High Performance Histograms on SIMT and SIMD Architectures

M.E.R. Berger

Delft University of Technology

High Performance Histograms on SIMT and SIMD Architectures

Master's Thesis in Computer Science

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

M.E.R. Berger

6th December 2015

Author
M.E.R. Berger

Title
High Performance Histograms on SIMT and SIMD Architectures

MSc presentation
16th December 2015

Graduation Committee

Prof. dr. ir. Henk Sips	Delft University of Technology
Dr. ir. Alexandru Iosup	Delft University of Technology
Dr. ir. Ana Varbanescu	University of Amsterdam.
Dr. ir. Frans Kanters, MBI	Eindhoven University of Technology
Dr. Jie Shen	Delft University of Technology

Abstract

Using the histogram procedure, this work studies performance determining factors in computing in parallel on SIMD and SIMT devices. Modern graphics processing units (GPUs) support SIMT, multiple threads running the same instruction, whereas central processing units (CPUs) use SIMD, in which one instruction operates on multiple operands. As part of this work, a cross-technology framework is developed that allows testing a single-source histogram implementation on multiple devices, providing insight into the performance of various API – hardware configurations. It is shown that in the presence of high contention, the implementation of atomic operations becomes of great influence on performance. This work provides guidelines for the choice between devices based on image features and hardware specifications.

Contents

Chapter 1

Introduction

In image processing and computer vision, the need of determining the occurrence frequency of color values is very common. The *histogram operation* sums the occurrence of values in the input data to produce a (graphical) representation of their frequency distribution. In real-time analysis of video feeds, for instance for on-the-fly color corrections, huge amounts of data may need to be processed quickly. This calls for processing these data in parallel.

In the work, we focus on two common ways of parallelization of execution code. The oldest, *single instruction, multiple data* (SIMD), was introduced on the CPU and basically entails wide instructions that perform the same operation on multiple operants simultaneously. *Graphics processing units* (GPUs) are affordable instances of massively parallel processors. The rise of the programmable GPU has helped bring affordable parallel compute capabilities to the masses. Originally starting off with SIMD, GPUs have introduced a new approach to parallelization called *single instruction, multiple threads* (SIMT), in which multiple processing elements execute the same instruction on different data in parallel.

This thesis uses the relatively simple histogram operation to gain a better understanding of the performance determining factors when computing in parallel on SIMD and SIMT devices. When confronted with an instance of the histogram problem, we want to know what combination of hardware architecture and algorithm implementation provides the best performance. As GPUs and CPUs differ fundamentally, in a given scenario, the question is when is it beneficial to chose a GPU over a traditional CPU based approach.

1.1 Problem Statement

We formulate the following research question: What are the performance determining factors that impact the design and deployment of parallel histogram calculation on multi- and many-core hardware?

In answering the research question, this thesis offers the following contributions:

1. A *parallel histogram procedure* is implemented in which

- the *image* is *distributed*, and each thread typically services *multiple* pixels;
- the histogram *bins* are *replicated* per *block* of threads, followed by a *reduction* step that merges these instances into a single histogram;

2. A *cross-technology framework* is developed that enables writing a single-source implementation that is compilable and runnable using,
   - on the CPU, either OpenCL or C++ (with C++11 threads);
   - on the GPU, either OpenCL or CUDA;
3. A study of *contention* in *images* from the point of view of the parallel histogram procedure;
4. Insight into the *platform impact* on histogram performance in presence of contention;
5. A study into the efficiency of *atomics* as implemented on the different platforms.

1.2 Thesis Organization

This work is structured as follows: In chapter 2 we provide some background and present related work. The overview of the parallel architectures we discus is given in Chapter 3. An overview of the Histogram procedure is given in Chapter 4. Our cross-technology framework for writing a single-source histogram implementation targeted at multiple platforms is described in Chapter 5. Chapter 6 discusses atomic operations and write collisions. Our experiments and their results are discussed in Chapter 7. Finally the conclusions and an outlook on future work are given in Chapter 8.

Chapter 2

Related Work

In this section we will discuss Histograms, their algorithms and origins, Multi-Device frameworks and Atomics and their performance analysis. We will also see how this work fits in those contexts and how it is different.

2.1 Histograms

The concept of a histogram was formally introduced by Karl Pearson in 1895 [28]. The basic algorithm is trivial and even the parallelization options are limited. Due to its prevalence and usefulness not only in statistics but also in video editing, image manipulation, machine vision and learning, the histogram operation is implemented countless times on computer systems. There is previous work on efficiently implementing histograms on GPUs [17] [33]. Histogram implementations have also been described in hardware (FPGA) [19] [31] and for SIMD architectures [38] [3] [32] Our work also uses the histogram operation to empirically show performance characteristics, yet it is different from these works as we use this operation across a range of different processors instead of focusing on a single class.

2.2 Multi Device Frameworks

Using a framework or additional libraries on top of the vendor supplied API is very common. Heterogeneous computing presents the programmer with more work to unlock the full potential of the hardware. There is a lot of work on using libraries and frameworks to overcome this issue. Particularity noteworthy in the context of this work are: *BlockLib* [2], a macro based library for the IBM Cell/BE processor [5], and the so-called skeleton library based approach as presented in *SkePU: a multi-back-end skeleton programming library for multi-GPU systems* [15] that supports CUDA, OpenCL, OpenMP from c++.

The OpenCL standard targets multiple devices [21] can be classified as a framework and does indeed support all hardware devices used in this work. However,

as we will show, the performance offered by vendor implementations is not always the same as can be obtained through other means of programming the hardware.

SkeCL [39] is an example of extending OpenCL to support multiple devices connected over a network. In this work, we use the device model brought forth in the OpenCL specification [21]. Our framework is novel as it is single source and does not use external source to source translation tools to meet the targets platform requirements. This core idea is inspired by work done by the Blender open source project to support multiple back-ends for its ray-tracing engine [42].

2.3 Performance analysis of atomics

Performance in shared memory processors has been researched since the 1960's [26]. With modern GPUs effectively being programmable multiprocessor computers, a lot of older multiprocessor work applies to them. In a lot of cases they suffer from the same synchronization challenges as shared memory multiprocessors[1]. There are a lot of ways to program these devices which have been compared extensively, for example in work by Jianbin Fang, Ana Lucia Varbanescu and Henk Sips on comparing CUDA and OpenCL. OpenCL and openMP have been compared in work by Jie Shen, Jianbin Fang, Henk Sips and Ana Lucia Varbanescu [36]. OpenCL performance on CPUs has also been done [37]. All of these compare the performance of two different ways of programming, mostly on a narrow set of hardware. Our work differs because it compares CUDA, OpenCL and C++ on different devices by multiple vendors, spanning multiple hardware generations. Apart from that, and maybe most importantly, we are the first work to focus entirely on atomics within this context.

Chapter 3

Parallel Computer Architectures

This thesis deals with performance characteristics of multi-core and many-core processors. As such, this chapter provides an overview of parallel architectures. From the historical context of parallel computer architectures, we outline important aspects of these architectures, focusing particularly on their execution models as well as their memory models. We further discuss programming languages for these architectures.

3.1 From a Serial to a Parallel Compute Model

The increase of the number of transistors on integrated circuits has followed the trend predicted by Moore's Law for more then 40 years, providing increasing speed-ups for processors. Packing the transistors closer together or effectively shrinking the design, measured in nanometers, allows the signals on the chip to travel farther within one clock tick which in turn allows for higher clock frequencies or more complex designs. Each successive generation of micro-processors has been increasingly complex and featured increases in clock speeds. Running at higher speeds implies that the transistors switch at a higher rate, emitting heat at each switch. At switching speeds somewhere between 4 GHz and 5 GHz, this heat production becomes such a problem that since the early 2000's, CPU operation frequencies have seen little or no increase. However, Moore's Law is still in effect, and during that same period the production process went from 0.13 µm (130 nm) to 14 nm. As we can no longer increase the clock frequency, we compensate with more cores which are also more complex. Even then, there is enough silicon real estate available to integrate heterogeneous cores, like specialist graphics processing cores, to the same die.

3.1.1 SIMD and SIMT

Two common ways of parallelizing code execution exist, denoted as *single instruction, multiple data* (SIMD) [6] and *single instruction, multiple threads* (SIMT) [9].

While similar in name, these approaches are only remotely related and not mutually exclusive; in theory both can be used together. SIMD relies on dedicated hardware that supports operations on multiple scalar memory elements together, so-called SIMD vectors (e.g., 4-float vectors in SSE, 8-float vectors in AVX, 16-float (512-bit) vectors in AVX3 [6]).

SIMD was introduced on CPUs. Modern GPUs are based on the SIMT model, abandoning native SIMD support in hardware and using multiple processing elements instead that use scalar operations in blocks of processing elements. Nevertheless, some GPUs still support 4-float SIMD vectors. On CPUs, SIMD continues to play a central role, e.g., in multimedia operations such as video encoding.

Composing program code that uses SIMD hardware, by programming in assembly language or using assembly-derived intrinsics in high-level languages such as C, is time consuming and error prone [25] [41]. While it provides the programmer with complete control of the low-level details, this goes at the expense of productivity. Alternatively, fixed-size vector data types are supported by specialized compilers such as Intel's SIMD compiler for C/C++ [7], but this sacrifices cross-compiler portability.

The process of vectorization can be automated through employing auto-vectorizing compilers (e.g., GCC, Intel C compiler (ICC), and the IBM XLC compiler) that automatically translate sequences of scalar operations, typically represented in the form of loops, into vector instructions [25]. However, it was shown that the impact of implicit vectorization introduced by these compilers is limited, despite auto-vectorizing 45-71% of the loops in a synthetic benchmark, in real-world applications auto-vectorization works in far fewer cases [25].

3.2 GPU Architectures

Specialized circuits designed towards the manipulation and creation of bits within a computer frame-buffer, i.e., for display purposes, have existed since the early 1980's [14]. By the end of that era, through developments adhering to Moore's Law, more advanced pixel processing became feasible [18, 29].

GPUs continued to advance in terms of speed and programmability, enabling them to be used for general-purpose computation other than graphics [4]. In November 2006, Nvidia released the GeForce 8800 GTX (equipped with the Nvidia G80 chip), a GPU that unified all the shader stages into a common floating-point core[8]. It also marked the introduction of the SIMT execution model. This brought with it a massive increase in programmability and, for the first time, proper general-purpose computational capabilities were available [24]. In January 2007, Nvidia released the *Compute Unified Device Architecture* (CUDA) language and SDK to program their GPUs. Around the same time, the notion *General-Purpose computing on Graphics Processing Units* (GPGPU) comes into play and starts to develop into a hot research area [27], as other vendors (such as AMD, ARM and Intel) are also attempting to bring hardware with many processing elements to the market

[30].

3.2.1 Nvidia

Looking back on the launch of the G80 chip in 2006, it is clear that its release ushered in a new era for GPU hardware. Combined with the introduction of the CUDA programming model and language, it really brought GPGPU forward. After

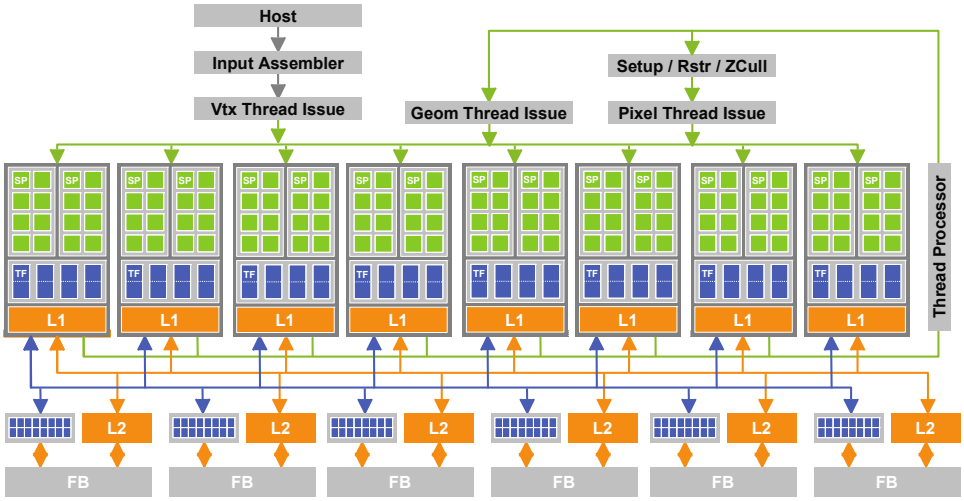

Figure 3.1: *Nvidia Tesla Architecture*
featuring 8 blocks of 16 processors from: [8]

the release of the G80 chip, the first incarnation of the *Tesla architecture* [8] (see Figure 3.1), Nvidia released Fermi [9], Kepler [10], and Maxwell [11] architectures. Unlike x86 processors, each hardware architecture has different *compute units* (CUs) (called *streaming multiprocessors* (SMs) in Nvidia terminology) and a different *instruction set architecture* (ISA). GPUs released with a certain chip then have features turned on and off and different sizes of RAM and memory bus-width, making different Nvidia GPUs much more different from each other than CPUs ever are. The G80 chip, with its Tesla architecture being the first designed for SIMT, also set off the move from 4-float wide SIMD to 16-float SIMT for GPUs, later increased to 32-float wide (1024-bit).

