PACC: a directive-based programming framework for out-of-core stencil computation on accelerators

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@ist.osaka-u.ac.jp
Email: ino@ist.osaka-u.ac.jp
Email: hagihara@ist.osaka-u.ac.jp
*Corresponding author

Abstract: We present a directive-based programming framework, i.e., the pipelined accelerator (PACC), to accelerate large-scale stencil computation on an accelerator device, such as a graphics processing unit (GPU). PACC provides a collection of extended OpenACC directives to facilitate out-of-core stencil computation accelerated using temporal blocking. The proposed framework includes a source-to-source translator capable of generating an out-of-core OpenACC code from the PACC code, i.e., large data is automatically decomposed into smaller chunks that are processed using limited capacity device memory. The generated code is optimised using a temporal blocking technique to minimise CPU-GPU data transfer. Furthermore, the code is accelerated using a multithreaded pipeline engine that maximises data copy throughput and overlaps GPU execution and data transfer. In experiments, we applied the proposed translator to three stencil computation codes. The out-of-core performance for 107 GB data on an NVIDIA Tesla K40 GPU with 12 GB memory reached 69.3 GFLOPS, which is 17% less than the in-core performance for 8 GB data. We believe that the proposed directive-based approach can be used to facilitate out-of-core stencil computation on a GPU.

Keywords: accelerator; directive-based programming; out-of-core execution; OpenACC; graphics processing unit; GPU.

Reference to this paper should be made as follows: Miki, N., Ino, F. and Hagihara, K. (2019) ‘PACC: a directive-based programming framework for out-of-core stencil computation on accelerators’, *Int. J. High Performance Computing and Networking*, Vol. 13, No. 1, pp.19–34.

Biographical notes: Nobuhiro Miki received his BE and ME in Information and Computer Sciences from the Osaka University, Osaka, Japan in 2015 and 2017, respectively. He is currently with Yahoo! Japan Corporation. His current research interests include high performance computing and system software.

Fumihiko Ino received his BE, ME, and PhD in Information and Computer Sciences from the Osaka University, Osaka, Japan in 1998, 2000, and 2004, respectively. He is currently an Associate 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.

Kenichi Hagihara received his BE, ME, and PhD in Information and Computer Sciences from the Osaka University, Osaka, Japan in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. His research interests include the fundamentals and practical application of parallel processing.

This paper is a revised and expanded version of a paper entitled ‘An Extension of OpenACC directives for out-of-core stencil computation with temporal blocking’, presented at WACCPD 2016, Salt Lake City, UT, USA, 14 November 2016.

1 Introduction

Accelerator devices are emerging as a key enabling technology in high-performance computers. For example, graphics processing units (GPUs) (NVIDIA Corporation, 2017a) and Xeon Phi multicore processors (Intel Corporation, 2017) provide powerful solutions for compute- or memory-intensive scientific problems. Such devices can have thousands of processing cores and five times higher memory bandwidth compared to CPUs. One of the most important classes in scientific computing is stencil computation in which array elements are updated iteratively according to a fixed pattern, i.e., *stencil*. For the stencil computation, finite-difference methods have been widely used to solve partial differential equations that describe the time evolution of variables. Stencil applications appear in a wide range of fields, such as computational fluid dynamics (Wu et al., 2004), computational electromagnetics (Adams et al., 2007), and image processing (Ikeda et al., 2014).

However, significant programming effort is required to develop high-performance stencil applications on accelerators. Typically, the compute unified device architecture (CUDA) (NVIDIA Corporation, 2017a) is used as a parallel programming framework for NVIDIA GPUs. A CUDA code consists of *host code* and *device code*, which run on a CPU and GPU, respectively. The host code invokes the device code, which runs in parallel, and transfers data between the host and the device. The key for achieving acceleration is to develop an efficient device code with device specific optimisation techniques. However, this process is time consuming because code and data structures usually must be adapted to the highly-threaded device architecture, which takes full advantage of memory latency hiding mechanisms. For example, arrays of structures must be transformed into structures of arrays to maximise memory access throughput on a GPU (Sung et al., 2012; Ino et al., 2014).

Therefore, directive-based programming frameworks have emerged as a promising approach that facilitates device code development. For example, OpenACC (OpenACC-Standard.org, 2015) provides a collection of compiler directives that can be added to the sequential code. According to the inserted directives, the OpenACC compiler automatically offloads bottleneck workloads from the host to an accelerator device that can perform parallel processing. Thus, the architecture-dependent code (i.e., inserted directives) is clearly separated from the generic code, which expresses the essential computation. In this way, directive-based approaches significantly lower the barrier to accelerated computing. Furthermore, the OpenACCcode retains performance portability such that the code achieves high performance on different architectures. Only a few directives will require modification if a new architecture is released.

However, the advantages of directive-based approaches, i.e., sharing code and data structures between the host and the device, can be problematic from a data size perspective. Accelerator-based implementations usually fail to solve a large problem previously processed by CPU-based implementations because the device memory generally has a smaller capacity than the host memory. This memory exhaustion problem can be resolved by rewriting the code to realise out-of-core computation. However, such code modification eliminates the advantages of directive-based approaches because the loop and data structures must be totally reorganised.

To tackle this issue, we propose a directive-based framework, i.e., a pipelined accelerator (PACC) (Kato et al., 2014; Miki et al., 2016) that can perform pipelined execution of out-of-core stencil computation on a CUDA-compatible GPU. PACC directives can be used to solve problems of the same size as those processed by CPU-based implementations. To realise this, PACC extends the OpenACC specification such that data decomposition and temporal blocking can be applied to stencil computation on a GPU automatically. Given a sequential C code with PACC directives, our source-to-source translator generates an OpenACC code that decomposes large data into smaller *chunks* automatically. These chunks are then processed in a pipelined manner to overlap CPU-GPU data transfer with kernel execution. The generated code is accelerated using a temporal blocking technique that minimises data transfer between the host and device. We extend our preliminary results (Miki et al., 2016) with a multithreaded copy engine that maximises data copy throughput on the host. Furthermore, we apply the presented translator to three stencil computation codes to investigate their out-of-core performance with more than 100 GB data.

The remainder of this paper is organised as follows. Section 2 introduces work related to acceleration of out-of-core stencil computation. Section 3 presents an overview of temporal blocking and summarises how this optimisation technique can be applied to OpenACC code. Section 4 describes the proposed PACC directives and translator. Section 6 provides experimental results. Finally, Section 7 concludes the paper and offers suggestions for future work.

2 Related work

OpenMP (OpenMP Architecture Review Board, 2015) provides a collection of directives that are useful for implementing multithreaded parallel applications on shared memory architectures such as multi-core CPUs. OpenMP 4.5 supports accelerators with OpenACC-like directives. However, host data cannot be partially mapped to device data. Therefore, similar to OpenACC, large problems cannot be solved easily due to device memory exhaustion.

XcalableACC (Nakao et al., 2014), a hybrid model comprising OpenACC and a partitioned global address space (PGAS) language (Murai and Sato, 2013; xcalablemp.org, 2017) realises directive-based programming for multi-node accelerator systems. Similar to the proposed approach that approach can implement a highly-efficient portable code by adding directives to the sequential code. Although the XcalableACC code runs efficiently on multi-node systems with minimum programming effort, the code structure must be modified to use temporal blocking, which is the key technique to minimise data transfer between the host and device.

Maruyama et al. (2011) presented Physis, a programming framework that provides 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 using the message passing interface (MPI) standard (Message Passing Interface Forum, 1994). In addition, they extended Physis to automatically apply temporal blocking to the generated code (Jin et al., 2014). This DSL-based approach is similar to the proposed directive-based approach, i.e., both approaches facilitate acceleration of stencil computation. However, the proposed directive-based approach requires less effort because the sequential code maintains the original code structure, i.e., the code does not need to be adapted to the target DSL.

Endo and Jin (2014) proposed a run-time library, called the hybrid hierarchical runtime (HHRT), which allows CUDA+MPI code to handle out-of-core data with temporal blocking (Endo et al., 2015). The HHRT library virtualises GPU memory with automated swapping to CPU memory. Because the HHRT library requires CUDA code as input, more effort is required to achieve parallelisation on a GPU. The proposed directive-based approach avoids this time-consuming process; however, execution efficiency can be degraded due to higher-level code description. For example, low-level optimisation, such as intrinsic functions (NVIDIA Corporation, 2017a), is not explicitly available in OpenACC code.

Midorikawa and Tan (2015) accelerated stencil computation for 512 GB data, which cannot be stored in either host memory or device memory. They applied temporal blocking to reduce data transfer between the storage device and host memory. Their out-of-core performance was only 13% less than in-core performance; however, application developers had to write their complicated code due to deeply nested loops.

