Prototyping an OpenCL Implementation for NEC SX Vector Systems

Hiroyuki Takizawa Ryusuke Egawa Hiroaki Kobayashi

Tohoku University
{tacky, egawa, koba}@isc.tohoku.ac.jp

Abstract

OpenCL is a new programming specification whose current implementations are mostly used for high-performance computing with graphics processing units(GPUs), so-called *GPU computing.* However, the OpenCL specification itself is not specialized for GPU computing. In this paper, therefore, we propose to use the OpenCL specification to describe the collaborative work of scalar systems and an NEC SX vector supercomputing system. Since there is no OpenCL implementation for the SX systems, we translate a part of an OpenCL code written in OpenCL C to a standard C++ code. After the translation, the generated code is compiled with a native SX C++ compiler so as to produce an executable program that runs on the SX system. This paper shows a prototype implementation of an OpenCL-to-C translator to evaluate the collaborative work of scalar systems and the SX systems according to the OpenCL specification. The evaluation results indicate that the performance of an SMP node of the SX system approaches the performance of a single GPU by manually tuning the code to improve the vectorization ratio. In addition, the SMP node can be used as an accelerator with a huge memory space. As data parallelism is explicitly described in an OpenCL C code, the performance of the code generated by the OpenCL-to-C translator is scalable with the number of SX processors.

1 Introduction

NEC SX vector supercomputing systems can provide excellent performance for a vast variety of large-scale scientific and engineering applications[5] because the computationally-expensive parts of those applications called *kernels* usually involve massive data parallelism with regular memory access patterns and are hence vectorization-friendly. However, if a part of the application code is totally unable to be vectorized, it might severely degrade the computational efficiency of the SX system. In general, vector processors are efficient for fine-grain data parallel workloads involving loop-level parallelism, while scalar processors are suited to sequential workloads.

In practical uses, a real application is often a mixture of sequential workloads and parallel workloads. To achieve efficient execution of such an application, therefore, we need to explore an effective way to collaborative work of scalar and vector systems. One important research topic is how to describe the collaboration of those systems. So far, a standard way to program their collaboration has not been established yet.

In addition to general-purpose scalar processors (CPUs), graphics processing units (GPUs) are recently used as accelerators for fine-grain data-parallel workloads. High-performance computing with GPUs is called *GPU computing.* Since a GPU is originally a special-purpose processor for accelerating computer graphics applications but is not a general-purpose processor, an external CPU fully controls the GPU for computation. Therefore, programming frameworks for GPU computing, such as CUDA[3] and OpenCL[8], are by nature designed to achieve effective collaboration of CPUs and GPUs.

The main idea of this paper is to use the GPU programming framework to describe the collaborative work of CPUs and SX systems. As with an SX system, GPUs are used for data-parallel workloads, and therefore the programming model for GPU computing would also be effective for the collaboration of CPUs and the SX systems. This paper shows our ongoing work to achieve efficient collaboration of CPUs and SX systems, showing a prototype implementation of OpenCL for SX and its early evaluation results.

The rest of this paper is organized as follows. Section 2 briefly reviews OpenCL. Then, Section 3 proposes an OpenCL implementation for SX systems and describes how the OpenCL specification is used for collaboration of CPUs and an SX system. Section 4 shows the feasibility of the implementation and discusses the performance gain. Finally. Section 5 gives concluding remarks of this paper.

2 OpenCL

OpenCL is a new programming standard for heterogeneous computing of different processors. Using OpenCL, a programmer can access various processors in a unified way. OpenCL assumes that a CPU works as a *host* to control a *compute device* such as a GPU.

A host and a compute device have their own memory spaces, host memory and device memory, respectively. A typical OpenCL application running on a host first initializes a compute device, allocates device memory chunks, copies host memory data to the allocated chunks, invokes a special "kernel" function to process the device memory data, and retrieves the computation results from the device memory. Only the kernel function is executed on the compute device, and the others are initiated by the host. In this way, a host and a compute device collaborate to run an OpenCL application.

A host code is written in standard programming languages such as C and C++. On the other hand, a device code of kernel functions offloaded to a compute device is written in a special language, called the OpenCL C language, which is C99 with some additional keywords and a lot of builtin functions. There are OpenCL API functions for a host code to dynamically load and compile a device code at runtime.

A host and a compute device asynchronously work, and the memory consistency between them are guaranteed only at their synchronization points. Thus, a host and a compute device are loosely coupled to run an OpenCL application and we can easily implement a wrapper library that transparently intercepts API calls to implicitly support advanced features such as transparent checkpointing[7]. Due to the loose collaboration nature of OpenCL, some mechanisms capable of transparently using remote compute devices have been recently developed [2, 1]. In such mechanisms, a host controls remote compute devices via TCP/IP network communication; the messages sent from a host are passed to the OpenCL implementation, i.e. OpenCL library, installed on a remote PC. As a result, the host can indirectly use the remote compute device. However, since there is no OpenCL implementation for SX systems, those existing mechanisms cannot achieve collaboration of scalar and SX systems.

