Ozana Silvia Dragomir

K-loops: Loop Transformations for Reconfigurable Architectures

K-loops: Loop Transformations for Reconfigurable Architectures

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

donderdag 16 juni 2011 om 15:00 uur

door

Ozana Silvia DRAGOMIR

Master of Science, Politehnica University of Bucharest
geboren te Ploieşti, Roemenië

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

Copromotor:
Dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J.Sips,	Technische Universiteit Delft, promotor
Dr. K.L.M.Bertels,	Technische Universiteit Delft, copromotor
Prof.dr. A.P. de Vries,	Technische Universiteit Delft
Prof.dr. B.H.H. Juurlink,	Technische Universität Berlin, Duitsland
Prof.dr. F. Catthoor,	Katholieke Universiteit Leuven, België
Prof.dr.ir. K. De Bosschere,	Universiteit Gent, België
Dr. J.M.P. Cardoso,	Universidade do Porto, Portugal
Prof.dr. C.I.M. Beenakker,	Technische Universiteit Delft, reservelid

Ozana Silvia Dragomir

K-loops: Loop Transformations for Reconfigurable Architectures
Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica
Met samenvatting in het Nederlands.

ISBN 978-90-72298-00-3

Subject headings: compiler optimizations, loop transformations, reconfigurable computing.

Cover image by Sven Geier.

Printed by Wöhrmann Print Service (www.wps.nl), The Netherlands.

To my beloved husband, to our wonderful son.

K-loops: Loop Transformations for Reconfigurable Architectures

Abstract

RECONFIGURABLE computing is becoming increasingly popular, as it is a middle point between dedicated hardware, which gives the best performance, but has the highest production cost and longest time to the market, and conventional microprocessors, which are very flexible, but give less performance. Reconfigurable architectures have the advantages of high performance (given by the hardware), great flexibility (given by the software), and fast time to the market.

Signal processing represents an important category of applications nowadays. In such applications, it is common to find large computation intensive parts of code (the application kernels) inside loop nests. Examples are the JPEG and the MJPEG image compression algorithms, the H264 video compression algorithm, the Sobel edge detection, etc. One way of increasing the performance of these applications is to map the kernels onto reconfigurable fabric.

The focus of this dissertation is on kernel loops (K-loops), which are loop nests that contain hardware mapped kernels in the loop body. In this thesis, we propose methods for improving the performance of such K-loops, by using standard loop transformations for exposing and exploiting the coarse grain loop level parallelism.

We target a reconfigurable architecture that is a heterogeneous system consisting of a general purpose processor and a field programmable gate array (FPGA). Research projects targeting reconfigurable architectures are trying to give answers to several problems: how to partition the application – decide which parts to be accelerated on the FPGA, how to optimize these parts (the kernels), what is the performance gain. However, only few try to exploit the coarse grain loop level parallelism.

This work goes towards automatically deciding the number of kernel instances to place into the reconfigurable hardware, in a flexible way that can balance between area and performance. In this dissertation, we propose a general framework that helps determine the optimal degree of parallelism for each hardware

mapped kernel within a K-loop, taking into account area, memory size and bandwidth, and performance considerations. In the future it can also take into account power. Furthermore, we present algorithms and mathematical models for several loop transformations in the context of K-loops. The algorithms are used to determine the best degree of parallelism for a given K-loop, while the mathematical models are used to determine the corresponding performance improvement. The algorithms are validated with experimental results. The loop transformations that we analyze in this thesis are loop unrolling, loop shifting, K-pipelining, loop distribution, and loop skewing. An algorithm that decides which transformations to use for a given K-loop is also provided. Finally, we also present an analysis of possible situations and justifications of when and why the loop transformations have or have not a significant impact on the K-loop performance.

Acknowledgements

The fact that I came to Delft to do my PhD is the result of a series of both fortunate and unfortunate events. I would like to thank to the main people that made it possible: Prof. Irina Athanasiu, Prof. Stamatis Vassiliadis, Vlad Sima, and Elena Panainte. Irina Athanasiu was my first mentor from the Politehnica University of Bucharest, and she taught many of us not only about compilers, but also valuable lessons of life. Stamatis Vassiliadis was the one that gave me the chance to do this PhD, and accepted me in his group. He was a great spirit and a great leader, and the group has never been the same without him. Vlad Sima has been near me in many crucial moments of my life. I thank him for all his patience and support throughout the years and, although our lives went on different paths, I will always think of him fondly. I am grateful to Elena Panainte and to her husband Viorel for welcoming me into their home, and for sharing their knowledge about life here, which made it easier for me to make the decision of coming to live here.

Next, I would like to thank my advisor, Dr. Koen Bertels, who believed in me more than I did. If it wasn't for him, I might not have found the strength to complete the PhD. I am also grateful to Stephan Wong and to Todor Stefanov, with whom I worked on my first papers. They critically reviewed my work and gave me helpful suggestions to improve it. Many thanks also to Carlo Galuzzi who reviewed the text of my dissertation.

Many people from the CE group made my life here in Delft easier and nicer, and I would like to thank all of them for the fun nights and social events, for the help when relocating (too many times already), for the time to chat, for the nice pictures, and for everything else. Many thanks to all the Romanians, to my past and present office mates, to Kamana, Yana, Filipa, Behnaz, Lu Yi, Demid, Roel, Dimitris, Sebastian, Stefan (Bugs), Cor, Ricardo, and to the CE staff (Lidwina, Monique, Erik, and Eef).

Special thanks go to Eva and Vašek, for opening up my mind regarding the terrifying subject of child rearing; to Marjan Stolk, for introducing us to the Dutch language and culture in a pleasant and fun way and for being more than

an instructor; to Martijn, for being the ‘best man’ and a true friend; and to Sean Halle, for insightful discussions, for believing in me, and for becoming an (almost) regular presence in our family.

Last but not least, I would like to thank my family. My parents have always been there for me and encouraged me to follow my dreams. I thank them for understanding that I cannot be with them now, and I miss them.

To my husband, Arnaldo, I would like to thank for his love and patience, and for our wonderful son, Alex. With them I am a complete person, and, I hope, a better person.

Ozana Dragomir Delft, The Netherlands, June 2011

Table of contents

List of Tables

List of Figures

List of Acronyms and Symbols

ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
CLB	Configurable Logic Block
CCU	Custom Computing Unit
CPU	Central Processing Unit
CGRA	Coarse-Grained Reconfigurable Array
CIF	Common Intermediate Format
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DF	Deblocking Filter
DFG	Data Flow Graph
DIL	Dataflow Intermediate Language
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSEP	Design Space Exploration
DSP	Digital Signal Processor
DWARV	DelftWorkbench Automated Reconfigurable VHDL Generator
EAPR	Early Access Partial Reconfiguration
FGRA	Fine-Grained Reconfigurable Architectures
FHD	Full High Definition
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GPP	General Purpose Processor
HD	High Definition
HW	Hardware
ICAP	Internal Configuration Access Port
ILP	Instruction Level Parallelism
I/O	Input/Output
IP	Intellectual Property
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
LAR	Large Area Requirements
LLP	Loop Level Parallelism
MB	MacroBlock
MDFG	Multi-Dimensional Data Flow Graph

MJPEG	Motion-JPEG
MPEG	Moving Picture Experts Group
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
PE	Processing Element
PLB	Processor Local Bus
QVGA	Quarter Video Graphics Array
RA	Reconfigurable Architecture
RC	Reconfigurable Computing
RP	Reconfigurable Processor
RTL	Register Transfer Level
RU	Reconfigurable Unit
OS	Operating System
SAR	Small Area Requirements
SD	Standard Definition
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SUIF	Stanford University Intermediate Format
SW	Software
TLP	Task Level Parallelism
VHDL	Very High Scale Integrated Circuits Hardware Description Language
VLE	Variable Length Encoder
VLIW	Very Long Instruction Word

1 Introducing K-Loops

THIS CHAPTER introduces the research context of this thesis and answers to the following questions: what is the problem that we are trying to solve, why is it important for the community, and what are the solutions that we propose. Section 1.1 presents the research context, Section 1.2 presents the problem definition, Section 1.3 presents the contributions of this thesis, and, finally, the thesis organization is presented in Section 1.4.

1.1 Research Context

Industry and society have been benefiting from steady improvements in computation power. While the hardware and software engineers have to develop new approaches to provide the best possible performance for increasingly complex applications, the threat of the power-wall has forced hardware to enter the multicore era. Because of this, a lot of effort has been directed to the domain of automatic parallelization, that would allow already existing applications to take advantage of the multicore characteristics. One solution for increasing the performance, but with less effort, is to use heterogeneous systems, which include General Purpose Processors (GPPs) and various other processing elements such as Digital Signal Processors (DSPs), and reconfigurable Processing Elements (PEs), such as Field Programmable Gate Arrays (FPGAs). In our work, we target a Reconfigurable Architecture (RA) that is a heterogeneous system consisting of a GPP and an FPGA.

Reconfigurable Computing (RC) is a middle point between Application Specific Integrated Circuits (ASICs) and conventional microprocessors. The ASIC hardware is a custom hardware, thus has the highest performance for the targeted application. However, it has very long and costly design times and no post-design modifiability. If a change needs to be made to the circuit, a new chip

(and probably a new device to use the chip) must be designed, manufactured, shipped and installed. Conventional microprocessors (also known as GPPs) have a low design cost and high post-design modifiability, but low performance when compared to dedicated architectures. RAs have the advantages of high performance (given by the hardware), great flexibility (given by the software), and fast time to the market. If a change needs to be made to the hardware in a RA, the FPGA only needs to be reprogrammed. Another advantage is that an FPGA can be programmed with different Custom Computing Units (CCUs), either at the same time (parallel execution) or at different times.

Examples of architectures that adhered to the multicore and heterogeneous paradigm are Xilinx's Virtex family with FPGA and PowerPC, and Altera's embedded technologies that combine FPGAs with embedded processors – the Cortex-M1 (FPGA with ARM) and the V1 ColdFire (FPGA with Freescale Semiconductor's ColdFire processor) [5], the announced MP32 (FPGA with MIPS) and 'Stellartone' (FPGA with Intel Atom E600) [6].

In heterogeneous systems, the applications' performance can be increased by mapping onto the different PEs, such as a Reconfigurable Processor (RP) or a DSP, the computation intensive parts of the application (the application kernels). In such systems, there are several problems to be answered: how to partition the application – decide which parts go on which PEs, how to optimize these parts (the kernels), what is the performance gain.

The focus of this thesis is on coarse grain loop level parallelism, which refers to kernel functions that are called from within a loop nest. This is the case when classical loop level parallelism meets standard task level parallelism, and traditional loop optimizations are applied for improving the performance.

In our work, a kernel refers to a piece of code (or a function) where the application spends a considerable amount of the execution time – more than 20%. This type of kernel is what we also call a 'large kernel', to emphasize that the kernels in this thesis refer to functions where a large part of the execution time of an application is spent. The applications that we target are from the signal processing domain, where large kernels in the innermost loop are common. Examples are the JPEG (Joint Photographic Experts Group) and the MJPEG (Motion-JPEG) image compression algorithms, the H264 video compression algorithm, the Sobel edge detection, and so on. Multimedia applications such as M/JPEG have also a streaming nature, which means that data enters at one point, and is then propagated through a series of tasks. The methodology that we propose in this thesis works better for applications that have such a task-chain structure, especially for multi-kernel K-loops.

This work goes towards automatically deciding the choice of kernel instances to place into the reconfigurable hardware, in a flexible way that can balance between area and performance. In the future it can also take into account power. The benefits to this approach are the following.

- It is labour-saving, because automatically generated kernel hardware code can be used. Even if the hardware kernel code is not optimal, compared to a hand tuned implementation, because of the parallelism between different kernel instances, a similar overall speedup of the loop nest can be obtained.
- It speeds up the design-exploration – a quick estimate of scalability limits can be achieved, without needing to hand-tune the kernels in order to investigate the achievable performance. An example from our experimental results is the DCT kernel.
- Third party library kernels can automatically run in parallel as well. Even aggressively optimized kernel implementations can benefit from running them in parallel.

While we focus in this thesis on FPGA mapped kernels, similar methods to those proposed in the next chapters can be applied for automatic parallelization of applications on multicore processors. In this case, the kernels are offloaded to other cores in the system, and these cores could be tailored for specific application domains. However, when applied for multicore processing, the methodology would be different regarding the memory aspects. On the reconfigurable platform that we are targeting, the memory is shared between the PEs and all kernels read and write from the same memory. This is not the case on multicore platforms, where each core has its own memory.

1.2 Problem Overview

In this thesis, a kernel loop (K-loop) is defined as a loop containing in the loop body one or more kernels mapped on the reconfigurable hardware. The loop may contain also code that is not mapped on the reconfigurable hardware, but will always execute on the general purpose processor (in software). The software code and the hardware kernels may appear in any order in the loop body. The number of hardware mapped kernels in the K-loop determines its size. An example of a size one K-loop is illustrated in Figure 1.1, where `SW` is the software function and `K` is the hardware mapped kernel.

```
for (i = 0; i < N; i++) do
    /* a pure software function                    */
    SW(blocks[i]);
    /* a hardware mapped kernel                    */
    K(blocks[i]);
end
```

Figure 1.1: K-loop with one hardware mapped kernel.

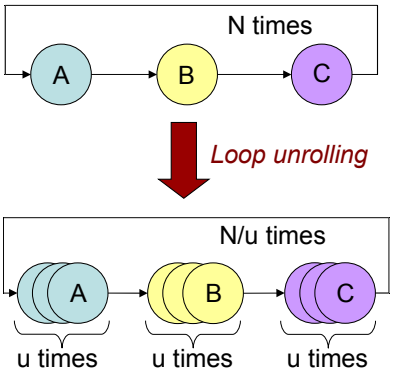


Figure 1.2: Loop unrolling.

The problem addressed in this thesis is improving the performance of such K-loops. This is important because, by definition, K-loops are the parts where applications spend the most of their execution times. The proposed approach is to use standard loop transformations and maximize the parallelism inside the K-loop.

Loop unrolling [1, 20, 24–26, 32, 39, 76, 79, 82, 121, 139] is a transformation that replicates the loop body and reduces the iteration number. In our work, we use unrolling to expose the loop parallelism, allowing us to execute concurrently multiple kernels on the reconfigurable hardware. Figure 1.2 illustrates the unrolling transformation, where each task inside the initial loop is replicated for the same number of times inside the unrolled loop.

Loop shifting [31,60,82] is a particular case of software pipelining. The shifting transformation moves operations from one iteration of the loop body to the previous iteration, as can be seen in Figure 1.3. The operations are shifted from the beginning of the loop body to the end of the loop body and a copy of these operations is also placed in the loop prologue. To be more specific, the prologue

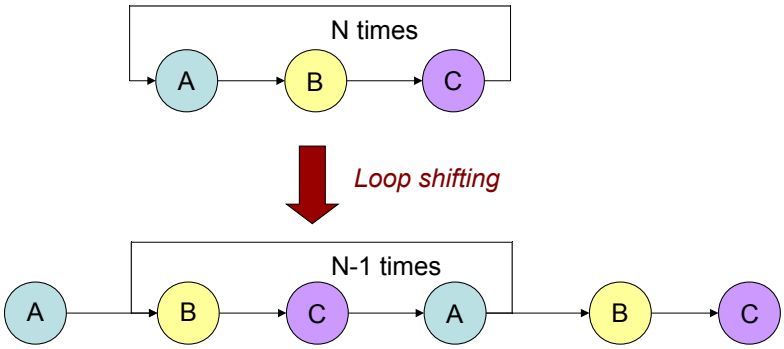


Figure 1.3: Loop shifting.

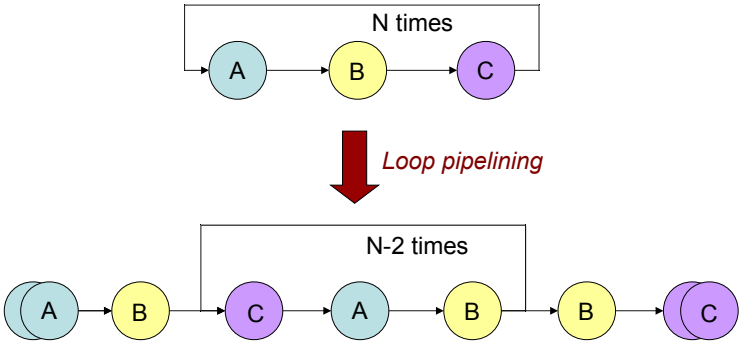


Figure 1.4: Loop pipelining.

consists of `A(1)`, the epilogue consists of `B(N)` and `C(N)`, and each iteration from the transformed loop body consists of `B(i)`, `C(i)`, and `A(i + 1)`. In our research, loop shifting means moving a function from the beginning of the K-loop body to the end, while preserving the correctness of the program. We use loop shifting to eliminate the data dependencies between software and hardware functions, allowing them to execute concurrently. We use the loop shifting transformation for single kernel K-loops.

Loop pipelining (or software pipelining [1,7,41,44,45,49,54,66,70,78,89,97,99, 113, 115–120, 128, 134, 135]) is used in our work for transforming K-loops with more than one kernel in the loop body. The loop pipelining transformation is illustrated in Figure 1.4. The prologue consists of `A(1)` followed by `A(2)` and `B(1)`. The epilogue consists of `B(N)` and `C(N − 1)`, followed by `C(N)`. Each iteration `i` from the transformed loop body consists of `C(i)`, `A(i + 2)`, and `B(i + 1)`. When applied to K-loops, we call this transformation K-pipelining.

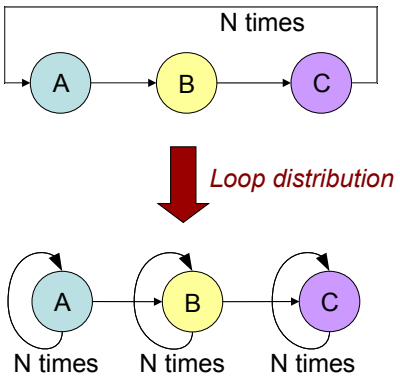
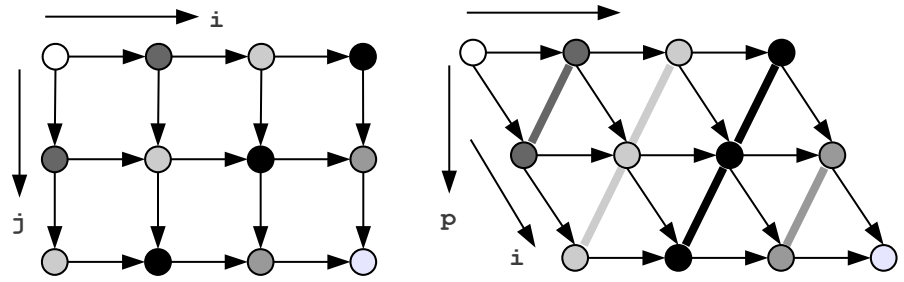


Figure 1.5: Loop distribution.

Its purpose is to eliminate the data dependencies between the software and hardware functions and allow them to execute in parallel. Assuming that the software and hardware functions alternate within the K-loop body, then half of the total number of function will need to be relocated.

Loop distribution [23, 72, 74, 75, 86, 92, 96] is a technique that breaks a loop into multiple loops over the same index range but each taking only a part of the loop's body. In our work, loop distribution is used to break down large K-loops (more than one kernel in the loop body) into smaller ones, in order to benefit more from parallelization with loop unrolling and loop shifting or K-pipelining. A generic representation of the loop distribution transformation is presented in Figure 1.5. In the top part of the figure, a loop with three tasks is illustrated. In the bottom part, each task is within its own loop and the execution order of the tasks has not been changed.

Loop skewing [11, 80, 96, 125, 126, 143] is a widely used loop transformation for nested loops iterating over a multidimensional array, where each iteration of the inner loop depends on previous iterations. This type of code cannot be parallelized or pipelined in its original form. Loop skewing rearranges the array accesses so that the only dependencies are between iterations of the outer loop, and enables the inner loop parallelization. Consider a nested loop whose dependencies are illustrated in Figure 1.6(a). In order to compute the element (i, j) in each iteration of the inner loop, the previous iteration's results $(i-1, j)$ must be available already. Performing the affine transformation $(p, t) = (i, 2*j+i)$ on the loop bounds and rewriting the code to loop on p and t instead of i and j, the iteration space and dependencies change as shown in Figure 1.6(b).



(a) Loop dependencies before skewing (b) Loop dependencies after skewing

Figure 1.6: Loop dependencies (different shades of gray show the elements that can be executed in parallel).

An algorithm that selects which transformations to use for a given K-loop is presented in Section 3.5.

Next, we propose a general framework for optimizing K-loops, and algorithms associated with the analyzed loop transformations. The algorithms are validated with experimental results. In the future it is desired that they are implemented in the compiler and enabled by specific compile time options.

1.3 Thesis Contributions

The contributions of this thesis are the following:

- A framework that helps determine the optimal degree of parallelism for each hardware mapped kernel within a K-loop, taking into account area, memory and performance considerations. The decision is taken based on profiling information regarding the available area, the kernel's area requirements, the memory bandwidth, the kernel's memory transfers, the kernel's software and hardware execution times, and the software execution times of all other pieces of code in the K-loop.
- Algorithms and mathematical models for each of the proposed loop transformations in the context of K-loops. The algorithms are used to determine the best degree of parallelism for a given K-loop, while the mathematical models are used to determine the corresponding performance improvement. The algorithms are validated with experimental results.

- An analysis of possible situations and justifications of when and why the loop transformations have or have not a significant impact on the K-loop performance.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 presents an overview of the most important loop optimizations and related research projects. In Chapter 3, we detail the notion of K-loop and the framework used in our work. In this chapter we present of general notations used throughout the thesis, together with area, memory and performance considerations. Next, the random test generator used for validating some of our transformations is introduced, followed by an overview of the different loop transformations presented in this thesis and an algorithm that decides which loop transformations to use on a given K-loop.

Chapter 4 is dedicated to loop unrolling, which is the main technique we use for exposing the available parallelism. By unrolling a K-loop without inter-iteration dependencies, several instances of the hardware mapped kernel can execute in parallel.

Chapter 5 and Chapter 6 are dedicated to loop shifting and K-pipelining. These two transformations are used for eliminating intra-iteration dependencies and enabling software-hardware parallelism. Loop shifting is a special case of pipelining and applies to K-loops with only one hardware mapped kernel. K-pipelining applies to K-loops with two or more kernels.

Chapter 7 addresses loop distribution, which represents an important step forward in accelerating large K-loops that are otherwise severely constrained by the area requirements. The method presented in this chapter proposes that large K-loops are split into smaller K-sub-loops, and each of these K-sub-loops is further optimized with the methods previously presented. If needed, the area is reconfigured in between K-sub-loops.

Chapter 8 focuses on loop skewing as the method for eliminating wavefront-like inter-iteration dependencies. The transformed (skewed) loop is then parallelized by means of loop unrolling. Two methods are proposed for scheduling the kernels execution within the skewed loop: one with all kernels running on the FPGA and one that schedules part of the kernels in software in order to have balance the software and hardware execution times.

Finally, Chapter 9 shows an overview of this thesis, draws the conclusions and presents the open issues and future work directions.

2
Related Work

IN MANY real life applications, loops represent an important source of optimization. Depending on the targeted architecture and on the class of applications, loops can be optimized for reducing the memory or the power consumption, or for increasing the application performance by speeding up the overall execution.

Various loop transformations, such as loop unrolling, software pipelining, loop shifting, loop distribution, loop merging, and loop tiling have been successfully used in the last decades for exploiting instruction level and loop level parallelism.

The focus of this thesis is on optimizing loops that contain kernels in the loop body (K-loops). For this purpose, traditional loop transformation such as the ones mentioned above are used. The kernels inside the K-loops are mapped onto reconfigurable hardware[1]. For this reason we are also interested in the related work in the field of Reconfigurable Computing.

In Section 2.1, we discuss the different types of parallelism that can be exploited within an application, and the loop transformations that can be used to enable and exploit that parallelism. Section 2.2 presents an overview of the most popular loop optimizations. Section 2.3 looks at several related research projects in Reconfigurable Computing and the way they address application optimization. Finally, in Section 2.5 we identify the open issues and present a choice of loop transformations to address these issues.

[1] More details about K-loops can be found in Section 3.1

2.1 Levels of Parallelism

There are various levels of parallelism that can be exploited within an application, depending on the application characteristics and on the target architecture.

The Instruction Level Parallelism (ILP) is the finest granularity. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which the instructions are executed. The most common architectures suitable for exploiting ILP are the superscalar and Very Long Instruction Word (VLIW) architectures, featuring multiple execution units that can be used for the parallel execution of instructions. Within a loop, the ILP can be exposed by using software pipelining techniques, which mimic the behaviour of the hardware pipelining. However, the pipelining can be performed only within basic blocks and the amount of parallelism is limited by the basic block size. Several loop transformations can be used to increase the basic block size. These include loop unrolling, loop tiling, and loop fusion.

Loop Level Parallelism (LLP) refers to the parallelism among iterations of a loop. It can be exploited only in the absence of loop-carried dependencies. Loop unrolling is used to expose LLP, while coarse-grain pipelining, loop shifting, loop peeling, loop fission and unimodular transformations can be used to break possible loop-carried dependencies. The data level parallelism is a particular case of LLP. It is a form of parallelization of computing across multiple processors in parallel computing environments.

Task Level Parallelism (TLP), also known as function level parallelism, emphasizes the distributed (parallelized) nature of the processing. The thread model is a particular case of TLP, targeting multiprocessor systems. Open Multi-Processing (OpenMP) [102] is a parallel programming paradigm for multiprocessor platforms with shared memory. Using specific language annotations (pragma-s for C for example) and API-s, it allows the developer to describe parallelism at a high level. It is suited for both data and task level parallelism. Message Passing Interface (MPI) [68] is an API specification that allows processes to communicate with each other by sending and receiving messages. It is typically used for parallel programs running on computer clusters and supercomputers. When TLP is associated with loops, the same transformations as in the case of LLP are needed to enable the parallelism.

In this thesis, we target task parallelism at loop level. As a result, our focus is on loop transformations that are suitable for LLP.

2.2 Loop Transformations Overview

In this section, an overview of the most popular loop optimizations is presented. It is difficult to give an exhaustive classification of the various loop transformations, as most of them can be used in different ways, depending on the architecture, on the purpose (expose parallelism, increase the parallelism, improve memory behaviour, minimize power consumption, etc.), on the type of exploited parallelism (ILP, LLP, etc.). While some transformations are used mainly for exposing parallelism (loop unrolling, fusion, etc.), there are others that work on data dependencies and loop bounds (loop bump, index split, etc.) and are used mainly for enabling the transformations from the first category.

2.2.1 Loop Unrolling

Loop unrolling, also known as loop unwinding, is a transformation that replicates the loop body and reduces the iteration number. The number of times that the loop is expanded is called unroll factor (u) and the iteration step will be u instead of 1. If the loop iteration count is not a multiple of the unroll factor, then the remainder of the loop iterations must be executed separately (the loop epilogue). Any loop can be unrolled, at both high and low levels. Unrolling can also be performed "by hand", and this is why some compilers perform loop rerolling before applying this transformation.

The benefits of unrolling have been studied on several different architectures [39]. Traditionally, the method is efficient because it removes branch instructions, reducing pipeline disruptions, and the index variable is modified fewer times. In reconfigurable computing it is used to expose parallelism. In all cases, it can reduce the number of computations and/or memory accesses when the same value is needed/referred multiple times, and can enable the use of other optimizations.

There are several approaches for unrolling loop nests. The main ones are unroll-and-jam [20] and loop quantization [1].

- **Unroll-and-jam** means unrolling an outer loop in a nested loop and fusing inner loop bodies together. Unlike the unrolling of a single loop, unrolling of multiple loops is not always legal[2]. The first unroll step can always be performed, but data dependencies may prevent the second fusion ('jam') step from being performed. Complex (non-linear)

[2]In the context of loops, a loop transformation is legal if it preserves the program correctness.

loop bounds can also affect the legality of performing a loop unrolling transformation. In a classical unroll-and-jam transformation, it is the responsibility of the fusion step to recognize when an illegal unrolling transformation is being attempted on a loop nest. The legality condition for unrolling multiple loops is equivalent to that of tiling [145]: given a set of k perfectly nested loops $i_1, \ldots, i_k$, it is legal to unroll the outer loop i_j if it is legal to permute the loop i_j to the innermost position. The unrolling of multiple loops can be seen as the division of the iteration space into small tiles. However, the iterations in an unrolled 'tile' execute copies of the loop body that have been expanded (unrolled) in place, rather than executing inner control loops as in tiling for cache locality.

- **Loop quantization** works by adding statements to the inner loop directly, to avoid replicated inner loops. Even though the structure of the loop nest remains the same, the iteration ranges are changed. Quantization is not always legal, as it changes the order of execution of the loop.

Several methods have been proposed for loop unrolling, for innermost loops only, but also for nested loops. For the work presented in this thesis it is important to use unroll methods that target nested loops, with a focus on maximizing data reuse and on computing the optimal unrolling factor that would bring the highest speedup, taking into account the hardware constraints.

Davidson and Jinturkar [32] were among the first to propose optimized versions of loop unrolling, answering questions such as how and when to unroll. The authors perform compile- and run-time analyses of the loop characteristics from a set of 32 benchmark programs. One of the key points was the importance of handling loops where the loop bounds are not known at compile-time. The analysis shows that most of the loops that are candidates for unrolling have bounds that are not known at compile-time (i.e., execution-time counting loops). Another factor analyzed was the control-flow complexity of loops that are candidates for unrolling. The analysis shows that unrolling loops with complex control-flow is as important as unrolling execution-time counting loops.

The problem of automatically selecting unroll factors for nested loops has been addressed first by Carr and Kennedy [26] and by Carr and Guan [25], and then by Koseki [76] and Sarkar [121]. Koseki unrolls outer loops simultaneously to achieve a higher level of instruction parallelism. The work takes into consideration loop dependencies and focuses on data reuse over iterations. The effect of code scheduling and register allocation is not taken into account. Sarkar addressed the problems of automatically selecting unroll factors for a set of perfectly nested loops, and generating compact code for the selected unroll factors.

The code generation algorithm for unrolling nested loops generates more compact code (with fewer remainder loops) than the unroll-and-jam transformation for the selected benchmarks. An algorithm for efficiently enumerating feasible unroll vectors is also presented.

Liao et al. propose in [82] a model for the hardware realization of kernel loops. The compiler is used to extract certain key parameters of the analyzed loop. From these parameters, by taking into account the resource constraints, the user is informed about the performance that can be obtained by unrolling the loop or applying loop unrolling together with software pipelining. The algorithm also suggests the optimal unroll factor to be used, but their method does not consider parallel execution. Besides, their model needs to be 'calibrated' by running several transformed loops in order to be able to make a prediction about the frequency and, thus, about the execution time.

Cardoso et al. propose in [24] a model to predict the impact of full loop unrolling on the execution time and on the number of required resources, without explicitly performing it. However, unroll-and-jam (unrolling one or more nested loops in the iteration space and fusing the inner loop bodies together) is not covered. The design space algorithm evaluates a set of possible unroll factors for multiple loops in the loop nest, searching for the one that leads to a balanced, efficient design. The estimation of needed resources for unrolled loops is performed simply by multiplying the resources for the loop body with the number of iterations, similar to the way we estimate the resource usage for multiple instances of the kernel. The execution time for the transformed loop (unrolled or pipelined) is computed by a complex formula, which takes into account the loop overhead, the length of the pipeline, and the number of successive accesses to the same memory. Nevertheless, their research does not consider parallel execution, nor memory bandwidth constraints.

The work of Weinhardt and Luk [137–139], known as pipeline vectorization, tries to extract parallelism from a sequential program and exploit this parallelism in hardware and it is based on software vectorizing compilers. In their approach, loop unrolling is an important technique used to increase basic block size, extending the scope of local optimizations. However, only inner loops are unrolled, in order to vectorize the next outer loop. The inner loop must have constant bounds in order to be completely unrolled. Partial unrolling is not suitable for pipeline vectorization and, as a result, it is not taken into account.

Lam et al. [79] apply loop fission in conjunction with loop unrolling, and across multiple loops. The advantage of considering unrolling and fission of all loops globally is that unrolled sub-loops from various loops can be potentially

executed in parallel. The target is a heterogeneous computing system, with two processing elements: one microprocessor and one FPGA. The addressed loops are the application kernels. The mapping and scheduling strategy considers that different instances of the same task can be scheduled on different PEs after unrolling. Partial reconfigurability is not considered.

In our work, loop unrolling is very important, as it is used to expose the parallelism and enable concurrent execution of multiple instances of the same kernel in hardware.

2.2.2 Software Pipelining

Software pipelining is another technique used to improve instruction level parallelism. The idea of software pipelining came from Patel and Davidson, who studied hardware pipelining and showed that by inserting delays between resources in the pipeline it is possible to eliminate resource collisions and improve the throughput.

In software pipelining, the operations in a loop iteration are broken into s stages. A single iteration executes stage 1 from iteration i, stage 2 from iteration $i-1$ and so on. One of the major difficulties of implementing software pipelining is the need for *prolog* – which initializes the pipeline for the first $s-1$ iterations – and *postlog*, which drains the pipeline for the last $s-1$ iterations. Note that for large loop bodies, prolog and postlog are very big and might affect performance due to cache misses.Software pipelining is often used in combination with loop unrolling as they can produce better results together than separately. By applying software pipelining after other loop transformations it is possible to have benefits in terms of data locality and reduced register pressure.

A survey on different methods used for software pipelining has been published in 1995 by Allan et al. [3]. In this survey, the authors explored relationships between the methods and highlight possibilities for improvements.

There are many criteria to classify the various approaches of software pipelining. The most important are:

- suitable for innermost loops ([54, 66, 97]) or for nested loops ([1, 48, 115]);
- resource constrained ([45]) or not ([115]);
- suitable for loops with control flow ([78]) or without ([1]);

- suitable for loops with conditional branches ([69,113]) or not (predicated execution);
- focused on achieving high throughput ([44,66]) or on minimizing register pressure ([44,69,78]) or on a low memory impact ([18,120,135]);
- the targeted architecture is: VLIW machines ([78]); Itanium ([97]); IA-64 ([118]); scalar architectures ([70]); etc.;
- the main method used for pipelining is based on:
  - kernel recognition: Perfect Pipelining ([1]), Petri nets ([49, 113]).Kernels recognition algorithms simulate loop unrolling and scheduling until a 'pattern' (a point where the schedule would be cyclic) appears. This pattern will form the kernel of the software-pipelined loop.
  - Heuristics: Modulo Scheduling ([44,45,66,78,89,97,115–120,134, 135]). The algorithms look for a solution with a cyclic allocation of resources. Every λ clock cycles, the resource usage will repeat. The algorithm looks thus for a schedule compatible with an allocation modulo λ, for a given λ (called the initiation interval). The value λ is incremented until a solution is found. Finding the optimal scheduling for a resource constrained scenario is a NP-complete problem. As a result, some researchers use software heuristics for scheduling loops that contain both intra- and inter-iterations data dependencies to achieve a high throughput.
  - Linear programming - Move-Then-Schedule algorithms ([7, 54, 66, 70]). The algorithms use an iterative scheme that alternatively schedules the body of the loop (loop compaction) and moves instructions across the back-edge of the loop as long as this improves the schedule.

