2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip

A Comparison of Performance Tunabilities between OpenCL and OpenACC

Makoto Sugawara*, Shoichi Hirasawa*†, Kazuhiko Komatsu*†, Hiroyuki Takizawa*†‡, and Hiroaki Kobayashi*
* Tohoku University, Sendai, Miyagi 980-8579, Japan
† Japan Science and Technology Agency, CREST
‡ E-mail: tacky@isc.tohoku.ac.jp

***Abstract*—To design and develop any autotuning mechanisms for OpenACC, it is important to clarify the differences between conventional GPU programming models and OpenACC in terms of available programming and tuning techniques, called *performance tunabilities*. This paper hence discusses the performance tunabilities of OpenACC and OpenCL. As OpenACC cannot synchronize threads running on GPUs, some important techniques are not available to OpenACC. Therefore, we also design an additional compiler directive for thread synchronization. Evaluation results show that both OpenCL and OpenACC need architecture-aware optimizations, and similar approaches to performance optimization are effective for both OpenCL and OpenACC. The additional directive can allow OpenACC to describe more tuning techniques in the same approach as OpenCL. As it is obvious that OpenACC is more productive than OpenCL especially for legacy application migration, OpenACC is a very promising programming model if it can achieve the same performance as the conventional GPU programming models such as CUDA and OpenCL.**

I. Introduction

Today, high-performance computing systems are equipped with accelerators in addition to general-purpose processors. Especially, Graphics Processing Units(GPUs) are powerful accelerators with high floating-point operation rates and power efficiency [1]. Therefore, there is a strong demand for using GPUs to accelerate numerical simulations in the scientific and engineering fields.

So far, use of GPUs had forced application developers to rewrite their existing codes using special programming models, called *GPU programming models*, such as Compute Unified Device Architecture(CUDA)[2] and Open Computing Language(OpenCL)[3]. Since practical application codes are massive, complex and often messy, it is hard to rewrite the existing codes from scratch by using the GPU programming models.

One promising approach is to use compiler directives for a compiler to generate a part of codes that manage accelerators. OpenACC[4] is the open standard of such compiler directives. Using OpenACC, application developers can migrate their applications to heterogeneous computing systems mainly by inserting several compiler directives into the existing codes. As a result, the approach can remarkably reduce the programming efforts required for the migration.

Although our final goal is to automate the performance tuning of OpenACC applications by developing an autotuning mechanism, before that, we need to understand the performance tuning techniques available in OpenACC programming. Various programming and tuning techniques have been reported for CUDA and OpenCL[5][6]. However, it is still unknown that all the techniques are available and effective in the OpenACC programming. If OpenACC is unable to express a certain technique, the performance impact due to the inability should be clarified. In this work, we discuss the differences between OpenCL and OpenACC in terms of available programming and tuning techniques, called *performance tunabilities*. Then, we also design an additional compiler directive to allow OpenACC to describe the same tuning techniques as OpenCL.

II. GPU Programming Models and Performance Tuning

A. OpenCL and OpenACC

Since processing elements are hierarchically organized in many accelerators, OpenCL and OpenACC adopt similar execution models for the hierarchical parallel processing, although their terminologies are different. Accelerators such as GPUs are called *devices*, and the processors controlling the devices are called *hosts*. In both OpenCL and OpenACC, hosts and devices are supposed to have their own memory spaces, host memory and device memory, respectively.

Based on directives in the code, OpenACC compilers usually translate a C or Fortran code with OpenACC directives to a CUDA or OpenCL code. Through the translation, a kernel loop is hierarchically decomposed into many parallel tasks that can be executed in parallel using different kinds of parallelism: vectors, workers, and gangs. OpenACC compilers map the vectors, workers, and gangs to work-items and work-groups of OpenCL. The mapping between them depends on the OpenACC compiler. In this paper, the finest-grain parallelism expressed by vectors (or workers) in OpenACC and work-items in OpenCL is denoted as thread-level parallelism. Thus, kernel loops are executed by a large number of *threads* running on a GPU, and those threads are grouped into some *thread blocks*[1].

The most important difference between OpenCL and OpenACC is that OpenACC does not require rewriting of a kernel loop while OpenCL needs to rewrite the loop as a

[1]Those terms are from the CUDA terminology[2].

