Evaluating Performance and Portability of OpenCL Programs

Kazuhiko Komatsu[1], Katsuto Sato[2], Yusuke Arai[2], Kentaro Koyama[2], Hiroyuki Takizawa[2,3], and Hiroaki Kobayashi[1,3]

[1] Cyberscience Center, Tohoku University
Sendai Miyagi 980-8578, Japan
komatsu@sc.isc.tohoku.ac.jp, koba@isc.tohoku.ac.jp
[2] Graduate School of Information Sciences, Tohoku University
Sendai Miyagi 980-8578, Japan
{katuto, kentaro}@sc.isc.tohoku.ac.jp, a_yousk@ic.is.tohoku.ac.jp,
tacky@isc.tohoku.ac.jp
[3] Japan Science and Technology Agency, CREST

Abstract. Recently, OpenCL, a new open programming standard for GPGPU programming, has become available in addition to CUDA. OpenCL can support various compute devices due to its higher abstraction programming framework. Since there is a semantic gap between OpenCL and compute devices, the OpenCL C compiler plays important roles to exploit the potential of compute devices and therefore its capability should be clarified. In this paper, the performance of CUDA and OpenCL programs is quantitatively evaluated. First, several CUDA and OpenCL programs of almost the same computations are developed, and their performances are compared. Then, the main factors causing their performance differences is investigated. The evaluation results suggest that the performances of OpenCL programs are comparable with those of CUDA ones if the kernel codes are appropriately optimized by hand or by the compiler optimizations. This paper also discusses the differences between NVIDIA and AMD OpenCL implementations by comparing the performances of their GPUs for the same programs. The performance comparison shows that the compiler options of the OpenCL C compiler and the execution configuration parameters have to be optimized for each GPU to obtain its best performance. Therefore, automatic parameter tuning is essential to enable a single OpenCL code to run efficiently on various GPUs.

1 Introduction

Nowadays, GPUs (Graphics Processing Units) are used not only for graphics applications, but also for non-graphics applications, so-called *GPU computing* or GPGPU (General-Purpose computation on GPUs). Thanks to their high floating-point operation rates and memory bandwidths, GPUs can accelerate various science and engineering computations [1–6].

In addition to rapid improvement in GPU performance, the programming flexibility of GPUs has improved dramatically. Particularly, CUDA (Compute

Unified Device Architecture) [7] released by NVIDIA in 2007 can significantly facilitate GPU programming, and thus has played a very important role to encourage broad use of GPU computing. AMD has also provided another programming framework, ATI Stream, for its GPUs [8]. Those programming frameworks allow a programmer to develop a GPU computing application without tricky graphics programming techniques. However, each programming framework forces a programmer to write code in its own programming language, which extends the standard C/C++ programming language with some special keywords. Therefore, CUDA programs and ATI Stream programs can run only on NVIDIA's GPUs and AMD's GPUs, respectively; they are not compatible.

OpenCL is a new open programming standard for various compute devices [9]. OpenCL is used to develop code not only for GPUs, but also for multi-core CPUs, Cell Broadband Engines, and other compute devices; a programmer can use these compute devices in a unified way. Thus, it can enable a programmer to avoid writing a vendor-specific code, resulting in improved code portability.

However, although OpenCL allows a programmer to use various compute devices, efficient coding and optimization methodologies for individual compute devices are not established yet. While both CUDA and ATI Stream have intensively been optimized to exploit the computing power of their own GPUs, there is a semantic gap between OpenCL and compute devices because OpenCL is vendor-independent and hence not specialized for any particular compute device. The sustained performance of a program developed with OpenCL might be lower than that with CUDA. Thus, it is still unclear that OpenCL can achieve the same performance as other programming frameworks that are designed for particular compute devices.

In this paper, we quantitatively compare the sustained performances of OpenCL programs with those of CUDA programs. To make a fair comparison, CUDA codes are ported to OpenCL codes as faithfully as possible, mostly by replacing CUDA's keywords and functions with the corresponding OpenCL's ones. Based on the performance comparison, this paper analyzes and discusses the main factors of causing the performance differences. In addition, this paper also shows the difference in sustained performance between an NVIDIA GPU and an AMD GPU. Since OpenCL can enable both GPUs to run the same code, a fair comparison of their performances can be performed.

The rest of this paper is organized as follows. Section 2 briefly reviews OpenCL and CUDA that are a vendor-independent programming environment and a vendor-specific environment, respectively. Section 3 discusses several factors that cause the performance differences between OpenCL and CUDA programs and shows the evaluation results to clarify the sustained performance of each programing environment. Section 4 shows that some parameters tuning is required for each compute device to achieve its best performance through the performance comparisons among three compute devices. This suggests that an automatic performance tuning mechanism is required to improve the performance portability of OpenCL programs. Finally, Section 5 gives conclusions of this paper.