3.2.2 AMD

While Nvidia released its G80 chip with unified shader model and scalar processors to the market, ATI, acquired by AMD in 2006, was still investing strongly in its SIMD-based *very long instruction word* architecture VLWW5. This hardware was

designed and is very well suited for performing DirectX-9-era graphics operations but less well suited for GPGPU. It took AMD until January 2012 to release hardware that was designed from the start with GPGPU in mind. AMD calls this design *Graphics Core Next* (GCN), see Figure 3.2

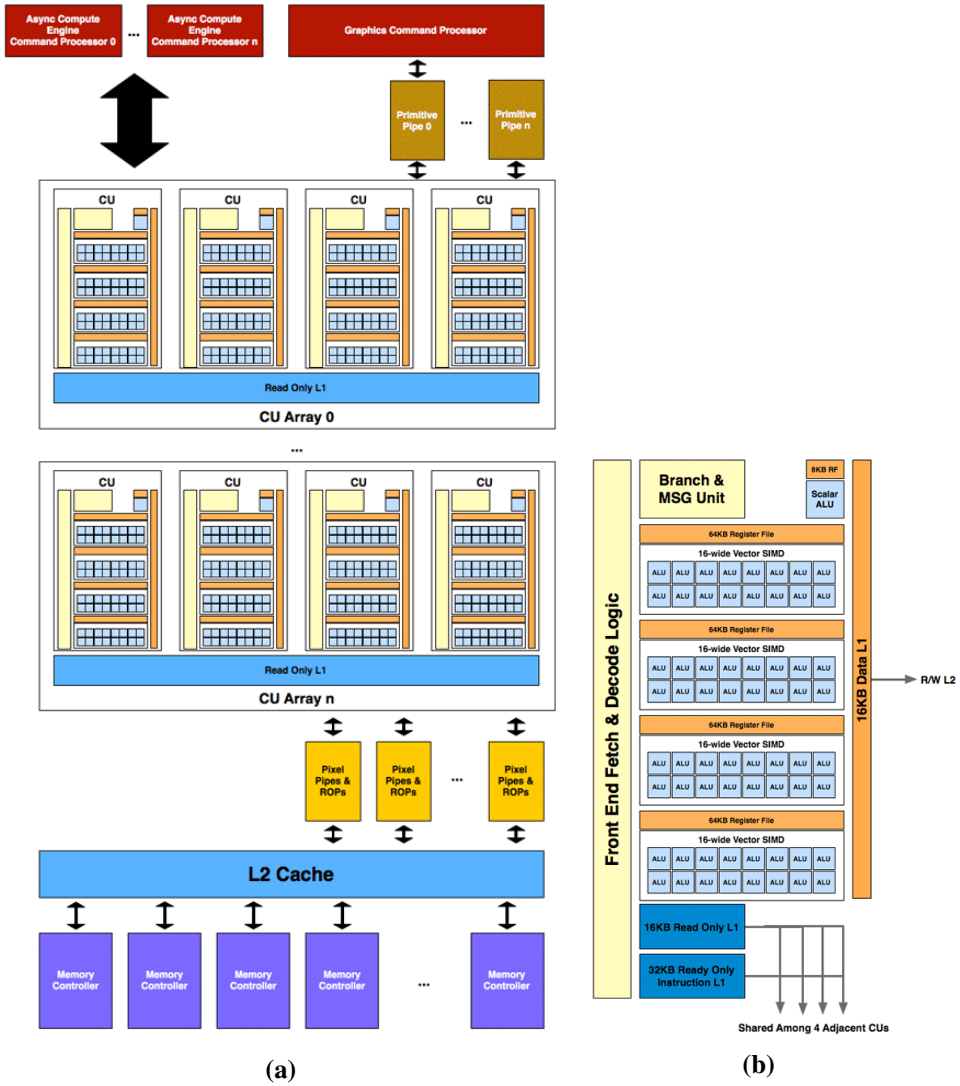

Figure 3.2: *AMD Graphics Core Next*. In (a), the GCN GPU architecture is displayed. In (b) is zoomed in on a GCN compute unit.

3.3 Parallel Programming Model

Programming for multi-core systems is in a lot of ways different and typically harder than writing serial code aimed at traditional PC-like systems. Data de-

pendencies and synchronization cost play a mayor role in the design of efficiently running algorithms for multi-processor machines. In programming GPUs, there is an explicit memory hierarchy to be taken into account, whereas it is implicit (i.e., hidden) in CPU programming. In SIMT, threads run in groups termed *thread blocks* or *work-groups*. Each thread runs on a SIMD or scalar unit, depending on the GPU architecture. Many different GPU architectures exist, typically not entirely compatable with each other, which makes the implementation very sensitive to the exact architecture used.

3.3.1 CUDA

Nvidia's CUDA progamming platform is leading in GPGPU programming. Released in June 2007, it is mature in the sense of robustness and relatively easy to use. CUDA represented great steps in making programming GPUs much easier and more accessible.

CUDA comes with two APIs: The higher-level *runtime API* and the lower-level *driver API*. In using the runtime API, CUDA code is mixed in C++ code. The **nvcc** compiler splits the CUDA code out and compiles it into PTX code, a low-level *parallel thread execution* virtual machine. The virtual *instruction set architecture* (ISA) enables using GPU-like devices as parallel computing machines [12]. The tool **ptxas** then converts the code for the actual ISA of the targeted device.

CUDA is now moving towards what is termed *unified memory*, which means implicit memory management in which the explicit division in global, per-block, and per thread memory is hidden from the programmer.

3.3.2 OpenCL

Open Compute Language (OpenCL) 1.0 was published in December 2008. In many regards, OpenCL is highly similar to CUDA, when targeting GPUs. However, OpenCL is also supported by vendors of CPUs and field-programmable gate arrays (FPGAs) for targeting those computing platforms. This is achieved by abstracting away some of the specifics of each platform.

The OpenCL platform model consists of a host to which one or multiple OpenCL devices are connected. Each device consists of one or more *compute units* (CUs) that contain *processing elements* (PEs), see Figure 3.3. OpenCL defines a memory model that consists of four different types of memory: *Global* memory is accessible from *work-items* in all *work-groups* and is writable. *Constant* memory is constant for the duration of the execution of a *kernel*. *Local* memory is local to the CU and *private* memory is local to an individual PE (also see Table 5.2). When programming an OpenCL device, there is also host (i.e., system) memory to consider.

In a lot of scenarios, one wants to do as little global memory operations as possible, as well as host-to-device operations as these are slow. This fact combined with the scarcity of the other types of memory on a lot of implementations leads to

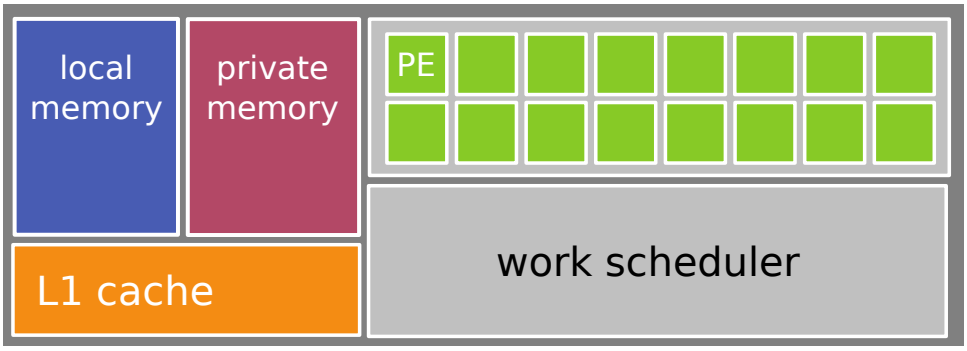

Figure 3.3: *OpenCL Compute Unit*. Prototypical OpenCL compute unit, showing a group of processing elements sharing memory and an instruction pointer.

an awkward situation where the OpenCL code *is* portable but decent performance with this portable code is *not* in a lot of situations [13, 23, 16].

3.4 General Remarks and Outlook

GPU architectures are changing significantly with each new hardware generation. New software and hardware features make programming them easier but at the same time abstract away more. The current logical model as expressed in the OpenCL and CUDA device models seems to continue to fit this class of hardware well, but even at the logical level, there is a trend towards more abstraction layers. GPUs are both powerful and highly complex. Reasoning about the performance of these systems is very different compared to that of single-core, single-threaded software, making comparative performance evaluation challenging.

While GPGPU is a relatively recent field that seems to have come into existence by accident, it can turn the average PC and mobile device into very powerful parallel computation machines. Getting good performance from such machines, however, is not easy. Massively parallel processors are not suited for all types problems, often requiring reformulating a task at hand to be able to efficiently make use of GPGPU. Nevertheless, as they provide great amounts of processing per Watt, they are there to stay and will most likely become increasingly prevalent.

CPU development is likely to continue to aim at bringing greater performance through including more smart logic such as *branch prediction caches*. This is the exact opposite strategy followed for GPUs, packing increasing amounts of relatively "dumb" logic on devices. However, GPUs will most likely become a bit smarter as the *thread management* hardware continues to evolve. In future architectures, we will see support for running multiple independent tasks on a single device. *Unified memory* will bring hardware assisted host–device data transfers. *Dynamic parallelism* will offer support for kernels launching other kernels, and there will be task priority support while running multiple kernels. Last but not least, faster *atomic operations* (see Chapter 6) will be unveiled.

Chapter 4

Histograms

Histograms are extremely common operations in the field of statistics, with applications in biology, applied sciences, and video processing. The concept of a histogram was formally introduced by Karl Pearson in 1895 [28]. Pearson, mathematician and biometrician, is seen as one of the fathers of modern statistics. Histograms are best known in their graphical representation, commonly drawn as a bar diagram. They are well-suited for representing complex statistical properties of a dataset such as the *probability density* of the *occurrence* of values. Figure 4.1 displays histograms over the color values of pixels in three channels.

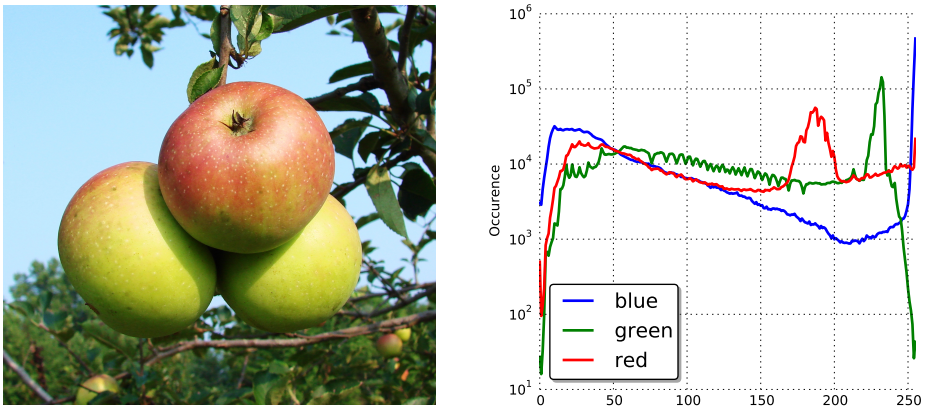

Figure 4.1: *Color Histograms.* Distribution of color values of pixels in a photo depicting apples, determined over three color channels.

Image processing and *computer vision* rely on histograms in a variety of operations. In analyzing high-resolution video feeds in real-time, excessive amounts of data are to be processed: It calls for the capability of running histogram operations on billions of data points, multiple times per second. These circumstances call for exploiting the strict absence of data interdependencies in the histogram operation, which allows to properly employ parallel computing resources.

In this chapter, we give a description of the histogram operation and present

two ways in which the parallelization of this algorithm can be achieved, explaining which strategy is recommended for what class of hardware and problem type.

4.1 Generic Histogram Procedure

Given a set of data points, the histogram operation defines a set of disjoint ranges called *bins*. Each data point is assigned to a bin based on a characterizing *value*, increasing the count of data points associated with this bin. The histogram of the entire data set is defined by these sums [28]. The basic procedure is laid out in Algorithm 1.

foreach *data point* **do**
    determine characterizing value
    calculate associated bin
    increment bin's counter
end

Algorithm 1: *Generic Histogram Procedure*

The proper number of bins depends on the phenomenon one aims to examine and on expectations concerning the input data. Even with this information available, choosing the optimal number of bins is a hard problem [40].

For the general case, bins are not required to have equally sized ranges. For the case of analyzing digital images, it is typical to use a single bin per 8-bit color. This leads to having 256 bins per color channel, a resolution that is also adequate when analyzing images of 10 of 12 bits per channel per pixel. In this thesis, separate histograms will be used for the channels red, green, and blue, with 256 bins per color channel.

4.2 Histogram Parallelization

For the histogram problem, there are roughly two main ways in which parallelization can be achieved: Each parallel process can evaluate a *full histogram* for a *part of the input* data, see Algorithm 2, or a *partial histogram* over the *whole input* data, see Algorithm 3. A partial histogram consists of a subset of the *bins*, i.e., registers associated to an interval in the input space, counting occurrences only for values falling within this subset.

For the case where the input data is split into multiple ranges, the maximal parallelism achieved is equal to the number of input elements. The maximal parallelism achieved for the partial histogram method is equal to the number of bins.
Algorithm 2 consists of two parts: First, each parallel process is given a part of the input data only, over which a full histogram is calculated. These histograms are then reduced to a single histogram in parallel, using maximally one parallel process per bin.

```
divide input data into segments
for segments do in parallel
    foreach data point do
        determine characterizing value
        calculate associated bin
        increment bin's counter
    end
end
divide bins into sets
for sets do in parallel
    foreach bin do
        calculate sum over corresponding bin in all histograms
    end
end
```

Algorithm 2: *Full Histogram – Partial Input Space*

```
divide bins into sets
for sets do in parallel
    foreach data point do
        determine characterizing value
        calculate associated bin
        if bin in set then
            increment bin's counter
        end
    end
end
concatenate the partial histograms
```

Algorithm 3: *Partial Histogram – Full Input Space*

Algorithm 3 consists of two parts as well. First, each parallel process reads *all* input data, but increments only the count of those bins assigned to it, after which the partial histograms are concatenated into a single full histogram.

