Design and Preliminary Evaluation of Omni OpenACC Compiler for Massive MIMD Processor PEZY-SC

Akihiro Tabuchi[1](✉), Yasuyuki Kimura[2], Sunao Torii[2], Hideo Matsufuru[3], Tadashi Ishikawa[3], Taisuke Boku[1,4], and Mitsuhisa Sato[1,5]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Japan
`tabuchi@hpcs.cs.tsukuba.ac.jp`
[2] ExaScaler Inc., Tokyo, Japan
[3] Computing Research Center, High Energy Accelerator Research Organization (KEK), Tsukuba, Japan
[4] Center for Computational Sciences, University of Tsukuba, Tsukuba, Japan
[5] RIKEN Advanced Institute for Computational Science, Kobe, Japan

Abstract. PEZY-SC is a novel massive Multiple Instruction Multiple Data (MIMD) processor used as an accelerator and characterized by high power efficiency. OpenACC is a standard directive-based programming model for accelerators, and programmers can concisely offload data and computation to the accelerators. In this paper, we present the design and preliminary implementation of an OpenACC compiler for a PEZY-SC. Our compiler translates C code with OpenACC directives to the corresponding PZCL code, which is the programming environment for PEZY-SC. The evaluation shows that the performance of the OpenACC version achieves over 98 % at N-body and up to 88 % at NAS Parallel Benchmarks CG than that of the PZCL version. In addition, we examined optimization techniques such as kernel merging and explicit context switching to exploit the PEZY-SC MIMD architecture, which differs from the single instruction multiple data graphics processing units. We found these optimizations useful in improving the performance and will be implemented in the future release.

Keywords: PEZY-SC · OpenACC · Compiler

1 Introduction

Accelerators are widely used in super computers for improving power efficiency because of limitation of power supply. At Green 500 [1], which is a ranking of the 500 most energy-efficient supercomputers, the top 10 systems were equipped with accelerators in November 2015. The top system is the Shoubu Supercomputer at RIKEN and is developed by ExaScaler in Japan. Its accelerator is a PEZY-SC processor, which has 1024 cores and 8192 threads that run in multiple instruction multiple data (MIMD) format. ExaScaler provides PZCL as a programming environment for PEZY-SC. PZCL is based on OpenCL [2] but its kernel description differs from that of OpenCL. Application development using PZCL is complicated because programmers must describe many codes for offloading data and computations to PEZY-SC. This degrades the application productivity.

In recent years, OpenACC [3] is being widely used for accelerator programming. It is a directive-based programming model for accelerators and allows a programmer to offload codes to accelerators to simplify the porting process for legacy CPU-based applications. Some commercial and research compilers support OpenACC for Graphics Processing Units (GPUs). If PEZY-SC is supported by OpenACC, it will be easier to use because users can rapidly develop applications and reuse the existing OpenACC code. Thus, we designed and preliminarily implemented an OpenACC compiler for PEZY-SC by using the Omni OpenACC compiler [4] originally targeted for NVIDIA GPU. We evaluated the performance of the compiler using the N-Body benchmark and NAS Parallel Benchmarks CG (NPB-CG) [5] and the productivities of PZCL and OpenACC.