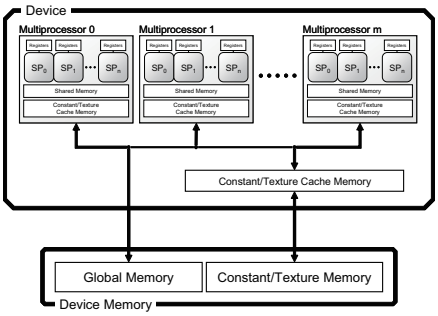

Fig. 1. GPU architecture in CUDA.

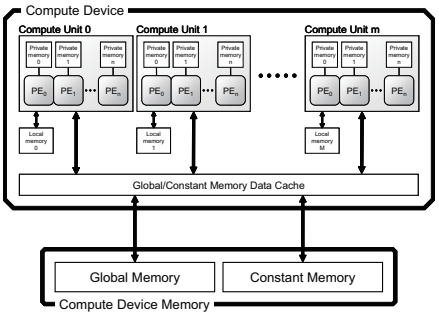

Fig. 2. GPU architecture in OpenCL.

2 Overview of OpenCL

2.1 Similarities and Differences between CUDA and OpenCL

CUDA is currently the de facto standard programming framework for GPU computing. CUDA enables a programmer to access GPUs without tricky graphics programming techniques, which are required for GPU programming with graphics APIs such as OpenGL [10] and DirectX [11]. Thus, CUDA significantly facilitates GPU programming to exploit the computing power of GPUs. However, CUDA is only available for NVIDIA's GPUs. Therefore, GPU programming with CUDA sacrifices the code portability to exploit the GPU computing power.

In 2009, OpenCL was announced as an open programming standard to access GPUs and other compute devices in a unified manner. The specifications and programming languages of OpenCL and CUDA have similarities in many aspects. Therefore, CUDA programmers can harness their experiences and skills to write an efficient OpenCL program.

Figures 1 and 2 show the architecture models of compute devices in CUDA and OpenCL, respectively. These figures show that their models have similar memory hierarchies and processing elements. One difference is that there are texture memories and caches only in CUDA. In CUDA, the texture data on the device memory are read-only from the processing elements and cached in the texture cache. On the other hand, texture memories and caches are not defined in the OpenCL specification, because OpenCL has to support not only GPUs but also other compute devices that may not use the textures. However, as with the texture memory of CUDA, some device memory data in OpenCL, called images, may be cached depending on the capabilities of compute devices.

Figures 3 and 4 show the execution models of CUDA and OpenCL, respectively. In CUDA, *grid* is a term to denote a set of all threads launched for execution of a *kernel* code. A grid is decomposed into several *thread blocks* of the same size, and each thread block is assigned to a Multi-Processor (MP). In CUDA, a thread block is further decomposed into *warps* [7], each consisting of 32 threads. Each warp is executed on a MP in a SIMD manner.

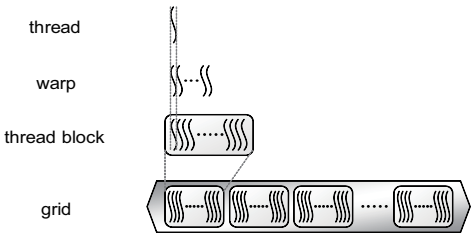

Fig. 3. Thread hierarchy in CUDA.

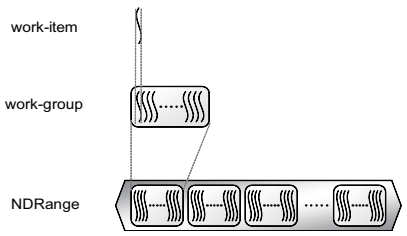

Fig. 4. Thread hierarchy in OpenCL.

In OpenCL, an *NDRange* is an N-dimensional index space, corresponding to a grid in CUDA. Besides, a *work-group* and a *work-item* correspond to a thread block and a thread in CUDA, respectively. Thus, the execution model of OpenCL is similar to that of CUDA.

Since OpenCL does not assume any particular compute device, it does not provide any concept corresponding to the warp specific to NVIDIA's GPUs, and *wavefronts* [12] specific to AMD's ones. By eliminating such vendor-specific definitions from the specification, OpenCL can support various kinds of compute devices. In practice, however, work-items of an OpenCL program are clustered into warps or wavefronts; work-items in a warp or a wavefront are simultaneously executed on actual GPUs in a SIMD manner. Thus, it is still important to consider vendor-specific hardware features, such as warps and wavefronts, to write an efficient OpenCL kernel for GPUs. Accordingly, optimization methodologies for individual GPUs may be different, even though the same programming interface is used in OpenCL code.

