Customizable Memory Schemes for Data Parallel Accelerators

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op

dinsdag 6 september 2011 om 10:00 uur

door

Chunyang GOU

Master of Science in Information and Communication Engineering
Tsinghua University, China
geboren te Sichuan, China

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. H. J. Sips

Copromotor:
Dr. ir. G. N. Gaydadjiev

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. H. J. Sips	Technische Universiteit Delft, promotor
Dr. ir. G. N. Gaydadjiev	Technische Universiteit Delft, copromotor
Prof. dr. P. Stenström	Chalmers University of Technology, Zweden
Prof. dr. A. Seznec	IRISA/INRIA, Frankrijk
Prof. dr. J. Takala	Tampere University of Technology, Finland
Prof. dr. K. G. Langendoen	Technische Universiteit Delft
Dr. G. K. Kuzmanov	Technische Universiteit Delft
Prof. dr. ir. A.-J. van der Veen	Technische Universiteit Delft, reservelid

ISBN 978-90-72298-23-2

Keywords: Computer Architecture, Data Parallel Accelerators, Parallel Memory Schemes, Customizable Memory Schemes

Copyright © 2011 Chunyang Gou
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.

Cover design by Hani Alers and Nike Gunawan (www.hanike.nl)
Printed in The Netherlands

This thesis is dedicated to my family.

Customizable Memory Schemes for Data Parallel Accelerators

Chunyang Gou

Abstract

Memory system efficiency is crucial for any processor to achieve high performance, especially in the case of data parallel machines. Processing capabilities of parallel lanes will be wasted, when data requests are not accomplished in a *sustainable* and *timely* manner. Irregular vector memory accesses can lead to inefficient use of the parallel banks/modules/channels and significantly degrade overall performance even when highly parallel memory systems are employed. This problem is also valid for many regular workloads exhibiting irregular vector accesses at runtime. This dissertation identifies the mismatch between the optimal access patterns required by the workloads and the physical data layout as one of the major factors for memory access inefficiency. We propose customizable memory schemes to address this issue in data parallel accelerators. More specifically, this thesis extends traditional approaches by proposing two new parallel memory schemes that alleviate bank conflicts for commonly used access patterns. We also propose a framework to capture and convey the access pattern information to the proposed parallel memory schemes. Furthermore, we describe techniques that dynamically adjust the instruction sequencer of a multithreaded vector architecture and customize the access patterns to improve on-chip, local memory efficiency. Last, we identify and exploit new locality type to dynamically adjust off-chip memory access granularity of manycore data parallel architectures, in order to improve main memory efficiency. We implemented our proposals as extensions of contemporary data parallel architectures and our evaluation results demonstrate that memory efficiency and overall system performance can be improved at minimal hardware cost, while at the same time programming overhead can be greatly reduced.

Acknowledgements

I acknowledge the help and contributions from many people during my PhD at the computer engineering (CE) laboratory of TU Delft. First of all, I owe deep gratitude to the guidance from my supervisor, Dr. Georgi Gaydadjiev. He insisted on high quality research and always motivated me to face challenging problems. He is always open-minded to let me follow my interests and was patient with me even when I entered panic mode. He taught me how to formulate problems and abstract complexity. He sacrificed many late hours and weekends working together with me to meet the deadlines. I truly enjoyed and greatly benefited from the past several years for being his student.

I would like to express my sincere gratitude to Dr. Georgi Kuzmanov for his daily supervision during the first half of my thesis work. He helped me develop academic thinking skills, identify the research problems at the beginning, and write technical papers. His rich experience and generous guidance has greatly benefited my early academic career and the influence will definitely continue.

I am sincerely grateful to late Prof. Stamatis Vassiliadis for giving me the opportunity to study in the CE group. It was my privilege knowing him; his memory will always remain in my heart. I thank Dr. Koen Beltels for setting me on the track of the first visit to the CE group and later the many useful suggestions on both academic work and life. I am also thankful to Prof. Henk Sips for serving as my promotor, and grateful to the thesis committee professors for their invaluable feedback and comments despite the tight time schedule.

I considered myself lucky to have nice colleagues in an international environment of the CE group, without whom the life at Delft would not be complete. Many thanks to Jae Young Hur, Ioannis Sourdis, Christos Strydis, Sebastian Isaza, Arnaldo Azevedo, Dimitrios Theodoropoulos, Roel Meeuws, Vlad-Mihai Sima, Laiq Hasan and Nor Zaidi Haron for their kind help and insightful discussions which refreshed my mind from time to time. My special thanks go to past and current officemates in HB.15.230: Elena Panainte, Carlo Galuzzi, Humberto Calderón, Zubair Nawaz, Pavel Zaykov, Aqeel Wahlah, and Karthik Chandrasekar, for creating such a harmonious work place and sharing the numerous technical and non-technical chats with me. I thank Catalin Ciobanu and Bogdan Spinean for the close collaboration in the SARC accelerator memory design. I also thank Filipa Duarte and Faisal Nadeem for the nice cooperation in organizing the Computer Engineering Colloquium over the years.

Special thanks are due to the CE secretary Lidwina Tromp for her administrative assistance and generous help. My thanks are also due to Bert, Erik and Eef, the past and current CE system administrators, especially for their technical support in operating the HPC clusters used to run our simulations.

I truly appreciate the friendship with many great Chinese friends during my stay in the Netherlands. Yizhi Zhao, Zhijiang Chang and Yi Lu kindly assisted me in finding an accommodation at Delft and setting up the new career when I joined the group. Their support continued ever since. Yao Wang and Changlin Chen's joining us later strengthened our Chinese "minority" in the CE group with whom I enjoyed pleasant chats, wonderful meals and long-distance biking. Outside the group I am also grateful to Qi Jia, Jun Hou, Hongchao Pan, Yang Yang, Shiming Xu, Jin He..., for their generous help and the many unforgettable gatherings and trips we enjoyed together.

It is my privilege to work in the SARC and ENCORE projects sponsored by the European Commission. I thank all the excellent people of these projects and the fruitful collaboration with them from which I benefited a lot.

I would like to thank many wonderful teachers and friends at Tsinghua University who created the premises for me to study in the Netherlands. Particularly, I am most grateful to Prof. Weimin Zheng for introducing me to computer architecture research. Sincere gratitude also goes to Prof. Yingning Peng for directing my steps toward Computer Engineering during my master thesis work.

Last but certainly not least, I would like to thank my family. I am forever indebted to my grandparents and parents, for their endless love and unwavering support throughout my life. They taught and encouraged me to become a good person to the society. I thank my wife, Ling Zhang, for her love, patience, support, understanding and sharing the wonderful life with me. I also thank my sister for always loving and supporting her only younger brother. I sincerely thank my parents in law for their love, support and trust. I honor the memory of my grandaunt and maternal grandmother. The everlasting love from my family has always been the source of peace and inspiration in my heart.

Chunyang Gou — Delft, The Netherlands, September 2011

Table of contents

Abstract . . . . **i**

Acknowledgments . . . . **iii**

List of Tables . . . . **ix**

List of Figures . . . . **xi**

List of Acronyms and Symbols . . . . **xv**

1 Introduction **1**
  1.1 Data Parallel Architecture Evolution 1
  1.2 Programmable Data Parallel Accelerators 3
  1.3 Processor-Memory Performance Gap 5
  1.4 Parallel Memory Organizations and Accesses 8
  1.5 Problem Formulation 12
  1.6 Machine Organization and Proposed Solutions 15
  1.7 Contributions 17
  1.8 Dissertation Organization 18

2 Related Work **21**
  2.1 Parallel Memory Schemes 21
  2.2 Access Pattern and Data Layout Optimizations 28
  2.3 Off-chip Memory Access Scheduling 30
  2.4 Memory Access Optimizations on GPUs 31
    2.4.1 Avoiding GPU Shared Memory Bank Conflicts 31
    2.4.2 GPU DRAM Access Optimizations 32
  2.5 Summary 33

3 Conflict-Free Parallel Memory Schemes **35**
  3.1 Motivation 35

3.1.1 Definitions 36
3.1.2 Non-Redundant Parallel Memory Schemes Limitations 37
3.1.3 Strided Access in 2D Environment 38
3.2 Single-Affiliation Multiple-Stride Memory Scheme 40
3.2.1 Moving from Conflict-Free to Low Degree of Affiliation 40
3.2.2 Hierarchical Single-Affiliation Parallel Memory Scheme 42
3.2.3 Solving Module Conflicts in Single-Affiliation Schemes 43
3.2.4 The Matched SAMS Scheme 45
3.3 2DSMM: 2D Strided Multiaccess Memory Scheme 46
3.4 Hardware Design and Implementation 49
3.4.1 SAMS Hardware Implementation 50
3.4.2 Implementation of 2DSMM Scheme 52
3.5 Discussion 65
3.6 Summary 67

4 Providing Multiple Views to Data 69
4.1 Introduction 69
4.2 Motivation 71
4.3 The Extended SAMS Scheme 74
4.3.1 Original SAMS Scheme 74
4.3.2 Proposed Extensions 75
4.4 Implementation and Integration 77
4.4.1 SAMS Organization and Implementation 77
4.4.2 Integration into the Cell SPE 80
4.5 Experimental Evaluation 83
4.6 Summary 92

5 Addressing On-chip Bank Conflicts 93
5.1 Introduction 93
5.2 Background and Motivation 95
5.2.1 Shared Memory Access on GPU 95
5.2.2 Motivating Example 97
5.3 Problem Analysis 98
5.3.1 Latency and Bandwidth Implications 98
5.3.2 Bank Conflicts Impact on Pipeline Performance 101
5.4 Elastic Pipeline Design 103
5.4.1 Safe Scheduling Distance and Conflict Tolerance . . . 104
5.4.2 Out-of-order Instruction Commitment 105

5.4.3 Extension for Large Warp Size 105
5.4.4 Hardware Overhead and Impact on Pipeline Timing . . 106
5.5 Bank-conflict Aware Warp Scheduling 107
5.5.1 Obtaining Bank Conflict Information 108
5.5.2 Bank Conflict History Cache 109
5.5.3 Proposed Warp Scheduling 111
5.5.4 Hardware Overhead 111
5.6 Experimental Evaluation 112
5.6.1 Effect on Pipeline Stall Reduction 115
5.6.2 Performance Improvements 117
5.6.3 Performance of Non-Conflicting Kernels 119
5.6.4 Interaction with Off-chip DRAM Access 120
5.7 Discussion 122
5.8 Summary 123

6 Improving DRAM Access Efficiency 125
6.1 Introduction 125
6.2 Background and Motivation 126
6.3 Horizontal Locality Aware DRAM Scheduling 130
6.3.1 Horizontal Locality 130
6.3.2 Compiler and Runtime Access Pattern Analyzer 134
6.3.3 Adaptive DRAM Access Granularity Scheduling . . . 136
6.4 Microarchitectural Extension 138
6.4.1 Vector MSHR (VMSHR) 138
6.4.2 Elastic MSHR with Deferred Reservation 141
6.5 Experimental Evaluation 143
6.5.1 Memory Bandwidth Requirement 146
6.5.2 DRAM Access Granularity Distribution 147
6.5.3 Improved DRAM Efficiency 149
6.5.4 Improvement in Overall Performance 150
6.5.5 Memory Access Utilization 150
6.5.6 Effect on a GPU with Less Capable Interconnect . . . 152
6.6 Discussion 153
6.6.1 Effect of Large DRAM Data Packets 153
6.6.2 Impact of HW Cache 154
6.6.3 Application to SPMD Barrel Processing 154
6.6.4 Possible Improvement 155
6.6.5 Contrast to Closely Related Art 155

6.7 Summary 157

7 Conclusions and Future Directions 159
7.1 Conclusions 159
7.2 Future Research Directions 161

A SAMS Conflict-Free Access Proof 165

B 2DSMM Properties and Formal Proof 173
B.1 Properties of the Basic XOR Scheme 173
B.2 Properties of the 2DSMM Scheme 174

Bibliography 185

List of Publications 195

Samenvatting 197

Curriculum Vitae 199

List of Tables

3.1 Delay and hardware usage of ATU 51
3.2 2DSMM scheme critical path and logic consumption 61
3.3 2DSMM scheme storage consumption 61

4.1 Stride control signals 78
4.2 Synthesis results of SAMS Multi-Layout Memory system . . . 79
4.3 Selected benchmark suit 84
4.4 SAMS instructions, intrinsics and macros 85
4.5 Changes over the original SPE and their impact 87
4.6 SPU dynamic instruction count and execution time 88

5.1 Benchmark shared memory BW requirement 100
5.2 Elastic pipeline HW overhead per GPU core 107
5.3 Hardware overhead of bank conflict prediction and warp mask generation (per GPU core) 111
5.4 The GPU processor configurations 113
5.5 Benchmark characteristics 114

6.1 Hardware cost of elastic MSHR with deferred reservation (per GPU core) 143
6.2 Baseline GPU configuration† 144
6.3 Benchmarks 145
6.4 DRAM access granularity distribution 148

List of Figures

1.1 Processor-memory performance gap 5

1.2 DRAM memory organization 8

1.3 1D array storage in 4 banks (uniform bank access timing) . . . 9

1.4 Bank conflicts in a multi-execution vector processing platform 10

1.5 1D array storage in 4 banks and memory accesses (non-uniform bank access timing) 11

1.6 Overview of the proposed customizable memory schemes . . . 16

1.7 Customizable memory schemes 18

2.1 Stride-2 vector access to three schemes: (a) low-order interleaving; (b) skewing (c) prime 22

2.2 Prime memory system with unused cells 23

3.1 Inherent limitation in multimodule memory assignment 37

3.2 SARC vector coprocessor memory addressing 39

3.3 SAMS single-affiliation scheme with $q = 2,\ s = 2$ 41

3.4 SAMS data arrangement example with 4 modules 44

3.5 2D addressing scheme 47

3.6 Example of 2DSMM module assignment for $N = 16, 2^p = 2, 2^q = 4, 2^{vs} = 2, 2^{hs} = 2$ 48

3.7 2DSMM access examples for $N = 32, 2^p = 2, 2^q = 4, 2^{vs} = 2, 2^{hs} = 2$ 49

3.8 Parallel memory system based on customizable HW schemes . 50

3.9 SAMS address translation logic: (a)Module assignment, (b)Row assignment, (c)Offset assignment 51
3.10 2DSMM address generation patterns 53
3.11 2DSMM address generation logic for six access patterns . . . 54
3.12 2DSMM address translation circuit with $M = 1024$, $N = 256$, $2^p = 2$, $2^q = 4$ 56
3.13 2DSMM properties for address & data routing simplification . 57
3.14 2DSMM address & data routing unit 58
3.15 Definition of access index for six 2DSMM access patterns . . 58
3.16 2DSMM address & data routing example ($2^p = 2$, $2^q = 2$) . . 59
3.17 2DSMM memory module depth scaling in Stratix-II with $2^p = 2$, $2^q = 4$, $data\_width = 64$ 62
3.18 2DSMM ALUTs consumption breakdown in Stratix-II with $2^p = 2, 2^q = 4, data\_width = 64$, memory module depth=8192 63
3.19 2DSMM data width scaling in Stratix-II with $2^p = 2$, $2^q = 4$, $memory\_module\_depth = 32K$ 64
3.20 2DSMM memory module number scaling in Stratix-II with $data\_width = 16$, $memory\_module\_depth = 16K$ 65

4.1 Vector-matrix multiplication: multiple working data sets . . . 71
4.2 Sample vector-matrix multiplication code 72
4.3 The proposed multi-layout memory 74
4.4 Internal data layouts in SAMS Multi-Layout Memory 75
4.5 SIMD memory organizations 77
4.6 SAMS Multi-Layout Memory Integration into the Cell SPE . . 81
4.7 Exemplary load/store pipeline optimization: (a)original SPE [57], (b)SAMS integration with optimizations 83

5.1 (a)CUDA threads hierarchy; (b)thread execution in GPU core pipeline; (c)GPU chip organization 96
5.2 AES source code 97
5.3 Effect of elastic pipeline in: (a)reducing pipeline stalls and (b)improving performance 98

5.4 Baseline in-order pipeline: (a)unified memory stages and (b)split memory stages 102
5.5 Elastic pipeline 103
5.6 Elastic pipeline logic diagram 106
5.7 Bank-conflict aware warp ready signal generation 109
5.8 Pipeline stall reduction. In each group: left bar: baseline GPU; right bar: elastic pipeline enhanced GPU 115
5.9 Performance improvement 117
5.10 Elastic pipeline vs hand-optimized code for conflicting kernels 119
5.11 Elastic pipeline performance for non-conflicting kernels 120
5.12 DRAM bandwidth impact 121

6.1 Hierarchy and memory accesses of worker threads 127
6.2 Baseline GPU organization with barrel processing 129
6.3 Matrix multiplication PC-Histogram: all 4 CTAs 132
6.4 Matrix multiplication PC-Histogram: 1 CTA 133
6.5 VMSHR design: (a)baseline; (b)modifications for elastic MSHR 139
6.6 Memory bandwidth requirement 146
6.7 DRAM efficiency 149
6.8 Speedup over the baseline 150
6.9 Memory access utilization 151
6.10 DRAM efficiency improvement and overall performance speedup (num_vcs=1) 153

A.1 Binary bits representation of $b = a + \delta \cdot 2^0$ when $1 \leq s \leq q$. 166
A.2 Binary bits representation of $b = a + \delta \cdot 2^1$ when $1 \leq s \leq q$. 167
A.3 The accessed addresses with stride $S = 2^s$ 169
A.4 Binary bits representation of $b = a + \delta$ when $s > q$ 170

B.1 Address sequence of strided row access 177
B.2 Address sequence of unit-stride row access 180
B.3 Address sequence of strided forward diagonal access 181

List of Acronyms and Symbols

AoS, SoA	Array of Structures, Structure of Arrays
ASIC	Application Specific Integrated Circuit
BW	Bandwidth
CLA	Carry Look-ahead Adder
CPU	Central Processing Unit
CSA	Carry Save Adder
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
DLP	Data Level Parallelism
FPGA	Field Programmable Gate Array
FR − FCFS	First Ready, First Come First Serviced
FIFO	First In, First Out
FLOP	FLoating-point OPeration
FLOPS	FLoating-point OPeration per Second
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GPGPU	General-purpose Processing on GPUs
HPC	High Performance Computing
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LUT	Lookup Table
MC	Memory Controller
MD	Molecular Dynamics
MIPS	Million Instructions Per Second
MSHR	Miss Status Holding Register
OoO	Out of Order
PCB	Printed Circuit Board
PMSHR	Primary Miss Status Holding Register
RTL	Register-Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SRAM	Static Random Access Memory
TLP	Task Level Parallelism

1

Introduction

In the past decades, the amount of digital data newly created and processed worldwide has been steadily growing. It is quite clear that this trend is going to continue in the future. This introduces tremendous challenges for the processing capabilities of modern computer systems. Many of the widely used workloads follow a particular processing paradigm: the *same* computation is repeated for a massive number of data elements. Such paradigm, called *data parallel computing*, exists in many applications ranging from consumer and desktop computing (such as multimedia and 3D graphics) up to the high-performance domain (e.g., financial analysis, bioinformatics, physics simulation and more). Data parallel processing, known also as *single instruction multiple data* (SIMD) [45], demands more efficient computing and memory hierarchy schemes when compared to scalar approaches.

1.1 Data Parallel Architecture Evolution

Vector supercomputers are the first best known examples of data parallel machines. Early successful implementations date back to the 1970s, with CDC Star-100 [33] and Texas Instruments ASC [148] as two representative examples. Such vector supercomputers were built with proprietary, dedicated vector processors and rather expensive, highly-banked memory arrays. Vector processing had played a central role in supercomputers and has been strikingly successful up to 1990s [6]. The most prominent vector supercomputers in history include the Cray series [126] and the NEC SX models [58].

The dominance of the vector supercomputers was gradually replaced by the “killer-micros” (high performance computers built using commodity microprocessors) since early 1990s [42] [1]. Nonetheless, the growth of data parallel processing paradigm did not stop. In fact, since then more data parallel architectures have arisen in the commodity computing market. The advent of such data parallel computing facilities is driven by the growing importance of data intensive computing on desktops [28, 41], and is enabled by the advances of the semiconductor technology.

On one hand, various types of SIMD extensions, such as SSE [140], 3DNow [9], and Altivec [40], have been introduced into the general purpose processors, in the form of short vector datapaths tightly coupled to the original scalar pipelines. The *general purpose processor* (GPP) SIMD extensions typically provide moderate computation capability limited by short vectors, but have the merit of being compatible with legacy code (including system software). On the other hand, data parallel accelerators appeared in application domains such as home entertainment (e.g., the Cell processor [68, 120]), computer graphics (such as the *graphics processing units* (GPUs) [96, 113], and visual computing accelerators [128]). In contrast to GPP SIMD extensions, there is less burden of backward comparability in terms of executing legacy code and the programming styles are still evolving. As a result, these data parallel accelerators can be *specialized* for a narrower set of workloads. In doing so, both the hardware designs and the programming models can be highly customized to improve performance, power and chip area efficiency.

Besides the GPP SIMD extensions and accelerators mentioned above, huge amount of special-purpose hardwired functional units implemented by *application specific integrated circuits* (ASICs) have also been designed and utilized to exploit data parallelism for dedicated applications. ASIC based systems are necessitated when extremely high performance is required, or in case rigid cost/power constraints render the programmability as less important. Good examples are the Anton multiprocessor for high-speed *molecular dynamics* (MD) simulations from DE Shaw Research [132] and the video codecs presented in [25, 91]. Often, the design objectives of such special-purpose computing systems determines architectural choices which largely sacrifice flexibility and programmability. Although some of the techniques presented in this dissertation are also directly applicable to ASIC based systems, we focus our discussion to programmable data parallel accelerators because of their wider scope and because we strongly believe that such accelerators are a promising approach to process data parallel workloads.

---

[1] Recently data parallel processors are returning to the Top 500 list [6] in modernized form (such as machines based on GPUs).

1.2 Programmable Data Parallel Accelerators

In general, a *compute accelerator* [119] is a hardware entity *separate from* the host general purpose processor, that offers higher performance and better silicon area/power efficiency as compared to the host processor. This is achieved usually for a restricted class of applications in a very specific domain [119]. To achieve this, the target applications characteristics are heavily exploited during the design process. Naturally, such design choices can provide combined performance/area/power benefits at the expense of flexibility[2]. In early compute accelerators [2] this often resulted in fixed, hardwired datapath designs.

Recently, there seems to be a consensus that high-level programmability using general purpose programming languages becomes a requirement for accelerators [113, 114, 135]. High-level programmability with matured programming styles/models not only improves programmer productivity and code portability, it also increases applicability of the same accelerator substrate to a wider set of application domains. This has been demonstrated by recent proposals in, e.g., *general-purpose processing on GPUs* (GPGPU) [1, 114], unified visual computing accelerators [128], and unified accelerator architectures for imaging, video and next-generation immersive multimodal applications [135]. The economic rationale driving the merging of domain-specific accelerators is that, expanding the application domains range allows for potentially larger product volume resulting from increased shares in the marketplace.

For all programmable accelerators, the common essential aim is to exploit parallelism in the targeted applications. Naturally, application code parallelism can be categorized into three major types: *instruction*-, *task*-, and *data-level*. While *instruction level parallelism* (ILP) has been intensively pursued by GPPs, task and data level parallelism are usually the primary targets of programmable accelerators. Task parallel accelerators scale well with irregular, dynamic and heterogeneous problems[3], that correspond to the global application scope situation. However, they suffer from being less capable of capturing the potential computing efficiency available in data parallel kernels.

In contrast, data parallel accelerators are concerned with (local scope) kernels with regular and homogeneous processing behavior. There are at least two important features inherent to the assumed data parallel processing paradigm: 1) it is relatively easy to get high utilization of throughput architectures (e.g., par-

---

[2] In this dissertation flexibility is used for expressing programmability and wide applicability.

[3] Irregular problems/applications/workloads in this thesis denote those with irregular data accesses and/or irregular control flows. (Ir)regular data access is elaborated in Section 1.5.

allel vector lanes); and 2) the pipeline frontend processing has high area/power efficiency due to the reduced frontend processing bandwidth requirements. The above two merits, plus the widespread data parallelism in emerging applications, naturally render data parallel architectures to be the primary choice for accelerator designs targeting regular and homogeneous workloads operating on large data sets.

Data and task parallelism are not mutually exclusive but rather complementary. Currently we have already seen commercial products with multicore GPPs and GPUs integrated on a single die [7, 12, 111]. These systems are able to provide suitable hardware substrates for combined data and task parallel processing. The more challenging issue of proper programming models and their efficient hardware interfaces, remains an open research question, the solution of which is still at its infancy. The merging process brings up a set of new challenges that have to be addressed in order to ensure the success of such co-exiting architectures. One particular example is the shared memory system infrastructure that has to efficiently support the memory accesses from both sides. The findings in this thesis can be viewed as one such step toward improving *memory access efficiency* (in terms of bandwidth utilization and access time) of such contemporary co-execution platforms.

Current data parallel processors often adopt local memories with no coherence support implemented in hardware (HW) and less provisions to reduce single thread latencies compared to GPPs. Such architectural choices often compromise the programmability to a certain extent, i.e., as restricted by specific programming styles/models. Generally speaking, the specific programming models exert nontrivial burdens on programmers, such as explicit management of parallelism and the memory hierarchy, which should be ideally avoided. This inspires a challenging research topic exploring novel programming models and architectures to relieve such burdens by striking the right balance between the hardware efficiency (in terms of performance gain/additional HW cost) and programming efforts. This was also the main motivation behind the Vector-Thread Architectures [80], Sequoal [43], Rigel [69], SARC [123] and ENCORE [4] research projects. The specialized programming models also provide opportunities for system-level optimizations, since, in general, more information about the application behavior is available when the code is written under specific constraints of programming styles/models. In addition, during execution additional information becomes available and can be utilized for performance optimizations. Based on the above observations, data parallel programming models/language extensions, compile-time analysis, runtime library optimizations, and architectural extensions will be jointly explored, to address the memory efficiency issues in this dissertation.

1.3 Processor-Memory Performance Gap

It is well known that memory density and capacity have grown along with the CPU computational capacity and complexity, however memory speed has not kept pace in this process. As a consequence, the processor-memory performance gap has steadily increased for decades, rendering memory access as a major bottleneck. Figure 1.1 shows the relative performance improvements of microprocessors and DRAM memories during the past 30 years, from both bandwidth and latency perspectives[4]. For memory graph the Asynchronous 16 bits memory system used by the industry in the eighties forms our baseline and for processors we have chosen Intel 80286 (without 80287 floating point coprocessor). All memory latencies are calculated under cold start assumptions. More precisely, we do not account for delays introduced by memory bank conflicts but consider only the accumulative numbers of Open Row and Col-

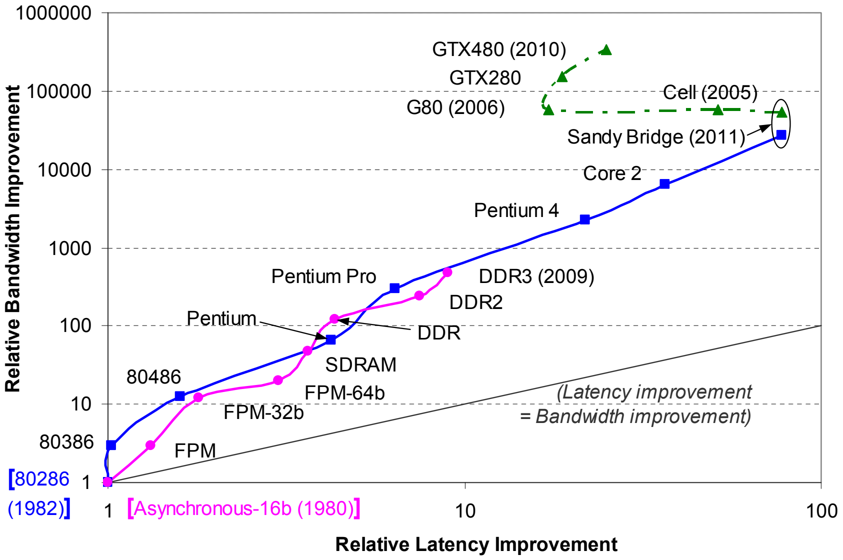


Figure 1.1: Processor-memory performance gap

---

[4] The idea of this plot and part of the source data are originated from [59]; other data are taken from public resources of respective vendors [64, 65, 112, 120].

umn access latencies. In respect to bandwidth, we considered a single channel memory configuration as adopted by the industry [65] at the corresponding time. The estimated bandwidth is the product of the frequency and the data port width. The microprocessor instruction latency is expressed by the absolute delay of the integer addition instruction. The instruction bandwidth is the product of the maximum achievable *instructions per cycle* (IPC) and the pipeline frequency. For data parallel processors[5] (the IBM *Cell Broadband Engine*, and three representative GPUs) we determine the latency similar to the microprocessors, while the relative bandwidth is in *floating point*[6] *operations per second* (FLOPS)[7]. We can safely relate data parallel architecture results to the 80286 base line since the frequencies of both pipelines (fixed/floating-point) are the same for the data parallel accelerators. The two graphs representing the microprocessors and the memories are aligned in time by using roughly the same start year, end year and similar time periods for the intermediate points.

We used the 4-core Intel *Sandy Bridge* processor to relate the raw computational capabilities of state of the art GPPs and data parallel accelerators. Note, that there is a difference between the instruction throughput and FLOPS metrics as shown in Figure 1.1. The 4-issue Sandy Bridge core implements *advanced vector extensions* (AVX) ISA which supports 256-bit vector operation (i.e., 8 FLOP) per cycle, hence, its peak FLOPS is twice as big as its peak instruction throughput. This shows two GPP trends: 1) that contemporary GPPs have the raw computational capabilities of data parallel accelerators more than 5 years old and 2) in order to cope with this delay, GPPs are heavily relying on wider SIMD lanes (exploiting more data parallelism).

Several observations can be drawn from Figure 1.1. First, the bandwidth has been improved at a significantly higher rate than latency, for both memories and processors. This is apparent by noticing that all three curves are well above the straight line representing identical bandwidth and latency growths. Moreover, it is quite clear that such disparity between the bandwidth and latency improvement ratio is going to continue. The reason for this is twofold. On one hand, since latency is constrained by laws of physics, its improvement can be mainly attributed to the process technology improvements resulting in increased clock frequency. This is becoming harder as we are approaching certain physical boundaries of the used technology. For example, power consumption is now becoming a major constraint that limits future clock frequency

---

[5] In the rest of this thesis, we use the terms *(programmable) data parallel accelerators* and *data parallel processors* interchangeably.

[6] We consider single precision floating point.

[7] Fused multiply-add is considered as a single FLOP in our calculations.

increases. On the other hand, improving bandwidth is relatively easier. Exponential growth of transistor density, pushed by *Moore's Law* [99], has been successfully driving the industry for the past thirty years and is still continuing today and is used to improve on-chip bandwidth. Off-chip bandwidth is more expensive (e.g., chip pin-counts are limited, have high parasitic impedances and PCB traces engineering is costly) but still manageable at reduced scale. Thus, it is not surprising that the industry has, particularly at present time, opted for *throughput computing* (e.g., wider SIMD lanes, massive HW multithreading) as the "easy" way towards *aggregate* performance improvements. In fact, the parallel processing paradigm has been so widely spread, that almost all computers today are eventually parallel. For example, even hand-held devices (such as smart phones) are powered by multicores [111, 121].

Second, DRAM performance improvement has not kept pace with the processors, in terms of both bandwidth and latency, for the past thirty years. In fact, memory latency has been reduced by 7.6% per year, while the microprocessor latency reduction amounts to 15.6% on average. Similarly, the DRAM bandwidth annual compound increment is 22.8%, whereas for microprocessors it is 40.5%. The bandwidth growth rate for data parallel accelerators is even more impressive – 55.8% for the past three GPU generations covering only 6 years. This trend has several important implications stated below.

To deal with the processor-memory latency gap, techniques to alleviate long memory latencies have become mainstream in microprocessor design. For example, caching and prefetching are widely adopted in microprocessors to *hide* memory latency and improve performance [59]. Interestingly, various prefetching techniques hide latency often at the expense of additional bandwidth consumption. In addition, hardware multithreading, a valid technique to *tolerate* long latencies, has been embodied in both CPUs and typical data parallel architectures (such as GPUs), as one of the main means to achieve high throughput. Hence, bandwidth is increased to compensate long latencies and provide high aggregate throughput.

The differences in processor-memory bandwidth improvements in time, have resulted in single memory unit being inadequate to feed its processor. In addition, the trend is clear that such disparity in processor and memory bandwidth is increasing (predicted from the last four memory/processor pairs). For sufficient memory bandwidth and reduced access latency, parallel memory systems with multiple memory units (e.g., banks/channels) are widely used in data parallel computers, from vector supercomputers [58, 126] to commodity data parallel accelerators [44, 51]. Despite the deployment of parallel memory units,

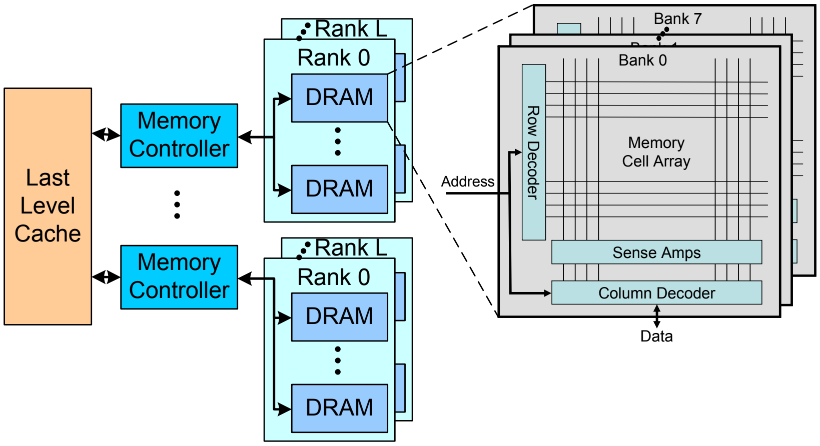


Figure 1.2: DRAM memory organization

efficient vector memory access[8] still remains one of the critical problems in data parallel computers.

1.4 Parallel Memory Organizations and Accesses

As described above, to mitigate the processor-memory performance gap, parallel memory organizations, with multiple memory units (such as banks, modules, and channels), have been proposed and successfully adopted in both early and contemporary high performance computers [83,87], microprocessors [89], and mobile processors [79,121]. In such memory organizations, parallel memory units can be independently indexed to service concurrent memory accesses, resulting in increased bandwidth and decreased latency. As an example, Figure 1.2 shows a typical DRAM memory subsystem organization, which consists of a few independent memory channels. Each channel, controlled by a memory controller (MC), manages a couple of independent memory *ranks* each of which employs several DRAM chips working in lock-step mode. Each DRAM chip is composed of several banks. While each bank can process MC commands independently, all banks inside the same chip must share their data/address buses.

Such parallel memory organizations, with abundant *raw* memory bandwidth,

---

[8] Parallel/vector/SIMD memory access denote the same memory access paradigm, and are used interchangeably in this thesis (unless explicitly stated otherwise).

can *potentially* improve system performance. One should note that it is often challenging to *effectively* utilize the available bandwidth and achieve high *sustained bandwidth*. This is especially the case in a common execution environment where thousands of in-flight memory accesses, from multiple processing elements, are *competing* for the memory resources and interfering with each other. Indeed, the parallel memory access inefficiency is not a new problem; for example, it has been extensively studied for traditional vector memory accesses, as discussed in the following.

In traditional vector machines, parallel memory organizations were often presented and analyzed by an abstract model as shown in Figure 1.3. The model consists of parallel banks, each of which has independent address decoding logic (i.e., each bank can be indexed independently). While accesses to different banks can be simultaneously serviced, accesses to the same bank must be serialized (but with *uniform access timing*).

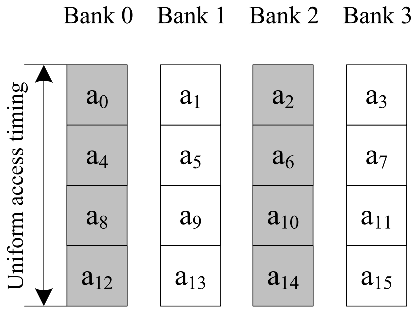


Figure 1.3: 1D array storage in 4 banks (uniform bank access timing)

In data parallel processors (such as vector processors), the data to be processed is usually stored in array structures. E.g., if a machine has m parallel memory units, we can store one-dimensional arrays across the units, in a format as Figure 1.3 shows, for $m = 4$. When concurrent memory accesses are to contiguous array elements, all memory units are working in parallel, allowing peak bandwidth. However, if only even array elements are accessed (the solid squares in Figure 1.3), the *effective bandwidth* is reduced by half, and, as a result, access latencies are penalized. This effect is usually referred to as "bank conflict" in the literature[9]. Depending on the microarchitecture, bank conflicts can result from interference either among operations executed by the same

---

[9] Also known as "partition camping" in the scenario of DRAM accesses that skewed towards a subset of available DRAM channels [151].

vector access (*intra-vector interference*), or among different vector memory accesses from the same/different execution context(s) (*inter-vector interference*), as illustrated in Figure 1.4.

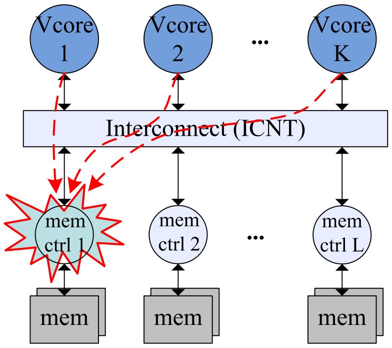


Figure 1.4: Bank conflicts in a multi-execution vector processing platform

It is clear that the memory inefficiency problem (such as bank conflict) also occur in scalar processing. The distinction between vector and scalar memory accesses, however, is that a single vector memory access comprises a group of scalar accesses, which are packed into a single instruction and executed together (normally in lockstep) following a specific pattern. As a result, interference may occur, not only among different vector memory accesses, but also among operations executed inside the same vector access. In fact, while inter-vector interference is similar to memory access interference in scalar processing, intra-vector interference is special in that it is decided by the vector access pattern, which is often predetermined (even for irregular access patterns) and usually predictable (even in case of dynamic patterns).

It should be pointed out that, there are also other sources of parallel memory inefficiencies beyond the bank conflicts modeled in Figure 1.3. Memory parallelism can be implemented at different HW levels, using various memory types, with different characteristics (such as uniform/non-uniform access timing, being fully pipelined or not). In fact, the simple model illustrated in Figure 1.3 did not take into account the timing diversity in accessing the same bank, limiting its application to only memory types with (relatively) uniform access timing (such as SRAM banks). The advantage of such model lies in its simplicity, which often results in closed-form mathematical formula that sufficiently describes the memory access behavior. On the other hand, however, for memory units with non-uniform access timing, the parallel memory model and thus the factors impacting the access performance become more complex.

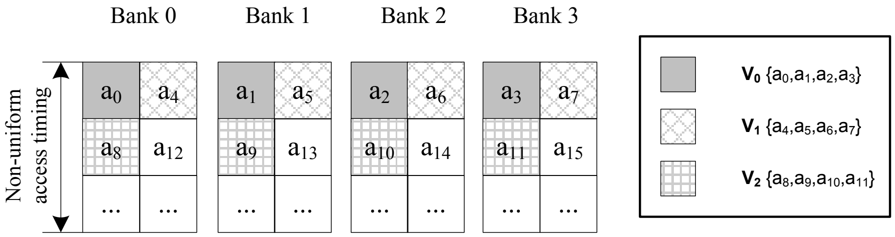


Figure 1.5: 1D array storage in 4 banks and memory accesses (non-uniform bank access timing)

For example, Figure 1.5 illustrates the parallel memory organization but with non-uniform bank access timing (DRAM). In this case, contiguous vector access is still *conflict-free* (similar to the case in Figure 1.3), however, the performance now depends also on the specific address sequence to each bank. E.g., when vector access $\{a_0, a_1, a_2, a_3\}$ (V_0) is followed by $\{a_4, a_5, a_6, a_7\}$ (V_1), peak throughput is achieved since the second accesses *hit* the DRAM page buffer and can be serviced with lowest latency possible. Whereas when V_0 is followed by $\{a_8, a_9, a_{10}, a_{11}\}$ (V_2), first the contents of the DRAM page (V_0 and V_1) will have to be written back to the cell array (the "Pregarge" command); then V_2 can be read out ("Activate") to the page buffer. The two additional steps (as compared to the page-hit case) incur wasted cycles on the data port and penalize memory performance. As a result, for the same vector accesses $\{V_0, V_1, V_2\}$, the ordering (e.g., $V_0 \rightarrow V_1 \rightarrow V_2$ vs $V_0 \rightarrow V_2 \rightarrow V_1$) can have drastic impact on performance. Thus such inter-vector access ordering is also considered as an important part of the vector access pattern in case of non-uniform bank access timing.

Memory access interference is a serious problem, since the low utilization of parallel memory resources often incurs non-trivial system performance degradation even in case of *balanced* processor-memory designs[10]. On the other hand, naive solutions, such as employing arbitrarily large number of memory units for excessive raw bandwidth, result in *overdesigned* memory systems which often lead to prohibitive cost and power consumption. In fact, we have observed the trend of decreasing BW/FLOPS ratio in recent microprocessor and GPU generations (Section 1.3), which necessitates more efficient utilization of memory resources.

---

[10] A *balanced* processor-memory design denotes that processor and memory bandwidths are matched.

From the above examples, it can be seen that both memory organization/data storage format and vector access patterns are of vital importance for high-performance parallel memory systems. When fixing one of the two, optimizing the other one can often improve system performance. To this end, *parallel memory schemes* have been designed for vector processing systems, usually optimized for a *given* set of vector access patterns. Traditionally, such schemes have been responsible mainly for two tasks. First, they manage specific organizations of parallel memory resources (e.g., at what memory hierarchy levels, and the number of parallel memory units at each level). Secondly, parallel memory schemes designate the corresponding low-level address mapping, from linear address space to the physical locations in the parallel units (e.g., row/bank/channel id). The goal of memory schemes is to increase access parallelism among multiple memory units and improve the performance, usually by supporting conflict-free vector accesses for the given set of vector access patterns. A more detailed study of parallel memory schemes in the literature will be introduced in Chapter 2.

1.5 Problem Formulation

As described in the sections above, memory system performance is crucial for any processor to achieve high performance, but especially for data parallel machines. The processing capability of parallel lanes can be efficiently utilized only when the data request is accomplished in a *sustainable* and *timely* manner. Unfortunately, even for parallel memory systems with adequate raw bandwidth, *irregular* access can lead to inefficient utilization of the parallel memory resources and significantly degrade overall performance.

Vector memory access regularity is completely determined by *parallel memory access patterns* and *physical data layout*[11]. From a high-level point of view, all memory operations are executed on data parallel machines after two transformations. First, memory operations on working data elements are bundled into vector accesses, and mapped onto parallel hardware lanes for execution. This is done through the vectorization/SIMDization process which largely determines the memory access patterns of the code[12]. Second, the working data sets are mapped to their physical locations on the (usually multi-dimensional)

---

[11] *Physical data layout* denotes the data layout in memory units and, as a result, the data access behavior (such as access timing).

[12] Memory access patterns are also affected by runtime dynamics in multithreaded/out-of-order execution, such as GPUs.

memory topology containing multiple rows/banks/channels. This is realized by multiple entities crossing many computing system abstraction layers, from data structures definition at the program level down to the data layout instantiation in memory hardware. For the entire access pattern set, each type of physical data layout defines its own optimal pattern subset. Memory accesses on the specific physical data layout can obtain peak performance, only if its optimal patterns are followed. For the rest, the memory efficiency degrades. A good example is the memory hierarchy of a typical computing system, whose (default) data layout exclusively favors contiguous memory accesses. As such, accesses to continuous addresses are often referred to as “regular accesses”.

In a single execution context (e.g., single-threaded vector processing), irregular memory accesses are typically originated from sparse data structures and/or highly irregular control flows. In addition, even applications which are generally regarded as regular and homogeneous can exhibit irregular vector memory accesses, e.g., access to a matrix column with row-majored storage format. In a multithreading scenario, concurrent memory accesses from multiple execution contexts can interfere with each other and end up with irregular accesses, even if memory access streams from individual execution are highly regular.

To summarize, in data parallel machines, memory accesses inefficiency roots in the mismatch between the actual parallel data access pattern and the one optimal for the physical data layout. Such a mismatch often results in memory bandwidth and latency inefficiencies, which penalize system performance. In particular, some typical scenarios of such inefficiencies include:

- For non-banked on-chip memory designs (adopted in some GPP SIMD extensions [140] and SIMD processors [68]), if the target addresses of a vector memory access are discontinuous, additional memory accesses and data rearrangement operations are required. This is widely known as the SIMDization data rearrangement overhead [11, 106, 124].

- For heavily banked on-chip memory designs (adopted in, e.g., GPUs), mismatch between the data access pattern and the physical data layout among banks incurs on-chip memory bank conflicts, similar to traditional vector processors [75]. Such bank conflicts result in pipeline stalls and overall performance degradation [54];

- For manycore architectures (such as GPUs), another issue is the interference among off-chip memory access streams from different cores [152]. Such interference often leads to hot DRAM channels, DRAM bank conflicts, and data bus read/write turn-around penalties [53].

As shown in Section 1.4, traditional parallel memory schemes utilize parallel hardware memory units, and manage low-level address mapping, to achieve *conflict-free* memory access for *given* vector access patterns. In this thesis, however, we notice that memory access patterns in contemporary data parallel processors are not necessarily hard-coded in the *instruction set architecture* (ISA) and exposed to the programmer (unlike the case in traditional vector processors). Thus, the access pattern information may not be taken for granted; instead, it is an important design consideration, from the system architect pointer of view. Therefore, we argue that, one should not be limited to just one memory hierarchy level when designing parallel memory schemes; instead, such schemes should be designed at system level, and consolidated with joint efforts often crossing the HW/SW boundaries. To this end, we propose *customizable memory schemes*, which extend conventional parallel memory schemes, by taking a holistic approach combining both memory access pattern *extraction* and *exploitation*. In such schemes, access pattern information is efficiently captured and utilized by customizing the low-level memory mapping with specific parallel memory schemes.

Additional techniques targeting parallel memory efficiency are also incorporated in our customizable memory schemes, in a broad sense. For example, with adequate memory access pattern information, the instruction sequencer in the pipeline front end (e.g., instruction/thread scheduling), the runtime scheduler (e.g., task scheduling), and the memory access scheduler (at the memory controller side) can also be customized, to better *adjust* the vector access patterns and *adapt* to the low level data layout and memory behavior for improved efficiency. Furthermore, vector access patterns and the physical data layout can be ideally coordinated for optimal performance, in a unified framework, similar in spirit to the compiler techniques that combine both vectorization and memory layout optimization [52, 106, 124].

The customizable memory schemes approach raises the following research questions, among others:

- How to design proper memory mapping schemes and provide a customizable yet generic mechanism for optimized layout of common data structures (e.g., 1D/2D arrays, *Array of Structures* (AoS));
- How to capture the minimal required information about the vector memory access pattern, and at which application phases;
- How to exploit the vector memory access pattern information to improve performance with low overhead, customizable hardware.

In answering the above questions, this dissertation aims at approaches to properly obtain the parallel memory access information, and leverage it to steer the co-designed hardware schemes. Such approaches are adapted to various contemporary data parallel architectures. In this way, customizable memory schemes significantly improve memory efficiency and overall performance, for both single data parallel core and manycore systems.

1.6 Machine Organization and Proposed Solutions

Figure 1.6 shows the organization of the baseline data parallel machine assumed in this thesis. This generic model is able to incorporate major contemporary multi/many-core data parallel accelerator variants. For example, the Cell processor [68] can be instantiated from Figure 1.6 with one GPP core and eight *synergistic processing element* (SPE) cores connected using a bidirectional ring bus. The SPE core has access to its local memory (called "local store" in the Cell SPE) and the attached DMA engine. Similarly, standalone GPUs can be instantiated with only accelerator cores connected by a crossbar, where both the local memory (called "shared memory" in NVIDIA GPUs) deployed at each core and the main memory are directly accessible by cores via specific load/store instructions. Furthermore, systems with GPU and CPU integrated on a single die [7, 12] can also be modeled with coexisting GPP and accelerator nodes as shown in Figure 1.6.

As discussed in Section 1.5, memory access inefficiency mainly roots in the mismatch between the parallel access patterns and the one favored by the physical data layout. This problem exists for both on-chip and off-chip memories, as highlighted in Figure 1.6. This will be addressed in Chapter 3, by introducing novel conflict-free parallel memory schemes (SAMS and 2DSMM). With such hardware parallel memory interleaving schemes, the physical data layout can be customized to support conflict-free vector access for strides from multiple stride families, in both 1D and 2D access environments.

Bank interleaving schemes have been adopted in traditional vector supercomputers. However, to the best of our knowledge, it is less explored in literature how such HW schemes can be *efficiently* integrated into modern short vector SIMD processors. Chapter 4 gives an example (called "SAMS Multi-Layout Memory") for such integration *at system level*. In this chapter, both the Cell SPE pipeline HW extensions and the programming interfaces for expressing *Array of Structure* (AoS) and *Structure of Array* (SoA) access types are presented. We illustrate how to capture such high-level information in a

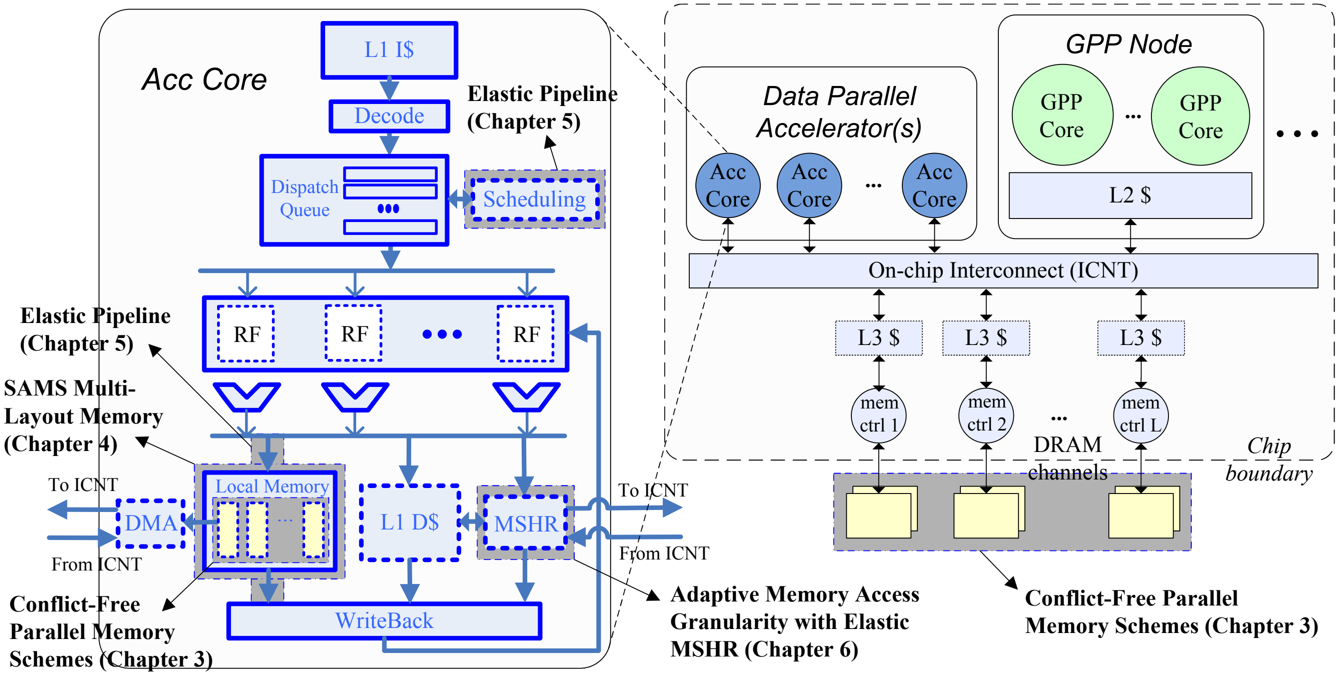

Acc Core
L1 I$
Decode
Elastic Pipeline (Chapter 5)
Dispatch Queue
Scheduling
Elastic Pipeline (Chapter 5)
RF
RF
RF
SAMS Multi-Layout Memory (Chapter 4)
To ICNT
From ICNT
DMA
Local Memory
...
L1 D$
MSHR
To ICNT
From ICNT
WriteBack
Conflict-Free Parallel Memory Schemes (Chapter 3)
Adaptive Memory Access Granularity with Elastic MSHR (Chapter 6)
Data Parallel Accelerator(s)
Acc Core
Acc Core
...
Acc Core
GPP Node
GPP Core
...
GPP Core
L2 $
On-chip Interconnect (ICNT)
L3 $
L3 $
L3 $
mem ctrl 1
mem ctrl 2
...
mem ctrl L
DRAM channels
Chip boundary
Conflict-Free Parallel Memory Schemes (Chapter 3)

Figure 1.6: Overview of the proposed customizable memory schemes

programmer-friendly way and utilize it to orchestrate the HW parallel memory schemes and improve memory access efficiency.

Multithreaded data parallel architectures are becoming popular, such as GPUs and their variants (e.g., Larrabee [128]). In a multithreaded in-order SIMD execution, on-chip, local memory access bank conflicts can be *hidden*, by the elastic pipeline design (Chapter 5) which decouples shared memory bank conflicts from pipeline stalls. Furthermore, clustered bank conflicts can be resolved by the co-designed bank-conflict aware warp scheduling technique, as shown at the top of Figure 1.6.

Another memory inefficiency is incurred by the interference among off-chip memory access streams from different cores. In barrel execution of *single program multiple data* (SPMD) programs (adopted in, e.g., GPUs), there exists abundant inter-thread data locality (named “horizontal locality” in Chapter 6). Horizontal locality is exploited to address DRAM access interference among cores and hence improve system performance. This is achieved by a holistic DRAM bandwidth efficiency optimization framework with combined compile-time, runtime and architectural efforts. On the hardware side, an elastic vector *miss status holding register* (MSHR) unit with deferred reservation is proposed, as shown in Figure 1.6.

1.7 Contributions

This dissertation proposes customizable memory schemes to address parallel memory access efficiency issues in data parallel accelerators, as illustrated in Figure 1.7. In this figure **V** denotes traditional vector architectures with memory access patterns hard-coded in the ISA and exposed to the programmer, while **V+** denotes contemporary data parallel processors such as the Cell and GPUs. In traditional parallel memory schemes (❶ in Figure 1.7), access pattern information is usually assumed as given and hence efforts are often dedicated only to designing hardware memory schemes/physical data layout[13] for fixed access patterns. This thesis extends traditional parallel memory schemes and makes the following contributions:

- two new parallel memory schemes (*SAMS* and *2DSMM*) to cope with bank conflicts for widely used access patterns (❷ in Figure 1.7);

---

[13] The HW memory scheme determines the layout/distribution of data in the memory units and hence the access behavior – i.e., the physical data layout.

