ISSN 1746-7659, England, UK
Journal of Information and Computing Science
Vol. 10, No. 2, 2015, *pp*.124-136

An In-depth Study on the Performance Impact of CUDA, OpenCL, and PTX Code

Puya Memarzia [1], Farshad Khunjush [2]

[1] School of Electrical and Computer Engineering, Shiraz University, pm@memarzia.com

[2] School of Electrical and Computer Engineering, Shiraz University, khunjush@shirazu.ac.ir

(*Received November 30, 2014, accepted February 26, 2015)*

Abstract. In recent years, the rise of GPGPU as a viable solution for high performance computing has been accompanied by fresh challenges for developers. Chief among these challenges is efficiently harnessing the formidable power of the GPU and finding performance bottlenecks. Many factors play a role in a GPU application's performance. This creates the need for studies performance comparisons, and ways to analyze programs from a fundamental level. With that in mind, our goal is to present an in-depth performance comparison of the CUDA and OpenCL platforms, and study how PTX code can affect performance. In order to achieve this goal, we explore the subject from three different angles: kernel execution times, data transfers that occur between the host and device, and the PTX code that is generated by each platform's compiler. We carry out our experiments using ten real-world GPU kernels from the digital image processing domain, a selection of variable input data sizes, and a pair of GPUs based on the Nvidia Fermi and Kepler architectures. We show how PTX statistics and analysis can be used to provide further insight on performance discrepancies and bottlenecks. Our results indicate that, in an unbiased comparison such as this one, the OpenCL and CUDA platforms are essentially similar in terms of performance.

Keywords: GPU; CUDA; OpenCL; PTX; Performance.

1. Introduction

The advent of General-Purpose computation on Graphical Processing Units (GPGPU) has long since opened the doors to utilizing GPUs for a wide variety of applications, and continues to gain momentum. GPUs are generally employed whenever an application requires heavy computational processing power, and has the potential to be split up into a large number of smaller tasks that can be performed in parallel. Software developers and researchers employ a variety of programming libraries and platforms to write GPGPU applications. Chief among these platforms is the Compute Unified Direct Architecture (CUDA) from Nvidia, and the Open Computing Language (OpenCL) from the Khronos Group. CUDA has established itself as the world's prominent GPU computing framework, and has traditionally received significantly higher attention from developers and researchers. Some studies have found CUDA to provide superior performance compared to OpenCL [6]. However, this appears to vary with the type of workload, and testing methodology [4]. In recent years, the OpenCL compiler and development tools have matured notably, and while it is still not a definitively superior programming platform, it is certainly more viable than it was in the past. GPGPU has been adopted on a broad variety of hardware, and the importance of portability (which is one of OpenCL's key features) and heterogeneous systems has risen dramatically. As a result, OpenCL has been increasingly gaining traction.

GPU applications are relatively complex compared to similar applications on the CPU. Many factors can affect a GPU application's performance. These include data transfers, access patterns, and the amount of parallelization. Writing efficient GPGPU applications is a challenging task. Different GPU architectures, programming platforms, algorithms, and memory spaces have the potential to influence an application's performance, significantly. Detailed analyses of these factors can help developers choose better hardware and programming platforms while writing better, and more efficient applications. They can also help with the development of better compilers and various developer tools, such as tuning guides, profilers, simulated GPUs, statistical performance models, modified compilers, and automatic optimization tools.

Nvidia GPUs are capable of executing both CUDA and OpenCL applications. This allows us to

implement a specific algorithm in CUDA, evaluate its performance on a set of data, and then port the code over to OpenCL, and repeat the procedure.

At their cores, the CUDA and OpenCL platforms share many similarities, such as the overall programming model, memory spaces, and a syntax that is very similar to C. One key similarity is the ability to save code to an intermediate language called Parallel Thread Execution (PTX). Since PTX is a low-level representation of the overlying code, it is directly related to performance, and can be useful for analyzing the different ways in which equivalent CUDA and OpenCL kernels are handled by their respective compilers. We combine PTX analysis with regular performance benchmarking. This provides us with a way to study the differences between CUDA and OpenCL from two different angles. PTX analysis can be beneficial to programmers because it can help them locate inefficiencies in the code. It can also be useful for compiler developers as it relates to compiler optimizations and performance.

Our goal is to compare and evaluate the differences between CUDA and OpenCL. We present a performance evaluation of a series of image processing kernels, implemented in both CUDA and OpenCL. The GPU kernels are implemented to be essentially identical.

The main contribution of this paper is to present a comprehensive performance comparison of the CUDA and OpenCL parallel programming platforms. Some of the key features of our work include: 1) we run the experiments on the latest GPU architectures from Nvidia (Fermi GF110 and Kepler GK104). 2) We measure kernel execution as well as data transfer times. 3) We perform statistical PTX analysis. 4) We take steps to ensure a fair comparison. 5) We explore the possibility of a correlation between the number of generated PTX instructions and the relative performance of the two platforms. We believe the sum of these features make our work distinctive from the prior work performed by others.

The reminder of this paper is organized as follows. Section 2 explores related work. A concise background on GPU architectures, GPU computing platforms, the PTX language, and image processing is provided in Section 3. We elaborate on our approach in Section 4, and describe our experimental methodology in Section 5. Our experimental results are presented and discussed in Section 6. We conclude the paper in Section 7, and outline future work in Section 8.

2. Related Work

Studying the performance impacts of GPGPU frameworks is not a novel concept. There have been relatively few comprehensive comparisons of the CUDA and OpenCL platforms. Some studies have focused on the performance differences between CUDA and OpenCL (with varying results), whereas others have attempted to explore the portability aspect of OpenCL. There has also been some work performed using simulated GPUs.

