A comprehensive performance comparison of CUDA and OpenCL

Conference Paper · September 2011

DOI: 10.1109/ICPP.2011.45 · Source: DBLP

3 authors:

Jianbin Fang
National University of Defense Technology
74 PUBLICATIONS **1,179** CITATIONS

SEE PROFILE

Ana Lucia Varbanescu
University of Twente
120 PUBLICATIONS **1,458** CITATIONS

SEE PROFILE

Henk J. Sips
Delft University of Technology
206 PUBLICATIONS **4,708** CITATIONS

SEE PROFILE

A Comprehensive Performance Comparison of CUDA and OpenCL

Jianbin Fang, Ana Lucia Varbanescu and Henk Sips
Parallel and Distributed Systems Group
Delft University of Technology
Delft, the Netherlands
Email: {j.fang, a.l.varbanescu, h.j.sips}@tudelft.nl

***Abstract*—This paper presents a comprehensive performance comparison between CUDA and OpenCL. We have selected 16 benchmarks ranging from synthetic applications to real-world ones. We make an extensive analysis of the performance gaps taking into account programming models, optimization strategies, architectural details, and underlying compilers. Our results show that, for most applications, CUDA performs at most 30% better than OpenCL. We also show that this difference is due to unfair comparisons: in fact, OpenCL can achieve similar performance to CUDA under a fair comparison. Therefore, we define a fair comparison of the two types of applications, providing guidelines for more potential analyses. We also investigate OpenCL's portability by running the benchmarks on other prevailing platforms with minor modifications. Overall, we conclude that OpenCL's portability does not fundamentally affect its performance, and OpenCL can be a good alternative to CUDA.**

***Index Terms*—Performance Comparison, CUDA, OpenCL.**

I. Introduction

In recent years, more and more multi-core/many-core processors are superseding sequential ones. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth, and this trend is likely to continue [1]. Particularly, today's GPUs (Graphic Processing Units), greatly outperforming CPUs in arithmetic throughput and memory bandwidth, can use hundreds of parallel processor cores to execute tens of thousands of parallel threads [2]. Researchers and developers are becoming increasingly interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for "General-Purpose computing on the GPU") [3], to rapidly solve large problems with substantial inherent parallelism.

Due to this large performance potential, GPU programming models have evolved from high-level shading languages such as Cg [4], HLSL [5], and GLSL [6] to modern programming languages, alleviating programmers' burden and thus enabling GPUs to gain more popularity. Particularly, the release of CUDA (Compute Unified Device Architecture) by NVIDIA in 2006 has eliminated the need of using the graphics APIs for computing applications, pushing GPU computing to more extensive use [7]. Likewise, APP (Advanced Parallel Processing) is a programming framework which enables ATI's GPUs, working together with the CPUs, to accelerate many applications beyond just graphics [8]. All these programming frameworks allow programmers to develop a GPU computing application without mastering graphic terms, and enables them to build large applications easier [9].

However, every programming framework has its unique method for application development. This can be inconvenient, because software development and related services must be rebuilt from scratch every time a new platform hits the market [10]. The software developers were forced to learn new APIs and languages which quickly became out-of-date. Naturally, this caused a rise in demand for a single language capable of handling any architecture. Finally, an open standard was established, now known as "OpenCL" (Open Computing Language). OpenCL, managed by the Khronos Group [11], is a framework that allows parallel programs to be executed across various platforms. As a result, OpenCL can give software developers portable and efficient access to the power of diverse processing platforms. Nevertheless, this also brings up the question of whether the performance is compromised, as it is often the case for this type of common languages and middlewares [10]. If the performance suffers significantly when using OpenCL, its usability becomes debatable (users may not want to sacrifice the performance for portability).

To investigate the performance-vs-portability trade-offs of OpenCL, we make extensive investigations and experiments with diverse applications ranging from synthetic ones to real-world ones, and we observe the performance differences between CUDA and OpenCL. In particular, we give a detailed analysis of the performance differences and then conclude that under a fair comparison, the two programming models are equivalent, i.e., there is no fundamental reason for OpenCL to perform worse than CUDA.

We focus on exploring the performance comparison of CUDA and OpenCL on NVIDIA's GPUs because, in our view, this is the most relevant comparison. First, for alternative hardware platforms it is difficult to find comparable models: on ATI's GPU, OpenCL has become the "native" programming model, so there is nothing to compare against; on the Cell Broadband Engine, OpenCL is still immature and a comparison against the 5-year old IBM SDK would be unfair "by design"; on the general purpose multi-core processors, we did not find a similar model (i.e., a model with similar low level granularity) to compare against. Second, CUDA and OpenCL, which are both gaining more and more attention from both researchers and practitioners, are similar to each other in many

aspects.

A. Similarities of CUDA and OpenCL

CUDA is a parallel computing framework designed only for NVIDIA's GPUs, and OpenCL is a standard designed for diverse platforms including CUDA-enabled GPUs, some ATI-GPUs, multi-core CPUs from Intel and AMD, and other processors such as the Cell Broadband Engine.

OpenCL shares a range of core ideas with CUDA: they have similar platform models, memory models, execution models, and programming models [7] [11]. To a CUDA/OpenCL programmer, the computing system consists of a host (typically a traditional CPU), and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units [12]. There also exists a mapping between CUDA and OpenCL in memory and execution terms, as is presented in Table I. Additionally, their syntax for various keywords and built-in functions are fairly similar to each other. Therefore, it is relatively straightforward to translate CUDA programs to OpenCL programs.

