PAPER Special Section on Parallel, Distributed, and Reconfigurable Computing, and Networking

A Data-Centric Directive-Based Framework to Accelerate Out-of-Core Stencil Computation on a GPU

Jingcheng SHEN[†a)], ***Nonmember***, **Fumihiko INO**[†], ***Member***, **Albert FARRÉS**[††], ***and*** **Mauricio HANZICH**[††], ***Nonmembers***

SUMMARY Graphics processing units (GPUs) are highly efficient architectures for parallel stencil code; however, the small device (i.e., GPU) memory capacity (several tens of GBs) necessitates the use of out-of-core computation to process excess data. Great programming effort is needed to manually implement efficient out-of-core stencil code. To relieve such programming burdens, directive-based frameworks emerged, such as the pipelined accelerator (PACC); however, they usually lack specific optimizations to reduce data transfer. In this paper, we extend PACC with two data-centric optimizations to address data transfer problems. The first is a direct-mapping scheme that eliminates host (i.e., CPU) buffers, which intermediate between the original data and device buffers. The second is a region-sharing scheme that significantly reduces host-to-device data transfer. The extended PACC was applied to an acoustic wave propagator, automatically extending the length of original serial code 2.3-fold to obtain the out-of-core code. Experimental results revealed that on a Tesla V100 GPU, the generated code ran 41.0, 22.1, and 3.6 times as fast as implementations based on Open Multi-Processing (OpenMP), Unified Memory, and the previous PACC, respectively. The generated code also demonstrated usefulness with small datasets that fit in the device capacity, running 1.3 times as fast as an in-core implementation.

key words: *stencil computation, out-of-core computation, data-centric optimizations, GPU*

1. Introduction

Stencil computation is one of the most important classes in scientific computing with a key principle of iteratively updating an input array by applying a fixed calculation pattern (i.e., *stencil*) to all the elements of the array. Stencil applications appear in a wide range of fields, including geophysics simulations [1], [2], computational electromagnetics [3], and image processing [4], [5]. Because computation time and memory consumption grow linearly with the size of input arrays, parallel implementations of stencil computation are of great importance [6]. Currently, the graphics processing unit (GPU) is considered to be the most efficient architecture for parallel stencil code [7]. Armed with thousands of cores and 5–10 times higher memory bandwidth than CPUs, GPUs provide powerful solutions for both compute- and memory-intensive scientific problems [8]–[11]. However, there are two main challenges in implementing GPU-accelerated stencil code: limited capacity of device (i.e., GPU) memory and considerable programming effort to implement GPU-accelerated code.

At several tens of GBs, the capacity of device memory is relatively small. Out-of-core methods are thus a straightforward option to process excess data. Although multi-node solutions are also used to handle large data, they are complex due to the need to reconcile intra- and inter-node programming models, such as Message Passing Interface (MPI) [12] and Open Multi-Processing (OpenMP) [13]. Out-of-core methods decompose large data into smaller chunks such that each chunk fits in the device memory. As a result, out-of-core methods involve frequently moving data between the host (i.e., CPU) and device. Therefore, the programmer must deliberately organize data transfer to and from the device; otherwise, the performance of parallel code is limited by data transfer.

Significant programming effort is required for the GPU-based parallelization of serial code, and the programming effort further increases if out-of-core computation needs to be implemented to handle excess data. Typically, the compute unified device architecture (CUDA) [14] is used as a parallel programming framework for NVIDIA GPUs. To develop efficient CUDA programs, programmers must possess an in-depth knowledge of GPU-specific optimization techniques that adapt serial code and data structures to the highly parallel device [15]. To reduce the programming effort to develop GPU-based out-of-core code, directive-based programming frameworks such as Open Accelerators (OpenACC) [16] have emerged as alternatives to CUDA. OpenACC provides users with a collection of compiler directives that can be simply inserted into the serial code. Such directives instruct the OpenACC compiler to automatically offload parallelizable workloads from the host to a parallel accelerator, such as a GPU. However, due to the small device memory capacity, the simplicity of OpenACC (i.e., sharing of identical code and data structures between the host and device) can be problematic. For example, assume that we use a 100 GB array in a CPU-based code. If we naively insert OpenACC directives into this code, the OpenACC compiler will fail to prepare an array of the same size on the device due to device memory exhaustion.

Out-of-core computation is necessary to solve the aforementioned memory exhaustion problems. Miki *et al.* [17] proposed the pipelined accelerator (PACC),

Manuscript received December 24, 2019.
Manuscript revised May 24, 2020.
Manuscript publicized September 7, 2020.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
[††]The authors are with the Barcelona Supercomputing Center, Barcelona 08034, Spain.
a) E-mail: jc-shen@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2020PAP0014

a directive-based framework that automatically generates out-of-core OpenACC code for large stencil applications. PACC decomposes large data into smaller chunks, overlaps data transfer with kernel execution, and performs *temporal blocking* to process on-device data for multiple times. However, the generated code failed to exploit the computational capabilities of state-of-the-art GPUs because the data transfer limits the performance, especially when the generated code was high-order stencil computation in which the calculation of an element relies on a wide neighboring area. Although the latest Tesla V100 GPUs have notably improved memory bandwidth (900 GB/s), host-device data transfer consumes significantly more time than kernel execution due to relatively slow interconnects (e.g., bandwidth of 16 GB/s for PCIe 3.0) and thus limits the performance. Moreover, the gap between device memory bandwidth and PCIe bandwidth widens over time [11]; therefore, we must focus more on the optimization for data transfer than computation. Thus, new data-centric optimizations are needed to evolve the PACC framework on the latest GPUs.

In this paper, we extend the PACC framework [17] with two data-centric optimizations to alleviate the performance limitation imposed by heavy data transfer. The first is a direct-mapping scheme that eliminates host buffers [17] used to transfer decomposed data between the host and device. The second is a region-sharing scheme that significantly reduces host-device data transfer by further reusing on-device data. The experimental results demonstrate that even a relatively small dataset that fits in device memory benefits from the region-sharing scheme. This reveals the efficacy of region-sharing scheme regardless of the data size, as long as the data parallelism is sufficient for GPU acceleration.

The main contributions of this paper are as follows:

1. Extension of the PACC framework with two data-centric optimizations that benefit large and high-order stencil applications.
2. Parallelization of a geophysics simulator [1]—a real world application—with the extended PACC framework.

The remainder of this paper is organized as follows. Section 2 introduces work related to GPU acceleration of out-of-core stencil computation, whereas Sect. 3 presents details of stencil computation with a real-world application—a geophysics simulator that was parallelized in this study. Section 4 introduces the PACC framework [17] that generates OpenACC-based out-of-core stencil code, whereas Sect. 5 elaborates on work that extends the PACC framework with data-centric optimizations. Section 6 provides the experimental results, whereas Sect. 7 concludes the paper and provides suggestions for future work.

2. Related Work

Sourouri *et al.* [18] proposed a compiler framework for three-dimensional (3D) stencil computation on GPU clusters. They provided directives and a source-to-source translator and thus reduced programming effort by automatically generating out-of-core stencil code from serial code. However, optimizations to reuse on-device data, such as temporal blocking, were not implemented.

Miki *et al.* [17] proposed PACC, an extension of OpenACC for out-of-core stencil computation on a GPU. The PACC framework automatically generates out-of-core code that decomposes the original data into chunks, each of which fits in the device memory. In addition, the generated code implements temporal blocking to reuse on-device data and performs pipeline execution to overlap data transfer with kernel execution. However, an intermediate-copying scheme is used in the generated code in which chunks are first copied to host buffers and then transferred to the device. Furthermore, the generated code transfers extra data (i.e., halo regions) to the device to perform temporal blocking. Therefore, the performance of the generated code degrades when the code handles high-order stencil computations and/or runs on the latest GPU due to the heavy data transfer cost. To address the data transfer problem, we integrated two data-centric optimizations into the PACC framework. First, we adopted OpenACC APIs to map regions of the original data to the device buffers, avoiding data copying between the original data and host buffers. Secondly, we designed a region-sharing scheme for contiguous chunks to share common regions, thus significantly reducing the amount of data transfer between the host and device.