3 OpenCL for SX

As reviewed in the previous section, there already exist several OpenCL implementations that enable collaboration among distinct PCs. However, they need an OpenCL implementation to be installed on each PC, because API calls on a host PC are just forwarded to the OpenCL implementation on remote PCs.

This section proposes an OpenCL implementation for effective collaboration between scalar and SX systems. As with Virtual OpenCL[2] and Hybrid OpenCL[1], the proposed implementation also assumes that all OpenCL API calls are forwarded to a remote SMP node of the SX system as shown in Figure 1. As a result, it allows scalar systems to remotely use the SMP node via OpenCL API calls.

The most important feature of the implementation is to translate a device code written in OpenCL C into a standard C++ code, called OpenCL-to-C translation. As a result of the translation, even the SX system that does not have any OpenCL implementation can run the translated code, and kernel execution of computationally-expensive data parallel workloads can be offloaded to the SX system from a scalar system.

In OpenCL C, the kernel part of an application is described as a kernel function. Once the kernel function is invoked, a bunch of threads are generated, and every thread executes the same function with different thread IDs for fine-grain data-parallel processing. On the other hand, in the stan-

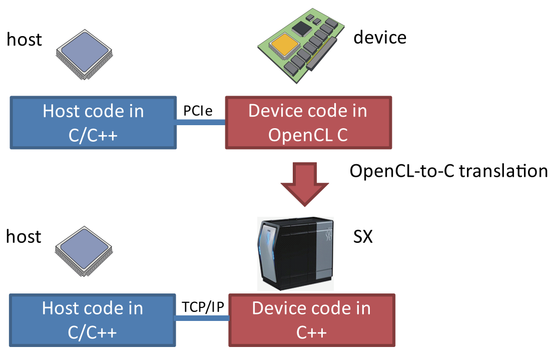
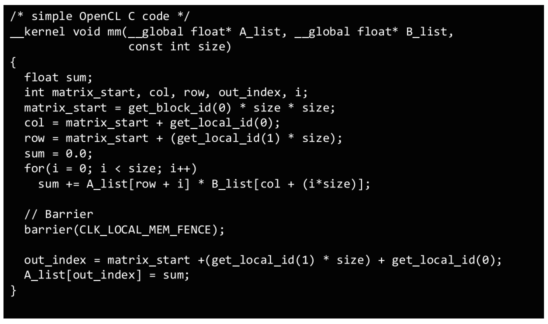


Figure 1: Overview of the vector-scalar collaboration using OpenCL.

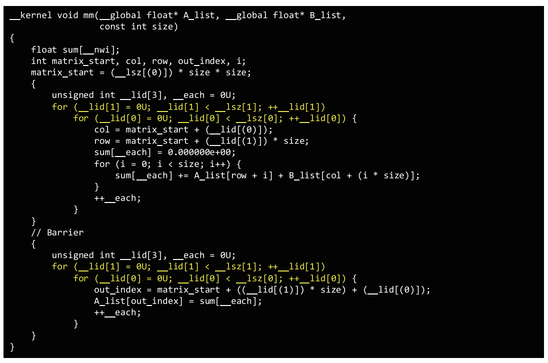


```
/* simple OpenCL C code */
__kernel void mm(__global float* A_list, __global float* B_list,
                 const int size)
{
  float sum;
  int matrix_start, col, row, out_index, i;
  matrix_start = get_block_id(0) * size * size;
  col = matrix_start + get_local_id(0);
  row = matrix_start + (get_local_id(1) * size);
  sum = 0.0;
  for(i = 0; i < size; i++)
    sum += A_list[row + i] * B_list[col + (i*size)];

  // Barrier
  barrier(CLK_LOCAL_MEM_FENCE);

  out_index = matrix_start +(get_local_id(1) * size) + get_local_id(0);
  A_list[out_index] = sum;
}
```

