An Extension of OpenACC Directives for Out-of-Core Stencil Computation with Temporal Blocking

Nobuhiro Miki, Fumihiko Ino, and Kenichi Hagihara
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {n-miki, ino}@ist.osaka-u.ac.jp

***Abstract*—In this paper, aiming at realizing directive-based temporal blocking for out-of-core stencil computation, we present an extension of OpenACC directives and a source-to-source translator capable of accelerating out-of-core stencil computation on a graphics processing unit (GPU). Out-of-core stencil computation here deals with large data that cannot be entirely stored in GPU memory. Given an OpenACC-like code, the proposed translator generates an OpenACC code such that it decomposes large data into smaller chunks, which are then processed in a pipelined manner to hide the data transfer overhead needed for exchanging chunks between the GPU memory and CPU memory. Furthermore, the generated code is optimized with a temporal blocking technique to minimize the amount of CPU-GPU data transfer. In experiments, we apply the proposed translator to three stencil computation codes. The out-of-core performance on a Tesla K40 GPU reaches 73.4 GFLOPS, which is only 13% lower than the in-core performance. Therefore, we think that our directive-based approach is useful for facilitating out-of-core stencil computation on a GPU.**

I. Introduction

Stencil computation is a class of iterative computation that frequently appears in scientific applications of a wide range of fields, including computational fluid dynamics [24], computational electromagnetics [1], image processing [5], and so on. This iterative computation typically updates array elements according to a fixed pattern, called stencil. For example, finite-difference methods have been widely used for solving partial differential equations that describe the time evolution of variables. In general, stencil applications are regarded as memory-intensive applications [6], [7], [19]. Therefore, many stencil codes have been implemented on accelerator devices, such as the graphics processing unit (GPU) [16] and Xeon Phi [8], which typically provide a 5 times higher memory bandwidth than the CPU.

Accelerator codes are usually developed with a unique programming language to maximize the performance on accelerator devices. For example, the compute unified device architecture (CUDA) [16] is widely used for NVIDIA GPUs. The CUDA code consists of *the host code* and *the device code*, each running on the CPU and the GPU, respectively. The key for achieving acceleration is to locate the performance bottleneck of the sequential code so that the bottleneck operations can be offloaded to the accelerator device with device-specific optimization techniques. However, this offloading process is time consuming because code and data structures usually have to be entirely rewritten to adapt themselves to the device architecture. Moreover, application developers are again enforced to rewrite their device-specific code when a new architecture will be released in the future. From this point of view, scientific codes should retain performance portability such that the codes achieve high performance on different architectures.

Such tremendous efforts can be minimized by directive-based programming (see Fig. 1), namely an emerging approach for developing parallel codes on an accelerator device. For example, OpenACC [20] provides a collection of compiler directives that are useful for offloading bottleneck workloads to an accelerator device. In other words, directives represent architecture-specific description, so that the architecture-dependent code is clearly separated from the generic code, which expresses the essentials of computation. Thus, OpenACC significantly lowers the barrier to accelerated computing. However, typical OpenACC-based implementations assume that the entire data are stored in device memory, which compromises the directive-based approach. Because the device memory generally has a smaller capacity than the host memory, OpenACC-based implementations usually fail to solve a large problem that has been processed by CPU-based implementations.

To relax this restriction, we developed a directive-based framework, named pipelined accelerator (PACC) [10], capable of pipelined execution of large-scale stencil computation on a GPU. According to PACC directives, our translator generates an OpenACC code such that large data are automatically decomposed into smaller chunks, which are then processed in a pipelined manner to overlap CPU-GPU data transfer with kernel execution. Thus, PACC directives are useful for solving a problem of the same size as that processed by CPU-based implementations. However, data decomposition increased the amount of CPU-GPU data transfer, which limited the performance of stencil applications, particularly those solving time evolution problems.

In this work, aiming at reducing the amount of data transfer between the CPU and GPU, we extend PACC directives [10]

WACCPD16; Salt Lake City, Utah, USA; November 2016
978-1-5090-6152-5/16/$31.00

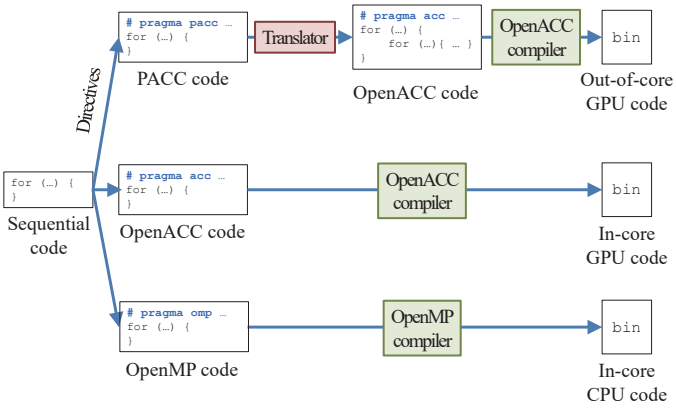


Fig. 1. Directive-based programming flow.

such that both data decomposition and temporal blocking can be automatically applied to stencil computation on a CUDA-compatible GPU. Given a sequential C code with PACC directives, our source-to-source translator generates an OpenACC code that can efficiently accelerate large-scale stencil computation with temporal blocking. The generated code can be rapidly processed on a GPU with several execution parameters such as the number of data decompositions. We apply the presented translator to three stencil computation codes [4] to investigate the impact of execution parameters.

The paper is structured as follows. Section II introduces related work regarding on acceleration of stencil computation. Section III presents an overview of temporal blocking and then summarizes how this optimization technique can be applied to an OpenACC code. Section IV describes the proposed PACC directives and translator. Section V shows experimental results. Finally, Section VI concludes the paper with future work.