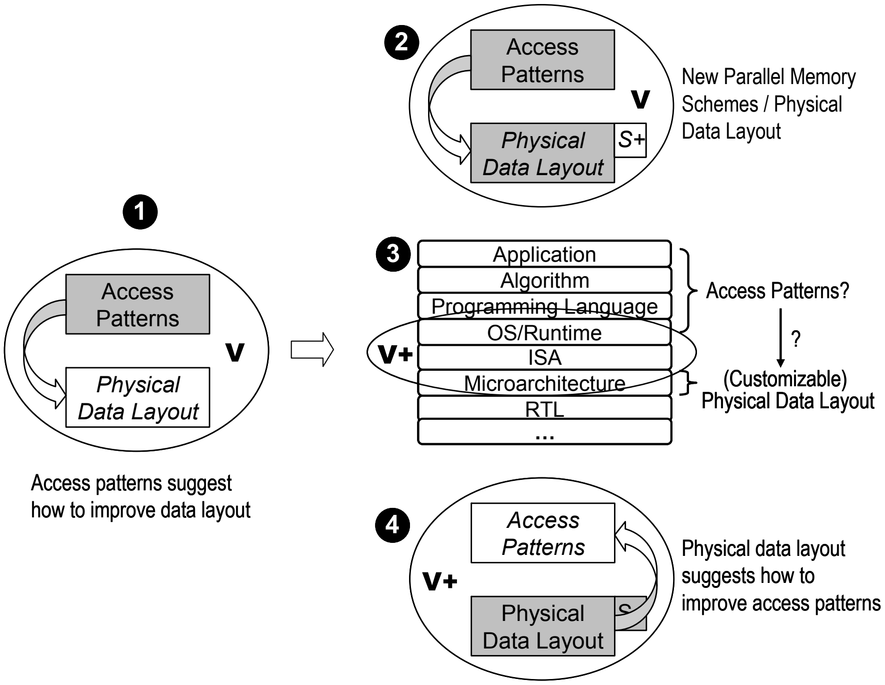


Figure 1.7: Customizable memory schemes

- a framework (*SAMS Multi-Layout Memory*) to capture and convey the access pattern information to the customizable parallel memory scheme integrated into contemporary data parallel architectures (❷ and ❸);
- *elastic pipeline* that dynamically adjusts the instruction sequencer of a multithreaded vector architecture to customize the access patterns and improve the on-chip, local memory efficiency (❸ and ❹);
- *elastic MSHR* to exploit horizontal locality and adjust off-chip memory access granularity of a manycore data parallel architecture, hence improve the main memory efficiency (❸ and ❹).

1.8 Dissertation Organization

This dissertation is organized as follows.

Chapter 2 gives an overview of the state of the art in solving memory access inefficiency problems in data parallel architectures. Specifically, it provides a survey on existing HW parallel memory schemes, techniques to reduce SIMDization data realignment overhead, techniques to relieve GPU on-chip memory bank conflicts, and techniques to address off-chip main memory efficiency issues in both GPUs and GPPs.

Chapter 3 introduces two novel hardware parallel memory schemes to cope with vector access memory bank conflicts. The first scheme (SAMS) deals with supporting conflict-free 1D vector memory access for strides from two different stride families. The second proposal (2DSMM) explores hardware schemes to support 2D stride conflict-free vector memory accesses. Besides formal description, the hardware prototyping of the proposed schemes is also presented. The mathematical proof of corresponding conflict-free properties is shown in Appendices A & B.

Chapter 4 proposes a system-level design to bridge the discrepancy between data representations in memory and those favored by SIMD processors by customizing the low-level address mapping. The HW SAMS parallel memory scheme is extended to provide both AoS and SoA views of the structured data accessed by the processor (multiple layout view). With such multi-layout memory, optimal SIMDization with low overhead, dynamically changing access patterns can be achieved. Experimental results show that the SAMS Multi-Layout Memory proposal has efficient hardware implementation while synthesized using standard cell library, and significantly decreases the dynamic instruction count and execution time, for representative workloads that operate heavily on array-based data structures.

Chapter 5 presents a novel *elastic pipeline* design that minimizes the negative impact of on-chip memory bank conflicts on system throughput. A hardware *bank-conflict aware warp scheduling* technique is also designed to avoid bank conflicts clustering. Simulation results show the elastic pipeline together with the co-designed warp scheduling dramatically reduces the pipeline stalls and improves overall system performance for benchmarks with heavy bank conflicts, at trivial hardware overhead.

Chapter 6 presents a holistic off-chip memory access efficiency optimization framework. Based on the framework, an adaptive DRAM access granularity scheme to exploit horizontal locality and reduce the memory access interference among cores is proposed. Experimental results show that the proposed techniques effectively improve the DRAM efficiency and overall system performance, with negligible hardware implementation cost.

Finally, **Chapter 7** summarizes our findings and directions for future research.

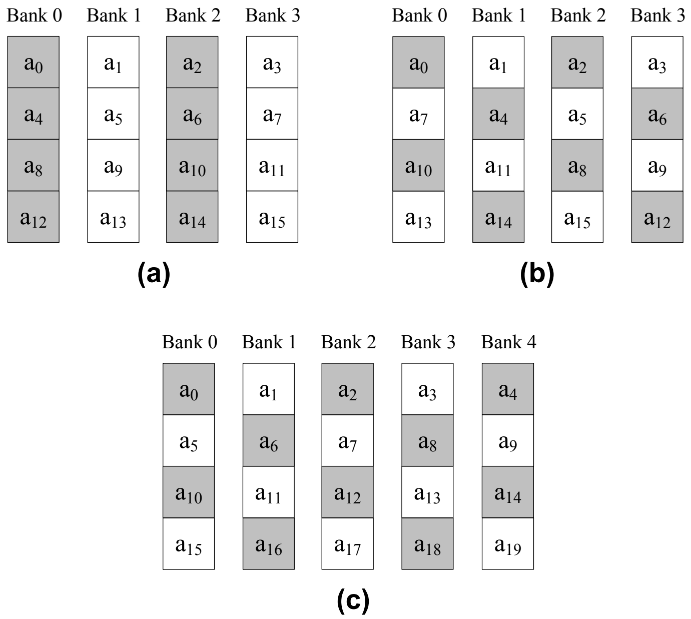
2

Related Work

As discussed in Chapter 1 one of the most critical design challenges in data parallel processors is imposed by the memory subsystem, expected to deliver sustained high bandwidth at reasonable latency [42, 79, 83]. In this chapter we will review the related state of the art in building data parallel centric memory subsystems. We will emphasize on the problems and existing solutions in contemporary mainstream data parallel architectures. Since this dissertation takes an access-pattern centric, systematic approach based on customizable memory schemes, we will first introduce the state of the art parallel memory schemes, which deal with the organization and low-level address mapping of memory subsystems with multiple memory banks/modules/channels. Then existing techniques on memory access patterns and data layout optimizations will be briefly summarized. At last, a survey of some recent progress in system-level memory access optimizations and, as a case study, the GPU memory access optimizations, is presented. Although the main focus of this dissertation is on programmable data parallel accelerators, we will also examine some relevant memory access optimization techniques from the GPP domain.

2.1 Parallel Memory Schemes

Parallel memories were introduced in early high performance processors [22] and later extensively adopted in vector supercomputers [58, 126]. Nowadays, there is a trend that general purpose systems are utilizing parallel memories in their memory hierarchy, such as the multibank on-chip cache organization in Niagara [77] and Opteron [70], multislice caches in Power processors [87, 133, 139], parallel on-chip eDRAM banks in the VIRAM processor [79], and interleaved DRAM banks in Rambus and other commercial-off-



Bank 0	Bank 1	Bank 2	Bank 3
a_0	a_1	a_2	a_3
a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}
a_{12}	a_{13}	a_{14}	a_{15}

(a)

Bank 0	Bank 1	Bank 2	Bank 3
a_0	a_1	a_2	a_3
a_7	a_4	a_5	a_6
a_{10}	a_{11}	a_8	a_9
a_{13}	a_{14}	a_{15}	a_{12}

(b)

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4
a_0	a_1	a_2	a_3	a_4
a_5	a_6	a_7	a_8	a_9
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}
a_{15}	a_{16}	a_{17}	a_{18}	a_{19}

(c)

Figure 2.1: Stride-2 vector access to three schemes: (a) low-order interleaving; (b) skewing (c) prime

the-shelf, monolithic DRAM modules.

As introduced in Chapter 1, parallel memory schemes are the main means to determine the parallel memory subsystem performance. At each memory hierarchy level, such schemes determine: 1) the memory resource organizations (e.g., the number of parallel memory units); and 2) the mapping from the linear address space to the physical locations (such as the bank id and local address inside the bank). The parallel memory schemes are also referred to as *interleaved schemes* [130], since an important task of the schemes is to distribute data to memory banks in an interleaved manner. The goal of such interleaving is to reduce the probability of bank access conflicts, and hence increase access parallelism among memory units and improve the memory performance.

Vector access, defined by an address stream with a constant offset between any two consecutive addresses[1], is one of the most important memory reference patterns in data parallel applications. However, vector access to parallel memories is often vulnerable to bank conflicts, which results in performance

---

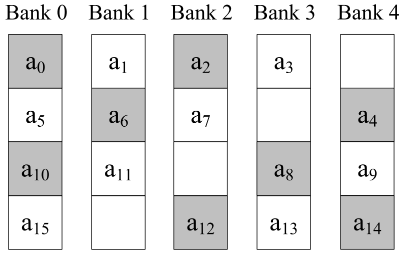
[1] *Vector (memory) access* was used in Chapter 1 in a more general sense to denote SIMD memory access.

degradation. For example, Figure 2.1 shows a stride-2 vector access to three parallel memory schemes. (a) is called *low-order interleaving*, the simplest and most common interleaving scheme optimized for contiguous memory access. For a power of two bank number N, low-order interleaving scheme uses the lowest $n = long_2 N$ bits of the address to select the bank. (b) is a specific case of the *skewing scheme* [36, 82], where each row is rotated right by one element. (c) belongs to the *prime memory scheme*, which employs prime number of banks to store the data elements, in an interleaved manner. Assume the memory subsystem is designed to service 4 accesses per cycle. As can be seen in Figure 2.1, the naive low-order interleaving scheme with 4 banks, will suffer from heavy bank conflicts, and, in consequence, the *effective bandwidth* will be reduced by 50%. In contrast, the other two schemes will successfully avoid such drawback and meet the high throughput requirement.

In fact, Figure 2.1 illustrates the two categories by which most parallel memory schemes can be classified: schemes with *prime* number of banks (e.g., (c)) and with *power of two* banks (e.g., (a) and (b)). The major difference, is whether or not division/modulo operations on prime numbers are involved in address generation[2]. For example, the address generation functions of Figures 2.1(c), a *prime memory system*, can be described as follows:

$$\begin{cases} m(a) = a\%5 \\ r(a) = \lfloor \frac{a}{5} \rfloor \end{cases} \tag{2.1}$$

where $m(a)$ is the bank id, and $r(a)$ is the local address.



Bank 0	Bank 1	Bank 2	Bank 3	Bank 4
a_0	a_1	a_2	a_3	
a_5	a_6	a_7		a_4
a_{10}	a_{11}		a_8	a_9
a_{15}		a_{12}	a_{13}	a_{14}

Figure 2.2: Prime memory system with unused cells

The prime memory systems have been considered in many research projects [81, 86, 117, 118], mainly because of their superiority in supporting conflict-free access for very wide ranges of strides. Indeed, it is straightforward

---

[2] Hence, all parallel memory schemes consisting of non power of two banks can be categorized as *prime schemes*.

to verify that, for a prime memory system with p banks, bank conflicts occur only when the vector access stride is multiple of p; while in all other cases, accesses are conflict-free, and peak throughput can be achieved [22]. However, the shortcoming is that they involve division/modulo operations on prime numbers (as exemplified by Equation 2.1), which are generally difficult to efficiently implement in hardware. Thus, *unused cells* are often introduced in early parallel computers employing prime memory schemes [81, 86], in order to reduce the HW complexity of address computation. Figure 2.2 illustrates the design concept that is adopted in the Burroughs Scientific Processor's prime memory system [81]. Note, the blank squares in Figure 2.2 denote physical memory cells that are not used by the system. The prime bank number p is set to be $2^q + 1$, and the address generation functions are defined as follows:

$$\begin{cases} m(a) = a\%p \\ r(a) = \lfloor \frac{a}{p-1} \rfloor = \lfloor \frac{a}{2^q} \rfloor \end{cases} \tag{2.2}$$

As a consequence, the division by a prime in Equation 2.1 is now converted into a division by a power of two (which is selected at design time), without any additional hardware cost. This simplification is a big advantage compared with the basic prime memory systems adopting low-order interleaving. However, the penalty is $1/p$ memory space being wasted, which is a clear drawback.

To cope with the memory storage utilization issue in the Burroughs Scientific Processor, Gao proposed a prime memory system based on Chinese Remainder Theorem [48]. Observing the number of rows in each bank, M, is power of two ($M = 2^m$) – thus coprime to p, the proposed scheme successfully avoided computing the division by replacing $r(a) = a/p$ with $r(a) = a\%M$. Since $a\%M$ is equal to the right most m bits of the address a, the local address generation involves no hardware logic (similar to the case in Equation 2.2). On the other hand, the bank address computation still remains. As a consequence, the hardware cost is the same, but no memory space is wasted, as compared with the original scheme.

In the meantime, Seznec et al. independently analyzed prime memory systems and revealed similar findings as above in [129]. Moreover, the specific properties of the permutation patterns between processor elements and memory banks were analyzed. Based on the analysis, an efficient permutation network, called *Chinese Remainder Network*, was proposed, for system configurations with N processor elements and memory banks, where N is a product of n distinct prime numbers $N = N_1 \cdot ... \cdot N_n$. Efficient hardware implementations of the Chinese Remainder Network were presented, for both *centralized control* and *self-routing* cases.

Besides prime memory systems, alternative interleaving schemes have also been proposed, without the constraint of the prime memory bank number. For 2D access in raster-graphics memories, Chor et al. proposed a memory organization based on a doubly periodic assignment of pixels to M memory banks according to a *Fibonacci lattice* [31]. The proposed memory organization has the property that the pixels in any rectilinearly oriented rectangle that contains no more than N pixels can be accessed simultaneously. Mathematical analysis was given, to guarantee $M < \sqrt{5}N$. And it was also revealed that M is even less than $2N$ for many practical values. Despite its solid theoretic foundation, however, the proposed scheme still wastes some bandwidth, since the number of banks is usually substantially larger than the number of accessed data. Also, the address computation logic was not discussed in detail in the paper.

In the similar context of 2D access, Kim et al. proposed a parallel memory scheme based on *perfect Latin square*, a new Latin square with good properties useful for parallel array access [72]. The condition for the existence of perfect Latin square was presented. The resulting scheme is able to support conflict-free row, column, diagonal, and square access patterns. The limitation, however, is that the perfect Latin square does not always exist for arbitrary square size, and the number of banks must be a square of an integer.

Another important type of interleaving schemes, *linear skewing schemes*, were proposed for parallel memories in [82] and analyzed in [131, 149]. In a linear skewing scheme, element $a_{i,j}$ of a 2D array is stored in memory bank $\lambda_1 \cdot i + \lambda_2 \cdot j$ $(\lambda_1, \lambda_2 \in \mathbb{N})$, a linear combination of the array coordinates. The linear scheme generalizes for higher dimension arrays [149]. Similar to the linear skewing schemes, Jorda et al. proposed a class of *semi-linear* skewing schemes, and in particular the *bi-base skewing scheme* which allows $(N-1) \times (N-1)$ matrices to be stored in N banks, and supports conflict-free access to row, columns, the forward diagonal and submatrices [66].

The above schemes assume either prime number of banks, or bank numbers to be integers in general. In such cases, the address generation may incur prime number operations (such as modulo, or even division). Although such penalty can be *partly removed*, it cannot be completely *eliminated* and often translates into cost in other forms (e.g., the increased interconnect routing complexity [129]). On the other hand, when the number of banks is power of two, the expensive prime number calculations can are naturally avoided. Therefore, such configurations are attractive and most widely seen in parallel memory organizations of contemporary machines. The drawback, however, is that it is difficult to support conflict-free memory access for many access patterns. For example, memory access with strides across *stride-families* can easily cause bank conflicts, in memory systems with power of two banks.

To cope with bank conflicts of vector accesses across stride families, several techniques have been proposed in the literature, including the use of buffers [34], dynamic memory schemes [35, 36], memory bank clustering [34] and intra-stream out-of-order access [144]. Under the parallel memory system with fully-pipelined memory banks, these methods still work but are subject to some limitations.

The use of buffers [34] is probably the most straightforward solution as it tolerates the bank conflicts by simply buffering the input addresses and output data and collecting the required data after some delay. The buffer depth required depends on the misalignment between the parallel memory scheme used and the vector access stride. If the access stream is distributed evenly between the memory bank of the system, then the peak throughput of one data per memory bank in one cycle might be achieved after a transient startup time. However, since the startup disparity is unavoidable, this solution introduces significant time penalties in case of short access streams. Moreover, the use of buffers and the logic for collecting the correct data items from the buffers could cause substantial hardware overheads.

The dynamic scheme proposed in [35, 36] works well only when the same data set is accessed with single stride family. However, if the data set is to be accessed using different stride families, the penalty of flushing and reloading data between the memory banks and the lower level in the memory hierarchy may not be amortized in some cases, which would result in performance degradation of the system.

The memory bank clustering [34] introduces inefficient use of bank control logic and data routing resources, as a number of memory banks may remain idle during each parallel memory access. For instance, under the assumption that the number of banks is power of two, the number of memory banks used for conflict-free access of two unmatched stride families may be no more than 50% of the available banks. This results in waste of logic resources and power in some cases.

The out-of-order vector access [144] is based on the observation that a long, strided memory reference stream with bank conflicts in sequential order could become conflict-free, if properly reordered. For instance, in a parallel memory system with conflict-free stride-4 access support, a stride-2 stream with 16 memory references could be accomplished by two stride-4 streams with 8 memory references each. Basically the original stride-2 stream is split into two stride-4 sub-streams and the memory system is accessed with by the alternating sub-streams. In this case, the access is conflict-free. The problem with intra-vector out-of-order access is that it requires long vectors for proper operation. In addition, as data items are read out of order, data permutation logic may introduce additional penalties [3].

Some recent research projects also consider manipulating the memory address mapping to improve memory access performance. In the Impulse project [24], physical addresses of discontinuous data are remapped to aliases which are contiguous in the *shadow space*, and references to the discrete data through the aliases are actually performed by the Impulse Memory Controller at the DRAM side. While improving the cache and memory bus utilization, it is not suitable for on-chip local stores, as data at the memory side still remain discontinuous and the efficiency of the data access remains low. Impulse memory also has the coherence problem since it creates aliases for discontinuous data. Similar to Impulse, the active memory system [71] uses the address remapping to create contiguous aliases for discontinuous data, and access the data with their aliases, to hijack the memory hierarchy for better cache behavior. Again, the active memory system is unable to improve the efficiency of the memory access at the physical memory banks/modules/channels. The Command Vector Memory System [32] proposes broadcasting vector access commands to all memory banks/modules, instead of sending individual addresses/data. Despite its inherent support for strided access, the Command Vector Memory System does not consider specialized address mapping schemes to improve memory access parallelism among multiple banks.

Regarding the data alignment problem in GPP SIMD extensions, studies have been done to improve the performance of SIMD devices by relieving the impact of non-contiguous and unaligned memory access from the hardware point of view. For example, Alvarez et al. analyzed the performance of extending the Altivec SIMD ISA with unaligned memory access support on H.264/AVC codec applications [11].

---

[3] Data permutation is not required in the original proposal [144] as there the assumed memory organization is that single datum is read out from the multiple memory banks per cycle, whereas in the organization considered in this thesis multiple data items (equal to the number of memory banks) are read per cycle.

2.2 Access Pattern and Data Layout Optimizations

In Section 2.1, we have briefly reviewed the historic and contemporary work in designing parallel memory schemes, which provides the hardware substrate for parallel memory organizations, and the corresponding low-level address mapping. All parallel memory schemes are optimized for specific subset of memory access patterns. Therefore, it is essential to correctly identify the access pattern information and exploit it for memory access efficiency improvement, by, e.g., utilizing the pattern information to steer the memory schemes.

There has been a large body of research studies and practices in properly obtaining the memory access pattern information. One common way to achieve this is by static analysis, such as the code analysis techniques used in software prefetching [23, 27, 94, 101]. Alternatively, memory access pattern information can also be obtained from programmer annotation. For example, CUDA-lite [143], a programming model enhancement to CUDA [105], enforces programmers to provide annotations describing certain properties of the data structures and code regions designated for GPU execution. The CUDA-lite tools analyze the code along with these annotations, and determine if memory bandwidth can be conserved and latency can be reduced by utilizing any special memory types and/or by massaging memory access patterns.

Memory access patterns can also be captured by profiling [95, 153] or runtime [10, 93], which make use of the information provided by standard hardware performance counters. Also, specialized hardware can be employed to dynamically detect the access patterns (e.g., strides) in a more direct manner, such as the cases of stream buffers [27, 116, 146].

Once memory access patterns have been statically identified, a following-up opportunity arises in that algorithms and/or data structures can be adapted, in order to suit the hardware memory hierarchy. The rationale of the former is that, access patterns are changed under different versions of the same algorithm – therefore the version most friendly to the memory hierarchy is chosen [14, 21, 49, 84]. The latter, adapting data structures to memory hierarchies, has led to an important area of memory-hierarchy conscious program data layout optimizations. Such optimizations can be done by hand, with optimized library support. For example, specialized data structures have been designed for database applications running on GPPs [21] and the Cell processor [14, 49]. In Glift [90], an abstraction and generic template library has been created for defining complex, random-access GPU data structures.

Beyond manually tuning data structures, recently we have also seen compiler optimizations to facilitate data reorganization. For example, to address the data alignment problem in GPP SIMD extensions, studies have been done to improve the performance of SIMD devices by relieving the impact of non-contiguous and unaligned memory access from the compiler point of view. Specifically, Ren et al. proposed a compiler framework to optimize data permutation operations by reducing the number of permutations in the source code with techniques such as permutation propagation and reduction [124]. Nuzman et al. developed an auto-vectorization compilation scheme for interleaved data access with constant strides that are powers of two, by reorganizing the key data structures [106].

It has been shown that the process of data layout optimization can also be automated, with customized memory allocation libraries and compiler analysis and transformations. Truong et al. proposed two data layout techniques to improve locality for heterogeneous data structures allocated dynamically [142]: *field reorganization*[4] and *instance interleaving*[5]. Chilimba proposed two strategies – *cache-conscious reorganization* and *cache-conscious allocation*, and the corresponding semi-automatic tools that use these strategies to produce cache-conscious pointer structure layouts [29, 30]. Zhong et al. proposed *array regrouping* and *structure splitting* using whole-program reference affinity, which measures how close a group of data are accessed together in a reference trace and gives a hierarchical partition of program data [153]. Golovanevsky et al. implemented C structure optimizations in GCC (i.e., the GCC *struct-reorg* pass), to adapt the layout of a data-structure to its access patterns in order to better utilize the cache by increasing spatial locality [52].

It is important to note that, despite the advantage of no hardware overhead, software data layout optimization techniques have certain drawbacks. Manual data structure manipulation and data reorganization put a huge overhead on the programmer, in requiring detailed knowledge about the low-level memory hierarchy. For example, the programmer has to take charge of the logical data layout and its mapping to (usually parallel) memory hardware. (Semi)automated approaches require less programmer intervening, however, are limited in scope since they require high-level program information that is hard to accurately obtain by profiling/static analysis. Indeed, most prefetching schemes either target only loops with statically-known strided accesses, or they rely on the access patterns detected during profiling being unchanged

---

[4] to group together data structure fields which are referenced together in the data structure declaration

[5] identical fields of different instances of a data structure often referenced together are grouped together dynamically

in real executions, which may not be the case since access patterns can be input-data dependent. Moreover, in the context of vector memory access in data parallel accelerators with parallel memory banks, conflict-free access can not be achieved by only software data layout adaptation without the support of specialized hardware memory schemes for many access patterns.

2.3 Off-chip Memory Access Scheduling

Off-chip memory access has become more and more important in modern computing systems, due to the expanding gap between the processor speed and off-chip memory access latency, and between increasing on-chip processing parallelism and off-chip memory bandwidth. In such systems, DRAM access requests are usually buffered inside the memory controller, forming a dynamic access window. Various scheduling policies can be applied, to select the proper access in the window to be issued to the DRAM chips, in the hope of improving DRAM access efficiency. This section will review the recent advances in off-chip memory access scheduling techniques.

Sophisticated out-of-order DRAM scheduling schemes have been extensively studied for DRAM efficiency improvement. Existing systems commonly employ variants of the FR-FCFS (first-ready, first-come first-serve) scheduling policy [125], which prioritizes row-hit requests over other requests. Although FR-FCFS was proposed originally in a thread-unaware context, recent work has shown that it is also effective in improving DRAM throughput and overall system performance in massive multithreaded GPUs [152].

Since thread-unaware memory access schedulers aim at maximizing DRAM throughput, they have been shown to be ineffective in guaranteeing fairness in general-purpose multicore and multithreaded systems [100, 104, 122]. In contrast, researchers have recently designed thread-aware memory schedulers to improve fairness as well as system throughput [73, 74, 102, 103]. Unlike thread-unaware memory schedulers, one of the major insights of thread-aware memory schedulers is, different threads have different characteristics regarding their DRAM access behavior (such as, bandwidth, regularity and the likelihood to interfere with others). Therefore, threads can be categorized into different classes (e.g., memory-intensive/non-memory-intensive), by either static analysis, or dynamic capturing, or both. With such classification, schedulers can try to generate the optimal memory access scheduling during threads execution, therefore improve both DRAM throughput and fairness.

2.4 Memory Access Optimizations on GPUs

This section reviews some recent studies on memory access optimizations on GPUs, which is a typical contemporary data parallel accelerator architecture. The GPU memory access suffers from two major problems: the on-chip shared memory bank conflicts, and the off-chip DRAM access efficiency, which will be discussed in the following.

2.4.1 Avoiding GPU Shared Memory Bank Conflicts

Bank conflicts form an important problem in vector processors and it has been studied intensively in the literature. To cope with bank conflicts of vector access across stride families, several techniques have been proposed, including the use of buffers [34], dynamic memory schemes [35, 36, 55, 56], memory banks clustering [34], and intra-stream out-of-order access [144], just to name a few. Some of the existing techniques may be considered for GPU memory bank conflict avoidance, however subject to certain limitations. For example, one possible solution based on existing techniques is to add buffers in front of each shared memory bank to create a small access window. Subsequently, out-of-order scheduling techniques may be applied to resolve the bank conflicts, within such window. Similar techniques have been successfully used in other scenarios, such as the DRAM memory controller scheduling [125]. However, in the context of GPU fine-grain multithreaded SIMD processing, this technique is not applicable, because distributed out-of-order accesses in parallel shared memory banks create diverged execution orders for threads inside a warp/subwarp, effectively breaking the subwarp boundaries in the SIMD datapath. This often leads to conflicts in the register file banks at the writeback stage, which stall the pipeline in the end. In this case, shared memory bank conflicts are not resolved but just *postponed* to later pipeline stages.

GPU on-chip shared memory efficiency is also impacted by physical data layout. Programming practices exist to avoid or relieve on-chip bank conflicts, by manually crafting the data layout at source code level (e.g., zero-padding for shared memory data structures) [76, 108]. While such optimizations have been adopted in practice, they lay a nontrivial burden on programmers. Detailed knowledge of the shared memory hardware is required, and sometimes major modifications to the source code is needed to apply such optimizations. Besides, they also create portability issues when platforms have different shared memory configurations. Moreover, static code optimizations are unable to relieve bank conflicts for patterns which cannot be determined statically. Examples are conflict patterns that change dynamically, or that are dependent on runtime parameters. Recently, we see some work in automating such manual optimizations [151]. Such high-level optimizations have the potential to relieve the shared memory bank conflicts burden from programmers, but are still limited by their static nature.

2.4.2 GPU DRAM Access Optimizations

Recently we have seen some **software optimization frameworks** to handle the complexity of GPU programming and optimizing. Ocelot [39] is a dynamic compilation framework designed to map the NVIDIA CUDA applications onto diverse multithreaded platforms. It includes a dynamic binary translator from PTX code to x86 and other ISA.

PTX transformations, such as thread-fusion used in MCUDA [136] and GPGPU Compiler [151], have also been proposed. The thread-fusion technique attempts to merge small threads into larger ones, so that more spatial locality, at various memory hierarchy levels (such as registers and shared memories), may be created for each kernel thread execution. By exploiting the extra locality, system performance can be improved.

Main memory access efficiency is also impacted by physical data layout in DRAM rows/banks/channels. Programming skills to improve off-chip memory access efficiency exist, such as manually changing the data layout of key data structures in main memory at source code level [76]. Unfortunately, automating such optimizations using program analysis for main memory data is more difficult than the on-chip shared memory case and such work is yet to be seen for GPUs.

From system architecture point of view, GPU memory performance optimizations have been addressed at different levels: between the GPU accelerator and host CPU, inside the GPU core, in GPU on-chip interconnection, and at the GPU memory controller side.

At the **CPU-GPU interconnect** level, current GPGPU platforms suffer significantly from the relatively low bandwidth between the host CPU and the accelerator GPU attached to the CPU through the system bus [76]. The research and development efforts can be classified into two categories: 1) to improve the efficiency of the CPU-GPU communication based on existing loosely-coupled system bus configuration [50]; and 2) to integrate the CPU and GPU onto the same die [7, 12].

At the **GPU core** level, there have been some studies in applying *prefetching* techniques for GPUs [127], however only data *inside* a thread was considered. A recent study on GPU prefetching proposed *Inter-Thread Prefetching (IP)* [88], recognizing the GPU specific locality among parallel threads. *IP* focused on *latency reduction* using speculation, assuming inadequate parallelism to hide memory latency.

Memory coalescing [108] is a hardware mechanism in NVIDIA GPUs to buffer and merge *intra-warp* memory accesses. It is an effective way to capture inter-thread data locality, however, its effectiveness is limited to a limited scope (e.g., half/single-warp, depending on the GPU generation).

At the **GPU on-chip interconnect** level, the work [152] also addresses the *memory access streams interleaving* problem, using a customized flow control design optimized for this scenario. A similar work [16] analyzed the *many-to-few-to-many* traffic pattern in typical GPU configurations, and proposed throughput-effective on-chip interconnection. The key observation is that, the traffic pattern is *unbalanced* between the GPU core nodes and memory controller nodes and such unbalance is fixed at design time. Therefore, conventional interconnect network designs, which assume a uniform traffic pattern, can be improved, with the unbalanced traffic pattern information. The paper explored such improvement and showed that the throughput per unit area is improved with the redesigned interconnect.

At the **GPU memory controller** side, recent work has shown that out-of-order DRAM scheduling schemes, such as FR-FCFS [125], are also effective in improving DRAM throughput and overall system performance in massive multithreaded GPUs [53, 152]. Needless to say, the performance of a DRAM controller can often be limited by its relatively small memory requests window size, which is constrained by the *out-of-order* (OoO) memory scheduler hardware cost. This is especially the case in the context of massive (at the order of thousands) in-flight DRAM accesses in typical GPUs, which tend to benefit from very large scheduler window sizes.

2.5 Summary

In this chapter, we have presented an overview of existing parallel memory schemes, which provide the hardware substrate for parallel memory banks/modules/channels organizations, and the corresponding low-level address mapping. Since parallel memory schemes optimize memory access according to specific access patterns, existing methodologies in extracting memory access pattern information were briefly discussed, with enumeration of software data layout optimizations based on the access patterns. Furthermore, off-chip memory scheduling techniques were also reviewed, due to the growing importance of off-chip memory access and the performance impact of off-chip memory access scheduling. Finally, we studied existing solutions on two major GPU memory issues: the on-chip shared memory bank conflicts, and the off-chip DRAM access efficiency.

3

Conflict-Free Parallel Memory Schemes

In this chapter, we analyze the problem of supporting conflict-free access for multiple stride families in parallel memory schemes targeted for SIMD processing systems. We propose two novel hardware parallel memory schemes to deal with memory bank conflicts incurred by vector memory access. The first scheme (SAMS) deals with supporting conflict-free 1D vector memory access for strides from two different stride families. The second one (2DSMM) moves one step further in that, it explores hardware schemes to support 2D stride conflict-free vector memory accesses. Besides formal descriptions of the proposed schemes, the hardware implementation prototyping are also presented. The corresponding mathematical proofs are listed in Appendixes A and B.

The remainder of the chapter is organized as follows. In Section 3.1, we present the motivation of the proposed parallel memory schemes. The construction procedure and the mathematical formulas of the SAMS and 2DSMM schemes are described in Sections 3.2 and 3.3, followed by their hardware implementation and synthesis results in Section 3.4. The major differences between our proposal and related works, the design space and applications are briefly discussed in Section 3.5. We conclude the chapter in Section 3.6.

3.1 Motivation

In this section, we will first introduce the limitation in non-redundant[1] parallel memory schemes, which motivates us to the *SAMS* scheme. Then we will present the conflict-free requirement in memory access to 2D data, which leads

---

[1] In this chapter *non-redundant* schemes refer to those with the same number of memory units as the required data elements number per each parallel access (usually equals $\#lanes$).

to the design of *2DSMM* scheme. To facilitate our discussion, we use the following definitions of some general terms in parallel memory systems, and the specific ones used in our SAMS scheme.

3.1.1 Definitions

Definition 1. A stream of independent memory accesses issued by the SIMD processor in parallel is called a **vector access**. A vector access could be either regular (with constant stride) or irregular (such as the scatter/gather memory access). However, we only discuss regular vector accesses in this chapter.

Definition 2. **Base address** is the first memory address in a given regular vector access.

Definition 3. **Stride** is the constant interval between subsequent memory addresses in a given regular vector access stream.

Definition 4. **Unit stride** denotes stride 1.

Definition 5. A **stride family** is a set of infinite number of strides, $\{S \| S = \sigma \cdot 2^s,\ s \in \mathbb{N},\ \sigma \text{ is odd}\}$. This follows the definitions given in [35, 36, 144].

Definition 6. The exponential part of the stride family $\{S \| S = \sigma \cdot 2^s,\ s \in \mathbb{N},\ \sigma \text{ is odd}\}$, s, is called the **stride family number**. The stride family number completely defines the set of strides belonging to a stride family. For example, stride family with family number 0 (we will use the phrase "stride family s" for short of "stride family with family number s" hereafter) is the stride set $\{1,\ 3,\ 5,\ 7, \cdots\}$ while stride family 1 is $\{2,\ 6,\ 10,\ 14, \cdots\}$.

Now we will give some definitions specific in our proposed SAMS scheme. Suppose a is the linear address and $m(a)$ is the module assignment function of a parallel memory scheme with N memory modules.

Definition 7. If address a satisfies $m(a) = m(a + \delta)$ (where $\delta < N$), address a has **forward-affiliation**.

Definition 8. If address a satisfies $m(a) = m(a - \delta)$ (where $\delta < N$), address a has **backward-affiliation**.

Definition 9. Forward-affiliation and backward-affiliation always occur in pairs. For instance, if address a has forward-affiliation ($m(a) = m(a + \delta)$) then the address $a + \delta$ has backward-affiliation. We call address a and its affiliated address ($a + \delta$) an **affiliation-pair**.

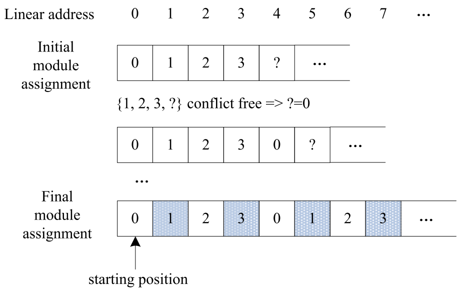


Figure 3.1: Inherent limitation in multimodule memory assignment

Now, let us examine the meaning of affiliation. If there exist forward/backward-affiliations in a memory scheme, then the scheme is not conflict-free for parallel N unit-stride accesses at arbitrary base addresses. For instance, unit-stride accesses starting at addresses with forward-affiliation will result in module conflicts.

Definition 10. If address a is associated with only one single instance of affiliation (backward or forward), then it is a **single-affiliation** address.

Definition 11. If there exist addresses in a non-redundant parallel memory scheme with single-affiliation and none of them has multi-affiliation, then it is a **single-affiliation scheme**.

3.1.2 Non-Redundant Parallel Memory Schemes Limitations

In traditional matched parallel memory schemes, it is impossible to simultaneously support both parallel unit-stride and arbitrarily strided[2] access orders [144]. Figure 3.1 shows an example with four memory modules[3]. Under the constraint of unit-stride conflict-free access, the module assignment function of the scheme is completely fixed. Note in Figure 3.1 the constant repeat of module assignment pattern of the first four addresses. When the system is accessed with stride 2, half of the memory modules are not utilized (the shadowed cells in Figure 3.1). One additional limitation, not shown in Figure 3.1, is that any interleaving scheme optimized for even-stride conflict-free access could not support conflict-free unit-stride access at arbitrary base addresses.

---

[2] By *strided* we refer to even strides in this chapter, as odd strides (including unit stride) conflict-free accesses are well supported by the simple low-order interleaving scheme [144].

[3] In this chapter *module* and *bank* are used interchangeably, to denote a memory bank with independent local address decoder.

There is a large number of strided vector accesses in many scientific and engineering applications which have significant impact on the performance of the workloads on traditional vector supercomputers [15]. In the meanwhile, we certainly could not neglect the unit-stride access pattern, as it is the most common one in vectorized scientific and engineering applications [60, 78, 138]. Even in vectorized SPEC95 benchmarks it is the second most frequent stride [115][4]. Furthermore, there are many occasions in which simultaneous support of both unit-stride and strided memory accesses is desired, as the same data block is accessed with different stride types. When we have to access data in parallel memories with both unit and even stride, the problem occurs that we have to either modify the interleaving scheme (that is, to redistribute data to memory modules in a different way), or to have the scheme optimized for conflict-free access with one type of access while suffer from the non-conflict-free access with the other. The former would incur data flushing into and reloading from the lower level memory in the memory hierarchy whenever there is a change of access stride, whereas the latter would introduce processor cycles wasted on waiting for the vector access.

As far as unit-stride access on stride-optimized parallel memory scheme is concerned, it is interesting to examine the *affiliation* properties of the scheme. Note that in single-affiliation schemes (defined in last section), a single-affiliation address belongs to only an affiliation-pair. Single-affiliation parallel memory schemes make sure the module conflicts under unit-stride access are moderate in the sense that, if an address in linear address space causes module conflict within access at one base address, then it will never cause any other conflict within access at a different base address. In Section 3.2, we shall illustrate how to make use of single-affiliation parallel memory schemes to construct a memory system capable of supporting conflict-free strided accesses from multiple stride families.

3.1.3 Strided Access in 2D Environment

Previous work on strided access is mostly focused on one dimensional case(i.e. strided vector access). However, with the increasing requirements for both stronger computational power and higher memory bandwidth from some engineering and scientific computing domains such as multimedia, radar data processing, and fluid dynamics, aggressive hardware support for more complicated memory access patterns is desirable, particularly for data-level multi-

---

[4] The most frequent access pattern is scalar access in both SpecInt95 and SpecFP95, defined as “stride-0” in [115].

processing systems such as vector and SIMD processors. For instance, in the SARC research project [123], the mechanism of two dimensional strided memory access is to be devised to speed up memory access for the vector coprocessor [85], as illustrated in Figure 3.2. B in Figure 3.2 denotes the *base address*, ES *element size*, HS/VS *horizontal/vertical stride*, and HGS/VGS *horizontal/vertical group stride*.

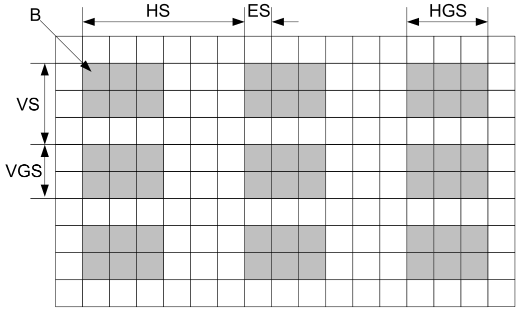


Figure 3.2: SARC vector coprocessor memory addressing

In this chapter, we propose the 2D strided Multiaccess Memory (2DSMM) scheme to address the issue of strided access in two dimensional environment. It has the following design goals:

1. ***Various access patterns:*** The 2DSMM should support as many useful access patterns in two dimensional environment as possible. It is identified that the strided row, strided column, strided diagonal and strided block access patterns are of most interest [118]. Besides strided access, it should also support continuous data access in parallel such as unit-stride access in one dimension or access to a continuous block. In our view these features are particularly desirable. On one hand, the 2DSMM could be used potentially as a buffer between the processor and the main DRAM memory and therefore it has to be able to exchange data with the DRAM memory efficiently. On the other hand, the continuous block access is a very frequently used access pattern in many applications of our concern.

2. ***Parallel strided access with no restrictions on the starting position:*** It should support strided access at any starting point. However some restrictions for continuous data access are indispensable in our scheme and we think they are acceptable for two reasons. First, as the continuous data access is mainly designed for the data exchange between

the 2DSMM buffer and DRAM, the restrictions on starting position will have no negative effects on the traffic between 2DSMM buffer and DRAM if the starting position is aligned to DRAM row boundaries. Second, the starting position restrictions in our scheme would probably not cause a big problem for the continuous access traffic between the processor and 2DSMM memory[5] for the domain-specific applications targeted by our scheme with the help of a compiler.

3. ***Simple hardware implementation:*** The implemented logic for the address generation should be simple and fast, because it is in the critical path of memory access.

4. ***No redundancy in memory module number:*** In most prime memory systems, there are more memory modules than the number of data accessed simultaneously. To increase the utilization of memory modules, our 2DSMM scheme would provide *matched* parallel access to all memory modules (i.e. #data elements required per parallel access = #memory modules). Another common problem with the prime memory systems are the existence of memory holes [86,117] which would also be avoided in our non-redundant system.

3.2 Single-Affiliation Multiple-Stride Memory Scheme

In this section, we propose SAMS, the Single-Affiliation Multiple-Stride conflict-free parallel memory scheme. SAMS aims at supporting conflict-free unit-stride and strided memory accesses simultaneously, by first constructing a single-affiliation interleaving scheme, and then making data lines wider to solve the module conflicting problem in unit-stride access, which exists in the single-affiliation scheme.

3.2.1 Moving from Conflict-Free to Low Degree of Affiliation

Many existing non-redundant, dynamic schemes, such as the XOR scheme [35, 36], can support conflict-free access for both unit-stride and strided patterns, with *dynamic (re)configuration* of the scheme parameters. However, when the scheme parameters are configured for strided access at one time, conflict-free unit-stride access is not supported simultaneously. Now that it is difficult to

---

[5] In this chapter, we use *2DSMM buffer* and *2DSMM memory* interchangeably, to denote the processor's local buffer.

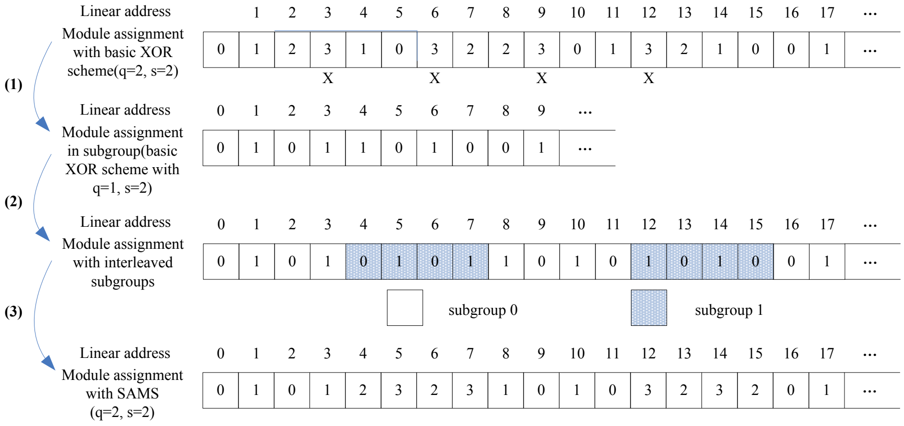


Figure 3.3: SAMS single-affiliation scheme with $q = 2,\ s = 2$

add conflict-free unit-stride access support at once, we propose to do this in two steps. First, we relax the constraint from *conflict-free* to *low degree of affiliation*, for unit-stride access. Second, for each and every memory module, we rearrange the groups of conflicting data items in unit-strided access into wide data lines so that they could be referenced during one access. The philosophy behind this idea is to restructure the memory modules. Traditionally, the memory module in a multimodule memory system is treated as a linear (one-dimensional) structure, and a memory interleaving scheme maps the linear address space from processor's view into multiple linearly structured storage units. In our approach, instead of one-dimensional, we model each memory module as a two-dimensional structure. Therefore, the aforementioned limitation in parallel memory schemes could hopefully be resolved by introducing a new dimension of access parallelism, namely the module data line. If the conflicting data items are located in the same data line, then they could be accessed in parallel with proper shuffling and selecting operations. The reason why low-affiliation schemes are preferable is that, as the degree of affiliation increases, the module data line grows wider because it has to be wide enough to accommodate all conflicting items. Consequently, the hardware for choosing the proper one(s) from the data line and shuffling data items from different modules becomes more complicated.

3.2.2 Hierarchical Single-Affiliation Parallel Memory Scheme

We propose to construct a single-affiliation scheme for SAMS hierarchically. For a parallel memory system with 2^q memory modules, when $s > q$ (s is the *stride family number* defined in Section 3.1.1), SAMS adopts Harper's XOR scheme [35] as it is single-affiliation in this case. However, as it is not a single-affiliation scheme when $s \leq q$, some modifications must be considered. Figure 3.3 illustrates one example of how to build SAMS single-affiliation scheme based on Harper's XOR scheme [35] (referred as "basic XOR scheme" in the figure). The construction process is described as follows:

(1) Divide 4 (i.e. 2^q) memory modules into 2 (i.e. 2^{q-s+1}) subgroups, with each group deployed with basic XOR scheme configured with 2 (i.e. 2^{s-1}) modules and stride 4 (i.e. 2^s).

(2) Interleave the two groups, at the granularity of 4 (i.e. 2^s). Now, the module assignment looks like that of the XOR scheme configured with 2 (i.e. 2^{s-1}) modules and stride 8 (i.e. 2^{q+1}).

(3) Combine the two groups and make uniform module index by merging the subgroup module index and the group index.

As Figure 3.3 suggests, there are cases of four items affiliated with each other (marked with "X"), when the basic XOR scheme is used. The affiliation problem is resolved when we shrink the number of modules in a group, which in turn introduces several (2 in the figure) subgroups of modules. Therefore, steps (2) and (3) in Figure 3.3 interleave them at the subgroup level and merge them into a unified scheme. After that, we could finally get a single-affiliation scheme, as shown at the bottom of the figure.

In the following, we provide the mathematical description of the above construction process.

- *module assignment function:*

$$m(a) = \begin{cases} a\%2^q, & s = 0 \\ \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}\right\rangle, & 1 \leq s \leq q \\ \left(a \otimes T_{H_{q,s}}\right)\%2^q, & s > q \end{cases} \qquad (s,\ q \in \mathbb{N}) \tag{3.1}$$

where, a is the n bit linear address, and 2^q is the number of memory modules in the SAMS scheme; s is the *stride family number*, which is the exponent part of the stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$ to be supported with conflict-free access by the scheme; a_i is the i-th bit of a; $m(a)$ is the module assignment function which has q bits. The notation $x\%y$ means x modulo y, and $< \ldots, \ldots >$ denotes binary bits concatenation. $T_{H_{x,y}}$ is the XOR scheme address transformation matrix taken from [35].

$$T_{H_{x,y}} = \prod_{k=0}^{\min(x,y)-1} T_{k+\max(x,y),\,k}$$

where $T_{i,j}$ is defined to be the identity matrix with a single off-diagonal 1 in $T(i,j)$. The binary matrix T is arranged in a form that the bottom-right element is $T(0,0)$, and the row index grows when moving up and the column index grows when moving left so that the top-left element is $T(l-1, l-1)$ (assume T is $l \times l$ in size). For example,

$$T = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = T_{1,0} \cdot T_{2,1}$$

The $\otimes$ symbol in this chapter is used for binary vector-matrix multiplication. For instance, consider

$$a = 7,\ T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

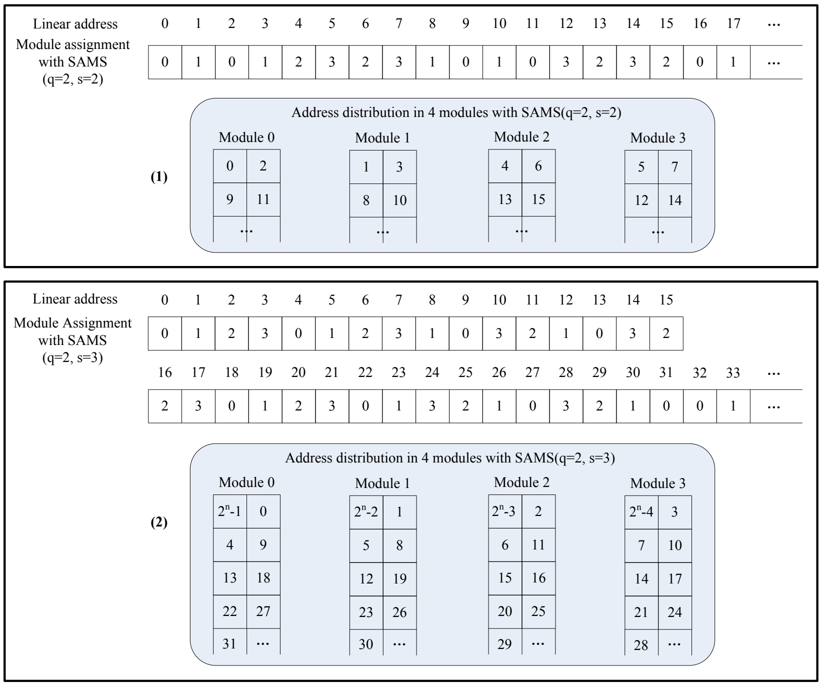
then

$$a \otimes T = [1\ 1\ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = [1\ 1\ 0] = 6\ .$$

The objective of SAMS module assignment function is to make sure that, on one hand, the scheme is conflict-free for stride family $\{S \| S = \sigma \cdot 2^s,\ \text{with}\ \sigma\ odd\}$; while on the other hand, there are at most two data references going to the same module on a parallel unit-stride access.

3.2.3 Solving Module Conflicts in Single-Affiliation Schemes

As described in Section 3.2.1, the construction of a single-affiliation scheme is just the first step of SAMS. To cope with the module conflicts in the single-affiliation scheme, we have to make the modules data lines wider in order to accommodate the conflicting data items. Furthermore, we have to arrange the


(1)
Address distribution in 4 modules with SAMS(q=2, s=2)
(2)
Address distribution in 4 modules with SAMS(q=2, s=3)

Linear address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	···
Module assignment with SAMS (q=2, s=2)	0	1	0	1	2	3	2	3	1	0	1	0	3	2	3	2	0	1	···

Module 0		Module 1		Module 2		Module 3	
0	2	1	3	4	6	5	7
9	11	8	10	13	15	12	14
···		···		···		···	

Linear address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
Module Assignment with SAMS (q=2, s=3)	0	1	2	3	0	1	2	3	1	0	3	2	1	0	3	2			
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	···
	2	3	0	1	2	3	0	1	3	2	1	0	3	2	1	0	0	1	···

Module 0		Module 1		Module 2		Module 3	
2^n-1	0	2^n-2	1	2^n-3	2	2^n-4	3
4	9	5	8	6	11	7	10
13	18	12	19	15	16	14	17
22	27	23	26	20	25	21	24
31	···	30	···	29	···	28	···

Figure 3.4: SAMS data arrangement example with 4 modules

data properly in the two-dimensional storage, as shown in Figure 3.4. Note 2^n in the figure is the capacity of the multimodule memory system. Part (1) of the figure shows the linear address distribution in 4 memory modules which satisfies both unit-stride and stride-4 family conflict free access, and part (2) illustrates the linear address distribution which satisfies both unit-stride and stride-8 family conflict free access. We could see from the figure that the idea of SAMS interleaving scheme virtually introduces a third dimension of address flexibility, the offset in the row (data line), besides the module index and *row address*[6]. The guideline of items placement in module row and offset is simple yet effective: to pack the conflicting items into the same row while maintaining the natural order of the items in the local modules. As there is at most a pair of conflicting items located in the same module on a parallel unit-stride access, a row with width of two is enough for holding them.

---

[6] *Row address* in this chapter denotes the *logical illusion* of module organization where each module consists of rows with the size of one data element or two (in case of the SAMS scheme). In real memory cell arrays, a *physical row* may have much larger size, in which case some bits of the scheme *logical row address* are used to index the *physical row offset*.

The SAMS interleaving scheme consists of three functions: (1) the module assignment function which assigns an item in linear address space to a specific module; (2) the row assignment function which determines the row in which the item is placed; and (3) the offset assignment function which calculates the offset of the item in the row. Since we have presented the module assignment function in Section 3.2.2, we introduce the row assignment and the offset assignment functions below:

- *row assignment function:*

$$r(a) = \begin{cases} \lfloor \frac{a}{2^{q+1}} \rfloor, & s = 0 \\ \lfloor \frac{a}{2^{q+1}} \rfloor, & 1 \leq s \leq q \\ \left(\left(\lfloor \frac{a}{2^{q}} \rfloor + 1 \right) \% 2^{n-q} \right) / 2, & s > q \end{cases} \qquad (s, \, q \in \mathbb{N}) \tag{3.2}$$

- *offset assignment function:*

$$o(a) = \begin{cases} a_q, & s = 0 \\ a_{s-1}, & 1 \leq s \leq q \\ \overline{a_q}, & s > q \end{cases} \qquad (s, \, q \in \mathbb{N}) \tag{3.3}$$

The notations x/y and $\frac{x}{y}$ mean the quotient of integer division between x and y. Note that n is the number of bits of the linear address of the 2^q memory modules. $r(a)$ has $n - q - 1$ bits, while $o(a)$ is a single bit, because we consider only two pieces of data per data line in the SAMS scheme.

We have proved that the SAMS scheme is capable of supporting both strided and unit-stride vector accesses without module conflicts. The detailed mathematical proofs of these properties are presented in Appendix A.

3.2.4 The Matched SAMS Scheme

In prior sections, we have introduced the SAMS scheme to simultaneously support conflict-free unit-stride and strided memory accesses from one *single* stride family. In this section, we will examine a special case of SAMS with the original hardware scheme input parameter s fixed at design time to q, called *Matched SAMS Scheme*, which is able to support conflict-free vector accesses for strides from *multiple*($>$ 2) stride families without changing the internal data layout.

With $s = q$, the module assignment function, the row assignment function and offset assignment function are simplified as follows:

$$\begin{cases} m(a) &= \left\langle a_q,\ \left(a \otimes T_{H_{q-1,\,q+1}}\right) \%2^{q-1}\right\rangle \\ r(a) &= \frac{a}{2^{q+1}} \\ o(a) &= a_{q-1} \end{cases} \qquad (q \in \mathbb{N}) \tag{3.4}$$

Let us consider $q = 2$, which means the Matched SAMS Scheme with 4 memory modules, for example. With $q = 2$ we have

$$\begin{aligned} m(a) &= \left\langle a_2,\ \left(a \otimes T_{H_{1,\,3}}\right) \%2\right\rangle \\ &= \langle a_2,\ (a \otimes T_{3,\,0})\,\%2\rangle \\ &= \langle a_2,\ a_3 \oplus a_0\rangle \end{aligned}$$

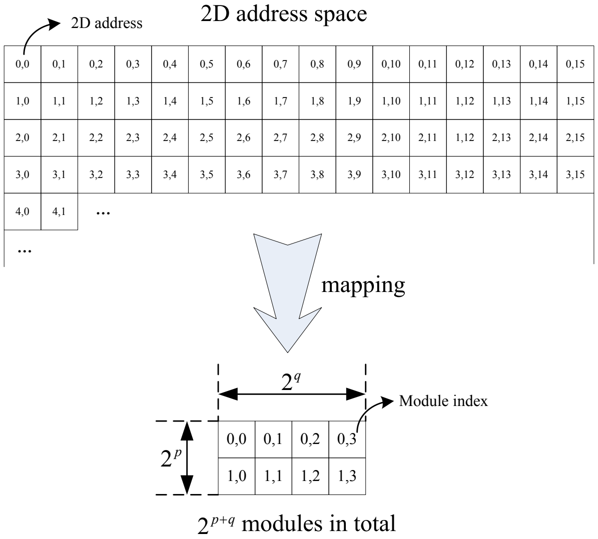
where $< \ldots, \ldots >$ denotes binary bits concatenation. Hence, the address mapping of the Matched SAMS Scheme with 4 memory modules is

$$\begin{cases} m(a) &= \langle a_2,\ a_3 \oplus a_0\rangle \\ r(a) &= a_{n-1:3} \\ o(a) &= a_1 \end{cases}$$

where the symbol $\oplus$ denotes the *binary exclusive or* (XOR) operation. The address mapping of the above example is illustrated in Figure 3.4 (1). Taking the base address 1 for example, the referenced linear address groups for stride 1, 2 and 4 vector accesses are $\{1,\ 2,\ 3,\ 4\}$, $\{1,\ 3,\ 5,\ 7\}$ and $\{1,\ 5,\ 9,\ 13\}$, respectively. As Figure 3.4 shows, all addresses in each group could be accessed in parallel within the Matched SAMS Scheme. Indeed, the Matched SAMS Scheme is capable of supporting conflict-free vector access with strides from more than two stride families. Mathematic proof of this feature is presented in Appendix A.