TABLE I
A COMPARISON OF GENERAL TERMS [13]

CUDA terminology	OpenCL terminology
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Thread	Work-item
Thread-block	Work-group

The rest of this paper is organized as follows: Section II presents some related work on performance comparison of parallel programming models on multi-core/many-core processors. Section III illustrates our methodology, the selected benchmarks and the testbeds. Section IV gives an overall performance comparison and identifies the main reasons for the performance differences. Then we define a fair comparison for potential performance comparisons and analyses of CUDA and OpenCL. OpenCL's ability in code-portability is shown in Section V. Section VI concludes this paper.

II. Related Work

There has been a fair amount of work on performance comparison of programming models for multi-core/many-core processors. Rick Weber et al. [14] presented a collection of Quantum Monte Carlo algorithms implemented in CUDA, OpenCL, Brook+, C++, and VHDL. They gave a systematic comparison of several application accelerators on performance, design methodology, platform, and architectures. Their results show that OpenCL provides application portability between multi-core processors and GPUs, but may incur a loss in performance. Rob van Nieuwpoort et al. [15] explained how to implement and optimize signal-processing applications on multi-core CPUs and many-core architectures. They used correlation (a streaming, possibly real-time, and I/O intensive application) as a running example, investigating the aspects of performance, power efficiency, and programmability. This study includes an interesting analysis of OpenCL: the problem of performance portability is not fully solved by OpenCL and thus programmers have to take more architectural details into consideration.

In [16], the authors compared programming features, platform, device portability, and performance of GPU APIs for cloth modeling. Implementations in GLSL, CUDA and OpenCL are given. They conclude that OpenCL and CUDA have more flexible programming options for general computations than GLSL. However, GLSL remains better for interoperability with a graphics API. In [17], a comparison between two GPGPU programming approaches (CUDA and OpenGL) is given using a weighted Jacobi iterative solver for the bidomain equations. The CUDA approach using texture memory is shown to be faster than the OpenGL version. Kamran Karimi et al. [18] compared the performance of CUDA and OpenCL using complex, near-identical kernels. They showed that there are minimal modifications involved when converting a CUDA kernel to an OpenCL kernel. Their performance experiments measure and compare data transfer time to and from the GPU, kernel execution time, and end-to-end application execution time for both CUDA and OpenCL. Only one application or algorithm is used in all the work mentioned above.

Ping Du et al. [19] evaluated many aspects of adopting OpenCL as a performance-portable method for GPGPU application development. The triangular solver (TRSM) and matrix multiplication (GEMM) have been selected for implementation in OpenCL. Their experimental results show that nearly 50% of peak performance could be obtained in GEMM on both NVIDIA Tesla C2050 and ATI Radeon 5870 in OpenCL. Their results also show that good performance can be achieved when architectural specifics are taken into account in the algorithm design. In [20], the authors quantitatively evaluated the performance of CUDA and OpenCL programs developed with almost the same computations. The main reasons leading to these performance differences are investigated for applications including matrix multiplication from the CUDA SDK and CP, MRI-Q, MRI-HD from the Parboil benchmark suite. Their results show that if the kernels are properly optimized, the performance of OpenCL programs is comparable with their CUDA counter-parts. They also showed that the compiler options of the OpenCL C compiler and the execution configuration parameters have to be tuned for each GPU to obtain its best performance. These two papers inspired us to analyze the performance differences by looking into intermediate codes.

Anthony Danalis et al. [21] presented a Scalable HeterOgeneous Computing (SHOC) benchmark suite. Its initial focus was on systems containing GPUs and multi-core processors, and on the new OpenCL programming standard. SHOC is a spectrum of programs that test the performance and stability of these scalable heterogeneous computing systems. At the lowest level, SHOC uses micro-benchmarks to assess architectural features of the system. At higher levels, SHOC uses

TABLE II
SELECTED BENCHMARKS

App.	Suite	Dwarf/Class*	Performance Metric	Description
BFS	Rodinia	Graph Traversal	sec	Graph breadth first search
Sobel	SELF	Dense Linear Algebra	sec	Sobel operator on a gray image in X direction
TranP	SELF	Dense Linear Algebra	GB/sec	Matrix transposition with shared memory
Reduce	SHOC	Reduce*	GB/sec	Calculate a reduction of an array
FFT	SHOC	Spectral Methods	GFlops/sec	Fast Fourier Transform
MD	SHOC	N-Body Methods	GFlops/sec	Molecular dynamics
SPMV	SHOC	Sparse Linear Algebra	GFlops/sec	Multiplication of sparse matrix and vector (CSR)
St2D	SHOC	Structured Grids	sec	A two-dimensional nine point stencil calculation
DXTC	NSDK	Dense Linear Algebra	MPixels/sec	High quality DXT compression
RdxS	NSDK	Sort*	MElements/sec	Radix sort
Scan	NSDK	Scan*	MElements/sec	Get prefix sum of an array
STNW	NSDK	Sort*	MElements/sec	Use comparator networks to sort an array
MxM	NSDK	Dense Linear Algebra	GFlops/sec	Matrix multiplication
FDTD	NSDK	Structured Grids	MPoints/sec	Finite-difference time-domain method

application kernels to determine system-wide performance including many systems features. SHOC includes benchmark implementations in both OpenCL and CUDA in order to provide a comparison of these programming models. Some of the benchmarks used in this work are selected from SHOC.

The majority of previous work has used very few applications to compare existing programming models. In our work, we tackle the problem by observing a large set of diverse applications to show the performance differences of CUDA and OpenCL. We also give a detailed analysis of the performance gap (if any) from all possible aspects. Finally, we discuss an eight-step fair comparison strategy to judge the performance of any applications implemented in both programming models.