Loop shifting is a particular case of software pipelining. Shifting means moving an operation from the beginning to the end of the loop body. To preserve the correctness of the program, a copy of the operation is placed in the loop prologue, while a copy of the loop body without the shifted operation is placed in the loop epilogue. Loop shifting is used in combination with scheduling algorithms to software pipeline loops.

In our work, shifting and pipelining can be performed at high (functional) level, to help eliminate data dependencies and enable hardware–software parallelism.

To expose more hardware parallelism, other techniques are used, such as loop unrolling.

Wang and Eisebeis [134] consider software pipelining as an instruction level transformation from a single dimension vector to a two-dimensional matrix (named DEcomposed Software Pipelining – DESP). The algorithm consists of two steps: the determination of the row numbers and the determination of the column numbers, which can be performed in any order. If the first step is to determine the row numbers, the algorithm is called FRLC, otherwise it is FCLR. For an operation, if the row number and column number have been determined, its position in the new loop body can be fixed. In conclusion, after determining the row numbers and the column numbers of all operations, the new loop body can be found. In the extended version of DESP, the algorithm exploits instruction level parallelism for loops with conditional jumps. In comparison with general algorithms like the one presented in [1], DESP with loop unrolling achieves the same time efficiency and it is much better from the space utilization point of view. Compared with algorithms that require that each loop body has identical schedules and initiation interval (restricted algorithms [128]), DESP is much more time efficient.

Stoodley and Lee [127] developed an algorithm called All Paths Pipelining (APP), focused on efficiently scheduling loops with control flows. APP uses techniques from both modulo scheduling and kernel recognition algorithms, extending the work in [1], [41], [128]. The algorithm has three steps: i) extract the iteration paths; ii) software pipelining each individual path using a modulo scheduling algorithm, and iii) insert code that is executed from one schedule to another when a subsequent iteration executes a different path - this step that combines the code being the main contribution of the research.

The work of Novack and Nicolau [99] in the domain of software pipelining is a different approach than previous heuristic-based strategies. They propose a technique called Resource-Directed Loop Pipelining (RDLP), which works by repeatedly exposing and exploiting parallelism in a loop until resource dependencies or loop-carried dependencies prevent further parallelization, or the balance between cost and desired performance is reached. Loop unrolling and loop shifting are the transformations used to expose parallelism.

Darte [31] is following the ideas developed in DESP. Loop shifting is used to move statements between iterations, thereby changing some loop independent dependencies into loop carried dependencies and vice versa. Then, the loop is compacted by scheduling multiple operations to execute concurrently, considering only the loop independent dependencies. Loop shifting is performed at

low (instruction) level, and the resulted loops are scheduled and mapped on the hardware. Similarly, [60] uses shifting to expose loop parallelism and then to compact the loop by scheduling multiple operations to execute in parallel.

Liao et al. propose in [82] a model for the hardware realization of kernel loops, where loop shifting is used in combination with loop unrolling. More details on this approach have been presented in Section 2.2.1.

2.2.3 Loop Fusion

Also known as loop merging, or loop jamming, loop fusion is the opposite of loop fission. It combines one or more loops with the same bounds into a single loop. Two loops can be fused if they have the same loop bounds. When they do not have the same bounds, it is sometimes possible to make them identical by using loop index split, loop bump, or other enabling loop transformations. However, it is not always legal to fuse two loops with the same bounds. Fusion is possible only if there are no statements S_1 in the first loop and S_2 in the second loop such that S_1 would depend on S_2 in the fused loop.

Loop fusion is extensively used for data locality. It increases variable reuse by bringing references closer together in time. This has an impact on memory space and energy consumption [38, 47, 64, 65, 75, 91, 112, 122, 132, 151].

Fusion is also used to increase the loop parallelism [75, 93, 122]. Fusion increases the granule size of parallel loops, which is desirable because it eliminates barrier synchronization points. Uniprocessors may also benefit from larger granule loops because they provide more fodder for instruction scheduling.

Although loop fusion reduces loop overhead, it does not always improve runtime performance, and may reduce it. For example, the memory architecture may provide better performance if two arrays are initialized in separate loops, rather than initializing both arrays simultaneously in one loop.

In the context of reconfigurable computing, we mention the work of Hammes et al. [61]. Loop fusion is applied to window-based loops, specific to the SA-C language [59, 98]. The SA-C compiler generates dataflow graphs, which are then compiled to VHDL. The aim of applying loop fusion is to fuse producer-consumer loops, as this often reduces data traffic and it may eliminate intermediary data structures.

In pipeline vectorization [139], loop fusion is one of the transformations used to increase basic block size and extend the scope of local optimizations.

In more recent works, loop fusion is used for reducing code size and improving

execution time [85, 86].

In our work, loop fusion is only theoretically useful, since we do not need to increase the granule of the K-loop (we are already considering large kernels). We foresee that small K-loops have more potential for improving the performance. However, it may be the case that by fusing two K-loops more hardware-software parallelism could be exploited. By fusing two or more K-loops, it would also be possible to schedule different kernels to run in parallel, but this is subject to future research.

2.2.4 Loop Fission

Loop fission, also known as loop distribution or loop (body) splitting, is a fundamental transformation used for optimizing the execution of loops. Loop distribution consists of breaking up a single loop (loop nest) into multiple loops (or loop nests), each of which iterates over distinct subsets of statements from the body of the original loop. The placement of statements into the loop must preserve the data and control dependencies of the loop. Loop fission may enable loop fusion, when one or more of the resulted sub-loops may be merged with other loops. This is desirable for increasing the parallelism granularity of loops.

Loop fission has been introduced by Muraoka [96] in the early '70s and it has been used traditionally in the following contexts.

- to break up a large loop that does not fit into the cache [74, 75];
- to improve memory locality in a loop that refers too many different arrays [75, 92];
- to enable other optimization techniques, such as loop fusion, loop interchange, and loop permutation [75, 86, 92];
- to eliminate dependencies in a large loop whose iterations cannot be run in parallel, by creating multiple sub-loops, some of which can be parallelized (automated vectorization).

Distribution may be applied to any loop as long as the dependencies are preserved. All statements belonging to the same dependence cycles must be placed in the same sub-loop. If statement S_2 depends on S_1 in the original loop, then the sub-loop containing S_1 must precede the sub-loop containing S_2.

More recent research targets embedded DSPs, such as the work of Liu et al. [86]. The authors propose to use loop distribution in conjunction with loop fusion

for improving the execution time and code size of loops. First, maximum loop distribution is applied on multi-level nested loops, taking into account the data dependencies. Direct loop fusion is then applied to the distributed loop in order to increase the performance of the execution, without increasing the code size. The technique, called MLD_DF, showed a timing improvement of 21% and a code size improvement of 7% for the selected loops, compared with the original. The performance improvement of MLD_DF is inferior to the authors work based only on loop fusion. However, with MLD_DF there is a decrease in code size, which was actually the desired outcome for the target architecture.

In reconfigurable computing, loop distribution has been used to break a loop into multiple tasks [72]. An extension of loop distribution, called loop dissevering, has been presented in [23]. The idea is the same, breaking the loop into multiple tasks for temporal partitioning. Each individual task will then be mapped onto the FPGA, making it possible to implement applications that exceed the size constraint of the FPGA. However, loop unrolling is not involved, thus there is no parallelism to exploit.

The research in [79] is more similar to our work. Loop fission is applied in conjunction with loop unrolling, and across multiple loops. The advantage of considering unrolling and fission of all loops globally is that unrolled sub-loops from various loops can be potentially executed in parallel. A more in-depth discussion of this work can be found in Section 2.2.1, where related works on loop unrolling are presented.

Loop distribution occupies an important place in our research. We use it to break large K-loops, containing multiple tasks, into smaller sub-K-loops that have more potential for performance improvement.

2.2.5 Loop Scheduling Transformations

For nested loops with wavefront-like dependencies, a higher level of parallelism is achieved by changing the execution sequence. This sequence of execution is commonly associated with a schedule vector s , which affects the order in which the iterations are performed. The iterations are executed along hyperplanes defined by s.

The wavefront processing method has been introduced by Lamport [80] and extended by Wolfe [143], and by Aiken and Nicolau [2]. Unimodular transformation is one of the major techniques for selecting an appropriate schedule vector. Other techniques are multi-dimensional retiming and loop striping. These techniques are detailed in the following.

Unimodular Transformations

The unimodular transformation technique combines various loop transformations, such as loop skewing, loop interchange and loop reversal, to achieve a particular goal, such as maximizing parallelism or data locality. The sequence of execution and the loop bounds are changed. One disadvantage is the added overhead: non-linear index bounds checking needs to be conducted on the new loop bounds to assure correctness, loop indexes become more complicated, and additional instructions are needed to calculate each new index.

This overhead can be more or less significant, depending on the architecture. In our case, where we target coarse-grain parallelism, such overhead will not have a noticeable impact on the performance. Among the unimodular transformations presented in this section, we have used only loop skewing for eliminating the dependencies for wavefront-like computations.

Loop Skewing. Loop skewing is a widely used loop transformation for wavefront-like computations [11, 80, 96, 143]. An example of wavefront-like dependency in a loop nest with the index variables `(i, j)` is: `A[i, j] = f(A[i − 1, j], A[i, j − 1])`. In order to compute the element `A(i, j)` in each iteration of the inner loop, the previous iteration's results `A(i − 1, j)` must be available already. Therefore, this code cannot be parallelized or pipelined as it is currently written. By performing the affine transformation `(p, t) = (i, 2 ∗ j + i)` on the loop bounds and by rewriting the code to loop on `p` and `t` instead of `i` and `j`, the iteration space is changed. The shape of the resulted loop changes (e.g. from rectangular to trapezoidal) and it can be parallelized because the dependencies are broken.

An implementation of the loop skewing transformation can be found in the Compaan [125, 126] compiler (method and tool set). Compaan transforms affine nested loops programs written in Matlab into a Kahn Process Network specification. The transformations supported by the Compaan compiler are: unfolding, plane cutting, skewing and merging. Compaan is used in conjunction with the Laura tool [153], which operates as a back-end for the compiler. Laura maps a Kahn Process Network (KPN) specification onto hardware, for example onto an FPGA.

Loop skewing is used also for increasing data locality, and is often used in conjunction with interchange, reversal, and tiling [71, 133, 141].

Loop Interchange. Loop interchange is the process of exchanging the positions of two loops in a loop nest, generally moving one of the outer loops to the innermost position. A major aim of loop interchange is to improve the cache performance for accessing array elements. Parallel performance can be improved by moving an independent loop outward in a loop nest to increase the granularity of each iteration. It is not always safe to exchange the iteration variables due to dependencies between statements or the order in which they must execute. To determine whether a compiler can safely interchange loops, a dependence analysis is required.

Loop interchange is used in conjunction with skewing, reversal, and tiling for improving data locality [141]. The theory of loop interchange is discussed by Allen and Kennedy in [4].

Loop Reversal. Reversal changes the direction in which the loop traverses the iteration range [136]. It is usually used in conjunction with other transformations, as it can modify the data dependencies [142]. In [133], it is used for improving data access regularity, together with loop skewing and loop permutation.

Multi-Dimensional Retiming

Passos et al. [104, 105] have proposed a type of loop restructuring that changes the structure of the iterations. The purpose is to achieve full parallelism within an iteration. Loop bodies are represented by Multi-Dimensional Data Flow Graphs (MDFG). Several algorithms for multi-dimensional retiming for uniform nested loops are presented: MD incremental retiming, MD chained retiming, and MD rotation scheduling.

Loop Striping

Loop striping has been proposed by Xue et al. in [148]. Loop striping groups iterations into stripes, where all iterations in a stripe are independent and can be executed in parallel. The original row-wise execution sequence does not change. The loop bounds remain unchanged, as opposite to unimodular transformations. The striping creates also a loop prologue and a loop epilogue. Experiments are performed on a set of two-dimensional digital filters. Loop striping gives better results than simple unrolling and than software pipelining. However, no comparison with the combination of the two or with loop skewing is performed.

We estimate that in the context of coarse-grain parallelism, the performance of loop striping is comparable to the performance of loop skewing. However, it is worth investigating more the potential of this loop transformation.

2.2.6 Enabler Transformations

A number of loop transformations such as strip mining, loop bump, loop extend/reduce, and loop index split are commonly used to enable other loop transformations (such as fusion, reverse, and interchange) that improve data regularity and locality.

Strip Mining

Strip mining is a method for adjusting the granularity of an operation by splitting a single loop into a nested loop. The resulting inner loop iterates over a section or strip of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's strip length.

Strip mining helps to ensure that the data used in a loop stays in the cache until it is reused. The partitioning of the loop iteration space leads to the partitioning of a large array into smaller blocks, thus fitting accessed array elements into the cache size, enhancing cache reuse and eliminating cache size requirements.

Used by itself, strip mining is not profitable. However, when it is used by loop tiling, it enhances parallelization.

Loop tiling, also known as loop blocking, strip mine and interchange, or supernode partitioning, is a loop optimization used by compilers to make the execution of certain types of loops more efficient. Loop tiling is the multi-dimensional generalization of strip mining.

The term loop tiling was introduced by Wolfe [142, 145]. He proposed the division of the iteration space of a loop into blocks or tiles with a fixed maximum size. Thus tiles become a natural candidate as the unit of work for parallel task scheduling. Synchronization between processors can be done between tiles, reducing the synchronization frequency (at some loss of potential parallelism). Parallelization is then a matter of optimization between tiles.

Loop tiling is one of the transformations used by Weinhardt and Luk in order to increase the basic block size and to extend the scope of local optimizations in their work on pipeline vectorization [139]. Loop tiling is often used in

conjunction with loop skewing, loop reversal and loop interchange, and used for improving regularity and locality [71, 141].

In our work, loop tiling is only theoretically useful, since we do not need to increase the granule of the K-loop (we are already considering large kernels).

Loop Bump

Loop bump modifies the loop bounds by adding/subtracting a fixed amount. The array indices inside the loop body are adjusted accordingly, buy subtracting/adding the same amount. It is used for enabling loop fusion, improving the data locality [27].

Loop Extend/Reduce

Loop extend expands the loop bounds, and introduces control code inside the loop body which tests that the loop index is within the old loop bounds. The original code will then be executed conditionally, within the control block. The indices used to access the data arrays inside the loop are not changed by this transformation. Loop reduce is the opposite of loop extend, and passes conditions existing inside the loop body into the loop manifest. The extend and reduce transformations are used for enabling loop fusion and improving the data locality [27].

Loop Index Split

This transformation splits the original loop into two or more loops that have the same loop body, but complementary iteration domains. It enables loop reverse (used for improving regularity) and loop interchange (used for regularity and locality) [27].

Loop peeling is a special case of loop index splitting. Loop peeling splits any problematic first (or last) few iterations from the loop and performs them outside of the loop body. It has two uses: it is used to remove dependencies created by the first (or last) few loop iterations, thereby enabling parallelization; and it is used to match the iteration control of adjacent loops to enable fusion.

Loop peeling was originally mentioned in [83], and automatic loop peeling techniques were discussed in [90]. August [10] showed how loop peeling could be applied in practice, and elucidated how this optimization alone may not increase program performance, but may expose opportunities for other

optimization leading to performance improvements. August used only heuristic loop peeling techniques.

Song [123, 124] proposes a technique for variable driven loop peeling, which is based on a static analysis for loop quasi-invariant variables, quasi-induction variables and their peeling lengths. The purpose is to determine the optimal peeling length needed to parallelize loops.

2.2.7 Other Loop Transformations

In addition to the loop transformations that held our interest so far, there are several more that need to be mentioned, although they do not find a direct application in our work.

Loop Unswitching

Loop unswitching [11] is applied when a loop contains a conditional with a loop-invariant test condition. It moves the conditional outside the loop by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional. This can improve the parallelization of the loop.

Loop-Invariant Code Motion

Loop-invariant code consists of statements that can be moved outside the body of a loop without affecting the semantics of the program. Loop-invariant code motion [11] is a compiler optimization which performs this movement automatically. While this transformation can improve performance in traditional programs, it does not show a potential to improve parallelism.

Loop Inversion

Loop inversion [11] is a loop transformation that replaces a while loop by an if block containing a do..while loop. Loop inversion allows safe loop-invariant code motion to be performed.

2.3 Loop Optimizations in Reconfigurable Computing

Several research projects develop C-to-VHDL frameworks, trying to exploit as much as possible the advantages of reconfigurable systems by maximizing the parallelism in targeted applications and accelerating kernel loops in hardware. We identify two types of reconfigurable architectures: fine-grained and coarse-grained architectures.

2.3.1 Fine-Grained Reconfigurable Architectures

Fine-grained reconfigurable logic devices allow designers to specialize their hardware down to the bit level. The configurability makes devices, like FPGAs, suitable to implement a large variety of functions directly. This configurability comes at a significant cost in terms of circuit area, power, and speed.

GARP (*University of California, Berkeley*)

The Garp project was the first one to develop a C compiler that automatically partitioned sequential code between software and hardware, automatically generated hardware for the custom logic array as well as the interface logic to reconfigure the array dynamically during execution.

The Garp compiler proposes a solution for automatic compilation of sequential ANSI C code to hardware, and their approach is hardly influenced by the target platform, the theoretical Garp chip ([22], [21]) – a MIPS processor with a reconfigurable coprocessor. The Garp compiler uses techniques that are similar to those used by VLIW compilers, with the *hyperblock* as a unit for scheduling.

A hyperblock is formed from many adjacent basic blocks, usually including basic blocks from different alternative control paths. For optimization purposes, a hyperblock will typically include only the common control paths rather than the rarely taken paths. By definition, a hyperblock has a single point of entry, but may have multiple exits. The Garp compiler uses the hyperblock structure for deciding which parts of a program are executed on the reconfigurable coprocessor, as well as for converting the operations in the selected basic blocks to a parallel form suitable for the hardware.

The unit of code that gets accelerated in hardware is a loop. General loop nesting is not supported (loops implemented in hardware cannot have inner loops).

The Garp compiler uses SUIF as front-end and a control flow graph library for

generating the basic block representation of the code. The compiler does not seem to perform any special transformations on the loops before selecting the hyperblocks.

The NAPA project (*National Semiconductor*)

The NAPA 1000 chip was the first reconfigurable co-processor, consisting of a RISC core and CLAy reconfigurable logic from National Semiconductors. The NAPA C compiler [51, 52] is pragma based, allowing the user to tag regions of code to be placed in the hardware. The compiler performs semantic analysis of the pragma-annotated program and co-synthesizes a conventional program executable combined with a configuration bit stream for the adaptive logic. Compiler optimizations include synthesis of hardware pipelines from pipelineable loops. This methodology is most effective when the application shows '90/10' behavior – 90% of execution time in 10% of the code. By placing such compute intensive blocks in hardware, the NAPA could achieve 1-2 orders of magnitude performance improvement over DSP processors.

The Cameron project (*University of California, Riverside*)

The Cameron project (highly active between 1998 and 2003) proposes an extension to the C language (called SA-C) and a compiler that maps high-level programs directly onto FPGA as a solution for reconfigurable computing [98], [59]. The targeted applications are mainly in the domain of image processing.

SA-C was not conceived to be a stand-alone language. It is assumed that developers would rewrite selected loops and functions of existing C programs in SA-C and incorporate them in the original program. The SA-C compiler would first analyze these segments to find parallel loops that can be executed on an FPGA, and translate all sequential code to C and include it in the host program. After that, it maps them to hardware (note that SA-C does not support any file I/O operations, it is assumed that such operations will be carried out in C).

Internally, the SA-C compiler uses several intermediate representations to bridge the gap between high-level algorithmic programs and FPGA configurations. After parsing the SA-C code, it transforms the program into a Data Dependence and Control Flow (DDCF) graph, which has nodes for complex operators such as sliding a window of data across an image. During optimization passes, this graph is transformed such that subgraphs of simpler nodes replace the complex nodes, until a more traditional data flow graph is obtained. The last steps consist

of generating and optimizing an abstract hardware architecture graph.

The compiler performs transformations on the intermediary data flow graphs, by applying traditional optimizations, as well as optimizations specific for generating hardware circuits. Among traditional optimizations, there are: common subexpression elimination, constant folding, invariant code motion and dead code elimination. The compiler also does specialized variants of loop strip mining, array value propagation, loop unrolling, function inlining, lookup tables, loop body pipelining, and array blocking, along with loop and array size propagation analysis [61].

The FPGA specific optimizations are:

- Producer/Consumer Loop Fusion. The fusion of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures. In simple cases, where arrays are processed element-by-element, fusion is straightforward. For some other cases, when loop do not iterate over the same index domain or have different number of iterations, loop fusion is also possible by examining the data dependencies.

- Temporal Common Subexpression Elimination. The SA-C compiler performs conventional CSE (common subexpression elimination within a block of code), but it also looks for values in one loop iteration that were previously computed in other loop iterations. In such cases, the redundant computation can be eliminated by holding the values in registers so that they are available later and do not need to be recomputed.

- Window Narrowing. When performing Temporal CSE, it may happen that one or more columns in the left part of the window become unreferenced, making it possible to eliminate those columns. By narrowing the window, the space required to store the window's values on the FPGA is decreased.

- Pipelining. After multiple applications of fusion and unrolling, the resulting loop often has a long critical path, resulting in a low clock frequency. By adding stages of pipeline registers it is possible to break up the critical path and thereby boost the frequency. The SA-C compiler uses propagation delay estimates, empirically gathered for each of the Data Flow Graph (DFG) node types, to determine the proper placement of pipeline registers. The user specifies to the compiler the number of register stages to place.

Results presented in [59] show that in terms of design time, SA-C's productivity is 10 to 100 times of that of handwritten VHDL. Anyhow, there are penalties in area (use of SA-C can double the device area utilization) and execution time: handwritten designs have 10% - 20% higher maximum clock frequencies and typically save about 40% in clock cycles with respect to SA-C generated circuits. Overall, the execution time of a handwritten design is approximately 50% of that of SA-C version.

ROCCC (*University of California, Riverside*)

ROCCC is a C-to-hardware compilation project whose objective is the FPGA-based acceleration of frequently executed code segments (loop nests), built upon the knowledge and experience of SA-C and Streams-C. It relies on a profiling tool that uses GCC to obtain a program's basic block counts and identifies, after the execution, the most frequently executed loops that form the computation kernel.

The compiler transforms modules and systems written in C into synthesizable VHDL. Modules are written in a subset of C and transformed into parallel, pipelined VHDL. Modules compiled by ROCCC are then exported back to the user and may be used to build up larger modules and complete systems that interface with memory through user specified channels. The main features of ROCCC 2.0 are:

- The modular (LEGO-like) code design supporting code re-use and compiler generated modular redundancy for increased reliability.
- The separation of user application code from the interface to external devices.
- The possibility to import of hard or soft IP cores into modules written in C code.
- The support for dynamic partial reconfiguration.

The high level compiler transformations are implemented using the SUIF toolset [30]. The hardware generation and low level optimizations passes are implemented using the LLVM toolset [111]. The front-end is based on GCC 4.0.2 [109]. All the information used by the compiler, such as the available or compiled modules, is stored in a database.

ROCCC can be used to generate VHDL only from perfectly nested loops. ROCCC 2.0 currently does not support:

- more than two nested loops;
- logical operators that perform short circuit evaluation;
- generic pointers;
- memory accesses that are dynamically determined;
- non-module functions, including C-library calls;
- shifting by a variable amount;
- non-for loops;
- non-boolean select style if statements.

The compiler performs regular loop unrolling and strip-mining transformations on loop nests in order to use efficiently the available area and memory bandwidth of the reconfigurable device. The results in [77] show that compile-time area estimation can be done within 5% accuracy and in less than one millisecond.

The compiler exploits both instruction level and loop level parallelism and pipelines the loop body to be able to execute multiple loops simultaneously [58].

ROCCC 2.0 performs the following loop optimizations: normalization, invariant code motion, peeling, unrolling, fusion, tiling, strip mining, interchange, unswitching, skewing, induction variable substitution, and forward substitution.

PARLGRAN (*University of California, Irvine*)

PARLGRAN [12, 13] is an approach that tries to maximize performance on reconfigurable architectures by selecting the parallelism granularity for each individual data-parallel task. The authors target task chains and make a decision on the parallelism granularity of each task. The task instances have identical area requirements but different workloads, which translates into different executions time (a task is split into several sub-tasks). The proposed algorithm takes into consideration the physical (placement) constraints and reconfiguration overhead at run-time, but without taking into account the memory bottleneck problem. Parallel execution of software and hardware is not considered.

2.3.2 Coarse-Grained Reconfigurable Architectures

Coarse-grained reconfigurable architectures have potential advantages in terms of circuit area, power, and speed. Many researchers have pursued the use of

Coarse-Grained Reconfigurable Arrays (CGRAs) considering that they would be easier targets for higher level development tools. However, as pointed out by Hartenstein [62], the challenge with the use of CGRAs is that there is no universal form that is effective for all applications. To be more efficient, CGRAs are optimized in terms of operators and routing for some specific problem domain or set of domains. Thus, an application must be well matched with the array to realize dramatic improvements over other reconfigurable solutions.

RaPiD (*University of Washington*)

The Reconfigurable Pipelined Datapath (RaPiD) [42, 43] aims at speeding up highly regular, computation-intensive tasks by deep pipelines on its linear array of data path units.

RaPiD is currently programmed using Rapid-C, a language that allows the programmer to schedule computations to the architecture while hiding most of the details of the architecture. In Rapid-C, extensions to the C language (like synchronization mechanisms and conditionals to identify first or last loop iteration) are provided to explicitly specify parallelism, data movement and partitioning. Rapid-C programs may consist of several nested loops describing pipelines. Outer loops are transformed into sequential code for address generators and inner loops are transformed into structural code for the RA. The Rapid-C compiler generates and optimizes the configured control circuits for the program.

PipeRench (*Carnegie Mellon University*)

PipeRench is one of the architectures proposed for reconfigurable computing, together with an associated compiler [19], [53]. PipeRench is particularly suitable for stream-based media applications or any applications that rely on simple, regular computations on large sets of small data elements.

The compiler is designed to trade off configuration size for compilation speed. For example, when handling the 1D discrete cosine transform, 4 times more bit operations are generated than when using the Xilinx tools. However, the compilation with the PipeRench compiler takes only 2.4 seconds, compared with 75 minutes with the Xilinx tools. The compilation process begins from an architecture description file. The source language is a Dataflow Intermediate Language (DIL), which is a single assignment language that can be easily used by the programmers, or as an intermediate language in a high level language compiler.

The compiler flow is as follows. After parsing, all modules are inlined and all loops are unrolled. Afterwards, a straight-line, single-assignment program is generated, which is converted into a graph upon which the rest of compiler operates. The compiler determines each operator's size and decomposes high-level operators and operators that exceed the target cycle time. There follows an operator recomposition pass that uses pattern matching to find subgraphs that it can map to parametrized modules. These modules take advantage of architecture-specific routing and PE capabilities to produce a more efficient set of operators.

Finally, a place-and-route algorithm is used to generate hardware configurations. This algorithm is a deterministic linear-time greedy algorithm and it represents the key to the compiler's speed.

Among the optimizations performed by the compiler, there are: common subexpression elimination, dead code elimination, bit level constant propagation, algebraic simplifications, register reduction and interconnection simplification.

ADRES (*IMEC*)

ADRES (Architecture for Dynamically Reconfigurable Embedded System) [33, 94] is a flexible architecture template that consists of a tightly coupled VLIW processor and a coarse-grained reconfigurable array; an architecture instance can be described in an XML-based description language and it is used by the compiler in the scheduling step. DRESC(Dynamically Reconfigurable Embedded System Compiler) is a retargetable C-compiler framework, which includes the compiler, the binary utilities (assembler and linker), and several simulators and RTL generators for the ADRES architecture.

When compiling an application for ADRES, the kernels which will be mapped on the reconfigurable array are identified in the profiling/partitioning step. In order to make the kernels pipelineable, some transformations are performed on the source code. On the intermediate representation, dataflow analysis and optimization and scheduling are performed.

In order to parallelize the kernels, the compiler performs software pipelining, using a modulo scheduling algorithm described in [95]. Traditional ILP scheduling techniques are applied to discover the available moderate parallelism for the non-kernel code.

The modulo scheduling algorithm by the DRESC compiler utilizes a data dependency graph representing the loop body and a graph-based architecture representation, called MRRG (Modulo Routing Resource Graph). MRRG

is needed in order to model resources in a unified way, to expose routing possibilities and to enforce modulo constraints. The algorithm combines ideas from FPGA placing and routing and VLIW modulo scheduling. It is based on congestion negotiation and simulated annealing methods. Starting from an invalid schedule that overuses resources, it tries to reduce overuse over time until a valid schedule is found.

The results show 18.8-44.8% utilization of Functional Units and 12-28.7% instructions per cycle. The speed-up reported for a MPEG2-decoder over an 8-issue VLIW is about 12 times for kernels and 5 times for the entire application.

The limitations of this modulo scheduling algorithm are that it can handle only the innermost loop of a loop nest and it cannot handle some architecture constraints (pipelined FUs and limited register files).

2.4 LLP Exploitation

Loop nests are the main source of potential parallelism, and loop level parallelization has been widely used for improving performance [15]. Automatic vectorization has long been shown a successful way of exploiting data parallelism for vector processors and supercomputers [152].

Following the same idea, modern microprocessors provide Single Instruction Multiple Data (SIMD) extensions to enable exploitation of data parallelism, characteristic of multimedia and scientific computations. These extensions perform the same operation simultaneously on multiple data elements packed in a SIMD word. Examples of such SIMD extensions include Intel®'s MMX™ [106] and SSE [114], AMD's 3DNow!™ [100], and IBM's AltiVec [37]. With the increasing computational demand for embedded applications, processors for embedded system also started featuring SIMD extensions, such as ARM®'s NEON™ [9]. Furthermore, the IBM/Sony/Toshiba Cell® processor [67] introduced the Synergistic Processing Elements (SPE) core [57] that operates exclusively in a SIMD fashion, without hardware support for scalar operations.

Software vectorization happens typically at loop level. In [14], loop peeling and loop distribution are proposed for transforming loops with data parallelism to enable vectorization. The work of Weinhardt and Luk [137–139], known as pipeline vectorization, tries to extract parallelism from a sequential program and exploit this parallelism in hardware and it is based on software vectorizing compilers. Auto-vectorization is performed also by the GCC compiler [50].

Parallelizing compiler prototypes have also provided valuable research results on loop transformations, such as Tiny [144] and Suif [30, 140]. Pips [8, 110], LooPo [55, 81, 101], LLVM [56, 111], and PoCC [46, 107, 108] are continuously developed and have adopted the polyhedral representation of loops. There are also a number of High-Level Synthesis tools that focus on efficient compilation of loops and are based on the polyhedral model: Alpha [36], Compaan [126], Paro [17].

In the domain of embedded architectures, [27] is a good source of references to loop transformations for LLP and other work in the embedded compiler and architecture community. In [73], the authors discuss how loop-level parallelism in embedded applications can be exploited in hardware and software. Specifically, it evaluates the efficacy of automatic loop parallelization and the performance potential of different types of parallelism, namely, true thread-level parallelism (TLP), speculative thread-level parallelism and vector parallelism, when executing loops. Additionally, it discusses the interaction between parallelization and vectorization. Applications from both the industry-standard EEMBC® 1.1, EEMBC 2.0 and the academic MiBench embedded benchmark suites are analyzed using the Intel® C compiler.

Bathen et al. [16] propose a methodology for discovering and enabling parallelism opportunities using code transformations such as loop fission, loop tiling, loop unrolling, as well as task/kernel level pipelining. The application's tasks are decomposed into smaller units of computation called kernels, which are distributed and pipelined across the different processing resources. Inter-kernel data reuse is taken into account, with the purpose of minimizing unnecessary data transfers between kernels, and load-balancing is performed in order to reduce power consumption and improve performance.

Targeting a multicore system, Zhong et al. [150] show that substantial amounts of loop level parallelism is available in general purpose applications, but it is difficult to see because of data and control dependencies. In addition to traditional techniques, the authors propose three new transformations to deal with dependencies: speculative loop fission, speculative prematerialization, and isolation of infrequent dependencies. A control-aware memory profiler is used to identify the cross iteration dependencies and the candidate loops.

Loop level parallelism has been studied also in the context of reconfigurable architectures. The ROCCC compiler, described in Section 2.3.2, performs well-known transformations meant to aggressively optimize loops. For the ROCCC compiler, loops are seen as kernels, and not as kernel containers, which is the case in our work.

Liu et al. [87] proposed a geometric framework for exploring data reuse (buffering) and loop level parallelism under the on-chip memory constraint in the context of FPGA-targeted hardware compilation, within a single optimization step. The data reuse decision is to decide at which levels of a loop nest to insert new on-chip arrays to buffer reused data for each array reference. The parallelization decision is to decide an appropriate strip-mining for each loop level in the loop nest, allowing a parametrized degree of parallelism.

Derrien et at. [35] use loop skewing and serialization (clustering) for pipelining the data-path, for improving array performance. The arrays are synthesized for FPGAs. A very fine grained parallelism is exploited in this paper, by pipelining down to the logic cell level.

The work presented in this thesis is different than other works targeting LLP. Loop optimizations for embedded systems usually have a very fine granularity, while we consider that kernels are large functions, and can also be 'black boxes', when only the VHDL implementation is available. The main concerns when compiling for embedded platforms are memory and power consumptions, while the work presented in this dissertation focuses more on performance in terms of speedup, and on reconfiguration.

Loop optimizations for reconfigurable architectures usually focus on aggressively optimizing loops that are seen as kernels, pipelining the data-path, and in some cases making use of vectorizing compilers. The granularity of our approach is different, as we consider loops to be containers for kernels. The performance gain in our work comes from executing several kernels in parallel, and not from maximizing the parallelism within a kernel.