3.3 2DSMM: 2D Strided Multiaccess Memory Scheme

In a 2D address space environment, 2D addressing schemes is necessitated to identify the locations of the data elements targeted by the 2D vector memory access. A 2D addressing scheme consists of two components: the 2D module assignment function which maps from a 2D address to the module index, and the row assignment function mapping from a 2D address to the module local address, as shown in Figure 3.5.



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	1,12	2,13	2,14	2,15
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15
4,0	4,1	...													
...															

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3

Figure 3.5: 2D addressing scheme

In the 2DSMM scheme, there are $2^{p+q}(1 \leq p \leq q)$ parallel processing lanes and their corresponding load/store units, and 2^{p+q} memory modules which are arranged into a $2^p \times 2^q$ array. All processing lanes access the memory modules simultaneously, with the vertical access stride $VS = \sigma_v \times 2^{vs}$($\sigma_v$ is odd) and horizontal stride $HS = \sigma_h \times 2^{hs}$($\sigma_h$ is odd). The addressing functions of the 2DSMM scheme are described as follows:

2D Module assignment function:

$$\begin{cases} m_v(i,j) = (i \otimes T_v + \alpha + \beta)\,\%2^p \\ m_h(j) = (j \otimes T_h)\%2^q \end{cases}$$

2D Row assignment function:

$$r(i,j) = \left(\frac{i}{2^p}\right) \cdot \left(\frac{N}{2^q}\right) + \frac{j}{2^q}$$

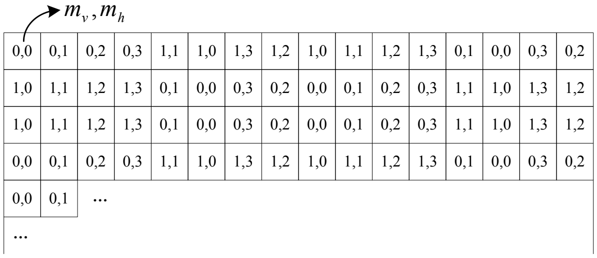
where

$$\alpha = \left(\frac{j}{2^{q+hs}}\right)\%2^p$$
$$\beta = \left(\frac{j}{2^q} \cdot 2^{p-\min(p,hs)}\right)\%2^p$$

$$T_v = \prod_{k=0}^{\min(p,vs)-1} T_{k+\max(p,vs),k}$$

$$T_h = \prod_{k=0}^{\min(q,hs)-1} T_{k+\max(q,hs),k}$$

Coordinates (i,j) is the 2D address in the 2DSMM address space with the size of $M \times N$, where the i coordinate is also referred as *vertical address* and the j coordinate *horizontal address* hereafter. $m_v(i,j)$ is vertical memory module assignment function and $m_h(j)$ is horizontal memory module assignment function. The pair (m_v, m_h) determine which memory module in the $2^p \times 2^q$ module array the address (i,j) will be mapped to, as illustrated in Figure 3.6.



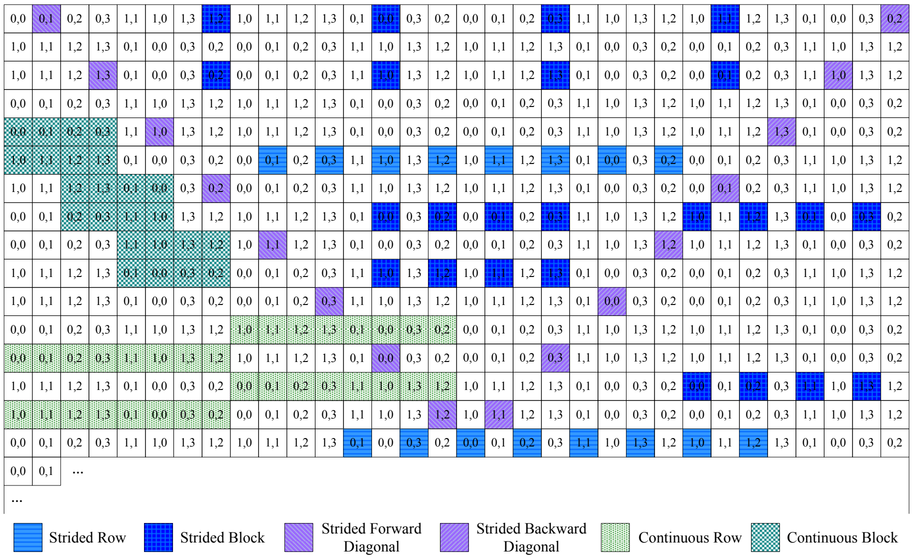
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	...													
...															

Figure 3.6: Example of 2DSMM module assignment for $N = 16, 2^p = 2, 2^q = 4, 2^{vs} = 2, 2^{hs} = 2$

m_v and m_h are also called *vertical module index* and *horizontal module index*, respectively. $r(i,j)$ determines the local address of element (i,j) in memory module (m_v, m_h). T_v and T_h are XOR scheme address transformation matrices taken from [36], as explained in Section 3.2.2. α and β are column rotation factors. α forces coarse grain rotation based on blocks of size $M \times 2^{q+hs}$ while β exerts fine grain rotation based on blocks of size $M \times 2^q$.

The 2DSMM memory works in the following manner. M, N, p and q are system design parameters which could not be modified after the system is designed. More specifically, M and N determine the total capacity of the 2DSMM memory, and p and q determine the number of memory modules. vs and hs are also system parameters which determine T_v and T_h, but they can be configured during runtime.

The proposed 2DSMM scheme supports conflict-free strided row, block, forward diagonal and backward diagonal access and continuous row and block access. Figure 3.7 shows some examples of different access patterns. The



0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	…																													
…																															

Figure 3.7: 2DSMM access examples for $N = 32, 2^p = 2, 2^q = 4, 2^{vs} = 2, 2^{hs} = 2$

detailed mathematical proofs of these properties are presented in Appendix B.

3.4 Hardware Design and Implementation

Above, we have presented the formulas of the SAMS and 2DSMM schemes, which can be *customized* to different access patterns, with the major parameter being the stride family s. For any parallel memory scheme to be practically useful, it is important to have efficient hardware implementation as the scheme logic is in the critical path of every memory access. In this section, we will examine the hardware implementation issues of the proposed scheme.

Figure 3.8 illustrates the organization of parallel memory system based on our customizable parallel memory schemes. The vector processor core issues memory access commands (base address and stride) to the Address Generation Unit (AGU), where the 2^q linear addresses are calculated in parallel and then sent down to the customizable parallel memory system. The eight linear addresses are resolved by the Address Translation Unit (ATU) into eight module assignments, eight row addresses and eight row offset addresses. After that, the eight groups of row-offset pair and eight data items from input data port (on a memory write) go to the address & data switch and get routed to

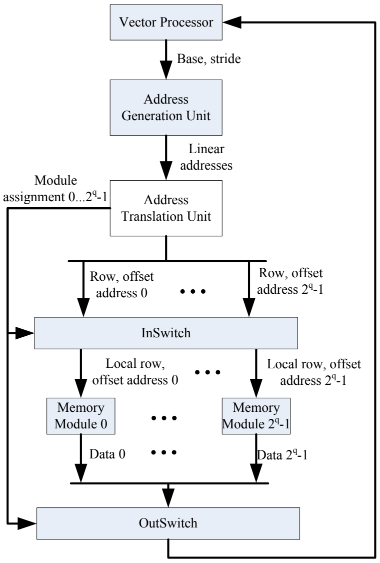


Figure 3.8: Parallel memory system based on customizable HW schemes

the proper memory modules according to their corresponding module assignments. In case of a read memory access, after the memory module read latency the eight read data are fed back to the vector processor via the data switch at the bottom of Figure 3.8.

3.4.1 SAMS Hardware Implementation

We will focus on the hardware implementation of ATU for SAMS, as it is the core of the parallel memory scheme. The remaining components of Figure 3.8 in the SAMS case are either trivial to implement (e.g., AGU), or are common in all parallel memory schemes (e.g., the input and output switches, the evaluation of which will be delayed in later sections), or are independent of parallel memory schemes (e.g., the discrete memory modules). As we have described in Section 3.2, the Address Translation Unit maps the linear addresses $a_i(0 \leq i \leq 2^q - 1)$ to the module-row-offset triples $m(a_i)$, $r(a_i)$ and $o(a_i)$. Figure 3.9 illustrates the logic implementation of the address translation process ($\oplus$ denotes XOR logic). Note, in Figure 3.9, the bits selections without the involvement of s are simply *static* wire selections. The latter which are completely fixed as n and q are fully determined by hardware (such as $a[n-1:q]$ in Figure 3.9 (b)), whereas those with involvement of s result

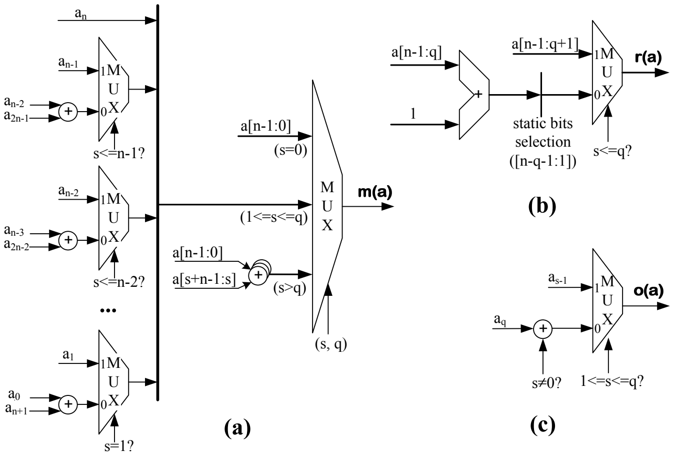


Figure 3.9: SAMS address translation logic: (a)Module assignment, (b)Row assignment, (c)Offset assignment

Table 3.1: Delay and hardware usage of ATU

Configuration		**Delay (ns)**			**Hardware Used**	
n	*q*	Logic Delay	Wire Delay	Total	Slices	LUTs
8	3	1.43	1.01	2.44	18	26
16	3	2.54	0.93	3.47	47	71
25	3	2.86	0.95	3.81	75	119
27	3	2.28	1.75	4.03	76	125
32	3	2.83	1.40	4.23	96	148
64	3	4.28	0.95	5.33	180	292
23	3	2.79	0.95	3.74	67	108
23	4	2.75	0.98	3.73	78	126
23	5	2.71	0.99	3.70	88	140

multiplexors (such as a_{s-1} in Figure 3.9 (c)). And the comparisons in $r(a)$ and $o(a)$ logic(i.e. $s \leq q$, $s \neq 0$ and $1 \leq s \leq q$) could be done and stored a priori, therefore they are not in the critical path. Consequently, the critical path of the row assignment logic is an $n - q$ bit CLA followed by a 2-to-1 multiplexer, and that of the offset assignment is an ($n - q$)-to-1(incurred by a_{s-1}, note $0 \leq s \leq n - q$) multiplexer and a 2-to-1 multiplexer. For the module address assignment function $m(a)$, we have analyzed that after collapsing and merging all the multiplexors in Figure 3.9 (c), we could get the simplified hardware implementation with q independent ($n - q + 1$)-to-1 multiplexors fed by 2-input XOR gates. Hence, the critical path of the module assignment function is a 2-input XOR gate followed by a ($n - q + 1$)-to-1 multiplexer. Notice that the module, row and offset assignment functions work independently, therefore the critical path of ATU is the longest one among the three, which is the $n - q$ bit Carry Look-ahead Adder followed by a 2-to-1 multiplexer in the row assignment function.

It should be noted that, the critical path analyzed above is the one for conflict-free access. For a bank-conflicting vector access, it will be divided into a sequence of conflict-free sub-vector accesses which are then serviced by the parallel memory system back-to-back.

We have implemented the ATU in Verilog and synthesized it using Xilinx ISE 9.1i. The target FPGA device is Virtex2-Pro XC2VP30-7FG676. Table 3.1 summarizes the performance results of our design in terms of delay and hardware utilization. The experiment is done under different configurations with various module capacities (denoted by n) and number of modules (denoted by q). For example, $n = 23$ means the address space (i.e. the capacity) of the multimodule memory system is $8M$ (2^{23}) words and $q = 3$ means there are 8 modules. We could see that the SAMS address translation logic has low critical path delay, which is in the proximity of 4 ns. In addition, the FPGA logic resources consumption is trivial - less than 1%. It is also shown in the table that the critical path delay and resource consumption scale well with the capacity of the parallel memory and the number of memory modules.

3.4.2 Implementation of 2DSMM Scheme

Address Generation Unit (AGU)

The data read/write command from the SIMD processor directly goes to the AGU (Figure 3.8), where the calculations of 2^{p+q} 2D access addresses from the base address and access pattern are carried out. Suppose the base address

1) *Strided Row*:

$$(i,j) \quad (i,j+HS) \quad (i,j+2\cdot HS) \quad \cdots \quad (i,j+(2^{p+q}-1)\cdot HS)$$

2) *Strided Block*:

$$\begin{array}{cccc} (i,j) & (i,j+HS) & \cdots & (i,j+(2^{q}-1)\cdot HS) \\ (i+VS,j) & (i+VS,j+HS) & \cdots & (i+VS,j+(2^{q}-1)\cdot HS) \\ \vdots & \vdots & \cdots & \vdots \\ (i+(2^{p}-1)\cdot VS,j) & (i+(2^{p}-1)\cdot VS,j+HS) & \cdots & (i+(2^{p}-1)\cdot VS,j+(2^{q}-1)\cdot HS) \end{array}$$

3) *Strided Forward Diagonal*

$$(i,j) \quad (i+VS,j+HS) \quad \cdots \quad (i+(2^{p+q}-1)\cdot VS,j+(2^{p+q}-1)\cdot HS)$$

4) *Strided Backward Diagonal*

$$(i,j) \quad (i+VS,j-HS) \quad \cdots \quad (i+(2^{p+q}-1)\cdot VS,j-(2^{p+q}-1)\cdot HS)$$

5) *Continuous Row*:

$$(i,j) \quad (i,j+1) \quad (i,j+2) \quad \cdots \quad (i,j+2^{p+q}-1)$$

6) *Continuous Block*:

$$\begin{array}{cccc} (i,j) & (i,j+1) & \cdots & (i,j+2^{q}-1) \\ (i+1,j) & (i+1,j+1) & \cdots & (i+1,j+2^{q}-1) \\ \vdots & \vdots & \cdots & \vdots \\ (i+2^{p}-1,j) & (i+2^{p}-1,j+1) & \cdots & (i+2^{p}-1,j+2^{q}-1) \end{array}$$

Figure 3.10: 2DSMM address generation patterns

is (i,j), then the addresses for the six different access patterns are listed in Figure 3.10. For these calculations, multiplications[7] of the following are needed.

$$2\cdot VS,\ 3\cdot VS,\ \dots,\ (2^{p+q}-1)\cdot VS$$
$$2\cdot HS,\ 3\cdot HS,\ \dots,\ (2^{p+q}-1)\cdot HS$$

Fortunately, they are not in the critical path, since VS and HS are configured by the vector processor core before the actual memory access starts. Therefore, the multiplications can be carried out in advance by the ALUs of the baseline AGU[8] and and stored in SRAM memories. With the multiplications results at hand, the address generation is implemented in the way as shown in Figure 3.11. We could see that the critical path of AGU contains one 5-input multiplexer and one $\max(\log_2 M, \log_2 N)$ -bit adder, and the logic resource consumption is in the order of 2^{p+q}.

---

[7] Since p and q are fixed at HW design time, hence the multiplications can be implemented with shifters and adders.

[8] Baseline AGU denotes the standard vector AGU without particular support for customizable parallel memory schemes such as SAMS and 2DSMM.

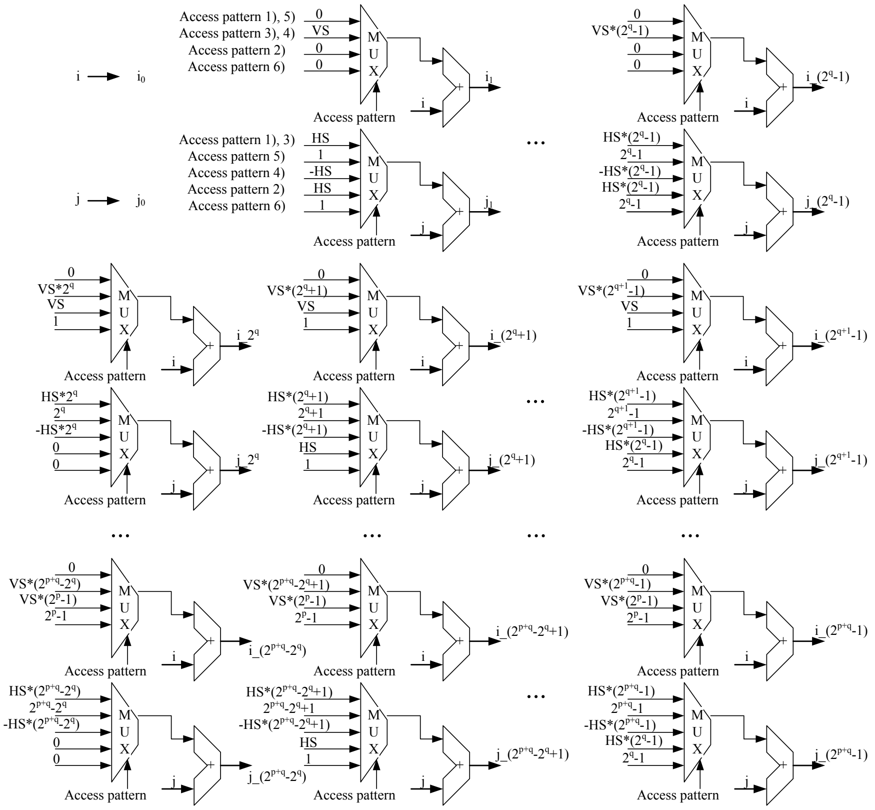


Figure 3.11: 2DSMM address generation logic for six access patterns

Address Translation Unit (ATU)

In the next step, the addresses generated by AGU go to the Address Translation Unit and get processed in parallel. Address translation is the core of the entire 2DSMM mechanism. It accepts two dimensional addresses and transforms them into the physical module numbers and module local addresses. For the sake of discussion convenience, we repeat the 2DSMM scheme formulas in the following.

Module assignment function:

$$\begin{cases} m_v(i,j) = \left[\left(i + (\alpha+\beta)\cdot 2^{vs}\right) \otimes T_v\right] \% 2^p \\ m_h(j) = (j \otimes T_h) \% 2^q \end{cases}$$

Row assignment function:

$$r(i,j) = \left(\frac{i}{2^p}\right)\cdot\left(\frac{N}{2^q}\right) + \frac{j}{2^q}$$

where

$$\begin{aligned} \alpha &= \left(\frac{j}{2^{q+hs}}\right)\%2^p \\ \beta &= \left(\frac{j}{2^q}\cdot 2^{p-\min(p,hs)}\right)\%2^p \\ T_v &= \prod_{k=0}^{\min(p,vs)-1} T_{k+\max(p,vs),k} \\ T_h &= \prod_{k=0}^{\min(q,hs)-1} T_{k+\max(q,hs),k} \end{aligned}$$

Under the constraints that both M and N are power of two, the row assignment function is actually implemented by an adder which adds the proper bit sections of 2D addresses i and j. So the major concern of address translation logic is the module assignment function. The binary vector-matrix multiplication where the structure of the matrix is variable could be implemented in XOR gate array with a multiplexer. As q is fixed, the $\frac{j}{2^q}$ part in both α and β is actually selection of a fixed bunch of bits of j, therefore no extra logic is needed. $\frac{j/2^q}{2^{hs}}$ part in α and $\frac{j}{2^q}\cdot 2^{p-\min(p,hs)}$ part in β could be implemented with multiplexers for dynamic bits selection. The $2^{p-\min(p,hs)}$ part of β would introduce another multiplexer into the critical path, however it could be avoided

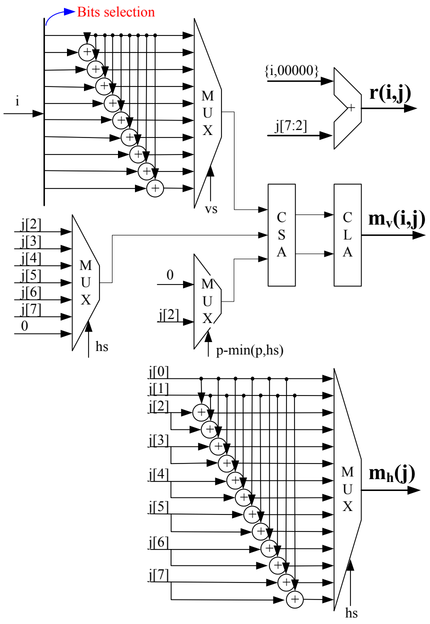
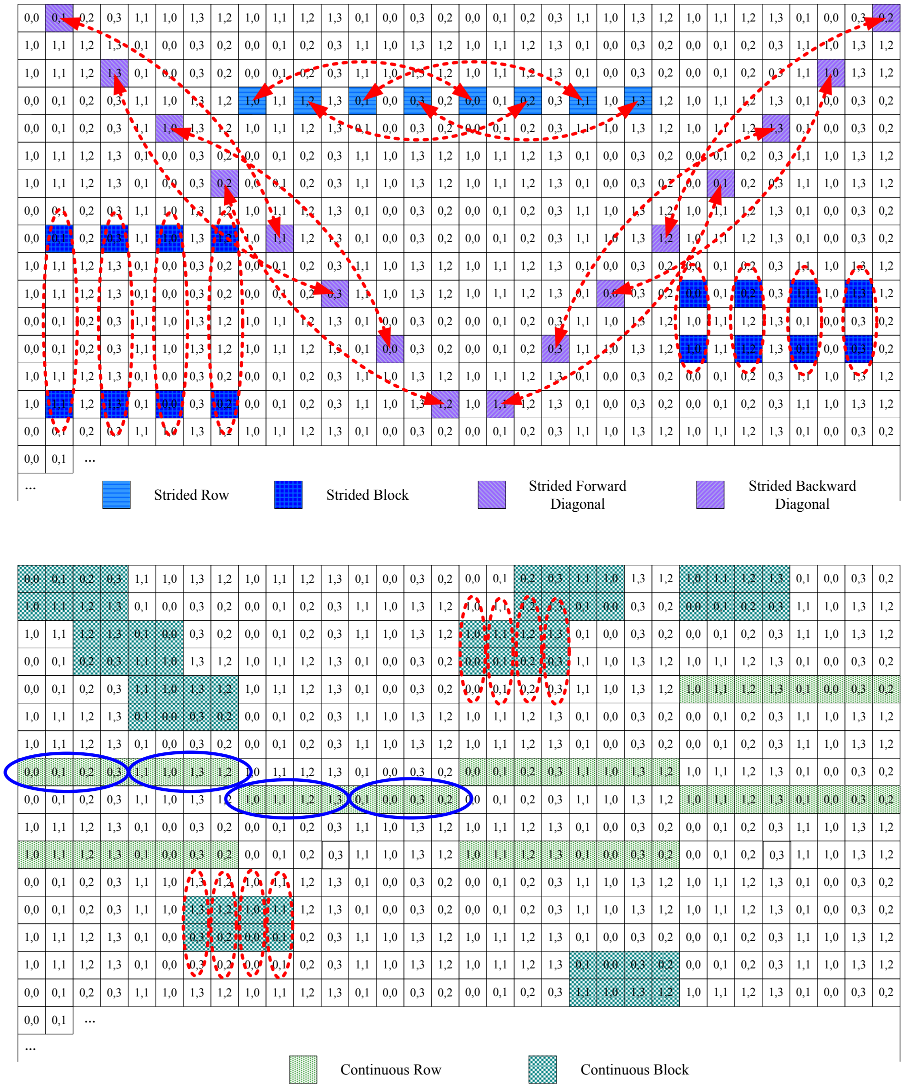


Figure 3.12: 2DSMM address translation circuit with $M = 1024$, $N = 256$, $2^p = 2$, $2^q = 4$

by calculating this in advance because p is constant and hs should be configured at least one clock cycle before the actual memory accesses of the 2DSMM memory would happen. One complete example of the address translation logic is shown in Figure 3.12 ($\oplus$ denotes XOR logic). In the example, the entire capacity of the 2DSMM memory is 1MB, which is consisted of $8(2 \times 4)$ memory modules with data width 32-bit. The entire 2D address space is 1024 rows × 256 columns.

In summary, the critical path of the module assignment function is comprised of one OR gate(2-input), one multiplexer($\log_2 M - p + 1$ -input), one CSA and one CLA (both are p-bit; note the gate delay of CSA is $O(1)$); while the critical path of the row address assignment function is one adder ($\log_2(M \cdot N) - p - q$ -bit). Both paths are independent from each other, therefore the critical path of the address translation logic is the longer one. For practical cases such as the example in Figure 3.12, the critical path is the $\log_2(M \cdot N) - p - q$ -bit adder.



0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	...																													
...																															

Strided Row

Strided Block

Strided Forward Diagonal

Strided Backward Diagonal

0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2
0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2	0,0	0,1	0,2	0,3	1,1	1,0	1,3	1,2	1,0	1,1	1,2	1,3	0,1	0,0	0,3	0,2
0,0	0,1	...																													
...																															

Continuous Row

Continuous Block

Figure 3.13: 2DSMM properties for address & data routing simplification

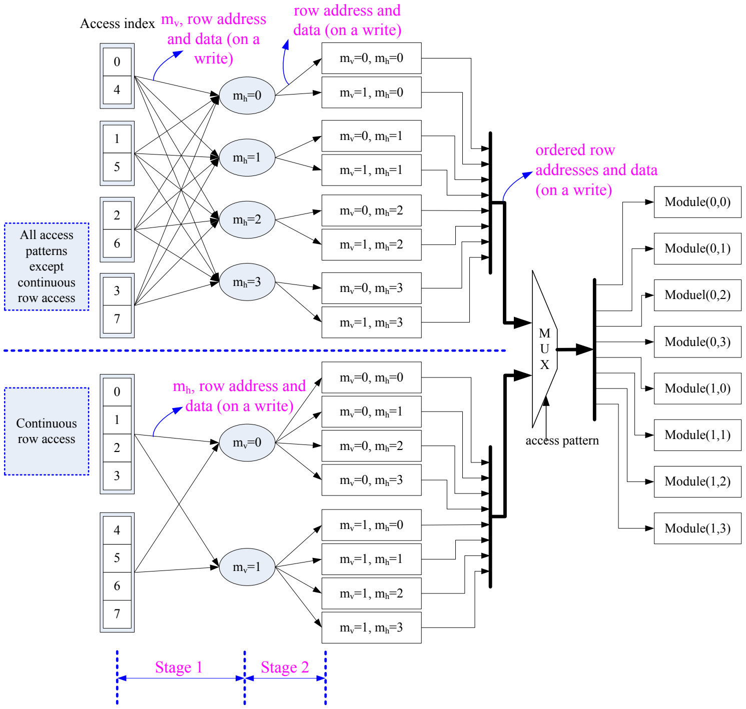


Figure 3.14: 2DSMM address & data routing unit

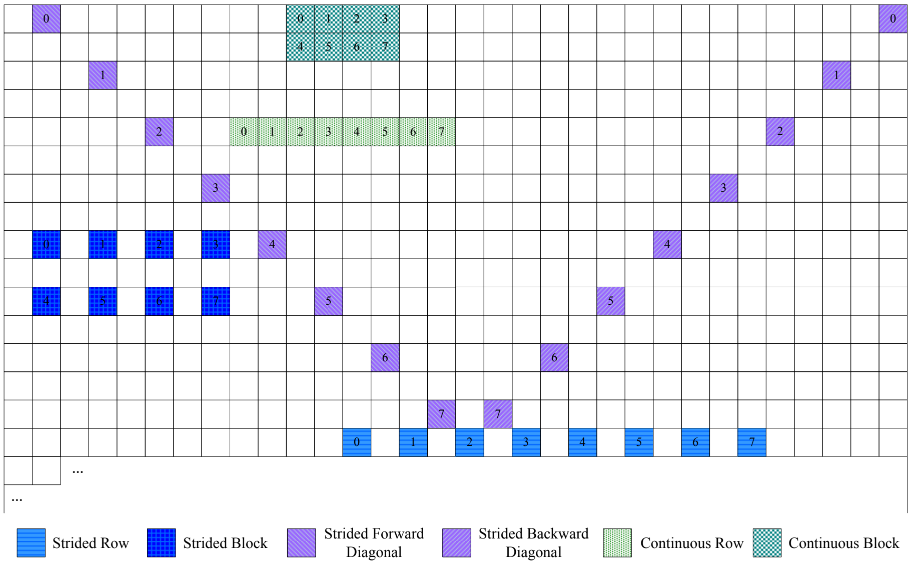


Figure 3.15: Definition of access index for six 2DSMM access patterns

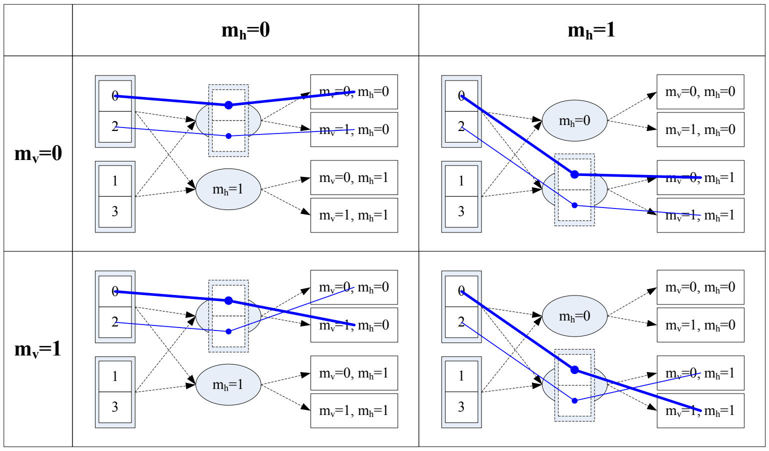


Figure 3.16: 2DSMM address & data routing example ($2^p = 2,\ 2^q = 2$)

Address & Data Routing Unit

The resolved addresses from the ATU have to be proper routed before they reach their destination memory modules. The function of *Address & Data Routing Unit* (Switch in Figure 3.8) is to route the local addresses and data items (on a memory write) to the module numbers and route data to output port on a memory read. It should be noted that, basically a full crossbar switch is indispensable if the 2^{p+q} addresses/data coming from the ATU go to the 2^{p+q} memory modules in an *ad hoc* manner, and the requirement of address/data switch exists for any multiple module memory systems regardless of the presence of the memory scheme. Fortunately, Theorem 7 guarantees that for all 2DSMM access patterns except continuous row, when the access sequence in the pattern are arranged in $2^p \times 2^q$ array, all accesses in the same column are assigned to the same m_h; for continuous row access, all accesses in the same row are assigned to the same m_v. Figure 3.13 is an example. In the figure, the data items connected by dotted ellipse or arrows are assigned to the same m_h, and data connected by continuous ellipse are assigned to the same m_v. The parameters in the example are $N = 32,\ 2^p = 2,\ 2^q = 4,\ 2^{vs} = 2,\ 2^{hs} = 2$.

With the help of the inherent properties of 2DSMM scheme, the address/data routing circuitry could be largely simplified, as Figure 3.14 illustrates. The "access index" at the left side of the figure denotes the predefined index in the six access patterns, counting from 0 to $2^{p+q} - 1$, as shown in Figure 3.15.

In Figure 3.14 there are 2^{p+q} incoming packages and each of them is comprised of a module index pair(m_v and m_h, from ATU), a module local address(from ATU) and a data item from input data port(on a memory write). In the simplified routing circuitry, all incoming packages are grouped in two different ways according to the access patterns. If the access is not the continuous row access, then the incoming data are arranged into 2^q groups according to their m_h(this is stage one). Now there are 2^p packages in each group and each package is consisted of a m_v, a module local address, and data(on a write). Then each group is *aligned* according to m_v of each package(this is stage two). After the alignment, only module local address and data(on a write) are kept and they are already ordered in line with the module index, therefore, the local addresses and data could go to the proper memory modules. This procedure corresponds to the upper data flow in Figure 3.14. For the continuous row access, the procedure corresponds to the lower data flow in the figure. We could notice that it is also divided into two stages, however, the routing information shifts for both stages, compared with the upper case. The two data flow merges and the proper data set is selected according to the access pattern and connected to the 2^{p+q} memory modules at the right side of the figure. Figure 3.16 shows an example for the routing process. For the sake of simplicity, the number of memory modules in the example is 2×2, and the routing shown is for all the access patterns except continuous row access(the upper path of Figure 3.14). m_v and m_h in Figure 3.16 are the resolved addresses of access index 0, which guide the two stage routing for it as illustrated in bold line. Note for the first stage routing, access index 2 are treated in the same way with access index 0 because they have the same m_h, which is shown in the figure by a thin line following exactly the same way of the bold line in stage one.

Above is the description of inward address/data routing mechanism. The philosophy of output data routing is simple: read data should go to where the address is from. In other words, the outward data routing is the inverse procedure of the inward case. Therefore for the outward data routing, the data flow goes from the right side of Figure 3.14 to the left. The routing is also organized into two stages just as the inward routing does, moreover it takes advantage of the same routing information(i.e. m_v and m_h) used in inward routing during the two stages. The only difference is that the stage 1 of inward routing becomes stage 2 of outward routing, and vice versa. The routing information is stored in some registers at the beginning of the next clock cycle after inward routing and it could therefore be used in the outward routing cycle, which is exactly the next clock cycle after the inward routing(note the synchronous SRAM module takes one cycle to read the data out).

The critical path of Address & Data Routing Unit is comprised of three 2-

Table 3.2: 2DSMM scheme critical path and logic consumption

Name	**Critical Path**	**Logic Consumption**
Address Generation Unit	one 5-input multiplexer and one $\max(\log_2 M, \log_2 N)$ -bit adder	$O(2^{p+q})$
Address Translation Unit	one $\log_2(M \cdot N) - p - q$ -bit adder	$O(2^{p+q})$
Address & Data Routing Unit	three 2-input multiplexers and two OR gates (2^p and 2^q -input each)	$O(2^{p+q} \cdot (data\_width + \log_2(M \cdot N) - p - q))$

Table 3.3: 2DSMM scheme storage consumption

Name	**SRAM Bits**
Address Generation Unit	$2^{p+q} \cdot \log_2(M \cdot N)$
Address & Data Routing Unit	$2^{p+q} \cdot (p + q)$

input multiplexers and two OR gates(2^p and 2^q -input each), among which each stage contributes one multiplexer and one OR gate, and the access pattern selection logic contributes another multiplexer. The logic consumption is in the order of $2^{p+q} \cdot (data\_width + local\_address\_width)$, where $data\_width$ is the data width of the 2DSMM memory, and $local\_address\_width$ is the number of address bits of each memory module, i.e. $\log_2(M \cdot N) - p - q$.

Critical Path and Logic Consumption of 2DSMM Scheme

The critical path and logic consumption of 2DSMM scheme are summarized in Table 3.2. Note, the logic consumption in the table is based on the count of the parallel critical path circuitry. Table 3.2 only gives the combinational logic consumption. For the 2DSMM scheme, the dominant SRAM consumption comes from the $VS \cdot i$ and $HS \cdot i(i = 0, 1, 2 \ldots, 2^{p+q} - 1)$ lookup table used in ATU and the storage for routing information in Address & Data Routing Unit, as shown in Table 3.3.

In summary, we could see that the 2DSMM scheme has relatively short critical path which basically increases in proportion to the logarithm of memory size, and logic consumption which grows roughly in proportion to the number of memory modules. Therefore, it could potentially have good scalability with system parameters of memory size and memory module number. We will examine this in more detail in the following section.

Implementation in FPGAs

In the following we will see how the critical path delay, LUTs (lookup table) consumption, SRAM bits consumption scale with the 2DSMM memory size. The total memory size is the multiplication of memory module number (i.e. 2^{p+q}), memory module depth (number of data items per memory module, i.e., $\log(M \cdot N) - p - q$), and memory module data width. We will investigate how the three factors independently influence the hardware implementation of 2DSMM memory.

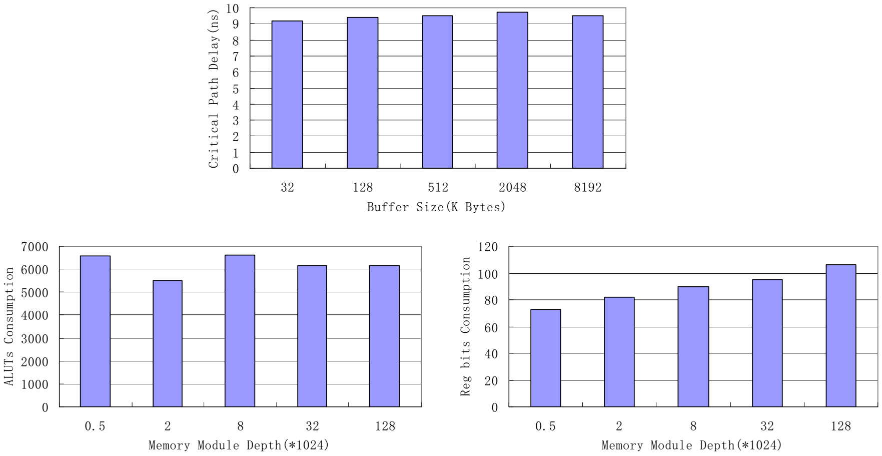


Figure 3.17: 2DSMM memory module depth scaling in Stratix-II with $2^p = 2$, $2^q = 4$, $data\_width = 64$

Figure 3.17 shows the scalability of critical path delay and logic consumption of the 2DSMM memory with memory module depth in Stratix-II FPGA. From the first plot in the figure we could see that there is a *very slight* increase in the critical path delay. This could be explained by the analysis in Table 3.2, which indicates that the critical path includes two adders of which the input bits increase linearly with respect to the increase of $\log_2 M$ and/or $\log_2 N$. Consider the fact that the delay of a CLA is in proportion to the logarithm of number of its input data bits, it is clear that the critical path delay grows in a logarithm(logarithm) manner with the increase of memory module depth. This results the very slight increase of the critical path delay in the first plot of Figure 3.17. As for the logic consumption, it is less than 8%

ALUTs[9] consumption and less than 2% SRAM bits of the FPGA resources even for the largest memory size configuration of the experiment(8MB). For the ALUTs consumption, we could see from the second plot of Figure 3.17 that there is no remarkable change during the scaling. Note the overwhelming majority of the ALUTs consumption comes from the Address & Data Routing Unit in the experiment configuration, as show in Figure 3.18. Therefore,

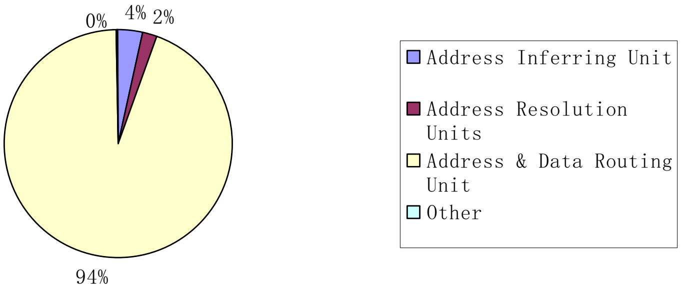


Figure 3.18: 2DSMM ALUTs consumption breakdown in Stratix-II with $2^p = 2, 2^q = 4$, $data\_width = 64$, memory module depth=8192

the scalability of 2DSMM memory combinational logic consumption is determined by that of the Address & Data Routing Unit. From Table 3.2 we know that the impact of memory module depth on the logic consumption is trivial, as long as the change of logarithm of memory module depth is small compared to data_width(i.e. $\triangle(\log_2(M \cdot N) - p - q) \ll data\_width$), which is the case for the experiment configurations and most of the real applications. For the SRAM bits consumption, from Table 3.3 we know that the SRAM bits consumption increases in a logarithm manner when scaling the memory module depth. This is shown in the third plot of Figure 3.17.

Now we will have a look at 2DSMM memory scalability regarding data width of memory modules, as shown in Figure 3.19. Theoretically the data width would not influence the critical path delay, however, we could see from the first plot that there is some increase in critical path delay during the scaling of data width. By further investigating the breakdown of critical path delay we found that the increased part comes from the FPGA interconnection delay. This could be further explained by the second plot of Figure 3.19. In the plot of ALUTs consumption, we could see that the combinational logic consumption grows linearly with the increase of data width (again, the X-axis is plotted

---

[9] ALUT: adaptive lookup table, a terminology from Altera for its LUTs in Stratix-II FPGAs. See http://www.altera.com/literature/hb/stx2/stx2_sii51002.pdf.

in logarithm manner), which is exactly in accordance with the analysis in Table 3.2 (Remember the majority logic consumption comes from the Address & Data Routing Unit). With the huge increase in the logic consumption, the 2DSMM logic becomes more and more interconnection bounded(particularly in the FPGA fabric), therefore the critical path grows as a consequence of increased interconnection delay. From the third plot of Figure 3.19 we could see that the data width does not influence the SRAM bits consumption, which is also indicated in Table 3.3.

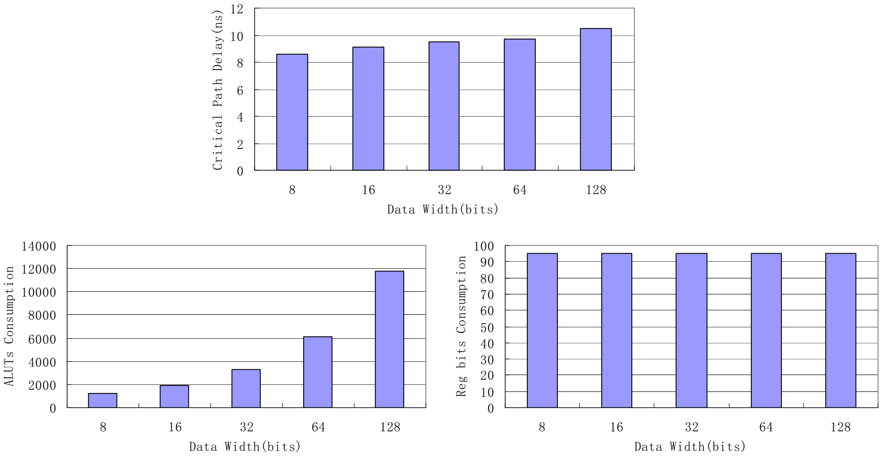


Figure 3.19: 2DSMM data width scaling in Stratix-II with $2^p = 2, 2^q = 4, memory\_module\_depth = 32K$

Finally we will check how 2DSMM scales with the number of memory modules, as shown in Figure 3.20. In this phase, the critical path delay grows roughly in logarithm manner and a major part of the growth is caused by the increase of FPGA interconnection delay due to the enormous logic expansion in the Address & Data Routing Unit. The resource consumption of both combinational logic and SRAM bits increases linearly during the scaling, which is indicated precisely in Table 3.2 and Table 3.3.

We have also implemented the 2DSMM memory in Xilinx Virtex4 XC4VLX200 FPGA. We found the properties of the hardware implementation results remain the same in Virtex-4, which further confirm the good scalability of the hardware implementation of 2DSMM memory.

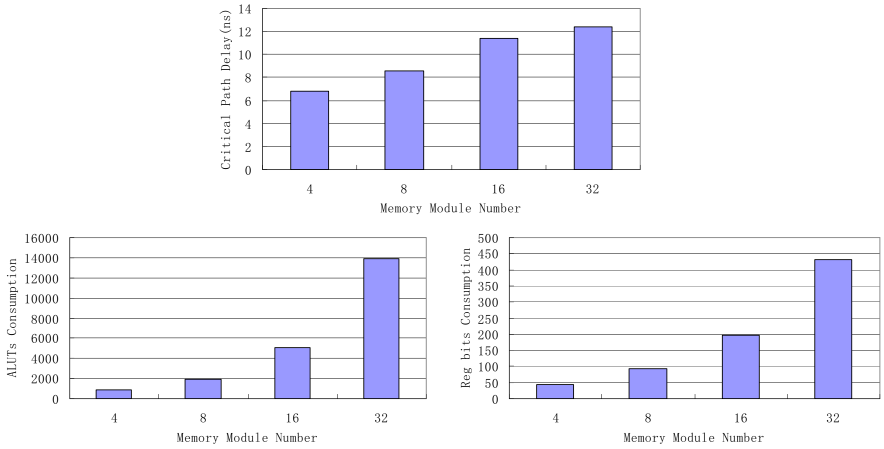


Figure 3.20: 2DSMM memory module number scaling in Stratix-II with $data\_width = 16$, $memory\_module\_depth = 16K$

3.5 Discussion

The most distinctive aspect of our scheme compared to the previous solutions is that it avoids the module conflicts when the memory reference patterns go back and forth between unit-stride and strided accesses, and thus truly-parallel data access is supported. Unlike the out-of-order vector access scheme, our proposal preserves the data sequence required by the vector load/store units, thus atomic parallel access is achieved for short vectors and peak performance could be sustained for vectors as short as 2^q elements. The SAMS is a memory scheme with no redundancy and high utilization of module resources. On the other hand, the SAMS scheme is complementary to the existing techniques, which means that it could also take their advantages to improve system performance. For instance, it adopts the idea of the dynamic scheme where s can be configured by the software at run time for different stride family access in different execution phases. For long vectors, it could also be augmented with out-of-order intra-vector access scheme to support conflict-free access for a wider spectrum of stride families.

It should be noted that SAMS is just one of the set of parallel memory schemes, which provide conflict-free access support for cross stride family vector accesses, under the configuration of 2D memory modules with wide data lines. It has not yet been proved that SAMS scheme is optimal in terms of the number of strides supported, and the complexity of hardware implementation. Therefore, it could be worthwhile to explore the design space of memory schemes under the configuration of wide memory modules, to hopefully find a scheme with better performance compared to SAMS.

Even inside the SAMS scheme itself, there is still space for performance improvement. As presented in the chapter, we have elaborated on the module assignment function; however we chose the row assignment and offset assignment functions straightforwardly. With the module assignment function fixed, there are no means to enlarge the number of supported conflict-free strides as the module conflict patterns are fixed with a fixed module assignment function. We also found that the row assignment function does not affect neither the module conflicts nor the hardware implementation of the SAMS scheme. However, the offset assignment function, which determines the relative positions of the data items in the same row, do have some impact on the hardware implementation. Namely, the offset assignment function determines the permutation patterns of accessed data which should be supported by the data routing circuitry of the SAMS scheme. Therefore, investigation for other offset assignment functions could also be helpful for better hardware implementation.

The simplicity of 2DSMM memory comes in the following way. First, it is the simplicity of address translation unit which makes parallel address translation possible. The other way of address translation in the context of parallel multi-module memory for SIMD access is to use lookup table, as adopted in [118]. In that work, as the prime memory scheme is used, the modulo operations on a prime number make the lookup table solution the only choice for module assignment function. However, the existence of lookup table not only results in longer critical path and larger die area consumption, but also wastes the opportunity of parallel address translation because it is impractical to provide each memory module with an independent lookup table. As a consequence another stage of lookup table is utilized to get the other module assignments from the one got from module assignment lookup table in [118]. Second, the inherent characteristics of the six 2DSMM access patterns provide with the a priori “access group” information which is very useful for the routing, therefore allows replacing the full crossbar switch with a much simpler two-stage switching circuitry.

Regarding the applications, both the SAMS and 2DSMM schemes are applicable wherever the data level parallelism is exploited, to boost the performance of data intensive applications with both unit-stride and strided memory accesses. For instance, the SAMS scheme is adopted as on-chip local store for the Cell

SPE, a typical SIMD processor [68], in Chapter 4 to improve the memory access flexibility. Similarly, they can also be considered for integration as on-chip buffer for GPP SIMD extensions, where the data alignment and permutation problem, which results from the lack of flexible memory access support, remains a bottleneck for many applications [106, 124]. It should be noted that the integration of a SIMD buffer into a GPP introduces coherence problem, as it will be deployed at the same level as the data caches in memory hierarchy. However, this problem could be solved by either snooping mechanisms or by the use of non-cacheable regions in the address space.

3.6 Summary

In this chapter, we proposed the SAMS scheme, which improves the previous non-redundant interleaving schemes (with power-of-two memory modules) by supporting both conflict-free access for both unit-stride and strided patterns. In the SAMS approach, we created a hierarchical way of composing single-affiliation memory scheme from a given multi-affiliation scheme. Furthermore, we added a new dimension of data access parallelism by representing each memory module as a 2D storage to resolve the unit-stride access conflicts in the single-affiliation scheme. In this way, SAMS provides atomic access time for unit stride access, while it could preserve all benefits of the existing cross-stride-family parallel memory schemes. To our best knowledge, SAMS is the first non-redundant, power-of-two parallel memory scheme supporting both strided and unit-stride conflict-free vector access.

In addition, we proposed the 2DSMM scheme based on the basic XOR scheme, which supports fully concurrent strided access patterns of horizontal, block, and diagonal access in 2D address space. Besides strided access, it also supports parallel continuous access patterns of horizontal and block access (on predefined boundaries).

Hardware implementation results in FPGA technology suggest short critical path of the SAMS address translation logic, which is a strong indication for the feasibility of the proposed SAMS scheme in practical parallel memory systems. Results also show that vector memory systems based on the 2DSMM scheme have the advantage of efficient hardware implementation as compared to related work. In particular, we have demonstrated the good scalability of 2DSMM hardware implementation in terms of both critical path delay and area consumption, with both theoretical analysis and implementation results. Furthermore, the proposed schemes achieve high memory module utilization, thanks to their non-redundancy nature.

Note. The contents of this chapter is based on the the following papers:

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **Matched SAMS Scheme: Supporting Multiple Stride Unaligned Vector Accesses with Multiple Memory Modules**, CE Technical Report, CE-TR-2008-06, pp. 1–27, Computer Engineering Lab, TU Delft, October 2008

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **SAMS: Single-Affiliation Multiple-Stride Parallel Memory Scheme**, Proceedings of the 2008 workshop on Memory access on future processors: a solved problem? (MAW'08), pp. 359–367, Ischia, Italy, May 2008

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **2DSMM: 2D Strided Multi-access Memory**, CE Technical Report, pp. 1–38, Computer Engineering Lab, TU Delft, July 2007

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **2D Stride Multiaccess Memory Scheme**, HiPEAC ACACES 2007, pp. 193–197, L'Aquila, Italy, July 2007

4

Providing Multiple Views to Data

Efficient SIMDization of applications is often hampered by the mismatch between data representations in memory and those favored by the SIMD processor. In this chapter, we propose to bridge the discrepancy between data representations in memory and those favored by the SIMD processor by *customizing the low-level address mapping*. To achieve this, we employ the extended Single-Affiliation Multiple-Stride (SAMS) parallel memory scheme at an appropriate level in the memory hierarchy. This level of memory provides both Array of Structures (AoS) and Structure of Arrays (SoA) views for the structured data to the processor, appearing to have maintained *multiple layouts* for the *same data*. With such multi-layout memory, optimal SIMDization can be achieved. Our synthesis results using TSMC 90nm CMOS technology indicate that the SAMS Multi-Layout Memory system has efficient hardware implementation, with a critical path delay of less than 1ns and moderate hardware overhead. Experimental evaluation based on a modified IBM Cell processor model suggests that our approach is able to decrease the dynamic instruction count by up to 49% for a selection of real applications and kernels. Under the same conditions, the total execution time can be reduced by up to 37%.

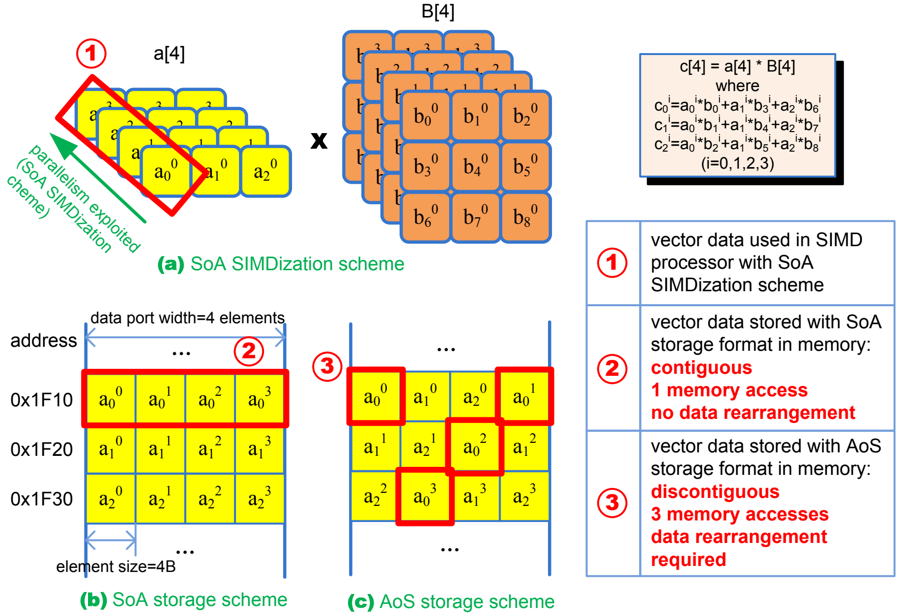
4.1 Introduction

One of the most critical challenges in SIMD processing is imposed by the data representation. By exploiting explicitly expressed data parallelism, SIMD processors tend to provide higher performance for computationally intensive applications with lower control overhead compared to superscalar microprocessors. However, SIMDization suffers from the notorious problems of difficult data alignment and arrangement, which greatly undermine its potential performance [11, 106, 124].

In both scientific and commercial applications, data is usually organized in a structured way. A sequence of structured data units could be represented either in AoS (Array of Structures) or in SoA (Structure of Arrays) format. Such a data representation predetermines, at the application level, the data layout and its continuity in the *linear memory address space*. It has been found that most SIMDized applications are in favor of operating on SoA format for better performance [46, 61]. However, data representation in the system memory is mostly in the form of AoS because of two reasons. First, AoS is the natural data representation in many scientific and engineering applications. Secondly, indirections to structured data, such as pointer or indexed array accesses, are in favor of the AoS format. Therefore, a pragmatic problem in the SIMDization arises: the need for dynamic data format transform between AoS and SoA, which results in significant performance degradation. To our best knowledge, no trivial solution for this problem has been previously proposed. Our SAMS Multi-Layout Memory system, presented in this chapter, supports contiguous data access for both AoS and SoA formats. The specific contributions of our proposal are:

- custom, low-level address mapping logic to manage individual internal data layout and provide efficient memory accesses for both AoS and SoA views;
- novel hardware/software interface for improved programmer productivity and additional performance gains;
- the SAMS scheme implementation in TSMC 90nm CMOS technology with affordable critical path (< 1 *ns*) and its integration into the IBM Cell SPE model;
- up to 49% improvement in dynamic instruction counts for real applications and kernels, which is translated into a 37% reduction of the overall execution time.