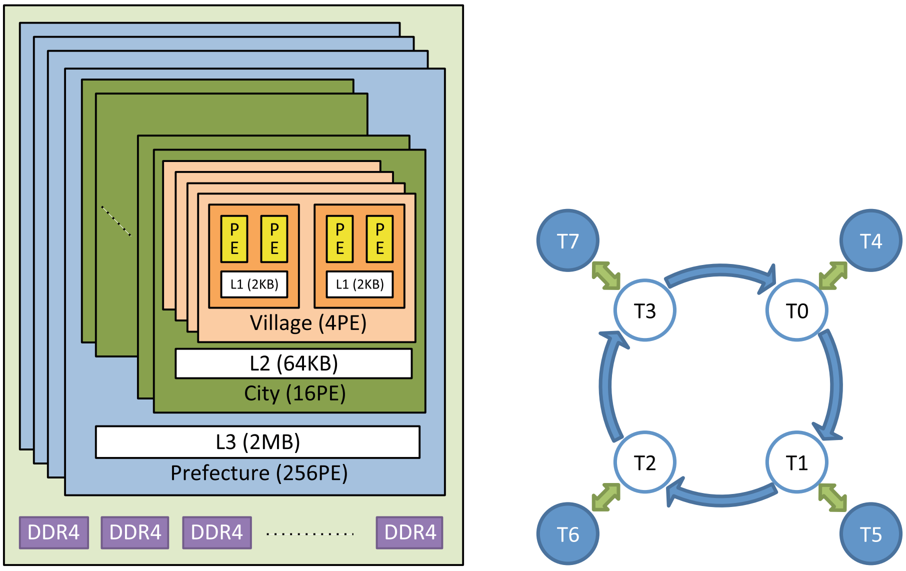
The contributions of this paper are summarized as follows:

– We have designed an OpenACC for the massive MIMD many-core processor, PEZY-SC. We determined that OpenACC is useful and provides a good programming model to improve the productivity for programmers of PEZY-SC.
– We propose some optimizations to exploit the PEZY-SC architecture, and show the effectiveness of these optimizations by comparing the performance of the nonoptimized version generated by our current preliminary OpenACC compiler. We examine the implementation of these optimizations by using the compiler and new additional directives.

The remainder of this paper is organized as follows. Section 2 introduces related works and Sect. 3 describes the architecture and programming of PEZY-SC. Section 4 describes the design and detailed implementation of the Omni OpenACC compiler for PEZY-SC. Further, we report the performance and productivity evaluation using two benchmarks in Sect. 5 and discuss on optimization for PEZY-SC and comparison with OpenMP in Sect. 6. Finally, we conclude our study in Sect. 7.

2 Related Work

Several open-sourced OpenACC compilers, such as accULL [6], OpenUH - OpenACC [7], OpenARC [8], and RoseACC [9] have been developed. Moreover, GCC supports OpenACC from version 5.0.1, and the development of future versions are under progress [10]. accULL is the first open-sourced OpenACC compiler, which translates code from OpenACC to CUDA or OpenCL with optimization through the YaCF compiler framework. OpenUH-OpenACC is based on the OpenUH compiler framework and translates OpenACC to CUDA or OpenCL. OpenARC is a compiler framework for accelerators and supports full features in OpenACC 1.0. The compiler is based on the Cetus compiler infrastructure, which



(a) Configuration (b) Two sides of threads

Fig. 1. PEZY-SC processor

has some code analysis and transformation features. RoseACC is a Rose Compiler based on the OpenACC compiler and translates code to OpenCL. These compilers mainly target GPUs but compilers that translate to OpenCL may be able to target various devices such as CPU, Xeon Phi, and FPGA. However, they are unavailable for PEZY-SC because the description of a kernel in PZCL is specialized for the PEZY-SC architecture and some built-in operations are not supported in OpenCL. We propose the implementation of the Omni OpenACC compiler to translate to PZCL and support PEZY-SC.

3 PEZY-SC

3.1 Architecture

PEZY-SC is a many-core processor developed by PEZY computing. Figure 1(a) shows the structure of the processor. The processor contains 1024 Processing Elements (PEs), which run in MIMD. In each PE, eight threads are executed by using simultaneous multithreading (SMT), that is, 8192 threads run on the processor. The processor is hierarchically constructed with PE, Village, City, and Prefecture in an ascending order. A PE has registers for eight threads, 16 KB local memory, two Arithmetic Logic Units (ALUs), two Floating-Point Units (FPUs), etc. In addition, a Village is constructed with four PEs and a 2 KB L1 cache per two PEs. Further, a City is constructed with four Villages, a Special

Function Unit (SFU), and a 64 KB L2 cache. Moreover, a Prefecture is constructed with 16 cities and a 2 MB L3 cache. The processor has four prefectures or 1024 PEs.

