Programmability Analysis of Image Processing Kernels on Multi-/Many-core Platforms

Sjoerd Hemminga

TU Delft Delft University of Technology

Programmability Analysis of Image Processing Kernels on Multi-/Many-core Platforms

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Sjoerd Hemminga

Wednesday 1 June, 2011

Author
Sjoerd Hemminga

Student number
1099876

Email
sjoerd@hemminga-online.nl

Title
Programmability Analysis of Image Processing Kernels on Multi-/Many-core Platforms

MSc presentation
June 8, 2011

Graduation Committee

Prof.dr.ir. H.J. Sips	Delft University of Technology, Faculty EEMCS
Dr.ir. H.X. Lin	Delft University of Technology, Faculty EEMCS
Dr. R.V. van Nieuwpoort	VU University Amsterdam, Faculty of Sciences
Dr. A.L. Varbanescu	Delft University of Technology, Faculty EEMCS

© 2011 Sjoerd Hemminga. All rights reserved.

Abstract

In the last few years, the computing industry has changed its course from ever higher clock speeds to multi- and many-core systems. These new parallel systems suffer a "programmability gap": there is a large gap between the performance a platform is theoretically capable of, and what the programmer is able to achieve.

Early programming models were very closely tied to the underlying hardware. A proliferation of programming models has lead to a push for standardisation. This standard became OpenCL.

In this thesis we investigated how the programmability of many-core systems has evolved in the past years. We started with the hardware-centric Cell SDK and compared it to the new standard OpenCL. We compared programmability on performance, portability and productivity. We found that OpenCL is a major improvement in portability and productivity, but that performance is still lacking.

Preface

There were times when writing this thesis felt like trying to land a jet plane: an important, yet very complex task that I have no clue how to do. And yet, even in the most frustrating and stressful moments, I thoroughly enjoyed every minute of it. From my first "bus error", 20 lines into my first Cell program, to the final "click" when everything just fell together.

I owe a great debt of gratitude to Ana Varbanescu, for her indispensable advise and guidance—from the other end of the world, if need be.

I also want to thank my parents, who have stood by me and taught me to aim to improve, and Lian Ien, for her undying love and care through it all.

Sjoerd Hemminga
Delft, The Netherlands
Wednesday 1 June, 2011

Contents

Preface **v**

Contents **vii**

List of Figures **xi**

List of Tables **xvii**

List of Algorithms **xix**

1 Introduction **1**
1.1 Research statement and approach 2
1.2 Related Work 3

2 Platforms **5**
2.1 Cell BE 5
2.1.1 PPE 6
2.1.2 SPEs 6
2.1.3 Programming the Cell 7
2.2 OpenCL 7
2.2.1 Memory 8
2.2.2 Programming OpenCL 8

3 Benchmarking **11**
3.1 Test cases 11
3.1.1 Scale down 11
3.1.2 Scale up 13
3.1.3 Mirror vertically 14
3.1.4 Mirror horizontally 14
3.1.5 Blend 15
3.1.6 Convolution 16
3.1.7 Enhance contrast 18
3.2 Benchmarking setup 20
3.2.1 Input data 20
3.2.2 Measurements 23

4 Cell SDK Implementations **25**
4.1 Fully sequential . . . 25
4.2 PPE vectorised . . . 26
4.3 SPEs . . . 29
4.4 SPEs double buffered . . . 32
4.5 SPEs vectorised . . . 32
4.6 Results . . . 32
4.6.1 Fully sequential . . . 32
4.6.2 PPE vectorised . . . 33
4.6.3 SPEs . . . 33
4.6.4 SPEs double buffered . . . 34
4.6.5 SPEs vectorised . . . 34
4.6.6 PPE and SPEs . . . 34
4.6.7 Summary . . . 34

5 OpenCL Implementations **37**
5.1 Fully sequential . . . 37
5.2 Naive implementation using `char` . . . 37
5.3 Naive implementation using other data types . . . 40
5.4 Parallel histogram . . . 43
5.5 Optimising memory accesses . . . 45
5.6 Vectorisation . . . 53
5.7 Using pinned memory . . . 61
5.8 Results . . . 61
5.8.1 Fully sequential . . . 61
5.8.2 Naive implementation using `char` . . . 62
5.8.3 Naive implementation using other data types . . . 63
5.8.4 Parallel histogram . . . 64
5.8.5 Using local memory . . . 64
5.8.6 Vectorisation . . . 64
5.8.7 Using pinned memory . . . 64
5.8.8 Summary . . . 65

6 Discussion **67**
6.1 Performance . . . 67
6.2 Portability . . . 69
6.3 Productivity . . . 70
6.4 Programmability . . . 71

7 Conclusions and Future Work **73**

Bibliography **77**

A Graphs **79**

A.1 Cell SDK 79
A.1.1 Fully sequential 79
A.1.2 PPE vectorised 82
A.1.3 SPEs 85
A.1.4 SPEs double buffered 91
A.1.5 SPEs vectorised 94
A.1.6 PPE and SPEs 97
A.2 OpenCL 100
A.2.1 Fully sequential 100
A.2.2 Naive implementation using `char` 104
A.2.3 Naive implementation using other data types 113
A.2.4 Parallel histogram 123
A.2.5 Using local memory 124
A.2.6 Vectorisation 131
A.2.7 Using pinned memory 137

List of Figures

2.1 An overview of the Cell BE processor 5

3.1 The scale down operation 12
3.2 The scale up operation 13
3.3 Vertical mirroring 14
3.4 Horizontal mirroring 15
3.5 Blend operation 17
3.6 Convolution with sharpen matrix 18
3.7 Convolution and edges 18
3.8 Enhance contrast 22

A.1 Sequential implementation of the scale down filter. 79
A.2 Sequential implementation of the scale up filter. 80
A.3 Sequential implementation of the mirror vertically filter. . . 80
A.4 Sequential implementation of the mirror horizontally filter. . 80
A.5 Sequential implementation of the blend filter. 81
A.6 Sequential implementation of the convolution filter. 81
A.7 Sequential implementation of the enhance contrast filter. . . 81
A.8 Sequential implementation of the scale down filter, compared to the PPE vectorised implementation. 82
A.9 Sequential implementation of the scale up filter, compared to the PPE vectorised implementation. 82
A.10 Sequential implementation of the mirror vertically filter, compared to the PPE vectorised implementation. 83
A.11 Sequential implementation of the mirror horizontally filter, compared to the PPE vectorised implementation. 83
A.12 Sequential implementation of the blend filter, compared to the PPE vectorised implementation. 83
A.13 Sequential implementation of the convolution filter, compared to the PPE vectorised implementation. 84
A.14 Sequential implementation of the enhance contrast filter, compared to the PPE vectorised implementation. 84
A.15 SPE-assisted implementation of the scale down filter, compared to the PPE vectorised implementation. 85

A.16 SPE-assisted implementation of the scale up filter, compared to the PPE vectorised implementation. 85
A.17 SPE-assisted implementation of the mirror vertically filter, compared to the PPE vectorised implementation. 86
A.18 SPE-assisted implementation of the mirror horizontally filter, compared to the PPE vectorised implementation. 86
A.19 SPE-assisted implementation of the blend filter, compared to the PPE vectorised implementation. 86
A.20 SPE-assisted implementation of the convolution filter, compared to the PPE vectorised implementation. 87
A.21 SPE-assisted implementation of the enhance contrast filter, compared to the PPE vectorised implementation. 87
A.22 A comparison of the number of SPEs used for the SPE-assisted implementation of the scale down filter. 88
A.23 A comparison of the number of SPEs used for the SPE-assisted implementation of the scale up filter. 88
A.24 A comparison of the number of SPEs used for the SPE-assisted implementation of the mirror vertically filter. . . . 89
A.25 A comparison of the number of SPEs used for the SPE-assisted implementation of the mirror horizontally filter. . . 89
A.26 A comparison of the number of SPEs used for the SPE-assisted implementation of the blend filter. 89
A.27 A comparison of the number of SPEs used for the SPE-assisted implementation of the convolution filter. 90
A.28 A comparison of the number of SPEs used for the SPE-assisted implementation of the enhance contrast filter. . . . 90
A.29 SPE-assisted implementation of the scale down filter, compared to the SPEs double buffered implementation. 91
A.30 SPE-assisted implementation of the scale up filter, compared to the SPEs double buffered implementation. 91
A.31 SPE-assisted implementation of the mirror vertically filter, compared to the SPEs double buffered implementation. . . 92
A.32 SPE-assisted implementation of the mirror horizontally filter, compared to the SPEs double buffered implementation. . . 92
A.33 SPE-assisted implementation of the blend filter, compared to the SPEs double buffered implementation. 92
A.34 SPE-assisted implementation of the convolution filter, compared to the SPEs double buffered implementation. 93
A.35 SPE-assisted implementation of the enhance contrast filter, compared to the SPEs double buffered implementation. . . 93
A.36 SPE vectorised implementation of the scale down filter, compared to the SPEs double buffered implementation. 94
A.37 SPE vectorised implementation of the scale up filter, compared to the SPEs double buffered implementation. 94

A.38 SPE vectorised implementation of the mirror vertically filter, compared to the SPEs double buffered implementation. . . 95
A.39 SPE vectorised implementation of the mirror horizontally filter, compared to the SPEs double buffered implementation. 95
A.40 SPE vectorised implementation of the blend filter, compared to the SPEs double buffered implementation. 95
A.41 SPE vectorised implementation of the convolution filter, compared to the SPEs double buffered implementation. . . 96
A.42 SPE vectorised implementation of the enhance contrast filter, compared to the SPEs double buffered implementation. . . 96
A.43 PPE and SPE vectorised implementations of the scale down filter compared. 97
A.44 PPE and SPE vectorised implementations of the scale up filter compared. 97
A.45 PPE and SPE vectorised implementations of the mirror vertically filter compared. 98
A.46 PPE and SPE vectorised implementations of the mirror horizontally filter compared. 98
A.47 PPE and SPE vectorised implementations of the blend filter compared. 98
A.48 PPE and SPE vectorised implementation of the convolution filter compared. 99
A.49 PPE and SPE vectorised implementations of the enhance contrast filter compared. 99
A.50 Sequential implementation of the scale down filter. 100
A.51 Sequential implementation of the scale up filter. 101
A.52 Sequential implementation of the mirror vertically filter. . . 101
A.53 Sequential implementation of the mirror horizontally filter. . 102
A.54 Sequential implementation of the blend filter. 102
A.55 Sequential implementation of the convolution filter. 103
A.56 Sequential implementation of the enhance contrast filter. . . 103
A.57 Naive implementation of the scale down filter, compared to the sequential implementation. 104
A.58 Naive implementation of the scale up filter, compared to the sequential implementation. 105
A.59 Naive implementation of the mirror vertically filter, compared to the sequential implementation. 105
A.60 Naive implementation of the mirror horizontally filter, compared to the sequential implementation. 106
A.61 Naive implementation of the blend filter, compared to the sequential implementation. 106
A.62 Naive implementation of the convolution filter, compared to the sequential implementation. 107

A.63 Naive implementation of the enhance contrast filter, compared to the sequential implementation. 107
A.64 Naive implementation of the scale down filter, kernel part of the run time. 108
A.65 Naive implementation of the scale up filter, kernel part of the run time. 109
A.66 Naive implementation of the mirror vertically filter, kernel part of the run time. 109
A.67 Naive implementation of the mirror horizontally filter, kernel part of the run time. 110
A.68 Naive implementation of the blend filter, kernel part of the run time. 110
A.69 Naive implementation of the convolution filter, kernel part of the run time. 111
A.70 Naive implementation of the enhance contrast filter, kernel part of the run time. 111
A.71 Naive implementation of the enhance contrast filter, histogram calculation part of the run time. 112
A.72 Naive implementation of the enhance contrast filter, normalisation part of the run time. 112
A.73 Different data types compared, for the scale down filter. . . 113
A.74 Different data types compared, for the scale up filter. 114
A.75 Different data types compared, for the mirror vertically filter. 114
A.76 Different data types compared, for the mirror horizontally filter. 115
A.77 Different data types compared, for the blend filter. 115
A.78 Different data types compared, for the convolution filter. . . 116
A.79 Different data types compared, for the enhance contrast filter. 116
A.80 Different data types compared, for the scale down filter, kernel run time only. 117
A.81 Different data types compared, for the scale up filter, kernel run time only. 118
A.82 Different data types compared, for the mirror vertically filter, kernel run time only. 118
A.83 Different data types compared, for the mirror horizontally filter, kernel run time only. 119
A.84 Different data types compared, for the blend filter, kernel run time only. 119
A.85 Different data types compared, for the convolution filter, kernel run time only. 120
A.86 Different data types compared, for the enhance contrast filter, kernel run time only. 120
A.87 Different data types compared, for the enhance contrast filter, histogram part of the kernel run time. 121

A.88 Different data types compared, for the enhance contrast filter, normalisation part of the kernel run time. 122
A.89 Comparing parallel histogram calculation with sequential histogram calculation. Histogram calculation times only. . . 123
A.90 Comparing the naive implementation to an implementation using local memory for the scale down filter, kernel run time only. 124
A.91 Comparing the naive implementation to an implementation using local memory for the scale up filter, kernel run time only. 125
A.92 Comparing the naive implementation to an implementation using local memory for the mirror vertically filter, kernel run time only. 126
A.93 Comparing the naive implementation to an implementation using local memory for the mirror horizontally filter, kernel run time only. 127
A.94 Comparing the naive implementation to an implementation using local memory for the blend filter, kernel run time only. 128
A.95 Comparing the naive implementation to an implementation using local memory for the convolution filter, kernel run time only. 129
A.96 Comparing the naive implementation to an implementation using local memory for the enhance filter, normalisation run time only. 130
A.97 Comparing a vectorised implementation to an implementation using local memory for the scale down filter, kernel run time only. 131
A.98 Comparing a vectorised implementation to an implementation using local memory for the scale up filter, kernel run time only. 132
A.99 Comparing a vectorised implementation to an implementation using local memory for the mirror vertically filter, kernel run time only. 133
A.100 Comparing a vectorised implementation to an implementation using local memory for the mirror horizontally filter, kernel run time only. 134
A.101 Comparing a vectorised implementation to an implementation using local memory for the blend filter, kernel run time only. 135
A.102 Comparing a vectorised implementation to an implementation using local memory for the convolution filter, kernel run time only. 136
A.103 Comparing a vectorised implementation to an implementation using local memory for the enhance filter, normalisation run time only. 136

A.104 Comparing a vectorised implementation to an implementation using pinned memory for the scale down filter. 137
A.105 Comparing a vectorised implementation to an implementation using pinned memory for the scale up filter. 138
A.106 Comparing a vectorised implementation to an implementation using pinned memory for the mirror vertically filter. . . 139
A.107 Comparing a vectorised implementation to an implementation using pinned memory for the mirror horizontally filter. 140
A.108 Comparing a vectorised implementation to an implementation using pinned memory for the blend filter. 141
A.109 Comparing a vectorised implementation to an implementation using pinned memory for the convolution filter. 142
A.110 Comparing a vectorised implementation to an implementation using pinned memory for the enhance filter. 142

List of Tables

3.1 Description of variables used 11
3.2 Sizes of the input images 22

5.1 OpenCL experiments 61

6.1 Comparison of filter run times 68
6.2 Peak performance and memory bandwidth 69
6.3 Programmability of the platforms summarised 71

List of Algorithms

3.1 Sequential pseudocode for the scale down filter 12
3.2 Sequential pseudocode for the scale up filter 13
3.3 Sequential pseudocode for the mirror vertically filter 14
3.4 Sequential pseudocode for the mirror horizontally filter 15
3.5 Sequential pseudocode for the blend filter 16
3.6 Sequential pseudocode for the convolution filter 19
3.6 Sequential pseudocode for the convolution filter (continued) . . 20
3.7 Sequential pseudocode for the enhance contrast filter 21

4.1 Basic structure of the sequential algorithms 25
4.2 Vectorised pseudocode for the scale down filter 26
4.3 Vectorised pseudocode for the scale up filter 27
4.4 Vectorised pseudocode for the mirror vertically filter 27
4.5 Vectorised pseudocode for the mirror horizontally filter 28
4.6 Vectorised pseudocode for the blend filter 28
4.7 Vectorised pseudocode for the convolution filter 30
4.8 Vectorised pseudocode for the enhance contrast filter 31

5.1 Pseudocode for the OpenCL scale down filter 38
5.2 Pseudocode for the OpenCL scale up filter 38
5.3 Pseudocode for the OpenCL mirror vertically filter 39
5.4 Pseudocode for the OpenCL mirror horizontally filter 39
5.5 Pseudocode for the OpenCL blend filter 40
5.6 Pseudocode for the OpenCL convolution filter 41
5.6 Pseudocode for the OpenCL convolution filter (continued) . . 42
5.7 Sequential pseudocode to find normalisation cut-off points . . 42
5.8 Pseudocode for the OpenCL enhance contrast filter 43
5.9 Parallel pseudocode to find normalisation cut-off points 44
5.10 Pseudocode for the OpenCL scale down filter using local memory 46
5.11 Pseudocode for the OpenCL scale up filter using local memory 47
5.12 Pseudocode for the OpenCL mirror vertically filter using local memory 48

5.13 Pseudocode for the OpenCL mirror horizontally filter using local memory 49
5.14 Pseudocode for the OpenCL blend filter using local memory . 50
5.15 Pseudocode for the OpenCL convolution filter using local memory 51
5.15 Pseudocode for the OpenCL convolution filter using local memory (continued) 52
5.16 Pseudocode for the OpenCL enhance contrast filter using local memory 53
5.17 Pseudocode for the vectorised OpenCL scale down filter 54
5.18 Pseudocode for the vectorised OpenCL scale up filter 55
5.19 Pseudocode for the vectorised OpenCL mirror vertically filter . 56
5.20 Pseudocode for the vectorised OpenCL mirror horizontally filter 57
5.21 Pseudocode for the vectorised OpenCL blend filter 57
5.22 Pseudocode for the vectorised OpenCL convolution filter . . . 58
5.22 Pseudocode for the vectorised OpenCL convolution filter . . . 59
5.23 Pseudocode for the vectorised OpenCL enhance contrast filter 60

Chapter 1

Introduction

In early 2005, a consortium of IBM, Sony and Toshiba released the first key documentation of their new Cell BE processor[11], with more details[19, 12] and an SDK[13] released later that year. The Cell BE is a processor based on IBM's POWER architecture, enhanced with eight newly designed accelerator processing cores[1]. This heterogeneous many-core[2] design was a novel approach and generated a lot of research attention. Developing applications for the Cell BE platform requires detailed knowledge of the platform. This makes it both hard to program and hard to port a Cell BE application to a new platform.

Around the same time the "megahertz wars" between mainstream desktop and server processor manufacturers reached its conclusion. The manufacturers moved on to produce multiple processing cores on the same die. Before this new trend, speed-ups could easily be achieved by using a newer generation of processors. Now applications need to be changed to exploit parallelism if they are to take advantage of the available computing resources.

In late 2006, both ATI[5] and NVIDIA[9] released their GPGPU[3] tool kits, allowing developers to exploit the computing power of the GPU for non-graphics purposes. The GPU works as an accelerator for the CPU. Like the Cell SDK, programming with these tool kits requires intimate knowledge of the hardware.

Because these platforms are so specific and their programming so hardware-centric, someone learning to work on a new platform will have to start at the beginning. Porting an application will generally come down to programming it from the ground up. Gaining good performance will require different

[1] Not all of these accelerator cores are available on all chips. The Cell BE chip used in the PlayStation 3 has one core reserved for the operating system and one core is disabled during production for quality control reasons.[27]

[2] We reserve the term "multi-core" for the general purpose multi-core processors used in modern day desktop and server machines. For the general case we use the term "many-core".

[3] General Purpose computing on GPUs (Graphics Processing Units).

strategies on each platform, sometimes radically so. If a programmer finds that good performance cannot be obtained on a platform, the whole process of programming and tuning will have to be restarted. Searching for a platform with good performance thus amounts to a depth-first search[30]. In many cases, the performance gap between what the hardware is capable of and what the programmer is able to achieve in a reasonable time is due to this difficult programming, and it is therefore known as the "programmability gap"[32].

In 2008, the Khronos Group, of OpenGL fame, announced the formation of their Compute Working Group[23], followed later that year by the release of the OpenCL 1.0 specification[22]. OpenCL is specifically designed to provide a standardised programming model for "multi-core CPUs, GPUs, Cell-type architectures and other parallel processors"[22]. It is blessed industry-wide, with NVIDIA[29], ATI/AMD[1], and IBM[14] integrating support into their tool kits within a year. The standardised model provided by OpenCL should reduce portability issues. By May 2011, all major hardware players have released their OpenCL 1.1 framework[21].

1.1 Research statement and approach

In this thesis we investigate how the programmability of many-core systems has evolved, starting with the hardware-centric Cell SDK and comparing it to the newer, application-centric OpenCL. We compare programmability based on performance, portability and productivity, as described in [32]:

Performance The primary reason to write many-core applications is performance. If a programmer cannot reach good performance on a platform that is theoretically capable of achieving it, the platform has low programmability. The difference in performance between what the platform is capable of and what the programmer is able to achieve is known as the "programmability gap".

Portability Because there are so many widely varying platforms, knowledge about one platform is often not very useful on another. If the application cannot be ported from one platform to another, the entire process of programming and tuning will have to be redone if another platform is desired, e.g. if the required performance cannot be achieved on the chosen platform.

Productivity We can define productivity loosely as the achieved performance per unit of time spent developing the application. A platform cannot be considered very programmable if writing an application with good performance for it takes a lot of time.

Our investigation is based on the empirical study of seven 2D image filters for both the Cell SDK and OpenCL. We have implemented the most common performance optimisations for both platforms. We will first describe the platforms in Chapter 2, and the image filters in Chapter 3. Chapters 4 and 5 describe the optimisations implemented for the Cell BE and OpenCL platforms, respectively. These chapters also include extensive performance measurements for each platform. Chapter 6 contains a full comparison of these platforms. Finally, in Chapter 7 we describe our conclusions.

1.2 Related Work

Many studies have done performance comparisons of CUDA applications and OpenCL ports of those applications, e.g. [20, 26, 6, 7]. Other studies have done application performance comparisons on Cell BE and GPUs[2, 8], and on Cell BE and multi-core processors[37, 36], using the native programming models. None of these studies look at programmability.

In [33], Varbanescu et al. survey a plethora of many-core programming models. They distinguish two classes of programming models: the hardware-centric models, including the Cell SDK, and the generic models, with OpenCL falling in that category. They find that OpenCL is one of two good candidates for a unified programming model for many-core applications. Their study does not look at specific applications, so it does not include performance measurements, but rather it focuses on the architectural features of each programming model.

Van Nieuwpoort and Romein[31] have looked at using many-core processors for correlating radio astronomy signals. Their application, like ours, is very memory intensive. They did a performance comparison between several hardware platforms, including the Cell BE, a multi-core processor, an NVIDIA GPU and an ATI GPU. These comparisons were done using the native programming models for these platforms: the Cell SDK for the Cell BE, CUDA for the NVIDIA card, and Brook+ and CAL for the ATI card. They did not include performance measurements for OpenCL, but they did include it in the programmability study they performed. They describe the strengths and weaknesses of each platform.

Chapter 2

Platforms

This chapter discusses the two platform we compare. In Section 2.1 we described the Cell BE and in Section 2.2 we describe OpenCL.

2.1 Cell BE

The Cell BE chip contains up to nine processing cores. Eight of those cores are called Synergistic Processing Element (SPEs). Those are identical. The ninth core is different and is called the PowerPC Processing Element (PPE). [15] The PPE is described in Section 2.1.1. The SPEs are described in Section 2.1.2.

The PPE and the SPEs are connected by a bus (the Element Interface Bus). See Figure 2.1 for an overview. The bus is also connected to the Memory Interface Controller. The Memory Interface Controller handles the communication to and from the main memory. Communication to other devices is handled by the Cell Broadband Engine Interface, which is also connected to the bus (not shown).

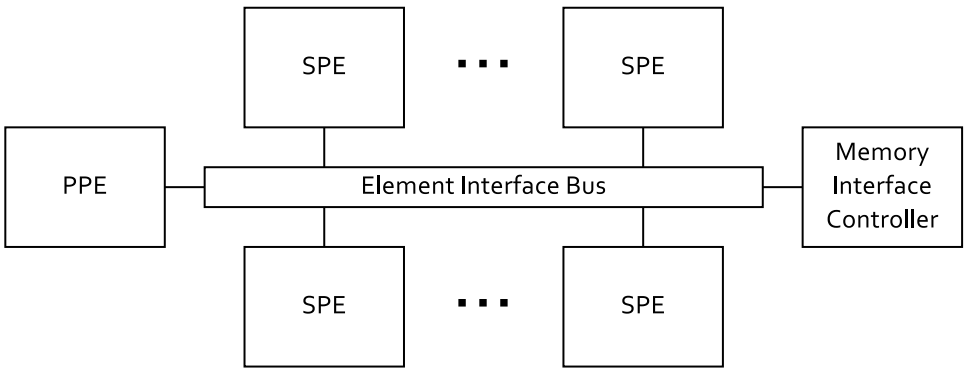


Figure 2.1 – *An overview of the Cell BE processor. Simplified from [15].*

2.1.1 PPE

The PPE's speciality is control and execution. It runs the operating system and instructs the SPEs on what it is supposed to do. The PPE conforms to the PowerPC architecture, which makes it possible for programs written for other PowerPC processors to run on the PPE without modification.

It is a 64-bit processor that can run in 32-bit mode. The PPE has 32 general-purpose registers and 32 floating-point registers, both of which are 64-bits wide. In 32-bit mode it can only use the lower 32 bits of the general-purpose registers.[16]

In addition to the standard PowerPC instruction set, it also has a SIMD[1] (or vector) instruction set. There are 32 vector registers available to work on. These are 128 bits wide.[16]

2.1.2 SPEs

The SPEs are sidekick processing cores. Their speciality is SIMD work. Its instruction set is specifically designed for 'data-rich, compute-intensive SIMD and scalar applications.'[16] Because of this new instruction, SPE programs must be written specifically for the SPE.

An SPE has no CPU cache and no direct access to the main memory of the machine. Instead each SPE has 256 KiB on-chip memory called the Local Store (Local Store). To copy data from main memory to the Local Store, an SPE needs to do DMA transfers.

To perform DMA transfers, the SPEs have a component called the Memory Flow Controller. It can work independently from the rest of the SPE, making it possible to transfer data and work on other data at the same time.[15] A single DMA transfer can copy up to 16 KiB into the Local Store.[16]

SPEs have 128 registers, which are used for all scalar and vector calculations. All registers are 128 bits wide. Scalar values smaller than 128 bits need to be stored in a designated area[2] of the 128-bit register to do scalar operations. If the value is not stored in this area, the value in the register will have to be rotated until it is, before scalar operations on it can be done.[16]

The pipeline on SPEs has simple rules, which can make branching in a program a costly endeavour if the branch prediction fails. A failed prediction would incur a penalty of at least 18 CPU cycles, whereas a successfully predicted branch would only cost 1 cycle.[16] The SPEs are therefore ill-suited for programs that require a lot of branching.

The PPE controls the SPEs on a high level. It loads a program into the memory of an SPE (or instructs an SPE to do so itself) and will then tell the SPE to run it.[16]

[1] Single Instruction, Multiple Data

[2] This is called the 'preferred scalar slot'.

2.1.3 Programming the Cell

The GNU Compiler Collection (GCC) has support for the PowerPC architecture and the SPE. This allows the Cell to be programmed in C and C++. Additional header files make it possible to use instructions that are not available using the C standard library, like the SIMD instructions and instructions for communication between SPEs and the rest of the system.

Running a program on an SPE is rather involved. The following steps need to be taken to run a program on SPEs:

1. Open an SPE program.
2. Prepare the data for the SPE.
3. Create an SPE context.
4. For each SPE to be used:
   (a) Load the program into the Local Store of the SPE.
   (b) Create a new thread (e.g. using the `pthread` library). In the new thread:
      i. Run the context. This will wait until the SPE program finishes.
5. For each SPE to be used:
   (a) Wait for the thread to finish.
   (b) Destroy the context.
6. Close the SPE program.

Each step requires its own error checking and handling, as is standard practise when programming in C.

2.2 OpenCL

OpenCL is a standard for general purpose parallel computing[25]. It is supported by all major hardware vendors.

The OpenCL model is based on a host with one or more connected accelerator devices. The host has no special features, but the accelerator devices can perform parallel computations. It runs many threads called "work-items", that are further grouped into "work-groups". Work-items can synchronise and communicate within a work-group, but they cannot synchronise and communicate with work-items in another work-group.

2.2.1 Memory

There are four types of memory in OpenCL [25]: private memory, local memory, global memory and constant memory. Private memory is private to each work-item. It is not accessible to any other thread. Local memory is shared between work-items in a work-group, but is not accessible by threads in other work-groups. Global memory is accessible to all threads. Constant memory is a region of global memory that cannot change during execution.

2.2.2 Programming OpenCL

The host can be programmed using standard programming languages, which interface with the accelerator device using a library. Currently, bindings exist for C and C++ [24]. To program a device we need to use a new programming language specified by the standard. The language is based on C and is very similar to it. Before an OpenCL program can be run, it needs to be compiled, by the host program. Normally, this will need to be done every time the program is run. However, the host program can choose to store a compiled representation of the program, and load that instead. This is faster, but requires more effort of the programmer.

Running an OpenCL program is even more involved than running a program on the Cell BE. A host program needs to perform the following actions, before it can run:

1. Select a platform
2. Select a device
3. Create a context
4. Create a command queue
5. Read the source code of the device program
6. Create a program object from source code
7. Build the device program
8. Create kernel program
9. Allocate buffers on device (once for each image, both input and output)
10. Enqueue a copy of input data to the buffer (once for each input image; actual copy is asynchronous)
11. Set kernel argument (once for each argument; allocated buffers need to be passed as well)
12. Run kernel

13. Enqueue a copy of output data back to the host memory (copy is asynchronous)
14. Free objects

Chapter 3

Benchmarking

3.1 Test cases

To test the two platforms, we have written programs for seven test cases, which are described in Sections 3.1.1 to 3.1.7. To describe the test cases we will use some variables. These are listed in Table 3.1.

Each test case modifies a bitmap image by running an image filter. The programs use the same library to read the images from the disc. The images are converted to greyscale when they are read into memory. Images are then represented as an array of width $\times$ height bytes.

Each byte represents a single pixel with a value in the range $[0, 255]$. 0 represents black, and a value of 255 represents a completely white pixel. Other shades of grey are represented by the values between these two extremes. Index $(0, 0)$ in the array corresponds to the pixel in the lower left corner of the image. This simple data representation makes working with it very convenient.

Variable	Description
B	Input or base image
O	Output image
w_X	Width of image X
h_X	Height of image X
$X_{ij}/X_{i,j}$	Pixel value of the pixel on position (i, j) in image X

Table 3.1 – *Description of variables used*

3.1.1 Scale down

The goal of the first filter is to scale down an image by a factor of 2 in both dimensions. For example, an image of size 128×96 will be scaled down to 64×48.

This is accomplished by taking square blocks of four pixels from the input image and calculating the arithmetic mean of those pixels. The result is a pixel of the output image. See Figure 3.1 for an example and Algorithm 3.1 for pseudocode.

In essence this is a simplified bilinear interpolation. We can use the arithmetic mean because the dimensions of the resulting image are exactly half of those of the input image. If we used another factor to scale the image, we would have to use a weighted average.

Algorithm 3.1: Sequential pseudocode for the scale down filter

Input : in: image of size $w \times h$
Output : out: image of size $\frac{w}{2} \times \frac{h}{2}$

```
1 for row ← 0 to h/2 − 1 do
2     for column ← 0 to w/2 − 1 do
3         sum ← in_{2·column,2·row}
4         sum ← sum + in_{2·column+1,2·row}
5         sum ← sum + in_{2·column,2·row+1}
6         sum ← sum + in_{2·column+1,2·row+1}
7         out_{column,row} ← sum/4
```

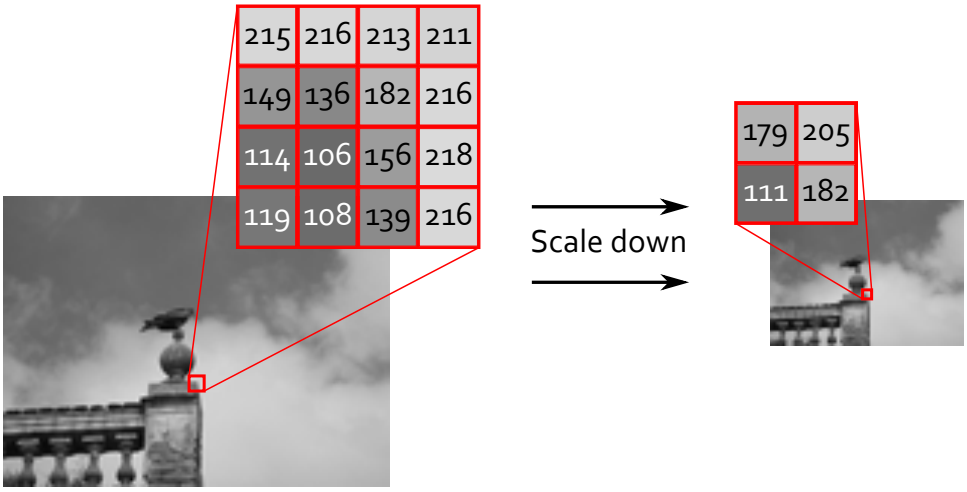


Figure 3.1 – *The scale down operation. The pixels of the smaller image are calculated by averaging four pixels of the image on the left. E.g., 179 is calculated by averaging 215, 216, 149 and 136.*

3.1.2 Scale up

The scale up filter is the opposite of the scale down filter. Instead of decreasing the dimensions of an image, we will increase both dimensions by a factor 2. For example, an image of size 128×96 will end up as an image of size 256×192.

We do this using nearest-neighbour interpolation. Each pixel in the output image will have the value of the closest pixel in the input image. Calculating from the input image: each input pixel has its value copied into a 2×2 square of output pixels. See Figure 3.2 for an example and Algorithm 3.2 for pseudocode.

Algorithm 3.2: Sequential pseudocode for the scale up filter

Input : in: image of size $w \times h$
Output : out: image of size $2w \times 2h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 do
3         out_{2·column,2·row} ← in_{column,row}
4         out_{2·column+1,2·row} ← in_{column,row}
5         out_{2·column,2·row+1} ← in_{column,row}
6         out_{2·column+1,2·row+1} ← in_{column,row}
```

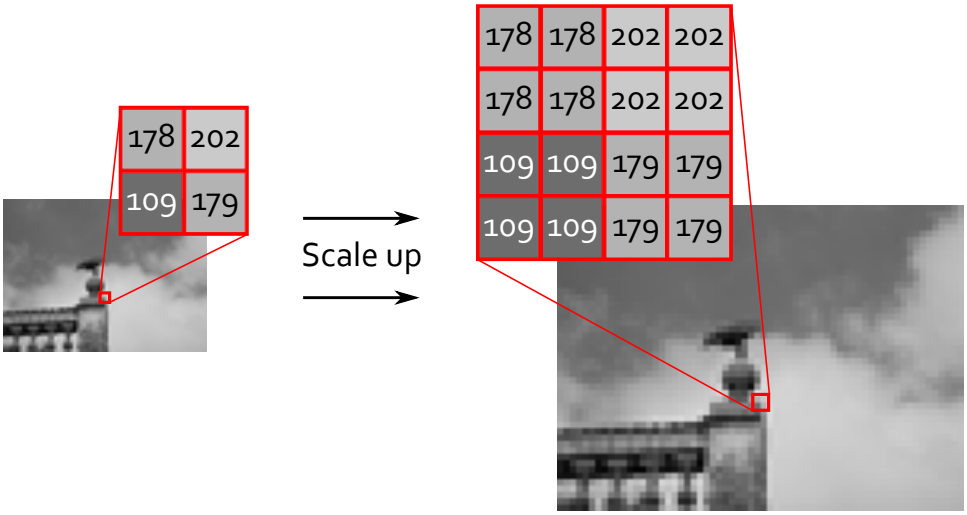


Figure 3.2 – *The scale up operation. Each pixel in the input image is copied into four pixels in the output image.*

3.1.3 Mirror vertically

In this filter the image is flipped around the x-axis: up becomes down and down becomes up. The algorithm is straight-forward: the top row of the input image is copied to the bottom row of the output image, the second row of the input to the next-to-last row of the output, and so on. See Figure 3.3 for an example and Algorithm 3.3 for pseudocode.

Algorithm 3.3: Sequential pseudocode for the mirror vertically filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 do
3         out_{column,h−row−1} ← in_{column,row}
```

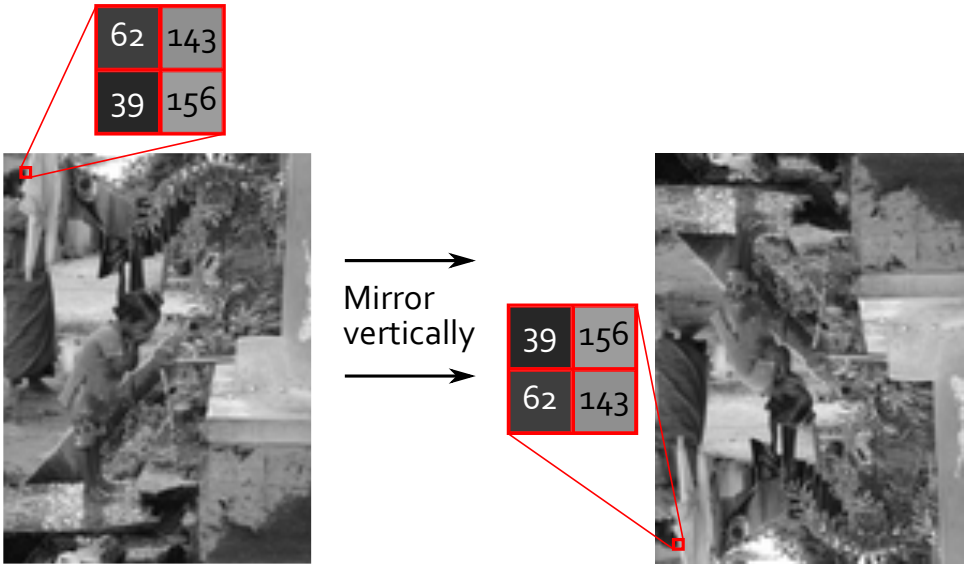


Figure 3.3 – *Vertical mirroring*

3.1.4 Mirror horizontally

In this filter we will flip the image around the y-axis. The output will look like the input image when it is viewed in a mirror: left becomes right, right becomes left.

For each row, the first pixel in the input image will be copied to the last pixel of the output image, the second input pixel is copied to the penultimate

pixel in the output, et cetera. See Figure 3.4 for an example and Algorithm 3.4 for pseudocode.