Table 1 summarizes the memory hierarchies of CUDA and OpenCL. In CUDA, the *local memory* is assigned to each thread. Only the assigned thread can access the local memory, which is a part of the off-chip device memory and is assigned when register spilling occurs. The *shared memory* is assigned to each thread block, and only threads in the thread block can access the shared memory, which is the small on-chip memory with a short access latency. In addition to the *global memory*, every thread can access *constant memory* and *texture memory*, each of which is the read-only off-chip memory with a cache mechanism. The global memory is the largest off-chip memory, but it needs a long access latency as well as the local memory. OpenCL assumes almost the same memory hierarchy as CUDA, even though their terminologies are different as shown in Table 1.

In CUDA, a kernel code executed on the compute device is statically compiled when the host code is compiled. On the other hand, as OpenCL uses the JIT compilation by default, it can generate an appropriate binary code after obtaining the system configuration. However, the compilation time is always included in the execution time of the OpenCL application, and hence time-consuming compiler optimizations may increase the total execution time.

Table 1. Memory hierarchy.

Memory accessability	CUDA	OpenCL	Readable/Writable
thread / work-item	local memory	private memory	R / W
CTA / work-group	shared memory	local memory	R / W
Grid / NDRange	global memory	global memory	R / W
	constant memory	constant memory	Readable
	texture memory	—	Readable

2.2 Parallel Thread Execution

Parallel Thread Execution (PTX) [13] defines a low-level virtual machine and instruction set architecture for general purpose parallel thread execution on NVIDIA GPUs. The CUDA C compiler can generate PTX codes, which will further be optimized for and translated to the GPU instructions. The translator and deriver enable GPUs to be used as programmable parallel computers.

In CUDA, a kernel code is usually written in high-level languages such as CUDA C and CUDA Fortran. Similarly, in OpenCL, the OpenCL C language is usually used to describe a kernel code. In the cases of GPU computing with NVIDIA GPUs, moreover, such a high-level kernel code can be translated into a PTX code in both CUDA and OpenCL environments. At runtime, the PTX code is then translated into the native code executable by the GPU device. As PTX is a lower-level representation than CUDA C and OpenCL C, analysis of a PTX code is helpful to discuss reasons of the performance difference in kernel execution between CUDA and OpenCL.

3 Performance Evaluation of OpenCL Compared with Vendor-dependent Framework

This section shows the evaluation results to clarify the difference in sustained performance between OpenCL and CUDA programs. To make their fair comparison, CUDA programs are ported to OpenCL ones as faithfully as possible. CUDA keywords and API functions are replaced with the corresponding OpenCL ones. Then, some minor modifications are applied to the OpenCL programs if necessary so that the programs can correctly work.

In the evaluation, two benchmark programs are selected from NVIDIA GPU Computing SDK 3.0. One is `bandwidthTest` that measures sustained bandwidths between a CPU and a GPU. The other is `matrixMul` that measures sustained performance of calculations of product of matrices. The size of each matrix in the CUDA program is set to the same size used in the OpenCL one.

In addition, three CUDA programs, Coulombic Potential (`CP`), Magnetic Resonance Imaging Q (`MRI-Q`), and Magnetic Resonance Imaging FHD (`MRI-FHD`), have been selected from the Parboil benchmark suite [14], and translated to OpenCL programs by replacing CUDA keywords and API functions with the corresponding ones in OpenCL. Those programs operate on data structures with

Table 2. PC specification.

CPU	Intel Core i7 920
GPU	NVIDIA Tesla C1060 AMD Radeon HD 5870
Chipset	Intel X58
Main Memory	12GB
Video Memory	Tesla 4GB, Radeon 1GB
OS	CentOS 5.4 kernel 2.6.18
Platform	NVIDIA CUDA toolkit 3.0 (OpenCL revision 1.0.48 support) ATI Stream SDK v2.01 with OpenCL 1.0 Support
Driver	NVIDIA driver version 195.36.15 AMD driver version 10.3

Table 3. Compute device specification.

Compute device	Intel Core i7 920	NVIDIA Tesla C1060	AMD Radeon HD 5870
# of processing elements	4	240	1600
Core frequency	2.67 GHz	1.3 GHz	850 MHz
Control units	-	30	20
Size of a work-group	8	32	64
Memory size	12GB	4GB	1GB
Memory bandwidth	25.6 GB/s	102 GB/s	153.6 GB/s
Peak performance (float)	42.56 Gflop/s	933 Gflop/s	2720 Gflop/s
Peak performance (double)	42.56 Gflop/s	78 Gflop/s	544 Gflop/s

simple data layouts, and thereby their array accesses are predictable and uniform; the memory access patterns are well-suited for the GPU hardware. Basically, their kernels are able to run without waiting for memory accesses, and hence their performance are limited by the instruction issue rates [5]. Therefore, the compiler optimization capabilities such as common subexpression elimination and loop invariant computation motion [15] influence their performances. All of the evaluation results are obtained using the PC shown in Tables 2 and 3.