978-0-7695-5086-2/13 $26.00 © 2013 IEEE
DOI 10.1109/MCSoC.2013.31

kernel function. In OpenACC, the execution of a region of an application code is offloaded to the device if the `parallel` or `kernels` directive is given to the region. Figure 1 shows a sample code in OpenACC. In the code, the triple-nested loop is translated to a kernel function. i.e. device code, and executed by the device. Since the device code is automatically generated by the OpenACC compiler, the performance of the OpenACC code strongly depends on the compiler optimization.

Although OpenACC as well as OpenCL needs architecture- and system-aware optimizations to fully exploit the system performance, it is unknown that all the OpenCL programming techniques used for such optimizations are available in OpenACC. Therefore, in the following, we review those performance optimization techniques for OpenCL effective to a wide range of applications. Through the discussions on how to write those techniques in OpenACC, we clarify the performance optimization techniques that are unavailable in the OpenACC programming.

B. Performance optimization techniques for GPUs

Popular performance optimization techniques for GPUs are roughly classified into parallelism optimizations and data movement optimizations[7]. The former includes kernel execution parameter tuning. The latter includes the CPU-GPU data transfer avoiding, off-chip memory access optimization by coalescing, and effective use of on-chip memory.

1) Redundant Host-Device Data Transfer Avoiding: As the data transfer between host memory and device memory is very time-consuming, it can easily be the performance bottleneck of an application. Therefore, it is crucially important to avoid redundant data transfers in both OpenCL and OpenACC.

The data management in OpenACC is more automated than that in OpenCL. In OpenCL, all the data transfers between host memory and device memory must explicitly be written

```
1  ////////////////////////////////////////////////////////////
2  // SGEMM : C = alpha * AB + beta * C
3  // N is A's width , A's height and B's height
4  ////////////////////////////////////////////////////////////
5  void sgemm(float alpha, float, beta, float* A , float* B, float
       * C)
6  {
7    int i,j,l;
8    float ab;
9  #pragma acc data copyin(A, B), copy(C)
10   {
11 #pragma acc parallel num_gangs(N), vector_length(N)
12     {
13 #pragma acc loop gang
14       for( i = 0 ; i < N ; i++ ) {
15  #pragma acc loop vector
16         for( j = 0 ; j < N ; j++ ) {
17           ab = 0.0f;
18             for( l = 0 ; l < N ; l++ ){
19               ab += A[ i * N + l ]*B[ l * N + j] ;
20             }
21           C[ i * N + j ]= alpha * ab + beta * C[ i * N + j
                 ];
22         }
23       }
24     }
25   }
26 }
```

Fig. 1. Matrix multiplication with OpenACC directives.

by application programmers. On the other hand, an OpenACC compiler can automatically generate the codes for transferring data between host memory and device memory. However, it is likely that the code generated by the OpenACC compiler is conservative and performs many redundant data transfers. In this case, application developers can use their knowledge about the code to avoid such redundant data transfers. More specifically, if the generated code is not optimal, application developers can use the `data` directive and/or some clauses to give precise instructions on the data transfers to the OpenACC compiler.

By default, every array accessed in the `parallel` or `kernels` region is copied to the device memory in advance of kernel execution, and copied back to the host memory after the kernel execution. For example, in Figure 1, array `C` is copied twice, before and after the kernel execution. The `copy` clause explicitly indicates the data copies. However, arrays `A` and `B` are not updated in the kernel; they do not need to be copied back to the host memory. Thus, the `copyin` clause is used for `A` and `B` to prevent `A` and `B` from being copied back to the host memory. In this way, redundant data transfers are avoided using OpenACC directives and their clauses.

2) Memory Coalescing: In general, data accessed by a kernel are first transferred from the host memory to device's off-chip memory, called the *global memory*. The global memory bandwidth is high, but too low to match the GPU's peak floating-point operation rate. As a result, the global memory access can easily become the performance bottleneck of an application.