In both methods, a reduction step is required in order to create the final histogram. In Algorithm 2, the runtime of the reduction step depends on the amount of parallelization (i.e., the number of histograms to sum over), and at a certain point, the additional slowdown introduced by the reduction step may surpass the speedup provided by additional parallelization. Algorithm 3's reduction step is negligible, but this approach requires *all* parallel processes to read *all* input data.

Chapter 5

Cross-Technology Framework

For facilitating the performance analysis of computing a histogram on different platforms, a *cross-technology framework* is designed, allowing for the use of a single code base. The main goal is the ability to run a single source implementation of an algorithm *unmodified* on the three types of platforms considered: *OpenCL*-based, *CUDA*-based and *C++*-based. This allows for faster development *and* prevents distorted results caused by errors present in certain platform implementations only. Furthermore, a platform implementation obtained through the framework has to achieve the same performance as directly coding it for the target platform.

The effort of composing the cross-technology framework is justifiable through providing an efficient, fast, and elegant way of identifying and characterizing the determining factors in platform performance. These range from *processing-element topology*, *memory topology*, availability and implementation of *atomic operations*, other hardware parameters and software settings like *execution topology*. The framework enables quick iteration over, and offers a handle for evaluation of, sample points in this search space of different hardware and software combinations.

5.1 Framework Requirements and Assumptions

In order to be able to determine the performance profile over multiple architectures, for a broad range of problem instances, we need a *unifying model* implemented in the form of a *cross-technology framework*. The framework allows us to use a single code base and single implementation of the algorithm. This provides for a better focus on the algorithm implementation as it needs be implemented only once and eliminates errors arising from multiple implementations.

In order to avoid inefficiencies, the framework needs to *minimal*, both in the sense of functionality included in the language abstraction, i.e., the level of the logic supported, as in the actual framework implementation. The dialect that we derive still needs to be *sufficiently expressive* and allow for the use of platform-specific methods in a general way. The use of platform-specific *fast-paths* is not

to be precluded for the sake of achieving an abstraction, and the actual algorithm implementation coded in the provided language abstraction should not be riddled with special platform-specific hacks.

Multiple logical cores (i.e., processing units) are assumed, thus the program to be written should be specifiable over multiple logical elements. Furthermore, an OpenCL-like *memory hierarchy* is adopted. The programmer should annotate memory in an OpenCL-like fashion.

5.2 Framework Definition

We are faced with a set of varying hardware (see Table 7.1) that supports partially overlapping subsets out of a set of three programming languages (see Table 5.1). A minimal language abstraction is created to be able to run a single algorithm implementation on all supported combinations of hardware and programming language. The framework exploits the fact that both OpenCL and CUDA are based on the C programming language, and that C++ can be seen as a superset of C. Thus, the three languages already have a large overlap, and through defining functionality required by one language in those where it is missing, the overlap is extended into a language dialect that fits our requirements. This yields a dialect that is both expressive and sufficiently close to the three target languages that using it is not a source of inefficiency for their compilers. We design the language dialect that the framework entails to be usable for *generic problems* (e.g., other than histogram) that can be written for execution by *two-dimensional* fields of threads with a *two-tier* abstraction of those threads.

Table 5.1: *The Availability of Languages per Hardware Platform.* OpenCL C runs on GPUs, as well as directly on the CPU through OpenCL runtimes such as *PoCL* and those provided by Intel and AMD. Performance differences arising from operating OpenCL C on different hardware types and differences inherent to the languages are examined in this work.

Hardware Platform	Languages Available
CPU	OpenCL C, C++
AMD GPU	OpenCL C
Nvidia GPU	OpenCL C, CUDA C

5.2.1 A Common Execution Hierarchy

GPUs define a different execution hierarchy than CPUs do. For supporting a cross-technology framework, the CPU model is mapped onto a model suitable for describing GPUs. CPU cores are taken as equivalent to *compute units* (CUs) with a single *processing element* (PE), see Mapping II in Figure 5.1. This choice is

motivated by their independence when compared to PEs. Different CUs operate independently, whereas PEs are highly dependent due to shared resources such as cache and the fact that they share their instruction pointer, i.e., within a single CU, all PEs *always* execute the same instruction. The CUDA process model is largely equivalent to its OpenCL counterpart and with the chosen mapping for CPUs, we have one consistent model that is applicable to all three platforms.

Table 5.2 provides the mapping between terminology of the different architectures used in defining the common execution hierarchy. In the remainder of this report, we will primarily use the terminology of OpenCL when referring to aspects of this model.

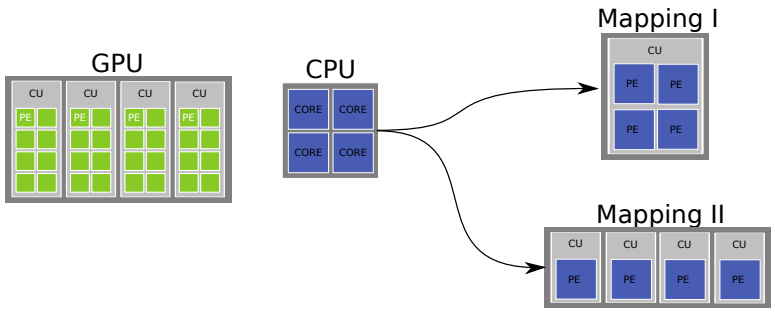

Figure 5.1: *Mapping CPU to GPU-like Architecture.* In mapping a multi-core CPU to a GPU-like architecture, it can be regarded as a single *compute unit* (CU) with multiple *process elements* (PEs) (see Mapping I), or multiple CUs with a single PE (Mapping II). We interpret a CPU using Mapping II, motivated by the relative independence of CPU cores.

Table 5.2: *Terminology Mapping between Technologies*

OpenCL	**CUDA**	**CPU**
Device	GPU	CPU
Compute unit	Multiprocessor (SM)	[a]
Processing element	Scalar core	[a]
Global memory	Global memory	Main memory
Local memory	Shared (per-block) memory	[a]
Private memory	Local memory	Registers / cache [b]
Program	Kernel	Function
Work-group	Block	[a]
Work-item	Thread	[a]

[a] No such concept present.
[b] No user control.

5.2.2 Unified Language Dialect

The language dialect needs to be interpretable by compilers implementing OpenCL C, CUDA C, and C++ with allowed preprocessing limited to that provided by their built-in C-preprocessors. In general, writing a C++ program that compiles well on different compilers and operating systems is already non-trivial. In practice, cross-platform C++ code is achieved by limiting the language to a subset of features supported by all C++ compilers. In cases where this subset is too restrictive, preprocessor and build-system handles are employed to obtain code compatible with the actual compilers aimed to be used. Finding such a supported subset and making things operate under different conditions is usually done in an iterative way, porting code cumulatively through trial-and-error.

The unified language dialect that we envision is created in a similar fashion. An important difference is the narrow overlap between *all* three languages: Unaugmented, the bare overlap does not even allow for the definition of a common empty program without any actual instructions.

Composing the Unified Dialect

As their names imply, OpenCL's and CUDA's programming languages are based on the C programming language. Pure C++ functionality such as *classes*, *templates*, and *namespaces* are therefore not available. C++ can be considered a superset of C, while not in the strict sense (i.e., not all C language constructs are allowed). The unified language dialect is composed through piecewise implementing the histogram operation, starting from the common language subset, that is, language elements that are *literally* present in all three languages, e.g., *basic types*, *inline functions*.

The majority of the necessary augmentation to this subset is obtained through *function-like preprocessor macros*. This is done from the perspective of OpenCL C, adding calls and elements to other languages through *wrapping* their language counterparts in macros (essentially renaming them) or adding new operations in line with the architecture mapping (see Figure 5.1). For instance, thread identification is available in OpenCL C through methods as **get_global_id**(), while CUDA C defines variables **blockIdx**, whereas C++ misses the notion of such *work-groups* altogether. It can also occur that mapping contructs to OpenCL C is not practical, in which case a new unifying element is introduced in all three languages.

Importantly, as stated, all differences to overcome are bridged without any kind of source-to-source compiler. We rely fully on the destination language's C-preprocessor and includes to make a program written in the unified dialect "appear valid" in the destination language. In order to achieve a minimal-size framework, the iterative method of compiling an increasingly complex program on all three language's compilers was chosen, iteratively expanding on a single-code implementation.

Continued compiling, testing, and adapting was used to keep the code working across these languages and various compilers for those languages. Language

elements that required notable implementation decisions to be taken are listed next.

SIMT Support

In the single instruction, multiple threads (SIMT [9]) model, multiple threads are bound to the same instruction pointer, requiring a means of identifying threads to diversify in instruction effect, e.g., to indicate the input data that the thread should execute the instruction on.

To this end, OpenCL C offers built-in functions (**get_global_id()**, **get_local_id()**), whereas CUDA C has built-in variables **threadIdx**, **blockIdx**, **blockDim**, **gridDim**, and **warpSize**. The latter can easily be wrapped, as well as adding the API to C++ by setting the **local_work_size** in all dimensions to 1 and **local_id** to 0.

SIMD Support

OpenCL and CUDA support single-instruction multi-element vector manipulation (single instruction, multiple data, SIMD [6]) by *swiveling*, i.e., the direct access and manipulation of components of fixed-size vector data-types, typically via letters like x, y, z, w. C++ does not have vectors allowing this type of access built-in. An example of swiveling is accessing elements of an OpenCL-C **float4** via **vec1.wzyx**, which has the same outcome as **float4(vec1[3], vec1[2], vec1[1], vec1[0])**, generating a new vector with elements reversed, in a single operation.

CPUs support SIMD operations via instruction sets such as SSE and AVX [6], but employing these in C++ in a *cross-compiler compatible* and *performance non-impacting* manner is not possible. Introducing swiveling in C++ essentially entails emulating the vector data-types through classes such as **float4** introducing the letter component addressing.

For swiveling operations, however, the float4 instances have to be kept aligned on the 128-bit boundary. Standard C++ compilers do not have handles to safeguard this, thereby requiring properly aligned copies of float4 instances to be generated prior to performing the SIMD operations. Swiveling could be introduced through employing Intel's SIMD compiler for C/C++ [7], albeit not cross-compiler compatible. It was therefore chosen not to include swiveling operations in the unified language dialect.

Textures

GPUs provide special containers for image data, *texture* objects that store pixels, i.e., multi-dimensional image data points providing color information. These texture objects are essentially arrays, two-dimensional in our use case, of 32-bit values. On GPUs, there is explicit support for sampling 2D textures. They provide hardware-assisted interpolation of pixels and efficient access through predicting the location of data points that are to be retrieved next.

CUDA and OpenCL allow for two distinctive ways of providing access to texture objects and samplers, *fixed* (set prior to execution) and *bindless* (allocatable during

execution) texture slots. It was chosen to implement the simpler fixed allocation scheme as it offers sufficient flexibility for our needs.

C++ does not include the concept of textures. Our C++ textures are implemented as plain two-dimensional arrays, lacking the intelligence of their GPU counterparts, but without decreasing performance compared to using arrays on the CPU.

Due to its hardware oriented nature, the texture API could not be unified from the OpenCL-C perspective. Therefore, a new unifying API was introduced with wrapper methods in all three languages.

Annotations

On the GPU, a hierarchy of three types of memory exists: Per thread (*local*, fastest type), per work-group (*shared*), and per device, accessible from all threads (*global*, slowest type). On the CPU, all addressable memory is accessible from all threads.

OpenCL and CUDA require the annotation of functions and objects to instruct the compiler where to place them in memory. The CUDA API is wrapped in the OpenCL notation, and for C++ statements are added that are nullified by the preprocessor.

5.2.3 Towards a Common Execution Model

Conceptually, the steps required in getting an OpenCL or CUDA-device to execute program code are highly similar. For the purpose of obtaining a single execution model, we will consider them equal and focus on bridging the divide between their execution model and that of CPUs. Effectively, this common execution *model* implements the common execution *hierarchy* as defined in Section 5.2.1.

Program / Kernel / Function

OpenCL and CUDA have the concept of a *program* (*kernel*) that needs to be compiled prior to execution. Both define an explicit memory hierarchy and require allocating and copying data to separate device memory in preparation of execution. Next, the program is loaded through API calls.

Conversely, there is no preparation for executing a CPU *function* from within the context of its containing CPU-based C++ program, other then passing it *arguments*. In support of the single execution model, a GPU-mimicing API has been implemented in C++, mostly through inert (i.e., empty) function implementations.

Thread Control

The GPU supports a two-level thread hierarchy and three dimensions per level. Specifying on how many threads per work-group and in how many work-groups code should run is an important performance aspect in executing a GPU program/kernel. Furthermore, one specifies the layout of the work-groups and the

threads within the work-groups conveniently for the task at hand, e.g., governed by the two-dimensional layout of the image data to be processed.

The CPU on the other hand has a flat, single level hierarchy with singular dimensionality. Threads need to be explicitly managed, unlike the implicit thread synchronization in the GPU. On GPUs, threading is inherent to the execution model, one only specifies the number of threads that are to be run in what topology. On CPUs, there is a cost associated with forking new threads and the joining of threads.

There exist solutions for implicit management of threads for C++ (e.g., *OpenMP*) but their offered execution model is unreconcilable with that of the GPU. As such, standard C++11 threading is used within the framework.