Figure 5 shows the sustained performance for each benchmark program on the NVIDIA GPU. The horizontal axis indicates the benchmark program names. The vertical axis indicates the total execution times, which include the initial setup time of kernels such as the JIT compilation in OpenCL. No compiler option is given to the JIT compiler invoked in the OpenCL programs. This figure shows that only the performance of the OpenCL-version `bandwidthTest` is almost the same as that of the CUDA-version. The other OpenCL-version programs run slower than the CUDA-version programs. The execution times for `matrixMul`, `CP`, `MRI-Q`, `MRI-FHD` of the CUDA programs are about 7.1, 3.2, 6.8, and 8.7 times shorter than those of the OpenCL programs, respectively.

The main reason why `bandwidthTest` behaves differently from the others is because it does not perform any calculations although it invokes some API functions for data transfers. These results suggest that the performance for invoking OpenCL API functions is almost the same as that for invoking CUDA API ones.

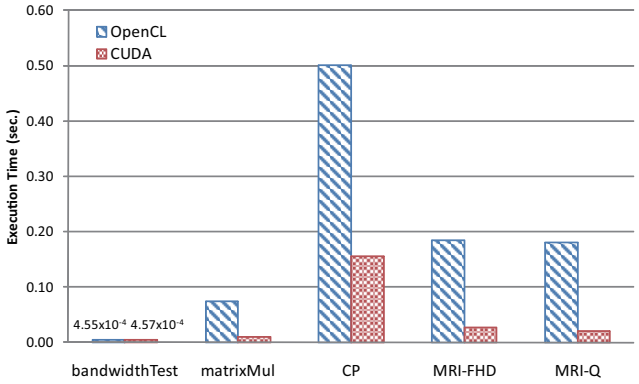

Fig. 5. Performance comparison between CUDA and OpenCL.

The CUDA programs except `bandwidthTest` perform much faster than the OpenCL programs. This is because the kernel execution times of CUDA programs are considerably shorter than those of OpenCL programs. Since the CUDA programs are faithfully ported to OpenCL ones, their kernel codes written in CUDA C and OpenCL C are very similar. Hence, it is obvious that their performance difference is caused by the difference between the kernel codes generated by the CUDA C compiler and the OpenCL C compiler. Therefore, by analyzing the `matrixMul`, `CP`, `MRI-FDH`, and `MRI-Q` kernel codes at the PTX level, the main reason of the performance difference is discussed below.

First, the main reason of the performance difference is discussed based on analysis of the `matrixMul` kernel code at the PTX level. In comparison between two PTX codes, we found that the PTX code generated by the CUDA C compiler is totally different from that by the OpenCL C compiler, even though those compilers translate essentially-identical kernel codes into PTX codes. The OpenCL C compiler generates a simple PTX code, while the CUDA C compiler generates its PTX code after several optimizations.

To confirm the effects of those optimization techniques applied by the CUDA C compiler, the same techniques are manually applied to the PTX code generated by the OpenCL C compiler. Figure 6 shows the performance improvements of the OpenCL program by optimizing its PTX code. The horizontal axis indicates the optimization techniques applied to the PTX code. The vertical axis indicates the kernel execution time.

Loop unrolling [15] is one of the most popular optimization techniques. It can reduce not only the numbers of conditional branches and index calculations, but also the number of memory accesses if the same memory address is accessed across iterations. Although the CUDA C compiler automatically unrolls the loop 16 times, the OpenCL C compiler does not unroll the loop. Hence, by manually unrolling the loop of the OpenCL-version PTX code, the total execution time of the OpenCL program is decreased by approximately 67.8%.

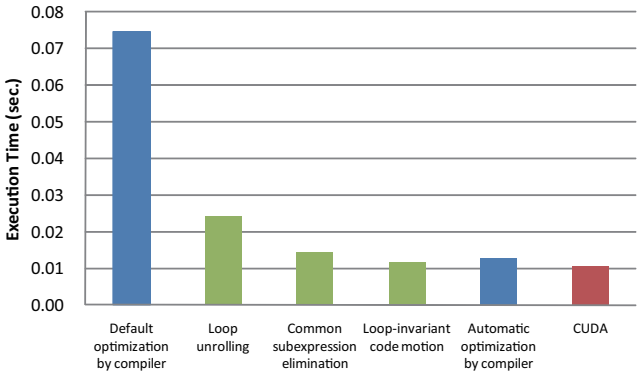

Fig. 6. Performance of `matrixMul` by manual optimization.

After manually unrolling the loop, the address calculation in the PTX code of OpenCL is still different from that of CUDA. In CUDA, common subexpression elimination [15] is adopted to avoid redundant calculations for the initial address of an array. On the other hand, in OpenCL, the initial address is always calculated before accessing the array. Hence, the execution time of the unrolled PTX code is further reduced by 40% by manually applying common subexpression elimination to the PTX code so as to avoid the redundant calculations.