II. Related Work

Maruyama *et al.* [12] presented a programming framework, named Physis, which provided a domain specific language (DSL) based solution to stencil computation. Given a DSL-based code, Physis generates a CUDA code for multi-node systems, where computing nodes communicate each other with the Message Passing Interface (MPI) standard [13]. Further, they extended Physis to automatically apply temporal blocking to the generated code [9]. This DSL-based approach is similar to our directive-based approach because both approaches facilitate acceleration of stencil computation. However, the directive-based approach is more useful for achieving this common goal with less efforts because directive-based approach keeps the original structure of the sequential code; there is no need to adapt the code to the target DSL.

Endo *et al.* proposed a run-time library, called hybrid hierarchical runtime (HHRT) [2], which allows CUDA+MPI codes to deal with out-of-core data with temporal blocking [3]. The HHRT library virtualized the GPU memory with automated swapping to the CPU memory. Because the HHRT library requires a CUDA code as its input, more efforts are necessary to achieve parallelization on the GPU. Our directive-based approach avoids this time-consuming process but execution efficiency can be degraded due to higher-level of code description. For example, low-level optimization such as intrinsic functions [16] is not explicitly available from an OpenACC code.

XcalableACC [15], a hybrid model of the OpenACC and a partitioned global address space (PGAS) language [14], [25], realizes directive-based programming for multi-node accelerator systems. Similar to our approach, their approach is useful for implementing a highly-efficient portable code by adding directives to a sequential code. Although the XcalableACC code efficiently runs on multi-node systems with minimum programming efforts, code structure must be modified to use temporal blocking, which is the key technique to minimize the amount of data transfer between the host and device.

OpenMP [21] provides a collection of directives that are useful for implementing multithreaded parallel applications on shared memory architectures such as multi-core CPUs. The latest OpenMP 4.5 supports accelerators with OpenACC-like directives. However, the host data cannot be partially mapped to the device data. Therefore, large problems cannot be solved due to device memory exhaustion.

Finally, NVIDIA has recently announced their latest architecture, called Pascal (P100) [18], which provides a single, unified virtual address space for CPU and GPU memory. This capability is useful to realize out-of-core stencil computation with a low effort. However, pipelined execution requires prefetching of chunks to overlap kernel execution with CPU-GPU data transfer. It is not clear whether such a software pipeline can be easily implemented with maintaining the original structure of the sequential code; the P100 architecture is not available at this time.

III. Stencil Computation and Temporal Blocking

Temporal blocking is a cache optimization technique for stencil applications that solve time evolution problems. This technique saves the memory bandwidth by performing reuse of on-cache data. To do so, the computational domain is decomposed into smaller blocks such that each block evolves k contiguous time steps at a time before proceeding to the next block. Hereafter, we call parameter k as *the blocking factor*. There are several approaches for realizing temporal blocking. In this paper, we consider an overlapped tiling approach for its simplicity. Other temporal blocking approaches such as wavefront temporal blocking and diamond tiling are useful for eliminating halos mentioned below (i.e., overheads in terms of computation at block boundaries).

Figure 2 shows an example code of four-point stencil computation. In this example, a four-point stencil is applied to $X \times Y$ elements stored in a two-dimensional (2-D) array (see Fig. 3); updating an element refers to itself together with its four neighbors. To realize block-based time evolution, the original `t` loop in Fig. 2 must be reorganized into double nested loops; the inner loop evolves k time steps within a block and the outer loop evolves every k time step for blocks.

```
1 for (t=0; t<T; t++) { // time evolution
2   for (x=1; x<X-1; x++) {
3     for (y=1; y<Y-1; y++)
4       q[x][y] = (p[x-1][y] + p[x+1][y] + p[x][y-1] + p[x][y
            +1]) * 0.25;
5   }
6   swap(p, q);
7 }
```

Fig. 2. Pseudocode of four-point stencil computation.

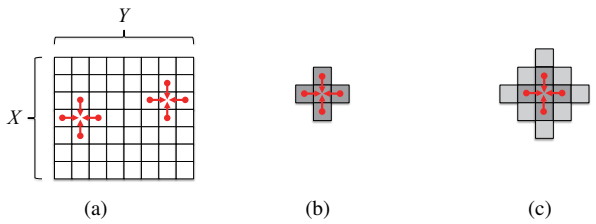


Fig. 3. Overview of stencil computation. (a) $X \times Y$ elements in the computational domain are updated with (b) a four-point stencil stored in a 3×3 region. (c) As compared with the original stencil in (b), applying temporal blocking refers more neighboring elements to compute the target element ($k = 2$, in this example).

Thus, temporal blocking exploits temporal locality to take advantage of caches, which have a low latency but with a limited capacity compared to the CPU memory. Such a tradeoff relation can be found between the CPU memory and GPU memory. Consequently, temporal blocking is frequently used for GPU-based implementations to reuse data blocks that have been transferred to the GPU memory. Evolving k time steps within the transferred data blocks reduces the amount of CPU-GPU data transfer to $1/k$.

Notice here that multiple updates within a block require *a halo region* around the block (see Fig. 4) because updating an element refers to its surrounding neighbors. Without this halo region, blocks cannot be processed independently, and thereby, preventing full parallelization. Suppose that a cross stencil is stored in a $(2r+1)\times(2r+1)$ region, i.e., updating an element refers to itself and its r neighbors in up/down/left/right directions: $r = 1$ for a four-point stencil, as depicted in Fig. 3(b). In this case, k updates for an element refers to its $(2rk+1)\times(2rk+1)$ region (Fig. 3(c)). This means that temporal blocking increases the amount of computation because halo regions of adjacent blocks overlap each other. Thus, there is a tradeoff relation between the amount of computation and the degree of available parallelism. Consequently, the blocking factor k must be optimized to maximize the performance gain of temporal blocking.

Hereafter, we use the term *chunk* to denote the region that contains a block and its overlapping halo region (Fig. 4).

A. OpenACC-based implementation

In this section, we present how data decomposition and temporal blocking can be applied to an OpenACC-based stencil code. In other words, the OpenACC code presented here is the output of our translator, which requires a PACC code as its input.