Jin *et al.* [19] proposed a multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU. In their method, they reused the intermediate computation results to eliminate redundant data (i.e., halo) transfers. Such a result-reusing scheme involves storing and restoring the intermediate results on the GPU for every two time steps, incurring more invocations for memory copy APIs than our region-sharing scheme. In our region-sharing scheme, one subdomain only needs to copy the overlapped regions for the consequent subdomain, before temporal blocking computations. The novelty of our work is that the region-sharing scheme is more succinct than the result-sharing scheme. That is, the region-sharing scheme has (1) fewer source lines and (2) simpler control that involves fewer invocations for CUDA APIs to copy data on the device. Although the region-sharing scheme is more succinct than the result-sharing scheme, the two schemes have almost the same performance because they have the same effect in reducing host-device data transfer that limits the performance. However, we must admit that compared to the region-sharing scheme, the result-reusing scheme reduces the size of device buffers by half the size of halo regions, because it can reuse the previous intermediate computation results in a shifting manner. The result-reusing scheme can also eliminate the redundant computations.

Shimokawabe *et al.* [20] proposed a stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. For excess data, the framework decomposes the entire domain stored in the host

memory into subdomains and then transfers the subdomains to the device for computation. However, this method transfers each subdomain with all halos required for temporal blocking to the device. Therefore, the cost of the data transfer can be high, especially for high-order stencil code. In contrast, our method allows a subdomain to share common regions with contiguous subdomains to reduce the amount of data transfer.

Reguly *et al.* [21] presented a cache-blocking tiling technique that efficiently processes large scale stencil code. Their implementations are based on OPS [22]–[24]—a domain specific language (DSL) that requires implementing stencil code based on its syntax. Instead, the PACC framework requires no modification aside from the insertion of directives into serial stencil code, thus reducing programming effort.

Hou *et al.* [25] proposed a framework to automatically generate stencil codes to access buffered data in the cached systems of GPUs. In their work, 2.5D block was adopted to decompose the original data; however, temporal blocking was not utilized for data reuse. In contrast, we adopted a 1.5D block scheme to decompose the original large data into blocks, fixing the sizes of blocks along the X and Y axes and varying only the size along the Z axis. We adopted the 1.5D block scheme for the convenience to apply temporal blocking because the scheme avoids discontinuous data intervened by halo regions along X and Y axes. Discontinuous data involve multiple invocations of host-device data transfer when the proposed region-sharing scheme is used to reduce transfers of halo regions.

Endo [26] proposed a recursive temporal blocking algorithm. Given multiple hierarchies of memory, he applied temporal blocking with variable block sizes. Data transfer and kernel execution did not overlap.

3. Acoustic Wave Propagator: Target Stencil Computation

In this section, we use the acoustic wave propagator [1] as an example of a reputed stencil application that is widely used for geophysics exploration in the petroleum industry. Acoustic wave propagation is governed by the following equation:

$$\frac{1}{V^2 x}\frac{\partial^2 p}{\partial t^2} = \nabla^2 p, \tag{1}$$

where $p(x, y, z, t)$ is the acoustic pressure at time t and the mesh specified by Cartesian coordination (x, y, z); $V(x, y, z)$ is the propagation speed, whereas ∇^2 is the Laplacian operator.

For our 3D application, $\nabla^2 p$ can be replaced by the sum of the second-order partial derivatives of p in each dimension. Therefore, we have

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}. \tag{2}$$

For the partial derivatives, we use a second-order central finite difference approximation [27] for time and an eighth-order approximation for space. We thus have

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} \approx \frac{1}{V^2}\frac{p(x, y, z, t+1) - 2p(x, y, z, t) + p(x, y, z, t-1)}{dt^2}. \tag{3}$$

With respect to the spatial terms on the right-hand side of (2), take the first term for instance, we have

$$\frac{\partial^2 p}{\partial x^2} \approx \frac{X}{dx^2}, \tag{4}$$

where

$$\begin{aligned} X = &-\frac{1}{560}(p(x+4, y, z, t) + p(x-4, y, z, t)) \\ &+ \frac{8}{315}(p(x+3, y, z, t) + p(x-3, y, z, t)) \\ &- \frac{1}{5}(p(x+2, y, z, t) + p(x-2, y, z, t)) \\ &+ \frac{8}{5}(p(x+1, y, z, t) + p(x-1, y, z, t)) \\ &- \frac{205}{72}p(x, y, z, t). \end{aligned} \tag{5}$$

The same procedure is then applied to the other two terms. In this way, we can discretize (1); that is, we can describe it in the form of a 25-point stencil computation (Fig. 1) in which calculation of an element relies on 24 surrounding elements (i.e., halo regions).

```
//c0 = -205.0f/72.0f, c1 = 8.0f/5.0f,
//c2 = -1.0f/5.0f, c3 = 8.0f/315.0f,
//c4 = -1.0f/560.0f;
//p3, p2, and p1 are the pressures at time t+1, t,
//and t-1, respectively
p3[z][y][x] = (1/dx2 + 1/dy2 + 1/dz2) * c0 * p[z][y][x];
p3[z][y][x] += 1/dx2 * (c4 * (p2[z][y][x+4] + p2[z][y][x-4]) + c3 * (p2[z][y][x+3] + p2[z][y][x-3])
                      + c2 * (p2[z][y][x+2] + p2[z][y][x-2]) + c1 * (p2[z][y][x+1] + p2[z][y][x-1]));
p3[z][y][x] += 1/dy2 * (c4 * (p2[z][y+4][x] + p2[z][y-4][x]) + c3 * (p2[z][y+3][x] + p2[z][y-3][x])
                      + c2 * (p2[z][y+2][x] + p2[z][y-2][x]) + c1 * (p2[z][y+1][x] + p2[z][y-1][x]));
p3[z][y][x] += 1/dz2 * (c4 * (p2[z+4][y][x] + p2[z-4][y][x]) + c3 * (p2[z+3][y][x] + p2[z-3][y][x])
                      + c2 * (p2[z+2][y][x] + p2[z-2][y][x]) + c1 * (p2[z+1][y][x] + p2[z-1][y][x]));
p3[z][y][x] = v[z][y][x] * v[z][y][x] * dt2 * p3[z][y][x] + 2 * p2[z][y][x] - p1[z][y][x];
```

Fig. 1 3D acoustic wave propagation described as 25-point stencil computation.

```
#pragma pacc init

#pragma pacc pipeline targetinout(p1) targetin(p2) size([0:y][0:x]) halo([1:1][1:1]) async
for(int t=0;t<total_time steps;t++){
  #pragma pacc loop dim(2)
  for(int j=0;j<y;j++){
      #pragma pacc loop dim(1)
      for(int i=0;i<x;i++){
          p2[j][i] = 0.25 * (p1[j][i+1] + p1[j][i-1] + p1[j+1][i] + p1[j-1][i]);
      }
  }
  //swap p1 and p2
  #pragma pacc loop dim(2)
  for(int j=0;j<y;j++){
      #pragma pacc loop dim(1)
      for(int i=0;i<x;i++){
          p1[j][i] = p2[j][i];
      }
  }
}
```

Fig. 2 Sample code with PACC directives for four-point stencil computation, i.e., finite difference solver for Laplace's equation.

4. PACC-Based Out-of-Core Stencil Computation

As explained in Sect. 1, to process excess data, naively inserting OpenACC directives into serial stencil code leads to device memory exhaustion. To avoid such failures, out-of-core stencil computation requires modifying the original code. A minimum modification includes data decomposition and host-device data transfer. In addition, optimizations such as temporal blocking must be implemented to improve performance. To reduce the programming effort caused by these code modifications, the PACC framework [17] automatically generates OpenACC-based out-of-core stencil code. In this section, we describe how the framework realizes out-of-core stencil computation.