Algorithm 3.4: Sequential pseudocode for the mirror horizontally filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 do
3         out_{w−column−1,row} ← in_{column,row}
```

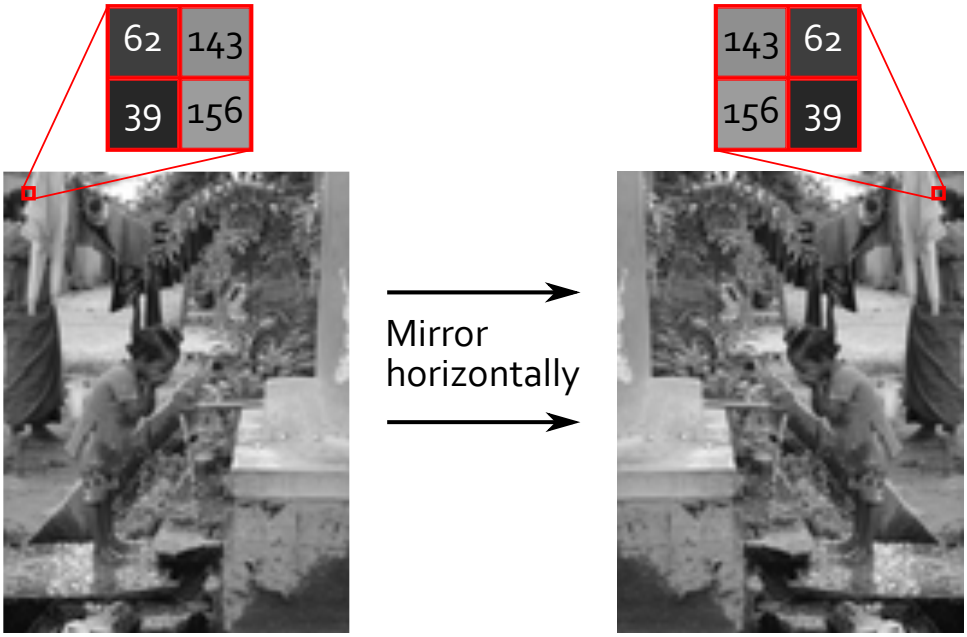


Figure 3.4 – *Horizontal mirroring*

3.1.5 Blend

The blend filter mixes two images together to form a new image. Apart from the base image B and the overlay image S (for small), it requires three parameters: x and y to indicate where the overlay image should have its lower left corner, and α as a blending factor. The blending factor, which is a value between 0 and 1, is used to calculate which image is the most dominant. The pixel of the base image is multiplied with $(1 - \alpha)$ and the pixel of the overlay image with α. Both are added together and stored as

the output pixel.

The calculations are only performed if the images need to be blended at that spot. If the overlay image is not defined at a pixel, the base image is simply copied.

Putting it all together:

$$O_{ij} = \begin{cases} B_{ij} & \text{if } i < x \text{ or } i \geq x + w_S \\ B_{ij} & \text{if } j < y \text{ or } j \geq y + h_S \\ B_{ij} \cdot (1 - \alpha) + S_{ij} \cdot \alpha & \text{otherwise} \end{cases} \tag{3.1}$$

See Figure 3.5 for an example and Algorithm 3.5 for pseudocode.

Algorithm 3.5: Sequential pseudocode for the blend filter

Input : base: image of size $w \times h$
overlay: image of size $w_s \times h_s$
x: $1 \leq x < w - w_s$, the x-coordinate to start blending
y: $1 \leq y < h - h_s$, the y-coordinate to start blending
α: $0 \leq \alpha \leq 1$, the blending factor

Output : out: image of size $w \times h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 do
3         if x ≤ column < x + w_s and y ≤ row < y + h_s then
4             out_column,row ←
              (1 − α) · base_column,row + α · overlay_column−x,row−y
5         else
6             out_column,row ← base_column,row
```

3.1.6 Convolution

The sixth filter is a convolution operation. Each pixel in an image and its eight immediate neighbours are treated as a 3×3 matrix. This matrix is then entrywise multiplied (called the Hadamard product) with a 3×3 convolution kernel C, after which all elements of the result are added to create the output pixel.

With $\circ$ indicating the entrywise product of two matrices, a complete description of the algorithm can be given as:

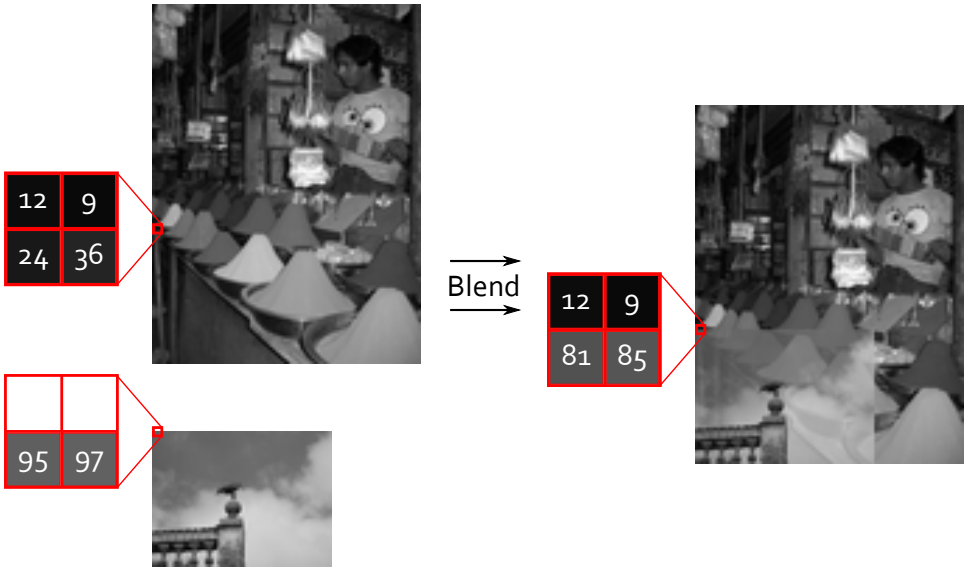


Figure 3.5 – *The blend operation for $x = 0$, $y = 0$ and $\alpha = 0.8$. The upper two pixels in the example are only available in the base image. They are simply copied. The lower two pixels in the base image are multiplied with $1 - \alpha = 0.2$. The two pixels in the overlay image are multiplied with α. The results of these multiplications are added together to create the output pixel.*

$$
\begin{aligned}
A &= \begin{bmatrix} B_{i-1,j-1} & B_{i,j-1} & B_{i+1,j-1} \\ B_{i-1,j} & B_{i,j} & B_{i+1,j} \\ B_{i-1,j+1} & B_{i,j+1} & B_{i+1,j+1} \end{bmatrix} \\
O' &= \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \circ \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \\
&= \begin{bmatrix} A_{11}C_{11} & A_{12}C_{12} & A_{13}C_{13} \\ A_{21}C_{21} & A_{22}C_{22} & A_{23}C_{23} \\ A_{31}C_{31} & A_{32}C_{32} & A_{33}C_{33} \end{bmatrix} \\
O_{ij} &= O'_{11} + O'_{12} + O'_{13} + \\
&\quad O'_{21} + O'_{22} + O'_{23} + \\
&\quad O'_{31} + O'_{32} + O'_{33}
\end{aligned}
\tag{3.2}
$$

For our benchmarks, we used a matrix with a sharpen effect:

$$
C = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{3.3}
$$

See Figure 3.6 for an example.

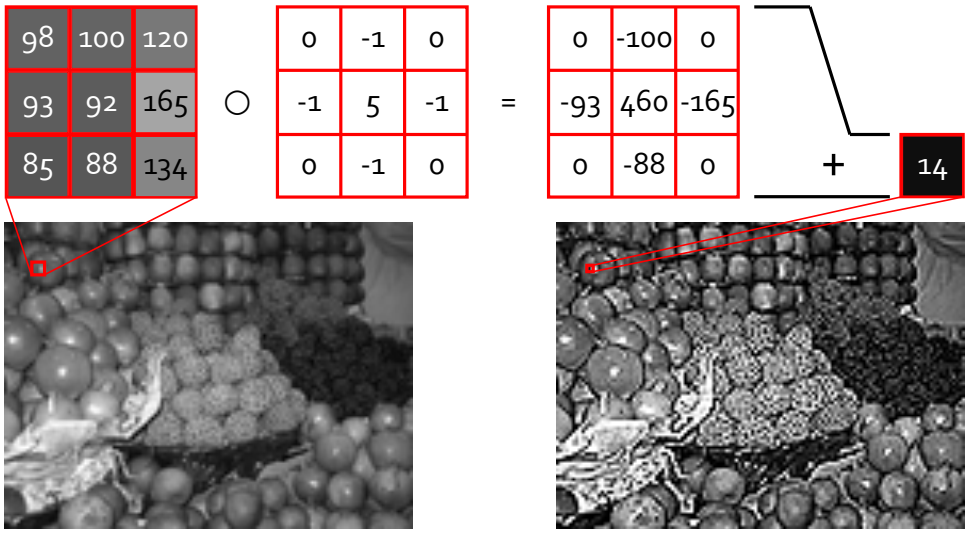


Figure 3.6 – *The convolution operation with a sharpen matrix. A 3×3 matrix is extracted from the image and element-wise multiplied with the sharpen matrix (indicated by $\circ$). All elements of the resulting matrix are added together to get the resulting pixel.*

This algorithm requires minor modifications at the edges of the image. To make edges work, we use the edge line twice. See Figure 3.7 for an example and Algorithm 3.6 for pseudocode of the entire algorithm.

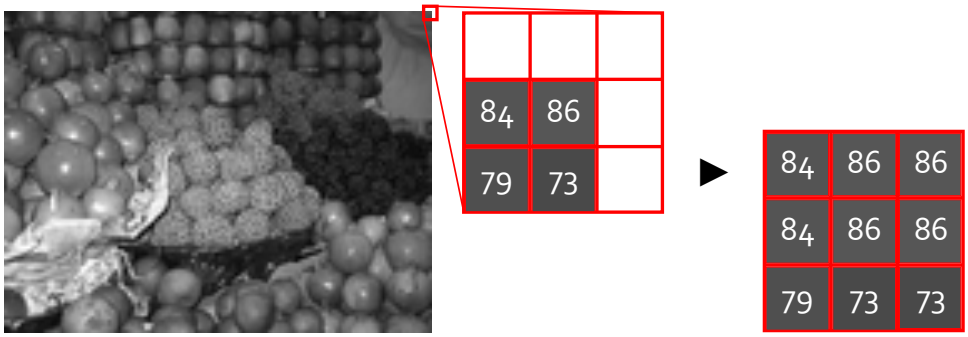


Figure 3.7 – *The convolution algorithm deals with edges and corners by copying the edge row (or column) to the empty row (or column).*

3.1.7 Enhance contrast

The last filter is image contrast enhancement, using the contrast stretching method.

Algorithm 3.6: Sequential pseudocode for the convolution filter

Input : in: image of size $w \times h$
C: 3×3 convolution kernel
Output : out: image of size $w \times h$

▷ *Bottom row*

1 $r \leftarrow \text{in}_{0,0} \cdot C_{0,0} + \text{in}_{0,0} \cdot C_{0,1} + \text{in}_{1,0} \cdot C_{0,2}$ ▷ *Bottom left corner*
2 $r \leftarrow r + \text{in}_{0,0} \cdot C_{1,0} + \text{in}_{0,0} \cdot C_{1,1} + \text{in}_{1,0} \cdot C_{1,2}$
3 $r \leftarrow r + \text{in}_{0,1} \cdot C_{2,0} + \text{in}_{0,1} \cdot C_{2,1} + \text{in}_{1,1} \cdot C_{2,2}$
4 $\text{out}_{0,0} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

5 **for** $col \leftarrow 1$ **to** $w - 2$ **do** ▷ *Bottom row*
6 $r \leftarrow \text{in}_{\text{col}-1,0} \cdot C_{0,0} + \text{in}_{\text{col},0} \cdot C_{0,1} + \text{in}_{\text{col}+1,0} \cdot C_{0,2}$
7 $r \leftarrow r + \text{in}_{\text{col}-1,0} \cdot C_{1,0} + \text{in}_{\text{col},0} \cdot C_{1,1} + \text{in}_{\text{col}+1,0} \cdot C_{1,2}$
8 $r \leftarrow r + \text{in}_{\text{col}-1,1} \cdot C_{2,0} + \text{in}_{\text{col},1} \cdot C_{2,1} + \text{in}_{\text{col}+1,1} \cdot C_{2,2}$
9 $\text{out}_{\text{col},0} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

10 $r \leftarrow \text{in}_{w-2,0} \cdot C_{0,0} + \text{in}_{w-1,0} \cdot C_{0,1} + \text{in}_{w-1,0} \cdot C_{0,2}$ ▷ *Bottom right corner*
11 $r \leftarrow r + \text{in}_{w-2,0} \cdot C_{1,0} + \text{in}_{w-1,0} \cdot C_{1,1} + \text{in}_{w-1,0} \cdot C_{1,2}$
12 $r \leftarrow r + \text{in}_{w-2,1} \cdot C_{2,0} + \text{in}_{w-1,1} \cdot C_{2,1} + \text{in}_{w-1,1} \cdot C_{2,2}$
13 $\text{out}_{w-1,0} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

▷ *Middle rows*

14 **for** $row \leftarrow 0$ **to** $h - 1$ **do**
15 $r \leftarrow \text{in}_{0,\text{row}-1} \cdot C_{0,0} + \text{in}_{0,\text{row}-1} \cdot C_{0,1} + \text{in}_{1,\text{row}-1} \cdot C_{0,2}$ ▷ *Left side*
16 $r \leftarrow r + \text{in}_{0,\text{row}} \cdot C_{1,0} + \text{in}_{0,\text{row}} \cdot C_{1,1} + \text{in}_{1,\text{row}} \cdot C_{1,2}$
17 $r \leftarrow r + \text{in}_{0,\text{row}+1} \cdot C_{2,0} + \text{in}_{0,\text{row}+1} \cdot C_{2,1} + \text{in}_{1,\text{row}+1} \cdot C_{2,2}$
18 $\text{out}_{0,\text{row}} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`
19 **for** $col \leftarrow 1$ **to** $w - 2$ **do** ▷ *Body*
20 $r \leftarrow \text{in}_{\text{col}-1,\text{row}-1} \cdot C_{0,0} + \text{in}_{\text{col},\text{row}-1} \cdot C_{0,1} + \text{in}_{\text{col}+1,\text{row}-1} \cdot C_{0,2}$
21 $r \leftarrow r + \text{in}_{\text{col}-1,\text{row}} \cdot C_{1,0} + \text{in}_{\text{col},\text{row}} \cdot C_{1,1} + \text{in}_{\text{col}+1,\text{row}} \cdot C_{1,2}$
22 $r \leftarrow r + \text{in}_{\text{col}-1,\text{row}+1} \cdot C_{2,0} + \text{in}_{\text{col},\text{row}+1} \cdot C_{2,1} + \text{in}_{\text{col}+1,\text{row}+1} \cdot C_{2,2}$
23 $\text{out}_{\text{col},\text{row}} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

▷ *Right side*

24 $r \leftarrow \text{in}_{w-2,\text{row}-1} \cdot C_{0,0} + \text{in}_{w-1,\text{row}-1} \cdot C_{0,1} + \text{in}_{w-1,\text{row}-1} \cdot C_{0,2}$
25 $r \leftarrow r + \text{in}_{w-2,\text{row}} \cdot C_{1,0} + \text{in}_{w-1,\text{row}} \cdot C_{1,1} + \text{in}_{w-1,\text{row}} \cdot C_{1,2}$
26 $r \leftarrow r + \text{in}_{w-2,\text{row}+1} \cdot C_{2,0} + \text{in}_{w-1,\text{row}+1} \cdot C_{2,1} + \text{in}_{w-1,\text{row}+1} \cdot C_{2,2}$
27 $\text{out}_{w-1,\text{row}} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

Continued on the following page. ▶

Algorithm 3.6: Sequential pseudocode for the convolution filter (continued)

▷ *Top row*

28 $r \leftarrow \text{in}_{0,h-2} \cdot C_{0,0} + \text{in}_{0,h-2} \cdot C_{0,1} + \text{in}_{1,h-2} \cdot C_{0,2}$ ▷ *Top left corner*

29 $r \leftarrow r + \text{in}_{0,h-1} \cdot C_{1,0} + \text{in}_{0,h-1} \cdot C_{1,1} + \text{in}_{1,h-1} \cdot C_{1,2}$

30 $r \leftarrow r + \text{in}_{0,h-1} \cdot C_{2,0} + \text{in}_{0,h-1} \cdot C_{2,1} + \text{in}_{1,h-1} \cdot C_{2,2}$

31 $\text{out}_{0,h-1} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

32 **for** $col \leftarrow 1$ **to** $w - 2$ **do** ▷ *Top row*

33 $\quad r \leftarrow \text{in}_{\text{column}-1,h-2} \cdot C_{0,0} + \text{in}_{\text{column},h-2} \cdot C_{0,1} + \text{in}_{\text{column}+1,h-2} \cdot C_{0,2}$

34 $\quad r \leftarrow r + \text{in}_{\text{column}-1,h-1} \cdot C_{1,0} + \text{in}_{\text{column},h-1} \cdot C_{1,1} + \text{in}_{\text{column}+1,h-1} \cdot C_{1,2}$

35 $\quad r \leftarrow r + \text{in}_{\text{column}-1,h-1} \cdot C_{2,0} + \text{in}_{\text{column},h-1} \cdot C_{2,1} + \text{in}_{\text{column}+1,h-1} \cdot C_{2,2}$

36 $\quad \text{out}_{\text{column},h-1} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

37 $r \leftarrow \text{in}_{w-2,h-2} \cdot C_{0,0} + \text{in}_{w-1,h-2} \cdot C_{0,1} + \text{in}_{w-1,h-2} \cdot C_{0,2}$ ▷ *Top right corner*

38 $r \leftarrow r + \text{in}_{w-2,h-1} \cdot C_{1,0} + \text{in}_{w-1,h-1} \cdot C_{1,1} + \text{in}_{w-1,h-1} \cdot C_{1,2}$

39 $r \leftarrow r + \text{in}_{w-2,h-1} \cdot C_{2,0} + \text{in}_{w-1,h-1} \cdot C_{2,1} + \text{in}_{w-1,h-1} \cdot C_{2,2}$

40 $\text{out}_{w-1,h-1} \leftarrow$ `Min(`*255,* `Max(`*0,* r`))`

Contrast stretching is a two-stage process. In the first phase a histogram is created by counting the occurrences of each greyscale value. Two cut-off points are calculated from the histogram. A cut-off point is found by starting at one end and moving towards the other while the number of occurrences is below a threshold. The cut-off point is the first greyscale value for which the number of occurrences in the image is larger than the threshold.

The other cut-off point is found by starting at the other end.

In stage two values between the two cut-off points are normalised to the entire greyscale range. Pixels with a value that is below the lower cut-off point are set to 0, and those with a value larger than the higher cut-off point are set to 255.

See Figure 3.8 for an example and Algorithm 3.7 for pseudocode.

3.2 Benchmarking setup

Each implementation of the filters is run multiple times implementations with the input data described in Section 3.2.1. Measurements are taken as described in Section 3.2.2.

3.2.1 Input data

All implementations are tested with a images of varying sizes. Each image has an aspect ratio of 4:3 and is landscape orientated. Before each run, a random image of the appropriate size is generated. See Table 3.2 for a list of image sizes.

Algorithm 3.7: Sequential pseudocode for the enhance contrast filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

▷ *Make histogram*
Initialise histogram values to 0
for $row \leftarrow 0$ **to** $h - 1$ **do**
    for $column \leftarrow 0$ **to** $w - 1$ **do**
        Add 1 to the count of $\text{in}_{column,row}$

▷ *Calculate threshold*
threshold $\leftarrow \frac{h \cdot w}{255}$

▷ *Find the lower cut-off point*
low $\leftarrow 0$
while *histogram value of low* $<$ *threshold* **do**
    Add 1 to low.

▷ *Find the upper cut-off point*
high $\leftarrow 255$
while *histogram value of high* $<$ *threshold* **do**
    Subtract 1 from high.

▷ *Normalise*
for $row \leftarrow 0$ **to** $h - 1$ **do**
    for $column \leftarrow 0$ **to** $w - 1$ **do**
        if $in_{column,row} < low$ **then**
            $\text{out}_{\text{column,row}} \leftarrow 0$
        else if $in_{column,row} > high$ **then**
            $\text{out}_{\text{column,row}} \leftarrow 255$
        else
            $\text{out}_{\text{column,row}} \leftarrow 255 \cdot \frac{\text{in}_{\text{column,row}} - low}{high - low}$

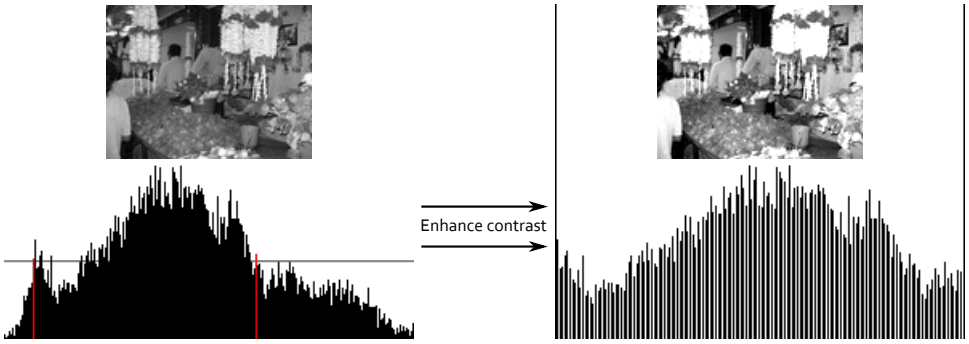


Figure 3.8 – *The enhance contrast operation. Each image has its histogram displayed underneath it. The histogram of the input image has the threshold marked in grey and the cut-off points in red. Note the gaps between bins in the right histogram. These are caused by rounding. Also note that the lines for the outer bins are truncated to save on space.*

Size	**Pixels**	**Bytes**
64 × 48	3 072	3 KiB
128 × 96	12 288	12 KiB
256 × 192	49 152	48 KiB
512 × 384	196 608	192 KiB
768 × 576	442 368	432 KiB
1024 × 768	786 432	768 KiB
1536 × 1152	1 769 472	1.7 MiB
2048 × 1536	3 145 728	3.0 MiB
3072 × 2304	7 077 888	6.8 MiB
4096 × 3072	12 582 912	12 MiB
6144 × 4608	28 311 552	27 MiB
8192 × 6144	50 331 648	48 MiB
12288 × 9216	113 246 208	108 MiB
16384 × 12288	201 326 592	192 MiB

Table 3.2 – *Sizes of the input images*

3.2.2 Measurements

Our measurements are done using wall clock time. For each implementation we have measured the time it takes to calculate the result. This timer—which we call the calculation timer—starts after the input image is generated and before the hardware is initialised. It stops after the result is calculated and stored in the host memory. Because OpenCL platforms take a long time to initialise the hardware and copy the input data into the device memory, we also have a timer in OpenCL implementation that only keeps track of the kernel run times—appropriately called the kernel timer. For OpenCL implementations of the enhance contrast filter we also keep track of separate timers for the histogram phase and the normalisation phase. For that filter we use the term "kernel timer" to mean the timer that measures the normalisation phase, "histogram timer" to measure the histogram phase and "kernels timer" (plural) to measure both.

Chapter 4

Cell SDK Implementations

On the Cell BE we have implemented five versions of each filter. The first, described in Section 4.1, is a reference sequential implementation. The second, described in Section 4.2, is a reference vectorised implementation. The third is a Cell-specific parallel implementation and is described in Section 4.3. The fourth and fifth are common optimisations on the Cell. They are described in Section 4.4 and Section 4.5, respectively.

4.1 Fully sequential

The first implementation is fully sequential, and no parallelism is exploited. This makes it possible to determine at which input size the sharing of work outweighs the increased communication.

The implementation are straightforward translations of the pseudocode shown in Algorithms 3.1 to 3.6. All these algorithms have a basic structure—shown in Algorithm 4.1—where we loop over the columns for each row. The choice to loop over the columns in the inner loop is not arbitrary. The pixels are stored in row-major order. Looping over the columns in the inner loop ensures that we loop over the pixels in the order that they are stored in memory. This causes fewer cache misses than the alternative where we would read the memory in large strides.

Algorithm 4.1: Basic structure of the sequential algorithms

Input : in: image of size $w \times h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 do
3         Perform action on pixel.
```

4.2 PPE vectorised

The PPE has special instructions and registers for vector/SIMD calculations, which are exploited in his implementation exploits that feature. Using the PPE to do SIMD calculations is not a common optimisation, but since one can save on communication between processing elements, there may be some interesting performance benefits.

Vectorised pseudocode for the scale down filter is shown in Algorithm 4.2. Note that, compared with Algorithm 3.1, the inner loop now uses a stride of 16: in the vectorised solution we can work on 16 pixels at the same time.

We first load vectors of pixels (lines 3 to 6), and then shuffle the pixels around (lines 7 to 10). The goal is to create four vectors, each containing one corner of 2×2 squares in the image. For example, $\vec{v}_1$ contains all top-left corners. These vectors can then be entrywise added and divided by 4 (line 11) to calculate the average of each 2×2 square.

Algorithm 4.2: Vectorised pseudocode for the scale down filter

Input : in: image of size $w \times h$
Output : out: image of size $\frac{w}{2} \times \frac{h}{2}$

```
1  for row ← 0 to h/2 − 1 do
2      for column ← 0 to w/2 − 1 stride 16 do
3          u⃗1 ← vector starting at in_{2·column,2·row}
4          u⃗2 ← vector starting at in_{2·column+16,2·row}
5          u⃗3 ← vector starting at in_{2·column,2·row+1}
6          u⃗4 ← vector starting at in_{2·column+16,2·row+1}
7          v⃗1 ← odd elements of u⃗1 and u⃗2
8          v⃗2 ← odd elements of u⃗3 and u⃗4
9          w⃗1 ← even elements of u⃗1 and u⃗2
10         w⃗2 ← even elements of u⃗3 and u⃗4
11         out_{column,row} ← (v⃗1+v⃗2+w⃗1+w⃗2)/4 (entrywise)
```

In Algorithm 4.3 we show vectorised pseudocode for the scale up filter. The goal of the code is to copy a pixel in the input image to a 2×2 square of pixels in the output image.

Like in the scale down filter, we start by loading pixel vectors (lines 1 to 3). We double the pixels in the horizontal direction in lines 4 and 5. By storing each vector twice (in lines 6 to 9), we take care of vertical doubling.

Vectorised pseudocode for mirror vertically is shown in Algorithm 4.4. The only difference to the sequential version (Algorithm 3.3) is that the

Algorithm 4.3: Vectorised pseudocode for the scale up filter

Input : in: image of size $w \times h$
Output : out: image of size $2w \times 2h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 stride 16 do
3         v⃗ ← vector starting at in_column,row
4         w⃗1 ← first 8 elements doubled of v⃗
5         w⃗2 ← last 8 elements doubled of v⃗
6         out_2·column,row ← w⃗1
7         out_2·column,row+1 ← w⃗1
8         out_2·column+16,row ← w⃗2
9         out_2·column+16,row+1 ← w⃗2
```

vectorised version copies 16 pixels at the same time.

Algorithm 4.4: Vectorised pseudocode for the mirror vertically filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

```
1 for row ← 0 to h − 1 do
2     for column ← 0 to w − 1 stride 16 do
3         out_column,h−row−1 ← vector starting at in_column,row
```

Vectorised pseudocode for mirror horizontally is shown in Algorithm 4.5. After loading the pixels (lines 1 to 3) we reverse the vector (line 4) and then stored (line 5).

Vectorised pseudocode for the blend filter is shown in Algorithm 4.6. In lines 1 and 2 we vectorise the blending factors α and $(1 - \alpha)$ and convert them to a `char` (an 8-bit integer data type). This conversion is done by multiplying with 128. We can do more calculations at the same time if we use `char`s. We multiply the blending factors with the pixels of the base image and the overlay image, respectively (lines 8 to 11).

Because the blending factor is a number between 0 and 1, we multiplied it by 128 to store it as a `char`. After multiplication with the input pixels we must divide by 128 to get the correct answer (lines 12 to 17). Once we have done that, we can add the two intermediate results and store the resulting pixels.

Algorithm 4.5: Vectorised pseudocode for the mirror horizontally filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

for *row* $\leftarrow 0$ **to** $h - 1$ **do**
  for *column* $\leftarrow 0$ **to** $w - 1$ **stride** *16* **do**
    $\vec{v} \leftarrow$ vector starting at $\text{in}_{\text{column,row}}$
    $\vec{w} \leftarrow$ reverse of $\vec{v}$
    $\text{out}_{w-\text{column}-16,\text{row}} \leftarrow \vec{w}$

Algorithm 4.6: Vectorised pseudocode for the blend filter

Input : base: image of size $w \times h$
  overlay: image of size $w_s \times h_s$
  x: $1 \leq x < w - w_s$, the x-coordinate to start blending
  y: $1 \leq y < h - h_s$, the y-coordinate to start blending
  α: $0 \leq \alpha \leq 1$, the blending factor
Output : out: image of size $w \times h$

$\vec{\alpha} \leftarrow$ vector with all elements set to `Round`$(128 \cdot \alpha)$
$\vec{\alpha}' \leftarrow$ vector with all elements set to `Round`$(128 \cdot (1 - \alpha))$
for *row* $\leftarrow 0$ **to** $h - 1$ **do**
  for *column* $\leftarrow 0$ **to** $w - 1$ **stride** *16* **do**
    if $x \leq \mathit{column} < x + w_s$ **and** $y \leq \mathit{row} < y + h_s$ **then**
      $\vec{b} \leftarrow$ vector starting at $\text{base}_{\text{column,row}}$
      $\vec{s} \leftarrow$ vector starting at $\text{overlay}_{\text{column}-x,\text{row}-y}$
      $\vec{b}'_1 \leftarrow$ multiply even elements of $\vec{b}$ and $\vec{\alpha}'$
      $\vec{b}'_2 \leftarrow$ multiply odd elements of $\vec{b}$ and $\vec{\alpha}'$
      $\vec{s}'_1 \leftarrow$ multiply even elements of $\vec{s}$ and $\vec{\alpha}$
      $\vec{s}'_2 \leftarrow$ multiply odd elements of $\vec{s}$ and $\vec{\alpha}$
      $\vec{b}'_1 \leftarrow \frac{\vec{b}'_1}{128}$ (entrywise)
      $\vec{b}'_2 \leftarrow \frac{\vec{b}'_2}{128}$ (entrywise)
      $\vec{s}'_1 \leftarrow \frac{\vec{s}'_1}{128}$ (entrywise)
      $\vec{s}'_2 \leftarrow \frac{\vec{s}'_2}{128}$ (entrywise)
      $\vec{b}' \leftarrow$ combine $\vec{b}'_1$ and $\vec{b}'_2$
      $\vec{s}' \leftarrow$ combine $\vec{s}'_1$ and $\vec{s}'_2$
      $\text{out}_{\text{column,row}} \leftarrow \vec{b}' + \vec{s}'$
    else
      $\text{out}_{\text{column,row}} \leftarrow$ vector starting at $\text{base}_{\text{column,row}}$

Vectorised pseudocode for the convolution filter is shown in Algorithm 4.7. The convolution filter calculates for each pixel the entrywise product of the 3×3 convolution kernel and the 3×3 area around a pixel. The entries of the resulting matrix are then added to calculate the output pixel.

To calculate 16 output pixels in one go, we multiply each pixel first with the left entry of the convolution kernel (lines 13, 19 and 25), then with the centre entry (lines 15, 21 and 27) and finally the right entry (lines 17, 23 and 29). Because this does not work properly at the edges of the vector, we use the pixel before the vectors (lines 12, 18 and 24) and the pixel after the vectors (lines 16, 22 and 28) when we need it. We then rotate the vectors (lines 30 and 31), before we add them together (line 32) to calculate the result.

Algorithm 4.8 contains vectorised pseudocode for the enhance contrast filter. Because we cannot gain enough precision with integer arithmetic, we need to use floating point arithmetic. We therefore need to convert our data from `char` vectors to `float` vectors. This conversion cannot be done directly. We have to convert the `char` vector to four `long` vectors first (lines 17 to 20). We then convert those vectors to `float` vectors (line 21). The real calculation is done in lines 22 to 24. We can then convert the `float` vectors back to a `char` vector, again using `long` vectors as an intermediate representation (lines 25 and 26). Finally, we store the result in the output image (line 27).

4.3 SPEs

The SPEs are major assets in the quest for performance gain on the Cell. The SPEs are accelerator cores designed for vector processing. These implementations are data parallel implementations of the original filters: the input data is split in as many equal pieces as there are SPEs available. These pieces are handed to the SPEs. Each SPE uses a simple serial algorithm to do its share of the work.

For most filters the splitting is very straightforward: each piece starts where the previous piece ends. As one of two exceptions, the convolution filter needs an extra line of context before and after each piece. The other exception is the blend filter, which has two input images. The overlay image may be smaller than the base image. The pieces are split according to the base image. This means that some SPEs could get no piece—or just a smaller piece—of the overlay image.

Each SPE copies a part of its piece to its Local Store using DMA transfers. After issuing a transfer command, it waits for the transfer to complete. It then proceeds to compute the result and issues a DMA write command, for which it then waits before continuing to copy the next part of its piece.

Algorithm 4.7: Vectorised pseudocode for the convolution filter

Input : in: image of size $w \times h$
C: 3×3 convolution kernel
Output : out: image of size $w \times h$

1 **for** *row* $\leftarrow 0$ **to** $h - 1$ **do**
2 $\quad \text{prev}_0 \leftarrow \text{in}_{0,\texttt{Max(}j\text{-}1,\ 0\texttt{)}}$
3 $\quad \text{prev}_1 \leftarrow \text{in}_{0,j}$
4 $\quad \text{prev}_2 \leftarrow \text{in}_{0,\texttt{Min(}j+1,\ h\text{-}1\texttt{)}}$
5 $\quad$ **for** *column* $\leftarrow 0$ **to** $w - 1$ **stride** *16* **do**
6 $\quad\quad \text{row}_0 \leftarrow$ vector starting at $\text{in}_{i,\texttt{Max(}j\text{-}1,\ 0\texttt{)}}$
7 $\quad\quad \text{row}_1 \leftarrow$ vector starting at $\text{in}_{i,j}$
8 $\quad\quad \text{row}_2 \leftarrow$ vector starting at $\text{in}_{i,\texttt{Min(}j+1,\ h\text{-}1\texttt{)}}$
9 $\quad\quad \text{next}_0 \leftarrow \text{in}_{\texttt{Min(}i\ +\ 16,\ w\ -\ 1\texttt{),Max(}j\text{-}1,\ 0\texttt{)}}$
10 $\quad\quad \text{next}_1 \leftarrow \text{in}_{\texttt{Min(}i\ +\ 16,\ w\ -\ 1\texttt{)},j}$
11 $\quad\quad \text{next}_2 \leftarrow \text{in}_{\texttt{Min(}i\ +\ 16,\ w\ -\ 1\texttt{),Min(}j+1,\ h\text{-}1\texttt{)}}$
12 $\quad\quad \texttt{Swap(}\mathit{prev}_0,\ \mathit{row}_{0,15}\texttt{)}$
13 $\quad\quad T_{0,0} \leftarrow \text{row}_0 \cdot C_{0,0}$ (entrywise)
14 $\quad\quad \text{row}_{0,15} \leftarrow \text{prev}_0$
15 $\quad\quad T_{0,1} \leftarrow \text{row}_0 \cdot C_{0,1}$ (entrywise)
16 $\quad\quad \texttt{Swap(}\mathit{next}_0,\ \mathit{row}_{0,0}\texttt{)}$
17 $\quad\quad T_{0,2} \leftarrow \text{row}_0 \cdot C_{0,2}$ (entrywise)
18 $\quad\quad \texttt{Swap(}\mathit{prev}_1,\ \mathit{row}_{1,15}\texttt{)}$
19 $\quad\quad T_{1,0} \leftarrow \text{row}_1 \cdot C_{1,0}$ (entrywise)
20 $\quad\quad \text{row}_{1,15} \leftarrow \text{prev}_1$
21 $\quad\quad T_{1,1} \leftarrow \text{row}_1 \cdot C_{1,1}$ (entrywise)
22 $\quad\quad \texttt{Swap(}\mathit{next}_1,\ \mathit{row}_{1,0}\texttt{)}$
23 $\quad\quad T_{1,2} \leftarrow \text{row}_1 \cdot C_{1,2}$ (entrywise)
24 $\quad\quad \texttt{Swap(}\mathit{prev}_2,\ \mathit{row}_{2,15}\texttt{)}$
25 $\quad\quad T_{2,0} \leftarrow \text{row}_2 \cdot C_{2,0}$ (entrywise)
26 $\quad\quad \text{row}_{0,15} \leftarrow \text{prev}_2$
27 $\quad\quad T_{2,1} \leftarrow \text{row}_2 \cdot C_{2,1}$ (entrywise)
28 $\quad\quad \texttt{Swap(}\mathit{next}_2,\ \mathit{row}_{2,0}\texttt{)}$
29 $\quad\quad T_{2,2} \leftarrow \text{row}_2 \cdot C_{2,2}$ (entrywise)
30 $\quad\quad$ Rotate $T_{0,0}$, $T_{1,0}$ and $T_{2,0}$ 1 position to the right
31 $\quad\quad$ Rotate $T_{0,2}$, $T_{1,2}$ and $T_{2,2}$ 1 position to the left
32 $\quad\quad O \leftarrow T_{0,0} + T_{0,1} + T_{0,2} + T_{1,0} + T_{1,1} + T_{1,2} + T_{2,0} + T_{2,1} + T_{2,2}$ (entrywise)
33 $\quad\quad \text{out}_{i,j} \leftarrow \texttt{Max(}0,\ \texttt{Min(}255,\ O\texttt{))}$ (entrywise)

Algorithm 4.8: Vectorised pseudocode for the enhance contrast filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

▷ *Make histogram*
Initialise histogram values to 0
for $row \leftarrow 0$ **to** $h - 1$ **do**
    for $column \leftarrow 0$ **to** $w - 1$ **do**
        Add 1 to the count of $\text{in}_{column,row}$

▷ *Calculate threshold*
threshold $\leftarrow \frac{h \cdot w}{255}$

▷ *Find the lower cut-off point*
low $\leftarrow 0$
while *histogram value of low* $<$ *threshold* **do**
    Add 1 to low.

▷ *Find the upper cut-off point*
high $\leftarrow 255$
while *histogram value of high* $<$ *threshold* **do**
    Subtract 1 from high.

▷ *Normalise*
$\vec{a} \leftarrow$ `float` vector with all elements set to $\frac{255}{\text{high}-\text{low}}$
$\vec{b} \leftarrow$ `float` vector with all elements set to high
for $row \leftarrow 0$ **to** $h - 1$ **do**
    for $column \leftarrow 0$ **to** $w - 1$ **stride** *16* **do**
        $\vec{v} \leftarrow$ vector starting at $\text{in}_{\text{column,row}}$
        $\vec{q_1} \leftarrow$ first 4 elements of $\vec{v}$ converted to `long`
        $\vec{q_2} \leftarrow$ second 4 elements of $\vec{v}$ converted to `long`
        $\vec{q_3} \leftarrow$ third 4 elements of $\vec{v}$ converted to `long`
        $\vec{q_4} \leftarrow$ last 4 elements of $\vec{v}$ converted to `long`
        Convert $\vec{q_1}$, $\vec{q_2}$, $\vec{q_3}$ and $\vec{q_4}$ to `float` vectors
        Subtract $\vec{b}$ from $\vec{q_1}$, $\vec{q_2}$, $\vec{q_3}$ and $\vec{q_4}$
        Multiply $\vec{q_1}$, $\vec{q_2}$, $\vec{q_3}$ and $\vec{q_4}$ with $\vec{a}$
        Make sure $\vec{q_1}$, $\vec{q_2}$, $\vec{q_3}$ and $\vec{q_4}$ have all values between 0 and 255
        Convert $\vec{q_1}$, $\vec{q_2}$, $\vec{q_3}$ and $\vec{q_4}$ back to `long` vectors
        $\vec{v} \leftarrow$ recombination of $\vec{q_1}, \vec{q_2}, \vec{q_3}$ and $\vec{q_4}$
        $\text{out}_{\text{column,row}} \leftarrow \vec{v}$

4.4 SPEs double buffered

The SPE can do calculations and memory transfers in parallel. The memory transfers are handled by the Memory Flow Controller while the SPU does other work. This parallelism can be exploited with a technique called double buffering.

As the name implies, a double buffered algorithm maintains twice as many buffers as a simpler algorithm. For most filters that means two input and two result buffers. After requesting the data for the first part, it requests the second part of its piece of the data. This second part is stored in a different buffer. Only after requesting that part will it wait for the transfer of the first part to complete. It will not wait for the second part to complete, but instead will proceed to compute the result for the first part. The result is stored in the first result buffer. The algorithm then issues a write request for that buffer, followed by a read request for the third part. Going into the next iteration, it will then wait for the transfer of the second part to complete. This will go on for the next parts. Before starting to compute the result, the SPE will wait until the result buffer it is about to use is available, i.e. until the write request of the result in the buffer is finished.

The convolution filter adds a little twist to the algorithm because the parts overlap. Each part's last line is used in the next part. To prevent lines from having to be transferred twice, we transfer two parts at the start and store them in consecutive buffers. We also request the third part as part of the double buffering algorithm. Once we are done with the first part, we copy the last line of the first part, and the entire second part to a buffer directly before the third part (and we request the fourth part). Once we are done with the second part, we do the same with the third part.