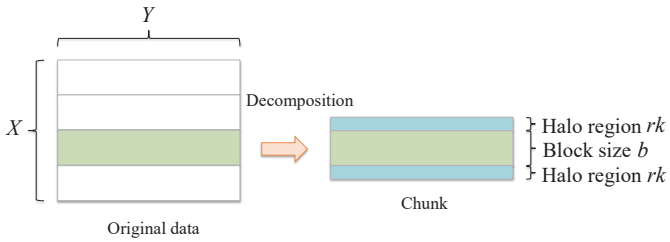


Fig. 4. A 1-D block decomposition scheme with halo region. Given a stencil of $(2r+1)\times(2r+1)$ elements, each block requires halos of size $rk \times Y$ to evolve k time steps for all elements within the block.

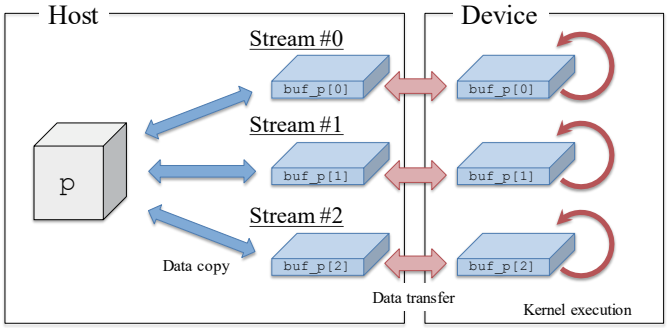


Fig. 5. A copy-based scheme for data decomposition. Host and device buffers are allocated for copying blocks and their halos from the original array.

1) Data decomposition: Data decomposition cannot be realized by simply adding OpenACC directives into the sequential code. Given a large array that exceeds the capacity of the device memory, a different array must be allocated in the host memory, as depicted in Fig. 5. This is due to the assumption of the OpenACC specification, which allocates the same variables on both the host memory and device memory. In this figure, large array `p` can be successfully processed on the GPU by copying chunks (i.e., blocks and their halos) to different buffers `buf_p`; memory exhaustion can be avoided if the total buffer size is smaller than the capacity of device memory. One drawback of this copy-based scheme is the copy overhead incurred on the host. However, this overhead can be hidden by pipelined execution, which overlaps the overhead with data transfer and kernel execution.

An alternative solution for avoiding device memory exhaustion is a map-based scheme, which maps the allocated device buffers to the original host array. This direct mapping allows GPU threads to update values in the original array, so that the host buffer (i.e., the copy overhead) can be eliminated. However, we decided to use the copy-based scheme because the map-based scheme failed to asynchronously transfer data on our experimental environment (see Section V); pipelined execution was not available with this scheme. Another drawback of the map-based scheme is that it may fail to realize multidimensional decomposition, which produces many mapping points; direct mapping is mainly designed for a contiguous memory region.

Figure 6 shows a pseudocode of OpenACC-based out-of-

```
1  Set blocking factor k, number d of decompositions, and number num_queue of queues from environmental variables;
2  Compute block size b from d and data size X;
3  Allocate buf_p[0], buf_p[1], ..., buf_p[num_queue-1] on host memory;
4  #pragma acc create (buf_p[0:num_queue][0:b+2*r*k], ...)
5  Allocate g[0], g[1], ..., g[num_queue-1] on host memory to store completed chunk numbers;
6
7  for (t=0; t<T; t+=k) { // outer loop for time evolution
8    for (c=0; c<d; i = (i+1) % num_queue) { // for each chunk
9
10     if (!acc_async_test(i)) // Select i as the ID of an idle queue
11       continue;
12
13     if (g[i] != NONE) {
14       Copy the g[i]-th chunk from buf_p[i] to p;
15       g[i] = NONE;
16     }
17
18     Copy the c-th chunk from p to buf_p[i];
19     #pragma acc update device (buf_p[i:1][0:b+2*r*k], ...) async (i)
20
21     for (s=0; s<k; s++) { // inner loop for time evolution
22       #pragma acc kernels present (buf_p[i:1][0:b+2*r*k], ...) async(i)
23       {
24         offset = r*(s+1);
25         xsize  = b+2*r*(k-1-s);
26
27         #pragma acc loop independent
28         for (x=offset; x<offset+xsize; x++)
29           #pragma acc loop independent
30           for (y=1; y<Y-1; y++)
31             #pragma acc loop independent
32             for (z=1; z<Z-1; z++)
33               buf_q[i][x*Y*Z+y*Z+z] += buf_p[i][(x+1)*Y*Z+y*Z+z] + ...;
34       }
35       #pragma acc kernels present (buf_p[i:1][0:b+2*r*k], ...) async(i)
36       {
37         #pragma acc loop independent
38         for (x=offset; x<offset+xsize; x++)
39           #pragma acc loop independent
40           for (y=1; y<Y-1; y++)
41             #pragma acc loop independent
42             for (z=1; z<Z-1; z++)
43               buf_p[i][x*Y*Z+y*Z+z] = buf_q[i][(x+1)*Y*Z+y*Z+z];
44       }
45     }
46
47     #pragma acc update host (buf_p[i:1][0:b+2*r*k], ...) async (i)
48     g[i] = c;
49     c++;
50   }
51 }
52 for (i=0; i<num_queue; i++) {// Clean up all queues
53   #pragma acc wait(i)
54   if (g[i] != NONE)
55     Copy the g[i]-th chunk from buf_p[i] to p;
56 }
```

Fig. 6. Pseudocode of out-of-core stencil computation with temporal blocking. This pseudocode applies a cross stencil of $(2r+1) \times (2r+1)$ elements to the computational domain of $(X-2r) \times Y \times Z$ elements. A 1-D block decomposition scheme is applied to the computational domain, so that the data is decomposed into d blocks of size $(X-2r)/d \times Y \times Z$.

core stencil computation that deploys a 1-D block decomposition scheme and temporal blocking. At line 3, `num_queue` buffers, `buf_p`, are allocated for the original array `p`, where `num_queue` is the number of queues used for pipelining. Similarly, device buffers with the same variable names are allocated by the `create` clause at line 4. Note that multiple buffers are necessary to realize efficient pipelining; a single buffer causes data dependence between the kernel execution and host-device data transfer, which avoids overlapped execution. Note also that these buffers are reused to minimize the allocation overhead during program execution; the buffers are allocated once at the beginning of an execution. At line 5, `num_queue` buffers, `g`, are allocated to store chunk numbers.