4.1 PACC Directives

The PACC framework provides OpenACC-like directives (i.e., *init*, *pipeline*, and *loop* constructs) to describe an out-of-core stencil code. Figure 2 presents an example of code with PACC directives for four-point stencil computation. Data decomposition and temporal blocking are not manually implemented but are automatically generated by the PACC framework. Brief descriptions for PACC directives are follows:

- The *init* construct (Line 1). This construct preserves variables used for out-of-core stencil computation such as the number of time steps for temporal blocking.
- The *pipeline* construct (Line 3). This construct is located directly before the outer loop which confines all iterations of the stencil computation. The *targetin* and *targetinout* clauses define read-only and read/write datasets. The *size* clause defines the size of datasets (i.e., range in each dimension) specified by the start index and length. The *halo* clause defines the size of halo regions in each dimension.
- The *loop* construct (Lines 5, 7, 13, and 15). This construct is located directly before each loop inside the kernel. The *dim* clause indicates the level of the loop inside the kernel. We decompose data at the top-level loop.

4.2 Rule-Based Translator

The PACC framework provides a translator that automatically generates out-of-core OpenACC code from a serial code implemented with PACC directives, based on rewriting rules. The translator was implemented with pure Python to achieve high usability by avoiding any package dependency issue. Code segments that perform data decomposition and temporal blocking are added by the translator. The translator functions as follows.

- Parses variables from the *pipeline* construct (e.g., the names of arrays and the size of halo regions).
- Adds variables and methods used to perform out-of-core computation (e.g., a method that maps data regions in the host memory to the device buffers).
- Replaces the *init* construct with code that initializes variables used to perform out-of-core computation (e.g., memory allocation for the device buffers).
- Adds an outer loop to the code block confined by the *pipeline* construct to control the chunk-wise out-of-core process.
- Rewrites all kernels in the code block confined by the *pipeline* construct (i.e., for each kernel, modifies the *loop* constructs and translates the data indices to process chunks rather than all of the data).

Figure 3 describes how the generated code is implemented with OpenACC directives, OpenACC APIs, and CUDA APIs.

```
allocate host dataset using cudaHostAlloc(h_p,h_sz*sizeof(float))
//p is pointer to original data, h_sz is size of data
allocate device buffers using d_buf[0-2]=acc_malloc(b_sz*sizeof(float))
//d_buf[0-2] are pointers to device buffers and b_sz is size of a device buffer
while number of remnant time steps is greater than zero
    reduce remnant time steps by temporal blocking time steps
    for each chunk
        map chunk to device buffer using acc_data_map(&h_p[ofs],d_buf[current],b_sz*sizeof(float))
        //ofs is offset from start of current chunk to start of original data
        #pragma data update device(h_p[ofs+r_sz:b_sz-r_sz])
        //r_sz is size of common regions
        copy common regions for next chunk
            using cudaMemcpyAsync(d_buf[current],d_buf[next]+ofs_b,r_sz*sizeof(float))
        //ofs_b is offset from start of device buffer to start of common regions
        for temporal blocking time steps
            #pragma acc kernels present(h_p[ofs:b_sz])
            parallel loops for stencil kernels
        #pragma data update host(h_p[ofs+r_sz/2:b_sz-r_sz])
        unmap data with acc_data_unmap(&h_p[ofs])
```

Fig. 3 A simplified description showing how PACC-generated code utilizes OpenACC directives, OpenACC APIs and CUDA APIs.

4.3 Data Decomposition and Temporal Blocking

Because the OpenACC compiler prepares identical variables for both the host and the device, a large host array can cause runtime failures due to device memory exhaustion. Therefore, the PACC framework decomposes excess data into smaller chunks and move the chunks between the host and device for out-of-core computation. To the best of our knowledge, there are two explicit ways and an implicit way to perform the chunk-wise process:

1. intermediate-copying. In the previous PACC framework, Miki *et al.* [17] prepared host buffers. Therefore, chunks are copied from the original data to the host buffers and then transferred to the device buffers with the *update device* directive. Once computation on a chunk completes, the chunk is transferred back to the host buffers with the *update host* directive.
2. Direct-mapping. In this study, we extend the PACC framework with a direct-mapping scheme. This scheme directly maps a chunk from the original data to the device buffers with OpenACC map APIs. In this way, data copying between the original data and host buffers is eliminated.
3. Unified Memory [28]. This scheme relies on the CUDA runtime to stream data to and from the device, avoiding device memory exhaustion. However, this scheme has two disadvantages that significantly impair the performance of out-of-core code. First, the scheme is unaware of the application-specific optimizations, such as temporal blocking. Secondly, the scheme results in low host-device bandwidths due to a complex page fault handling mechanism if prefetching hints are not explicitly specified [29].

In most cases, stencil code is time-evolving, which signifies that all of the data must be calculated for multiple time steps; therefore, temporal blocking must be implemented to reduce the amount of host-device data transfer. As mentioned in Sect. 2, we decompose the original data using a 1.5D block scheme for the convenience to implement temporal blocking. For high usability, the previous PACC framework implemented temporal blocking with a succinct overlapped tiling scheme. This scheme simply wraps each chunk with halo regions (Fig. 4) and thus avoids analyzing data dependencies across multiple arrays, which must be performed if temporal blocking is implemented using wavefront [23], [30]–[32], trapezoidal [33], and diamond [34] tiling schemes.

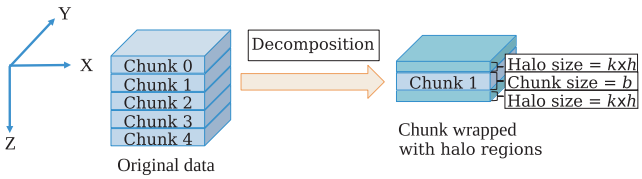

Fig. 4 Data decomposition using 1.5D block scheme. The original data are decomposed into chunks along the main axis. Note that halo regions are required to be attached to each chunk for temporal blocking. In this figure, k and h denote the number of temporal blocking time steps and the size of halo regions, respectively.

However, halo transfer increases linearly with the number of time steps to update a chunk on the device. For high-order stencil code that has a large halo area, transferring extra data can be incredibly time consuming. To address this problem, we improved the performance of the overlapped tiling scheme with a data reuse optimization scheme that significantly reduces host-to-device (HtoD) data transfer, which is described in greater detail in Sect. 5.2.

4.4 Pipeline Execution

At runtime, the process of a chunk has three stages: (1) HtoD transfer, (2) kernel execution, and (3) device-to-host

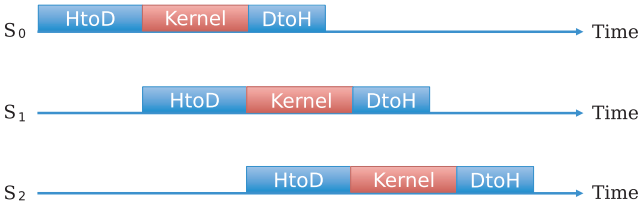

Fig. 5 Asynchronous CUDA streams used to overlap data transfer with kernel execution. S_0, S_1, and S_2 are streams used for the i-th, $(i+1)$-th, and $(i+2)$-th chunks, respectively. Bars marked as "HtoD," "DtoH," and "Kernel" represent host-to-device data transfer, device-to-host data transfer, and kernel execution, respectively.

(DtoH) transfer. The previous PACC framework [17] thus adopted a three-stage pipeline to process three chunks in parallel with asynchronous CUDA streams. For each chunk, the framework submits operations to the corresponding stream (Fig.5). In this way. data transfer is overlapped with kernel execution.

5. Data-Centric Optimizations

As mentioned in Sect. 4.3, data transfer limits the performance of high-order stencil code generated by the PACC framework [17], because a large amount of halo data is transferred between the host and device at runtime. This limitation is more noticeable on new GPUs due to a widened gap between device memory bandwidth and PCIe bandwidth [11].