III. Methodology and Experimental Setup

In this section, we explain the methodologies we adopt in this paper. The used benchmarks and experimental testbeds are also explained.

A. Unifying Performance Metrics

In order to compare the performance of CUDA and OpenCL, we define a normalized performance metric, called $PerformanceRatio(PR)$, as follows:

$$PR = \frac{Performance_{OpenCL}}{Performance_{CUDA}} \quad (1)$$

For $PR < 1$, the performance of OpenCL is worse than its counter-part; otherwise, OpenCL will give a better or the same performance. In an intuitive way, if $|1 - PR| < 0.1$, we assume CUDA and OpenCL have similar performance.

When it comes to different domains, performance metrics have different meanings. In memory systems, the bandwidth of memories can be seen as an important performance metric. The higher the bandwidth is, the better the performance is. For sorting algorithms, performance may refer to the number of elements a processor finishes sorting in unit time. Floating-point operations per second (Flops/sec) is a typical performance metric in scientific computing. Exceptionally, performance is inversely proportional to the time a benchmark that takes from start to end. Therefore, we have selected specific performance metrics for different benchmarks, as illustrated in Table II.

B. Selected Benchmarks

Benchmarks are selected from the SHOC benchmark suite, NVIDIA's SDK, and the Rodinia benchmark suite [22]. We also use some self-designed applications. These benchmarks fall into two categories: synthetic applications and real-world applications.

1) Synthetic Applications: Synthetic applications are those which provide ideal instructions to make full use of the underlying hardware. We select two synthetic applications from the SHOC benchmark suite: MaxFlops and DeviceMemory, which are used to measure peak performance (floating-point operations and device-memory bandwidth) of GPUs in GFlops/sec and GB/sec. In this paper, peak performance includes theoretical peak performance and achieved peak performance. Theoretical peak performance (or theoretical performance) can be calculated using hardware specifications, while achieved peak performance (or achieved performance) is measured by running synthetic applications on real hardware.

2) Real-world Applications: Such applications include algorithms frequently used in real-world domains. The real-world applications we select are listed in Table II. Among them, Sobel, TranP in both CUDA and OpenCL, and BFS in OpenCL are developed by ourselves (denoted by "SELF"); others are selected from the SHOC benchmarks suite ("SHOC"), NVIDIA's CUDA SDK ("NSDK") and the Rodinia benchmark suite (only BFS in CUDA, denoted by "Rodinia"). Following the guidelines of the 7+ Dwarfs [23], different applications fall into different categories. Their performance metrics and descriptions are also listed in the table.

C. Experimental Testbeds

We obtain all our measurement results on real hardware using three platforms, called Dutijc, Saturn, and Jupiter. Each platform consists of two parts: the host machine (one CPU) and its device part (one or more GPUs). Table III shows the detailed configurations of these three platforms. A short comparison of the three GPUs we have used (NVIDIA GTX280, NVIDIA GTX480, and ATI Radeon HD5870) is presented in Table IV (MIW there stands for Memory Interface Width). Intel(R) Core(TM) i7 CPU 920@2.67GHz (or Intel920) and Cell Broadband Engine (or Cell/BE) are also used as OpenCL devices. For the Cell/BE, we use the OpenCL implementation from IBM. For the Intel920, we use the implementation from AMD (APP v2.2), because Intel's implementation on Linux is still unavailable at the moment of writing.

TABLE III
DETAILS OF UNDERLYING PLATFORMS

	Saturn	**Dutijc**	**Jupiter**
Host CPU	Intel(R) Core(TM) i7 CPU 920@2.67GHz		
Attached GPUs	GTX480	GTX280	Radeon HD5870
gcc version	4.4.1	4.4.3	4.4.1
CUDA version	3.2	3.2	—
APP version	—	—	2.2

TABLE IV
SPECIFICATIONS OF GPUS

	GTX480	**GTX280**	**HD5870**
Architecture	Fermi	GTX200s	Cypress
#Compute Unit	60	30	20
#Cores	480	240	320
#Processing Elements	—	—	1600
Core Clock(MHz)	1401	1296	850
Memory Clock(MHz)	1848	1107	1200
MIW(bits)	384	512	256
Memory Capacity(GB)	GDDR5 1.5	GDDR3 1	GDDR5 1

IV. PERFORMANCE COMPARISON AND ANALYSIS

A. Comparing Peak Performance

1) Bandwidth of Device Memory: TP_{BW} (Theoretical Peak Bandwidth) is given as follows:

$$TP_{BW} = MC * (MIW/8) * 2 * 10^{-9} \quad (2)$$

where MC is the abbreviation for Memory Clock. Using Equation 2 we calculate TP_{BW} of GTX280 and GTX480 to be 141.7 GB/sec and 177.4 GB/sec, respectively.

AP_{BW} (Achieved Peak Bandwidth) is measured here by reading global-memory in a coalesced manner. Moreover, our experimental results show that AP_{BW} depends on work-group-size (or block-size), which we set to 256. The results of the experiments with DeviceMemory on Saturn (GTX480) and Dutijc (GTX280) are shown in Figure 1. We see that OpenCL outperforms CUDA in AP_{BW} by 8.5% on GTX280 and 2.4% on GTX480. Further, the OpenCL implementation achieves 68.6% and 87.7% of TP_{BW} on GTX280 and GTX480, respectively.