Eight threads in a PE are actually the two sides of 4 threads, as shown in Fig. 1(b). One side of each of the four threads is executed in a round-robin manner every clock cycle, and some synchronizations or explicit switching operations switch the threads to the opposite side. Caches L1–L3 have no coherency. Thus, if a PE reads some values written by another PE, programmers need to flush cached data through a flush operation before the read.

3.2 Programming

PEZY computing provides PZCL, which is a language based on OpenCL as a programming environment for PEYZ-SC. Thus, programmers also need to write the host and device codes; however, some points differ from those of OpenCL.

Host Code. A host code is written using C APIs, which is a subset of OpenCL 1.1. It supports almost all APIs that conventional OpenCL code uses; however, the building and launching of kernels are limited. A conventional OpenCL code usually uses online-compilation, which compiles kernel code at runtime; whereas PZCL only supports offline-compilation, which compiles kernel code at compile time. When launching a kernel, *global_work_size* must be a multiple of 128 because PEZY-SC executes threads by the City and *local_work_size* must be eight because a number of threads in a PE is fixed at eight. Moreover, we can specify at least three dimensions for the shape of a work item in OpenCL, whereas we can specify only one dimension in PZCL. Notably, *global_work_size* should be 8192 or less, otherwise the kernel is launched multiple times.

Device Code. A device code is written in C/C++. Kernels, which are launched from the host, are described as functions same as those in OpenCL. The qualifiers `__kernel` and `__global` or `__local` must be respectively added to the kernel functions and kernel function parameters in OpenCL. Although we must add "pzc_" prefix to the function name and retain the kernel parameters in PZCL, the memory shared among threads in a PE cannot be defined (it corresponds to local memory in OpenCL). In the function body, computation is parallelized using PE and thread IDs, which are obtained through get_pid() and get_tid(), respectively. These correspond to get_group_id(0) and get_local_id(0) in OpenCL. The number of PEs and threads can be obtained using get_maxpid() and get_maxtid(),which correspond to get_num_groups(0) and get_local_size(0) respectively in OpenCL.

PZCL provides some functions for PEZY-SC specific features as follows. The function chgthread() switches the thread to its opposite side explicitly, and sync() synchronizes all threads in a processor. In addition, functions sync_L1(), sync_L2(), and sync_L3() synchronize threads by the Village, City, and Prefecture, respectively. Further, flush() synchronizes all threads in a processor and flushes all cached data. Moreover, flush_L1() synchronizes threads by the Village and flushes L1 cache data, and flush_L2() synchronizes threads by the City and flushes L1 and L2 cache data. We need to optimize kernel functions using these built-in functions for obtaining the best performance. Here, we describe two optimizations used in this study.

Kernel merging. In CUDA or OpenCL, there is no synchronization among thread-blocks (CUDA) or work-groups (OpenCL); thus, we need to divide a kernel code into several kernel functions at global synchronization points. This leads to the increase of kernel launch overhead. However, PZCL provides global synchronization in a processor (sync()); therefore, we do not need to divide the kernel code and the kernel launch overhead can be reduced.

Explicit thread switching. This optimization has two benefits. (1) The improvement of cache utilization. Paired threads tend to access data that are closer because the difference between the thread numbers is four. The explicit switching allows the opposite side to access data before the cache is removed and improves the cache hit ratio. (2) The second benefit is latency hiding of memory access. Switching to the opposite side reduces stall through memory access.

4 Omni OpenACC Compiler

This section describes the design and implementation of the Omni OpenACC compiler for PEZY-SC.

4.1 Design

OpenACC supports C, C++, and Fortran; however, our compiler only supports C. The Omni OpenACC compiler is a source-to-source compiler, and thus translates an OpenACC C code to a host C code and a kernel PZCL code. Thus, the device code for PEZY-SC can be generated by the PZCL compiler. In the host code, the directives are translated to some runtime library calls for maintaining portability. In the kernel code, the offloaded code is translated using functions for simplification and commonization.