To alleviate this limitation, we extend PACC with two data-centric optimizations—direct-mapping and region-sharing—that notably reduce the amount of data transfer. The direct-mapping scheme maps and transfers regions of the original data to the device buffers, avoiding data copying between the original data and host buffers. Furthermore, the region-sharing scheme evolves the overlapped tiling scheme used by the previous PACC framework by eliminating all halo transfer between the host and device.

5.1 Direct-Mapping

The previous PACC framework uses an intermediate-copying scheme that prepares host buffers that mediate between the original data and device buffers (Fig. 6). In contrast, the proposed direct-mapping scheme uses OpenACC APIs (i.e., *acc_data_map* and *acc_data_unmap*) to map regions of the original data to the device buffers (Fig. 7). The mapped data regions are then transferred between the host and device with OpenACC directives (i.e., *data update device* and *data update host*). Thus, the direct-mapping scheme avoids data copying between the original data and host buffers.

Nevertheless, we take advantage of page-locked (pinned) memory for high speed of data transfer between host and device. Similar to the CUDA implementations [35]–[37], the direct-mapping uses pinned memory by allocating the host datasets with CUDA API, but it differs

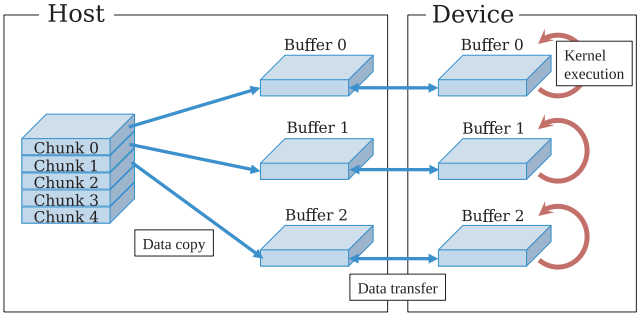

Fig. 6 intermediate-copying scheme [17]. Host buffers are used as "transfer stations" between original data and device buffers.

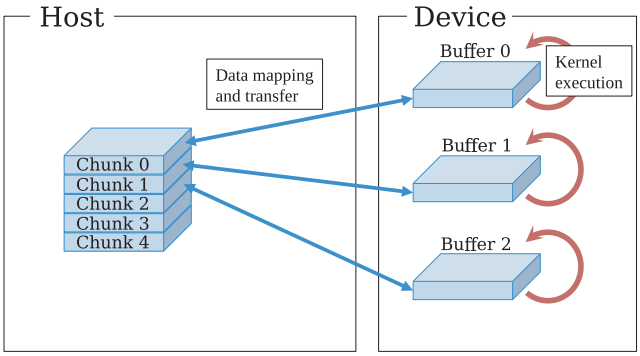

Fig. 7 Direct-mapping scheme implemented with data mapping APIs provided by OpenACC. This scheme eliminates data copying between the original data and host buffers.

from [35]–[37] in using OpenACC APIs to map host data to device buffers (Fig. 3). Moreover, because the generated code deploys OpenACC directives, we compiled the generated code with the PGI compiler option to enable pinned memory. Without the appropriate option, the direct-mapping uses pageable memory. Note also that the direct-mapping scheme consumes more pinned memory than the intermediate-copying scheme does. In the intermediate-copying scheme, only three host buffers, each with a size of a chunk and corresponding halo regions, need to be allocated with pinned memory. However, in the direct-mapping scheme, all of the original data must be allocated with pinned memory. This drawback thus requires dynamically allocating pinned memory if the size of the original data is beyond the host memory capacity.

5.2 Region-Sharing

The previous PACC framework performs temporal blocking with the overlapped tiling scheme to avoid analyzing data dependencies across multiple arrays. However, as we mentioned in Sect. 4.3, the overlapped tiling scheme requires attaching extra data (i.e., halos) to each chunk. The extra data transfer increases with the number of time steps of temporal blocking and is time-consuming for high-order stencil code due to large halo areas. We therefore propose a region-sharing scheme to eliminate the extra data transfer, which

Algorithm 1 Performing out-of-core stencil computation for T time steps using the region-sharing scheme. Note that we assume that the sizes of top halo H_i^{top} and bottom halo H_i^{bottom} for any chunk C_i are equal. For simplicity, we avoid special descriptions for uneven top and bottom halo regions such as that of the first chunk.

Input: (1) $\bigcup_{i=0}^{n-1} C_i$: original data, which is decomposed into n chunks, (2)S_0, S_1, S_2: CUDA streams to perform HtoD data transfer and common regions copying for three chunks, (3)S_3, S_4, S_5: CUDA streams to perform kernel execution and DtoH data transfer for three chunks, (4) T and K: number of total time steps and number of time steps for temporal blocking, respectively.

Output: $\bigcup_{i=0}^{n} C_i$: data that has been updated for T time steps.

```
while T > 0 do
    k ⇐ min(T, K)
    T ⇐ T − k
    i ⇐ 0
    while i < n do
        sid ⇐ i (mod 3)                                   ▷ id of first stream for current chunk
        prev ⇐ sid − 1                                    ▷ id of first stream for previous chunk
        if prev ≠ −1 then                                 ▷ transfer updated previous chunk to host
            transfer C_{i−1} to host using S_{prev+3}
        end if
        if i = 0 then                                     ▷ for first chunk, transfer it with upper and lower halo regions
            transfer H_i^top ∪ C_i ∪ H_i^bottom to device using S_sid
        else                                              ▷ for each of other chunks, transfer remnant of it with lower halo regions
            transfer (C_i ∪ H_i^bottom − H_{i−1}^bottom) to device using S_sid
        end if
        if prev ≠ −1 then
            wait for S_prev to complete copying common regions
        end if
        if i ≠ n − 1 then                                 ▷ if current chunk is not last one, copy common regions to next chunk
            copy H_{i+1}^top ∪ H_i^bottom for C_{i+1} on device using S_sid
        end if
        wait for S_sid to complete HtoD data transfer
        while k > 0 do
            process C_i on device using S_{sid+3}
            k ⇐ k − 1
        end while
        i ⇐ i + 1
    end while
end while
```

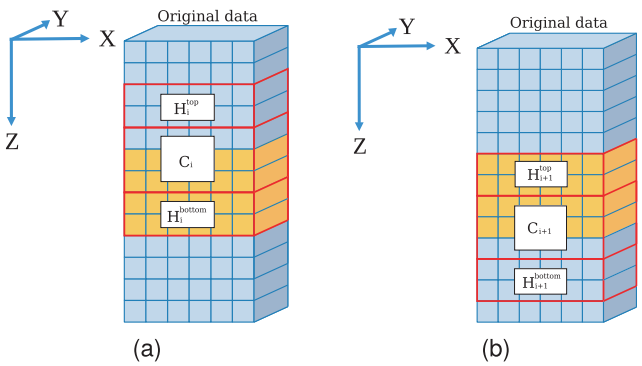

Fig. 8 Regions shared by two contiguous chunks (a) C_i and (b) C_{i+1}, $H_{i+1}^{top} \cup H_i^{bottom}$. H_{i+1}^{top} denotes the top halo regions of C_{i+1}, whereas H_i^{bottom} denotes the bottom halo regions of C_i. Note that the original data are decomposed along the Z axis, and each chunk therefore has top and bottom halo regions. Chunks and halo regions are outlined in red, whereas common regions are colored yellow.

constitutes a large proportion of the HtoD data transfer.

The region-sharing scheme takes advantage of the fact that two contiguous chunks share common regions (Fig. 8). We assume that a stencil has equal sizes of halo regions in up and down directions, which is the most common case in stencil computation. As a result, the region-sharing scheme can copy the common regions of a chunk that has arrived on the device to the device buffers that would store the next chunk. The scheme then only needs to transfer the remnant of the next chunk with corresponding halo regions. Note that the size of the remnant equals that of a chunk without halo regions; thus, the scheme effectively eliminates all halo transfer between the host and device.