Memory

GPUs are often layed out as (PCI Express) expansion cards, so-called *discrete* GPUs. These boards come with their own memory modules and memory controller. To execute a GPU program, API calls need to be made to allocate GPU memory, copy data from the host memory to the device and copy the program's results back to the host. The operating system has no direct control over this memory space.

There exist GPUs that are part of the same *die* as the host's processor and that effectively share the main memory space with the CPU. For this case, memory needs to be reserved in the main pool and made available to the device. Such GPU memory is physically shared but logically separated.

In both OpenCL and CUDA, there is a feature called *unified memory* that hides the device/host separation, but such unified memory, especially in the discrete-GPU case, continues to exhibit the reduced performance of a separated layout. Within the common execution model, we have chosen to expose the separated layout and require the programmer to perform the explicit copying between memories, although the implementation in the CPU case consists of inert methods.

5.3 Practical Technical Challenges

This section lists the notable practical issues that arise when working to reconcile OpenCL, CUDA, and C++ in a cross-technology framework. There are differences to overcome in compilation workflow, linking, and precision (next to handling platform-specific bugs and oddities).

Compilation Workflow

C++ uses *ahead-of-time* (AOT) compilation for the CPU source and embedding of the GPU program into the application. CUDA uses AOT compilation as well (see below), whereas OpenCL uses compilation at runtime. Prior to executing code implemented using the cross-technology framework, there is a required compilation

step, regardless of the target language that was selected. This step handles AOT compilation where necessary.

When developing in CUDA, the choice is to be made between using the low-level *CUDA Driver API* or higher-level *CUDA Runtime API* (see [12]). The CUDA Runtime API is not fit for our purposes as it hides device access and handling in an attempt to decrease the effort of programming on the GPU: CUDA C statements are to be mixed with CPU code in the C/C++ source files, and at compile time the **main()** function is modified and all kinds of support functions added in a developer-opaque way. Instead, we employ the CUDA Driver API with separate .cu source files, compiled to .cubin files and accessed from the CPU source. From CUDA version 7 on, it is possible to use runtime compilation instead of AOT.

Runtime Linking

Both OpenCL and CUDA require the compiled application to be dynamically linked to a library that implements the API. In a lot of cases, this yields a binary that cannot be used with other versions of the same library and or cannot be used on other systems.

In employing *OpenGL*, it is common to use a library called *glew* (OpenGL Extension Wrangler) that finds and links to the correct API-implementing library at runtime. It happens that there are *clew* [34] and *cuew* [35] variants that offer this functionality for OpenCL and CUDA. These were used in order to facilitate easy switching between API-implementing libraries and systems, one of the goals of developing the cross-technology framework.

Numerical Precision

Not all hardware guarantees the same level of numerical precision. This can lead to differing outcome in running equivalent code on multiple hardware platforms.

The *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754) standardizes how results of floating-point arithmetic should be approximated [43]. CPUs perform floating point operations using either the *x87* or *SSE* instruction set [6], where the latter follows IEEE 754 more strictly [43]. GPUs typically do not follow IEEE 754 in all modes of operation, allowing for optional less precise and faster computation [20, 22].

To make sure that results are equal, independent of the target language and hardware platform chosen, we adopt 32-bit floating point numbers and operations, using SSE on the CPU and not employing fast-math routines on the GPU.

5.4 Employing the Framework Illustrated

We illustrate the usage of the cross-technology framework by implementing a simple function and creating entry points for calling it from OpenCL C, CUDA C, and C++.

```
#ifdef __KERNEL_CPU
#include "cpu_kernel.h"
#endif

DEVICE_NAMESPACE_BEGIN

dev_inl void sum_partial_results_uint8(
                              dev_global uint *partial_histogram,
                              int num_groups,
                              dev_global uint *histogram)
{
  int tid = (int)get_global_id(0);
  int group_indx;
  int n = num_groups;
  uint tmp_histogram;

  tmp_histogram = partial_histogram[tid];

  group_indx = NUM_BINS*3;
  while (--n > 0)
  {
    tmp_histogram += partial_histogram[group_indx + tid];
    group_indx += NUM_BINS*3;
  }

  histogram[tid] = tmp_histogram;
}

DEVICE_NAMESPACE_END
```

Figure 5.2: *Implementation*. Function that sums results over partial histograms, implemented in the cross-technology framework. **num_groups** is the number of work-groups that was used to compute partial histograms, **partial_histogram** is an array of **num_groups** $\cdot$ 256 $\times$ 3 $\times$ 32-bits entries; 256 bins for values of red, followed by 256 bins for values of green, and 256 bins for values of blue.

```
dev_inl void sum_partial_results_uint8(
                              dev_global uint *partial_histogram,
                              int num_groups,
                              dev_global uint *histogram);
```

Figure 5.3: *Header File*. Header of function that sums results over partial histograms, implemented in the cross-technology framework.

After implementing the function, see Figure 5.2, a header file is to be created for it, see Figure 5.3. Through including this header file, entry points are obtained for the target languages that have the following layout (see Figure 5.4, 5.5, and 5.6):

include compat_myplatform.h

include algorithm.h

address_space qualifier void kernel_name(kernel_args) {
    real_algorithm(kernel_args);
}

After compiling it for at least one back-end per target language, the function can be run.

```
#ifdef OCL_USE_ATOMICS
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable
#endif

#include "compat/compat_opencl.h"
#include "sum_partial_results_uint8.h"

kernel void histogram_sum_partial_results_uint8(dev_global uint *
    partial_histogram, int num_groups, dev_global uint *histogram)
    {
     sum_partial_results_uint8(partial_histogram, num_groups,
    histogram);
}
```

Figure 5.4: *OpenCL Entry Point*

```
#include "compat/compat_cuda.h"
#include "sum_partial_results_uint8.h"

__global__ void histogram_sum_partial_results_uint8(uint *
    partial_histogram, int num_groups, unsigned int *histogram) {
  sum_partial_results_uint8(partial_histogram, num_groups,
    histogram);
}
```

Figure 5.5: *CUDA Entry Point*

```
#include "compat/compat_cpu.h"
#include "sum_partial_results_uint8.h"

void histogram_sum_partial_results(uint *partial_histogram, uint *
    histogram, uint num_threads){
  for(int i = 0; i < num_threads; ++i){
    for(int j =  0; j < 256*3; ++j ){
       histogram[j] += partial_histogram[i * 256 * 3 + j];
    }
  }
}
```

Figure 5.6: *C++ Entry Point*

Chapter 6

Atomics

Implementing a program that has to run concurrently on multiple *processing elements* (PEs, see Section 5.2.1) adds challenges when compared to a single-threaded program. A very important class of these challenges arises from utilizing shared resources (e.g., OpenCL's work-group shared *local* memory, see Section 5.2.1). Data race conditions can and will occur unless special care is taken to avoid them.

A lot of computer hardware and programming languages have special support for making sure that data races cannot happen, but enforcing this decreases performance. A common approach of mitigating this is to have the programmer indicate which calls are to be protected. One way used to convey the intent that an operation should not be susceptible to a data race is through the concept of an *atomic operation*.

Atomic operations, like their name implies, are always executed as a whole; other operations competing for the same resources are prevented access until the atomic operation has finished. In reality, most operations are not inherently atomic but they can be made to appear so through both software and hardware. When supporting these kinds of operations with hardware, the performance is usually impacted positively as compared to using for example *mutual exclusion* in software to achieve the same effect.

The parallel calculation of (partial) histograms relies heavily on these atomic operations, particularity where we lookup the appropriate bin in the intermediary result and then increment its counter (see Algorithm 2 and 3 in Chapter 4). With these steps performed for each pixel in the image under analysis, the performance of this operation is essential to the total runtime of the histogram procedure.

In this chapter, we will first illustrate the impact of write collisions, then lay out a common way of implementing hardware-assisted atomics, after which we present a model that simulates the impact of software locking.

6.1 Write Collisions

To get an indication of the chance of write collisions to occur within the histogram procedure, we turn to the well-known *Birthday Paradox*. In a group of 21 people, there is an approximately 50% chance of a collision in birthdays to occur. The Birthday Paradox is defined as follows, for b bins and a group size of n,

$$P_{b,n} = 1 - \left(\frac{n!}{b^n} \cdot \binom{b}{n} \right). \tag{6.1}$$

Reducing the number of bins to 256 (color values), and increasing the group size to, e.g., 32 (number of processing elements within a compute unit), the chance of a collision to occur goes up,

$$P_{256,32} = 1 - \left(\frac{32!}{256^{32}} \cdot \binom{256}{32} \right) \approx 0.87. \tag{6.2}$$

Furthermore, as the color values within a block of data are obviously unevenly distributed, we can expect collisions to occur in determining histograms.

6.1.1 Latency Hiding

GPUs can mitigate stalling resulting from write collisions through *latency hiding*. This works through *over-allocation* and efficient *context switching*. It is possible to allocate more threads on a compute unit (CU) (see Table 5.2) than it has processing elements (PEs) available, in the form of more threads per block or multiple blocks. When certain threads stall, the CU switches to different threads, allowing the stalled threads to get unblocked in the meantime. Latency hiding is less effective in a situation where locking is not achieved through hardware-assisted atomics but relies on an software scheme, inherently less efficient. This is the case on Nvidia Fermi and Kepler cards.

6.1.2 Compare and Swap

As atomics play an important role getting performance out of multi-processor systems, hardware vendors have started to add support for atomics quickly after the first multi-processor machines became available. For GPUs, atomic support has to be implemented per non-private memory type. The Nvidia Fermi architecture does not include atomic support for global and per-block shared memory. Nvidia Kepler has global atomic support, whereas Maxwell introduced atomics for per-block shared memory.

One of the most common ways to implement atomics is the so called *compare-and-swap* operation. The basic idea is that one updates the target value if and only if the current value equals the expected value, hence, the value at the start of the operation. In case the check fails, the operation is repeated. This leaves a vulnerability, however, for the so-called *ABA* pattern.

The ABA pattern entails that we aim to do an update that is valid in the context of the state being A, while in the meantime, other operations change the state to B and successively back to A. In this case, it is not guaranteed that, semantically, our update is still valid.

A means of effectively preventing this problem from occurring is extending the compare operation to check for a counter of the number times that the value was updated as well. At some point, this counter will overflow and wrap around, but with increasing bit width, it becomes exceedingly unlikely of both value and counter being what is expected after ABA-like patterns occurring. This type of augmented compare-and-swap operation is used in modern Intel processors.

6.2 Absence of Hardware Atomics Simulated

In general, for GPU-like cases, it is advised to have a high ratio of computation versus memory access. This is based on a relatively high cost of doing memory operations, especially on global memory, in which on some cards the per-block shared memory is implemented as well.

We use a mathematical model to simulate the absence of hardware atomics. The histogram procedure is assumed to operate on per-block shared memory. All memory write operations, whether successful or not, are assumed equally expensive and reads are free. Without hardware-assisted atomics, threads need to obtain a software lock. With hardware atomics, locking is done so efficiently that it is no longer a dominant factor on performance. Of each compute unit (CU), only one processing element (PE) can have a lock for a memory location at the same time. The actual number of PEs per CU is used, that is, over-allocation is not included in the model. Furthermore, as in the histogram procedure the number of compute operations is highly limited, the compute time is omitted in the simulation model altogether to better visualize the impact of the lock operations.

Figure 6.1 displays the minimum amount of expected collisions per block of pixels in a two-dimensional image (the example image analyzed in Figure 4.1 is revisited). Most 4×8-sized blocks of pixels have expected number write collisions in the range 6–8.

6.2.1 Simulated Impact on Performance

We can easily compare expected CU performance for the average and worst case in the absence of hardware atomics. Assuming a perfectly random image, statistics practically guarantees (see Section 6.1) the occurrence of blocks with two pixels sharing the same value, whereas blocks with three pixels with the same value would be relative rare. In the worst case, all color values in a block for a certain channel are equal.

We can calculate the impact on performance, for example, for the Nvidia Fermi CU, which is 32 PEs wide. In the worst case, all operations would happen as if the

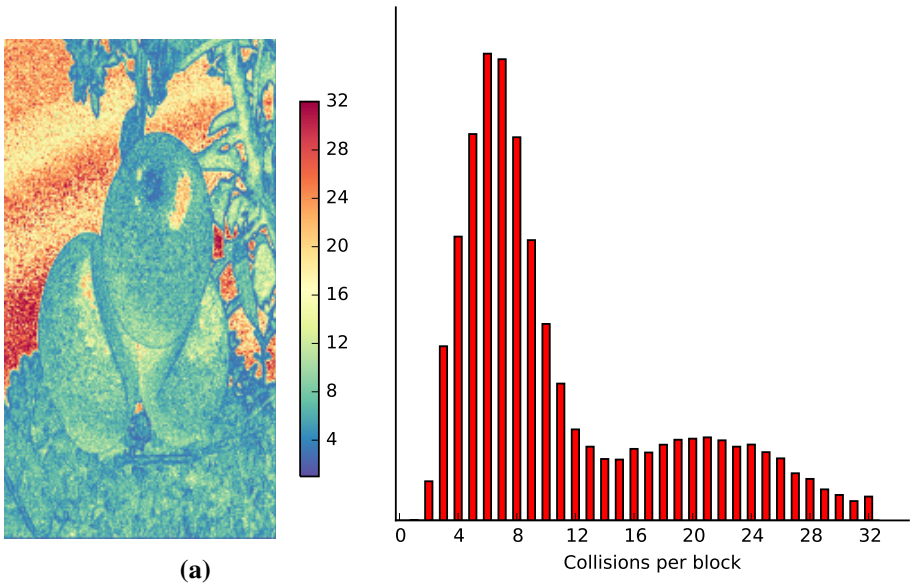

Figure 6.1: *Occurrence of Write Collisions Illustrated.* Of a two-dimensional image, per 4×8-sized block of pixels, the number of occurrences of the most-occurring color value in the red channel is reported. This serves as a minimum for the expected amount of write collisions that will happen in this block in generating a histogram using a CU containing 32 PEs. In (a), the minimum number of expected write collisions is plotted per block of pixels (displayed as a single pixel), in (b), the occurrence count of blocks with a certain minimum number of expected write collisions is displayed.