Fang et al. [4] presented an in-depth comparison of the CUDA and OpenCL platforms. They evaluated the performance of 16 benchmarks, consisting of real world and synthetic benchmarks, on the CUDA and OpenCL platforms. The tests were performed on an Nvidia GTX 480, GTX 280, and an ATI Radeon 5870. The authors used statistical PTX analysis to find out why their FFT benchmark exhibited the largest performance gap. Their experimental results suggest that OpenCL can perform on par with CUDA if the tests are performed in a fair manner.

Weber et al. [20] compared the performance and programmability of a Monte Carlo application on several platforms including Nvidia CUDA, and OpenCL. Their results indicate that the OpenCL platform provides portability between CPUs and GPUs, but may cause a drop in performance.

Karimi et al. [6] compared the performance of CUDA and OpenCL on a Monte Carlo simulation using near-identical kernels. They also explained the process involved in converting a CUDA kernel to an OpenCL kernel. Their results show CUDA consistently outperforming OpenCL in both data transfer times and kernel execution times. They used an Nvidia GTX 260 for their experiments.

A study on the performance and portability of OpenCL kernels was performed by Komatsu et al. [7]. The authors evaluate the performance several equivalent CUDA and OpenCL programs, and investigate the reasons behind their performance differences. Their results initially suggest that the CUDA kernels are significantly faster than their OpenCL counterparts are. They then demonstrate that OpenCL programs can perform similarly to CUDA programs, only if they are optimized by hand, or if the OpenCL compiler parameters are manually tuned for each device. The authors performed their experiments using a Radeon 5870, and an Nvidia Tesla C1060.

Du et al. [19] evaluated the OpenCL platform and investigated its performance and the behavior of its compiler, compared to CUDA. Their results show CUDA consistently performing better than OpenCL. The authors conclude that OpenCL is a good platform for performance-portable applications, but recommend auto-tuning as an ideal solution to get better. The experiments were performed on NVIDIA Tesla C2050 and ATI Radeon 5870.

Ker et al. [2] proposed a set of metrics for GPU workloads and used these metrics to analyze the behavior of GPU programs. They developed a full function emulator that implements the NVIDIA virtual machine referred to as PTX. Their results highlighted the importance of various optimizations and located opportunities for additional parallelism.

Bakhoda et al. [1] extended the simulator, GPGPU-Sim to simulate Nvidia's PTX instruction set architecture. The authors used this simulator to run a series of twelve non-trivial CUDA applications on a simulated Nvidia Geforce 8600GTS, analyze the performance of each application, and locate the primary bottlenecks.

An analysis on the primary factors in implementing and evaluating image-processing algorithms was performed by Park et al. [3]. The authors proposed and evaluated a set of metrics aimed at helping programmers predict the characteristics of an algorithm's parallel implementation as well as its appropriateness. The metrics were tested on image processing algorithms from four distinct domains. The authors used NVIDIA G92 and G200 GPU hardware. The paper does not compare CUDA and OpenCL, but its conclusions regarding the implementation of image processing algorithms are relevant to our work.

In summary, some studies have compared the performance of CUDA and OpenCL on a variety of physical GPUs ranging from the G80 architecture to Fermi [4], [6], [7], [19], and [20]. Of the aforementioned studies, some of them also examined PTX statistics [4], or compiler behavior and tuning [7] and [19]. The remainder of the related work examines the use of PTX analysis on simulated GPUs [1, 2], or the process of implementing image processing algorithms on GPUs [3]. In contrast with the related work, our work compares the performance of CUDA and OpenCL, using physical hardware based on the latest Nvidia GPU architectures (GF110 Fermi and GK104 Kepler). One or both of these GPU architectures are not present in the related work, mainly because they were not available to the authors at the time. We measure kernel execution times as well as data transfer times, and combine these results with PTX statistics and analysis. Our experiments are entirely centered on algorithms from the digital image-processing domain, which provide a diverse, data access intensive selection of benchmarks for our comparison.

3. Background

In this section, we provide an overview on the core topics and concepts that are relevant to our work. These topics consist of GPGPU, CUDA, OpenCL, PTX, and Digital image processing.

3.1. General-Purpose Computation on Graphics Processing Units (GPGPU)

Nowadays, GPUs are used for much more than just 2D and 3D acceleration. A GPU can significantly accelerate many general tasks, especially tasks with a high degree of parallelizability. A typical GPGPU application can be summarized as follows: First the CPU (known as the host), having completed all other necessary initialization tasks, performs a data transfer to a GPU (referred to as a device). Then, the GPU uses the many threads that is has at its disposal to run a piece of code that resembles a function (called a GPU kernel) on the data. Finally, the results of the GPU's work are transferred back to the CPU.

There are several current and upcoming GPGPU programming platforms. CUDA and OpenCL are two well-established platforms, for developing, and running GPGPU applications. A brief description of each of these programming platforms is provided in sections 3.3 and 3.4.

On the hardware level, a GPU contains several stream multiprocessors. Each multiprocessor is capable of executing a large number of threads, in parallel. The exact number of threads depends on the hardware. Threads on a GPU are organized into blocks called thread blocks. Each thread block can be independently executed by a multiprocessor, and threads within a block can use shared memory to communicate and share data. A collection of thread blocks are organized into a grid, the size of which depends on the size of the data. This grid forms the basis for the GPU application's execution [16]. In OpenCL, threads are known as work-items and thread blocks are called work-groups.