PTX provides a multiply-and-addition (`mad`) arithmetic instruction. The `mad` instruction is frequently used in the PTX code generated by the CUDA C compiler. On the other hand, the PTX code of OpenCL executes two instructions, `add` and `mul`, to perform the same multiply-and-add operation. As the kernel is compute-intensive, it is likely that a reduction in the number of instructions improves the performance. However, we experimentally confirmed that use of two instructions instead of one `mad` instruction does not affect the performance. Even if the combination of `mul` and `add` is manually replaced by one `mad` instruction, the performance does not change. Therefore, the combination of `mul` and `add` would internally be replaced with one `mad` instruction in the driver.

Loop invariant code motion [15] is one optimization technique that moves the calculation outside the loop if the calculation result is unchanged during the loop execution. The CUDA C compiler automatically applies this optimization technique to the PTX code, while the OpenCL C compiler does not. Hence, by manually applying the technique to the PTX code of the OpenCL C compiler, the execution time is further reduced by approximately 20%.

The above discussions are based on manual optimization of the PTX code, which is generated by the OpenCL C compiler with no compiler option. Meanwhile, the OpenCL C compiler also provides some compiler options for automatic optimizations. By enabling the `-cl-fast-relaxed-math` option, the execution time of the OpenCL-version `matrixMul` kernel becomes almost the same as that of the CUDA-version kernel. Although the execution time of the automatically-generated PTX code is somewhat longer than that of the manually-optimized

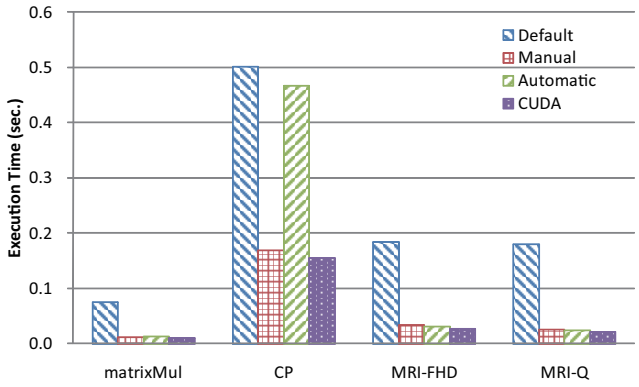

Fig. 7. Performance of benchmarks by manual optimizations.

code, the performance difference is small because almost the same optimization techniques such as loop unrolling, common subexpression elimination, and loop invariant motion are automatically applied to the code. The small performance difference comes from the difference of loop invariant motion optimization. In automatically-generated PTX, some invariant codes still remain in the loop.

Figure 7 shows the sustained performance of `matrixMul`, `CP`, `MRI-FHD`, and `MRI-Q`. The PTX codes of the programs are manually optimized. In addition to the results of the manual optimizations, those of the automatic optimizations by the JIT compiler of OpenCL with `-cl-fast-relaxed-math` option are also shown. In the cases of `CP`, `MRI-FHD`, and `MRI-Q`, we manually apply common subexpression elimination, loop invariant motion, use of intrinsic instructions such as `mad`, `rsqrt`, and trigonometrical instructions, because these techniques are already applied to the PTX code of the CUDA program by the compiler. Especially, use of `rsqrt` instructions is effective to improve the performance in `CP`, because it allows the PTX code to avoid using the division instruction, which is one of the most time-consuming instructions. In the cases of `MRI-FHD` and `MRI-Q`, use of trigonometrical intrinsic instructions is effective because calculations of a trigonometrical function is costly.

In comparison among the manually-optimized OpenCL codes, the automatically-optimized ones, and CUDA ones, the execution times are almost the same except in the case of `CP`. The execution time of the automatically-optimized `CP` kernel is obviously longer than those of the manually-optimized and CUDA-version kernels. The reason is that `rsqrt` instructions are not utilized at all in the automatically-optimized `CP`. Therefore, there is room to further improve the automatic optimization capabilities of the OpenCL C compiler.

These results indicate that the OpenCL programs can achieve the comparable performance with the CUDA programs by manually optimizing the PTX codes. Automatic compiler optimizations of the OpenCL C compiler can also be helpful to achieve the comparable performance, depending on the applications. Consequently, the sustained performances of OpenCL programs can become compa-

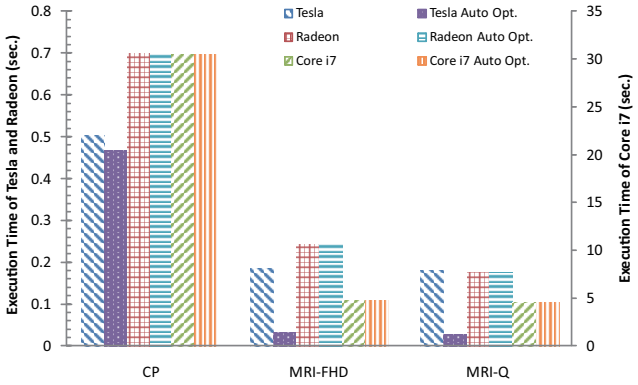

Fig. 8. Comparison among NVIDIA GPU, AMD GPU, and CPU.

rable with those of CUDA programs when the OpenCL C compiler have further matured. In this case, CUDA programs can easily be translated into OpenCL programs by replacing the keywords and API functions with the corresponding ones. Only by such a simple translation, we can benefit from OpenCL's features, reusing existing CUDA programs.