4.5 SPEs vectorised

The SPEs support vector/SIMD instructions and registers similar to the PPE. The algorithms used on the PPE can be reused on the SPEs. The only changes that need to be made are the names of the intrinsics, as their functionality is similar enough.

These implementations are double buffered as well, and are therefore the most optimised implementations we have tested using the Cell SDK.

4.6 Results

4.6.1 Fully sequential

The benchmarks for the sequential implementation are shown in Figures A.1 to A.7. The relation between the input size and the run-time is linear, as is expected with the used algorithm. Note that the PlayStation 3 results do

not include the larger image sizes. The PlayStation 3 only has 256 MiB of memory, which is not enough to store both input and output data for those sizes. The QS22 results for the scale up and scale down also lack the higher sizes. This is because the algorithm on the SPEs cannot deal with them, making it pointless to compare them with a sequential version.

4.6.2 PPE vectorised

The results for the vectorised implementation are compared to the sequential version in Figures A.8 to A.14. These results show that vectorisation is an improvement for all filters on both platforms. In general we can calculate 16 pixels at the same time, which would lead us to expect a 16-fold speedup. In reality we reach speedups as low as 2.5 times (for the convolution kernel) and as high as 35.6 times (for the blend kernel). When running the blend filter we saved many operations, which caused the big speedup. In the convolution filter we have to reorder the pixels quite a lot before they are in an order where we can use them efficiently.

4.6.3 SPEs

Next, we have enabled our kernels to use the SPEs as accelerators, and we benchmarked the new versions. The plots in Figures A.15 to A.21 show that using the SPEs only pays off, performance-wise, for the large(st) image sizes. For smaller sizes the calculation time is overshadowed by the time required to initialise the SPEs. For the blend filter and the enhance contrast filter, using the SPEs does not bring any performance improvement, because these are already quite fast on the PPE.

Note that the plots are made using the maximum number of SPEs available on each system: 16 for the QS22 and 6 for the PlayStation 3. In Figures A.22 to A.28 we can see that using fewer SPEs leads to shorter initialisation time and therefore for better run-times (i.e. improved performance) for smaller image sizes. We note that for each of the kernels, there is an "inflection point"—i.e., a certain image size—at which larger images will be computed faster by the SPE-enabled versions become faster, and smaller images will be computed faster by the vectorised PPE ones. This essentially means that one could implement a solution that can switch the execution between the PPE and the SPEs, depending on the size of the input image. Such a solution would not only give better overall execution time, but it will also provide better resource utilisation. Therefore, we consider this idea an interesting future work direction towards auto-tuning.

Note that the inflection points differ per kernel and system, but is never below 768×576. For the QS22 it is usually a little bigger than for the PlayStation 3, because the former uses more SPEs and therefore spends more time setting up and starting SPEs.

4.6.4 SPEs double buffered

The next optimisation we have applied is double buffering. We have benchmarked the new versions of the filters, and our results (see Figures A.29 to A.35) show virtually no performance improvement for the double buffered version. This result is not surprising, as double buffering works by overlapping memory accesses with computation, and these kernels do not have enough computation for this overlap.

4.6.5 SPEs vectorised

Finally, we have benchmarked the SPE vectorised version. Figures A.36 to A.42 show a comparison between the double buffered implementation and the SPEs vectorised implementation. The results show important performance improvements for the large image sizes, and only minor changes for the small and medium ones. The effect is relatively minor for the scale down filter (1.3 times) and the mirror filters (1.9 times for both vertically and horizontally), but very big for the blend (21.4 times) and the enhance contrast filter (58.9 times). While the numbers are more extreme than for the vectorised PPE, the general trend is the same. Given that vectorisation infers no performance penalty for small images and leads to an important increase for large images, we consider it a successful optimisation.

4.6.6 PPE and SPEs

Finally, we have compared the vectorised versions running on the PPE and on the SPEs. Vectorisation made a big difference on both the PPE and the SPEs. Figures A.43 to A.49 compare these. For smaller sizes the initialisation time for the SPEs makes the PPE outperform the SPEs. For larger sizes the additional computing power of the SPEs makes them outperform the PPE. Specifically, the SPE performance gain for the filters ranges between 2.5 and 13.8 times. We note that there is a similar inflection point seen in the PPE-SPE comparison in Section 4.6.3: for small images, the filters should run on the PPE, and for larger images should be processed using the SPEs. Note that the inflection point has shifted to the left for most kernels.

4.6.7 Summary

Overall, our results with the Cell SDK show the following:

- The reference sequential versions of the kernels, benchmarked on the PPE, show a linear increase of the execution time with the increase of the input image size.
- Vectorising the PPE code improves performance significantly (between 2.5 and 36 times).

- Enabling the use of SPEs for these kernels only pays off for large image sizes; the size at which the SPEs overtake the PPE in terms of performance differs largely per kernel.
- The impact of the traditional low-level SPE code optimisations varies: double buffering had very little impact, while vectorisation helped a lot (improving performance up to 59 times).
- The performance on the PlayStation 3 is comparable with the performance on the QS22 for the PPE and equal number of SPEs (6); QS22 performs better for large images and using all its 16 SPEs.

Finally, we note that there is a "performance balance point" for each kernel: before it, the PPE version performs better, and after it the SPE-enabled version performs better. Essentially, these results show that the Cell BE is a good solution for improving the performance of such filters for large images, but it is most likely "overkill" for small and medium-sized images.

Chapter 5

OpenCL Implementations

We have implemented seven versions of each filter, and one more for the enhance contrast filter. In this chapter we discuss these implementations. We start with a reference sequential implementation in Section 5.1, followed by a naive parallelisation in Section 5.2. We discuss two versions in Section 5.3 where we experimented with different data types. In Section 5.4, we discuss parallelisation of the histogram code used for the enhance contrast filter. In Section 5.5, we look at memory optimisations. Section 5.6 discusses vectorisation and Section 5.7 discusses an optimisation called "pinned memory". Finally, in Section 5.8 we show results.

5.1 Fully sequential

This is exactly the same implementation as described in Section 4.1. We have made no changes to the code, but simply compiled it for the other machines. We can use this implementation to determine the input size at which the sharing of work outweighs the increased communication.

5.2 Naive implementation using `char`

The first implementation that uses OpenCL is a naive one. The inner loops of the sequential algorithms are translated to OpenCL programs. We start one OpenCL thread per pixel, i.e., each pixel becomes a work-item. The threads are grouped into work-groups. Each work-group calculates one row of the output, unless the maximum size of a work-group is smaller than the number of pixels per row. In that case, the maximum number of work-items per work-group is used.

Each pixel is represented by an `unsigned char`. This is an 8-bit unsigned integer data type which can store values in the range 0 to 255. It is the data format used by the BMP library.

Algorithm 5.1: Pseudocode for the OpenCL scale down filter

Input : in: image of size $w \times h$
Output : out: image of size $\frac{w}{2} \times \frac{h}{2}$

1 idx $\leftarrow$ `get_global_id()`
2 column $\leftarrow$ `Remainder(`*idx,* $\frac{w}{2}$`)`
3 row $\leftarrow \frac{\text{idx}-\text{column}}{\frac{w}{2}}$
4 sum $\leftarrow \text{in}_{2\cdot\text{column},2\cdot\text{row}}$
5 sum $\leftarrow$ sum $+ \text{in}_{2\cdot\text{column}+1,2\cdot\text{row}}$
6 sum $\leftarrow$ sum $+ \text{in}_{2\cdot\text{column},2\cdot\text{row}+1}$
7 sum $\leftarrow$ sum $+ \text{in}_{2\cdot\text{column}+1,2\cdot\text{row}+1}$
8 $\text{out}_{\text{column,row}} \leftarrow \frac{\text{sum}}{4}$

The code shown in Algorithm 5.1 runs for every pixel in the output image. It is very similar to the inner loop of Algorithm 3.1. The differences are in lines 1 to 3.

Each thread has a unique global identifier between 0 and the total number of threads. This identifier is stored in the variable idx in line 1. This identifier is used in line 2 and 3 to calculate the row and column of the pixel calculated by this thread. `Remainder(`a`,` b`)` is the remainder of the division $\frac{a}{b}$.

Threads also have a local identifier and a work-group identifier. The local identifier is a number between 0 and the work-group size and is unique for all threads in the same work-group. The work-group identifier is a number between 0 and the number of work-groups. Each thread in the same work-group has the same work-group identifier. For smaller images, where each row is handled by one work-group, these two identifiers could be used to find the row and column numbers. However, larger images require multiple work-groups per row, which is why we use the global identifier instead.

Algorithm 5.2: Pseudocode for the OpenCL scale up filter

Input : in: image of size $w \times h$
Output : out: image of size $2w \times 2h$

1 idx $\leftarrow$ `get_global_id()`
2 column $\leftarrow$ `Remainder(`*idx,* w`)`
3 row $\leftarrow \frac{\text{idx}-\text{column}}{w}$
4 $\text{out}_{2\cdot\text{column},2\cdot\text{row}} \leftarrow \text{in}_{\text{column,row}}$
5 $\text{out}_{2\cdot\text{column}+1,2\cdot\text{row}} \leftarrow \text{in}_{\text{column,row}}$
6 $\text{out}_{2\cdot\text{column},2\cdot\text{row}+1} \leftarrow \text{in}_{\text{column,row}}$
7 $\text{out}_{2\cdot\text{column}+1,2\cdot\text{row}+1} \leftarrow \text{in}_{\text{column,row}}$

The algorithm shown in Algorithm 5.2 runs for every pixel in the input image. Like the scale down algorithm, it is very similar to the inner loop of its sequential version, shown in Algorithm 3.2. Lines 1 to 3 are added to calculate the row and column numbers from the global identifier. The calculation is almost identical to the one in the Algorithm 5.1. The only difference is that it uses the width of the input image, where the scale down filter uses the width of output image.

The same three lines are added to the inner loops in Algorithms 3.3 to 3.5 to create Algorithms 5.3 to 5.5.

Algorithm 5.3: Pseudocode for the OpenCL mirror vertically filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

1 idx $\leftarrow$ `get_global_id()`
2 column $\leftarrow$ `Remainder(`*idx, w*`)`
3 row $\leftarrow \frac{\text{idx}-\text{column}}{w}$
4 $\text{out}_{\text{column},h-\text{row}-1} \leftarrow \text{in}_{\text{column,row}}$

Algorithm 5.4: Pseudocode for the OpenCL mirror horizontally filter

Input : in: image of size $w \times h$
Output : out: image of size $w \times h$

1 idx $\leftarrow$ `get_global_id()`
2 column $\leftarrow$ `Remainder(`*idx, w*`)`
3 row $\leftarrow \frac{\text{idx}-\text{column}}{w}$
4 $\text{out}_{w-\text{column}-1,\text{row}} \leftarrow \text{in}_{\text{column,row}}$

Algorithm 5.6 shows pseudocode for the convolution filter. The sequential convolution algorithm described in Algorithm 3.6 deals with the edge cases in different stages of the algorithm. It first calculates the bottom left corner, then the non-corner pixels on the bottom row, et cetera.

The OpenCL implementation calculates all pixels at the same time. It does so in one thread per pixel. The thread must therefore check if the pixel is an edge case and if so, calculate it differently. These checks are done in lines 4, 9, 14, 19, 24, 29, 34 and 39. If the pixel is not an edge case, the else clause starting on line 44 is executed.

The enhance contrast filter works in two stages. The first stage calculates normalisation cut-off points from a histogram. The second stage normalises the image. The first stage is shown in Algorithm 5.7. It is a sequential

Algorithm 5.5: Pseudocode for the OpenCL blend filter

Input : base: image of size $w \times h$
overlay: image of size $w_s \times h_s$
x: $1 \leq x < w - w_s$, the x-coordinate to start blending
y: $1 \leq y < h - h_s$, the y-coordinate to start blending
α: $0 \leq \alpha \leq 1$, the blending factor
Output : out: image of size $w \times h$

```
1 idx ← get_global_id()
2 column ← Remainder(idx, w)
3 row ← (idx − column) / w
4 if x ≤ column < x + w_s and y ≤ row < y + h_s then
5 |   out_column,row ← (1 − α) · base_column,row + α · overlay_column−x,row−y
6 else
7 |   out_column,row ← base_column,row
```

implementation in OpenCL. Parallelisation is discussed in Section 5.4. The second stage is shown in Algorithm 5.8. The row and column are calculated from the global identifier using the method described above. The rest of the second stage consists of the inner loop of the normalisation part of Algorithm 3.7.

The histogram stage is run sequentially, while the normalisation stage is run in parallel with one pixel per thread. Threads are grouped in many work-groups. OpenCL provides no way to synchronise data between work-groups, other than to return from the kernel and starting a new kernel. The low and high variables need to be communicated to the normalisation stage, so synchronisation is required. Both stages are therefore run as separate kernels.

There is no need for data transfers from device memory to host memory between the stages. The low and high variables can be stored in global memory and then accessed again by the second stage. Similarly, it is not necessary to copy the input image to the device again. It can be reused.

5.3 Naive implementation using other data types

The previous implementation used `unsigned char`s to store pixels. We experimented with two other data types to see if some systems behaved differently when using those. The two data types we selected were `float`s—a 32-bit floating point data type—and `unsigned int`s—a 32-bit unsigned integer data type, capable of storing values in the range 0 to 4 294 967 295. Both these data types use four times as much space as the `unsigned char`s do. Neither the precision offered by `float`s, nor the larger range offered by

Algorithm 5.6: Pseudocode for the OpenCL convolution filter

Input : in: image of size $w \times h$
C: 3×3 convolution kernel
Output : out: image of size $w \times h$

```
1  idx ← get_global_id()
2  col ← Remainder(idx, w)
3  row ← (idx−col)/w
4  if row = 0 and col = 0 then                                   ▷ Bottom left corner
5      r ← in_{0,0} · C_{0,0} + in_{0,0} · C_{0,1} + in_{1,0} · C_{0,2}
6      r ← r + in_{0,0} · C_{1,0} + in_{0,0} · C_{1,1} + in_{1,0} · C_{1,2}
7      r ← r + in_{0,1} · C_{2,0} + in_{0,1} · C_{2,1} + in_{1,1} · C_{2,2}
8      out_{0,0} ← Min(255, Max(0, r))
9  else if row = 0 and col = w − 1 then                          ▷ Bottom right corner
10     r ← in_{w−2,0} · C_{0,0} + in_{w−1,0} · C_{0,1} + in_{w−1,0} · C_{0,2}
11     r ← r + in_{w−2,0} · C_{1,0} + in_{w−1,0} · C_{1,1} + in_{w−1,0} · C_{1,2}
12     r ← r + in_{w−2,1} · C_{2,0} + in_{w−1,1} · C_{2,1} + in_{w−1,1} · C_{2,2}
13     out_{w−1,0} ← Min(255, Max(0, r))
14 else if row = 0 then                                          ▷ Bottom row
15     r ← in_{col−1,0} · C_{0,0} + in_{col,0} · C_{0,1} + in_{col+1,0} · C_{0,2}
16     r ← r + in_{col−1,0} · C_{1,0} + in_{col,0} · C_{1,1} + in_{col+1,0} · C_{1,2}
17     r ← r + in_{col−1,1} · C_{2,0} + in_{col,1} · C_{2,1} + in_{col+1,1} · C_{2,2}
18     out_{col,0} ← Min(255, Max(0, r))
19 else if row = h − 1 and col = 0 then                          ▷ Top left corner
20     r ← in_{0,h−2} · C_{0,0} + in_{0,h−2} · C_{0,1} + in_{1,h−2} · C_{0,2}
21     r ← r + in_{0,h−1} · C_{1,0} + in_{0,h−1} · C_{1,1} + in_{1,h−1} · C_{1,2}
22     r ← r + in_{0,h−1} · C_{2,0} + in_{0,h−1} · C_{2,1} + in_{1,h−1} · C_{2,2}
23     out_{0,h−1} ← Min(255, Max(0, r))
24 else if row = h − 1 and col = w − 1 then                      ▷ Top right corner
25     r ← in_{w−2,h−2} · C_{0,0} + in_{w−1,h−2} · C_{0,1} + in_{w−1,h−2} · C_{0,2}
26     r ← r + in_{w−2,h−1} · C_{1,0} + in_{w−1,h−1} · C_{1,1} + in_{w−1,h−1} · C_{1,2}
27     r ← r + in_{w−2,h−1} · C_{2,0} + in_{w−1,h−1} · C_{2,1} + in_{w−1,h−1} · C_{2,2}
28     out_{w−1,h−1} ← Min(255, Max(0, r))
29 else if row = h − 1 then                                      ▷ Top row
30     r ← in_{col−1,h−2} · C_{0,0} + in_{col,h−2} · C_{0,1} + in_{col+1,h−2} · C_{0,2}
31     r ← r + in_{col−1,h−1} · C_{1,0} + in_{col,h−1} · C_{1,1} + in_{col+1,h−1} · C_{1,2}
32     r ← r + in_{col−1,h−1} · C_{2,0} + in_{col,h−1} · C_{2,1} + in_{col+1,h−1} · C_{2,2}
33     out_{col,h−1} ← Min(255, Max(0, r))
```

Continued on the next page. ▶

Algorithm 5.6: Pseudocode for the OpenCL convolution filter (continued)

```
34 else if col = 0 then                                          ▷ Left side
35     r ← in_{0,row−1} · C_{0,0} + in_{0,row−1} · C_{0,1} + in_{1,row−1} · C_{0,2}
36     r ← r + in_{0,row} · C_{1,0} + in_{0,row} · C_{1,1} + in_{1,row} · C_{1,2}
37     r ← r + in_{0,row+1} · C_{2,0} + in_{0,row+1} · C_{2,1} + in_{1,row+1} · C_{2,2}
38     out_{0,row} ← Min(255, Max(0, r))
39 else if col = w − 1 then                                      ▷ Right side
40     r ← in_{w−2,row−1} · C_{0,0} + in_{w−1,row−1} · C_{0,1} + in_{w−1,row−1} · C_{0,2}
41     r ← r + in_{w−2,row} · C_{1,0} + in_{w−1,row} · C_{1,1} + in_{w−1,row} · C_{1,2}
42     r ← r + in_{w−2,row+1} · C_{2,0} + in_{w−1,row+1} · C_{2,1} + in_{w−1,row+1} · C_{2,2}
43     out_{w−1,row} ← Min(255, Max(0, r))
44 else                                                          ▷ Body
45     r ← in_{col−1,row−1} · C_{0,0} + in_{col,row−1} · C_{0,1} + in_{col+1,row−1} · C_{0,2}
46     r ← r + in_{col−1,row} · C_{1,0} + in_{col,row} · C_{1,1} + in_{col+1,row} · C_{1,2}
47     r ← r + in_{col−1,row+1} · C_{2,0} + in_{col,row+1} · C_{2,1} + in_{col+1,row+1} · C_{2,2}
48     out_{col,row} ← Min(255, Max(0, r))
```

Algorithm 5.7: Sequential pseudocode to find normalisation cut-off points

Input : in: image of size $w \times h$
Output : low: integer, $0 \leq \text{low} \leq \text{high}$, the lower cut-off point
high: integer $\text{low} \leq \text{high} \leq 255$, the upper cut-off point

```
    ▷ Make histogram
1   Initialise histogram values to 0
2   for row ← 0 to h − 1 do
3       for column ← 0 to w − 1 do
4           Add 1 to the count of in_{column,row}
    ▷ Calculate threshold
5   threshold ← (h·w)/255
    ▷ Find the lower cut-off point
6   low ← 0
7   while histogram value of low < threshold do
8       Add 1 to low.
    ▷ Find the upper cut-off point
9   high ← 255
10  while histogram value of high < threshold do
11      Subtract 1 from high.
```

Algorithm 5.8: Pseudocode for the OpenCL enhance contrast filter

Input : in: image of size $w \times h$
low: integer, $0 \leq \text{low} \leq \text{high}$, the lower cut-off point
high: integer $\text{low} \leq \text{high} \leq 255$, the upper cut-off point
Output : out: image of size $w \times h$

```
1 idx ← get_global_id()
2 column ← Remainder(idx, w)
3 row ← (idx−column)/w
4 if in_{column,row} < low then
5 |   out_{column,row} ← 0
6 else if in_{column,row} > high then
7 |   out_{column,row} ← 255
8 else
9 |   out_{column,row} ← 255 · (in_{column,row} − low)/(high − low)
```

`unsigned int`s is used. Pixels only hold integer values in the 0 to 255 range.

The data is still generated or read as `unsigned char`s, but is immediately converted to the used data type. The conversion is not included in the calculation and kernel timers.

The extra memory used by these data types causes some devices to run out of memory on smaller images than they would if we used `unsigned char`s. Because of this, and because the `unsigned char` data type is used by the BMP library, all our following implementations use that data type.

5.4 Parallel histogram

The histogram implementation shown in Algorithm 5.7 is sequential, but it is possible to use more threads to achieve a parallel one. The solution shown in Algorithm 5.9 uses multiple threads in one work-group.

The histogram algorithm has three different stages. The first two calculate a histogram of the image. The third stage finds the cut-off points. In stage one each thread builds a histogram of a part of the image. These histograms are stored in local memory so they can be accessed by other threads in the next stage. Unfortunately, local memory is limited. Not all threads can therefore build a histogram. The number of threads that build a histogram is calculated (in line 1) of Algorithm 5.9 as the ratio between the available local memory and the amount of memory one histogram requires. The thread histograms are build in lines 4 to 7. Threads read the pixels from the image in strides equal to the number of active threads. This improves memory coalescing by making adjacent threads read adjacent pixels.

In the second stage all thread histograms are merged into one complete

Algorithm 5.9: Parallel pseudocode to find normalisation cut-off points

Input : in: image of size $w \times h$
memSize: the amount of available local memory
Output : cutoffPoints$_0$: integer, $0 \leq$ low $\leq$ high, the lower cut-off point
cutoffPoints$_1$: integer low $\leq$ high ≤ 255, the upper cut-off point

▷ *Find number of active threads due to memory concerns*
1 nrThreads $\leftarrow \lfloor \frac{\text{memSize}}{\text{memory used by one histogram}} \rfloor$

▷ *Make thread histograms*
2 local_idx $\leftarrow$ `get_local_id()`
3 **if** *local_idx* $<$ *nrThreads* **then**
4 Initialise thread histogram values to 0
5 **for** *row* $\leftarrow 0$ **to** $h - 1$ **do**
6 **for** *column* $\leftarrow$ *local_idx* **to** $w - 1$ **stride** *nrThreads* **do**
7 Add 1 to the count of $\text{in}_{column,row}$
8 **barrier**

▷ *Merge histograms*
9 **for** $i \leftarrow$ *local_idx* **to** 256 **stride** *work-group size* **do**
10 $\text{histogram}_i = 0$
11 **for** $j \leftarrow 0$ **to** *nrThreads* **do**
12 $\text{histogram}_i = \text{histogram}_i + \text{thread}_j.\text{histogram}_i$

▷ *Calculate threshold*
13 threshold $\leftarrow \frac{h \cdot w}{255}$
14 **barrier**

▷ *Find cut-off points*
15 **if** *local_idx* < 2 **then**
16 $i = 255 \cdot \text{local_idx}$
17 **while** $\textit{histogram}_i < \textit{threshold}$ **do**
18 $i \leftarrow i + (\text{local_idx} = 0) - (\text{local_idx} \neq 0)$
19 $\text{cutoffPoints}_{\text{local_idx}} = i$

histogram. This is done by the first 256 threads (or all threads if there are fewer). Each thread sums a single bin of all histograms. If there are fewer than 256 threads, each thread will do this for multiple bins, separated by a stride equal to the work-group size (see the for-statement in line 9, and the summing done in lines 10 to 12).

The third stage (lines 15 to 19) is executed by only two threads: those with local_idx = 0 or local_idx = 1. Each thread calculates one cut-off point: thread 0 searches from the lower bin (bin 0) upwards, while thread 1 searches from the top bin (bin 255) downwards. Note that to do so in parallel for both threads, we use arithmetic evaluation of logical conditions (in line 18).

Between stages, threads synchronise with barriers. The first barrier (line 8) makes sure that all thread histograms are complete before merging; the second barrier (line 14) makes sure that the histogram is properly merged before searching for the cut-off points.

This algorithm is used in all enhance contrast implementations described in the rest of this thesis.

5.5 Optimising memory accesses

Up until now we have used data directly from global memory, with hardly any regard for memory access optimisations. One such optimisation is memory coalescing, which means that threads access adjacent memory at the same time. Because the hardware can read multiple adjacent bytes at the same time from memory, these memory accesses are faster than those that are read in strides.

Local memory accesses are not influenced by memory coalescing, so using it to temporarily store data can improve performance. Threads will then read data from global memory properly coalesced, store it in local memory and then proceed to use the data in a non-coalesced way. After calculations the results are temporarily stored in local memory. Finally, when all computation is ready, the results are written back to global memory, properly coalesced.

This procedure only shows performance gains if data is used non-coalesced, or if data is used more than once. In the latter case it saves on accesses to global memory. The behaviour described is very similar to how caches work and in fact some of the systems use a cache to improve performance in cases where the programmer did not pay attention to the memory access pattern.

We have implemented this procedure for all algorithms, even for those that do not reuse data or access it in non-coalesced patterns.

Because these algorithms rely on all threads working on the same row, we have changed the work-group size calculation somewhat. Instead of using the minimum of the image width and the maximum work-group size, we now make sure that there is an integer number of work-groups per row. The full calculation is in Equation 5.1.

$$\text{work-group size} = \frac{\text{number of pixels per row}}{\lceil \frac{\text{number of pixels per row}}{\text{maximum work-group size}} \rceil} \tag{5.1}$$

Algorithm 5.10: Pseudocode for the OpenCL scale down filter using local memory

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $\frac{w}{2} \times \frac{h}{2}$
Local : inputcache1: cache for even (0-based) input lines
inputcache2: cache for odd (0-based) input lines

```
1  idx ← get_global_id()
2  local_idx ← get_local_id()
3  col ← Remainder(idx, w/2)
4  row ← (idx−col)/(w/2)
5  inputcache1_local_idx ← in_col,row
6  inputcache1_local_idx+wgsize ← in_col+wgsize,row
7  inputcache2_local_idx ← in_col,row+1
8  inputcache2_local_idx+wgsize ← in_col+wgsize,row+1
9  barrier
10 sum ← inputcache1_local_idx
11 sum ← sum + inputcache1_local_idx+1
12 sum ← sum + inputcache2_local_idx
13 sum ← sum + inputcache2_local_idx+1
14 out_col,row ← sum/4
```

Algorithm 5.10 shows the scale down filter using local memory. It uses two arrays that are stored in local memory: inputcache1 and inputcache2. In line 2 we set the variable local_idx to contain the local identifier, which is used as an index into the two local memory arrays.

In lines 5 to 8 the input pixels are loaded. In the previous version (Algorithm 5.1) each thread loaded the pixels used for its own calculation, thus odd and even pixels were read one after the other. In Algorithm 5.10, adjacent pixels are read at the same time, improving memory coalescing and thereby increasing memory bandwidth. Because it is now likely that a pixel is loaded by a different thread than the thread that uses it for its calculation, we need a barrier (line 9) to make sure that the data is in memory before we use it. The computation (lines 10 to 13) now uses inputcache1 and inputcache2 as the source of data, but have otherwise not been changed.

Algorithm 5.11: Pseudocode for the OpenCL scale up filter using local memory

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $2w \times 2h$
Local : inputcache: cache for input lines

1 idx $\leftarrow$ `get_global_id()`
2 local_idx $\leftarrow$ `get_local_id()`
3 wgbase $\leftarrow$ idx $-$ local_idx
4 wgbasecol $\leftarrow$ `Remainder(`*wgbase, w*`)`
5 col $\leftarrow$ `Remainder(`*idx, w*`)`
6 row $\leftarrow \frac{\text{idx}-\text{col}}{w}$
7 $\text{inputcache}_{\text{local_idx}} = \text{in}_{\text{col,row}}$
8 **barrier**
9 $\text{out}_{2\cdot\text{wgbasecol}+\text{local_idx},2\cdot\text{row}} \leftarrow \text{inputcache}_{\frac{\text{local_idx}-\texttt{Remainder}(\textit{local_idx},\,2)}{2}}$
10 $\text{out}_{2\cdot\text{wgbasecol}+\text{local_idx}+\text{wgsize},2\cdot\text{row}} \leftarrow$
$\text{inputcache}_{\frac{\text{wgsize}+\text{local_idx}-\texttt{Remainder}(\textit{local_idx},\,2)}{2}}$
11 $\text{out}_{2\cdot\text{wgbasecol}+\text{local_idx},2\cdot\text{row}+1} \leftarrow \text{inputcache}_{\frac{\text{local_idx}-\texttt{Remainder}(\textit{local_idx},\,2)}{2}}$
12 $\text{out}_{2\cdot\text{wgbasecol}+\text{local_idx}+\text{wgsize},2\cdot\text{row}+1} \leftarrow$
$\text{inputcache}_{\frac{\text{wgsize}+\text{local_idx}-\texttt{Remainder}(\textit{local_idx},\,2)}{2}}$

Algorithm 5.11 shows the scale up filter using local memory. Compared with the previous filter (the scale down filter in Algorithm 5.10), we have added wgbase and wgbasecol. wgbase stores the global identifier of the first thread in the current work-group, and it is used to calculate wgbasecol, which contains the first pixel of the block assigned to this work-group. Further, the calculation of the row and col are updated to the requirements of the new filter.

Next, data from the image stored in the local memory is copied to the inputcache array in local memory (line 7). A barrier is used (line 8) to ensure that the copying has finished (i.e., all data is in the local memory) before the operation can begin.

Remember that for a 2×2 scale-up, each pixel needs to be "doubled" both horizontally and vertically. Therefore, the pixel $\frac{\text{local_idx}-\texttt{Remainder(}\textit{local_idx,2}\texttt{)}}{2}$ is taken from the inputcache and copied into the output (line 9). Note, however, that pairs of consecutive threads take the same pixel: threads 0 and 1 both take pixel 0, threads 2 and 3 take pixel 1 and so on. This takes care of the horizontal doubling of the input pixels. The pixel is first stored in the output on the even row(s) and then on the odd row(s), which takes care of the vertical doubling.

Algorithm 5.12: Pseudocode for the OpenCL mirror vertically filter using local memory

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache: cache for input lines

1 idx $\leftarrow$ `get_global_id()`
2 local_idx $\leftarrow$ `get_local_id()`
3 col $\leftarrow$ `Remainder(`*idx, w*`)`
4 row $\leftarrow \frac{\text{idx}-\text{col}}{w}$
5 $\text{inputcache}_{\text{local_idx}} = \text{in}_{col,row}$
6 **barrier**
7 $\text{out}_{\text{col},h-\text{row}-1} \leftarrow \text{inputcache}_{\text{local_idx}}$

Algorithm 5.12 is an implementation of the mirror vertically filter using local memory. Reads and writes to global memory of the naive implementation shown in Algorithm 5.3 are already coalesced. The implementation using local memory exists only for completeness.

In Algorithm 5.13 the data is first copied into a local memory array in line 8. In line 10 the data is then copied to the output image, but in reverse

Algorithm 5.13: Pseudocode for the OpenCL mirror horizontally filter using local memory

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache: cache for input lines

```
1 idx ← get_global_id()
2 local_idx ← get_local_id()
3 col ← Remainder(idx, w)
4 row ← (idx−col)/w
5 inputcache_local_idx = in_col,row
6 barrier
7 out_col,row ← inputcache_wgsize−local_idx−1
```

order: the first thread copies the last pixel, and the last thread copies the first pixel.

Algorithm 5.14 is an implementation of the blend filter using local memory. It is presented solely for completeness, as the memory accesses in Algorithm 5.5 are already properly coalesced.

The convolution filter is the only filter that reuses data. That makes it a prime candidate to use local memory. Instead of loading a value multiple times from global memory, it can be loaded once and cached in local memory. The implementation shown in Algorithm 5.15 does that.

The centre pixel of the 3×3 area is loaded in line 5. This pixel is also used as the centre right pixel of the previous 3×3 area and the centre left pixel of the next, if these are calculated by this work-group. The first and last thread of this work-group do not have their centre side pixels loaded by adjacent threads, so they must load them themselves. Line 6 tests for this and line 7 copies the centre pixel to the sides, to handle the edges of the image as shown in Figure 3.7. If this thread is not working on the edge of the image, we load the appropriate pixel from global memory (line 9).

Lines 10 to 19 load the bottom row of the 3×3 area. Lines 11 to 13 are executed if the bottom row does not exist. Line 11 copies the centre pixels. The test in line 12 is identical to the in line 6. It executes line 13 for the first and last thread of the work-group. Line 13 copies the centre side pixels to the bottom corners of the 3×3 area. It is important that lines 6 and 12 test for the same threads. If lines 7 to 9 are executed by a different thread from the one that executes line 13, the data may not have been loaded by lines 7 to 9 when line 13 reads it. By using the same thread, we remove the

Algorithm 5.14: Pseudocode for the OpenCL blend filter using local memory

Input : base: image of size $w \times h$
overlay: image of size $w_s \times h_s$
x: $1 \leq x < w - w_s$, the x-coordinate to start blending
y: $1 \leq y < h - h_s$, the y-coordinate to start blending
α: $0 \leq \alpha \leq 1$, the blending factor
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : basecache: cache for base image lines
ovlcache: cache for overlay image lines
outcache: cache for output image lines

```
1  idx ← get_global_id()
2  local_idx ← get_local_id()
3  col ← Remainder(idx, w)
4  row ← (idx−col)/w
5  basecache_local_idx ← base_col,row
6  if x ≤ col < x + w_s and y ≤ row < y + h_s then
7  |  ovlcache_local_idx ← overlay_col−x,row−y
8  else
9  |  ovlcache_local_idx ← basecache_local_idx
10 outcache_local_idx ← (1 − α) · basecache_local_idx + α · ovlcache_local_idx
11 output_col,row ← outcache_local_idx
```

Algorithm 5.15: Pseudocode for the OpenCL convolution filter using local memory

Input : in: image of size $w \times h$
C: 3×3 convolution kernel
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache1: cache for pixels below current pixel
inputcache2: cache for pixels on the same line as current pixel
inputcache3: cache for pixels above current pixel

```
1  idx ← get_global_id()
2  local_idx ← get_local_id()
3  col ← Remainder(idx, w)
4  row ← (idx−col)/w
   ▷ Load centre pixel
5  inputcache2_{local_idx + 1} ← in_{col,row}
   ▷ Load outer pixels on the middle row
6  if local_idx = 0 or local_idx = wgsize - 1 then
7      inputcache2_{(local_idx≠0)·(wgsize+1)} ←
       inputcache2_{(local_idx=0)+(local_idx≠0)·(wgsize+1)}
8      if col ≠ 0 and col ≠ w − 1 then
9          inputcache2_{(local_idx≠0)·(wgsize+1)} ←
           in_{col+−1·(local_idx=0)+(local_idx=0),row}
   ▷ Load bottom pixels
10 if row = 0 then
11     inputcache1_{local_idx+1} ← inputcache2_{local_idx+1}
12     if local_idx = 0 or local_idx = wgsize − 1 then
13         inputcache1_{(local_idx≠0)·(wgsize+1)} ←
           inputcache2_{(local_idx≠0)·(wgsize+1)}
```

Continued on the following page. ▶

Algorithm 5.15: Pseudocode for the OpenCL convolution filter using local memory (continued)

```
14 else
15     inputcache1_{local_idx+1} ← in_{col,row−1}
       ▷ Load outer pixels on the bottom row
16     if local_idx = 0 or local_idx = wgsize - 1 then
17         inputcache1_{(local_idx≠0)·(wgsize+1)} ←
           inputcache1_{(local_idx=0)+(local_idx≠0)·(wgsize+1)}
18         if col ≠ 0 and col ≠ w − 1 then
19             inputcache1_{(local_idx≠0)·(wgsize+1)} ←
               in_{col+−1·(local_idx=0)+(local_idx=0),row−1}
   ▷ Load top pixels
20 if row = 0 then
21     inputcache3_{local_idx+1} ← inputcache2_{local_idx+1}
22     if local_idx = 0 or local_idx = wgsize − 1 then
23         inputcache3_{(local_idx≠0)·(wgsize+1)} ←
           inputcache3_{(local_idx≠0)·(wgsize+1)}
24 else
25     inputcache3_{local_idx+1} ← in_{col,row+1}
       ▷ Load outer pixels on the top row
26     if local_idx = 0 or local_idx = wgsize - 1 then
27         inputcache3_{(local_idx≠0)·(wgsize+1)} ←
           inputcache3_{(local_idx=0)+(local_idx≠0)·(wgsize+1)}
28         if col ≠ 0 and col ≠ w − 1 then
29             inputcache3_{(local_idx≠0)·(wgsize+1)} ←
               in_{col+−1·(local_idx=0)+(local_idx=0),row+1}
30 barrier
31 result ← inputcache1_{local_idx} · C_{0,0} + inputcache1_{local_idx+1} · C_{0,1} +
   inputcache1_{local_idx+2} · C_{0,2} + inputcache2_{local_idx} · C_{1,0} +
   inputcache2_{local_idx+1} · C_{1,1} + inputcache2_{local_idx+2} · C_{1,2} +
   inputcache3_{local_idx} · C_{2,0} + inputcache3_{local_idx+1} · C_{2,1} +
   inputcache3_{local_idx+2} · C_{2,2}
32 out_{col,row} ←Min(255,Max(0, result))
```

need to add a barrier between these points.

Algorithm 5.16: Pseudocode for the OpenCL enhance contrast filter using local memory

Input : in: image of size $w \times h$
low: integer, $0 \leq \text{low} \leq \text{high}$, the lower cut-off point
high: integer $\text{low} \leq \text{high} \leq 255$, the upper cut-off point
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache: cache for input lines
outputcache: cache for output lines

```
1  idx ← get_global_id()
2  local_idx ← get_local_id()
3  col ← Remainder(idx, w)
4  row ← (idx − column) / w
5  inputcache_local_idx ← input_col,row
6  if inputcache_local_idx < low then
7  |  outputcache_local_idx ← 0
8  else if inputcache_local_idx > high then
9  |  outputcache_local_idx ← 255
10 else
11 |  outputcache_local_idx ← 255 · (inputcache_local_idx − low) / (high − low)
12 output_col,row ← outputcache_local_idx
```

Algorithm 5.16 is an implementation of the normalisation stage of the enhance contrast filter using local memory. The histogram stage is unchanged from Algorithm 5.9. Algorithm 5.16 is presented for completeness only, as reads and writes to global memory of the naive implementation shown in Algorithm 5.8 are already properly coalesced.

5.6 Vectorisation

OpenCL has support for explicit vectorisation by the programmer. We have tested several implementations that use this feature. Each implementation in this section uses vectors of four 8-bit unsigned integer (`unsigned char`) elements. While OpenCL supports vectors of 2, 4, 8 and 16 elements[25], we found that for these filters and our `unsigned char` data types, four element vectors provide the best performance.

Algorithm 5.17: Pseudocode for the vectorised OpenCL scale down filter

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $\frac{w}{2} \times \frac{h}{2}$
Local : inputcache1: cache for even (0-based) input lines
inputcache2: cache for odd (0-based) input lines

1 idx $\leftarrow$ `get_global_id()`
2 local_idx $\leftarrow$ `get_local_id()`
3 col $\leftarrow$ `Remainder(`*idx*, $\frac{\frac{w}{2}}{4}$`)`
4 row $\leftarrow \frac{\text{idx}-\text{col}}{\frac{\frac{w}{2}}{4}}$
5 $\text{inputcache1}_{\text{local_idx}} \leftarrow \text{in}_{4\cdot\text{col,row}}$ (as vector)
6 $\text{inputcache1}_{\text{local_idx+wgsize}} \leftarrow \text{in}_{4\cdot(\text{col+wgsize}),\text{row}}$ (as vector)
7 $\text{inputcache2}_{\text{local_idx}} \leftarrow \text{in}_{4\cdot\text{col,row+1}}$ (as vector)
8 $\text{inputcache2}_{\text{local_idx+wgsize}} \leftarrow \text{in}_{4\cdot(\text{col+wgsize}),\text{row+1}}$ (as vector)
9 **barrier**
10 $E1 \leftarrow$ `Join(`$inputcache1_{2\cdot local_idx}.even, inputcache1_{2\cdot local_idx+1}.even$`)`
11 $O1 \leftarrow$ `Join(`$inputcache1_{2\cdot local_idx}.odd, inputcache1_{2\cdot local_idx+1}.odd$`)`
12 $E2 \leftarrow$ `Join(`$inputcache2_{2\cdot local_idx}.even, inputcache2_{2\cdot local_idx+1}.even$`)`
13 $O2 \leftarrow$ `Join(`$inputcache2_{2\cdot local_idx}.odd, inputcache2_{2\cdot local_idx+1}.odd$`)`
14 result $= \frac{E1+O1+E2+O2}{4}$ (entry-wise)
15 $\text{out}_{4\cdot\text{col,row}} \leftarrow$ result (as vector)

Algorithm 5.17 is a vectorised implementation of the shrink filter, based on Algorithm 5.1. However all actions are now performed on/with vectors. Instead of pixels, lines 5 to 8 load vectors of four pixels into the local memory arrays inputcache1 and inputcache2, using the OpenCL function `vload4`. `vload4` loads four elements from a memory location and converts them to a vector before returning it.

