An OpenCL-like Offload Programming Framework for SX-Aurora TSUBASA

Hiroyuki Takizawa*, Shinji Shiotsuki†, Naoki Ebata†, Ryusuke Egawa*
* Cyberscience Center, Tohoku University, Japan.
E-mail: {takizawa, egawa}@tohoku.ac.jp
† Graduate School of Information Sciences, Tohoku University, Japan.
E-mail: {shinji.shiotsuki.q6, naoki.ebata.r7}@dc.tohoku.ac.jp

***Abstract*—This paper presents an OpenCL-like offload programming framework for NEC SX-Aurora TSUBASA (SX-Aurora). Unlike traditional vector systems, one node of an SX-Aurora system consists of a host processor and some vector processors on PCI-Express cards, which are called a *vector host* and *vector engines*, respectively. Since the standard OpenCL execution model does not naturally fit in the vector engine, this paper discusses how to adapt the OpenCL specification to SX-Aurora while considering the trade off between performance and code portability. Performance evaluation results clearly demonstrate that, with a moderate programming effort, the proposed framework can express the collaboration between a vector host and a vector engine so as to make a good use of both of the two different processors. By delegating the right task to the right processor, an OpenCL-like program can fully exploit the performance of SX-Aurora.**

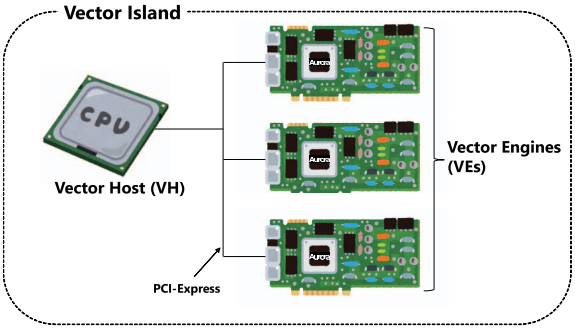

Fig. 1. The Hardware Configuration of an SX-Aurora System.

I. Introduction

Today, a high-performance computing (HPC) system is often equipped with different kinds of processors mainly for improving the energy efficiency. On such a "heterogeneous" HPC system, an application program can be executed by the collaboration of those different processors; a *host* processor launches an application program, and only data-parallel parts of the program are offloaded to other processors, called *computing devices*.

The latest vector computing system, NEC SX-Aurora TSUBASA (SX-Aurora) [1], would usually be used as a "homogeneous" computing system of vector processors. However, its physical hardware configuration is almost the same as those of heterogeneous computing systems equipped with different kinds of processors. Figure 1 illustrates the hardware configuration of SX-Aurora. SX-Aurora's vector processors on PCI-Express cards, called *vector engines* (VEs), are controlled by a host processor, called a *vector host* (VH). A set of one VH and one or more VEs is called a *vector island*. A VH is an x86 processor and runs the standard x86 Linux operating system (OS), while some user processes compiled for VEs are automatically executed on VEs. As the whole user process is automatically executed on a VE, users do not need to be aware of the heterogeneous hardware configuration. However, the two kinds of processors, a VH and a VE, have different processor architectures and thus different performance characteristics. Generally, a VE works best for executing "vector-friendly" loops of a program, which are vectorizable with larger vector lengths. If a large part of a program is not vector-friendly, it would be better to offload only vector-friendly loops and/or kernels of the program to VEs. Accordingly, offload programming is needed to fully exploit the performance of an SX-Aurora system by delegating the right task to the right processor. However, since traditional vector systems did not have such a heterogeneous hardware configuration, offload programing for vector systems have hardly been discussed so far.

Open Compute Language (OpenCL) is the open standard for cross-platform offload programming [2]. A standard OpenCL program consists of a host code and a device code, which are executed by a host and a computing device, respectively. OpenCL provides OpenCL host functions for a host code to control computing devices, and also the OpenCL C/C++ language for writing a device code to be executed on computing devices. Although OpenCL has already supported various computing systems, it is not available on SX-series processors yet because they traditionally did not assume to execute only parts of programs running on host processors.