The remainder of the chapter is organized as follows. In Section 4.2, we provide the motivation for this work. In Section 4.3, the original SAMS scheme and the proposed extensions are briefly described. The hardware implementation and synthesis results of the SAMS Multi-Layout Memory system and its integration to the IBM Cell SPE are presented in Section 4.4. Simulated performance of the SAMS memory in applications and kernels is evaluated in Section 4.5. Finally, Section 4.6 summarizes the chapter.


B[4]
①
a[4]
x
Parallelism exploited (SoA SIMDization scheme)
c[4] = a[4] * B[4]
where
$c_0^i=a_0^i*b_0^i+a_1^i*b_3^i+a_2^i*b_6^i$
$c_1^i=a_0^i*b_1^i+a_1^i*b_4^i+a_2^i*b_7^i$
$c_2^i=a_0^i*b_2^i+a_1^i*b_5^i+a_2^i*b_8^i$
(i=0,1,2,3)
① vector data used in SIMD processor with SoA SIMDization scheme
② vector data stored with SoA storage format in memory: contiguous 1 memory access no data rearrangement
③ vector data stored with AoS storage format in memory: discontiguous 3 memory accesses data rearrangement required
address
data port width=4 elements
0x1F10
0x1F20
0x1F30
element size=4B
②
③

(a) SoA SIMDization scheme

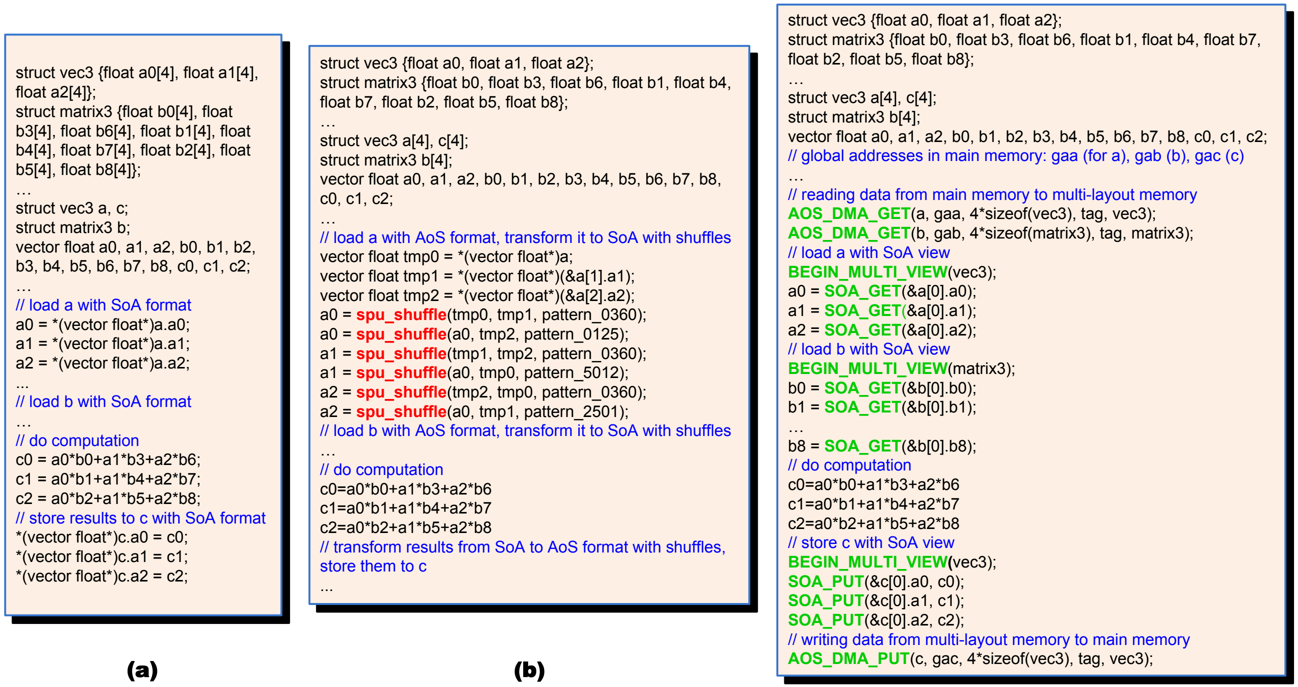
(b) SoA storage scheme

(c) AoS storage scheme

Figure 4.1: Vector-matrix multiplication: multiple working data sets

4.2 Motivation

Motivation Example: We shall examine the SIMDization of vector-matrix multiplication, $c = a \times B$, where a and c are 1x3 vectors and B is a 3x3 matrix with column-major storage. Although the involved computations are quite simple, SIMDizing them to achieve optimal speedup is very difficult. Assuming a 4-way SIMD processor, the first apparent drawback is that only 75% of the available bandwidth could be utilized during vector multiplications for the inner products. Afterwards, all the three elements of the vector multiplication result have to be accumulated. However, this is not straightforward because the three elements are located in different vector lanes while a vector operation could be done in SIMD processors only when its operands are distributed in the same vector lane. Therefore, a sequence of data shuffle operations is necessary to rearrange the elements to be accumulated in the same lanes. Moreover, due to memory alignment restrictions in many practical SIMD systems, neither the second, nor the third column of B can be accessed within a single vector load; instead, they require additional load and shuffle instructions to fetch and rearrange the data into the right format. As a consequence of this rearrangement, performance is penalized. Zero-padding can be used in some applications to



```
struct vec3 {float a0[4], float a1[4],
float a2[4]};
struct matrix3 {float b0[4], float
b3[4], float b6[4], float b1[4], float
b4[4], float b7[4], float b2[4], float
b5[4], float b8[4]};
…
struct vec3 a, c;
struct matrix3 b;
vector float a0, a1, a2, b0, b1, b2,
b3, b4, b5, b6, b7, b8, c0, c1, c2;
…
// load a with SoA format
a0 = *(vector float*)a.a0;
a1 = *(vector float*)a.a1;
a2 = *(vector float*)a.a2;
...
// load b with SoA format
…
// do computation
c0 = a0*b0+a1*b3+a2*b6;
c1 = a0*b1+a1*b4+a2*b7;
c2 = a0*b2+a1*b5+a2*b8;
// store results to c with SoA format
*(vector float*)c.a0 = c0;
*(vector float*)c.a1 = c1;
*(vector float*)c.a2 = c2;
```

(a)
SoA storage +
SoA SIMDization

```
struct vec3 {float a0, float a1, float a2};
struct matrix3 {float b0, float b3, float b6, float b1, float b4,
float b7, float b2, float b5, float b8};
…
struct vec3 a[4], c[4];
struct matrix3 b[4];
vector float a0, a1, a2, b0, b1, b2, b3, b4, b5, b6, b7, b8,
c0, c1, c2;
…
// load a with AoS format, transform it to SoA with shuffles
vector float tmp0 = *(vector float*)a;
vector float tmp1 = *(vector float*)(&a[1].a1);
vector float tmp2 = *(vector float*)(&a[2].a2);
a0 = spu_shuffle(tmp0, tmp1, pattern_0360);
a0 = spu_shuffle(a0, tmp2, pattern_0125);
a1 = spu_shuffle(tmp1, tmp2, pattern_0360);
a1 = spu_shuffle(a0, tmp0, pattern_5012);
a2 = spu_shuffle(tmp2, tmp0, pattern_0360);
a2 = spu_shuffle(a0, tmp1, pattern_2501);
// load b with AoS format, transform it to SoA with shuffles
…
// do computation
c0=a0*b0+a1*b3+a2*b6
c1=a0*b1+a1*b4+a2*b7
c2=a0*b2+a1*b5+a2*b8
// transform results from SoA to AoS format with shuffles,
store them to c
…
```

(b)
AoS storage +
SoA SIMDization

```
struct vec3 {float a0, float a1, float a2};
struct matrix3 {float b0, float b3, float b6, float b1, float b4, float b7,
float b2, float b5, float b8};
…
struct vec3 a[4], c[4];
struct matrix3 b[4];
vector float a0, a1, a2, b0, b1, b2, b3, b4, b5, b6, b7, b8, c0, c1, c2;
// global addresses in main memory: gaa (for a), gab (b), gac (c)
…
// reading data from main memory to multi-layout memory
AOS_DMA_GET(a, gaa, 4*sizeof(vec3), tag, vec3);
AOS_DMA_GET(b, gab, 4*sizeof(matrix3), tag, matrix3);
// load a with SoA view
BEGIN_MULTI_VIEW(vec3);
a0 = SOA_GET(&a[0].a0);
a1 = SOA_GET(&a[0].a1);
a2 = SOA_GET(&a[0].a2);
// load b with SoA view
BEGIN_MULTI_VIEW(matrix3);
b0 = SOA_GET(&b[0].b0);
b1 = SOA_GET(&b[0].b1);
…
b8 = SOA_GET(&b[0].b8);
// do computation
c0=a0*b0+a1*b3+a2*b6
c1=a0*b1+a1*b4+a2*b7
c2=a0*b2+a1*b5+a2*b8
// store c with SoA view
BEGIN_MULTI_VIEW(vec3);
SOA_PUT(&c[0].a0, c0);
SOA_PUT(&c[0].a1, c1);
SOA_PUT(&c[0].a2, c2);
// writing data from multi-layout memory to main memory
AOS_DMA_PUT(c, gac, 4*sizeof(vec3), tag, vec3);
```

(c)
AoS storage + SoA SIMDization
+ Multi-layout memory

Figure 4.2: Sample vector-matrix multiplication code

alleviate the data alignment problem, but at the cost of wasted memory space and additional memory bandwidth which can become prohibitively expensive for some applications (e.g. the Wilson-Dirac kernel).

Fortunately, it is common in applications with high data parallelism that the processing is to be operated upon multiple independent data sets, just as Figure 4.1a suggests. The SIMDization method which exploits data parallelism in single data set processing is still applicable, which will be referred to as "AoS SIMDization scheme" in the chapter. However, we can also map each SIMD operation to a batch of data sets to exploit inter-dataset parallelism, which is referred to as "SoA SIMDization scheme"[1]. If the data storage scheme is SoA, illustrated in Figure 4.1b, optimal performance gain of four times speed-up could be potentially achieved. The example code for this case is shown in Figure 4.2a. However, if the data format in memory is AoS, as suggested in Figure 4.1c, the data rearrangement is inevitable, resulting in performance degradation. The example code for this case (Figure 4.2b) suggests that, e.g., 6 shuffles are required to rearrange 3 vector elements of **a** loaded from memory, which are apparently non-trivial overhead compared to the actual vector-matrix multiplication.

Problem Statement: From the above example, it can be observed that parallel processing of a batch of N data sets is more favorable for better utilization of SIMD parallel datapath and thus results in higher performance. Therefore, the SoA SIMDization scheme is preferable in most cases. Unfortunately, the most common data layout in the main memory is AoS (as briefly discussed in Section 6.1). This data representation discrepancy poses a significant overhead of dynamic data format conversions between the AoS and SoA.

Proposed Solution: The essential reason for the data format mismatch is that there is *no single optimal data layout* for different data access patterns. For operations based on indirections, the AoS storage scheme is preferable since access to fields inside a data set is contiguous; while for SIMD operations in most cases, the SoA storage scheme is favorable. To bridge the data representation gap, our idea is to design a memory system, which preserves the benefits of both AoS and SoA layouts. We call such a system "multi-layout memory" and its position and main functionalities are shown in Figure 4.3. In such a system, the multi-layout memory is deployed between the main memory and the SIMD processor, working as an intermediate data storage to provide contiguous data access to both data fields within the same data set (like the AoS layout) and the same field across consecutive data sets (like the SoA lay-

---

[1] Also known as "outer-loop vectorization" [107].

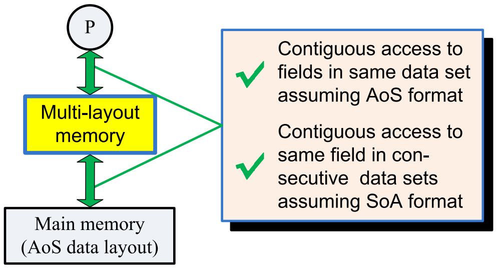

P
Multi-layout memory
Main memory
(AoS data layout)
✓ Contiguous access to fields in same data set assuming AoS format
✓ Contiguous access to same field in consecutive data sets assuming SoA format

Figure 4.3: The proposed multi-layout memory

out). Therefore, the penalty of dynamic conversion between the AoS and SoA data representations is completely avoided with the help of this multi-layout memory. The vector-matrix multiplication code for this case is shown in Figure 4.2c. It can be observed that, the programmer can easily express multiple views of the data arrays and the shuffle overhead is completely avoided.

It is worthy to mention that, assuming AoS layout in the linear address space, the AoS view of data requires unit-stride access, while the SoA view requires strided access, where the stride is determined by the size of the working data structure. We will address this issue in the following section.

4.3 The Extended SAMS Scheme

4.3.1 Original SAMS Scheme

Given a specific physical memory organization and resources, parallel memory schemes determine the mapping from the linear address space to physical locations, such as the module/bank number and row address. Vector access is one of the most important memory reference patterns in SIMDized applications. Traditional parallel memory schemes in vector computers provide conflict-free access for a *single* stride family. To solve the module conflicts encountered with the cross stride family accesses, several enhancements have been previously proposed in literature, such as the use of dynamic memory schemes [35, 36], use of buffers [34], use of more memory modules [34], and out-of-order vector access [144].

In Chapter 3, a parallel memory scheme, SAMS, was proposed to simultaneously support conflict-free unit-stride and strided memory accesses. The SAMS scheme is mathematically described by three functions: the *module as-*

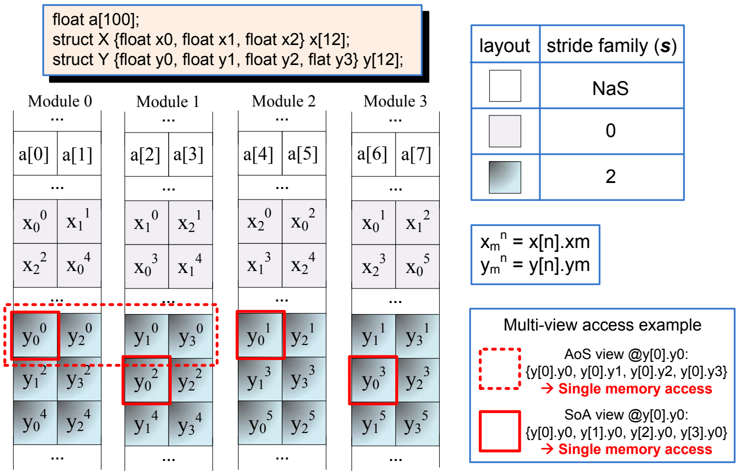

float a[100];
struct X {float x0, float x1, float x2} x[12];
struct Y {float y0, float y1, float y2, flat y3} y[12];
Module 0
Module 1
Module 2
Module 3
a[0] a[1]
a[2] a[3]
a[4] a[5]
a[6] a[7]
x_0^0 x_1^1 x_2^2 x_0^4
x_1^0 x_2^1 x_0^3 x_1^4
x_2^0 x_0^2 x_1^3 x_2^4
x_0^1 x_1^2 x_2^3 x_0^5
y_0^0 y_2^0 y_1^2 y_3^2 y_0^4 y_2^4
y_1^0 y_3^0 y_0^2 y_2^2 y_1^4 y_3^4
y_0^1 y_2^1 y_1^3 y_3^3 y_0^5 y_2^5
y_1^1 y_3^1 y_0^3 y_2^3 y_1^5 y_3^5
layout
stride family (s)
NaS
0
2
x_m^n = x[n].xm
y_m^n = y[n].ym
Multi-view access example
AoS view @y[0].y0:
{y[0].y0, y[0].y1, y[0].y2, y[0].y3}
→ Single memory access
SoA view @y[0].y0:
{y[0].y0, y[1].y0, y[2].y0, y[3].y0}
→ Single memory access

Figure 4.4: Internal data layouts in SAMS Multi-Layout Memory

signment function (Equation 3.1); the *row assignment function* (Equation 3.2); and the *offset assignment function* (Equation 3.3). Detailed treatment of the scheme and the functions was discussed in Section 3.2.

4.3.2 Proposed Extensions

We have made two important extensions to the original SAMS scheme (Chapter 3), in order to better meet the requirements and constraints of practical SIMD systems, related to (1)multi-layout support and (2)non-strided access.

(1)Multiple Data Layouts Support: In Chapter 3, it was assumed that the entire SAMS memory system adopts a single low-level address mapping (linear address↔module/row/offset) scheme and therefore manages unified internal data layout pattern. Although this simplifies the memory access since it does not need to indicate the stride family for which the accessed data is optimized (such information is maintained at the global scope), it significantly limits the SAMS applications, since there are many applications with multiple structured data, which require different internal data layouts for optimal access efficiency. The *Point* and *PointData* in streamcluster [20], and the *spinor* and *gauge_link* in Wilson-Dirac kernel [63] are examples[2] for such a requirement. Therefore, instead of maintaining a single low-level address mapping at the global scope for all data, our approach *customizes the address mapping logic* and manages

---

[2] See Section 4.5 for details.

an *individual internal data layout* for each application data, as illustrated in Figure 4.4. Figure 4.4 suggests that the *stride family* is an essential parameter in the SAMS scheme. Strided accesses with strides belonging to the stride family supported by the internal data layout could be accomplished in a single access; while accesses with strides from other stride families may cause module conflicts. Furthermore, it is the stride family that *configures* the low-level address mapping and the resulting internal data layout in the memory. On the other hand, the internal data layout/address mapping *determines* what stride family it supports, as illustrated in Figure 4.4.

The configuration granularity of the internal data layout/ address mapping is a complete 32 bytes data line. This equals 2×*sizeof*(vector register), which is determined by the SAMS hardware. Since most relevant applications tend to use large arrays, such granularity is well suited.

Obviously, with the extension to multiple data layouts, we have to keep track of the appropriate access strides and stride families for different data. Fortunately, this is not difficult as the information of the data structure and organization is static in most cases. Therefore, it is quite feasible to provide the programmer with some abstractions, e.g., C macros or library functions, to facilitate capturing such structural information, as Figure 4.2c illustrates. Furthermore, it is also possible for the compiler to automate the multi-layout memory usage with proper compile-time analysis and optimizations.

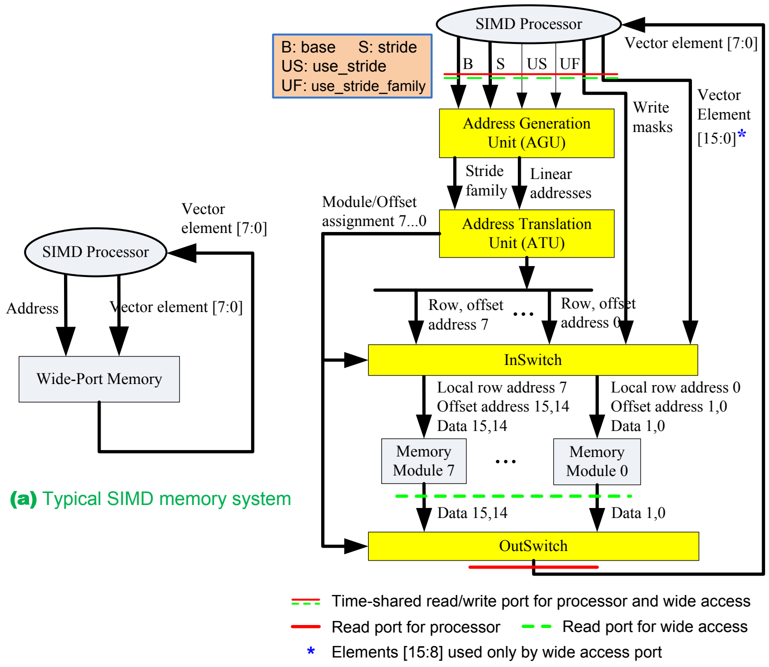
(2)Definition for *NaS*: By convention, the stride family $s \geq 0$. We extend this definition by introducing a special symbol NaS(not a stride), which indicates a special non-strided data storage pattern:

$$\begin{cases} m(a) & = & \frac{a}{2}\%2^q \\ r(a) & = & \frac{a}{2^{q+1}} \\ o(a) & = & a_0 \end{cases}$$

as demonstrated by the layout of **a** in Figure 4.4. The NaS pattern is a simple yet efficient layout for data not touched by any strided memory access. The intuition for this extension is the concern of power efficiency. For aligned and continuous accesses, it is unnecessary to invoke the majority of the AGU, ATU and In/OutSwitch logic in Figure 4.5b[3]. Therefore, those components may be bypassed or even shutdown to save power, when the program does not need unaligned or strided access. In the particular case of the SAMS integration into the Cell SPE, the system further benefits from the NaS pattern. In the Cell SPE, the local store is responsible for feeding instructions as well as data to

---

[3] See Section 4.4.1 for details.



(a) Typical SIMD memory system

(b) SAMS Multi-Layout Memory system

Figure 4.5: SIMD memory organizations

SPU, where the instruction fetch (IF) is always aligned and continuous - at the granularity of 64 bytes [44]. Therefore, the instruction fetch engine can use the NaS layout and completely remove the OutSwitch in Figure 4.5b from the IF pipeline. The DMA engine can also use this pattern for regular data accesses.

4.4 Implementation and Integration

In this section, we investigate the implementation of the SAMS Multi-Layout Memory system and present its integration into the IBM Cell SPE.

4.4.1 SAMS Organization and Implementation

Figure 4.5a illustrates a typical memory system of a SIMD processor. To reduce hardware complexity, a logically monolithic memory module with wide data port is used to feed vector elements, which are contiguous in memory

Table 4.1: Stride control signals

<use_stride_family, use_stride>	00	01	10	11
semantics	narrow port normal access	wide port access	narrow port unit-stride access (e.g. AoS view)	narrow port strided access (e.g. SoA view)
stride used in AGU	1	AGU bypassed	1	stride from processor
stride family used in ATU	NaS	ATU bypassed	stride_family generated by AGU	stride_family generated by AGU
# of accessed elements	2^q	2^{q+1}	2^q	2^q

space, to the SIMD processor core. Figure 4.5b illustrates the organization of a multi-layout memory system based on the extended SAMS scheme. The vector processor core issues memory access commands, together with the base address and stride (note the vector length VL=8) to the Address Generation Unit(AGU). The eight linear addresses are generated in parallel in AGU and they are then resolved by the Address Translation Unit(ATU) into eight module assignments, eight row addresses and eight row offset addresses. Afterwards, the eight groups of row-offset pair and eight data elements from input data port (on a memory write) go to the InSwitch and get routed to the proper memory modules according to their corresponding module assignments. In case of a memory read access, after the read latency of the memory modules[4], eight read data are fed back to the vector processor through the OutSwitch at the bottom of Figure 4.5b. Two additional latencies are incurred by the integration of extended SAMS scheme to the original SIMD memory system of Figure 4.5a: a)“inbound path”, which includes AGU, ATU and InSwitch; b)“outbound path”, which consists of the OutSwitch only.

Address Generation Unit (AGU): AGU is responsible for parallel generation of the addresses of the 2^q vector elements, namely: $\{base,\ base + stride,\ base + 2 \cdot stride,\ \cdots,\ base + (2^q - 1) \cdot stride\}$. Also, it computes the stride family from the stride designated by the SIMD processor. Note there are two control signals from the processor: *use_stride* and *use_stride_family*. Table 4.1 shows the semantics of these signals.

Address Translation Unit (ATU): determines the internal data layout of the

---

[4] The access latency of a memory module may be more than one clock cycle. In the chapter, we assume the memory modules are fully pipelined.

SAMS Multi-Layout Memory and input/output data permutation patterns used in InSwitch and OutSwitch. ATU consists of three independent components: the module assignment logic, the row assignment logic and the offset assignment logic. Therefore, the critical path of ATU is the longest of the three, which is the $n-q$ bit adder followed by a 2-to-1 multiplexor in the row assignment logic [56].

Table 4.2: Synthesis results of SAMS Multi-Layout Memory system

	Critical Path Delay [ns]			**Logic Complexity [# of gates]**		
	$q=2$	$q=3$	$q=4$	$q=2$	$q=3$	$q=4$
SAMS memory logic	0.76	0.87	1.01	6,906	26,784	82,538
equivalent # of 32-bit adders	2.0	2.3	2.7	1.5	5.9	18.1

In/OutSwitch: In the SAMS Multi-Layout Memory system, the InSwitch is a $2^q \times 2^{q+1}$ crossbar, while the OutSwitch is a $2^{q+1} \times 2^q$ crossbar [56].

Unaligned Vector Access: Unaligned vector memory access is one of the critical problems in SIMD processing systems [106, 124]. The SAMS Multi-Layout Memory system supports unaligned unit-stride and strided vector loads and stores. Details of a similar technique can be found in [11].

Memory Store Granularity: With 2^q memory modules instead of a monolithic memory module, the store granularity of the SAMS Multi-Layout Memory system is reduced from an entire vector of 2^q elements to a single element. For example, the monolithic local store of IBM Cell SPE only supports loads/stores at the granularity of 128 bits; while with the SAMS scheme with four memory modules and element size of 32 bits, stores of 1, 2 or 4 32-bit elements are well supported.

Wide Port Support: The SAMS scheme utilizes wide data lines to tolerate module conflicts [56]. More specifically, each of the eight modules in Figure 4.5b has a data port width of two elements and the eight memory modules are capable of servicing 16 elements per access, under the condition that it is aligned to 16 elements boundary. To avoid additional hardware complexity, the wide access port in Figure 4.5b is not responsible for reordering the 16 data elements during a wide access. Indeed, the wide port behaves the same as an ordinary linear memory interface: it directly reads or writes all the 16 data elements from/to the 8 memory modules with the row address of $\frac{base}{32}$(assuming 4B element size), effectively bypassing all the SAMS logic. Therefore, for a read of a full data line of 16 elements from the SAMS Multi-Layout Memory, the external data consumer has to do a post read shuffle after reading the data. For a write, a pre-write shuffle is also necessary, since the internal data layout of the SAMS Multi-layout Memory has non-linear structure as indicated in Figure 4.4. For the external data provider/consumer of the SAMS memory, there is a trade-off between the bandwidth and hardware complexity. We shall further discuss this in Section 4.4.2.

Implementation and Synthesis Results: We have implemented the SAMS Multi-Layout Memory system using Verilog and synthesized it for TSMC 90nm low-K process technology using Synopsis Design Compiler. Synthesis results are provided in Table 4.2 for SAMS memory systems with 4, 8 and 16 memory modules, i.e., $q = 2$, 3 and 4, which target 4-way, 8-way and 16-way SIMD processing systems respectively. The critical path delays in Table 4.2 actually present the inbound path. We also calculated the relative delay and area consumption of the SAMS system compared to a 32-bit adder synthesized on the same technology node. Further investigation into the synthesis results indicates that the ATU, which is the core of the SAMS scheme, has quite fast and compact hardware implementation: it only contributes to approximately $\frac{1}{5}$ of the entire critical path delay and its area overhead is even smaller.

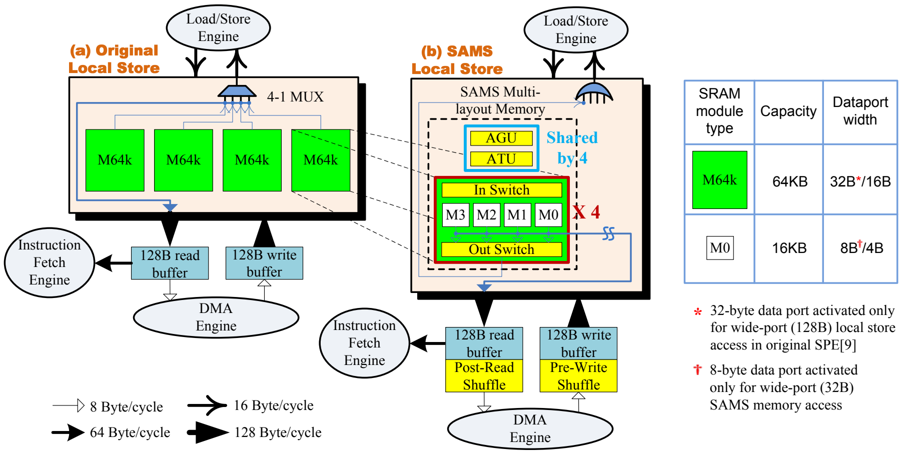
4.4.2 Integration into the Cell SPE

To validate the performance of the proposed SAMS Multi-Layout Memory in real applications, we implemented it in a model of the IBM Cell processor, aiming at computation intensive applications with high data parallelism [68, 120]. The local store of the Cell Synergistic Processing Element (SPE) is chosen for the deployment and implementation of the multi-layout memory system. Figure 4.6a depicts the original local store memory organization in the Cell SPE. The fully pipelined 256KB local store is composed of four *M64k* SRAM modules[5]. Note, the SRAM arrays are themselves single-ported, therefore, the local store is accessed in a time-shared manner, as sketched in Figure 4.6a (only the load path is shown for simplicity).

The integration of SAMS Multi-Layout Memory is illustrated in Figure 4.6b. It is also referred to as “SAMS local store” in our experiments in Section 4.5. Note, although in Figure 4.6b each M64k module is split into four submodules, the total size of the SAMS local store is kept the same as the original one. The

---

[5] *M64k* is the 64KB SRAM module used in SPE local store which runs at the same speed as the SPU core [38].



SRAM module type	Capacity	Dataport width
M64k	64KB	32B*/16B
M0	16KB	8B†/4B

* 32-byte data port activated only for wide-port (128B) local store access in original SPE[9]

† 8-byte data port activated only for wide-port (32B) SAMS memory access

Figure 4.6: SAMS Multi-Layout Memory Integration into the Cell SPE

“splitting” of SRAM arrays may not incur additional engineering effort, since the original M64k is composed of 32 subarrays in the Cell local store physical implementation [38].

An important change in hardware due to the SAMS integration is on the 128B wide port buffers. As high bandwidth of the wide port is extraordinarily desirable for both instruction fetch and DMA in SPE, we choose to provide full bandwidth of the SAMS memory for the wide port. As discussed in Section 4.4.1, in this case the wide port data of each SAMS duplicate needs to be aligned to 32B boundary (this is guaranteed by the 128B access granularity of the original local store wide port), and the data format needs to be adjusted to the internal layout in SAMS memory modules. The latter requires two macros, the Post-Read Shuffle (PRS) and the Pre-Write Shuffle (PWS), to be added to the system, as suggested in Figure 4.6b. The critical path delay and area of PRS and PWS are comparable to those of the SAMS ATU, which has less than one cycle latency and trivial hardware consumption as discussed in 4.4.1.

The major impact on the SPU microarchitecture with the incorporation of the SAMS Multi-Layout Memory in the SPE local store is that the local store pipeline is lengthened, since the SAMS logic introduces additional delay. According to our synthesis results in Table 4.2, the critical path delay of the SAMS inbound path for four memory modules (each with 64-bit port width) is 0.76 ns, which corresponds to two times the latency of a 32-bit adder. In the Cell SPU, 32-bit addition in the Vector Fixed Point Unit (FPU) fixed-point is accomplished in a single cycle [97]. Therefore, we project the deployment of the SAMS memory in SPE's local store will introduce 2(very stringent) or 3(considering pipeline latches and retiming costs) additional pipeline stages for the inbound path. The outbound path takes one additional cycle since it has a critical path delay of 0.35ns which is less than the 0.38ns latency of a 32-bit adder in our study. To summarize, 4 cycles for a load (inbound and outbound paths) and 3 cycles for a store (only inbound path involved) is a realistic estimation for the extra latency incurred by the integration of the SAMS Multi-Layout Memory logic inside the SPU pipeline in our study. As in the original SPE [120] the load and store instructions take 6 and 4 clock cycles respectively, their costs will become 10 and 7 cycles in the modified SPE pipeline.

SAMS Memory Pipeline Optimization: Certain optimizations can be considered to *hide* the SAMS pipeline latency when taking into account target microarchitecture pipeline specifics. For example, in our experiments with the Cell SPE, there are two pipeline stages dedicated for register read, as shown in Figure 4.7a. Note, for a narrow port local store access (Section 4.4.1), only the base address (B) and data ($\mathit{Vector\ element}$[7:0], in case of a local store write) need to be read from SPU *general-purpose vector register file* (GPVRF), as Figure 4.5b indicates. As a result, operations not dependent on B and $\mathit{Vector\ element}$ (e.g. calculations $\{2\cdot \mathit{stride},\ 3\cdot \mathit{stride},\ \cdots,\ (2^q-1)\cdot \mathit{stride}\}$, and the computation of $\mathit{stride\_family}$ from stride) can be performed as soon as the RF read starts. Based on this observation, SAMS memory system can start working after memory access instructions are issued (IS stages), resulting portion of the SAMS delay effectively hidden by the RF stages, as illustrated in Figure 4.7b. Note also the stride register is a dedicated register, hence writing it takes one cycle (done at the beginning of RF2 for *setstride* instruction) and reading it incurs no delay at the beginning of RF1 for SAMS memory access instructions, which allows for back-to-back issue of *setstride* and SAMS memory access instructions. More aggressive optimizations may even hide the entire AGU and ATU logic in vector RF read stages (as long as their latency fits), by using a dedicated base register instead of GPVRF to expose only InSwitch and OutSwitch (see Figure 4.5b) to the load/store pipeline. Furthermore, part of the original 6 cycles load/store pipeline may be overlapped with the SAMS logic (e.g. there is an adder macro in the original SPE load/store pipeline to add the base and offset addresses for loads/stores with *-d/x/l* forms [38]), as Figure 4.7b shows. Note, although not shown in Figure 4.7b, IF and all backend pipelines are also increased by 2 and 3 stages, respectively. The former is because instruction engine fetches instructions from local store - thus it also suffers the exposed SAMS latency (2 stages in the example as the last OutSwitch stage is bypassed in a wide port read). The latter is necessary since

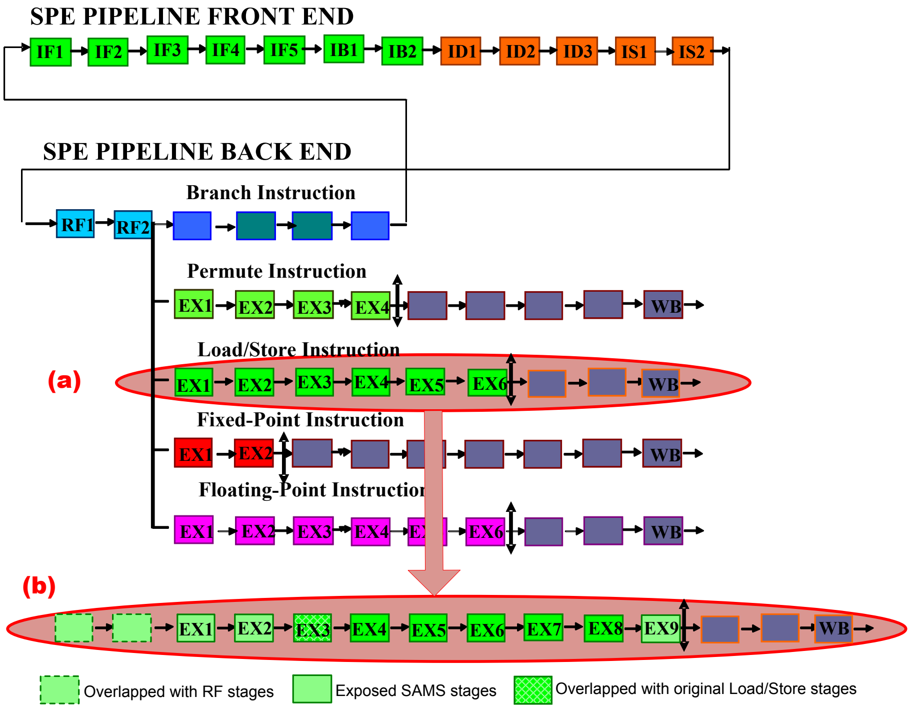


Figure 4.7: Exemplary load/store pipeline optimization: (a)original SPE [57], (b)SAMS integration with optimizations

SPE is an in-order issue processor hence equal number of pipeline stages before RF writeback are required to maintain writeback order.

4.5 Experimental Evaluation

Experimental Setup: We use CellSim developed at BSC [3], which is a cycle-accurate full system simulator for IBM Cell/BE processor. The benchmarks of our experiments consist of some full applications from PARSEC [20], the Wilson-Dirac kernel from INRIA [63], and some micro kernels from IBM Cell SDK [5]. These applications/kernels are selected since they are representative of application codes which operate heavily on array-based data structures. This type of code is widely used in scientific and engineering applications. Table 4.3 lists the major features of the selected benchmarks. Streamcluster from PARSEC is an online clustering kernel which takes streaming data points as input and uses a modified k-median algorithm to do the online clustering. The parameters of streamcluster workload in our study are set as follows: 1024 input points, block size 256 points, 10 point dimensions, 5-10 centers, up to 500 intermediate centers allowed. Fluidanimate is an Intel RMS application and it uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes [20]. The fluidanimate workload in our experiments uses the *simsmall* input set provided by PARSEC. The 3D working domain for one SPE has been shrunk to 9x9x30 cells and the maximal number of particles inside a cell has been reduced to 10. The simulation runs for one time step to compute one frame. Computing the actions of Wilson-Dirac operator contributes most of the CPU time in the simulation of Lattice Quantum Chromodynamics (Lattice QCD), which aims at understanding the strong interactions that bind quarks and gluons together to form hadrons [63]. The experiments in our study with the Wilson-Dirac kernel focus on single SPE for floating point data, with the 4-way runtime data fusion scheme proposed in [63]. The problem size for the Wilson-Dirac kernel for single SPE is 128 output spinors, with a computation intensity of 1608 FP (floating point operations) per output spinor. Besides full applications and large kernels, we also include some micro kernels, including complex number multiplication [62] with workload set to 10K multiplications and 4x4 matrix transpose [5] (with workload set to 10K transposes).

Table 4.3: Selected benchmark suit

Benchmark	**Source**	**Type**	**Application Domain**	**Working Set**
streamcluster	PARSEC	kernel	data mining	medium
fluidanimate	PARSEC	application	animation	large
Wilson-Dirac Operator	INRIA	kernel	quantum physics	medium
complex multiplication	Cell SDK	micro kernel	-	small
matrix transpose	Cell SDK	micro kernel	-	small

To compile the C code, we use two stand-alone compilers: PPU toolchain 2.3 (based on gcc 3.4.1) and SPU toolchain 3.3 (based on gcc 4.1.1). All benchmark applications and kernels are compiled with the -O1 option.

To make the functionalities of the SAMS Multi-Layout Memory available to software, we have extended the SPU ISA and the programming interface. Table 4.4 lists some of the new instructions, C intrinsics and macros for the en-

Table 4.4: SAMS instructions, intrinsics and macros

Name	**Type**	**Operation**
setstride	instruction	set stride register
lqwsd/x/a	instruction	load a quad word with $stride = stride\ register$ and $s = log_2(stride)^\dagger$, d/x/a-form
stqwsd/x/a	instruction	store a quad word with $stride = stride\ register$ and $s = log_2(stride)^\dagger$, d/x/a-form
lqwsfd/x/a	instruction	load a quad word with $stride = 1$ and $s = log_2(stride\ register)^\dagger$, d/x/a-form
stqwsfd/x/a	instruction	store a quad word with $stride = 1$ and $s = log_2(stride\ register)^\dagger$, d/x/a-form
spu_sams_setstride (imm)	intrinsic	set stride register to value imm
spu_sams_lqws (a)	intrinsic	load a quad word at base address a with $stride = stride\ register$ and $s = log2(stride)^\dagger$ (unified form for lqwsd/x/a)
spu_sams_stqws (a,val)	intrinsic	store quad word val at base address a with with $stride = stride\ register$ and $s = log2(stride)^\dagger$ (unified form for stqwsd/x/a)
spu_sams_lqwsf (a)	intrinsic	load a quad word at base address a with $stride = 1$ and $s = log2(stride\ register)^\dagger$ (unified form for lqwsfd/x/a)
spu_sams_stqwsf (a,val)	intrinsic	store quad word val at base address a with with $stride = 1$ and $s = log2(stride\ register)^\dagger$ (unified form for stqwsfd/x/a)
BEGIN_MULTI_ VIEW(str)	macro	spu_sams_setstride(sizeof(str))
SOA_GET(a)	macro	spu_sams_lqws(a)
SOA_PUT(a,val)	macro	spu_sams_stqws(a,val)
AOS_GET(a)	macro	spu_sams_lqwsf(a)
AOS_PUT(a,val)	macro	spu_sams_stqwsf(a,val)
AOS_DMA_GET (la,ga,size,tag,str)	macro	spu_sams_mfcdma64(la,mfc_ea2h(ga),mfc_ea2l(ga),size,tag, MFC_GET_CMD,log2sizeof(str)‡)
AOS_DMA_PET (la,ga,size,tag,str)	macro	spu_sams_mfcdma64(la,mfc_ea2h(ga),mfc_ea2l(ga),size,tag, MFC_PUT_CMD,log2sizeof(str)‡)

† Stride family(s) calculation is done in AGU (see Figure 4.5b).

‡ *log2sizeof* is a new C keyword we implemented in spu-gcc (log2sizeof(str)=log_2(sizeof(str)), where log_2 is done at compile time).

hanced SPE with SAMS integration (also referred to as “SAMS SPE”). To reflect the changes in the architecture, we have modified the spu-gcc backend to generate optimal code for the SAMS SPE, including automatic selection of appropriate instructions for unaligned memory access and flexible access granularity. The load latency and branch penalty have also been updated for proper instruction scheduling. Besides the compiler, the CellSim simulator has also been modified accordingly.

Before elaborating on the experiments, we compose a list on major changes of the SAMS SPE over the original SPE and their cons and pros in Table 4.5. Note, the pipeline optimizations discussed in Section 4.4.2 are not applied in our evaluation.

It should be noted that although there are eight SPEs available in the Cell processor, we only use a single one in our experiments, since we want to focus on the performance impact of the SAMS Multi-Layout Memory on SIMDization. Our techniques are orthogonal to those for efficient parallelization of data parallel applications on multiple processor cores.

Benchmarks SIMDization: In both applications of streamcluster and fluidanimate, the house keeping work (such as data preparation), scalar code dominated by branches (such as building the neighbor table in fluidanimate), and work suited to be done in the global scope (e.g., rebuilding the grid in fluidanimate) is done by PPU, while the majority of the computation is offloaded to the SPU. When data are ready in the main memory, the PPU triggers SPU to start processing. SPU reads a portion of data into its local store by DMA transfers, processes them and writes (using DMA) the results back to main memory. The baseline Wilson-Dirac kernel is already SIMDized and heavily optimized on the original SPE using SoA SIMDization scheme. Therefore, our optimizations based on the SAMS SPE only involve the elimination of dynamic data format conversion overhead. The micro kernels of complex number multiplication and matrix transpose are normally used as one step in a sequence of data operations in the SPE local store. Therefore, DMA transfers are not invoked in our experiments with them.

It should be noted that for all experiments except streamcluster, the SoA SIMDization scheme is adopted since it gives better performance over AoS, in both the original and the SAMS SPEs. As for the memory access patterns, the AoS access of the SAMS local-store is used in all benchmarks for data transfers between the main memory and the local-store (illustrated in Figure 4.1c). During execution, however, four benchmarks access the local-store with the SoA access and one - streamcluster - uses both SoA/AoS.

Table 4.5: Changes over the original SPE and their impact

	Changes over original SPE	**Impact on overall performance, hardware cost and programming**
cons	Latency of store is increased from 4 cycles to 7 cycles	Normally no noticeable (negative) impact on performance unless there is serious register spilling.
	Latency of load is increased from 6 cycles to 10 cycles	May incur significant performance degradation if instruction execution could not hide the long load latency.
	Penalty of a taken branch is increased from 18 cycles to 21 cycles[*]	Impact on overall performance is usually negligible since the portion of (taken) branches in total executed instructions is normally small in SPU.
	Area overhead	Trivial - less than 2 32-bit adders (see Table 4.2).
pros	Reduction of number of executed loads/stores as a result of hardware support for flexible memory access alignment, granularity and stride	Reduction in execution time and memory traffic (This is particularly favorable in Cell SPE since reduction in LS traffic also decreases the chance of LS port conflict[⋆]).
	Reduction of *glue* instructions for data rearrangement (e.g. data format transform between AoS and SoA)	Reduction in execution time can be considerable when a nontrivial number of glue instructions are incurred in SPU code due to data format rearrangement.
	Support for unaligned and strided memory accesses with intrinsics; Support for multiple views of data in memory with easy-to-use abstractions	Programmer and compiler friendly: hardware takes care of details of low-level data layouts in memory and data format conversions; programmers and compilers are relieved from such burdens and therefore can focus on high-level optimizations of the application.

[*] Only the inbound path(3 stages) are exposed to instruction fetch.

[⋆] Note the single-port LS is shared by data load/store, instruction fetch and DMA transfer.

Table 4.6: SPU dynamic instruction count and execution time

Benchmark	**Memory Instruction Count**			**Total Instruction Count**			**Execution Time (cycles)**		
	Original SPE (load/store)	SAMS SPE (load/store)	R	Original SPE	SAMS SPE	R	Original SPE	SAMS SPE	R
streamcluster	12,790,400 (11,084,211/ 1,706,189)	10,331,056 (9,079,100/ 1,251,956)	19%	78,850,693	45,117,363	43%	251,783,087	224,622,745	11%
fluidanimate	4,888,924 (3,043,109/ 1,845,815)	1,772,207 (680,436/ 1,091,771)	64%	33,106,575	22,515,460	32%	95,284,379	74,148,296	22%
Wilson-Dirac Operator	13,863 (13,077/786)	11,375 (10,551/824)	18%	77,038	63,510	18%	75,975	62,035	18%
complex multiplication	15,392 (10,256/5,136)	15,419 (10,272/5,147)	0%	42,008	26,693	36%	96,242	73,137	24%
matrix transpose	81,950 (40,974/40,976)	81,979 (40,992/40,987)	0%	167,766	85,893	49%	283,121	179,690	37%

Experimental Results: We evaluate the performance by measuring the application execution time on SPU. Table 4.6 suggests the experimental results.

For the streamcluster benchmark, it is not obvious whether the AoS or the SoA SIMDization scheme gives better performance, since it depends on the input data, the load latency and the quality of spu-gcc instruction scheduling. Although the critical loop is SIMDized with the AoS SIMDization scheme, for the rest of the code, the loops involving distance calculation are SIMDized with SoA scheme, to achieve better SIMD datapath utilization. The major performance improvement of the SAMS SPE comes from the support for unaligned vector and scalar memory accesses in the SAMS local store, as shown by the 19% reduction of memory instructions and 43% of total instructions. However, as the two-level indirection has serious negative impact on performance especially in SAMS SPE (it has longer load latency), and the large number of branches in the source code which could not be well handled by SPU also incur substantial performance overhead (around 14% execution time is on IF stall for the original SPE and 20% for the SAMS SPE since it has longer IF latency), the overall performance gain is only 11%, as shown in Table 4.6. For such applications, further effort to make major modifications to both the data representation and the control flow at algorithm level would pay-off for better SIMDization performance, on both the original and the SAMS SPEs. Nonetheless, streamcluster represents a class of applications where both AoS and SoA SIMDization schemes are applied on the same data at different application phases. In such cases, the SAMS Multi-Layout Memory's capability of providing multiple data views with high efficiency enables flexible choices of optimal SIMDization schemes in different scenarios.

In fluidanimate, most execution time is spent on computing the density and acceleration for each particle, by accumulating densities and forces between the current particle and all neighboring particles in valid range. Since the data parallelism in computing a single particle pair is very limited, the SoA SIMDization scheme is used to vectorize the code, so that in each batch of processing the interoperation between 4 particles in the current cell and a particle in a neighboring cell are evaluated. Although optimizations are equally deployed in both the original and the SAMS SPEs, the SAMS SPE gives superior performance for three major reasons. First, the 3D position, velocity and acceleration data are maintained in AoS format in main memory, therefore the dynamic format conversion overhead is incurred in the original SPE. Second, as the 3D particle data components are not aligned to 16B memory addresses, an alignment problem occurs in the original SPE. Third, frequent scalar memory access in the code incurs significant performance overhead in the original

SPE. All three overheads are removed in the SAMS SPE, leading to the 22% reduction of execution time as shown in Table 4.6. Note, the number of stores (shown in 3rd column of Table 4.6) in our SoA SIMDized fluidanimate is significantly larger than loads. The reason is explained as follows. As the actual number of particles in current cell (maximal 10) is known only at runtime, therefore 3 batches of processing (dealing with 4, 4, and 2 particles in the current cell respectively) are necessary for computing one neighboring particle. Furthermore, each batch has to update the density and acceleration of the current particles and the neighboring particle. As a result, the stores of the particle data is doubled for current cells with 5∼8 particles and tripled for current cells with 9∼10 particles, compared to current cells with only 1∼4 particles, for each neighboring particle. In contrast to the duplicated stores, the loads of particles in current cell is shared among all particles in all neighboring cells, and the load of the neighboring particle is invoked only once regardless of the number of particles in current cell. This explains the unusual disparity between the number of dynamic loads and stores in the SAMS SPE. In the original SPE, the number of loads (shown in 2nd column in Table 4.6) are larger than stores because each unaligned load incurs two aligned loads and each unaligned store incurs a load-modify-store sequence.

The main problem that prevents SIMDizing the computation of Wilson-Dirac operator efficiently is the multiple patterns of accessing the same spinor and gauge link data [63]. For efficient SIMDization, the authors introduce the runtime data fusion technique in [63], which is basically a SoA SIMDization scheme with rearrangement of data from AoS to SoA format at runtime. Consequently, it also suffers from the overhead of dynamic data format conversion. With our SAMS Multi-Layout Memory, the spinor and gauge link data are accessed in both AoS and SoA formats (for computation) with high efficiency, therefore the data rearrangement overhead is completely eliminated. Additionally, there are 80 loads overhead per 4 output spinors (therefore 20*128=2560 loads overhead in total for 128 spinors) due to partial use of the loaded spinor and gauge filed data in the original code with the original SPE (the partial use of memory bandwidth also results from the mismatch between the data layout in the local store and that used in the SPU). With the SAMS local store, such overhead is effectively removed, resulting the reduced number of executed load instructions. Altogether, the execution time is reduced by 18% in the SAMS SPE, as shown in Table 4.6.

In complex number multiplication, with the common AoS representation of complex number array, load and store of the real/imaginary part of consecutive complex numbers require SoA view of the array. With the multiple view capability of the SAMS memory, the real vector and imaginary vector can be loaded directly with one single strided access, instead of loading the mixture of them and extracting the real and imaginary parts using shuffle instructions, as the source code in [62] did. Therefore, although the memory accesses count is the same for both the original and the SAMS SPEs, the kernel still achieved a significant performance gain with the SAMS SPE as a result of the reduction of the glue instructions.

In 4x4 matrix transpose, if each row (4 elements) of the matrix is treated as a basic structure, and the original matrix is stored in the AoS format, then accessing a row requires AoS view of the row array, while accessing a column requires the SoA view. With the SAMS local store, the transpose procedure is accomplished in a simple manner: first load 4 columns of the input matrix directly (with SoA view) into registers (afterwards the matrix has been effectively transposed), and then store them to the output buffer. Any shuffle instructions to pack the transposed rows from the original rows in the original SPE are completely eliminated. This explains the significant performance improvement by the SAMS SPE with the 37% suggested in Table 4.6.

To summarize, experimental results demonstrate that the SAMS Multi-Layout Memory is a feasible solution for many data rearrangement problems (such as format conversion between AoS and SoA) popular in SIMD processors. Multiple views of array-based data structures provide both programming efficiency and performance improvement for applications operating on such data structures. Additionally, the support for unaligned vector and scalar memory accesses is also a plus for most applications on SIMD processors. Although, the SAMS memory inevitably introduces certain hardware complexity and a longer local store pipeline, with careful consideration of the hardware design trade-offs, the local store latency can be controlled. Further, the SIMD processor can exploit the typical streaming nature of vectorized applications by fetching multiple vectors back-to-back, so that the memory latency can be amortized and tolerated. Therefore the performance degradation due to the longer local store latency are be minimized[6].

---

[6] E.g., 12% performance loss in Wilson-Dirac kernel when local store latency increases from 6 cycles to 10 cycles, with the same code running on the original SPE. Note the overall performance results in Table 4.6 have already taken this factor into account.

4.6 Summary

In this chapter, we proposed the SAMS Multi-Layout Memory to solve the data rearrangement problem in general and to reduce the dynamic data format conversion overhead in particular. The idea is to easily express the preferred view of the data structures in software and let the hardware customize the low-level address mapping logic for optimal data access using this information. Synthesis results for TSMC 90nm CMOS technology node suggest reasonable latency and area overhead of the proposed SAMS memory. To investigate the performance improvement gains, SAMS was integrated into the IBM Cell SPE model, and simulated using real applications. Experiments suggest that, for applications which require dynamic data format conversions between AoS and SoA, our multi-layout memory hardware together with the accompanying software abstractions improve the system performance by up to 37% and simplify the program SIMDization.

Note. The contents of this chapter is based on the the following paper:

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **SAMS Multi-Layout Memory: Providing Multiple Views of Data to Boost SIMD Performance**, Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10), pp. 179–188, Tsukuba, Japan, June 2010. **Best Paper Award**

5

Addressing On-chip Bank Conflicts

One of the major problems with the GPU on-chip shared memory is bank conflicts. We analyze that the throughput of the GPU processor core is often constrained neither by the shared memory bandwidth, nor by the shared memory latency (as long as it stays constant), but is rather due to the *varied latencies* caused by memory bank conflicts. This results in conflicts at the writeback stage of the in-order pipeline and causes pipeline stalls, thus degrading system throughput. Based on this observation, we investigate and propose a novel *elastic pipeline* design that minimizes the negative impact of on-chip memory bank conflicts on system throughput, by decoupling bank conflicts from pipeline stalls. Simulation results show that our proposed elastic pipeline together with the co-designed *bank-conflict aware warp scheduling* reduces the pipeline stalls by up to 64.0% (with 42.3% on average) and improves the overall performance by up to 20.7% (on average 13.3%) for our benchmark applications, at trivial hardware overhead.

5.1 Introduction

The trend is quite clear that multi/many-core processors are becoming pervasive computing platforms nowadays. GPU is one example that uses massive lightweight cores to achieve high aggregated performance, especially for highly data-parallel workloads. Although GPUs are originally designed for graphics processing, the performance of many well tuned general-purpose applications on GPUs have established them among one of the most attractive computing platforms in a more general context – leading to the GPGPU (*General-purpose Processing on GPUs*) domain [1].

In manycore systems such as GPUs, massive multithreading is used to hide long latencies of the core pipeline, interconnect and different memory hierarchy levels. On such heavily multithreaded execution platforms, the overall system performance is significantly affected by the efficiency of both on-chip and off-chip memory resources. As a rule, the factors impacting the on-chip memory efficiency have quite different characteristics compared to the off-chip case. For example, on-chip memories tend to be more sensitive to dynamically changing latencies, while bandwidth limitations are more severe for off-chip memories. In the particular case of GPUs, the on-chip first level memories, including both the software managed shared memories and the hardware caches, are heavily banked, in order to provide high bandwidth for the parallel SIMD lanes. Even with adequate bandwidth provided by the parallel memory banks, however, applications can still suffer drastic pipeline stalls, resulting in significant performance losses. This is due to unbalanced accesses to the on-chip memory banks. This increases the overhead in using on-chip shared memories, since the programmer has to consider the bank conflicts. Furthermore, often the GPU shared memory utilization is constrained by such overhead.