The four vectors are entry-wise added and divided by four to get the result (line 14). Before they are added, the vectors are first converted to vectors of four unsigned 32-bit integers to prevent overflows from occurring. After dividing by four, the result is converted back to `unsigned char`s.

Algorithm 5.18: Pseudocode for the vectorised OpenCL scale up filter

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $2w \times 2h$
Local : inputcache: cache for input lines

1 idx $\leftarrow$ `get_global_id()`
2 local_idx $\leftarrow$ `get_local_id()`
3 col $\leftarrow$ `Remainder(`*idx,* $\frac{w}{4}$`)`
4 row $\leftarrow \frac{\text{idx}-\text{col}}{\frac{w}{4}}$
5 $\text{inputcache}_{\text{local_idx}} = \text{in}_{4\cdot\text{col},\text{row}}$ (as vector)
6 **barrier**
7 $O1_0 = \text{inputcache}_{\frac{\text{local_idx}-(\texttt{Remainder(}local_idx,2\texttt{)})}{2}}.xxyy$
8 $O1_1 = \text{inputcache}_{\frac{\text{local_idx}-(\texttt{Remainder(}local_idx,2\texttt{)})}{2}}.zzww$
9 $O2_0 = \text{inputcache}_{\frac{\text{wgsize}+\text{local_idx}-(\texttt{Remainder(}local_idx,2\texttt{)})}{2}}.xxyy$
10 $O2_1 = \text{inputcache}_{\frac{\text{wgsize}+\text{local_idx}-(\texttt{Remainder(}local_idx,2\texttt{)})}{2}}.zzww$
11 $\text{out}_{4\cdot2\cdot\text{col},2\cdot\text{row}} \leftarrow O1_{\texttt{Remainder(}local_idx,2\texttt{)}}$ (as vector)
12 $\text{out}_{4\cdot(2\cdot\text{col}+1),2\cdot\text{row}} \leftarrow O1_{\texttt{Remainder(}local_idx,2\texttt{)}}$ (as vector)
13 $\text{out}_{4\cdot2\cdot\text{col},2\cdot\text{row}+1} \leftarrow O2_{\texttt{Remainder(}local_idx,2\texttt{)}}$ (as vector)
14 $\text{out}_{4\cdot(2\cdot\text{col}+1),2\cdot\text{row}+1} \leftarrow O2_{\texttt{Remainder(}local_idx,2\texttt{)}}$ (as vector)

Algorithm 5.18 contains a vectorised implementation of the scale up filter. The data is loaded from global memory and converted to a vector (line 5). In lines 7 to 10 the output is prepared. The indexes into the inputcache array are the same as in Algorithm 5.11.

For vectors of four or fewer elements, OpenCL allows a suffix of a dot followed by any combination of the letters x, y, z and w for convenient addressing of elements 0, 1, 2 and 3 respectively. In lines 7 and 9 we use that to take the first two elements of a vector and double them by using

`.xxyy`. In lines 8 and 10 we used `.zzww` to get the last two elements and double them.

In lines 11 to 14 the result is written to the output image. The even threads only write the `.xxyy` vectors, while the odd threads only write the `.zzww` vectors. The resulting write pattern is properly coalesced.

Algorithm 5.19: Pseudocode for the vectorised OpenCL mirror vertically filter

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache: cache for input lines

1 idx $\leftarrow$ `get_global_id()`
2 local_idx $\leftarrow$ `get_local_id()`
3 col $\leftarrow$ `Remainder(`*idx, w*`)`
4 row $\leftarrow \frac{\text{idx}-\text{col}}{w}$
5 $\text{inputcache}_{\text{local_idx}} = \text{in}_{4\cdot\text{col,row}}$
6 **barrier**
7 $\text{out}_{4\cdot\text{col},h-\text{row}-1} \leftarrow \text{inputcache}_{\text{local_idx}}$

Algorithm 5.19 contains a vectorised mirror vertically filter. The only difference with Algorithm 5.12 is that the vectorised implementation reads and writes four pixels at the same time.

As opposed to the previous implementation of the mirror horizontally filter, the vectorised filter in Algorithm 5.20 1) reads and writes four elements at the time, and 2) it reverses the four elements before writing them.

Algorithm 5.21 contains a vectorised implementation of the blend filter. It is very similar to Algorithm 5.14, but all global memory accesses are done as four element vectors. The calculation on line 10 is performed entry-wise.

The vectorised implementation of the convolution filter is shown in Algorithm 5.22. Apart from the loading of vectors instead of pixels, the only differences with Algorithm 5.15 until line 30 are in lines 7, 17 and 27: we do not simply copy the vector, but we also reverse it. This is because we only need a copy of the border pixel, but get a complete vector. Because the copy needs to be adjacent to the original, we reverse the vector.

Lines 31 to 39 calculate the result using vectors. They shift the vectors left or right before multiplication such that values for the same output pixel are in the same spot. Empty spots are filled with the appropriate element from the

Algorithm 5.20: Pseudocode for the vectorised OpenCL mirror horizontally filter

Input : in: image of size $w \times h$
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache: cache for input lines

```
1 idx ← get_global_id()
2 local_idx ← get_local_id()
3 col ← Remainder(idx, w/4)
4 row ← (idx−col)/(w/4)
5 inputcache_local_idx = in_4·col,row (as vector)
6 barrier
7 O ← inputcache_wgsize−local_idx−1.wzyx
8 out_4·(w−col−1),row ← O (as vector)
```

Algorithm 5.21: Pseudocode for the vectorised OpenCL blend filter

Input : base: image of size $w \times h$
overlay: image of size $w_s \times h_s$
x: $1 \leq x < w - w_s$, the x-coordinate to start blending
y: $1 \leq y < h - h_s$, the y-coordinate to start blending
α: $0 \leq \alpha \leq 1$, the blending factor
wgsize: the work-group size
Output : out: image of size $w \times h$

```
1 idx ← get_global_id()
2 local_idx ← get_local_id()
3 col ← Remainder(idx, w/4)
4 row ← (idx−col)/(w/4)
5 B ← base_col,row (as vector)
6 if x ≤ col < x + w_s/4 and y ≤ row < y + h_s then
7 |   S ← overlay_4·(col−x+),row−y (as vector)
8 else
9 |   S ← B
10 O ← (1 − α) · B + α · S (entry-wise)
11 output_col,row ← O (as vector)
```

Algorithm 5.22: Pseudocode for the vectorised OpenCL convolution filter

Input : in: image of size $w \times h$
C: 3×3 convolution kernel
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : I1: cache for pixels below current pixel
I2: cache for pixels on the same line as current pixel
I3: cache for pixels above current pixel

```
1  idx ← get_global_id()
2  local_idx ← get_local_id()
3  col ← Remainder(idx, w/4 w)
4  row ← (idx−col)/(w/4)
   ▷ Load centre pixel
5  I2_(local_idx + 1) ← in_(4·col,row) (as vector)
   ▷ Load outer pixels on the middle row
6  if local_idx = 0 or local_idx = wgsize - 1 then
7      I2_((local_idx≠0)·(wgsize+1)) ← I2_((local_idx=0)+(local_idx≠0)·(wgsize+1)).wzyx
8      if col ≠ 0 and col ≠ w/4 − 1 then
9          I2_((local_idx≠0)+(wgsize+1)) ← in_(4·(col+−1·(local_idx=0)+(local_idx=0)),row)
           (as vector)
   ▷ Load bottom pixels
10 if row = 0 then
11     I1_(local_idx+1) ← I2_(local_idx+1)
12     if local_idx = 0 or local_idx = wgsize − 1 then
13         I1_((local_idx≠0)·(wgsize+1)) ← I2_((local_idx≠0)·(wgsize+1))
14 else
15     I1_(local_idx+1) ← in_(4·col,row−1) (as vector)
       ▷ Load outer pixels on the bottom row
16     if local_idx = 0 or local_idx = wgsize - 1 then
17         I1_((local_idx≠0)+(wgsize+1)) ←
           I1_((local_idx=0)+(local_idx≠0)·(wgsize+1)).wzyx
18         if col ≠ 0 and col ≠ 4/w − 1 then
19             I1_((local_idx≠0)+(wgsize+1)) ←
               in_(4·(col+−1·(local_idx=0)+(local_idx=0)),row−1) (as vector)
```

Continued on the next page. ▶

Algorithm 5.22: Pseudocode for the vectorised OpenCL convolution filter

▷ *Load top pixels*

if $row = 0$ **then**

  $\text{I3}_{\text{local_idx+1}} \leftarrow \text{I2}_{\text{local_idx+1}}$

  if $local_idx = 0$ **or** $local_idx = wgsize - 1$ **then**

    $\text{I3}_{(\text{local_idx}\neq 0)\cdot(\text{wgsize}+1)} \leftarrow \text{I3}_{(\text{local_idx}\neq 0)\cdot(\text{wgsize}+1)}$

else

  $\text{I3}_{\text{local_idx+1}} \leftarrow \text{in}_{col,row+1}$

  ▷ *Load outer pixels on the top row*

  if *local_idx = 0* **or** *local_idx = wgsize - 1* **then**

    $\text{I3}_{(\text{local_idx}\neq 0)+(\text{wgsize}+1)} \leftarrow$
    $\text{I3}_{(\text{local_idx}=0)+(\text{local_idx}\neq 0)\cdot(wgsize+1)}.wzyx$

    if $col \neq 0$ **and** $col \neq \frac{w}{4} - 1$ **then**

      $\text{I3}_{(\text{local_idx}\neq 0)+(\text{wgsize}+1)} \leftarrow$
      $\text{in}_{4\cdot(\text{col}+-1\cdot(\text{local_idx}=0)+(\text{local_idx}=0)),\text{row}+1}$ (as vector)

barrier

result $\leftarrow$`Join(`$I1_{local_idx}.w, I1_{local_idx+1}.xyz$`)`$\cdot C_{0,0}$

result $\leftarrow$ result $+ \text{I1}_{\text{local_idx+1}} \cdot C_{0,1}$

result $\leftarrow$ result+ `Join(`$I1_{local_idx+1}.yzw, I1_{local_idx+2}.x$`)`$\cdot C_{0,2}$

result $\leftarrow$ result+`Join(`$I2_{local_idx}.w, I2_{local_idx+1}.xyz$`)`$\cdot C_{1,0}$

result $\leftarrow$ result $+ \text{I2}_{\text{local_idx+1}} \cdot C_{1,1}$

result $\leftarrow$ result+ `Join(`$I2_{local_idx+1}.yzw, I2_{local_idx+2}.x$`)`$\cdot C_{1,2}$

result $\leftarrow$ result+`Join(`$I3_{local_idx}.w, I3_{local_idx+1}.xyz$`)`$\cdot C_{2,0}$

result $\leftarrow$ result $+ \text{I3}_{\text{local_idx+1}} \cdot C_{2,1}$

result $\leftarrow$ result+ `Join(`$I3_{local_idx+1}.yzw, I3_{local_idx+2}.x$`)`$\cdot C_{2,2}$

result $\leftarrow$`Min(`255,`Max(`0, *result*`))` (entry-wise)

$\text{out}_{col,row} \leftarrow$ result (as vector)

previous or following vector. For example, in line 31 the bottom left pixels of the 3×3 areas are multiplied with $C_{0,0}$. The first three elements of the vector are prepended by the last element of the previous vector, effectively shifting the all elements one position to the right. They are then multiplied by $C_{0,0}$ and the outcome is stored in result variable. In line 32 the vector is not shifted but immediately multiplied with $C_{0,1}$ and added to result, so a pixel multiplied by $C_{0,1}$ is added to its neighbour to the left multiplied by $C_{0,0}$. In line 33 the elements are shifted to the left and the empty position is filled with the first element of the next vector. It is multiplied with $C_{0,2}$ and added to result. The same procedure is repeated for the middle and top row in lines 34 to 36 and 37 to 39 respectively.

Before multiplication the vectors are converted to vectors of unsigned 32-bit integers to prevent overflow. When converting the result back to unsigned 8-bit integers, we use a saturated conversion. Values larger than 255 are set to 255. Those lower than 0 are set to 0. This saves us the explicit calls of the `Min()` and `Max()` functions in line 40.

Algorithm 5.23: Pseudocode for the vectorised OpenCL enhance contrast filter

Input : in: image of size $w \times h$
low: integer, $0 \leq \text{low} \leq \text{high}$, the lower cut-off point
high: integer $\text{low} \leq \text{high} \leq 255$, the upper cut-off point
wgsize: the work-group size
Output : out: image of size $w \times h$
Local : inputcache: cache for input lines
outputcache: cache for output lines

1 idx $\leftarrow$ `get_global_id()`
2 local_idx $\leftarrow$ `get_local_id()`
3 col $\leftarrow$ `Remainder(`*idx*, $\frac{w}{4}$`)`
4 row $\leftarrow \frac{\text{idx}-\text{column}}{\frac{w}{4}}$
5 $f \leftarrow \frac{255}{\text{high}-\text{low}}$
6 $\text{inputcache}_{\text{local_idx}} \leftarrow \text{input}_{4 \cdot col, row}$ (as vector)
7 result $\leftarrow f \cdot (\text{inputcache}_{\text{local_idx}} - low)$
8 $\text{outputcache}_{\text{local_idx}} \leftarrow$`Min(`255,`Max(`0, *result*`))` (entry-wise)
9 $\text{output}_{4 \cdot \text{col,row}} \leftarrow \text{outputcache}_{\text{local_idx}}$ (as vector)

Algorithm 5.23 contains a vectorised implementation of the normalisation part of the enhance contrast filter. Line 5 pre-computes the normalisation constant. It is stored as a 32-bit floating point value. Line 6 loads the one four element vector as input data. In line 7 the calculation is performed.

The input data is converted to a vector of 32-bit floating point values using `convert_float4` before the calculation is performed. In line 8 the result is converted back to a vector of unsigned 8-bit integers, using a saturated conversion to take care of the `Min()` and `Max()` calls. Line 9 stores the result in the output image.

5.7 Using pinned memory

Using pinned memory is a technique recommended by NVIDIA in [28]. While we cannot directly control if buffers are allocated in pinned memory, we can create memory buffers with certain flags that make it likely they are allocated in pinned memory. These buffers are allocated by the host. This technique requires no changes in the OpenCL kernels.

The normal method to upload input data to the device memory is to allocate a buffer on the device and then copy the data into it. When using pinned memory, we allocate two buffers. One is the same buffer we would otherwise allocate; the other is a buffer mapped into the host memory. We can then copy data into that buffer as if it was a normal region in the host memory. After that, we must still upload the data to the device.

Using pinned memory does not seem likely to improve performance at all. But since it is a best practice, we have included it in our benchmarks.

5.8 Results

Experiments
Fully sequential
Naive implementation using `char`
Naive implementation using `float`
Naive implementation using `int`
Parallel histogram (only for enhance contrast)
Local memory
Vectorisation
Pinned memory

Table 5.1 – *OpenCL experiments*

5.8.1 Fully sequential

The results for the sequential implementation are shown in Figures A.50 to A.56. Only the calculation timer is shown. Because there is no initialisation of the hardware there would be no difference with a kernel timer. The relation

between the number of input pixels—i.e., the image size—and the run-time is linear, as you would expect based on the algorithms used (all $O(n)$).

There are bumps in the GeForce GTX480 results for the scale down (Figure A.50a), scale up (Figure A.51a) and mirror vertically (Figure A.52a) filters, at 3072×2304, 1536×1152 and 2048×1536, respectively. These bumps are not caused by single outliers, but rather by a group of "runs" that are slower than the others. Each run uses the same input size.

It is unlikely that these bumps are caused by malfunctioning hardware, because these three filters have all been tested on a different machine. The configuration of these machines is the same configuration as that of the machines that ran the Xeon E5620 benchmarks in Figures A.50c, A.51c and A.52c, which do not exhibit this behaviour. The only difference between these two benchmarks is the unused OpenCL library that is linked: the NVIDIA library for the GeForce GTX480 and the ATI/AMD library for the Xeon E5620.[1]

5.8.2 Naive implementation using `char`

The results of the naive implementation (a simple OpenCL parallelisation) are shown in Figures A.57 to A.63. These times are compared to the run times of the sequential version and include the time taken to initialise the hardware and to transfer the data from and to the device memory. The plots show that for smaller sizes the constant initialisation times overshadow the size-dependant transfer and calculation times. This is especially so for the GPUs, but to a lesser degree also for the Xeon E5620 and the Cell BE. The Radeon HD 5870 plots do not show results for the larger sizes—8192×6144 and up for the scale up filter, 12288×9216 and up for the enhance contrast filter, and 16384×12288 for the other filters. This is because the device does not have enough memory to store the input and output data for these sizes. For most filters the programs crashed with an OpenCL error message stating that not enough memory was available. For the enhance contrast filter this caused the program to hang. It could not be killed and the machine had to be rebooted to make the card available again.

To further compare the run times between implementations, we have also tracked the time spent in the kernel. This timer is started after the hardware is initialised and the input data is transferred to the device and stopped before the output is downloaded from the device. These results are displayed in Figures A.64 to A.70. The timings for the enhance contrast filter are further specified in a sequential part where the histogram is calculated—shown in Figure A.71—and a parallel normalisation part—shown in Figure A.72.

[1] We linked these libraries because we wanted to use the same Makefile for all tests on a machine to make sure we did not accidentally change compiler flags for only one implementation.

The results for the GeForce GTX480 show a jump for all filters. This jump is between 1024×768 and 1536×1152 for the scale down filter, between 512×384 and 768×576 for the scale up and blend filters, and between 768×576 and 1024×768 for the other filters. The amount of memory used by both input and output data is at most 960 KiB before the jump, which is also the total amount of local memory available to the Streaming Multiprocessors.

The Radeon HD 5870 has unexplained drops and spikes in run time. When keeping the filter and the image size constant, we can see a jump in run time on the Radeon HD 5870: the first few runs take a short time, but the rest of the runs take a long time. E.g., for the scale up filter and an input image of size 1024×768, we noticed that the first five runs took between 1723 µs and 1749 µs, the fifth run took 2273 µs, and the last four runs took between 11 812 µs and 12 252 µs. We believe this big jump is caused by overheating of the card.

The Xeon E5620 results form a straight line apart from the smaller images. The run times for the smaller sizes are so small, that they all are about equally fast.

The run times for the Cell BE results have a similar shape as the Xeon E5620 results, but here the flat part of the curve is not caused by internal variance in the results. We guess that this is happens because the calculation and memory transfers cannot be overlapped due to a lack of data.

5.8.3 Naive implementation using other data types

We tested each filter with three different data types: `unsigned char`, an unsigned 8-bits integer type, `unsigned int`, an unsigned 32-bits integer type and `float`, a 32-bit floating point type. The run times for these types are compared in Figures A.73 to A.79.

The run time is, for nearly all platforms and filters, larger for the implementations using `int` and `float` than for the implementation using `char`. This is caused by the extra time needed to transfer the larger data types. The larger size of `int` and `float` is also the reason that the larger image sizes are missing in some of the plots: the devices do not have enough memory to store that much data. The results for the `float` implementation of the convolution filter on the Cell BE are missing because the filters could not be compiled due to a compiler bug.

The only test that shows an improvement in run time is the convolution filter on the Radeon HD 5870. This improvement is caused by faster kernel run times. The kernel run times are plotted in Figures A.80 to A.86. The Radeon HD 5870 is the only platform we tested that benefits from the use of 32-bits data types. The other platforms are equally fast or a bit slower for the 32-bits data types when compared to the 8-bits integer. There is no difference between the 32-bits `int` type and the `float` on any platform.

5.8.4 Parallel histogram

So far we have the histogram calculations sequentially. They can also be done in parallel. The results are shown in Figure A.89. The GPUs benefit from the parallel calculation, while the Xeon E5620 is somewhat slower with it. The histogram calculation is done in a single work-group. While the GPUs have multiple shader processors per streaming multiprocessor and can therefore do work in parallel, the Xeon E5620 has only one core per work-group. The additional overhead of keeping track of multiple threads causes the additional delays. There are no results for the Cell BE, because its compiler could not fit the thread histograms into the Local Store.

5.8.5 Using local memory

The results of using local memory are displayed in Figures A.90 to A.96. The Xeon E5620 is slower using local memory. This can be attributed to the fact that both global memory and local memory are memory areas in the main memory of the machine. There are no performance benefits to using data from local memory as opposed to global memory, so the overhead of copying the data is wasted effort.

The Cell BE implementation is a little faster for the scale down, blend and convolution filters, and a little slower for the mirror vertically and mirror horizontally filters. There is no difference for the scale up filter. There are no results for the enhance contrast filter for the reason mentioned in the previous section.

The Radeon HD 5870 is generally faster using local memory. For the GeForce GTX480 there is only a difference for the enhance contrast filter, and then only for 1024×768 and smaller.

5.8.6 Vectorisation

Vectorisation is an improvement for most platforms (Figures A.97 to A.103). Only the Radeon HD 5870 performs worse after vectorisation for some filters and never better than the local memory implementation. However, the results for the Radeon HD 5870 are rather erratic, and are showing the same overheating signs we identified earlier. We believe that the Radeon HD 5870 would perform better if it was cooled down before testing.

5.8.7 Using pinned memory

The results for using pinned memory are displayed in Figures A.104 to A.110. Like we predicted in Section 5.7 this implementation is not an improvement on any platform. The additional data copies are only adding overhead.

5.8.8 Summary

Overall, our results for the OpenCL implementations show the following:

- The overhead of copying data from host memory to device memory makes using OpenCL parallelisation generally not worth it. Only for a few filters and the very largest image sizes does it improve performance.
- Using `float` or `unsigned int` instead of `unsigned char` data types does not improve performance.
- The Radeon HD 5870 slows down under load. We believe this is due to overheating. It causes unpredictable results.
- Using local memory only works on the Radeon HD 5870. It hurts performance for non-GPU cards, because these do not have a natural "local memory" area.
- Vectorisation is a useful optimisation that works on all platforms.
- Despite being suggested by the NVIDIA OpenCL best practises, using pinned memory only slows the filters down. It takes more effort to implement and can therefore not be recommended.

Chapter 6

Discussion

In this chapter we compare the programmability of the Cell SDK and OpenCL based on the three indicators discussed in Chapter 1. Each indicator has its own section dedicated to it: performance in Section 6.1, portability in Section 6.2 and productivity in Section 6.3. We tie these metrics together in Section 6.4.

6.1 Performance

Performance can be measured in a variety of ways. The easiest way is to compare the wall clock time between applications. This is the time between reading the input and writing the output. It gives a measure of the entire platform stack: hardware, compiler and programming model. The user is likely to care the most about this measure, since it is the amount of time he has to wait to get an answer. Table 6.1 compares these times for an 4096×3072 image. That image size is the largest that worked for all platforms and filters. For the OpenCL platforms we have shown the run times of the implementations using pinned memory. For the Cell SDK platforms we have shown those of the SPE-assisted vectorised implementations. Where those implementations crashed, we have shown an earlier implementation. All available SPEs were used: 16 on the QS22 and 6 on the PlayStation 3.

From the table we can conclude that the hardware-centric Cell SDK implementations are faster by an order of magnitude. Using OpenCL, the QS22 and the Radeon HD 5870 reach around similar performance, and the Xeon E5620 is generally a bit faster. The GeForce GTX480 results are disappointing. Most of the extra delay is caused during initialisation of the hardware. Emulating the copying of data from host memory to device memory is probably slowing down the Xeon E5620 and QS22 OpenCL implementations somewhat.

Another way to compare performance is to look at how much of the

	OpenCL				Cell SDK	
Filter	**NV**	**ATI**	**Xeon**	**QS22**	**QS22**	**PS3**
Scale down	1.42 s	0.26 s	0.12 s	0.18 s	19 ms	11 ms
Scale up	1.45 s	0.31 s	0.28 s	0.53 s	19 ms	12 ms
Mirror vertic.	1.42 s	0.27 s	0.23 s	0.27 s	17 ms	10 ms
Mirror horiz.	1.43 s	0.28 s	0.23 s	0.27 s	17 ms	10 ms
Blend	1.45 s	0.29 s	0.13 s	0.32 s	17 ms	15 ms
Convolution	1.45 s	0.28 s	0.25 s	(0.27 s)[a]	23 ms	27 ms
Enh. contrast	1.50 s	0.56 s	(0.14 s)[b]	(0.46 s)[c]	27 ms	36 ms

[a] Implementation using local memory.
[b] Parallel histogram implementation.
[c] Naive implementation using `char`.

Table 6.1 – *Comparison of the wall clock time spent on running the filter. Note the difference in units between OpenCL and Cell SDK implementations.*

peak performance your application is reaching. This shows how well the programming model works on the hardware platform. Unfortunately, this approach does not work well for our application. Peak performance for a system is usually reported using flop/s, while our application does not use floating point operations.

Still, it is useful to look at the peak performance of these cards, even if we can only do so in a less formal way. The peak performance metrics of the systems are shown in Table 6.2. If we look at both Tables 6.1 and 6.2, we can see that the compute power of the GPUs is a lot bigger than the other systems. Yet the other systems can compete very well on run time, with the QS22 reaching a similar run time to the Radeon HD 5870 and the Xeon E5620 outperforming them by a small margin. The best explanation for that is that the GPUs are limited by their interface to host memory. While their interface to device memory is quite fast, input data needs to be put in device memory first. The bandwidth to the host memory is only 8.0 GiB/s, which is low compared to the other systems. This does not explain the difference in performance between the GeForce GTX480 and the Radeon HD 5870, since both suffer the same limitation. In light of Table 6.2, the difference between Cell SDK implementations and OpenCL implementations is even larger.

We must conclude that the hardware-centric Cell SDK programming model performs better than the OpenCL model. In part this is caused by higher bandwidth to the main memory, but there is also a gap between the OpenCL model and the Cell BE hardware, which is not bridged by the compiler and run time environment.

System	Peak performance	Memory bandwidth
GeForce GTX480	1.35 Tflop/s [35]	177.4 GiB/s[a] [35]
Radeon HD 5870	2.72 Tflop/s [35]	153.6 GiB/s[a] [35]
Xeon E5620	38.4 Gflop/s [17]	25.6 GiB/s [18]
QS22 (OpenCL)	460 Gflop/s[b] [10]	51.2 GiB/s [34]
PlayStation 3	160[c] Gflop/s[b] [4]	25.6 GiB/s [4]

[a]This is the memory bandwidth to the device memory. When transferring data from and to the host, the bandwidth is limited by the PCIe bus to 8.0 GiB/s.

[b]Single precision

[c]153.6 Gflop/s for the SPEs and 6.4 Gflop/s for the PPE.

Table 6.2 – *Peak performance and memory bandwidth*

6.2 Portability

If a programmer cannot reach the required performance on one platform, he has no other recourse than to use another platform. Without portability this comes down to starting over, which is obviously undesirable.

We define three types of portability: code portability, parallelism portability and performance portability. Code portability means that the programmer can run the same program on another platform. With parallelism portability the same parallelisation strategy works on both platforms. Performance portability means that the program will reach a similar level of performance on both platforms.

Implementations for the Cell SDK that do not use the SPEs can run on other processors with the POWER architecture. While vectorisation on the PPE is a valid optimisation, it is not the most common parallelisation strategy for the Cell SDK. When using the SPEs, however, there is no longer any portability with other platforms. The parallelisation strategy is unique because there are at present no other platforms that use the same heterogeneous architecture.

OpenCL has better code portability because it is standardised and implemented on multiple platforms. However, to ensure that applications are portable, the programmer needs to query the device, instead of assuming certain resources are available. The most prominent examples of these are work-group sizes and the amount of memory (both local and global) available to the application.

Code portability was harmed by bugs in the platforms. The most serious of this was a repeatable hang in the Radeon HD 5870. This hang left the GPU driver in an unusable state and the machine had to be reset.[1] The only other platform with major issues is the OpenCL implementation for

[1]We did not notice any problems with the ATI/AMD implementation for multi-core processors, suggesting that the stability problems for the GPU are either in the hardware or in the hardware-specific software (e.g. drivers and firmware).

the Cell BE. Several programs failed to compile, for varying reasons[2]. Still, we must conclude that OpenCL has no structural code portability problems. The only problems that we identified were bugs in the implementations, not problems in the specification.

OpenCL has parallelism portability by construction. The added layer of abstraction means that the programmer can use the same parallelisation strategy on all platforms. It is the responsibility of the compiler to map the OpenCL parallelism model to the hardware.

From Section 6.1 we can see that there is a huge difference in performance between platforms, so there is no performance portability. Platforms will also respond differently to optimisations, which makes the choice of which optimisations to implement a platform-dependant choice.

Still, we can only conclude that OpenCL is a major improvement on the Cell SDK when it comes to portability. The programmer has to re-*tune* the application for a new platform, but does not have to re-*implement* it.

6.3 Productivity

OpenCL code is simpler than the SPE code for the same filters. In OpenCL it is the control process' responsibility to load the data into the device memory. On the Cell BE, it is the responsibility of the SPE code. The hardest part of the memory transfers on the SPEs is that the data is transferred in parts. Doing so requires keeping track of pointers into main memory, and, for double buffered solutions, into the Local Store. Bugs in the pointer arithmetic were a common source of bugs.

Debugging Cell BE programs is easier than debugging OpenCL programs. OpenCL has no debugging tools available and no easy way to output data, whereas Cell BE has a debugger and the ability to simply use `printf`.

The Cell BE platform has fewer bugs by itself. When an application fails, there is always a bug in the program. In contrast, GPUs tend to be a bit more flaky. From time to time a reboot would cause a program to start working, while it failed before the reboot.

Overall, we find that programming OpenCL is much easier and much less error-prone than programming with the Cell SDK. The few marks against it (many bugs; lacking debugging tools) are caused by immaturity of the platform and not by inherent problems in the design. These are likely to be solved over time.

[2] Some OpenCL function were not properly defined; some valid syntax was considered invalid; and the compiler could sometimes not fit the program in memory.

6.4 Programmability

The aim of our programming exercise was to determine the programmability of two extremely different programming models: a hardware-centric one, the Cell SDK, and a generic one, OpenCL. We have compared their performance, productivity, and portability, aiming to evaluate their differences with respect to programmability (see Chapter 1 for details). We found that the Cell SDK provides much better performance than OpenCL (on the Cell BE), at the expense of lower productivity and no portability. On the other hand, OpenCL offers code and parallelism portability, with good productivity. In terms of performance, for the Cell BE, the results are disappointing: the combination of an immature compiler and a different parallelisation approach lead to an unacceptable performance gap of an order of magnitude. We conclude that the actual OpenCL realisation is not a good alternative for programming the Cell BE.

For NVIDIA GPUs, OpenCL is a natural fit for parallelisation and productivity. We did not compare the performance with alternative programming models, but we note that other studies[7] show little performance gaps when compared with CUDA. Therefore, we believe that OpenCL is a good alternative to CUDA: it adds portability while inducing very little performance penalties.

For ATI/AMD GPUs, OpenCL is the programming model of choice. It answers well to the programmability and portability requirements, but it is not yet mature enough performance-wise.

Finally, for CPUs, OpenCL provides a quick solution to run a parallel application already available, but the difference in hardware granularity and memory systems causes a performance drop (the mapping between the OpenCL platform and a multi-core CPU is a non-trivial one). Given alternatives such as OpenMP [3] it is difficult to recommend programmers to implement their parallel CPU applications using OpenCL only for the sake of portability.

We summarise these findings in Table 6.3.

	SDK	**OpenCL**			
Platform	**Cell**		**NV**	**ATI**	**CPU**
Performance	★★★★★	★★	★★★★[a]	★★★★	★★★
Productivity	★★	★★★	★★★★	★★★★	★★★★
Portability	—	★★★	★★★	★★★	★★★
Programmability	★★★	★★	★★★★	★★★★	★★★

[a] We ignored the long initialisation time of the NVIDIA card for this table.

Table 6.3 – *Programmability of the platforms summarised*

Overall, we note that OpenCL provides an important improvement for programmability. However, there are still details to be taken into account

and improved before it becomes a fully usable model. Cell SDK remains the "assembly" language of the Cell BE: difficult to use, but able to provide close to optimal performance. Despite the end-of-life of the Cell BE, its programmability remains a lesson well learnt, and we believe that any new heterogeneous hardware platform should use these lessons for building a better programming model.

Chapter 7

Conclusions and Future Work

In the past years manufacturers made big strides in developing and popularising many-core systems. The cheap computing power offered by these new products has been made available with a variety of tool kits. The early tool kits were very tied to the specific hardware they were released for. This has led to a standardisation process aiming to provide a single standard for many-core computing. Such a standard has finally appeared in 2008, under the name of OpenCL and under the supervision of a large consortium of hardware and software vendors.

In this thesis we have investigated how the programmability of many-core systems has evolved. We started by looking at the hardware-centric Cell SDK and compared it to the newer OpenCL standard, using a set of seven 2D image processing filters as benchmarks. We compared these on performance, portability and productivity. We found that using OpenCL improved portability and productivity. However, this came at the cost of reduced performance.

Performance The Cell SDK applications outperformed the OpenCL implementations by an order of magnitude on the same Cell BE machine. We believe this behaviour has two main reasons:

- The OpenCL programming model involves a layer of abstraction over the Cell BE hardware. The mapping between the OpenCL model and the hardware is non-trivial due to the large difference in granularity between the two. This mapping is, most likely, sub-optimal. I.e., the hand-tuned mapping in Cell SDK is much better tuned to these filters than the basic OpenCL one.
- The tool chain for OpenCL programming for the Cell BE platform is not mature enough. If these tools were to be developed further, the

performance loss could probably be meliorated.

When we ran the tests on the GPUs we found an even larger difference in performance with the hand-tuned Cell SDK implementations, despite the fact that these cards have much higher peak compute performance. We attribute the difference mainly to the very low bandwidth op the PCIe bus (the bus that connects the GPUs to the host and is used to transfer the input and output data from the host memory to the device memory). Furthermore, as the kernels are memory-intensive, they are only using a small fraction of the GPU computation power, resulting in very limited performance.

Portability The hardware model of the Cell BE is unique, and using the Cell SDK requires unique strategies to parallelise our applications to make proper use of the SPEs. In this case, the code is platform-specific, and therefore not portable.

OpenCL, by contrast, is portable by design. The code compiles and runs on all supported platforms (if we ignore compiler bugs) and the parallelising strategy is the same for each platform. However, performance portability does not exist. We noticed large differences in performance between systems. The systems also responded differently to optimisations. Some changes would improve performance on one system, but harm it on another. Porting an application from one system to another would therefore involve tuning the program again. Still, it is an improvement to using hardware-centric models like the Cell SDK where porting an application comes down to reimplementing it.

Productivity The immaturity of the platform is the biggest productivity loss when writing OpenCL programs. The lack of debugging tools, especially, can make developing with OpenCL painful. However, OpenCL tool chains are under active development and it seems likely these problems will be addressed in the future. The Cell SDK on the other hand suffers from structural problems. The programmer must manually load data from memory which is a complex process with many possibilities to make mistakes. Furthermore, to gain good performance, the programmer has to manage several levels of parallelism in the hardware, further complicating development.

The most important reason to run an application on a many-core system is to improve performance. At first glance this would mean that we cannot consider OpenCL to be an improvement to the Cell SDK for our applications. However, two other points need to be made.

1. The tool chain for OpenCL on the Cell BE platform is immature. There are some major compiler bugs. It stands to reason that performance could get a lot better if the tool chain were developed further.

2. The OpenCL programming model is much more suitable for GPUs than for other platforms. From another study, we know that OpenCL can reach the same performance an CUDA, the native platform for NVIDIA GPUs[7].

So while the performance difference could be expected to grow smaller over time, it is likely a difference in performance would remain. This does not completely kill the programmability of OpenCL. Rather, it could be considered a higher level model. The improved productivity and portability can create a similar relation between OpenCL and Cell SDK as exists between C and assembler.

Overall, we conclude that the five years of effort towards closing the programmability gap of many-core processors have led to a standardised solution that is promising in terms of productivity and portability. However, efforts are still to be made to improve OpenCL's performance.

Kernels The kernels we used in our experiments have a low arithmetic intensity, which was the major cause of low performance on all the multi-core platforms. Applications with a higher arithmetic intensity would definitely have better performance, and the comparison will likely be more favourable towards the GPUs. However, our programmability analysis is affected by this limitation in terms of generality—i.e., more complex applications and different application classes would make the comparison more generic.

Platforms In terms of platforms, there are several observations we need to make, and we present them in order of their impact.

First of all, the behaviour of the Radeon HD 5870 was very irregular. We noticed the card slowing down significantly over time, in situations where the input size, filter and implementation were kept constant. This was only noticeable in the time spent on calculations. The time to transfer data from and to the host memory was unaffected. We suspect this is due to overheating of the card.

The GeForce GTX480 has a very slow initialisation time. It took more than 1 second to create an OpenCL context. This would seem to be a bug in the NVIDIA OpenCL implementation. Pure calculation time of the GeForce GTX480 is better than on the Radeon HD 5870, although the ATI/AMD card has higher peak compute performance. This is an effect of the difference in local memory bandwidth in favour of the NVIDIA board, which pays off for kernels that are so severely memory-bound.

The Xeon E5620 ran out of memory for a few test cases, even though the machine had enough physical memory. We still achieved good performance when using it, even though the OpenCL programming model is not a natural fit to the multi-core environment (in terms of granularity).

Finally, some OpenCL code could not be properly compiled for the QS22 Cell blades. We have isolated the cause of this behaviour as being a compiler problem. Compared to other OpenCL platforms, the QS22 performed quite well, performance-wise for a 5-years old platform.

Future Work In this work we have used seven kernels with low arithmetic intensity. This choice had a major influence on our results. A higher arithmetic intensity will likely increase performance on most of the many-core platforms, making the optimisations more interesting to apply, and the results (potentially) more diverse. One way an increased complexity can be achieved is to combine several of these filters into synthetic benchmarks. Measuring the performance of such a benchmark will give more insights into both low-level and high-level optimisations and their effects. Furthermore, it will give a new dimension to evaluating productivity and portability, as the programming models need to tackle an additional application-level parallelism layer.

Our observations on performance were also limited by the image sizes. We encountered limits in memory size on several OpenCL platforms. The Cell SDK implementations were limited in the width of the image by the maximum size of DMA transfers. Both of these limits could be worked around with some effort. It would be an interesting experiment to see how these workarounds would impact performance. Using OpenCL especially, we would like to see the impact of analysing input data partitioning and, related to this, the potential improvement in performance and suitability in productivity of multi-node implementations.

Finally, we believe our comparison in programmability between the two models has also been limited by the specific class of kernels we have chosen (i.e., 2D image processing). Therefore, we recommend similar comparisons for other classes of applications, allowing for more generic conclusions. While this is certainly a good investment for OpenCL kernels, Cell SDK might be replaced by more modern languages/platforms, such as OpenMP (for CPUs) and CUDA (for GPUs).

Bibliography

[1] AMD delivers and submits for certification industry's first OpenCL™ software development platform for x86 CPUs. Press release, Aug. 2009.

[2] Zachary K. Baker, Maya B. Gokhale, and Justin L. Tripp. Matched filter computation on FPGA, Cell and GPU. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007*, pages 207–218. IEEE, 2007.

[3] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0, May 2008.