In this work, we propose an OpenCL-like offload programming framework for SX-Aurora by considering both performance and code portability. In the proposed framework, a host code is written in C/C++ using standard OpenCL host functions. As a result, existing OpenCL host codes written for other computing systems can easily be ported to an SX-Aurora system; a large part of OpenCL host codes are reusable for SX-Aurora. Another important benefit of using standard

978-1-7281-2616-6/19/$31.00 ©2019 IEEE
DOI 10.1109/PDCAT.2019.00059

OpenCL host functions is that many programmers are already familiar with OpenCL programming. The proposed framework uses C++ not only for writing host codes but also for device codes, which are executed on VEs. In the proposed framework, therefore, both host and device codes are written in C++, and only the collaboration between a VH and a VE is expressed using OpenCL host functions. As a result, only vectorizable parts of a program are offloaded to VEs, and we can effectively use both a VH and VEs with considering their performance characteristics.

The main contributions of this work are as follows.

1) For SX-Aurora, the first offload programming framework partially but reasonably compatible with standard OpenCL is proposed;
2) Performance benefits from offload programming on SX-Aurora are quantitatively demonstrated using a real application;
3) The code migration costs are discussed by migrating three benchmark programs to the proposed framework.

II. Related Work

SX-Aurora is the latest vector computing system released in 2018 [1]. Unlike traditional NEC SX-series vector systems running with the proprietary OS, SX-Aurora uses the de-facto standard x86 Linux OS by introducing the heterogeneous hardware configuration. Once a program is compiled with NEC's compiler, its process is automatically launched on a VE. Then, the VE is devoted to executing such a user process. On the other hand, a VH is responsible for executing the Linux OS and handling all system calls from user processes running on VEs. As a result, users can run a program as if it is executed in a standard Linux environment, and implicitly benefit from the use of VEs without being aware of the hardware configuration.

So far, several case studies [3][4][5] have reported that SX-Aurora can achieve significantly high sustained performance, especially when executing various scientific applications whose performances are limited by the sustained memory bandwidths. This is mainly because SX-Aurora's VE has an outstanding memory bandwidth of 1.228 GB/s enabled by the world's first implementation of six High Bandwidth Memory 2 (HBM2) modules on a processor chip [1]. However, it is well known that any vector architectures can exert their potentials only for vector-friendly codes, which are vectorizable with large average vector lengths. On the other hand, a practical scientific application is usually a mix of vector-friendly and vector-unfriendly kernels, and hence it is obvious that we need to assign the right processor, either of a VH or a VE, to each kernel. Accordingly, we need offload programming to execute a program on a VH and offload only vector-friendly kernels to VEs.

OpenMP is one promising candidate for offload programming of SX-Aurora [6]. However, the OpenMP implementation is still under development, and thus the performance evaluation results have not been reported yet. In addition, although there exist various libraries and system software dedicated to VEs, the OpenMP approach does not allow an application code to use them because the OpenMP compiler is not able to compile only a part of the code in a totally-different way, e.g., linking a special library only to the part. Therefore, low-level APIs such as OpenCL host functions as well as OpenMP are needed so that programmers can directly express the offloading and thereby fully exploit the VE functionalities.

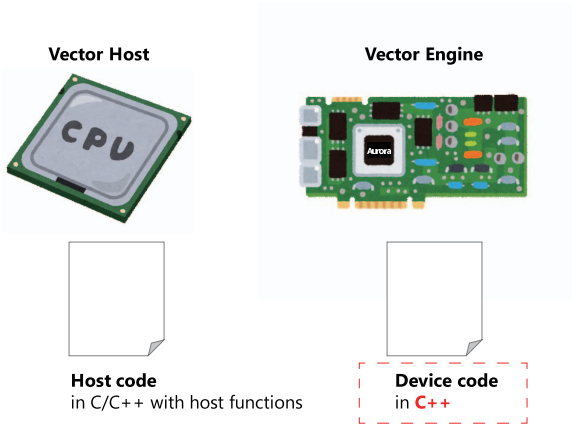

Fig. 2. The proposed OpenCL-like programming.