Theoretically, the more halo regions are used, the more benefit can be drawn from the region-sharing scheme. That is, more time can be saved that would otherwise be consumed by data transfer. However, in practice, more halo regions require larger device buffers; thus, the region-sharing scheme is also subject to limitations of device memory capacity.

The region-sharing scheme requires (1) copying common regions between chunks on the device and (2) maintaining an inter-chunk execution order (i.e., computation on a chunk must wait for the previous chunk to complete copying common regions). To fulfill the first requirement, we use

Algorithm 2 Performing out-of-core stencil computation for T time steps using the result-reusing scheme [19].

Input: (1) $\bigcup_{i=0}^{n-1} C_i$: original data, which is decomposed into n chunks, (2)S_0, S_1, S_2: CUDA streams to perform HtoD and DtoH data transfers for three chunks, (3)S_3, S_4, S_5: CUDA streams to perform reading and writing intermediate results and kernel execution for three chunks, (4) T and K: number of total time steps and number of time steps for temporal blocking, respectively.

Output: $\bigcup_{i=0}^{n} C_i$: data that has been updated for T time steps.

```
1: while T > 0 do
2:     k ⇐ min(T, K)
3:     T ⇐ T − k
4:     i ⇐ 0
5:     while i < n do
6:         sid ⇐ i (mod 3)                          ▷ id of first stream for current chunk
7:         prev ⇐ sid − 1                           ▷ id of first stream for previous chunk
8:         if prev ≠ −1 then                        ▷ transfer updated previous chunk to host
9:             transfer C_{i−1} to host using S_prev
10:        end if
11:        if i = 0 then                            ▷ for first chunk, transfer it with upper and lower halo regions
12:            transfer H_i^top ∪ C_i ∪ H_i^bottom to device using S_sid
13:        else                                     ▷ for each of other chunks, transfer remnant of it with lower halo regions
14:            transfer (C_i ∪ H_i^bottom − H_{i−1}^bottom) to device using S_sid
15:        end if
16:        wait for S_sid to complete HtoD data transfer
17:        if prev ≠ −1 then
18:            wait for S_{prev+3} to complete writing intermediate results
19:        end if
20:        while k > 0 do
21:            if i ≠ 0 and k (mod 2) = 0 then
22:                read the on-device intermediate results of the common regions computed by C_{i−1}
23:                write the intermediate results of the common regions on device for C_{i+1}
24:            end if
25:            process C_i on device using S_{sid+3}
26:            k ⇐ k − 1
27:        end while
28:        i ⇐ i + 1
29:    end while
30: end while
```

cudaMemcpyAsync to perform data copying on the device. To fulfill the second requirement, we control the pipeline execution in smaller granularity than that used in the previous PACC framework.

Algorithm 1 provides the details of out-of-core stencil computation using the region-sharing scheme. We assume that upper and lower halo regions have the same size. Six streams are used for three chunks. Specifically, we use two streams to perform operations for a chunk. The first stream transfers the chunk from the host to the device and copies the common regions of the chunk to the next chunk. The second stream computes on the chunk and transfers the computation results back to the host. Note that the inter-chunk execution order requires extra synchronizations (Lines 16–18). For instance, considering three contiguous chunks, C_0, C_1, and C_2, C_1 must wait for C_0 to complete copying common regions before C_1 can copy common regions to C_2.

Algorithm 2 shows the details of a result-reusing scheme [19] mentioned in Sect. 2. In the temporal blocking loops of this scheme, for every two time steps, a chunk reads the intermediate results of common regions computed by the previous chunk (line 22), and writes the intermediate results of common regions computed by itself for the next chunk (lines 23). In contrast, the region-sharing scheme is more succinct (i.e., having fewer lines of source code) and reduces the invocations of APIs to perform on-device copy, which is because the region-sharing scheme avoids maintaining additional device buffers to store intermediate results and involves copying the common regions only once before the temporal blocking loops. Moreover, the two schemes have the same effect in eliminating halo transfers, resulting in almost the same performance achievement, which will be demonstrated with experimental results in Sect. 6.5.

However, the region-sharing scheme has a disadvantage. If the code runs in a multi-node environment, the inter-chunk execution order prevents the assignment of chunks to different nodes in an arbitrary order because contiguous chunks must be assigned to the same node to share common regions. This constraint reduces the flexibility with which load balancing is performed in a multi-node environment. Nevertheless, the performance improvement produced by the region-sharing scheme outweighs the reduction in programming flexibility.

6. Experimental Results

We parallelized an acoustic wave propagator introduced in Sect. 3 using the extended PACC framework that

automatically generated out-of-core OpenACC-based code from the original serial code. To demonstrate the applicability to other stencil kernels, we also applied the extended PACC framework to the Himeno benchmark [38], which is widely used in measuring computation speed of different architectures. Himeno benchmark has much smaller halo size (i.e., one) than the acoustic wave propagator. The generated code implemented the proposed optimization techniques, and we evaluated the generated code on a latest NVIDIA GPU with respect to performance.

6.1 Experimental Setup

To verify the effectiveness of the proposed optimization schemes on a latest NVIDIA GPU (Table 1), we designed five experiments with two datasets for the acoustic wave propagator (Table 2) and one dataset for the Himeno benchmark (Table 3). A brief introduction to the four experiments is as follows.

- The first experiment aims to detect the performance improvement achieved by the PACC-generated code compared with other implementations, such as OpenMP and Unified Memory based programs.
- The second experiment aims to analyze the speedups achieved by the direct-mapping and region-sharing schemes.
- The third experiment aims to investigate the improvement (or degradation) attributed to PACC-generated out-of-core code compared with an in-core implementation.
- The fourth experiment aims to compare the region-sharing scheme with a previous result-reusing scheme in terms of performance.
- The fifth experiment aims to demonstrate the number of source lines automatically generated by the PACC framework.

Table 1 Testbed: a latest NVIDIA GPU.

Device	Tesla V100-PCIe
Device architecture	Volta
Device memory capacity	16 GB
Host	Xeon Silver 4110
OS	Ubuntu 16.04.6
CUDA	9.2
PGI compiler	19.10
Interconnect	PCIe 3.0 ×16

Table 2 Two datasets for acoustic wave propagator.

	Out-of-core	In-core
Size	$1160 \times 1160 \times 1160$	$820 \times 820 \times 820$
Data type	Float	Float
Number of arrays	4	4
Total amount	24 GB	8 GB

Table 3 Dataset for Himeno benchmark.

Size	$613 \times 613 \times 1225$
Data type	Float
Number of arrays	14
Total amount	24 GB

6.2 Comparison with OpenMP, Unified Memory, and Intermediate-Copying Based Implementations

In this experiment, we compared the out-of-core acoustic wave propagation code and Himeno benchmark generated by the extended PACC framework with OpenMP (i.e., CPU parallelization), Unified Memory, intermediate-copying [17], and direct-mapping based implementations. Note that in our experiments, we prepared the OpenMP and OpenACC based implementations by inserting OpenMP or OpenACC directives into the original serial code. Unified Memory was enabled for the OpenACC-based implementation by compiling the source code with the Unified Memory option. Techniques, such as loop transformation and Unified Memory prefetching hints, were not used because they require many manual modifications to the original code, such as dividing the original data into tiles. In contrast, PACC-based out-of-core implementations were automatically generated by the proposed framework. All PACC-based implementations used pinned memory by enabling pinned memory when we compiled the codes. We set d (the number of chunks) to eight, and k (the number of temporal blocking time steps) to 12 for the acoustic wave propagator, whereas we set d to 10, and k to 16 for the Himeno benchmark. These settings were experimentally determined to be optimal. The datasets used in this experiment were 24 GB (referring to out-of-core data in Table 2 and Table 3).