3.2. CUDA

CUDA is a parallel computing and programming platform, created by Nvidia, and utilized to create applications for Nvidia GPUs. The CUDA platform combined with Nvidia GPUs form a tightly coupled and well-supported ecosystem. CUDA integrates well with existing development environments and provides an extensive set of tools and documentation to assist programmers in the challenging task of creating efficient programs for the GPU. Consequently, CUDA benefits from a higher degree of popularity among GPGPU researchers [23]. One potential downside to using CUDA is that portability is limited to other Nvidia GPUs.

3.3. OpenCL

OpenCL is another parallel computing platform, created by the Khronos Group. OpenCL and CUDA appear to serve similar purposes, but OpenCL has one significant difference: portability. Portability is regarded as OpenCL's most notable feature. As the name suggests, the OpenCL specification is open source, and as a result, has been ported to multiple hardware and software platforms. OpenCL provides a portable API, which enables different hardware platforms such as CPUs, GPUs, and even FPGAs to support OpenCL.

3.4. PTX

The PTX ISA (Parallel Thread Execution) is an intermediate language developed by Nvidia. It is somewhat similar to assembly in both appearance and in function. On a GPU, a PTX program specifies the instructions that must be executed on any given thread. PTX is generated from high-level source code by the CUDA or OpenCL compiler, and then assembled at runtime into the target device code by a PTX assembler (a component of the GPU driver). Consequently, the performance of a PTX program relies heavily on the compiler and driver. For more information regarding the PTX ISA, please refer to [18].

Figure 1 illustrates an overview of the PTX generation and assembling process. By default, the intermediate PTX code is deleted after it has served its purpose. However, both the CUDA and OpenCL platforms provide options to dump a program's PTX code. The PTX code can then be analyzed to better understand a program's behavior, and potentially locate some of its bottlenecks. It can also provide useful insights into the CUDA and OpenCL compilers, such as different approaches to optimization.

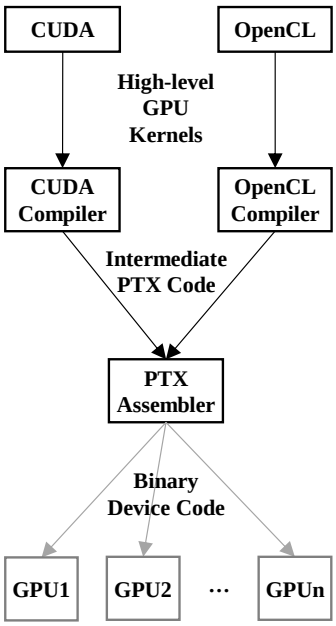


Fig.1: Overview of the PTX code process

PTX code analysis can be performed in a variety of ways. One such method is statistical analysis, whereupon we count the occurrences of specific instructions, for a given kernel within the PTX file. Identical CUDA and OpenCL programs will produce dissimilar PTX code. By performing this analysis on the PTX code, we can assess the behaviors of the OpenCL and CUDA compilers.

3.5. Digital Image Processing

Digital image processing refers to the use of computer algorithms to perform image processing on digital images. A digital image can be regarded as a matrix of colored dots called pixels. A typical serial image-processing algorithm operates by looping through these pixels and performing computation, accordingly. Each output pixel will often rely on calculations performed on multiple input pixels. Consequently, performance is sensitive to memory performance and compiler optimizations.
Digital image processing is ideal for parallel programming because the workloads typically involve massive grids of pixels that can be mapped to the large number of processing cores present on a GPU with relative ease.

4. Our Approach

In order to represent a fair comparison, and obtain a baseline comparison for each of these platforms, we avoid performing manual, framework-specific optimizations on the kernels. Consequently, each frameworks respective compiler handles the burden of optimization. In terms of high-level code, the kernels are essentially identical. Measuring the kernel execution and data transfer times gives us a picture of how CUDA and OpenCL differ in terms of performance, but it does not give us the reasons behind this. To shed more light on the matter, we also investigate the PTX code that is generated by each of the compilers. We do this by separating the kernels, and then counting the PTX instructions in each kernel. The results of these statistics combined with the kernel execution results guide us towards directly investigating PTX code itself. This allows us to discover some of the finer details, such as how loops and data movement are optimized, and observe the different subsets of instructions that are being used.

5. Experimental Setup and Methodology

In this section, we elaborate on our setup, testing methodology, metrics, and selected benchmarks.

5.1. Experimental Setup

All experiments are performed on a system equipped with an Intel 2500k processor running at 3.2GHz, 4GBs of RAM, an NVIDIA GTX 570 video card, and an NVIDIA GTX 670 video card. Table 1 shows the exact hardware and software configuration of the system used for our experiments, and the specifications of the GPUs are displayed in Table 2. Due to the removal of all OpenCL related material from version 5.5 of the Nvidia CUDA SDK, we also use version 4.2 of the Nvidia GPU computing SDK, which contains OpenCL samples, libraries, and documentation.