Vector Engine Offloading (VEO) [7] and Vector Host Call [8] are provided by NEC to describe the VH-VE collaboration. However, they are non-standard and do not provide any code portability with other frameworks at all.

III. OpenCL-like Offload Programming for SX-Aurora TSUBASA

A VH and a VE have totally different performance characteristics. Since one application program could be a mix of vector-friendly and vector-unfriendly computations, it is clear that there is room to improve the SX-Aurora performance by offloading only vector-friendly computations to VEs, and executing the other vector-unfriendly computations on the VH. To this end, we need a way of programming to specify which parts of a program are offloaded to VEs, also to specify data transfers between VHs and VEs. In this paper, therefore, we propose an OpenCL-like offload programming framework for SX-Aurora. Since the standard OpenCL execution model does not naturally fit in the VE, this paper discusses how to adapt the OpenCL specification to SX-Aurora while keeping the code portability and reusability as much as possible.

A. Offload Programming Framework

Basically, an OpenCL host code initializes computing devices, sends necessary data to computing devices, initiates kernel computation on computing devices, retrieves the computation results from computing devices, and finalizes the computing devices. An OpenCL host code would always play a similar role, and hence have a similar structure even if it is written for a totally different kind of computing devices. Accordingly, an OpenCL host code is portable across different systems, to some extent.

On the other hand, a device code is substantially not portable across different computing devices, even if the code is written

```
.
.
.
/* read the device source code from a file */
char * source = (char *)calloc(sourcesize, sizeof(char));
char * tempchar = "./kmeans.cl";
FILE * fp = fopen(tempchar, "rb");
fread(source + strlen(source), sourcesize, 1, fp);
fclose(fp); .
/* create a program from a device code in OpenCL C/C++ */
const char * slist[2] = { source, 0 };
cl_program prog
     = clCreateProgramWithSource(ctx,1,slist,NULL,&err);
.
.
.
/* launch n_points threads to execute kernel2 */
size_t gw[3] = { n_points, 1, 1 };
err = clEnqueueNDRangeKernel(cmd,kernel2,1,NULL,gw,...);
```

(a) The original host code.

```
.
.
.
/* read the shared object from a file to bin */
const unsigned char* bin = 0;
size_t blen;
bin = read_to_array("./kmeans_kernel.so",&blen); .
/* create a program from a device code in OpenCL C/C++ */
cl_program prog
     = clCreateProgramWithBinary(ctx,1,dlst,&blen,&bin...)
.
.
.
/* launch one thread to execute kernel2 */
size_t gw[3] = { 1, 1, 1 };
err = clEnqueueNDRangeKernel(cmd,kernel2,1,NULL,gw...);
```

(b) The modified host code.

Fig. 3. A host code migrated to the proposed framework.

in the OpenCL C/C++ language. In practice, a device code must be specialized and tailored for each kind of computing devices to achieve high performance. The OpenCL C/C++ language and its execution model would originally be designed keeping GPU architectures in mind, and does not necessarily fit in other architectures. For example, although the OpenCL C/C++ language is available for writing device codes for FPGAs, a device function for FPGAs is often written as a *single-work-item kernel* [9], which looks similar to a standard C/C++ function and is executed by a single work-item to exploit the parallelism within the function body.

Apparently, the OpenCL execution model does not fit in the VE, because the VE strongly relies on loop-level parallelism exploited by vectorizing a loop and thus each thread itself needs to execute a long loop. For a VE to efficiently execute a device function written for GPUs, it would be necessary to reconstruct the loop structure from the device function to exploit the loop parallelism. Indeed, some OpenCL implementations [10] thus support a code translation to reconstruct a loop from a device function. In such a case, however, programmers cannot directly optimize the loop structure, resulting in less efficient vectorization. This is a critical problem to exploit the VE performance. Consequently, in the proposed "OpenCL-like" programming framework, we have decided to use standard C++, instead of OpenCL C/C++, for describing device codes as illustrated in Figure 2. As a result, we can fully exploit the compiler's automatic vectorization capability by

```
__kernel void
kmeans_swap(__global float  *feature,
            __global float  *feature_swap,
            int     npoints,
            int     nfeatures)
{
  unsigned int tid = get_global_id(0);
  for(int i = 0; i <  nfeatures; i++) {
    feature_swap[i * npoints + tid]
      = feature[tid * nfeatures + i];
  }
}
```