In this chapter, we analyze that the throughput of the GPU processor core is often hampered neither by the on-chip memory bandwidth, nor by the on-chip memory latency (as long as it stays constant), but rather by the varied latencies due to memory bank conflicts, which end up with writeback conflicts and pipeline stalls in the in-order pipeline, thus degrading system throughput. To address this problem, we will investigate novel *elastic pipeline* design that minimizes the negative impact of on-chip memory bank conflicts on system throughput. More precisely, this chapter makes the following contributions:

- careful analysis of the impact of GPU on-chip shared memory bank conflicts on pipeline performance degradation;
- a novel *elastic pipeline* design to alleviate on-chip shared memory conflicts and boost overall system throughput;
- co-designed *bank-conflict aware warp scheduling* technique to assist our elastic pipeline hardware;
- pipeline stalls reductions of up to 64.0% leading to overall system performance improvement of up to 20.7% under realistic scenario.

The remainder of the chapter is organized as follows. In Section 5.2, we provide the background and motivation for this work. In Section 5.3, we analyze the GPU shared memory bank conflicts problem from latency and bandwidth perspective, and identify the mechanism through which shared memory bank conflicts degrade GPU pipeline performance. Based on the findings, we discuss our proposed elastic pipeline design in Section 5.4. The co-designed bank-conflict aware warp scheduling technique is elaborated in Section 5.5. Simulated performance of our proposed elastic pipeline in GPGPU applications is evaluated in Section 5.6, followed by some general discussions of our simulated GPU core architecture along with the elastic pipeline in Section 5.7. Finally, Section 5.8 summarizes the chapter.

5.2 Background and Motivation

In this section, we will first introduce some GPU related background and their shared memory accesses. Then we provide a motivating example.

5.2.1 Shared Memory Access on GPU

GPU utilization has spanned far beyond graphics rendering, covering a wide spectrum of general-purpose computing known as GPGPU [1]. The programming models of GPU (such as OpenCL [8] and CUDA [109]) are generally referred to as *explicitly-parallel, bulk-synchronous* SPMD (*Single Program Multiple Data*). In such programming models, the programmer extracts the data-parallel section of the sequential application code, identifies the basic working unit (typically an element in the problem domain), and explicitly expresses the same sequence of operations on each working unit in a *kernel*. Multiple kernel instances (called *threads* in CUDA) are running independently on GPU cores. The parallel threads are organized into a two-level hierarchy, in which a kernel (*grid* in CUDA) consists of parallel CTAs (*Cooperating Thread Array*, or *block* in CUDA), with each CTA composed of parallel threads, as shown in Figure 5.1(a). Explicit, localized synchronizations and on-chip data sharing mechanisms (such as CUDA shared memory) are supported inside each CTA.

During execution, a batch of threads from the same CTA are grouped into a *warp*, which is the smallest unit for the pipeline front-end processing (i.e., warp scheduling, fetching and decoding stages in Figure 5.1(b)) in GPU cores, as illustrated in Figure 5.1. For high efficiency, warps are executed on the fine-grain multithreaded GPU core in a SIMD fashion. Figure 5.1 shows a warp configuration of 5 threads per warp and a SIMD data path consisting of five lanes. The warps are scheduled and issued to the pipeline in an *interleaved* manner which is also known as *barrel processing* [141].

GPUs rely mainly on massive hardware multithreading to hide external DRAM

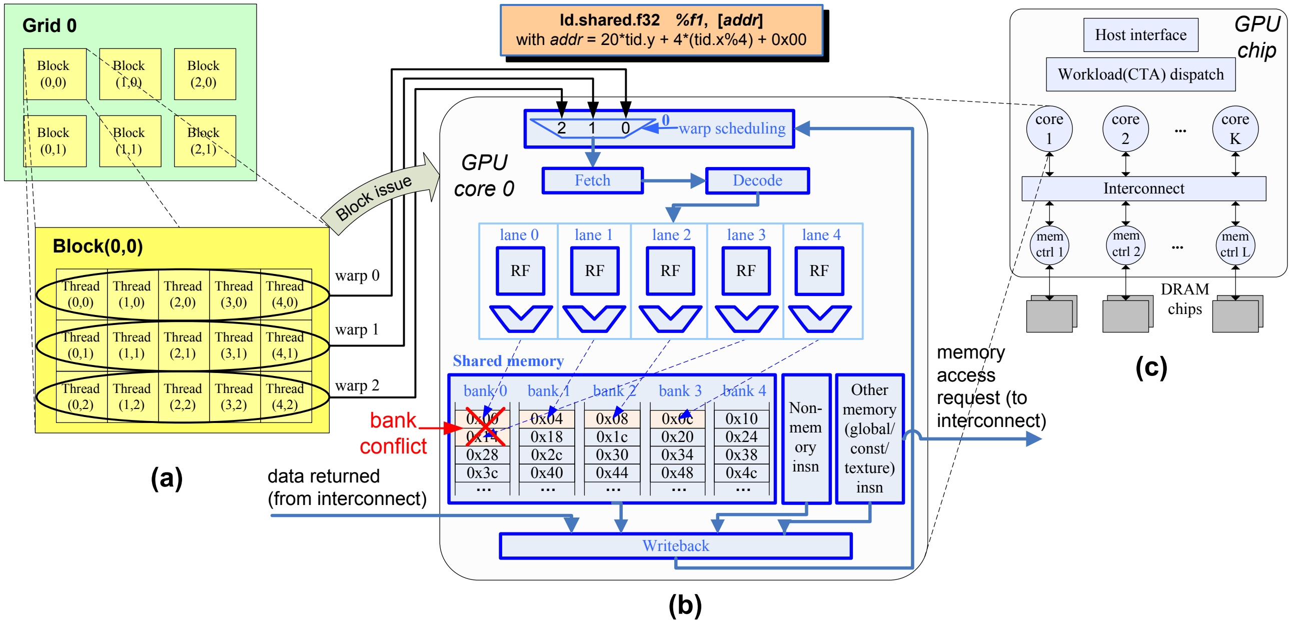
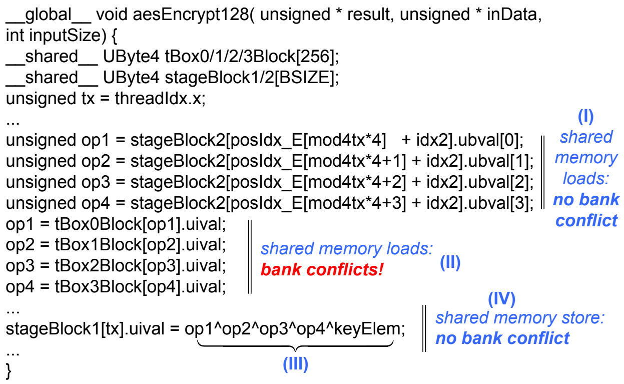


Figure 5.1: (a)CUDA threads hierarchy; (b)thread execution in GPU core pipeline; (c)GPU chip organization



```
__global__ void aesEncrypt128( unsigned * result, unsigned * inData,
int inputSize) {
__shared__ UByte4 tBox0/1/2/3Block[256];
__shared__ UByte4 stageBlock1/2[BSIZE];
unsigned tx = threadIdx.x;
...
unsigned op1 = stageBlock2[posIdx_E[mod4tx*4]   + idx2].ubval[0];
unsigned op2 = stageBlock2[posIdx_E[mod4tx*4+1] + idx2].ubval[1];
unsigned op3 = stageBlock2[posIdx_E[mod4tx*4+2] + idx2].ubval[2];
unsigned op4 = stageBlock2[posIdx_E[mod4tx*4+3] + idx2].ubval[3];
op1 = tBox0Block[op1].uival;
op2 = tBox1Block[op2].uival;
op3 = tBox2Block[op3].uival;
op4 = tBox3Block[op4].uival;
...
stageBlock1[tx].uival = op1^op2^op3^op4^keyElem;
...
}
```

Figure 5.2: AES source code

latencies. In addition, on-chip memory hierarchies are also deployed in GPUs in order to provide high bandwidth and low latency. Such on-chip memories include, software managed caches (shared memory), or hardware caches, or a combination of both [51]. To provide adequate bandwidth for the GPU parallel SIMD lanes, the shared memory is heavily banked. However, when accesses to the shared memory banks are unbalanced, shared memory bank conflicts occur. For example, with the memory access pattern shown on top of Figure 5.1(b), data needed by both lanes 0 and 4 reside in the same shared memory bank 0. In this case a *hot bank* is formed at bank 0, and the two *conflicting* accesses have to be *serialized*, assuming a single-port shared memory design[1]. As a result, the GPU core throughput may be substantially degraded, as to be exemplified by the following example.

5.2.2 Motivating Example

A snapshot of the AES encryption kernel source is shown in Figure 5.2. The code shown there deals with the second encryption stage. First, the stage input data indexes are loaded from shared memory region *stageBlock2* (phase I). Then the stage input data are loaded from shared memory regions *tBox*Block* (phase II), with the indexes from phase I. Afterwards the data is processed (phase III), and finally stored to the shared memory region *stageBlock1* (phase IV). The other stages of the encryption process work similarly. In phase II an irregular access pattern called *indirection* or *gather* is required, causing

---

[1] Even with dual-port shared memory banks, such serialization can not be completely avoided when the bank conflict degree is higher than two.

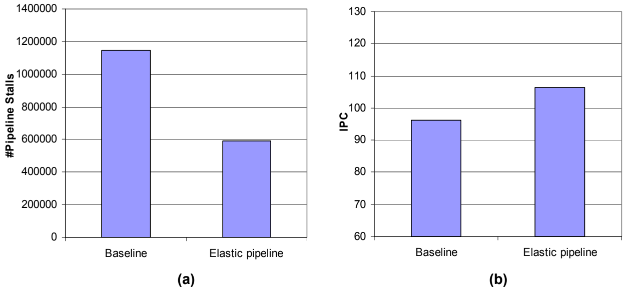


Figure 5.3: Effect of elastic pipeline in: (a)reducing pipeline stalls and (b)improving performance

shared memory bank conflicts during AES execution. As a result, the kernel suffers from a large number of pipeline stalls and non-trivial performance loss. With our proposed elastic pipeline design (Section 5.4) together with the bank-conflict aware warp scheduling technique (Section 5.5), the number of pipeline stalls is reduced by 48.2%, which translates to an overall performance improvement of 10.5%, as Figure 5.3 shows.

5.3 Problem Analysis

In this section, we will first analyze the latency and bandwidth implications of GPU shared memory bank conflicts, then identify and analyze the mechanism how shared memory bank conflicts degrade GPU pipeline performance.

5.3.1 Latency and Bandwidth Implications

GPUs use a large number of hardware threads to hide both function unit and memory access latency. Such extreme multithreading requires a large amount of parallelism. The needed parallelism, using “Little’s law” [92], can be calculated as follows:

$$\mathit{Needed\_parallelism} = \mathit{Latency} \times \mathit{Throughput} \tag{5.1}$$

For GPU throughput cores, this means the required number of in-flight operations to maintain peak throughput equals the product of pipeline latency and SIMD width. For the *strict* barrel processing (See Section 5.7) where all in-flight operations are from different HW thread contexts, this directly determines the required amount of concurrent threads. For other cases in general, the $needed\_parallelism$ is proportional to the concurrent threads number. As a result, a moderate increase in pipeline latency can be effectively *hidden* by running more threads. For example, the GPU core configuration used in our evaluation employs a 24-stage pipeline with SIMD width of 8[2]. Hence, assuming four extra stages for tolerating shared memory bank conflicts, the pipeline depth is increased from 24 stages to 28. In this case, the number of threads to hide the pipeline latency (function unit/shared memory) is penalized, by an increment from 192 to 224, according to Equation 5.1. This is normally not a problem, for both the GPU cores (where there are adequate hardware thread contexts), and the application domains targeted by GPUs (with enough parallelism available).

It is also worth noting that, unlike CPUs, program control flow speculation is not needed in GPUs thanks to the barrel processing model [141]. Therefore, the increase in pipeline latency will not incur pipeline inefficiency associated to deeper pipelines [134].

On the other hand, the peak bandwidth of the shared memory is designed to feed the SIMD core, as illustrated by the pipeline model shown in Figure 5.1(b)), where the number of shared memory banks equals $simd\_width$[3]. Therefore, bandwidth is naturally not a problem when GPU shared memory works at peak BW. However, intuitively, when the shared memory bank conflicts are severe, the *sustained* bandwidth can drop dramatically. Moreover, it may eventually become the bottleneck of the entire kernel execution. In such cases no microarchitectual solution exists without increasing the shared memory raw bandwidth.

To facilitate our discussion, we use the following definition of *bank conflict degree* of SIMD shared memory access:
Bank conflict degree: the maximal number of simultaneous accesses to the same bank during the same SIMD shared memory access from the parallel lanes. Following this definition, the conflict degree of a SIMD shared memory access ranges from 1 to *simd_width*. For example, the SIMD shared memory access in Figure 5.1(b) has a conflict_degree of 2. In general, it takes $\left\lceil \frac{conflict\_degree}{\#shared\_memory\_ports} \right\rceil$ cycles to read/write all data for a SIMD shared memory access.

---

[2] This is in accordance with a contemporary NVIDIA GTX280 architecture [108]. Similar pipeline configurations are also widely used in research GPU models [17, 150].

[3] Note, in practical implementations, the number of shared memory banks can be multiple of (e.g., $2X$) the SIMD width, all running at a lower clock frequency compared to the core pipeline. The bottom line is, the peak BW of the shared memory is at least be capable of feeding the SIMD core [108].

Table 5.1: Benchmark shared memory BW requirement

Benchmark name	r	$conflict\_degree_{avg}$	$r \times conflict\_degree_{avg}$
AES_encypt	36.1%	1.54	0.56
AES_decypt	35.9%	1.53	0.55
Reduction	4.0%	3.07	0.12
Transpose	3.7%	4.50	0.17
DCT	21.6%	3.33	0.72
IDCT	21.2%	3.33	0.71
DCT_short	10.3%	2.75	0.28
IDCT_short	10.3%	2.75	0.28

Naturally, it depends on an application's shared memory access *intensity* and *conflict degree* whether or not it is shared memory BW bound. Assume the available shared memory BW, BW_{avail}, allows access of one data element per core cycle for all SIMD lanes. In this case, a single shared memory instruction's bandwidth requirement equals its $conflict\_degree$. This is due to the fact that this instruction occupies the shared memory for $conflict\_degree$ cycles during the execution. Suppose the ratio between #executed shared memory access instructions and all instructions is r. Then the shared memory BW can become a bottleneck *if and only if* the available BW is smaller than the required BW, for the entire GPU kernel execution:

$$BW_{avail} < BW_{req}$$

i.e.,

$$1 < r \times IPC_{nor} \times conflict\_degree_{avg}$$

Considering the normalized IPC per SIMD lane, IPC_{nor}, is no larger than 1, we see that the shared memory BW is a bottleneck *iff*

$$r \times conflict\_degree_{avg} > 1 \tag{5.2}$$

Table 5.1 shows the values of r (denoted as "shared memory intensity" in Table 5.5) and $conflict\_degree_{avg}$ ("average bank conflict degree" in Table 5.5) in real kernel execution. As can be observed in Table 5.1, Equation 5.2 holds for none of the GPU kernels in our experiments. This indicates we have large shared memory *BW margin*, as far as bank conflicts are concerned. In other words, we have sufficient amount of shared memory BW to sustain peak IPC, even bank conflicts.

This insight is very important, since it reveals the opportunity to improve overall performance, without increasing the raw shared memory BW. In the rest of the chapter, we will see how moderate microarchitectural refinement can be created to solve the problem.

5.3.2 Bank Conflicts Impact on Pipeline Performance

The baseline in-order, single-issue GPU core pipeline configuration is illustrated on the top of Figure 5.4(a). The warp scheduling stage is not shown, and only one of the parallel SIMD lanes of the execution/memory stages is depicted in Figure 5.4(a) for simple illustration[4]. Meanwhile, although only sub-stages of the memory stage (*MEM0/1*) are explicitly shown in the figure, other stages are also pipelined for increased execution frequency. t_i denotes execution time in cycle i, and W_i denotes warp instruction fetched in cycle i.

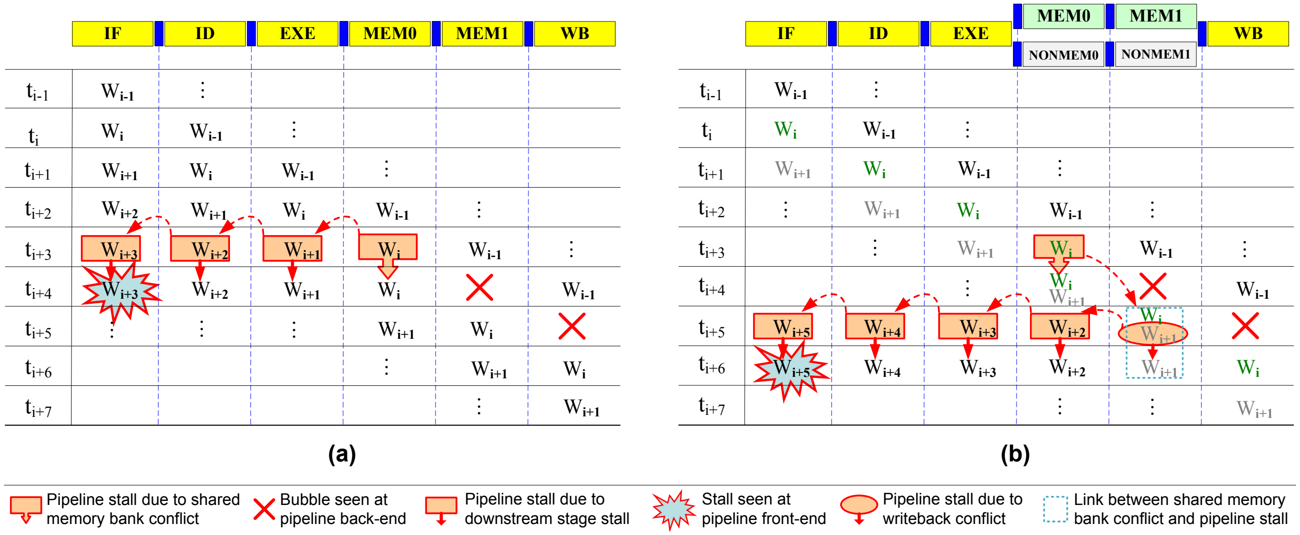
As Figure 5.4(a) shows, W_i is a shared memory access with a conflict degree of 2, and it suffers from shared memory bank conflict in cycle $i+3$, at *MEM0* stage. The bank conflict holds W_i at *MEM0* for an additional cycle (assuming single port shared memory), until it gets resolved at the end of cycle $i+4$. In the baseline pipeline configuration with unified memory stages, the bank conflict in cycle $i+3$ has two consequences: (1) it blocks the upstream pipeline stages in the same cycle, thus incurring a pipeline stall which is finally observed by the pipeline front-end in cycle $i+4$; (2) it introduces a bubble into the *MEM1* stage in cycle $i+4$, which finally turns into a writeback bubble in cycle $i+5$.

Notice the fact that W_{i+1} execution does not have to be blocked by W_i, if W_{i+1} is not a shared memory access. Thus a possible pipeline configuration which is able to eliminate the above mentioned consequence (1) is possible, as Figure 5.4(b) shows. With the help of the extra *NONMEM* path, W_{i+1} is now no longer blocked by W_i, instead it steps into the *NONMEM* path while W_i is waiting at stage *MEM0* for the shared memory access conflict to be resolved, as Figure 5.4(b) shows. Unfortunately, this cannot avoid the writeback bubble in cycle $i+5$. Moreover, the bank conflict of W_i in cycle $i+3$ causes writeback conflict[5] at the beginning of cycle $i+6$, which finally incurs a pipeline stall at fetch stage in the same cycle, as shown in Figure 5.4(b).

---

[4] Please refer to Figure 5.1(b) for the pipeline details.

[5] Note the writeback throughput for a single issue pipeline is 1 instruction/cycle at maximum.



	IF	ID	EXE	MEM0	MEM1	WB
t_{i-1}	W_{i-1}	⋮				
t_i	W_i	W_{i-1}	⋮			
t_{i+1}	W_{i+1}	W_i	W_{i-1}	⋮		
t_{i+2}	W_{i+2}	W_{i+1}	W_i	W_{i-1}	⋮	
t_{i+3}	W_{i+3}	W_{i+2}	W_{i+1}	W_i	W_{i-1}	⋮
t_{i+4}	W_{i+3}	W_{i+2}	W_{i+1}	W_i	✗	W_{i-1}
t_{i+5}	⋮	⋮	⋮	W_{i+1}	W_i	✗
t_{i+6}				⋮	W_{i+1}	W_i
t_{i+7}					⋮	W_{i+1}

(a)

	IF	ID	EXE	MEM0 NONMEM0	MEM1 NONMEM1	WB
t_{i-1}	W_{i-1}	⋮				
t_i	W_i	W_{i-1}	⋮			
t_{i+1}	W_{i+1}	W_i	W_{i-1}	⋮		
t_{i+2}	⋮	W_{i+1}	W_i	W_{i-1}	⋮	
t_{i+3}		⋮	W_{i+1}	W_i	W_{i-1}	⋮
t_{i+4}			⋮	W_i W_{i+1}	✗	W_{i-1}
t_{i+5}	W_{i+5}	W_{i+4}	W_{i+3}	W_{i+2}	W_i W_{i+1}	✗
t_{i+6}	W_{i+5}	W_{i+4}	W_{i+3}	W_{i+2}	W_{i+1}	W_i
t_{i+7}				⋮	⋮	W_{i+1}

(b)

Pipeline stall due to shared memory bank conflict
Bubble seen at pipeline back-end
Pipeline stall due to downstream stage stall
Stall seen at pipeline front-end
Pipeline stall due to writeback conflict
Link between shared memory bank conflict and pipeline stall

Figure 5.4: Baseline in-order pipeline: (a)unified memory stages and (b)split memory stages

Our observation: Through the above analysis, we can see that the throughput of the GPU core is constrained neither by the shared memory bandwidth, nor by the shared memory latency (as long as it stays constant), but rather by the varied execution latencies due to blocking memory bank conflicts. The variation in execution latency incurs writeback bubbles and writeback conflicts, which further causes pipeline stalls in the in-order pipeline. As a result of the above the system throughput is decreased.

5.4 Elastic Pipeline Design

Based on the analysis of the GPU shared memory bank conflict problem, we will introduce our elastic pipeline design in this section. Its implementation will be presented, with emphasis on the conflict tolerance, hardware overhead and pipeline timing impact.

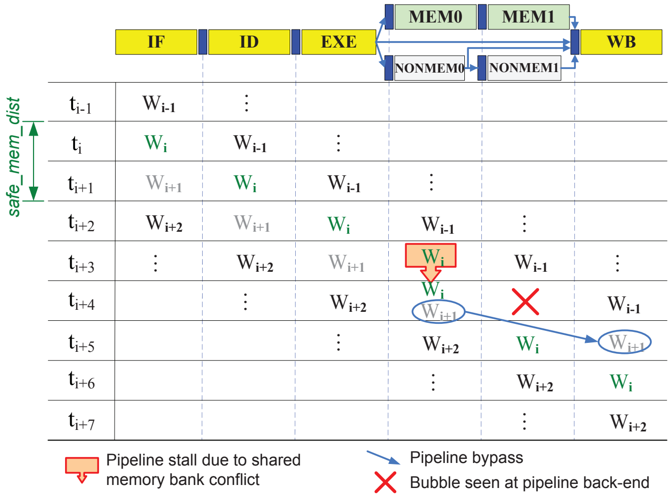


Figure 5.5: Elastic pipeline

To address the problem discussed above, we propose an elastic pipeline design which is able to eliminate the negative impact of shared memory bank conflicts on system throughput, as shown in Figure 5.5. Compared with the baseline pipeline with split memory stages in Figure 5.4(b), the major change is the added buses to forward *matured instructions* from *EXE* and *NONMEM0/1* stages to the writeback stage. This effectively turns the original 2-stage *NONMEM* pipeline into a 2-entry FIFO queue (we will refer to it as "*NONMEM* queue" hereafter). Note, the output from the *EXE* stage can be forwarded directly to writeback only if it is **not** a memory instruction, whereas forwarding from *NONMEM0* to writeback is always allowed. Such non-memory instructions can bypass some or all memory stages, simply because they do not need any processing by the memory pipeline. As Figure 5.5 shows, by forwarding matured instructions in the *NONMEM* queue to the writeback stage, the writeback conflict is removed, and thus the link between bank- and writeback conflicts is cut and the associated pipeline stall is eliminated.

5.4.1 Safe Scheduling Distance and Conflict Tolerance

For ease of discussion, we first define the following warp types:
Memory warp: a warp which is ready for pipeline scheduling and is going to access any type of memory (e.g., shared/global/ constant) in its next instruction execution.
Shared memory warp: a ready warp which is going to access on-chip shared memory in its next instruction execution.
Non-memory warp: a ready warp which is not going to access any memory type during its next instruction execution.

In Figure 5.5 it is assumed that W_{i+1} is a non-memory instruction. Otherwise, W_{i+1} will be blocked at *EXE* stage in cycle $i+4$, since W_i is pending at *MEM0* in the same cycle, due to its shared memory bank conflicts. Such a problem exists even if W_{i+1} is not a shared memory access[6]. To avoid this problem, we have the constraint of *safe memory warp schedule distance*, defined as:
Safe memory warp schedule distance: the minimal number of cycles between the scheduling of a shared memory warp and a following memory warp, in order to avoid pipeline stall due to shared memory bank conflicts.

It is easy to verify the relationship between $safe\_mem\_dist$ (short for "safe memory warp schedule distance") and the shared memory bank conflict degree, in the following equation:

$$safe\_mem\_dist = \left\lceil \frac{conflict\_degree}{\#shared\_memory\_ports} \right\rceil \tag{5.3}$$

The safe memory warp schedule distance constraint requires that memory warps should not be scheduled for execution in next $safe\_mem\_dist - 1$ cycles after a bank-conflicting shared memory warp is scheduled. For example,

---

[6] Note, in such case, even if there exists a third path (with fixed number of stages) for that memory access type, writeback bubbles cannot be avoided, due to the same phenomenon illustrated in Figure 5.4(b).

$\mathit{safe\_mem\_dist}$ for W_i in Figure 5.5 is $\lceil \frac{2}{1} \rceil = 2$, which means that in the next cycle, only non-memory warps can be allowed for scheduling.

It is important to point out that, the elastic pipeline handles bank conflicts of *any degree* without introducing pipeline stalls, as long as the $\mathit{safe\_mem\_dist}$ constraint is satisfied. We will discuss this in more detail in Section 5.5.

5.4.2 Out-of-order Instruction Commitment

In Figure 5.5, the elastic pipeline shows the behavior of out-of-order instruction commitment. For GPU cores with *strict barrel processing* (Section 5.7) (assumed in our evaluation), it is not a problem since the in-flight instructions are from different warps. In the case of *relaxed barrel processing* in which consecutively issued instructions may come from the same warp (but without data dependence), out-of-order instruction commitment within the same execution context may occur. This is penalized by the pipeline being unable to support precise exception. Possible solutions are discussed in Section 5.7.

5.4.3 Extension for Large Warp Size

Above we have assumed *#warp_size=simd_width*. In real GPU implementations, however, the number of threads in a warp can be a multiple of GPU core pipeline SIMD width[7]. In this case, a warp is divided into smaller *subwarps* with the size of each equaling the number of SIMD lanes. All subwarps from the same warp are executed by the SIMD pipeline consecutively. Therefore, *warp_size/simd_width* free issue slots are needed for a warp to be completely issued into the pipeline. Moreover, each warp will occupy the SIMD pipeline for at least *warp_size/simd_width* cycles during execution. Consider for example $\mathit{warp\_size}/\mathit{simd\_width} = 2$. In this case both W_i and W_{i+1} in Figure 5.5 will have to execute the same shared memory access instruction for the first and second half of the same warp, respectively. Since W_i is blocked at stage *MEM0* in cycle $i + 4$, W_{i+1} is unable to step into *MEM0* from the *EXE* stage at the beginning of the same cycle. This results W_{i+1} being blocked at *EXE* and all upstream pipeline stages being blocked in cycle $i + 4$, thus incurring a pipeline stall.

To solve this problem, an extension has to be adopted to the elastic pipeline shown at the top of Figure 5.5. In the extension, we introduce another source

---

[7] For example, there are 32 threads per warp in CUDA and 8 SIMD lanes in NVIDIA GPUs before the Fermi [51] generation.

of elasticity to the *MEM* path, by placing before the *MEM0* stage a ($warp\_size/simd\_width - 1$)-entry FIFO queue ("*PREMEM* queue" in Figure 5.6). With the help of the *PREMEM* queue, the elastic pipeline can handle all consecutive, back-to-back issued bank-conflicting SIMD shared memory accesses from the same warp, regardless of the conflict degree of each.

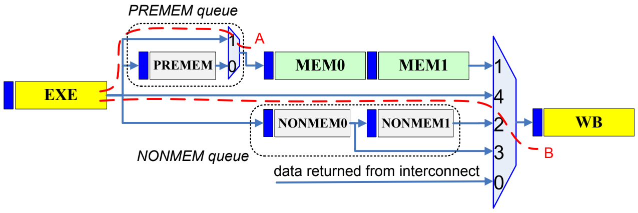


Figure 5.6: Elastic pipeline logic diagram

The logic diagram of the final elastic pipeline with the extension for large warp size is shown in Figure 5.6, for the case with two memory stages and 1-entry *PREMEM* queue. The numbers inside the multiplexers denote the MUX inputs priority (smaller numbers have higher priorities). For example, the data returned from interconnect is assigned with the highest priority (it is loaded from the external main memory after hundreds of cycles delay); whereas only when there is no available data from elsewhere can the data output from stage *EXE* be written back (if it is not a memory access).

With the elastic pipeline configuration of Figure 5.6, W_{i+1} in Figure 5.5 will be buffered in the *PREMEM* queue in cycle $i+4$, while W_{i+2} will directly step into writeback stage at the beginning of cycle $i+5$.

To summarize, the elastic pipeline adds two FIFO queues to the baseline pipeline: the *NONMEM* queue with a depth of M and the *PREMEM* queue with a depth of N, where

$$M = \#MEM\_stages \tag{5.4}$$

$$N = \left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1 \tag{5.5}$$

5.4.4 Hardware Overhead and Impact on Pipeline Timing

The additional hardware overhead as compared with the baseline pipeline is summarized in Table 5.2. The metric for the logic complexity of pipeline latches is that of a pipeline latch in a single SIMD lane. As we can see in

Table 5.2: Elastic pipeline HW overhead per GPU core

Type	Logic complexity	Quantity
Pipeline latches	$simd\_width$	M+N
(M+3)-to-1 MUX	M+2	1
(N+1)-to-1 MUX	N	1

Table 5.2, the area consumption of the additional pipeline latches is in the order of $(M + N) \cdot simd\_width$. Considering small Ms and Ns in realistic GPU core pipeline designs (e.g. M=4, N=3 in our evaluation), this additional cost is well acceptable. The hardware overhead of the two multiplexers is negligible.

The control paths of the two multiplexers are not shown in Figure 5.6, since they are simply valid signals from relevant pipeline latches at the beginning of each stage, and are therefore not in the critical path. Compared with the baseline pipeline, all other pipeline stages' timing is untouched, with only one exception of the *EXE* stage, as illustrated in Figure 5.6[8]. There are two separate paths in which the *EXE* stage is prolonged: path A and B, as marked by the two dash lines in Figure 5.6. A is the (N+1)-to-1 multiplexer and B is the (M+3)-to-1 multiplexer listed in Table 5.2. With standard critical path optimizations such as the *priority on late arriving signal* technique [19], both A and B only incur an additional latency of 2-to-1 MUX for the *EXE* stage. Therefore, the increased latency to stage *EXE* is that of a 2-to-1 MUX in total, which will not noticeably affect the target frequency of the pipeline in most cases (assumed in our experimental evaluation).

5.5 Bank-conflict Aware Warp Scheduling

As discussed in Section 5.4.1, in order to completely avoid the pipeline stall due to shared memory bank conflicts, the constraint of *safe memory warp schedule distance* must be satisfied. Otherwise, two consequences will happen: 1) the *PREMEM* queue will get saturated, which results in pipeline stalls; and 2) the *NONMEM* will get emptied, which results in writeback starvation. In the end the pipeline throughput is degraded. In order to cope with this problem, warp scheduling logic should prevent any memory warp from being

---

[8] Note, although not shown in Figure 5.4(a), there is a MUX at the end of *MEM1* stage in the baseline pipeline, since an arbitration to select writeback data from either inside the GPU core pipeline or from the interconnect is needed.

scheduled in the time frame of *warp_safe_mem_dist* (Equation (5.7)) cycles after a bank-conflicting shared memory warp is scheduled for execution. This is called "*bank-conflict aware warp scheduling*", discussed next.

5.5.1 Obtaining Bank Conflict Information

In order to apply bank-conflict aware warp scheduling, we have to first find out which instructions will cause shared memory bank conflicts, and their corresponding conflict degree. This information may be obtained in two ways: 1) static program analysis; 2) dynamic detection. We chose dynamic bank conflict detection instead of compile-time analysis in our implementation for two reasons. First, some shared memory access patterns (and therefore the bank conflict patterns) are only known at runtime. This is the case for regular access patterns (e.g. 1D strided) whose pattern parameters (e.g., the stride) are not known at compile time, or irregular accesses whose bank conflict patterns can not be identified statically (such as the AES example). Second, there is no additional hardware cost incurred directly by the dynamic detection, as the shared memory bank conflict detection logic is needed in the baseline pipeline[9].

Note, for warp sizes larger than the number of SIMD lanes, the bank conflict degree of the *entire warp* is the accumulation of all subwarp SIMD accesses, as given by the following equation:

$$warp\_bkconf\_degree = \sum_{i=0}^{\left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1} conflict\_degree_i \tag{5.6}$$

where $conflict\_degree_i$ is the shared memory bank conflict degree of subwarp i, which is measured by the hardware dynamically. $warp\_bkconf\_degree$ is obtained by an accumulator and a valid result is generated at fastest every $\left\lceil \frac{warp\_size}{simd\_width} \right\rceil$ cycles (if there is no pipeline stall during that time).

Accordingly, the safe memory warp schedule distance in Equation (5.3) is extended in the following:

$$warp\_safe\_mem\_dist = \left\lceil \frac{\sum_{i=0}^{\left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1} conflict\_degree_i}{\#shared\_memory\_ports} \right\rceil \tag{5.7}$$

---

[9] The shared memory has to identify the conflict degree of each SIMD shared memory access (i.e., $conflict\_degree_i$ in Equation (5.6)) in order to resolve it.

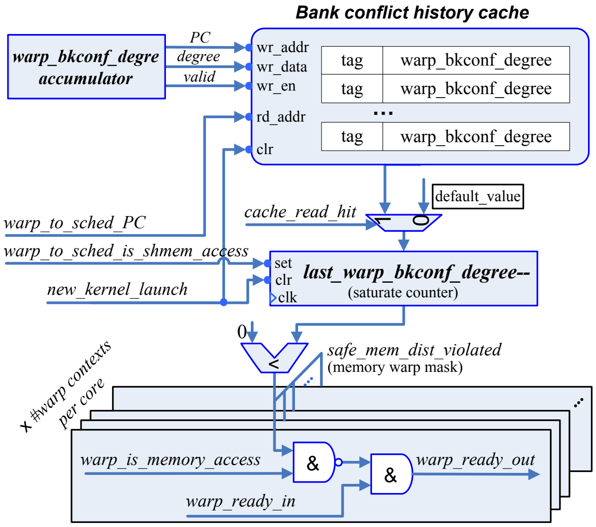


Figure 5.7: Bank-conflict aware warp ready signal generation

And the safe memory warp schedule distance constraint now requires that the scheduling interval between bank-conflicting shared memory warp and memory warp should be no less than *warp_ safe_mem_dist* cycles.

It is very important to note that, the bank conflict degree of the last scheduled shared memory warp can not be obtained *in time* by simply checking the *warp_bkconf_degree* accumulator on the fly. This is due to the fact that it may have not reached memory stages or finished shared memory accesses yet when its bank conflict information is needed by the warp scheduling logic. Therefore, we need to **predict** *warp_bkconf_degree* for a shared memory warp before the real value becomes valid, by only its shared memory instruction PC. In our design, we implement a simple prediction scheme which predicts the bank conflict degree of a shared memory instruction to be the one measured during the *last execution* of the same instruction.

5.5.2 Bank Conflict History Cache

In order to maintain the historic conflict degree information, we implement a small private *bank conflict history cache* distributed among the GPU cores, as shown in Figure 5.7. Each time a new kernel is launched, both the bank conflict history cache and the *last_warp_bkconf_degree* counter are cleared.

The cache is updated whenever a warp execution of shared memory instruction gets resolved and the *warp_bkconf_degree* accumulator generates a valid value for it[10]. Whenever a shared memory warp is scheduled, the *last_warp_bkconf_degree* counter is set to its last warp bank conflict degree in history, by checking it in the conflict history cache. If a cache miss occurs, then the *last_warp_bkconf_degree* counter is set to a default value (0 in our design). The memory warp mask is generated by checking if the safe memory warp schedule distance constraint is violated. Note, in our design we assume the warp scheduling stage knows whether or not a ready warp is a shared memory access (the *“warp_to_sched_is_ shmem_access”* signal in Figure 5.7), or a memory access (the *“warp_is_memory_ access”* signal). This can be done easily with negligible overhead. For example, we can look up the committing warp’s next instruction type in a per-core type bit-vector (initialized at kernel launch time) at pipeline writeback stage (only 2 bits per PC per kernel are enough for this purpose), and setup the 2-bit type register associated with the committing warp (only 2 bits per a hardware warp context).

When we use the bank conflict history cache to predict the conflict degree of scheduled shared memory access, the result is incorrect in two situations: (1) when a cache miss happens (e.g., compulsory misses due to the cold cache after a new kernel is launched) and unfortunately the default output value generated is different from the actual conflict degree; (2) when the conflict degree of the same shared memory instruction varies among consecutive execution. Case (1) is unavoidable for any kernel. Fortunately, its impact on overall performance is usually negligible. Case (2) occurs only in kernels with irregular shared memory access patterns and dynamically changing conflict degree (e.g., AES). It is important to note that, incorrect prediction of the shared memory bank conflict degree in the elastic pipeline will not necessarily result in pipeline stalls. Indeed, the pipeline will be stalled only when the predicted value is *smaller* than the actual conflict degree and there is at least one memory warp scheduled which violates the safe memory warp schedule distance constraint. The impact of incorrect bank conflict degree prediction on pipeline stalls is shown in Section 5.6.1.

---

[10] Note, in our design the conflict degree value from the accumulator has been decreased by $\lceil \frac{warp\_size}{simd\_width} \rceil$ before written to the conflict history cache, in order to *align* the value for instruction with no bank conflict to zero.

5.5.3 Proposed Warp Scheduling

With the bank access conflict history for each shared memory instruction maintained in the conflict history cache, bank-conflict aware warp scheduling can apply the same scheduling scheme as the baseline pipeline to schedule the ready warps for execution. The only difference is that if a previously scheduled warp *will be*/*is still being* blocked at the memory stages due to shared memory bank conflicts, then all memory warps are excluded from the ready warp pool, as Figure 5.7 shows.

Once it is guaranteed by the warp scheduling logic that there is no memory warp violating the safe memory warp schedule distance constraint, then shared memory bank conflicts incurred by a single warp can be effectively handled by the elastic pipeline design, as discussed in Section 5.4. Otherwise, the elastic pipeline will get saturated and stalls due to bank conflicts will occur. We will see the impact of the bank-aware warp scheduling on overall performance in Section 5.6.2. It should be noted that, warp scheduling just by itself is unable to reduce pipeline stalls caused by shared memory bank conflicts, without the elastic pipeline infrastructure.

5.5.4 Hardware Overhead

Table 5.3: Hardware overhead of bank conflict prediction and warp mask generation (per GPU core)

Type	**Logic complexity**	**Quantity**
Bank conflict history cache	$\#cache\_lines \cdot (log_2(warp\_size)+14-log_2(\#cache\_sets))$ bits, dual port (1R+1W)	1
AND/NAND gate	–	2·#warp contexts per core
Accumulator	$log_2(warp\_size)$ bits	1
2-to-1 MUX	–	1
Counter	$log_2(warp\_size)$ bits	1
Comparator	$log_2(warp\_size)$ bits	1

As discussed above, our bank-conflict aware warp scheduling does not incur any additional overhead in scheduling logic – it simply utilizes the same scheduling as the baseline. However, the warp ready signal generation logic needs to be modified to make it aware of in-flight bank-conflicting shared memory accesses and enforce the constraint on following warps to be scheduled, as shown in Figure 5.7. Table 5.3 summarizes the hardware overhead incurred by the bank conflict degree prediction and bank-conflict aware warp ready signal generation. The main contributor in Table 5.3 is the bank conflict degree history cache. Assuming 14 bits PC (which is able to handle kernels with up to 16K instructions – large enough from our experience), this turns to 14-$log_2(\#cache\_sets)$ bits cache tag size. Remember, each $conflict\_degree_i$ in Equation (5.6) takes $log_2(simd\_width)$ bits (Section 5.3.2), therefore the cache content *warp_bkconf_degree* occupies $log_2(warp\_size)$ bits. The total size of the bank conflict history cache is summarized in Table 5.3. In our design, we implemented a 2-way set associative conflict history cache with 256 sets, which is capable of removing all capacity and conflict misses for all kernels in our evaluation. In this case, the conflict history cache consumes only 704 bytes (with *warp_size*=32), which is quite trivial.

Regarding the timing impact, the increase in the warp ready signal generation delay observed by the default warp scheduler is only that of one *AND* gate, as shown in Figure 5.7.

5.6 Experimental Evaluation

Experimental Setup: We use a modified version of GPGPU-Sim [17], which is a cycle-level full system simulator for GPUs implementing ptx ISA [110]. We model GPU cores with a 24-stage pipeline similar to contemporary implementations [108, 150]. The detailed configuration of the GPU processor is shown in Table 5.4. The GPU processor with the baseline pipeline ("baseline GPU") and the case with the proposed elastic pipeline ("enhanced GPU") are evaluated in this chapter. They differ only in the core pipeline configurations and warp scheduling schemes, as Table 5.4 shows. The number of memory pipeline stages and the *warp_size/simd_width* ratio are 4 (see Table 5.4). Therefore the queue depth is set to 4 for the *NONMEM* queue, and 3 for the *PREMEM* queue in the elastic pipeline, according to Equations (5.4) and (5.5).

We selected eight shared memory intensive benchmarks from CUDA SDK [109] and other public sources [98]. Table 5.5 lists the main characteristics of the selected benchmarks. The instruction count in columns *total instructions* and *shared memory instructions* shows two numbers, with the first being the number of dynamic instructions executed by all 128 scalar pipelines (i.e., SIMD lanes) of 16 GPU cores, and the second number being the ptx instruction count in the compiled program. *Shared memory intensity* is the ratio of dynamic shared memory instructions to total executed instructions. The

Table 5.4: The GPU processor configurations

Number of Cores	16
Core Configuration	8-wide SIMD execution pipeline, 24 pipeline stages (with 4 memory stages); 32 threads/warp, 1024 threads/core, 8 CTAs/core, 16384 registers/core; execution model: strict barrel processing (Section 5.7) warp scheduling policy: Round-robin (**baseline GPU**) **vs** bank-conflict aware warp scheduling (**enhanced GPU**) pipeline configuration: baseline pipeline (**baseline GPU**) **vs** elastic pipeline (**enhanced GPU**)
On-chip Memories	16KB software managed cache (i.e., shared memory)/core, 8 banks, 1 access per core cycle per bank
DRAM	4 GDDR3 memory channels, 2 DRAM chips per channel, 2KB page per DRAM chip, 8 banks, 8 Bytes/channel/transmission (51.2GB/s BW in total), 800 MHz bus freq, 32 DRAM request buffer entries memory controller policy: out-of-order (FR-FCFS) [125]
Interconnect	crossbar, 32-Byte flit size

average bank conflict degree field shows the average number of cycles spent on a SIMD shared memory access for each benchmark application. This is collected by running the benchmarks on the baseline GPU. *Theoretic speedup* calculates, assuming IPC=1 (normalized to a single scalar pipeline/SIMD lane) for all instructions except shared memory accesses (i.e., all pipeline inefficiency comes from pipeline stalls caused by shared memory bank conflicts), the speedup that can be gained by eliminating all pipeline stalls. *CTAs per core* denotes the maximal number of concurrent CTAs that can be allocated on each GPU core. A **Y** in the *Irregular shared memory patterns* column indicates kernels with shared memory instructions with irregular access patterns and dynamically varied bank conflict degree.

Note, the kernel names followed by a ⋇ denote the CUDA code which has originally been hand-optimized to avoid shared memory bank conflicts, by changing the layout of the data structures in shared memory (e.g., by padding one additional column to a 2D array). We adopt the code but *undo* such optimizations in our evaluation of elastic pipeline performance in Sections 5.6.1 and 5.6.2. There are two reasons for this. First, we found that in practice if the shared memory bank conflict is a problem, the programmer will either remove it (by the above mentioned hand-optimizations), or simply avoid using the shared memory. Due to this we were unable to find many existing

Table 5.5: Benchmark characteristics

Name	**Source**	**Grid Dim**	**CTA Dim**	**CTAs/core**	**Total Insns**
AES_encypt	[98]	(257,1,1)	(256,1,1)	2	35132928/534
AES_decypt	[98]	(257,1,1)	(256,1,1)	2	35527680/540
Reduction	CUDA SDK	(16384,1,1)	(256,1,1)	4	415170560/49
Transpose⋇	CUDA SDK	(16,16,1)	(16,16,1)	4	3538944/54
DCT⋇	CUDA SDK	(16,32,1)	(8,4,2)	7	7274496/222
IDCT⋇	CUDA SDK	(16,32,1)	(8,4,2)	7	7405568/226
DCT_short⋇	CUDA SDK	(16,16,1)	(8,4,4)	7	10223616/337
IDCT_short⋇	CUDA SDK	(16,16,1)	(8,4,4)	7	10190848/336

Name	**Sh-mem Insns**	**Sh-mem Intensity**	**Avg. Conf. Degree**	**Theoretic Speedup**	**Irregular Sh-mem Pattern**
AES_encypt	12697856/193	36.1%	1.54	1.19	Y
AES_decypt	12763648/194	35.9%	1.53	1.19	Y
Reduction	16744448/5	4.0%	3.07	1.09	Y
Transpose⋇	131072/2	3.7%	4.50	1.13	N
DCT⋇	1572864/48	21.6%	3.33	1.50	N
IDCT⋇	1572864/48	21.2%	3.33	1.50	N
DCT_short⋇	1048576/40	10.3%	2.75	1.18	N
IDCT_short⋇	1048576/40	10.3%	2.75	1.18	N

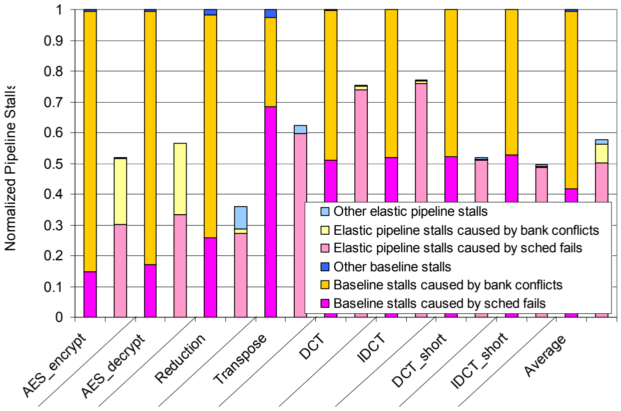


Figure 5.8: Pipeline stall reduction. In each group: left bar: baseline GPU; right bar: elastic pipeline enhanced GPU

codes with heavy shared memory bank conflicts. That is why we manually *roll back* the shared memory hand-optimizations for these kernels and use them in our initial evaluation presented in this chapter. Second, assuming the elastic pipeline is adopted in the GPU core, we also want to inspect how it performs for these kernels, without shared memory optimizations from the programmer.

We use the NVCC toolchain [110] to compile the CUDA application code. The toolchain first invokes *cudafe* to extract and separate the host C/C++ code and device C code from the CUDA source, then it invokes two stand-alone compilers: *gcc* to compile the host C/C++ code running on the CPU and *nvopencc* to compile the device code running on the GPU. All benchmarks are compiled with *-O3* option.

It has to be pointed out that, although we use CUDA code and the corresponding toolchain in our experiments, our proposed elastic pipeline and bank-conflict aware warp scheduling do not rely on any particular GPU programming model. Further, the application of our proposal is not limited to GPGPU applications – graphics kernels can also benefit from it where on-chip shared memory bank conflict is a concern.

5.6.1 Effect on Pipeline Stall Reduction

Figure 5.8 shows the proposed elastic pipeline and the bank-conflict aware warp scheduling effect on reducing pipeline stalls. The results are per kernel, with the left bar of each group showing the number of pipeline stalls in the baseline GPU, and the right bar showing the stalls in the enhanced GPU. The number of stalls are normalized to the baseline GPU. Inside each bar, the pipeline stalls are broken down into three categories (from bottom to top): warp scheduling fails, shared memory bank conflicts, and other reasons (i.e., writeback conflicts incurred by data returned from interconnect). Note, GPU core warp scheduling fails if the ready warp pool is empty, which can be incurred by: (1) the core pipeline latency or other long latencies (e.g., due to main memory access) which are not hidden by the parallel warp execution (i.e., not enough concurrent CTAs active on chip); (2) the barrier synchronization; (3) warp control-flow re-convergence mechanisms [47].

As discussed before, shared memory bank conflicts create writeback bubbles which finally incur pipeline stalls. Note, although the portion of shared memory instructions is small for some kernels (such as Reduction and Transpose, with less than 5% as shown in Table 5.5), some of the involved SIMD shared memory accesses result in very high bank conflict degrees (up to 8). Therefore the total number of bank conflicts and pipeline stalls is quite significant in the baseline, as shown in Figure 5.8.

As Figure 5.8 shows, the number of pipeline stalls are significantly reduced by the elastic pipeline. In all kernels except AES encrypt/decrypt, the pipeline stalls caused by bank conflicts are almost completely removed in the enhanced GPU. Remember, the bank conflict stalls in the elastic pipeline may occur, only if the conflict degree prediction made by the bank conflict history cache is incorrect (Section 5.5.2). The bank conflict history cache was unable to produce constantly precise conflict degree prediction for the highly irregular shared memory access patterns in the AES kernels. This results in a large number of bank conflict stalls. Figure 5.8 shows that the impact of compulsory misses in bank conflict history cache is negligible in the elastic pipeline.

On the other hand, the number of pipeline stalls due to warp scheduling failures are increased for some kernels. This is expected, since the bank-conflict aware warp scheduling masks off the ready warps which violate the constraint of safe memory warp schedule distance. Contrary to our expectation, the number of warp scheduling fails is actually reduced for Transpose and DCT/IDCT_short kernels. Detailed investigation reveals that this is related to the inter operation between our elastic pipeline design and the rest of the GPU processor, such as the on-chip synchronization and control flow re-convergence mechanisms, and off-chip DRAM organizations. For example, drastic DRAM channel conflicts are observed during the Transpose kernel execution on the baseline GPU. Whereas in the GPU enhanced by the elastic pipeline and the bank-conflict

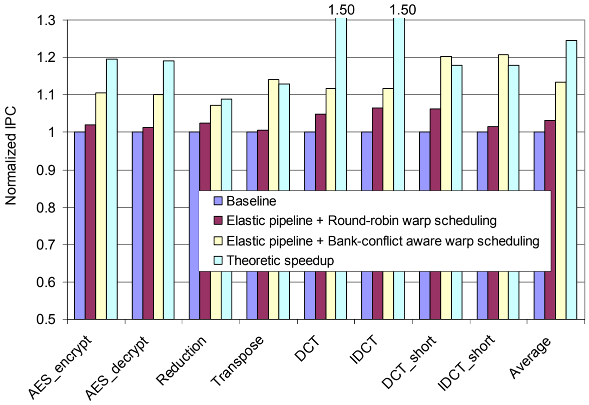


Figure 5.9: Performance improvement

aware warp scheduling, such channel conflicts are substantially reduced and DRAM efficiency is improved.

The last type of pipeline stalls (*other pipeline stalls* in Figure 5.8) is caused by writeback conflicts incurred by data returned from interconnect. The number is slightly increased in the elastic pipeline as shown in Figure 5.8. This is because the number of such conflicts is relatively small, and a large portion of them are well *hidden* by the large amount of bank conflict stalls at the upstream of the pipeline, in the baseline GPU.

5.6.2 Performance Improvements

Figure 5.9 compares the performance of the baseline GPU, the enhanced GPU with pure elastic pipe design (with default warp scheduling), the enhanced GPU with elastic pipeline augmented by bank-conflict aware warp scheduling, and the theoretic speedup. We can see that the performance is improved by the pure elastic pipeline only slightly (3.2% on average), without the assistance of proper warp scheduling. While with the co-designed bank-conflict aware warp scheduling, an additional 10.1% improvement is gained, leading to the average performance improvement of the elastic pipeline design by 13.3%, as compared to the baseline. This confirms our analysis in Section 5.5. For AES encrypt/decrypt, the achieved speedup by the elastic pipeline is substantially smaller than the theoretical bound, mainly because a large portion of bank conflict stalls still remains in the elastic pipeline, as shown in Figure 5.8. DCT and IDCT see a *huge* gap between the actually achieved performance gain by elastic pipeline and the theoretic bound. This is due to the number of pipeline stalls caused by warp scheduling fails is significantly increased, also shown in Figure 5.8. It is interesting to see that the speedup of our elastic pipeline design exceeds the theoretic bound, for kernels Transpose and DCT/IDCT_short. This results from the fact that the number of warp scheduling fails is reduced, thanks to the positive interaction between the elastic pipeline and the rest of the system in these cases, as discussed in Section 5.6.1.

In order to find out how our elastic pipeline performs in relieving the overhead of reducing shared memory bank conflicts from the software side, we compared the performance of un-optimized code (i.e., CUDA SDK code with shared memory bank conflict optimizations removed by us) running on the enhanced GPU versus the hand-optimized code (i.e. the original CUDA SDK code) running on the baseline GPU, as shown in Figure 5.10. As we can see from the figure, on average the performance of un-optimized kernel running on elastic pipeline cores is on par with the optimized kernel running on baseline cores. However, we also found that for the DCT/IDCT kernels, the performance gap is quite large (e.g., 14.9% less performance on the elastic pipeline+un-optimized kernel combination for IDCT). In-depth analysis reveals that this is due to the change of warp execution order by the elastic pipeline interacts poorly with the global memory access which results in degraded DRAM access efficiency. This actually leaves room for further optimizations. For example, a simple variant of our bank-conflict aware warp scheduling allows issuing of memory instructions *violating* the safe memory warp schedule distance, if there are no ready warps to execute non-memory instruction[11]. This variant essentially trades more bank conflict stalls for fewer scheduling fails. Theoretically, the performance should not change since the number of total pipeline stalls is kept the same. However, the performance of IDCT with the variant is increased by 5.1% as compared with the original bank-conflict aware warp scheduling, simply due to the change of warp execution order[12]. This could be further improved by taking into account also the main memory bandwidth efficiency in our design. For example, it may be possible to create more efficient warp scheduling schemes which are aware of not only on-chip shared memory bank-conflict, but also global memory efficiency. More details about such interactions between on/off-chip memory accesses at system level are discussed in Section 5.6.4.

Nonetheless, the results in Figure 5.10 suggest the strong potential of our elas-

---

[11] In the original bank-conflict aware warp scheduling, the ready warp pool is masked to empty in this case and pipeline will be stalled due to scheduling fails.

[12] We did not adopt this variant as it degrades the performance for other kernels.