4.2 Implementation

To realize the code translation, we used the Omni compiler infrastructure [11], which is a set of programs for a source-to-source compiler with code analysis and transformation. Figure 2 shows the flow for compilation. An OpenACC translator is used to translate an input OpenACC C code to host C and kernel PZCL codes. The host code is compiled using a general C compiler (e.g., GCC and ICC) and linked with the Omni OpenACC runtime library for PZCL. The kernel code is compiled to a kernel binary code using a PZCL compiler, and the binary is loaded at runtime. Note that our preliminary implementation does not currently support the optimizations for PEZY-SC.

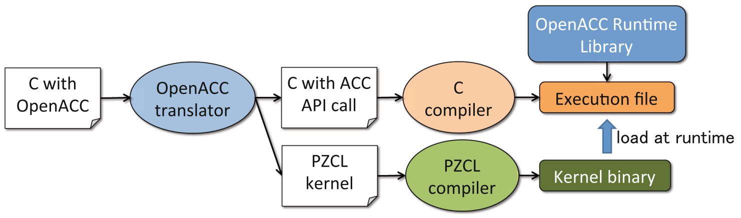


Fig. 2. Compilation flow

```
int a[100], b;
#pragma acc data copy(a) copyout(b)
{
  /* some codes using a and b */
}
```

(a) data construct

```
int a[100], b;
{
  void *DESC_a,*DEV_ADDR_a,*DESC_b,*DEV_ADDR_b;
  unsigned long long _lower[] = {0};
  unsigned long long _length[] = {100};
  _ACC_init_data(&(DESC_a),&(DEV_ADDR_a),a,sizeof(int),1,_lower,_length);
  _ACC_init_data(&(DESC_b),&(DEV_ADDR_b),&(b),sizeof(int),0,NULL,NULL);
  _ACC_copy_data(DESC_a,_ACC_HOST_TO_DEVICE,_ACC_ASYNC_SYNC);
  {
    /* some codes using a and b */
  }
  _ACC_copy_data(DESC_a,_ACC_DEVICE_TO_HOST,_ACC_ASYNC_SYNC);
  _ACC_copy_data(DESC_b,_ACC_DEVICE_TO_HOST,_ACC_ASYNC_SYNC);
  _ACC_finalize_data(DESC_a);
  _ACC_finalize_data(DESC_b);
}
```

(b) translated code

Fig. 3. Code translation of data construct

Translation of Data Construct. A `data` construct declares data on an accelerator in the following region. According to clauses, device memory is allocated and data is transferred from host to device at the beginning, and data is transferred from device to host and device memory is released at the end. The compiler translates these operations to runtime library calls.

Figure 3 illustrates a `data` construct and the translated code. This data construct specifies that an array *a* and a variable *b* are allocated on a device at the beginning of the region and freed at the end. The array *a* in the `copy` clause is transferred at the beginning and end of the region, and the variable *b* in the `copyout` clause is transferred at the end of the region. The function _ACC_init_data allocates data on the device. If the data is an array, the lower bound and length of each dimension are passed. The variable DEV_ADDR_*name* is a pointer to the device memory that corresponds to the host memory of *name*. The DESC_*name* is a pointer to the structure, which has the host address, device address, shape, element size, etc. The function _ACC_copy_data transfers data between the host and device, and the function _ACC_finalize_data frees data on the device.

Translation of Parallel Construct. A `parallel` construct offloads the following region to a device. OpenACC supports three-level parallelism: gang, worker, and vector. However, PZCL provides only the PE and thread. Although our preliminary implementation maps the gang to PE and the vector to thread in PEZY-SC and does not use worker-level parallelism, this correspondence might be changed in the future. The compiler generates a kernel function from the offloading region. Variables, which are accessed in the region, become the function parameters, unless they are specified in the `private` clauses. The function is launched from the host code with device memory objects or values as arguments. Additionally, the number of launch PEs is specified by the `num_gangs` clause. Alternatively, it may be determined by the number of loop iteration in the region.