Finally, the latest NVIDIA architecture, called Pascal (P100) (NVIDIA Corporation, 2016), provides a single, unified virtual address space for CPU and GPU memory. This capability can realise out-of-core stencil computation with the CUDA. However, explicit data prefetching is required to implement an efficient pipeline, which overlaps kernel execution with CPU-GPU data transfer. This software pipeline requires significant modification of the sequential code.

3 Stencil computation

Figure 1 shows an example of the four-point stencil code. In this example, a four-point stencil [Figure 2(b)] is iteratively applied to $X \times Y$ elements stored in a two-dimensional (2D) array. Assume that a cross stencil is stored in a $(2r + 1) \times (2r + 1)$ region, i.e., updating an element references itself and its r neighbours in up/down/left/right directions. Here, $r = 1$ for a four-point stencil [Figure 2(b)].

Figure 1 Four-point stencil computation pseudocode

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

Figure 2 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) compared to the original stencil in (b), applying temporal blocking references more neighbouring elements to compute the target element ($k = 2$, in this example) (see online version for colours)

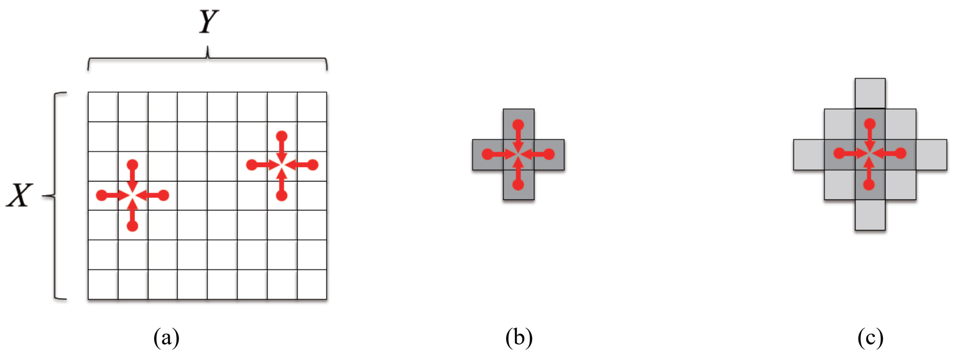
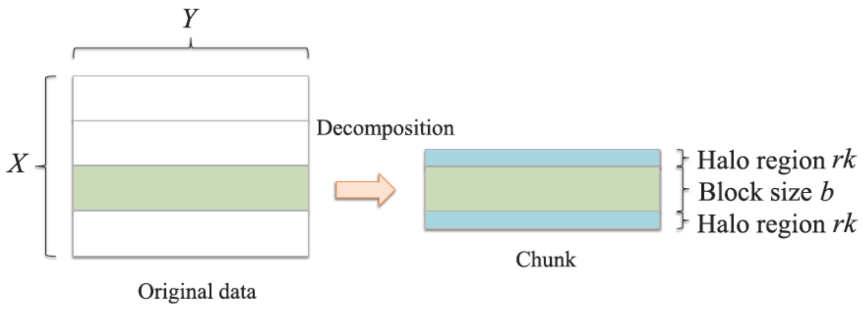
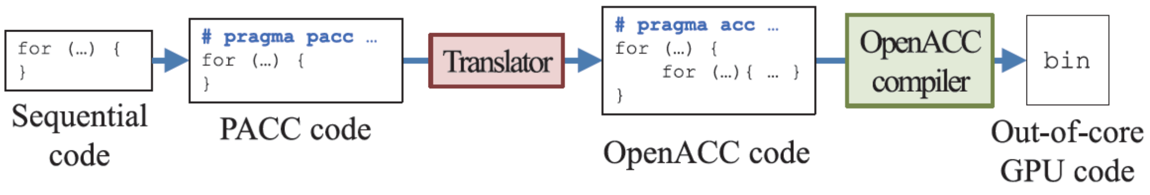


Figure 3 1D block decomposition scheme with halo region (see online version for colours)



Note: 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.

Figure 4 Programming flow with PACC framework (see online version for colours)



Typically, stencil computation is a memory-intensive application; thus, that a cache optimisation technique, called temporal blocking, is usually applied to stencil solvers that deal with time evolution problems. This technique saves memory bandwidth by reusing on-cache data. The computational domain is decomposed into smaller blocks such that each block evolves k contiguous time steps before proceeding to the next block. Hereafter, we refer to parameter k as *the blocking factor*. To realise block-based time evolution, the original `t` loop in Figure 1 must be reorganised into double nested loops; the inner loop evolves k time steps within a block and the outer loop evolves every k time steps for blocks. There are several approaches to realise temporal blocking. In this paper, we consider an overlapped tiling approach due to its simplicity. Other temporal blocking approaches, such as wavefront temporal blocking (Lamport, 1974) and diamond tiling (Orozco and Gao, 2009), are useful for eliminating halos, i.e., overhead related to computation at block boundaries.

Thus, temporal blocking exploits temporal locality to take advantage of caches, which have low latency but limited capacity compared to CPU memory. This trade-off relationship occurs between CPU and GPU memory. Consequently, in GPU-based implementations, temporal blocking is frequently employed to reuse data blocks that have been transferred to GPU memory. Evolving k time steps within the transferred data blocks reduces CPU-GPU data transfer to $1 / k$.

Note that multiple updates within a block require *a halo region* around the block (Figure 3) because updating an element requires its surrounding neighbours. Without this halo region, blocks cannot be processed independently, which prevents full parallelisation. For a cross stencil stored in a $(2r + 1) \times (2r + 1)$ region, k updates for an element refers to its $(2rk + 1) \times (2rk + 1)$ region [Figure 2(c)]. This means that temporal blocking increases the amount of computation because halo regions of adjacent blocks overlap. Thus, there is a trade-off relationship between the amount of computation and the degree of available parallelism. Consequently, blocking factor k must be optimised to maximise 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 (Figure 3).

4 PACC framework

Figure 4 summarises the programming flow with the proposed PACC framework. Application developers must add several PACC directives to sequential C code to realise out-of-core stencil computation. Then, PACC code is input to the PACC translator to generate out-of-core OpenACC code automatically. An OpenACC compiler can be used to obtain an executable file.

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

The proposed translator assumes that the target stencil code satisfies the following constraints.

- Data decomposition constraints. Application developers must not to manually decompose data in the PACC code. The translator also assumes that the data is sufficiently small to be stored in host memory; however, data greater than device memory capacity are acceptable.

- Temporal blocking constraints. The number of total time steps *T* must be fixed at the beginning of program execution. Accordingly, the target program cannot use a `while` loop to terminate execution according to a user-defined error threshold. Furthermore, the device evolves data with *k* time steps at a time; thus, intermediate values between every *k* time steps cannot be accessed from the host.

4.1 PACC directive

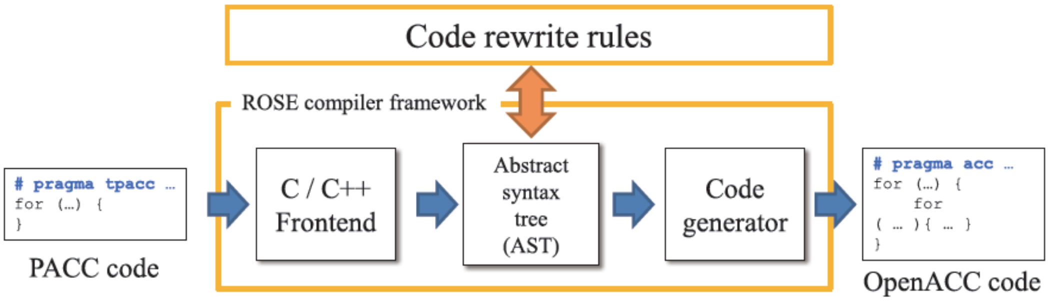
The proposed PACC extends OpenACC (OpenACC-Standard.org, 2015) with three constructs, i.e., the `init`, `pipeline`, and `loop` constructs. Figure 5 shows example PACC code that implements out-of-core stencil computation with temporal blocking. Similar to OpenACC directives, a PACC directive begins with `#pragma pacc`. The extended directives are as follows.