Furthermore, the sustained global memory bandwidth depends on the memory access pattern. Note that, in NVIDIA GPUs, 16 threads simultaneously access the global memory, which is decomposed into segments of 32, 64, and 128 bytes. If all the threads access data within one segment, their memory accesses are coalesced to organize one memory transaction. The coalesced memory access is more efficient than non-coalesced multiple accesses of individual threads, because the hardware of NVIDIA GPUs can access the global memory only by those segments. On the other hand, if the memory region accessed by the 16 thread is located across two segments, the GPU needs to fetch data of the two segments even if most of fetched data are not actually used in the kernel. Thus, misaligned memory access causes redundant data transfers. If the 16 threads simultaneously access more segments, the global memory access pattern becomes more wasteful. In the case of stride memory access, the memory addresses to be accessed by the 16 threads are widely distributed over multiple segments, and the GPU needs to fetch all the segments. As a result, the GPU cannot achieve a high effective bandwidth. Accordingly, it is always important to consider how each memory access in a kernel is translated to a memory access pattern of 16 threads.

One popular performance optimization technique is *data padding* to align the accessed memory region to the segment boundary. Especially, the memory accesses to a multi-dimensional array often lead to a misaligned memory access pattern. By data padding, therefore, the array is padded so that the array elements actually accessed are aligned to a segment

boundary. In both OpenCL and OpenACC, data padding is available and effective to improve the performance because the performance is limited by the sustained global memory bandwidth in most cases.

3) Kernel execution parameter tuning: In GPUs, a kernel is executed by threads in a work-sharing fashion. To achieve a high performance, moreover, GPUs hide the memory access latencies by concurrently running a huge number of threads[8]. Hence, the number of threads should basically be increased to exploit the thread-level parallel processing capability. However, the performance degrades if there are too many threads because all the threads execute the same kernel, a part of which might be unnecessary and redundant for some of the threads. Accordingly, the number of threads is one of important kernel execution parameters that needs to be tuned.

Various hardware resources of a GPU such as registers and on-chip memory spaces are allocated to each thread block, and shared by the threads in the same thread block. The threads within the same thread block can also synchronize if necessary. Therefore, if a kernel requires data sharing and/or frequent synchronization among threads, the number of threads in a thread block called the *thread block size* should be set to a larger value. However, larger thread blocks generally need more resources. Hence, fewer thread blocks can be executed at the same time on the GPU, and hence might lead to underutilization of the hardware resources. Therefore, the thread block size is also an important parameter to be carefully tuned.

In OpenCL, application developers are supposed to appropriately determine those kernel execution parameters. Similarly, in OpenACC, application developers can specify the thread count and the thread block size through some clauses of compiler directives such as the `parallel` and `loop` directives. Hence, in both OpenACC and OpenCL, application developers can adjust the kernel execution parameters.

However, for thread-level parallel execution of a loop with inter-iteration data dependency, threads must be synchronized before exchanging their data. Although the GPU hardware allows the threads within a thread block to synchronize, OpenACC does not provide any way to specify the thread synchronization. As a result, if there is dependency among iterations of a loop, OpenACC cannot easily parallelize the loop even if OpenCL can parallelize it. Consequently, such a loop might prevent OpenACC from using the optimal parameters, though the kernel execution parameters can be adjusted in both OpenCL and OpenACC. In this sense, the performance tunability of OpenACC is obviously lower than that of OpenCL.

4) Use of On-chip Memory: GPUs have small and fast on-chip memory spaces. In OpenCL, the on-chip memory space is called the *local memory space*, and application developers are responsible for explicitly and manually putting data on the local memory space. On the other hand, in OpenACC, data with the `cache` directive are automatically copied to the local memory space if possible. Using the on-chip memory, application developers can reduce the number of time-consuming off-chip memory accesses, resulting in performance improvement.

In OpenCL, multiple threads of a thread block work together to copy data from the global memory space to the local memory space. Barrier synchronization using a built-in function, `barrier()`, is performed to ensure that the data copy is finished hence the local memory data are available to any threads in the thread block. In OpenCL kernels using the local memory, thus, `barrier()` is called whenever the data copy from/to the local memory space is performed.

On the other hand, there is no OpenACC directive to specify such barrier synchronization of threads. Even though various performance optimization techniques require the local memory, OpenACC cannot explicitly use it. OpenACC just provides the `cache` directive as a hint for the OpenACC compiler to copy data from/to the local memory space. Although the compiler-generated code may not be optimal to a specific application, there is no standard way for application developers to revise it. Accordingly, due to the lack of thread synchronization support, the performance tunability of OpenACC is lower than that of OpenCL. Whenever performance tuning techniques using thread synchronization are effective, OpenCL potentially attains a higher performance than OpenACC.