[4] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the PlayStation 3. *University of Tennessee Knoxville*, 2007.

[5] Jason Cross. ATI steps into Stream computing. *ExtremeTech*, Oct. 2006.

[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Technical Report CS-10-656, University of Tennessee, Aug. 2010.

[7] Jianbin Fang, Ana Lucia Varbanescu, and Henk J. Sips. A comprehensive performance comparison of cuda and opencl. In *40th International Conference on Parallel Processing*, 2011.

[8] R. Ginjupalli and G. Khanna. High-precision numerical simulations of rotating black holes accelerated by CUDA. *Arxiv preprint arXiv:1006.0663*, 2010.

[9] Mark Hachman. Nvidia's GeForce 8800 adds 'CUDA' tech; can work as CPU. *ExtremeTech*, Nov. 2006.

[10] IBM BladeCenter QS22.

[11] IBM, Sony, Sony Computer Entertainment Inc. and Toshiba disclose key details of the Cell chip. Press release, Feb. 2005.

[12] IBM, Sony, Sony Computer Entertainment Inc., and Toshiba unveil details of the Cell microprocessor. Press release, Aug. 2005.

[13] Linux(tm) extensions, key software development tools released for Cell Broadband Engine microprocessor. Press release, Nov. 2005.

[14] IBM releases OpenCL drivers for POWER6 and Cell/B.E., Oct. 2009.

[15] IBM Systems and Technology Group. *Cell Broadband Engine Programming Tutorial Version 2.1*, Mar. 2007.

[16] IBM Systems and Technology Group. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor Version 1.11*, May 2008.

[17] Intel microprocessor export compliance metrics.

[18] Intel Xeon processor E5620 specifications.

[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[20] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *Computing Research Repository*, abs/1005.2581, 2010.

[21] Khronos adopter conformant products. `http://www.khronos.org/adopters/conformant-products/`. Retrieved May 29th, 2011.

[22] The Khronos Group releases OpenCL 1.0 specification. Press release, Dec. 2008.

[23] Khronos launches heterogeneous computing initiative. Press release, June 2008.

[24] Khronos OpenCL Working Group. *The OpenCL C++ Wraper API, version 1.1.* The Khronos Group Inc., fourth edition, June 2010.

[25] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1.* The Khronos Group Inc., thirty-sixth edition, Sept. 2010.

[26] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of OpenCL programs. In *The Fifth International Workshop on Automatic Performance Tuning (iWAPT2010)*, 2010.

[27] Martin Linklater. Optimizing Cell code—a case study. *Game Developer*, pages 15–18, Apr. 2007.

[28] NVIDIA. *NVIDIA OpenCL Best Practices Guide, Version 1.0.* NVIDIA, Aug. 2009.

[29] NVIDIA releases OpenCL driver to developers. Press release, Apr. 2009.

[30] Miloš Rašković, Ana Lucia Varbanescu, Wouter Vlothuizen, Maarten Ditzel, and Henk Sips. OCL-BodyScan: A case study for application-centric programming of many-core processors, 2011.

[31] Rob V. van Nieuwpoort and John W. Romein. Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming*, 39:88–114, Feb. 2011.

[32] Ana Lucia Varbanescu. *On the effective parallel programming of multi-core processors.* Ph.D. dissertation, Delft University of Technology, Dec. 2010.

[33] Ana Lucia Varbanescu, Pieter Hijma, Rob van Nieuwpoort, and Henri Bal. Towards an effective unified programming model for many-cores, 2011.

[34] J.S. Vogt, R. Land, H. Boettiger, Z. Krnjajic, and H. Baier. IBM BladeCenter QS22: design, performance, and utilization in hybrid computing systems. *IBM Journal of Research and Development*, 53(5):3:1–3:14, 2009.

[35] Scott Wasson. Nvidia's GeForce GTX 480 and 470 graphics processors. *The Tech Report*, Mar. 2010.

[36] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *IEEE International Symposium on Parallel and Distributed Processing, 2008*, pages 1–14. IEEE, 2008.

[37] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

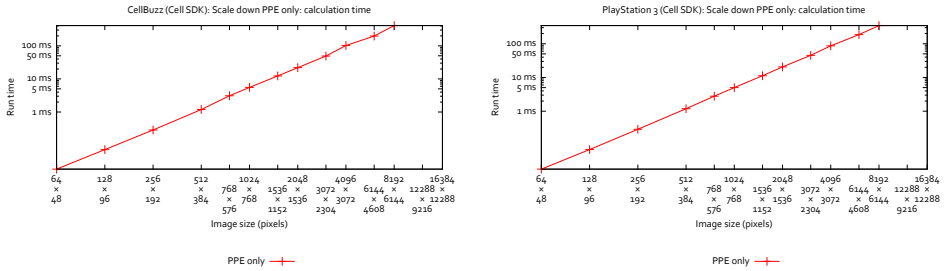
Appendix A

Graphs

A.1 Cell SDK

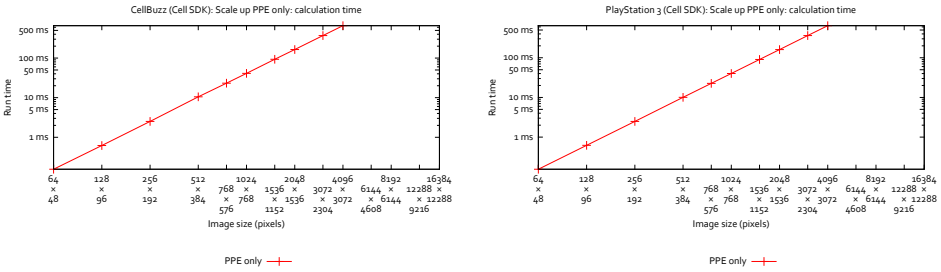
A.1.1 Fully sequential

In this section we show the detailed results of running the fully sequential version of the Cell SDK code.



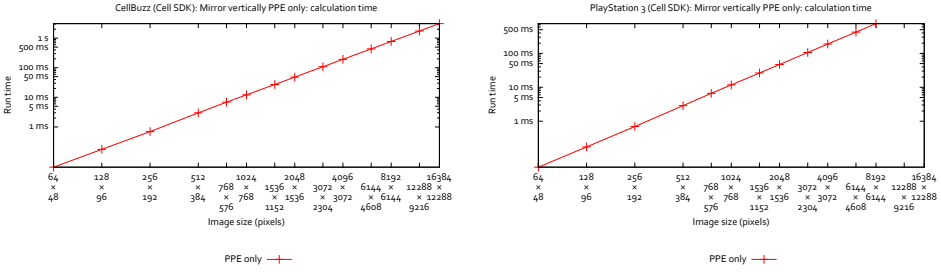
QS22 *PlayStation 3*

Figure A.1 – *Sequential implementation of the scale down filter. Note that the scales of these plots differ.*



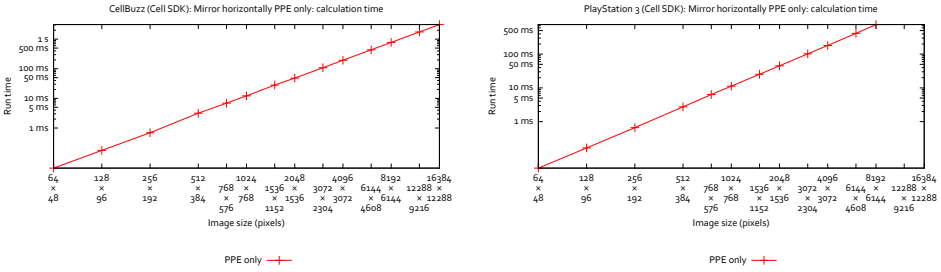
QS22 *PlayStation 3*

Figure A.2 – *Sequential implementation of the scale up filter. Note that the scales of these plots differ.*



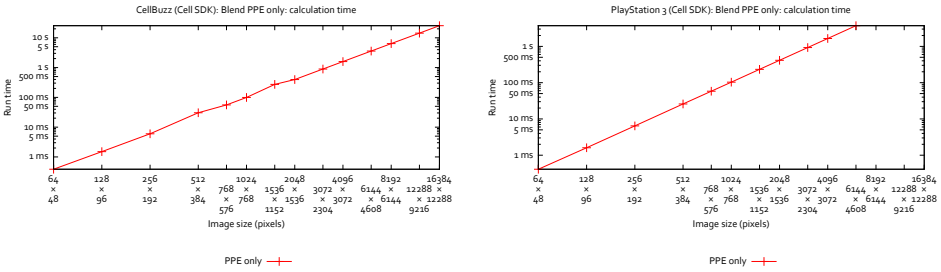
QS22 *PlayStation 3*

Figure A.3 – *Sequential implementation of the mirror vertically filter. Note that the scales of these plots differ.*



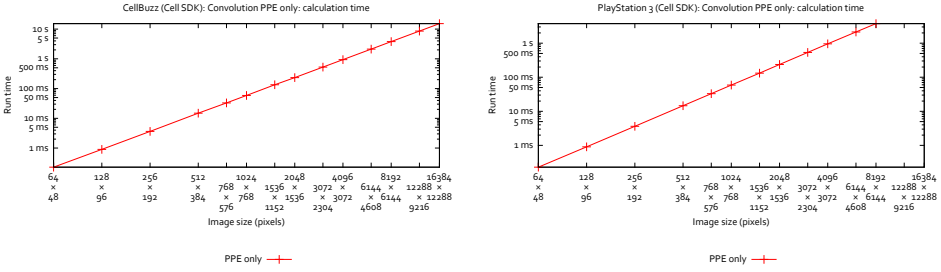
QS22 *PlayStation 3*

Figure A.4 – *Sequential implementation of the mirror horizontally filter. Note that the scales of these plots differ.*



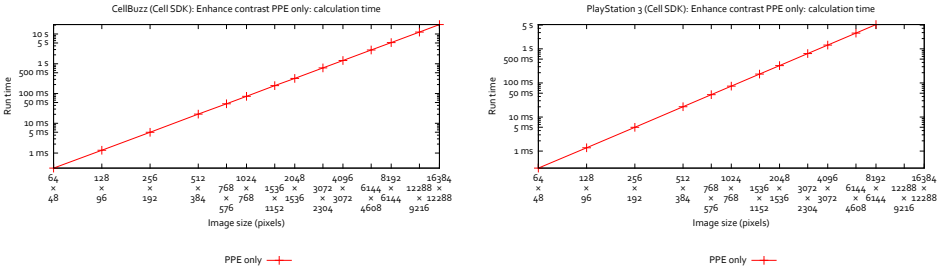
QS22 *PlayStation 3*

Figure A.5 – *Sequential implementation of the blend filter. Note that the scales of these plots differ.*



QS22 *PlayStation 3*

Figure A.6 – *Sequential implementation of the convolution filter. Note that the scales of these plots differ.*

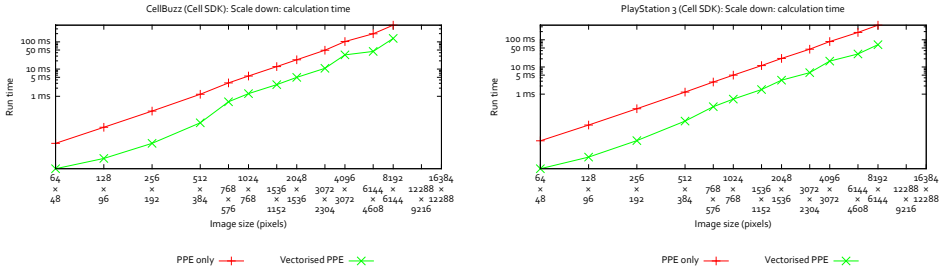


QS22 *PlayStation 3*

Figure A.7 – *Sequential implementation of the enhance contrast filter. Note that the scales of these plots differ.*

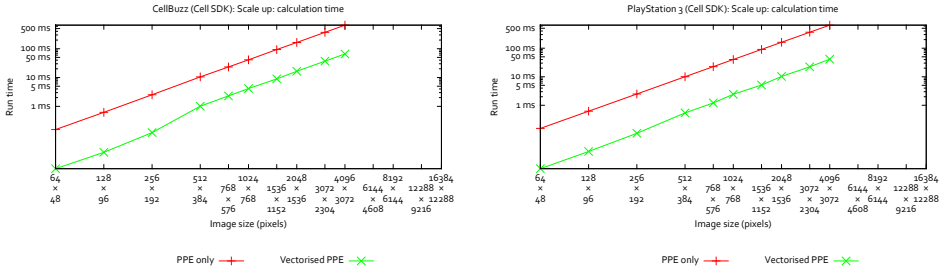
A.1.2 PPE vectorised

In this section we show the detailed results of running the PPE vectorised version of the Cell SDK code.



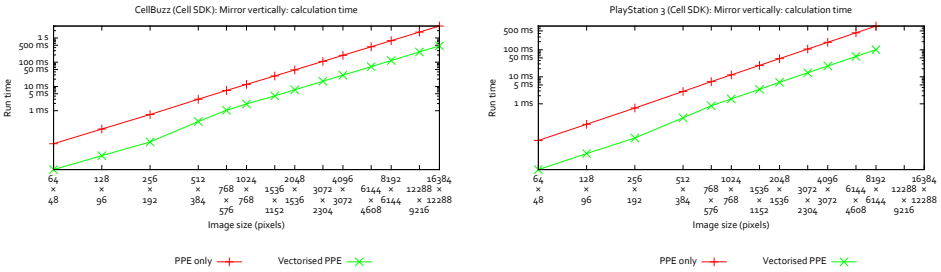
QS22 *PlayStation 3*

Figure A.8 – *Sequential implementation of the scale down filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



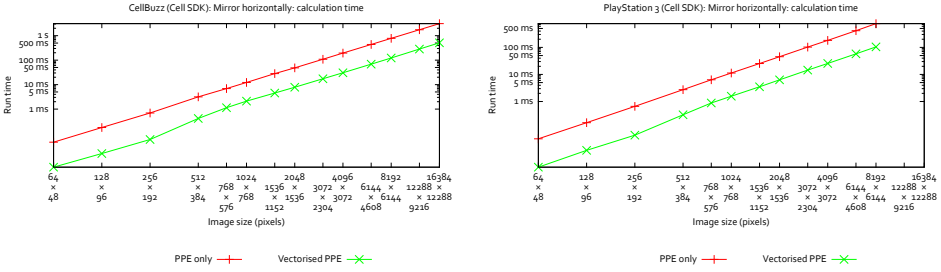
QS22 *PlayStation 3*

Figure A.9 – *Sequential implementation of the scale up filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



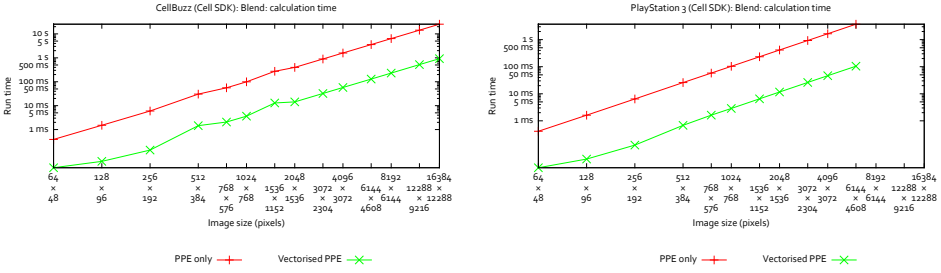
QS22 *PlayStation 3*

Figure A.10 – *Sequential implementation of the mirror vertically filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



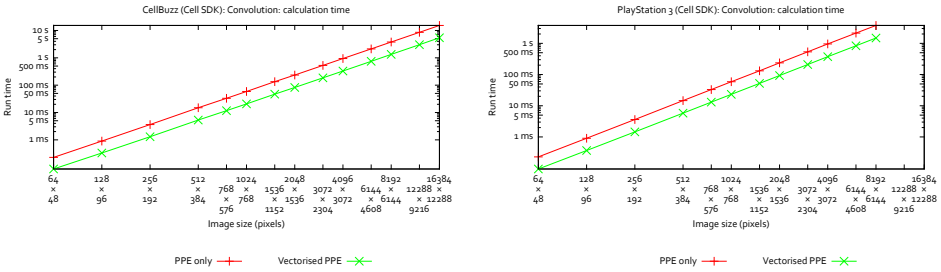
QS22 *PlayStation 3*

Figure A.11 – *Sequential implementation of the mirror horizontally filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



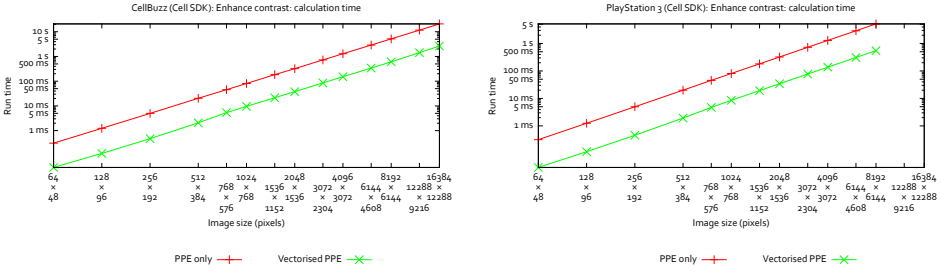
QS22 *PlayStation 3*

Figure A.12 – *Sequential implementation of the blend filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



QS22 *PlayStation 3*

Figure A.13 – *Sequential implementation of the convolution filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*

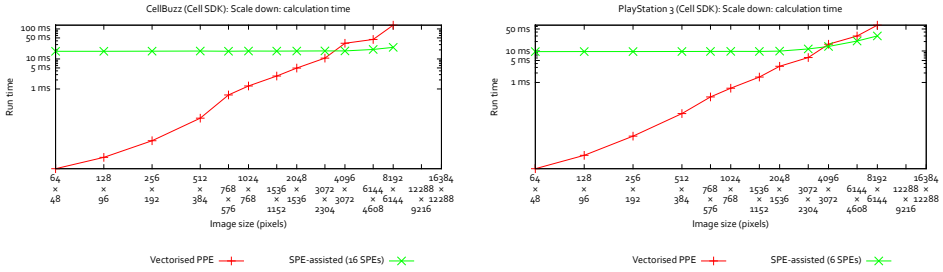


QS22 *PlayStation 3*

Figure A.14 – *Sequential implementation of the enhance contrast filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*

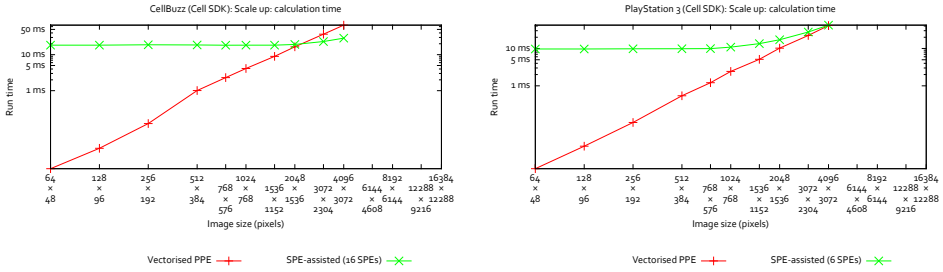
A.1.3 SPEs

In this section we show the detailed results of running the SPEs version of the Cell SDK code.



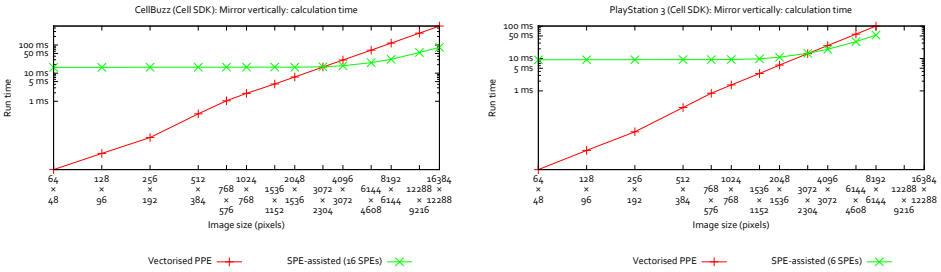
QS22 *PlayStation 3*

Figure A.15 – *SPE-assisted implementation of the scale down filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



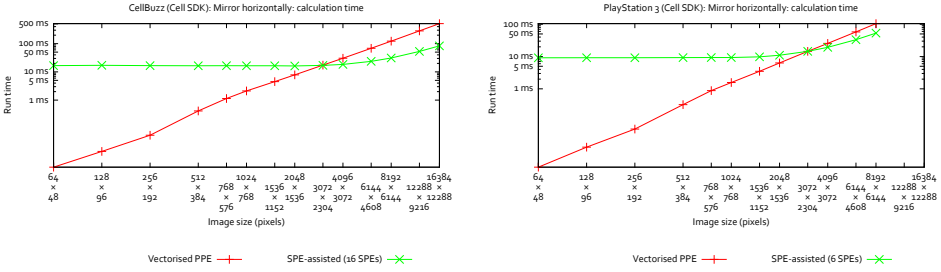
QS22 *PlayStation 3*

Figure A.16 – *SPE-assisted implementation of the scale up filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



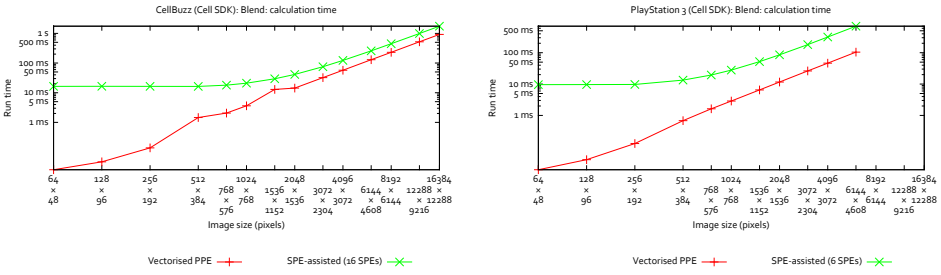
QS22 *PlayStation 3*

Figure A.17 – *SPE-assisted implementation of the mirror vertically filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



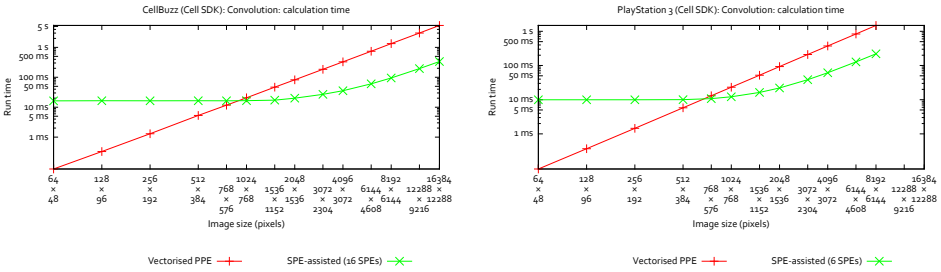
QS22 *PlayStation 3*

Figure A.18 – *SPE-assisted implementation of the mirror horizontally filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



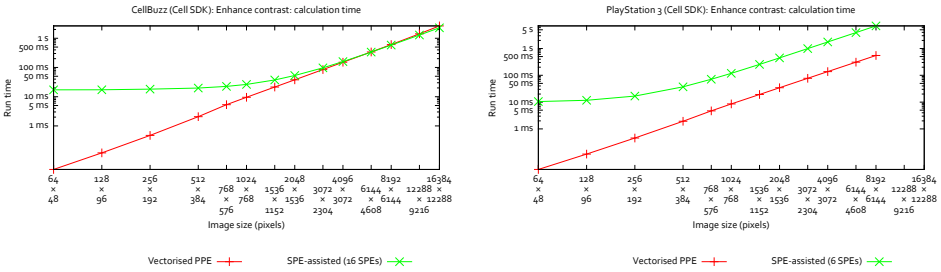
QS22 *PlayStation 3*

Figure A.19 – *SPE-assisted implementation of the blend filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



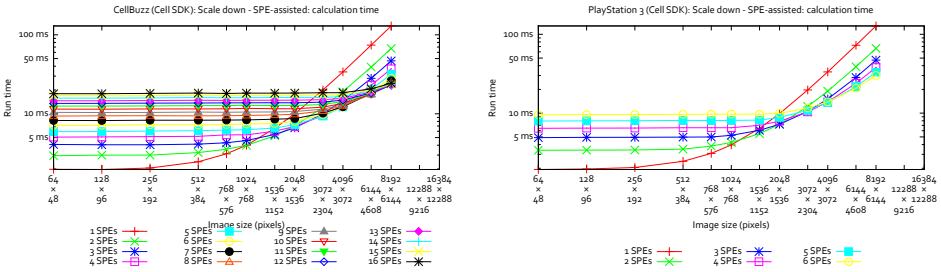
QS22 *PlayStation 3*

Figure A.20 – *SPE-assisted implementation of the convolution filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



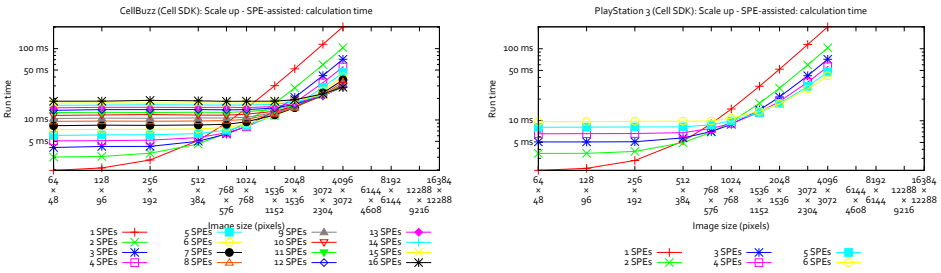
QS22 *PlayStation 3*

Figure A.21 – *SPE-assisted implementation of the enhance contrast filter, compared to the PPE vectorised implementation. Note that the scales of these plots differ.*



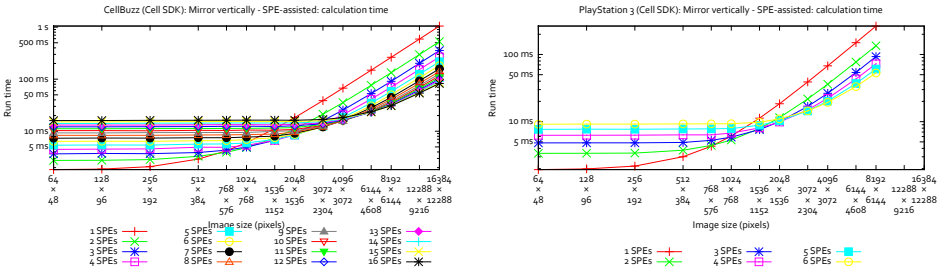
QS22 *PlayStation 3*

Figure A.22 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the scale down filter. Note that the scales of these plots differ.*



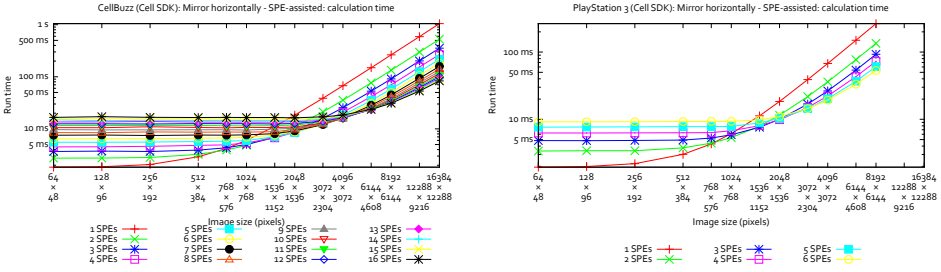
QS22 *PlayStation 3*

Figure A.23 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the scale up filter. Note that the scales of these plots differ.*



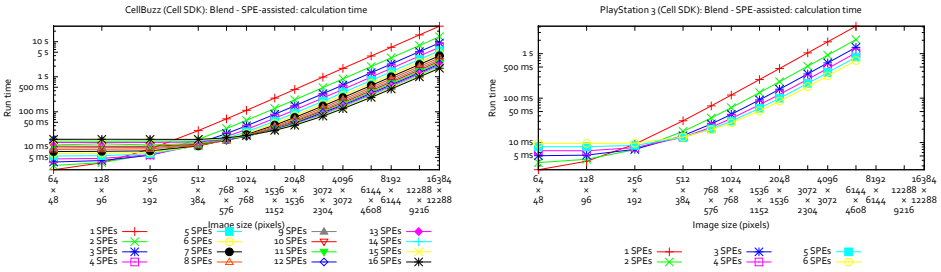
QS22 *PlayStation 3*

Figure A.24 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the mirror vertically filter. Note that the scales of these plots differ.*



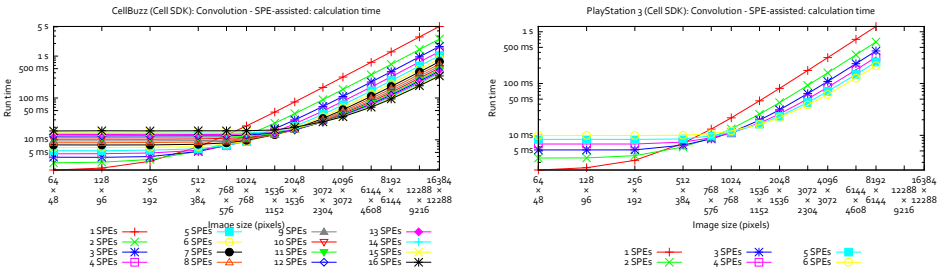
QS22 *PlayStation 3*

Figure A.25 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the mirror horizontally filter. Note that the scales of these plots differ.*



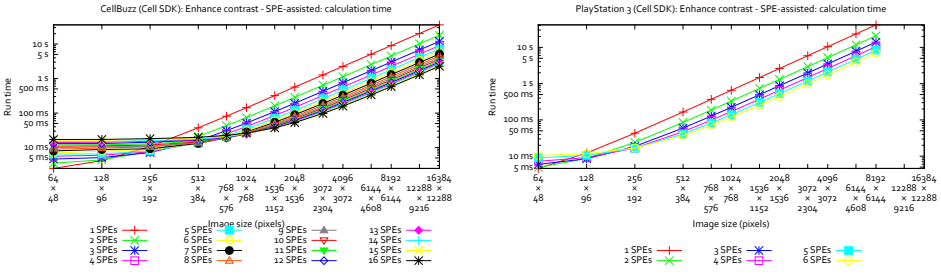
QS22 *PlayStation 3*

Figure A.26 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the blend filter. Note that the scales of these plots differ.*



QS22 *PlayStation 3*

Figure A.27 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the convolution filter. Note that the scales of these plots differ.*

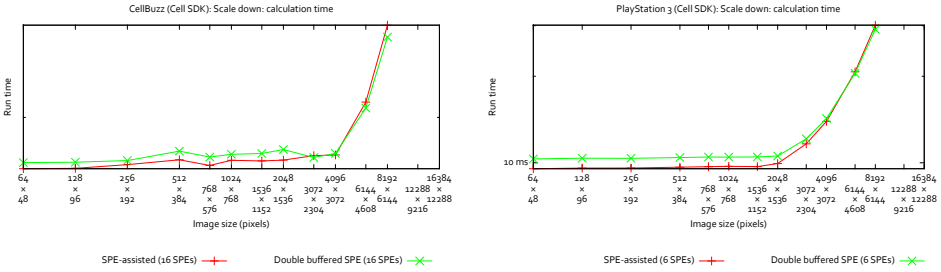


QS22 *PlayStation 3*

Figure A.28 – *A comparison of the number of SPEs used for the SPE-assisted implementation of the enhance contrast filter. Note that the scales of these plots differ.*

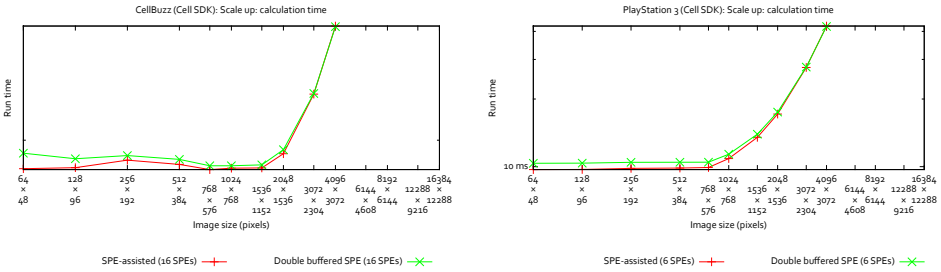
A.1.4 SPEs double buffered

In this section we show the detailed results of running the SPEs double buffered version of the Cell SDK code.



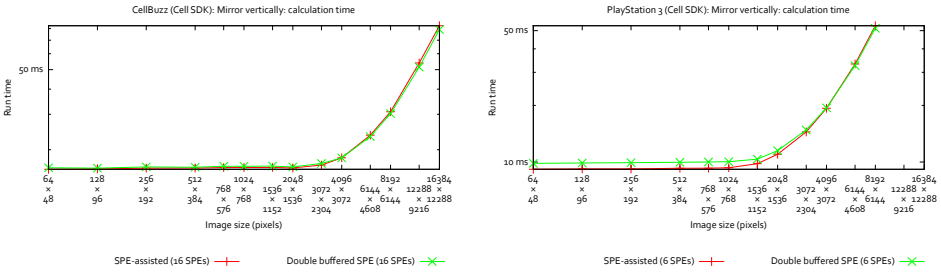
QS22 *PlayStation 3*

Figure A.29 – *SPE-assisted implementation of the scale down filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



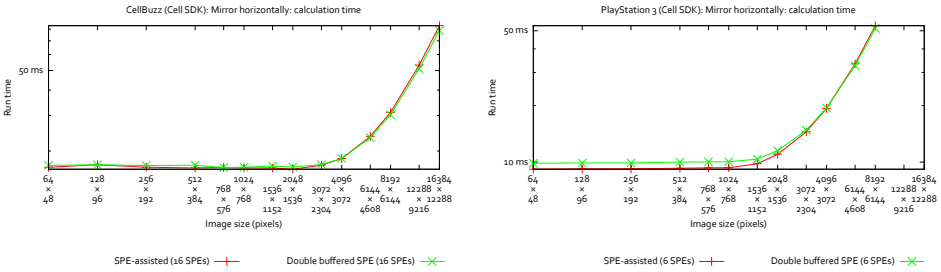
QS22 *PlayStation 3*

Figure A.30 – *SPE-assisted implementation of the scale up filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



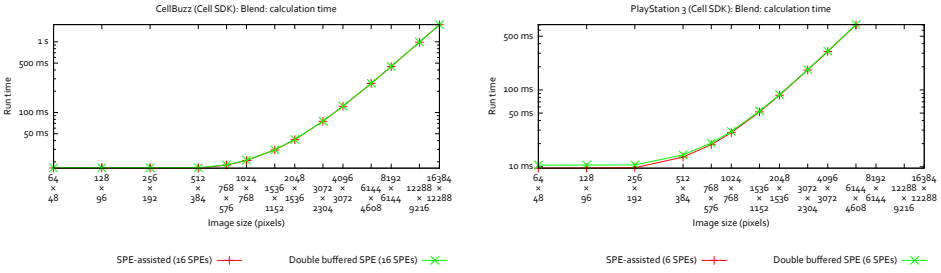
QS22 *PlayStation 3*

Figure A.31 – *SPE-assisted implementation of the mirror vertically filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



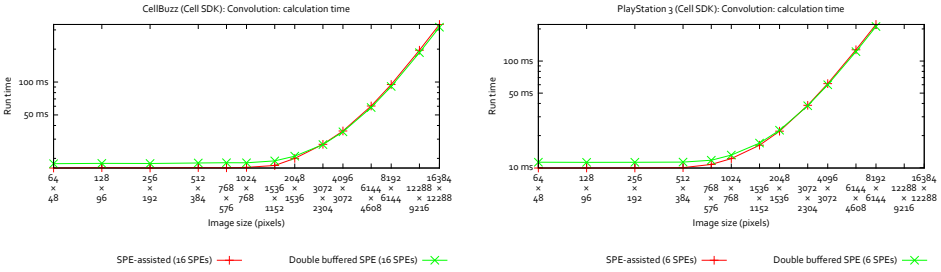
QS22 *PlayStation 3*

Figure A.32 – *SPE-assisted implementation of the mirror horizontally filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



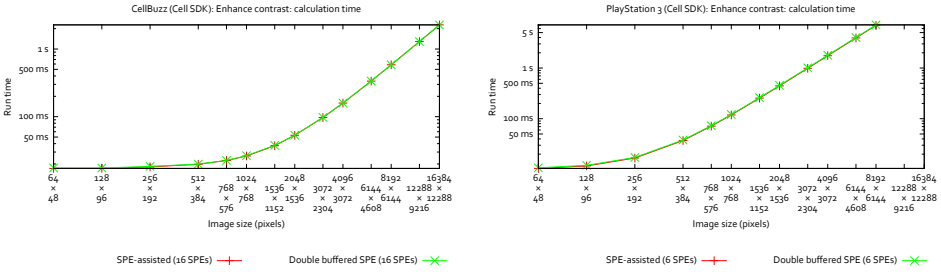
QS22 *PlayStation 3*

Figure A.33 – *SPE-assisted implementation of the blend filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



QS22 *PlayStation 3*

Figure A.34 – *SPE-assisted implementation of the convolution filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*

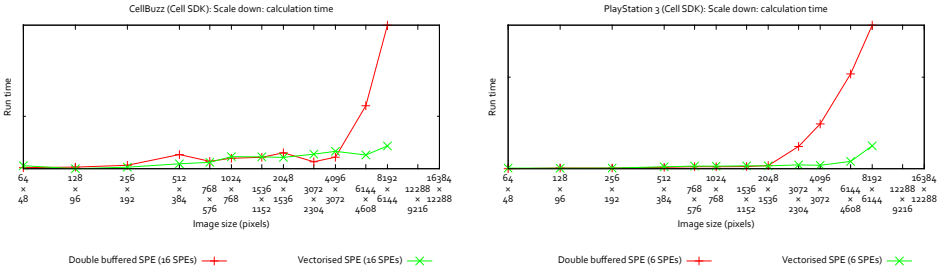


QS22 *PlayStation 3*

Figure A.35 – *SPE-assisted implementation of the enhance contrast filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*

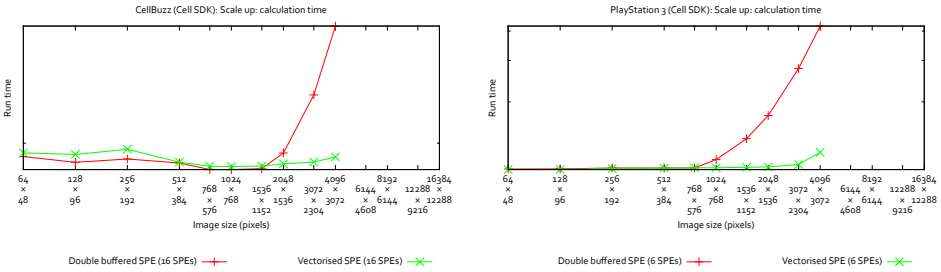
A.1.5 SPEs vectorised

In this section we show the detailed results of running the SPEs vectorised version of the Cell SDK code.