CU only consists of a single PE, thus 32 times slower than the best case. Compared to the average case, the performance penalty would be $\frac{2}{32} = 16$ times slower.

The model used to simulate performance of a CU with certain size is given in Figure 6.6. The function **create_threshold_image()** provides control to test performance from the average case (perfectly random image) to the worst case (all color values per channel are equal) through setting all color values smaller than or equal to a *threshold value* to 0. Increasing the threshold leads to a larger number of expected write collisions per block of data. Blocks of pixels with random color values are generated, with increasing portion of the value 0, thereby increasing the expected number of write collisions. The count of the most-occurring color value is used to express the minimum number of tries required to enable all simulated threads to complete their write operation. Performance is expressed as the inverse of the number of tries needed to finish all the work, a higher performance score thus indicating better performance. 1000 runs are used to smoothen the results per threshold setting.

Through determining the expected number of data collisions, gradually moving from average case to worst case, we can determine device-specific performance curves governed by the number of PEs per CU, see Figure 6.2. As can be seen, the

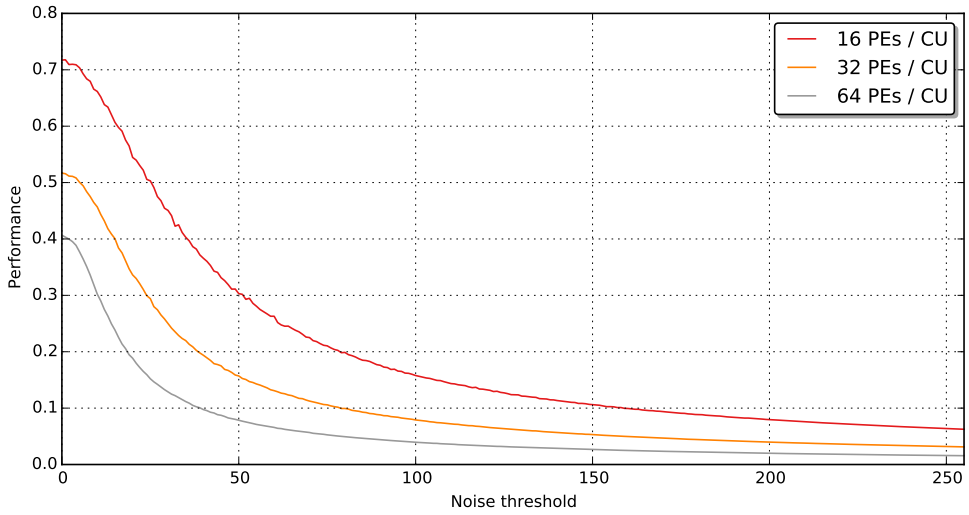

Figure 6.2: *Simulated Impact of Collisions on Performance.* For increasingly homogeneous random images, the performance resulting from expected write collisions is simulated for differently sized CUs, in absence of hardware atomics. The noise theshold indicates which part of the color values are set to 0. Performance is reported as the inverse of the number of rounds needed to allow all threads to finish their write operation. A smaller CU's performance is impacted less by increasing homogeneity of the input image.

simulated performance drop in running the histogram procedure is most present in wide SIMT architectures and is data dependent. Uniformly distributed data or non-posterized photos should not pose a problem. Wide SIMT architectures benefit most from hardware-assisted atomics.

This would advocate the use of wider SIMT architectures over narrower ones. However, in these results, compute times required for the reduction step (see Chapter 4) are omitted. These are greater for narrower CUs, as the input data is divided into more blocks. Furthermore, using smaller CUs in hardware is less economic with respect to the total amount of control logic required.

```
def create_threshold_image(width, height, threshold):
    ''' Threshold determines the values that are set to 0 '''
    threshold_img = numpy.vectorize(lambda x: x if x > threshold
    else 0)
    return threshold_img(randint(0,256,(width,height,3)))
```

Figure 6.3: *Model Simulating the Absence of Hardware Atomics I.*The creation an image where each pixel is either zero or some random value depending on the input threshold value.

```
def process_pixel_block(block):
    ''' Create r,g,b histograms from the block of pixels '''
    r_hist = defaultdict(int) # Initializes to 0 for all values
    g_hist = defaultdict(int)
    b_hist = defaultdict(int)
    for i in range(block.shape[0]):
        for j in range(block.shape[1]):
            r,g,b = block[i,j,:] # Obtain rgb values (see line 4)
            r_hist[r] += 1 # Increase counter of bin 'r' (256 bins
            g_hist[g] += 1 # per channel)
            b_hist[b] += 1
    return r_hist, g_hist, b_hist
```

Figure 6.4: *Model Simulating the Absence of Hardware Atomics II.*Function that processes 3 channels in a block of pixels returning 3 histograms

```
def hist_to_tries(hist):
    ''' Tries required = count of the most-occurring value '''
    return max(hist.values(), default=1) # Minimum of 1 as
                                         # performance = 1 / tries
```

Figure 6.5: *Model Simulating the Absence of Hardware Atomics III.*We take the frequency of the most occurring value as our model predicts this will be the dominating performance indicator.

```
def simulate_CU_performance(width=4, height=8, runs=1000):
    ''' Performance of CU with n PEs, n = width * height,
        modeled using blocks of pixels of width x height '''
    performance = []
    for threshold in range(256): # Gradually increase portion of
        tries = []               # value 0
        for i in range(runs):
            block = create_threshold_image(width,height,threshold)
            r_hist, g_hist, b_hist = process_pixel_block(block)
            tries.append(hist_to_tries(r_hist) +
                         hist_to_tries(g_hist) +
                         hist_to_tries(b_hist))
        performance.append(1.0 / average(tries))
    return r
```

Figure 6.6: *Model Simulating the Absence of Hardware Atomics IV.* Python code for simulating the performance of a CU with a certain number of PEs.

Chapter 7

Empirical Study

The aim of this empirical study is to determine the performance factors that impact the histogram operation, providing insight in what execution configuration to use for a certain input. In order to achieve this, we run our histogram repeatedly on different configurations of hardware and software, and with a diverse set of input images.

The testing corpus is made up of both synthetic and real-life images. The non-synthetic images have primarily been sourced from the image sharing website Flickr. The synthetic images have been constructed to study certain corner cases, as well as the influence of varying spatial distributions of equal values.

First, we lay out the setup for the study, then we explain the experiments performed, after which we analyze the performance determining factors in the results.

7.1 Testing Setup and Preliminaries

Experiments are carried out on implementations of Algorithm 2 (*Full Histogram – Partial Input Space*), described in Chapter 4. When run single-threaded, this algorithm is equal to Algorithm 1. The histogram kernel is implemented in C++, OpenCL C, and CUDA C using the cross-technology framework (Chapter 5). The implementations are bound to Python for easy access and analysis of results, as shown in Figure 7.1. The main testing loop is run in Python on the CPU, while all the actual work happens on the device being tested.

For each experiment, we make sure that the GPU is not drawing to the display or is loaded with other tasks. Prior to each test, a full tear-down and reconstruction of the device and API under investigation is performed, including context creation, device initialization, and kernel compilation, to make sure that the device's state is fully reset. The test is then run for a number of times, typically 1000.

Table 7.1 lists the tested API and hardware combinations. In testing GPU performance, the choice of system CPU is of negligible influence. Therefore, we used three different machines for the tests:

- Intel Core i7-2600K (Sandy Bridge) and an AMD R9 290X (GCN 1.1);

```
import histogram
import numpy as np

# Pick the first device, can be a CPU, OpenCL or CUDA device
dev = histogram.device_list()[0]

# Create a 1024 x 1024 x 4 'image'
img = np.zeros((1024,1024,4))

# Run histogram kernel for uint8 input 100 times over our 'image'
res_time = dev.test_histogram_uint8(img, runs=100)
```

Figure 7.1: *Test Execution Illustrated.* The test suite is accessed through bindings for Python, importable as the **histogram** module. The available devices are probed and made available through **histogram.device_list()**. In the example, the histogram kernel for uint8 input images is executed a 100 times on the first device from the list.

Table 7.1: *Tested API – Hardware Configurations*

API	Driver	Hardware	Architecture
C++		Intel Core i7-2600K	Sandy Bridge
		Intel Core i7-4770K	Sandy Bridge
OpenCL	Intel 2014 SDK	Intel Core i7-2600K	Sandy Bridge
	Intel 2014 SDK	Intel Core i7-4770K	Sandy Bridge
	Nvidia 352	Nvidia GeForce GTX 750	Maxwell [11]
	Nvidia 352	Nvidia GeForce GTX 660	Kepler [10]
	Nvidia 352	Nvidia GeForce GTX 570	Fermi [9]
	AMD 15.04	AMD R9 290X	GCN 1.1
CUDA	Nvidia 352	Nvidia GeForce GTX 750	Maxwell [11]
	Nvidia 352	Nvidia GeForce GTX 660	Kepler [10]
	Nvidia 352	Nvidia GeForce GTX 570	Fermi [9]

- Intel Core i7-2600K (Sandy Bridge) and a Nvidia Geforce GTX 660 (Kepler);
- Intel Core i7-4770K (Haswell) and a Nvidia Geforce GTX 570 (Fermi) and a Nvidia Geforce GTX 750 (Maxwell).

All machines ran Ubuntu 14.04, and five compilers were used:

- C++: GCC 4.8 with glibc 2.19;
- OpenCL C: Intel 2014 SDK compiler;
- OpenCL C: Nvidia CUDA SDK 6.5.19;

- OpenCL C: AMD OpenCL 2.0 compiler;
- CUDA C: Nvidia CUDA SDK 6.5.19.

7.1.1 Execution Stages

Depending on the hardware – API platform that the test suite is executed on, we execute a (sub)set of the following steps. The steps required on discrete GPUs running OpenCL or CUDA are (almost) identical.

Compile, Link, Load

First, the kernel needs to be compiled and linked for the device. OpenCL allows for both runtime compilation and loading from a pre-compiled binary. CUDA allows for runtime compilation starting from version 7. We used runtime compilation with OpenCL and pre-compilation with CUDA version 6.5.19.

Transfer Input Data

Once the device-specific binary code has been loaded into the driver, we need to transfer the input data to the device. For discrete GPUs, this involves copying memory contents from the host memory over to the device, such that these are stored in a physically separated DRAM that is located on the device.

Invoke the Kernel

Then, the kernel can be called with the launch parameters. Via the API, any shared memory, read-only ranges, and other things required are set up so that the kernel can begin its execution. Given that this is done for potentially tens of thousands of threads and is a non-trivial operation, this might take anywhere from a few micro- up to milliseconds, depending on the actual device and the launch parameters.

Run the Kernel

Then the kernel runs. Once all threads are done (note that they might not all finish at the same time), the device hardware has to notify the host that it has completed the kernel.

Transfer Output Data

Lastly, the output data has to be retrieved from the device's DRAM to the main system DRAM.

When executing the same kernel on different data in rapid succession, these steps are usually pipelined in order to achieve a better throughput: Transfer of data and

actual computation are performed overlapping so that either the computation or the transfer can be the bottleneck, but not the serial result of both.

For integrated GPUs that share memory with the host CPU, the overhead of copying data back and forth might be reduced to passing ownership of a few pages of RAM plus any associated overheads in the OS or driver. This way, the data transfer is then no longer linear in cost to the size of the data. This is known as *zero-copy*.

For a CPU, the execution flow is radically simpler. All stages up to starting the actual kernel are non-existent, and compilation has already been done, at compile time of the host application. The actual execution is just a function call and the completion is the return of that call.

7.1.2 Kernel Time vs. Application Time

In order to better reason about time required for executing the kernels, we introduce two notions of execution time, see Figure 7.2:

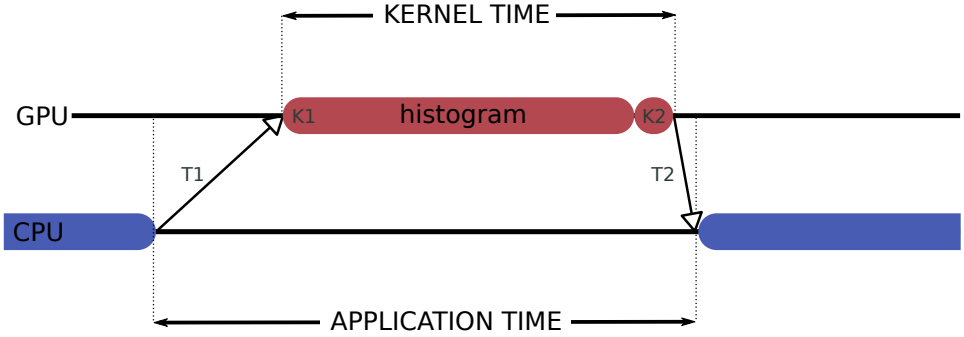

Figure 7.2: *Notions of Execution Time.* Kernel time is the duration of executing the histogram kernel, where K1 generates the histograms over the divided input data and K2 combines these histograms into one (see Algorithm 2, Chapter 4). Application time is the kernel time plus the duration of the input data transfer T1 and the output data transfer T2.