2.5 Open Issues

The general strategy for accelerating an application that is partially mapped on reconfigurable hardware is to find the most time consuming parts of that application (namely, the application kernels), and aggressively optimize them while generating the corresponding hardware code. When the kernels are (nested) loops, the loop optimizations presented in this chapter are used to transform the loop prior to generating the hardware. This strategy exploits the instruction level parallelism of the applications. Task parallelism is generally not sought in conjunction with loops. An example of loop level parallelism in the related work is that in the ROCCC project, where multiple loops can be executed simultaneously. This differs from our target, the parallel execution of tasks inside loops.

There are several aspects that have not received adequate attention in literature as a source of performance improvement.

First, coarse grain loop level parallelism (parallel execution of tasks inside loops) is mostly associated with TLP. For a single threaded application, optimizing compilers usually focus on a given instruction set and try to exploit the VLIW or SIMD parallelism. In the case of RAs, loops usually represent the kernels and the hardware/software compiler must generate efficient hardware code for mapping the loop onto the reconfigurable processor. K-loops are considered to be kernel containers, where the kernel hardware implementation is already available and not subject to change, or where the kernel hardware implementation is automatically generated and not hand-tuned. Optimizing such K-loops gives another level of optimization, which can boost up performance regardless of how optimized the application kernels' implementations are.

Second, the exploitation of software-hardware loop level parallelism, where a part of a loop (e.g. a kernel) is accelerated on the hardware, while the rest of the loop executes on the GPP, is usually considered with a different granularity.

Last, partial reconfigurability is one of the topics of interest in the reconfigurable community. However, not much work has been carried in the context of partial hardware mapping. By partial hardware mapping we refer to the fact that only the kernels are mapped on the reconfigurable hardware, while the rest of the code contained in the K-loop body will always execute on the GPP. The rest of the loop body will execute in software.

In this thesis, we address coarse grain LLP, software-hardware LLP and partial reconfigurability at loop level. For this purpose, we make use of some of the traditional loop transformations presented earlier in this chapter. However, K-loops must be viewed from a perspective that differs from that of normal loops, because the parallelism inside them is coarse-grained. In traditional computing, some of the most common scenarios for speeding up an application is to use loop transformations that improve cache hits and/or reduce branching. In the context of K-loops, loop transformations that might not be interesting in traditional computing because of the large overhead they introduce might be in our advantage. An example is the loop fission transformation, which may require the use of (large) intermediate buffers to communicate the data from one sub-loop to another.

The following loop transformations have been the subject of the research presented in this thesis, and they can improve K-loops performance. Loop unrolling (Chapter 4) will be used for exposing intra-loop parallelism, which will enable different instances of the same task (kernel) to execute in parallel in the hard-

ware. Loop shifting (Chapter 5) and pipelining (Chapter 6) will be used for breaking intra-iteration dependencies and enable software-hardware parallelism. Loop skewing (Chapter 8) will also be used for breaking wavefront-like dependencies. Loop distribution (Chapter 7) will be used for breaking large loops into smaller sub-loops that could be accelerated more easily. Partial reconfiguration may be used between the sub-loops.

3
Methodology

IN THIS CHAPTER we detail the notion of K-loop, a loop containing in the loop body one or more kernels mapped on the reconfigurable hardware. Later on, we present the framework that helps determine the optimal degree of parallelism for each hardware mapped kernel within a K-loop. Optimizing K-loops with the methods presented in the following chapters answers to the open issues that we have identified in the previous chapter: coarse grain loop level parallelism, software-hardware parallelism, and considering partial reconfigurability while accelerating parts of a loop nest.

Assumptions regarding the K-loops framework and the targeted applications are presented in Subsection 3.1.2. General notations that are used in the mathematical models of the following chapters are presented in Section 3.3.1, while the area, memory and performance considerations for the proposed methods are presented in Subsection 3.3.2, Subsection 3.3.3, and Subsection 3.3.4, respectively. In Subsection 3.3.5, Amdahl's Law is adapted to our framework such that we can compare the speedup achieved by loop unrolling or unrolling plus shifting, etc., with the theoretical maximum speedup assuming maximum parallelism for the hardware. In Section 3.4, the random test generator used for showing the potential improvement of K-pipelining and loop distribution is presented. An overview of the proposed loop optimizations, including an algorithm that can be used to decide which transformations to use for a given K-loop, is given in Section 3.5.

3.1 Introduction

In this section, we detail the notion of K-loops and introduce some general assumptions regarding the targeted applications and the framework.

3.1.1 The Definition of K-loop

Within the context of Reconfigurable Architectures, we define a **kernel loop** (K-loop) as a loop containing in the loop body one or more kernels mapped on the reconfigurable hardware. The loop may also contain code that is not mapped on the FPGA and that will always execute on the GPP (in software). The software code and the hardware kernels may appear in any order in the loop body. The number of hardware mapped kernels in the K-loop determines its size.

```
for (i = 0; i < N; i++) do
    /* a pure software function                */
    SW(blocks[i]);
    /* a hardware mapped kernel                */
    K(blocks[i]);
end
```

Figure 3.1: A K-loop with one hardware mapped kernel.

A simple example of a size one K-loop is illustrated in Figure 3.1. In our work, we focus on improving the performance for such loops by applying standard loop transformations to maximize the parallelism inside the K-loop. Loop unrolling is used for hardware parallelism, as shown in Chapter 4. Loop shifting, presented in Chapter 5, and K-pipelining, presented in Chapter 6, are used for hardware-software parallelism and they can be combined with loop unrolling. Loop skewing, presented in Chapter 8, is used for eliminating data dependencies for wavefront-like applications and is combined with loop unrolling for parallelizing. Loop distribution, presented in Chapter 7, is used to split large K-loops into smaller K-sub-loops where the mentioned techniques can be applied in a more efficient manner to the K-sub-loops.

3.1.2 Assumptions Regarding the K-loops Framework

Our assumptions regarding the application and the framework can be divided into three categories: K-loop structural assumptions, memory assumptions and area assumptions.

At the beginning of our research, the K-loop structure was restricted to perfectly nested loops with no inter-iteration dependencies. Nevertheless, our research has shown that some inter-iteration dependencies can also be included in the

analysis since they can be eliminated by using loop transformations such as loop reversal, loop interchange, loop skewing, and so on. The requirement that the loop bounds are known at compile time stays, as the loop dimensions can have a large impact on the achievable speedup and on the chosen degree of parallelism.

There are several assumptions regarding the memory accesses. First, we assume that all the necessary data are read by the kernel at the beginning. After that, the kernel processes them, and, finally, it writes the output data. This model is needed to simplify the estimation of the number of kernel instances running in parallel for which the memory bandwidth will becomes the bottleneck. For this same reason, but also for architectural reasons, there are no memory transactions performed in parallel (e.g. two kernel instances reading at the same time from the memory, or one reading and another one writing). For a more accurate model, the polyhedral representation of loops can be used. This is a subject of future research. More details about the memory model can be found in Subsection 3.3.3.

Another assumption regarding the memory is that there is no need for external memory, and all external data needed by the kernel is available in the on-chip shared memory. This is important for the estimation of the speedup, since external memories have different access times. The use of an external memory is not within the scope of this thesis. Nevertheless, it will be addressed in our future work. Kernel's private data are stored in the FPGA's local memory, as decided by the VHDL generator.

As far as it concerns the area and the placement, the main assumption is that the design fits on the available area without creating routing issues, thus shape of the design is not considered. However, only 90% of the total area can be used by the Custom Computing Units (CCUs), in order to avoid the routing issues. More details about the area limitations can be found in Subsection 3.3.2. Our methods based on loop unrolling, shifting, pipelining and skewing do not require that the area is reconfigured while the K-loop executes. Therefore, we assume that the hardware configuration is performed well in advance (by a scheduler), such that the configuration latency is hidden. When loop distribution is used to split a K-loop into several K-sub-loops, the initial configuration latency is not taken into account.

These assumptions are summarized in Table 3.1.

Table 3.1: Assumptions for the experimental part.

K-loop structure:
⋆ no inter-iteration dependencies, except for wavefront-like dependencies; ⋆ K-loop bounds known at compile time.
Memory accesses:
⋆ memory reads/writes at the beginning/end; ⋆ on-chip memory shared by the GPP and the CCUs used for program data; ⋆ all necessary data available in the shared memory; ⋆ all memory reads/writes from/to the shared memory performed sequentially; ⋆ kernel's local data stored in the FPGA's local memory, but not in the shared part of the memory.
Area & placement:
⋆ shape of design not considered; ⋆ initial hardware configuration decided by a scheduling algorithm, such that the configuration latency is hidden.

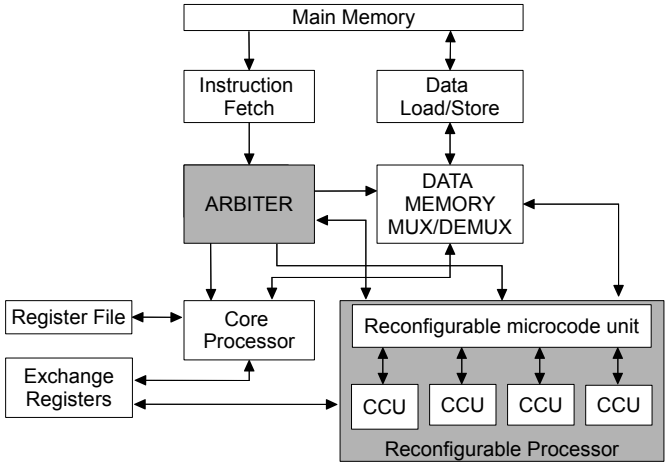


Figure 3.2: The Molen machine organization.

3.2 Framework

The work presented in this thesis is part of the Delft WorkBench (DWB) project [34]. The DWB is a semi-automatic toolchain platform for integrated hardware-software co-design in the context of Custom Computing Machines (CCM), which targets the Molen polymorphic machine organization [131] and the Molen programming paradigm [130]. The Molen programming paradigm is a paradigm that offers an abstraction of the available resources to the programmer, together with a model of interaction between the components. The Molen machine organization is illustrated in Figure 3.2. By using a 'one time' architectural or operating system extension, the Molen programming paradigm allows for a virtually infinite number of new hardware operations to be executed on the reconfigurable hardware.

The DWB supports the entire design process, as follows. The kernels are identified in the first stage of profiling and cost estimation. Next, the Molen compiler [103] generates the executable file, and replaces the function calls to the kernels implemented in hardware with specific instructions for hardware reconfiguration and execution, according to the Molen programming paradigm. An automatic tool for hardware generation, the Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) [149], is used to transform the selected kernels into VHDL code targeting the Molen platform.

3.3 Theoretical Background

In this section, the methodology used in the following chapters of the thesis is introduced. Since the main performance improvement in our work comes from parallel execution of kernel instances on the reconfigurable hardware, different aspects that limit the degree of parallelism need to be taken into account. First, the degree of parallelism is limited by the available area. Second, the degree of parallelism is limited by the memory bandwidth, since this usually is a bottleneck. Last, we impose an extra bound to the degree of parallelism, which is given by the achievable speedup, more exactly by a trade-off between speedup and area consumption.

The way Amdahl's Law for parallelization is applied to our work for determining the theoretical maximum achievable speedup is presented in the end of this section.

Before introducing the different constraints that will be taken into account, the main notations used throughout this thesis are listed.

3.3.1 General Notations

The notations used in the methodology and mathematical models for various loop transformations fall mainly into two categories: kernel related notations and K-loop related notations. The basic notations, common to all mathematical models presented in the following chapters, are presented in this section. The notations introduced for specific loop transformations are presented in the corresponding chapter.

Note that throughout this thesis, we refer to the execution time of various functions, or to the K-loop's execution time. When in the context of K-loops, the execution time always refers to the number of cycles, as all the notations in this section refer to the number of cycles.

Kernel related notations:

- T_{r} — the number of cycles required by a kernel K running in hardware to read all the necessary data from the memory;
- T_{w} — the number of cycles required by a kernel K running in hardware to write all the necessary data to the memory;

- $T_{\text{min(r,w)}}$/$T_{\text{max(r,w)}}$ – the minimum/maximum of T_{r} and T_{w}:

$$T_{\text{min(r,w)}} = \texttt{min}\,(T_{\text{r}}, T_{\text{w}}) \tag{3.1}$$

$$T_{\text{max(r,w)}} = \texttt{max}\,(T_{\text{r}}, T_{\text{w}})\,; \tag{3.2}$$

- T_{c} — the number of cycles required by a kernel K running in hardware to perform the computations;

- $T_{K\text{(sw)}}$/$T_{K\text{(hw)}}$ — the number of cycles required by a kernel K running in software/hardware (either the average of several instances or the maximum, depending on the case). In particular:

$$T_{K\text{(hw)}} = T_{\text{r}} + T_{\text{w}} + T_{\text{c}}\,; \tag{3.3}$$

- $T_{K\text{(hw)}}(u)$ — the number of cycles required by u instances of the kernel K running in hardware.

K-loop related notations:

- M, N — the dimensions of a nested K-loop before any transformation. Without loss of generality, we assume that $M \leq N$. In the case of simple K-loops, $M = 1$;

- R — the reminder of the division of N by the unroll factor u;

- T_{sw} — the number of cycles required by one instance of the software function (the function that is always executed by the GPP – in our example, the `SW` function);

- T_{prolog} — the number of cycles for the prologue of a transformed K-loop (when applying loop shifting or K-pipelining for instance);

- T_{body} — the number of cycles for the loop body of a transformed K-loop;

- T_{epilog} — the number of cycles for the epilogue of a transformed K-loop;

- $T_{\text{loop(sw)}}$ — the number of cycles for the K-loop executed completely in software. Note that it does not depend on the unroll factor. For a simple K-loop, the software time is:

$$T_{\text{loop(sw)}} = (T_{\text{sw}} + T_{K\text{(sw)}}) \cdot N \cdot M\,; \tag{3.4}$$

- $T_{\text{unroll}}(u)$ — the number of cycles for the K-loop when parallelized with loop unrolling, with unroll factor u and the kernel instances running in hardware;
- $S_{\text{unroll}}(u)$ — the K-loop speedup when parallelized with loop unrolling, with unroll factor u;
- $T_{\text{shift}}(u)$ — the number of cycles for the K-loop when parallelized with loop unrolling and shifting, with unroll factor u and the kernel instances running in hardware;
- $S_{\text{shift}}(u)$ — the K-loop speedup when parallelized with loop unrolling and shifting, with unroll factor u;
- $T_{\text{pipe}}(u)$ — the number of cycles for the K-loop when parallelized with loop unrolling and K-pipelining, with unroll factor u and the kernel instances running in hardware;
- $S_{\text{pipe}}(u)$ — the K-loop speedup when parallelized with loop unrolling and K-pipelining, with unroll factor u;
- T_{distr} — the number of cycles for the K-loop when transformed via the distribution algorithm, with the kernel instances running in hardware;
- S_{distr} — the K-loop speedup when transformed via the distribution algorithm;
- $T_{\text{h/h}}(u)$ — the number of cycles for the skewed K-loop when all kernel instances are running in hardware, assuming the unroll factor u;
- $T_{\text{h/s}}(u)$ — the number of cycles for the skewed K-loop when part of the kernels are running in hardware and part in software, assuming the unroll factor u;
- $S_{\text{h/h}}(u)$ — the speedup for the skewed K-loop when all the kernels instances are running in hardware, assuming the unroll factor u.

$$S_{\text{h/h}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{h/h}}(u) + T_{\text{sw}} \cdot M \cdot N}; \tag{3.5}$$

- $S_{\text{h/s}}(u)$ — the speedup for the skewed K-loop when part of the kernels are running in hardware and part in software, assuming the unroll factor u:

$$S_{\text{h/s}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{h/s}}(u) + T_{\text{sw}} \cdot M \cdot N}. \tag{3.6}$$

Notations for K-loops with more than 1 kernel:

- N_f — the K-loop's number of functions;
- N_k — the number of kernels, which is half of the number of functions: $N_k = \lfloor N_f/2 \rfloor$;
- $\mathtt{K}_i$ — the i-th kernel in the K-loop;
- $\mathtt{SW}_i$ — the i-th software function in the K-loop (the pre- function of $\mathtt{K}_i$);
- T_{K_i} — the number of cycles for $\mathtt{K}_i$ running in hardware;
- T_{SW_i} — the number of cycles for $\mathtt{SW}_i$.

3.3.2 Area considerations

The hardware implementation of each of the selected kernels required a certain amount of area on the reconfigurable device. As the area available on the reconfigurable hardware is always limited, area becomes the first limitation to parallelism. Additionally, although a 100% use of the reconfigurable area may allow a higher degree of parallelism than a partial use of the reconfigurable device, it may cause routing issues. For this reason, we aim at using no more than 90% of the available area.

By taking into account the area constraints and by assuming no constraint regarding the placement of the kernel, we can devise an area-related upper bound for the unroll factor as follows:

$$\mathrm{u_a} = \left\lfloor \frac{Area_{(available)}}{Area_{(K)}} \right\rfloor \text{, where:} \tag{3.7}$$

Where:

- $Area_{(available)}$ is the available area, taking into account the resources utilized by Molen and by other configurations; We assume that the overall interconnect area grows linearly with the number of kernels;
- $Area_{(K)}$ is the area utilized by one instance of the kernel, including the storage space for the values read from the shared memory. All kernel instances have identical area requirements.

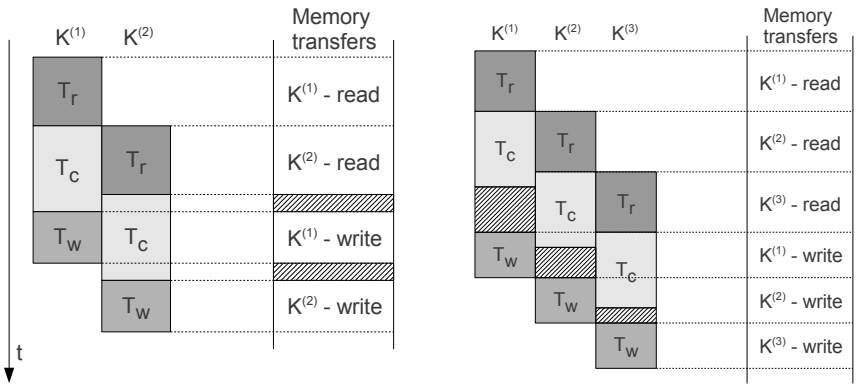
3.3.3 Memory Considerations

Ideally, all data would be available immediately at request and the degree of parallelism would be limited only by the available area. However, for many applications, the memory bandwidth is an important bottleneck in achieving the maximum theoretical parallelism. As specified in Section 3.3.1, T_r, T_w, and T_c are the number of cycles required by a kernel K running in hardware to read data, to write data, and to perform the computations, respectively, as indicated by the profiling information. The number of cycles for one instance of the kernel K on the reconfigurable hardware $T_{K(hw)}$ is the sum of the total number of cycles for reading, writing, and performing the computations.

As specified in 3.1.2, we assume that within any kernel K, the memory reads are performed at the beginning and memory writes in the end. This is needed in order to simplify the mathematical model, which gives the estimation of the memory bound and the performance estimation. For a more accurate model, the polyhedral representation of loops can be used. Hu et al. [63] give a high-level memory usage estimation, including accesses and size, that is used for global loop transformations. Then, as illustrated in Figure 3.3,a new instance of K can start only after a time T_r. The kernel instances are denoted in Figure 3.3 by $K^{(1)}$, $K^{(2)}$, etc.

There are several possible cases to consider regarding the relation between T_r, T_w, and T_c. Figure 3.3 represents only one of them, more precisely the case when $T_w \leq T_r < T_c$ and $T_w + T_r > T_c$. However, in all cases, the performance stops increasing at the moment when the computation is fully overlapped by the memory transfers performed by the kernel instances running in parallel. We denote with u_m the unroll factor where this case happens. As a result, u_m sets another bound for the degree of unrolling on the reconfigurable hardware. A further increase of the unroll factor gives a converse effect when computation stalls occur due to waiting for the memory transfers to finish, as it can be seen in Figure 3.3(b).

Note that this is actually the worst case scenario in our analysis and our algorithm will always give results which are on the safe side. More precisely, using the above assumption to compute the unroll factor bound u_m with respect to the assumed memory accesses is just a very quick worst case estimation, which does not require further study of the target application. Moreover, determining u_m with this assumption guarantees that for any unroll factor which is less than u_m, there will be no computation stalls due to the memory accesses/transfers.Depending on the hardware implementation, the real threshold value u_m for the unroll factor regarding memory transfers will be more re-



(a) $T_{K(hw)}(u) = T_c + T_w + u \cdot T_r$ (b) $T_{K(hw)}(u) = u \cdot (T_r + T_w)$

Figure 3.3: Parallelism on the reconfigurable hardware. In this case, $T_w \leq T_r < T_c$ and $T_w + T_r > T_c$.

laxed than in our estimation. The time for running u instances of K on the reconfigurable hardware is:

$$T_{K(hw)}(u) = \begin{cases} T_c + T_{min(r,w)} + u \cdot T_{max(r,w)}, & \text{if } u \leq u_m \\ u \cdot (T_r + T_w), & \text{if } u > u_m \end{cases} \tag{3.8}$$

The speedup at kernel level is $S_K(u) = \dfrac{u \cdot T_{K(hw)}(1)}{T_{K(hw)}(u)}$. For $u > u_m$, the speedup is:

$$S_K(u > u_m) = \frac{u \cdot (T_r + T_w + T_c)}{u \cdot (T_r + T_w)} = \frac{T_r + T_w + T_c}{T_r + T_w}, \tag{3.9}$$

which means that the speedup is constant. Therefore it is not worth to unroll more. Performance increases with the unroll factor if the following condition is satisfied:

$$T_c + T_{min(r,w)} + u \cdot T_{max(r,w)} < u \cdot (T_{min(r,w)} + T_{max(r,w)}). \tag{3.10}$$

The memory bound can be derived as follows:

$$u \leq u_m = \left\lfloor \frac{T_c}{T_{min(r,w)}} \right\rfloor + 1 \tag{3.11}$$

When applied to the example in Figure 3.3, $u_m = 2$. The case $u \leq u_m$ corresponds to Figure 3.3(a) and the case $u > u_m$ corresponds to Figure 3.3(b).

In our example, $T_w \leq T_r$. As a result, $T_{max(r,w)} = T_r$ and $T_{min(r,w)} = T_w$. In Figure 3.3(a), the time for running in parallel two kernel instances ($K^{(1)}$ and $K^{(2)}$) is given by the time for $K^{(1)}$, $T_c + T_r + T_w$, plus the necessary delay for $K^{(2)}$ to start (T_r). In Figure 3.3(b), $K^{(1)}$ writing to memory is delayed because of $K^{(3)}$ reading from memory. In this case, the actual kernel computation is hidden by the memory transfers and the hardware execution time depends only on the memory accesses ($u \cdot (T_r + T_w)$).

3.3.4 Performance Considerations

When multiple kernels are mapped on the reconfigurable hardware, the goal is to determine the optimal unroll factor for each kernel, which would lead to the maximum performance improvement for the application. For this purpose, we introduce a new parameter in the model: the calibration factor F, a positive number decided by the application designer, which determines a limitation of the unroll factor according to the targeted trade-off. For example, one may not want to increase the unrolling, if the gain in speedup would be with a factor of 0.1%, but the area usage would increase with 15%. The simplest relation to be satisfied between the speedup and necessary area is:

$$\Delta S(u+1, u) \geq \Delta A(u+1, u) \cdot F \tag{3.12}$$

where $\Delta A(u+1, u)$ is the relative area increase, which is constant since all kernel instances are identical:

$$\Delta A(u+1, u) = A(u+1) - A(u) = Area_{(K)} \in (0, 1) \tag{3.13}$$

and $\Delta S(u+1, u)$ is the relative speedup increase between consecutive unroll factors u and $u+1$:

$$\Delta S(u+1, u) = \frac{S(u+1) - S(u)}{S(u)}. \tag{3.14}$$

Note that only in the ideal case $\frac{S(u+1)}{S(u)} = \frac{u+1}{u}$. This means that

$$S(u+1) < 2 \cdot S(u), \ \forall u \in \mathbb{N}, \ u > 1 \tag{3.15}$$

and the relative speedup satisfies the relation:

$$\Delta S(u+1, u) \in [0, 1), \ \forall u \in \mathbb{N}, \ u > 1. \tag{3.16}$$

Thus, F is a threshold value which sets the speedup bound for the unroll factor (u_s). How to choose a good value for F is not within the scope of this research. However, it should be mentioned that a greater value of F would lead to a lower bound, which translates to 'the price we are willing to pay in terms of area compared to the speedup gain is small'. Additionally, the upper limit for F is $F_{max} = \frac{\Delta S(2,1)}{Area_{(K)})}$. If $F = F_{max}$, then the unroll factor will be $u = 2$. The case $F > F_{max}$ incurs that $u = 1$, which means that no unroll will be performed. In conclusion, the possible values of F are:

$$F \in \left[0, \frac{\Delta S(2,1)}{Area_{(K)})}\right]. \tag{3.17}$$

The speedup bound is defined as:

$$u_s = \min(u) \quad \text{such that} \quad \Delta S(u_s + 1, u_s) < F \cdot Area_{(K)}. \tag{3.18}$$

Local optimal values for the unroll factor u may appear when u is not a divisor of N, but $u+1$ is. To avoid this situation, as S is a monotonic increasing function for $u < u_m$, we add another condition for u_s:

$$\Delta S(u_s + 2, u_s + 1) < F \cdot Area_{(K)}. \tag{3.19}$$

When the analyzed kernel is the only one running in hardware, it might make sense to unroll as much as possible, given the area and memory bounds (u_a and u_m), as long as there is no performance degradation. In this case, we set $F = 0$ and $u_s = u_m$.

Binary search can be used to compute in $O(\log N)$ time at compile-time the value of u_s. The value of u_s must satisfy the conditions $\Delta S(u_s + 1, u_s) < F \cdot Area_{(K)}$ and $\Delta S(u_s + 2, u_s + 1) < F \cdot Area_{(K)}$.

Figure 3.4 illustrates the speedup achieved for different unroll factors for the DCT kernel, as presented in [40]. The area for one instance of DCT represents approximately 12% of the area on Virtex-II Pro. With respect to these data, Figure 3.5 shows how different values of F influence the unroll factor. Note that in this case, $\frac{\Delta S(2,1)}{Area_{(K)})} = 6.23$, which means that for $F > 6.23$ there will be no unroll (meaning that $u_s = 1$).

3.3.5 Maximum Speedup by Amdahl's Law

For each analyzed kernel, we must compare the speedup achieved when using the proposed methods with the theoretical maximum achievable speedup using

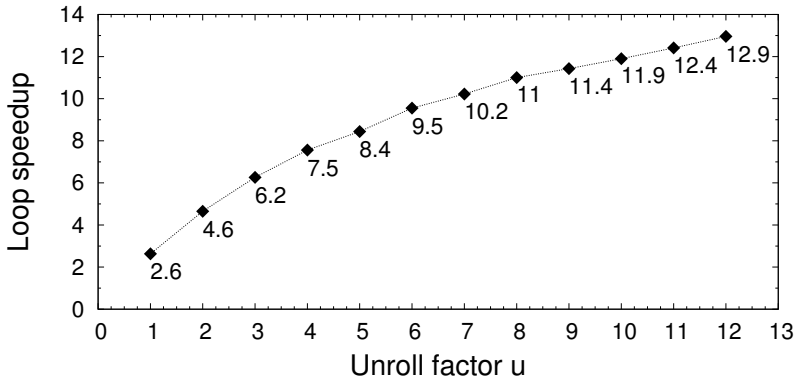


Figure 3.4: DCT K-loop speedup.

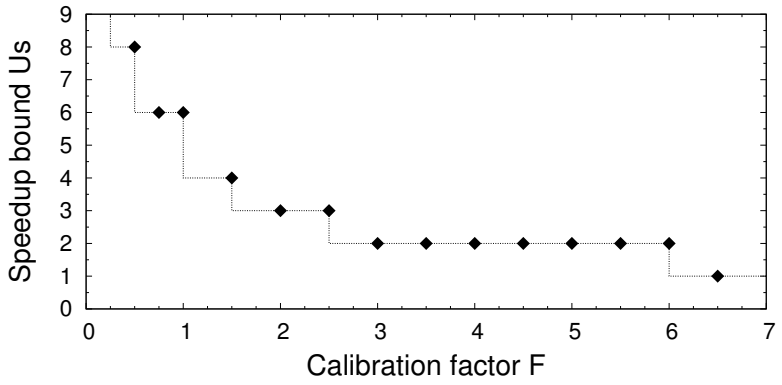


Figure 3.5: Influence of the calibration factor F on the speedup bound u_S.

Amdahl's Law for parallelization. When computing the value with Amdahl's Law, we consider maximum parallelism for the kernels executed in hardware (*e.g.*, full unroll). The serial part of the program (the part that cannot be parallelized) consists of the N instances (N is the number of iterations in the K-loop) of the software function `SW` plus one instance of the kernel running in hardware. The parallelizable part consists of the N instances of the kernel running in software minus one instance of the kernel running in hardware, because the execution time of N kernels in hardware is at least the time for running a single kernel — and that is the ideal case. We denote by P_{K} the percentage of the loop time that can be parallelized:

$$P_{\mathrm{K}} = \frac{N \times T_{K(\mathrm{sw})} - T_{K(\mathrm{hw})}}{T_{\mathrm{loop(sw)}}} \tag{3.20}$$

From now on, we will refer to the computed theoretical maximum speedup using Amdahl's Law as **the speedup by Amdahl's Law**, which is:

$$S_{Amdahl} = \frac{1}{1 - P_{\mathrm{K}} + \frac{P_{\mathrm{K}}}{N}} \tag{3.21}$$

Note that Amdahl's Law neglects potential bottlenecks, such as memory bandwidth.

3.4 The Random Test Generator

In order to study the impact of various loop transformations over K-loops, more test data than the available ones are needed. While new programming models that are emerging to fit the multicore era and future applications in various domains (such as audio-video) are more likely to have large K-loops, not many of the present applications meet our requirements and, thus, the need for a test generator arose. A random test generator has been developed in order to be able to study the potential for performance improvement of K-pipelining (Chapter 6) and loop distribution (Chapter 7) and possibly other transformations, which are part of the future work. The generated tests simulate K-loops of any size (the size can be specified at compile time or at run time). A part of the random test generator consists of a computational module, which determines the speedup for the analyzed K-loop using the techniques that are described in this thesis: loop unrolling (Chapter 4), loop shifting (Chapter 5), K-pipelining (Chapter 6), and loop distribution (Chapter 7).

A test case is determined by the following parameters:

- N — the number of iterations;
- $T_{\text{sw}}[j]$ — the number of cycles for each software function SW_j;
- MAX(sw) — the maximum of the number of cycles of the software functions:
$$\text{MAX(sw)} = \max_j(T_{\text{sw}}[j]);$$
- $T_{K(\text{sw})}[i]$ — the number of cycles for kernel K_i running in software;
- $S[i]$ — the speedup for kernel K_i ($S[i] \in \mathbb{R}$);
- $A[i]$ — the area occupied by kernel K_i in hardware, in percent ($A[i] \in \mathbb{R}$). Additionally,
$$\sum(A[i]) \leq 90\%;$$
- $T_{K(\text{hw})}[i]$ — the number of cycles for kernel K_i running in hardware:
$$T_{K(\text{hw})}[i] = (\text{int})\ \frac{T_{K(\text{sw})}[i]}{S[i]};$$
- $T_{\text{r}}[i]$ — the time for memory read for kernel K_i running in hardware (cycles);
- $T_{\text{w}}[i]$ — the time for memory write for kernel K_i running in hardware (cycles);
- $T_{\text{c}}[i]$ — the time for computation for kernel K_i running in hardware (cycles):
$$T_{\text{c}}[i] = T_{K(\text{hw})}[i] - T_{\text{r}}[i] - T_{\text{w}}[i].$$

The computational module can be used to determine the speedup of any given K-loop whose parameters are known.

For the randomly generated tests, the values for the presented parameters have been generated according to the Table 3.2. Note that in some cases, the maximum value of a parameter depends on the generated value of another parameter (for instance, the number of cycles needed by a kernel for reading the data from memory is at most one third of the total number of cycles for the kernel running in hardware).

To study the impact that the area requirements have on performance, the maximum value for the area for a kernel is set first to 12%, and later on to 60%.

Table 3.2: Parameter values for randomly generated tests.

Parameter	Min. value	Max. value
N	64	256*256
$T_{\text{sw}}[j]$	0	800
$T_{K(\text{sw})}[i]$	1.5 * MAX(sw)	41.5 * MAX(sw)
$S[i]$	2.0	10.0
$A[i]$	1.2%	12.0%/60%
$T_{\text{r}}[i]$	1	$T_{K(\text{hw})}[i]/3$
$T_{\text{w}}[i]$	1	$T_{K(\text{hw})}[i]/6$

3.5 Overview of the Proposed Loop Transformations

In this section, we present an algorithm used to decide which loop transformations to apply to a given K-loop. Table 3.3 shows an overview of the different loop transformations analyzed in this thesis. For each transformation, the following information is shown:

- the K-loop size that it is suitable for;
- the impact it has on the code (such as enabling parallelism or eliminating dependencies);
- the transformations that can be used in conjunction with it;
- the transformations that are exclusive with it;
- whether the transformation can schedule kernels to run in software or not.

Table 3.3: Summary of loop transformations for K-loops.

Transformation	K-loop size	impact	reconfig	can be used with	exclusive with	can schedule kernels in sw
(1) unrolling	any	enable hw parallelism	no	(2)–(5)	–	no
(2) shifting	1	enable hw-sw parallelism	no	(1), (4), (5)	(3)	no
(3) K-pipelining	≥ 2	enable hw-sw parallelism	no	(1), (4)	(2)	no
(4) distribution	≥ 2	split the K-loop into K-sub-loops that are individually optimized	yes/no	(1)–(3), (5)	–	no
(5) skewing	1	eliminate dependencies, enable unrolling	no	(1), (2)	–	yes

The 'Loop_Trans' algorithm in Figure 3.6 can be used to decide which loop transformations to use on a given K-loop (*KL*).

If the K-loop size is 1, depending on the existing loop dependencies, the K-loop is transformed with either loop skewing and/or loop shifting. Loop shifting is used to enable the software-hardware parallelism. If loop shifting is performed, it is important to look at the ratio between the execution times of the software part and of the hardware part of the K-loop, in order to understand if loop unrolling, which exposes the hardware parallelism, should be performed. If the software execution time is twice (or more) more than the hardware execution time, unrolling will bring no benefit because the hardware execution would be completely hidden by the software execution. If the ratio between the software and the hardware execution times is less than the threshold value ($Eps = 2$), unrolling will be performed.