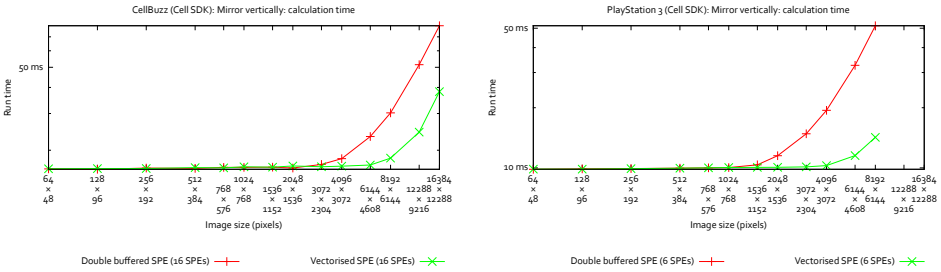
QS22 *PlayStation 3*

Figure A.36 – *SPE vectorised implementation of the scale down filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



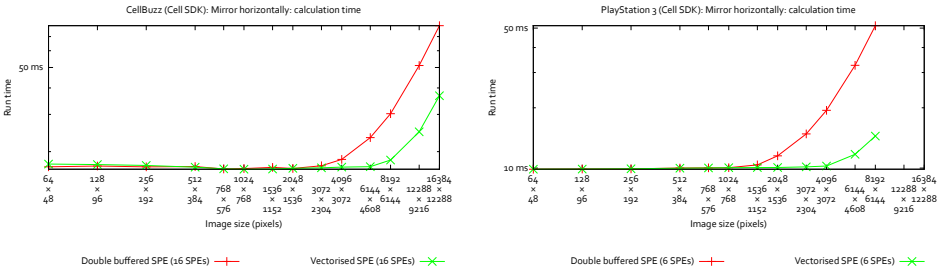
QS22 *PlayStation 3*

Figure A.37 – *SPE vectorised implementation of the scale up filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



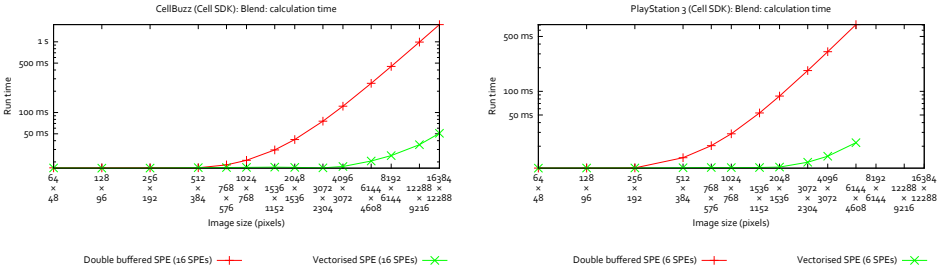
QS22 *PlayStation 3*

Figure A.38 – *SPE vectorised implementation of the mirror vertically filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



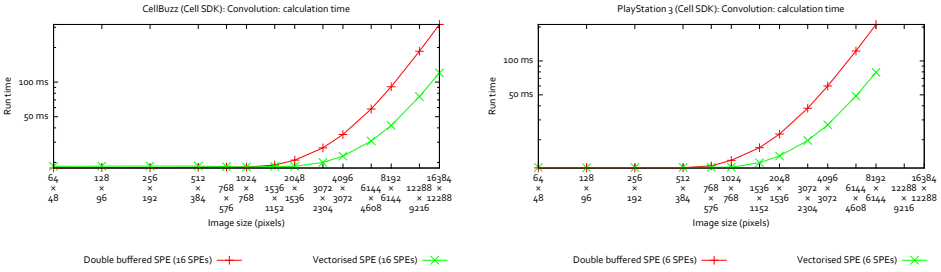
QS22 *PlayStation 3*

Figure A.39 – *SPE vectorised implementation of the mirror horizontally filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



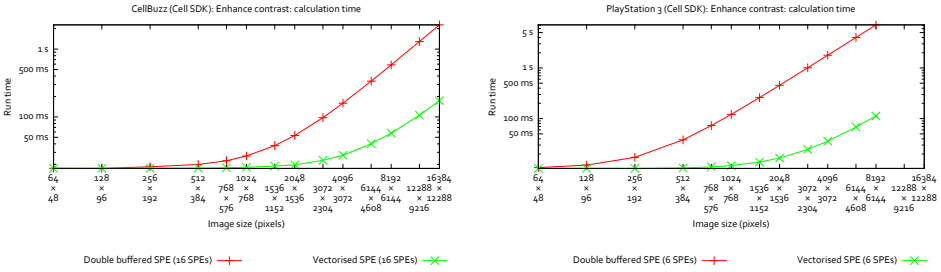
QS22 *PlayStation 3*

Figure A.40 – *SPE vectorised implementation of the blend filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*



QS22 *PlayStation 3*

Figure A.41 – *SPE vectorised implementation of the convolution filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*

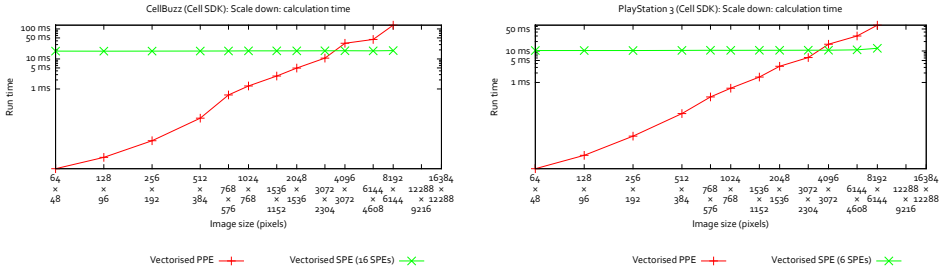


QS22 *PlayStation 3*

Figure A.42 – *SPE vectorised implementation of the enhance contrast filter, compared to the SPEs double buffered implementation. Note that the scales of these plots differ.*

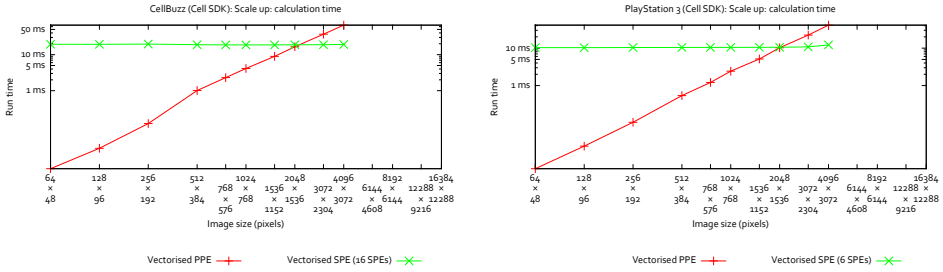
A.1.6 PPE and SPEs

In this section we show a detailed comparison of the PPE vectorised and SPEs vectorised versions of the Cell SDK code.



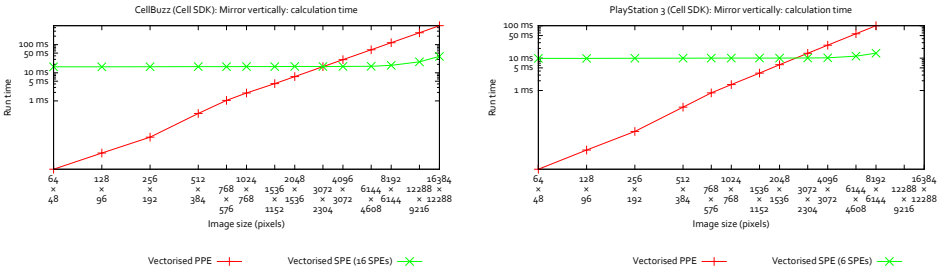
QS22 *PlayStation 3*

Figure A.43 – *PPE and SPE vectorised implementations of the scale down filter compared. Note that the scales of these plots differ.*



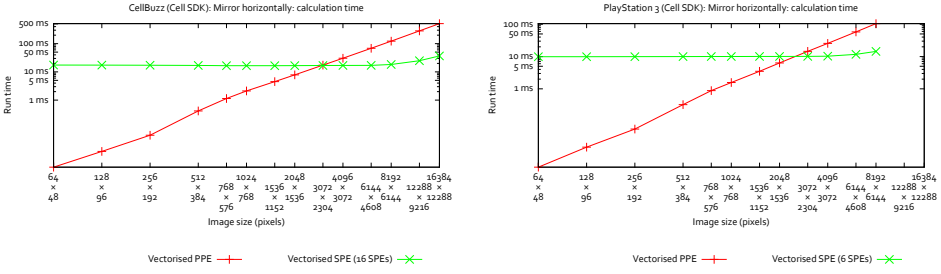
QS22 *PlayStation 3*

Figure A.44 – *PPE and SPE vectorised implementations of the scale up filter compared. Note that the scales of these plots differ.*



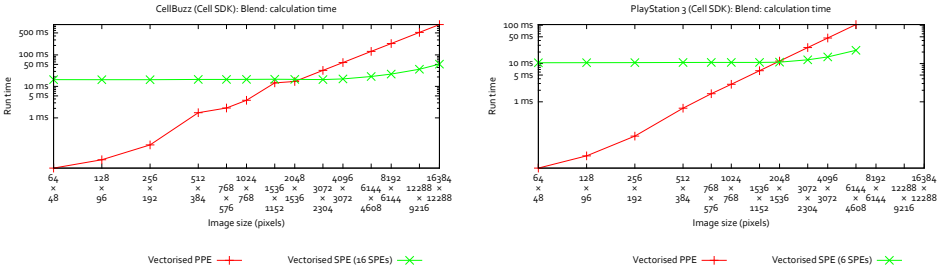
QS22 *PlayStation 3*

Figure A.45 – *PPE and SPE vectorised implementations of the mirror vertically filter compared. Note that the scales of these plots differ.*



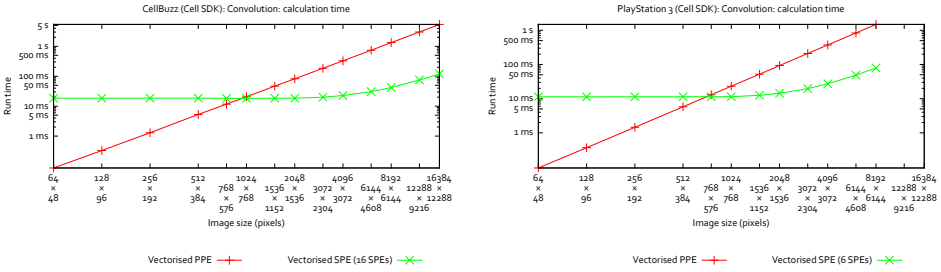
QS22 *PlayStation 3*

Figure A.46 – *PPE and SPE vectorised implementations of the mirror horizontally filter compared. Note that the scales of these plots differ.*



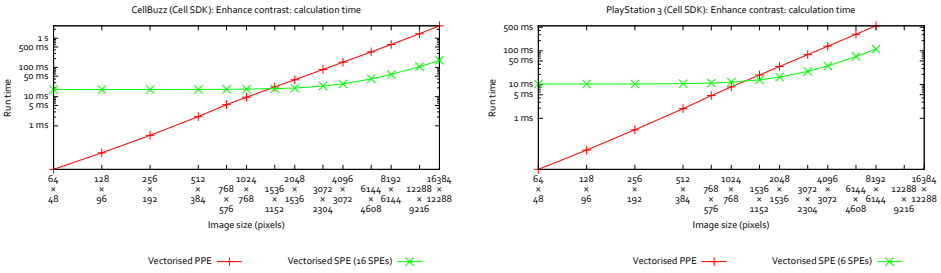
QS22 *PlayStation 3*

Figure A.47 – *PPE and SPE vectorised implementations of the blend filter compared. Note that the scales of these plots differ.*



QS22 *PlayStation 3*

Figure A.48 – *PPE and SPE vectorised implementation of the convolution filter compared. Note that the scales of these plots differ.*



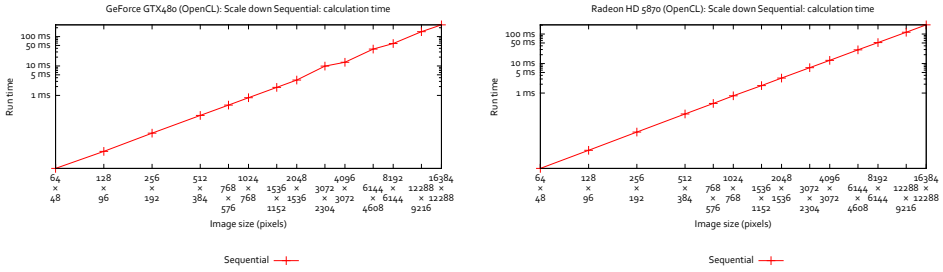
QS22 *PlayStation 3*

Figure A.49 – *PPE and SPE vectorised implementations of the enhance contrast filter compared. Note that the scales of these plots differ.*

A.2 OpenCL

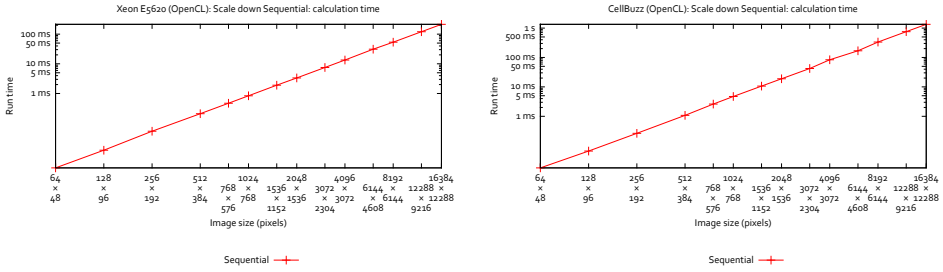
A.2.1 Fully sequential

In this section we show the detailed results of running the fully sequential version of the OpenCL code.



GeForce GTX480

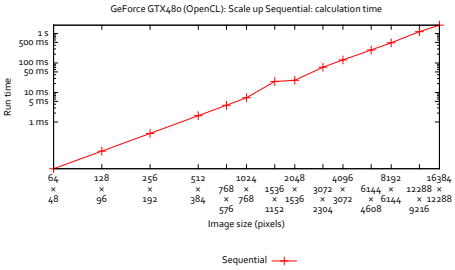
Radeon HD 5870



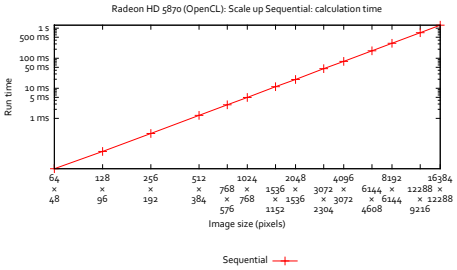
Xeon E5620

Cell BE

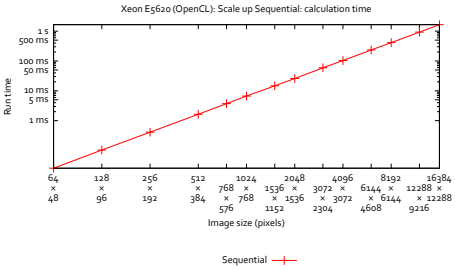
Figure A.50 – *Sequential implementation of the scale down filter. Note that the scales of these plots differ.*



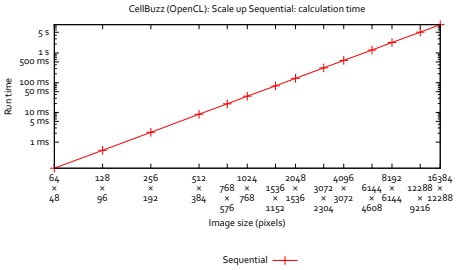
GeForce GTX480



Radeon HD 5870

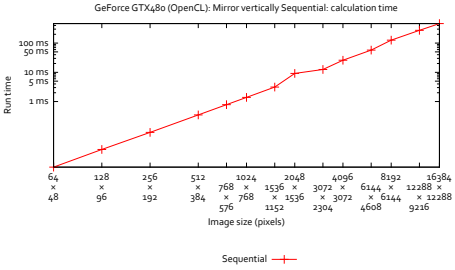


Xeon E5620

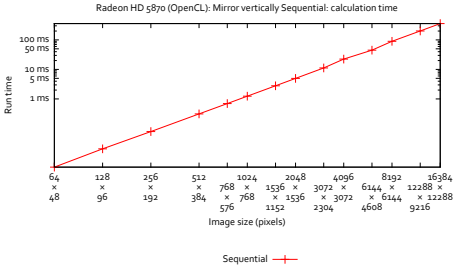


Cell BE

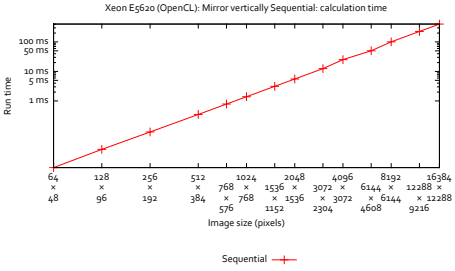
Figure A.51 – *Sequential implementation of the scale up filter. Note that the scales of these plots differ.*



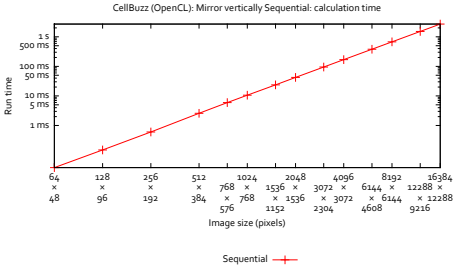
GeForce GTX480



Radeon HD 5870

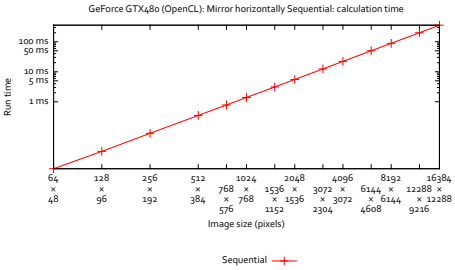


Xeon E5620

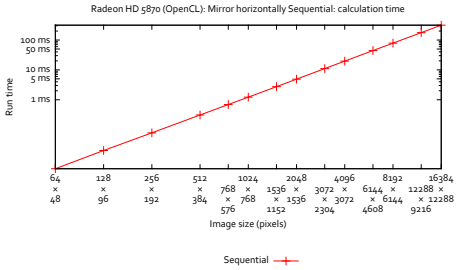


Cell BE

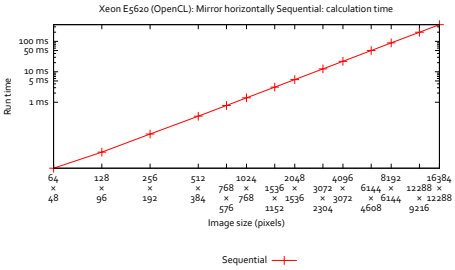
Figure A.52 – *Sequential implementation of the mirror vertically filter. Note that the scales of these plots differ.*



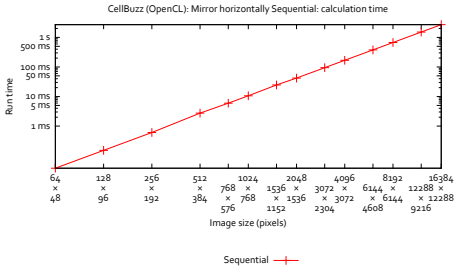
GeForce GTX480



Radeon HD 5870

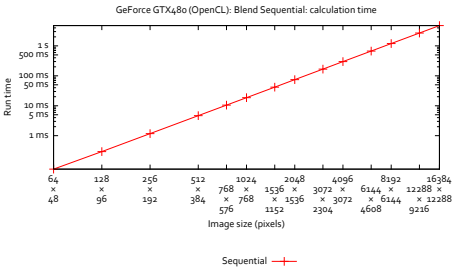


Xeon E5620

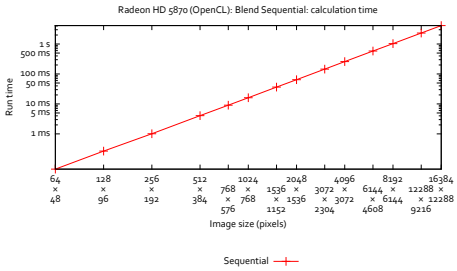


Cell BE

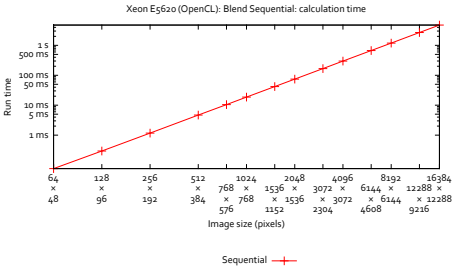
Figure A.53 – *Sequential implementation of the mirror horizontally filter. Note that the scales of these plots differ.*



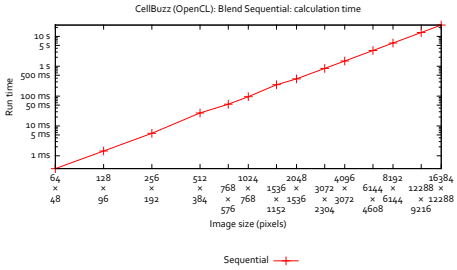
GeForce GTX480



Radeon HD 5870

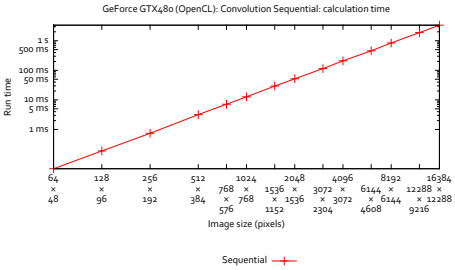


Xeon E5620

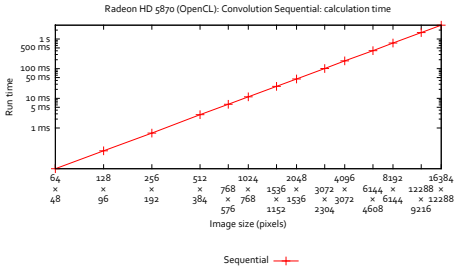


Cell BE

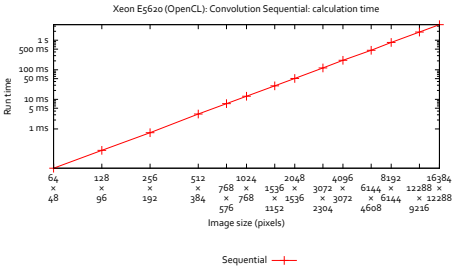
Figure A.54 – *Sequential implementation of the blend filter. Note that the scales of these plots differ.*



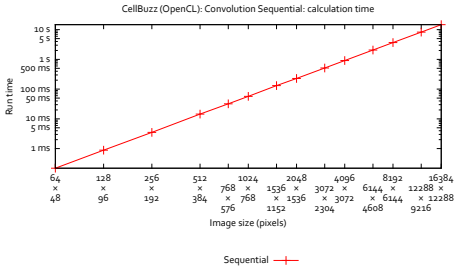
GeForce GTX480



Radeon HD 5870

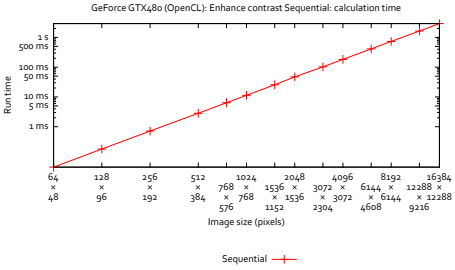


Xeon E5620

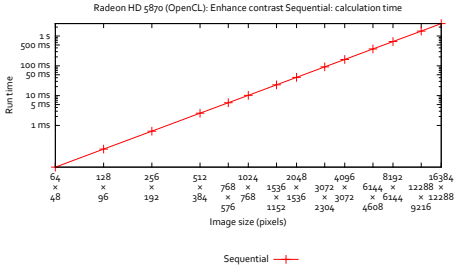


Cell BE

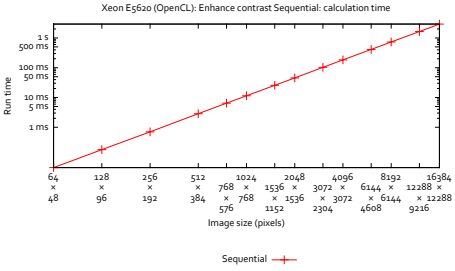
Figure A.55 – *Sequential implementation of the convolution filter. Note that the scales of these plots differ.*



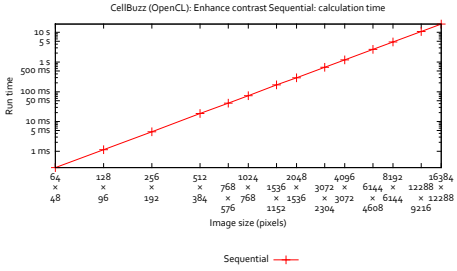
GeForce GTX480



Radeon HD 5870



Xeon E5620

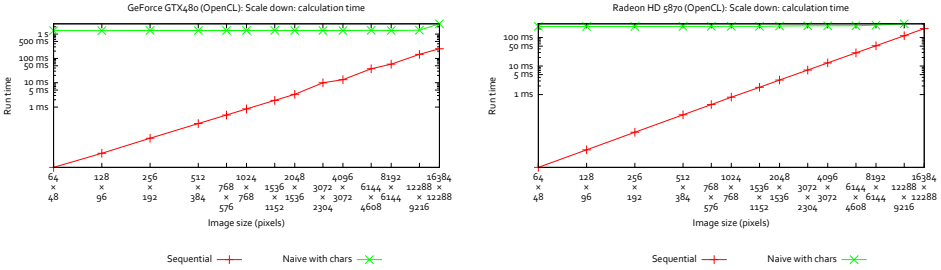


Cell BE

Figure A.56 – *Sequential implementation of the enhance contrast filter. Note that the scales of these plots differ.*

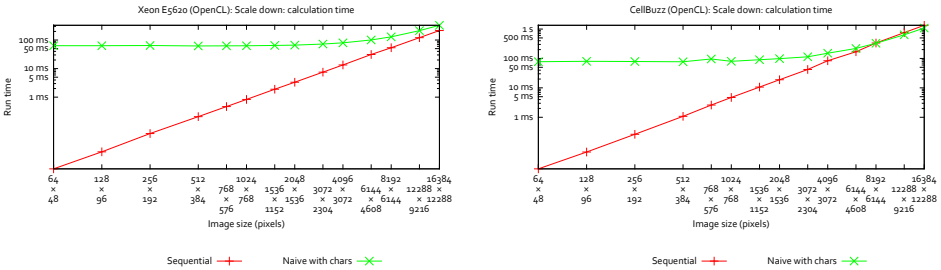
A.2.2 Naive implementation using `char`

In this section we show the detailed results of running the naive implementation using `char` of the OpenCL code.



GeForce GTX480

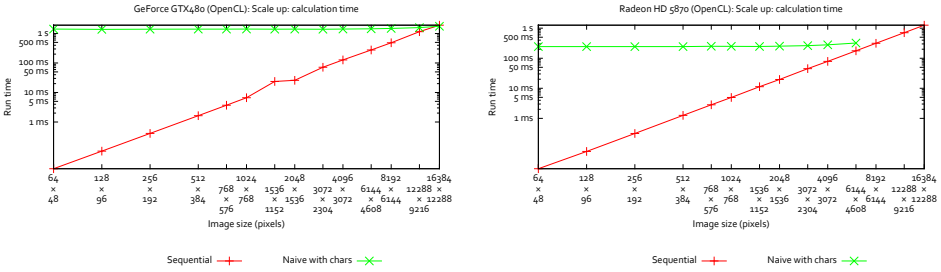
Radeon HD 5870



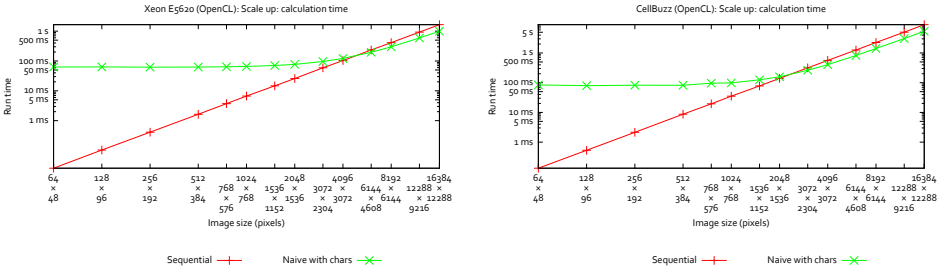
Xeon E5620

Cell BE

Figure A.57 – *Naive implementation of the scale down filter, compared to the sequential implementation. Note that the scales of these plots differ.*

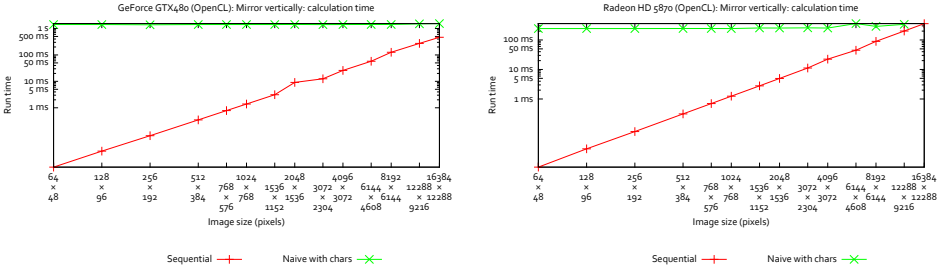


GeForce GTX480 *Radeon HD 5870*

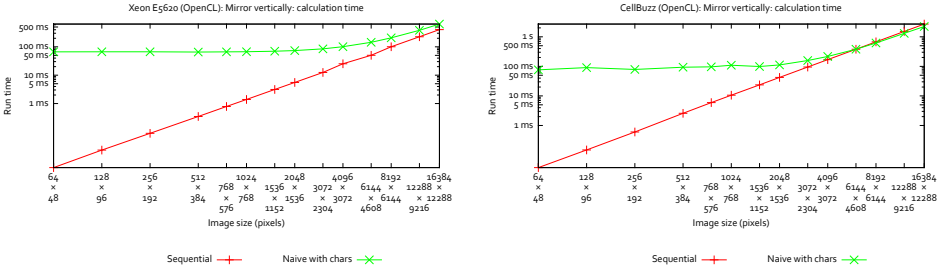


Xeon E5620 *Cell BE*

Figure A.58 – *Naive implementation of the scale up filter, compared to the sequential implementation. Note that the scales of these plots differ.*

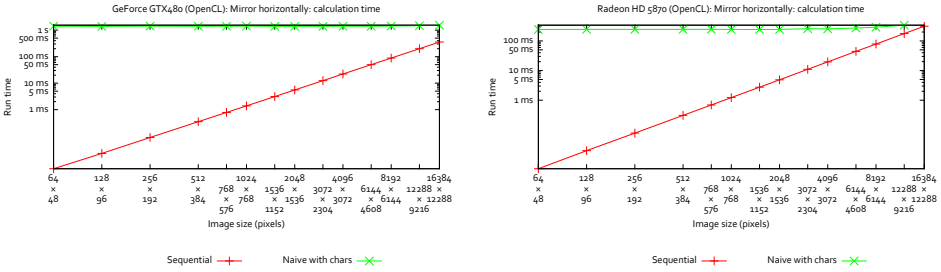


GeForce GTX480 *Radeon HD 5870*

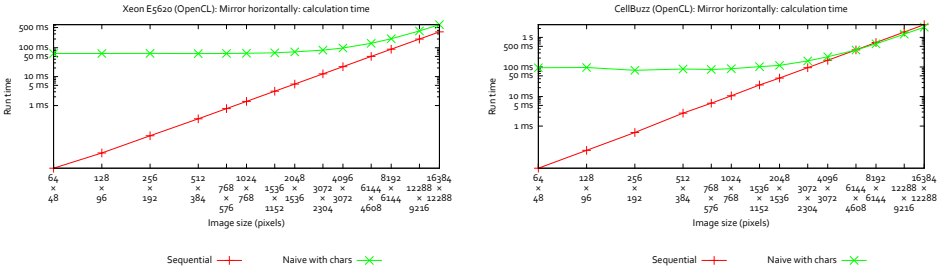


Xeon E5620 *Cell BE*

Figure A.59 – *Naive implementation of the mirror vertically filter, compared to the sequential implementation. Note that the scales of these plots differ.*

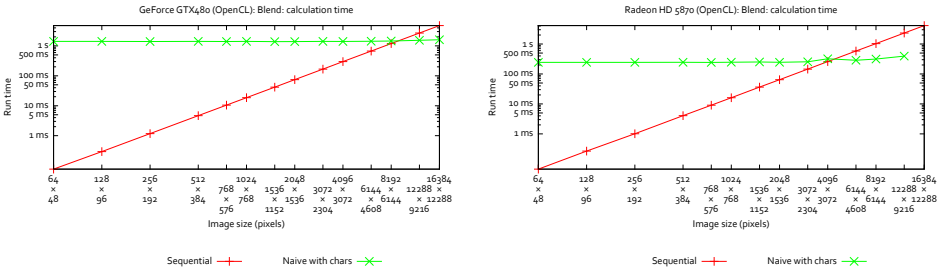


GeForce GTX480 *Radeon HD 5870*

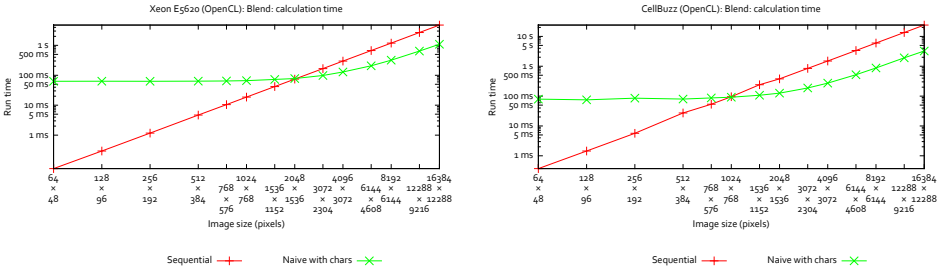


Xeon E5620 *Cell BE*

Figure A.60 – *Naive implementation of the mirror horizontally filter, compared to the sequential implementation. Note that the scales of these plots differ.*

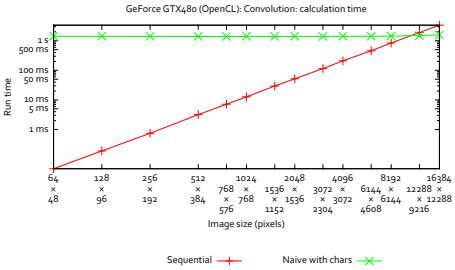


GeForce GTX480 *Radeon HD 5870*

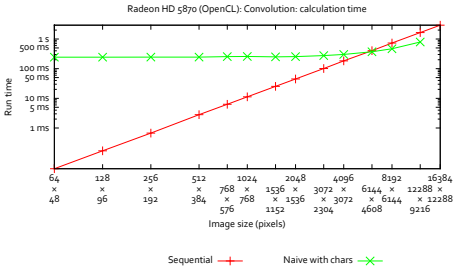


Xeon E5620 *Cell BE*

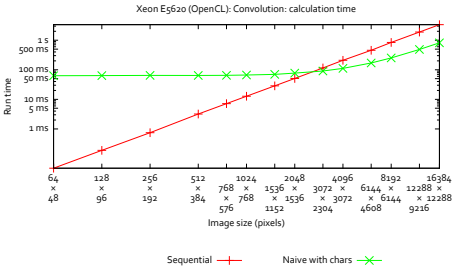
Figure A.61 – *Naive implementation of the blend filter, compared to the sequential implementation. Note that the scales of these plots differ.*



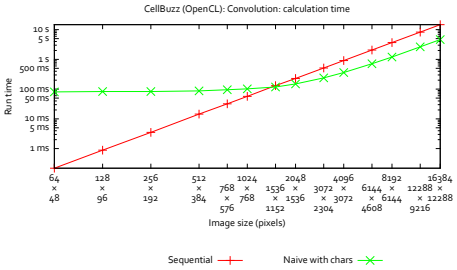
GeForce GTX480



Radeon HD 5870

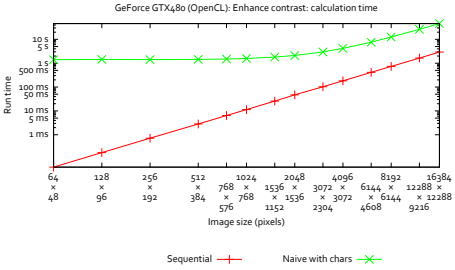


Xeon E5620

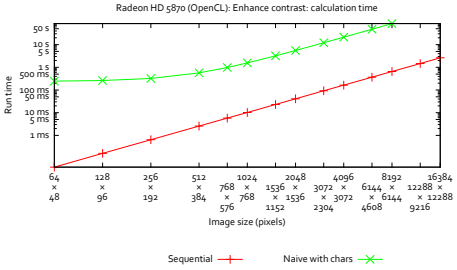


Cell BE

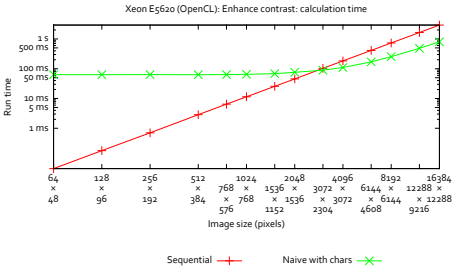
Figure A.62 – *Naive implementation of the convolution filter, compared to the sequential implementation. Note that the scales of these plots differ.*



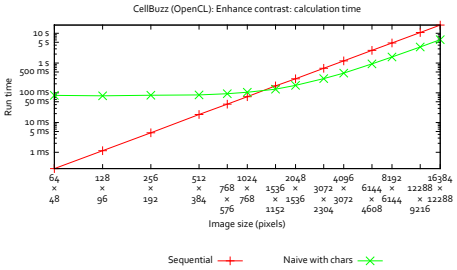
GeForce GTX480



Radeon HD 5870

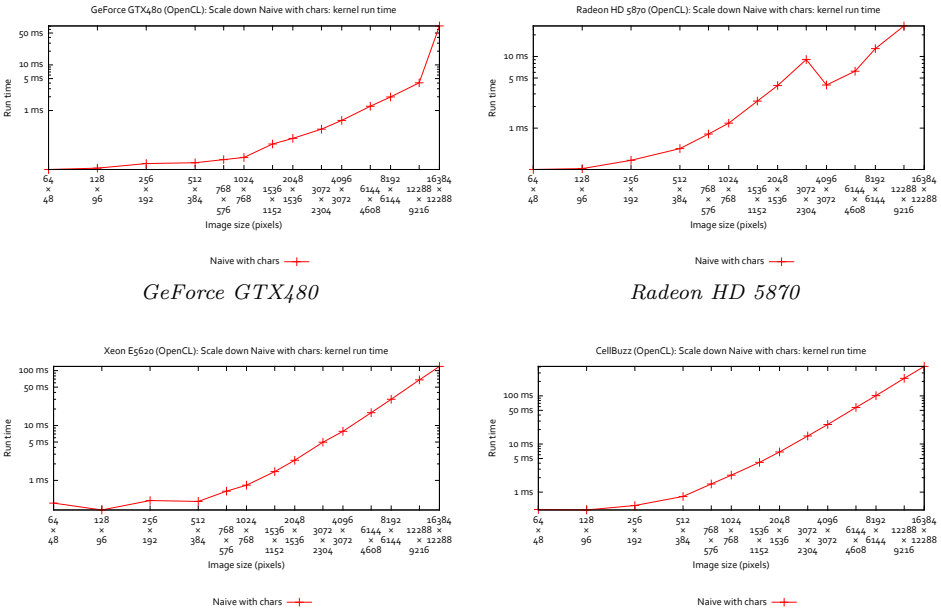


Xeon E5620



Cell BE

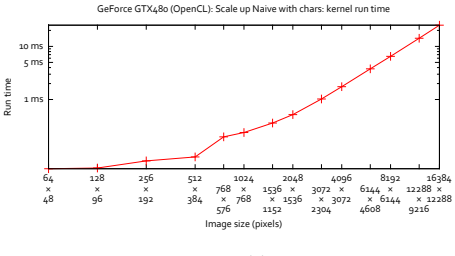
Figure A.63 – *Naive implementation of the enhance contrast filter, compared to the sequential implementation. Note that the scales of these plots differ.*



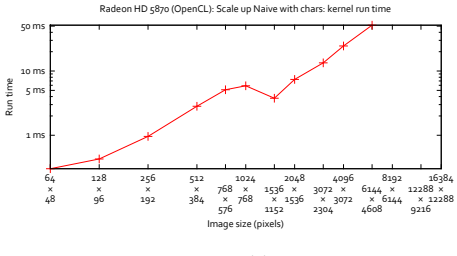
GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

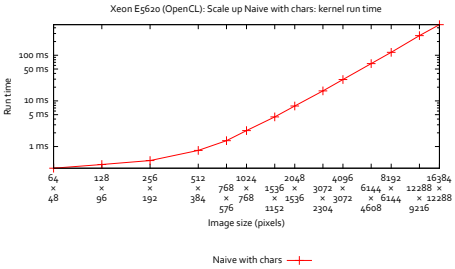
Figure A.64 – *Naive implementation of the scale down filter, kernel part of the run time. Note that the scales of these plots differ.*