- **Kernel time**: The 'naked' kernel execution time. This is the time it takes to run the actual kernel, not accounting for data copy overheads. The kernel time encompasses the time needed to generate full histograms over the divided input data and the following reduction step that combines these full histograms into a single histogram for the whole input data (see Algorithm 2, Chapter 4). Under normal conditions, the first kernel's execution time is linear to the input data size. The second kernel is also linear in its execution time, but requires no locking mechanism (i.e., hardware atomics or software mutexing) and the input data size is two orders of magnitude smaller.

- **Application time**: Total execution time of the kernel call. This includes only the kernel time and the time required for data transfer. We are able to measure this by pre-running the kernel once prior to doing measurements.

This way, it is already compiled and loaded, image data has been read from disk, and the time required for this does not show up in the results.

7.1.3 Fast High-Resolution Time Measurement

Multiple calls to the application (i.e., kernel execution and data transfers) are started simultaneously, essentially queuing them as they are forced to wait on each other due to data dependencies. These multiple calls are timed as a whole. This approach makes sure that time introduced by possible notifications and interrupts from the operating system is omitted. For measuring kernel time, the same approach is used: Multiple kernel calls, omitting the data transfers, are executed and timed together.

The expected kernel runtimes are between a few microseconds and a few seconds. Accurately measuring elapsed time less than a millisecond is not trivial. We utilize a timer based on the RDTSC instruction that reads out a high-precision counter register in a single CPU clock tick, available on modern Intel processors (see Section 17.13.1 3b of [6]). This timer gives us 3×10^9 increments in one second on a 3 GHz machine and a resolution greater than one nanosecond, which is more than enough to accurately measure phenomena in the microsecond range. An amount of 1000 calls was chosen for measuring kernel time and application time, as this yields a high enough total runtime for accurate measuring while keeping the runtime sort of reasonable. The whole suite of tests completes within a few hours on most hardware.

Next to image characteristics, execution time is likely to be influenced by the image size, i.e., the number of pixels contained in it. Performance is therefore reported in pixels per second. The kernel time will be used to determine these performance numbers over.

7.2 Experiments

We list the experiments performed in this study in the order in which they were performed. As a first comparison, the histogram procedure (Algorithm 2, Chapter 4) is performed on six real-world images, shown in Figure 7.3. The GPU configurations show clearly superior performance compared to their CPU counterparts, with, strikingly, the best performance being reached by an Nvidia GeForce GTX 750 Ti card operating under OpenCL instead of Nvidia's 'native' CUDA. On the higher performing devices, it is shown that images A and B are special cases, because they lead to collisions in processing. The effect of contention is most visible on the Nvidia GeForce GTX 570 and 660, and less so on the AMD R9 290X and Nvidia GeForce 750 Ti.

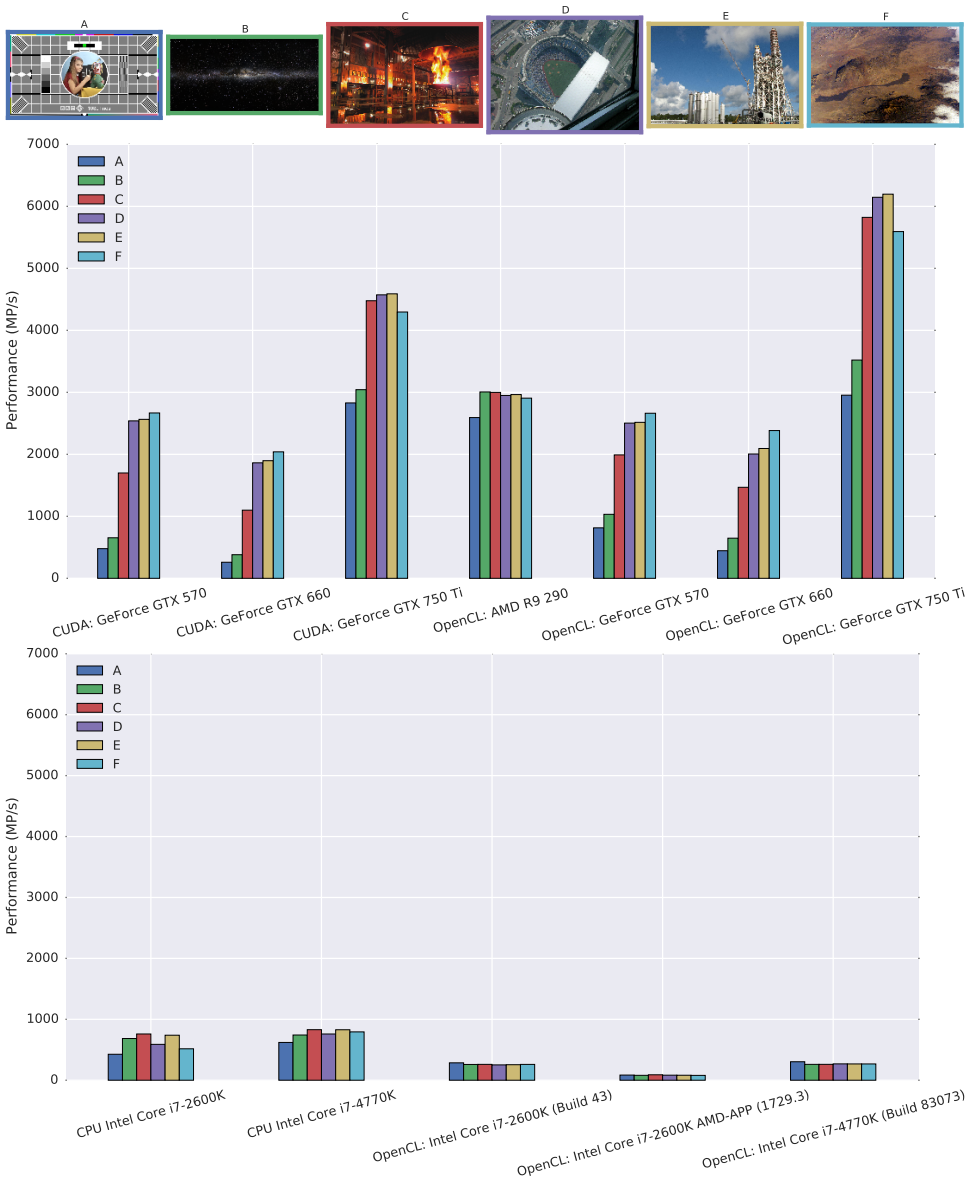

Figure 7.3: *Comparison on Real-life Image Set.* Three GPU – CUDA, four GPU – OpenCL, two CPU – C++ (C++11 threads), and two three CPU – OpenCL configurations are compared on a real-life set of six images. The GPU configurations clearly show better performance than the CPU configurations. Results are normalized for image size, showing that images A en B are special input cases that hinder performance.

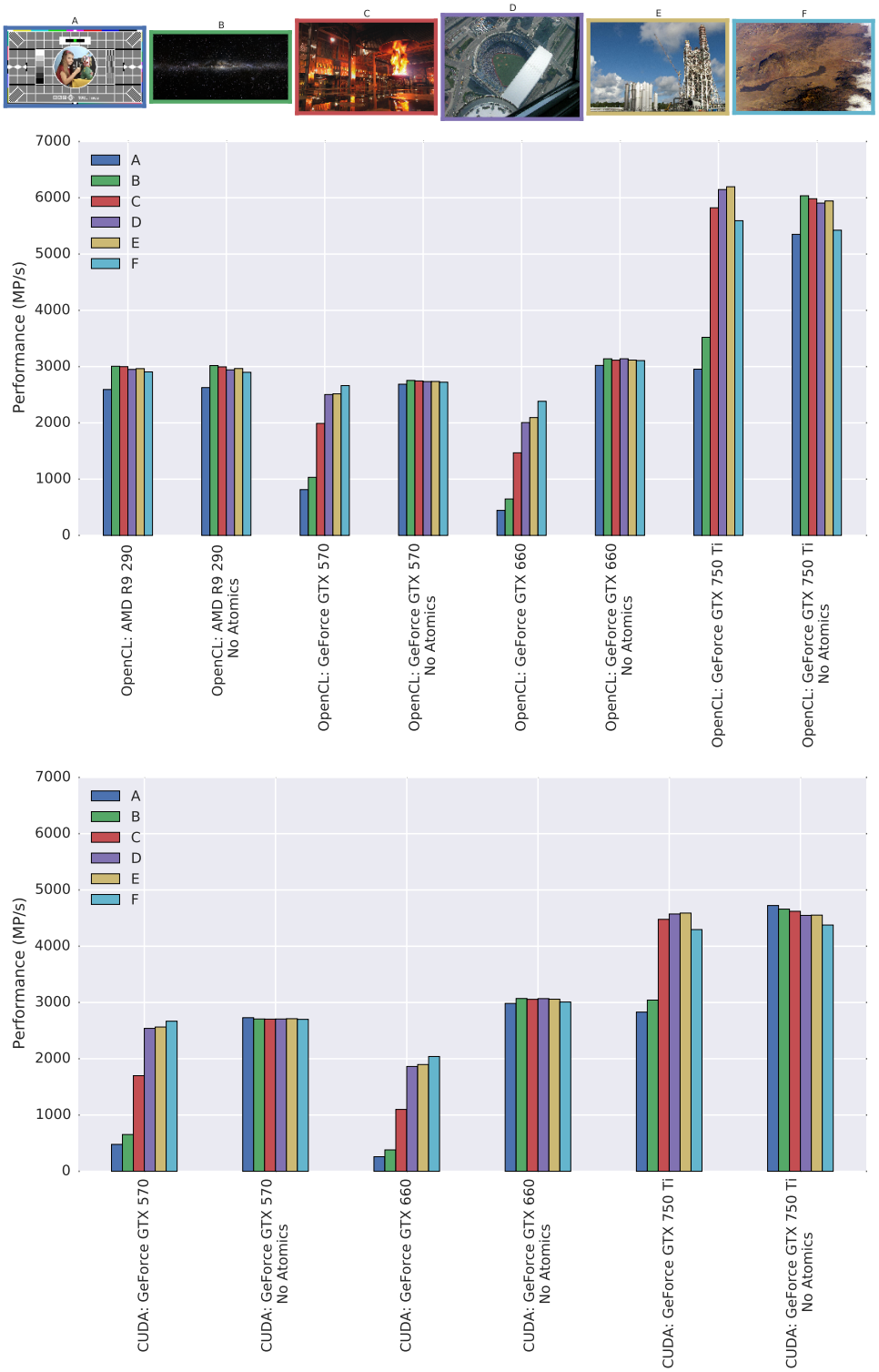

Figure 7.4: *Cost of Employing Atomics.* On GPU configurations, we determine the overhead of using atomic operations. Employing (the necessary) thread-safe operations clearly decreases performance. Again, the difference in speed is most-visible on images A and B.

7.2.1 Cost of Employing Atomics

In the follow-up experiment performed, we examine the computational overhead induced by employing atomic operations versus their non-thread-safe counterparts; only on GPU configurations, as these cannot be omitted on CPUs. Note that using non-thread-safe operations generates invalid outcome in case of write collisions, but here we are interested in comparing speed of operation only.

The results are shown in Figure 7.4. Atomic operations introduce overhead in the form of checking and setting the mutex, and in requiring threads to wait for other threads to complete their operation on the same memory address.

Atomic operations are either hardware-assisted (on Nvidia Maxwell architecture: GeForce GTX 750 Ti, and on AMD's GCN architecture: R9 290X) or implemented through software mutexing (on Nvidia's Fermi and Kepler architectures, GeForce GTX 570 and 660). This is visible as R9 290X does not show performance decrease and GTX 750 Ti only shows performance drops on contention-heavy images A en B, whereas GTX 570 and GTX 660 show overall better performance in not employing atomics.

GTX 660 is impacted more by enabling atomics than GTX 570. As GTX 570 comes with more CUs, this possibly allows for more effective latency hiding (see Section 6.1.1) as less data needs to be processed per CU.

7.2.2 Performance under Increasing Contention

Next, we examine the performance under increasing number of collisions in generating valid histograms, that is, using atomic operations. We run the histogram procedure on images consisting of pixels with uniform randomly picked color values, i.e., white noise, all color values are equally likely. The idea is that if we generate unbiased white noise, we will get really low pressure on the atomics (but not zero collisions, in a block of 32 pixels, likeliness of a collision to occur is 87%, see Section 6.1).

Thresholded Random

In generating an image containing white noise, we set a *threshold value* below which all color values will be set to 0. As such, a threshold value of 3 means that no pixel will have color values in the range $[1, 3]$ and that the value 0 is four times as likely to occur as the remaining color values. This is the method of controlling the expected number of collisions described in Section 6.2.1. All *added* write collisions occur in a single bin.

The results in running all platform configurations on images of 2048 pixels are plotted in Figure 7.5. GeForce GTX 570 and GTX 660 show exponentially decreasing performance, while it kicks in later on the GTX 570, probably due to the larger number of CUs. On GeForce GTX 750 Ti, the decrease is less than exponential and smaller percentage-wise. The R9 290X shows little impact of increased

contention, and on the CPUs, Intel's OpenCL driver shows best performance, followed by C++ and AMD's OpenCL driver coming in last. The jitter shown in the performance curves can be due to the cards getting overheated or another temporary suboptimal system state.