(a) An OpenCL C code.



```
__kernel void mm(__global float* A_list, __global float* B_list,
                 const int size)
{
    float sum[__nwi];
    int matrix_start, col, row, out_index, i;
    matrix_start = (__lsz[(0)]) * size * size;
    {
        unsigned int __lid[3], __each = 0U;
        for (__lid[1] = 0U; __lid[1] < __lsz[1]; ++__lid[1])
            for (__lid[0] = 0U; __lid[0] < __lsz[0]; ++__lid[0]) {
                col = matrix_start + (__lid[(0)]);
                row = matrix_start + (__lid[(1)]) * size;
                sum[__each] = 0.000000e+00;
                for (i = 0; i < size; i++) {
                    sum[__each] += A_list[row + i] + B_list[col + (i * size)];
                }
                ++__each;
            }
    }
    // Barrier
    {
        unsigned int __lid[3], __each = 0U;
        for (__lid[1] = 0U; __lid[1] < __lsz[1]; ++__lid[1])
            for (__lid[0] = 0U; __lid[0] < __lsz[0]; ++__lid[0]) {
                out_index = matrix_start + ((__lid[(1)]) * size) + (__lid[(0)]);
                A_list[out_index] = sum[__each];
                ++__each;
            }
    }
}
```

(b) An automatically-generated C++ code.

Figure 2: An example of OpenCL-to-C translation.

dard C/C++, the kernel part is usually described as a loop. Therefore, in OpenCL-to-C translation, a kernel function is converted into a kernel loop, called a *thread loop*.

One major difficulty in the translation is how to deal with barrier synchronizations. If there is a barrier synchronization in a kernel function, the kernel must be partitioned and converted into two thread loops. If each thread has its own value, we have to propagate the value from the preceding thread loop to the subsequent thread loop. Accordingly, variable analysis is required to find thread-dependent variables.

The programming model translation from CUDA to C has been proposed by the IMPACT group of the university of Illinois at Urbana-Champaign [6]. Our OpenCL-to-C translator uses a simplified version of their CUDA-to-C translation. An abstract syntax tree (AST) of the source code is built in advance of the OpenCL-to-C translation. Then, the translation procedure can be briefly summarized as follows:

1. The AST is first traversed to build a parent-child map of the tree elements.

2. Using the parent-child map, function inlining is applied to all function calls in every kernel function.

3. The AST is traversed again for variable analysis or data dependency analysis to find thread-dependent variables.

4. The kernel function is translated to one or more thread loops.

Figure 2 illustrates an example of converting a kernel function body to two thread loops. First, the kernel is partitioned because there is a barrier synchronization in the kernel. After partitioning, `for` statements are appropriately inserted so that each partition is surrounded by a `for` loop. An array is automatically declared and used if there is a thread-dependent variable that lives across multiple thread loops. Based on the variable analysis, reference to a thread-dependent variable is replaced with that to an element of the array. For example, `sum` in Figure 2(a) is such a variable and hence replaced with an array element, `sum[__each]` in Figure 2(b). The array must keep all thread-specific values of a thread block, and the array size is thus the same as the thread block size determined at the kernel invocation.

A more detailed description of the translation can be found in [6].

Another possible (and more straightforward) approach might be to develop an OpenCL C compiler that can produce a binary code of SX systems from an OpenCL C code. However, this work does not employ the approach, because the translation of kernel functions to kernel loops is anyhow required even in the case of developing such a compiler.

4 Early Evaluation and Discussions

This section shows some early evaluation results to demonstrate the performance of the SX system for an OpenCL program. We have developed a prototype implementation of OpenCL for SX that consists of an OpenCL runtime library and an offline OpenCL-to-C translator. Just-in-time compilation of device codes is not supported, and a device code is thus converted to a C++ code in advance of the execution.

Three benchmark programs written in CUDA, `CP`, `MRI-Q`, `MRI-FHD`, are selected from the Parboil benchmark suite and manually ported to OpenCL. Then, they are converted into standard C++ codes by using our OpenCL-to-C translator.

Figure 3 shows the performance evaluation results for the `CP` program. In the figure, the left-side axis and the right-side axis represent the execution time and the scalability, respectively. The execution time of a single GPU of NVIDIA Tesla C1060 is also shown for comparison.

In the kernel code, function `sqrt` is used to calculate the double-precision value of the squared root of a single-precision value that is stored in a single-precision variable. As a result, unnecessary type casts are inserted to a thread loop and inhibit the compiler from vectorizing the loop. In this case, the performance of the SX system is somewhat disappointing. It is widely known that the vectorization ratio is crucial for high efficiency of vector processors.

In the evaluation, the automatically-generated code is tuned manually so as to avoid those unnecessary type casts and thus to enhance the vectorization ratio. Due to this simple tuning, the performance of the SMP node of 16 processors is significantly improved and approaches the GPU performance, even though the original code is completely optimized for GPUs. Although OpenCL allows a programmer to access various compute devices in a unified manner, it does not mean that a single code can achieve the best performance of each device. Therefore, these results indicate that several automatic performance tuning capabilities to improve the vectorization ratio should be incorporated into the translator.