- The `init` construct. This construct allocates host and device buffers to realise data decomposition (Section 5). Consequently, the `init` construct must be placed before the `pipeline` and the `loop` constructs. Furthermore, the proposed multithreaded copy engine requires that the `init` construct be applied to the `for` loop in which the original data to be decomposed are accessed first. We consider that the first access is usually performed for data initialisation (line 6 of Figure 5).
- The `pipeline` construct. The `pipeline` construct specifies the code block to be processed in a pipeline. In Figure 5, the `pipeline` construct is applied to the `for` loop in line 9, 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, Figure 5 specifies `size([0:X][0:Y])` in line 8 to update all elements in an $X \times Y$ array. Finally, the halo clause defines the stencil size for each dimension. In Figure 5, `halo([1:1][1:1])` in line 8 defines a four-point stencil, which accesses left/right/up/down neighbours to update an element. Finally, the `async` clause declares that the code block must be processed asynchronously to realise 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, i.e., `dim`, that associates the loop control variable with the dimension of the array data. For example, the `dim(2)` clause in line 10 indicates that the loop control variable `x` in line 11 corresponds to the second dimension of the array data. Here, we assume that the first (last) dimension has minimum (maximum) stride between consecutive elements. This association is used to adjust the loops for decomposed data. The proposed translator currently deploys a 1D block scheme that decomposes the array data in terms of the last dimension. In Figure 5, the array data will be decomposed along the x axis to yield $b \times Y$ blocks. Note that block size *b* is given by environmental variables.

Figure 5 Example PACC code that implements out-of-core four-point stencil computation

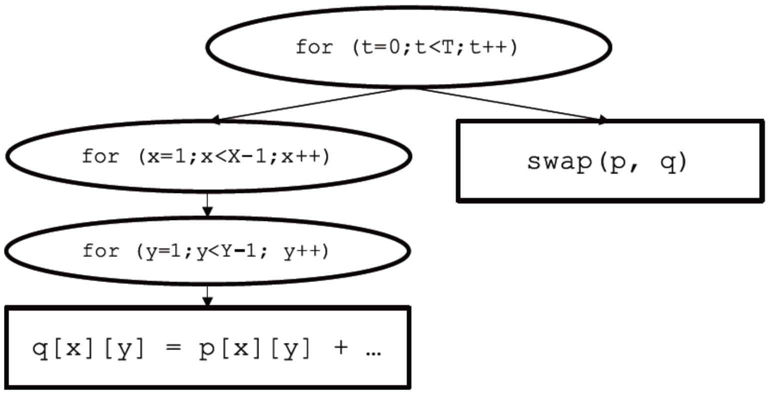
```
 1 float p[X][Y], q[X][Y];
 2
 3 #pragma pacc init
 4 for (x=0; x<X; x++)
 5     for (y=0; y<Y); y++)
 6         p[x][y] = q[x][y] = initial_value;
 7
 8 #pragma pacc pipeline targetinout(p,q) size([0:X][0:Y]) halo([1:1][1:1]) async
 9 for (t=0; t<T; t++){
10     #pragma pacc loop dim(2)
11     for (x=1; x<X-1; x++)
12         #pragma pacc loop dim(1)
13         for (y=1; y<Y-1; y++)
14             q[x][y] = (p[x-1][y] + p[x+1][y] + p[x][y-1] + p[x][y+1]) * 0.25;
15
16     #pragma pacc loop dim(2)
17     for (x=1; x<X-1; x++)
18         #pragma pacc loop dim(1)
19         for (y=1; y<Y-1; y++)
20             p[x][y] = q[x][y];
21 }
```

Figure 6 Overview of the proposed translator (see online version for colours)

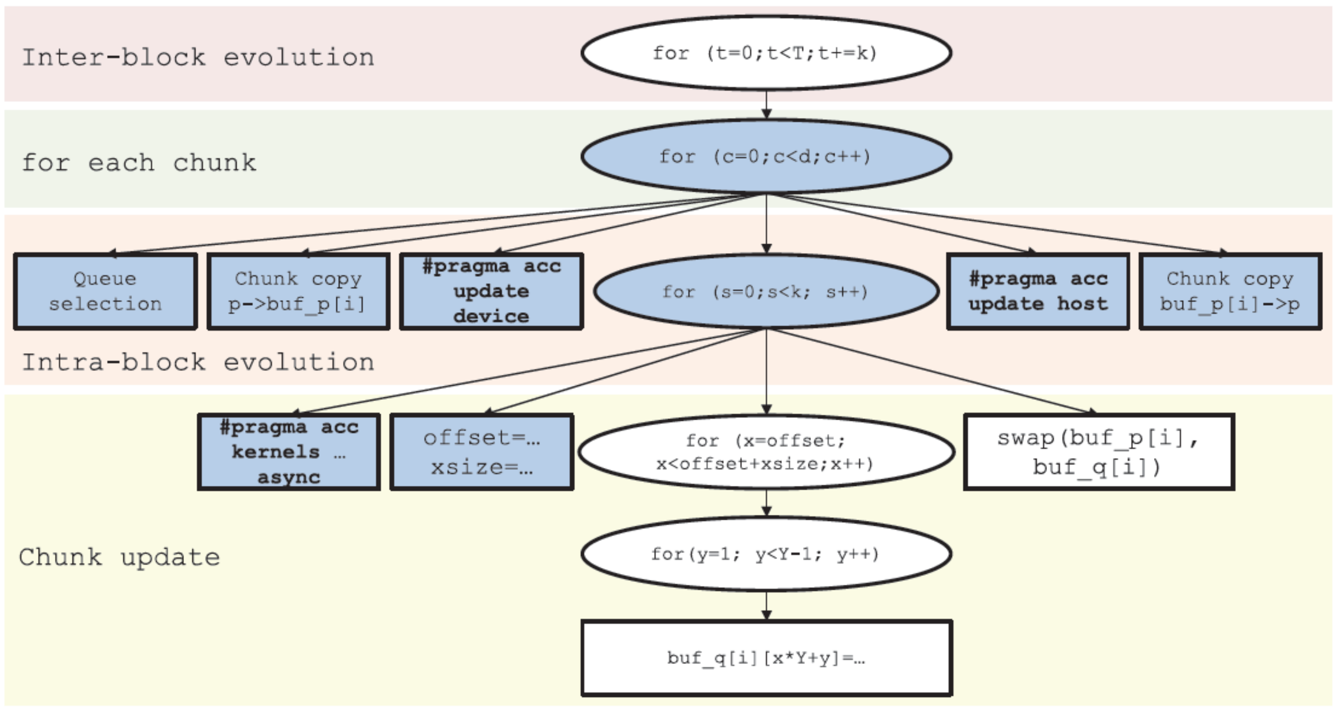


Note: AST-based transformation is performed on the PACC code to generate out-of-core OpenACC code.

Figure 7 Example AST transformation, (a) the translator generates an AST from PACC code (b) the translator then applies code rewrite rules to obtain OpenACC code that can deal with large data with data decomposition and temporal blocking (see online version for colours)

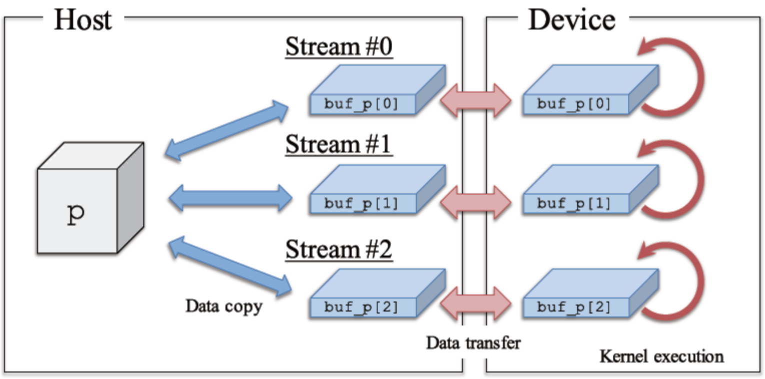


(a)



(b)

Figure 8 Copy-based scheme for data decomposition (see online version for colours)



Note: Host and device buffers `buf_p` are allocated for copying blocks and their halos from the original array `p`.

4.2 PACC translator

The proposed PACC translator was implemented using the ROSE compiler infrastructure (Liao et al., 2010; rosecompiler.org, 2017), which provides a C/C++ frontend to generate an abstract syntax tree (AST) of the input code (Figure 6). The generated AST is then traversed to detect PACC directives, i.e., nodes with the `pacc` attribute. During this traversal, detected directives are parsed to retrieve the key information given by succeeding clauses, such as stencil size *r*, variables to be decomposed, and their data size. The detected nodes are marked explicitly for modification and code rewrite rules are applied to the detected nodes in the next traversal. Finally, the rewritten AST is input to a code generator to obtain the pipelined OpenACC code.