(a) The original device code.

```
void
kmeans_swap(float  *feature,
            float  *feature_swap,
            int     npoints,
            int     nfeatures)
{
 /* the loop structure is restructured manually */
 #pragma omp parallel for
 for(unsigned int tid = 0;tid < npoints;tid++){
    for(int i = 0; i <  nfeatures; i++){
      feature_swap[i * npoints + tid]
        = feature[tid * nfeatures + i];
  }}
}
```

(b) The modified device code.

Fig. 4. A device code migrated to the proposed framework.

writing vectorizable C++ loops in device functions. Since host codes are portable, the proposed framework employs OpenCL host functions, while the framework needs to rewrite device codes for VEs because device codes are not performance portable by nature.

Another important reason of not using OpenCL C/C++ in the proposed framework is that OpenCL C/C++ is too restrictive to use VE's architecture-specific features and libraries. Unlike OpenCL C/C++, use of standard C++ for device codes also enables to adopt standard parallel programming models such as OpenMP [11] in device codes.

Figures 3 and 4 show how a host code and its corresponding device function are modified for the proposed framework. By not employing OpenCL C/C++, a host code does not need to launch a lot of work-items upon a device function call by invoking `clEnqueueNDRangeKernel`. In the proposed framework, a host code usually creates only one thread on a VE (Figure 3(b)), and then the thread spawns other threads in a standard way such as using OpenMP directives if necessary (Figure 4(b)). This makes data sharing among threads easier in comparison with creating independent threads on the VE at a device function call. More specifically, instead of launching `n_points` threads on the VE (Figure 3(a)), a host code is modified to launch only a single thread on the VE (Figure 3(b)), and its corresponding function is also modified to have a loop of `n_points` iterations (Figure 4(b)). It is a very common situation that the number of threads is adjusted for each computing device.

B. Implementation

VEO is a low-level programming interface provided by NEC to control a VE from a VH [7]. Generally, VEO and OpenCL host functions provide similar features, and hence it is mostly

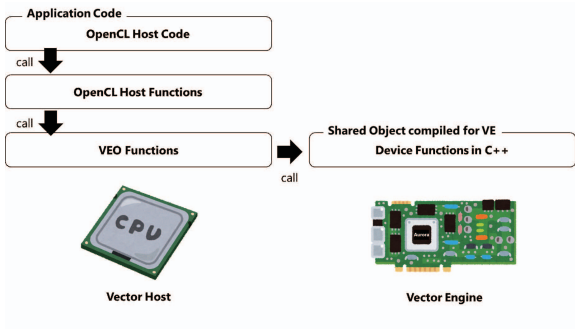

Fig. 5. Software stack of the proposed framework. From an application code, device functions of the application's kernels are invoked via OpenCL host functions and internally via VEO functions.

straightforward to implement OpenCL host functions by internally using VEO functions. Figure 5 illustrates the software stack of the proposed framework. The proposed framework can be build on top of VEO to improve code reusability and portability from standard OpenCL applications, which have originally been developed assuming other platforms such as GPUs. In the proposed framework, both host and device codes are written in C++, and only vector-friendly kernel parts of an application are expressed as device functions. The device functions are compiled for VEs, and their object codes are packed into a shared object. The shared object is dynamically loaded by calling a VEO function, `veo_load_library`, so that the device functions can be invoked by using another VEO function, `veo_call_async`. VEO also provides API functions for data transfers and synchronizations between a VH and a VE, which are internally used to implement OpenCL host functions of the proposed framework.