Figure 9 presents the results of running acoustic wave propagation codes obtained on the Tesla V100 GPU. The execution time of out-of-core code generated by the extended PACC framework was 7.5 s, which was 41.0, 22.1, 3.6, and 1.4 times as fast as the OpenMP (307.4 s), Unified

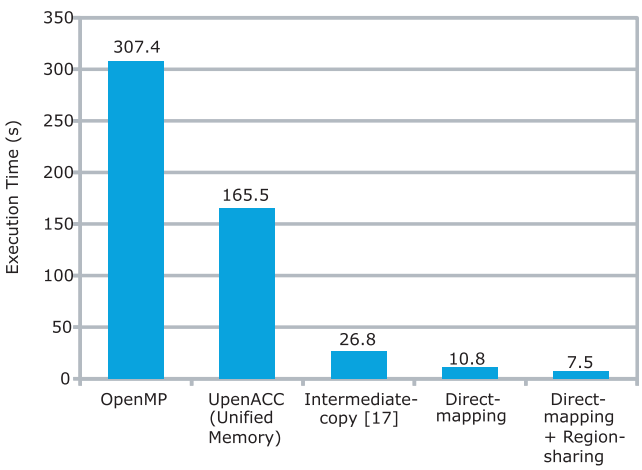

Fig. 9 Comparison of the acoustic wave propagation code generated by the extended PACC framework (Direct-mapping + Region-sharing) with other implementations on a Tesla V100 GPU in terms of performance.

Memory (165.5 s), intermediate-copying [17] (26.8 s), and direct-mapping (10.8 s) based implementations, respectively.

Figure 10 demonstrates that the extended PACC framework obtained even better speedup for the Himeno benchmark than for the acoustic wave propagator on the Tesla V100 GPU. The execution time of out-of-core code generated by the extended PACC framework was 4.5 s, which was 103.0, 88.9, 2.7, and 1.5 times as fast as the OpenMP (508.6 s), Unified Memory (400.1 s), intermediate-copying [17] (12.1 s), and direct-mapping (6.9 s) based implementations, respectively. Note that the transition of speedups was similar for both acoustic wave propagator and Himeno benchmark, as we incrementally added the optimization schemes to the PACC framework, which proved the general usefulness of the extended PACC framework for different stencil kernels.

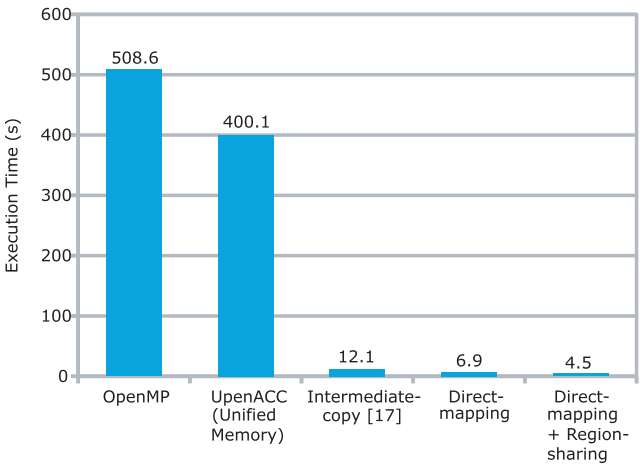

Fig. 10 Comparison of the Himeno benchmark code generated by the extended PACC framework (Direct-mapping + Region-sharing) with other implementations on a Tesla V100 GPU in terms of performance.

These results demonstrate the effectiveness of the extended PACC framework on the state-of-the-art GPU, owing to the direct-mapping and region-sharing schemes. Detailed analyses for the improvements achieved by the two schemes are given in the following section.

6.3 Detailed Analyses of Data-Centric Optimizations

To discuss the performance bottleneck and achievements of the data-centric optimizations, we analyzed the performance improvement by using the direct-mapping and region-sharing schemes. In this experiment, we ran the intermediate-copying, direct-mapping, and direct-mapping plus region-sharing based implementations on the Tesla V100 GPU. Because the extended PACC framework obtained similar benefits for both acoustic wave propagtor and Himeno benchmark, we considered only the acoustic wave propagtor in this experiment.

Figure 11 reveals that the replacement of the intermediate-copying scheme by the direct-mapping scheme attained a 2.7× speedup (26.8 s/10.1 s). The performance (i.e., total execution time) of intermediate-copying based code was apparently limited by the data copying between the original data and host buffers (i.e., reading and writing the host buffers). Note that in the intermediate-copying [17] based code, although reading and writing the host buffers were performed by multiple threads, reading was not overlapped with writing, and vice versa. In contrast, the performance of the direct-mapping based code was limited by the HtoD data transfer, which required less time than reading

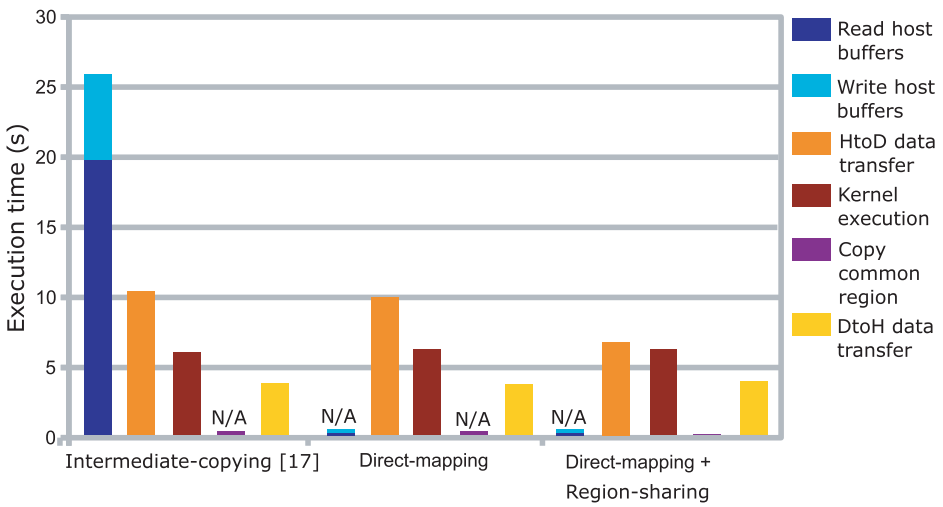

Fig. 11 Comparison of the breakdown of the execution time of three versions of out-of-core code, implemented with intermediate-copying [17], direct mapping, and both direct mapping and region sharing, respectively. Bars of different colors denote the time consumed by different operations. "N/A" denotes that no time consumption for that operation. Direct-mapping based code eliminated data copying between the original data and host buffers, and thus ran 2.7 times as fast as intermediate-copying based code; moreover, region-sharing achieved a further 1.4× speedup because it eliminated all halo transfer.

and writing the host buffers did. The amount of HtoD data transfer was the same for both schemes; thus, a reduction in data transfer is necessary for further improvement.

Moreover, for the code implemented with direct-mapping and that implemented with both direct-mapping and region-sharing, the bounding operation is HtoD data transfer. Therefore, we are interested in analyzing the theoretical and practical speedups for HtoD data transfer achieved by the region-sharing scheme. We obtained the theoretical speedup by calculating the amount of HtoD data transfer reduced by the region-sharing scheme. In this experiment, the data consisted of four arrays. Each array had $1160 \times 1160 \times 1160$ float-type elements. Because we decomposed the data along one axis, we considered the data as 1160 planes, where each plane had 1160×1160 float-type elements. We decomposed the data into eight chunks and processed each chunk on the device for 12 times. Therefore, for the first and last chunks, the halo region had $4 + 4 \times 12 = 52$ planes, whereas for the other chunks, the halo regions had $2 \times 4 \times 12 = 96$ planes. Because the region-sharing scheme effectively avoids transferring halo regions, the amount of HtoD transfer equals that of the original data: 1160 planes. However, if the region-sharing scheme is not used, $(1160 + 52 \times 2 + 96 \times 6 = 1{,}840$ planes must be transferred to the device to process all of the data for 12 time steps. Summarily, the theoretical speedup of HtoD transfer should be $1{,}840/1160 = 1.6\times$. However, the practical speedup for HtoD transfer was $1.5\times$ (= 9.8 s/6.7 s). In order to explain the discrepancy, we recorded the data transfer behaviors of both implementations. The results indicate that the increased number of operation streams and synchronizations required by the region-sharing scheme resulted in a small achieved bandwidth compared with the implementation without region-sharing. For the HtoD data transfer, the implementation with region-sharing achieved 10.5 GB/s, whereas the implementation without region-sharing achieved 11.2 GB/s. We can thus verify the validity of the practical speedup, i.e., $(1{,}840/11.2)/(1160/10.5) = 1.5$ times. In terms of the total execution time, the code implemented both direct-mapping and region-sharing (7.5 s) ran $1.4\times$ as fast as that using direct-mapping alone (10.1 s).