After this allocation, chunks are copied from the original array `p` to the host buffers `buf_p`, which are then transferred to the device buffers `buf_p` by using the `update` clause at line 19. Using these buffers, a kernel function is invoked at line 22,35 to update elements in the chunks for k time steps. To do this, a `kernels` construct is deployed to specify a code block to be offloaded from the host to the device. In addition, the `present` clause at line 22,35 indicates that chunks to be accessed have already been sent to the device buffers `buf_p`, so that additional data transfer can be avoided at kernel invocation. The updated elements are then transferred back to the host buffers at line 47. Host buffers are copied to the original array `p` at line 14 when the same queue is selected

next time.

Notice that the code modification mentioned above is necessary to avoid device memory exhaustion, which results in an execution failure. Without this code modification, the original code fails to run if array `p` is too large to fit in the device memory. However, as shown in Fig. 6, this modification reorganizes the loop structure of the original code (Fig. 2), diminishing the benefits of directives. That is, the modified code degrades the performance portability because it leads to inefficient run on other machines equipped with a large-capacity memory.

2) Temporal blocking: Efficient temporal blocking can be implemented by realizing the following points.

- Block-based time evolution. The time evolution loop must be separated into two loops such that one is responsible for intra-block and the other is responsible for inter-block, as shown in Fig. 6; the outer `t` loop at line 7 is responsible for processing blocks at every k time steps while the inner `s` loop at line 21 is responsible for processing a block for consecutive k time steps. To facilitate automated parallelization, `x`, `y`, and `z` loops in the code block have a `loop` construct with an `independent` clause, which notifies the OpenACC compiler that iterations can be efficiently executed without synchronization.
- Pipelined execution. Asynchronous APIs are deployed to realize software-based pipelining that overlaps kernel execution with host-device data transfer. In Fig. 6, a chunk is assigned to one of the `num_queue` asynchronous queues. At line 10, a queue is tested whether all associated operations of the queue have completed or not using an `acc_async_test` API. That is, chunks are transferred at line 19 with an `async` clause and its argument `i`, which specifies the queue ID to be used. As mentioned before, the host and device buffers `buf_p[i]` are dedicated to the `i`-th queue for realizing concurrent execution. Thus, different queues have no data dependence between their tasks.

With respect to the tasks to be queued, each queue is responsible for processing the following steps in order.

1) Data copy step. The host copies a chunk from the original array to the host buffer (line 18).
2) Data transfer step (host to device). The host transfers the copied chunk from the host buffer to the device buffer (line 19).
3) Kernel execution step. The device iteratively executes the kernel to evolve the transferred chunk for k time steps (lines 21–45).
4) Data transfer step (device to host). The updated chunk is transferred from the device buffer to the host buffer (line 47).
5) Data copy step. The host copies the updated chunk to the original array (line 14).

Note that the pseudocode code of Fig. 6 cannot overlap the first step with the last step because it uses a single CPU thread to copy data. However, the kernel execution step can be overlapped with the data transfer if chunks are assigned to different queues. A full overlap can be established with multiple CPU threads, but we leave this issue for a future work.

```
1  #pragma pacc init
2
3  #pragma pacc pipeline targetinout(p,q) size([0:X][0:Y])
       halo([1:1][1:1]) async
4  for (t=0; t<T; t++){
5    #pragma pacc loop dim(2)
6    for (x=1; x<X-1; x++)
7      #pragma pacc loop dim(1)
8      for (y=1; y<Y-1; y++)
9        q[x][y] = (p[x-1][y] + p[x+1][y] + p[x][y-1] + p[x
             ][y+1]) * 0.25;
10
11   #pragma pacc loop dim(2)
12   for (x=1; x<X-1; x++)
13     #pragma pacc loop dim(1)
14     for (y=1; y<Y-1; y++)
15       p[x][y] = q[x][y];
16 }
```

Fig. 7. An example of a PACC code.

IV. Pipelined Accelerator (PACC)

The proposed PACC directives allow application developers to specify key information, such as the stencil size and the data to be decomposed, which are required to automate applying data decomposition and temporal blocking. Moreover, several execution parameters, such as the blocking factor k and the number d of decompositions, can be flexibly specified by environmental variables at the beginning of a program execution.

Our translator assumes that the target stencil code satisfies the following constraints.

- Constraints on data decomposition. Application developers are prohibited to manually decompose data in the PACC code. The translator also assumes that the data are small enough to be stored in the host memory.
- Constraints on temporal blocking. The number T of total time steps must be fixed at the beginning of a program execution. Accordingly, the target program is prohibited to use a `while` loop to terminate its execution according to a user-defined threshold on the error. Furthermore, the device evolves data with k time steps at a time, so that the intermediate values between every k time steps cannot be accessed from the host.

A. PACC directive

The proposed PACC extends OpenACC [20] with three constructs: the `init`, `pipeline`, and `loop` constructs. Figure 7 shows an example of a PACC code that implements out-of-core stencil computation with temporal blocking. Similar to OpenACC directives, a PACC directive starts with `#pragma pacc`. The extended directives are as follows.