4 Performance Analysis of OpenCL on Various Compute Devices

4.1 Performance Comparison among Various Compute Devices

This section investigates the sustained performance of different compute devices by using OpenCL programs in order to clarify whether a single OpenCL code can efficiently run on various compute devices.

To evaluate the sustained performances of different compute devices, NVIDIA Tesla C1060 (Tesla), AMD Radeon HD 5870 (Radeon), and Intel Core i7 920 (Core i7) are examined in the following evaluation. The other experimental conditions are the same in Section 3. NVIDIA's OpenCL implementation is used for Tesla, and AMD's implementation is used for the others[4]. The sustained performance of each compute device is evaluated using OpenCL-version `CP`, `MRI-FHD`, and `MRI-Q`. As OpenCL is a vendor-independent programming framework, fair comparisons among compute devices of multiple vendors can be conducted.

Figure 8 shows the evaluation results. The horizontal axis indicates the benchmark names, the left vertical axis shows the execution times of Tesla and Radeon, and. the right vertical axis shows the execution time of Core i7. In each compute device, the execution times without the compiler option and `-cl-fast-relaxed-math` option are measured. These results show that the sustained performances of the GPUs are about 20 to 183 times higher than those

[4] AMD's implementation can use a CPU as a compute device as well as a GPU.

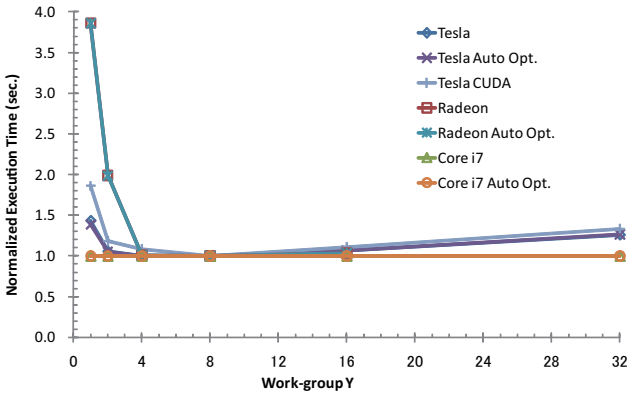

Fig. 9. Normalized time of `CP` in various work-group size.

of the CPU. This is because these programs involve massive data parallelism and regular memory access patterns, resulting in efficient use of many processing elements in GPUs without waiting for memory operations [5]. In comparison between the two GPUs, Tesla outperforlms Radeon for `CP` and `MRI-FHD`, while it does not for `MRI-Q`. These results indicate that their performance differences depend on the applications.

Furthermore, the execution times with the compiler optimization option are shorter than those with no compiler option in Tesla and Core i7. Thus, for Tesla and Core i7, the automatic optimizations by the OpenCL C compiler is effective to improve the performance. However, in Radeon, the both execution times with the compiler optimization option and with no compiler option are almost the same. The automatic optimizations by the OpenCL C compiler does not work for Radeon. From these results, it is obvious that the OpenCL C compiler is still under development in terms of the compiler optimization. Therefore, there is room to improve the performance by maturing the OpenCL C compiler.

Figures 9, 10, and 11 show the sustained performance of each compute device for `CP`, `MRI-FHD`, and `MRI-Q`, respectively. In the figures, the sustained performance is measured by changing the work-group size to investigate the performance sensitivity of each compute device to the work-group size. The work-group size should be large enough to hide the memory latency, and a small work-group size results in decreasing the number of active threads. However, if the work-group size is too large, the number of active threads decreases due to the lack of hardware resources such as the shared memory and registers in a GPU. In each compute device, the execution times with no compiler option and `-cl-fast-relaxed-math` option are measured. In each figure, the horizontal axis indicates the work-group size, and the vertical axis shows the normalized execution time, which is the kernel execution time of each parameter configuration normalized by the execution time of the best parameter configuration. In the case of `CP`, although the work-group has a two-dimensional index space