III. A Directive for Improving the Performance Tunability of OpenACC

If the performance achieved by OpenACC is comparable with that by CUDA and OpenCL, OpenACC is obviously helpful to port the existing codes of large-scale simulations to heterogeneous computing systems. As reviewed in Section II, many important programming techniques used in OpenCL can be expressed by inserting OpenACC directives plus minor modification of the original code.

However, some techniques are not available in OpenACC because the current version of OpenACC does not provide any directives for explicitly synchronizing threads, even though not a few OpenCL applications are implemented using `barrier()` for the synchronization. For example, 22 out of 36 sample codes in NVIDIA SDK[9] use the built-in function. Therefore, this paper presents a compiler directive, `barrier`, which is translated to `barrier()` of OpenCL for barrier synchronization[2]. The directive enables application developers to explicitly specify barrier synchronization in the OpenACC programming and thereby improve the performance tunability of OpenACC.

The `barrier` directive explicitly specifies that threads in a thread block are blocked until all the threads reach the synchronization point. The directive is available only in a `parallel` or `kernels` region. A sample code of using the directive is shown in Figure 2. Using the `cache` directive, this code expects that a local memory region is allocated to array `sa`. Then, the data of array `a` in the global memory space are copied to `sa`. The data copy is done by multiple threads. After that, hence, barrier synchronization is performed to ensure that the data copy is finished and thus the local memory data are available. Then, kernel computation is executed using the local memory data. Finally, another barrier synchronization is

[2]A similar directive can be found in HMPPCG directives[10]. The purpose of using this directive is to discuss the usability of such a synchronization directive in the OpenACC programming.

```
1
2  #pragma acc parallel copy(a)
3  {
4  #pragma acc cache
5    float sa[N];
6  #pragma acc loop
7    for( i = 0 ; i < N ; i++ ) {
8      sa[i] = a[i];
9    }
10 #pragma acc barrier(gang) local
11   // processing the local memory data
12 #pragma acc barrier(gang) local
13
14 #pragma acc loop
15   for( i = 0 ; i < N ; i++ ) {
16     a[i] = sa[i];
17   }
18 }
```

Fig. 2. A sample code of the `barrier` directive.

done and the local memory data are copied back to the global memory space. Accordingly, by combining the `barrier` directive with the `cache` directive, OpenACC can use the local memory space in a similar way to OpenCL as mentioned in Section II-B4.

In this work, we prototyped a translator to use the `barrier` directive. The translator puts a special mark (comments) on the location of the `barrier` directive, and expects that the mark remains in the device code generated by some OpenACC compilers such as the CAPS compiler[11]. Then, the mark is replaced with OpenCL's `barrier()` function at runtime by implicitly hijacking some OpenCL API calls as in [12]. In this way, we have implemented the directive working with the commercial OpenACC compiler; the compiler is regarded as a black box. Since the `barrier()` function takes one argument, the `barrier` directive also has clauses, `local` and `global`, that correspond to `CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE` of OpenCL, respectively.

IV. Evaluation and Discussions

This section shows performance evaluation results to discuss the effects of OpenCL and OpenACC programming techniques on their performances. The system configurations are listed in Table I. The following evaluation uses three generations of NVIDIA GPUs: Tesla C1060, Tesla C2070, and Tesla K20. The PGI compiler and the CAPS compiler are used for the compilation. However, only the CAPS compiler is used for evaluating the `barrier` directive described in Section III because the current implementation of our translator does not work for the PGI compiler.

In this paper, the nanopowder growth simulation [13] is used as an example of real applications. In the simulation, the entire growth process of binary alloy nanopowders in thermal plasma synthesis is simulated considering various phenomena. The code is written in Fortran, and approximately 90% of the total execution time is spent for executing the `Coagulation` subroutine, which simulates that particles mutually collide and merge into larger particles. This routine involves high parallelism, and is potentially suited for parallel processing by GPUs. However, architecture-aware performance optimization of the subroutine is required to achieve a reasonable

TABLE I
System configurations.