Table 1: Experimental Setup Specifications

CPU	Intel Core i5 2500k @ 3.2GHz
RAM	8GB DDR3-1333
Chipset	Intel P67
GPU	Nvidia GTX 570
	Nvidia GTX 670
Video Driver	Nvidia Geforce 331.58
Operating System	Windows 7 64-bit
Frameworks	NVIDIA CUDA 5.5 / 4.2
	OpenCL 1.1

Table 2: GPU Specifications

Model	Nvidia GTX 570	Nvidia GTX 670
Architecture	Fermi (GF110)	Kepler (GK104)
Stream Processors	480	1344
Memory (GB)	1.25GB	2GB
Core Clock (MHz)	750	915
Memory Clock (MHz)	975 (3900)	1502 (6008)
Shader Clock (MHz)	1500	N/A

5.2. Selected Benchmarks

The benchmarks used in this paper are based on a selection of image processing kernels that we implemented in CUDA and OpenCL. The image processing kernels perform a diverse variety of digital image processing operations that are commonly used in image manipulation software. Great care is taken to ensure that the kernels for each platform are as close to identical as possible. The benchmarks differ in terms of memory usage and access patterns, computational intensity, and data reuse. Table 3 lists the kernels and

specifies the parameters used for each of them.

Table 3: Kernel Parameters

Kernel Name	Parameters
Bloom	Radius1 = 1, Radius2 = 3
Blur	Radius = 1
Dilation	Radius = 3
Dithering	BayerSize = 8
Erosion	Radius = 3
Median	Radius = 1
Oil Painting	Nbins = 20, Radius = 2
Ripples	N/A (coordinates based function)
Sharpen	Radius = 1, Strength = 2.3
Swirl	Swirlfactor = 0.02

The selected benchmarks and their descriptions are as follows

- *Bloom Effect*: The Bloom effect creates the illusion of light sources glowing and bleeding light into the surrounding area. This effect frequently occurs when using real-world cameras. We reproduce Bloom by applying multiple convolution filters.
- *Blur filter*: The blur filter is a fundamental building block in many image-processing algorithms. There are various ways to implement this filter. Our implementation uses a common 2D box filter.
- *Dilation*: Morphological dilation causes the lighter regions in the image to grow larger. To differentiate this kernel from the Erosion kernel, we implement it for RGB image.
- *Dithering*: Dithering is commonly used to eliminate color banding and for color reduction. Ordered Dithering uses a threshold matrix of an arbitrary size called a Bayer matrix to determine the pixel's final color. We use an 8x8 Bayer matrix for our benchmarks.
- *Erosion*: Morphological erosion causes the darker regions in the image to grow larger. This algorithm can be implemented to output binary, greyscale, or RGB images. We opt for the greyscale variant.
- *Median Filter*: Median Filters are often used to eliminate noise from an image, or as a building block in other, larger image processing algorithms. Each pixel is replaced by the median of its neighboring pixels.
- *Oil Painting*: The oil painting algorithm receives an image and outputs a rendition that looks like an oil painting. A common implementation of this algorithm counts the colors in a radius surrounding each pixel. It then finds the maximum repeated color and uses writes that to the output image.
- *Ripples*: The ripples effect uses a simple, fixed trigonometric function to generate ripples based on the coordinates of each pixel. These ripples are then merged with the input image to generate the output image.
- *Sharpening Filter*: The sharpening filter enhances an image by making details stand out. One commonly used implementation method sharpens an image by first applying a blur filter to each pixel, and then subtracting the resulting values from the original image. A strength factor controls the intensity of the effect.
- *Swirl*: Swirl is type a displacement filter. It creates a swirling effect by taking each pixel and moving it to a different position based on its relative position to the center of the image. The final result is similar to the effect that one would get from twisting a piece of cloth.

5.3. Input Data

We use a series of bitmap images with a diverse variety of resolutions, for our experiments. Table 4 depicts the specifications of these images. The exact same input images are used for the CUDA and OpenCL experiments. Unless stated otherwise, our selection of image processing algorithms do not depend on the actual contents of these images. Consequently, the varying image sizes are the main point of interest. These are illustrated again in our data transfer experiments.

Table 4: Specifications of test images

Row	Image Resolution	Megapixels	File Size (KB)
1	640 x 480	0.3	901
2	1024 x 768	0.8	2305
3	1920 x 1080	2	6076
4	2560 x 1440	4	10801
5	3735 x 3648	14	39926
6	4896 x 4188	21	60072
7	6587 x 8336	55	160892

5.4. Performance Evaluation Metrics

In this section, we outline the methods we use to measure performance and the metrics we use to present those measurements. In this context, a test configuration refers to a combination of one of programming platforms and one of the GPUs. Since we are testing two of each, this gives a total of four test configurations.

For our kernel execution experiments, we measure the execution time (t) for each platform, in seconds. In order to give us a better idea of how the two platforms compare we each other, we calculate their Relative Performance (RP). This is defined as follows.

$$RP = \frac{t_{CUDA}}{t_{OpenCL}} \tag{1}$$

On each test configuration, we measure the kernel execution time for each image. The test is looped 100 times. We then use the average of those results to calculate the Relative Performance for that particular kernel. This results in one set of results for each tested GPU. This metric is then used to calculate another metric called RP_{Total}, which is defined as follows.

$$RP_{Total} = \frac{\sum RP_i}{n} \tag{1}$$

In equation (2), n denotes the total number of benchmarks and RP_i is the relative performance for each individual benchmark. Our data transfer experiments are performed by measuring the transfer times in seconds. We separately measure the time it takes to transfer the input image to the GPU (host to device), and the time it takes to transfer the result image back (device to host). We only measure the data transfer itself, excluding any other operations, such as buffer allocation. Each test is repeated 100 times. We then calculate the average times for each of the four configurations.