- The `init` construct. This construct allocates host and device buffers for realizing data decomposition (see Section III-A). Consequently, the `init` construct must be placed

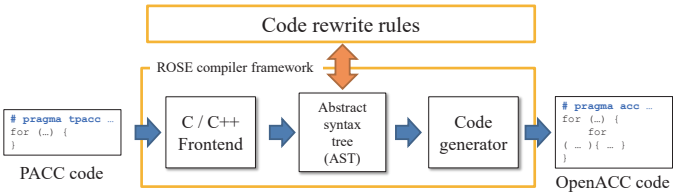


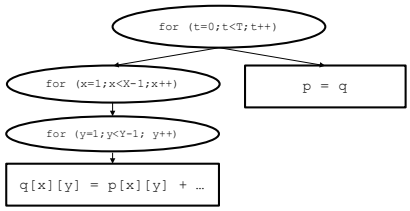
Fig. 8. An overview of the proposed translator.

before the `pipeline` construct and the `loop` construct mentioned below.

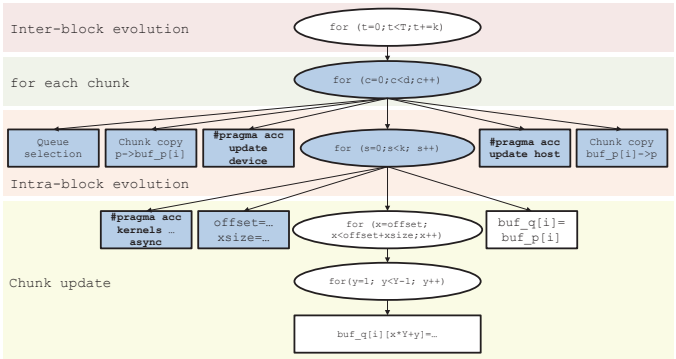
- The `pipeline` construct. The `pipeline` construct specifies the code block to be processed in a pipeline. In Fig. 7, the `pipeline` construct is applied to the `for` loop at line 4, which is responsible for time evolution. This construct is similar to the `data` construct of OpenACC and can have additional clauses such as the `targetin`, `targetinout`, `size`, and `halo` clauses. The `targetin` and `targetinout` clauses define read-only and read/write variables, respectively. The `size` clause defines an array range specification with start and length for each dimension. For example, Fig. 7 specifies `size([0:X][0:Y])` at line 3 to update all elements in an array of size $X \times Y$. Finally, the `halo` clause defines the stencil size for each dimension. In Fig. 7, `halo([1:1][1:1])` defines a four-point stencil, which accesses left/right/up/down neighbors to update an element. Finally, the `async` clause declares that the code block must be asynchronously processed to realize pipelined execution.
- The `loop` construct. The PACC `loop` construct is an extension of the OpenACC `loop` construct. The extended construct can have an extended clause, `dim`, which associates the loop control variable with the dimension of the array data. For example, the `dim(2)` clause at line 5 indicates that the loop control variable `x` at line 6 corresponds to the second dimension of the array data; we assume here that the first (last) dimension has the minimum (maximum, respectively) stride between consecutive elements. This association is used for adjusting the loops for decomposed data. Our translator currently deploys a 1-D block scheme that decomposes the array data in terms of the last dimension. In Fig. 7, the array data will be decomposed along the `x` axis.

B. PACC translator

Our PACC translator was implemented using the ROSE compiler infrastructure [11], [22], which provides a C/C++ frontend to generate an abstract syntax tree (AST) of the input code (see Fig. 8). The generated AST is then traversed to detect PACC directives, i.e., the nodes with the `pacc` attribute. During this traverse, detected directives are parsed to retrieve the key information given by succeeding clauses, such as stencil size, the variables to be decomposed, and their data size. The detected nodes are marked explicitly for



(a)



(b)

Fig. 9. An example of AST transformation. (a) The presented translator generates an AST from a PACC code. (b) The translator then applies code rewrite rules to obtain an OpenACC code that can deal with large data with data decomposition and temporal blocking.

modification, so that code rewrite rules are applied to them in the next traversal. Finally, the modified AST is given to a code generator to obtain the pipelined OpenACC code.

The code rewrite rules can be summarized as follows.

1) The rule for the `init` construct. The AST node that corresponds to the `init` construct is replaced with AST nodes that are responsible for (1) obtaining the blocking factor k and the number d of data decompositions via environmental variables, (2) computing the block size b from d and the array size X to be decomposed, and (3) allocating host and device buffers.
2) The rule for the `pipeline` construct. The original `for` loop is reorganized into double nested `for` loops to realize temporal blocking. To achieve this, as shown in Fig. 9, the code rewrite rule replaces the AST node that corresponds to the `for` loop of time evolution with AST nodes that correspond to (1) the `for` loop of inter-block evolution, (2) that of per-chunk operations, and (3) that of intra-block evolution. Furthermore, several AST nodes are added as children of the second AST node (i.e., per-chunk operations) to select an idle queue, copy chunks on the host buffers, and exchange them between the host and device buffers. Finally, memory accesses to the original arrays are replaced with those to the buffers by updating the attributes of AST nodes appropriately.
3) The rule for the `loop` construct. This rule updates the transformed double nested loops with new initialization and condition to adapt the loop structure for block-based evolution. To do this, the rule first locates the `for` loop that must be updated due to data decomposition. For

TABLE I
STENCIL COMPUTATION CODES USED FOR EXPERIMENTS. NOTATIONS T, f, l, s, AND G REPRESENT THE NUMBER OF TIME STEPS, THAT OF FLOATING POINT OPERATIONS PER ELEMENT, THE AMOUNT OF WRITES PER ELEMENT, THE AMOUNT OF READS PER ELEMENT, AND THE ARITHMETIC INTENSITY [23], RESPECTIVELY. THE ARITHMETIC INTENSITY G IS GIVEN BY $G = f/(l + s)$.