	PC 0	PC 1	PC 2
CPU	Core i7 920	Core i7 920	Core i7 930
GPU	Tesla C1060	Tesla C2070	Tesla K20
Compiler	GCC 4.1.2	GCC 4.4.4	GCC 4.1.2
CAPS Compiler	3.3.0	3.3.0	3.3.0
PGI Compiler	12.10	12.10	12.10
OpenCL	1.0	1.1	1.1
CUDA	5.0	5.0	5.0
Compiler Option	O3	O3	O3

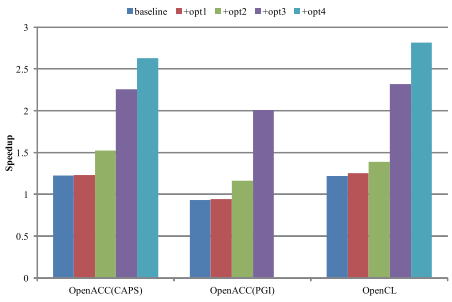

Fig. 3. Speedup ratios of Tesla C1060.

performance, because it requires a large memory size of 1.8 Gbytes, several kinds of memory access patterns, the reduction operation, and data locality awareness. Therefore, the nanopowder growth simulation is used to evaluate the effect of each performance optimization technique reviewed in Section II.

The `Coagulation` subroutine includes a deeply-nested loop. The nested loop can be decomposed into 2173 independent tasks, and each task has a reduction operation. A version of the simulation code, in which each independent task of the nested loop is executed by one thread of the GPU, is called the baseline version. In this evaluation, four optimization techniques listed in Table II are incrementally applied to the baseline version, and then their effects on the performance are discussed via their speedup ratios compared to the 4-thread execution performance of Core i7 920. Figures 3, 4, and 5 show the speedup ratios of Tesla C1060, C2070, and K20, respectively. In the figures, we do not intend to compare the two commercial compilers. The point in comparison of the two compilers is that the performance significantly changes depending on the compiler. Note that the PGI compiler used in the evaluation is not the latest version while the CAPS compiler is the latest one.

The first performance optimization, opt1, is to reduce the data transfers between CPU's host memory and GPU's device memory. Although the `Coagulation` subroutine accesses data of 1.8 Gbytes, most of the data are unchanged during the simulation. Hence, in opt1, those data are transferred only once, and the `data` directive is used to avoid redundant data transfers during the simulation. Since another subroutine, `Coefficient`, generates data of about 42 Mbytes from other data of about 40 Kbytes, the transferred data size is further reduced by executing the subroutine on the GPU. Although the

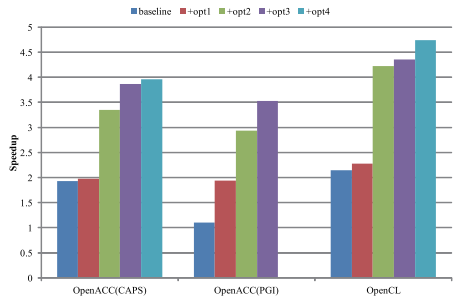

Fig. 4. Speedup ratios of Tesla C2070.

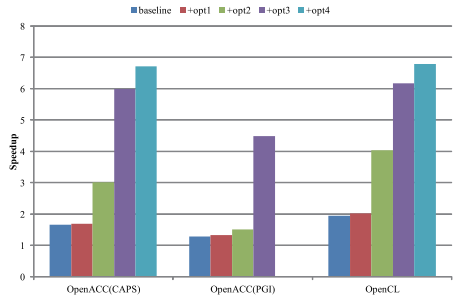

Fig. 5. Speedup ratios of Tesla K20.

GPU cannot accelerate the `Coefficient` subroutine at all and the GPU performance for this routine is almost the same as the CPU performance, a reduction in the transferred data size leads to a big performance gain in many cases. In our evaluation, the performance improvement is not remarkable because the kernel execution time is dominant in the total execution time. However, it is clear that both OpenCL and OpenACC can use opt1 and their performances (slightly) improve.