5.5. Selected PTX Instructions

The PTX instructions selected for our statistics are inspired by the FFT PTX statistics presented in [4]. We only omit the instructions that relate to shared memory and synchronization, because our naïve kernels do not utilize them. Table 5 lists and categorizes these instructions and provides a short description on the function of each instruction. Please note that some instructions may have several sub-variants. As an example, a subset of data movement instructions supports vectors as operands, as opposed to scalar operands. For a more comprehensive representation of these instructions, refer to Chapter 8.7 of the PTX ISA documentation [18].

Table 5: List of PTX instructions

Category	Instruction	Description
Arithmetic	add	add two values
	sub	subtract one value from another
	mul	multiply two values
	div	divide one value by another
	mad	multiply-add (a*b+c)
	fma	fused multiply-add
	neg	arithmetic negation
Logical	and	bitwise and

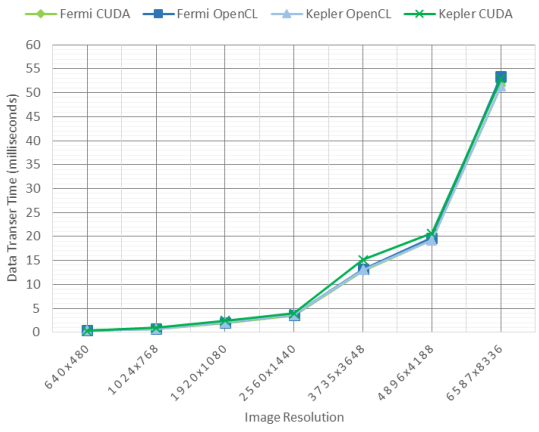
	or	bitwise or
	not	one's compliment
	xor	bitwise xor
	shl	shift bits left, zero fill
	shr	shift bits right, zero fill
Data Movement	cvt	convert a value from one type to another.
	mov	copy data between registers
	ld.param	kernel parameter -> register
	ld.local	local memory -> register
	ld.const	constant memory -> register
	ld.global	global memory -> register
	st.local	register -> local memory
	st.global	register -> global memory
Flow Control	setp	compare two numeric values with a relational operator
	selp	select operand, based on the value of the predicate operand.
	bra	branch to target

6. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we present the results of our experiments, discuss the significance of them, and speculate on the factors that may have had an impact on the results. The results consist of a series of data transfer experiments, kernel execution experiments, and an analysis of the generated PTX code.

6.1. Data Transfer Results

In this section, we present and discuss the results of our data transfer experiments. Figure 2 depicts the data transfer times measured for each of the configurations, for transfers that occur from the host to the device. Figure 3 provides this information for device to the host data transfers. The numbers state the average time each test configuration takes to transfer all of the images from Table 3. These results are displayed on a per image resolution basis.



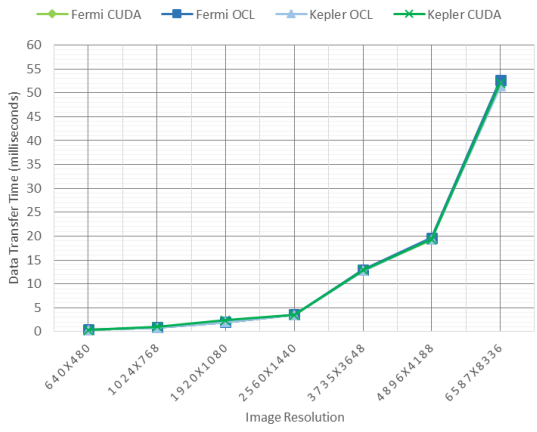


Fig. 1: Host to device data transfer results Fig. 2: Device to host data transfer results

Based on these results, we can draw several conclusions. Firstly, it is apparent that the performance gap between the various test configurations is extremely narrow. There are no noticeable trends in the data to suggest that any of the test configurations are behaving differently to the rest. We notice a few minor discrepancies, but they can be attributed to measurement error, and are negligible in size. The transfer times remain relatively consistent when switching between host to device and device to host data transfers, and this holds true for all of the testing configurations. Overall, we can conclude that, for regular data transfers, all the test configurations in this experiment behave in a similar and consistent manner.

6.2. Kernel Execution Results

Figures 4 and 5 depict the Relative Performance for each of the kernel execution experiments. The

Relative Performance metric is such that numbers greater than one indicate better performance on OpenCL and numbers below one represent superior performance on CUDA. It is worth reiterating that our intention is not to compare the GTX 570 and 670 GPUs, but rather, to compare the CUDA and OpenCL platforms, on each of these GPUs.

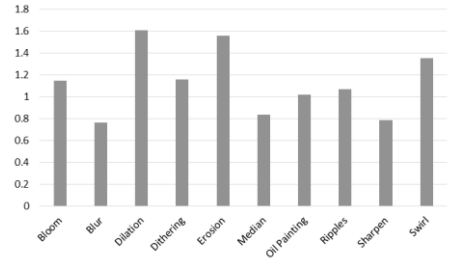


Fig. 4: Relative Performance (RP) – GTX 570

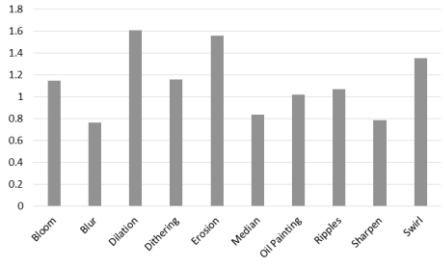


Fig. 5: Relative Performance (RP) – GTX 670

From looking at these results, it becomes immediately apparent that there some kernels seem to favor OpenCL, while others perform better on CUDA.