Code	Number of arrays	Array size	Data size (GB)	Stencil	T (step)	f (FLOP)	l (B)	s (B)	G (FLOP/B)
Jacobi	2	$48,000 \times 48,000$	18.4	4 point	2048	4	16	4	0.20
Himeno	14	$512 \times 512 \times 1024$	15.0	18 point	256	34	128	4	0.26
CIP	8	$22,000 \times 22,000$	15.5	9 point	256	91	120	12	0.69

example, the `for` loop at line 6 of Fig. 7 is selected for updating its initialization and condition because this loop is associated with a `dim(2)` clause that has the largest value (`2`) as the argument of `dim` clauses. The new initialization and condition is appropriately given by variables `offset` and `xsize`, as shown in Fig. 6.

V. EXPERIMENTAL RESULTS

We added PACC directives to three stencil computation codes and evaluated their performance on an experimental machine. Our experimental machine had an Intel Xeon E5-2680 v2 processor, 512 GB main memory, and a Tesla K40 GPU with 12 GB device memory. The Tesla K40 GPU had two direct memory access (DMA) engines, so that data transfer from the host to device was overlapped with that of the opposite direction. We used the PGI Compiler 15.5 [17], CUDA 7.0 [16], and Ubuntu 15.3.

The stencil computation codes used for experiments were the Jacobi method, Himeno benchmark [4], and constraint interpolation profile (CIP) method [26] summarized in Table I. The Jacobi method is an iterative solver for a system of linear equations. The Himeno benchmark is a linear solver for 3-D pressure Poisson equations. The CIP method is a solver for hyperbolic partial differential equations. All the three experimental codes processed at least 15 GB of data, which could not be stored entirely on the device memory.

A. Performance analysis

We first manually implemented PACC codes by adding PACC directives appropriately to the sequential codes. The PACC codes were then given to the proposed translator to automatically generate out-of-core OpenACC codes. Using the generated codes, we measured their effective performance to investigate the impact of execution parameters such as the blocking factor k and the block size b.

1) Jacobi method: We solved a problem size of $X \times Y = 48,000 \times 48,000$ elements, which consumed 18.4 GB of memory space, with the total time steps of $T = 2048$. Figure 10(a) shows the effective performance E given by $E = 4(X-2)(Y-2)T/t$, where t is the execution time, which includes the data transfer time between the CPU and GPU.

In Fig. 10(a), the maximum performance of the Jacobi method was 28.5 GFLOPS, obtained when the block size and the blocking factor were $b = 8000$ and $k = 32$, respectively. This figure also shows that the performance was determined by the blocking factor k rather than the block size b.

We then measured the effective performance with varying the blocking factor k at fixed block size $b = 8000$ (Fig. 11(a)). The effective performance did not monotonically increase with the blocking factor k due to the tradeoff relation mentioned in Section III. To investigate this behavior, we analyzed the breakdown of execution time, which we show in Fig. 12(a). We found that the copy overhead determined the entire performance when $k < 32$. This overhead was inversely proportional to k, so that the effective performance increased with k when $k < 32$. In contrast, the effective performance slightly decreased as we increased k from 32. This performance degradation was caused by temporal blocking, which increased kernel execution time due to redundant computation. Thus, the best tradeoff point was obtained when $k = 32$, where the data transfer and copy overheads were fully overlapped with kernel execution time.

2) Himeno benchmark: We solved the problem size XL ($512 \times 512 \times 1024$, 15 GB) with $T = 256$ time steps. Given 3-D data of $X \times Y \times Z$ elements, the effective performance E is given by $E = 34(X-2)(Y-2)(Z-2)T/t$.

In Fig. 10(b), the effective performance of the Himeno benchmark monotonically increased with the blocking factor k. As a result, the maximum performance of 37.5 GFLOPS was obtained when $b = 102$ and $k = 16$, where the buffer size was maximized; when $k > 16$, we failed to execute the PACC code due to device memory exhaustion. A larger memory capacity was required to find the best blocking factor that might be obtained when $k > 16$.

Thus, the tradeoff point was not clearly observed for the Himeno benchmark because we failed to increase the blocking factor k from 16. In other words, device memory exhaustion occurred though the data were decomposed for saving memory consumption. This issue can be resolved by realizing multidimensional decomposition. With our 1-D decomposition scheme, a chunk consists of $(b+2rk) \times Y \times Z$ elements. Therefore, the amount of memory consumption linearly increased with Y and Z, which restricted the blocking factor k such that $k \leq 16$. Therefore, a multidimensional decomposition scheme is necessary for our experimental machine to successfully run the benchmark with $k > 16$.

As for multidimensional decomposition, data pack and unpack procedures are required to retrieve a small multidimensional array from a large multidimensional array. Therefore, the PACC translator must be extended such that it can (1) pack several data segments into a host buffer and (2) unpack a device buffer onto the original array. Furthermore, memory references in the kernel must be updated to access the packed

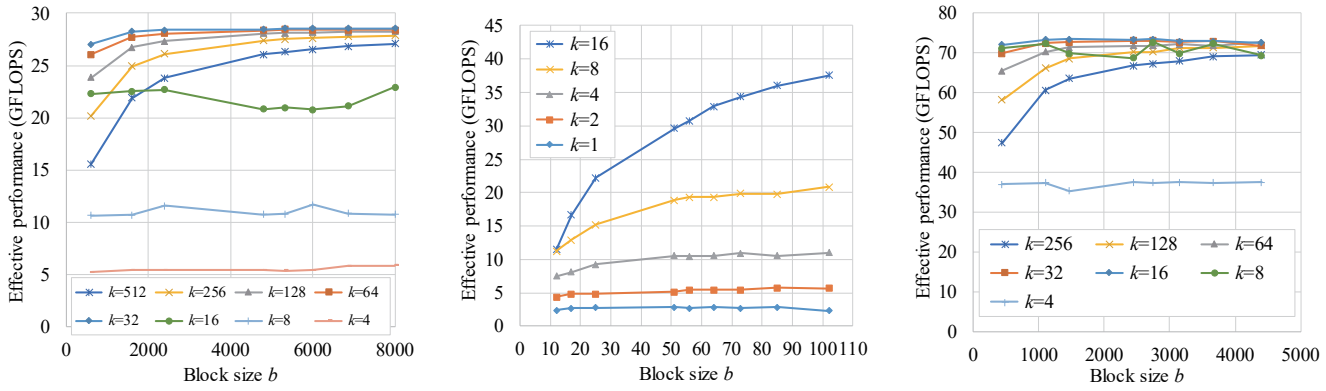


Fig. 10. Effective performance with different block size b and blocking factor k. Results for (a) Jacobi method, (b) Himeno benchmark, and (c) CIP method.