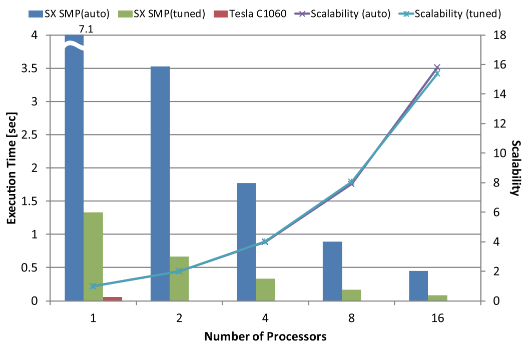


Figure 3: Performance and speedup ratio.

As data parallelism is explicitly described in an OpenCL C code, the performance of the generated code is almost perfectly scalable with the number of SX processors. An SMP node of the SX system has a huge memory space of 1TB. Hence, it can be beneficial to use the SX system instead of a GPU if an application program needs a large memory space. The SX system can be used as a compute device with a 1TB device memory.

Although the benchmark programs used in the evaluations are compute-intensive[4], use of the SX system will be also beneficial for memory-intensive applications, because its aggregated memory bandwidth of 4TB/s is much higher than those of any other compute devices. For example, the peak memory bandwidth of Tesla C1060 is 102GB/s, and 40 times less than the aggregated peak memory bandwidth of the SX SMP node. Therefore, we will evaluate our approach using memory-intensive applications in our future work.

5 Conclusions

This paper applied OpenCL to collaboration of scalar and vector systems. As there is no OpenCL implementation for the SX system, this paper has discussed OpenCL-to-C translation to enable an SX system to run a device code. A prototype implementation shows the feasibility of this approach. The performance evaluation results with the implementation indicate that an SMP node of the SX system can be used as an accelerator with a huge memory space. This may become a useful alternative of GPUs as compute devices because the memory size of a GPU is much smaller than the host memory size. Besides, since data parallelism is explicitly described in an OpenCL code, the performance of the translated code is scalable with the number of processors. The prototype implementation uses OpenMP directives to exploit the whole of an SMP node as a compute device. An SMP node of the SX system has up to 16 vector processors, and the OpenCL-to-C translation can benefit from the large SMP node.

However, this work is ongoing and there still remain at least three difficulties in using OpenCL for effective collaboration of scalar and SX systems. One is that the current SX operating system does not support dynamic linkage. Although a C++ code translated from a device code can be compiled at runtime, there is no way to dynamically link the object file with the host program. Therefore, a device code must be translated, compiled and linked with the host code in advance of the execution. Another difficulty is that an SX processor uses the big endian while CPUs usually use the little endian. Hence, the byte order of the host memory data must be changed before those are transferred to an SX system. The endian conversion is nontrivial because some API functions receive a pointer to the data of an unknown variable type and hence cannot determine the size of each data element in the address. The other difficulty is that the SX system may require performance tuning techniques different from GPUs. Although OpenCL allows a programmer to use various compute devices in a unified way, it does not mean that a single code can bring out the best performance of each device. Automatic performance tuning of OpenCL codes for each device is a challenging and important open problem.

Acknowledgments

The authors would like to thank Wenmei W. Hwu of University of Illinois at Urbana-Champaign for helpful comments on the OpenCL-to-C translation.

This research was done with the NEC SX-9 supercomputing system installed in Cyberscience Center, Tohoku University.

This research was partially supported by Grants-in-Aid for Young Scientists (B) #21700049; 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).

References

[1] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid opencl: Enhancing opencl for distributed processing. In *9th IEEE International Symposium on Parallel and Distributed Processing with Applications*, ISPA'11, pages 149–154, 2011.

[2] Amnon Barak, Tal Ben-Nun, Ely Levy, and Amnon Shiloh. A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC), IEEE Cluster 2010*, 2010.

[3] NVIDIA Corporation. CUDA Zone – The resource for CUDA developers.

[4] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-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*, PPoPP '08, pages 73–82, 2008.

[5] Takashi Soga, Akihiro Musa, Youichi Shimomura, Ryusuke Egawa, Ken'ichi Itakura, Koki Okabe Hiroyuki Takizawa, and Hiroaki Kobayashi. Performance evaluation of nec sx-9 using real science and engineering applica-

tions. In *ACM/IEEE Supercomputing Conference 2009*, SC09, 2009.

[6] John A. Stratton, Vinod Grover, Jaydeep Marathe, Baastian Aarts, Mike Murphy, Ziang Hu, and Wen mei W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *2010 International Symposium on Code Generation and Optimization*, CGO2010, 2010.

[7] Hiroyuki Takizawa, Kentaro Koyama, Katuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. Checl: Transparent checkpointing and process migration of opencl applications. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS'11, 2011.

[8] The Khronos Group. OpenCL 1.0 specification, 2008.