The code rewrite rules can be summarised as follows:

1 The `init` construct rule. The AST node that corresponds to the `init` construct is replaced with AST nodes responsible for:
   - obtaining blocking factor *k* and the number of data decompositions *d* via environmental variables
   - computing block size *b* from *d* and the array size *X* to be decomposed
   - allocating host and device buffers
   - performing physical data allocation such that the data copy throughput is maximised according to a first touch policy (Section 5.4).

2 The `pipeline` construct rule. The original `for` loop is reorganised into double nested for loops to realise temporal blocking. To achieve this, the AST is transformed as shown in Figure 7. The code rewrite rule replaces the AST node that corresponds to the time evolution `for` loop with AST nodes that correspond to the inter-block evolution, per-chunk operations, and intra-block evolution `for` loops. 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 in the host buffers, and exchange them between host and device buffers. Finally, memory accesses to the original arrays are replaced with accesses to the buffers by updating AST node attributes appropriately.

3 The `loop` construct rule. This rule updates the transformed double nested loops with a new initialisation and condition to adapt the loop structure for block-based evolution. The rule first locates the `for` loop that must be updated due to data decomposition. For example, the `for` loop in line 11 of Figure 5 is selected to update its initialisation and condition because this loop is associated with a `dim(2)` clause that has the greater value (2) as the argument of dim clauses. The new initialisation and condition is appropriately given by variables `offset` and `xsize`, which we present in Section 5.

5 Out-of-core stencil computation with OpenACC

Out-of-core stencil computation with OpenACC requires a data decomposition scheme, which divides large data into smaller chunks such that each chunk can be entirely stored in device memory. In this section, we describe how our generated out-of-core code realises data decomposition and temporal blocking with OpenACC.

5.1 Data decomposition

Data decomposition cannot be realised by simply adding OpenACC directives to the sequential code. Given a large array that exceeds the device memory capacity, a different array must be allocated in the host memory, as shown in Figure 8. This is due to the assumption of the OpenACC specification, which allocates the same variables for both host and device memory. As can be seen in Figure 8, large array `p` can be processed successfully 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 device memory capacity. 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.

Figure 9 Pseudocode of out-of-core stencil computation with temporal blocking

```
Set blocking factor k, number d of decompositions, and number num_queue of queues from
      environmental variables;
Compute block size b from d and data size X;
Allocate buf_p[0], buf_p[1], ..., buf_p[num_queue-1] on host memory;
#pragma acc create (buf_p[0:num_queue][0:b+2*r*k], ...)
Allocate g[0], g[1], ..., g[num_queue-1] on host memory to store completed chunk numbers;

for (t=0; t<T; t+=k) { // outer loop for time evolution
    for (c=0; c<d; i = (i+1) % num_queue) { // for each chunk

        if (!acc_async_test(i)) // Select i as the ID of an idle queue
            continue;

        if (g[i] != NONE) {
            Copy the g[i]-th chunk from buf_p[i] to p in parallel;
            g[i] = NONE;
        }

        Copy the c-th chunk from p to buf_p[i] in parallel;
        #pragma acc update device (buf_p[i:1][0:b+2*r*k], ...) async (i)

        for (s=0; s<k; s++) { // inner loop for time evolution
            #pragma acc kernels present (buf_p[i:1][0:b+2*r*k], ...) async(i)
            {
                offset = r*(s+1);
                xsize  = b+2*r*(k-1-s);

                #pragma acc loop independent
                for (x=offset; x<offset+xsize; x++)
                    #pragma acc loop independent
                    for (y=1; y<Y-1; y++)
                        #pragma acc loop independent
                        for (z=1; z<Z-1; z++)
                            buf_q[i][x*Y*Z+y*Z+z] += buf_p[i][(x+1)*Y*Z+y*Z+z] + ...;
            }
            #pragma acc kernels present (buf_p[i:1][0:b+2*r*k], ...) async(i)
            {
                #pragma acc loop independent
                for (x=offset; x<offset+xsize; x++)
                    #pragma acc loop independent
                    for (y=1; y<Y-1; y++)
                        #pragma acc loop independent
                        for (z=1; z<Z-1; z++)
                            buf_p[i][x*Y*Z+y*Z+z] = buf_q[i][(x+1)*Y*Z+y*Z+z];
            }
        }

        #pragma acc update host (buf_p[i:1][0:b+2*r*k], ...) async (i)
        g[i] = c;
        c++;
    }
}
for (i=0; i<num_queue; i++) { // Clean up all queues
    #pragma acc wait(i)
    if (g[i] != NONE)
        Copy the g[i]-th chunk from buf_p[i] to p in parallel;
}
```

Notes: This applies a cross stencil of $(2r + 1) \times (2r + 1)$ elements to the computational domain of $(X - 2r) \times Y \times Z$ elements. A 1D block decomposition scheme is applied to the computational domain such that the data is decomposed into d blocks of size $(X - 2r) / d \times Y \times Z$.

An alternative solution to avoid device memory exhaustion is a map-based scheme that maps the allocated device buffers to the original host array with the `acc_map_data()` API (OpenACC-Standard.org, 2015). This direct mapping approach allows GPU threads to update values in the original array such that the host buffer (i.e., the copy overhead) can be eliminated. However, we decided to use the copy-based scheme because the PGI compiler 15.10, which we used for preliminary evaluation in 2015, allocated pageable memory (NVIDIA Corporation, 2017a) if the `acc_map_data()` function was used in the OpenACC code; the map-based scheme was executed in a synchronous manner, which avoids pipelined execution. Notice that the latest PGI compiler successfully produces the map-based code that can run asynchronously. Another drawback of the map-based scheme is that it may fail to realise multidimensional decomposition, which produces many mapping points. Direct mapping is primarily useful for a contiguous memory region.

Figure 9 shows the pseudocode for an OpenACC-based out-of-core stencil computation that deploys a 1D block decomposition scheme and temporal blocking. In line 3, `num_queue` buffers, i.e., `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 (line 4). Note that multiple buffers are necessary to realise efficient pipelining. A single buffer causes data dependence between kernel execution and host-device data transfer, which avoids overlapped execution. Note that these buffers are reused to minimise allocation overhead during program execution. The buffers are allocated once at the beginning of execution. In line 5, `num_queue` buffers, i.e., `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` using the `update` clause in line 19. Using these buffers, a kernel function is invoked (lines 22 and 35) to update elements in the chunks for *k* time steps. To achieve 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 (lines 22 and 35) indicates that chunks to be accessed have already been sent to the device buffers `buf_p`; thus, additional data transfer can be avoided at kernel invocation. The updated elements are then transferred back to the host buffers (line 47). Host buffers are copied to the original array `p` (line 14) when the same queue is selected next time.

Note that the code modification mentioned above is necessary to avoid device memory exhaustion, which results in execution failure. Without this code modification, the original code fails to run if array `p` is too large to fit in device memory. However, as shown in Figure 9, this modification reorganises the loop structure of the original code (Figure 1), thereby diminishing the benefits of directives. In other words, the modified code degrades the performance portability because it leads to inefficient execution on machines equipped with large-capacity memory.

5.2 *Temporal blocking*

Efficient temporal blocking can be implemented by realising block-based time evolution. To achieve this, 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 Figure 9. The outer `t` loop (line 7) is responsible for processing blocks every *k* time steps while the inner `s` loop (line 21) is responsible for processing a block for consecutive *k* time steps. To facilitate automated parallelisation, `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 executed efficiently without synchronisation.

5.3 *Pipelined execution*

Asynchronous APIs are deployed to realise software-based pipelining that overlaps kernel execution with host-device data transfer. In Figure 9, a chunk is assigned to one of the `num_queue` asynchronous queues. In line 10, a queue is tested to determine whether all associated (asynchronous) operations of the queue have completed using the `acc_async_test()` API (OpenACC-Standard.org, 2015). This non-blocking API immediately returns a value; thus, multiple chunks can be processed simultaneously with different queues. Note that different queues are responsible for data-independent tasks because the host and device buffers `buf_p[i]` are dedicated to the `i`-th queue to realise concurrent execution. For example, chunks are transferred in line 19 with an `async` clause and its argument *i*, which specifies the queue ID to be used.

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). This step is parallelised with OpenMP (OpenMP Architecture ReviewBoard, 2015) (Section 5.4).

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 executes the kernel iteratively 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).

Similar to the first step, the last step is also parallelised with OpenMP. Note that the pseudocode code in Figure 9 cannot overlap the first step with the last step because each step uses all CPU cores to copy data. However, the kernel execution step can be overlapped with the data transfer steps if chunks are assigned to different queues. A full overlap can be established if the first step and the last step are assigned asynchronously to different CPU cores. However, we decided to occupy all CPU cores for either the first step or the last step because the data copy throughput was maximised when using all CPU cores (Section 5.4). Overlapping the first step with the last step, i.e., performing data copy steps for different chunks simultaneously, decreases data copy throughput due to lower data locality.