If the K-loop size is larger than 1, the loop distribution algorithm will determine the best partitioning of the K-loop. The algorithm works in a Greedy manner, selecting the most promising kernel in terms of speedup and adding functions around it to create a K-sub-loop, while constantly checking the performance of the K-sub-loop against that of the original loop. It is possible that the partitioning that gives the best performance is the original K-loop. If the decision is to split the K-loop into smaller K-sub-loops, each of the K-sub-loops will be transformed according to the same 'Loop_Trans algorithm. If the loop distribution algorithm has decided that the original K-loop would perform best, then it will be transformed with K-pipelining and unrolling.

Table 3.4 summarizes the formulas for computing the K-loop execution time (in cycles) for each of the proposed loop transformations. The formulas presented for the loop unrolling, loop shifting, K-pipelining and loop skewing are dependent on the unroll factor, assuming that we are always interested in exploiting the hardware parallelism. In the case of loop distribution, the total execution time is the sum of the execution times of the K-sub-loops and their reconfiguration times, and each of the K-sub-loops may have a different degree of parallelism and different formula to compute its execution time. For this reason, we cannot explicitly write the formula in the case of loop distribution.

```
Eps = 2;
shift = 0;
Loop_Trans KL:
    if (KL.Size == 1) then
        if (KL.Dependencies & WAVEFRONT) then
            Skew(KL);
        end
        if (KL.Dependencies & TASKCHAIN) then
            Shift(KL);
            shift = 1;
        end
        if shift && (KL.SW_ExecTime() / KL.HW_ExecTime() <
        Eps) then
            Unroll(KL);
        end
    else
        go = KL.Distribute();
        if (go) then
            while (KL_i[i] = KL.GetNextSubLoop()) do
                Loop_Trans (KL_i[i]);
                i++;
            end
        else
            KPipeline(KL);
            Unroll(KL);
        end
    end
end
```

Figure 3.6: The loop transformations algorithm.

Table 3.4: Summary of the execution times (in cycles) per loop transformation.

Transformation	Execution time
(1) unrolling	$T_{\text{unroll}}(u) = N \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R)$
(2) shifting	$T_{\text{shift}}(u) = \begin{cases} u \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R), & u < U_1 \\ \lfloor N/u \rfloor \cdot u \cdot T_{\text{sw}} + \max\left(R \cdot T_{\text{sw}}, T_{K(\text{hw})}(u)\right) + T_{K(\text{hw})}(R), & u \geq U_1 \end{cases}$
(3) K-pipelining	$T_{pipe}(u) = (N/u - 1) \cdot (\sum_{i=1}^{N_k} \max(T_{K_i}(u), u \cdot T_{SW_i}) + u \cdot T_{SW_{N_k+1}}) + \sum_{i=1}^{N_k} T_{K_i}(R) + + \sum_{i=1}^{\lceil N_f/2 \rceil} u \cdot T_{SW_i} + \sum_{i=1}^{N_k} \max(T_{K_i}(u), R \cdot T_{SW_i}) + (R+u) \cdot T_{SW_{N_k+1}}$
(4) distribution	T_{distr} ‘ $= T_{KL_0} + \sum_{i=1}^{L-1}(T_{reconfig_i} + T_{KL_i})$, when the K-loop is split into L K-sub-loops ($KL_{0..L-1}$) (T_{KL_i} is computed like in [1], [2], [3] or [5], depending on the case; $T_{reconfig_i}$ is the reconfiguration time)
(5) skewing	$T_{\text{h/h}}(u) = 2 \cdot q \cdot \sum_{i=1}^{u-1} T_{K(\text{hw})}(i) + 2 \cdot \sum_{i=1}^{r-1} T_{K(\text{hw})}(i) + (N - M + 1) \cdot T_{K(\text{hw})}(r) + + q \cdot (N - u + 1 + r) \cdot T_{K(\text{hw})}(u)$, with $M = q \cdot u + r,\ r < u$
	$T_{\text{h/s}}(u) = 2 \cdot \sum_{i=1}^{u-1} T_{K(\text{hw})}(i) + \left(2 + 2 \cdot \sum_{i=u+1}^{M-1} q_v(i) + (N - M + 1) \cdot q_v(M)\right) \cdot T_{K(\text{hw})}(u) + + \left(2 \cdot \sum_{i=u+1}^{M-1} T_{K(\text{hw})}(r_v(i))\right) + (N - M + 1) \cdot T_{K(\text{hw})}(r_v(M))$

3.6 Conclusions

In this chapter, we laid the foundation for the methods presented in the following chapters, suitable for optimizing applications for heterogeneous platform comprising a reconfigurable processor.

In Section 3.1, we explained the notion of K-loops and the assumptions associated with the K-loops framework and with the targeted applications. The platform used for our experiments is presented in Section 3.2. The main notations used throughout this thesis are an important part of the methodology and are presented in Section 3.3. The area, memory and speedup considerations, which are common to the algorithms illustrated in this thesis, are also presented in Section 3.3.

A random test generator used to study the impact of various loop transformations on K-loops has been developed and it is described in Section 3.4. The experimental results presented in Chapter 6 and Chapter 7 make use of the test cases created with this random test generator.

In Section 3.5, we have presented an overview of the loop transformations proposed in this thesis, and an algorithm that decides which of these transformations to use for a given K-loop.

The next chapter of this dissertation is dedicated to loop unrolling. The mathematical model of using loop unrolling in the context of K-loops will be presented, followed by experimental results.

4

Loop Unrolling for K-loops

IN THIS CHAPTER, we discuss how loop unrolling can be used in the context of K-loops. In Section 4.1.1 we show how a simple K-loop that contains only one kernel can be parallelized via loop unrolling. In Section 4.2, we present the mathematical analysis that allows to determine the K-loop speedup that can be achieved after parallelizing with loop unrolling. Experimental results for several kernels extracted from well-known applications are presented in Section 4.3. Finally, conclusions are presented in Section 4.4.

4.1 Introduction

Loop unrolling is a transformation that replicates the loop body and reduces the iteration number. In traditional computing, it is used to eliminate the loop overhead, thus improving the cache hit rate and reducing branching. In reconfigurable computing, loop unrolling is used to expose parallelism. In this thesis, we use loop unrolling to expose the loop-level parallelism, which allows us to concurrently execute multiple kernels on the reconfigurable hardware. A generic representation of the loop unrolling transformation is presented in Figure 4.1. In this figure, each task inside the initial loop is replicated for the same number of times inside the unrolled loop.

4.1.1 Example

We illustrate the loop unrolling technique on a motivational example illustrated in Figure 4.2. In this example, data dependencies between `SW` and K may exist in each iteration. In order to be able to apply loop unrolling and run in parallel multiple instances of the kernel, data dependencies between K(i) and K(j), for any iterations i and j, with $i \neq j$, may not exist. For instance,

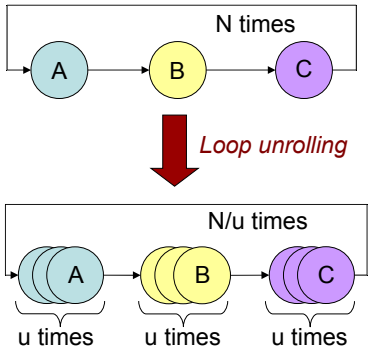


Figure 4.1: Loop unrolling.

`SW` can be the code that computes the parameters for the kernel instance to be executed in the same iteration. The unrolled and parallelized loop is illustrated in Figure 4.3. This figure shows the case $N \bmod u = 0$, when the loop unrolling transformation does not create an epilogue.

```
for (i = 0; i < N; i++) do
    /* a pure software function                    */
    SW(blocks[i]);
    /* a hardware mapped kernel                    */
    K(blocks[i]);
end
```

Figure 4.2: Original K-loop with one kernel.

4.2 Mathematical Model

The speedup at loop nest level for the K-loop transformed with loop unrolling is defined as the ratio between the execution time for the pure software execution and the execution time for the loop with the kernel running in hardware:

$$S_{\text{unroll}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{unroll}}(u)}. \tag{4.1}$$

In the example in Figure 4.3, the total execution time for the K-loop with the kernel running in hardware is:

$$T_{\text{unroll}}(u) = (T_{\text{sw}} \cdot u + T_{K(\text{hw})}(u)) \cdot (N/u). \tag{4.2}$$

```
for (i = 0; i < N; i += u) do
    /* u instances of SW(), sequentially       */
    SW(blocks[i + 0]);
    ...
    SW(blocks[i + u − 1]);
    /* u instances of K in parallel            */
    #pragma parallel
        K(blocks[i + 0]);
        ...
        K(blocks[i + u − 1]);
    #end
end
```

Figure 4.3: Parallelized K-loop after unrolling with factor u.

For the general case where u is not a divisor of N, the remainder instances of the software function and hardware kernel will be executed in the loop epilogue. We denote by R the remainder of the division of N by u:

$$R = N - u * \lfloor N/u \rfloor \,, 0 \leq R < u \,. \tag{4.3}$$

We define $T_{K(\text{hw})}(R)$ as:

$$T_{K(\text{hw})}(R) = \begin{cases} 0, & R = 0 \\ T_c + T_{\text{min(r,w)}} + R \cdot T_{\text{max(r,w)}}, & R > 0. \end{cases} \tag{4.4}$$

Then, $T_{\text{unroll}}(u)$ is:

$$\begin{aligned} T_{\text{unroll}}(u) &= \lfloor N/u \rfloor \cdot (T_{\text{sw}} \cdot u + T_{K(\text{hw})}(u)) + (R \cdot T_{\text{sw}} + T_{K(\text{hw})}(R)) \\ &= N \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R) \end{aligned} \tag{4.5}$$

which, by expanding $T_{K(\text{hw})}(u)$ from (3.8) with $\text{u} \leq \text{u}_\text{m}$, is equivalent to:

$$T_{\text{unroll}}(u) = (T_{\text{sw}} + T_{\text{max(r,w)}}) \cdot N + (T_{\text{c}} + T_{\text{min(r,w)}}) \cdot \lceil N/u \rceil \tag{4.6}$$

Since $T_{\text{loop(sw)}}(u)$ is constant and $T_{\text{unroll}}(u)$ is a monotonic decreasing function, then $S_{\text{loop}}(u)$ is a monotonic increasing function for $u < \text{u}_\text{m}$.

This means that performance increases with the unroll factor. However, our experimental results show that the performance increase after a certain unroll factor tends to be smaller and smaller and can be considered at some point

negligible. The application designer will then decide which is the desirable area-performance trade-off. The optimum unroll factor is determined by applying the general methodology described in Chapter 3 regarding area, memory and performance.

4.3 Experimental Results

The purpose of this section is to illustrate loop unrolling for K-loops, by applying it to several kernels from well known applications. The relative speedup obtained by running multiple instances of a kernel in parallel compared to running a single instance is computed taking into account the profiling information. The optimal unroll factor is then automatically determined, based on the area and memory constraints and the desired trade-off. The achieved performance depends on the kernel implementation, but is also subject to Amdahl's Law.

4.3.1 Selected Kernels

Four different kernels have been selected to illustrate the parallelization with the loop unrolling method. For some kernels, multiple implementations have been tested. The dependencies between different iterations of the loops containing the kernels have been eliminated, if there were any. The VHDL code for the analyzed kernels has been automatically generated with the DWARV tool [149] and synthesized with Xilinx XST tool of ISE 8.1. The kernels are:

1. **DCT (Discrete Cosine Transformation)** - a 2-D integer implementation, extracted from the MPEG2 encoder multimedia benchmark. The DCT operates on 8×8 memory blocks and there are 64 memory reads and 64 memory writes performed by each kernel instance. The `SW` function consists in a pre-processing of the blocks by adjusting the luminance/chrominance.

2. **Convolution** - extracted from the Sobel algorithm. We have chosen not to implement in hardware the whole Sobel algorithm as most of its execution time is spent inside a loop which uses the convolution. By accelerating the convolution, a large speedup for the whole Sobel algorithm can be obtained, but with a much smaller area consumption. As the kernel consists only of the convolution, the part of the algorithm that adjusts the values of the pixels to the interval [0, 255] is contained in the `SW` function.

Table 4.1: Profiling information for the DCT, Convolution, SAD, and Quantizer kernels.

Kernel	Area [%]	T_r/T_w (cycles)	$T_{K(sw)}$ (cycles)	$T_{K(hw)}$ (cycles)	N iter.	$T_{loop(sw)}$ (cycles)	T_{sw} (cycles)
DCT	12.39	192/64	106626	37278	6*16	10751868	5292
Convolution	3.70	24/1	2094	204	126*126	35963184	168
SAD-area	6.81	330/1	4013	2908	11*13	619392	84
SAD-time	13.17	330/1	4013	1305	11*13	619392	84
Quantizer-1	2.98	192/64	12510	2970	64*16	20925786	8112
Quantizer-2	4.35	192/64	12510	1644	64*16	20925786	8112
Quantizer-4	7.08	192/64	12510	1068	64*16	20925786	8112
Quantizer-8	12.13	192/64	12510	708	64*16	20925786	8112

3. **SAD (Sum of Absolute Differences)** - extracted also from the MPEG2 encoder. For this kernel, we have chosen two VHDL implementations: an implementation that is faster but occupies more area, and another one that is slower but takes less area. We will call them SAD-time and SAD-area, respectively. Note that the VHDL code was automatically generated and the performance does not compare to a hand-written implementation. The loop nest containing the SAD has been transformed into a perfect nest, and the execution times for one instance of SAD on GPP/FPGA used in the algorithm are taken as the weighted average for the execution times of all instances within the loop nest. The length of the execution of a SAD function is determined by the values of some parameters, which are updated after each execution. As a result, `SW` is the post-processing code which updates the values of these parameters.

4. **Quantizer** - an integer implementation, extracted from the JPEG application. For this kernel, we have four implementations, denoted by Q-1, Q-2, Q-4 and Q-8. The one with the smallest area consumption is the slowest (Q-1), and the fastest one also requires the most area (Q-8). The `SW` is a post-processing function, and it performs a Zig-Zag transformation of the matrix.

Table 4.1 summarizes the profiling information for the kernels considered in these experiments. It includes the area occupied by one instance of the kernel, memory transfer times, various execution times, and the number of

iterations in the loop nest. The execution times have been measured using the PowerPC timer registers. For the kernels running on FPGA, the times include the parameter transfer using exchange registers. The measured execution times are the following:

1. the execution time for a single instance of each kernel running in software / hardware — T_K(sw) / T_K(hw);
2. the execution time for the whole loop — $T_{\text{loop(sw)}}$;
3. the execution time for the software part (in one iteration) — T_{sw}.

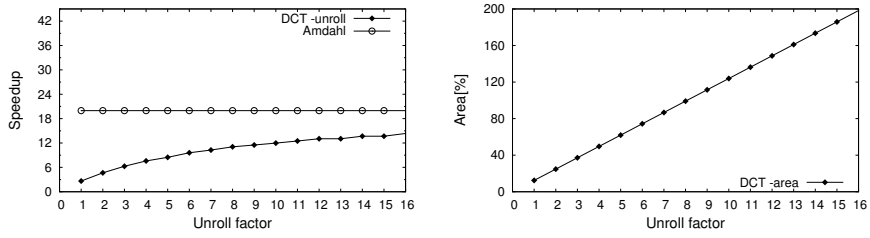
The times for the memory transfers (T_{r} / T_{w}) are computed considering 3 cycles per memory read (in the automatically generated VHDL, the memory transfers are not pipelined) and 1 cycle per memory write.

The experiments have been performed with one instance of the kernel running on the FPGA. The results for the execution time of the loop for higher unroll factors are computed using (4.6), while the speedup is computed using (4.1).

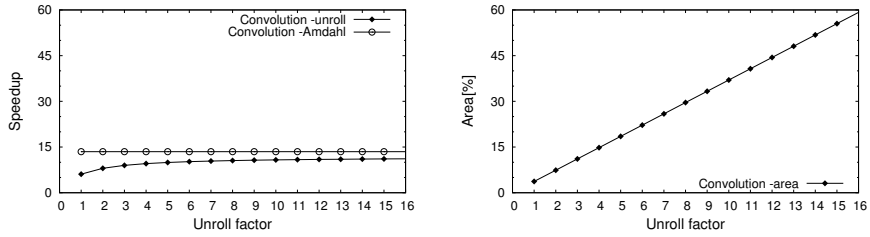
4.3.2 Analysis of the Results

The results for the kernels considered in our experiments are illustrated in Figure 4.4. The left-side figures illustrate, for each kernel, the speedups for different unroll factors and the speedup by Amdahl's Law. The area consumption for each kernel for various unroll factors is shown in the right-side figures. Taking into account also area requirements, the optimum unroll factor depends on the desired trade-off between area consumption and performance. If, for instance, the metric chosen for the optimum unroll factor is to have a speedup increase with a factor of 0.5 compared to the previous unroll factor, then the optimum unroll factors are: 8 for DCT, 4 for Convolution, 6 for SAD-area and 5 for SAD-speed, 1 for Q-1–Q-8 (no unroll). Since 8 instances of DCT would occupy 99% of the VirtexII-Pro area and create routing problems, the unroll factor 7 is selected for this platform. Table 4.2 summarizes these results, showing for each kernel the optimal unroll factor, the area requirements, the achieved speedup, the maximum obtainable speedup by Amdahl's Law and, finally, how close we are to that maximum.

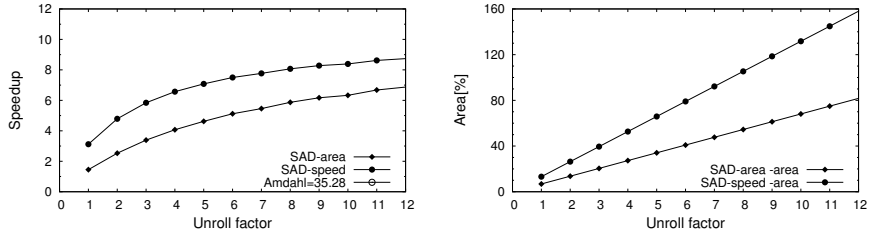
1. **DCT (MPEG2)**. As it can be seen in Figure 4.4(a), $S_{\text{loop}}(u)$ is monotonously increasing, with the maximum value 19.07 for $u = N$. For comparison, the speedup at kernel level – which would be achieved



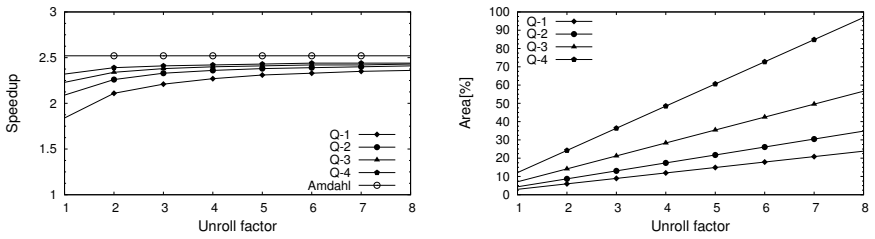
(a) DCT



(b) Convolution



(c) SAD



(d) Quantizer

Figure 4.4: Loop unrolling: speedup and area consumption for: a) DCT; b) Convolution; c) SAD; d) Quantizer.

Table 4.2: Loop unrolling: results summary for the analyzed kernels.

Kernel	U opt.	Area usage [%]	Speedup	Amdahl	Percent
DCT	7	87.0	10.27	19.97	51.4 %
Convolution	4	14.8	9.56	13.48	70.9 %
SAD-area	6	40.9	5.12	35.28	14.51 %
SAD-time	5	65.9	7.08	35.28	20.06 %
Q-8	1	12.1	2.32	2.52	92.06 %

if the loop would not have a software part – is drawn with a dashed line on the same figure, while the speedup by Amdahl's Law is represented as a horizontal continuous line with the value 19.97.

If the area is not a constraint, the optimum unroll factor can be decided considering the desired speedup increase between two consecutive unroll factors. For a speedup increase of minimum 0.4, the unroll factor is 12, with the speedup 13.05. For a speedup increase of 0.5, the unroll factor is 8, with the speedup 11.06.

For Virtex II-Pro, considering the DCT and Molen area requirements, the optimal unroll factor is 7, with an area consumption 87% and speedup of 10.27. This represents 51.4% of the speedup by Amdahl's Law.

2. **Convolution (Sobel)**. The speedups for Sobel are illustrated in Figure 4.4(b). $S_{\text{loop}}(u)$ is an increasing function, reaching its maximum value of 11.8 for $u = N$. The shape of the function is due to the fact that the execution time for the software part of the loop is 82% of the kernel's execution time in hardware.

   The speedup at kernel level is drawn with a dashed line on the same figure. The shape is due to the fact that the memory transfers represent a significant amount of the kernel's hardware execution time, almost 12%.

   If a speedup increase of 0.5 between consecutive unroll factors is desired, then the optimum unroll factor is 4. This leads to a speedup of 9.56 for the whole loop and an area consumption of 14.8% of the Virtex II-Pro area.

3. **SAD (MPEG2)**. For SAD we have two different implementations: the first one is time-optimized, while being also area-consuming (SAD-time),

and the second one is area-optimized, while being more time-consuming (SAD-area). The results for both implementations are illustrated in Figure 4.4(c). If a speedup increase of 0.5 between consecutive unroll factors is desired, then the optimal unroll factor for SAD-time is 5, with a speedup of 7.08 and with area consumption of 65.9%. For SAD-area, the optimal unroll factor is 6, with an area consumption of 40.9% and speedup of 5.12.

By looking at the experimental results, it is a wise decision to choose the SAD-time implementation, which gives better performance and requires a reasonable amount of area. Nevertheless, we must mention that the decision on which implementation to use differs from case to case, as it cannot be taken without comparing the best solutions for the different implementations.

The maximum Amdahl based speedup is 35.28, and for the sake of clarity it is not illustrated in the figure, and the maximum value of the Y axis is 12. For SAD, the performance achieved with our method is only 20% of the theoretical maximum. Part of the explanation is that Amdahl's Law does not take into account memory bottlenecks. Memory accesses represent an important part of the total execution time of the SAD kernel (11.34% for SAD-area and 25.28% for SAD-time), as they are not optimized at all in the hardware implementations generated by the DWARV tool. For this reason, the speedup increases slowly when using the loop unrolling.

In conclusion, this is an example of K-loop that would benefit more from an optimized hardware implementation of the kernel able to reduce the memory access time.

4. **Quantizer (JPEG)**. Out of the four Quantizer implementations we tested (Q-1, Q-2, Q-4 and Q-8), Q-1 is the slowest but requires the least area, and Q-8 is the fastest and requires the most area. However, T_{sw} (the execution time for the software function) is significantly larger than $T_{K(\text{hw})}$ (the execution time for the kernel running in hardware). This leads to a very small performance increase, as can be seen in Figure 4.4(d).

   Kernel speedup and area consumption are represented separately. We notice that a kernel speedup of 40 is possible, with area consumption of 50% (Q-4 for $u = 7$), but the loop speedup for the same unroll factor is only 2.4.

   As a conclusion, this loop example shows that if the software part of the loop needs more time to execute than the kernel's time in hardware,

there will be very little or no gain in unrolling, no matter how optimized the kernel implementation is. This is nothing but a practical proof for Amdahl's Law. The speedup achieved with Q-8 without unrolling is 2.32, that is very close to the theoretical maximum given by Amdahl's Law, $\approx$ 2.53. The area consumption is 12.13% of the Virtex II-Pro area.

4.4 Conclusions

Loop unrolling is a very useful and common transformation for exposing parallelism. In the context of K-loops, loop unrolling is used for exposing hardware parallelism, and enabling parallel execution of identical kernel instances on the reconfigurable hardware.

We have analyzed four well-known kernels, and compared the achieved results with the theoretical maximum speedup computed with Amdahl's Law, by assuming maximum parallelism (full unroll) for the hardware.

Depending on the ratio between the execution time of the software part of the K-loop and that of the kernel executing in hardware, we see different aspects of Amdahl's Law. If the software is faster, as in the case of DCT and SAD, different results will be obtained for different kernel implementations, depending on how much optimized they are. However, the speedup cannot be larger than the theoretical maximum given by Amdahl's Law. If the execution time of the software part is much larger than the kernel's execution time in hardware, then unrolling does not bring any benefit, as in the case of the Quantizer kernel.

For I/O intensive kernels, the performance that can be achieved by applying our methods with automatically generated VHDL is quite far from the theoretical maximum. The main reasons for this are that Amdahl's Law does not consider memory bottlenecks, and that in the current stage, the DWARV tool does not optimize the memory accesses.

The next chapter of this dissertation is dedicated to loop shifting. The mathematical model of using loop shifting in the context of K-loops will be presented, followed by experimental results.

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

O.S. Dragomir, E. Moscu Panainte, K. Bertels, S. Wong, **Optimal Unroll Factor for Reconfigurable Architectures**, ARC 2008
O.S. Dragomir, T. Stefanov, K. Bertels, **Optimal Loop Unrolling and Shifting for Reconfigurable Architectures**, TRETS 2009

5

Loop Shifting for K-loops

IN THIS CHAPTER, we discuss how loop shifting can be used in the context of K-loops. In Section 5.1.1, we show how a simple K-loop with one hardware kernel and intra-iteration dependencies is parallelized via loop unrolling plus shifting. In Section 5.2, we present the mathematical part of determining the K-loop speedup after parallelizing with loop unrolling and shifting. Experimental results for several kernels extracted from real-life applications are presented in Section 5.3, while final conclusions are presented in Section 5.4.

5.1 Introduction

Loop shifting is a transformation that moves operations from one iteration of the loop body to the previous iteration, as shown in Figure 5.1. Loop shifting is a particular case of software pipelining, The operations are shifted from the beginning of the loop body to the end of the loop body and a copy of these operations is also placed in the loop prologue. In our research, loop shifting means moving a function from the beginning of the K- loop body to its end, while preserving the correctness of the program. While loop unrolling is used to expose the parallelism at hardware level (e.g., by running multiple kernels in parallel), loop shifting is used to eliminate the data dependencies between software and hardware functions, which, as a result, allows to concurrently execute on the GPP and FPGA.

Loop shifting can be used together with loop unrolling for enhanced performance. Anyhow, cases exist when loop unrolling does not bring any benefit after performing loop shifting.

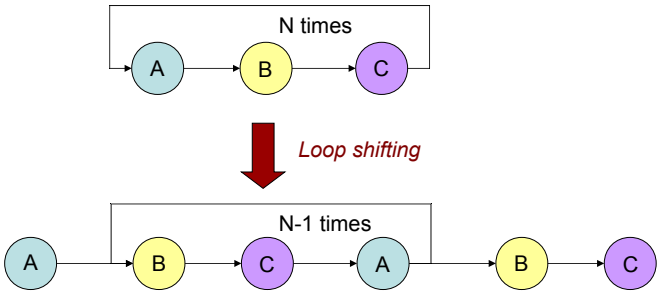


Figure 5.1: Loop shifting.

5.1.1 Example

We illustrate the loop shifting technique on a motivational example illustrated in Figure 5.2. Assuming that there are data dependencies between the software function `SW` and the hardware kernel K within the same iteration but not inter-iterations, the objective is to break the dependencies and enable concurrent execution of the software function and the hardware kernel. The proposed loop shifting moves the software part of the K-loop to the end of the loop body and each iteration i of the shifted K-loop contains the hardware kernel K_i and the software function SW_{i+1}. The parallelized shifted K-loop is shown in Figure 5.3.

```
for (i = 0; i < N; i++) do
    /* a pure software function                    */
    SW(blocks[i]);
    /* a hardware mapped kernel                    */
    K(blocks[i]);
end
```

Figure 5.2: Original K-loop with one kernel.

The loop unrolling and loop shifting can be combined for better results. In order to be able to apply loop unrolling, inter-iteration data dependencies between K(i) and K(j) ($i \neq j$) are not allowed. Figure 5.4 illustrates the parallelized K-loop resulted by applying the loop unrolling and the loop shifting techniques. The K-loop is unrolled with a factor u and then the software part of the K-loop is shifted to the end of the K-loop body. In each new iteration, u sequential executions of the function `SW` are executed in parallel with u identical kernel instances. The first u calls of `SW` are executed before the K-loop body and they determine the K-loop prologue. The last u kernel instances are executed after

```
SW(blocks[0]);
for (i = 1; i < N; i++) do
    #pragma parallel
        /* the hardware mapped kernel from the
           previous iteration                    */
        K(blocks[i − 1]);
        /* the software function                 */
        SW(blocks[i]);
    #end
end
K(blocks[N − 1]);
```

Figure 5.3: The parallelized shifted K-loop.

the K-loop body (the K-loop epilogue). Note that if the unroll factor u is not a divisor of the total number of iterations N, the reminder of functions that do not make a full iteration will be found in the K-loop epilogue. For clarity, the code Figure 5.4 shows the case when $N \bmod u = 0$.

5.2 Mathematical Model

In this section, we show how the speedup of the parallelized K-loop in Figure 5.4 is computed. We assume that the K-loop is transformed via loop shifting and loop unrolling with factor u. If only loop shifting is desired, the value of u should be 1. The speedup at loop nest level when unrolling and shifting are used is defined as the ratio between the execution time for the pure software execution and execution time for the K-loop with the kernel running in hardware:

$$S_{\text{shift}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{shift}}(u)}. \tag{5.1}$$

The total execution time ($T_{\text{shift}}(u)$) for the K-loop transformed with unrolling and shifting with the kernel running in hardware is the sum of the execution times of the prologue, transformed loop body, and epilogue:

$$T_{\text{shift}}(u) = T_{\text{prolog}}(u) + T_{\text{body}}(u) + T_{\text{epilog}}(u). \tag{5.2}$$

By looking at Figure 5.4, the three components of the total execution time are computed as follows.

```
SW(blocks[0]);
...
SW(blocks[u − 1]);
for (i = u; i < N; i += u) do
    /* u instances of K in parallel with the
       software                                    */
    #pragma parallel
        K(blocks[i − u]);
        ...
        K(blocks[i − 1]);
        /* sequential execution in software      */
        {
            SW(blocks[i + 0]);
            ...
            SW(blocks[i + u − 1]);
        }
    #end
end
#pragma parallel
    K(blocks[N − u]);
    ...
    K(blocks[N − 1]);
#end
```

Figure 5.4: Parallelized K-loop after shifting and unrolling with factor u.

(i) $T_{\text{prolog}}(u)$ is the time for the K-loop prologue:

$$T_{\text{prolog}}(u) = u \cdot T_{\text{sw}}; \tag{5.3}$$

(ii) $T_{\text{body}}(u)$ is the time for the transformed K-loop body, consisting of parallel hardware and software execution:

$$T_{\text{body}}(u) = \left(\lfloor N/u \rfloor - 1\right) \cdot \max\left(u \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)\right); \tag{5.4}$$

(iii) $T_{\text{epilog}}(u)$ is the time for the K-loop epilogue.

For the simplified case in Figure 5.4, the epilogue consists of the hardware parallel execution of u kernel instances: $T_{\text{epilog}}(u) = T_{K(\text{hw})}(u)$.

For the general case ($N \mod u \neq 0$), the time for the epilogue is:

$$T_{\text{epilog}}(u) = \max\left(R \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)\right) + T_{K(\text{hw})}(R), \tag{5.5}$$

where $T_{K(\text{hw})}(R)$ is defined in (4.4).

In order to compute $T_{\text{body}}(u)$ from (5.4) and $T_{\text{epilog}}(u)$ from (5.5) – where the max function is used –, there are different cases depending on the relations between T_{sw}, T_{c}, $T_{\text{min(r,w)}}$ and $T_{\text{max(r,w)}}$. For values of u greater than a threshold value U_1, the execution on the reconfigurable hardware in one iteration will take less time than the execution on GPP. If $T_{\text{sw}} \leq T_{\text{max(r,w)}}$, then the execution time for the software part inside a loop iteration increases slower than the hardware part and the threshold value is $U_1 = \infty$. Otherwise, the execution time of the software part increases faster than the hardware part and we have to compute the threshold value. The execution time for one iteration is:

$$\max\left(u \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)\right) = \begin{cases} T_{K(\text{hw})}(u), & u < U_1 \\ u \cdot T_{\text{sw}}, & u \geq U_1. \end{cases} \tag{5.6}$$

If $u \leq U_1$ then:

$$u \cdot T_{\text{sw}} \leq T_{\text{c}} + T_{\text{min(r,w)}} + u \cdot T_{\text{max(r,w)}}. \tag{5.7}$$

This determines the threshold value U_1 for the case $T_{\text{sw}} > T_{\text{max(r,w)}}$ as:

$$U_1 = \left\lceil \frac{T_{\text{c}} + T_{\text{min(r,w)}}}{T_{\text{sw}} - T_{\text{max(r,w)}}} \right\rceil. \tag{5.8}$$

The execution time for the K-loop transformed with unrolling and shifting is:

$$T_{\text{shift}}(u) = \begin{cases} u \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R), & (u < U_1) \\ \lfloor N/u \rfloor \cdot u \cdot T_{\text{sw}} + \max\left(R \cdot T_{\text{sw}}, T_{K(\text{hw})}(u)\right) + T_{K(\text{hw})}(R). & (u \geq U_1) \end{cases} \quad (5.9)$$

Intuitively, we expect that the unroll factor that gives the smallest execution time and, thus, the largest speedup is the one where the software and hardware execute concurrently in approximatively the same amount of time. This happens in the close vicinity of U_1 (we define the close vicinity as the set $\{U_1 - 1, U_1, U_1 + 1\}$), depending if any of these values is a divisor of N or not. More specifically, if U_1 is a divisor of N, then it is the value that maximizes the speedup function. Otherwise, this value might be found for $u > U_1$ when u is a divisor of N, but then $T_{\text{shift}}(u)$ is significantly smaller (more than 10%) then $T_{\text{shift}}(U_1)$ only if $\lfloor N/u \rfloor \geq 10$.

Unrolling and shifting vs. unrolling We compare $T_{\text{shift}}(u)$ from (5.9) with $T_{\text{unroll(hw)}}(u)$ from (4.5).

$$T_{\text{unroll(hw)}}(u) - T_{\text{shift}}(u) = \begin{cases} (N - u) \cdot T_{\text{sw}}, & \text{if } u < U_1 \\ R \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) - & \\ \quad \max(R \cdot T_{\text{sw}}, T_{K(\text{hw})}(u)), & \text{if } u \geq U_1. \end{cases} \quad (5.10)$$

This means that the execution time for the two methods is the same when using the maximum unroll factor ($u = N$) or when the K-loop has no software part. Otherwise, the execution time for the K-loop transformed with unrolling and shifting is smaller than the one for unrolling only, and, as a result, the performance is improved.

5.3 Experimental Results

The analyzed kernels are DCT, Convolution, SAD and Quantizer. Details about their implementation and profiling information are provided in Section 4.3. The results for parallelizing the selected kernels with loop unrolling and loop shifting are illustrated in Figure 5.5. For each kernel, we show the possible speedups for different unroll factors, and the speedup by Amdahl's Law. Table 5.1 summarizes these results, showing the optimal unroll factor, the area

Table 5.1: Loop shifting: results summary for the analyzed kernels.