As in other OpenCL/OpenCL-like implementations, in our implementation, OpenCL objects of computing resources on devices are encapsulated as C++ class objects, and passed to VEs via OpenCL host function calls. One technical issue on this implementation approach is that a `cl_mem` object representing an allocated memory region on the VE side cannot simply be passed to a device function when the object is given as a function argument by calling the `clSetKernelArg` function. Although the `clSetKernelArg` function can take a `cl_mem` object as a `void` pointer, a device function is expecting to receive the memory address kept in the object. Therefore, we need to check the signature of each device function, and if a function parameter (formal argument) is a pointer type, its actual argument should be a memory address on the VE side that is kept in a memory object in the host code.

In the proposed framework, a device code is written in C++ and compiled to produce a shared object. Hence, we can obtain the type information of each function argument incorporated in the function symbol name (mangled name) when loading the shared object. If a device function has a pointer-type argument, the actual argument given as a `cl_mem` object is converted to a memory address on the VE side, and then the memory address is passed as the actual argument.

TABLE I
SYSTEM SPECIFICATIONS.

	NEC SX-Aurora TSUBASA A300-2
VH Processor	Intel Xeon Gold 6126 (Skylake)
Memory Capacity	98 Gbytes
Compiler	GNU C++ compiler 4.8.5
VE Processor	NEC Vector Engine Type-10B
Memory Capacity	48 Gbytes
Compiler	NEC C++ compiler 2.1.2
Operating system	CentOS Linux 7.5 1804
Storage device	SuperMicro SSD SOB20R

IV. PERFORMANCE EVALUATION AND DISCUSSIONS

A. Evaluation Setup

In this work, we have implemented the proposed framework on top of Portable Open Compute Language (POCL) version 1.2 [10]. We developed a POCL driver for VEs so that POCL internally uses VEO to support a subset of OpenCL 1.0 features, such as data transfers and offloading between VH and VE.

The specifications of the system used in the following evaluations are listed in Table I. Using the prototype implementation of the proposed framework on this system, this work discusses the performance gain and portability of our OpenCL-like offload programming approach.

B. Performance Evaluation Results

This work discusses the performance gain by offloading with the proposed approach. In this paper, the hybrid input-output (HIO) algorithm is considered as a practical application widely used for retrieving the phases in coherent diffraction imaging [12]. Our implementation of the HIO algorithm consists of vector-friendly and vector-unfriendly parts. Since the HIO algorithm iteratively updates image data, intermediate image data need to be saved to files at a certain interval. For the image file writing, the implementation uses the `fstream` class, which is the C++ standard file I/O class but not well optimized for SX-Aurora yet. On the other hand, computationally-intensive kernels of the implementation are FFT, IFFT, and some array operations that are all suited for vector processing and OpenMP parallelization. Accordingly, for appropriate division of workloads, the file writing and kernel computation should be assigned for the VH and the VE, respectively.

Figure 6 shows a simplified version of the `main` function of our HIO implementation. The `main` function is compiled for offloading if the C preprocessor macro `OFFLOAD` is defined. Three functions, `setup`, `initial_step`, and `optimization_step`, are offloaded to a VE by using OpenCL host functions as shown in Figure 7[1]. In this application program, it should be emphasized that the original C++ functions of `setup`, `initial_step`, and `optimization_step` are simply reused as device functions without any modifications. The original C++ functions

[1]Error handling is removed for simplicity. Most of function arguments are also omitted. Kernel arguments need to be set up before each kernel invocation.

```
int main(int argc, char** argv)
{
  // initial setup and image file loading
#ifndef OFFLOAD
  setup(...);
#else
  setup_opencl(...);
#endif

  // first optimization step
#ifndef OFFLOAD
  initial_step(...);
#else
  inis_opencl(...);
#endif

  for(img=0; img<NUM_FILES; img++){
    for(itr=1; itr<INTERVAL; itr++){
      // optimization step
      // (FFT, IFFT, and some array operations)
#ifndef OFFLOAD
      optimization_step(...) ;
#else
      opts_opencl(...);
#endif
    }

    // writing an intermediate image to a file
#ifdef OFFLOAD
    read_data(...);
#endif
    write_image(...);
  }
}
```