On the GTX 570 (Fermi) GPU, CUDA performs noticeably better in five of the 10 kernels: Sharpen, Bloom, Blur, Oil Painting, and Ripples. CUDA's Performance is worse in four of the kernels: Dithering, Swirl, Dilation, and Erosion. Performance is on par with OpenCL in just one of the kernels: Median.

The results of the GTX 670 (Kepler) GPU largely mirror those of the GTX 570. However, a few of the kernels change positions. The Median kernel goes from performing similarly on both platforms to favoring CUDA. On the other side, the Oil Painting and Ripples kernels perform equally well on both platforms, instead of performing better on CUDA. The Bloom kernel switches from favoring CUDA to favoring OpenCL. The reasons for these changes would require a comparative analysis of the underlying GPU architectures and device code, which we intend to pursue in our future work.

To get a better idea of the big picture when it comes to overall performance, we calculate the RP metric again, this time using the sum of all the kernel execution times to calculate RP_{Total} (as shown in section 5.4, equation 2). This provides a summary of the relative performance between the CUDA and OpenCL platforms. The results are displayed in Table 6.

Table 6: Relative Performance - Summarized

RP_{Total}	
GTX 570	***GTX 670***
0.950	1.070

The results indicate that, on average, CUDA performs better than OpenCL perform by 5% on the GTX 570. On the GTX 670, performance is narrowly in favor of OpenCL, by approximately 7%. Overall, the results are reasonably close. The fact that OpenCL is able to perform so close to CUDA may come as a surprise, considering how often CUDA is cited as a superior solution by professionals. However, it is worth noting that the OpenCL backend is also a product of Nvidia.

Let us consider a performance difference of less than 10% to indicate identical performance. Therefore, the summarized performance of the CUDA and OpenCL platforms is identical on both GPUs. Any remaining differences are likely to be a result of other factor's such as each platform's respective PTX assembler. We conclude that this comparison shows both platforms to be quite similar in terms of performance, in a fair comparison using basic image processing kernels. This can have significant implications in situations where portability is preferable.

6.3. PTX Statistics

As discussed in Section 3, exploring PTX code can help discover certain factors that can affect an application's performance. However, this is not the whole story, because the PTX code may receive further optimizations when it is compiled to device code. Examining device code optimizations is something that we are considering for future work. A summary of the PTX instruction statistics for each of the GPU kernels can be seen in Table 7, and Table 8. The statistics apply to both of our tested GPUs, because both are compiled using the exact same parameters. This is possible because PTX code can be assembled to work with multiple

DOI https://doi.org/2024-JICS-22555 | Generated on 2025-04-09 07:03:19

different GPUs (see Figure 1 in section 3.4). Overall, it becomes apparent from the results that the CUDA and OpenCL compilers produce roughly the same total number of PTX instructions from these kernels.

Table 7: CUDA PTX Statistics

	Instruction Count									
Instruction Group	***Bloom***	***Blur***	***Dilation***	***Dither***	***Erosion***	***Median***	***Oil Painting***	***Ripples***	***Sharpen***	***Swirl***
Arithmetic	57	27	18	19	24	66	44	22	38	36
Logical	11	8	8	18	8	31	11	4	8	10
Flow Control	35	15	19	23	16	34	19	5	21	34
Data Movement	59	32	40	11	27	148	126	31	37	52
Total	162	82	85	71	75	279	200	62	104	132

Table 8: OpenCL PTX Statistics

	Instruction Count									
Instruction Group	***Bloom***	***Blur***	***Dilation***	***Dither***	***Erosion***	***Median***	***Oil Painting***	***Ripples***	***Sharpen***	***Swirl***
Arithmetic	46	22	21	26	20	65	44	24	33	40
Logical	10	5	6	8	6	11	10	1	5	8
Flow Control	42	19	25	10	21	38	23	4	25	33
Data Movement	56	37	32	24	32	135	130	31	46	42
Total	154	83	84	68	79	249	207	60	109	123

6.4. Relationship between PTX and Performance

In this section, we combine the results presented in the PTX Analysis and Kernel Execution sections, to explore the possibility of a correlation between a GPU kernel's number of PTX instructions and its relative performance. As mentioned in the previous sections, the instructions measured for the PTX statistics cover a subset of the total PTX specifications. However, we confirmed that this selection of instructions adequately covers the instructions used by our kernels. As a result, the total sum of these instructions can be considered a viable representation of the amount of PTX code that is generated for a particular kernel. In order to compare this info with the Relative Performance metric, we subtract the sum of OpenCL instructions from the sum of CUDA instructions to calculate a PTX difference (denoted as PTX Diff.). We then use a combo chart in order to obtain a side-by-side comparison of the PTX difference and the relative performance for each kernel. The resulting charts are depicted in Figure 6 and Figure 7. Positive PTX difference values indicate that there are a greater number of PTX instructions in the CUDA PTX output than OpenCL, and vice versa. As discussed in the previous sections, the relative performance value shows the performance of the OpenCL implementation in relation to the CUDA implementation, so a value of greater than one means that a particular kernel is performing better in OpenCL, and a value of less than one indicates that it is performing better in CUDA.