Kernel	U opt.	Area usage [%]	Speedup	Amdahl	Percent
DCT	7	87.0	18.7	19.97	93.6 %
Convolution	2	7.4	13.48	13.48	99.9 %
SAD-time	6	79.0	8.71	35.28	24.7 %
Quantizer	1	12.1	2.52	2.52	99.9 %

requirements, the achieved speedup, and the speedup by Amdahl's Law for each kernel.

1. **DCT (MPEG2)**. In Figure 5.5(a) it can be seen that $S_{\text{shift}}(u)$ is growing significantly faster than $S_{\text{loop}}(u)$ until unroll factor $u = U_1 = 8$. At this point, $S_{\text{loop}}(u) = 11.06$ and $S_{\text{shift}}(u) = 19.65$, which is the maximum value. For $u > 8$, $S_{\text{shift}}(u)$ has values between 18.5 and 19.65, while $S_{\text{loop}}(u)$ is monotonously increasing, with the maximum value 19.07 for $u = N$.

   For comparison purposes, the speedup at kernel level – which would be achieved if the loop would not have a software part – is drawn with a dashed line on the same figure, while the speedup by Amdahl's Law is represented as a horizontal continuous line with the value 19.97.

   In conclusion, if the area is not constraint, the best solution is unrolling and shifting with unroll factor 8. For Virtex II-Pro, considering only the area requirements for DCT and for Molen, the optimal unroll factor is 7, with an area consumption of 87% and speedup of 18.7.

2. **Convolution (Sobel)**. The execution time for the software part of the loop is 82% of the kernel's execution time in hardware, which leads to $U_1 = 2$. Indeed, as suggested by Figure 5.5(b), the maximum for $S_{\text{shift}}(u)$ is achieved for $u = 2$. $S_{\text{loop}}(u)$ is an increasing function, reaching its maximum value of 11.8 for $u = N$. Note that the speedup obtained by combining unrolling with shifting is approximatively equal to the speedup by Amdahl's Law, 13.48.

   The speedup at kernel level is drawn with a dashed line on the same figure. The shape is due to the fact that the memory transfers represent a significant part of the kernel's execution time in hardware, almost 12%.

In conclusion, unrolling combined with shifting gives an optimal unroll factor of 2, which leads to a speedup of 13.48 for the whole loop and an area consumption of 7.4% of the Virtex II-Pro area.

3. **SAD (MPEG2)**. The software part of the loop executes faster than the memory transfers performed by the hardware kernel. As a result, U_1 is set to ∞. As a consequence, $S_{\text{shift}}(u)$ looks like an increasing function in the interval $[0, N]$, with a maximum at $u = N$. This can be seen in Figure 5.5(c).

   As seen in the previous chapter, two different implementations of SAD are considered. The first one is time-optimized and area-consuming (SAD-time), and the second one is area-optimized and time-consuming (SAD-area). We note that for SAD-time, the optimal unroll factor is 6 with a speedup of 8.71 and with an area consumption of 79% (92.2% occupied area for unroll factor 7 would have been too much and it would have made the design slow). For SAD-area, the optimal unroll factor is 13, with an area consumption of 88.5% and speedup of 8.08.

   The decision is to use SAD-time, which gives better performance and requires less area. Nevertheless, we must mention that the decision on which implementation to use differs from case to case as it cannot be taken without comparing the best solutions for the different implementations.

   The maximum Amdahl based speedup is 35.28. In this case, the performance achieved with our method is only at 24.7% of the theoretical maximum. As specified in the previous chapter, a part of the explanation is that Amdahl's Law does not take into account memory bottlenecks. Memory accesses represent an important part of the total execution time of the SAD kernel, leading to only a slow speedup increase when unrolling. Also, the time for the software function (T_{sw}) is smaller than the time for the memory transfers $T_{\text{max(r,w)}}$, meaning that the execution on the GPP is completely hidden by the execution on the FPGA when loop unrolling and shifting are performed.

   In conclusion, this is an example of a K-loop that would benefit from an optimized hardware implementation of the kernel that would be able to reduce the memory access time.

4. **Quantizer (JPEG)**. Out of the four Quantizer implementations we tested (Q-1, Q-2, Q-4 and Q-8), Q-1 is the slowest but requires the least area, and Q-8 is the fastest and requires the most area. As explained in the previous chapter, since the execution time for the software function is

significantly larger than the execution time for the kernel running in hardware, the performance increase is negligible, as it can be seen from Figure 5.5(d). When applying also loop shifting, the speedup slightly increases compared to when only loop unrolling is used, but S_{shift} is $\approx$ 2.52 for all unroll factors and even for all Quantizer implementation.

Kernel speedup and area consumption are represented separately. We notice that a kernel speedup of 40 is possible, with an area consumption of 50% (Q-4 for $u = 7$), but the loop speedup for the same unroll factor is only 2.4.

As a conclusion, the best solution in this context is to apply shifting without unrolling. Transforming the K-loop with loop shifting leads to a speedup of 2.52, slightly better than the speedup of 2.32 achieved for Q-8 without any transformation, only with the kernel accelerated in hardware. This performance is also very close to the theoretical maximum given by Amdahl's Law, which is $\approx$ 2.53. The area consumption is 12.13% of the Virtex II-Pro area.

The experiments show the following possible situations:

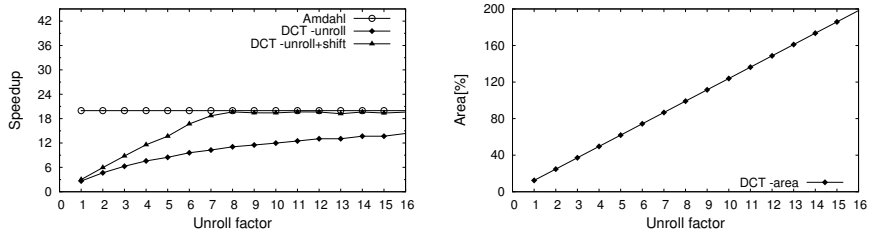
1. $T_K(\text{hw}) >> T_{\text{sw}}$ (DCT, SAD). In this case, loop unrolling plus shifting will be the most beneficial. It will also matter how much optimized the kernel implementation is, but only up to the point when $T_K(\text{hw})$ becomes comparable to or less than T_{sw}. When that happens, the conclusions for the second or third category apply (see below).

2. $T_K(\text{hw}) \approx T_{\text{sw}}$ (Convolution). Most probably, a combination between shifting and unrolling with a factor 2 is the best solution.

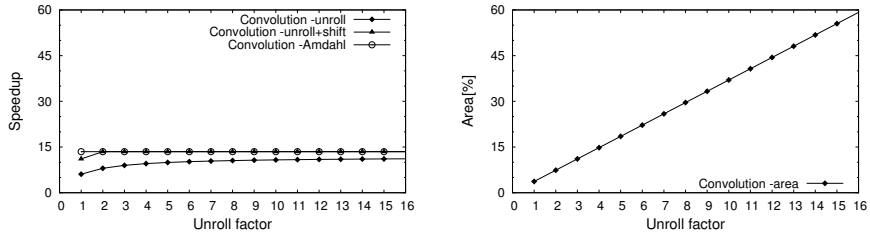
3. $T_K(\text{hw}) << T_{\text{sw}}$ (Quantizer). In this case, unrolling is not beneficial, which is also seen from Amdahl's Law. By applying loop shifting without unrolling there will be a slight performance improvement, while the area requirements will not change.

5.4 Conclusions

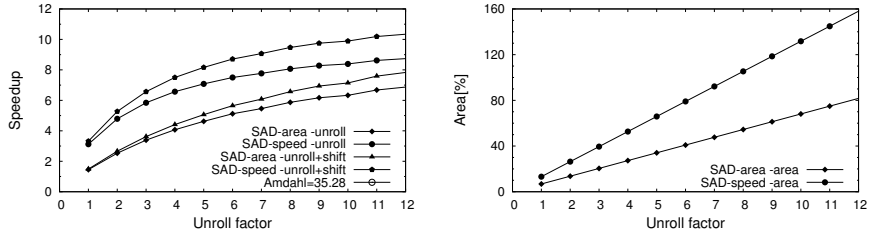
Loop shifting is used for eliminating data dependencies for K-loops with one hardware-mapped kernel. As a consequence, loop shifting exposes the software-hardware parallelism and enables concurrent execution of the software functions and the hardware-mapped kernels.



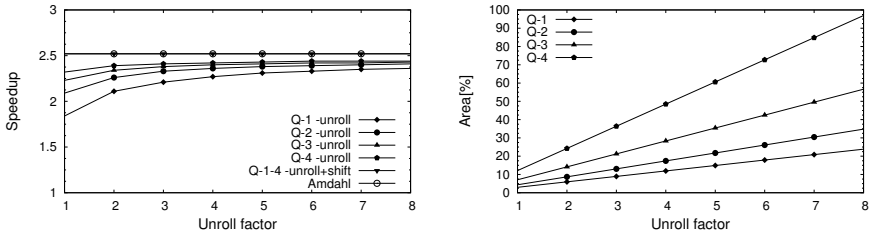
(a) DCT



(b) Convolution



(c) SAD



(d) Quantizer

Figure 5.5: Loop shifting: speedup with unrolling and shifting for: a) DCT; b) Convolution; c) SAD; d) Quantizer.

In Section 5.2, we have proven that combining loop shifting with loop unrolling always gives results that are superior (or, in the worst case, equal) to those when only loop unrolling is used. There are various situations that can appear, when the best performance is achieved by using a combination of loop unrolling and loop shifting, or when loop shifting is used by itself.

In the experimental part of this chapter, we looked at the same kernels that were considered in Chapter 4, where the K-loops were parallelized by using loop unrolling. The results in Table 5.1 show that when the execution time of the software part of the K-loop is much larger than that of the kernel executing in hardware, as in the case of Quantizer, or when the kernel is I/O intensive, unrolling is not beneficial and the K-loop should be transformed only with loop shifting. When the execution time of the hardware kernel is much larger than that of the software part, then several instances of the software function can be 'hidden' by a kernel running in hardware, and a combination of loop unrolling with loop shifting gives the best performance. This was the case for the DCT and SAD K-loops. When the software and hardware have similar execution times in an iteration of the original K-loop, most probably a combination between shifting and unrolling with a factor 2 is the best solution, as seen in the case of Convolution.

Loop shifting is a particular case of software pipelining and is suitable for K-loops with a single hardware-mapped kernel. For larger K-loops, the transformation called K-pipelining, presented in the next chapter, should be used. First, we will show the mathematical model of using K-pipelining, followed by theoretical results on randomly generated test cases, and by experimental results for the MJPEG application.

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

O.S. Dragomir, T. Stefanov, K. Bertels, **Loop Unrolling and Shifting for Reconfigurable Architectures**, FPL 2008
O.S. Dragomir, T. Stefanov, K. Bertels, **Optimal Loop Unrolling and Shifting for Reconfigurable Architectures**, TRETS 2009

6 K-Pipelining

IN THIS CHAPTER, we present the K-pipelining technique, a version of loop pipelining specifically adapted for K-loops. In Section 6.1.1, the difference between loop shifting and K-pipelining is illustrated on a large K-loop with two hardware mapped kernels and three software functions .

Section 6.2 presents the methodology for applying K-pipelining and unrolling for the parallelization of large K-loops. In order to study the potential for performance improvement of using the K-pipelining on K-loops, a test suite composed of randomly generated test cases, and a K-loop extracted from the MJPEG application are used. The related experimental results are presented in Section 6.3. Finally, Section 6.4 concludes this chapter.

6.1 Introduction

The previous chapters focused on simple K-loops containing only one hardware kernel to accelerate on the FPGA and some software code that always executes on the GPP. For larger loops that contain several software functions more hardware-software parallelism is possible. This parallelism can be exposed using the K-pipelining technique, which is presented in this chapter.

The loop pipelining transformation is illustrated in Figure 6.1. It consists of relocating (shifting) part of the functions in the loop – in our example, two out of the three functions are relocated. In the loop prologue, a first instance of function `A` is executed, followed by a second instance of function `A` and by the first instance of function `B`. As a result, in the loop body there are now function calls from different iterations: `A(i + 2)`, `B(i + 1)` and `C(i)`, which can now execute in parallel. The execution order of functions `A`, `B` and `C` is preserved.

The purpose of the K-pipelining transformation is to eliminate the data depen-

dencies between the software and hardware functions and allow them to execute in parallel. Assuming that the software and hardware functions alternate within the K-loop body, then half of the total number of functions will need to be relocated. The differences between the K-pipelining technique and loop shifting and software pipelining are illustrated in Section 6.1.1.

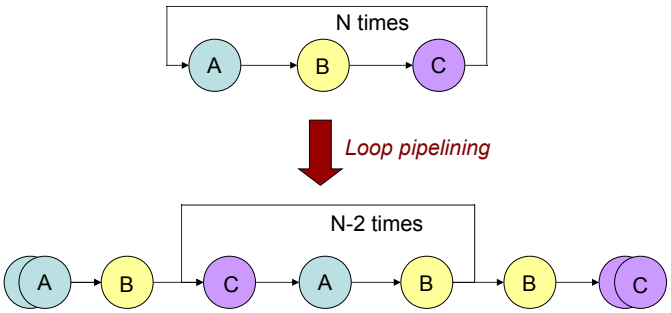


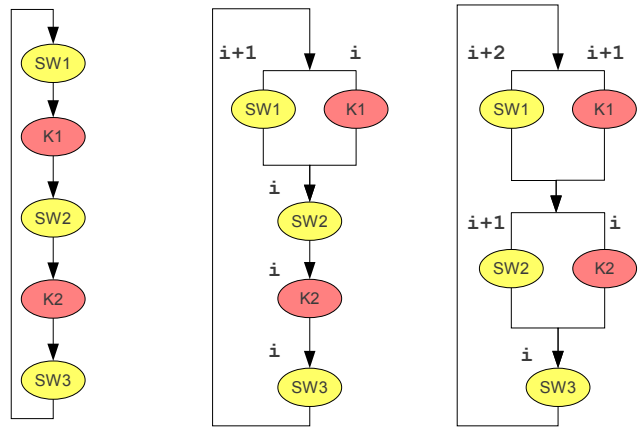
Figure 6.1: Loop pipelining for K-loops (K-pipelining).

6.1.1 Example

A generic K-loop is illustrated in Figure 6.2(a). The example shows a K-loop with several functions. The SW_j functions are always executed on the GPP, while the K_i functions are the application kernels that are meant to be accelerated in hardware. These can be viewed as a task chain, where we assume that there are dependencies between consecutive tasks in the chain (intra-iteration dependencies), but not between any two tasks from different iterations (no inter-iteration dependencies).

In Figure 6.2(b), we illustrate the execution model of the K-loop when the simple loop shifting technique is applied. In this case, the first kernel of the K-loop body executes in parallel with the first software function from a different iteration. The K-loop prologue and epilogue resulted from the shifting are not shown on the figure. Considering that the base iteration is `i`, the corresponding original iterations (`i`, `i + 1`) for each function call are showed on the figure.

In Figure 6.2(c), the execution model of the K-loop transformed by K-pipelining is illustrated. Each of the kernels in the K-loop executes in parallel with its preceding software function. As in the case of simple loop shifting, the K-loop prologue and epilogue are not shown on the figure. Considering that the base iteration is `i`, the corresponding original iterations (`i`, `i + 1`,`i + 2`) for each function call are showed on the figure.



(a) Loop (b) Shifted loop (c) K-pipelined loop

Figure 6.2: K-loop with 2 hardware mapped kernels and 3 software functions.

The code in Figure 6.3 is used to illustrate the difference in parallelization via simple loop shifting, full loop pipelining (software pipelining correspondent at function level), and the newly introduced K-pipelining. The shifted loop is presented in Figure 6.4, the fully pipelined loop in Figure 6.5, and the K-pipelined loop in Figure 6.6. The meaning of the curly brackets ({ and }) is that the functions called in between them are scheduled to be executed in parallel. For more clarity, all function calls from a prologue or epilogue iteration are put on the same line.

```
for (i = 0; i < N; i++) do
    A(i);
    B(i);
    C(i);
    D(i);
    E(i);
end
```

Figure 6.3: Original loop to be parallelized.

```
A(0);
for (i = 0; i < N − 1; i++) do
    { A(i+1); B(i); }
      C(i);
      D(i);
      E(i);
end
B(N − 1); C(N − 1); D(N − 1); E(N − 1) ;
```

Figure 6.4: Shifted loop.

```
  A(0);
{ A(1); B(0); }
{ A(2); B(1); C(0); }
{ A(3); B(2); C(1); D(0); }
for (i = 0; i < N − 4; i++) do
    { A(i+4); B(i+3); C(i+2); D(i+1); E(i); }
end
{ B(N − 1); C(N − 2); D(N − 3); E(N − 4); }
{ C(N − 1); D(N − 2); E(N − 3); }
{ D(N − 1); E(N − 2); }
  E(N − 1);
```

Figure 6.5: Fully pipelined loop.

```
  A(0);
{ A(1); B(0); } C(0);
for (i = 0; i < N − 1; i++) do
    { A(i+2); B(i+1); }
    { C(i+1); D(i); }
      E(i);
end
  B(N − 1); { C(N − 1); D(N − 2); } E(N − 2);
  D(N − 1); E(N − 1);
```

Figure 6.6: K-pipelined loop.

6.2 Mathematical Model

The loop shifting technique presented in Chapter 5 is suitable for single-kernel K-loops. For larger K-loops, more parallelism can be exploited if more functions can be relocated in order to break the dependencies. We are not interested in a fully parallel K-loop body. That would imply that different kernels execute in parallel. Instead, we are interested in a half-parallel loop body, which allows for each kernel to execute in parallel with its preceding software function. Therefore, we do not need to implement a full coarse-grain software pipelining technique, but a simpler type of pipelining, which we refer to as K-pipelining.

There are several K-loop requirements for the K-pipelining transformation to make sense:

1. no inter-iteration dependencies are allowed between consecutive functions;
2. intra-loop dependencies between consecutive functions should exist;
3. the hardware mapped kernels and the software scheduled functions should alternate within the K-loop body, like in Figure 6.2.

In order to compute the total execution time for the K-pipelined loop, we assume that the first function in the K-loop is a software only function. We use the notations in Section 3.3.1 and, in addition, the notations presented below.

- T_{ep} (T_{ep}^{odd} / T_{ep}^{even} for N_f odd/even) — the execution time for the prologue and epilogue (cycles);
- T_{iter} (T_{iter}^{odd} / T_{iter}^{even} for N_f odd/even) — the execution time for one iteration of the K-pipelined loop (cycles);
- N_{iter} — the number of iterations of the new K-loop body:

$$N_{iter} = N - N_k\,. \tag{6.1}$$

The execution time for a K-loop with size larger than 1, running completely in software, is:

$$T_{\text{loop(sw)}} = (\sum_{i=1}^{N_k} T_{K_i} + \sum_{i=1}^{\lceil N_f/2 \rceil} T_{SW_i}) \cdot N\,. \tag{6.2}$$

When a software function and a hardware kernel execute in parallel, the maximum of their execution times is taken into account when computing the overall time.

The execution time for one iteration when N_f is even ($N_f = 2 \times N_k$) is:

$$T_{iter}^{even} = \sum_{i=1}^{N_k} \max(T_{K_i}, T_{SW_i}). \tag{6.3}$$

When N_f is an odd number, *e.g.* $N_f = 2 \times N_k + 1$, the execution time for the last software function needs to be taken into account. The execution time for one iteration is:

$$T_{iter}^{odd} = T_{iter} + T_{SW_{N_k+1}}. \tag{6.4}$$

Each function is called N_k times in the added prologue and epilogue. This means that the added size of the prologue and epilogue for K-pipelining is $N_k \times N_f$ function calls (for full pipelining it would be $N_f \times N_f$ function calls).

For maximum performance, we exploit the hardware-software parallelism in the prologue and epilogue as well. Except for one occurrence, all hardware functions called in the prologue and epilogue will execute in parallel with the corresponding software functions. For K-loops with an odd number of functions, the last one will not have a corresponding parallel function in the prologue/epilogue, as it does not have in the transformed K-loop body.

The execution time for the prologue and epilogue when N_f is even is:

$$T_{ep}^{even} = (N_k - 1) \cdot (\sum_{i=1}^{N_k} \max(T_{K_i}, T_{SW_i})) + \sum_{i=1}^{N_k} T_{K_i} + \sum_{i=1}^{N_k} T_{SW_i}; \tag{6.5}$$

$$T_{ep}^{even} = (N_k - 1) \cdot T_{iter} + \sum_{i=1}^{N_k} T_{K_i} + \sum_{i=1}^{N_k} T_{SW_i}. \tag{6.6}$$

Similarly, the execution time for the prologue and epilogue when N is odd is:

$$T_{ep}^{odd} = (N_k - 1) \cdot T_{iter}^{odd} + \sum_{i=1}^{N_k} T_{K_i} + \sum_{i=1}^{N_k+1} T_{SW_i}. \tag{6.7}$$

In general, the total execution time is:

$$T_{pipe} = T_{ep} + N_{iter} \cdot T_{iter}; \tag{6.8}$$

$$T_{pipe} = (N - 1) \cdot T_{iter} + \sum_{i=1}^{N_k} T_{K_i} + \sum_{i=1}^{\lceil N_f/2 \rceil} T_{SW_i}. \tag{6.9}$$

The K-loop speedup is computed by dividing the execution time for the pure software K-loop by the execution time of the K-pipelined K-loop:

$$S_{\text{pipe}} = \frac{T_{\text{loop(sw)}}}{T_{\text{pipe}}}. \tag{6.10}$$

When combining the K-pipelining with loop unrolling, the following formulas are used to compute the execution time for the prologue and epilogue, as well as the total execution time, for a certain unroll factor u. The number of iterations for the transformed loop body with unroll factor u is:

$$N_{iter}(u) = (N - N_k \cdot u)/u\,. \tag{6.11}$$

The execution time for one iteration is:

$$T_{iter}^{even}(u) = \sum_{i=1}^{N_k} \max(T_{K_i}(u), u \cdot T_{SW_i}); \tag{6.12}$$

$$T_{iter}^{odd}(u) = T_{iter}(u) + u \cdot T_{SW_{N_k+1}}\,. \tag{6.13}$$

The execution time for the prologue and epilogue is:

$$T_{ep}^{even}(u) = (N_k - 1) \cdot T_{iter}(u) + \sum_{i=1}^{N_k} T_{K_i}(R) + \sum_{i=1}^{\lceil N_f/2 \rceil} u \cdot T_{SW_i} + \\ + \sum_{i=1}^{N_k} max(T_{K_i}(u), R \cdot T_{SW_i}); \tag{6.14}$$

$$T_{ep}^{odd}(u) = T_{ep}^{even}(u) + (R + u) \cdot T_{SW_{N_k+1}}\,. \tag{6.15}$$

The total execution time when unroll factor u is used is:

$$T_{pipe}(u) = T_{ep}(u) + N_{iter}(u) \cdot T_{iter}(u); \tag{6.16}$$

$$T_{pipe}(u) = (N/u - 1) \cdot T_{iter}(u) + \sum_{i=1}^{N_k} T_{K_i}(R) + \sum_{i=1}^{\lceil N_f/2 \rceil} u \cdot T_{SW_i} + \\ + \sum_{i=1}^{N_k} max(T_{K_i}(u), R \cdot T_{SW_i}) + (R + u) \cdot T_{SW_{N_k+1}}\,. \tag{6.17}$$

6.3 Experimental Results

This section presents two sets of results. In the first part, the random test generator presented in Section 3.4 was used to generate a test suite and study the potential for performance improvement of unrolling combined with K-pipelining on arbitrary K-loops. Results for the randomly generated tests are presented in Section 6.3.1.

Experimental results for a K-loop extracted from the MJPEG application are presented in Section 6.3.2.

6.3.1 K-pipelining on Randomly Generated Tests

A test suite composed of randomly generated test cases has been used in order to study the impact of K-pipelining on K-loops with sizes between 2 and 8. For each loop size, 1500 tests have been generated. For each test, we compare the performance improvement when applying loop unrolling with the performance improvement when unrolling and K-pipelining are applied to the original K-loop.

To study the impact of the area requirements, we made two sets of tests. For the first set, all kernels have small area requirements, between 1.2% and 12%. For the second test, the maximum area requirement for a kernel is 60%. However, the kernels should fit altogether on the available area. Therefore, the sum of all areas should be less than the maximum allowed (in our tests, this maximum is 90%).

In Figure 6.7, we illustrate the performance improvement for the different loop sizes when applying the K-pipelining method combined with loop unrolling, compared to loop shifting and unrolling. For Figure 6.7(a), the set of tests was generated with Small Area Requirement (SAR) for each kernel, between 1.2% and 12%. In Figure 6.7(b), the displayed results are for the case of Larger Area Requirements (LAR), maximum 60% per kernel. In both cases, the K-pipelining brings no improvement for K-loops of size 2 and shows variable improvement for all K-loop sizes larger than 2. Little performance improvement (less than 10%) is shown in only 1% of the SAR cases, compared to 20% for LAR cases. An improvement of 10–20% is shown for 12% of the SAR cases and the majority (56%) of the LAR cases. A better performance gain, between 20–40% is shown for the majority of the SAR cases (61%) and for 21.5% of the LAR cases. A significant performance improvement of more than 40% is shown for more than 25% of the SAR cases and only 2% of the LAR cases.

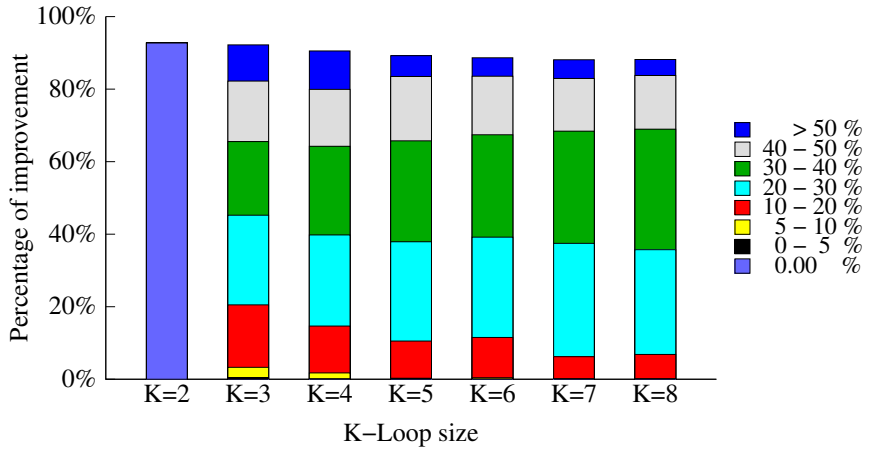
The conclusion of this experiment is that the K-pipelining technique combined with loop unrolling has good potential for improving K-loops performance, for K-loops that contain more than two hardware mapped kernels. The results show that there is a larger probability of a better speedup when all kernels have small area requirements. The reason for this is simple: large area requirements means little or no hardware parallelism because of the area constraints. In this case, all the performance gain comes from the hardware-software parallelism enabled by the K-pipelining. An interesting fact is also that the numbers are quite evenly distributed for various K-loop sizes larger than 2. The reason for this is that the K-pipelining is scaled to the number of kernels in the K-loop.

6.3.2 K-pipelining on the MJPEG K-loop

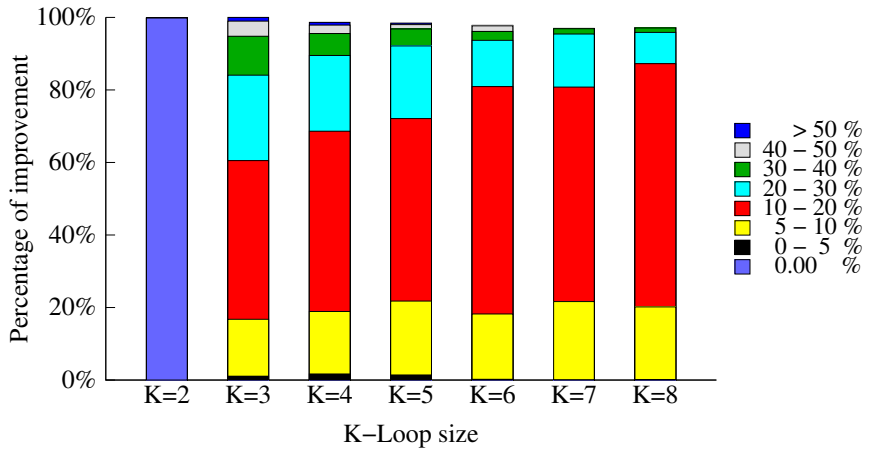
Motion JPEG is a video format where each video frame or interlaced field of a digital video sequence is separately compressed as a JPEG image. It uses a lossy form of intra-frame compression based on the Discrete Cosine Transform (DCT). This mathematical operation converts each frame/field of the video source from the time domain into the frequency domain. A perceptual model based loosely on the human psycho-visual system discards high-frequency information, i.e. sharp transitions in intensity, and color hue. In the transform domain, the process of reducing information is called quantization. Basically, quantization is a method for optimally reducing a large number scale (with different occurrences of each number) into a smaller one, and the transform-domain is a convenient representation of the image because the high-frequency coefficients, which contribute less to the over picture than other coefficients, are characteristically small-values with high compressibility. The quantized coefficients are then sequenced and packed into the output bitstream.

The main parts of the MJPEG algorithm are: the Discrete Cosine Transform (DCT), the Quantization (Q), and the Variable Length Encoder (VLE). The DCT and Quantizer functions have been previously used to illustrate the use of loop unrolling and loop shifting. The VLE function is based on the Huffman encoder, and consists of two main parts: the EncodeAC() function, that takes as input the pixel block and encodes it by passing the values of the codes found to the Huffman package, and the EncodeDC() function, that encodes the input coefficient to the stream using the currently installed DC Huffman table. In addition to these functions, there are some auxiliary functions that are used for pre- or post-processing of the data. These functions are:

- PreShift — subtracts 128 (2048) from all matrix elements. This results



(a) Area requirements between 1.2% and 12%.



(b) Area requirements between 1.2% and 60%.

Figure 6.7: K-pipelining: performance distribution for the randomly generated tests.

Table 6.1: Profiling information for the MJPEG functions.

Kernel	$T_{K(\text{sw})}$(cycles)	$T_{K(\text{hw})}$(cycles)
PreShift	3096	–
DCT	106626	37278
Bound	1953	–
Quantizer	7854	2862
ZigZag	8112	–
VLE	51378	6252

in a balanced DCT without any DC bias;

- Bound — clips the Dct matrix such that it is no larger than a 10 bit word (1023) or a 14 bit word (4095);
- ZigZag — performs a zig-zag translation on the input matrix.

The MJPEG main loop is illustrated in Figure 6.8. The data transmitted from task to task is a 8×8 pixel block. All functions need to perform the reading and writing of each pixel in the data block. This means that there are 64 memory reads and 64 memory writes per function. All functions, except VLE, have a constant execution time, independent of the input data.

```
for (i = 0; i < VNumBlocks; i++) do
    for (j = 0; j < HNumBlocks; j++) do
        PreShift(blocks[i][j]);
        DCT(blocks[i][j]);
        Bound(blocks[i][j]);
        Quantizer(blocks[i][j]);
        ZigZag(blocks[i][j]);
        VLE(blocks[i][j]);
    end
end
```

Figure 6.8: The MJPEG main K-loop.

The VHDL code for the DCT, Quantizer and VLE functions has been automatically generated with the DWARV [149] tool and synthesized using the

Table 6.2: Synthesis results for the MJPEG kernels.

Kernel	Area (slices)	% VirtexIIPro	%Virtex4
DCT	1692	12.35	4.72
Quantizer	409	2.98	1.14
VLE	10631	77.62	29.66

Xilinx XST tool of ISE 11.2 for the Xilinx VirtexIIPro-XC2VP30 and Virtex4-XC4VLX80 boards. The code was executed only on the Virtex II Pro-XC2VP30 board and the execution times were measured using the PowerPC timer registers. For the considered implementation, the shared memory has an access time of 3 cycles for reading and storing the value into a register and 1 cycle for writing a value into memory. The profiling information for all the functions in the MJPEG K-loop is presented in Table 6.1. The software and hardware execution times for VLE that are presented in the profiling information are the average of the execution times of several instances of the VLE kernel. The synthesis results for the MJPEG kernels are presented in Table 6.2.

Note that the ZigZag function, although has a quite large execution time, is a software only function, as it is I/O intensive and its implementation in hardware would not be beneficial.

Figure 6.9 illustrates the speedup (compared to the pure software execution) that can be achieved for the MJPEG K-loop with various techniques. The K-pipelining combined with unrolling is compared with simple unrolling, and with shifting plus unrolling. Note that due to the area constraints, the chosen unroll factor for the Virtex-II Pro platform is 1 (no unroll), while for Virtex-4 it is 2. The K-pipelining performs better than the shifting plus unrolling, and significantly better than the loop unrolling (more than 20% improvement). The final speedup for the Virtex-II Pro is 3.69 and for the Virtex-4 it is 6.16.

The performance improvement depends heavily on the ratio between the execution time of the hardware kernels and the execution time of the software functions, which influence the efficiency of the hardware-software parallelism. Because some of the software functions in the MJPEG K-loop have a large execution time, the achieved performance improvement is not large, but only satisfactory.

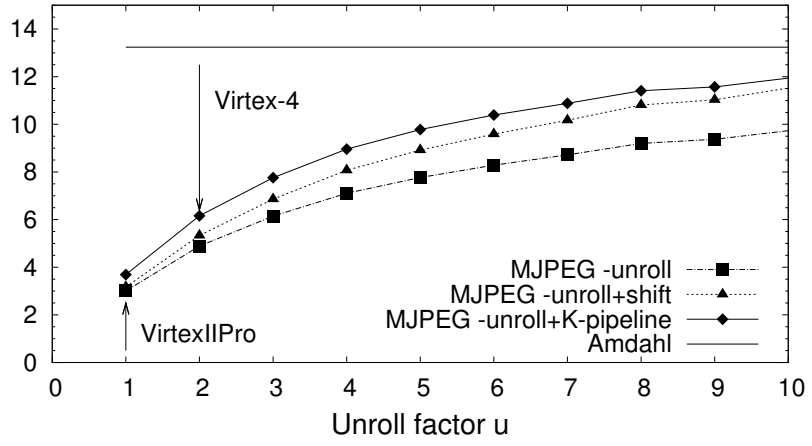


Figure 6.9: K-pipelining: MJPEG speedup with unrolling, unrolling + shifting, and unrolling + K-pipelining.

