International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers

Mohammed Sourouri · Scott B. Baden · Xing Cai

the date of receipt and acceptance should be inserted later

Abstract This paper describes a new compiler framework for heterogeneous 3D stencil computation on GPU clusters. Our framework consists of a simple directive-based programming model and a tightly integrated source-to-source compiler. Annotated with a small number of directives, sequential stencil codes originally written in C can be automatically parallelized for large-scale GPU clusters. The most distinctive feature of the compiler is its capability to generate state-of-the-art hybrid MPI+CUDA+OpenMP code that uses concurrent CPU+GPU computing to unleash the full potential of powerful GPU clusters. At the same time, the auto-generated hybrid codes hide the overhead of various data motion by overlapping them with computation. Test results on the Titan supercomputer and the Wilkes cluster show that auto-translated codes from our compiler can achieve about 90% of the performance of highly optimized handwritten codes, for both a simple stencil benchmark and a real-world application in cardiac modeling. We thus believe that the user-friendliness and performance delivered by our domain-specific compiler framework allow com-

Mohammed Sourouri
Simula Research Laboratory, Norway
Department of Informatics,
University of Oslo, Norway
E-mail: mohamso@simula.no

Scott B. Baden
Dept. of Computer Science and Engineering
University of California, San Diego La Jolla, CA, USA
E-mail: baden@eng.ucsd.edu

Xing Cai
Simula Research Laboratory, Norway
Department of Informatics,
University of Oslo, Norway
E-mail: xingca@simula.no

putational scientists to harness the full power of GPU-accelerated supercomputing without painstaking coding effort.

1 Introduction

Manycore processors such as GPUs and the Xeon Phis possess high levels of compute power per Watt, thus causing large clusters that use these accelerators currently in strong demand. Recently revealed plans for future supercomputers, such as ORNL Summit [27] and ANL Aurora [2], show that future supercomputers will be heterogeneous systems equipped with both general-purpose CPUs and accelerators. Looking ahead, it is also hypothesized by Ang et al. [1] that future Exascale systems will continue to adopt a similar system design.

So far, much attention has been paid to effectively using the accelerators in heterogeneous clusters. In such systems, the CPU's role has mainly been to perform tasks that accelerators are not able to do on their own or cannot perform effectively. However, as CPUs continue to scale with Moore's law, their computational performance and memory bandwidth have reached a level that can no longer be neglected.

Not surprisingly, the latest research has switched to combining CPUs and accelerators for improved performance and energy efficiency [25]. A number of studies have demonstrated the benefit of concurrent CPU+GPU execution, for example in stencil computations [31, 14, 38, 37].

A well-known feature of stencil applications is that the performance is often limited by the memory bandwidth [46]. From a practical point of view, combining CPUs and accelerators means that the memory bandwidth provided by the CPU and the accelerator can be aggregated. The increase of memory bandwidth thus motivates involving the CPUs in addition to the accelerators.

Despite the potential advantages of this strategy, there is, to the best of our knowledge, no programming model or compiler that can reap the benefit of this approach. To address this challenge, we propose Panda, a framework comprising a programming model and a compiler that effectively transforms serial C stencil code for parallel execution on heterogeneous CPU-GPU clusters. Panda uses CUDA and OpenMP to express intra-node parallelism, and MPI to express inter-node parallelism.

Our primary goal is to provide a tool that is easy to use, and thus promotes productivity. To make Panda user-friendly, we have developed a programming model that uses compiler directives to implicitly express parallelism in a sequential code, thus guiding the compiler during the subsequent code translation.

Our secondary goal is to provide a tool that satisfies the performance requirements of both novice and expert users. Frameworks such as OpenACC [29] and OpenMP [30] have demonstrated that achieving high-performance using a generic approach is challenging. Previous domain-specific solutions [42, 5] have outcompeted the generic approach. We have therefore decided to restrict Panda's applicability to 3D stencil computations on structured grids. While we acknowledge that this restriction will limit Panda's outreach, delivering high performance in such a large application space is considered important enough to justify the decision.

This paper makes the following contributions:

- We introduce a programming model that abstracts the complexity of writing parallel code for heterogeneous CPU-GPU clusters. The model consists of a set of compiler directives that implicitly express parallelism in serial C code, allowing the user to focus on the domain science instead of parallelization (Section 2).
- We create a source-to-source compiler that implements the programming model (Section 3).
- We demonstrate the framework's versatility by generating code that targets different cluster scenarios, including pure MPI, MPI+CUDA and **MPI+CUDA+OpenMP** for concurrent CPU+GPU execution on heterogeneous CPU-GPU clusters (Section 3).
- We experimentally evaluate the performance of our framework. Compared with highly optimized handwritten code, we observe a performance realization close to 90% under weak scalability experiments for both a simple stencil benchmark and a real-world application in cardiac modeling (Section 4).