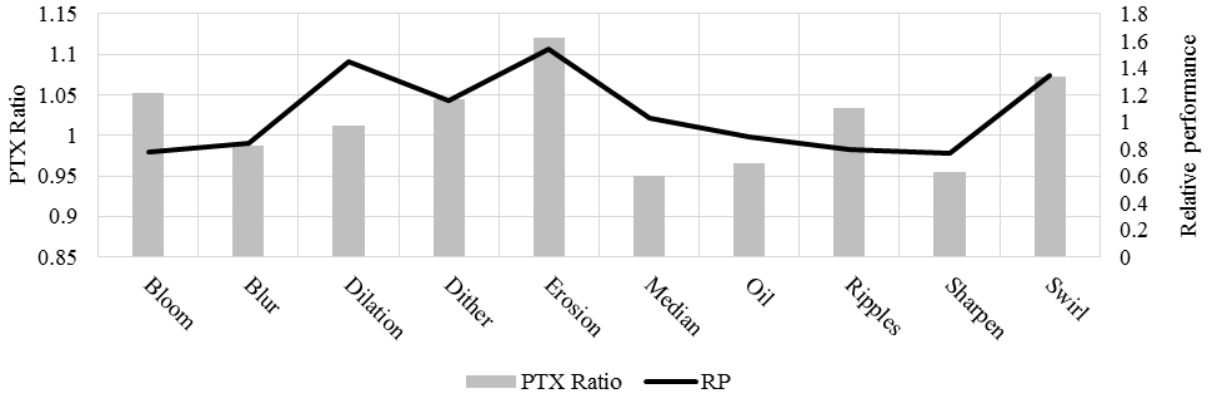


Fig. 6: PTX Ratio and Relative Performance – GTX 570

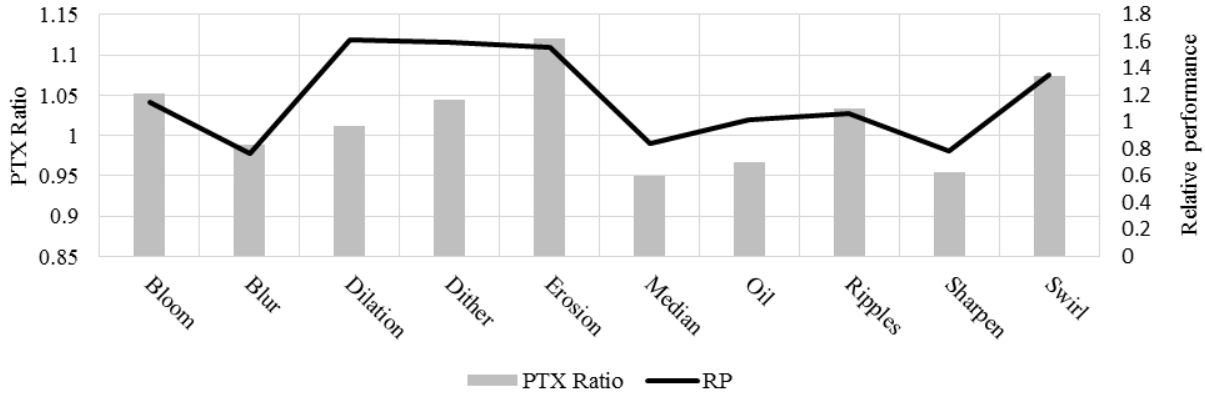


Fig. 7: PTX Ratio and Relative Performance – GTX 670

The results seem to indicate that there is some correlation between the PTX ratio metric and relative performance. Some kernels exhibit a stronger correlation than others do. For example, the Erosion kernel displays a high PTX ratio, and a high corresponding RP value. However, the same cannot be said for some of the other kernels. This indicates that PTX analyses and comparisons are by themselves insufficient to estimate a kernel's relative performance, accurately. We can speculate that there are at least three different reasons for this. Firstly, the PTX assembler performs further optimizations on the code and that has the significantly affect performance. Secondly, different types of compiler optimizations may increase or decrease the number of instructions. As an example, consider loop unrolling. Loop unrolling can increase performance, but also increases the amount of intermediate code generated by the compiler. In contrast, loop fusion would reduce the number of instructions, while also having the potential to improve performance. Thirdly, this model only counts PTX instructions. It does not take into account the fact that different PTX instructions have different execution costs. Considering the above issues, if any solutions are found to resolve or alleviate them, the use of PTX statistics in performance analysis has potential.

6.5. Using PTX Code to Identify Performance Bottlenecks

Sometimes analyzing PTX code can reveal a severe flaw in a program. This can be attributed to either the programmer, or the way the compiler handles code. Prior to being fixed for the framework comparison in section 6.2, the dithering kernel was one such example. It contained notably more PTX instructions in the CUDA version, compared to the OpenCL version.

The dithering kernel had exhibited a significant performance advantage in OpenCL in the kernel execution experiments (up to x20). Given the fact that this kernel is a naïve implementation, it is reasonable to assume that this difference is caused by the behavior of the compiler. Directly investigating the PTX code confirms this, by revealing that the CUDA compiler places the Bayer matrix in local memory (CUDA terminology for what OpenCL refers to as private memory), while OpenCL places it in constant memory. As a reminder, the Bayer matrix is a series of thresholds that each thread in order to calculate the value of the final pixel. It is never modified by any of the threads, so placing it in the constant memory makes more sense. Table 9 shows the statistics for the data movement instructions in the Dither kernel.

Table 9. Dithering Kernel PTX Statistics – Data Movement Instructions

Instruction	CUDA	OpenCL
Cvt	2	6
Mov	88	9
ld.param	5	5
ld.local	1	0
ld.const	0	1
ld.global	1	2
st.local	64	0
st.global	2	1

There are 64 st.local instructions in the CUDA PTX, which is a relatively clear hint that the 8x8 Bayer matrix is being stored in local memory. Local memory is slower than constant memory because it is allocated separately for each thread, resulting in higher memory usage and overhead. We confirm that this is

the primary cause of CUDA's slower performance, by explicitly declaring the Bayer matrix as a global constant, using the __constant__ variable type qualifier, and running the experiment again. This causes the st.local instructions to disappear from the PTX code, and results in a noticeable performance boost. The results for the Dithering kernel, after performing this modification, are shown in Table 10.