6.4 Comparison with In-Core Implementation

In this experiment, we aimed to determine the improvement or degradation caused by PACC-generated out-of-core code, compared with in-core OpenACC code. We ran OpenACC, Unified Memory, direct-mapping, and direct-mapping plus region-sharing based implementations on the Tesla V100 GPU. The direct-mapping plus region-sharing based implementation implemented the two data-centric optimization schemes to process data that fit in the device memory (in-core data in Table 2). The direct-mapping based implementation processed the same data.

Figure 12 reveals that the Unified Memory based code had almost the same performance as OpenACC based code without using Unified Memory, implying that the Unified Memory technique fails to sufficiently overlap data transfer with kernel execution if prefetching hints are not explicitly implemented. The PACC-generated code using the direct-mapping scheme alone led to a degradation of 15% compared with the in-core code due to extra data (i.e., halo regions) transfer. Fortunately, the code using both the direct-mapping and region-sharing schemes ran 1.3 times as fast as the in-core code, because the proposed schemes notably reduce the amount of data transfer and thus improve the effect of overlapping data transfer with kernel execution.

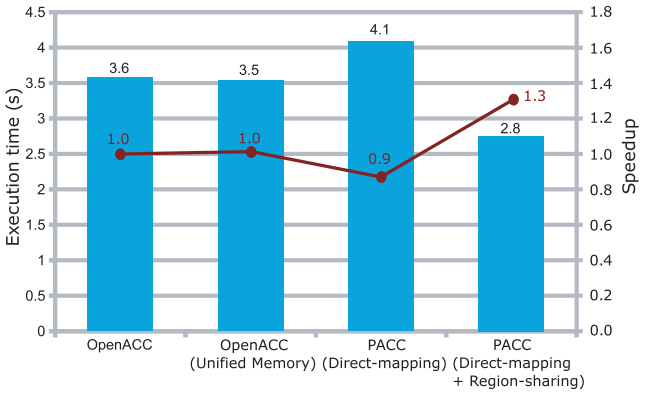

Fig. 12 Analysis of the performance of PACC-generated code to process data that fit in the device memory.

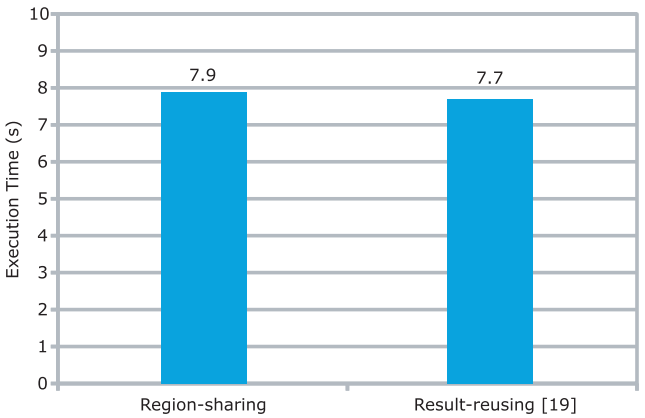

Fig. 13 Comparison between region-sharing and result-reusing [19] schemes in terms of performance.

6.5 Comparison with Result-Reusing Scheme

In this experiment, we compared the region-sharing scheme with result-reusing scheme [19]. Both implementations used direct-mapping scheme.

Figure 13 shows that similar performances were obtained by the two schemes, which was because the two schemes had the same effect in eliminating the halo transfers. The performances were limited by HtoD data transfer time for both schemes. Precisely, the result-reusing scheme had shorter HtoD data transfer time (6.9 s) than region-sharing scheme (7.3 s), because the result-sharing scheme took advantage of smaller device buffers. However,

Table 4 Comparison between source lines of code (SLOC) of PACC-generated out-of-core code (429 lines) and that of serial code (185 lines).

	Serial code	Code with PACC directives	Generated out-of-core code
SLOC	185	193	429

Table 5 Details of generated out-of-core code. "Changed" and "Unchanged" denote lines that are changed and not changed, respectively, compared with serial code.

	Changed				Unchanged
	Adding new variables and methods	Data transfer (direct mapping)	Region sharing	Kernel execution (temporal blocking)	
SLOC	78	132	25	90	104

the region-sharing scheme has shorter on-device data copy (i.e., common region copy) time (0.1 s) than result-reusing scheme (0.2 s), and the region-sharing scheme also showed better effect in overlapping operations, because the region-sharing scheme had fewer CUDA API invocations for on-device data copy.

6.6 Evaluation of Programming Effort Benefits

In this experiment, we examined on the programming effort reduced by using the extended PACC framework. We considered the source lines of code (SLOC) as a metric for programming effort. Table 4 demonstrates that the serial code of the acoustic wave propagator had 193 lines, whereas the PACC-generated out-of-core code had 429 lines. Therefore, the PACC code generation process reduced the programming effort by automatically extending the serial code 2.3 times in length. In fact, the framework was even more helpful because 325 out of 429 generated lines were either new or modified compared with the original serial code (Table 5). From this perspective, the PACC framework exempts the users from manually modifying 75% of the final program.

7. Conclusion

In this study, we extended the PACC framework using two data-centric optimization schemes for GPU acceleration of out-of-core stencil computation. The direct-mapping scheme avoids data copying between the original data and host buffers, whereas the region-sharing scheme avoids halo region transfer between the host and device. We thus retain the high usability of this directive-based framework and generate efficient out-of-core code with the framework.

The experimental results demonstrate that out-of-core code generated by the extended PACC framework outperformed OpenMP and Unified Memory based implementations by a factor of 10 on a latest GPU, thus verifying the usefulness of the extended PACC framework. With respect to the data-centric optimizations, the replacement of intermediate-copying [17] by the direct-mapping scheme contributed to a 2.7× speedup. Furthermore, the region-sharing scheme achieved an additional 1.4× speedup. Although being aimed at out-of-core stencil computation, the extended PACC framework can be applied to in-core code, achieving a 1.3× speedup. With respect to the extent of programming effort reduction, we determined that the PACC framework automatically extended the original serial code 2.3 times in length to obtain the out-of-core parallel code. In addition, 75% of the extended code was different from the original serial code.

Future work includes adapting the PACC framework to a multi-node environment. Because the region-sharing scheme involves an inter-chunk execution order, sophisticated offloading algorithms are required that consider the data dependency between contiguous chunks. Tuning schemes are also worthy of investigation to automatically determine the parameters for temporal blocking.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers JP15H01687, JP16H02801, and JP20K21794.

References

[1] M. Serpa, E. Cruz, M. Diener, A. Krause, A. Farrés, C. Rosas, J. Panetta, M. Hanzich, and P. Navaux, "Strategies to Improve the Performance of a Geophysics Model for Different Manycore Systems," Proc. 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pp.49–54, Campinas, 2017.

[2] A. Farres, C. Rosas, M. Hanzich, M. Jordà, and A. Peña, "Performance Evaluation of Fully Anisotropic Elastic Wave Propagation on NVIDIA Volta GPUs," Proc. 81st EAGE Conference and Exhibition, 2019.