2 The Panda Programming Model

In this section we describe the principal design goals of our framework, and the programming model that adheres to it.

The main goal of our framework is to reduce the complexity of developing large-scale stencil applications by an automated approach. Our target hardware systems are GPU clusters where each node is equipped with one or more GPUs.

We regard directive based programming as a developer-friendly model that requires minimal user programming effort and high level of abstraction [17]. Another benefit of such an approach is backward compatibility. Compilers that do not implement specific directives will simply ignore them. As a result, the user will always have a working code base.

2.1 Target Computations

The fundamental assumption of the Panda framework is that 3D stencil computations are done over logically 3D data arrays. Moreover, triple loop nests are assumed for updating the values of these data arrays, where iterations of such a triple loop nest can be concurrently carried out, thus giving rise to full parallelism. A loop nest can have more than three levels, such as a time loop being the outermost level that has to be carried out in sequence. The

Panda compiler uses static analyses, with support of directives, to automatically identify the parallelism, which is subsequently realized by MPI, CUDA and OpenMP programming.

2.2 Panda Directives

panda distribute(list) size(list)

For performance reasons, Panda supports only flattened arrays that are logically 3D. We can consequently not assume that the iterators of a triple loop nest (such as lines 3-10 in Listing 1) are directly used as array index expressions. This performance-oriented design decision makes reference extraction difficult. Therefore, the **distribute** directive of Panda (such as line 1 in Listing 1) allows the user to annotate all the logically 3D arrays while, more importantly, explicitly marking the variables used to define the length of the arrays through the **size** clause.

```
1  #pragma panda distribute(u_old, u_new) size(Nx+2,Ny+2,Nz+2)
2    for(int t = 0; t < iterations; t++) {
3      for (int k = 1; k < Nz+1; k++)
4        for (int j = 1; j < Ny+1; j++)
5          for (int i = 1; i < Nx+1; i++) {
6            int idx = i + j*(Nx+2) + k*(Nx+2)*(Ny+2);
7            u_new[idx] = kC1 * u_old[idx] + kC0 *
8            (u_old[idx-1] + u_old[idx+1] + u_old[idx-(Nx+2)]
9            + u_old[idx+(Nx+2)] + u_old[idx-(Nx+2)*(Ny+2)]
10           + u_old[idx+(Nx+2)*(Ny+2)]); }
11     #pragma panda wait
12     std::swap(u_old, u_new);
13   }
```

Listing 1 A sample 7-point stencil computation benchmark annotated with Panda directives.

panda boundary(list) size(list)

Panda assumes that a double loop nest is used to enforce the boundary condition on each side of the physical boundary (six possibilities in total). Unlike an automatically detected triple loop nest that traverses the entire 3D volume of an array, Panda relies on the user to insert a special **boundary** directive on top of each double loop nest that updates one side of the physical boundary.

```
  #pragma panda boundary(zmin) size(n,n)
  for (int j=1; j <=n; j++)
    for (int i=1; i <=n; i++)
      int index = i + j * (n+2) * 0 + (n+2)*(n+2);
      E_prev[index] = E_prev[index+2*(n+2)*(n+2)];
```

Listing 2 Computations on the physical boundary that is annotated with the special boundary directive.

The input to the **boundary** directive is a list that consists of the following variables: *xmin*, *xmax*, *ymin*, *ymax*, *zmin* and *zmax*, which represent the three directions in a Cartesian coordinate system. With help of (a subset of) these variables, Panda can detect the applicable spatial direction, and thus auto-generate the correct parallel code in the context of distributed memory. The **size** clause is used both for validation purposes and for deriving the correct indices inside the boundary-condition double loop nest. Listing 2 illustrates the use of the boundary directive.

Currently, the boundary region is limited to boundaries that does not require off-node communication such as periodic boundary conditions. We plan to extend the functionality of the Panda framework to handle physical boundary conditions that require off-node communication in the future.

panda reduction(operator:list)

Many stencil applications need reduction operations, for example, to compute an inner product. Interactions (implicitly) enforced between the threads of a CPU or a GPU are needed to carry out a local reduction. Globally, on distributed memory, a reduction requires additional interaction with MPI, which a novice user may not be aware of. Thus, Panda supports (like OpenMP and OpenACC) a **reduction** directive, which automatically takes care of necessary intra-node and inter-node data exchanges.

panda wait

Code regions that can only be executed sequentially on either the host or the GPU are marked by the **wait** directive. The translated implementation depends on the translator's mode of operation. For example, when generating MPI+CUDA code, the **wait** directive translates into a `cudaDeviceSynchronize` call. When generating MPI+CUDA+OpenMP code, the **wait** directive will result in the insertion of a call to `cudaDeviceSynchronize` plus an OpenMP `#pragma omp master` directive, followed by `#pragma omp barrier`.