Fig. 6. A simplified `main` function of the HIO implementation.

```
1  void setup_opencl(int w, int h, double* spc, int*  pix)
2  {
3    int r;
4    const unsigned char* bin = 0;
5    size_t blen;
6    cl_int status;
7    int n = w*h;
8
9    clGetPlatformIDs(...);
10   clGetDeviceIDs(...);
11   ctx = clCreateContext(...);
12
13   bin = read_to_array("./kernel.so",&blen);
14   cl_program prog = clCreateProgramWithBinary(...);
15   clBuildProgram(...);
16
17   kern_init = clCreateKernel(prog,"setup",&r);
18   kern_inis = clCreateKernel(prog,"initial_step",&r);
19   kern_opts = clCreateKernel(prog,"optimization_step",&r);
20
21  /* initialization of other OpenCL objects */
22 }
23
24 void inis_opencl(int w, int h)
25 {
26   clEnqueueTask(cq, kern_inis, 0, NULL, NULL);
27 }
28
29 void opts_opencl(int w, int h)
30 {
31   clEnqueueTask(cq, kern_opts, 0, NULL, NULL);
32 }
33
34 void read_data(int size, double* pi)
35 {
36   clEnqueueReadBuffer(cq, d_spc, CL_TRUE,
37         0, size*sizeof(double), pi, 0, NULL, NULL);
38 }
```

Fig. 7. Code for offloading the kernel functions.

```
void optimization_loop(...)
{
  for(int i(1); i<maxi; i++){
     optimization_step(...);
  }
  return;
}
```

Fig. 8. Additional simple device function for "loop offloading."

are compiled to generate a shared object, which is loaded by a user-defined function `read_to_array` in Figure 7 (Line 13), and then passed to `clCreateProgramWithBinary` in Line 14. Then, a `cl_kernel` object is created for each of those functions in Lines 17 to 19, and used for calling the corresponding function using `clEnqueueTask` or `clEnqueueNDRangeKernel`. Accordingly, for migrating the HIO implementation to the proposed framework, we just write the collaboration between the VH and the VE, i.e., the code in Figure 7, and the other codes are almost unchanged and just reused. Only one important difference from the original `main` function is that the `read_data` function is called before writing images because the image data are updated on the VE side and need to be transferred to the VH side before the VH executes the image file writing.

In Figure 6, the `optimization_step` function is invoked on the VE at every iteration of the innermost loop (hereafter, this implementation is called *function offloading*). Since the VE can execute a standard C++ function, we can offload the whole of the innermost loop to reduce the number of function calls, and hence the function call overhead, i.e., the innermost loop can be separated from the loop nest and implemented as a function, `optimization_loop` in Figure 8. By invoking this new function, we need to call `clEnqueueTask` or `clEnqueueNDRangeKernel` only once for generating one intermediate image (this implementation is called *loop offloading*). Therefore, the function call overhead can be reduced in return for the small additional programming effort.

Figure 9 shows the total execution time of each implementation for executing 10,000 iterations of the HIO algorithm on an image of 1,024x1,024 pixels. Intermediate image data of approximately 1.1 MB are saved at every 100 iterations. The results show that the total execution time of the VE is longer than that of the VH, even though the VE outperforms the VH in terms of kernel computation performance. This is because of the following two reasons. One reason is that the VE needs a larger file I/O access overhead. The other is that the file I/O performance of the VE becomes extremely low if the `fstream` class is used for the image file writing. This is probably because its internal implementation is more complicated and hence vector-unfriendly than the C standard I/O library functions such as `fwrite`. As SX-Aurora is the brand-new vector computing system, the implementation of the `fstream` class for SX-Aurora may not be matured yet. Figure 6 clearly shows the performance gain from our offloading approach. Accordingly, by delegating the right task to the right processor, the proposed framework allows the OpenCL-like program to make a good use of SX-Aurora as a heterogeneous computing system.