6.4 Conclusions

K-pipelining provides the means for exploiting more hardware-software parallelism for K-loops that contain more than one hardware mapped kernel. Experimental results on randomly generated tests show that the K-pipelining technique combined with loop unrolling has good potential for improving K-loops performance, for K-loops that contain more than 2 hardware mapped kernels. The results show that there is a larger probability of a better speedup when all kernels have small area requirements, since large area requirements means little or no hardware parallelism. When the area constraints limit the hardware parallelism, the performance gain comes from the hardware-software parallelism enabled by the K-pipelining. The results also show that the K-pipelining is scaled to the number of kernels in the K-loop.

Experimental results on the MJPEG K-loop show a performance improvement of 16% compared to unrolling plus shifting and more than 20% compared to unrolling only. The performance could be improved if the software functions would be optimized such that more software-hardware parallelism would be possible.

The next chapter of this dissertation is dedicated to loop distribution, also known as loop splitting or loop fission. This transformation is suitable for large K-loops. The motivation for using loop distribution is that for a large K-loop,

the degree of parallelism may be more limited than the degrees of parallelism of the resulted K-sub-loops. An algorithm of applying loop distribution in the context of K-loops will be presented, followed by theoretical results on randomly generated test cases, and by experimental results for the MJPEG application.

7

Loop Distribution for K-loops

IN THIS CHAPTER, we discuss another traditional loop transformation, loop distribution, and see how it can be used in the context of K-loops. In Section 7.2 we propose an algorithm for distributing K-loops that contain more than one kernel. The purpose is to split large K-loops into smaller K-sub-loops and apply the previously proposed loop transformations - e.g. unrolling and shifting/K-pipelining to the K-sub-loops. In order to study the potential for performance improvement of using the loop distribution on K-loops, we make use of a suite of randomly generated test cases. The results of applying the distribution algorithm to the randomly generated tests are shown in Subsection 7.3.1. The algorithm is also validated with a K-loop extracted from the Motion JPEG (MJPEG) application in Subsection 7.3.2.

7.1 Introduction

Loop distribution is a technique that breaks a loop into multiple loops over the same index range but each taking only a part of the loop's body. In our work, loop distribution is used to break down large K-loops (more than one kernel in the loop body) into smaller ones, in order to benefit more from parallelization with loop unrolling and loop shifting or K-pipelining. Note that according to Chapter 5 the term shifting is used when the loop is shifted one position only (suitable for K-loops with one hardware-mapped kernel) and K-pipelining is the generalized version, used for K-loops with more than one kernel.

A generic representation of the loop distribution transformation is presented in Figure 7.1. In the top part of the figure, a loop with three tasks is illustrated. In the bottom part, each task is within its own loop and the execution order of the tasks has not been changed.

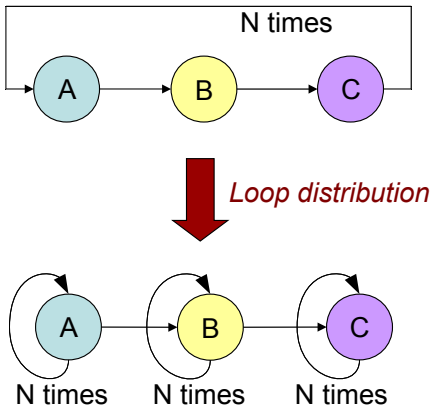


Figure 7.1: Loop distribution.

7.1.1 Motivation

In the previous chapters we focused mostly on simple K-loops with one hardware-mapped kernel, which is accelerated on the FPGA, and some software code, which always executes on the GPP. For this kind of K-loops, we proposed algorithms for loop unrolling (Chapter 4) and loop unrolling plus shifting (Chapter 5) to determine which would be the best unroll factor that would allow the maximum parallelization and performance. For generic K-loops with an arbitrary number of kernels and pieces of software code occurring in between the kernels, the K-pipelining method combined with loop unrolling has been proposed in Chapter 6. However, this method does not exploit the hardware parallelism at its best, as it forces the same degree of parallelism for all kernels in the K-loop. An example of a K-loop containing two hardware mapped kernels and three software functions illustrated in Algorithm 7.2.

```
for (i = 0; i < N; i++) do
    SW1(blocks[i]);// software function
    K1(blocks[i]);// hardware mapped kernel
    SW2(blocks[i]);// software function
    K2(blocks[i]);// hardware mapped kernel
    SW3(blocks[i]);// software function
end
```

Figure 7.2: K-loop with 2 kernels.

The K-loop in the example contains several functions – the SW_j functions will always execute on the GPP, while the K_j functions are the application kernels, which are accelerated in hardware. This K-loop can be viewed as a task chain, where we assume that there are dependencies between consecutive tasks in the chain, but not between any two tasks from different iterations. Inter-iteration dependencies can be handled with other loop transformations such as loop skewing and are not within the scope of this thesis. As specified in Chapter 6, the software functions situated before and after the kernel are called the kernel's pre- and post- functions. The loops that result from the splitting of the original K-loop are called K-sub-loops.

It is obvious that for large K-loops, unrolling and/or K-pipelining have limitations in regard to exposing the available parallelism. In the case of loop unrolling, different kernels in the K-loop may benefit from different unroll factors, due to various factors: the kernel's I/O, the kernel's area requirements, the kernel's acceleration factor, the relation between the kernel and the pre-/post- software functions. Since different kernels could benefit from different unrolling factors, it means that performance could be improved if the unrolling and shifting/K-pipelining would be applied on smaller K-loops.

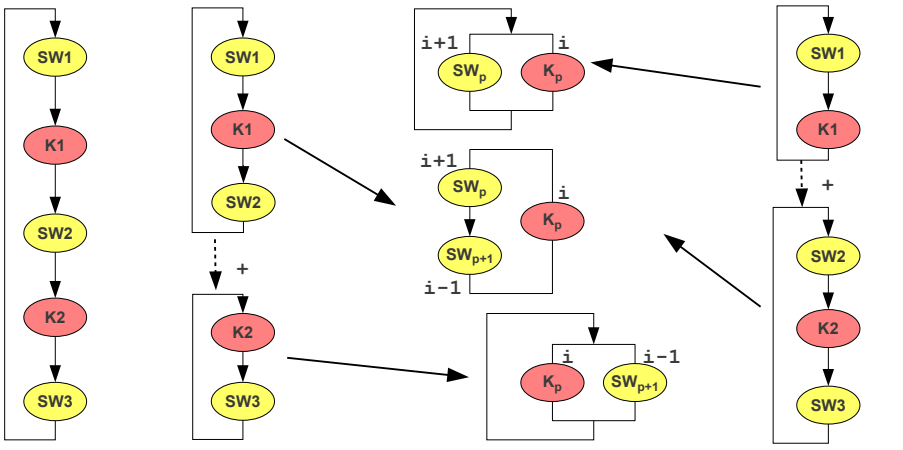


Figure 7.3: Possibilities of loop distribution.

7.2 Mathematical Model

In this section, we propose a method for splitting a K-loop containing interleaved kernels and software functions. A K-sub-loop will contain one or more kernels. The K-loop breaking points are considered to be the points between two subsequent kernels that will be placed in different K-sub-loops. Assuming that the software functions and the kernels are placed evenly within the K-loop such that between any two kernels there is a software task, a decision has to be made whether to distribute the software task with the first kernel or with the second one. This kind of decision can be taken at each breaking point of the loop.

Figure 7.3 shows a loop with two hardware kernels and three software functions. The possibilities of breaking this loop between the two kernels are illustrated in Figure 7.3b) and Figure 7.3d). In Figure 7.3c) we illustrated the parallel execution model for each of the three cases of a loop containing one hardware kernel inside, depending on the position of the software code, where $i-1$, i, and $i+1$ are the iteration numbers.

The algorithm uses the profiling information about memory transfers, execution times for all the software and hardware functions, area requirements for the kernels, memory bandwidth and available area. The proposed algorithm is a Greedy type of algorithm, applied to the sorted list of kernels. The sorting is performed according to a heuristic based on the hardware execution time, the memory constraints and the relation with the pre-/post- software function for each kernel. The first kernel in the list is the most promising kernel in terms of speedup. The algorithm is illustrated in Algorithm 7.4.

The sorted list of kernels is processed in a Deep First Search manner. Since each function (either kernel or software) can belong to only one K-sub-loop, the function will be processed (added to the current K-sub-loop) only if it has not been previously selected in a K-sub-loop. The order of the functions is preserved in the distributed K-loop because of intra-iteration dependencies. We will call a function that does not belong to a K-sub-loop a 'free' function. For each free kernel in the list, the following steps are performed.

1. A K-sub-loop is created (denoted by L), containing the current kernel. The pre- and post- functions are added to L if they are free.
2. While the speedup of L is less than the speedup of the original (undistributed) K-loop, and while the next kernel and its pre- function are free, they are added to L.

```
S = Loop.GetSpeedup(); // get loop speedup
Klist.OrderKernels(); // order kernel list
Llist.Init(); // initialize loop list
// start with the first kernel in the list
K = Klist.First();
while (K != NULL) do
    if (!K.added) then
        L = new Loop();
        L.AddKernel(K);
        L.AddSwFunc(K);
        S_temp = L.GetSpeedup();
        if (S_temp < S) then
            next = K.GetNext();
            prev = K.GetPrev();
            while (next && !next.added && S_temp < S) do
                L.AddKernel(next);
                L.AddSwFunc(next);
                next = next.GetNext();
                S_temp = L.GetSpeedup();
            end
            while (prev && !prev.added && S_temp < S) do
                L.AddKernel(prev);
                L.AddSwFunc(prev);
                prev = prev.GetPrev();
                S_temp = L.GetSpeedup();
            end
        end
        Llist.AddLoop(L);
    end
    K = Klist.Next();
end
```

Figure 7.4: The distribution algorithm.

3. While the speedup of L is less than the speedup of the original (undistributed) K-loop, and while the previous kernel and its post- function are free, they are added to L.

4. L is saved in the list of K-sub-loops.

The following considerations apply regarding the software functions.

1. If the first function in the K-loop is a software function, it will always be grouped with the first kernel.

2. If the last function in the K-loop is a software function, it will always be grouped with the last kernel.

3. The software functions that are called in between kernel functions will be grouped with either the previous or the following kernel function. This decision is taken at the stage of kernel ordering.

7.2.1 Reconfigurability Issues

The loop distribution can be performed either considering that all kernel instances should fit on the FPGA or considering that the FPGA will be (partially) reconfigured in between K-sub-loops to accommodate the new kernels.

Of course trying to fit all kernel instances on the FPGA imposes severe area constraints, especially in the case of small FPGAs such as the VirtexIIPro-XC2VP30. This may result in poor or no improvement when parallelizing the split loop because of the limitations imposed on the unroll factors.

However, larger FPGAs such as the Virtex4-XC4VLX80 may offer enough space for configuring all the needed kernel instances at once. Then the true potential for performance improvement of loop distribution can be seen.

Reconfiguration (either partial or total) can be expensive in terms of CPU cycles. Although part of the reconfiguration time can be hidden by the K-sub-loops prologue/epilogue (that appear because of loop unrolling and loop shifting/K-pipelining), the reconfiguration overhead might be so big that the performance decreases instead of increasing.

Because of the different technologies involved, different platforms have different reconfiguration times. For the random tests we have considered so far that the reconfiguration latency does not exist or can be hidden. In the future, we will refer to a specific platform and compute what is the reconfiguration latency that

we can afford in order to still have an improvement when applying the loop distribution technique.

Reconfiguration on Virtex II Pro. The Virtex-II Pro configuration memory is arranged in vertical frames that are one bit wide and stretch from the top edge of the device to the bottom. These frames are the smallest addressable segments of the Virtex-II Pro configuration memory space; therefore, all operations must act on whole configuration frames [147].

Configuring the entire device takes approx. 20ms. However, Virtex-II Pro devices allow partial reconfiguration – rewriting a subset of configuration frames. We consider that the time needed to reconfigure only a part of the device is proportional to the size of the module to be configured.

When computing the reconfiguration latency for the kernels in our experiments, we will be considering 2 strategies. The first one uses the official Xilinx reconfigurability facilities, with no further optimizations. The second considers the work of Claus et al [28, 29] who have developed ways to accelerate partial reconfiguration on Xilinx FPGAs. The authors first developed Combitgen, a tool that generates optimized partial bit-streams. It reduces the reconfiguration time for VirtexIIPro by a factor of 96.9, as the new bitstream can be configured in 0.20ms, compared to 19.38ms when using the Xilinx Early Access Partial Reconfiguration (EAPR) to create the bitstream. In [29], a method to calculate the expected reconfiguration throughput and latency is presented. In addition, the authors propose a PLB ICAP (Processor Local Bus Internal Configuration Access Port) controller that enables fast on-chip Dynamic Partial Reconfiguration (DPR) close to the maximum achievable speed. Compared to an alternative state-of-the art realization, an increase in speed by a factor of 18 in the case of VirtexIIPro can be obtained.

Reconfiguration on Virtex-4. The Virtex-4 FPGA family marks a significant change in layout over previous devices. The configuration architecture is still frame-based, but a frame spans 16 rows of Configurable Logic Blocks (CLBs) rather than the full device height. These frames are the smallest addressable segments of the Virtex-4 configuration memory space, and all operations must therefore act upon whole configuration frames [146].

The number of frames that need to be configured determines the size of the reconfiguration bitstream. There is also a configuration overhead of 1312 words. In order to compute the configuration time, the throughput must also be known.

The idealized throughput on the Virtex-4 chip is 400MB/s, at 100MHz. The work of Liu et al [88] proposed a Direct Memory Access (DMA) engine design that yields a throughput of 399MB/s. Considering that the size of a frame is 41 words, the reconfiguration time is $0.411\mu s$/frame. When adding the overhead, the total reconfiguration time is

$$T_{reconfig} = \text{Number\_of\_frames} * 0.411 + 13.15(\mu s) \quad (7.1)$$

7.3 Experimental Results

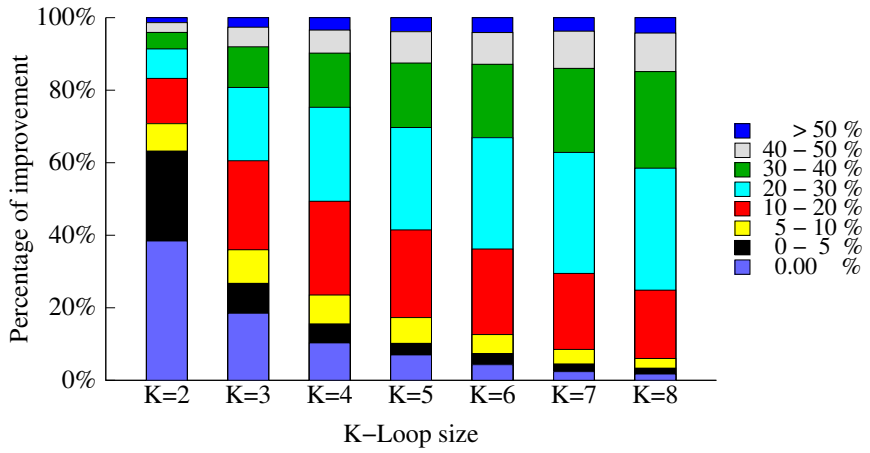
In order to study the potential for performance improvement of distributing K-loops, we make use of the random test generator. The results of applying the distribution algorithm to the randomly generated tests are shown in Subsection 7.3.1. The algorithm is also validated with a K-loop extracted from the MJPEG application in Subsection 7.3.2.

7.3.1 Distribution on Randomly Generated K-loops

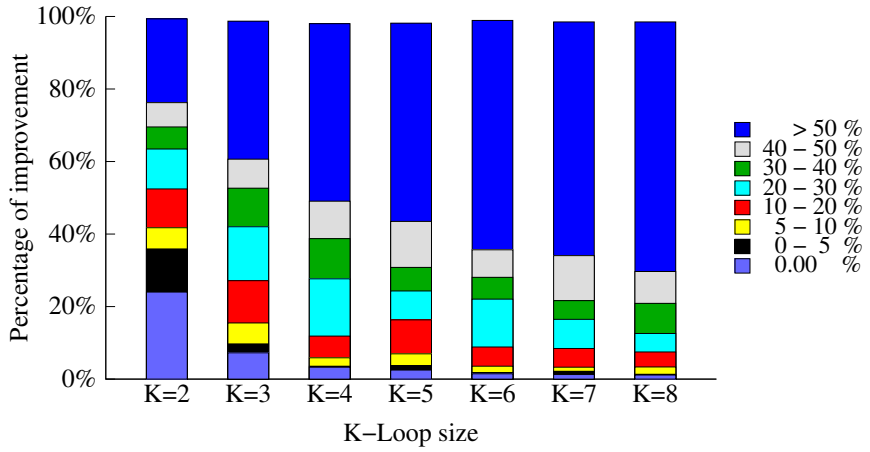
A suite of randomly generated tests has been used in order to study the impact of loop distribution on K-loops with sizes between 2 and 8. For each loop size, 1500 tests have been generated. For each test we compare the performance improvement when applying unrolling and K-pipelining to the original K-loop with the performance improvement when distributing the K-loop using the above algorithm and unrolling and shifting/K-pipelining the resulted K-sub-loops. For these experiments, the reconfiguration latency is not taken into account. Details about the random test generator and the parameters that determine a test case can be found in Section 3.4.

Parallel hardware execution (enabled by loop unrolling) is a great source of performance improvement. The kernels' area requirements influence the maximum degree of parallelism, thus the performance. To study the impact of the area requirements, two sets of tests with K-loop sizes between 2 and 8 were used. For the first set, all kernels have small area requirements, between 1.2% and 12%, which allows for a parallelism degree of 7 or more. For the second test, the maximum area requirement for a kernel is 60%, which can lead to no hardware parallelism. All kernels in a K-loop should fit together on the available area, therefore the sum of all areas should be less than the maximum allowed (in our tests, this maximum is 90%, in order to avoid routing issues).

In Figure 7.5 we illustrate the performance improvement for the different loop



(a) Area requirements between 1.2% and 12%



(b) Area requirements between 1.2% and 60%

Figure 7.5: Loop distribution: performance distribution for the randomly generated tests.

sizes when applying the loop distribution method, compared to the unrolling and K-pipelining method. For Figure 7.5(a) the set of tests was generated with Small Area Requirements (SAR) for each kernel, between 1.2% and 12%. In Figure 7.5(b), the displayed results are for the case of Large Area Requirements (LAR), maximum 60% per kernel.

The distribution algorithm brings no improvement for K-loops of size 2 for 38% of the SAR cases and 24% of the LAR cases, but these numbers decrease with the K-loop size, down to less than 2% for K-loop size 8. Little performance improvement (less than 10%) is shown for 11.2% of the SAR cases and 3.6% of the LAR cases. An improvement of 10–20% is shown for 23% of the SAR cases and 7% of the LAR cases. A performance gain between 20–40% is shown for most of the SAR cases (47.6%) and 19% of the LAR cases. A significant performance improvement of more than 40% is shown for 12% of the SAR cases and most of the LAR cases (67.3%).

The results show that the area requirements have a great impact on performance when loop distribution is taken into account. When the K-loop contains kernels with high area requirements, it is beneficial to split it into smaller K-sub-loops. This way, the smaller kernels can benefit from higher unroll factors.

Also, a large K-loop size is a good reason for splitting the loop. Having small K-sub-loops brings more benefit from loop shifting/K-pipelining, when the software and hardware execute concurrently.

7.3.2 Distribution on the MJPEG K-loop

Details about the Motion JPEG application, as well as profiling and synthesis information, are presented in Section 6.3.2. The MJPEG K-loop consists of three hardware mapped kernels (DCT, Quantizer and VLE) and three software functions (PreShift, Bound and ZigZag).

The MJPEG main loop is illustrated in Algorithm 7.6.

In the remaining text, we will use the notation u for the unroll factor used for the whole K-loop. The u_i notations are used when the K-loop is split into several K-sub-loops and each u_i corresponds to the i-th kernel in the original loop. Then, u_1 is the unroll factor for DCT, u_2 is the unroll factor for Quantizer and u_3 is the unroll factor for VLE. In order to optimize the execution of the MJPEG K-loop, three scenarios are possible:

1. Unroll and parallelize the kernel execution in the main loop.

```
for (i = 0; i < VNumBlocks; i++) do
    for (j = 0; j < HNumBlocks; j++) do
        PreShift(blocks[i][j]);
        DCT(blocks[i][j]);
        Bound(blocks[i][j]);
        Quantizer(blocks[i][j]);
        ZigZag(blocks[i][j]);
        VLE(blocks[i][j]);
    end
end
```

Figure 7.6: The MJPEG main K-loop.

2. Apply K-pipelining such that DCT can execute in parallel with PreShift, and ZigZag can execute in parallel with VLE (concurrent hardware-software execution). Then unroll to parallelize hardware execution.

3. Apply the distribution algorithm to find the optimal loop distribution. Parallelize the resulted K-sub-loops by shifting/K-pipelining and unrolling.

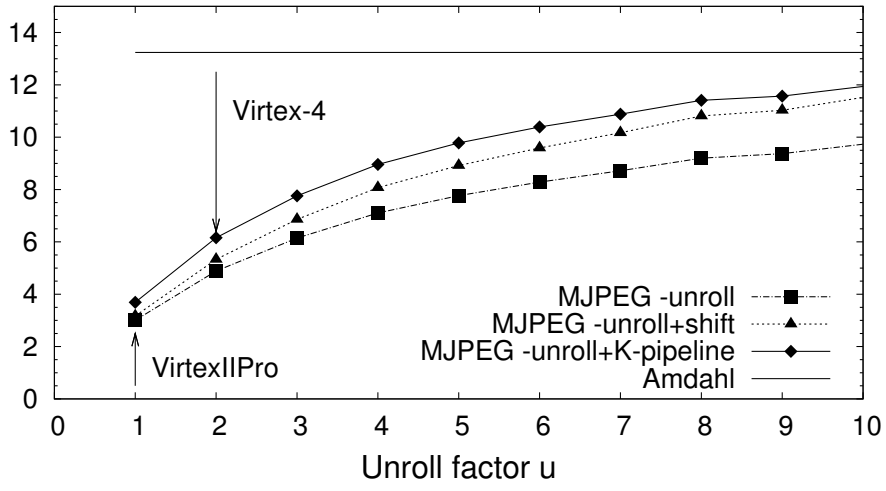


Figure 7.7: Loop distribution: speedup for scenario 1 (loop unrolling) and scenario 2 (loop unrolling + loop shifting/K-pipelining).

Scenarios 1 and 2. Unrolling and K-pipelining. When the MJPEG K-loop is parallelized with loop unrolling or loop unrolling combined with shifting or K-pipelining, the speedup varies with the unroll factor according to Figure 7.7. Note that both these methods constrain that the same parallelization factor applies to all kernels (namely, the unroll factor). All kernel instances must be configured at the same time, no reconfiguration is considered. The final speedup depends on the specific area constraints and/or desired area-speedup trade-off.

No area constraints. The maximum speedup that can be achieved using unrolling and K-pipelining is 12.93, for the unroll factor $u = 28$. A very similar performance with less area usage is possible with $u = 13$, when the speedup is 12.81. Note that the maximum achievable speedup by Amdahl's Law for parallel hardware execution is 13.23. However, this does not take into account the concurrent hardware-software execution, nor the memory constraints.

The VirtexIIPro area constraints require that no unrolling is performed. This leads to a speedup of only 3.56.

The Virtex-4 area constraints require a maximum unroll factor of 2, leading to a speedup of 5.88.

Scenario 3. Loop distribution, followed by unrolling and shifting/K-pipelining. The loop distribution algorithm decides to partition the K-loop into two K-sub-loops. The first K-sub-loop ($KL1$) consists of the PreShift, DCT and Bound functions, while the second K-sub-loop ($KL2$) consists of the Quantizer, ZigZag and VLE functions. The maximum achievable speedup by Amdahl's Law for parallel hardware execution for $KL1$ is 20.91 and for $KL2$ is 8.23. Again, it does not take into account the concurrent hardware-software execution, nor the memory constraints.

The two K-sub-loops have now different optimal parallelization factors. For $KL1$, the optimal factor (assuming no area constraints) is $u_1 = 20$, leading to a speedup of 22.11 for $KL1$.

For $KL2$, the optimal factor is $u_2 = u_3 = 1$, with speedup of 8.23. The reason for this optimal unroll factor follows. Performing K-pipelining allows the software functions to execute in parallel with the hardware kernel. Since the execution time of the software will then be approximately the same as the execution time of the hardware VLE, there is no gain in unrolling in the case of $KL2$.

The K-loop speedup depends now on whether we choose to map all the needed kernel instances on the FPGA (no reconfigurability) or map the K-sub-loops kernels separately (partial reconfigurability before $KL2$).

Loop distribution with no reconfigurability.

No area constraints. The K-loop speedup for the optimal factors $u_1 = 20$, $u_2 = 1$ is 13.53.

The VirtexIIPro area constraints allow no unrolling for either $KL1$ or $KL2$, therefore applying loop distribution would bring no improvement. The speedup is 3.56.

The Virtex-4 area constraints allow a maximum unroll factor $u_1 = 13$ for $KL1$, while $KL2$ does not benefit from unrolling, hence the optimal factor is 1 ($u_2 = u_3 = 1$). The speedup for $KL1$ will be 21.65, while the K-loop speedup after distribution is 13.41. The final unroll factors are (u_1, u_2, u_3) = (13, 1, 1). Compared to the speedup of 5.88 when loop distribution is not used on Virtex4, this is a 228% performance improvement.

Loop distribution with partial reconfigurability. It makes no sense to consider partial reconfigurability with no area constraints, therefore we will analyze only the VirtexIIPro and Virtex-4 cases.

The VirtexIIPro area constraints allow for a maximum of 7 DCT kernels to be configured at a time (we aim at occupying no more than 90% of the area of the XC2VP30 in order to avoid routing issues). The speedup for $KL1$ is 18.93 and ideally, the speedup for the K-loop with unroll factors (7, 1, 1) will be 12.71.

The reconfiguration latency is not negligible in this case. The VLE kernel from $KL2$ will need to be configured after the DCTs have finished. Since the Quantizer takes only 2.98% of the area and we are willing to minimize the reconfiguration time, the Quantizer can be configured before the end of $KL1$, since the DCTs occupy only 86.45% of the total area.

Considering the reconfiguration time of 19.39ms for the whole board, it would take 14.93ms to map VLE onto the FPGA. This means 985400 cycles at the recommended frequency of 66MHz. The maximum VLE reconfiguration latency that can be hidden by the epilogue of $KL1$ and prologue of $KL2$ and that is equal to only 14880 cycles. The MJPEG speedup increases when using loop distribution with partial reconfiguration from 3.56 to 8.22.

The Virtex-4 area constraints impose an unroll factor $u_1 = 18$ for $KL1$. Ignoring the reconfiguration overhead, the speedup for the K-loop with unroll factors (18, 1, 1) is 13.51. Similar to the case of VirtexIIPro, the Quantizer can be configured before the end of $KL1$ and the VLE kernel will need to be configured after the DCTs have finished. The maximum VLE reconfiguration latency that can be hidden by the epilogue of $KL1$ and prologue of $KL2$ and that is equal to 28551 cycles. It differs from the one computed for VirtexIIPro because the $KL1$

Table 7.1: Performance results for different area constraints and different methods ([1]-loop unrolling; [2]-K-pipelining; [3]-loop distribution).

Area constraints	(u_1,u_2,u_3)	methods	reconfig.	S_{K-loop}
∞ (max)	(28,28,28)	[1]+[2]	no	12.93
∞ (trade-off)	(13,13,13)	[1]+[2]	no	12.81
∞ (max)	(20, 1, 1)	[3]+[1]+[2]	no	13.53
∞ (trade-off)	(8, 1, 1)	[3]+[1]+[2]	no	13.24
VirtexIIPro	(1, 1, 1)	[1]+[2]	no	3.56
VirtexIIPro	(1, 1, 1)	[3]+[1]+[2]	no	3.56
VirtexIIPro	(7, 1, 1)	[3]+[1]+[2]	yes	8.23 / 12.34[1]
Virtex-4	(2, 2, 2)	[1]+[2]	no	5.88
Virtex-4	(13, 1, 1)	[3]+[1]+[2]	no	13.41
Virtex-4	(18, 1, 1)	[3]+[1]+[2]	yes	11.95

epilogue is larger for $u_1 = 13$ (Virtex-4) than for $u_1 = 7$ (VirtexIIPro).

To load the VLE on Virtex-4, a total of 883.9KB need to be transferred on the FPGA. The reconfiguration time of the VLE kernel computed with (7.1) is 2.21ms at 100MHz, that is the equivalent of 221000 CPU cycles. The K-loop speedup with partial reconfiguration is 11.95, less than the speedup achieved with static mapping of the kernels on the Virtex-4.

The speedup when using loop distribution followed by unrolling and shifting/K-pipelining (scenario 3) is illustrated in Figure 7.8. The three area requirements cases are shown: no area constraints, VirtexIIPro constraints, Virtex-4 constraints, while considering both cases of reconfigurability/no reconfigurability. Only the global optimum is shown for the case of no area constraints. Only the best value (with the least reconfiguration time) is shown for VirtexIIPro. Table 7.1 summarizes the results of mapping the MJPEG K-loop onto reconfigurable hardware, using the presented techniques and various area constraints. Note that ∞ means no area constraints. In this case, two types of results are shown. The (max) type means the global optimum in terms of performance. The (trade-off) type means a lower unroll factor that gives a performance 2-3%

[1]The two values for the speedup on VirtexIIPro when using partial reconfiguration correspond to the different times for reconfiguring the chip using first the traditional method (with the Xilinx EAPR) and second using Combitgen for an optimized bitstream([28]).

less than that of the global optimum.

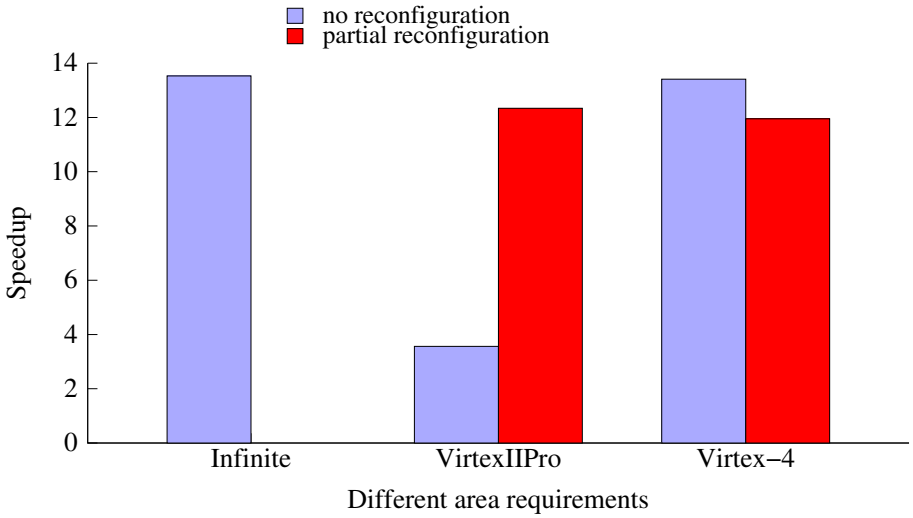


Figure 7.8: Loop distribution: speedup for scenario 3 (loop distribution followed by unrolling and shifting/K-pipelining).

The results on VirtexIIPro show that loop distribution with partial reconfiguration gives a significant performance improvement when the area is very restrictive (speedup increase from 3.56 to 8.22). Partial reconfiguration allows for a higher unroll factor in the first K-sub-loop and thus more parallelism. If the reconfiguration latency could be decreased, the performance would improve with maximum 54% (up to 12.71 speedup on VirtexIIPro).

The results of mapping MJPEG on the Virtex-4 show that loop distribution with no reconfigurability gives the best performance when the area is not a big constraint. The performance increased by 228%(from 5.88 to 13.41), which represents 99% of the maximum speedup (13.53) that could be achieved with infinite resources.

7.3.3 Interpretation of the Results

Some observations regarding various possible situations influencing the results follow.

- The relation between the kernel hardware time and the pre-/post- software function time is essential in determining the usefulness of loop unrolling.

If the hardware kernel time and the software function time are comparable, unrolling with a factor larger than 2 is not beneficial and in some cases shifting or K-pipelining alone will suffice to get to the maximum speedup.

- High area requirements for certain kernels are a good reason for splitting the loop. This way, the other kernels can benefit from higher unroll factors.
- A large K-loop size may also be a good reason for splitting the loop. This way, different kernels can benefit from different unroll factors.
- On small chips such as the VirtexIIPro, the reconfiguration latency is a big overhead. However, even with the cost of the partial reconfiguration, a large performance improvement can be achieved, because more unrolling is permitted.
- On the Virtex-4 chips the reconfiguration latency has a smaller impact. However, the performance without reconfiguration can be superior to the performance after reconfiguration when the area is not such a big constraint.

7.4 Conclusions

In this chapter, we discussed the use of loop distribution in the context of K-loops. When the kernels from a large K-loop are distributed into smaller K-sub-loops, they can benefit from different unroll factors and therefore more from the parallelism enabled by unrolling and shifting/K-pipelining.

The results of experimental data that was randomly generated show that the K-loop size and the area constraints have a high impact on the performance. Splitting is very beneficial for large K-loops and for K-loops that contain kernels with high area requirements.

In more than 60% of the random test cases, distributing the K-loop lead to a significant performance improvement (more than 40%) compared to when loop unrolling and K-pipelining are applied to the K-loop without splitting it. Note that the partial reconfiguration overhead was not taken into account for the randomly generated tests because they were not designed for any platform in particular. Since Virtex-4 family chips have a small reconfiguration delay, we consider that for our tests this delay can be hidden by the K-sub-loops prologue and epilogue (resulted from loop unrolling and loop shifting/K-pipelining).

The distribution algorithm proposed in this paper has been validated with the MJPEG K-loop. Our method proves to be efficient in both configurations: together with partial reconfiguration on small chips (the VirtexIIPro), as well as without reconfiguration on large chips (the Virtex-4).

The performance of the MJPEG K-loop on VirtexIIPro improved from 3.56 speedup without loop distribution to 8.22 speedup with loop distribution and partial reconfiguration in between the K-sub-loops.

On the Virtex-4, the best performance is achieved with loop distribution and static mapping of the kernels on the reconfigurable hardware (speedup of 13.41). This is a performance increase of more than 100% compared to the speedup of 5.88 when the MJPEG K-loop is parallelized only by unrolling and K-pipelining.