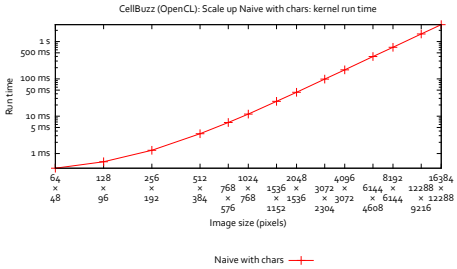
GeForce GTX480



Radeon HD 5870

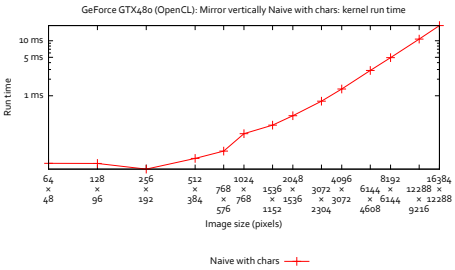


Xeon E5620

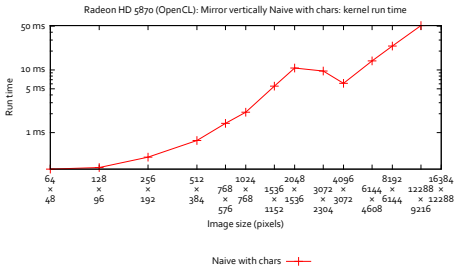


Cell BE

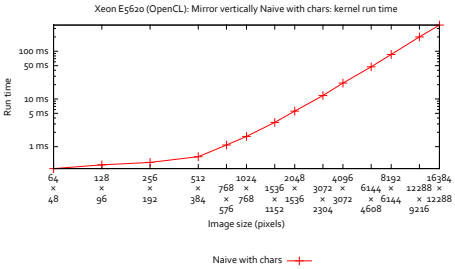
Figure A.65 – *Naive implementation of the scale up filter, kernel part of the run time. Note that the scales of these plots differ.*



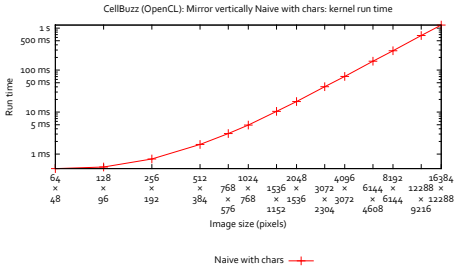
GeForce GTX480



Radeon HD 5870

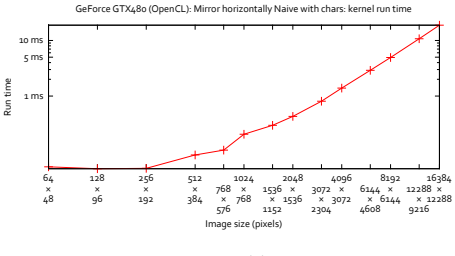


Xeon E5620

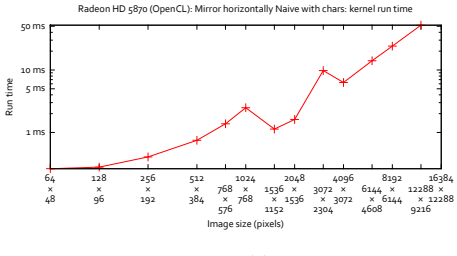


Cell BE

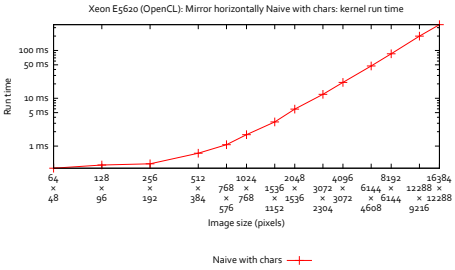
Figure A.66 – *Naive implementation of the mirror vertically filter, kernel part of the run time. Note that the scales of these plots differ.*



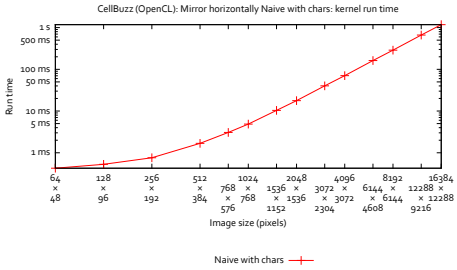
GeForce GTX480



Radeon HD 5870

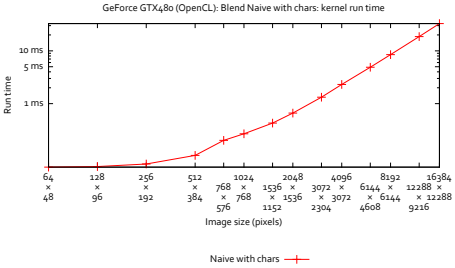


Xeon E5620

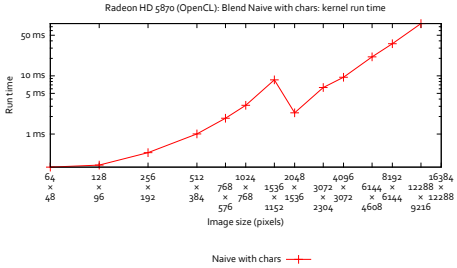


Cell BE

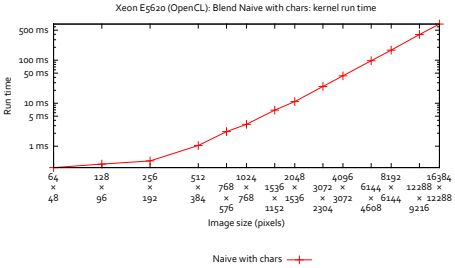
Figure A.67 – *Naive implementation of the mirror horizontally filter, kernel part of the run time. Note that the scales of these plots differ.*



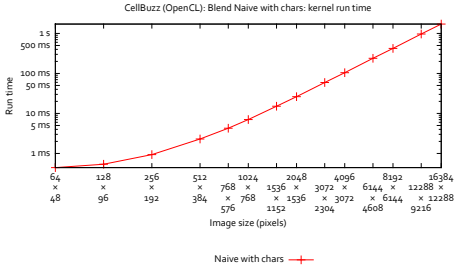
GeForce GTX480



Radeon HD 5870

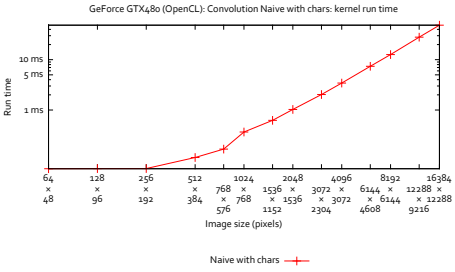


Xeon E5620

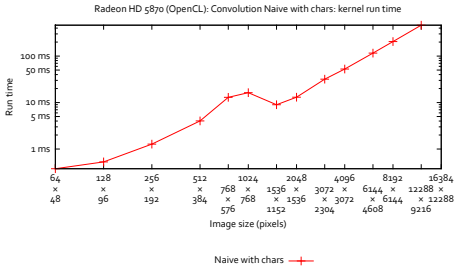


Cell BE

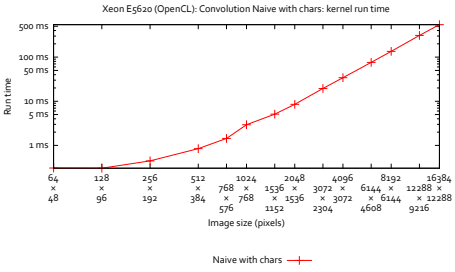
Figure A.68 – *Naive implementation of the blend filter, kernel part of the run time. Note that the scales of these plots differ.*



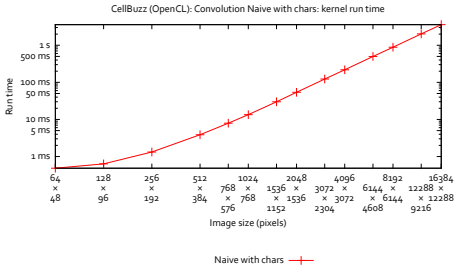
GeForce GTX480



Radeon HD 5870

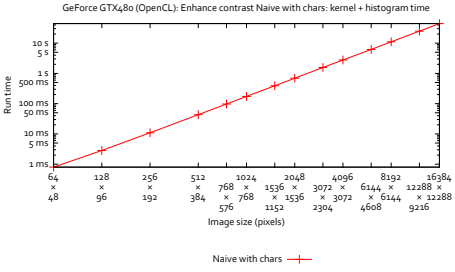


Xeon E5620

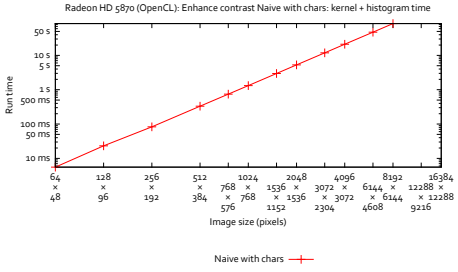


Cell BE

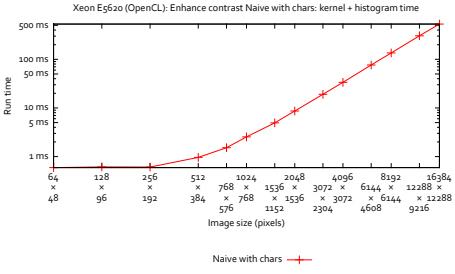
Figure A.69 – *Naive implementation of the convolution filter, kernel part of the run time. Note that the scales of these plots differ.*



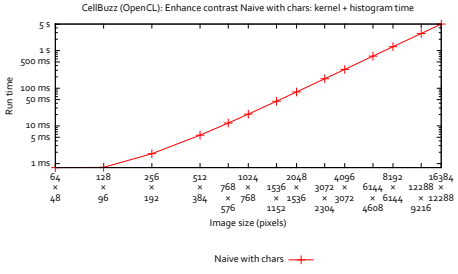
GeForce GTX480



Radeon HD 5870

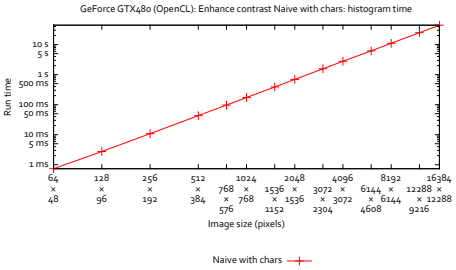


Xeon E5620

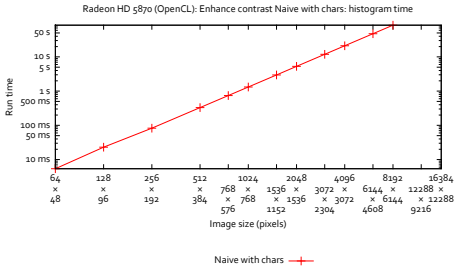


Cell BE

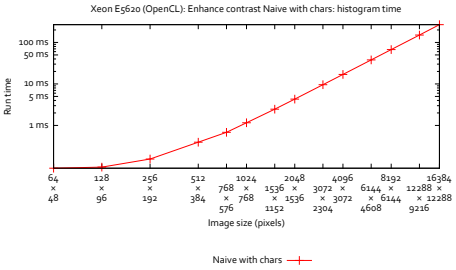
Figure A.70 – *Naive implementation of the enhance contrast filter, kernel part of the run time. Note that the scales of these plots differ.*



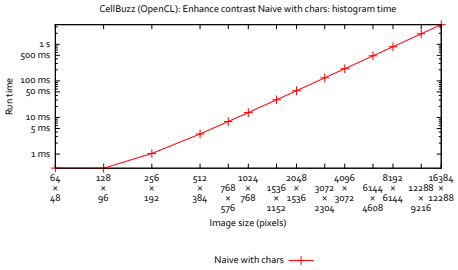
GeForce GTX480



Radeon HD 5870

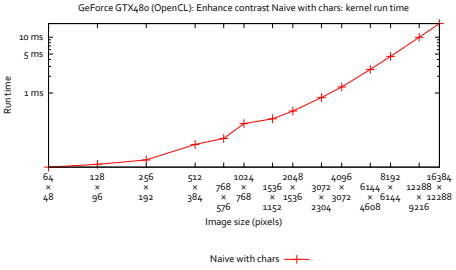


Xeon E5620

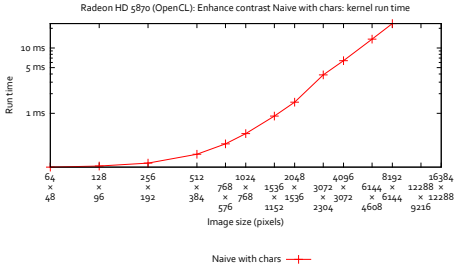


Cell BE

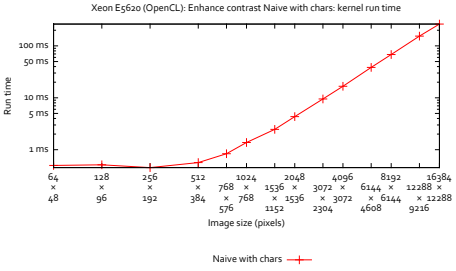
Figure A.71 – *Naive implementation of the enhance contrast filter, histogram calculation part of the run time. Note that the scales of these plots differ.*



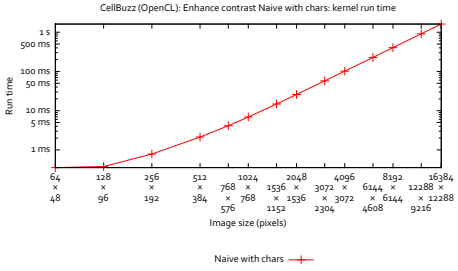
GeForce GTX480



Radeon HD 5870



Xeon E5620

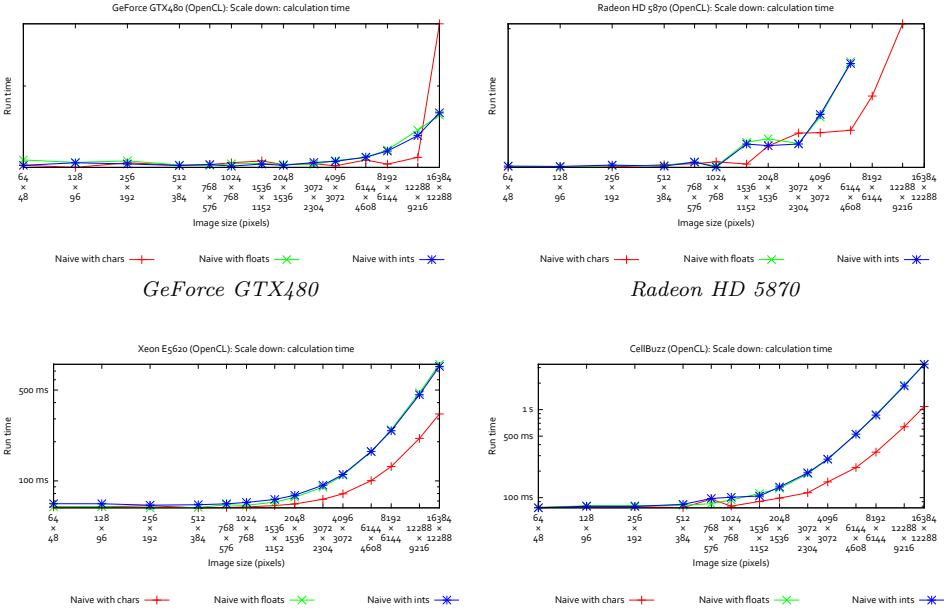


Cell BE

Figure A.72 – *Naive implementation of the enhance contrast filter, normalisation part of the run time. Note that the scales of these plots differ.*

A.2.3 Naive implementation using other data types

In this section we show the detailed results of running the naive implementation using other data types of the OpenCL code.



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

Figure A.73 – *Different data types compared, for the scale down filter. Note that the scales of these plots differ.*

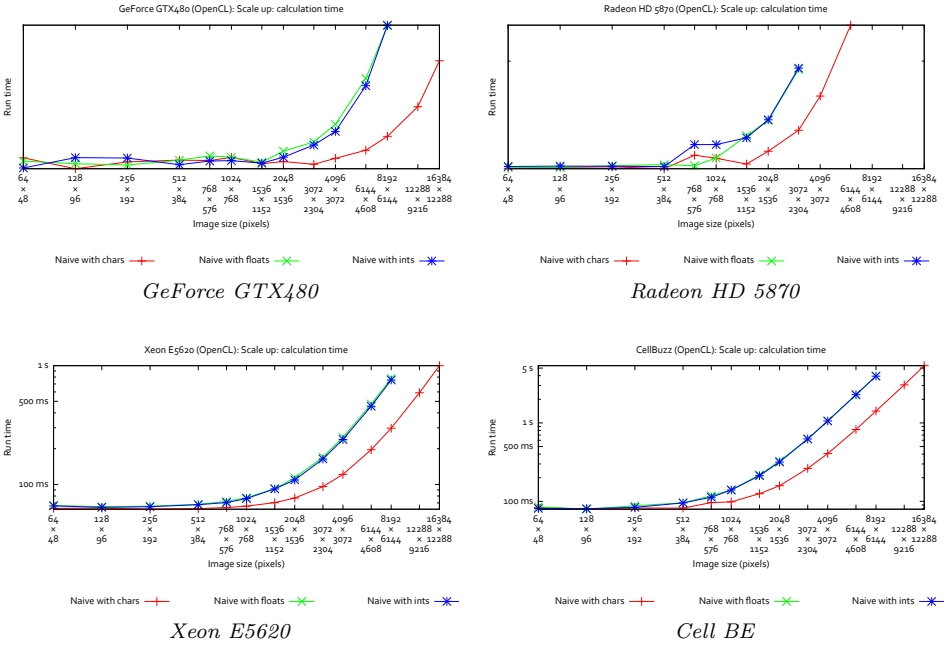


Figure A.74 – *Different data types compared, for the scale up filter. Note that the scales of these plots differ.*

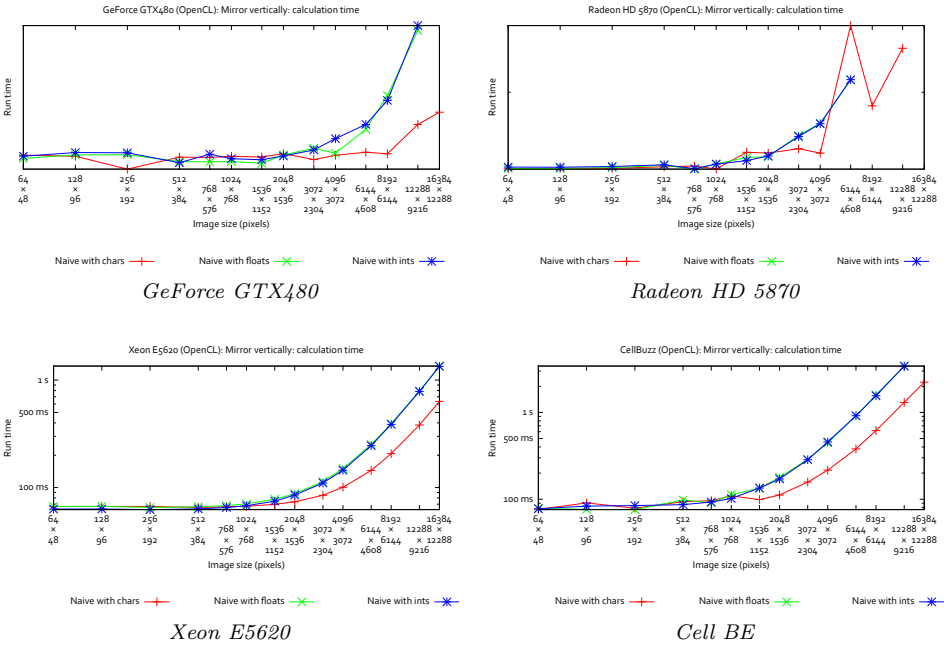


Figure A.75 – *Different data types compared, for the mirror vertically filter. Note that the scales of these plots differ.*

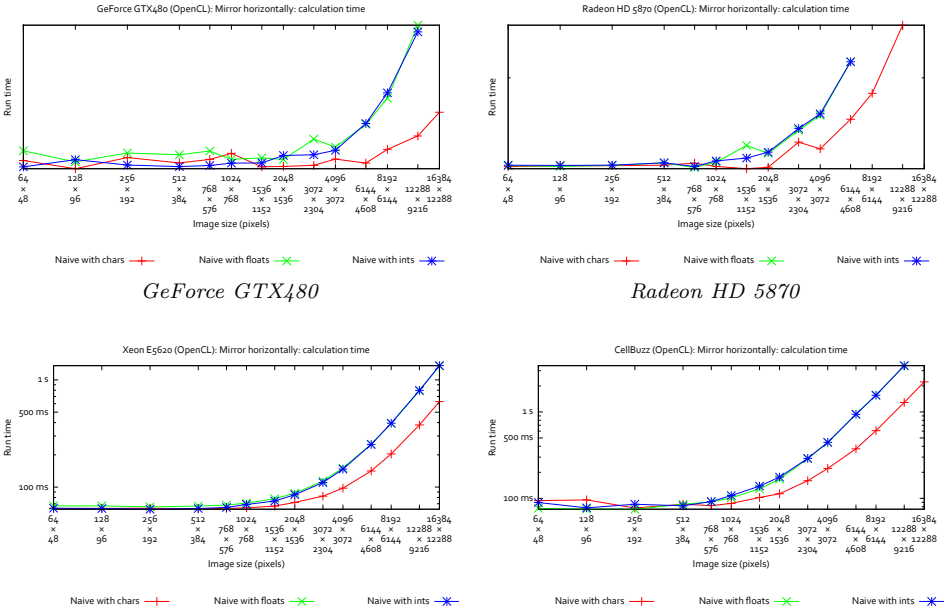


Figure A.76 – *Different data types compared, for the mirror horizontally filter. Note that the scales of these plots differ.*

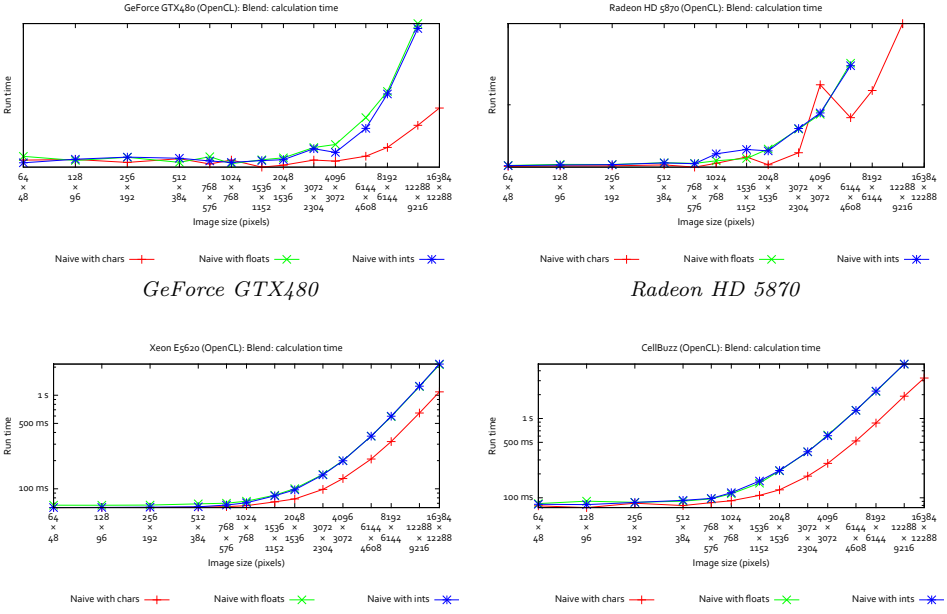
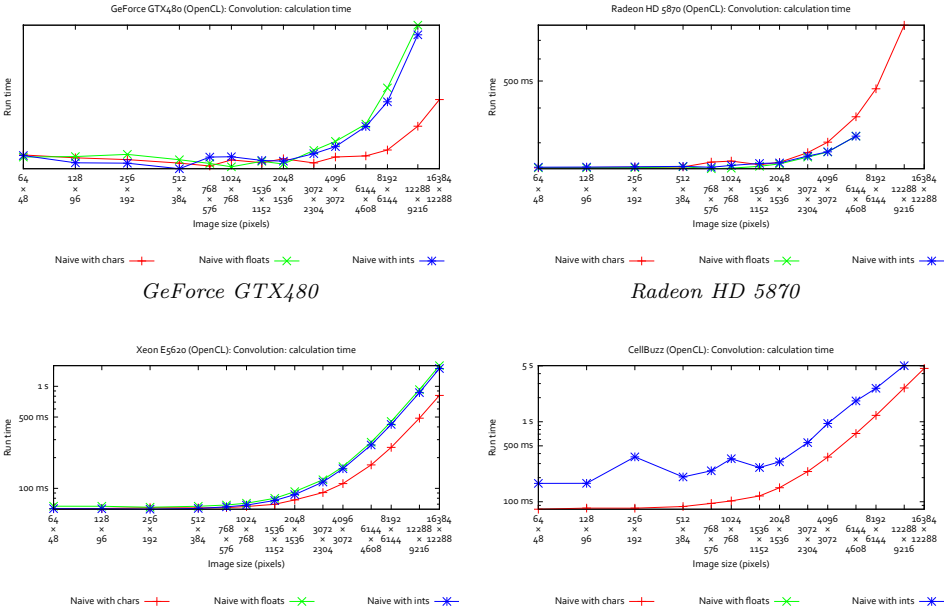


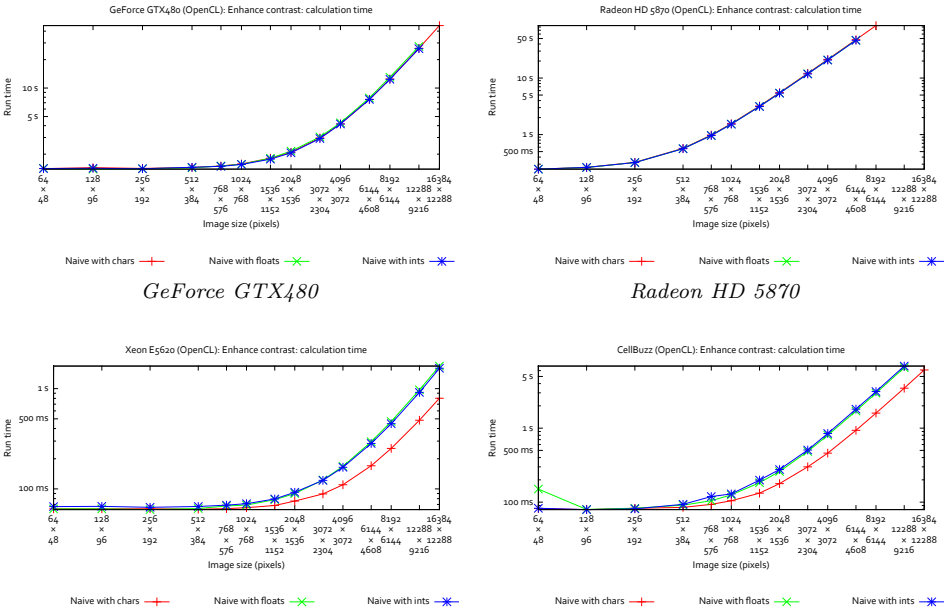
Figure A.77 – *Different data types compared, for the blend filter. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

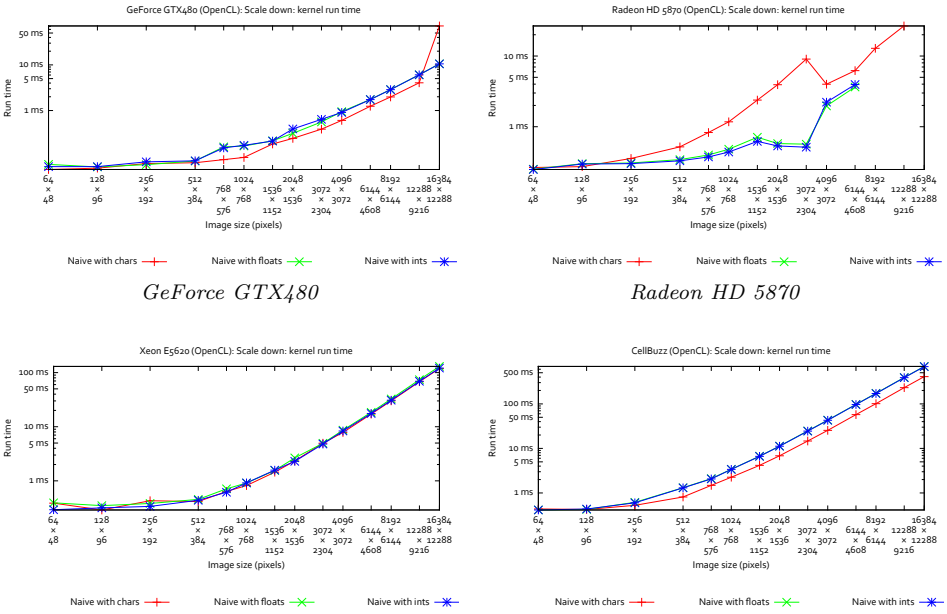
Figure A.78 – *Different data types compared, for the convolution filter. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

Figure A.79 – *Different data types compared, for the enhance contrast filter. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

Figure A.80 – *Different data types compared, for the scale down filter, kernel run time only. Note that the scales of these plots differ.*

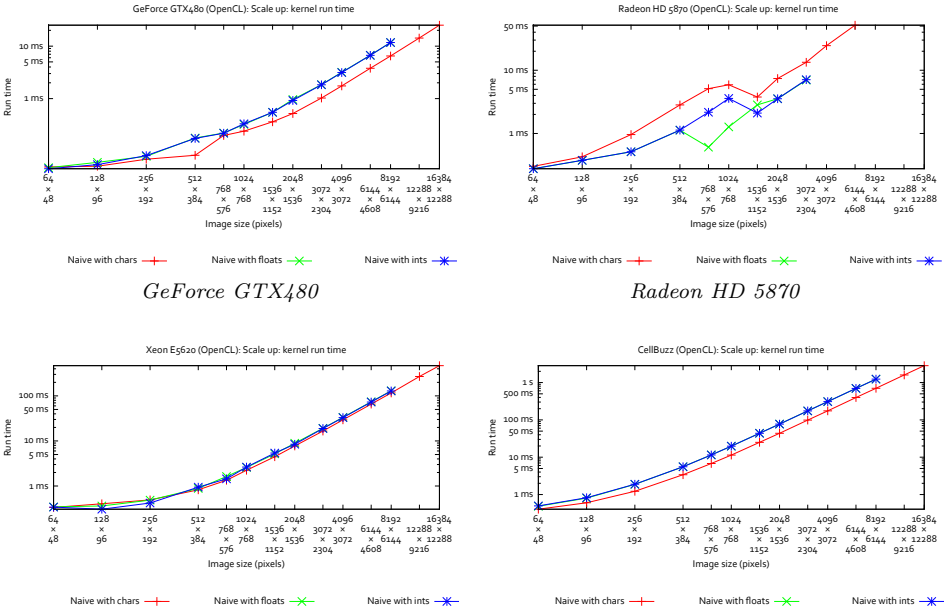


Figure A.81 – *Different data types compared, for the scale up filter, kernel run time only. Note that the scales of these plots differ.*

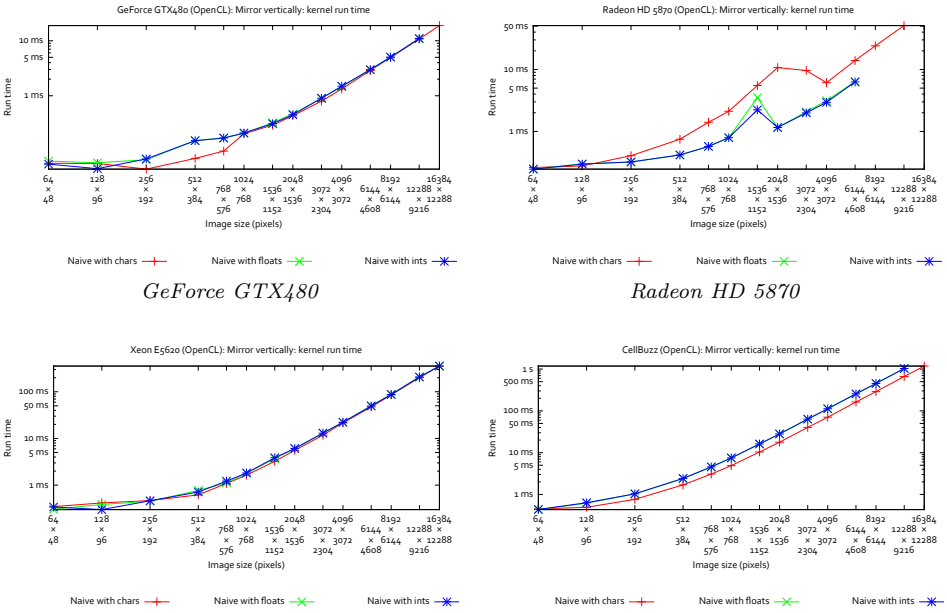
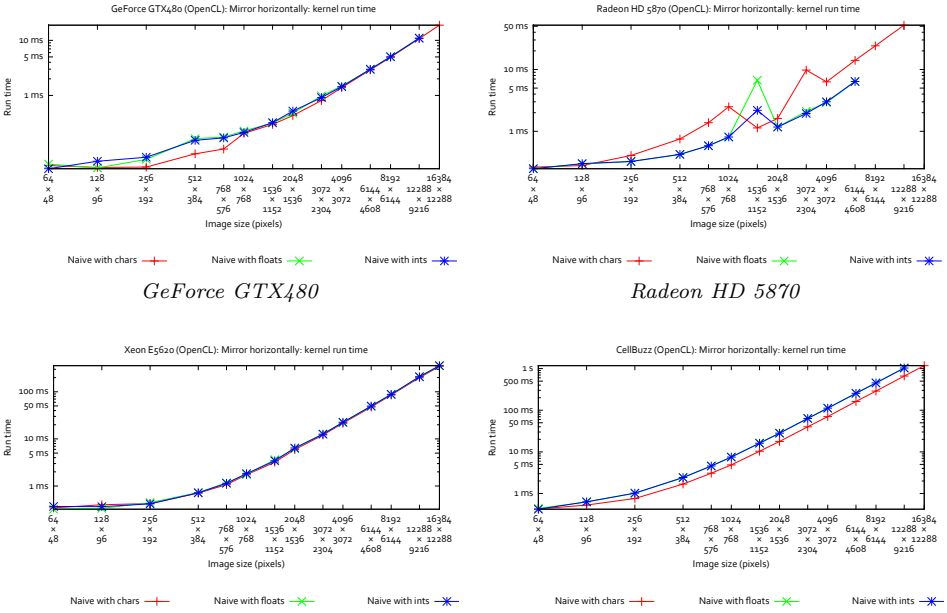


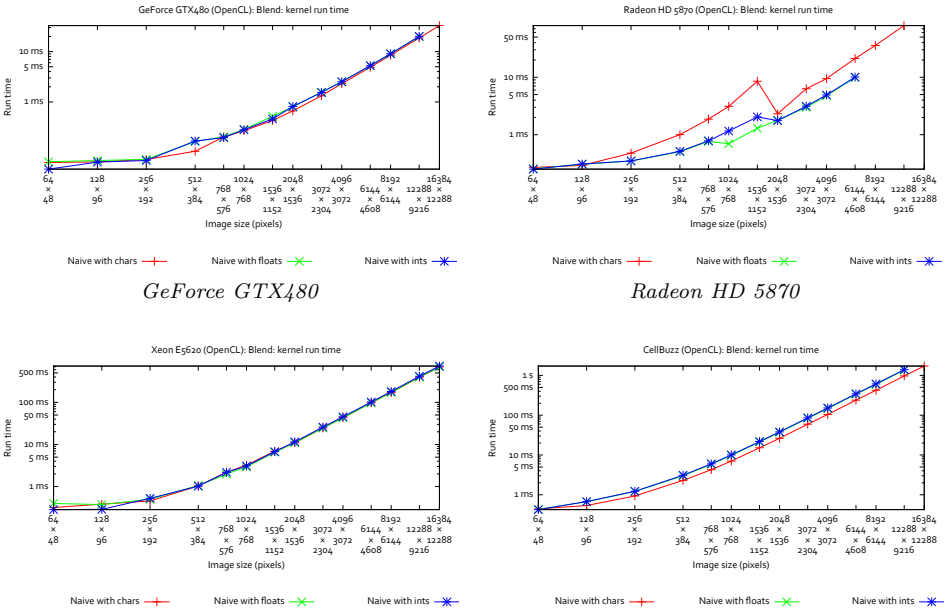
Figure A.82 – *Different data types compared, for the mirror vertically filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

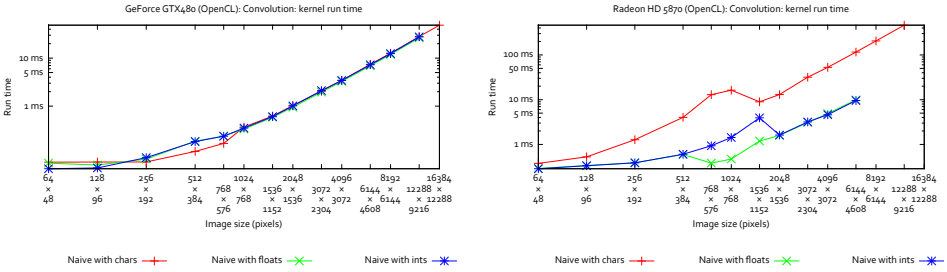
Figure A.83 – *Different data types compared, for the mirror horizontally filter, kernel run time only. Note that the scales of these plots differ.*



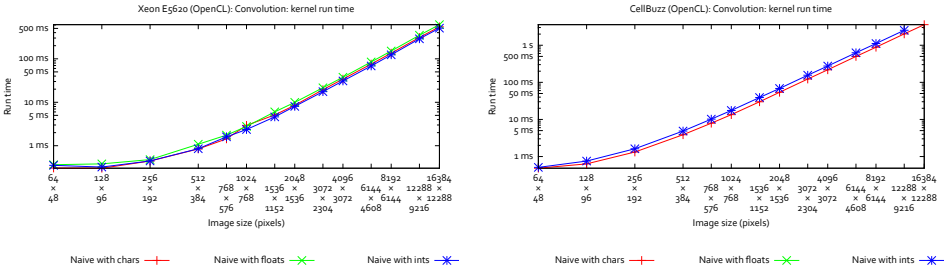
GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

Figure A.84 – *Different data types compared, for the blend filter, kernel run time only. Note that the scales of these plots differ.*

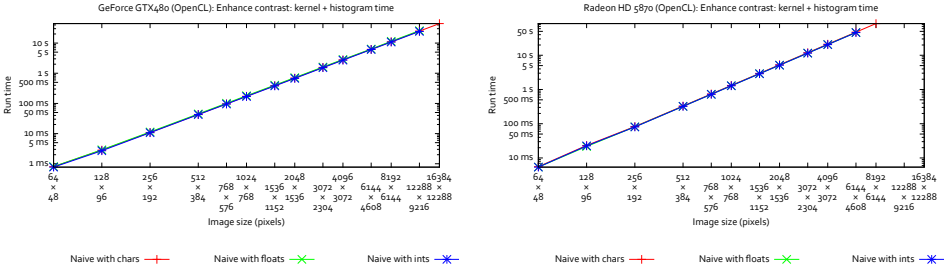


GeForce GTX480 *Radeon HD 5870*

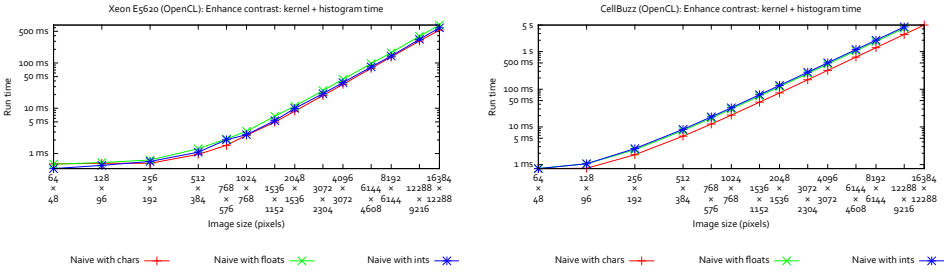


Xeon E5620 *Cell BE*

Figure A.85 – *Different data types compared, for the convolution filter, kernel run time only. Note that the scales of these plots differ.*

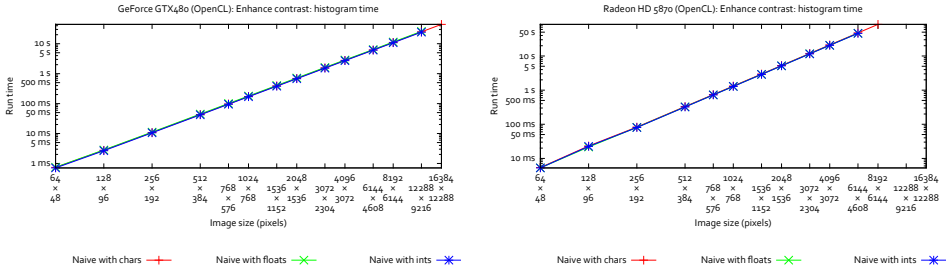


GeForce GTX480 *Radeon HD 5870*



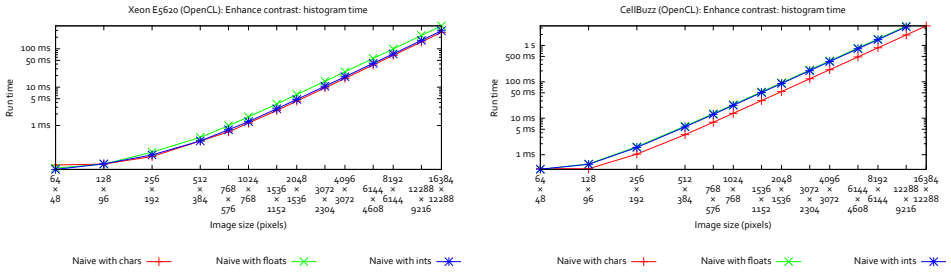
Xeon E5620 *Cell BE*

Figure A.86 – *Different data types compared, for the enhance contrast filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480

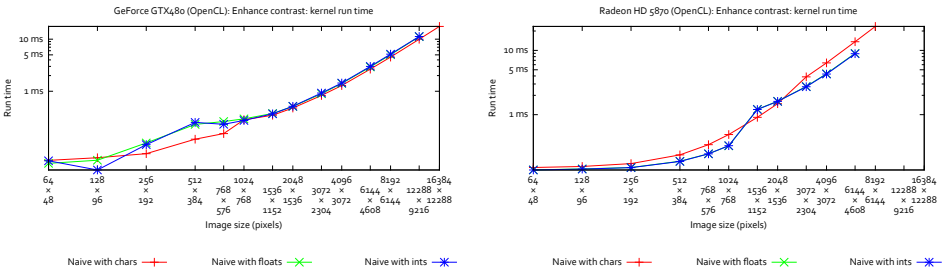
Radeon HD 5870



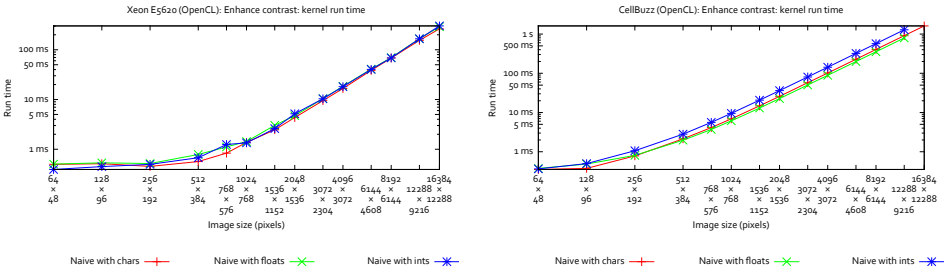
Xeon E5620

Cell BE

Figure A.87 – *Different data types compared, for the enhance contrast filter, histogram part of the kernel run time. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

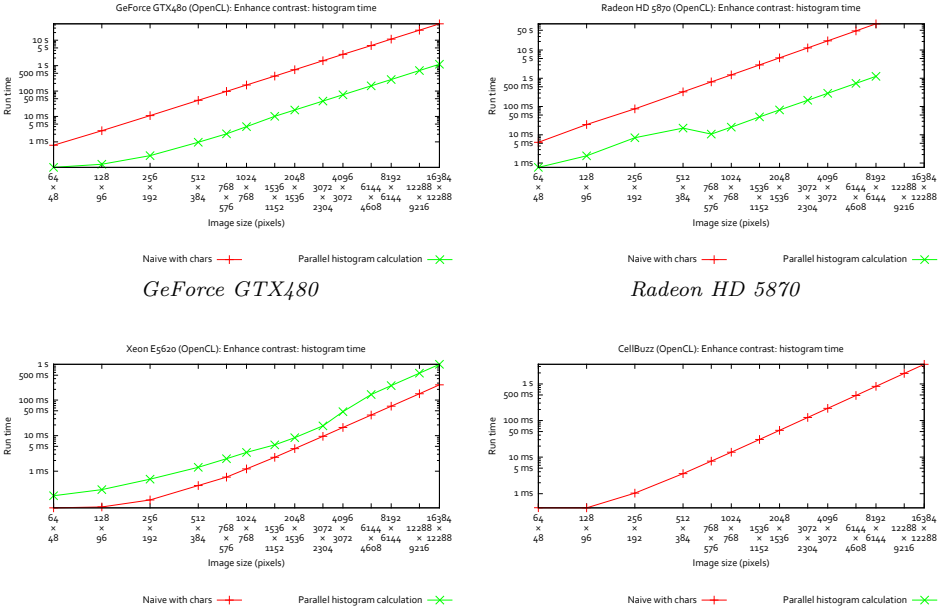


Xeon E5620 *Cell BE*

Figure A.88 – *Different data types compared, for the enhance contrast filter, normalisation part of the kernel run time. Note that the scales of these plots differ.*

A.2.4 Parallel histogram

In this section we show the detailed results of running the parallel histogram version of the OpenCL code.