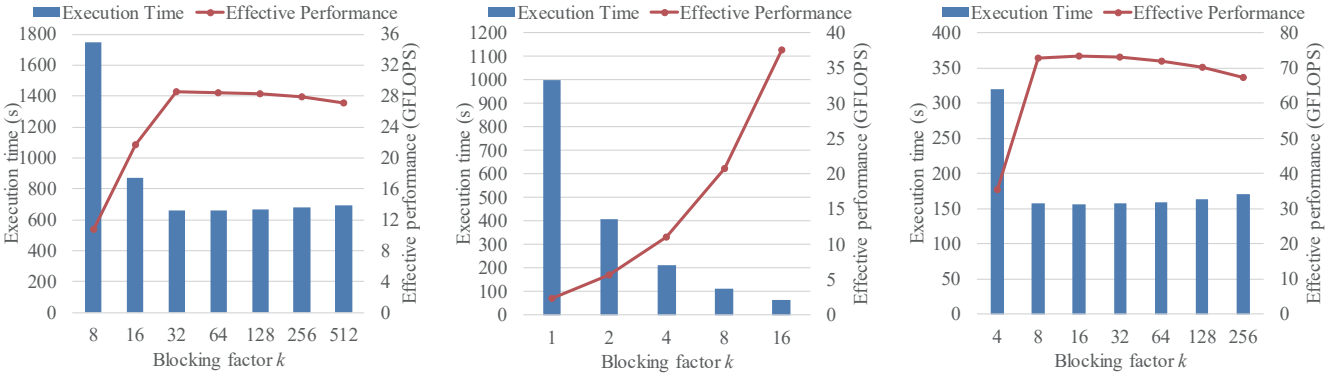


Fig. 11. Effective performance with different blocking factor k for fixed block size b. Results for (a) Jacobi method, (b) Himeno benchmark, and (c) CIP method.

small array accordingly.

Figure 12(b) shows the breakdown of execution time. We found that the data copy time spent on the host was almost the same as the kernel execution time when $k = 16$. Consequently, the blocking factor of $k = 16$ seems to be the best configuration for the Himeno benchmark.

3) CIP method: We solved a problem size of $22,000 \times 22,000$ elements with $T = 256$ time steps. Given 2-D data of $X \times Y$ elements, the effective performance E is given by $E = 91(X-2)(Y-2)T/t$.

In Fig. 10, the effective performance reached 73.4 GFLOPS when $b = 2750$ and $k = 16$. As compared with the Jacobi method and the Himeno benchmark, the CIP method maximized its performance with relatively a small blocking factor, $k = 8$, due to its high arithmetic intensity [23] (see Table I). That is, the ratio of the kernel execution time over the CPU-GPU data transfer time was relatively high, and thereby, the impact of temporal blocking was rapidly maximized with low k. Therefore, temporal blocking failed to demonstrate a significant improvement when $k \geq 8$.

Figures 11(c) and 12(c) show the effective performance with different blocking factor k and the breakdown of execution time, respectively. As we estimated above, the best tradeoff point was found at $k = 8$ in Fig. 11(c), where the copy overhead was close to the kernel execution time, as shown in Fig. 12(c).

B. Comparison with in-core implementation

Finally, we compared our out-of-core implementation with an in-core implementation that processed small data (Table II). The in-core versions of the Jacobi, Himeno, and CIP methods solved problem sizes of $24,000 \times 24,000$ (4.6 GB), $128 \times 128 \times 256$ (0.2 GB), $16,000 \times 16,000$ (8.2 GB), respectively. Each in-core data was iteratively updated with the same number of time steps as the out-of-core implementation. Note here that the effective performance was derived from the execution time that included the data transfer time between the CPU and GPU.

The highest in-core performance reached 83.9 GFLOPS, which was 13% higher than the performance achieved by our out-of-core implementation. Because data transfer time usually limits the performance of GPU applications, we think this 13% slowdown is acceptable for realizing highly-efficient, out-of-core stencil computation with a directive-based approach.

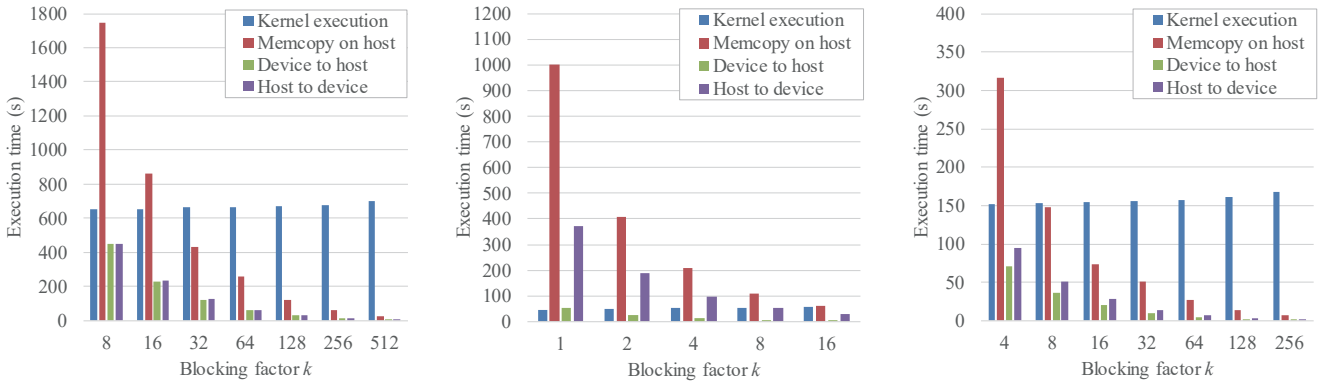


Fig. 12. Breakdown of execution time with different blocking factor k for fixed block size b. Results for (a) Jacobi method, (b) Himeno benchmark, and (c) CIP method.