The next chapter of this dissertation is dedicated to loop skewing. The mathematical model of using loop skewing in the context of K-loops will be presented, followed by experimental results on the Deblocking Filter K-loop of the H.264 encoder/decoder.

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

O.S. Dragomir, K. Bertels, **Loop Distribution for K-Loops on Reconfigurable Architectures**, DATE 2011

8

Loop Skewing for K-loops

IN THIS CHAPTER, we discuss the loop skewing transformation in the context of reconfigurable computing. In Section 8.1.1, we present the motivation for using the loop skewing for K-loops. In Section 8.2, two techniques for skewing simple K-loops are presented. Section 8.3 shows experimental results for the Deblocking Filter kernel, while conclusions are drawn in Section 8.4.

8.1 Introduction

Loop skewing is a widely used loop transformation for wavefront-like computations [11, 143]. A wavefront-like computation means a nested loop iterating over a multidimensional array, where each iteration of the inner loop depends on previous iterations. Loop skewing rearranges the array accesses such that the only dependencies are between iterations of the outer loop. Consider the example in Figure 8.1. In order to compute the element `A(i, j)` in each iteration of the inner loop, the previous iteration's results `A(i − 1, j)` must be available already. Therefore, this code cannot be parallelized or pipelined as it is currently written. Performing the affine transformation `(p, t) = (i, 2 ∗ j + i)` on the loop bounds and rewriting the code to loop on `p` and `t` instead of `i` and `j`, we obtain the "skewed" loop in Figure 8.2. The iteration space and dependencies for the original and skewed loops are showed in Figure 8.3 a) and b), respectively.

8.1.1 Motivational Example

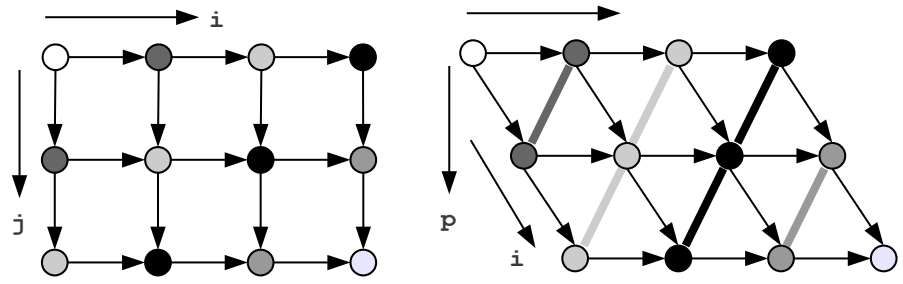
Throughout this chapter, we will use the motivational example in Figure 8.4. It consists of a K-loop with two functions: `SW` — which is executed always on

```
for (j = 0; j < N; j++) do
    for (i = 0; i < M; i++) do
        A(i,j) = F(A(i-1,j), A(i,j-1));
    end
end
```

Figure 8.1: Loop with wavefront-like dependencies.

```
for (t = 1; t < M + 2 * N; t++) do
    pmin = max(t%2, t-2*N+2);
    pmax = min(t, M-1);
    for (p = pmin; p <= pmax; p+= 2) do
        i = p;
        j = (t-p)/2;
        A(i,j) = F(A(i-1,j), A(i,j-1));
    end
end
```

Figure 8.2: Skewed (dependency-free) loop.



(a) Loop dependencies before skewing (b) Loop dependencies after skewing

Figure 8.3: Loop dependencies (different shades of gray show the elements that can be executed in parallel).

the GPP — and K, which is the application's kernel and will be executed on the reconfigurable hardware. Implicitly, the execution time for SW is smaller than the execution time of K on the GPP.

```
for (j = 0; j < N; j++) do
    for (i = 0; i < M; i++) do
        /* a pure software function                    */
        SW(blocks, i, j);
        /* a hardware mapped kernel                    */
        K(blocks, i, j, i − 1, j − 1);
    end
end
```

Figure 8.4: K-loop with one hardware mapped kernel and inter-iteration data dependencies.

We assume that in each iteration (i, j) data pointed by $(blocks, i, j)$ are updated based on data previously computed in iterations $(i-1, *)$ and $(*, j-1)$. Thus there are data dependencies between instances of K in different iterations, similar to the dependencies shown in Figure 8.3(a).

8.2 Mathematical Model

We use the notations presented in Section 3.3.1. The case $u > u_{\max}$ is not considered because there is no speedup increase for the hardware kernels. In addition, we use the following notations:

- $T_{it}(t)$ — the hardware execution time at iteration level in iteration t, assuming the unroll factor u;
- N_{it} — the number of iterations of the inner loop after skewing.

The K-loop in our motivational example has the same iteration domain as the loop in Figure 8.1, therefore after skewing the iteration domain will be the same as in Figure 8.2. The outer loop will have $N + M - 1$ iterations. For simplicity, we assume that $M \leq N$. The number of iterations of the inner loop (N_{it}) varies according to 8.5. The maximum number of iterations of the inner loop is $\min(M, N) = M$ and there are $(N - M + 1)$ such iterations. This means that M is the maximum available parallelism. A mathematical formulation for

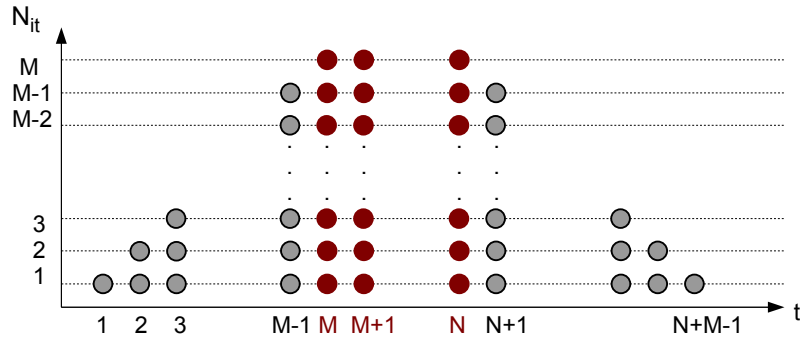


Figure 8.5: Loop skewing: the number of iterations of the inner skewed loop.

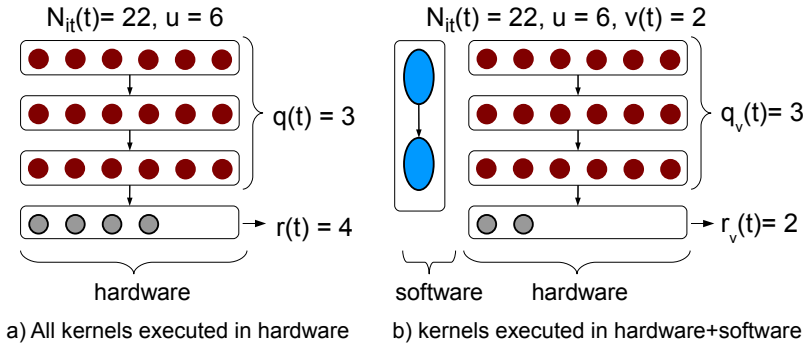


Figure 8.6: Loop skewing: kernels' execution pattern in an iteration.

N_{it} is:

$$N_{\text{it}}(t) = \begin{cases} M, & M \leq t \leq N \\ t, & 1 \leq t < M \text{ or } N < t < M+N \end{cases} \tag{8.1}$$

Next, we compute the hardware execution time for the skewed K-loop assuming the unroll factor u as the sum of the execution times in each outer iteration t. First, we consider the method with all kernel instances running in hardware and second, the method with part of the kernels running in software.

With all kernels in hardware The hardware execution time for the skewed K-loop with all kernel instances running in hardware when unrolling with factor u is:

$$T_{\text{h/h}}(u) = \sum_{t=1}^{M+N-1} T_{\text{it}}(t) \tag{8.2}$$

$T_{\text{it}}(t)$ represents the execution time of the $N_{\text{it}}(t)$ kernels in the iteration t. Considering $N_{\text{it}}(t)$ from (8.1), $T_{\text{h/h}}(u)$ becomes:

$$T_{\text{h/h}}(u) = 2 \cdot \sum_{t=1}^{M-1} T_{\text{it}}(t) + (N - M + 1) \cdot T_{\text{it}}(M) \tag{8.3}$$

After skewing, the inner loop can be parallelized to improve the performance. Depending on the relation between $N_{\text{it}}(t)$, u_m and u, the $N_{\text{it}}(t)$ kernels in outer iteration t should execute in groups of u concurrent kernels and not all kernels in parallel. The groups execute sequentially, as depicted in Figure 8.6a). The number of groups with u kernels is the quotient of the division of $N_{\text{it}}(t)$ by u, denoted by $q(t)$. One more group consisting of $r(t)$ kernels will be executed, where $r(t)$ is the remainder of the division of $N_{\text{it}}(t)$ by u. Then, the time to execute the $N_{\text{it}}(t)$ kernels in hardware is:

$$T_{\text{it}}(t) = q(t) \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(r(t)), \tag{8.4}$$

Assuming the unroll factor u, let q and r be the quotient and the remainder of the division of M by u, respectively. Then $M = q \cdot u + r,\ r < u$. The hardware execution time becomes:

$$\begin{array}{rcl} T_{\text{h/h}}(u) & = & 2 \cdot [\, T_{\text{it}}(1) \quad + \ldots + \quad T_{\text{it}}(u) \,] \\ & + & 2 \cdot [\, T_{\text{it}}(u+1) \quad + \ldots + \quad T_{\text{it}}(u+u) \,] \\ + \ldots & + & 2 \cdot [\, T_{\text{it}}(q \cdot u + 1) \quad + \ldots + \quad T_{\text{it}}(q \cdot u + r) \,] \\ & & + (N - M - 1) \cdot T_{\text{it}}(q \cdot u + r) \end{array} \tag{8.5}$$

By expanding each $T_{\text{it}}(t)$ from (8.5) according to (8.4), we obtain:

$$\begin{array}{rcl} T_{\text{h/h}}(u) & = & 2 \cdot [\, T_{K(\text{hw})}(1) + \ldots + T_{K(\text{hw})}(u) \,] \\ & + & 2 \cdot [\, T_{K(\text{hw})}(u) + \ldots + 2 \cdot T_{K(\text{hw})}(u) \,] \\ + \ldots & + & 2 \cdot [\, q \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(1) + \ldots + \\ & & + q \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(r) \,] \\ & & + (N - M - 1) \cdot [\, q \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(r) \,] \end{array} \tag{8.6}$$

Then, by grouping all the similar terms, $T_{\text{h/h}}(u)$ becomes:

$$\begin{aligned} T_{\text{h/h}}(u) = {} & 2 \cdot q \cdot \sum_{i=1}^{u-1} T_{K(\text{hw})}(i) + 2 \cdot \sum_{i=1}^{r-1} T_{K(\text{hw})}(i) \\ & + (N - M + 1) \cdot T_{K(\text{hw})}(r) \\ & + q \cdot (N - u + 1 + r) \cdot T_{K(\text{hw})}(u) \end{aligned} \tag{8.7}$$

The speedup at K-loop level will be:

$$S_{\text{h/h}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{h/h}}(u) + T_{\text{sw}} \cdot M \cdot N}. \tag{8.8}$$

With part of the kernels in software It is possible that better performance is achieved if not all kernel instances from an iteration are executed in hardware. Since the degree of parallelism varies with the iteration number according to 8.5, balancing the number of kernels that run in software and in hardware for each iteration would lead to a significant overhead. Assuming a certain unroll factor u ($u < M$), we look only at the iterations t with $N_{\text{it}}(t) > u$ and compute the number of kernels (denoted by $v(t)$) that need to run in software, such that the iteration execution time is minimized. There are $N + M - 2 \cdot u - 1$ such iterations, namely the iterations t with $u < t \leq N + M - u$.

Then, $v(t)$ is the number of kernels for which the kernels running in software execute in approximately the same amount of time as the kernels running in hardware. The time to execute $v(t)$ kernels in software is:

$$T_{K(\text{sw})}(v(t)) = v(t) \cdot T_{K(\text{sw})} \tag{8.9}$$

The execution pattern for $N_{\text{it}}(t) = 22$, with $v(t) = 2$ kernel instances running in software is depicted in 8.6b). In this context, $q_v(t)$ and $r_v(t)$ are the quotient and the reminder of $N_{\text{it}}(t) - v(t)$ divided by u, respectively.

The time to execute the remaining $N_{\text{it}}(t) - v(t)$ kernels in hardware is:

$$T_{K(\text{hw})}(t, v(t)) = q_v(t) \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(r_v(t)) \tag{8.10}$$

By using (3.8) in (8.10), $T_{K(\text{hw})}(t, v(t))$ becomes:

$$\begin{aligned} T_{K(\text{hw})}(t, v(t)) = {} & (N_{\text{it}}(t) - v(t)) \cdot T_{\max(\text{r,w})} + \\ & + \left\lceil \frac{N_{\text{it}}(t) - v(t)}{u} \right\rceil \cdot (T_{\text{c}} + T_{\min(\text{r,w})}) \end{aligned} \tag{8.11}$$

Then, the optimum number of kernels that should run in software in the iterations t with $N_{\text{it}}(t) > u$ is the closest integer to the value of $v(t)$ that satisfies the relation:

$$T_{K(\text{sw})}(v(t)) = T_{K(\text{hw})}(t, v(t)) \iff \tag{8.12}$$

$$\begin{aligned} v(t) \cdot (T_{K(\text{sw})} + T_{\max(\text{r,w})}) = {} & N_{\text{it}}(t) \cdot T_{\max(\text{r,w})} + \\ & + \left\lceil \frac{N_{\text{it}}(t) - v(t)}{u} \right\rceil \cdot (T_{\text{c}} + T_{\min(\text{r,w})}) \end{aligned} \tag{8.13}$$

The value of $v(t)$ depends on the number of kernels in iteration t ($N_{\text{it}}(t)$), which represents the available parallelism, and on the weight of the memory transfers ($T_{\max(\text{r,w})}$). The larger the value of $N_{\text{it}}(t)$ and the weight of the

memory transfers compared to the kernel's execution time (in software and hardware), the larger the value of $v(t)$. The unroll factor u also influences the number of kernels to execute in software, as a larger value of u implies that more kernels will execute in hardware and fewer in software. These considerations are illustrated in Section 8.3, where we present the experimental results.

By rounding the value of $v(t)$ to the nearest lower integer, the execution time for the kernels in software in iteration t will not be larger than the execution time for the kernels in hardware. Therefore, the overall time for executing all the kernel instances in iteration t is given by the hardware time:

$$T_{\text{it}}(t, v(t)) = T_{K(\text{hw})}(t, v(t)) \qquad \Rightarrow \tag{8.14}$$

$$T_{\text{it}}(t, v(t)) = q_v(t) \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(r_v(t)) \tag{8.15}$$

The hardware execution time for unroll factor u when part of the kernels execute in software is:

$$T_{\text{h/s}}(u) = 2 \cdot \sum_{t=1}^{u} T_{\text{it}}(t) + \sum_{t=u+1}^{M+N-u} T_{\text{it}}(t, v(t)) \tag{8.16}$$

$$T_{\text{h/s}}(u) = 2 \cdot \sum_{t=1}^{u} T_{\text{it}}(t) + 2 \cdot \sum_{t=u+1}^{M-1} T_{\text{it}}(t, v(t)) + \sum_{t=M}^{N} T_{\text{it}}(M, v(M)) \tag{8.17}$$

Then, by expanding each T_{it} and grouping all the similar terms, $T_{\text{h/s}}(u)$ becomes:

$$\begin{aligned} T_{\text{h/s}}(u) \;=\; & 2 \cdot \sum_{i=1}^{u-1} T_{K(\text{hw})}(i) + \\ & + \left(2 + 2 \cdot \sum_{i=u+1}^{M-1} q_v(i) + (N - M + 1) \cdot q_v(M) \right) \cdot T_{K(\text{hw})}(u) + \\ & + \left(2 \cdot \sum_{i=u+1}^{M-1} T_{K(\text{hw})}(r_v(i)) \right) + (N - M + 1) \cdot T_{K(\text{hw})}(r_v(M)) \end{aligned} \tag{8.18}$$

Similarly to (8.8), the speedup at K-loop level when balancing the kernel execution in software and hardware will be:

$$S_{\text{h/s}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{h/s}}(u) + T_{\text{sw}} \cdot M \cdot N}\,. \tag{8.19}$$

Note that (8.8) and (8.19) are valid under the assumption that the software code (SW) and the hardware-mapped kernel execute sequentially. In Chapter 5 it

has been proven that it is always beneficial to apply loop shifting for breaking intra-iteration dependencies between the SW function and the kernel, in order to enable the software-hardware parallelism. The same technique can be applied also in conjunction with the loop skewing and thus the final execution time and the speedup could be improved by concurrently executing the software and the hardware code.

8.3 Experimental Results

In this section, we illustrate the methods presented in Section 8.2.

```
foreach frame f do
    for (i = 0; i < nCols; i++) do
        for (j = 0; j < nRows; j++) do
            compute_MB_params(&MB);
            filter_MB(MB);
        end
    end
end
```

Figure 8.7: The Deblocking Filter K-loop.

The analysis was performed on one of the most time consuming parts of the H.264 video codec, Deblocking Filter (DF) [84], which is a part of both encoder and decoder. The code presented in Figure 8.7 represents the main K-loop of the DF.

In the H.264 standard, the image is divided in blocks of 4×4 pixels. The color format is YCbCr 4:2:0, meaning that the chrominance (chroma) components are sub-sampled to half the sample rate of the luminance (luma) - in both directions. The blocks are grouped in MacroBlocks (MB), each MB being a 4×4 block matrix for the luma and 2×2 block matrix for the chroma components. Each block edge has to be filtered, the DF being applied to each decoded block of a given MB for luma and chroma samples.

The pseudocode for the DF is shown in Figure 8.8. For the processed MB, first vertical filtering is applied (luma and chroma), followed by horizontal filtering. The vertical and horizontal filtering cannot be interchanged or executed in parallel because of data dependencies, however the luma and chroma filtering can be performed in parallel.

```
foreach Edge in MB.edges[VERT] do
    if (is_picture_edge(Edge) == FALSE) then
        filter_MB_edge(MB, VERT);
        if (odd == TRUE) then
            odd = FALSE;
            filter_MB_edge_cv(MB, VERT, Cr);
            filter_MB_edge_cv(MB, VERT, Cb);
        else
            odd = TRUE;
        end
    end
end
foreach Edge in MB.edges[HORIZ] do
    if (is_picture_edge(Edge) == FALSE) then
        filter_MB_edge(MB, HORIZ);
        if (odd == TRUE) then
            odd = FALSE;
            filter_MB_edge_ch(MB, HORIZ, Cr);
            filter_MB_edge_ch(MB, HORIZ, Cb);
        else
            odd = TRUE;
        end
    end
end
```

Figure 8.8: Deblocking Filter applied at MB level.

Table 8.1: Synthesis results for the Deblocking Filter kernel.

Platform	Slices	[%]	Freq[MHz]
XC2VP30	2561	18.70	178.296
XC2VP70	2552	7.72	177.953
XC2VP100	2606	5.91	151.400

Table 8.2: Execution times for the Deblocking Filter.

	Software	Hardware
average	87119	106802
maximum	103905	119166

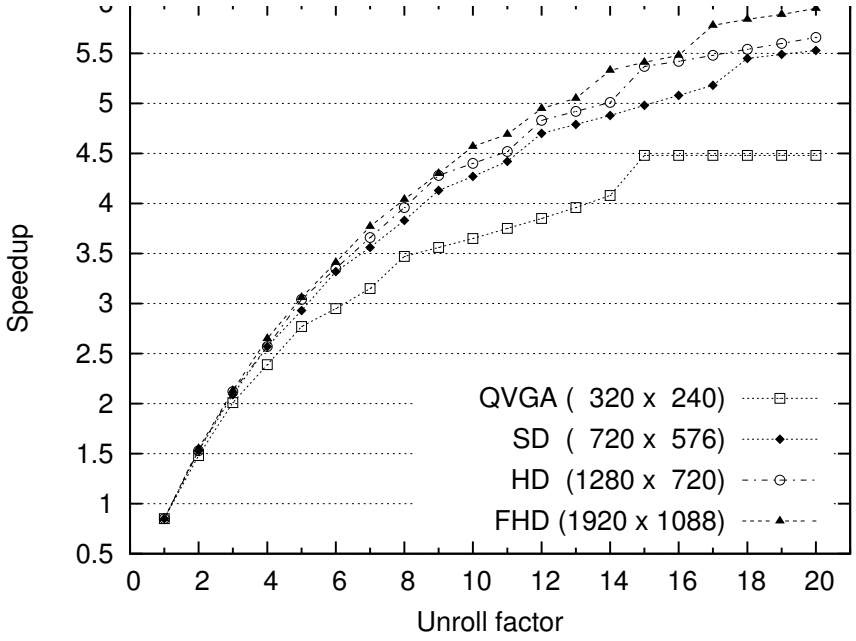
The VHDL code for the filter was automatically generated with the DWARV [149] tool. The synthesis results using Xilinx XST tool of ISE 8.1 for different Xilinx boards are presented in Table 8.1. The code was executed on the Virtex II Pro–XC2VP30 board. The execution times were measured using the PowerPC timer registers. For the considered implementation, the shared memory has an access time of 3 cycles for reading and storing the value into a register and 1 cycle for writing a value to memory;

In the profiling stage, the execution times for the DF running in both software and hardware for 80 different MBs were measured. The average and the maximum values that were measured are presented in Table 8.2. Also, the profiling indicates the (maximum) number of memory transfers per kernel – 2424 memory reads and 2400 memory writes. The software function that computes the parameters executes in 2002 cycles.

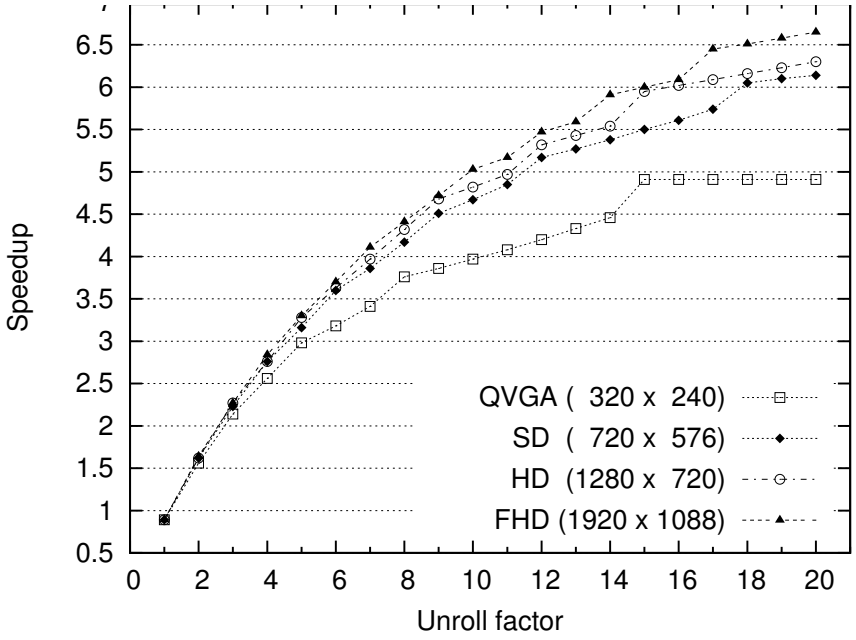
Figures 8.9(a) and 8.9(b) illustrate the results when the first of the presented methods is considered, with all kernel instances running on the FPGA. The following picture formats have been considered: Quarter Video Graphics Array (QVGA) – 320 $\times$ 240 pixels; Standard Definition (SD) – 720 $\times$ 576 pixels; High Definition (HD) – 1280 $\times$ 720 pixels; and Full High Definition (FHD) – 1920 $\times$ 1088 pixels.

Equations (3.4), (8.7) and (8.8) were used for computing the different speedups for the skewed and unrolled DF K-loop for various image sizes and unroll factors. The speedup considering the average execution times for the kernel are illustrated in Figure 8.9(a), while the speedups for the maximum execution times are presented in Figure 8.9(b).

The results are heavily influenced by the performance of the kernel implementation in hardware. As shown in Table 8.2, the hardware implementation is slower than the software one, which results in a performance decrease (0.85 and 0.89 speedup) when no unrolling is performed. However, unrolling even with



(a) Deblocking Filter speedup for different picture sizes (average MB execution time).



(b) Deblocking Filter speedup for different picture sizes (maximum MB execution time).

Figure 8.9: Loop skewing: Deblocking Filter speedup for different picture sizes.

a factor of 2 already compensates the slow hardware implementation, giving a speedup of 1.5 – 1.6. The performance does not increase linearly with the unroll factor because of the variable number of iterations of the skewed K-loop, being influenced also by the relation between the unroll factor and M (M is the maximum degree of parallelism for most of the iterations of the skewed K-loop).

The results show that better performance is obtained when parallelizing K-loops with a large number of iterations. The reason is that for a large total number of iterations, there will be a large number of iterations that have a higher available degree of parallelism (M). Note that for the QVGA picture format (320×240 pixels), the speedup saturates at the unroll factor 15. This happens because the DF K-loop number of iterations is equal to the picture size divided by 16, meaning that in this case $N = 20$ and $M = 15$.

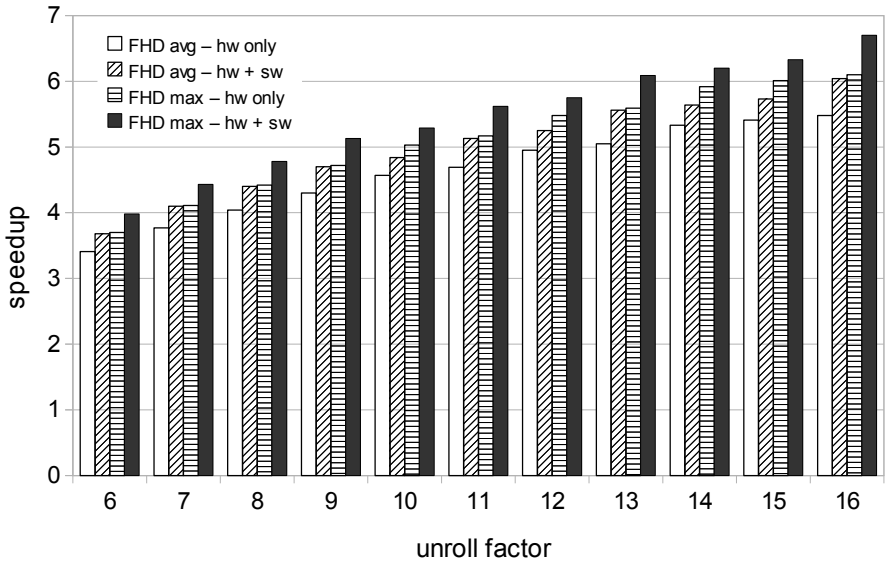


Figure 8.10: Loop skewing: Deblocking Filter speedup for FHD format for different unroll factors.

Figure 8.10 presents the speedup for the FHD picture format for unroll factors between 6 and 16, considering both types of execution presented in Section 8.2. The first type of execution consists of running all kernel instances in hardware, while the second places part of the kernel instances in software. A performance increase between 4% and 15% is noted for the second type. The performance would increase more for a larger number of iterations or if the memory I/O

($T_{\max(r,w)}$) would represent more of the execution time.

Figure 8.11 shows how the number of kernel instances that execute in software varies with the unroll factor for the FHD picture format. The depicted numbers represent the sum of software executed kernels across all loop iterations, approx. 2–2.45% of the total number of kernels for the FHD format. The general trend is that a larger unroll factor means fewer kernels to execute in software, because more kernels will execute in hardware in each group. See Figure 8.6 for the kernels' execution pattern, where one line of kernels represents a group. On the same figure it can be seen that the total number of kernel instances that execute in software is larger for FHD-avg than for FHD-max. This happens because the memory I/O represents a larger percentage from the total execution time for FHD-avg, compared to FHD-max. The reason is that the weight of the I/O time from the total execution time directly influences the number of software scheduled kernels, as seen from equation (8.13).

On the basis of one iteration which has a variable number of kernels over time (see Figure 8.5), Figure 8.12 presents how the number of kernel instances that execute in software varies with the total number of kernels in that iteration. When the total number of kernels in one iteration increases, the total execution time increases and therefore more kernel instances can be scheduled for software execution.

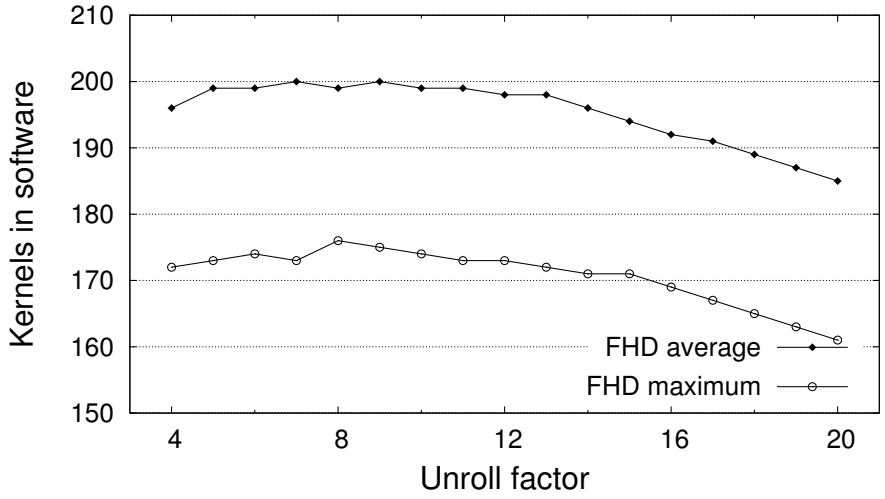


Figure 8.11: Loop skewing: how the number of kernels in software varies with the unroll factor.

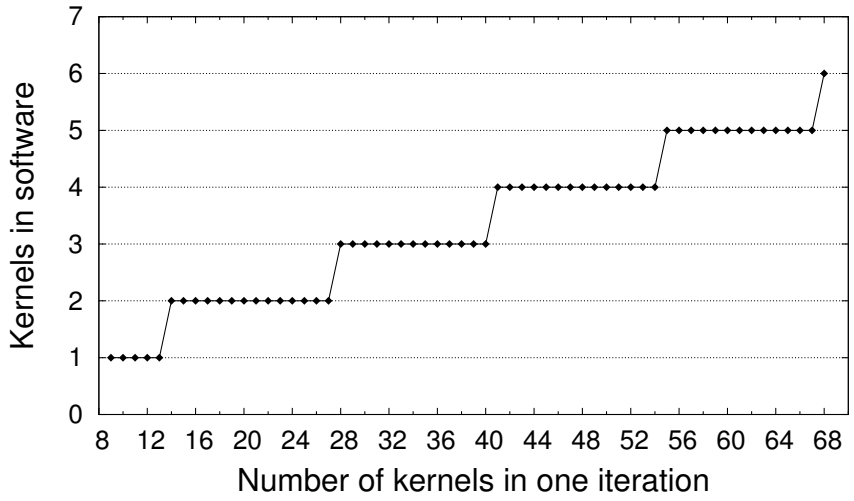


Figure 8.12: Loop skewing: the software scheduled kernels vs. the total number of kernels.

Table 8.3: Loop skewing: Deblocking Filter speedups for the unroll factor u=8.

	QVGA	SD	HD	FHD
average (hw only)	3.47	3.83	3.96	4.04
average (hw + sw)	3.65	4.11	4.20	4.40
maximum (hw only)	3.77	4.18	4.32	4.42
maximum (hw + sw)T	3.93	4.47	4.55	4.78

Taking into account also area requirements, the optimum unroll factor depends on the desired trade-off between area consumption and performance. If for instance the metric chosen for the optimum unroll factor is to have a speedup increase of at least 0.3 comparing to the previous unroll factor, then the optimum unroll factor for most picture sizes in our example is 8. The estimated speedups achieved for the unroll factor `u = 8` for all pictures sizes and using the average/maximum kernel execution times are depicted in Table 8.3.

8.4 Conclusions

In this chapter, we addressed parallelizing K-loops with wavefront-like dependencies, thus relaxing the constraints imposed by using only loop unrolling and/or shifting/K-pipelining for loop parallelization. Two methods have been presented, with different scheduling scenarios for the kernel instances. For the first method, all instances of a kernel K are scheduled to run on the reconfigurable hardware. For the second method, part of the kernel instances are scheduled to run in hardware and part in software. Both methods are based on loop skewing and unrolling for computing the optimal number of kernel instances that will run in parallel on the reconfigurable hardware. The second method is more suitable for a large number of iterations and/or when the I/O represents a significant part of the kernel, limiting the degree of parallelism.

In the experimental part, we analyzed the case of the Deblocking Filter K-loop of the H.264 encoder/decoder. Using an automatically generated VHDL code for the edge filtering part of the DF, we showed that a speedup between 3.4 and 4.8 can be achieved with the unroll factor 8, depending on the input picture size. Better performance is achieved when the K-loop bounds are larger, because a higher number of iterations allows for a higher degree of parallelism.

In the next chapter, we present a summary of this dissertation and the contributions it brings. They are followed by conclusions and open issues.

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

O.S. Dragomir, K. Bertels, **K-Loops: Loop Skewing for Reconfigurable Architectures**, FPT 2009

9 Conclusions

IN THIS THESIS, we addressed loop optimizations that are suitable for K-loops in the context of reconfigurable architectures under the Molen Programming Paradigm. A K-loop is a loop that contains one or more hardware mapped kernels (which can execute on the reconfigurable hardware), together with pure software code (which executes only on the GPP). The work in this thesis targeted coarse grain loop level parallelism, hardware-software loop level parallelism and partial reconfigurability while only a part of the loop (nest) is executed on the reconfigurable hardware. The purpose of the presented methods is to improve the performance of applications, by maximizing the parallelism inside K-loops. This chapter is organized as follows. Section 9.1 presents a summary of the dissertation. The major contributions of this thesis are presented in Section 9.2, while the main conclusions are drawn in Section 9.3. The open issues and the future directions of research are presented in Section 9.4.

9.1 Summary

In this dissertation, we investigated loop optimizations that address performance improvement for K-loops in the context of reconfigurable architectures. Several loop transformations have been proposed for optimizing K-loops. These loop transformations have different purposes: to parallelize the K-loop, to eliminate specific data dependencies in order to allow parallelization, to split the K-loop into smaller K-sub-loops that can be individually optimized. The performance gain comes from the parallel execution of multiple kernel instances on the reconfigurable hardware, but also from parallel execution of the hardware and software parts of the K-loop, when it is possible. The methodology used is similar for all the proposed loop transformations, and takes into account area,

memory and performance considerations. The proposed algorithms are based on profiling information about the application and about the K-loop.