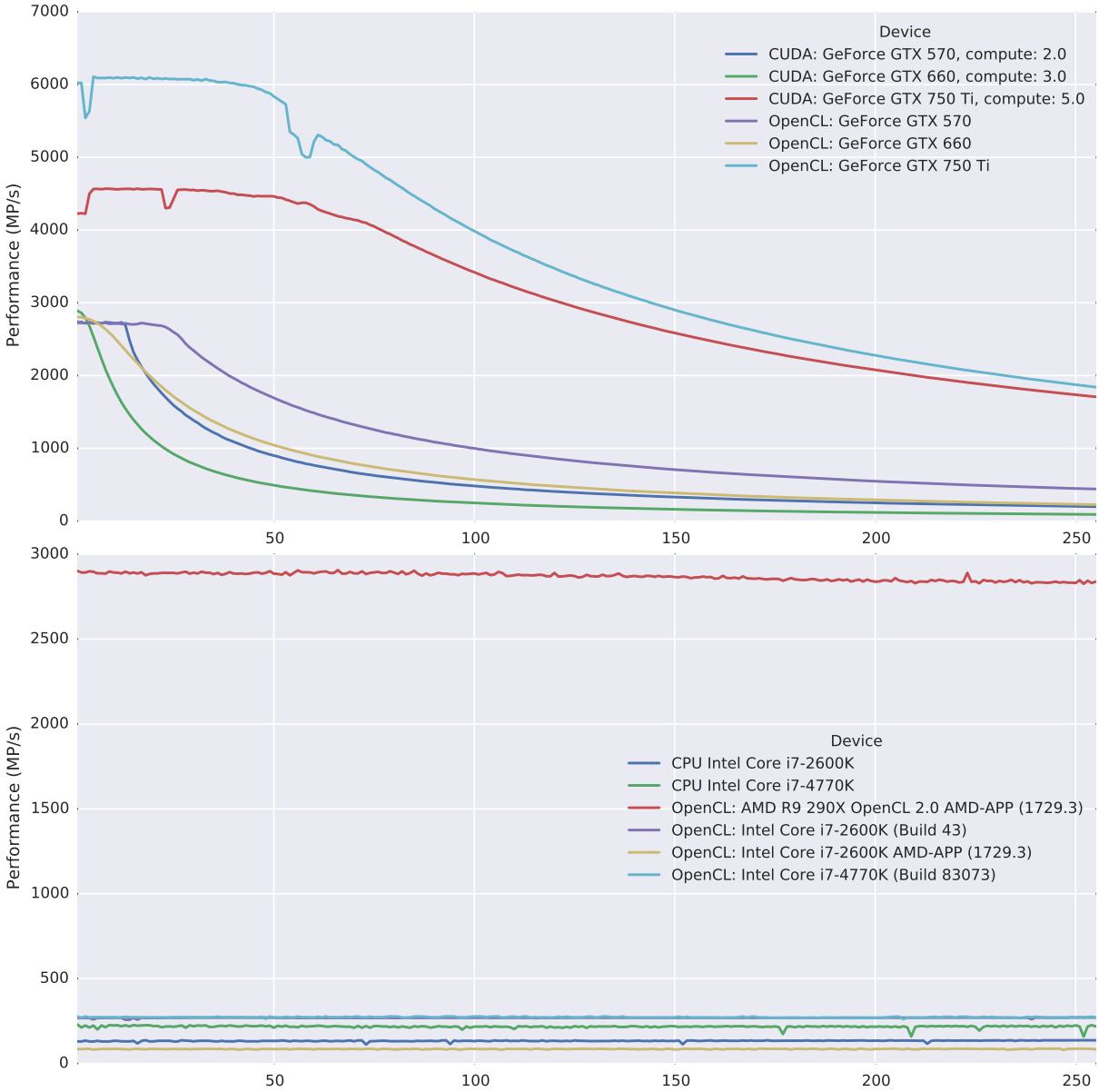

Figure 7.5: *Performance under Increasing Contention.* On random images of 2048 pixels, the noise threshold below which all values are set to 0 is gradually increased, increasing contention. There is exponentially decreasing performance on GeForce GTX 570 and GTX 660, whereas GeForce GTX 750 Ti shows less decrease and R9 290X appears almost unhindered by increasing contention.

7.2.3 Inherent Cost of Employing Atomics

In the fourth experiment, we assure zero collisions through pre-processing input images, blockwise eliminating the possibility of collisions to occur by iteratively changing pixel values. As stated, this is required as random images still give rise to a minimal number of collisions. Furthermore, it is difficult to concisely determine which parts of an image are prone to generating collisions with the naked eye. In ensuring zero collisions, the inherent cost of atomic operations can be measured, that is, the time required for checking and setting the mutex only.

Depending on the work-group size and layout set for a device, we can execute a routine that slightly modifies the input image to make sure there will occur no collisions in processing each block of pixels. The procedure is as follows: In a first pass, per block of pixels to be processed by a single work-group, a histogram is generated. In the second pass, for each pixel is checked whether its color values have a count of more than one in the histogram. If so, with increasing distance is sought for a color value that is still unused, after which the histogram is adjusted.

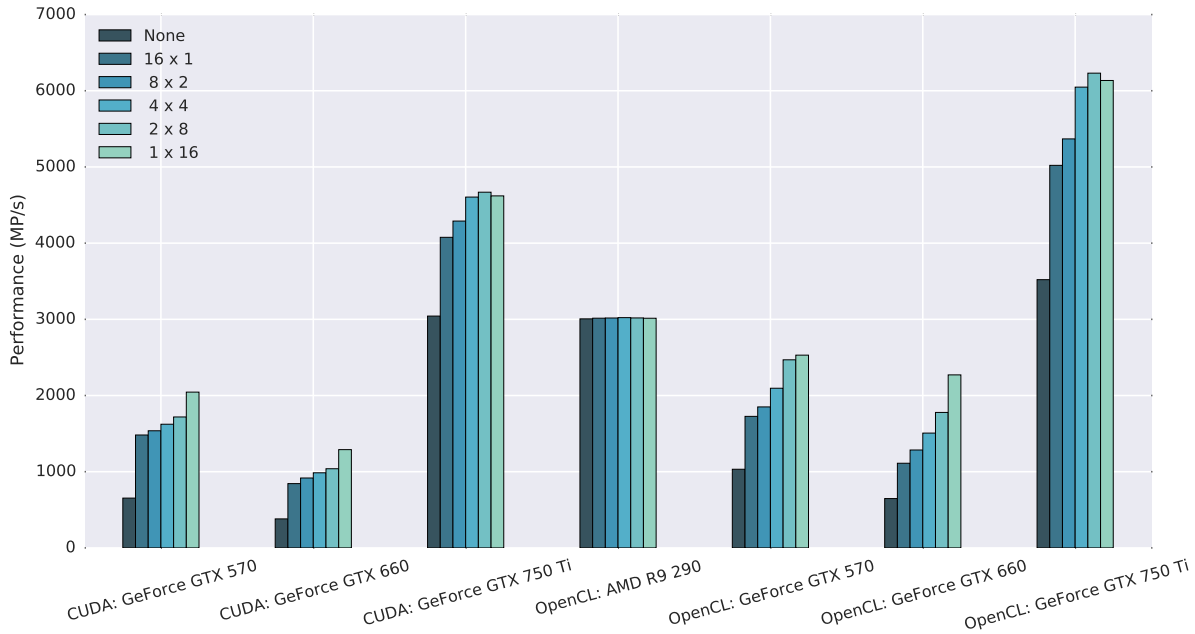

Figure 7.6: *Collision Elimination Compared.* Different strategies of traversing a 16-by-16 block of pixels and clearing it of collisions are compared. The legend describes how 16 pixels per step are cleared of collisions relatively. Strikingly, there is difference in performance after traversing in, e.g., 16×1 and 1×16.

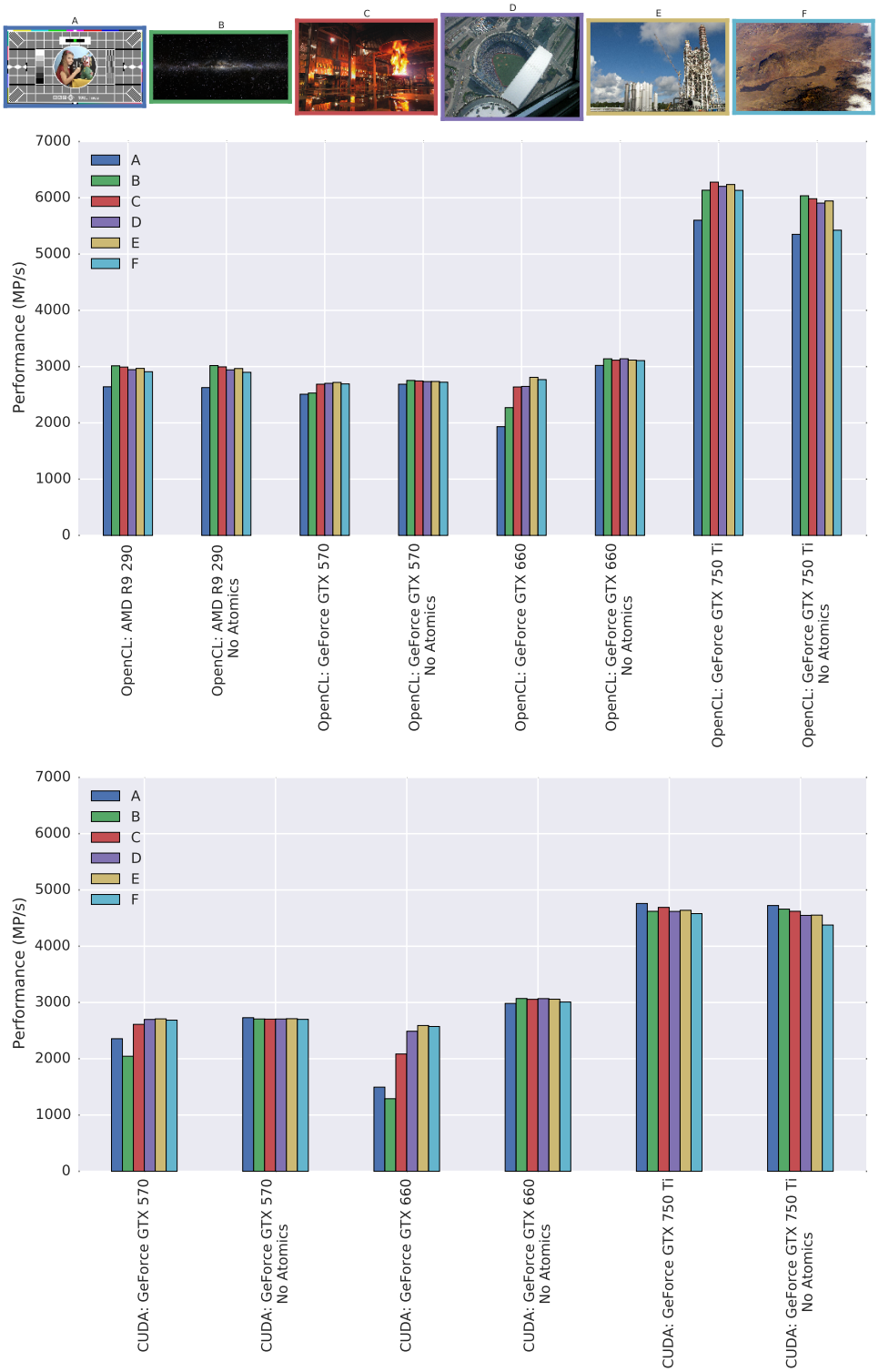

Figure 7.7: *Inherent Cost of Employing Atomics*. After eliminating collisions, most devices show similar performance in employing atomic and non-atomic operations. GeForce GTX 570 and 660 show performance lag in employing atomics, while GeForce GTX 750 Ti seems to in non-atomic operations.

On image B, see Figure 7.7, different ways of eliminating collisions are compared, see Figure 7.6. There is difference in performance given rise to by the way in which a block of 16-by-16 pixels is subdivided into 16 pixel blocks that are made free of collisions. This gives insight in how the over-allocated CUs (see Section 6.1.1) divide the 16-by-16 block over the actual PEs. Again, we see that AMD's R9 290X is unaffected by different levels of contention, but is greatly outperformed by Nvidia's GeForce GTX 750 Ti, both shipping with hardware atomics.

We return to the six real-life images and clear them of atomics using 1×16 traversal, after which devices should show equivalent performance in employing atomic and non-atomic operations. The results are shown in Figure 7.7. Software mutexing GeForce GTX 570 and 660 show less performance decrease in employing atomic operations. While this includes the inherent cost of running their atomic operations, it should be noted that possibly the images are not entirely collision free as this would require traversal in strides of size 32. Again, GeForce GTX 570 is hindered less than 660 due to more CUs. R9 290X shows the expected equivalent performance, employing its hardware-assisted atomics has no inherent cost, whereas GeForce GTX 750 Ti even seems to show better performance in employing its hardware-assisted atomic operations than in its non-atomic operations.

7.3 Results Analyzed: When to Use What?

Based on the outcome of the experiments, the tested devices can be divided in three distinct classes:

- **Immune**: The first class shows consistent performance that is unaffected by changes in the expected number of collisions. These devices are not impacted by anything other than the image size. This class consists of the non-SIMT hardware, only the CPUs in our case, both in running OpenCL and C++ implementations of the histogram procedure;

- **Weakly influenced**: The second class is somewhat impacted in case of extreme (i.e., artificially high) pressure on the atomic operations. The hardware in this category is AMD's GCN-based card and Nvidia's Maxwell card. These cards have hardware support for atomics and the tests show the usefulness of this feature;

- **Strongly influenced**: The third class is the hardware strongly influenced by the performance of the atomic operations. The Nvidia Fermi and Kepler cards are in this category, both in running OpenCL and CUDA. They show orders of magnitude worse performance under extreme collisions than the weakly influenced devices. There is a strong correlation between the expected number of collisions and the resulting runtime, due to relying on software mutexing for handling atomics within a work-group.