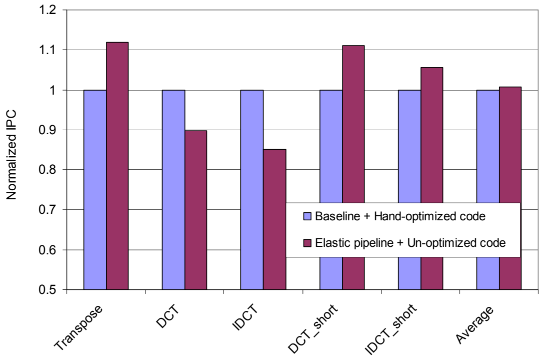


Figure 5.10: Elastic pipeline vs hand-optimized code for conflicting kernels

tic pipeline design to relieve the burden of avoiding shared memory bank conflicts from the programmer. Note also, static program analysis and optimizations are unable to avoid bank conflicts caused by irregular conflict patterns, which can be effectively handled by our proposal as demonstrated by the substantial performance improvement by elastic pipeline for the AES and Reduction kernels in Figure 5.9. Therefore, we can safely draw the conclusion that, our elastic pipeline proposal is capable of relieving the shared memory bank conflict issue for both regular and irregular access patterns, and thus enables more GPGPU applications to exploit the on-chip shared memory for improved performance and efficiency which is not possible without our proposal.

5.6.3 Performance of Non-Conflicting Kernels

Besides bank-conflicting kernels, we also would like to find out to which extent the proposed elastic pipeline will affect the execution of normal kernels without on-chip shared memory bank conflicts. Note, in this case, the bank-conflict aware warp scheduling behaves exactly the same as the default warp scheduling, since the conflict degree predicted by the bank conflict history cache is constantly zero (Figure 5.7). The *non-conflicting* kernels examined in this section are the five kernels whose names are followed by a ⋇ in Table 5.5, with the hand-optimization to avoid shared memory bank conflict (i.e., the original CUDA SDK code is used).

The performance of the non-conflicting kernels (i.e. the original CUDA SDK source code) execution on both the baseline and elastic pipeline cores is shown in Figure 5.11. As we can see in the figure, the difference in performance is

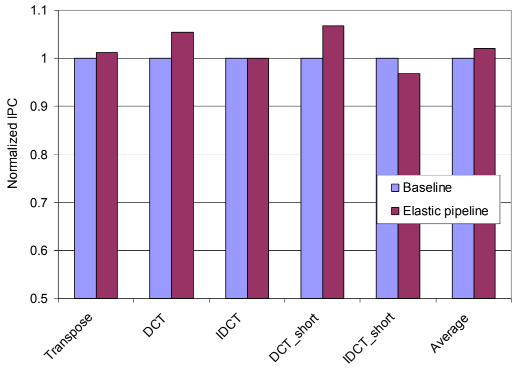


Figure 5.11: Elastic pipeline performance for non-conflicting kernels

negligible. The performance difference between the baseline pipeline and the elastic pipeline for kernels without any bank conflict is due to: (1) the elastic pipeline can hide some of the writeback conflicts caused by the competition between core pipeline instructions (e.g., non-global memory instructions) and global memory loads (Figure 5.1); (2) the writeback MUX in the elastic pipeline (Figure 5.6) changes the default warp completion order of baseline in some cases (e.g., when the *MEM* and *NONMEM* paths compete for writeback, or, when there is a pipeline bypass in the *NONMEM* path (Figure 5.5)), which will further affect warp scheduling and execution order later. Factor (1) is always beneficial while factor (2) can contribute either positively or negatively to overall performance, depending on other subtle conditions (e.g. varied global memory access efficiency, synchronization efficiency and control flow re-convergence efficiency, under different warp execution orders).

5.6.4 Interaction with Off-chip DRAM Access

At first glance, it seems that on-chip shared memory access is decoupled from off-chip DRAM access. Counterintuitively, however, we have already observed quite some inter-operation between them, is the warp execution order (and the subsequent DRAM access patterns): as discussed in Sections 5.6.1 and 5.6.2. Indeed, it would be interesting to inspect the relationship between our proposed elastic pipeline and the kernel DRAM access behavior. Figure 5.12 tries to unveil it in a quantitatively way. The curve at the top shows the ratio between theoretic speedup and the speedup actually gained by our elastic pipeline design (data from Figure 5.9). And the one at the bottom shows the

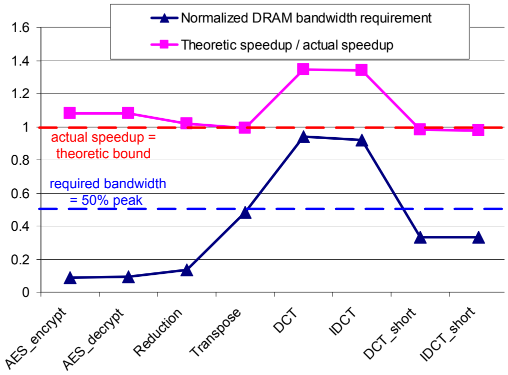


Figure 5.12: DRAM bandwidth impact

required DRAM bandwidth by each kernel, normalized to GPU DRAM peak bandwidth (Table 5.4). The required bandwidth is calculated by dividing the total amount of global memory data access by the execution time assuming IPC=1 for each SIMD lane. We choose the *required* bandwidth as the metric instead of the *actual* bandwidth utilization, since the latter has already been coupled with the interaction between the core pipeline behavior and external DRAM access patterns.

Interestingly, the two curves in Figure 5.12 show quite strong correlation between each other. Roughly speaking, the higher off-chip bandwidth is required, the larger the gap between the speedup of our elastic pipeline and the theoretic bound – in other words, the more difficult to reclaim the performance loss due to bank-conflict pipeline stalls. For the benchmarks examined in our experiments and the off-chip DRAM configuration in our GPU processor, we can see that some rough threshold, say, the 50% DRAM peak bandwidth bar, separates the benchmarks into *high bandwidth* group (DCT/IDCT) and *low bandwidth* group (the other kernels), as illustrated in Figure 5.12. For the low bandwidth group, the performance loss due to bank conflict stalls is relatively easy to be reclaimed by our proposed elastic pipeline design (indeed the theoretic bound is even surpassed in cases of Transpose and DCT/IDCT_short). While for the high bandwidth group, the elastic pipeline performance gain is far from ideal. The reason seems to be that, standalone core pipeline techniques (such as our elastic pipeline proposal) ignorant to the main memory access efficiency are unable to exploit the full potential of the hardware and the parallelism inherent in the software. This also explains the relatively large performance loss for DCT/IDCT kernels in Figure 5.10 with our elastic pipeline.

To summarize, it can be anticipated that, cooperative optimization schemes which take care of both core pipeline optimization and off-chip DRAM bandwidth efficiency are highly desirable, to further improve the overall performance for kernels with both heavy on-chip shared memory bank conflicts and off-chip bandwidth requirement.

5.7 Discussion

In this chapter we assume *barrel processing* [141], which lays the basis for contemporary GPU execution models [47]. In barrel processing, an instruction from a different hardware execution context is launched at each clock cycle in an *interleaved* manner. Consequently, there is no interlock or bypass associated with the barrel processing, thanks to the non-blocking feature of the execution model. Despite its advantages, *strict* interleaved multithreading has the drawback of requiring large on-chip execution contexts to hide latency, which can be improved in some ways. One such improvement is to allow multiple *independent* instructions to be issued into pipeline from the same execution context. In GPU cores, that is to allow multiple independent instructions from the same warp to be issued back-to-back (instead of the *strict* barrel execution model in which consecutively issued instructions are from different warps). Such execution is also adopted by some contemporary GPUs [147]. We call this extension “relaxed barrel execution model”. The intention of the *relaxed barrel processing* in GPUs is to exploit ILP inside the thread, in order to reduce the minimal number of independent hardware execution contexts (active warps) required to hide pipeline latency.

In the case of relaxed barrel execution, there can be two choices to make our proposed Elastic Pipeline still work. First, we can still allow elasticity in the pipeline backend, which means that the consecutively issued instructions from the same warp commit out of the program order. This flexibility comes at the cost of the pipeline being unable to support precise exception handling. This can be resolved by adding a re-order buffer (ROB), however at extra hardware cost. A second choice is to forbid out-of-order writeback for instructions from the same warp. In order to make elastic pipeline still effective in reducing pipeline stalls, it is the responsibility of the scheduling logic not to execute any more instruction from the same warp, if current shared memory instruction will cause any bank conflict. This can be easily integrated into our bank-conflict aware warp scheduling technique.

This chapter also assumes the execution stages and memory stages are not overlapped, therefore our proposed elastic pipeline design can make use of the existing "spare" MEM pipeline registers as the source of *elasticity* to tolerant the varied pipeline latency due to shared memory bank conflicts. However, this is not a mandatory requirement of our elastic pipeline proposal. For example, in the pipeline configuration with parallel execution/memory stages, we can insert some additional *spare pipeline stages* between the end of the parallel execution/memory stages and the beginning of writeback stages. With the extra stages as the source of elasticity our proposal can still work. Note the extra spare stages do not introduce any additional pipeline latency for ordinary execution without bank conflicts, thanks to the bypass buses (Figure 5.4(b)). The only overhead is the hardware cost of the pipeline stage registers of the additional pipeline stages.

Although only the effect of elastic pipeline for on-chip explicitly managed shared memory is evaluated in this chapter, we believe the first level hardware cache can also benefit from our proposal. The reason is that the heavily-banked hardware cache also suffers from the dynamically varied cache access delay due to unbalanced bank accesses, which is similar to the shared memory case. We leave the evaluation of elastic pipeline for L1 cache as future work.

For out-of-order processors, the *pipeline elasticity* realized by our elastic pipeline proposal in this chapter is actually enabled by the out-of-order engine. The OoO engine provides a small instruction window, which handles the variation of execution latency similar to a dataflow machine. The associated reorder buffer enforces the in-order instruction commitment. For architectures based on in-order pipelines, our elastic pipeline can be applied for a wide range of designs adopting barrel processing and SIMD data path, besides GPUs. The reason is that the on-chip bank conflict problem exists generally in such architectures. Furthermore, although we target the varied execution latencies caused by shared memory bank conflicts in this chapter, elastic pipeline can also be applied to cope with on-chip execution latency variation due to other shared resource conflicts (e.g., accelerator (such as FPU) access, interconnect buffers allocation, miss status holding registers (MSHRs) allocation, etc.). In such cases, pipeline elasticity can be exploited to tolerate the resource conflicts and maximize the SIMD datapath throughput.

5.8 Summary

In this chapter, we analyzed the shared memory bank conflict problem, and identified how the bank conflicts are translated into pipeline performance degradation. Based on the observation, we proposed a novel elastic pipeline design that minimizes the negative impact of on-chip memory conflicts on system throughput, by decoupling bank conflicts from pipeline stalls. Simulation results show that our elastic pipeline with the co-designed bank-conflict aware warp scheduling significantly reduces the pipeline stalls by up to 64.0% and improves overall performance by up to 20.7%, with trivial hardware overhead. Besides the performance advantage, our proposal also leads to reduced GPU programming complexity by relieving the burden of avoiding shared memory bank conflicts from the programmer.

Note. The contents of this chapter is based on the the following papers:

C. Gou, G. N. Gaydadjiev, **Addressing GPU On-chip Shared Memory Bank Conflicts Using Elastic Pipeline**, invited to *Special Issue of International Journal of Parallel Programming on SI: Computing Frontiers 2011 Best Papers* (in press)

C. Gou, G. N. Gaydadjiev, **Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts**, Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11), pp. 1–11, Ischia, Italy, May 2011. **Nominated for the Best Paper Award by the Program Committee**

6

Improving DRAM Access Efficiency

In this paper, we analyze a particular spatial locality case (called *horizontal locality*) inherent to GPUs employing barrel execution of SPMD kernels. We then propose an adaptive DRAM access granularity scheme to exploit and enforce the horizontal locality in order to reduce GPU cores memory interference and hence improve GPU DRAM efficiency. With the proposed technique, DRAM efficiency grows by 1.42X on average, leading to 12.3% overall performance gain, for a set of representative memory intensive GPGPU applications.

6.1 Introduction

Off-chip memory bandwidth is becoming a scarce resource in current and future manycore processors due to chip pin count limitations. Particularly, the bandwidth can be a severe bottleneck for *graphics processing units* (GPUs) due to their high ALU density design. Furthermore, DRAM access streams from different GPU cores can easily incur destructive interference among them. Therefore, efficient DRAM bandwidth utilization is crucial, especially for a growing number of data/memory-intensive applications. Memory access optimization techniques from the general purpose computing domain, such as prefetching and DRAM access scheduling, have demonstrated their effectiveness for GPUs [88,152]. However, as a platform originally designed for graphics processing, GPUs have their specific characteristics when used for general purpose workloads (aka *general-purpose processing on GPUs* (GPGPU) [1]). Hence, traditional solutions not aware of GPU programming and execution model specifics can lead to sub-optimal decisions and result in inefficient off-chip memory bandwidth utilization.

In this paper, we leverage a spatial locality typical for GPUs, called "horizontal locality", to improve external memory access efficiency. We propose a holistic DRAM bandwidth optimization framework for GPUs with combined compile-time, run-time, and architectural efforts. Our technique takes advantage of per memory instruction access pattern information generated by a compiler and runtime analyzer, and schedules the optimal access granularity accordingly at kernel launch time. With the co-designed hardware support, adaptive memory access granularity is achieved, DRAM efficiency is increased and the overall performance improved.

This chapter makes the following specific contributions:

- novel method to improve GPU DRAM efficiency, using combined compile-time and runtime efforts along with specific microarchitectural extension;
- adaptive, *inter-thread/warp locality* aware DRAM access granularity scheduling technique to improve DRAM efficiency;
- trivial hardware overhead *elastic vector MSHR* design with *deferred reservation*;
- DRAM efficiency improvement by 1.42X on average (corresponding to 12.3% overall performance boost) for a set of representative workloads.

The remainder of this chapter is organized as follows. In Section 6.2, we provide the background and motivation for this work. In Section 6.3, the horizontal locality is analyzed, and the memory access pattern analyzer is briefly introduced, followed by the elaboration of our proposed adaptive memory access granularity scheduling. The microarchitectural extension to support adaptive memory access granularity is discussed in Section 6.4. Simulation results for a set of memory-intensive CUDA benchmarks are presented in Section 6.5. Finally, Section 6.7 summarizes the chapter.

6.2 Background and Motivation

GPU is a manycore platform employing large number of lightweight cores to achieve high aggregated performance, originally for graphics workloads. Nowadays, its utilization has spanned far beyond graphics rendering, covering a wide spectrum of general-purpose applications (referred to as *GPGPU* [1]). GPUs often adopt *bulk synchronous programming* (BSP) [145] programming

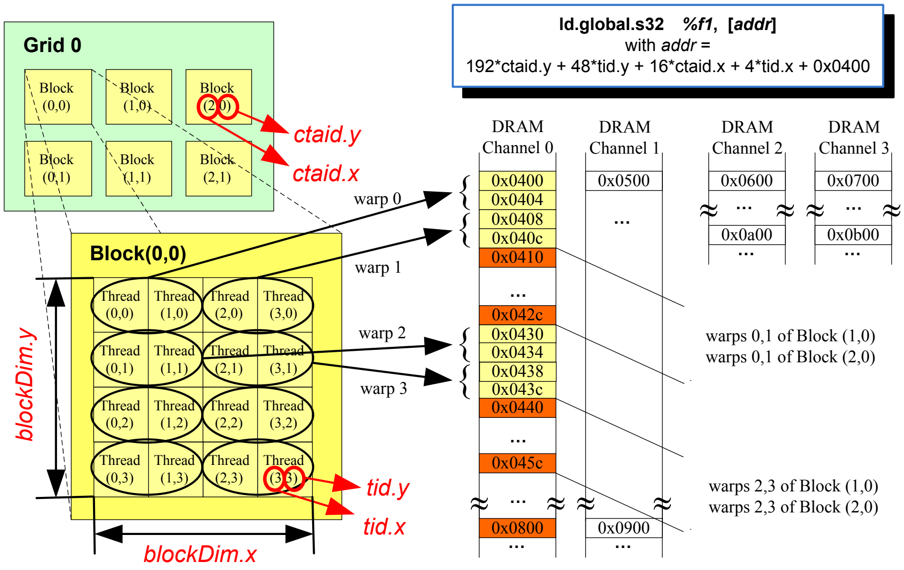


Figure 6.1: Hierarchy and memory accesses of worker threads

models. The execution of BSP programs on GPUs often employs *barrel processing* [141] due to its low pipeline implementation overhead.

Programming Model Properties: The BSP model has been widely adopted in programming languages targeting manycore accelerator architectures, e.g., CUDA [105] and OpenCL [8]. In such languages, the parallelism of the application's computation intensive kernels is explicitly expressed in a *single program multiple data* (SPMD) manner. In such *explicitly-parallel, bulk-synchronous* SPMD programming models, the programmer extracts the data-parallel section of the application code, identifies the basic working unit (typically an element in the problem domain), and explicitly expresses the same sequence of operations on each working unit in a *kernel*. Multiple kernel instances (called *threads* in CUDA) are running independently on the GPU cores.

In CUDA, the parallel threads are organized into a two-level hierarchy, in which a kernel (also called *grid*) consists of parallel CTAs (*cooperating thread array*, aka *block*), with each CTA composed by parallel threads, as shown in Figure 6.1[1]. Explicit, localized synchronization and on-chip data sharing mechanisms (such as CUDA shared memory) are supported inside each CTA.

---

[1] This is an example based on CUDA, with the warp size reduced from 32 to 2 to simplify the illustration.

Baseline Manycore Barrel Processing Organization: Figure 6.2 shows our baseline organization. On the right the high-level system organization is shown. The system consists of an GPU node with K cores and a memory subsystem with L DRAM channels, connected by the on-chip interconnect. Depending on the implementation, the GPU node itself may form a chip (similar to discrete GPUs), or the GPU node(s) and host CPU(s) can be integrated on the same die (e.g., [12]). The host processor offloads computation intensive kernels to the GPU node during execution. The kernel code and parameters are transferred from host processor using the host interface, and the workloads are dispatched at the CTA/block granularity. Concurrent CTAs are executed on GPU cores independent of each other.

The left part of Figure 6.2 illustrates a single GPU core. During execution, a batch of threads from the same CTA are grouped into a *warp*, the smallest unit for the pipeline front-end processing. Each core maintains a set of on-chip hardware execution contexts and switches at the warp granularity. The context switching, also called *warp scheduling*, is done in an interleaved manner, also known as *barrel processing* [141]. Warps are executed by the core pipelines in a SIMD fashion for improved pipeline front-end processing efficiency. In practical designs, the threads number in a warp can be multiple of the SIMD width. In this case, a warp is composed of multiple slices (*subwarps* here), with each subwarp size equaling the SIMD width. Subwarps from the same warp are processed by the SIMD pipeline back-to-back.

Warps can access two types of memory: on-chip shared memory and off-chip memory. When there is an off-chip memory access (in systems without hardware data caches, such as the one evaluated in Section 6.5), or a cache miss (in case on-chip L1 data cache is adopted), the execution is taken care of by a *miss status holding register* (MSHR) unit as shown in Figure 6.2. The memory access information is logged by the allocated MSHR entry, and warp execution is put into inactive status. After being fired into the on-chip interconnect, off-chip memory access streams from GPU cores start competing shared resources against each other, often resulting in nontrivial interference at DRAM side and inefficient DRAM bandwidth utilization.

DRAM Bandwidth Utilization Inefficiency: The interference among GPU cores has various forms: *Hot DRAM Channels:* when multiple GPU cores access only one or a few DRAM channels, leaving the others idle. This directly degrades memory system performance since the bandwidth of “cold”(idle) channels is wasted. *DRAM bank conflicts:* when memory accesses from multiple cores compete reading or writing to different rows of the same DRAM

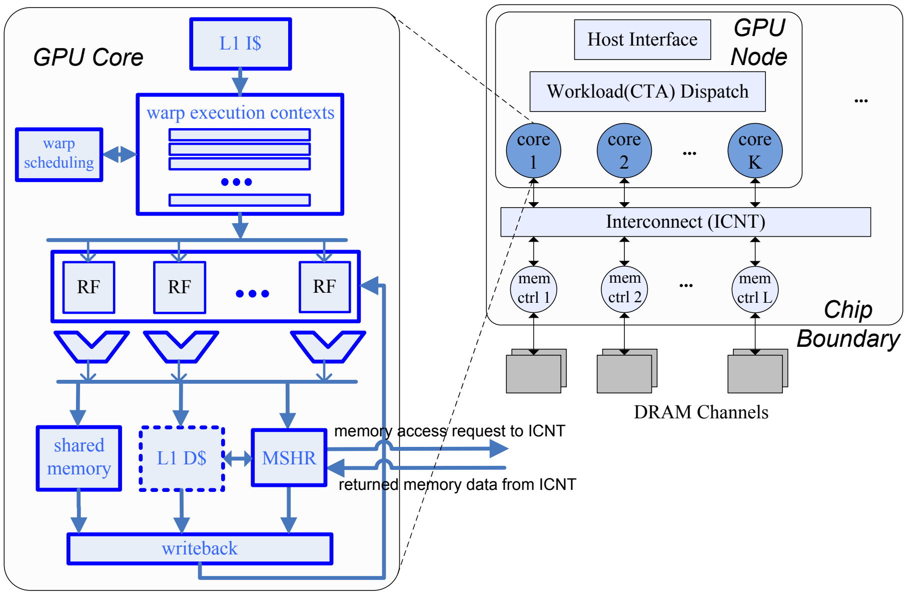


Figure 6.2: Baseline GPU organization with barrel processing

bank. This causes frequent opening and closing row operations, which also degrades DRAM performance due to their high penalty. *Bus read/write transition cost:* Shifting between read and write in the same DRAM channel introduces not only additional latency but also bandwidth losses. This is due to the data bus turn-around time which is necessary for the shared input/output data bus design adopted in most contemporary DRAM chips. All of the above penalties are often non-negligible, especially for data intensive applications.

Motivation: As a platform originally designed for graphics processing, GPUs have their specific characteristics when used for accelerating general-purpose workloads. Therefore, in order to better address the off-chip memory bandwidth inefficiencies, the unique characteristics of both the GPU programming and execution models should be exploited. In this chapter, we leverage a spatial locality typical for SPMD barrel processing, called *horizontal locality*, to improve external memory access efficiency. Specifically, we propose a holistic DRAM bandwidth optimization framework for GPUs with combined compile-time, run-time, and architectural efforts. Our technique utilizes memory instruction access pattern information provided by a static access pattern analyzer, and runtime scheduling for optimal access granularity based on the available horizontal locality determined by the access pattern. With the sup-
]

port of co-designed hardware, adaptive memory access granularity is achieved and the DRAM efficiency and the overall performance are improved.

6.3 Horizontal Locality Aware DRAM Scheduling

In this section, we will first analyze horizontal locality, a spatial inter-thread/warp locality typical for GPU execution. Then we use a compiler and runtime analyzer to extract the kernel memory access patterns. With the precise address pattern information, a scheduler is created to identify the available horizontal locality, and find out the most profitable external memory access granularity at kernel launch time.

6.3.1 Horizontal Locality

Within GPU barrel execution of SPMD kernels, the memory access behavior is determined not only by a *single* thread/warp, but also by concurrent warps execution. Figure 6.1 shows a typical 2D address pattern in a CUDA kernel. One important observation is that, the access pattern not only guides each worker thread to its working field (data address in memory), but also binds the relationship among threads' memory accesses. For example, memory addresses of warps 0 and 1 are always contiguous, for the given access pattern. Therefore, spatial locality can be exploited. Different from the spatial locality in general-purpose processors, this new locality type has two distinct characteristics: 1) it is an *inter-thread/warp locality* among multiple independent threads/warps; 2) if captured, it only benefits the neighbor threads/warps but not the memory access initiator. In this chapter, we call such inter-thread/warp spatial locality "horizontal locality", whereas the spatial locality inside a thread will be referred to as "vertical locality". We will see in Section 6.5 that both locality types are important in optimizing memory access for GPU throughput.

Above we have assumed that all warps are executing the same memory instruction using the same access pattern. In fact, this originates from the combination of the SPMD programming model and the barrel execution model adopted in contemporary GPUs. Within *strict barrel execution*, an instruction from each warp execution context is launched at each clock cycle in an *interleaved* manner. Moreover, all warps are executing the same SPMD kernel code. In this way, the execution of concurrent warps in a core is *highly correlated* in that the in-flight instructions are similar in many cases. Even with *relaxed barrel execution*, where consecutive instructions from the same warp are allowed to issue into the SIMD pipeline, since the average size of *independent instruction blocks*[2] is small (1$\sim$3 instructions in our benchmarks), the horizontal locality can still be captured in time[3] before the requested data arrives.

A real example: Figures 6.3 and 6.4 shows the PC-Histogram of a GPU core running matrix multiplication kernel (MM in Section 6.5), drawn using a modified version of AerialVison [13]. The one on top shows the PC-Histogram of all warps execution, while the bottom one illustrates that of warps from a *single* CTA. The x-axis of Figures 6.3 and 6.4is the core cycle time, while the y-axis is the kernel *parallel thread execution* (PTX) source line number. The PC histogram represents the portion of the program touched by the concurrent threads (i.e., PTX instructions executed or on which the threads are pending) during each sampling interval[4]. The color intensity of Figures 6.3 and 6.4 indicates the thread count. During any interval, the aggregate thread count remains constant (1024 in Figure 6.3 and 256 in Figure 6.4). The PC-Histograms at different intervals differ only in the invoked PTX instructions and their count.

Several observations can be made from the example: 1) unlike a traditional vector processor, concurrent warps are executing **not** in *lock-step* (Figure 6.3), even with the Round-robin warp scheduler (Table 6.2); 2) even though warps execution is *interwined* across CTAs, warps inside a single CTA are highly correlated (Figure 6.4); 3) external memory accesses (e.g., global loads at PTX source lines 118 and 121) effectively *re-align* the CTA warps execution, similar to barrier synchronizations (e.g., PTX source line 124).

The above observations confirm the analysis at the beginning of this section, in that the GPU SPMD barrel processing indeed correlates the CTA warp execution. Besides, the other major reasons for the correlated execution include: bulk synchronizations common in BSP programs, and the rather long external memory latency (e.g., > 1000 cycles in above example) incurred by insufficient *transient* external memory bandwidth during burst memory accesses.

It should be pointed out that, the MM example shown in Figures 6.3 and 6.4 is a typical regular application, with intrinsic locality. However, even with less regular memory access patterns, memory accesses from *close neighbor warps* often exhibit spatial locality, which can be also captured (e.g., FWT in Section 6.5). Admittedly, for applications with highly irregular control flow (such

---

[2] A sub-basic block with mutually independent instructions.

[3] It takes 384 cycles for all 32 warps to issue an independent instruction block of 3 instructions, a time much shorter than average main memory access delay for memory intensive kernels.

[4] 128 cycles sampling interval is chosen, during which all 1024 threads can execute exactly one instruction in the 8-way SIMD pipeline (Table 6.2).

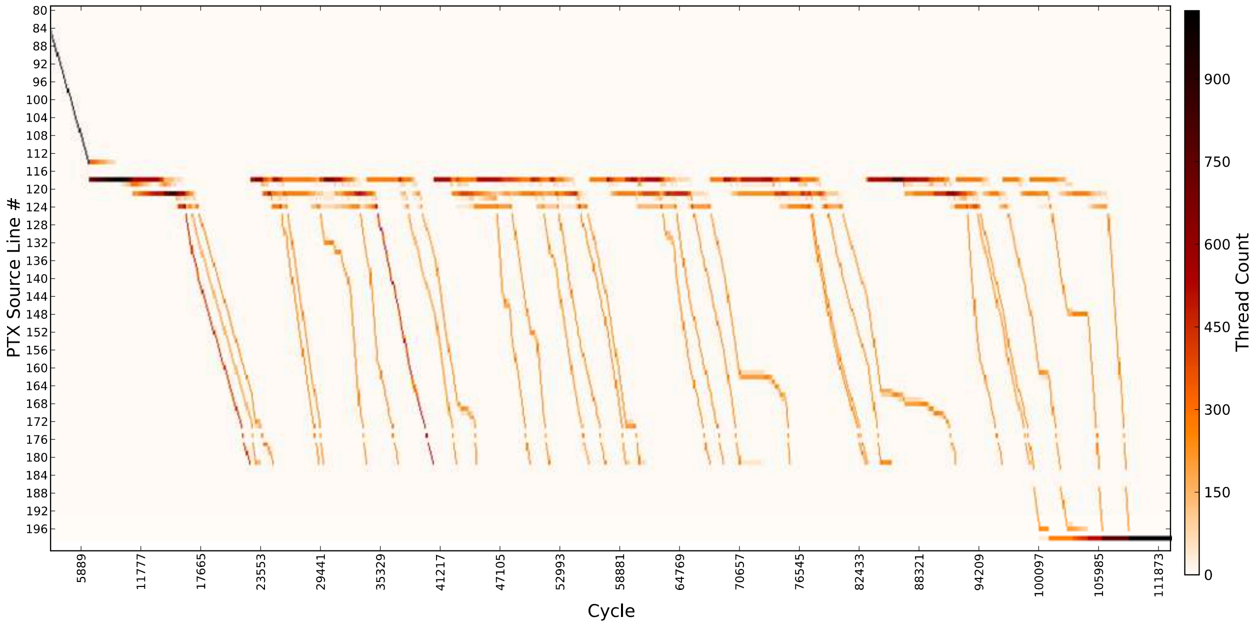


Figure 6.3: Matrix multiplication PC-Histogram: all 4 CTAs

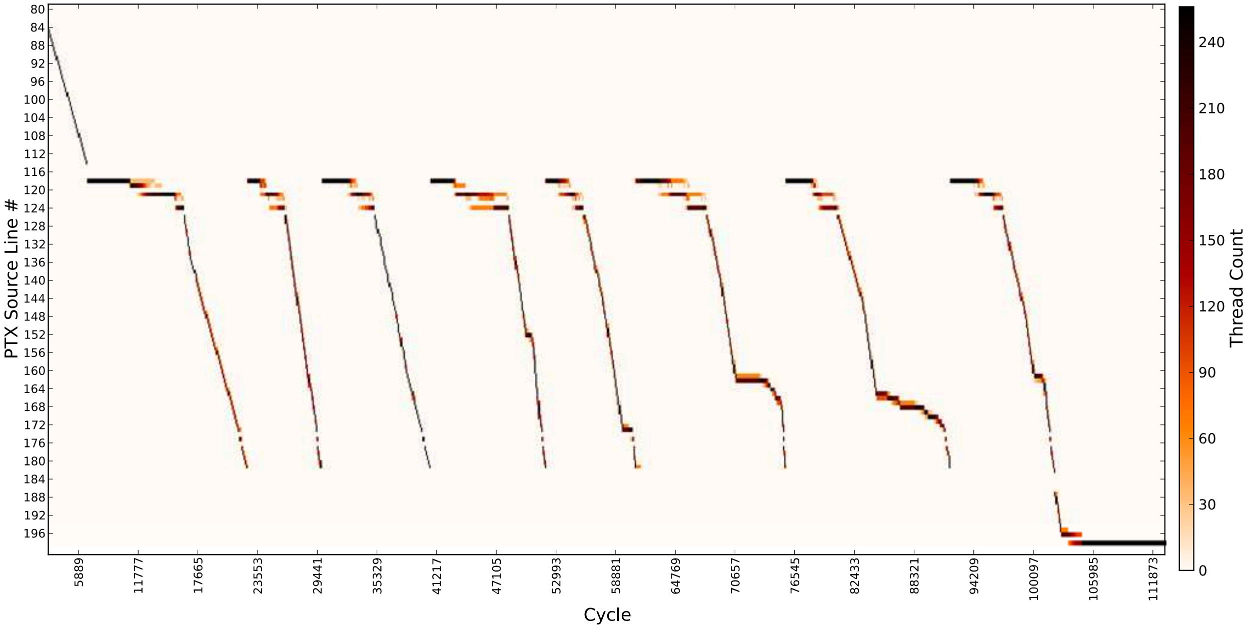


Figure 6.4: Matrix multiplication PC-Histogram: 1 CTA

as BFS and LPS), it is difficult to precisely identify the horizontal locality, since the available locality changes dynamically during warp execution.

As stated in Section 6.2, concurrent memory accesses of different cores can interfere in GPUs. For example, even if accesses from warps 0 and 1 of block (0,0) in Figure 6.1 are issued back-to-back, they can be *separated* along the way to DRAM, e.g., by on-chip interconnect or memory controllers. As a consequence, their horizontal locality is broken, resulting in extra DRAM bank conflicts and memory bus transition penalties. In the following, we propose a novel way to *exploit* and *enforce* the horizontal locality in GPUs and improve external memory bandwidth efficiency.

6.3.2 Compiler and Runtime Access Pattern Analyzer

In an explicitly-parallel, bulk-synchronous SPMD program, in order for the worker thread to identify its working set, a mapping between the thread id and the working set is designated in the code: $addr = \Phi(tid.z, ctaid.y, tid.y, ctaid.x, tid.x)$, as illustrated in Figure 6.1. Ideally Φ can be of any arbitrary form, however, the number of patterns used in programmers practice is rather small, and the complexity of address generation is often limited. Leveraging this observation, we have prototyped a framework able to detect and exploit the most common memory access patterns for CUDA kernels. Our framework employs static control- and dataflow analysis at compile time to detect the access *skeleton* type, and build the corresponding parameters expressions. A skeleton is defined as a *parameterized* address mapping function, which is able to generate *a class of* memory access patterns. Our runtime library will evaluate the parameters expressions provided by compiler analysis, based on the CUDA kernel dimensions and input parameters available at launch time. We implemented our prototype as additional NVCC compiler pass and library extensions to CUDA runtime environment. Currently, our framework supports two most common skeletons –

① *1D/2D contiguous/strided/block-strided access:*

$$\begin{aligned}
&\eta(a, b, c, d, e, f) \\
=&(a, b, c, d, e, f) \cdot (tid.z, ctaid.y, tid.y, ctaid.x, tid.x, 1) \\
=&a \cdot tid.z + b \cdot ctadid.y + c \cdot tid.y + d \cdot ctaid.x + e \cdot tid.x + f
\end{aligned}$$

with parameters $a, b, c, d, e, f \in \mathbb{N}$ (used in regular memory access patterns);

and ② *skewed access:*

$$y = \eta(a, b, c, d, e, f)$$
$$\varphi(h1, l1, s1, h0, l0, s0, \alpha, \beta, y) = (y[h1 : l1] \ll s1 \mid y[h0 : l0] \ll s0) \cdot \alpha + \beta$$

where, $h1, l1, s1, h0, l0, s0, \alpha, \beta \in \mathbb{N}$ (used in irregular applications). Skewed access skeleton is constructed using 1D/2D access (y), and contains two bits sections: the higher section, taken from bits $h1$ to $l1$ of y and shifted left by $s1$ bits; the lower section, generated similarly. In the final form, the two sections and y are *normalized* so that $l0 = s0 = 0$.

The compile-time access pattern analysis employs static control- and dataflow analysis to trace back the use-def chain of the memory instruction. In order to do so, a set of *transition operations*, such as {+, -, ·, /, ≪(shift left), ≫(shift right), &(logic and), |(logic or), ∧(logic not)}, are defined for the above skeletons. Ideally, transition operations should be completely *composable* without limitations. However, this requires more complex address skeletons, becoming more costly in terms of analysis time. In practice, trade-offs are made in defining the operations composability, in order to reduce the analysis complexity while at the cost of less analysis accuracy. Admittedly, this compromises the compiler capability in handling various memory address generation types. During analysis, if an unsupported transition operation or operation composition occurs, the process simply stops and returns an analysis fail.

Since some kernel parameters may be available only at runtime, thus the compiler analysis outputs skeleton id (① or ②) and builds the internal representations for each skeleton parameter. This process can be formalized as follows: given

$$\begin{cases} \Theta & \leftarrow \{kernel\ input\ params\} \\ nctaid & \leftarrow \{CUDA\ grid\ dimension\} \\ ntid & \leftarrow \{CUDA\ block\ dimension\} \end{cases}$$

for $I \in \{kernel\ memory\ insn\}$, the compiler analysis generates

$$\begin{cases} skeleton\_id_I \\ \tau = \zeta_{I,\tau}(\Theta, nctaid, ntid) \\ \qquad \tau \in \begin{cases} \{a, b, c, d, e, f\}, & skeleton\_id_I = ① \\ \{a, b, \ldots, f, h1, l1 \ldots, \alpha, \beta\}, & skeleton\_id_I = ② \end{cases} \end{cases}$$

Accordingly, the interface between the compiler analysis and runtime is $skeleton\_id_I$ and $\zeta_{I,\tau}$. Skeleton type is known after static analysis; while $\zeta_{I,\tau}$

is composed of a *tree representation*, with each leave node pointing to one of the Θ, $nctaid$, and $ntid$ parameters, and other nodes being arithmetic operations. Normally, address generation for GPU kernel memory instructions is not complex, therefore the tree representation usually contains only few nodes. At kernel launch time, since all parameters Θ, $nctaid$, and $ntid$ are known, thus the precise access pattern is extracted by simply evaluating tree representations for all $\zeta_{I,\tau}$.

Note, at kernel scope, the memory address use-def chain may vary along different control flow paths. In this case, *multiple access pattern versions* may exist for the same instruction I, with each corresponding to a different control flow path. In a general sense, such multiple versions per memory instruction case can be viewed as another form of access pattern, although it can not be efficiently expressed in *closed form*. Also it creates major challenges in GPU execution. We will see this in Section 6.5.

6.3.3 Adaptive DRAM Access Granularity Scheduling

Alg. 6.1 Adaptive memory access granularity scheduling

```
$max\_gran \leftarrow$ DRAM channel interleaving granularity
$min\_gran \leftarrow$ DRAM minimal access granularity
$data\_size \leftarrow$ data size of memory access instruction
$blockDim.x \leftarrow$ block size in dimension x
$d, e, is\_skewed\_access, l1, s1, h0 \leftarrow$ access pattern parameters

if $e! = data\_size$ or $blockDim.x \times e! = d$ or ($is\_skewed\_access$ and ($l1 \leq h0$ or $s1 \leq h0$ or $2^{h0} \leq max\_gran$)) then
    return $min\_gran$
else if $blockDim.x \times e > max\_gran$ then
    return $max\_gran$
else if $blockDim.x \times e < min\_gran$ then
    return $min\_gran$
else
    return $blockDim.x \times e$
end if
```

To reduce bank conflicts and bus turn-around times, *adaptive DRAM access granularity scheduling* (Alg. 6.1) is created to determine the memory access granularity, based on access pattern information generated by the memory pattern analyzer described in Section 6.3.2. The essential idea of the adaptive granularity scheduling is as follows. Since the address pattern determines the locality among neighbor warps' memory accesses, we can choose the maximal access granularity *allowed* by the horizontal locality, for a given memory instruction. When a warp memory instruction execution misses, it initiates the external memory access with the *large granularity*. Due to the horizontal locality, the extra requested data will often be used by neighbor warps, hence, boosting memory efficiency, *without the penalty of wasted memory bandwidth*.

The key in the adaptive memory access granularity is to correctly identify the available horizontal locality. Line 6 of Alg. 6.1 examines whether or not a memory instruction has the appropriate horizontal locality. Currently, only instructions accessing contiguous memory addresses among neighbor threads in a block's x-axis (Figure 6.1) are considered – a rather conservative decision made in this work. Lines 8 to 13 determine the proper access granularity under three constraints: 1) the minimal access granularity, set by the DRAM interface (32B for GDDR3 in our study); 2) the maximal granularity, set by DRAM channel interleaving granularity; and 3) the *block boundary* constraint ($e \times blockDim.x$), due to the fact that a warp should not fetch data outside the block boundary. E.g., in Figure 6.1, warp 0 of block(0,0) should not fetch data for warp 0 of block(1,0) even if they are neighbors in the grid map, since the scheduler at kernel launch time does not know whether or not the two independent blocks will be concurrently executed on the same core.

Example: For the global memory load instruction shown in Figure 6.1, our compiler pass identified *2D access* skeleton, with parameters $(a, b, c, d, e, f) = (0, 192, 48, 16, 4, 0)$ calculated by the runtime by evaluating the corresponding tree representations. At kernel launch time, first the memory access contiguity among warps inside a block is determined. This is done by checking if the accesses from neighbor warps fail to cover the contiguous address space ($e \neq data\_size$ or $blockDim.x \times e \neq d$, false in this case). At this point, the access pattern is assured to have *strong* horizontal locality where contiguous $blockDim.x \times e$ (16 in this case) bytes data will be *fully* utilized by the load instruction warps execution in the block. Therefore, an MSHR entry is allocated, and 16 bytes external memory request is fired into the interconnect when warp 0 is run. In case of no hardware data cache, warp 0 has also to book MSHR space for the fired load request, *however only for its own requested data* (8B data segment X). Later on when warp 1 is invoked for the load execution, it simply checks the MSHR tag array to find out an ongoing load which *covers* its request (8B data segment Y). At this point, it allocates an MSHR entry and reserve the data space for Y. Afterwards, it is put into inactive status, waiting for data Y to return from DRAM. The hardware mechanism supporting this process is called *elastic MSHR with deferred reservation*, which will be addressed in the following section.

6.4 Microarchitectural Extension

In this section, we present the necessary microarchitectural extension to support adaptive external memory access granularity. We will first describe the vector MSHR unit which the extension is built upon, and then the elastic MSHR hardware design.

It should be mentioned that, the access granularity values, determined by the runtime scheduling, can be easily transferred from the host CPU to the GPU through host interface (Figure 6.2), and stored in a specific, on-chip *access granularity buffer* shared by all GPU cores. This can be achieved easily with trivial architectural extension, thus the details will be omitted in this Section.

6.4.1 Vector MSHR (VMSHR)

Before going into details, first we shall introduce some definitions for the ease of discussion: **Vector memory access:** a group of memory accesses requested by the scalar threads from within the same *subwarp*(Section 6.2) execution. **Primary load and primary MSHR (PMSHR):** A vector load is called "*primary load*" if there is no in-flight memory read access targeting/covering the same address range as this memory load. When a primary load is processed by the MSHR unit, an MSHR entry, called "*primary MSHR*", has to be allocated to keep track of the load access address. Note, for each primary load and PMSHR allocation, an off-chip memory access has to be fired. **Secondary load and secondary MSHR (SMSHR):** A vector load is called "*secondary load*" if there is at least one in-flight memory load access targeting/covering the same address range as this memory load. When a secondary load is processed by the MSHR unit, an MSHR entry associated with the corresponding PMSHR of the overlapping in-flight load, called "*secondary MSHR*", has to be allocated. Note, a SMSHR only has to record the target register address. Unlike primary load, a secondary load does **not** need to fire any off-chip memory access, since it is covered by its primary load already in flight. **Vector MSHR (VMSHR):** A MSHR unit design which is capable of processing vector memory access. The major difference between a *VMSHR* and a *scalar MSHRs* is that the former is required to allocate vector memory accesses, and commit the ready data to vector registers, both in parallel. Depending on the detailed implementation, the main challenge of a VMSHR design can be a pure bandwidth problem, or it may be a more complex issue requiring sophisticated data rearrangement when committing data to vector registers. Since our proposal does not change the MSHR data writeback and deallocation behavior, we will

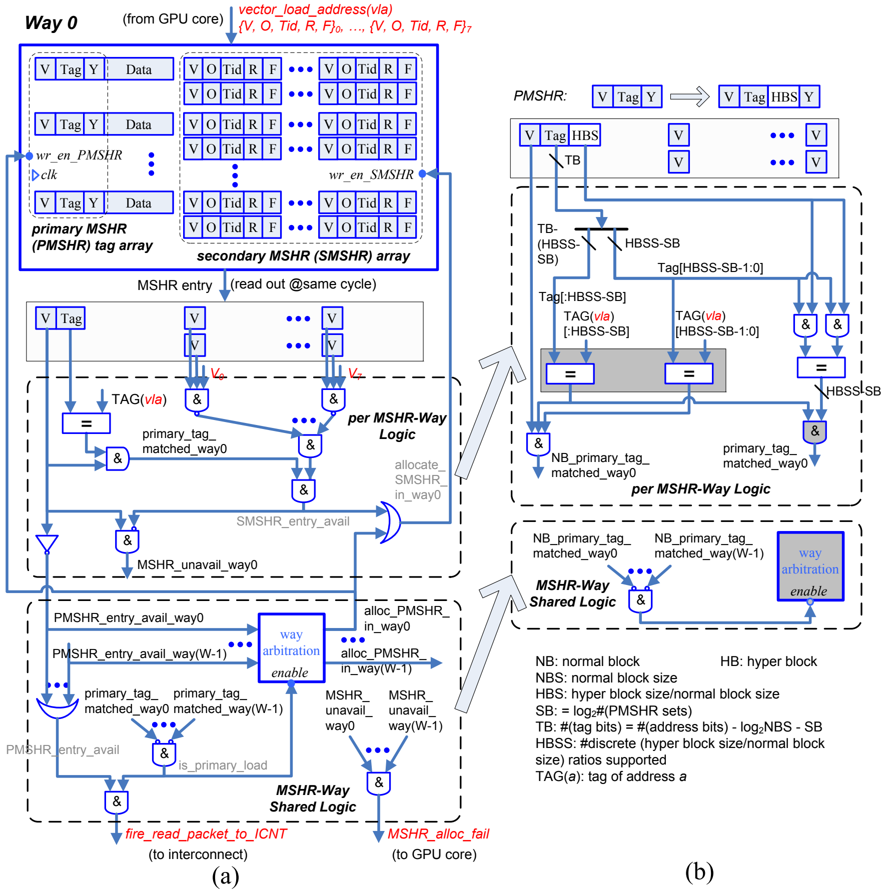


Figure 6.5: VMSHR design: (a)baseline; (b)modifications for elastic MSHR

focus on the MSHR storage scheme and allocation logic in this chapter. **Normal block:** the memory access granularity of a normal load and store (32B in our configuration). **Hyper block:** If the load granularity determined by the runtime scheduler (Section 6.3.3) is larger than the *normal block*, then it is called a *hyper block*.

We modeled a source addressed MSHR design [67] for GPU core, with the storage scheme similar to [18], as shown on top of Figure 6.5(a). Each PMSHR entry includes a valid bit (V) and an address tag. Since our modeled GPU core does not include a on-chip hardware data cache, a storage (the *Data* field) has to be provided for each primary load, together with a data ready bit (Y), Each SMSHR is composed of a valid bit (V), an offset (O) indicating the offset inside the *Data* field, a thread id (Tid) recording the hardware thread id which requests the memory access, a target register address (R), and a data format (F) field. Since MSHRs can be implemented with non fully-associative structures, we show a set associative MSHR design on top of Figure 6.5(a), for the purpose of discussion generality.

In our design, we associate 2 SMSHRs to each PMSHR, in order to avoid performance losses due to the pipeline being blocked by a secondary load [67]. This also makes it possible for the MSHR unit to capture certain temporal locality, using the second SMSHR entry[5].

MSHR allocation: For a vector MSHR allocation, the vector load address[6] together with the corresponding access information from each lane[7] are sent to the MSHR unit. Then the load address is searched among all PMSHR tag arrays, to identify the vector access type and check PMSHR/SMSHR entry availability accordingly. In the end, two output signals are generated. *MSHR_alloc_fail=1* informs the core pipeline that a valid MSHR entry is not found for the vector memory access, which will block the pipeline. Otherwise, the allocation is successful, and in consequence the MSHR unit will take charge for the vector memory access and the core pipeline will be released. The other signal, *fire_read_packet_to_ICNT=1* indicates: 1) current load is a primary load; 2) PMSHR allocation is successful; 3) an interconnect buffer is needed. In this case, the MSHR unit will wait for an interconnect buffer, and

---

[5] Note, a SMSHR entry is also allocated when a PMSHR is allocated for a primary load, to keep track of the target register addresses [67].

[6] Vector memory stores are simply packed by the MSHR unit and directly injected into the interconnect.

[7] If lanes requests are targeting different memory blocks, then the vector has to be split into sub-vectors, with each accessing a block. Then the sub-vector requests are sent to the MSHR unit one by one.

then a DRAM read request is fired into the interconnect if a buffer is available. At this point, the first stage of this load instruction execution, sending out the request, is finished.

Note, we are **not** proposing new vector MSHR designs here. Instead, we just model one the possible design, with the intention to show how our proposed adaptive DRAM access granularity can be actually implemented in hardware and integrated into the baseline vector MSHR design.

6.4.2 Elastic MSHR with Deferred Reservation

In order to support adaptive access granularity for horizontal locality aware DRAM access granularity scheduling, the vector MSHR needs to be extended, which we call "*Elastic MSHR*" in this chapter. The idea of elastic MSHR is straightforward. Since the primary MSHR already maintains the access address, we can also extend it with an extra field to keep track of access granularity. However, the problem occurs when we allocate for a primary load. Since we also need to reserve the storage for outstanding primary loads, we are facing the problem of proper data storage size to be reserved, which is changing dynamically. A naive solution will be a MSHR design which associates a storage to each PMSHR entry with a large size able to satisfy all dynamic size requirements. Unfortunately, this results not only in chip area waste, but also complicates the already sophisticated data rearrangement problem during vector MSHR writeback.

Our observation is, since the primary vector load which initiates the hyper block access will only consume its own required data (i.e., a normal block), it is **not** necessary for the primary load to reserve the *extra storage space*, for a hyper block memory access. Based on this observation, we propose a simple yet effective MSHR data storage reservation policy for GPU cores without hardware L1 data cache, which we call "*deferred reservation*". In the deferred reservation, each PMSHR is associated with a storage of a normal block size. When a PMSHR is allocated, only this storage is reserved, irrespective of the hyper block size. By the time a hyper block returns from DRAM, if the corresponding storage space for some hyper block part has not yet been reserved (indicating that the part has never been requested by any secondary load), then that part is simply discarded. In this case, system performance may decrease due to the additional, unused off-chip trafic. Fortunately, we found such occasions rare in our experiments (Section 6.5.5).

Microarchitectural Modifications

With the deferred reservation scheme, the aforementioned elastic MSHR data storage size problem is resolved, with only slight modifications to the baseline vector MSHR design. As shown on top of Figure 6.5(b), a *hyper block size* (HBS) field is added to the PMSHR tag, defined as the ratio between the requested memory load (being either normal block or hyper block) granularity and the normal block size. The two dashed squares in Figure 6.5(b) show the modifications to the original vector MSHR allocation logic in order to support elastic MSHR with deferred reservation, assuming $\log_2(\#PMSHR\_sets) < \#different\ \frac{hyper\_block\_size}{normal\_block\_size}$.[8]

It is important to note that, with non fully-associative MSHR implementations (i.e., $\#PMSHR\_sets > 1$), it may occur that a secondary load is falsely identified and processed as a primary load by the MSHR unit, when the real primary load (which covers that secondary load) is allocated in some MSHR set different from the one the secondary load is mapped to[9]. In this case, an unnecessary off-chip memory request is fired into the interconnect, resulting in system performance degradation. Essentially, the trade-off here is between the MSHR hardware complexity and off-chip bandwidth utilization. Since off-chip memory bandwidth is a serious problem in manycore GPUs, we prioritize memory performance over MSHR hardware complexity, and thus assume fully-associative MSHR for designs with practical MSHR sizes[10]. In this case the above problem does not exist.

Hardware Cost and Timing Overhead

The shaded components in Figure 6.5(b) already exist in the baseline vector MSHR in Figure 6.5(b). Note, the single comparator in Figure 6.5(a) is split into two (in shadow) in Figure 6.5(b). Assuming fully-associative MSHR design (which incurs the highest hardware cost), and the optional hyper block sizes 64, 128 and 256B (Section 6.5), the hardware cost is summarized in Table 6.1. M in the table denotes the PMSHR entry number. As we can see in Table 6.1, the extra hardware cost of our *elastic MSHR with deferred reservation* is trivial, for practical MSHR designs. By examining Figure 6.5, it is easy

---

[8] In our evaluation it is $0 < 3$ with fully-associative MSHR and 3 hyper block sizes. When $\log_2(\#PMSHR\_sets) >= \#different\ \frac{hyper\_block\_size}{normal\_block\_size}$, no modification is needed.

[9] This is a scenario similar to the address disambiguation issue in configurable caches.

[10] In our evaluation, 64 MSHR (PMSHR) entries per GPU core is adopted (Table 6.2), similar to [16].

Table 6.1: Hardware cost of elastic MSHR with deferred reservation (per GPU core)

Type	Quantity	Type	Quantity
2-input AND Gate	x 6M	3-bit Comparator	x M
3-input AND Gate	x M	M-input NOR Gate	x 1

to verify that the critical path is the one that passes through the *way arbitration unit*, which is **not** affected by the additional logic of our proposal.

6.5 Experimental Evaluation

Experimental Setup: We use a modified version of GPGPU-Sim [17], a cycle-level full system simulator implementing PTX virtual ISA [110] of NVIDIA CUDA programming model and target architectures. The detailed configuration of the modeled GPU is shown in Table 6.2. The prototyped access pattern analyzer and the adaptive access granularity scheduling run on the host CPU. When a kernel is to be launched, the scheduled optimal access granularity values for memory instructions are transferred to the accelerator. At the accelerator node, the baseline and our proposal execution differ mainly in memory access granularity: the baseline memory accesses are 32B[11], while our proposal memory access granularity is *determined* by the launch time scheduler (Section 6.3.3) implemented in our runtime library, and *realized* by the elastic MSHR design (Section 6.4.2) implemented in our microarchitecture. We evaluated only adaptive memory **load** granularity.

We used 17 memory intensive benchmarks from CUDA SDK [109], Rodinia benchmark suit [26], and [17], which are widely used in GPU architecture research [16, 89]. The selected benchmarks are listed in Table 6.3. The selection criteria is that the memory bandwidth requirement (Section 6.5.1) is larger than 1 Byte per core per cycle. Similar criteria has been used in prior work [16]. Note, we only considered CUDA *global* and *local* memory accesses in evaluating the bandwidth requirement, since this work focuses on generic GPU memory systems. Memory accesses generated by graphics-specific subsystems, such as texture and constant memories, are **not** counted. This has excluded memory intensive kernels such as *kmeans* [26] and *MUMerGPU* [26] from our benchmarks.

---

[11] Appropriate for the simulated 8-way SIMD pipeline.

Table 6.2: Baseline GPU configuration[†]

Number of Cores	16 @1296.0 Mhz
Core Configuration	8-wide SIMD execution pipeline, 24 pipeline stages 32 threads/warp, 1024 threads/core, 8 CTAs/core, 16384 registers/core warp scheduling policy: Round-robin, execution model: strict barrel processing (Section 6.3.1)
On-chip Memories	16KB software managed cache (i.e., shared memory)/core, 8 banks, 1 access per core cycle per bank 64 MSHRs/core, with 32B data field per MSHR entry (no hardware cache)
DRAM	4 GDDR3 memory channels, 2 DRAM chips per channel, 2KB page per DRAM chip, 8 banks per DRAM chip 8 Bytes/channel/transmission, 68.2 GB/s aggregate bandwidth @1066 Mhz bus freq, 256B channel interleaving GDDR3 memory timing: t_{CL}=12, t_{RP}=12, t_{RC}=41, t_{RAS}=29, t_{RCD}=14, t_{RRD}=10 memory controller policy: out-of-order (FR-FCFS) [125], 32 DRAM request buffer entries per controller
Interconnect Network	crossbar (2-ary 5-fly butterfly [37]) @602.0 Mhz, 16-Byte flit size, 2 Virtual Channels, 8 buffers per Virtual Channel

[†] a *downsized* NVIDIA GeForce GTX 280 version, with 16 vs 30 processor cores and half of its aggregated DRAM bandwidth (68.2 instead of 141.7 GB/s)

Table 6.3: Benchmarks

Name	**Abbr.**	**Name**	**Abbr.**	**Name**	**Abbr.**
Transpoes [109]	TRA	Speckle Reducing Anisotropic Diffusion [26]	SRAD	Weather Prediction [17]	WP
Fast Wavelet Transform [109]	FWT	Parallel Reduction [109]	RD	LIBOR Monte Carlo [17]	LIB
Scalar Product [109]	SP	Streamcluster [26]	SC	CFD Solver [26]	CFD
BFS Graph Traversal [26]	BFS	RAY Tracing [17]	RAY	3D Laplace Solver [17]	LPS
Separable Convolution [109]	COV	Matrix Multiplication [109]	MM	Black-Scholes Option Pricing [109]	BS
Nearest Neighbor [26]	NN	Back Propogation [26]	BP		

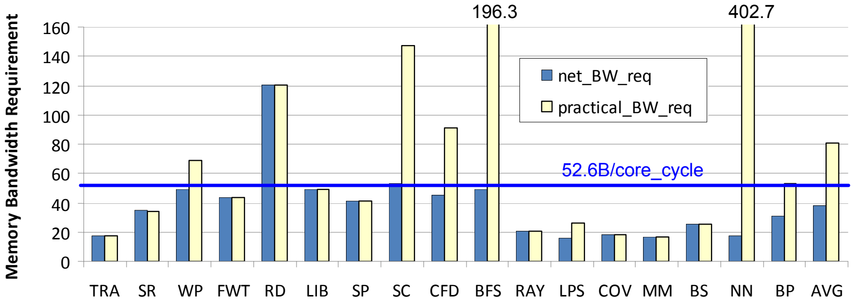


Figure 6.6: Memory bandwidth requirement

Some benchmarks consist of more than one kernel. All results presented here are aggregation of all kernels invoked in benchmark execution.