Chapter 1 introduced the research context and the notion of K-loop. It continued with an overview of the problem that has been addressed in this dissertation, and with the dissertation outline.

In Chapter 2, an overview of the most well-known loop transformations and projects addressing reconfigurable architectures has been presented. After looking at related projects, we could identify the open issues, which have been addressed later on in this thesis.

In Chapter 3, we detailed the notion of K-loop and introduced the framework used in our work. The methodology, comprising of general notations used throughout the thesis, area, memory and performance considerations has been presented here. Next, the random test generator that we used for validating some of our transformations has also been presented. The last part of the chapter has focused on the description of an algorithm that decides which loop transformations to use, depending on the K-loop's characteristics.

Chapter 4 discussed the use of the loop unrolling transformation. In the context of K-loops, loop unrolling is used for exposing hardware parallelism, enabling parallel execution of identical kernel instances on the reconfigurable hardware. Experiments have been performed for K-loops containing well-known kernels from real-life applications: DCT, SAD, Quantizer and Convolution.

The achieved results have been compared with the theoretical maximum speedup computed with Amdahl's Law assuming maximum parallelism (full unroll) for the hardware. The results showed that the software part of the K-loop has a large influence on the optimal degree of parallelism. If the software is faster, different results are obtained for different kernel implementations, depending on how much optimized they are.

Chapter 5 discussed the loop shifting transformation, and Chapter 6 presented the extension of loop shifting, called K-pipelining. The main difference between these two transformations is that loop shifting applies to single kernel K-loop, while K-pipelining is suitable for K-loops with multiple hardware mapped kernels. These two transformations are used for exposing hardware-software parallelism, by eliminating certain inter-iteration loop dependencies that prevent the software functions and the hardware kernels to execute concurrently. Loop shifting and K-pipelining can also be combined with loop unrolling for further exploitation of the available parallelism, as loop unrolling enables for multiple kernel instances to execute concurrently on the reconfigurable hardware.

Experiments for loop shifting have been performed for the same K-loops as in the case of loop unrolling, with the DCT, SAD, Quantizer and Convolution kernels. The mathematical model and the results have proven that it is always beneficial to apply loop shifting to a K-loop (if possible), while loop unrolling may not bring any additional performance improvement if loop shifting is used. This is the case of K-loops where the software part is much larger than the execution time of the hardware kernel.

The experimental results for K-pipelining are of two types: theoretical, using randomly generated tests, and practical, using the MJPEG K-loop, which contains three kernels (DCT, Quantizer and VLE). The results for the randomly generated test have shown a good potential of performance improvement when K-pipelining is used in conjunction with loop unrolling for large K-loops, but the speedup depends highly on the kernels' area requirement as they limit the degree of parallelism. The results for the MJPEG K-loop have shown that the performance improvement depends much on the ratio between the execution time of the hardware kernels and the execution time of the software functions, because it influences the efficiency of the hardware-software parallelism.

So far, the strategy for speeding up large K-loops consisted of using the K-pipelining transformation and the loop unrolling. In Chapter 7 we have proposed a more efficient method for speeding up large K-loops, based on loop distribution. An algorithm for deciding the optimal splitting points for a K-loop has been proposed. The resulted K-sub-loops are accelerated by using the previously presented methods, e.g. loop unrolling and/or loop shifting/K-pipelining. One of the advantage of using loop distribution is that, after splitting, different kernels can benefit from different unroll factors. After the distribution is performed, each K-sub-loop can also have the whole area available due to the possibility of reconfiguration, and this allows, in many cases, a higher degree of parallelism.

Similar to K-pipelining, the experimental results are of two types: theoretical, using randomly generated tests, and practical, for the MJPEG K-loop. The results for the randomly generated tests have shown a good potential for performance improvement, larger in average than for K-pipelining. The results for the MJPEG K-loop have shown that the reconfiguration time has a large impact in the overall performance, however the performance when using loop distribution is superior to the performance when using the combination of K-pipelining with unrolling.

In Chapter 8, loop skewing has been proposed for eliminating wavefront-like dependencies in K-loops. The skewed K-loop is then parallelized by means of

loop unrolling. Two scenarios of scheduling the hardware kernels have been proposed: one that considers that all kernel instances will run on the reconfigurable hardware, and one that considers that part of the kernel instances may run on the GPP for better performance. The degree of parallelism varies, as the inner skewed loop has variable loop bounds. Experiments have been performed on the Deblocking Filter K-loop extracted from the H.264 encoder/decoder. The results have shown that better performance can be achieved when the K-loop bounds are larger, because a higher number of iterations allows for a higher degree of parallelism.

9.2 Contributions

The main contributions of the research presented in this thesis can be summarized as follows.

- A framework that helps determine the optimal degree of parallelism for each hardware mapped kernel within a K-loop, taking into account area, memory and performance considerations. The decision is taken based on profiling information regarding the available area, the kernel's area requirements, the memory bandwidth, the kernel's memory transfers, the kernel's software and hardware execution times, and the software execution times of all other pieces of code in the K-loop.
- Algorithms and mathematical models for each of the proposed loop transformations in the context of K-loops. The algorithms are used to determine the best degree of parallelism for a given K-loop, while the mathematical models are used to determine the corresponding performance improvement. The algorithms have been validated with experimental results.
- An analysis of possible situations and justifications of when and why the loop transformations have or have not a significant impact on the K-loop performance.

9.3 Main Conclusions

The main conclusions of the research presented in this dissertation can be summarized as follows.

- Performance of kernel loops can be further improved even when the kernels inside have already been optimized.
- One of the main aspects that limits the degree of parallelism is the memory bandwidth. I/O intensive kernels do not benefit much from hardware parallelization.
- The area allowance also limits the degree of parallelism. Choosing a larger FPGA to execute on can give a performance boost.
- Parallel execution of multiple kernel instances on the reconfigurable hardware does not always improve the performance. This is the case when the execution time of the software part is larger than the execution time of the hardware part.
- Whenever the data dependencies allow it, parallel hardware-software execution should be performed.
- Splitting large K-loops and applying the parallelizing transformation on the smaller K-sub-loops can reduce the overall execution time, despite the reconfiguration overhead. The larger the K-loop size, the higher the probability that K-loop splitting improves the performance.
- Reconfiguration is expensive in terms of computing cycles, nevertheless it may be worth it in terms of performance.
- The K-loops with wavefront-like dependencies need a large number of iterations in order to benefit from parallelization after they are transformed with loop skewing.
- For K-loops with wavefront-like dependencies, it is beneficial to schedule part of the kernel instances to run on the GPP if the kernel is I/O intensive or if the K-loop has very large bounds.

9.4 Open Issues and Future Directions

From the research presented in this dissertation, some open issues have been identified. In this section, we list these issues, together with directions for the future research.

- As we have already mentioned in Chapter 2, the impact of several loop transformations that can enable parallelization of K-loops need to be

investigated. These transformations include loop bump, extend, index split, tiling, fusion, multi-dimensional retiming, striping.

- Considering that there are several aspects of interest when accelerating K-loops, such as memory size, bandwidth, power consumption, etc., it would be useful to see how various loop transformations interact with each other, and the impact that enabling or disabling a loop transformation has on the memory requirements or power consumption.

- The proposed methodology consists of static, compile-time algorithms. For the future, a parametrized compile-time approach should be investigated.

- The loop shifting/K-pipelining and the loop skewing transformations are used for eliminating only certain types of data dependencies. Other types of dependencies should also be considered.

- In signal processing applications, it is not unusual for different functions to have common code parts. For instance, an application may contain several FIR filter blocks that differ only in the number of taps and the set of filter coefficients [129]. For such cases, hardware reuse should be taken into account.

- In our experiments, only on-chip memory has been used. This type of memory is the fastest, but imposes severe constraints on the amount of application data that can fit into the memory. On today's platforms, other types of memory, such as the DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory), are available. While the external memory has the advantage of size, its latency is much higher than in the case of on-chip memory. A direction of future research would be to investigate the use of different types of memory, and propose new algorithms and mathematical models that take them into account.

- Relaxing the memory-related assumptions, such as the need for sequential memory transfers, can increase the range of targeted applications, and give more accurate performance estimations. A pre-processing stage for memory oriented optimizations may be useful.

- The proposed algorithms and mathematical models target reconfigurable architectures similar to the Molen architectures. Since the only processing element besides the GPP is the FPGA, the scheduling of the kernels is straightforward. While the parallelization algorithms are scalable, new

mathematical models are needed for different types of architectures, where the communication, the memory models and/or the number of processing elements differ from those that are on Molen. For an increased number of PEs, an algorithm for kernel scheduling would be required.

Bibliography

[1] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: A new loop parallelization technique. *ESOP '88: Proceedings of the 2nd European Symposium on Programming*, pages 221–235, 1988.

[2] Alexander Aiken and Alexandru Nicolau. Fine-grain parallelization and the wavefront method. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, 1990.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, pages 367–432, 1995.

[4] John R. Allen and Ken Kennedy. Automatic loop interchange. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 233–246, New York, NY, USA, 1984. ACM.

[5] Altera. Embedded Products Overview. `http://www.altera.com/technology/embedded/overview/emb-overview.html`.

[6] Altera. Intel Atom Industrial Reference Platform. `http://www.altera.com/end-markets/industrial/io-hub/intel/ind-msc-platform.html`.

[7] Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: software pipelining in the presence of structural hazards. *SIGPLAN Not.*, 30:139–150, 1995.

[8] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPs Is not (just) Polyhedral Software. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, 2011.

[9] ARM. Neon. `http://www.arm.com/products/processors/technologies/neon.php`.

[10] D. I. August. Hyperblock performance optimizations for ILP processors. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[11] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[12] S. Banerjee, E. Bozorgzadeh, and N. Dutt. PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In *Design Automation, 2006. Asia and South Pacific Conference on*, page 6, 24–27 Jan. 2006.

[13] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Exploiting application data-parallelism on dynamically reconfigurable architectures: Placement and architectural considerations. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(2):234 –247, feb. 2009.

[14] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[15] Utpal K. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[16] Luis Angel D. Bathen, Yongjin Ahn, Nikil D. Dutt, and Sudeep Pasricha. Inter-kernel data reuse and pipelining on chip-multiprocessors for multimedia applications. In *ESTImedia'09*, pages 45–54, 2009.

[17] Marcus Bednara and Jürgen Teich. Automatic synthesis of FPGA processor arrays from loop algorithms. *Journal of Supercomputing*, 26:149–165, September 2003.

[18] Michael Bedy, Steve Carr, Soner Onder, and Philip Sweany. Improving software pipelining by hiding memory latency with combined loads and prefetches. In *Interaction between Compilers and Computer Architectures*. Kluwer Academic Publishers, 2001.

[19] Mihai Budiu and Seth Copen Goldstein. Fast compilation for pipelined reconfigurable fabrics. *FPGA '99: Proceedings of the 1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*, pages 195–205, 1999.

[20] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.

[21] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *The Computer Journal*, pages 62–69, 2000.

[22] Timothy J. Callahan and John Wawrzynek. Instruction-level parallelism for reconfigurable computing. *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 248–257, 1998.

[23] J.M.P. Cardoso. Loop dissevering: A technique for temporally partitioning loops in dynamically reconfigurable computing platforms. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 181, Washington, DC, USA, 2003. IEEE Computer Society.

[24] J.M.P. Cardoso and P.C. Diniz. Modeling loop unrolling: approaches and open issues. *the 4th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS'04)*, pages 224–233, 2004.

[25] Steve Carr and Yiping Guan. Unroll-and-jam using uniformly generated sets. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–357, Washington, DC, USA, 1997. IEEE Computer Society.

[26] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[27] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P.G. Kjeldsberg, T. Van Achteren, and T. Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, 2002.

[28] C. Claus, F. H. Muller, J. Zeppenfeld, and W. Stechele. A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–7, 2007.

[29] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker. A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 535 –538, sept. 2008.

[30] The SUIF Compiler. The SUIF2 Compiler System. `http://suif.stanford.edu/suif/suif2/`.

[31] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. *Languages and Compilers for Parallel Computing*, pages 415–431, 1999.

[32] Jack W. Davidson and Sanjay Jinturkar. An aggressive approach to loop unrolling. *Technical Report: CS-95-26, 2001*, 1995.

[33] Bjorn De Sutter, Bingfeng Mei, Andrei Bartic, Tom Aa, Mladen Berekovic, Jean-Yves Mignolet, Kris Croes, Paul Coene, Miro Cupac, Assa Couvreur, Andy Folens, Steven Dupont, Bert Van Thielen, Andreas Kanstein, Hong-Seok Kim, and Suk Kim. Hardware and a tool chain for ADRES. In *Reconfigurable Computing: Architectures and Applications*, volume 3985 of *Lecture Notes in Computer Science*, pages 425–430. Springer Berlin / Heidelberg, 2006.

[34] Delft WorkBench. http://ce.et.tudelft.nl/DWB/.

[35] Steven Derrien, Sanjay Rajopadhye, and Susmita Sur Kolay. Combined instruction and loop parallelism in array synthesis for FPGAs. *Proceedings of the 14th international symposium on Systems Synthesis*, pages 165–170, 2001.

[36] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter High-Level Synthesis of Loops Using the Polyhedral Model. Springer Netherlands, 2008.

[37] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *Micro, IEEE*, 20(2):85–95, 2000.

[38] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 2004.

[39] J. J. Dongarra and A. R. Hinds. Unrolling loops in Fortran. *Software: Practice and Experience*, 1979.

[40] Ozana Silvia Dragomir, Elena Moscu Panainte, Koen Bertels, and Stephan Wong. Optimal unroll factor for reconfigurable architectures. In *Proceedings of the 4th International Workshop on Applied Reconfigurable Computing (ARC'08)*, pages 4–14, March 2008.

[41] K. Ebcioğlu and Toshio Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 213–229, London, UK, UK, 1990. Pitman Publishing.

[42] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping applications to the RaPiD configurable architecture. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 106, Washington, DC, USA, 1997. IEEE Computer Society.

[43] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD – Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, London, UK, 1996. Springer-Verlag.

[44] Alexandre E. Eichenberger and Edward S. Davidson. Stage scheduling: a technique to reduce the register requirements of a modulo schedule. *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 338–349, 1995.

[45] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum modulo schedules for minimum register requirements. *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 31–40, 1995.

[46] H. Munk et al. ACOTES project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming (IJPP)*, 2010. Special issue on European HiPEAC network of excellence member's projects. To appear.

[47] Antoine Fraboulet, Karen Kodary, and Anne Mignotte. Loop fusion for memory space optimization. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 95–100, New York, NY, USA, 2001. ACM.

[48] G. Gao, Q. Ning, and V. Van Dongen. Software pipelining for nested loops. Technical report, Advanced Compilers, Architectures and Parallel Systems (ACAPS) Technical Memo 53, School of Computer Science, McGill University, Montreal, 1993.

[49] Guang R. Gao, Yue-Bong Wong, and Qi Ning. A timed Petri-net model for fine-grain loop scheduling. *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation*, 1991.

[50] GCC. Auto-vectorization in gcc. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[51] Maya B. Gokhale and Janice M. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, 1998.

[52] Maya B. Gokhale, Janice M. Stone, and Edson Gomersall. Co-synthesis to a hybrid RISC/FPGA architecture. *The Journal of VLSI Signal Processing*, 24(2-3):165–180, March 2000.

[53] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Tayor. PipeRench: A reconfigurable architecture and compiler. *The Computer Journal*, 33:70–77, 2000.

[54] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 85–94, New York, NY, USA, 1994. ACM.

[55] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo. In *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21, pages 311–320. Forschungszentrum, 1996.

[56] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly - polyhedral optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, 2011.

[57] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[58] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from C codes for FPGAs. *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, 2005.

[59] Zhi Guo, Dinesh Chander Suresh, and Walid A. Najjar. Programmability and efficiency in reconfigurable computer systems. *Workshop on Software Support for Reconfigurable Systems*, 2003.

[60] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, 2004.

[61] Jeffrey Hammes, A. P. Wim Böhm, Charlie Ross, Monica Chawathe, Bruce A. Draper, Bob Rinker, and Walid A. Najjar. Loop fusion and temporal common subexpression elimination in window-based loops. *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, 2001.

[62] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE '01, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[63] Qubo Hu, Erik Brockmeyer, Martin Palkovic, Per Gunnar Kjeldsberg, and Francky Catthoor. Memory hierarchy usage estimation for global loop transformations. In *Proceedings of IEEE Norchip Conference*, pages 301–304, 2004.

[64] Qubo Hu, Martin Palkovic, and Per Gunnar Kjeldsberg. Memory requirement optimization with loop fusion and loop shifting. In *Euromicro Symposium on Digital System Design (DSD'04)*, pages 272–278, 2004.

[65] Qubo Hu, Arnout Vandecappelle, Martin Palkovic, Per Gunnar Kjeldsberg, Erik Brockmeyer, and Francky Catthoor. Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 606–611, Piscataway, NJ, USA, 2006. IEEE Press.

[66] Richard A. Huff. Lifetime-sensitive modulo scheduling. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–267, 1993.

[67] IBM. The Cell project at IBM research. `http://www.research.ibm.com/cell/`.

[68] Message Passing Interface. The Message Passing Interface (MPI) standard. `http://www.mcs.anl.gov/research/projects/mpi/`.

[69] H. Itoga, T. Haraikawa, Y. Yamashita, and Ikuo Nakata. Register allocation methods of improved software pipelining for loops with conditional branches. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 2006.

[70] Suneel Jain. Circular scheduling: a new technique to perform software pipelining. *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation*, pages 219–228, 1991.

[71] Guohua Jin, John Mellor-Crummey, and Robert Fowler. Increasing temporal locality with skewing and recursive blocking. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[72] Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan, and Iyad Ouaiss. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 616–622, New York, NY, USA, 1999. ACM.

[73] Arun Kejariwal, Alexander V. Veidenbaum, Alexandru Nicolau, Milind Girkar, Xinmin Tian, and Hideki Saito. On the exploitation of loop-level parallelism in embedded applications. *ACM Transactions in Embedded Computing Systems (TECS)*, 8(2):1–34, 2009.

[74] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 407–416, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[75] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, volume 768, pages 301–320, 1993.

[76] A. Koseki, H. Komastu, and Y. Fukazawa. A method for estimating optimal unrolling times for nested loops. *ISPAN '97: Proceedings of the*

1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97), page 376, 1997.

[77] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems. *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 239, 2002.

[78] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[79] Yuet Ming Lam, José Gabriel F. Coutinho, Chun Hok Ho, Philip Heng Wai Leong, and Wayne Luk. Multiloop parallelisation using unrolling and fission. *International Journal of Reconfigurable Computing*, 2010.

[80] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974.

[81] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR '93, Lecture Notes in Computer Science 715*, pages 398–416. Springer-Verlag, 1993.

[82] Jirong Liao, Weng-Fai Wong, and Tulika Mitra. A model for hardware realization of kernel loops. *13th International Conference on Field-Programmable Logic and Applications (FPL'03)*, 2003.

[83] D. C. Lin. Compiler support for predicated execution in superscalar processors. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[84] Peter List, Anthony Joch, Jani Lainema, Gisle Bjøntegaard, and Marta Karczewicz. Adaptive Deblocking Filter. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):614 – 619, July 2003.

[85] Meilin Liu, Qingfeng Zhuge, Zili Shao, and Edwin H.-M. Sha. General loop fusion technique for nested loops considering timing and code size. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 190–201, New York, NY, USA, 2004. ACM.

[86] Meilin Liu, Qingfeng Zhuge, Zili Shao, Chun Xue, Meikang Qiu, and Edwin H.-M. Sha. Maximum loop distribution and fusion for two-level loops considering code size. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:126–131, 2005.

[87] Qiang Liu, G.A. Constantinides, K. Masselos, and P.Y.K. Cheung. Combining data reuse exploitationwith data-level parallelization for FPGA targeted hardware compilation: A geometric programming framework. In *International Conference on Field Programmable Logic and Applications, 2008. FPL 2008.*, pages 179 –184, sept. 2008.

[88] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Energy reduction with run-time partial reconfiguration (abstract only). In *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 292–292, New York, NY, USA, 2010. ACM.

[89] Josep Llosa, Mateo Valero, Eduard Ayguade, and Antonio Gonzallez. Hypernode reduction modulo scheduling. *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 350–360, 1995.

[90] S. A. Mahlke. *Exploiting instruction level parallelism in the presence of conditional branches*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[91] Naraig Manjikian. Combining loop fusion with prefetching on shared-memory multiprocessors. *Parallel Processing, International Conference on*, 1997.

[92] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[93] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *SPAA '97: Proceedings of the 9th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, New York, NY, USA, 1997. ACM.

[94] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. *Field-Programmable Logic and Applications (FPL'03)*, pages 61–70, 2003.

[95] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, 2003.

[96] Yoichi Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1971. AAI7121189.

[97] Kalyan Muthukumar and Gautam Doshi. Software pipelining of nested loops. *Compiler Construction : 10th International Conference, CC 2001*, 2001.

[98] Walid A. Najjar, Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. *The Computer Journal*, pages 63–69, 2003.

[99] Steven Novack and Alexandru Nicolau. Resource directed loop pipelining: exposing just enough parallelism. *The Computer Journal*, 40:311–321, 1997.

[100] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: Architecture and implementations. *Micro, IEEE*, 19(2):37–48, 1999.

[101] Department of Informatics and Mathematics of the University of Passau. The polyhedral loop parallelizer: Loopo. `http://www.infosun.fim.uni-passau.de/cl/loopo/`.

[102] OpenMP.org. The OpenMP API specification for parallel programming. `http://openmp.org/wp/`.

[103] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. The PowerPC backend Molen compiler. *14th International Conference on Field-Programmable Logic and Applications (FPL'04)*, 2004.

[104] Nelson Luiz Passos and Edwin Hsing-Mean Sha. Full parallelism in uniform nested loops using multi-dimensional retiming. In *ICPP '94: Proceedings of the 1994 International Conference on Parallel Processing*, pages 130–133, Washington, DC, USA, 1994. IEEE Computer Society.

[105] Nelson Luiz Passos, Edwin Hsing-Mean Sha, and Steven C. Bass. Loop pipelining for scheduling multi-dimensional systems via rotation. In *DAC '94: Proceedings of the 31st annual Design Automation Conference*, pages 485–490, New York, NY, USA, 1994. ACM.

[106] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *Micro, IEEE*, 16(4):42–50, 1996.

[107] Louis-Noël Pouchet. PoCC: the Polyhedral Compiler Collection. `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`.

[108] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, pages 549–562, Austin, TX, January 2011. ACM Press.

[109] GNU Project. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`.

[110] PIPS Project. Pips: Automatic parallelizer and code transformation framework. `http://cri.ensmp.fr/pips/`.

[111] The LLVM Compiler Infrastructure Project. LLVM Overview. `http://llvm.org/`.

[112] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 249–258, New York, NY, USA, 2006. ACM.

[113] M. Rajagopalan and V. H. Allan. Efficient scheduling of fine grain parallelism in loops. *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 2–11, 1993.

[114] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *Micro, IEEE*, 20(4):47–57, 2000.

[115] J. Ramanujam. Optimal software pipelining of nested loops. *Proceedings of the 8th International Symposium on Parallel Processing*, pages 335–342, 1994.

[116] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, 1981.

[117] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, 1994.

[118] Hongbo Rong, Alban Douillet, R. Govindarajan, and Guang R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. *CGO '04: Proceedings of the international symposium on Code Generation and Optimization*, page 175, 2004.

[119] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multidimensional loops. In *CGO '04: Proceedings of the international symposium on Code Generation and Optimization*, 2004.

[120] Jesus Sanchez and Antonio Gonzalez. Software data prefetching for software pipelined loops. *Journal of Parallel and Distributed Computing*, 58:236–259, 1999.

[121] Vivek Sarkar. Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 2001.

[122] S. K. Singhai and K. S. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40:340–355, 1997.

[123] L. Song and K. Kavi. A technique for variable dependence driven loop peeling. *Algorithms and Architectures for Parallel Processing, International Conference on*, 0:0390, 2002.

[124] Litong Song, Robert Glck, and Yoshihiko Futamura. Loop peeling based on quasi-invariance/induction variables. *Wuhan University Journal of Natural Sciences*, 6:362–367, 2001.

[125] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *CODES '02: Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 7–12, New York, NY, USA, 2002. ACM.

[126] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, page 10340, Washington, DC, USA, 2004. IEEE Computer Society.

[127] Mark G. Stoodley and Corinna G. Lee. Software pipelining loops with conditional branches. *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 262–273, 1996.

[128] Boronr Su, Shiyuan Dink, Jian Vank, and Jinshi Xia. GURPR - a method for global software pipelining. *ACM SIGMICRO Newsletter*, 19, 1988.

[129] Wonyong Sung, Junedong Kim, and Soonhoi Ha. Memory efficient software synthesis from dataflow graph. In *System Synthesis, 1998. Proceedings. 11th International Symposium on*, pages 137 –142, dec 1998.

[130] Stamatis Vassiliadis, Georgi Gaydadjiev, Koen Bertels, and Elena Moscu Panainte. The Molen programming paradigm. In *the 3rd International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, pages 1–7, July 2003.

[131] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, November 2004.

[132] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors*, pages 17–27, 2003.

[133] Sven Verdoolaege, Koen Danckaert, Francky Catthoor, Maurice Bruynooghe, and Gerda Janssens. An access regularity criterion and regularity improvement heuristics for data transfer optimization by global loop transformations. In *in 1st Workshop on Optimization for DSP and Embedded Systems, ODES*, 2003.

[134] Jiang Wang and Christine Eisenbeis. Decomposed software pipelining: A new approach to exploit instruction level parallelism for loop programs. *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on*

Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, 1993.

[135] Zhong Wang, Michael Kirkpatrick, and Edwin Hsing-Mean Sha. Optimal loop scheduling for hiding memory latency based on two level partitioning and prefetching. *Design Automation Conference*, 2000.

[136] Dorothy Wedel. Fortran for the Texas Instruments ASC system. In *Proceedings of the conference on Programming languages and compilers for parallel and vector machines*, pages 119–132, New York, NY, USA, 1975. ACM.

[137] Markus Weinhardt and Wayne Luk. Pipeline vectorization for reconfigurable systems. *FCCM '99: Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 52–, 1999.

[138] Markus Weinhardt and Wayne Luk. Memory access optimization for reconfigurable systems. *Computers and Digital Techniques, IEEE Trans. Comput. Proceedings*, 148:105–112, 2001.

[139] Markus Weinhardt and Wayne Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[140] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University, Stanford, CA, USA, 1994.

[141] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM.

[142] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.

[143] Michael Wolfe. Loop skewing: the wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[144] Michael Wolfe. The Tiny loop restructuring research tool. In *International Conference on Parallel Processing*, pages 46–53, 1991.

[145] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

[146] Xilinx Inc. Virtex-4 FPGA User Guide. `http://www.xilinx.com/support/documentation/user_guides/ug070.pdf`.

[147] Xilinx Inc. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. `http://www.xilinx.com/support/documentation/user_guides/ug012.pdf`.

[148] Chun Xue, Zili Shao, Meilin Liu, Mei Kang Qiu, and Edwin Hsing-Mean Sha. *Loop Striping: Maximize Parallelism for Nested Loops*, pages 405–414. Springer-Verlag New York, Inc., 2006.

[149] Yana Yankova, Georgi Kuzmanov, Koen Bertels, Georgi Gaydadjiev, Yi Lu, and Stamatis Vassiliadis. DWARV: DelftWorkbench automated reconfigurable VHDL generator. *the 17th International Conference on Field Programmable Logic and Applications (FPL'07)*, 2007.

[150] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[151] YongKang Zhu, Grigorios Magklis, Michael L. Scott, Chen Ding, and David H. Albonesi. The energy impact of aggressive loop fusion. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 153–164, Washington, DC, USA, 2004. IEEE Computer Society.

[152] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM, New York, NY, USA, 1991.

[153] Claudiu Zissulescu, Todor Stefanov, and Bart Kienhuis. Laura: Leiden Architecture Research and Exploration Tool. In *the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, pages 1–3, 2003.

List of Publications

International Journals

1. O.S. Dragomir, T. P. Stefanov, K.L.M. Bertels, **Optimal Loop Unrolling and Shifting for Reconfigurable Architectures**, *ACM Transactions on Reconfigurable Technology and Systems*, Vol. 2, Issue 4, September 2009, pp. 1–24.

International Conferences

1. O.S. Dragomir, K.L.M. Bertels, **Loop Distribution for K-Loops on Reconfigurable Architectures**, *Proceedings of the International Conference on Design, Automation and Test in Europe 2011 (DATE 2011)*, Grenoble, France, March 2011.

2. O.S. Dragomir, K.L.M. Bertels, **K-Loops: Loop Skewing for Reconfigurable Architectures**, *Proceedings of the International Conference on Field-Programmable Technology (FPT 2009)*, Sydney, Australia, December 2009.

3. O.S. Dragomir, T. P. Stefanov, K.L.M. Bertels, **Loop Unrolling and Shifting for Reconfigurable Architectures**, *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL08)*, Heidelberg, Germany, September 2008.

4. O.S. Dragomir, E. Moscu Panainte, K.L.M. Bertels, S. Wong, **Optimal Unroll Factor for Reconfigurable Architectures**, *Proceedings of the 4th International Workshop on Applied Reconfigurable Computing (ARC2008)*, London, UK, March 2008.

International Workshops

1. O.S. Dragomir, K.L.M. Bertels, **Generic Loop Parallelization for Reconfigurable Architectures**, *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC08)*, Veldhoven, The Netherlands, November 2008.

2. O.S. Dragomir, K.L.M. Bertels, **Extending Loop Unrolling and Shifting for Reconfigurable Architectures**, *Architectures and Compilers for Embedded Systems (ACES)*, Edegem, Belgium, September 2008.

3. O.S. Dragomir, T. P. Stefanov, K.L.M. Bertels, **Loop Optimizations for Reconfigurable Architectures**, *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, L Aquila, Italy, July 2008.

4. O.S. Dragomir, E. Moscu Panainte, K.L.M. Bertels, **Loop Parallelization for Reconfigurable Architectures**, *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC07)*, Veldhoven, The Netherlands, November 2007.

Papers Unrelated to This Thesis

1. Z. Nawaz, O.S. Dragomir, T. Marconi, E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Recursive Variable Expansion: A Loop Transformation for Reconfigurable Systems**, *Proceedings of the International Conference on Field-Programmable Technology (FPT 2007)*, Kokurakita, Kitakyushu, JAPAN, December 2007.

Samenvatting

RECONFIGURABLE computing wordt steeds populairder, aangezien het een middenweg is tussen dedicated hardware, dat de beste prestaties levert, maar de hoogste productiekosten en langste time-to-market heeft, en tussen conventionele microprocessors, die erg flexibel zijn, maar minder prestaties leveren. Herconfigureerbare architecturen hebben als voordeel, dat ze hoge prestaties (gegeven door de hardware), grote flexibiliteit (gegeven door de software), en snelle time-to-market hebben.

Signaalverwerking vertegenwoordigt op dit moment een belangrijke categorie applicaties. In dergelijke applicaties vind met grote verwerkingsintensieve delen in de code (de application kernels) gewoonlijk binnen geneste loops. Voorbeelden zijn de JPEG en MJPEG beeldcompressie-algoritmen, het H.264 videocompressie-algoritme, Sobel Edge Detection, etc. n manier om de prestaties van deze applicaties te verbeteren is het toewijzen van de kernels op herconfigureerbare structuren.

De nadruk van deze dissertatie ligt op kernel loops (K-loops); dit zijn geneste loops, waar de loop body kernels bevat die aan hardware componenten zijn toegewezen. In deze dissertatie stellen we methoden voor om de prestaties van zulke K-loops te verbeteren door standaard loop transformaties te gebruiken om grofmazig parallellisme op loop-niveau bloot te leggen en toe te passen.

We richten ons op een herconfigureerbare architectuur dat een heterogeen systeem is, bestaande uit een General Purpose Processor en een Field Programmable Gate Array (FPGA). Onderzoeksprojecten die zich richten op herconfigureerbare architecturen proberen verschillende problemen aan te pakken: hoe de applicatie te partitioneren of hoe te besluiten welke onderdelen versneld zullen worden op de FPGA, hoe deze onderdelen (de kernels) te versnellen, en wat is de prestatiewinst? Slechts enkele, echter, trachten het grofmazige parallellisme op loop-niveau te benutten.

In dit onderzoek werken we toe naar het automatisch kiezen van het aantal kernel instanties om op de herconfigureerbare hardware te plaatsen op een flexibele manier dat een balans kan vinden tussen oppervlakte en prestaties. In deze dissertatie stellen we een generiek raamwerk voor dat helpt de optimale mate van parallellisme voor elke kernel binnen een K-loop die op hardware is geplaatst te bepalen, rekeninghoudend met oppervlakte-, geheugen-, en prestatieoverwegingen. Vervolgens presenteren we algoritmen en wiskundige

modellen voor verscheidene loop-transformaties met betrekking tot K-loops. De algoritmen worden gebruikt om de beste mate van parallellisme te bepalen voor een bepaalde K-loop, terwijl de wiskundige modellen gebruikt worden om de bijbehorende verbetering van de prestaties te bepalen. De algoritmen worden gevalideerd met experimentele resultaten. De loop-transformaties die we in deze dissertatie analyseren zijn loop unrolling, loop shifting, K-pipelining, loop distribution en loop skewing. Ook wordt er een algoritme geboden dat beslist welke transformaties gebruikt dienen te worden voor een bepaalde K-loop. Tenslotte geven we ook een analyse van mogelijke situaties waar de loop-transformaties wel of niet een significante impact hebben op de prestaties van de K-loop en mogelijke verklaringen waarom dit gebeurt.

About the Author

Ozana Silvia Dragomir was born in 1981 in Ploieşti, Romania. She received the BSc and MSc degrees in Computer Science at the Politehnica University of Bucharest in June 2004 and June 2005, respectively. The bachelor and master thesis were developed under the guidance of Prof. Dr. Ing. Irina Athanasiu, and in collaboration with Motorola/Freescale Semiconductors. In September 2006, she joined Prof. Dr. Stamatis Vassiliadis's group at the Computer Engineering department at Delft University of Technology, and started her PhD under the guidance of Dr. Koen Bertels. Her work has focused on loop optimizations in the context of reconfigurable architectures. The results of this research is presented in this thesis.

Ozana assisted in various courses as course instructor, such as 'Languages for distributed programming', 'Modeling and simulation', etc. at the Politehnica University of Bucharest, and 'System programming in C' and 'Embedded systems' at TU Delft.

Her research interests include compiler construction and optimizations, parallel programming models and paradigms, hardware/software co-design, image processing, embedded systems, reconfigurable computing, heterogeneous and multicore architectures.