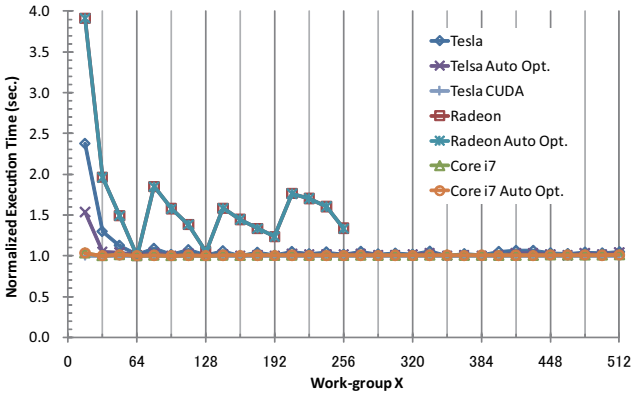

Fig. 10. Normalized time of `MRI-FHD` in various work-group size.

(x, y), the kernel code assumes $x = 16$ and hence only y is changed from 1 to 32 in the evaluation. In `MRI-FHD` and `MRI-Q`, the work-groups are one-dimensional. Hence, the work-group sizes are changed from 16 to the maximum number of work-items. In `MRI-FHD`, the maximum number is 256. In `MRI-Q`, it is 512 for Tesla and 256 for Radeon.

These figures show that the sustained performance of the CPU in every benchmark does not depend on the work-group size. On the other hand, the work-group size obviously affects the sustained performances of the GPUs irrespective of the compiler options.

Figure 9 shows that the sustained performance of `CP` increases until $y = 4$. As the work-group size grows, the number of work-items, i.e. threads, increases. As a result, the sustained performance improves, because executing a number of work-items is necessary to hide the memory access latency and to keep many processing elements busy. However, the larger the work-group size, the fewer the number of work-groups for the kernel execution. Moreover, as all work-items in a work-group share hardware resources of a MP, the number of work-groups assigned to one MP decreases in the case of the hardware resource shortage; some of work-groups are sequentially executed if the work-group size is too large to assign all work-groups to MPs at once. As a result, the number of work-groups running in parallel decreases, which results in the performance degradation. Thus, the sustained performance of `CP` decreases for a too large work-group size.

Figures 10 and 11 show that the performances of `MRI-FHD` and `MRI-Q` are more sensitive to the work-group size. The sustained performances degrade for a small work-group size, i.e. less than 32 for Tesla and 64 for Radeon due to the same reason in the case of `CP`. In addition, the sustained performance for `MRI-FDH` increases if the work-group size is a multiple of 32 for Tesla, and of 64 for Radeon. In the case of Tesla, 32 threads in a warp simultaneously execute the same instruction in a single-instruction multiple-data (SIMD) manner. Similarly, in the case of Radeon, 64 threads are packed into a wavefront and run in a SIMD

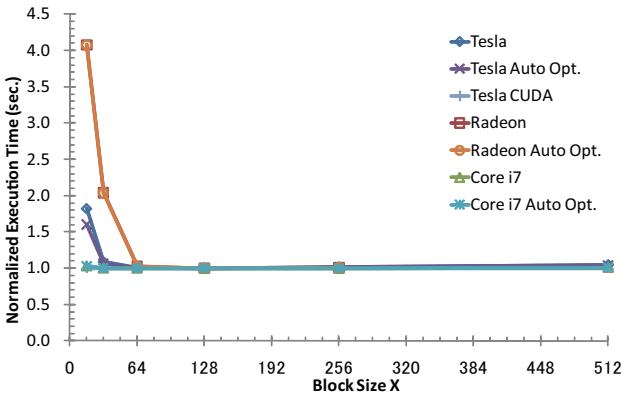

Fig. 11. Normalized time of `MRI-Q` in various work-group size.

manner. Hence, GPUs efficiently run if the work-group size is a multiple of their SIMD execution size. The results in Figures 10 and 11 coincide with these hardware specific behaviors.

In order to compare the sustained performance between Tesla and Radeon, Figure 12 shows the kernel execution time of each compute device for `MRI-Q`. This figure shows that the optimal work-group size does not depend on the compiler option, although automatic optimization can significantly improve the sustained performance. In performance comparison between Tesla and Radeon for the non-optimized programs, the faster GPU may change depending on the work-group size. If the work-group size is larger than 144, Radeon works faster than Tesla. On the other hand, if the work-group size becomes small, Tesla outperforms Radeon. Accordingly, we have to adjust the work-group size for individual GPUs, even though OpenCL allows programmers to access various GPUs in a unified fashion.

4.2 Automatic Tuning for OpenCL among Various Compute Devices

From the results in Section 4.1, it is demonstrated that the sustained performance of an OpenCL program drastically changes, depending on various factors such as compute devices, compiler optimization options, and configuration parameters for kernel execution. Since the optimal configuration of some parameters, such as the work-group size, obviously depends on individual applications and GPU devices, runtime automatic tuning would be required to enable a single OpenCL program to run efficiently on various GPUs. However, an application code may be written assuming a specific parameter configuration, e.g. CP assumes that the width of the work-group size is 16. Therefore, automating tuning of execution parameters needs to ensure that the code still works correctly after changing the parameters.