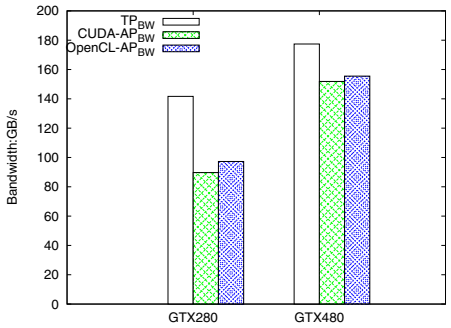


Fig. 1. A comparison of the peak bandwidth for GTX280 and GTX480

2) Floating-Point Performance: TP_{FLOPS} (Theoretical Peak Floating-Point Operations per Second) is calculated as follows:

$$TP_{FLOPS} = CC * \#Cores * R * 10^{-9} \quad (3)$$

where CC is short for Core Clock and R stands for maximum operations finished by a scalar core in one cycle. R differs depending on the platforms: it is 3 for GTX280 and 2 for GTX480, due to the dual-issue design of the GT200 architecture. As a result, TP_{FLOPS} is equal to 933.12 GFlops/sec and 1344.96 GFlops/sec for these two GPUs, respectively.

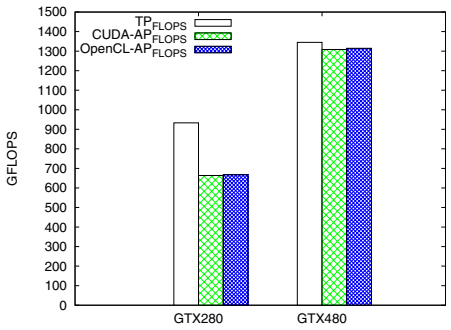


Fig. 2. A comparison of the peak FLOPS for GTX280 and GTX480

AP_{FLOPS} (Achieved Peak FLOPS) in MaxFlops is measured in different ways on GTX280 and GTX480. For GTX280, a `mul` instruction and a `mad` instruction appear in an interleaved way (in theory they can run on one scalar core simultaneously), while only `mad` instructions are issued for GTX480. The experimental results are compared in Figure 2. We see that OpenCL obtains almost the same AP_{FLOPS} as CUDA for GTX280 and GTX480, accounting for approximately 71.5% and 97.7% of the corresponding TP_{FLOPS}.

Thus, CUDA and OpenCL are able to achieve similar peak performance (to be precise, OpenCL even performs slightly better), which shows that OpenCL has the same potential to use the underlying hardware as CUDA.

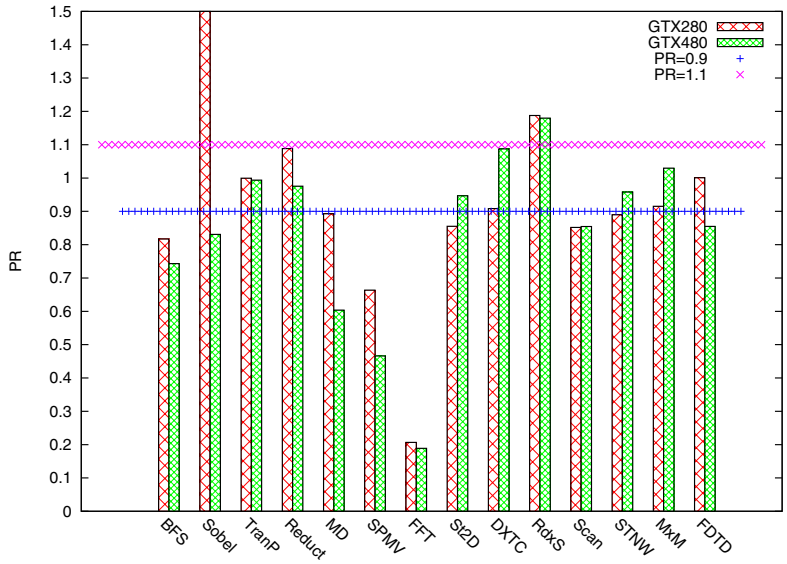
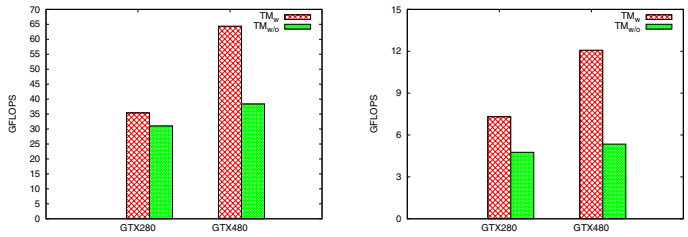


Fig. 3. A performance comparison of selected benchmarks. When the top border of a rectangle lies in the area between Line $\{PR = 0.9\}$ and Line $\{PR = 1.1\}$, we assume CUDA and OpenCL have similar performance. (Note that on GTX280, the PR for Sobel is 3.2)

B. Performance Comparison of Real-world Applications

The real-world applications mentioned in Section III-B are selected to compare the performance of CUDA and OpenCL. The PR of all the real-world applications without any modifications is shown in Figure 3. As can be seen from the figure, PR varies a lot when using different benchmarks and underlying GPUs. We analyze these performance differences using the following criteria.

1) Programming Model Differences: as is shown in Section I-A, CUDA and OpenCL have many conceptual similarities. However, there are also several differences in programming models between CUDA and OpenCL. For example, NDRange in OpenCL represents the number of work-items in the whole problem domain, while GridDim in CUDA is the number of blocks.