Figure 4 illustrates a `parallel` construct and the translated code. The function pzc__ACC_kernel_0 is a kernel function. There are 16 PEs from the `num_gangs` clause and the number of threads is fixed at 8. The array _ACC_args contains the kernel arguments and the array _ACC_argsizes contains the sizes of the arguments. Further, the function _ACC_launch launches the kernel function. The first argument _ACC_program is a pointer of structure, which contains kernel objects loaded from a kernel binary file, and the second argument is the kernel number. In the function _ACC_launch, the kernel is launched after the number of total threads is adjusted to multiples of 128 and 8192 or less for PEZY-SC.

Translation of Loop Construct. A `loop` construct specifies parallelization of the following loop in an offloaded region. If parallelisms (gang or vector) are specified, the compiler parallelizes the loop with them; otherwise, the compiler

```
#pragma acc parallel present(a) num_gangs(16)
{
  /* codes in parallel region */
}
```

(a) parallel construct

```
/* host code */
{
  int _ACC_ngangs = 16;
  int _ACC_nworkers = 1;
  int _ACC_veclen = 8;
  int _ACC_conf[] = {_ACC_ngangs, _ACC_nworkers, _ACC_veclen};

  void* _ACC_args[] = {&DEV_ADDR_a};
  size_t _ACC_argsizes[] = {sizeof(void*)};
  _ACC_launch(_ACC_program, 0, _ACC_conf, ACC_ASYNC_SYNC, 1, args, arg_sizes);
}

/* kernel function in device code */
void pzc__ACC_kernel_0(int *a)
{
  /* codes in parallel region */
}
```

(b) translated code

Fig. 4. Code translation of parallel construct

```
/* inside parallel region */
#pragma acc loop vector reduction(+:sum)
for(i = 0; i < N; i++){
  a[i]++;
  sum += a[i];
}
```

(a) loop construct

```
/* inside kernel function */
int _niter_i, _idx, _init, _cond, _step, _red_sum;
_ACC_init_reduction_var(&_red_sum,0);
_ACC_calc_niter(&_niter_i, 0, N, 1);
_ACC_init_thread_iter(&_init,&_cond,&_step,_niter_i);
for(_idx = _init; _idx < _cond; _idx += _step){
  int i;
  _ACC_calc_idx(_idx, &i, 0, N, 1);
  a[i]++;
  _red_sum += a[i];
}
_ACC_reduction_thread(sum,_red_sum, 0);
```

(b) translated code

Fig. 5. Code translation of loop construct

automatically determines parallelisms. The loop iterations are cyclically scheduled for both PE and thread. If `reduction` clauses are present, private variables are prepared before the loop, and their reduced values are stored in reduction variables after the loop.

Figure 5 illustrates a `loop` construct and the translated code. The function _ACC_calc_niter calculates the number of iterations of the loop, and the function _ACC_init_thread_iter obtains the initial, conditional, and step values of the iteration for its thread. In the loop body, the function _ACC_calc_idx obtains the value of the loop variable. For reduction, the function _ACC_init_reduction_var initializes the thread local variable, and then the function _ACC_reduction_thread reduces the value of the variable among threads in the PE.

5 Evaluation

This section presents the evaluation of our compiler's performance and productivity of OpenACC.

5.1 Benchmark

For evaluation, we used the N-body benchmark and NPB-CG. The N-body benchmark simulates the motion of particles that interact and calculates the interactions of all pairs in a simple manner using a single-precision floating-point value. NPB-CG is a benchmark, which evaluates the smallest eigenvalue of a large sparse symmetric positive definite matrix by using the conjugate gradient method. We developed the following versions of the benchmarks.

PZCL (Base). The baseline code written in PZCL. This uses kernel functions separated at global synchronization points.