Next, we analyze the performance determining factors in more detail, from the perspective of collisions, by comparing CUDA with OpenCL and OpenCL with C++, and by comparing CPU and GPU.

7.3.1 The Impact of Contention

For two images of identical size, the amount of work a device needs to do to build a histogram in terms of the number of executed statements is equal. There exist cases, however, where two images of identical size give rise to wildly differing runtimes on the same device.

The main reason lies in contention for the same memory location. Since the algorithm needs atomic operations on shared memory, threads in a work-group trying to update the same value end up in a sequential queue. The time needed to service all colliding threads depends on the speed of the atomic operation. During this time, the whole work-group of threads, also those finished already or those that had different values to update, is stalled.

The tests with synthetic images allow us to control the expected number of collisions, enabling us to visualize the relation between this pressure on the atomics and the runtime. Based on the results, we can say that this relation is architecture specific: Most affected are the Nvidia Fermi (GeForce GTX 570) and Kepler (GeForce GTX 660) cards, Nvidia Maxwell (GeForce 750 Ti) and AMD GCN (R9 290X) cards are not so much affected. CPUs have a single PE per CU in our used Mapping II, see Section 5.2.1, Figure 5.1, thus cannot have collisions occurring.

The results put Fermi and Kepler in their own subclass over all hardware tested. The reason behind this is the absence of hardware atomics in these architectures, therefore relying on slower software mutexing, which makes that the cost of the atomic operations becomes the dominant factor in the results. On test case Theshol-ded Random, see Section 7.2.2, both in employing OpenCL and CUDA, similar exponentially descreasing behavior is shown. This is exactly the trend predicted in our artificial model simulating performance in absence of hardware atomics, see Section 6.2. Here was assumed that taking the inverse of the maximum number of collisions occurring in a block of pixels assigned to a single work-group predicts performance.

7.3.2 CUDA vs. OpenCL, OpenCL vs. C++

Given the fact that CUDA is only available on Nvidia hardware, any comparison between CUDA and OpenCL is highly limited. Within Nvidia's platform, both OpenCL and CUDA are implemented on top of Nvidia's pseudo-assembly language *Parallel Thread Execution* (PTX). While typically new features are supported in CUDA first, this does not influence our results, in which operating under OpenCL typically shows better performance than under CUDA.

The same holds for comparing OpenCL to GCC-compiled C++, the comparison can be done on the three used CPUs only. The framework maps each core in

a CPU to a compute unit (CU) with a single processing element (PE). All tested CPUs support $4 \times 32 = 128$ bit SIMD instructions, and while explicit vectorization was not used in the framework (see Section 5.2.2), both OpenCL and GCC attempt to auto-vectorize instructions (see Section 3.1.1), effectively giving rise to CUs containing four PEs. Intel's OpenCL CPU driver is able to execute more efficiently than GCC-compiled C++, possibly due to the fact that vectorization is more native to OpenCL and that more information on thread interdependency is provided via the program code. AMD's OpenCL driver, however, is outperformed by C++, while it should be able to run on Intel CPUs without sacrificing performance, which points at a less mature OpenCL implementation.

7.3.3 CPU vs. GPU

When run on the CPU, the runtime of the histogram kernels (K1: generation, K2: reduction, see Figure 7.2) is unaffected by the image data itself and appears to be linear with the input image size. This is inherent to the used mapping, while the tested CPUs do come with hardware atomic support. As data transfers to and from a discrete device are not applicable here, kernel time equals application time (see Section 7.1.2).

When run on the GPU, the kernel time scales linear with the input image size for images with comparable color value distributions. Furthermore, the kernel time is impacted by the actual image data, severely on cards that come without hardware support for atomics. Moreover, transfer time of the input data has to be taken into account, as well as the transfer time of the output data but this is negligible in comparison.

In general, operating on GPUs comes with a high startup cost, but afterwards, calculations are done much faster than on CPUs. As such, only from a certain input image size, it becomes beneficial doing processing on a GPU. As a rule of thumb, input images with size less than 2 megapixels are best processed on the CPU. For larger images, the image size divided by the average GPU performance plus the time needed for transfers has to be smaller than the image size divided by the CPU performance.

Chapter 8

Conclusions

Using the relatively simple histogram procedure, this work studies performance determining factors in computing in parallel on SIMD and SIMT devices. Modern GPUs support SIMT, multiple threads running the same instruction, whereas CPUs use SIMD, in which one instruction manipulates multiple memory locations at once.

Image Size

GPUs allow for a much greater scale of parallelization than CPUs. While GPUs offer a lot of power, getting high performance even for simple algorithms is non-trivial. Their application is limited due to certain fixed costs (e.g., transfer of data to the device, kernel invocation) requiring a minimal problem size to be able to offer an performance increase through increased parallelization.

Image Content

In processing typical real-life images, contention for the same memory address is not occurring at such a scale that performance of atomic operations becomes a factor of importance. For specific cases, however, they can dominate performance, in which hardware-assisted atomic operations becomes a very useful feature, as compared to relying on software mutexing. For the histogram problem, the best way of dealing with the lack of shared-level hardware atomics is to keep the number of threads per work-group sufficiently small in order to keep the maximal impact of contention manageable.

Hardware-related Factors on Performance

The way in which per work-group shared memory is implemented on a device has a great impact on performance. At the slow end of the spectrum, it is physically part of the global memory, whereas on the fast side, it is implemented as per-CU cache. Furthermore, the amount of parallelization that a device offers through

the number of PEs per CU, the number of CUs available, and the (possibly over-allocated) number of threads that can be scheduled are determining factors. The availability of hardware atomics and their actual implementation specifics can also greatly impact performance in high-contention corner cases.

Performance Prediction

Performance on different types of GPUs is difficult to predict. Optimal layout parameters for the kernel invocation are highly device and problem-instance specific. As such, choosing an high-performing runtime parameters for a broad range of problems on a large set of platforms is not feasible.

Images smaller than two megapixels should be processed on the CPU. After determining average GPU performance and the time needed for data transfers, as is done in this study for a number of devices, it can be determined whether it is beneficial to perform processing for a certain image size on the GPU or CPU.

Glossary

C++ . vi, 2, 4, 6, 9, 15, 16, 18–22, 25, 33, 34, 38, 41, 44–46

CPU central processing unit. 1, 2, 4–7, 9, 10, 16, 20–22, 33, 37, 38, 40, 41, 44–48

CU OpenCL Compute Unit. 7, 9, 28–32, 40, 44–48

CUDA Compute Unified Device Architecture, a parallel computation language and framework by Nvidia. v, vi, 2, 4, 6, 7, 9, 10, 15–22, 24, 35, 37, 38, 44, 45, 53

CUDA C . 16, 18, 19, 22, 33, 35

FPGA field-programmable gate array. 9

GPGPU General-Purpose computing on Graphics Processing Units. 6–10

GPU graphics processing unit. 1–4, 6–10, 16, 19–22, 28, 33, 35–40, 45–48

OpenCL Open Compute Language, a parallel computation language and framework by the Khronos group consortium. v, vi, 2, 4, 9, 10, 15–22, 24, 27, 35, 37, 38, 41, 44–46, 53

OpenCL C . 16, 18, 19, 22, 33–35

OpenGL . 22

PE OpenCL Processing Element. 9, 27–30, 32, 44–46, 48

posterized Posterization is the process of decreasing the number of distinct colors in an image. Originally used as an artistic effect to create posters, it also occurs as a side effect of lossy image compression. 31

ptxas Nvidia optimizing assembler for PTX. 9

Python . 33, 34

SIMD Single Instruction, Multiple Data. v, 1, 5–7, 9, 19, 46, 47

SIMT Single Instruction, Multiple Threads. v, 1, 5–7, 9, 19, 31, 44, 47

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] Juan Alemany and Edward W Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 1992.
Cited on page 4.

[2] Markus Ålind, Mattias V Eriksson, and Christoph W Kessler. Blocklib: a skeleton library for cell broadband engine. In *Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14. ACM, 2008.
Cited on page 3.

[3] Michel Auguin and G Giraudon. Image processing on a simd/spmd architecture: Opsila. In *Pattern Recognition, 1988., 9th International Conference on*, pages 430–433. IEEE, 1988.
Cited on page 3.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.
Cited on page 6.

[5] Thomas Chen, Ram Raghavan, Jason N Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementationa performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.
Cited on page 3.

[6] Intel Cooperation. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, Feb. 2014.
Cited on pages 5, 6, 19, 22, and 37.

[7] Intel Cooperation. Intel SPMD Program Compiler. https://ispc.github.io/index.html, 2015.
Cited on pages 6 and 19.

[8] NVIDIA Corporation. NVIDIA GeForce 8800 GPU Architecture Overview. http://www.nvidia.com/object/IO_37100.html, Nov. 2006.
Cited on pages 6 and 7.

[9] NVIDIA Corporation. NVIDIA's CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, Sept. 2009.
Cited on pages 5, 7, 19, and 34.

[10] NVIDIA Corporation. NVIDIA's CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, Mar. 2012.
Cited on pages 7 and 34.

[11] NVIDIA Corporation. NVIDIA's CUDA Compute Architecture: Maxwell GM107 GM204. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, Mar. 2014.
Cited on pages 7 and 34.

[12] NVIDIA Corporation. Programming Guide :: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Mar. 2015.
Cited on pages 9 and 22.

[13] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.
Cited on page 10.

[14] Nick England. Advanced architecture for graphics and image processing. *System*, 1024:24, 1982.
Cited on page 6.

[15] Johan Enmyren and Christoph W Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.
Cited on page 3.

[16] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.
Cited on page 10.

[17] Oliver Fluck, Shmuel Aharon, Daniel Cremers, and Mikael Rousson. Gpu histogram computation. In *ACM SIGGRAPH 2006 Research posters*, page 53. ACM, 2006.
Cited on page 3.

[18] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. *Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories*, volume 23. ACM, 1989.
Cited on page 6.

[19] Edgard Garcia. Implementing a histogram for image processing applications. *Xcell Journal*, 38:46–47, 2000.
Cited on page 3.

[20] Dominik Göddeke, Robert Strzodka, and Stefan Turek. *Accelerating double precision FEM simulations with GPUs*. Univ. Dortmund, Fachbereich Mathematik, 2005.
Cited on page 22.

[21] Khronos OpenCL Working Group et al. The opencl specification. (version: 1.2):22–23, 2012.
Cited on pages 3 and 4.

[22] Mark Harris. Mapping Computational Concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.
Cited on page 22.

[23] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.
Cited on page 10.

[24] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. GPGPU: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006.
Cited on page 6.

[25] S. Maleki, Yaoqing Gao, M.J. Garzaran, T. Wong, and D.A. Padua. An Evaluation of Vectorizing Compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, Oct 2011.
Cited on page 6.

[26] Alastair JW Mayer. The architecture of the burroughs b5000: 20 years later and still ahead of the times? *ACM SIGARCH Computer Architecture News*, 10(4):3–10, 1982.
Cited on page 4.

[27] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.
Cited on page 6.

[28] Karl Pearson. Contributions to the mathematical theory of evolution. II. Skew variation in homogeneous material. *Philosophical Transactions of the Royal Society of London. A*, pages 343–414, 1895.
Cited on pages 3, 11, and 12.

[29] Michael Potmesil and Eric M Hoffert. The pixel machine: A parallel image computer. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 69–78. ACM, 1989.
Cited on page 6.

[30] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: A many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.
Cited on page 7.

[31] Asadollah Shahbahrami, Jae Young Hur, Ben Juurlink, and Stephan Wong. Fpga implementation of parallel histogram computation. In *2nd HiPEAC Workshop on Reconfigurable Computing, Göteborg, Sweden*, pages 63–72, 2008.
Cited on page 3.

[32] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. Simd vectorization of histogram functions. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 174–179. IEEE, 2007.
Cited on page 3.

[33] Ramtin Shams and RA Kennedy. Efficient histogram algorithms for nvidia cuda compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422. Citeseer, 2007.
Cited on page 3.

[34] Arkady Shapkin. OpenCL Extension Wrangler.
`https://github.com/OpenCLWrangler/clew`, 2015.
Package version 0.1.
Cited on page 22.

[35] Sergey Sharybin. CUDA Extension Wrangler.
https://github.com/CudaWrangler/cuew, 2015.
Package version 0.1.
Cited on page 22.

[36] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 116–125. IEEE, 2012.
Cited on page 4.

[37] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance traps in opencl for cpus. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 38–45. IEEE, 2013.
Cited on page 4.

[38] Leah J Siegel, Howard Jay Siegel, and Philip H Swain. Performance measures for evaluating algorithms for simd machines. *Software Engineering, IEEE Transactions on*, (4):319–331, 1982.
Cited on page 3.

[39] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Towards a portable multi-gpu skeleton library. *Private Communication, June*, 2010.
Cited on page 4.

[40] Herbert A Sturges. The Choice of a Class Interval. *Journal of the American Statistical Association*, 21(153):65–66, 1926.
Cited on page 12.

[41] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
Cited on page 6.

[42] Brecht van Lommel et. al. Cycles Rendering Engine.
http://code.blender.org/index.php/2011/04/modernizing-shading-and-rendering/, 2011.
Cited on page 4.

[43] Nathan Whitehead and Alex Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.
https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf, 2011.
Cited on page 22.