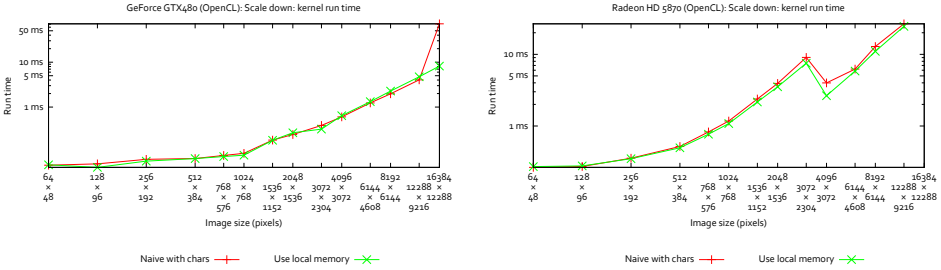
GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

Figure A.89 – *Comparing parallel histogram calculation with sequential histogram calculation. Histogram calculation times only. Note that the scales of these plots differ.*

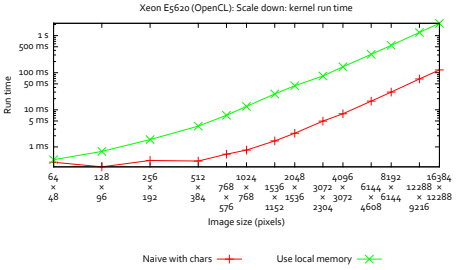
A.2.5 Using local memory

In this section we show the detailed results of running the local memory version of the OpenCL code.

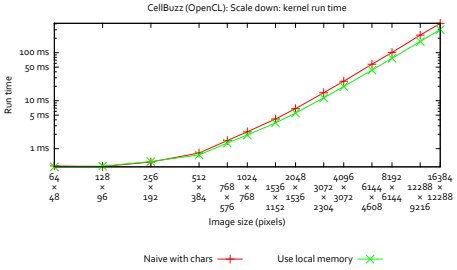


GeForce GTX480

Radeon HD 5870

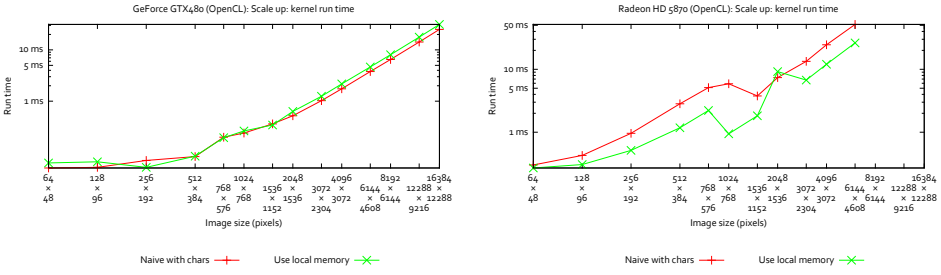


Xeon E5620



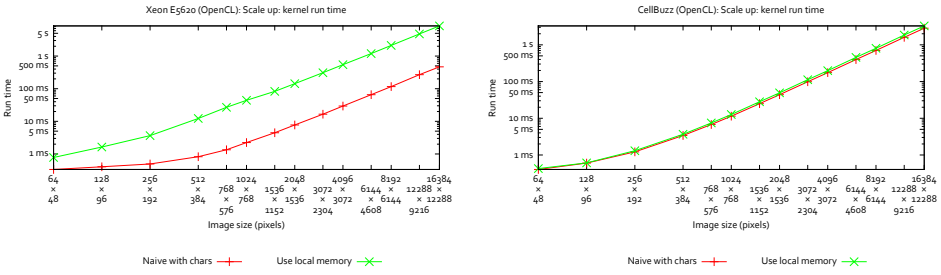
Cell BE

Figure A.90 – *Comparing the naive implementation to an implementation using local memory for the scale down filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480

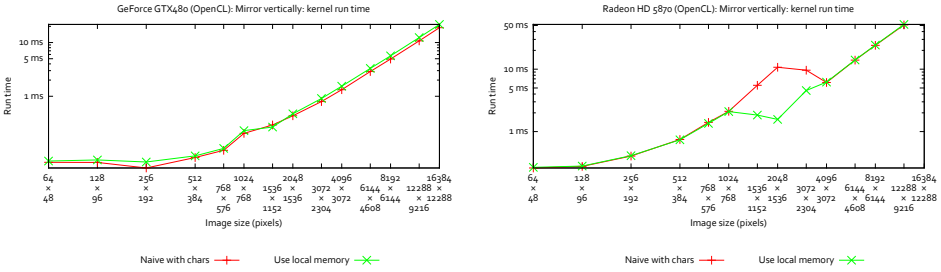
Radeon HD 5870



Xeon E5620

Cell BE

Figure A.91 – *Comparing the naive implementation to an implementation using local memory for the scale up filter, kernel run time only. Note that the scales of these plots differ.*



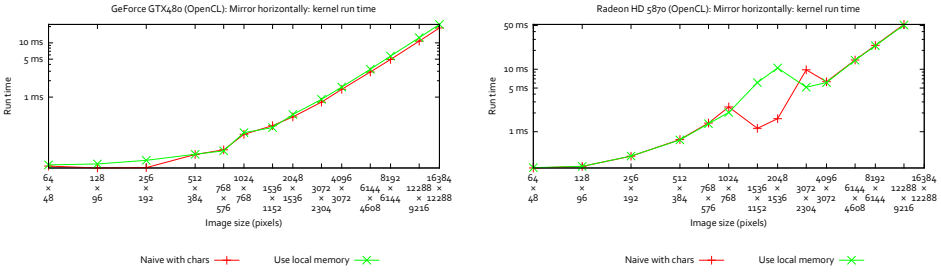
GeForce GTX480

Radeon HD 5870

Xeon E5620

Cell BE

Figure A.92 – *Comparing the naive implementation to an implementation using local memory for the mirror vertically filter, kernel run time only. Note that the scales of these plots differ.*



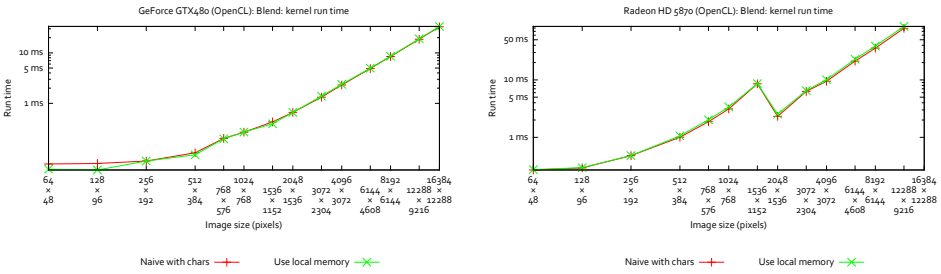
GeForce GTX480

Radeon HD 5870

Xeon E5620

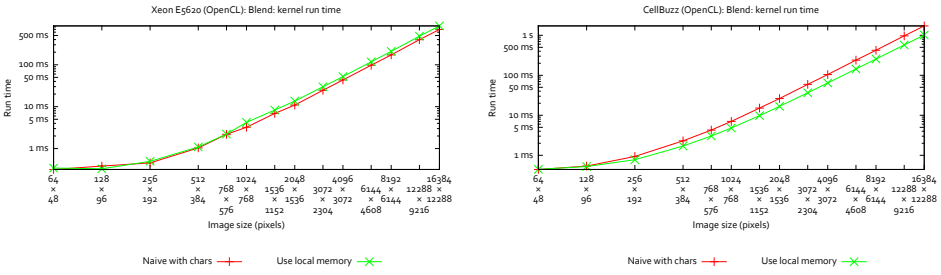
Cell BE

Figure A.93 – *Comparing the naive implementation to an implementation using local memory for the mirror horizontally filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480

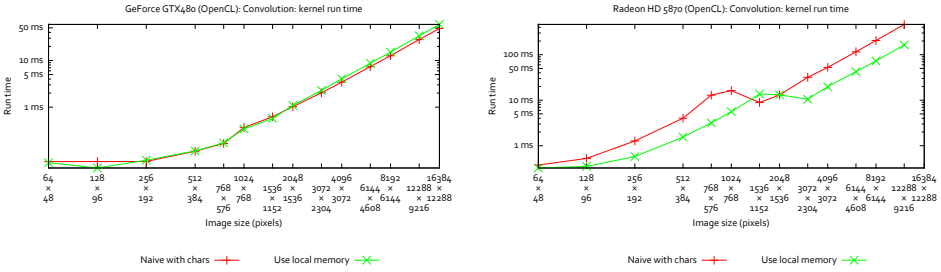
Radeon HD 5870



Xeon E5620

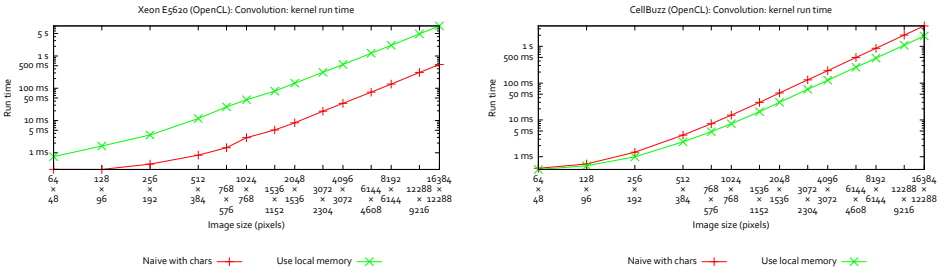
Cell BE

Figure A.94 – *Comparing the naive implementation to an implementation using local memory for the blend filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480

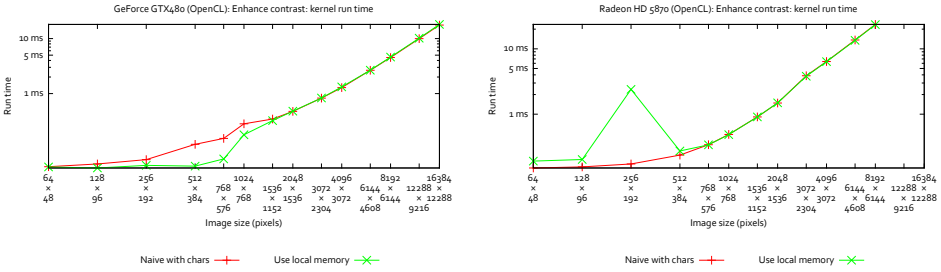
Radeon HD 5870



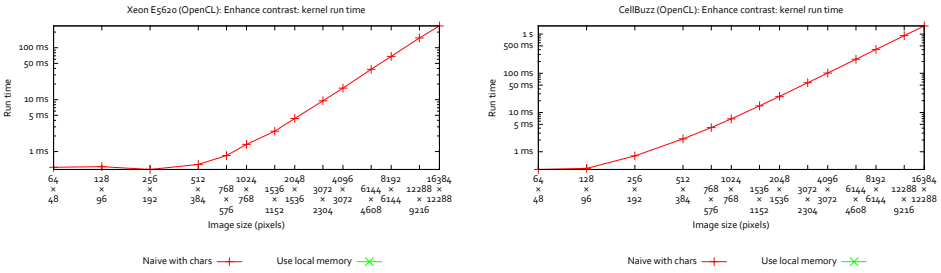
Xeon E5620

Cell BE

Figure A.95 – *Comparing the naive implementation to an implementation using local memory for the convolution filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

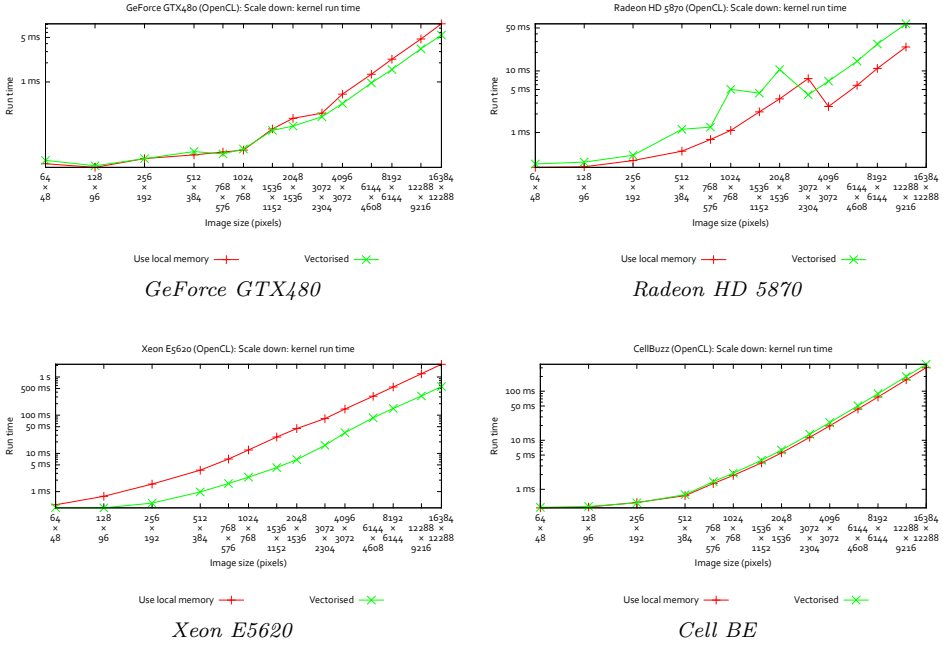


Xeon E5620 *Cell BE*

Figure A.96 – *Comparing the naive implementation to an implementation using local memory for the enhance filter, normalisation run time only. Note that the scales of these plots differ.*

A.2.6 Vectorisation

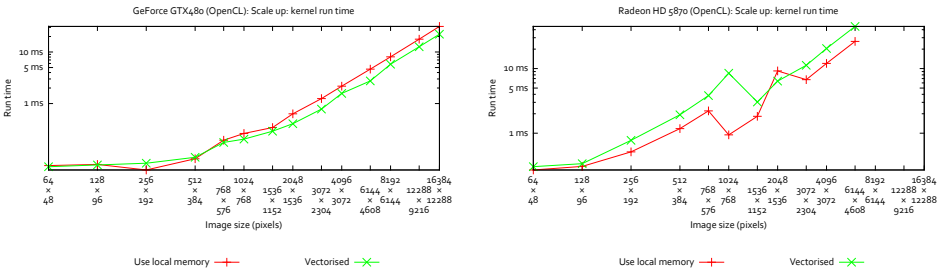
In this section we show the detailed results of running the vectorised version of the OpenCL code.



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

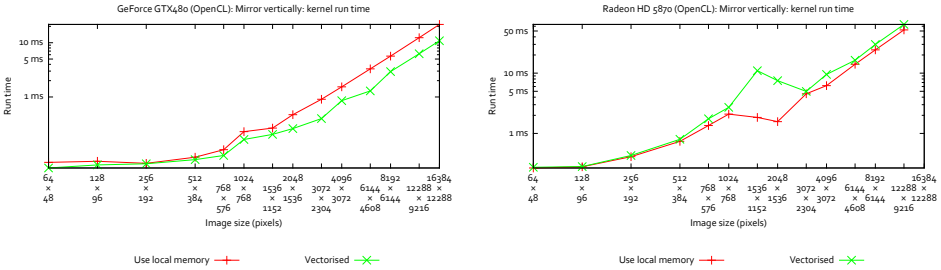
Figure A.97 – *Comparing a vectorised implementation to an implementation using local memory for the scale down filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

Xeon E5620 *Cell BE*

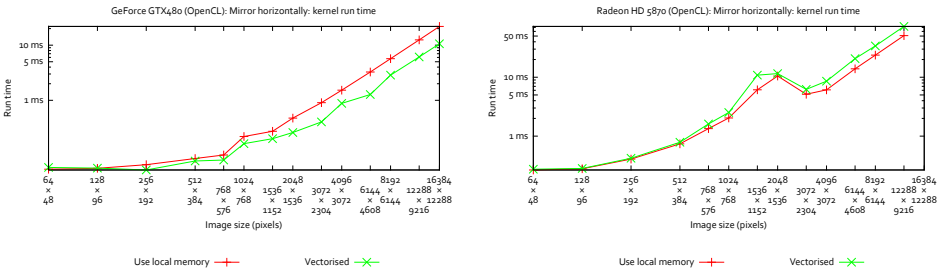
Figure A.98 – *Comparing a vectorised implementation to an implementation using local memory for the scale up filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

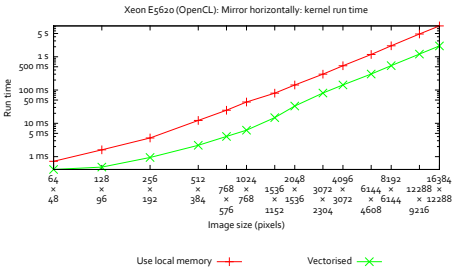
Xeon E5620 *Cell BE*

Figure A.99 – *Comparing a vectorised implementation to an implementation using local memory for the mirror vertically filter, kernel run time only. Note that the scales of these plots differ.*

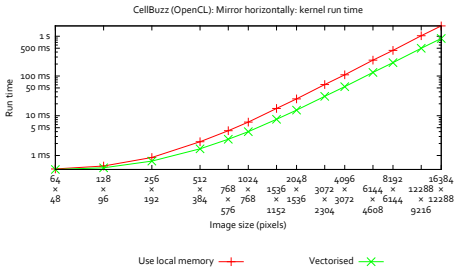


GeForce GTX480

Radeon HD 5870

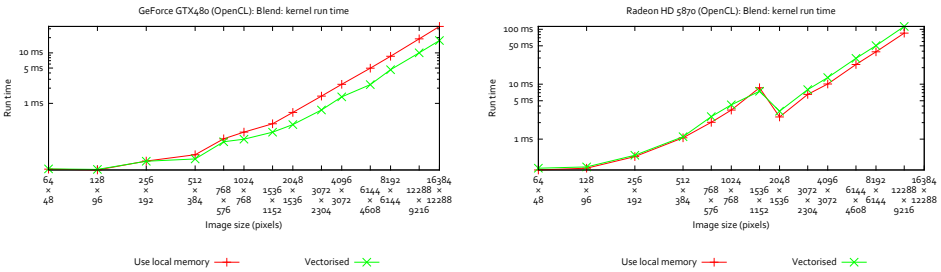


Xeon E5620



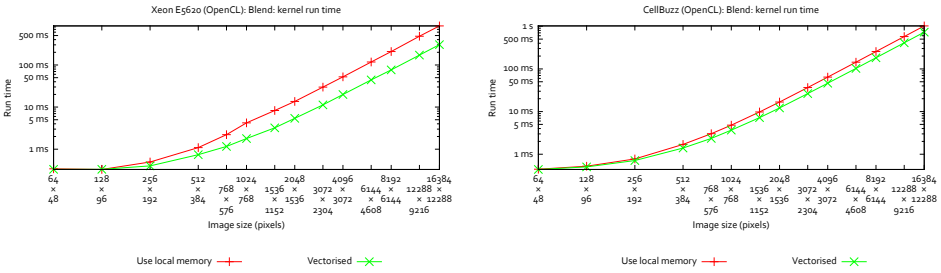
Cell BE

Figure A.100 – *Comparing a vectorised implementation to an implementation using local memory for the mirror horizontally filter, kernel run time only. Note that the scales of these plots differ.*



GeForce GTX480

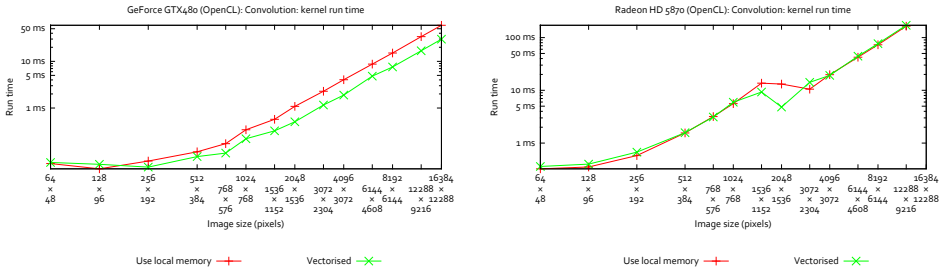
Radeon HD 5870



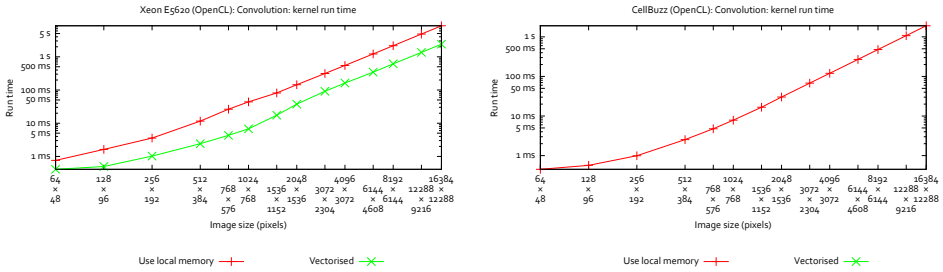
Xeon E5620

Cell BE

Figure A.101 – *Comparing a vectorised implementation to an implementation using local memory for the blend filter, kernel run time only. Note that the scales of these plots differ.*

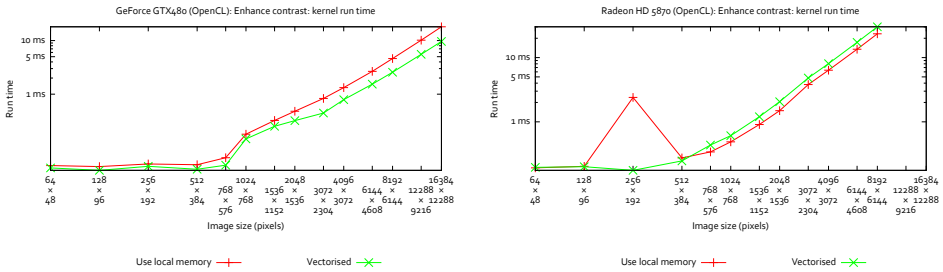


GeForce GTX480 *Radeon HD 5870*



Xeon E5620 *Cell BE*

Figure A.102 – *Comparing a vectorised implementation to an implementation using local memory for the convolution filter, kernel run time only. Note that the scales of these plots differ.*

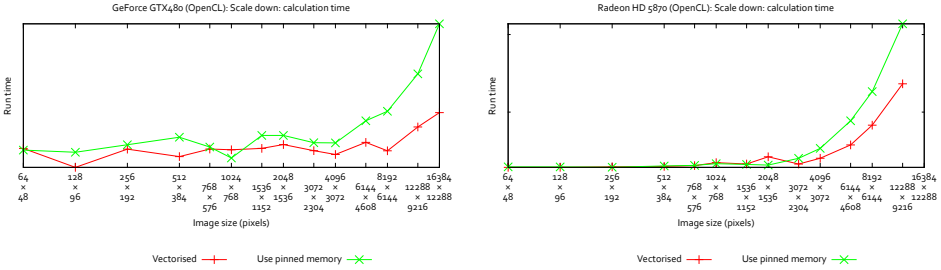


GeForce GTX480 *Radeon HD 5870*

Figure A.103 – *Comparing a vectorised implementation to an implementation using local memory for the enhance filter, normalisation run time only. Note that the scales of these plots differ.*

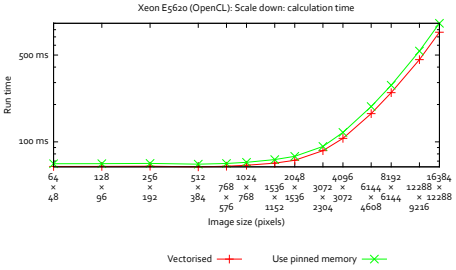
A.2.7 Using pinned memory

In this section we show the detailed results of running the pinned memory version of the OpenCL code.

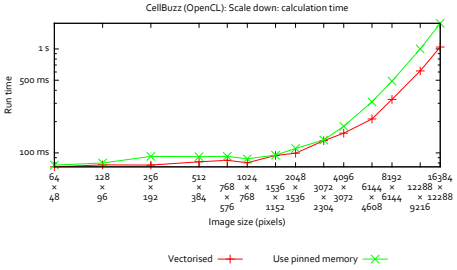


GeForce GTX480

Radeon HD 5870

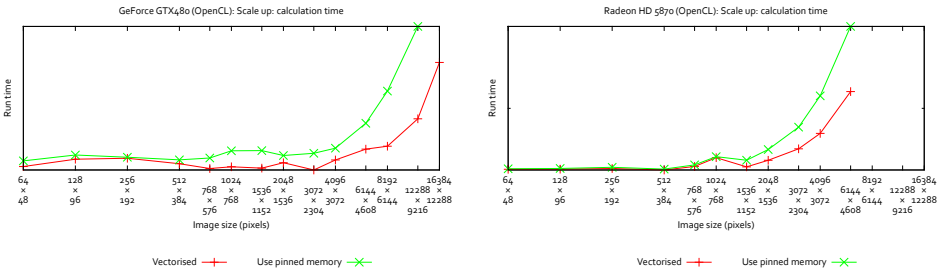


Xeon E5620



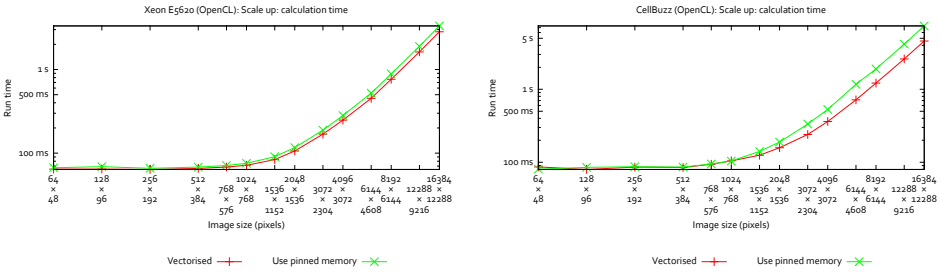
Cell BE

Figure A.104 – *Comparing a vectorised implementation to an implementation using pinned memory for the scale down filter. Note that the scales of these plots differ.*



GeForce GTX480

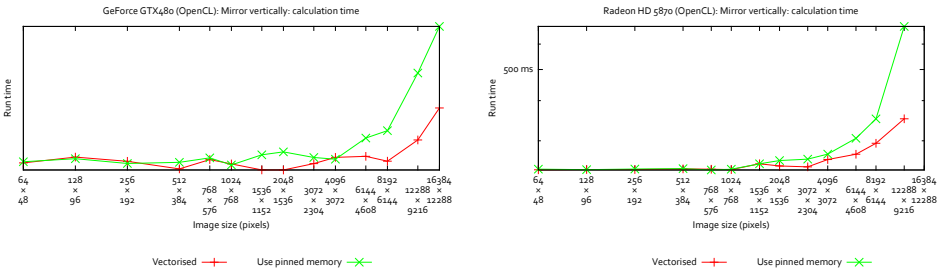
Radeon HD 5870



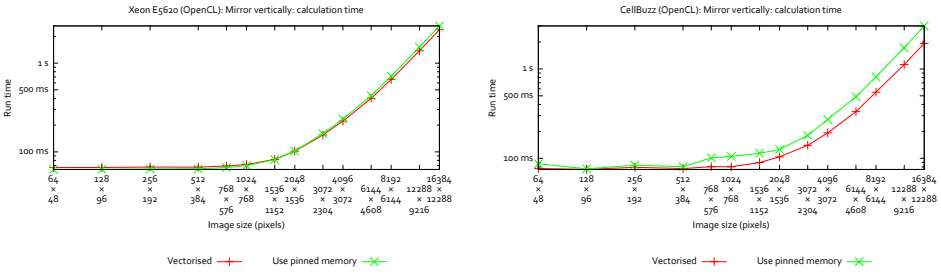
Xeon E5620

Cell BE

Figure A.105 – *Comparing a vectorised implementation to an implementation using pinned memory for the scale up filter. Note that the scales of these plots differ.*

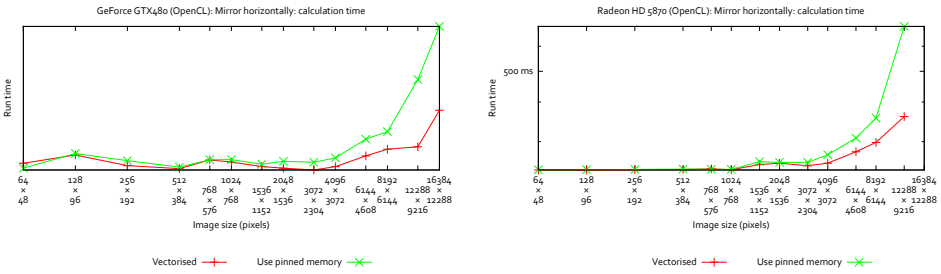


GeForce GTX480 *Radeon HD 5870*



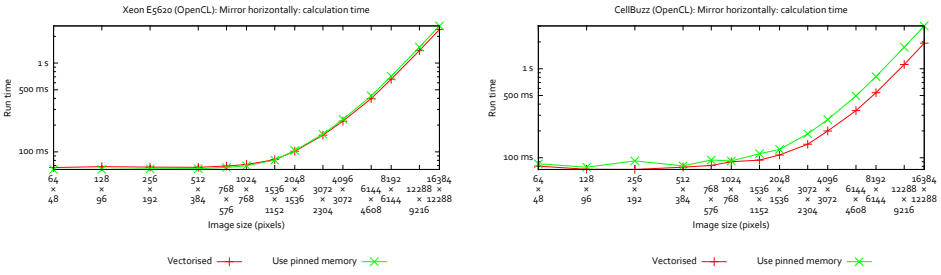
Xeon E5620 *Cell BE*

Figure A.106 – *Comparing a vectorised implementation to an implementation using pinned memory for the mirror vertically filter. Note that the scales of these plots differ.*



GeForce GTX480

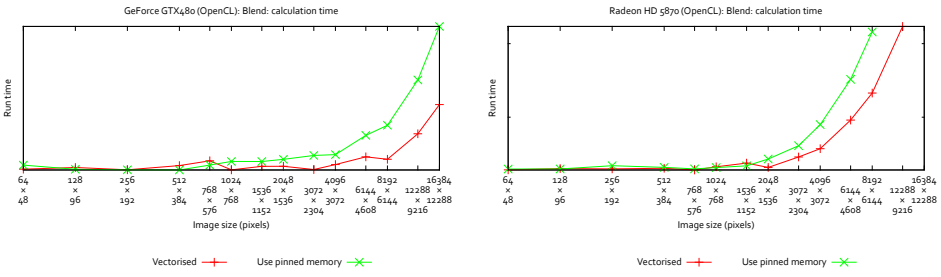
Radeon HD 5870



Xeon E5620

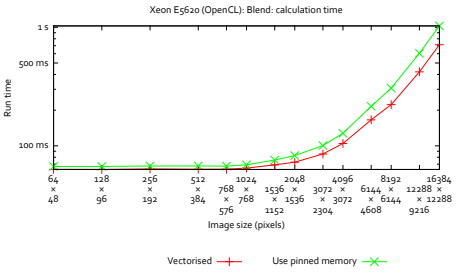
Cell BE

Figure A.107 – *Comparing a vectorised implementation to an implementation using pinned memory for the mirror horizontally filter. Note that the scales of these plots differ.*

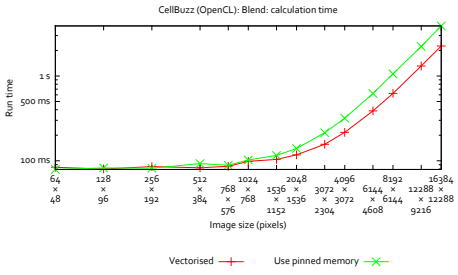


GeForce GTX480

Radeon HD 5870

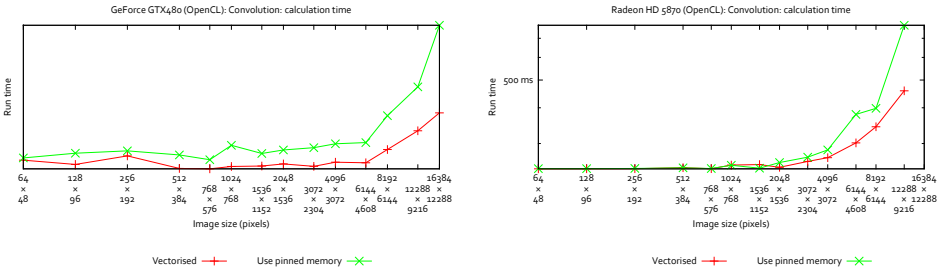


Xeon E5620

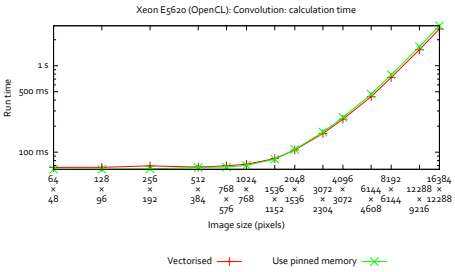


Cell BE

Figure A.108 – *Comparing a vectorised implementation to an implementation using pinned memory for the blend filter. Note that the scales of these plots differ.*

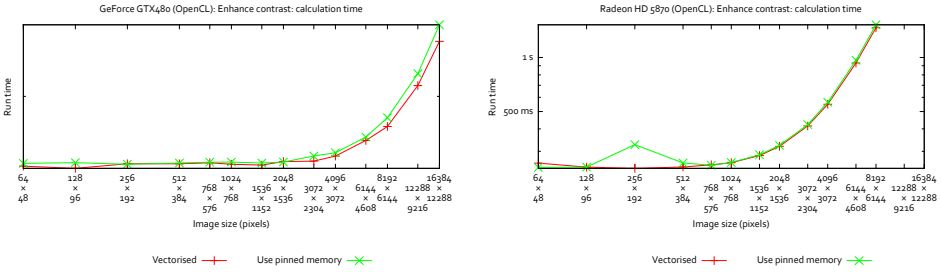


GeForce GTX480 *Radeon HD 5870*



Xeon E5620

Figure A.109 – *Comparing a vectorised implementation to an implementation using pinned memory for the convolution filter. Note that the scales of these plots differ.*



GeForce GTX480 *Radeon HD 5870*

Figure A.110 – *Comparing a vectorised implementation to an implementation using pinned memory for the enhance filter. Note that the scales of these plots differ.*