5.4 *Multithreaded copy engine*

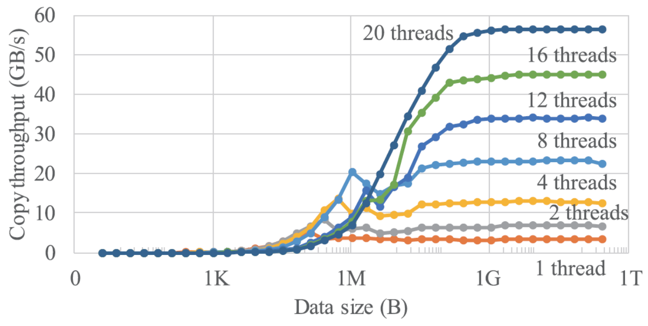
The basic idea for accelerating the data copy steps is to:

1 parallelise copy operations

2 minimise traffic between different CPU sockets.

Firstly, the data copy steps mentioned above can be accelerated using multiple CPU threads in the non-uniform memory access architecture because memory access throughput depends on the number of CPU threads participating in the memory access operation. Figure 10 shows benchmarking results using our copy engine multithreaded with OpenMP (OpenMP Architecture Review Board, 2015). The data to be copied were dynamically allocated with the `posix_memalign()` function. The benchmark was compiled with the PGI compiler 16.10 and executed with setting an environmental variable `MP_BIND` to `y`, which binds threads to physical CPU cores. As shown in Figure 10, the effective bandwidth reached 56 GB/s, which was close to that achieved with the standard STREAM benchmark (McCalpin, 1995); however, the achieved bandwidth seems to be far from the peak memory bandwidth of 102.4 GB/s, so that it may be further increased by considering a write allocate policy. Thus, data copy operations must be parallelised to yield full memory access performance. To achieve this, we multithreaded copy operations with OpenMP, as shown in Figure 11.

Figure 10 Multithreaded data copy throughput with different numbers of CPU threads (see online version for colours)



Notes: Two Xeon E5-2680 v2 (Ivy Bridge-EP) CPUs were deployed for measurement. The peak memory bandwidth was 102.4 GB; 51.2 GB/s per socket. The speedup over the single-thread version reached a factor of 16.7 when using 20 threads.

Secondly, efficient parallelisation requires physical memory allocation to be optimised such that copy operations exchange minimum data between different CPU sockets. Memory performance is maximised when CPU cores access their local memory (rather than remote memory belonging to other CPU sockets). For physical memory allocation, Linux operating systems deploy:

1 a local allocation scheme, which allocates memory on the node of the CPU that triggered the allocation

2 this allocation is carried out when the page is first touched by a thread (Lameter, 2013).

Consequently, memory access throughput in the data copy steps depends on how the data copy is initialised in the code. This explains why the `init` construct must be applied to the `for` loop in which the data are accessed first (Figure 5). In other words, our translator rewrites this initialisation loop such that it has the same structure as the loop that implements the data copy step. Thus, we realise the localisation by using the same loop structure and OpenMP directives for both data initialisation and data copy operations. Furthermore, the generated code must be executed by setting the environmental variable `MP_BIND` to `y`, as mentioned above, so that thread migration is avoided during execution.

Figure 11 shows how our multithreaded copy engine implements the data copy step. Here, the c^{th} 2D chunk is copied from the original array `p` to the host buffer `buf_p` using CPU threads. First, the array data must be initialised with OpenMP directives to take advantage of the first touch policy. The multithreaded part is then assigned to CPU threads by a block-cyclic distribution with a block size of 4 KB, i.e., the page size, which maximises the copy throughput. To realise this distribution, the `collapse` clause is specified with $v - 1$, where v is the dimension of the data to be copied. We use the `static` schedule to assign 4 KB data blocks to threads. For 4-byte data, this can be realised by specifying `schedule(static, 1024)`. Secondly, the main block in the chunk is copied using OpenMP directives accordingly. Then, the left and right halos are copied with a single thread.

Note here that the page size can be increased to 2 MB and 1 GB if transparent huge pages are activated due to large amount of host memory. We found that our experimental machine always activated this capability but there was no significant difference between the results for 2 MB data blocks (i.e., `schedule(static, 1024*512)`) and those for 4 KB data blocks.

6 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 has two direct memory access engines; thus, data transfer from host to device overlapped that of the opposite direction. We used the PGI Compiler 16.10 (NVIDIA Corporation, 2017b), CUDA 8.0 (NVIDIA Corporation, 2017a), and Ubuntu 14.04. The peak bandwidth of device memory is 288 GB/s on the K40 card, which yielded an effective bandwidth of 182.8 GB/s according to the `bandwidthTest` benchmark (NVIDIA Corporation, 2017a).

The stencil computation codes used for the experiments were the Jacobi method, the Himeno (2017) benchmark, and the constraint interpolation profile (CIP) method (Yabe et al., 2001), which are summarised in Table 1. The Jacobi method is an iterative solver for a system of linear equations. The Himeno benchmark is a linear solver for 3D pressure Poisson equations. The CIP method is a solver for hyperbolic partial differential equations. All three experimental codes processed out-of-core data that could not be stored entirely in device memory.

6.1 Scalability analysis

First, we 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 experimentally investigated the best execution parameters, i.e., blocking factor *k* and block size *b*, for each data size. We then executed the generated codes to analyse their scalability and effective performance with varying data size (Figure 12).

Figures 12(a) and 12(c) show that PACC maintained effective performance for the Jacobi and CIP methods with 107 GB data, which was 8.5 times greater than the device memory capacity. For these methods, we successfully found the best parameter values for *k* and *b* that fully overlapped CPU-GPU data transfer with GPU execution. As can be seen in these figures, pipelined execution significantly increased the effective performance.

Figure 11 Example code for multithreaded copy operations for a 2D chunk, (a) data initialisation (b) data copy

```
#pragma omp parallel for schedule(static, 1024) private(y) collapse(1)
for (x=0; x<b; x++)  // multithreaded
    for (y=0; y<Y; y++) // vectorized
        p[x][y] = initial_value;
```

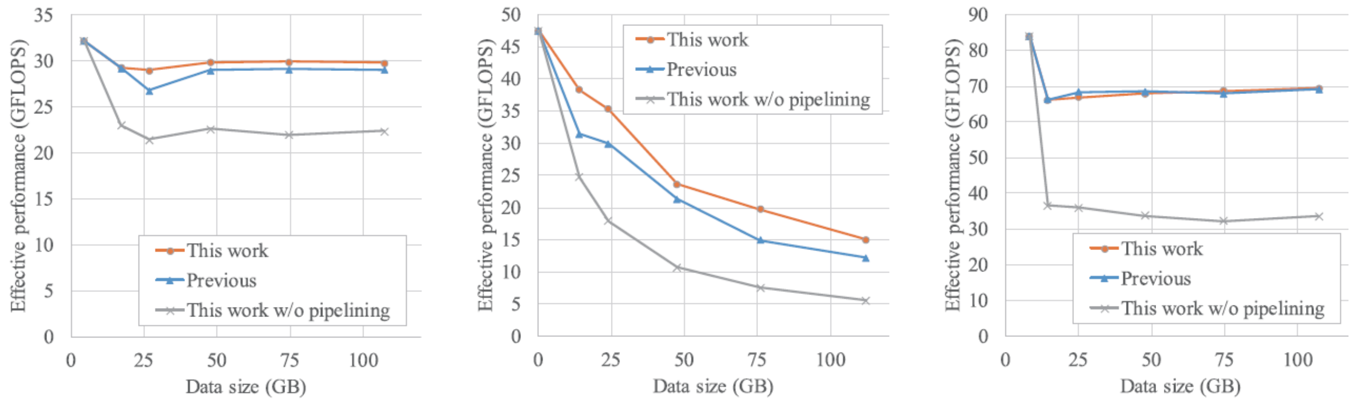
(a)

```
offset = c*b + r; // c: the chunk ID, b: block size, r: stencil size

// data copy for block
#pragma omp parallel for schedule(static, 1024) private(y) collapse(1)
for (x=0; x<b; x++)  // multithreaded
    for (y=0; y<Y; y++) // vectorized
        buf_p[i][(x + r*k) * Y + y] = p[x + offset][y];

// data copy for left halo
for (x=-r*k; x<0; x++)
    for (y=0; y<Y; y++)
        buf_p[i][(x + r*k) * Y + y] = p[x + offset][y];

// data copy for right halo
for (x=b; x<r*k; x++)
    for (y=0; y<Y; y++)
        buf_p[i][(x + r*k) * Y + y] = p[x + offset][y];
```

(b)

Note: To take advantage of the first touch policy, the same loop structure and OpenMP directives are used for both data initialisation and data copy operations.