Table 1. Evaluation environment

	Suiren Blue	HA-PACS/TCA
CPU	Intel Xeon-E5 2618Lv3 2.3 GHz	Intel Xeon-E5 2680v2 2.8 GHz×2
Memory	DDR4 64 GB, 1866 MHz	DDR3 128 GB, 1866 MHz
Accelerator	PEZY-SC×4	Tesla K20X×4
-Peak perf.	SP: 3 TFlops, DP: 1.5 TFlops	SP: 3.95 TFlops, DP: 1.31 TFlops
-Memory	DDR4 16 GB, 1866 MHz	GDDR5 6 GB
-Memory BW	153.6 GB/s	250 GB/s
Compiler	ICC 14.0.2, PZSDK 2.1, Omni OpenACC compiler 0.9.3 for PEZY-SC	PGI 15.10, CUDA 7.5, Omni OpenACC compiler 0.9.3

PZCL (A). PZCL (Base) with optimization A (Kernel merging)
PZCL (A, B). PZCL (Base) with optimizations A and B (Explicit thread switching)
OpenACC. The code written in C with OpenACC

The performance of OpenACC should be equal to that of PZCL (Base) because our compiler does not support the optimizations.

5.2 Performance

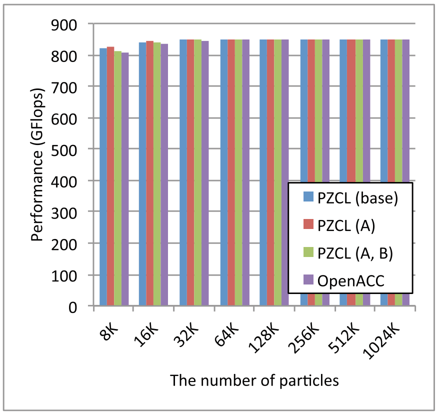
We measured the performance using Suiren Blue and HA-PACS/TCA, and the evaluation environments are shown in Table 1.

Figure 6(a) shows the performance of the N-body benchmark on PEZY-SC.

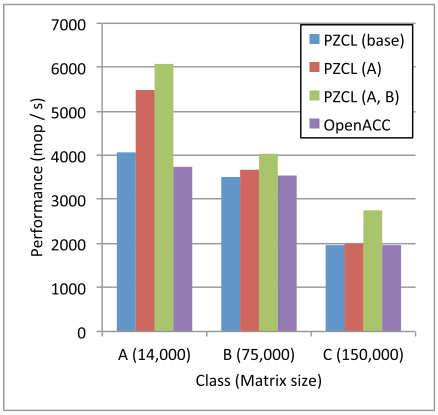
The results show flops under an assumption of 38 floating-point operations per interaction [12]. The performance of OpenACC version is 97.8–100.0 % of PZCL versions. The effects of optimizations A and B are slight because the calculation of interactions is dominant.

Figure 6(b) shows the performance of the NPB-CG benchmark on PEZY-SC. “mop/s” implies mega operations per second. The performance of OpenACC version is over 91.9 % of PZCL (base) version. Moreover, the OpenACC version has unnecessary transfers related to reduction kernels. The value of the reduction variable is copied from the host to device despite the initial value being always zero. The optimization A is effective when the matrix size is small, especially because the ratio of kernel launch overhead is large. The optimization B has a good effect when the matrix size is large because the opposite-side threads can utilize cached data before it is removed. The performance of OpenACC version is 61.6–87.5 % of the PZCL (A,B).

Finally, for comparison, we measured the performance of the benchmarks of the OpenACC version on Tesla K20X and compared it with that of the benchmarks on PEZY-SC, as shown in Fig. 7(a) and (b). For K20X, we measured the performance using both PGI and Omni compilers. For the N-body benchmark,