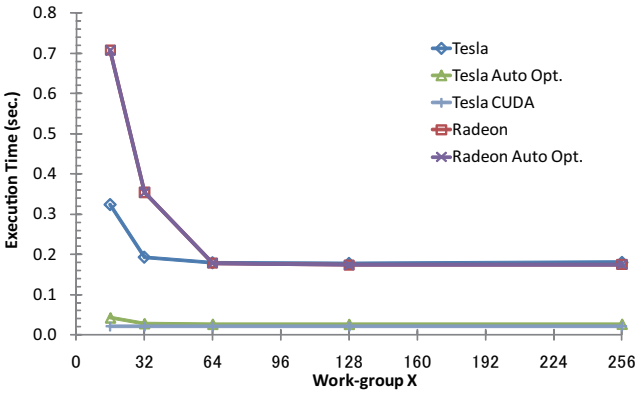

Fig. 12. Execution time of `MRI-Q` in various work-group size.

We are planning to explore the automatic performance tuning methodology to enhance the performance portability of OpenCL applications. By exploring the parameter space automatically based on profiling, the optimal parameter configuration can effectively be predicted. This automatic tuning methodology helps to exploit the potential of GPU computation.

5 Conclusions

This paper has discussed the sustained performance of OpenCL programs in comparison with CUDA. The quantitative evaluation results indicate that the sustained performance of every OpenCL program with the default mode is much lower than that of the equivalent CUDA program. To clarify the reason the performance differences between OpenCL program and CUDA one, this paper has also analysis, and pointed out that the performance differences come from the difference in the compiler optimization capabilities. By manually optimizing the kernel codes of the OpenCL programs and/or using automatic optimizations in the OpenCL C compiler, their sustained performances become almost the same as the performances of the CUDA ones.

In addition, as OpenCL enables us to make a fair comparison between various GPUs, this paper has compared an NVIDIA GPU with an AMD GPU in terms of sustained performance for executing the same programs. The evaluation results indicate that the optimization option of the OpenCL C compiler and the work-group size need to be adjusted for each GPU to obtain the best performance. Several other optimization factors would also depend on individual GPUs. Therefore, although OpenCL allows us to access various compute devices in a unified manner, we have to further optimize the code for each of those devices. We are planning to explore the automatic performance tuning methodology based on profiling to enhance the performance portability of OpenCL applications.

Acknowledgments

The authors would like to thank to the reviewers for their thoughtful review and helpful comments. This research was partially supported by Grant-in-Aid for Scientific Research (S) #21226018; Grant-in-Aid for Young Scientists(B) #21700049; Grant-in-Aid for Scientific Research (B) #21300007; NAKAYAMA HAYAO Foundation for Science & Technology and Culture; Core Research of Evolutional Science and Technology of Japan Science and Technology agency (JST); and Excellent Young Researcher Overseas Visit Program of Japan Society for the Promotion of Science (JSPS). The authors would like to acknowledge support from the Tohoku University Global COE Program on World Center of Education and Research for Transdisciplinary Flow Dynamics.

References

1. John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
2. N.K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *the 2006 ACM/IEEE conference on Supercomputing (SC06)*, November 2006.
3. Ian Buck et al. Gpubench: Evaluating gpu performance for numerical and scientic applications. In *2004 ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–20, 2004.
4. S. Che, J. Meng, J. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors. In *The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
5. Wen-Mei W. Hwu, Christopher Rodrigues, Shane Ryoo, and John Stratton. Compute unified device architecture application suitability. *Computing in Science and Engineering*, 11(3):16–26, 2009.
6. Hiroyuki Takizawa and Hiroaki Kobayashi. Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. *The Journal of Supercomputing*, 38(3):219–234, 2006.
7. NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture programming guide 3.0*, 2010. `http://developer.nvidia.com/object/cuda.html`.
8. AMD Corporation. *ATI STREAM ATI Stream Computing User Guide version 1.4 beta*, April 2009.
9. The Khronos OpenCL Working Group. *The OepnCL Specification version 1.0*, 2008. `http://www.khronos.org/oepncl/`.
10. Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 3.0 and 3.1.* Addison-Wesley Professional, 7th edition, 2009.
11. Microsoft Corporation. *DirectX*. `http://www.microsoft.com/windows/directx/`.
12. AMD Corporation. *ATI STREAM SDK v2.01*, February 2010.
13. NVIDIA Corporation. PTX : Parallel Tread Execution ISA Version 1.4, 2009.
14. The IMPACT Research Group. Perboil Benchmark suite . `http://impact.crhc.illinois.edu/parboil.php`.
15. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2nd edition, 2007.