The second optimization, opt2, is to make the global memory access more efficient by using memory coalescing. In the `Coagulation` subroutine, elements of a 3-dimensional array `SOCO(I,J,K)` are sequentially accessed by each thread, i.e. `I` is the loop variable of the innermost loop in the thread. In this case, the data simultaneously accessed by 16 threads are widely distributed over many segments. The memory accesses are not coalesced, and thus GPUs cannot efficiently access the data. Therefore, the data layout of `SOCO(I,J,K)` is changed to `SOCO(K,J,I)` and data padding is also applied to the transposed array so that memory accesses of 16 threads are coalesced and performed as one memory transaction accessing a continuous memory region aligned to a segment boundary. This optimization, opt2, can significantly improve the sustained memory bandwidth and hence the performance. Figure 6 clearly shows that a lot of global memory accesses are coalesced to be fewer memory transactions by data padding and aligning. This performance optimization technique is available to both OpenCL and OpenACC. However, it should be noted that the original code is considerably modified by opt2 even in the case of OpenACC.

In the third optimization, opt3, the thread count and the thread block size are tuned to improve the performance. Note

TABLE II
FIVE VERSIONS OF THE NANOPOWDER GROWTH SIMULATION CODE.

	Version Name
baseline	Reference(naive implementation)
+opt1	CPU-GPU data transfer avoiding
+opt2	Data padding for memory coalescing
+opt3	Kernel execution parameter tuning
+opt4	Use of local memory

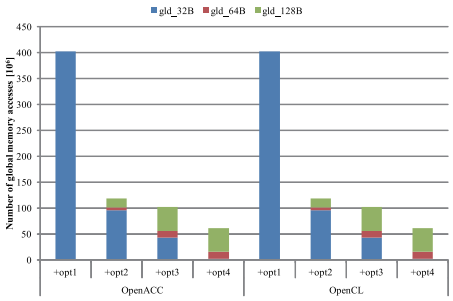

Fig. 6. Global memory accesses in Tesla C1060.

that, in the baseline version, the kernel loop is decomposed into completely-independent tasks, each of which is assigned to a thread. However, each task has a reduction operation. Hence, if the reduction operation can be parallelized, more threads can be used for the kernel execution to improve the performance.

In OpenCL, a reduction operation is manually coded with several advanced programming techniques[14]. On the other hand, in OpenACC, a reduction operation can easily be written using the `reduction` clause. Using the reduction clause, OpenACC can parallelize loops with reduction operations in addition to loops without any inter-iteration dependencies. As a result, OpenACC can adjust the number of threads more flexibly. Therefore, in opt3, the thread count and the thread block size are adjusted so as to increase the occupancy given by NVIDIA's OpenCL profiler. For each GPU, the thread block size with the maximum occupancy is used in opt3. Since opt3 changes the memory access pattern of the kernel, data padding and aligning are again applied so as to improve the memory access performance of the optimized kernel.

The forth optimization, opt4, uses the local memory for caching reusable data. In the `Coagulation` subroutine, all the threads access the same data. Once the data are fetched to the local memory space, the data can be reused several times by different threads. In the case of not using the local memory, the data are always read from the global memory. On the other hand, when the local memory is used, the data are read from the global memory to the local memory only once, and then the data in the local memory are used for the computation. As a result, opt4 can reduce the access latency to the data and also the number of global memory accesses.

For using the local memory, barrier synchronization is required to ensure that all threads finish reading the data from the global memory to the local memory. In OpenCL, a built-in function of `barrier()` is used for the synchronization. However, the standard OpenACC specification does not pro-

TABLE III
THE NUMBER OF LINES FOR NANOPOWDER GROWTH SIMULATION.

	OpenACC		OpenCL		OpenMP
baseline	2979	(8)	3464	(487)	2977
+opt1	2982	(8)	3570	(108)	-
+opt2	2986	(33)	3570	(46)	-
+opt3	2991	(50)	3615	(80)	-
+opt4	2997	(7)	3620	(6)	-

vide the synchronization, and hence is not able to use opt4. The `barrier` directive allows the OpenACC programming to use the synchronization. As a result, it can significantly improve the performance of an OpenACC application as shown in Figures 3, 4, and 5.

The performance difference between OpenACC and OpenCL still remains even if opt4 is applied to the OpenACC version. This is mainly because the reduction operation implementation generated by simply using the OpenACC's `reduction` clause is slower than the parallel reduction implementation written in OpenCL[14]. As OpenACC compilers are still under active development, this performance problem will be reduced by maturing OpenACC compilers.