(a) MD Benchmark (b) SPMV Benchmark

Fig. 4. Performance impact of texture memory

Additionally, they have different abstractions of device memory hierarchy, where CUDA explicitly supports specific hardware features which OpenCL avoids for portability reasons. Through analyzing kernel codes, we find that texture memory is used in the CUDA implementations of MD and SPMV. Both benchmarks have intensive and irregular access

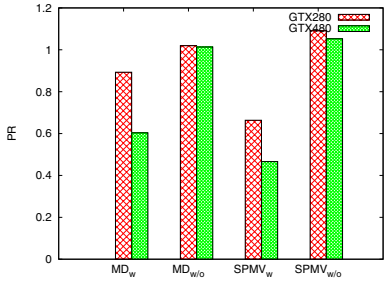


Fig. 5. Performance ratio before and after removing texture memory

to a read-only global vector, which is stored in the texture memory space. Figure 4 shows the performance of the two applications when running with and without the usage of texture memory. As can be seen from the figure, after the removal of the texture memory, the performance drops to about 87.6%, 65.1% on GTX280 and 59.6%, 44.3% on GTX480 of the performance with texture memory for MD and SPMV, respectively. We compare the performance of OpenCL and CUDA after removing the usage of texture memory. The results of this comparison are presented in Figure 5, showing similar performance between CUDA and OpenCL. It is the special support of texture cache that makes the irregular access look more regular. Consequently, texture memory plays an important role in performance improvement of kernel programs.

2) Different Optimizations on Native Kernels: in [24], many optimization strategies are listed: (i) ensure global memory accesses are coalesced whenever possible; (ii) prefer shared memory access wherever possible; (iii) use shift operations to avoid expensive division and modulo calculations; (iv) make it easy for the compiler to use branch prediction instead of loops, etc.

One of the important optimization to be performed in kernel codes is to reduce the number of dynamic instructions in the run-time execution. Loop unrolling is one of the techniques that reduces loop overhead and increases the computation per loop iteration [25]. NVIDIA's CUDA provides an interface to unroll a loop fully or partially using the pragma `unroll`. When analyzing the native kernel codes of FDTD (as is illustrated in the following list), we find these two codes are the same except that the CUDA code uses the pragma `unroll` at both unroll points a and b, while the OpenCL one unrolls the loop only at point b.

```
// Code segment of FDTD kernel
// Step through the xy-planes
#pragma unroll 9   //unroll point: a
for (int iz=0; iz<dimz; iz++){
  // some work here
#pragma unroll RADIUS   //unroll point: b
  for (int i=1; i<=RADIUS; i++){
    // some work here
  }
  // some work here
}
```

The performance of the application (in CUDA only) with and without the pragma `unroll` at point a is shown in Figure 6. We can see that the performance without the pragma `unroll` drops to 85.1% and 82.6% of the performance with it for GTX280 and GTX480. We then remove the pragma at point a from the CUDA version and present a performance comparison between CUDA and OpenCL in Figure 7. It can be seen that they achieve similar performance on GTX480, while OpenCL outperforms CUDA by 15.1% on GTX280. Moreover, we observe that when adding the pragma `unroll` at unroll point a of the OpenCL implementation, the performance degrades sharply to 48.3% and 66.1% of that of the CUDA implementation for GTX280 and GTX480, also shown in Figure 7.

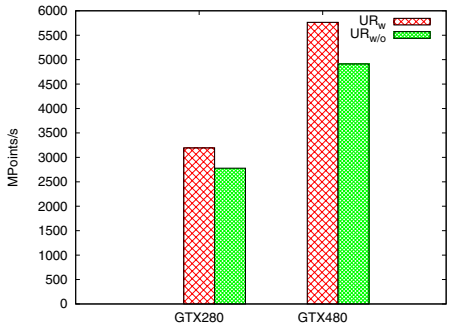


Fig. 6. Performance impact of loop-unrolling (CUDA only)

3) Architecture-related Differences: since the birth of the original G80, the Fermi architecture can be seen as the most remarkable leap forward for GPGPU computing. It differs from the previous generations by, e.g, (i) improved double precision performance; (ii) ECC support; (iii) true cache hierarchy; (iv) faster context switching [26].

The introduction of the cache hierarchy has a significant impact on Fermi's performance. When looking at Figure 3, we

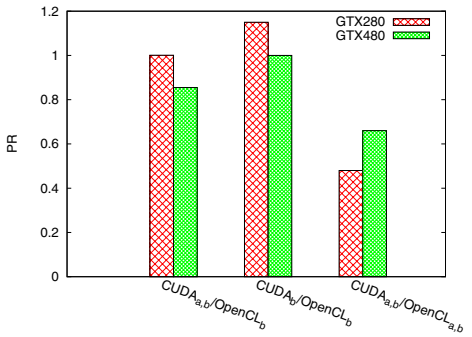


Fig. 7. A performance comparison of FDTD with/without loop-unrolling at different points ($CUDA_x$ represents we execute loop-unrolling at point x, and it is the same for OpenCL. For example, the third group $CUDA_{a,b}/OpenCL_{a,b}$ represents we unroll the loop at both points for CUDA and OpenCL).

see that the values diverge remarkably for Sobel on GTX280 and GTX480. On GTX280, the OpenCL version runs three times faster than the CUDA one, but it only obtains 83% of CUDA's performance when the benchmark runs on GTX480. These differences are caused by the constant memory and the cache. In the implementation with OpenCL, constant memory is employed to store the "filter" in Sobel, while it is not in the CUDA version.