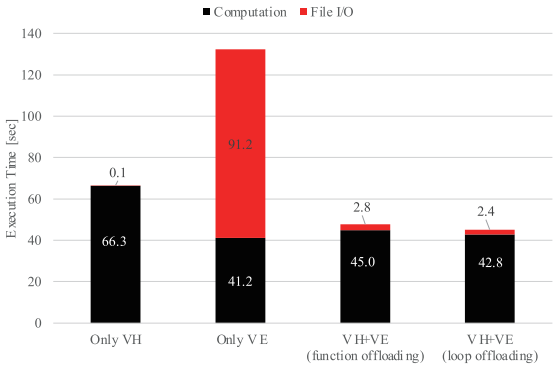

Fig. 9. Performance comparison among HIO implementations.

TABLE II
THE TOTAL NUMBER OF CODE LINES.

	OpenMP	OpenCL	Proposal
`kmeans`	487	295	294
(kernels)	(243)	(56)	(55)
`lud`	149	453	252
(kernels)	(28)	(163)	(19)
`nw`	253	592	419
(kernels)	(32)	(203)	(63)

TABLE III
THE NUMBER OF MODIFIED CODE LINES.

	Host code	Device code	Original kernel
`kmeans`	13	17	OpenCL
`lud`	69	9	OpenMP
`nw`	53	53	OpenMP

Image data are transferred from the VE to the VH via PCI-Express before the image file writing on the VH side. As a result, the data transfer overhead slightly increases the image file writing time in comparison with the case of using only the VH. By comparing the two offloading versions, function offloading and loop offloading, we can discuss the device function call overhead required by the proposed framework. The function offloading version calls the `optimization_step` function 100 times before writing an image, while the loop offloading version calls the `optimization_loop` function only once. As a result, the latter version can achieve a higher performance than the former. In the case of our HIO implementation, the performance difference is relatively small. Therefore, we believe that the runtime overhead of the proposed framework itself is small enough for practical use.

C. Discussions on Code Reusability and Portability

Using the Rodinia benchmark suite, this work also discusses the code reusability and portability of the proposed OpenCL-like approach. The Rodinia benchmark suite provides CUDA, OpenACC, OpenCL, and OpenMP versions of each benchmark program. Basically, a benchmark program can be migrated to the proposed OpenCL-like framework by mostly reusing the host code of the OpenCL version and offloading only the kernel parts of the OpenCL or OpenMP version.

Table II shows the total number of code lines of three benchmark programs, `kmeans`, `lud`, and `nw`, selected from the Rodinia benchmark suite. The number of code lines in the kernel parts to be offloaded is also shown in the table. To discuss the code reusability, the existing codes are reused as much as possible for adapting the programs to the proposed framework. Table III summarizes how many code lines are modified for migrating the three benchmark programs to the proposed framework.

In the OpenCL version of each benchmark program, all device codes are written in OpenCL C/C++. At runtime, the device codes are loaded from files and then compiled to create `cl_program` objects as shown in Figure 3(a). On the other hand, the proposed framework assumes that all device functions are defined in a shared object, e.g., `kmeans_kernel.so` in Figure 3(b).

For simple device functions, such as the function in Figure 4(a), that make all work-items execute exactly the same operations on different data, a loop is simply inserted for such a function to execute the computations of all OpenCL work-items as shown in Figure 4(b). Such simple device functions can be found in several benchmark programs such as `kmeans`, `bfs`, `cfd`, `gaussian`, and `nn`.

A code often becomes complicated by adapting to the OpenCL specification. For example, comparing OpenMP and OpenCL versions of the `lud` benchmark program, we can find that the OpenMP version is much simpler than the OpenCL version. In the proposed approach, the simple OpenMP kernel can be reused as the device code. In the case of `lud`, we can reduce the number of code lines by about 44.4% in total when adapting the OpenCL code to the proposed framework.