Table 10. Relative Performance of CUDA Dithering Kernel – Before and After Modifications

	RP	
	GTX 570	**GTX 670**
Before (Local Bayer Matrix)	20.252	16.484
After (Constant Bayer Matrix)	1.591	1.158

We observe that the applied modification results in a significant reduction in the relative performance metric, bringing CUDA's dither kernel performance much closer to the OpenCL version. Perhaps future versions of the CUDA compiler will be able to resolve this issue automatically, by detecting non-modified variables and switching to a more suitable memory space.

7. Conclusion

In this paper, we presented a comparison of the CUDA and OpenCL parallel programming platforms. We explained the difficulty of efficiently utilizing the GPU for general-purpose computation, and the potential benefits of studies that explore and analyze the performance of these platforms. We described the differences and similarities between CUDA and OpenCL. We then evaluated the performance of these two platforms on 10 basic image-processing kernels, using a series of images as input data.

Our experiments consisted of three segments: data transfers, kernel execution, and PTX statistics and analysis. Our data transfer results showed no notable difference between the two platforms. Our kernel execution results contained wins for both sides, with overall performance being slightly skewed in OpenCL's favor due to anomalous performance from the CUDA dithering kernel. We used PTX analysis to focus on this kernel and showed an example of how PTX analysis can be used to help track down and eliminate a performance bottleneck. After fixing the dithering kernel, we recalculated the relative performance metric and found the CUDA and OpenCL platforms to perform within 10% of each other. We considered this to indicate that both platforms perform similarly. Lastly, we combined the PTX statistics with the relative performance results in order to find out if there is any correlation between the two. We did not find any correlation between these two metrics. We speculated that different types of compiler optimizations and post-PTX code optimizations might have affected the results, eliminating any tangible relationships

8. FUTURE WORK

Our future work will involve expanding the experiments to involve other GPU architectures, such as the recently released Maxwell architecture. We also plan to port the code to other hardware platforms such as multicore CPUs. The data transfer experiments could be expanded to include other data transfer methods and optimizations, such as concurrent streams and pinned memory. We would also like to explore manual optimizations on PTX code, and possibly even go beyond that and look at the device code that is generated from the PTX code. Another possibility is to use GPG-PU simulators, such as GPGPU-sim to analyze PTX code generated using the CUDA and OpenCL compilers.

9. References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pages 163 –174, 2009.

[2] A. Kerr, G. Diamos, and S. Yalamanchili. A Characterization and Analysis of PTX Kernels. In Proceedings of 2009 IEEE International Symposium on Workload Characterization (IISWC). 2009. pp. 3–12.

[3] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim. Design and Performance Evaluation of Image Processing Algorithms on GPUs. IEEE Transactions on Parallel and Distributed Systems. pp. 91-104. 2011.

[4] J. Fang, A. L. Varbanescu and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. International Conference on Parallel Processing (ICPP), pp.216-225, September 2011.

[5] J. Stam. Maximizing GPU Efficiency in Extreme Throughput Applications. In proceedings of the GPU Technology Conference 2009. 2009.

[6] K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. arXiv preprint, arXiv:1005.2581V2 (2010).

[7] K. Komatsu , K. Sato , Y. Arai , K. Koyama , H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In Proceeding of the Fifth international Workshop on Automatic Performance Tuning (iWAPT2010), USA, June 2010.

[8] Khronos Group. The OpenCL Specification Version 1.0. A. Munshi, ed. Khronos Group, 2009.

[9] M. J. Harvey, and G. De Fabritiis. Swan: A Tool for Porting CUDA Programs to OpenCL. Computer Physics Communications 182, no. 4 (2011): 1093-1099.

[10] M. J. Misic, D. M. Durdevic, and M. V. Tomasevic. Evolution and trends in GPU computing. In MIPRO, 2012 Proceedings of the 35th International Convention, pp. 289-294. IEEE, 2012.

[11] NVIDA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[12] NVIDA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[13] NVIDIA Corporation. OpenCL Quickstart Guide.

[14] NVIDIA Corporation. Tuning CUDA Applications for Fermi, Version 1.5, 2011.

[15] NVIDIA Corporation. Tuning CUDA Applications for Kepler, Version 1.1, 2013.

[16] NVIDIA CUDA C Best Practices Guide from CUDA Toolkit 5.0.

[17] NVIDIA CUDA Programming Guide, Version 5, NVIDIA Corporation, Santa Clara, CA, 2012

[18] NVIDIA Inc., Parallel Thread Execution ISA Version 3.1, November 2012.

[19] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. Parallel Computing 38, no. 8 (2012): 391-407.

[20] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A case study. Parallel and Distributed Systems, IEEE Transactions on 22, no. 1 (2011): 58-68.

[21] S. Gupta, P. Xiang, and H. Zhou. Analyzing Locality of Memory References in GPU Architectures. ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '13). 2013.

[22] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. An Experimental Study on Performance Portability of OpenCL kernels. In 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10). 2010.

[23] Statistics, N.D. Retrieved December 5th 2013, from High Performance Computing on Graphics Processing Units (HGPU): http://hgpu.org/?page_id=3529

[24] The Khronos OpenCL Working Group. "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems." http://www.khronos.org/opencl/, December 2012