After removing the usage of constant memory, we do the same experiments on these two GPUs. The execution time is presented in Figure 8. On the one hand, we see the kernel execution time drops to one quarter of that without using constant memory on GTX280. On the other hand, there are few changes on GTX480 due to the availability of the global-memory cache in the Fermi architecture. Overall, CUDA and OpenCL achieve similar performance with/without constant memory on GTX480.

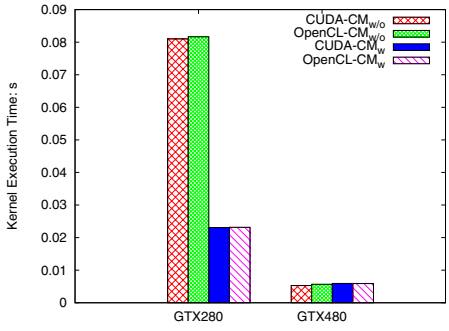


Fig. 8. A performance comparison for Sobel with and without constant memory on GTX280 and GTX480

4) Compiler and Run-time Differences: among all the benchmarks, the performance gap between OpenCL and CUDA is the biggest for the FFT. Their native kernel codes are exactly the same. However, when looking into their PTX codes, we find notable differences between them. A quantitative comparison of these two PTX kernels is presented in Table V. The statistics are gathered for the "forward" kernel

TABLE V
STATISTIC FOR PTX INSTRUCTIONS

		Instruction Count				Instruction Count	
Class	**Instructions**	**CUDA**	**OpenCL**	**Class**	**Instructions**	**CUDA**	**OpenCL**
Arithmetic	add	93	191	Data Movement	cvt	16	16
	sub	83	95		mov	687	88
	mul	33	138		ld.param	1	1
	div	0	2		ld.local	97	64
	fma	0	37		ld.shared	32	32
	mad	2	22		ld.const	0	24
	neg	9	36		ld.global	8	8
	and	1	291		st.local	250	78
Sub-total		220	521		st.shared	32	32
Logic Shift	or	2	33		st.global	8	8
	not	0	4	Sub-total		1131	351
	xor	0	4	Flow Control	setp	2	80
	shl	0	50		selp	0	40
	shr	1	43		bra	2	68
Sub-total		4	163	Sub-total		4	188
Synchronization	bar	7	7	Total		1366	1230

of the FFT implementation.

From Table V, the differences between these two PTX codes become visible. The OpenCL front-end compiler generates two times more arithmetic instructions than its CUDA counterpart. There are rarely any logic-shift instructions in CUDA, while there are 163 such instructions in the OpenCL kernel. A similar situation happens with the flow-control instructions: there are many more for OpenCL than for CUDA. Although there are many more data-movement instructions for CUDA, most of them are `mov`, simply moving data to or from registers or local memories. Finally, we note that all time-consuming instructions such as `ld.global` and `st.global` are exactly the same.

We can explain this situation by assuming that the front-end compiler for CUDA has been used and optimized more heavily, thus is more mature, than that of OpenCL. As a result, when it comes to some kernels like "forward" in FFT, OpenCL performs worse than CUDA.

BFS is also an interesting example here. It has to invoke the kernel functions several times to solve the whole problem. Thus, the kernel launch time (the time that a kernel takes from entering the command-queue until starting its execution) plays a significant role in the overall performance. Our experimental results show that the kernel launch time of OpenCL is longer than that of CUDA (the gap size depends on the problem size), due to differences in the run-time environment. The longer kernel launch time may also explain why OpenCL performs worse than CUDA for applications like BFS.

In the previous analysis, we only identify the most influential factor for each application that shows an observable performance difference. It is important to note that several factors may often affect the program performance together, leading to larger performance discrepancies. An analysis of such combinations, as well as the investigation of lower level factors (such as compiler optimizations), is left for future work.

C. A Fair Comparison

So far, we have shown that the performance gaps between OpenCL and CUDA are due to programming model differences, different optimizations on native kernels, architecture-related differences, and compiler differences. It has been shown that performance can be equalized by systematic code changes. Therefore, we present an eight-step fair comparison approach for CUDA and OpenCL applications from the original problem to its final solution, which provides guidelines for investigating the performance gap between CUDA and OpenCL (if any). A schematic view of this approach is shown in Figure 9.

1) Problem Description: this step describes what the problem is and what form the solutions could be.

2) Algorithm Translation: how to address the problem is given using certain algorithms. The algorithms can be described in pseudo-code which is environment-independent and easier for humans to understand.

3) Implementation: in this step, the algorithms mentioned above are implemented with different programming models or languages. As for GPU programs, there are two parts: one is the host program and the other is the kernel code running on GPUs. On NVIDIA GPUs, CUDA+C and OpenCL+C are usually adopted to implement GPU programs. If two implementations use similar APIs to access the same type of hardware resources, we consider these two implementations to be the same. Note that two implementations also have to use the same type of timers to measure performance.

4) Native Kernel Optimizations: after implementation, architecture-dependent optimizations on kernel programs are

executed. For example, whether to use the shared memory (or local memory in OpenCL), whether to employ vectorization, whether to unroll loops, whether to reduce bank-conflicts, whether to use texture memory in CUDA, and whether to access global memory in a coalesced way. are decisions that should be taken into account. On the one hand, optimizations on native kernels is a time-consuming and error-prone job; on the other hand, it can contribute to performance improvement significantly.

5) First-Stage Compilation and Optimization: the first-stage compiler adopted in CUDA is called NVOPENCC. There is a similar front-end compiler for OpenCL in this stage. This stage compiles kernel codes into PTX codes, a low-level parallel thread execution virtual machine and instruction set architecture (ISA) developed by NVIDIA [27]. Some advanced optimizations are also executed in this stage.