(a) N-body



(b) NPB-CG

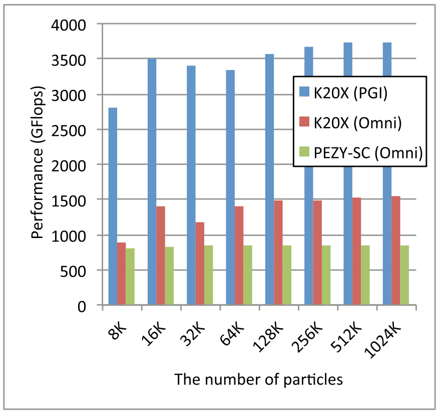
Fig. 6. Performance of benchmarks on PEZY-SC

the performance of Omni version is less than half of the performance of the PGI version because particle data is loaded as scalars in the Omni version and as a vector in the PGI version. For the NPB-CG benchmark, the Omni version outperforms the PGI version because the Omni compiler utilizes warp shuffle operation during reduction and the adjustment of the number of thread-blocks is better than in PGI. Even when considering that the OpenACC compiler for PEZY-SC is not optimized, the performance of PEZY-SC is unsatisfactory compared with that of K20X. This may be because PEZY-SC has small number of SFUs, which calculate the reciprocal square root for N-body, and the memory bandwidth is low for NPB-CG.

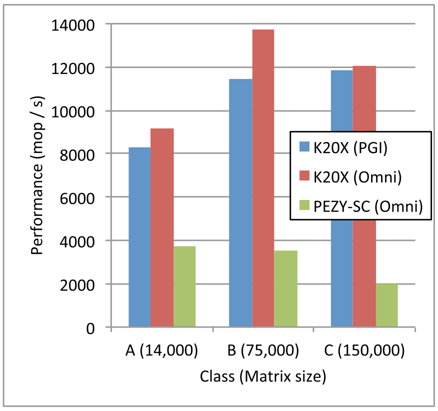
5.3 Productivity

In PZCL, programmers need to manage memory and kernel execution on the accelerator using many complex API calls same as in OpenCL, and write parallelized kernel functions in the PZCL-specific description. Contrastively, in OpenACC, programmers can directly offload and parallelize parts of serial code by using several simple directives. Moreover, OpenACC is a standard specification; therefore, OpenACC code is available for all accelerators.

We counted the Source Lines Of Code (SLOC) to measure the productivity quantitatively. Table 2 shows the SLOC of N-body and NPB-CG. The SLOC of OpenACC version are 48 % and 45 % of those of the PZCL version for N-Body and NPB-CG respectively, and these are almost the same as their serial codes. Therefore, OpenACC shows better productivity than PZCL.



(a) N-body



(b) NPB-CG

Fig. 7. Performance of OpenACC version on K20X and PEZY-SC

Table 2. SLOC of N-Body and NPB CG. The number in parentheses represent the lines of directives.

	N-Body	NPB CG
Serial	109	418
PZCL (A, B)	240	1001
OpenACC	114 (5)	447 (25)

6 Discussion

6.1 Optimization for PEZY-SC

We have not yet implemented the proposed optimizations in our OpenACC compiler. In this subsection, we describe how to apply these optimizations.

The optimization of kernel merging can be implemented in the translation of `kernels` constructs. Similar to a `parallel` construct, a `kernels` construct offloads the following region to a device. While the `parallel` construct launches a single kernel, the `kernels` construct launches multiple kernels. Our current compiler translates a `kernels` region to some separated kernels, in the same manner as for GPU. We will modify our compiler to translate the region to the single kernel when the target is PEZY-SC. In the kernel, our compiler automatically inserts sync() or flush() at the ends of each loop parallelized among gangs, not breaking the semantics of the OpenACC program.

For explicit thread switching, we propose an additional directive corresponding to chgthread(). Another method is for our compiler to automatically insert chgthread() at appropriate points, such as at the end of a loop body, but it is not always effective because it may lead to additional overhead due to thread switching or cache miss. Therefore, it is preferable to give an API for programmers to specify thread switching explicitly by using some directives. As there is no OpenACC directive that corresponds this, we will introduce the additional directive to OpenACC. In addition, our compiler will replace the directive to chgthread() when the target is PEZY-SC, or ignore it.

Although we did not exploit the hierarchy of PEs on PEZY-SC in this study, we will utilize that by also using `worker` level parallelism in OpenACC. For example, it is considered to map `gang` to City, `worker` to PE, and `vector` to thread.