[3] S. Adams, J. Payne, and R. Boppana, "Finite Difference Time Domain (FTDT) Simulations Using Graphics Processors," Proc. High Performance Computing Modernization Program Users Group Conf. (HPCMP-UGC), pp.334–338, Pittsburgh, PA, 2007.

[4] K. Ikeda, F. Ino, and K. Hagihara, "Efficient Acceleration of Mutual Information Computation for Nonrigid Registration using CUDA," IEEE J. Biomed. Health Inform., vol.18, no.3, pp.956–968, 2014.

[5] S. Tabik, M. Peemen, and L. Romero, "A tuning approach for iterative multiple 3d stencil pipeline on GPUs: Anisotropic Nonlinear Diffusion algorithm as case study," The Journal of Supercomputing, vol.74, no.4, pp.1580–1608, 2018.

[6] K. Datta, "Auto-tuning Stencil Codes for Cache-Based Multicore Platforms," Technical Report No. UCB/EECS-2009-177, 2009.

[7] A. Schäfer and D. Fey, "High Performance Stencil Code Algorithms for GPGPUs," Proc. International Conference of Computer Science,

pp.2027–2036, 2011.
[8] T. Okuyama, M. Okita, T. Abe, Y. Asai, H. Kitano, T. Nomura, and K. Hagihara, "Accelerating ODE-based Simulation of General and Heterogeneous Biophysical Models using a GPU," IEEE Trans. Parallel Distrib. Syst., vol.25, no.8, pp.1966–1975, 2014.
[9] F. Ino, K. Shigeoka, T. Okuyama, M. Motokubota, and K. Hagihara, "A Parallel Scheme for Accelerating Parameter Sweep Applications on a GPU," Concurrency and Computation: Practice and Experience, vol.26, no.2, pp.516–531, 2014.
[10] Y. Mitani, F. Ino, and K. Hagihara, "Parallelizing Exact and Approximate String Matching via Inclusive Scan on a GPU," IEEE Trans. Parallel Distrib. Syst., vol.28, no.7, pp.1989–2002, 2017.
[11] J. Shen, K. Shigeoka, F. Ino, and K. Hagihara, "GPU-based Branch-and-Bound Method to Solve Large 0-1 Knapsack Problems with Data-centric Strategies," Concurrency and Computation: Practice and Experience, vol.31, no.4, e4954, 2019.
[12] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: portable parallel programming with the message-passing interface, MIT press, 1999.
[13] R. Pas, E. Stotzer, and C. Terboven, Using OpenMP–The Next Step: Affinity, Accelerators, Tasking, and SIMD, MIT press, 2017.
[14] NVIDIA Corporation, CUDA C Programming Guide, 2019. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[15] Y. Lu, F. Ino, and K. Hagihara, "Cache-Aware GPU Optimization for Out-of-Core Cone Beam CT Reconstruction of High-Resolution Volumes," IEICE Trans. Inf. & Syst., vol.E99-D, no.12, pp.3060–3071, 2016.
[16] PGI Compilers & Tools, OpenACC Getting Started Guide, 2019. https://www.pgroup.com/resources/docs/19.1/pdf/openacc19_gs.pdf.
[17] N. Miki, F. Ino, and K. Hagihara, "PACC: a directive-based programming framework for out-of-core stencil computation on accelerators," International Journal of High Performance Computing and Networking, vol.13, no.1, pp.19–34, 2019.
[18] M. Sourouri, S. Baden, and X. Cai, "Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers," International Journal of Parallel Programming, vol.45, no.3, pp.711–729, 2017.
[19] G. Jin, T. Endo, and S. Matsuoka, "A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU," Proc. 2013 IEEE International Symposium on Parallel Distributed Processing (IPDPS), Workshops and Phd Forum, pp.1080–1087, 2013.
[20] T. Shimokawabe, T. Endo, N. Onodera, and T. Aoki, "A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers," Proc. 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp.525–529, Hawaii, USA, 2017.
[21] I. Reguly, G. Mudalige, and M. Giles, "Beyond 16GB: out-of-core stencil computations," Proc. Workshop on Memory Centric Programming for HPC (MCHPC), pp.20–29, Denver, CO, 2017.
[22] I. Reguly, G. Mudalige, M. Giles, D. Curran, and S. McIntosh-Smith, "The OPS domain specific abstraction for multi-block structured grid computations," Proc. 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), pp.58–67, New Orleans, LA, 2014.
[23] I. Reguly, G. Mudalige, and M. Giles, "Loop tiling in large-scale stencil codes at run-time with OPS," IEEE Trans. Parallel Distrib. Syst., vol.29, no.4, pp.873–886, 2017.
[24] G. Mudalige, I. Reguly, S. Jammy, C. Jacobs, M. Giles, and N. Sandham, "Large-scale performance of a DSL-based multi-block structured-mesh application for Direct Numerical Simulation," Journal of Parallel and Distributed Computing, vol.131, pp.130–146, 2019.
[25] K. Hou, H. Wang, and W. Feng, "Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus," Proc. Computing Frontiers Conference (CF), pp.107–116, Siena, Italy, 2017.
[26] T. Endo, "Applying Recursive Temporal Blocking for Stencil Computations to Deeper Memory Hierarchy," Proc. IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), pp.19–24, Hakodate, Japan, 2018.
[27] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," Mathematics of computation, vol.51, no.184, pp.699–706, 1988.
[28] S. Deldon, J. Beyer, and D. Miles, "OpenACC and CUDA Unified Memory," Proc. Cray User Group (CUG), Stockholm, Sweden, 2018. https://cug.org/proceedings/cug2018_proceedings/includes/files/pap115s2-file1.pdf.
[29] N. Sakharnykh, "Maximizing Unified Memory Performance in CUDA," 2017. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/.
[30] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.
[31] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," Proc. 14th International Parallel and Distributed Processing Symposium (IPDPS), pp.171–180, Cancun, Mexico, 2000.
[32] T. Muranushi and J. Makino, "Optimal temporal blocking for stencil computation," Procedia Computer Science, vol.51, pp.1303–1312, 2015.
[33] T. Grosser, A. Cohen, P. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs: automatic parallelization using trapezoidal tiles," Proc. 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU), pp.24–31, Houston, TX, 2013.
[34] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," IEEE Trans. Parallel Distrib. Syst., vol.28, no.5, pp.1285–1298, 2016.
[35] M. Harris, "How to Optimize Data Transfers in CUDA C/C++." https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/.
[36] V. Allada, T. Benjegerdes, and B. Bode, "Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster," Proc. 2009 IEEE International Conference on Cluster Computing and Workshops, pp.1–9, 2009.
[37] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: GPU Based High Speed Regular Expression Matching Engine," Proc. 2011 5th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp.366–370, 2011.
[38] R. Himeno, Himeno benchmark, 2015. http://accc.riken.jp/en/supercom/himenobmt/.

Jingcheng Shen was born in 1990, in Chongqing, China. He received a B.E. degree in the College of Software Engineering, Southeast University, China and an M.E. degree in the Graduate School of Information Science and Technology, Osaka University, Japan. He is now working for a PhD degree in Osaka University. He is currently doing research on adapting out-of-core GPU-accelerated applications to rapidly developing parallel machines.

Fumihiko Ino received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently a Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.

Albert Farrés is an engineer at Barcelona Supercomputing Center, the Spanish National Supercomputing Institute. He is currently researching and developing seismic imaging tools for the oil industry. He has an MSc degree and a Bachelor in Computer Science from the Universitat Politècnica de Catalunya.

Mauricio Hanzich is a senior researcher at Barcelona Supercomputing Center, the Spanish National Supercomputing Institute. He is currently researching and developing seismic imaging tools for the oil industry. Prior to this position, he was a professor at the Universitat Autónoma de Barcelona and a information technology consultant for the Argentinian government. Raised in Neuquén, Argentina, Mauricio now lives in Barcelona. He has a PhD degree from the Universitat Autónoma de Barcelona and a Bachelor in Computer Science from Universidad del Comahue (Neuquén).