6) Second-Stage Compilation and Optimization: PTXAS (the back-end compiler) translates PTX codes into binary format in this step and it may execute some additional optimizations.

7) Program Configuration and Start-up: before executing the program prepared so far, we need to configure two kinds of parameters: (1) problem parameters (the parameters of the problem to be solved such as the size of the matrix), and (2) algorithmic parameters (for example, block-size or work-group size). Although these parameters don not change the correctness of final results, they can have a significant impact on the performance of the application.

8) Running on GPUs: With the help of drivers, the binary codes are finally scheduled to run on the GPUs.

These eight steps make up the application development flow from an original problem to its final solution. Based on this, we define that a comparison for CUDA and OpenCL is "fair" when configurations in all the eight steps of the comparison are the same. According to the analysis in previous subsection, OpenCL can obtain similar performance to CUDA in the case of "a fair comparison". In real-world, programmers are responsible for steps (1) - (4) and compilers take charge of steps (5), (6). Finally, users will employ the application through steps (7) and (8), as is illustrated in Figure 9. Each of the eight steps is probably executed by different programmers (they have different programming habits, abilities and choices) or different compilers (they may execute different optimizations) or different users (they have different requirements and investments). All those lead to the difficulty of making sure that a performance comparison is fair for CUDA and OpenCL.

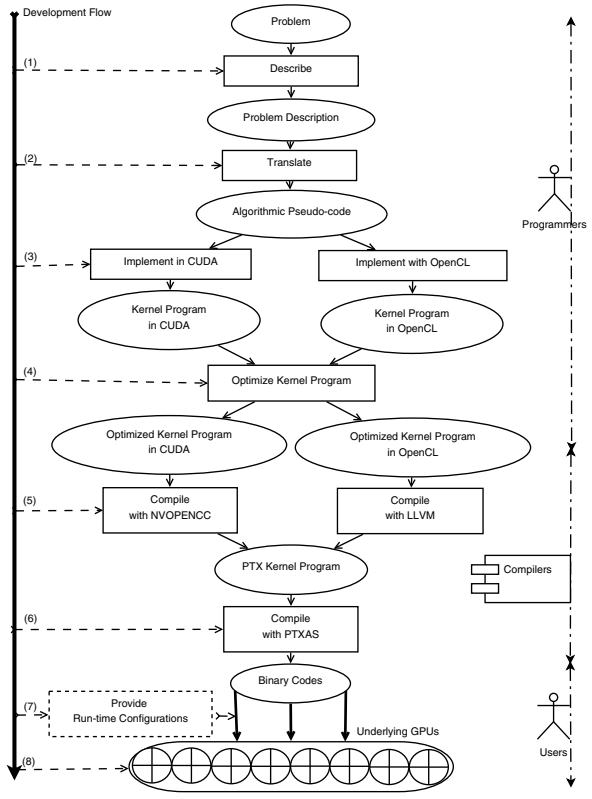


Fig. 9. Development flow of GPU kernel programs (The ellipses represent entities such as a program or a description and the rectangles represent actions on the entities. We categorize three types of roles participating the whole process: programmers, compilers, and users.)

V. A Brief Evaluation of OpenCL's Portability

We have seen so far that for a set of 16 benchmarks, OpenCL implementations differ from the CUDA ones on performance. Given that OpenCL's portability is typically invoked as a good reason for performance drops, we investigate if the portability claim holds by porting all the real-world benchmarks from NVIDIA's GPUs to HD5870, Intel920 and Cell/BE. All performance data is listed in Table VI (the performance units are the same as those shown in Table II).

When comparing performance on NVIDIA's GPUs, we find that most benchmarks, without additional optimizations on HD5870, achieve comparable performance with that on GTX280. An exceptional example is TranP on HD5870 which performs much worse than it does on GTX280.

When benchmarks run on Intel920 and Cell/BE, we have to make some minor modifications of changing CL_DEVICE_TYPE_GPU to CL_DEVICE_TYPE_CPU or to CL_DEVICE_TYPE_ACCELERATOR for benchmarks selected from the CUDA SDK. Moreover, there are more programming constraints on Cell/BE (e.g. `get_local_id` or `cosine` functions are not allowed within inline definition of another function). When it comes to performance on Intel920, we observe that the bandwidth of TranP drops from 2.411 GB/sec to 0.2150 GB/sec because of using local memory: all OpenCL memory objects for CPU are cached implicitly by hardware and thus explicitly using local memory just introduces unnecessary overhead. Another interesting observation is that SPMV sees a performance degradation from 3.805 GFlops/sec to 0.1247 GFlops/sec when employing warp-oriented optimization (using a warp of threads to work together on one matrix row). We believe this happens because there are orders of magnitude less processing cores in CPUs than in GPUs.

TABLE VI
PERFORMANCE DATA ON PREVAILING PLATFORMS

	BFS	Sobel	TranP	Reduct	MD	SPMV	FFT	St2D	DXTC	RdxS	Scan	STNW	MxM	FDTD
HD5870	0.0246	0.0048	5.951	114.4	28.60	4.665	36.10	1.666	14.50	FL	177.6	42.43	205.3	3352
Intel920	0.1455	0.1553	2.411	0.9936	2.597	3.805	1.424	238.4	14.48	FL	1.071	0.7605	0.8857	3787
Cell/BE	1.159	5.425	0.1993	0.0528	0.1264	0.0809	ABT	0.1178	ABT	ABT	1.620	ABT	1.473	19.15

In Table VI, "ABT" means the programs (FFT, DXTC, RdxS, and SNTW) exit, showing "aborted". It is mainly because there are not enough resources on the Cell/BE. For example, DXTC shows "CL_OUT_OF_RESOURCES" when invoking `clEnqueueNDRangeKernel` because of insufficient registers or local memories. The possible solution we can imagine now is to make the input problem size smaller.