6.2 Comparison with OpenMP

OpenMP supports offloading since version 4.0, and that is similar to OpenACC. Even if we implement an OpenMP compiler for PEZY-SC, we will obtain almost the same performance and productivities for the benchmarks.

However, some difference between OpenMP and OpenACC may affect performance. OpenMP `parallel` construct can be multiple nested; whereas OpenACC has only three-level parallelism. Therefore, if we describe five-nested `parallel` constructs, they correspond to all levels of PEZY-SC hierarchy one-to-one. Further, OpenMP provides `barrier` and `flush` directives which OpenACC does not. The directives correspond to sync and flush functions in PZCL, respectively and allow programmers to synchronize threads and flush cached data at each level of the hierarchy. However, the correspondences between OpenMP `parallel` constructs and the hierarchy levels are implicit when using less than five nested `parallel` constructs especially. In OpenACC, programmers can clarify parallelisms by using `gang`, `worker`, and `vector` clauses.

Therefore, specifying parallelism level extension for `parallel` construct can be considered for OpenMP, and more parallelism levels and synchronization and flushing directives extension can be considered for OpenACC.

7 Conclusion

In this paper, we preliminarily designed and implemented an OpenACC compiler for PEZY-SC to improve productivity. The compiler is based on our Omni OpenACC compiler, and we implemented it to translate OpenACC code to PZCL for PEZY-SC. In the evaluation, the performance of the OpenACC version was over 98 % at N-body and up to 88 % at NPB-CG of that of the PZCL version. At NPB-CG, some optimizations such as kernel merging and explicit thread switching were effective for improving performance. From the viewpoint of productivity, OpenACC outperforms PZCL because OpenACC allows programmers to offload work to accelerators by adding directives to the serial version of code, and the SLOC of OpenACC version are less than half of the PZCL version at both N-Body and NPB-CG.

In our future work, we will optimize our compiler using PEZY-SC-specific features. We plan to improve the translation of `kernels` construct and introduce an additional directive for thread switching.

Acknowledgment. The present study was supported in part by the JST/CREST program entitled "Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era" in the research area of "Development System Software Technologies for post-Peta Scale High Performance Computing".

References

1. The green500. http://www.green500.org
2. Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/
3. OpenACC-Standard.org. OpenACC Home. http://www.openacc.org
4. Tabuchi, A., Nakao, M., Sato, M.: A source-to-source OpenACC compiler for CUDA. In: an Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 178–187. Springer, Heidelberg (2014)
5. NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html
6. Reyes, R., López-Rodríguez, I., Fumero, J.J., de Sande, F.: `accULL`: an OpenACC implementation with CUDA and OpenCL support. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 871–882. Springer, Heidelberg (2012)
7. Tian, X., Xu, R., Yan, Y., Yun, Z., Chandrasekaran, S., Chapman, B.: Compiling a high-level directive-based programming model for GPGPUs. In: Caşcaval, C., Montesinos-Ortego, P. (eds.) LCPC 2013. LNCS, vol. 8664, pp. 105–120. Springer, Heidelberg (2014)
8. Lee, S., Vetter, J.S.: Openarc: open accelerator research compiler for directive-based, efficient heterogeneous computing. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC 2014, New York, NY, USA, pp. 115–120. ACM (2014)
9. University of Delaware and LLNL. RoseACC. http://roseacc.org/
10. GCC. OpenACC - GCC Wiki. https://gcc.gnu.org/wiki/OpenACC
11. RIKEN AICS and University of Tsukuba. Omni Compiler Project. http://omni-compiler.org
12. Warren, M.S., Salmon, J.K., Becker, D.J., Goda, M.P., Sterling, T., Winckelmans, W.: Pentium pro inside: I. a treecode at 430 gigaflops on asci red, ii. price/performance of $50/mflop on loki and hyglac. In: ACM/IEEE 1997 Conference on Supercomputing, p. 61, November 1997