Figure 12 Effective performance with different data sizes, (a) Jacobi method (b) Himeno benchmark (c) CIP method (see online version for colours)



(a) (b) (c)

Notes: Plots for the smallest data represent in-core performance. Plots for data greater than 12 GB represent out-of-core performance.

Figure 13 Effective performance with different block size *b* and blocking factor *k*, (a) Jacobi method (b) Himeno benchmark (c) CIP method (see online version for colours)

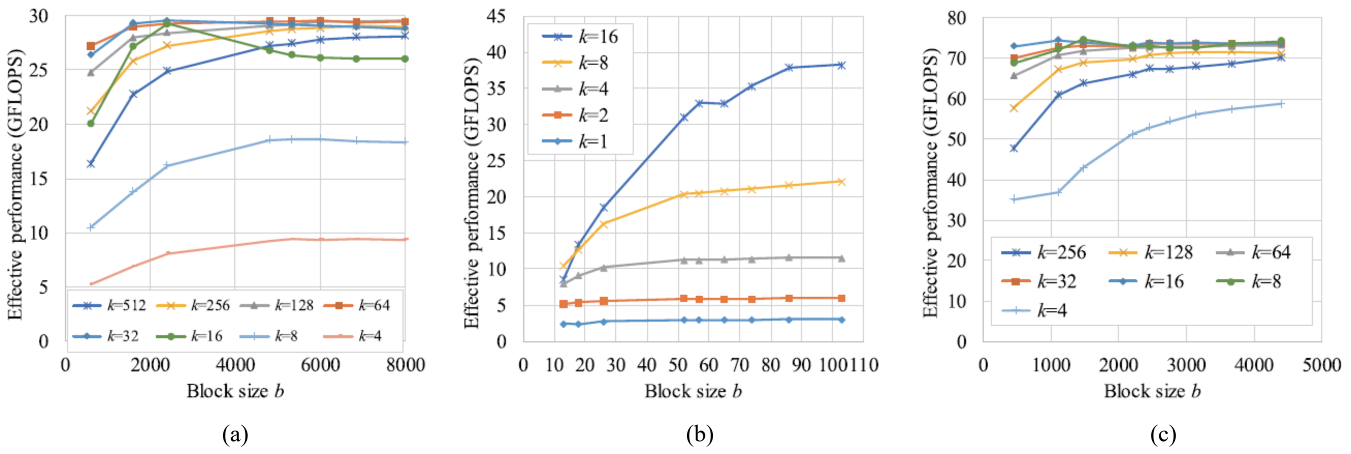
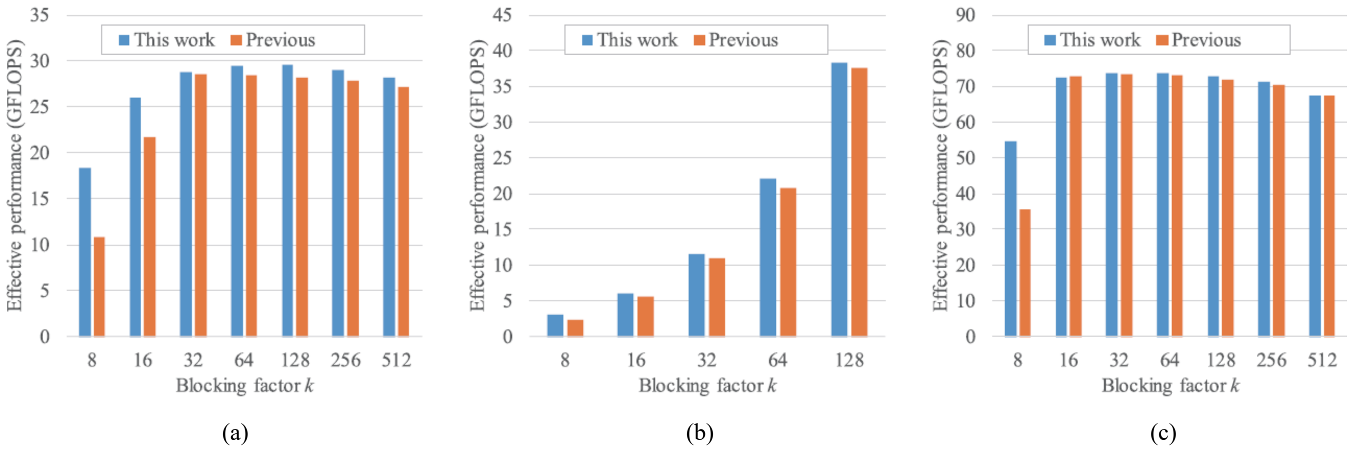


Figure 14 Effective performance with different blocking factor *k* for fixed block size *b*, results for (a) Jacobi method, (b) Himeno benchmark and (c) CIP method (see online version for colours)



In contrast, the performance of the Himeno benchmark decreased monotonically as the data size increased. This performance degradation was due to our 1D block scheme, which prevented the best execution parameters *b* and *k* to be selected for large data. In other words, the best parameter values were only selectable with small chunks, which cannot be produced with our 1D block scheme. The Himeno benchmark also allocated many arrays (Table 1), which avoided *b* and *k* from being sufficiently large due to the limitation of device memory capacity. A multidimensional block scheme was required to deal with this issue. A detailed analysis is presented in Section 6.2.

Finally, we compared our out-of-core implementation with an in-core implementation that processed small data (Figure 12). The in-core versions of the Jacobi, Himeno, and CIP methods solved problem sizes of 24,000 × 24,000 (4.6 GB), 128 × 128 × 256 (0.2 GB), 16,000 × 16,000 (8.2 GB), respectively. The in-core data were updated iteratively with the same number of time steps as the out-of-core implementation. Note that effective performance was derived from the execution time that included the data transfer time between the CPU and GPU. The highest in-core performance for the CIP method reached 83.9 GFLOPS, which was 17% greater than the performance achieved by our out-of-core implementation. Data transfer time usually limits the performance of GPU applications; therefore, we think that this 17% slowdown is acceptable for highly-efficient, out-of-core stencil computation with a directive-based approach.

6.2 *Performance analysis*

Using the generated codes, we measured their effective performance to investigate the impact of execution parameters, such as blocking factor *k* and block size *b*.

6.2.1 *Jacobi method*

We solved a problem of size $X \times Y = 48{,}000 \times 48{,}000$ elements, which consumed 18.4 GB of memory space in $T = 2{,}048$ time steps. Figure 13(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. Note that effective performance *E* does not consider cache effects.

As shown in Figure 13(a), the maximum performance of the Jacobi method was 29.5 GFLOPS, which was obtained when the block size and the blocking factor were $b = 8{,}000$

and k = 128, respectively. This figure also shows that performance was determined by blocking factor k rather than block size b.