As shown in Table III, we reuse most of host and device codes of the `kmeans` program, and the code modification is needed mostly to adopt the offline compilation. In the cases of `lud` and `nw`, the kernel parts of their OpenMP versions are reused to create the device functions for the proposed framework. As the original kernel of `lud` is a separate function, the function is simply used as a device function, while kernel loops are extracted to organize two functions and used as device functions in `nw`. Then, their host codes are simplified to launch those device functions. In the case of `lud`, three device functions are replaced with only one C++ function, and hence the host code is simplified more significantly than that of `nw`. Accordingly, the proposed framework can mostly reuse and even simplify existing application codes at code migration.

In general, OpenCL is not performance portable at all [13]. Thus, code migration by reusing the existing codes might not result in a significant performance gain. Code migration and optimization of practical applications with the proposed framework will be further discussed in our future work.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed an OpenCL-like offload programming framework for SX-Aurora while considering performance and code portability. Our evaluation results clearly show the performance benefit of using both of a VH and a VE in an appropriate way. Since a practical application is often a

mix of vector-friendly and vector-unfriendly computations, by delegating the right task to the right processor, we can fully exploit the potential of SX-Aurora. Although this work focuses on SX-Aurora, the proposed approach would be useful also for other accelerators, to which the OpenCL execution model does not fit well.

To discuss the code reusability, this paper assumes that OpenMP and OpenCL versions of a code are already available. The OpenMP version is reusable for the proposed framework without major code modifications as discussed in this paper. However, in such a case, the performance of SX-Aurora strongly depends on how vector-friendly the code is, and vector-aware code tuning could be required for high performance in practice. Therefore, we will extend the framework to support architecture-specific code optimizations for SX-Aurora. These will be discussed in our future work.

ACKNOWLEDGMENTS

The authors would like to thank Yasuhisa Masaoka, Yoko Isobe, Yuuta Watanabe, Souya Fujimoto, and Erich Focht of NEC for meaningful discussions and suggestions on this work.

This work is partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program "R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications" and Grant-in-Aid for Scientific Research(B) #16H02822 and #17H01706.

REFERENCES

[1] Y. Yamada and S. Momose, "Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA," in *Hot Chips: A Symposium on High Performance Chips*, 2018.

[2] Khronos OpenCL Working Group, "The OpenCL Specification version:1.0 document revision: 48," http://www.khronos.org/registry/cl/, 2009.

[3] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance evaluation of a vector supercomputer SX-Aurora TSUBASA," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18)*, 2018, pp. 685–696.

[4] K. Yamaguchi, T. Soga, Y. Shimomura, T. Reimann, K. Komatsu, R. Egawa, A. Musa, H. Takizawa, and H. Kobayashi, "Performance evaluation of different implementation schemes of an iterative flow solver on modern vector machines," *Supercomputing Frontiers and Innovations*, vol. 6, no. 1, pp. 36–47, 2019.

[5] Y. Shimomura, M. Kano, T. Soga, K. Yamaguchi, A. Musa, Y. Mizuno, S. Takahashi, R. Egawa, and H. Takizawa, "Optimization of a gas-particle flow solver on vector supercomputers," in *The 31st International Conference on Parallel Compoutational Fluid Dynamics (ParCFD2019)*, 2019.

[6] T. Cramer, "NEC and RWTH Aachen University Collaboration: OpenMP Offload Programming Model for NEC SX-Aurora TSUBASA," 2018, presented at NEC Aurora Forum at ISC High Performance 2018.

[7] NEC Corporation, "SX-Aurora TSUBASA Vector Engine Offloading (VEO)," https://github.com/veos-sxarr-NEC/veoffload.

[8] E. Focht, "Example for VHcall offloading from VE to VH for SX-Aurora TSUBASA," https://github.com/efocht/vhcall-example.

[9] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-based computing systems with OpenCL*. Spinger International Publishing, 2017.

[10] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, 2015.

[11] OpenMP Architecture Review Boards, "The OpenMP API specification for parallel programming," https://www.openmp.org/.

[12] J. Fienup, "Reconstruction of an object from the modulus of its fourier transform," *Optics Letters*, vol. 3, no. 1, pp. 27–29, 1978.

[13] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of OpenCL programs," in *The 5th International Workshop on Automatic Performance Tuning (iWAPT2010)*, 2010.