TABLE II
COMPARISON OF EFFECTIVE PERFORMANCE OF IN-CORE AND OUT-OF-CORE IMPLEMENTATIONS.

Code	In-core p_1 (GFLOPS)	Out-of-core p_2 (GFLOPS)	Ratio p_2/p_1 (%)
Jacobi	32.2	28.5	89
Himeno	47.5	37.5	79
CIP	83.9	73.4	87

VI. CONCLUSION

We presented an extension of OpenACC directives, named PACC, and its source-to-source translator capable of accelerating out-of-core stencil computation with temporal blocking on a GPU. Given a PACC code, our translator generates an OpenACC code such that the code decomposes large data into smaller chunks, which are then processed in a pipelined manner to hide the copy overhead incurred on the CPU. Furthermore, the generated code is accelerated with temporal blocking, which reduces the amount of data transfer between the CPU and GPU.

In experiments, we added PACC directives to three stencil computation codes, the Jacobi, Himeno, and CIP methods. We found that the out-of-core performance reached 73.4 GFLOPS on a Tesla K40 GPU, which was only 13% lower than the in-core performance. Thus, PACC directives not only facilitate out-of-core stencil computation on a GPU but also achieve high performance on a GPU.

Future work includes an automated framework for finding the best execution parameters b and k, namely the block size and the blocking factor.

ACKNOWLEDGMENTS

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15K12008, 15H01687, 16H02801, and the Japan Science and Technology Agency CREST program, “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems.” The authors would like to thank the anonymous reviewers for helpful comments to improve their paper.

REFERENCES

[1] S. Adams, J. Payne, and R. Boppana. Finite difference time domain (FTDT) simulations using graphics processors. In *Proc. High Performance Computing Modernization Program Users Group Conf. (HPCMP-UGC'07)*, pages 334–338, June 2007.

[2] T. Endo and G. Jin. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. In *Proc. 16th IEEE Int. Conf. Cluster Computing (CLUSTER'14)*, pages 132–139, Sept. 2014.

[3] T. Endo, Y. Takasaki, and S. Matsuoka. Realizing extremely large-scale stencil applications on GPU supercomputers. In *IEEE Trans. Parallel and Distributed Systems*, pages 625–632, Dec. 2015.

[4] R. Himeno. Himeno benchmark, 2015. http://accc.riken.jp/en/supercom/himenobmt/.

[5] K. Ikeda, F. Ino, and K. Hagihara. Efficient acceleration of mutual information computation for nonrigid registration using CUDA. *IEEE J. Biomedical and Health Informatics*, 18(3):956–968, May 2014.

[6] T. Ikuzawa, F. Ino, and K. Hagihara. Reducing memory usage by the lifting-based discrete wavelet transform with a unified buffer on a gpu. *J. Parallel and Distributed Computing*, 93/94:44–55, July 2016.

[7] F. Ino, Y. Munekawa, and K. Hagihara. Sequence homology search using fine grained cycle sharing of idle GPUs. *IEEE Trans. Parallel and Distributed Systems*, 23(4):751–759, Apr. 2012.

[8] Intel Corporation. Intel Xeon Phi product family.

[9] G. Jin, M. Wahib, N. Maruyama, T. Endo, and S. Matsuoka. Locality optimizations for stencil computations: Algorithms and implementations. In *1st Workshop Programming Abstractions for Data Locality (PADAL'14)*, Apr. 2014.

[10] T. Kato, F. Ino, and K. Hagihara. PACC: An extension of OpenACC for pipelined processing of large data on a GPU. In *Poster 27th Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'14)*, Nov. 2014.

[11] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Proc. 23rd Int'l Workshop Languages and Compilers for Parallel Computing (LCPC'10)*, pages 308–322, Oct. 2010.

[12] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'11)*, Nov. 2011. 12 pages.

[13] Message Passing Interface Forum. MPI: A message-passing interface standard. *Int'l J. Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, 1994.

[14] H. Murai and M. Sato. An efficient implementation of stencil communication for the XcalableMP PGAS parallel programming language. In *Proc. 7th International Conference on PGAS Programming Models (PGAS'13)*, pages 142–156, Oct. 2013.

[15] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato. XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In *Proc. 2nd Workshop Accelerator Programming using Directives (WACCPD'14)*, pages 27–36, Nov. 2014.

[16] NVIDIA Corporation. CUDA C Programming Guide Version 7.0, March 2015.

[17] NVIDIA Corporation. PGI compiler, 2015.

[18] NVIDIA Corporation. NVIDIA Tesla P100, June 2016.

[19] D. Okada, F. Ino, and K. Hagihara. Accelerating the Smith-Waterman algorithm with an interpair pruning method for all-pairs comparison of base sequences. *BMC Bioinformatics*, 16(321), Oct. 2015. 15 pages.

[20] OpenACC-Standard.org. The OpenACC application programming interface, version 2.5, Oct. 2015.

[21] OpenMP Architecture Review Board. OpenMP application programming interface, version 4.5, Nov. 2015.

[22] rosecompiler.org. ROSE compiler infrastructure, 2015. http://rosecompiler.org/.

[23] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, 52(4):65–76, Apr. 2009.

[24] E. Wu, Y. Liu, and X. Liu. An improved study of real-time fluid simulation on GPU. *Computer Animation and Virtual Worlds*, 15(3/4):139–146, July 2004.

[25] xcalablemp.org. XcalableMP. http://www.xcalablemp.org/.

[26] T. Yabe, F. Xiao, and T. Utsumi. The constrained interpolation profile method for multiphase analysis. *J. Computational Physics*, 169(2):556–593, May 2001.