We also find that RdxS can end normally, but get wrong results (denoted by "FL" in the Table VI) on HD5870 and Intel920. The benchmark uses the four-step radix sort in each pass proposed in papers [28] [29]. However, the implementation of RdxS depends on warp-size in CUDA, i.e., wavefront-size in APP. The warp-size is 32 in CUDA, while it is 64 in APP. Therefore, only one half warp of threads are able to map keys into buckets and the other half are not when it comes to APP, leading to incorrectly sorted sequences. This is a typical example of hiding platform specific details into programs, and can be considered as a programmer's mistake.

To sum up, all the benchmarks compile correctly and most of them run properly on the other platforms, illustrating OpenCL's cross-platforms portability. In order to make OpenCL programs run on more platforms, programmers are encouraged to use vendor-independent terms (for example, CL_DEVICE_TYPE_ALL) and provide users with optional choices. After all, even minor modifications and additional debugging can be time-consuming. When it comes to a specific architecture, an auto-tuner could be used to boost performance [30]. Finally, we note that OpenCL is very useful as a prototyping tool, enabling portability while still achieving good performance.

VI. CONCLUSIONS

From the results and analysis above, we can see that there is no reason for OpenCL to obtain worse performance than CUDA under a fair comparison. Several benchmarks also show the interesting performance gaps. The reasons behind the gaps are analyzed thoroughly and they can all be essentially related to various behaviors of programmers, compilers and users. We also port all the real-world benchmarks to other platforms with minor modifications to show OpenCL's potential for portability.

Since it has been shown in this paper that OpenCL is a good alternative to CUDA, we would like to develop an auto-tuner to adapt general-purpose OpenCL programs to all available specific platforms to fully exploit the hardware.

ACKNOWLEDGMENT

We would like to thank the authors from the SHOC benchmark suite, the CUDA SDK and the Rodinia benchmark suite for their valuable benchmarks. We would like to thank Professor David Patterson from UC Berkeley for his instant reply and comments on the 7+ dwarfs. We thank our group member Mehdi Chitchian for his valuable remarks on this paper. We are also thankful to the reviewers for their comments. This work is partially funded by the CSC (China Scholarship Council).

REFERENCES

[1] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, pp. 13–27, July 2008.

[2] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, pp. 56–69, March 2010.

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, pp. 80–113, March 2007.

[4] NVIDIA Inc., "NVIDIA Cg Toolkit." http://developer.nvidia.com/page/cg_main.html, Februay 2011.

[5] Microsoft Inc., "Reference for HLSL." http://msdn.microsoft.com/en-us/library/bb509635(v=VS.85).aspx, Februay 2011.

[6] Khronos Group, "OpenGL Shading Language." http://www.opengl.org/documentation/glsl/, Februay 2011.

[7] NVIDIA Inc., "CUDA Toolkit 3.2." http://developer.nvidia.com/object/cuda_3_2_downloads.html, Februay 2011.

[8] AMD Inc., "AMD Accelerated Parallel Processing (APP) SDK." http://developer.amd.com/gpu/amdappsdk/pages/default.aspx, Februay 2011.

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.

[10] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki, *The OpenCL Programming Book*. Fixstars Corporation, March 2010.

[11] The Khronos OpenCL Working Group, "OpenCL - The open standard for parallel programming of heterogeneous systems." http://www.khronos.org/opencl/, Februay 2011.

[12] D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Feb. 2010.

[13] AMD Inc., "Porting CUDA Applications to OpenCL." http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx, Februay 2011.

[14] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 58–68, January 2011.

[15] R. van Nieuwpoort and J. Romein, "Correlating radio astronomy signals with Many-Core hardware," *International Journal of Parallel Programming*, vol. 39, pp. 88–114, Feb. 2011.

[16] T. I. Vassilev, "Comparison of several parallel API for cloth modelling on modern GPUs," in *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, CompSysTech '10, (New York, NY, USA), pp. 131–136, ACM, 2010.

[17] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos, "Comparing CUDA and OpenGL implementations for a Jacobi iteration," pp. 22–32, June 2009.

[18] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," May 2010.
[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," tech. rep., Department of Computer Science, UTK, Knoxville Tennessee, September 2010.
[20] K. Komatsu1, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi1, "Evaluating Performance and Portability of OpenCL Programs," in *Proceedings of the Fifth international Workshop on Automatic Performance Tuning(iWAPT2010)*, (Berkeley, USA), June 2010.
[21] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, (New York, NY, USA), pp. 63–74, ACM, 2010.
[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," pp. 44–54, October 2009.
[23] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.
[24] NVIDIA Inc., *OpenCL Best Practices Guide*, May 2010.
[25] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 225–234, ACM, 2008.
[26] NVIDIA Inc., *NVIDIAs Next Generation CUDA Compute Architecture: Fermi*, 2009.
[27] NVIDIA Inc., *PTX: Parallel Thread Execution ISA Version 2.2*, October 2010.
[28] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 712–721, ACM, 1991.
[29] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2009.
[30] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, (New York, NY, USA), pp. 86–97, ACM, 2010.