Finally, the productivities of OpenCL and OpenACC are compared based on the number of code lines shown in Table III. The numbers specified in parentheses are the numbers of code lines modified for each optimization. The OpenACC baseline version requires only eight compiler directives to achieve the comparable performance with OpenCL. Since the original code is written in Fortran, OpenCL needs to rewrite a part of the code in C and 286 lines are for redefinition of variables in the C code. This means that, in the case of porting a legacy Fortran program to a heterogeneous computing system, OpenCL requires rewriting a considerable part of the code. Accordingly, it is clear that OpenACC is more productive than OpenCL especially for legacy application migration.

V. CONCLUSIONS

In this paper, we have first evaluated the performance tunabilities of OpenACC and OpenCL. Although most programming optimization techniques are available in both OpenCL and OpenACC, some important techniques are not available because OpenACC does not have any method to synchronize threads. Therefore, we have also presented an additional compiler directive, the `barrier` directive, to explicitly specify thread synchronization and thereby improve the performance tunability of OpenACC.

Our evaluation results suggest a conclusion that both OpenCL and OpenACC need architecture-aware optimizations and thus we can basically take a similar approach whichever programming model is used. The discussions on performance would come to the same conclusion even if CUDA [2] instead of OpenCL is used for the comparison, because it has been reported that there is no reason for OpenCL to obtain worse performance than CUDA under a fair comparison [15]. As it is obvious that OpenACC is more productive than OpenCL and CUDA especially for legacy application migration, OpenACC is a very promising programming model if it has the same performance tunability as those conventional GPU programming models.

Even in the case of OpenACC, we need to modify the original source code to achieve a high performance, and hence to reduce the code portability. The OpenACC programming is never just to insert compiler directives. The code and performance portabilities are further sacrificed by modifying the code to efficiently use the synchronization directive because thread synchronization is not required in sequential execution and other many environments except for GPUs. In our future work, therefore, we will develop an autotuning mechanism that can improve the performance tunability of OpenACC in a more portable way.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Mayasa Shigeta of Tohoku University for allowing us to use his simulation code in the performance evaluation. The authors would also like to thank the RIKEN Integrated Cluster of Clusters (RICC) at RIKEN for the user supports and the computer resources used for the performance evaluation.

This research was partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems" and Grant-in-Aid for Scientific Research(B) #25280041

REFERENCES

[1] O. Schenk, M. Christen, and H. Burkhart, "Algorithmic performance studies on graphics processing units," *Journal of Parallel Distribbuted Compututing*, vol. 68, no. 10, pp. 1360–1369, Oct. 2008.
[2] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide v5.0*, October 2012.
[3] Khronos OpenCL Working Group, *The OpenCL Specification Version 1.2*, November 2011. [Online]. Available: http://www.khronos.org/
[4] *The OpenACC Application Programming Interface Version 1.0*, 2011. [Online]. Available: http://openacc.org
[5] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08, 2008, pp. 73–82.
[6] K. Komatsu, T. Soga, R. Egawa, H. Takizawa, H. Kobayashi, S. Takahashi, D. Sasaki, and K. Nakahashi, "Parallel processing of the building-cube method on the GPU platform," *Computers & Fluids Special Issue "22nd International Conference on Parallel Computational Fluid Dynamics"*, vol. 45, no. 1, pp. 122–128, June 2011.
[7] NVIDIA Corporation, *OpenCL Best Practices Guide*, 2011.
[8] ——, *OpenCL Programming for the CUDA Architecture Version 4.2*, 2012.
[9] ——, "NVIDIA OpenCL SDK Code Samples," 2012.
[10] CAPS entreprise, *CAPSCompilers-3.3 HMPPCG Directives Reference-Manual*, 2012.
[11] ——, *CAPSCompilers-3.3 OpenACC-ReferenceManual*, 2012.
[12] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "CheCL: Transparent checkpointing and process migration of OpenCL applications," in *Proceedings of IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, May 2011, pp. 864–876.
[13] M. Shigeta and T. Watanabe, "Growth model of binary alloy nanopowders for thermal plasma synthesis," *Journal of Applied Physics*, vol. 108, no. 4, pp. 043 306–1 – 043 306–15, Aug 2010.
[14] M. Harris, *Optimizing Parallel Reductiion in CUDA*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/index.html
[15] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *Proceedings of 2011 International Conference on Parallel Processing*, September 2011, pp. 216–225.