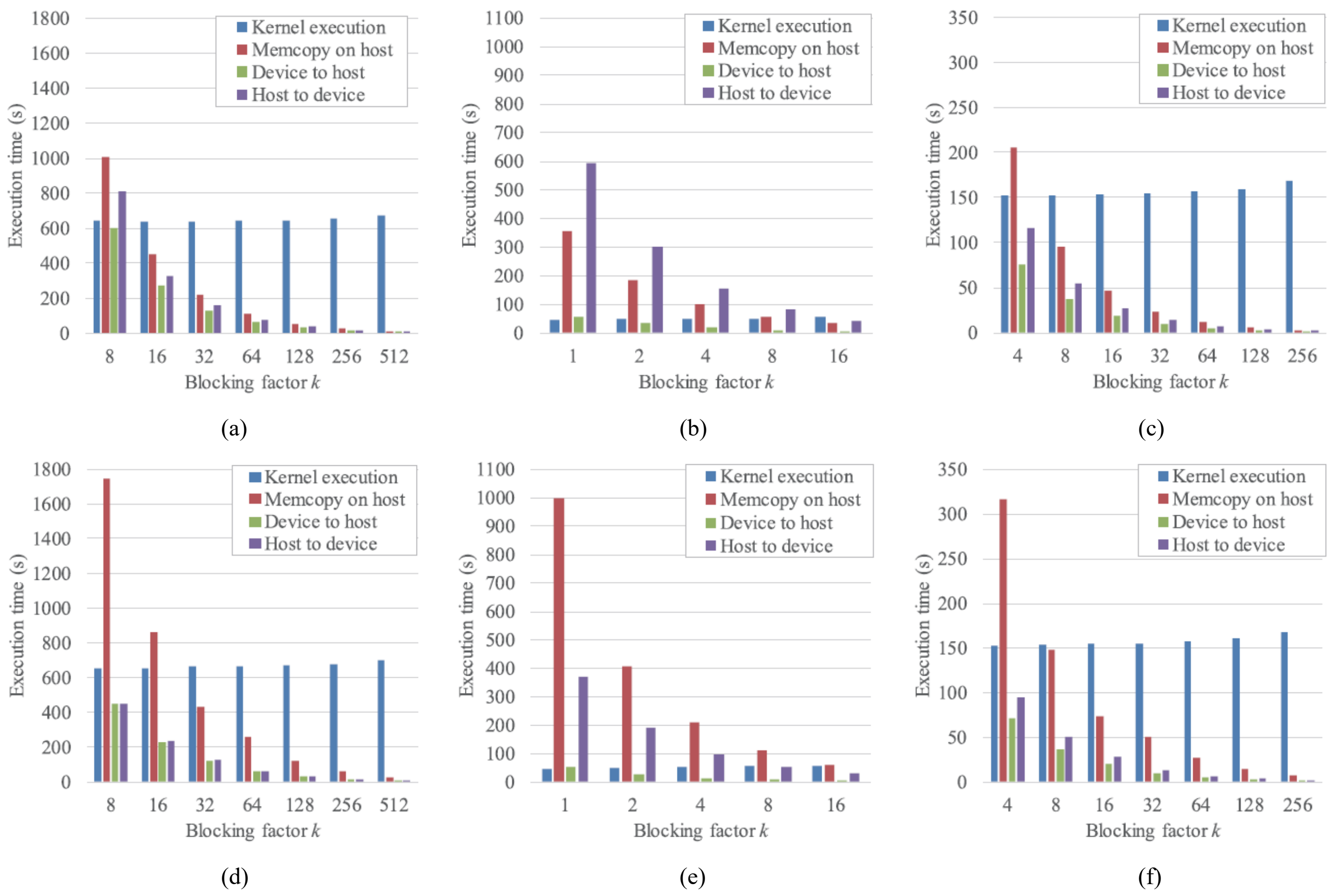
We then measured the effective performance with varying blocking factor k at fixed block size b = 8,000 [Figure 14(a)]. The effective performance did not increase monotonically with blocking factor k due to the trade-off discussed in Section 3. To investigate this behaviour, we analysed the breakdown of execution time, which we show in Figure 15(a). We found that the copy overhead determined overall performance when $k < 16$. This overhead was inversely proportional to k; thus, the effective performance increased with k when $k < 16$. In contrast, the effective performance decreased slightly as we increased k from 16. This performance degradation was caused by temporal blocking, which increased kernel execution time due to redundant computation. Thus, the best trade-off point was obtained when $k = 16$, where the data transfer and copy overheads were fully overlapped with the kernel execution time.

The impact of the multithreaded copy engine can be seen by comparing Figure 14(a) and Figure 14(d). When $k = 8$, our multithreaded engine successfully decreased the copy overhead from 1,747 ms to 1,007 ms. However, this accelerated copy operations increased the data transfer time from 450 ms (449 ms) to 808 ms (599 ms) for the host-to-device direction (the device-to-host direction, respectively). Because the data transfer between the host and device consumes not only the PCIe bus bandwidth but also the host memory bandwidth, these timing behaviour clearly shows that the host memory bandwidth limits the entire performance if blocking factor k is not appropriately tuned; the available host memory bandwidth must be higher than the PCIe bandwidth to obtain the maximum PCIe bandwidth because the data on the PCIe bus comes from the host memory (or sent to the host memory). For example, the PCIe bandwidth can be decreased from 10 GB/s to 6 GB/s if the engine copies data at 50 GB/s out of 56 GB/s; in this case, 6 GB/s is left for the data transfer.

6.2.2 *Himeno benchmark*

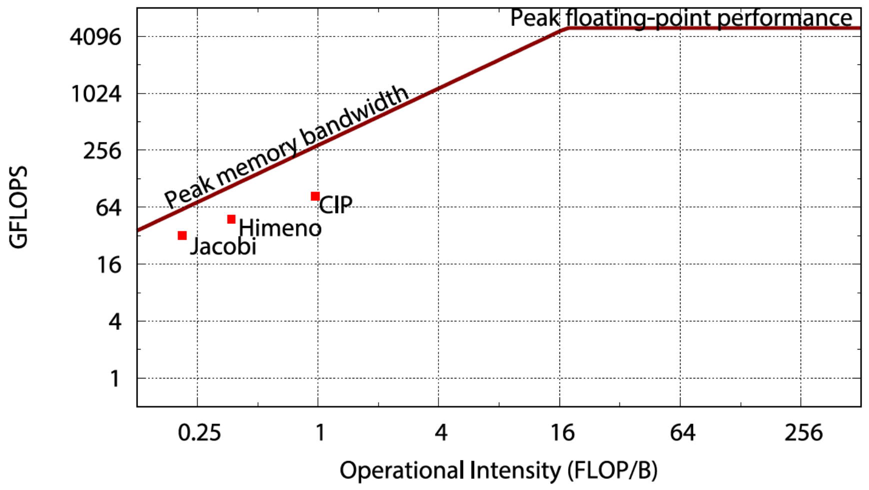
Here, we solved a problem size XL (512 × 512 × 1,024, 15 GB) with $T = 256$ time steps. Given 3D data of $X \times Y \times Z$ elements, the effective performance E is given by $E = 34(X - 2)(Y - 2)(Z - 2)T / t$.

Figure 15 Breakdown of execution time with different blocking factor k for fixed block size b, (a) Jacobi method (b) Himeno benchmark (c) CIP method with the proposed method (d) Jacobi method (e) Himeno benchmark (f) CIP method with the previous work (see online version for colours)



Source: Miki et al. (2016)

Figure 16 Roofline model for our experimental machine (Tesla K40), which had the peak memory bandwidth of 288 GB/s and the peak floating-point performance of 5,000 GFLOPS (see online version for colours)



Note: Increasing cache hit rate (i.e., the operational intensity) is required to achieve high performance for the three benchmarks.

Table 1 Stencil computation codes used for experiments

Code	*#of arrays*	*Array size*	*Data size (GB) (GB)*	*Stencil*	*T (step)*	*f (FLOP)*	*l (B)*	*s (B)*	*G (FLOP/B)*	*G′*
Jacobi	2	120 K × 120 K	107	4 point	2048	4	16	4	0.14	0.21
Himeno	14	1 K × 1 K × 2 K	112	18 point	256	34	128	4	0.24	0.37
CIP	8	60 K × 60 K	107	9point	256	91	120	12	0.58	0.97

Notes: T, f, l, s, and G represent the number of time steps, the number of floating point operations per element, the amount of writes per element, the amount of reads per element, and arithmetic intensity (Harris, 2005), respectively. The arithmetic intensity G is given by $G = f / (l + s + 8)$, which do not consider cache effects. By contrast, the operational intensity (Williams et al., 2009), G', considers cache effects according to profiling results.

As shown in Figure 13(b), the effective performance of the Himeno benchmark increased monotonically with blocking factor k. As a result, the maximum performance of 38.2 GFLOPS was obtained when b = 103 and k = 16, where the buffer size was maximised. When $k > 16$, we failed to execute the PACC code due to device memory exhaustion. Greater memory capacity was required to find the best blocking factor that might be obtained when $k > 16$.

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

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 pack several data segments into a host buffer and unpack a device buffer to the original array. Furthermore, memory references in the kernel must be updated to access the packed small array accordingly.

Figure 15(b) shows the breakdown of execution time. We found that the data copy time spent on the host was nearly the same as the kernel execution time when k = 16. Consequently, blocking factor k = 16 seems to be the best configuration for the Himeno benchmark of the problem size XL.

As for the kernel performance, we investigated the efficiency of kernel execution using the roofline model (Williams et al., 2009). The arithmetic intensity (Harris, 2005) and operational intensity (Williams et al., 2009), G and G', respectively, shown in Table 1 consider swap overheads incurred at every time step. The arithmetic intensity G is based on traffic between the processor and cache, and thus cache effects are not considered. By contrast, the operational intensity G' considers cache effects according to the traffic between the cache and memory. To obtain the traffic for the operational intensity, we used the `nvprof` tool (NVIDIA Corporation, 2017a) with the in-core data to measure four metrics: `gputime`, `dram_read_throughput`, `dram_write_throughput`, and `flop_count_sp`. Figure 16 shows the roofline model for our experimental machine. As shown in this figure, the peak memory bandwidth rather than the peak floating-point performance determines the kernel performance. Given the operational intensity of 0.37, the performance of the Himeno benchmark was limited by 106.6 GFLOPS, whereas the measured performance reached 47.5 GFLOPS.