6.5.1 Memory Bandwidth Requirement

Figure 6.6 shows the bandwidth requirement of our benchmarks, with the metric being Bytes per GPU core cycle. Note, the "memory bandwidth requirement" in this section denotes bandwidth to feed data to GPU core pipelines, **not** off-chip memory bandwidth. The left bar of each group shows the *net* memory bandwidth requirement (*net_BW_req*), calculated by dividing the net memory access (assuming 1B access granularity) by the execution time assuming normalized IPC=1. *net_BW_req* actually gives the lower bound of sustained memory bandwidth necessary to keep the GPU core pipeline busy. The right bar, *practical_BW_req*, is calculated by dividing the required data (assuming 32B access granularity) by the execution time on GPU cores with perfect memory system, but taking into account warp control flow divergence penalty [47].

The disparity between the *net* and *practical* benchmark bandwidth requirement reflects its *memory utilization efficiency* (Section 6.5.5). Among all benchmarks, NN shows the largest gap, because the kernel code exhibits extremely high vertical locality (intra-thread spacial locality, Section 6.3.1), with 32 memory loads from within each thread accessing contiguous 32 Bytes global memory. Unfortunately, vertical locality is not captured by our modeled GPU core (with no hardware cache). For such benchmarks, it is desirable to adopt on-chip hardware data caches to reduce the off-chip bandwidth requirement.

The solid bar in Figure 6.6 lines out the peak bandwidth of our baseline DRAM subsystem, 52.6B/core_cycle. Considering there are 128 scalar lanes in total from 16 8-way SIMD cores, this translates into a bandwidth-FLOP ratio of 0.41. Interestingly, Figure 6.6 shows that, the peak bandwidth of the baseline DRAM system indeed exceeds most benchmarks' bandwidth requirements. However, as we will see in following sections, most of them are actually memory bound during execution, due to low *DRAM efficiency*.

6.5.2 DRAM Access Granularity Distribution

Table 6.4 shows the static instruction count scheduled at each granularity level by our runtime scheduler for each benchmark in our experiments. Note, 32, 64, 128 and 256B denote the possible granularity of the *subwarp vector accesses* (Section 6.4.1) generated by the corresponding memory instructions. As we can see in Table 6.4, no particular pattern in the granularity type distribution is observed across all benchmarks – their optimal granularity spread across all four levels. Moreover, 10 out of the 17 benchmarks require at least two access granularity types. This suggests that in general a *single* optimal access granularity for all memory accesses in a kernel is not feasible, and, confirms the need for scheduling memory instructions separately, based on access patterns.

The last row of Table 6.4, AF (analysis fail), indicates the number of memory instructions for which the access pattern analysis failed in our experiments. The reasons for analysis fails include: 1) skeleton transition operations or operation compositions used in address generation are not supported by the compiler; or 2) access patterns are dependent on the control flow path (Section 6.3.2). In either case, an analysis fail will result in the minimal access granularity (32B in our experiments) being scheduled (note the first row of data in Table 6.4 already includes such AF instructions). Specially, in case 2), the obtained multiple access pattern versions (Section 6.3.2) are simply dropped by the runtime scheduler, because it runs on the host CPU before the kernel launches, having no clue about the dynamic path taken by the warp execution. The only exception is addresses generated in loops with constant steps across loop iterations – in this case, multiple pattern versions differ only in skeleton parameter f (Section 6.3.2), and are treated as a single valid pattern by the runtime granularity scheduler.

Note, a granularity larger than 32B in our experiments indicates that the subwarp vector memory accesses generated by the corresponding memory instruction will fetch data more than needed by themselves. Therefore, Table 6.4 also confirms that the horizontal locality widely exists in GPGPU applications.

Table 6.4: DRAM access granularity distribution

	TRA	**SR**	**WP**	**FWT**	**RD**	**LIB**	**SP**	**SC**	**CFD**	**BFS**	**RAY**	**LPS**	**COV**	**MM**	**BS**	**NN**	**BP**
32B	0	12	100	3	0	12	0	108	60	48	8	2	0	0	0	40	3
64B	1	26	0	0	0	0	0	0	0	0	5	0	12	2	0	0	11
128B	0	0	0	0	2	0	0	0	0	0	0	2	0	0	0	0	0
256B	0	0	0	62	2	15	2	180	86	16	0	0	0	0	30	0	0
AF	0	0	86	3	0	2	0	108	60	48	8	2	0	0	0	0	0

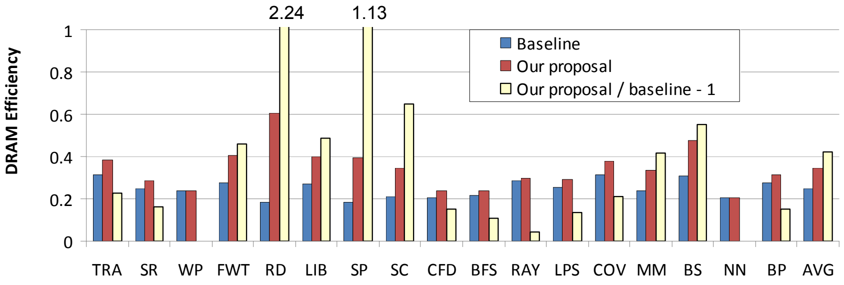


Figure 6.7: DRAM efficiency

6.5.3 Improved DRAM Efficiency

DRAM efficiency is defined as the ratio between DRAM data bus actual transfer cycles and the number of cycles with pending DRAM access:

$$E_{tot} = \frac{\sum_{k=0}^{\#kernels-1} \sum_{i=0}^{\#channels-1} \#bus\_transactions_{k,i}}{2 \cdot \sum_{k=0}^{\#kernels-1} \sum_{i=0}^{\#channels-1} \#active\_cycles_{k,i}}$$

where $\#bus\_transactions_{k,i}$ is the number of accomplished DRAM bus transactions in channel i during the execution of kernel k, and $\#active\_cycles_{k,i}$ is the number of DRAM bus cycles during which channel i is not *completely idle*. Here completely idle denotes the DRAM channel status with no pending accesses in its request queue and any ongoing memory accesses. The factor *2* in the equation is due to the fact that there are two bus transactions in one cycle for *dual-data-rate* memories.

Fig. 6.7 shows the overall DRAM efficiency of our adaptive DRAM access granularity scheme for all benchmark applications. Within each group, there are three bars. The left and middle show the DRAM efficiency with the baseline and our proposal, respectively. The right bar shows the net gain. The RD and SP kernels see the largest DRAM efficiency improvement, since they have very few memory access instructions with simple access patterns, and demonstrate high horizontal locality. Meanwhile, we have observed that most fetched data are efficiently utilized by neighbor warps in such kernels, even at the largest access granularity of 256B (Figure 6.9). Other benchmarks also show varied efficiency increment, with the only exceptions being WP and NN. The reason is that the *identified* access patterns do not have exploitable horizontal locality, while for the rest instructions the analyzer failed to extract the

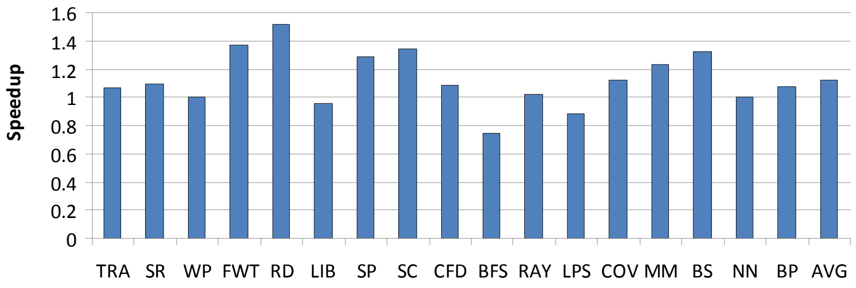


Figure 6.8: Speedup over the baseline

precise access patterns as they are control flow dependent. As a result WP and NN are executed with the same access granularity as the baseline. Nonetheless, the average DRAM efficiency is improved by our adaptive memory granularity scheme by 1.42X, a strong indication that our approach is capable of efficiently exploiting the available horizontal locality in the benchmarks.

6.5.4 Improvement in Overall Performance

Figure 6.8 shows the overall performance improvement with our adaptive memory granularity scheme. On average, the performance is improved by 12.3% over the baseline. Note, some benchmarks, such as BFS and LPS, see substantial overall performance degradation. Detailed analysis reveals that sever *inter-warp control flow divergence* occurs during kernel execution. In this case, neighbor warps execute along different control flow path, rendering the extra data fetched at large granularity being wasted. As a result, memory bandwidth is wasted and system performance is degradable, especially for BFS (which is heavily memory bound). This can be alleviated by utilizing both static analysis results from the pattern analyzer, and also the control flow information avaialbe at *warp execution time*. We leave this as our future work.

6.5.5 Memory Access Utilization

Memory access utilization is defined as the portion of data loaded from off-chip memory which is *truly utilized* by the processor core pipeline. Fundamentally, imperfect memory access utilization results from the *mismatch* between the data manipulation granularity in program world and the *much larger* data

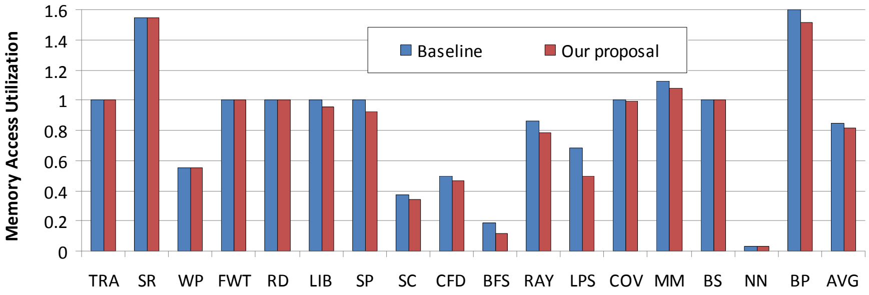


Figure 6.9: Memory access utilization

movement granularity in a real machine. For example, the most frequently used data types are of 1 to 8 Bytes in CUDA C code, whereas the smallest burst size of GDDR3 interface is 32 Bytes.

Figure 6.9 shows the memory access utilization for both the baseline and our adaptive access granularity scheme. In general, when an application employs *irregular* memory access patterns, the memory access efficiency drops. For example, SC and BFS make intensive use of pointer accesses inside a thread (SIMD lane), forming scatter/gather accesses during SIMD pipeline subwarp execution, resulting in very low memory access efficiency. As mentioned in Section 6.5.1, NN exhibits exclusively vertical locality not captured in our GPU model, therefore it suffers the lowest memory access utilization (in most cases only one single Byte out of 32B DRAM data is utilized). Nonetheless, irregular memory access is an open issue, not only for data parallel architectures (in forms of scatter/gather), but also for general-purpose processors (e.g., pointer-chasing). For GPU predicated SIMD execution, another source of low memory access utilization is warp divergence. In such case, only part of the SIMD lanes and the data transfer bandwidth are utilized.

Interestingly, we have observed a good correlation between the memory access under-utilization and the portion of AF(analysis fail) memory instructions (the last row of Table 6.4) in total memory instructions (sum of other 4 data rows in the same column). Remember, one major cause of pattern analysis fails is access pattern being dependent on control flow paths (Section 6.5.2). Therefore, such correlation suggests that control flow divergence is actually a major contributing factor to GPU memory under-utilization. The impact is in two fold: 1) it *directly* degrades memory access utilization, as described in above paragraph; 2) it *indirectly* decrease memory utilization, by creating irregular memory access patterns in different control flow paths.

Memory access utilization should be taken good care of, especially in our adaptive granularity proposal. With the increased DRAM access granularity, the utilization tends to decrease. Fortunately, on average our approach did not substantially decrease memory access efficiency, as shown in Figure 6.9, thanks to the conservative scheduling we have adopted (Section 6.3.3). However, some benchmarks, such as BFS and LPS, still see significant memory access efficiency degradation.

Another interesting observation in Figure 6.9 is that, despite its relatively small size (as compared to a normal L1 data cache), the vector MSHR managed capturing certain temporal locality in some of the benchmarks, such as SR, BP and MM. Investigation in the kernel code reveals different temporal locality sources. E.g., with SR, the temporal locality results from the threads inside a same CTA accessing the surrounding elements overlapping each other; while for MM, the data sharing occur among CTAs accessing same matrix tiles. For the other applications, the temporal locality is either limited, or not captured by the vector MSHR.

6.5.6 Effect on a GPU with Less Capable Interconnect

In order to see how our proposal interacts with different interconnection capabilities, we did the same experiments with an interconnect with the same topology and intersection bandwidth, with the number of virtual channels reduced from 2 to 1. Figure 6.10 shows the DRAM efficiency improvement (on the top) and overall performance speedup (on the bottom). It can be seen that the similar trend for both efficiency and performance improvement is also maintained in this cases. On average, our proposed adaptive memory access granularity scheme gives a improvement of 35.1% and 77.2%, for DRAM efficiency and overall performance, respectively. Further analysis reveals that, the effect of our adaptive memory access granularity is twofold in GPUs with less capable on-chip interconnect. First, our adaptive memory access granularity generates less traffic, with lower packet overhead[12], when horizontal locality exists in the kernel execution. Second (more importantly), the original memory access stream from a GPU core gets *interleaved* in the less capable interconnect by common fairness-oriented flow control policies. As a result, the locality in the original memory access streams (if any) is broken at the memory controller

---

[12] E.g., for our 16B flit size, the packet overhead for 32B data is 50% (32B data + 8B header requires 3 flits), which is reduced to 6.25% for a 256B data packet.

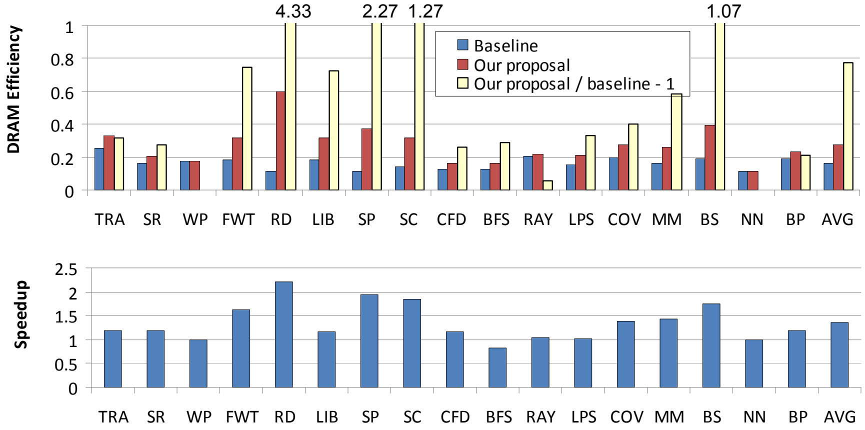


Figure 6.10: DRAM efficiency improvement and overall performance speedup (*num_vcs*=1)

side[13]. This is particularly true for GPUs with less capable interconnect, where the interconnect is saturated and consequently such *inter-core memory access interleaving* is more frequent. Our adaptive memory access granularity scheme works perfectly in this scenario, in that it *enforces* the locality from each core's memory access stream, *adapted to available horizontal locality*. This explains the higher performance improvement in a less capable interconnect GPU.

6.6 Discussion

6.6.1 Effect of Large DRAM Data Packets

One possible drawback of our proposal is that, it might degrade the interconnect performance, since our proposal fetches large DRAM data packets.

In general, large packets can degrade interconnect performance, since they tend to cause traffic congestion, and decrease the utilization of channel resource and buffers. Besides, large packets also cause QoS problems [37]. However, in the particular context of throughput-oriented GPU accelerators,

---

[13] This is true, even for a memory controller with a capable out-of-order access scheduling (such as the one modeled in our evaluation), since the request queue depth (scheduling window size) is limited.

we anticipate that moderate increase in memory access delay for a single thread/warp does not decrease system throughput, as long as it can be hidden by concurrently active warps execution. On the other hand, we have observed that the interconnect traffic congestion resulting from large read data packets is quite rare, probably due to the fact that the data traffic from memory to GPU core forms a *few-to-many* pattern [16]. Furthermore, solutions to alleviate the side effect of large packets exist, e.g., the *split packet* technique [37].

6.6.2 Impact of HW Cache

On-chip HW data caches are recently adopted in commercial GPUs, e.g., NVIDIA Fermi. Please note that, for spatial locality among threads, although caches can capture certain amounts of it by the (wide) cache line access, are unable to dynamically optimize DRAM access granularity. More specifically, the data cache can capture only one row in Table 6.4 (say, 64B cache line size). For memory accesses preferring access granularity different from the cache line size, the cache block access either wastes DRAM bandwidth when the access granularity is smaller than the cache line size (e.g., the 3 irregular access patterns in FWT in Table 6.4) or, it incurs unnecessary inter-core memory access interference at the memory controller side. In the former case the use of HW data cache is questionable and indeed our approach has no impact due to dominating cache block accesses. In the latter case, when the preferred access granularity is larger than cache lines (e.g., for the 62 regular access patterns in FWT in Table 6.4) our approach will work by correctly adjusting DRAM access granularity to the access patterns. In addition, with the help of on-chip L1 data cache, spatial locality inside each thread can also be exploited by our scheme, which is otherwise difficult (Section 6.5.1). This said, we consider our approach general and not limited by the fact that our experiments consider only GPUs with no L1 data caches.

6.6.3 Application to SPMD Barrel Processing

In Section 6.3.1 we have analyzed the horizontal locality being inherent in barrel execution of SPMD programs. GPU is the most prominent example of manycore architectures which embody both SPMD programming and the barrel execution model, at present time. This work experimented with a holistic framework with compile-/runtime analysis, scheduling and the co-designed architectural support, to exploit and enforce the horizontal locality and improve memory efficiency. Although the techniques are implemented and evaluated with a GPU model, we believe our proposal is also effective for other architectures employing SPMD barrel execution, in capturing the horizontal locality.

6.6.4 Possible Improvement

In this chapter, we presented a holistic framework for GPU memory access optimization. By combined compile-time access pattern analysis and runtime pattern extraction, we obtained the detailed access pattern information for *local* and *global* memory access instructions. With this information, we optimize the memory access performance by calculate and enforce the optimal granularity for each off-chip memory access pattern, to exploit horizontal locality and improve DRAM efficiency. Based on this work, two directions can be immediately explored: (1) to reduce runtime overhead. For example, since the importance of individual memory accesses may be different (depending on, e.g., high/low invocation frequency which can be roughly identified by compile time analysis), it may not be necessary to *extract* access patterns for all instructions. Rather, only those *critical accesses*, the access granularity of which may have strong impact on overall performance (as suggested in Table 6.4), need to be further investigated at runtime. (2) to identify more opportunities for increasing DRAM efficiency and overall performance. In fact, we have utilized only a small portion of the valid access pattern information (e.g., the access stride parameters in dimension-Y, b and c, are not used in the access granularity scheduling experimented in this work). Naturally, the extra access information characterizes certain aspects of the dynamic behavior of kernel execution. Therefore, it can be further exploited, e.g., to reduce GPU DRAM channel conflicts and interconnect congestion, by appropriate memory aware CTA issue and warp scheduling schemes.

6.6.5 Contrast to Closely Related Art

Ocelot [39] is a dynamic compilation framework to map the NVIDIA CUDA applications onto diverse multithreaded platforms. It includes a dynamic binary translator from PTX code to x86 ISA and others. While Ocelot is a general framework, our access pattern analyzer is dedicated for external memory access analysis, and designed with minimal runtime overhead objective in mind. Moreover, our runtime scheduling is assisted by the co-designed hardware which is essential in *realizing* the adaptive memory access granularity.

PTX transformations, such as thread-fusion used in MCUDA [136] and GPGPU Compiler [151], have also been proposed to change CUDA code memory access patterns, in order to optimize their execution on existing GPUs. In our view, such techniques make a good effort in *creating* horizontal locality. In contrast, our proposal focuses on *exploiting* and *enforcing* the horizontal locality in existing code. Therefore, PTX transformations are complementary to our work, and we believe the combination of two have the potential for further GPU performance improvement.

Memory coalescing [108] is a hardware mechanism in NVIDIA GPUs to buffer and merge *intra-warp* memory accesses. In this way variable-sized loads, depending on access locality, are also supported. However, the effectiveness of coalescing is limited to half/single-warp[14]. In contrast, our approach takes advantage of the high level access pattern information, and captures horizontal locality both inside and among warps even when **not** issued back-to-back (Section 6.3.1). Thus our scheme generalizes memory coalescing, and the ”coalescing rules” are effectively relaxed in our system.

A recent study on GPU prefetching proposed *Inter-Thread Prefetching (IP)* [88], somehow similar to our work in spirit, both recognizing the GPU specific locality among parallel threads. *IP* focused on *latency reduction* using speculation, assuming inadequate parallelism to hide memory latency. In contrast, our work emphasizes on *off-chip memory access efficiency* using *accurate access pattern* information *without speculation*, in a context where the off-chip memory bandwidth efficiency becomes the performance bottleneck[15]. Moreover, our work is novel in analyzing the horizontal locality in SPMD barrel execution embodied by GPUs.

At the GPU on-chip interconnect level, the work [152] also addresses the *memory access streams interleaving* problem, using a customized flow control design optimized for this scenario. Different form their work, our approach utilizes high-level access pattern information at the source (GPU cores) of the traffic, and adjust memory access granularity according to the available horizontal locality. Also, our adaptive memory access granularity scheme does not change the behavior of kernel execution with less or no horizontal locality.

---

[14] depending on GPU generation

[15] For example, for the same application “Black-Scholes”, #max warps/core is 12 (see Table III in [88]) in the GPU modeled in [88] (NVIDIA GeForce 8800GT). While in the GPU modeled in our study (GeForce GTX 280) #max warps/core becomes 24, resulting in doubled available parallelism (with per-core off-chip bandwidth not significantly increased). Hence, the memory efficiency, rather than latency, becomes the first-order issue.

6.7 Summary

In this chapter, we analyzed horizontal locality found in GPUs with SPMD barrel execution. We proposed an adaptive DRAM access granularity scheme to exploit this locality and reduce memory access interference among GPU cores to improve DRAM efficiency. Our scheme comprises a compiler and runtime access pattern analyzer, a runtime granularity scheduler, and the co-designed elastic MSHR HW support. Our results show that, on average our proposal improves DRAM efficiency by 1.42X and the overall performance 12.3%, for a set of representative GPGPU applications.

Note. The contents of this chapter is based on the the following papers:

C. Gou, G. N. Gaydadjiev, **Exploiting SPMD Horizontal Locality to Improve Memory Efficiency**, IEEE Computer Architecture Letters, Vol. 99, DOI: http://doi.ieeecomputersociety.org/10.1109/L-CA.2011.5, 2011

C. Gou, G. N. Gaydadjiev, **Improving GPU DRAM Efficiency Using Access Granularity**, submitted to the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44), 2011

7

Conclusions and Future Directions

7.1 Conclusions

This dissertation identified that one major factor of the memory access inefficiency in data parallel accelerators is the mismatch between the access pattern required by the workloads and the one optimal for the physical data layout. We proposed customizable memory schemes to address the access efficiency problem of parallel memories in terms of bandwidth utilization and access time. Different from conventional approaches, customizable memory schemes employ systematic approach with well coordinated hardware and software efforts. Our approach leverages the extra information about the application and its execution characteristics available within code written under the corresponding programming styles/models of data parallel accelerators. Furthermore, our approach extracts such information at proper application phases using customized programming interfaces, compile-time analysis, runtime optimizations and architectural extensions, to steer the memory scheme hardware and improve memory efficiency.

To support conflict-free vector access for multiple stride families frequently required in workloads with irregular memory accesses, this dissertation proposed two novel hardware parallel memory schemes in Chapter 3. The first one, the SAMS hardware scheme, supports both unit-stride and strided conflict-free 1D vector memory access. To the best of our knowledge, the proposed SAMS scheme is the first of its kind that supports atomic parallel access without limiting the vector length and without using more memory banks than the parallel access degree. The second access scheme we proposed was the 2DSMM scheme able to support conflict-free vector access in a 2D environment.

To illustrate how such hardware parallel memory schemes can be integrated into the system level of a contemporary data parallel accelerators, this dissertation proposed a parallel memory design to bridge the discrepancy between data representations in memory and those favored by the SIMD processor by customizing the low-level address mapping. The key in the proposed design is a customizable memory scheme (called "SAMS Multi-Layout Memory") in the memory hierarchy backed up by the SAMS hardware that provides both *Array of Structures* (AoS) and *Structure of Arrays* (SoA) views of structured data to the processor, appearing to have maintained multiple layouts for the same data. With such multi-layout memory, optimal SIMDization with dynamically changing access patterns can be achieved. We did show in Chapter 4 how the customizable memory scheme enalbed by SAMS can significantly improve both memory access efficiency and system performance. Our scheme achieved all of the above performance targets while reducing programming efforts.

To minimize the negative impact of on-chip memory bank conflicts on system throughput, we proposed a novel elastic pipeline design which decouples bank conflicts from pipeline stalls in multithreaded vector execution. We described in Chapter 5 how the proposed elastic pipeline together with the co-designed bank-conflict aware warp scheduling substantially reduce pipeline stalls and improve the overall performance for workloads characterized with many on-chip shared memory bank conflicts, when applied to GPUs. This is all achieved with trivial hardware overhead. In the mean time, our proposal also leads to reduced GPU programming complexity by relieving the burden of avoiding shared memory bank conflicts from the programmer.

To improve external memory access efficiency in the case of data parallel manycore accelerator architectures adopting barrel execution of SPMD programs, we proposed a holistic framework for off-chip memory access optimizations in Chapter 6. We analyzed the horizontal locality among parallel execution of independent worker threads in a core. To prevent the available horizontal locality from being broken by the interference among accelerator cores memory accesses, this dissertation described an adaptive memory access granularity scheme to exploit and enforce such horizontal locality, based on the proposed holistic memory optimization framework. We showed in Chapter 6 that with the proposed techniques, DRAM efficiency is dramatically increased and overall system performance is improved.

To summarize, this dissertation demonstrated: 1) both memory access patterns and physical data layout can be optimized to improve memory efficiency; 2) optimal solutions require striking the right balance between adjusting either memory access patterns or physical data layout, with respect to programming efforts and hardware efficiency (in terms of performance gain/additional HW cost). These two findings, within the context of contemporary data parallel accelerators, form the core of our customizable memory schemes. Especially, a key factor of customizable memory techniques is how to properly capture the access pattern information. We showed in this thesis that it can be achieved by well coordinated HW/SW efforts across multiple computing system abstraction layers: customized programming interfaces, static code analysis, runtime optimziations and architectural extensions.

In Chapter 1 we posed three research questions which we answered throughout this thesis. Here are their summarized answers:

- Conflict-free accesses for common 1D and 2D vector patterns can be supported, by extending traditional parallel memory schemes (SAMS and 2DSMM);
- Systematic approaches should be leveraged to capture the memory access pattern information, with well coordinated HW/SW efforts across computing system abstraction layers (SAMS Multi-Layout Memory, elastic pipeline and elastic MSHR);
- Access pattern information can be exploited using customizable HW, by adapting the physical data layout to access patterns (SAMS Multi-Layout Memory); by decoupling on-chip memory access irregularities from pipeline stalls (elastic pipeline); and by enforcing the horizontal locality and reducing inter-core memory interference (elastic MSHR).

At a broader scope, we argue that the techniques proposed in this thesis move one step forward in expanding the effectiveness scope of contemporary data parallel accelerators. Such techniques can also be useful for the long-term merging of the data parallel architectures with other parallel architectures (such as task parallel), toward unified accelerator architectures.

7.2 Future Research Directions

This dissertation introduced customizable memory schemes to increase parallel memory access efficiency. The customizable memory techniques comprise two major components: 1) customizable hardware memory schemes, and 2) customizable parallel memory access scheduling. There are several possible future research directions to improve these two components and to further increase memory efficiency of contemporary data parallel accelerators:

- In spite of various existing hardware parallel memory schemes in the literature (including the SAMS and 2DSMM introduced in this thesis), there is still room for innovation in new parallel memory schemes to support more conflict-free patterns. However, two issues should be kept in mind. First, the hardware scheme design should be investigated at system-level. Especially, questions should be asked: how is it related to mainstream data parallel architectures? How important are the new access pattens supported by the scheme, with respect to real/emerging applications? The second issue is the additional cost due to the new schemes, in terms of extra latency/silicon area and power consumption;

- With the additional memory access flexibility provided by conflict-free hardware memory schemes (such as SAMS and 2DSMM), it may be worthwhile to revisit relevant program flow analyses/transformations, to help renovate automatic SIMDization;

- In the SAMS Multi-Layout Memory design, we have shown that, a customizable memory scheme which is adaptive for individual data structures/access patterns, has the potential to boost the performance of emerging applications on SIMD processors. However, as our work targeted the access of array-based data structures as a first step, it would be interesting to apply similar approaches to more difficult problems (e.g., linked-list/trees), with proper customizable hardware address mapping and the corresponding software abstractions;

- For the elastic pipeline design, it would be interesting to evaluate its effect on GPU cores with banked on-chip hardware caches. Moreover, it would be desirable to improve the accuracy of bank conflict degree prediction for irregular shared memory access patterns for better warp scheduling efficiency. Another interesting follow-up research direction could be the co-optimization of on/off-chip memory access efficiency;

- Current implementation of the adaptive memory access granularity proposal only focuses on memory loads; however, our framework also provides the access pattern information for stores. Since system performance can also be improved by optimizing memory writes [137], it would be interesting to explore mechanisms using access pattern information to optimize memory access efficiency for both loads and stores. Moreover, since only part of the memory access pattern information is utilized in our adaptive memory access granularity scheme, it would be worthwhile to further exploit the holistic memory optimization framework to orchestrate GPU warp execution for better memory efficiency and overall performance, by techniques such as memory efficiency aware block issuing, and warp scheduling;

- The effectiveness of hardware parallel memory schemes are only evaluated for on-chip SRAMs. Since DRAM access efficiency is also sensitive to the low level address mapping, it is desirable to expand the utilization of such parallel memory schemes to DRAMs, with necessary extensions suited for DRAM organizations;
- Another interesting work is to optimize both memory access patterns and physical data layout in a unified framework. We believe this co-design has the potential to bring extra benefits, by leveraging on techniques from both memory access scheduling and parallel memory schemes;
- Finally, with the emerging trend toward unified accelerator architectures [69, 113, 128, 135], both accelerator HW substrate and programming models able to capture data and task parallelism are jointly explored, for better balance between programming efforts and HW efficiency. This introduces new challenges for customizable memory schemes to further improve programmability while maintaining efficiency.

A

SAMS Conflict-Free Access Proof

In this section, we will present the mathematical foundations for the basic SAMS scheme, and its derivative, the Matched SAMS Scheme, both proposed in Chapter 3. “Conflict-free” conventionally means that data to be referenced in one access are located in different modules so that they could be accessed in parallel. However, the concept of conflict-free in the SAMS context is slightly extended such that it includes the cases where there are two references located in the same module which reside in the same row (therefore they could also be accessed in parallel). For the sake of clarity, we will use the term “strictly conflict-free” when we refer to the conventional meaning. Now we will first illustrate some properties of the SAMS scheme, and then give the proof of its capability of supporting conflict-free access in theorems later.

Property 1. The period of SAMS module assignment function is 2^{q+s}.

Proof. There are three cases in the SAMS scheme.

I) $s = 0$. $m(a) = a\%2^q = \left(a + 2^{q+s}\right)\%2^q = m(a + 2^{q+s})$.

II) $1 \leq s \leq q$.

$$\begin{aligned} m(a) &= \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}\right\rangle \\ &= \left\langle a_q \cdots a_s,\ \left(\left(a + 2^{q+s}\right) \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}\right\rangle \qquad \text{(A.1)} \\ &= m\left(a + 2^{q+s}\right). \end{aligned}$$

Equation (A.1) stands because the module assignment function of Harper’s XOR scheme, namely $\left(a \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}$, has a period of $2^{(s-1)+(q+1)} = 2^{q+s}$ [36].

III) $s > q$. In this case the module assignment function is precisely that of Harper’s XOR scheme configured with 2^q banks and stride 2^s, which has a period of 2^{q+s} [36]. ■

Property 2. When $1 \leq s \leq q$, the SAMS scheme is conflict-free for any stride

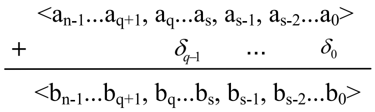


Figure A.1: Binary bits representation of $b = a + \delta \cdot 2^0$ when $1 \leq s \leq q$

$S = 2^{s'}(0 \leq s' \leq s-1)$.

Proof. The SAMS scheme is conflict-free, if we could guarantee that the conflicting items in linear address space are mapped to the same row. Described in mathematics, given two different addresses a and b, we need prove $r(a) = r(b)$ under the conditions $m(a) = m(b)$ and $b = a + \delta \cdot 2^{s'}(1 \leq \delta \leq 2^q - 1)$(without loss of generality we assume $b > a$).

First we examine the equation $m(a) = m(b)$. Note when $1 \leq s \leq q$ $m(a) = m(b)$ means

$$\left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} \right\rangle = \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} \right\rangle \ ,$$

i.e.

$$a_q \cdots a_s = b_q \cdots b_s \qquad \text{(A.2)}$$

$$\left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} = \left(b \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} \ . \qquad \text{(A.3)}$$

According to the definition, equation (A.3) could be further expanded as

$$\begin{cases} a_0 \oplus a_{q+1} = b_0 \oplus b_{q+1} \\ a_1 \oplus a_{q+2} = b_1 \oplus b_{q+2} \\ \ldots \\ a_{s-2} \oplus a_{q+s-1} = b_{s-2} \oplus b_{q+s-1} \end{cases} . \qquad \text{(A.4)}$$

When $s' = 0$, we have $b = a + \delta$. The binary bits representation of the addition process is depicted in Figure A.1. Consider $a_q = b_q$(from equation (A.2)) together with Figure A.1, we could see that there is no carry input from bit $q-1$ to bit q during the addition. Accordingly, the high order bits(from bit q on) of a are not affected by the addition of δ, which means $a_{n-1} \cdots a_{q+1} = b_{n-1} \cdots b_{q+1}$, i.e. $\frac{a}{2^{q+1}} = \frac{b}{2^{q+1}}$, which means $r(a) = r(b)$.

When $s' = 1$, we have $b = a + 2 \cdot \delta$, which is depicted in Figure A.2. Note the addition on bit 0 is $b_0 = a_0 + 0$, thus $b_0 = a_0$. Combined with equation (A.4), we have $b_{q+1} = a_{q+1}$, which indicates that there is no carry input from bit q to bit $q+1$. Hence $a_{n-1} \cdots a_{q+1} = b_{n-1} \cdots b_{q+1}$, i.e. $\frac{a}{2^{q+1}} = \frac{b}{2^{q+1}}$, which means $r(a) = r(b)$.

$$\begin{array}{rl} & \langle a_{n-1}\ldots a_{q+1}, a_q\ldots a_s, a_{s-1}, a_{s-2}\ldots a_0\rangle \\ + & \qquad\qquad \delta_q \qquad \cdots \qquad \delta_0\ 0 \\ \hline & \langle b_{n-1}\ldots b_{q+1}, b_q\ldots b_s, b_{s-1}, b_{s-2}\ldots b_0\rangle \end{array}$$

Figure A.2: Binary bits representation of $b = a + \delta \cdot 2^1$ when $1 \leq s \leq q$

Similarly, for $s' = k(k = 2, \ldots, s-1)$, i.e. $b = a + \delta \cdot 2^k$, we have

$$\begin{cases} a_0 &= b_0 \\ a_1 &= b_1 \\ \cdots & \\ a_k &= b_k \end{cases} . \tag{A.5}$$

by examing the process of addition. Considering (A.5) together with (A.4), we know

$$\begin{cases} a_{q+1} &= b_{q+1} \\ a_{q+2} &= b_{q+2} \\ \cdots & \\ a_{q+k+1} &= b_{q+k+1} \end{cases} . \tag{A.6}$$

This indicates that there is no carry input from bit $q+k$ to bit $q+k+1$. Therefore the high order bits(from bit $q+k+1$ on) of a are kept untouched during the addition. Consequently we have $a_{n-1} \cdots a_{q+1} = b_{n-1} \cdots b_{q+1}$, i.e. $\frac{a}{2^{q+1}} = \frac{b}{2^{q+1}}$, which means $r(a) = r(b)$. ■

Property 2 reveals a very interesting feature of the SAMS scheme: it could potentially support conflict-free vector accesses with strides across multiple stride families, under the condition $s \leq q$. Moreover, it is exactly where the idea of the Matched SAMS Scheme originates. We will see how this feature works for the Matched SAMS Scheme later.

Before proving the conflict-free access support of SAMS scheme, first we have to prove that it is a bijection on $\mathbb{E}^n$[1]. Only when the interleaving scheme is a bijection from the linear address space to the transformed space(the module-row-offset trinity in SAMS) could it be consistent in both theory and practice.

Theorem 1. The mapping from linear address a to the module-row-offset trinity in the SAMS scheme is a bijection on $\mathbb{E}^n$.

Proof. As there are three cases in the SAMS scheme, we will discuss them one by one.

---

[1] Denote $\mathbb{E}^n = \{0, 1, \ldots, 2^n - 1\}$.

I) $s = 0$. By concatenating the binary bits of the memory module assignment, row assignment, and offset assignment, we get

$$\begin{aligned} & \langle m(a),\ r(a),\ o(a)\rangle \\ =\ & \left\langle a\%2^q,\ \frac{a}{2^{q+1}},\ a_q \right\rangle \\ =\ & \left\langle a\%2^q,\ \frac{a}{2^q} \right\rangle . \end{aligned}$$

It's clear that the mapping from a to the trinity $\langle m(a),\ r(a),\ o(a)\rangle$ is a bijection on $\mathbb{E}^n$.

II) $1 \leq s \leq q$. By concatenating the binary bits of the memory module assignment, row assignment, and offset assignment, we get

$$\langle m(a),\ r(a),\ o(a)\rangle$$

$$= \left\langle a_q \ldots a_s,\ \left(a \otimes T_{H_{s-1,s+1}}\right)\%2^{s-1},\ \frac{a}{2^{q+1}},\ a_{s-1} \right\rangle \tag{A.7}$$

$$\overset{bijection}{\Leftrightarrow} \left\langle \left(a \otimes T_{H_{s-1,s+1}}\right)\%2^{s-1},\ \frac{a}{2^{q+1}},\ a_q \ldots a_s,\ a_{s-1} \right\rangle \tag{A.8}$$

$$= \left\langle \left(a \otimes T_{H_{s-1,s+1}}\right)\%2^{s-1},\ \frac{a}{2^{s-1}} \right\rangle . \tag{A.9}$$

Note expression (A.9) is virtually the Harper XOR scheme configured with 2^{s-1} memory modules and stride 2^{s+1}(the first part of the binary concatenation is the module assignment function, and the last part is the row assignment function). Therefore, mapping from a to (A.9) is a bijiection on $\mathbb{E}^n$. As the transform between (A.7) and (A.8) is also a bijiection, hence mapping from a to $\langle m(a),\ r(a),\ o(a)\rangle$ is a bijection on $\mathbb{E}^n$.

III) $s > q$. By concatenating the binary bits of the memory module assignment, row assignment, and offset assignment, we get

$$\begin{aligned} & \langle m(a),\ r(a),\ o(a)\rangle \\ =\ & \left\langle \left(a \otimes T_{H_{q,s}}\right)\%2^q,\ \left(\left(\frac{a}{2^q}+1\right)\%2^{n-q}\right)/2,\ \overline{a_q} \right\rangle \\ =\ & \left\langle \left(a \otimes T_{H_{q,s}}\right)\%2^q,\ \left(\left(\frac{a}{2^q}+1\right)\%2^{n-q}\right)/2, \right. \\ & \qquad \left. \left(\left(\frac{a}{2^q}+1\right)\%2^{n-q}\right)\%2 \right\rangle \end{aligned}$$

$$= \left\langle \left(a \otimes T_{H_{q,s}}\right)\%2^q,\ \left(\frac{a}{2^q}+1\right)\%2^{n-q} \right\rangle \tag{A.10}$$

$$\overset{bijection}{\Leftrightarrow} \left\langle \left(\frac{a}{2^q}+1\right)\%2^{n-q},\ \left(a \otimes T_{H_{q,s}}\right)\%2^q \right\rangle \tag{A.11}$$

$$= \left[\left\langle \frac{a}{2^q},\ \left(a \otimes T_{H_{q,s}}\right)\%2^q \right\rangle + 2^q\right]\%2^n . \tag{A.12}$$

$$\begin{matrix} b & b+2^{q+1} & \cdots & b+2^{q+1}\cdot\left(2^{s-1}-1\right) \\ b+2^{s} & b+2^{s}+2^{q+1} & \cdots & b+2^{s}+2^{q+1}\cdot\left(2^{s-1}-1\right) \\ \vdots & \vdots & \ddots & \vdots \\ b+2^{s}\cdot\left(2^{q-s+1}-1\right) & b+2^{s}\cdot\left(2^{q-s+1}-1\right)+2^{q+1} & \cdots & b+2^{s}\cdot\left(2^{q-s+1}-1\right)+2^{q+1}\cdot\left(2^{s-1}-1\right) \end{matrix}$$

Figure A.3: The accessed addresses with stride $S = 2^s$

Note the transform between (A.10) and (A.11) is a bijection. As we know the Harper's XOR scheme which maps linear address a to the row-module concatenation $\left\langle \frac{a}{2^q},\ \left(a \otimes T_{H_{q,s}}\right) \%2^q \right\rangle$ is a bijection on $\mathbb{E}^n$, therefore the mapping from a to $\left[\left\langle \frac{a}{2^q},\ \left(a \otimes T_{H_{q,s}}\right) \%2^q \right\rangle + 2^q\right] \%2^n$ is also a bijection with fixed q. Therefore, the mapping from linear address a to $\langle m(a),\ r(a),\ o(a)\rangle$ is a bijection on $\mathbb{E}^n$. $\blacksquare$

Theorem 2. The SAMS scheme is conflict-free for both unit-stride and stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

Proof. I) $s = 0$. In this case the stride family becomes $\{S \| S = \sigma,\ \sigma\ odd\}$, which includes the unit stride. The SAMS module assignment function when $s = 0$ is the same as that of the simple low-order interleaving scheme, hence it is conflict-free for all odd stride accesses.

II) $1 \leq s \leq q$.

a) Strided access with stride $S = 2^s$.

Suppose the starting address of the strided access is b. Then the accessed items in linear address space are $b,\ b + 2^s,\ \dots,\ b + 2^s(2^q - 1)$, which is shown in Figure A.3. In the figure the address sequence is rearranged into a matrix, where the address increases in a column-major manner, and each row consists of all the references to the same subgroup (Note there are 2^{q-s+1} subgroups in total.). Now we will prove each and every item in the matrix is distributed into a different memory module, by the SAMS module assignemnt function $\left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} \right\rangle$. By examing the address matrix we could see that item $k \cdot 2^{q+1} (k = 0, 1, \dots 2^s - 1)$ does not affect $a_q \cdots a_s$, and the only determinant factor is item $b + k \cdot 2^s (k = 0, 1, \dots 2^{q-s+1} - 1)$, which results different $a_q \cdots a_s$ for different k. In other words, the high order bits of the module assignment function is different for different rows. Now we look into the row. For the i-th row($i = 0, 1, \dots, 2^{q-s+1} - 1$), the address sequence is precisely that generated by the strided access with starting address $b + i \cdot 2^s$ and stride 2^{q+1}. Consequently, the second part of the SAMS module assignment function, which is actually the Harper's XOR scheme configured

$$\begin{array}{lr} & \langle a_{n-1}\ldots a_{q+s},\ a_{q+s-1}\ldots a_{s},\ a_{s-1}\ldots a_{q},\ a_{q-1}\ldots a_{0}\rangle \\ + & \delta_{q-1}\ldots\delta_{0} \\ \hline & \langle b_{n-1}\ldots b_{q+s},\ b_{q+s-1}\ldots b_{s},\ b_{s-1}\ldots b_{q},\ b_{q-1}\ldots b_{0}\rangle \end{array}$$

Figure A.4: Binary bits representation of $b = a + \delta$ when $s > q$

with conflict-free access for stride 2^{q+1}, designates different module indices to different address items in the same row. In other words, $\left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1}$ is different for each and every items in the same row. Together with the fact that $a_q \cdots a_s$ is different for different rows, we could know that each and every items in the address sequence referenced by the strided access are assigned to different memory modules. Hence the SAMS scheme is *strictly* conflict-free for access stride $S = 2^s$.

b) Strided access with stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

Since the SAMS scheme is strictly conflict-free for stride $S = 2^s$ in the sense that all referenced addresses in one access are distributed in different modules, it is also strictly conflict-free for the stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$, according to Theorem 3 in [36].

c). Unit-stride access.

This has already been proved in Property 2.

III) $s > q$.

a) Strided access with stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

When $s > q$, as the SAMS scheme adopts the module assignment function from Harper's XOR scheme directly, therefore it is *strictly* conflict-free for strided access with stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

b) Unit-stride access.

To prove the SAMS scheme to be conflict-free for unit-stride access, we only have to prove that the conflicting items in linear address space are mapped to the same row. Depicted in mathematics, given two different addresses a and b, we need prove $r(a) = r(b)$ under the conditions $m(a) = m(b)$ and $1 \le |b - a| \le 2^q - 1$.

Assume $b = a + \delta (1 \le \delta \le 2^q - 1)$. Consider the binary bits representation of $b = a + \delta$, as depicted in Figure A.4. From the figure we could see that, if $a_{q-1} \ldots a_0 = b_{q-1} \ldots b_0$, then $\delta = 0$, which means $a = b$. Therefore

$a_{q-1} \ldots a_0 \neq b_{q-1} \ldots b_0$. In addition we know $m(a) = m(b)$, i.e.

$$\begin{cases} a_0 \oplus a_s & = & b_0 \oplus b_s \\ a_1 \oplus a_{s+1} & = & b_1 \oplus b_{s+1} \\ \ldots \\ a_{q-1} \oplus a_{q+s-1} & = & b_{q-1} \oplus b_{q+s-1} \end{cases} .$$

Therefore we know $a_{q+s-1} \ldots a_s \neq b_{q+s-1} \ldots b_s$. Consequently we know that there should be a carry input from bit $s-1$ to bit s, which is the only possible way to make the equation in Figure A.4 stand. Furthermore, as the carry outcome of bit $s-1$ could only come from that of bit $q-1$, therefore we have

$$a_{s-1} \ldots a_q = 1 \ldots 1 \tag{A.13}$$

$$b_{s-1} \ldots b_q = 0 \ldots 0 \tag{A.14}$$

$$b_{n-1} \ldots b_q = a_{n-1} \ldots a_q + 1 \, . \tag{A.15}$$

As $b_q = 0$(from Equation (A.14)), we could further get

$$\frac{b_{n-1} \ldots b_q}{2} = \frac{b_{n-1} \ldots b_q + 1}{2} \, . \tag{A.16}$$

Combining equation A.15 and Equation A.16, we know

$$\frac{a_{n-1} \ldots a_q + 1}{2} = \frac{b_{n-1} \ldots b_q + 1}{2} \, ,$$

i.e.

$$\left(\frac{a}{2^q} + 1\right)/2 = \left(\frac{b}{2^q} + 1\right)/2 \, ,$$

which indicates that $r(a) = r(b)$. This means that any conflicting items(items located in the same memory module) under unit-stride access in the SAMS scheme are assigned to the same row, therefore they could be referenced simultaneously in one access. ■

Corollary 1. The Matched SAMS Scheme is conflict-free for stride 1(unit stride), $2, \ldots, 2^{q-1}$ and stride family $\{S \| S = \sigma \cdot 2^q, \ \sigma \ odd\}$.

Proof. In the Matched SAMS Scheme, the parameter s is set to q, therefore Corollary 1 is virtually the direct application of Property 2 and Theorem 2. ■

The Matched SAMS Scheme is simple yet powerful, because of the large stride range it covers. In general, the Matched SAMS Scheme is capable of supporting conflict-free accesses with strides from $log_2(\#modules) + 1$ families. For example, if we have a parallel memory system with 8 memory modules which deploys the Matched SAMS Scheme, then it could provide conflict-free access for unit stride, stride 2, stride 4, and any stride of $8 \cdot \sigma(\sigma\ odd)$. Thus the potential benefit is very promising in high performance vector processing systems, where a large number of processor clocks are spent on loading, packing and unpacking data from memory.

Note. The contents of this appendix is based on the the following report:

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **Matched SAMS Scheme: Supporting Multiple Stride Unaligned Vector Accesses with Multiple Memory Modules**, CE Technical Report, CE-TR-2008-06, October 2008

B

2DSMM Properties and Formal Proof

B.1 Properties of the Basic XOR Scheme

The 2DSMM is based on the XOR scheme proposed by Harper III[Harper92](we will refer to it as *the basic XOR scheme* hereafter), which provides conflict free strided vector access with multiple memory modules. The main properties of the basic XOR scheme is enumerated in the following. The basic XOR scheme is

$$\begin{cases} m(a) = (a \otimes T)\%2^n \\ r(a) = a/n \end{cases}$$

where $T = \prod_{k=0}^{\min(n,s)-1} T_{k+\max(n,s),k}$. a is the address in linear address space, 2^n the number of memory modules, and 2^s the power of two part of the access stride. $m(a)$ is the module assignment function and $r(a)$ is the row assignment function.

Property 1 Any 2^n strided accesses with stride $S = \sigma \cdot 2^s(\sigma$ is odd) are assigned to 2^n different modules by the memory assignment function $m(a)$[Harper92]. In other words, $(b \otimes T)\%2^n$, $((b + \sigma \cdot 2^s) \otimes T)\%2^n$, ..., $((b + (2^n - 1) \cdot \sigma \cdot 2^s) \otimes T)\%2^n$ are different(between 0 and $2^n - 1$) for any arbitrary base address b and odd σ. This is indeed the *conflict free* requirement which is the basic XOR scheme designed for.

Property 2 Any 2^n unit-stride accesses are assigned to 2^n different modules by the memory assignment function $m(a)$, under the condition that the base address(the address of the starting position) is at the boundaries of $\{\tau \cdot 2^{\min(n,s)} \| \tau \in \mathbb{N}\}$[Gou07]. In other words, $((\tau \cdot 2^{\min(n,s)}) \otimes T)\%2^n$, $((\tau \cdot 2^{\min(n,s)} + 1) \otimes T)\%2^n, \ldots, ((\tau \cdot 2^{\min(n,s)} + 2^n - 1) \otimes T)\%2^n$ are different(between 0 and $2^n - 1$).

Property 3 The basic XOR scheme is periodic with the period of 2^{n+s} when accessed with stride $S = \sigma \cdot 2^s$(σ is odd)[Harper92]. In other words, $((a + k \cdot 2^{n+s}) \otimes T)\%2^n = (a \otimes T)\%2^n$ $(k \in \mathbb{N})$.

It is important to point out that as T_v[1] and T_h of the 2DSMM scheme are instances of the basic XOR scheme, all the three properties above are also applicable to them, with parameters 2^n replaced by 2^p(for T_v) or 2^q(for T_h), and 2^s replaced by 2^{vs} or 2^{hs}, respectively.

B.2 Properties of the 2DSMM Scheme

Property 4 The period of $m_v(i,j)$ (with fixed j) is 2^{p+vs}; the period of $m_h(j)$ is 2^{q+hs}; the period of $(m_v,\ m_h)$ is $2^{p+vs} \times 2^{p+q+hs}$. In other words, $m_v(i + k \cdot 2^{p+vs}, j) = m_v(i,j);\ m_h(j + k \cdot 2^{q+hs}) = m_h(j);\ (m_v(i + k \cdot 2^{p+vs}, j + l \cdot 2^{p+q+hs}),\ m_h(j + l \cdot 2^{p+q+hs})) = (m_v(i,j),\ m_h(j))\,(k, l \in \mathbb{N})$.

Proof. Periodicity of $m_v(i,j)$(j is fixed) and $m_h(j)$ comes directly from Property 3. For the period of $(m_v,\ m_h)$, it is easy to verify that $(m_v(i,j),\ m_h(j)) = (m_v(i + k \cdot 2^{p+vs}, j + l \cdot 2^{p+q+hs}),\ m_h(j + l \cdot 2^{p+q+hs}))(k, l \in \mathbb{N})$. Suppose the period of $(m_v(i,j),\ m_h(j))$ is $2^{p+vs} \times P$ and $P = \tau \cdot 2^{q+hs}(1 \leq \tau \leq 2^p)$. By examining the definition of α and β we know

$$\begin{aligned} \alpha(j + l \cdot P) &= (\alpha(j) + l \cdot \tau)\%2^p \\ \beta(j + l \cdot P) &= \beta(j) \end{aligned}$$

which means that

$$\begin{aligned} & m_v(i + k \cdot 2^{p+vs},\ j + l \cdot P) \\ = \ & m_v(i,\ j + l \cdot P) \\ = \ & [i \otimes T_v + \alpha(j + l \cdot P) + \beta(j + l \cdot P)]\,\%2^p \\ = \ & [i \otimes T_v + \alpha(j) + \beta(j) + l \cdot \tau]\,\%2^p \end{aligned}$$

To make $m_v(i + k \cdot 2^{p+vs},\ j + l \cdot P) = m_v(i,j)$, it should be satisfied that

$$[i \otimes T_v + \alpha(j) + \beta(j) + l \cdot \tau]\,\%2^p = [i \otimes T_v + \alpha(j) + \beta(j)]\,\%2^p$$

which means that $(l \cdot \tau)\%2^p = 0$ for $\forall\, l$. So we know $\tau = 2^p$ and $P = 2^{p+q+hs}$, and the period of $(m_v(i,j),\ m_h(j))$ is $2^{p+vs} \times 2^{p+q+hs}$. ■

---

[1] As we could see from the definition, the basic XOR scheme is completely determined by the address transformation matrix T. So we could refer to an XOR scheme by just giving the name of the transformation matrix.

Lemma 1 Elements of the same column in the 2D address space are assigned to modules with the same horizontal module index.

Proof. All elements of a column in the 2D address space have the same horizontal address $j = j_0$. So they are assigned to the same $m_h(j_0)$, from the definition of m_h. ■

Lemma 2 Every column could be accessed in parallel for 2^p strided accesses with stride $VS = \sigma_v \cdot 2^{vs}(\sigma_v$ is any arbitrary odd number).

Proof. For any column $j = j_0$, the 2^p addresses of the strided accesses are $\{(b_v, j_0), (b_v + VS, j_0), \dots, (b_v + (2^p - 1) \cdot VS, j_0)\}$. According to Lemma 1 they are assigned to the same m_h. Now consider m_v. According to Property 1 the accesses are assigned to different modules under the basic XOR scheme T_v, i.e. $b_v \otimes T_v$, $(b_v + VS) \otimes T_v, \dots, (b_v + (2^p - 1)VS) \otimes T_v$ are different. Moreover, α and β are fixed with given j_0, so $b_v \otimes T_v + \alpha + \beta$, $(b_v + VS) \otimes T_v + \alpha + \beta, \dots, (b_v + (2^p - 1)VS) \otimes T_v + \alpha + \beta$ are different. In effect, all 2^p accesses are assigned to 2^p different vertical module indexes, so they could be accessed in parallel. ■

Lemma 3 Every column could be accessed in parallel for 2^p unit-stride accesses, under the condition that the vertical base address is at the boundaries of $\{\tau \cdot 2^{\min(p,vs)} \| \tau \in \mathbb{N}\}$.

Proof. Proof of Lemma 3 is much like the proof of Lemma 2. As the horizontal coordinates(j) of all elements in the same column are the same, m_v of the 2DSMM scheme is equal to the module assignment m'(denote $m' = i \otimes T_v$) of the basic XOR scheme T_v plus a constant value $\alpha + \beta$, i.e. $m_v = (m' + \alpha + \beta)\%2^p$. According to Property 2, the address sequence of the access are assigned to different m', therefore they are assigned to different m_v. In effect, all 2^p accesses are assigned to 2^p different vertical module indexes, so they could be accessed in parallel. ■

Lemma 4 Any 2^q unit-stride row accesses[2] whose horizontal base address is at the boundaries of $\{\tau \cdot 2^q \| \tau \in \mathbb{N}\}$ are assigned to modules with the same vertical module index.

---

[2] The expressions "unit-stride row access" and "continuous row access" would be used alternatively in this report for the same concept.

Proof. The address sequence of the row access is $\{(i_0, \tau \cdot 2^q), (i_0, \tau \cdot 2^q + 1), \ldots, (i_0, (\tau+1) \cdot 2^q - 1)\}$. Consider $m_v(i_0, \tau \cdot 2^q + k) \quad (0 \leq k \leq 2^q - 1)$.

$$\begin{aligned}
\alpha &= \left(\frac{\tau \cdot 2^q + k}{2^{q+hs}}\right) \%2^p = \left(\frac{\tau}{2^{hs}}\right) \%2^p \\
\beta &= \left(\frac{\tau \cdot 2^q + k}{2^q} \cdot 2^{p-\min(p,hs)}\right) \%2^p \\
&= (\tau \cdot 2^{p-\min(p,hs)}) \%2^p
\end{aligned}$$

We could see that neither α nor β is related to k, which means that $m_v(i_0, \tau \cdot 2^q + k) = \left[\left(i_0 + (\alpha+\beta) \cdot 2^{vs}\right) \otimes T_v\right] \%2^p$ is not related to k. Therefore address sequence $\{(i_0, \tau \cdot 2^q), (i_0, \tau \cdot 2^q + 1), \ldots, (i_0, \tau \cdot 2^q + 2^q - 1)\}$ are assigned to modules with the same vertical module index. ■

Theorem 1 Every row could be accessed in parallel for 2^{p+q} accesses with stride $HS = \sigma_h \cdot 2^{hs}$($\sigma_h$ is any arbitrary odd number).

Proof. For any row $i = i_0$, the address sequence of the strided row access is $\{(i_0,\ b_h), (i_0,\ b_h + HS), \ldots, (i_0,\ (2^{p+q} - 1) \cdot HS)\}$($b_h$ is the horizontal base address).

i)Address sequence $\{(i_0,\ b'_h), (i_0,\ b'_h + HS), \ldots, (i_0,\ b'_h + (2^q - 1) \cdot HS)\}$($b'_h$ is the horizontal base address) are assigned to 2^q modules with different m_h. This is true according to Property 1 of T_h.

ii)Address sequence $\{(i_0,\ b_h + k \cdot HS), (i_0,\ b_h + k \cdot HS + 2^q \cdot HS), \ldots, (i_0,\ b_h + k \cdot HS + (2^p - 1) \cdot 2^q \cdot HS)\}(0 \leq k \leq 2^q - 1)$ are assigned to 2^q modules with the same m_h but different m_v.
Consider m_h. It is periodic with period of 2^{q+hs} according to Property 3. Note $2^q \cdot HS = 2^q \cdot \sigma_h \cdot 2^{hs}$, so the elements of the address sequence have an interval which is multiple(σ_h times) of the period of m_h. So they are assigned to modules with the same horizontal index by m_h. Now consider $m_v(i_0, b_h +$

$k \cdot HS + l \cdot 2^q \cdot HS)$.

$$
\begin{aligned}
\alpha &= \left(\frac{b_h + k \cdot HS + l \cdot 2^q \cdot \sigma_h \cdot 2^{hs}}{2^{q+hs}}\right) \% 2^p \\
&= (B_k + l \cdot \sigma_h) \% 2^p \qquad \left(\text{Denote } B_k = \frac{b_h + k \cdot HS}{2^{q+hs}}\right) \\
\beta &= \left(\frac{b_h + k \cdot HS + l \cdot 2^q \cdot HS}{2^q} \cdot 2^{p-\min(p,hs)}\right) \% 2^p \\
&= \left((C_k + l \cdot \sigma_h \cdot 2^{hs}) \cdot 2^{p-\min(p,hs)}\right) \% 2^p \\
& \qquad \left(\text{Denote } C_k = \frac{b_h + k \cdot HS}{2^q}\right) \\
&= \left(C_k \cdot 2^{p-\min(p,hs)}\right) \% 2^p \\
&= D_k \qquad \left(\text{Denote } D_k = \left(C_k \cdot 2^{p-\min(p,hs)}\right) \% 2^p\right)
\end{aligned}
$$

So we have

$$
\begin{aligned}
& m_v(i_0,\ b_h + k \cdot HS + l \cdot 2^q \cdot HS) \\
= \ & (i_0 \otimes T_v + \alpha + \beta) \% 2^p \\
= \ & (i_0 \otimes T_v + B_k + D_k + l \cdot \sigma_h) \% 2^p
\end{aligned}
$$

It is obvious that for any fixed k, m_v is different for each $0 \le l \le 2^p - 1$ with odd σ_h.

Now consider the entire address sequence of the 2^{p+q} accesses, enumerated in the address matrix in Figure B.1(only the horizontal address of the address pair is shown because the vertical address is the same for all the accesses). Suppose any two elements of the access sequence are assigned to modules (m'_v, m'_h) and (m''_v, m''_h), respectively. Assume they are assigned to the same module, i.e. $m'_v = m''_v$ and $m'_h = m''_h$. According to $i)$ and $ii)$, $m'_h = m''_h$ could be possible

$$
\begin{array}{cccc}
b_h & b_h + HS & \ldots & b_h + (2^q - 1) \cdot HS \\
b_h + 2^q \cdot HS & b_h + (2^q + 1) \cdot HS & \ldots & b_h + (2^{q+1} - 1) \cdot HS \\
\vdots & \vdots & \ldots & \vdots \\
b_h + (2^p - 1) \cdot 2^q \cdot HS & b_h + ((2^p - 1) \cdot 2^q + 1) \cdot HS & \ldots & b_h + (2^{p+q} - 1) \cdot HS
\end{array}
$$

Figure B.1: Address sequence of strided row access

only when they are in the same column of the address matrix. In this case, $m'_v = m''_v$ could be satisfied only when the two are the same, according to ii). That is to say, $m'_v = m''_v$ and $m'_h = m''_h$ could never happen at the same time for two different accesses in the access stream. This proves Theorem 1. ■

Theorem 2 Every row could be accessed in parallel for 2^{p+q} unit-stride accesses, under the condition that the horizontal base address is at the boundaries of $\{\tau \cdot 2^{q+\min(p,hs)} \| \tau \in \mathbb{N}\}$.

Proof. For any row $i = i_0$, the address sequence of the unit-stride row access is $\{(i_0,\ \tau \cdot 2^{q+\min(p,hs)}), (i_0,\ \tau \cdot 2^{q+\min(p,hs)} + 1), \dots, (i_0,\ \tau \cdot 2^{q+\min(p,hs)} + 2^{p+q} - 1)\}(\tau \in \mathbb{N})$.

i)Address sequence $\{(i_0,\ \delta \cdot 2^q), (i_0,\ \delta \cdot 2^q + 1), \dots, (i_0,\ \delta \cdot 2^q + 2^q - 1)\}(\delta \in \mathbb{N})$ are assigned to 2^q modules with the same m_v but different m_h. According to Lemma 4, the accesses are assigned to modules with the same vertical module indexes. As the base address is at the boundaries of $\{\tau \cdot 2^q \| \tau \in \mathbb{N}\}$, the access sequence are assigned to 2^q different horizontal modules, i.e. different m_h, according to Property 2.

ii)Address sequence $\{(i_0, \tau \cdot 2^{p+q} + k), (i_0, \tau \cdot 2^{p+q} + k + 2^q), \dots, (i_0, \tau \cdot 2^{p+q} + k + (2^p - 1) \cdot 2^q)\}(0 \leq k \leq 2^q - 1)$ are assigned to 2^p modules with different m_v.

Consider $m_v(i_0, \tau \cdot 2^{p+q} + k + l \cdot 2^q)(0 \leq l \leq 2^p - 1)$.

$$\begin{aligned}
\alpha &= \left(\frac{\tau \cdot 2^{p+q} + k + l \cdot 2^q}{2^{q+hs}}\right)\%2^p \\
&= \left(\frac{\tau \cdot 2^p + l}{2^{hs}}\right)\%2^p \\
\beta &= \left(\frac{\tau \cdot 2^{p+q} + k + l \cdot 2^q}{2^q} \cdot 2^{p-\min(p,hs)}\right)\%2^p
\end{aligned}$$

We could see that neither α nor β is related to k.

$$(\alpha + \beta)\%2^p = \begin{cases} \left(\frac{\tau}{2^{hs-p}} + l\right)\%2^p & p \leq hs \\ \left(\tau \cdot 2^{p-hs} + \frac{l}{2^{hs}} + (l\%2^{hs}) \cdot 2^{p-hs}\right)\%2^p & p > hs \end{cases}$$

When $p \leq hs$, it is clear that $(\alpha + \beta)\%2^p$ traverses $\mathbb{E}^q$[3] when l traverses $\mathbb{E}^q$ with fixed τ. Now let's consider the case $p > hs$. By examining the structure of $\frac{l}{2^{hs}} + (l\%2^{hs}) \cdot 2^{p-hs}$, it could be noticed that as $p > hs$ and $0 \leq l \leq 2^p - 1$,

---

[3] Denote $\mathbb{E}^q = \{0, 1, \dots, 2^p - 1\}$.

it moves the highest $p - hs$ bits of l to the lowest $p - hs$ bits(as the effect of $\frac{l}{2^{hs}}$), and moves the lowest hs bits to the highest hs bits(as the effect of $(l\%2^{hs}) \cdot 2^{p-hs}$). So when l traverses $\mathbb{E}^q$, $\frac{l}{2^{hs}} + (l\%2^{hs}) \cdot 2^{p-hs}$ also traverses $\mathbb{E}^q$. As τ is fixed, we know $(\alpha + \beta)\%2^p$ traverses $\mathbb{E}^q$ in both cases of $p \leq hs$ and $p > hs$. Therefore $m_v(i_0,\ \tau \cdot 2^{p+q} + k + l \cdot 2^q) = (i_0 \otimes T_v + \alpha + \beta)\,\%2^p$ traverses $\mathbb{E}^q$, for $0 \leq l \leq 2^q - 1$. This proves $ii)$.

$iii)$Address sequence $\{(i_0, \tau \cdot 2^{q+hs} + k), (i_0, \tau \cdot 2^{q+hs} + k + 2^q), \ldots, (i_0, \tau \cdot 2^{q+hs} + k + (2^p - 1) \cdot 2^q)\}(0 \leq k \leq 2^q - 1)$ are assigned to 2^p modules with different m_v.
Consider $m_v(i_0, \tau \cdot 2^{q+hs} + k + l \cdot 2^q)(0 \leq l \leq 2^p - 1)$.

$$\begin{aligned}
\alpha &= \left(\frac{\tau \cdot 2^{q+hs} + k + l \cdot 2^q}{2^{q+hs}}\right)\%2^p \\
&= \left(\tau + \frac{l}{2^{hs}}\right)\%2^p \\
\beta &= \left(\frac{\tau \cdot 2^{q+hs} + k + l \cdot 2^q}{2^q} \cdot 2^{p-\min(p,hs)}\right)\%2^p \\
&= \left((\tau \cdot 2^{hs} + l) \cdot 2^{p-\min(p,hs)}\right)\%2^p \\
&= \begin{cases} l\%2^p & p \leq hs \\ (\tau \cdot 2^p + l \cdot 2^{p-hs})\%2^p & p > hs \end{cases}
\end{aligned}$$

So we have

$$(\alpha + \beta)\%2^p = \begin{cases} (\tau + l)\%2^p & p \leq hs \\ \left(\tau + \frac{l}{2^{hs}} + (l\%2^{hs}) \cdot 2^{p-hs}\right)\%2^p & p > hs \end{cases}$$

We could see the structure of $(\alpha + \beta)\%2^p$ remains the same as the case of $ii)$ with respect to l. Therefore the analysis in $ii)$ also applies here, and we know $iii)$ is true.

With the results of $ii)$ and $iii)$, it is obvious that address sequence $\{(i_0, \tau \cdot 2^{q+\min(p,hs)} + k), (i_0, \tau \cdot 2^{q+\min(p,hs)} + k + 2^q), \ldots, (i_0, \tau \cdot 2^{q+\min(p,hs)} + k + (2^p - 1) \cdot 2^q)\}(0 \leq k \leq 2^q - 1)$ are assigned to 2^p modules with different m_v. Now consider the entire address sequence of the 2^{p+q} accesses, enumerated in the address matrix in Figure B.2(only the horizontal address of the address pair is shown because the vertical address is the same for all the accesses). According to $ii)$ and $iii)$, different rows are assigned to modules with different m_v, and elements in each row are assigned to modules with different m_h according to $i)$. So it is clear that all the 2^{p+q} unit-stride accesses are assigned to different memory modules. ■

$$\begin{array}{cccc}
\tau\cdot 2^{q+\min(p,hs)} & \tau\cdot 2^{q+\min(p,hs)}+1 & \ldots & \tau\cdot 2^{q+\min(p,hs)}+2^q-1 \\
\tau\cdot 2^{q+\min(p,hs)}+2^q & \tau\cdot 2^{q+\min(p,hs)}+2^q+1 & \ldots & \tau\cdot 2^{q+\min(p,hs)}+2^{q+1}-1 \\
\vdots & \vdots & \ldots & \vdots \\
\tau\cdot 2^{q+\min(p,hs)}+(2^p-1)\cdot 2^q & \tau\cdot 2^{q+\min(p,hs)}+(2^p-1)\cdot 2^q+1 & \ldots & \tau\cdot 2^{q+\min(p,hs)}+2^{p+q}-1
\end{array}$$

Figure B.2: Address sequence of unit-stride row access

Theorem 3 Every forward diagonal could be accessed in parallel for 2^{p+q} accesses with vertical stride $VS = \sigma_v \cdot 2^{vs}$ and horizontal stride $HS = \sigma_h \cdot 2^{hs}$($\sigma_v$ and σ_h are any arbitrary odd numbers).

Proof. The address sequence of strided forward diagonal access is $\{(b_v, b_h), (b_v + VS, b_h + HS), \ldots, (b_v + (2^{p+q} - 1) \cdot VS, b_h + (2^{p+q} - 1)] \cdot HS\}$($b_v$, b_h are vertical and horizontal base addresses respectively).

i)Addresses $\{(b'_v, b'_h), (b'_v + VS, b'_h + HS), \ldots, (b'_v + (2^q - 1) \cdot VS, b'_h + (2^q - 1)] \cdot HS\}$($b'_v$, b'_h are vertical and horizontal base addresses respectively) are assigned to modules with different m_h. This is true according to Property 1 of T_h.

ii)Addresses $\{(b_v + k \cdot VS, b_h + k \cdot HS), (b_v + (k + 2^q) \cdot VS, b_h + (k + 2^q) \cdot HS), \ldots, (b_v + (k + (2^p - 1) \cdot 2^q) \cdot VS, b_h + (k + (2^p - 1) \cdot 2^q) \cdot HS)\}(0 \le k \le 2^q - 1)$ are assigned to modules with the same m_h but different m_v. Consider m_h. It is periodic with period of 2^{q+hs} according to Property 3 of T_h. Note $2^q \cdot HS = 2^q \cdot \sigma_h \cdot 2^{hs}$, which means the interval of the address sequence is multiple(σ_h times) of the period of m_h. So the accesses are assigned to modules with the same horizontal index. Now consider $m_v(b_v + (k + l \cdot 2^q) \cdot VS, b_h + (k + l \cdot 2^q) \cdot HS)(0 \le l \le 2^p - 1)$.

$$\begin{aligned}
\alpha &= \left(\frac{b_h + (k + l \cdot 2^q) \cdot \sigma_h \cdot 2^{hs}}{2^{q+hs}}\right) \% 2^p \\
&= (B_k + l \cdot \sigma_h) \% 2^p \qquad \left(\text{Denote } B_k = \frac{b_h + k \cdot \sigma_h \cdot 2^{hs}}{2^{q+hs}}\right) \\
\beta &= \left(\frac{b_h + (k + l \cdot 2^q) \cdot \sigma_h \cdot 2^{hs}}{2^q} \cdot 2^{p-\min(p,hs)}\right) \% 2^p \\
&= C_k \quad (\text{Denote } C_k = \left(\frac{b_h + k \cdot \sigma_h \cdot 2^{hs}}{2^q} \cdot 2^{p-\min(p,hs)}\right) \% 2^p)
\end{aligned}$$

So we have

$$\begin{aligned} & m_v(b_v + (k + l \cdot 2^q) \cdot VS,\ b_h + (k + l \cdot 2^q) \cdot HS) \\ = \ & [(b_v + k \cdot VS + l \cdot 2^q \cdot \sigma_v \cdot 2^{vs}) \otimes T_v + B_k + C_k + l \cdot \sigma_h] \,\%2^p \end{aligned}$$

Note the factor $l \cdot 2^q \cdot \sigma_v \cdot 2^{vs}$. Under the general constraint $p \leq q$ of 2DSMM scheme, it is the multiple of 2^{p+vs}, the period of T_v according to Property 4. So we have

$$\beta = [(b_v + k \cdot VS) \otimes T_v + B_k + C_k + l \cdot \sigma_h] \,\%2^p$$

from which we could see that m_v traverses $\mathbb{E}^p$ when l traverses $\mathbb{E}^p$, with any fixed k and odd σ_h. This means that the elements of address sequence in ii) are assigned to modules with different vertical indexes.

Now consider the entire address sequence of the 2^{p+q} accesses, enumerated in the address matrix in Figure B.3. Suppose two elements of the access sequence are assigned to modules (m'_v, m'_h) and (m''_v, m''_h), respectively. According to i) and ii), $m'_h = m''_h$ could be possible only when they are in the same column of the address matrix. In this case, $m'_v = m''_v$ could happen only when they are the same, according to ii). In consequence, $m'_v = m''_v$ and $m'_h = m''_h$ could never happen at the same time for two different accesses in the access stream. This justifies Theorem 3. ■

$$\begin{array}{cccc}
\begin{pmatrix} b_v, \\ b_h \end{pmatrix} & \begin{pmatrix} b_v + VS, \\ b_h + HS \end{pmatrix} & \cdots & \begin{pmatrix} b_v + (2^q - 1) \cdot VS, \\ b_h + (2^q - 1) \cdot HS \end{pmatrix} \\
\begin{pmatrix} b_v + 2^q \cdot VS, \\ b_h + 2^q \cdot HS \end{pmatrix} & \begin{pmatrix} b_v + (2^q + 1) \cdot VS, \\ b_h + (2^q + 1) \cdot HS \end{pmatrix} & \cdots & \begin{pmatrix} b_v + (2^{q+1} - 1) \cdot VS, \\ b_h + (2^{q+1} - 1) \cdot HS \end{pmatrix} \\
\vdots & \vdots & \cdots & \vdots \\
\begin{pmatrix} b_v + (2^p - 1) \cdot 2^q \cdot VS, \\ b_h + (2^p - 1) \cdot 2^q \cdot HS \end{pmatrix} & \begin{pmatrix} b_v + ((2^p - 1) \cdot 2^q + 1) \cdot VS, \\ b_h + ((2^p - 1) \cdot 2^q + 1) \cdot HS \end{pmatrix} & \cdots & \begin{pmatrix} b_v + (2^{p+q} - 1) \cdot VS, \\ b_h + (2^{p+q} - 1) \cdot HS \end{pmatrix}
\end{array}$$

Figure B.3: Address sequence of strided forward diagonal access

Theorem 4 Every backward diagonal could be accessed in parallel for 2^{p+q} accesses with vertical stride $VS = \sigma_v \cdot 2^{vs}$ and horizontal stride $HS = \sigma_h \cdot 2^{hs}$($\sigma_v$ and σ_h are any arbitrary odd numbers).

Proof. The address sequence of backward diagonal access is $\{(b_v, b_h), (b_v + VS, b_h - HS), \ldots, (b_v + (2^{p+q} - 1) \cdot VS, b_h - (2^{p+q} - 1) \cdot HS\}$$(b_v, b_h$ are vertical and horizontal base addresses respectively, and $b_h \geq (2^{p+q} - 1) \cdot HS)$. There is no essential difference between this sequence and the sequence generated by forward diagonal access, and it is easy to verify that the proof of Theorem 3 also holds for Theorem 4. ■

Theorem 5 Every strided block with the size of $2^p \times 2^q$ in which the intervals of the elements are $VS = \sigma_v \cdot 2^{vs}$ in vertical and $HS = \sigma_h \cdot 2^{hs}$ in horizontal could be accessed in parallel(σ_v and σ_h are any arbitrary odd numbers).

Proof. We know all elements of a column in the strided block are assigned to modules with the same m_h(Lemma 1), and different column holds different m_h(Property 1 of T_h). So if two different addresses in the strided block are assigned to modules with the same m_h, they should be in the same column. Once two accesses are in the same column, they could never have the same m_v because Lemma 2 indicates that all elements of a column in the strided block are assigned to modules with different m_v. So all elements of the accessed block are assigned to different modules, which means that they could be accessed in parallel. ■

Theorem 6 Every continuous block with the size of $2^p \times 2^q$ could be accessed in parallel, under the condition that the vertical base address is at the boundaries of $\{\tau \cdot 2^{\min(p,vs)} \| \tau \in \mathbb{N}\}$ and the horizontal vertical address is at the boundaries of $\{\tau \cdot 2^{\min(q,hs)} \| \tau \in \mathbb{N}\}$.

Proof. Each column of the accessed block is assigned to the same m_h(Lemma 1), and different column is assigned to different m_h when the horizontal base address is at the boundaries of $\{\tau \cdot 2^{\min(q,hs)} \| \tau \in \mathbb{N}\}$(Property 2 of T_h). Moreover, each element of the column in the accessed block is assigned to a different m_v when the vertical base address is at the boundaries of $\{\tau \cdot 2^{\min(p,vs)} \| \tau \in \mathbb{N}\}$(Property 2 of T_v). So all elements of the accessed block are assigned to different modules and they could be accessed in parallel. ■

Theorem 6 additionally justifies the row assignment function $r(a)$ of the 2DSMM scheme because it guarantees that all the elements which are assigned to the same row address by $r(a)$ are assigned to different memory modules.

Theorem 7 When the access sequence are arranged in $2^p \times 2^q$ array in row-major manner, for all the strided access patterns and the continuous block access, all accesses in same column are assigned to same m_h; For continuous row access, all accesses in same row are assigned to same m_v.

Proof. When the access sequence are arranged in $2^p \times 2^q$ array in row-major manner, for the strided block access and the continuous block access, the accesses in same column have the same horizontal address j, so they are assigned to the same m_h, according to Lemma 1; For the strided row, strided forward diagonal and strided backward diagonal access patterns, any two accesses in the same column have their horizontal addresses related in the form of $j_1 = j_0 + k \cdot 2^q \cdot HS = j_0 + k \cdot \sigma_h \cdot 2^{q+hs}\,(k \in \mathbb{N})$. According to Property 4, $m_h(j_0) = m_h(j_1)$. That is to say, in all the strided access patterns and the continuous block access, accesses in the same column are assigned to the same m_h.
Now consider the continuous row access. From Theorem 2 we know the horizontal base address is at the boundaries of $\{\tau \cdot 2^{q+\min(p,hs)} \| \tau \in \mathbb{N}\}$. Therefore when the access sequence are arranged in $2^p \times 2^q$ array in row-major manner, each row will start at the horizontal boundaries of $\{\tau \cdot 2^q \| \tau \in \mathbb{N}\}$. According to Lemma 4, the elements of the same row will be assigned to the same m_v. ■

Theorem 7 reveals an opportunity to exploit the inherent regularities within module assignment functions of the six access patterns to simplify the address and data switching circuitry. We have already seen this in the 2DSMM implementation presented in Section 3.4.2.

Note. The contents of this appendix is based on the the following report:

C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **2DSMM: 2D Strided Multi-access Memory**, CE Technical Report, Computer Engineering Lab, TU Delft, July 2007

Bibliography

[1] GPGPU home page. http://gpgpu.org.

[2] http://en.wikipedia.org/wiki/3dfx_interactive.

[3] http://pcsostres.ac.upc.edu/cellsim/doku.php.

[4] http://www.encore-project.eu/.

[5] http://www.ibm.com/developerworks/power/cell/.

[6] http://www.top500.org/.

[7] Intel Sandy Bridge, Intel processor roadmap, 2010.

[8] OpenCL home page. http://www.khronos.org/opencl/.

[9] 3DNOW TECHNOLOGY MANUAL, (2000).

[10] ADL-TABATABAI, A.-R., HUDSON, R. L., SERRANO, M. J., AND SUBRAMONEY, S. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation* (New York, NY, USA, 2004), PLDI '04, ACM, pp. 267–276.

[11] ALVAREZ, M., SALAMI, E., RAMIREZ, A., AND VALERO, M. Performance impact of unaligned memory operations in SIMD extensions for video codec applications. In *ISPASS '07: Proceedings of the 2007 International Symposium on Performance Analysis of Systems and Software* (Los Alamitos, CA, USA, 2007), vol. 0, IEEE Computer Society, pp. 62–71.

[12] AMD. http://sites.amd.com/us/fusion/apu/pages/fusion.aspx.

[13] ARIEL, A., FUNG, W., TURNER, A., AND AAMODT, T. Visualizing complex dynamics in many-core accelerator architectures. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (2010), pp. 164 –174.

[14] BADER, D., AGARWAL, V., AND MADDURI, K. On the design and analysis of irregular algorithms on the Cell processor: A case study of list ranking, march 2007.

[15] BAILY, D. Vector computer memory bank contention. *IEEE Trans. Computers 36* (Mar. 1987), 293–298.

[16] BAKHODA, A., KIM, J., AND AAMODT, T. M. Throughput-effective on-chip networks for manycore accelerators. In *Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 421–432.

[17] BAKHODA, A., YUAN, G., FUNG, W., WONG, H., AND AAMODT, T. Analyzing CUDA workloads using a detailed GPU simulator, 2009.

[18] BATTEN, C., KRASHINSKY, R., GERDING, S., AND ASANOVIC, K. Cache refill/access decoupling for vector machines. In *Proceedings of the 37th annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2004), MICRO 37, IEEE Computer Society, pp. 331–342.

[19] BHATNAGAR, H. *Advanced ASIC Chip Synthesis Using Synopsys Design Compiler Physical Compiler and PrimeTime*, 2nd ed. Kluwer Academic Publishers, 2001.

[20] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), ACM, pp. 72–81.

[21] BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), VLDB '99, Morgan Kaufmann Publishers Inc., pp. 54–65.

[22] BUDNIK, P., AND KUCK, D. The organization and use of parallel memories. *IEEE Trans. Computers C, 20* (Dec. 1971), 1566–1569.

[23] CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1991), ACM, pp. 40–52.

[24] CARTER, J., HSIEH, W., STOLLER, L., SWANSON, M., ZHANG, L., BRUNVAND, E., DAVIS, A., KUO, C.-C., KURAMKOTE, R., PARKER, M., SCHAELICKE, L., AND TATEYAMA, T. Impulse: Building a smarter memory controller. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 70–79.

[25] CHANG, H.-C., LIN, C.-C., AND GUO, J.-I. A novel low-cost high-performance VLSI architecture for MPEG-4 AVC/H.264 CAVLC decoding. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on* (May 2005), pp. 6110 – 6113 Vol. 6.

[26] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium 0* (2009), 44–54.

[27] CHEN, T.-F., AND BAER, J.-L. A performance study of software and hardware data prefetching schemes. In *ISCA '94: Proceedings of the 21st annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 223–232.

[28] CHEN, Y.-K., CHHUGANI, J., DUBEY, P., HUGHES, C., KIM, D., KUMAR, S., LEE, V., NGUYEN, A., AND SMELYANSKIY, M. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE 96*, 5 (May 2008), 790 –807.

[29] CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 13–24.

[30] CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. Cache-conscious structure layout. *SIGPLAN Not. 34* (May 1999), 1–12.

[31] CHOR, B., LEISERSON, C. E., RIVEST, R. L., AND SHEARER, J. B. An application of number theory to the organization of raster-graphics memory. *J. ACM 33* (January 1986), 86–104.

[32] CORBAL, J., ESPASA, R., AND VALERO, M. Command vector memory systems: High performance at low cost. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 68–77.

[33] CORP., C. D. CDC STAR-100 instruction execution times, rev. 2.

[34] D. T. HARPER III. Block, multistride vector and FFT accesses in parallel memory systems. *IEEE Trans. Parallel and Distributed Systems 2*, 1 (1991), 43–51.

[35] D. T. HARPER III. Increased memory performance during vector accesses through the use of linear address transformations. *IEEE Trans. Computers 41*, 2 (1992), 227–230.

[36] D. T. HARPER III, AND LINEBARGER, D. A. Conflict-free vector access using a dynamic storage scheme. *IEEE Trans. Computers 40*, 3 (1991), 276–283.

[37] DALLY, W. J. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[38] DHONG, S., TAKAHASHI, O., WHITE, M., ASANO, T., NAKAZATO, T., SILBERMAN, J., KAWASUMI, A., AND YOSHIHARA, H. A 4.8GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor. In *Proceedings of IEEE Int'l Solid-State Circuits Conference 2005* (2005), pp. 486–612.

[39] DIAMOS, G. F., KERR, A. R., YALAMANCHILI, S., AND CLARK, N. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 353–364.

[40] DIEFENDORFF, K., DUBEY, P. K., HOCHSPRUNG, R., AND SCALES, H. Altivec extension to powerpc accelerates media processing. *IEEE Micro 20*, 2 (2000), 85–95.

[41] DUBEY, P. Recognition, mining and synthesis moves computers to the era of tera. *Technology Intel Magazine* (Feb. 2005).

[42] ESPASA, R., VALERO, M., AND SMITH, J. E. Vector architectures: past, present and future. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing* (New York, NY, USA, 1998), ACM, pp. 425–432.

[43] FATAHALIAN, K., KNIGHT, T. J., HOUSTON, M., EREZ, M., HORN, D. R., LEEM, L., PARK, J. Y., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006).

[44] FLACHS, B., ASANO, S., DHONG, S., HOTSTEE, P., GERVAIS, G., KIM, R., LE, T., LIU, P., LEENSTRA, J., LIBERTY, J., MICHAEL, B., OH, H., MUELLER, S., TAKAHASHI, O., HATAKEYAMA, A., WATANABE, Y., AND YANO, N. A streaming processing unit fors a Cell processor. In *Proceedings of IEEE Int'l Solid-State Circuits Conference 2005* (2005), pp. 134–135.

[45] FLYNN, M. J. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on C-21*, 9 (1972), 948 –960.

[46] FORSYTH, T. SIMD programming with Larrabee. *http://software.intel.com/file/15545*.

[47] FUNG, W. W. L., SHAM, I., YUAN, G., AND AAMODT, T. M. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 407–420.

[48] GAO, Q. S. The Chinese remainder theorem and the prime memory system. In *Proceedings of the 20th annual International Symposium on Computer Architecture* (New York, NY, USA, 1993), ISCA '93, ACM, pp. 337–340.

[49] Gedik, B., Bordawekar, R. R., and Yu, P. S. Cellsort: high performance sorting on the Cell processor. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB '07, VLDB Endowment, pp. 1286–1297.

[50] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W.-m. W. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 347–358.

[51] Glaskowsky, P. N. Nvidia's Fermi: The first complete GPU computing architecture.

[52] Golovanevsky, O., and Zaks, A. Struct-reorg: Current status and future perspectives. In *GCC Summit Proceedings* (2007).

[53] Gou, C., and Gaydadjiev, G. Exploiting SPMD horizontal locality to improve memory efficiency. *IEEE Computer Architecture Letters 99*, RapidPosts (2011).

[54] Gou, C., and Gaydadjiev, G. N. Elastic pipeline: addressing gpu on-chip shared memory bank conflicts. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (New York, NY, USA, 2011), CF '11, ACM, pp. 3:1–3:11.

[55] Gou, C., Kuzmanov, G., and Gaydadjiev, G. N. SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 179–188.

[56] Gou, C., Kuzmanov, G. K., and Gaydadjiev, G. N. SAMS: single-affiliation multiple-stride parallel memory scheme. In *Proceedings of the 2008 Workshop on Memory Access on future processors: a solved problem?* (New York, NY, USA, 2008), MAW '08, ACM, pp. 350–368.

[57] Gschwind, M., Hofstee, H. P., Flachs, B., Hopkins, M., Watanabe, Y., and Yamazaki, T. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro 26*, 2 (2006), 10–24.

[58] Hammond, S. W., Loft, R. D., and Tannenbaum, P. D. Architectural and application: the performance of the NEC SX-4 on the NCAR benchmark suite. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1996), IEEE Computer Society, p. 22.

[59] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[60] Hsu, W.-C., and Smith, J. Performance of cached DRAM organizations in vector supercomputers. *ACM SIGARCH Computer Architecture News 21* (May 1993), 327–336.

[61] IBM Systems and Technology Group. Cell BE programming tutorial v3.0. *http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788*.

[62] IBM Systems and Technology Group. Developing code for Cell - SIMD. *www.cc.gatech.edu/˜bader/cell/Day1-06_DevelopingCodeforCell-SIMD.ppt*.

[63] Ibrahim, K. Z., and Bodin, F. Implementing Wilson-Dirac operator on the Cell Broadband Engine. In *ICS '08: Proceedings of the 22nd annual International Conference on Supercomputing* (New York, NY, USA, 2008), ACM, pp. 4–14.

[64] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. No. 253669-033US. December 2009.

[65] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. http://www.jedec.org.

[66] JORDA, J., MZOUGHI, A., AND LITAIZE, D. Semi-linear and bi-base storage schemes classes: general overview and case study. In *Proceedings of the 9th International Conference on Supercomputing* (New York, NY, USA, 1995), ICS '95, ACM, pp. 299–307.

[67] JOUPPI, N. P. *Destination indexed miss status holding registers*. US patent, 1998.

[68] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development 49*, 4.5 (2005), 589 –604.

[69] KELM, J. H., JOHNSON, D. R., JOHNSON, M. R., CRAGO, N. C., TUOHY, W., MAHESRI, A., LUMETTA, S. S., FRANK, M. I., AND PATEL, S. J. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *SIGARCH Comput. Archit. News 37* (June 2009), 140–151.

[70] KELTCHER, C., MCGRATH, K., AHMED, A., AND CONWAY, P. The AMD Opteron processor for multiprocessor servers. *IEEE Micro* (Mar. 2003), 66–76.

[71] KIM, D., CHAUDHURI, M., HEINRICH, M., AND SPEIGHT, E. Architectural support for uniprocessor and multiprocessor active memory systems. *IEEE Trans. Computers 53*, 3 (2004), 288–307.

[72] KIM, K., AND PRASANNA-KUMAR, V. K. Perfect latin squares and parallel array access. In *Proceedings of the 16th annual International Symposium on Computer Architecture* (New York, NY, USA, 1989), ISCA '89, ACM, pp. 372–379.

[73] KIM, Y., HAN, D., MUTLU, O., AND HARCHOL-BALTER, M. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (jan. 2010), pp. 1 –12.

[74] KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 65–76.

[75] KIRK, D. B., AND MEI W. HWU), W. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[76] KIRK, D. B., AND MEI W. HWU, W. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[77] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro* (Mar. 2005), 21–29.

[78] KONTOTHANASSIS, L., SUGUMAR, R., FAANES, G., SMITH, J., AND SCOTT, M. Cache performance in vector supercomputers. In *Proceedings of the ACM/IEEE Conference on Supercomputing 1994* (1994), pp. 255–264.

[79] KOZYRAKIS, C. *Scalable Vector Media-Processors for Embedded Systems*. PhD thesis, UC Berkeley, Berkeley, CA, USA, May 2002.

[80] KRASHINSKY, R., BATTEN, C., HAMPTON, M., GERDING, S., PHARRIS, B., CASPER, J., AND ASANOVIC, K. The vector-thread architecture. In *Proceedings of the 31st annual International Symposium on Computer Architecture* (Washington, DC, USA, 2004), ISCA '04, IEEE Computer Society, pp. 52–.

[81] KUCK, D., AND STOKES, R. The Burroughs scientific processor (BSP). *Computers, IEEE Transactions on C-31*, 5 (may 1982), 363 –376.

[82] KUCK, D. J. ILLIAC IV software and application programming. *IEEE Trans. Comput. 17* (August 1968), 758–770.

[83] KUCK, D. J. A survey of parallel machine organization and programming. *ACM Comput. Surv. 9* (March 1977), 29–59.

[84] KUMAR, V. *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[85] KUZMANOV, G., GAYDADJIEV, G., AND VASSILIADIS, S. Multimedia rectangularly addressable memory. *Multimedia, IEEE Transactions on 8*, 2 (2006), 315 – 322.

[86] LAWRIE, D. H., AND VORA, C. R. The prime memory system for array access. *IEEE Trans. Comput. 31* (May 1982), 435–442.

[87] LE, H. Q., STARKE, W. J., FIELDS, J. S., O'CONNELL, F. P., NGUYEN, D. Q., RONCHETTI, B. J., SAUER, W. M., SCHWARZ, E. M., AND VADEN, M. T. IBM POWER 6 microarchitecture. *IBM J. Res. & Dev. 51*, 6 (2007), 639–662.

[88] LEE, J., LAKSHMINARAYANA, N. B., KIM, H., AND VUDUC, R. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 213–224.

[89] LEE, V. W., KIM, C., CHHUGANI, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., SINGHAL, R., AND DUBEY, P. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 451–460.

[90] LEFOHN, A. E., SENGUPTA, S., KNISS, J., STRZODKA, R., AND OWENS, J. D. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph. 25* (January 2006), 60–99.

[91] LIN, Y.-K., LI, D.-W., LIN, C.-C., KUO, T.-Y., WU, S.-J., TAI, W.-C., CHANG, W.-C., AND CHANG, T.-S. A 242mW, 10mm2 1080p H.264/AVC high profile encoder chip. In *Proceedings of the 45th annual Design Automation Conference* (New York, NY, USA, 2008), DAC '08, ACM, pp. 78–83.

[92] LITTLE, J. D. C. A proof for the queuing formula: L= w. *Operations Research 9*, 3 (1961), pp. 383–387.

[93] LU, J., CHEN, H., FU, R., HSU, W.-C., OTHMER, B., YEW, P.-C., AND CHEN, D.-Y. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2003), MICRO 36, IEEE Computer Society, pp. 180–.

[94] LUK, C.-K., AND MOWRY, T. C. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS-VII, ACM, pp. 222–233.

[95] LUK, C.-K., MUTH, R., PATIL, H., WEISS, R., LOWNEY, P. G., AND COHN, R. Profile-guided post-link stride prefetching. In *Proceedings of the 16th International Conference on Supercomputing* (New York, NY, USA, 2002), ICS '02, ACM, pp. 167–178.

[96] MACEDONIA, M. The GPU enters computing's mainstream. *IEEE Computer Magazine 36*, 10 (Oct. 2003), 106–108.

[97] MÄDING, N., LEENSTRA, J., PILLE, J., SAUTTER, R., BÜTTNER, S., EHRENREICH, S., AND HALLER, W. The vector fixed point unit of the synergistic processor element of the Cell architecture processor. In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe* (3001 Leuven, Belgium, Belgium, 2006), European Design and Automation Association, pp. 244–248.

[98] MANAVSKI, S. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on* (2007), pp. 65 –68.

[99] MOORE, G. E. Readings in computer architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, ch. Cramming more components onto integrated circuits, pp. 56–59.

[100] MOSCIBRODA, T., AND MUTLU, O. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), USENIX Association, pp. 18:1–18:18.

[101] MOWRY, T. C., LAM, M. S., AND GUPTA, A. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1992), ASPLOS-V, ACM, pp. 62–73.

[102] MUTLU, O., AND MOSCIBRODA, T. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 146–160.

[103] MUTLU, O., AND MOSCIBRODA, T. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th annual International Symposium on Computer Architecture* (Washington, DC, USA, 2008), ISCA '08, IEEE Computer Society, pp. 63–74.

[104] NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM international symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 208–222.

[105] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable parallel programming with CUDA. *Queue 6* (March 2008), 40–53.

[106] NUZMAN, D., ROSEN, I., AND ZAKS, A. Auto-vectorization of interleaved data for SIMD. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), ACM, pp. 132–143.

[107] NUZMAN, D., AND ZAKS, A. Outer-loop vectorization: revisited for short simd architectures. In *PACT '08: Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), ACM, pp. 2–11.

[108] NVIDIA. CUDA best practice guide, edition 3.0.

[109] NVIDIA. http://developer.nvidia.com/object/cuda_3_0_downloads.html.

[110] NVIDIA. The CUDA compiler driver NVCC, edition 2.2.

[111] NVIDIA. Bringing high-end graphics to handheld devices. *Whitepaper* (2011), 1–28.

[112] NVIDIA. *NVIDIA CUDA Programming Guide 4.0.* 2011.

[113] OWENS, J., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J., AND PHILLIPS, J. GPU computing. *Proceedings of the IEEE 96*, 5 (May 2008), 879 –899.

[114] OWENS, J., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRUGER, J., LEFOHN, A., AND PURCELL, T. A survey of general purpose computation on graphics hardware. *Computer Graphics Forum 26* (2007), 80–113.

[115] PAJUELO, A., GONZALEZ, A., AND VALERO, M. Speculative dynamic vectorization. In *Proceedings of the 29th Ann. Int'l Symp. Computer Architecture* (2002), pp. 271–280.

[116] PALACHARLA, S., AND KESSLER, R. E. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21st annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 24–33.

[117] PARK, J. W. An efficient buffer memory system for subarray access. *IEEE Trans. Parallel Distrib. Syst. 12* (March 2001), 316–335.

[118] PARK, J. W. Multiaccess memory system for attached SIMD computer. *IEEE Trans. Comput. 53* (April 2004), 439–452.

[119] PATEL, S., AND HWU, W.-M. W. Accelerator architectures. *Micro, IEEE 28*, 4 (2008), 4 –12.

[120] PHAM, D., AIPPERSPACH, T., BOERSTLER, D., BOLLIGER, M., CHAUDHRY, R., COX, D., HARVEY, P., HARVEY, P., HOFSTEE, H., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., PHAM, M., PILLE, J., POSLUSZNY, S., RILEY, M., STASIAK, D., SUZUOKI, M., TAKAHASHI, O., WARNOCK, J., WEITZEL, S., WENDEL, D., AND YAZAWA, K. Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor. *Solid-State Circuits, IEEE Journal of 41*, 1 (2006), 179 – 196.

[121] QUALCOMM. http://www.qualcomm.com/news/releases/2011/02/14/qualcomm-announces-next-generation-snapdragon-mobile-chipset-family.

[122] RAFIQUE, N., LIM, W.-T., AND THOTTETHODI, M. Effective management of DRAM bandwidth in multicore processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (Washington, DC, USA, 2007), PACT '07, IEEE Computer Society, pp. 245–258.

[123] RAMIREZ, A., CABARCAS, F., JUURLINK, B., ALVAREZ MESA, M., SANCHEZ, F., AZEVEDO, A., MEENDERINCK, C., CIOBANU, C., ISAZA, S., AND GAYDADJIEV, G. The SARC architecture. *Micro, IEEE 30*, 5 (2010), 16 –29.

[124] REN, G., WU, P., AND PADUA, D. Optimizing data permutations for SIMD devices. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), ACM, pp. 118–131.

[125] RIXNER, S., DALLY, W., KAPASI, U., MATTSON, P., AND OWENS, J. Memory access scheduling. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (2000).

[126] RUSSELL, R. M. The CRAY-1 computer system. *Communications of the ACM* (Jan. 1978), 63–72.

[127] RYOO, S., RODRIGUES, C. I., STONE, S. S., BAGHSORKHI, S. S., UENG, S.-Z., STRATTON, J. A., AND HWU, W.-M. W. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2008), CGO '08, ACM, pp. 195–204.

[128] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph. 27* (August 2008), 18:1–18:15.

[129] SEZNEC, A., AND LENFANT, J. Odd memory systems may be quite interesting. *SIGARCH Comput. Archit. News 21* (May 1993), 341–350.

[130] SEZNEC, A., AND LENFANT, J. Interleaved parallel schemes. *Parallel and Distributed Systems, IEEE Transactions on 5*, 12 (dec 1994), 1329 –1334.

[131] SHAPIRO, H. D. Theoretical limitations on the efficient use of parallel memories. *IEEE Trans. Comput. 27* (May 1978), 421–428.

[132] SHAW, D. E., DENEROFF, M. M., DROR, R. O., KUSKIN, J. S., LARSON, R. H., SALMON, J. K., YOUNG, C., BATSON, B., BOWERS, K. J., CHAO, J. C., EASTWOOD, M. P., GAGLIARDO, J., GROSSMAN, J. P., HO, C. R., IERARDI, D. J., KOLOSSVÁRY, I., KLEPEIS, J. L., LAYMAN, T., MCLEAVEY, C., MORAES, M. A., MUELLER, R., PRIEST, E. C., SHAN, Y., SPENGLER, J., THEOBALD, M., TOWLES, B., AND WANG, S. C. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 1–12.

[133] SINHAROY, B., KALLA, R. N., TENDLER, J. M., EICKEMEYER, R. J., AND JOYNER, J. B. POWER 5 system microarchitecture. *IBM J. Res. & Dev. 49*, 4/5 (2005), 505–521.

[134] STARK, J., BROWN, M. D., AND PATT, Y. N. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (New York, NY, USA, 2000), MICRO 33, ACM, pp. 57–66.

[135] STMICROELECTRONICS, AND CEA. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. *Whitepaper* (November 2010), 1–26.

[136] STRATTON, J. A., GROVER, V., MARATHE, J., AARTS, B., MURPHY, M., HU, Z., AND HWU, W.-M. W. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 111–119.

[137] STUECHELI, J., KASERIDIS, D., DALY, D., HUNTER, H. C., AND JOHN, L. K. The virtual write queue: coordinating DRAM and last-level cache policies. In *Proceedings of the 37th annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 72–82.

[138] SUN, T., AND YANG, Q. A comparative analysis of cache designs for vector processing. *IEEE Trans. Computers 48* (Mar. 1999), 331–344.

[139] TENDLER, J. M., DODSON, J. S., FIELDS, J. S., JR., H. L., AND SINHAROY, B. POWER 4 system microarchitecture. *IBM J. Res. & Dev. 46*, 1 (Jan. 2002), 5–25.

[140] THAKKAR, S. T., AND HUFF, T. Internet streaming simd extensions. *Computer 32*, 12 (1999), 26–34.

[141] THISTLE, M. R., AND SMITH, B. J. A processor architecture for horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 1988), Supercomputing '88, IEEE Computer Society Press, pp. 35–41.

[142] TRUONG, D., BODIN, F., AND SEZNEC, A. Improving cache behavior of dynamically allocated data structures. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on* (oct 1998), pp. 322 –329.

[143] UENG, S.-Z., LATHARA, M., BAGHSORKHI, S. S., AND HWU, W.-M. W. Languages and compilers for parallel computing. Springer-Verlag, Berlin, Heidelberg, 2008, ch. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15.

[144] VALERO, M., LANG, T., PEIRON, M., AND AYGUADE, E. Conflict-free access for streams in multimodule memories. *IEEE Trans. Computers 44* (1995), 634–646.

[145] VALIANT, L. G. A bridging model for parallel computation. *Commun. ACM 33* (August 1990), 103–111.

[146] VANDERWIEL, S. P., AND LILJA, D. J. Data prefetch mechanisms. *ACM Comput. Surv. 32* (June 2000), 174–199.

[147] VOLKOV, V. Better performance at lower occupancy. In *GPU Technology Conference 2010* (2010), GTC '10.

[148] WATSON, W. J. The TI ASC: a highly modular and flexible super computer architecture. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I* (New York, NY, USA, 1972), AFIPS '72 (Fall, part I), ACM, pp. 221–228.

[149] WIJSHOFF, H. A. G., AND VAN LEEUWEN, J. The structure of periodic storage schemes for parallel memories. *IEEE Trans. Computers 34*, 6 (1985), 501–505.

[150] WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (2010), pp. 235 –246.

[151] YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 86–97.

[152] YUAN, G., BAKHODA, A., AND AAMODT, T. Complexity effective memory access scheduling for many-core accelerator architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009), pp. 34 –44.

[153] ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation* (New York, NY, USA, 2004), PLDI '04, ACM, pp. 255–266.

List of Publications

International Journals

1. C. Gou, G. N. Gaydadjiev, **Addressing GPU On-chip Shared Memory Bank Conflicts Using Elastic Pipeline**, invited to *Special Issue of International Journal of Parallel Programming on SI: Computing Frontiers 2011 Best Papers* (in press)
2. C. Gou, G. N. Gaydadjiev, **Exploiting SPMD Horizontal Locality**, *IEEE Computer Architecture Letters*, Volume 10, Issue 1, pp. 20-23, JANUARY-JUNE 2011, DOI: 10.1109/L-CA.2011.5
3. C. Galuzzi, C. Gou, D.R.H. Calderón, G. N. Gaydadjiev, S. Vassiliadis, **High-bandwidth Address Generation Unit**, *Journal of Signal Processing Systems*, Vol. 57, Number 1, pp. 33–44, 2008

International Conference Proceedings

1. C. Gou, G. N. Gaydadjiev, **Improving GPGPU DRAM Efficiency Using Access Granularity**, submitted to *the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, 2011
2. C. Gou, G. N. Gaydadjiev, **Alleviating On-chip Shared Memory Bank Conflicts in Data Parallel Architectures**, to appear in *Electronics - ET2011*, Sozopol, Bulgaria, September 2011
3. C. Gou, G. N. Gaydadjiev, **Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts**, *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11)*, pp. 1–11, Ischia, Italy, May 2011. Acceptance rate: 18.2% (22/121). **Nominated for the Best Paper Award by the Program Committee**
4. C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **SAMS Multi-Layout Memory: Providing Multiple Views of Data to Boost SIMD Performance**, *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*, pp. 179–188, Tsukuba, Japan, June 2010. Acceptance rate: 17.8% (32/180). **Best Paper Award**
5. C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **SAMS: Single-Affiliation Multiple-Stride Parallel Memory Scheme**, *Proceedings of the 2008*

workshop on Memory access on future processors: a solved problem? (MAW'08), pp. 359–367, Ischia, Italy, May 2008

Technical Reports

1. C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **Matched SAMS Scheme: Supporting Multiple Stride Unaligned Vector Accesses with Multiple Memory Modules**, *CE Technical Report*, *CE-TR-2008-06*, pp. 1–27, Computer Engineering Lab, TU Delft, October 2008
2. C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **2DSMM: 2D Strided Multiaccess Memory**, *CE Technical Report*, pp. 1–38, Computer Engineering Lab, TU Delft, July 2007

Non Peer-Reviewed Conference Proceedings

1. C. Gou, G. Kuzmanov, G. N. Gaydadjiev, **2D Stride Multiaccess Memory Scheme**, *HiPEAC ACACES 2007*, pp. 193–197, L'Aquila, Italy, July 2007

Samenvatting

De efficiëntie van het geheugensysteem is van cruciaal belang voor elke processor om hoge prestaties te bereiken, vooral in het geval van data parallelle machines. Dataverwerkingsmogelijkheden van parallelle hardware units zullen onbenut blijven wanneer gegevens niet aanhoudend en op tijd bereikt kunnen worden. Onregelmatige vector geheugenaccessen kunnen leiden tot een inefficiënt gebruik van de parallelle banken / modules / kanalen en de algemene prestaties aanzienlijk verslechteren, zelfs wanneer snelle, parallelle geheugensystemen worden gebruikt. Dit probleem geldt ook voor vele regelmatige programmas die onregelmatige vector geheugenaccessen vertonen tijdens het uitvoeren van het programma. Dit proefschrift identificeert de mismatch tussen de optimale geheugenaccesspatronen vereist door het programma en de fysieke data lay-out, als een van de belangrijkste factoren voor de inefficiëntie van geheugenaccess. Wij stellen configureerbare geheugenschemas voor om dit probleem aan te pakken in data parallelle acceleratoren. Meer specifiek, dit proefschrift schetst een uitbreiding van traditionele benaderingen door het voorstellen van twee nieuwe parallelle geheugenschema's die de bank access conflicten verminderen voor de meest gebruikte accesspatronen. We stellen ook een ontwerpmethode voor om de informatie over de geheugenaccesspatroon over te brengen naar de voorgestelde parallelle geheugenschemas. Verder beschrijven we technieken die dynamisch de instructiesequencer van een multithreaded vector architectuur aanpassen en de geheugenaccesspatronen benvloeden om de efficiëntie van accessen naar het on-chip geheugen te verbeteren. Als laatste identificeren we een nieuw data lokaliteitstype en exploiteren deze voor het dynamisch aanpassen van de off-chip geheugenaccessgranulariteit van manycore dataparallelle architecturen, om de efficiëntie van het hoofdgeheugenaccess te verbeteren. We implementeerden onze voorstellen als een uitbreiding van moderne dataparallelle architecturen en onze evaluatie resultaten tonen aan dat efficiëntie van het geheugen en de algemene prestaties van het systeem verbeterd kunnen worden tegen minimale hardware-kosten, terwijl tegelijkertijd de programmeringsoverhead sterk kan worden verminderd.

Curriculum Vitae

Chunyang Gou was born on March 1, 1981 in Sichuan, China. He received his secondary education between 1993 and 1999 at No. 1 Middle School of Bazhong in Sichuan, China. From 1999 to 2003 he studied at the School of Communications and Information Engineering, University of Electronic Science and Technology of China in Chengdu, China. He received Bachelor's degree in 2003, and continued in the same year to study at the Department of Electronic Engineering, Tsinghua University in Beijing, China. He received his Master's degree in Information and Communication Engineering in 2006.

In October 2006, he joined the Computer Engineering laboratory of Delft University of Technology in the Netherlands, and, under the advisory of Assistant Professor Georgi N. Gaydadjiev, he started his PhD study, working on parallel memory schemes for data parallel architectures. The research work was funded by the European Commission in the context of the SARC project and was continued by the ENCORE project. The results of this work are presented in the current dissertation.

Chunyang's research interests include compilers and programming systems, program optimizations and performance tuning, multicore and manycore, GPU computing, data locality and memory hierarchy, with particular focus on data parallel architectures and high-performance parallel memory schemes.