We also investigated the efficiency of kernel execution in terms of effective memory bandwidth. As mentioned in Section 6, the maximum bandwidth is given by 182.8 GB/s. On the other hand, the effective memory bandwidth B can be given by $B = G' \times E$. Given in-core performance of $E = 47.5$ GFLOPS, the effective memory bandwidth B reached 195.6 GB/s, which seems reasonable compared to the maximum bandwidth mentioned above. However, this result implies that the OpenACC compiler can be further improved to maximise cache effects for higher performance. A tiling approach may also be useful for tackling this issue.

6.2.3 CIP method

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

In Figure 13(b), the effective performance reached 73.4 GFLOPS when $b = 2750$ and $k = 16$. Compared to the Jacobi method and Himeno benchmark, the CIP method maximised performance with a relatively small blocking factor $k = 8$ due to its high arithmetic intensity (Harris, 2005) (Table 1). In other words, the ratio of kernel execution time to CPU-GPU data transfer time was relatively high; thus, the impact of temporal blocking was maximised rapidly with low k. Therefore, temporal blocking failed to demonstrate significant improvement when $k \geq 8$.

Figures 14(c) and 15(c) show the effective performance with different blocking factor k and the breakdown of execution time, respectively. As estimated above, the best trade-off point was found for $k = 8$ [Figure 14(c)], where the copy overhead was close to the kernel execution time, as shown in Figure 15(c).

Table 2 Code lengths in lines

Code	*Original C*	*In-core OpenACC*	*PACC*	*Out-of-core OpenACC*
Jacobi	98	104	104	314
Himeno	220	235	236	614
CIP	149	164	160	458

6.3 Code length analysis

Table 2 compares the lengths of the original C code, the in-core (handwritten) OpenACC code, the PACC code, and the out-of-core (generated) OpenACC code. As can be seen, the PACC code requires several additional lines over the original C code to process out-of-core data on the GPU. In contrast, the out-of-core OpenACC code generated by the PACC translator doubled in length, implying that significant modification is necessary over the in-core OpenACC code. Thus, we think that the proposed PACC framework is useful for reducing programming effort for out-of-core stencil computation on a GPU.

7 Conclusions

We have presented PACC, an extension of OpenACC directives, and its source-to-source translator. The proposed PACC method can accelerate out-of-core stencil computation with temporal blocking on a GPU. Given PACC code, the translator generates OpenACC code such that the code decomposes large data into smaller chunks, which are then processed in a pipelined manner to hide 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. We have also presented a multithreaded copy engine required to maximise memory copy throughput on the host.

In experiments, we added PACC directives to three stencil computation codes, i.e., the Jacobi, Himeno, and CIP methods.We found that out-of-core performance for data greater than 100 GB reached 69.3 GFLOPS on a Tesla K40 GPU, which was only 17% less than the in-core performance with 8 GB data. Thus, the proposed PACC directives facilitate high-performance out-of-core stencil computation on a GPU. However, we found that our 1D block scheme was insufficient for selecting the best execution parameters for the Himeno benchmark, which deals with many 3D data during execution.

In future, we plan to develop an automated framework for finding the best execution parameters for block size b and blocking factor k. We also plan to support a multidimensional decomposition scheme to increase the effective performance of the Himeno benchmark, which deals with several multidimensional arrays.

Acknowledgements

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

Adams, S., Payne, J. and Boppana, R. (2007) ‘Finite difference time domain (FTDT) simulations using graphics processors’, in *Proc. High Performance Computing Modernization Program Users Group Conf. (HPCMP-UGC’07)*, pp.334–338.

Endo, T. and Jin, G. (2014) ‘Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations’, in *Proc. 16th IEEE Int. Conf. Cluster Computing (CLUSTER’14)*, pp.132–139.

Endo, T., Takasaki, Y. and Matsuoka, S. (2015) 'Realizing extremely large-scale stencil applications on GPU supercomputers', in *Proc. 21st Int'l. Conf. Parallel and Distributed Systems (ICPADS'15)*, pp.625–632.

Harris, M. (2005) 'Mapping computational concepts to GPUs', in Pharr, M. and Fernando, R. (Eds.): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Chapter 31, Addison-Wesley, Reading, MA.

Himeno, R. (2017) *Himeno Benchmark* [online] http://accc.riken.jp/en/supercom/himenobmt/ (accessed 30 June 2017).

Ikeda, K., Ino, F. and Hagihara, K. (2014) 'Efficient acceleration of mutual information computation for nonrigid registration using CUDA', *IEEE J. Biomedical and Health Informatics*, Vol. 18, No. 3, pp.956–968.

Ino, F., Shigeoka, K., Okuyama, T., Motokubota, M., Ino, F. and Hagihara, K. (2014) 'A parallel scheme for accelerating parameter sweep applications on a GPU', *Concurrency and Computation: Practice and Experience*, Vol. 26, No. 2, pp.516–531.

Intel Corporation (2017) *Intel Xeon Phi Product Family* [online] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html (accessed 30 June 2017).

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

Kato, T., Ino, F. and Hagihara, K. (2014) '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)*.

Lameter, C. (2013) 'An overview of non-uniform memory access', *Communications of the ACM*, 56(9), 59–54.

Lamport, L. (1974) 'The parallel execution of DO loops', *Communications of the ACM*, Vol. 17, No. 2, pp.83–93.

Liao, C., Quinlan, D.J., Vuduc, R. and Panas, T. (2010) '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)*, pp.308–322.

Maruyama, N., Nomura, T., Sato, K. and Matsuoka, S. (2011) '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)*, 12pp.

McCalpin, J.D. (1995) 'Memory bandwidth and machine balance in current high performance computers', *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pp.19–25 [online] http://www.cs.virginia.edu/~mccalpin/papers/balance/ (accessed 30 June 2017).

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

Midorikawa, H. and Tan, H. (2015) 'Locality-aware stencil computations using flash SSDS as main memory extension', in *Proc. 15th IEEE/ACM Int'l. Symp. Cluster, Cloud and Grid Computing (CCGRID'15)*, pp.1163–1168.

Miki, N., Ino, F. and Hagihara, K. (2016) 'An extension of OpenACC directives for out-of-core stencil computation with temporal blocking', in *Proc. 3rd Workshop Accelerator Programming using Directives (WACCPD'16)*, pp.36–45.

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

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

NVIDIA Corporation (2016) *NVIDIA Tesla P100* [online] https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (accessed 30 June 2017).

NVIDIA Corporation (2017a) *CUDA C Programming Guide Version 8.0* [online] http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (accessed 30 June 2017).

NVIDIA Corporation (2017b) *PGI Compiler* [online] http://www.pgroup.com/ (accessed 30 June 2017).

OpenACC-Standard.org (2015) *The OpenACC Application Programming Interface, Version 2.5* [online] https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf (accessed 30 June 2017).

OpenMP Architecture Review Board (2015) *OpenMP Application Programming Interface, Version 4.5* [online] http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (accessed 30 June 2017).

Orozco, D. and Gao, G. (2009) *Diamond tiling: A Tiling Framework for Time-Iterated Scientific Applications* [online] http://www.capsl.udel.edu/pub/doc/memos/memo091.pdf (accessed 30 June 2017).

rosecompiler.org (2017) *ROSE Compiler Infrastructure* [online] http://rosecompiler.org/ (accessed 30 June 2017).

Sung, I-J., Liu, G.D. and Hwu, W-M.W. (2012) 'DL: a data layout transformation system for heterogeneous computing', in *Proc. Innovative Parallel Computing (InPar'12)*, 11pp.

Williams, S.W., Waterman, A. and Patterson, D.A. (2009) 'Roofline: an insightful visual performance model for floating-point programs and multicore architectures', *Communications of the ACM*, Vol. 52, No. 4, pp.65–76.

Wu, E., Liu, Y. and Liu, X. (2004) 'An improved study of real-time fluid simulation on GPU', *Computer Animation and Virtual Worlds*, Vol. 15, Nos. 3/4, pp.139–146.

xcalablemp.org (2017) *Xcalablemp* [online] http://www.xcalablemp.org/ (accessed 30 June 2017).

Yabe, T., Xiao, F. and Utsumi, T. (2001) 'The constrained interpolation profile method for multiphase analysis', *J. Computational Physics*, Vol. 169, No. 2, pp.556–593.