3 The Panda Source-to-Source Compiler

The fundamental building blocks of our framework are a programming model (described in the preceding section) and a compiler. In this section we will describe the source-to-source compiler that translates Panda-annotated serial stencil code to parallel and distributed forms.

3.1 Overview

Panda generates three types of parallel code: pure MPI for homogeneous CPU clusters, MPI+CUDA for GPU clusters, and MPI+CUDA+OpenMP for con-

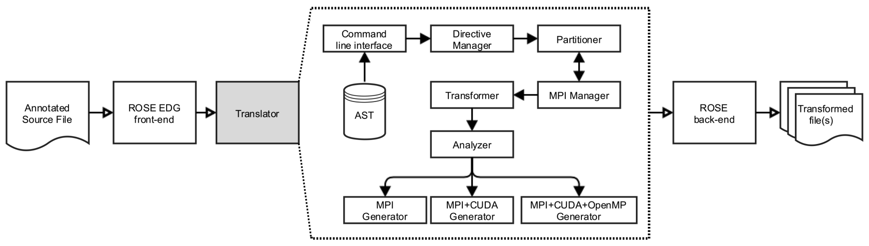


Fig. 1 An architectural view of the Panda source-to-source compiler, which adopts a modular design. Each module may consist of numerous sub-modules, but for brevity, only the most important sub-modules are depicted.

current CPU+GPU execution on GPU clusters. A common trait for these versions is that they gradually extend each other. For example, the MPI+CUDA version is similar to the pure MPI version, but the main difference is that Panda generates CUDA kernels instead of CPU functions (i.e. the CPUs do no computations). In the MPI+CUDA+OpenMP version, both CPU functions annotated with OpenMP directives and CUDA kernels are generated by Panda.

In order to deal with the generation of different code versions, a command line interface (CLI) will parse the options passed to the compiler. A command-line "translation mode" flag determines which modules of the Panda source-to-source compiler are utilized to ensure that correct analyses are performed. Currently, three command-line options are allowed:

– –mpi generates pure MPI (CPU only) code
– –gpu generates MPI+CUDA (GPU-only) code
– –hybrid generates MPI+CUDA+OpenMP (CPU+GPU) code

The user input to our compiler is a serial C source file, annotated with Panda directives. Panda makes use of the EDG front-end bundled with **ROSE** [15] to construct an abstract syntax tree (AST), which expresses the structure of the input code as a graph.

Panda adopts a modular design, and the workflow between the different modules is shown in Figure 1. Once the AST has been generated, the CLI will pass the translation mode to the *Directive Manager* module.

The role of the *Directive Manager* module is twofold: verification and extraction. First, it traverses the AST to verify the correctness of the directives. Assuming that all directives are correctly formulated, the *Directive Manager* will proceed to extract information from them. The extracted information is used to generate local C++ objects that are stored for future access by other modules.

Once the *Directive Manager* has completed its tasks, it will call the *Partitioner* module to decompose the global domain. The default domain partitioning strategy is 3D, meaning that the global domain is partitioned into smaller cuboids. Moreover, in the CPU+GPU mode, each subdomain is partitioned an additional time using 1D decomposition along the z-axis, as described by

Sourouri et al. [38]. In the CPU+GPU mode, the user can dynamically control the partitioning and thus the workload distribution via the command line, because the *Partitioner* will generate command line arguments code that reads user input from `argc` and `argv` at runtime.

Before the domain can be successfully partitioned, the *MPI Manager* module is called to inject the required `<mpi.h>` header and to generate calls to functions such as `MPI_Init`, `MPI_Comm_rank`, `MPI_comm_size` and `MPI_Finalize`. Some of these MPI function calls require the generation of additional variables that the *Partitioner* module depends on. These variables are needed to complete the domain partitioning. Once the domain has been successfully partitioned, the *Transformer* module ensures that for example references to the global domain are substituted with references to local subdomains.

Next, the Panda compiler calls the *Stencil Analyzer*. The task of this module is to reveal important details about the stencil reach, which are needed to generate CPU functions/GPU kernels for halo boundary computations, and corresponding MPI function calls. For example, if the stencil shape reaches beyond 7 points, it is necessary to generate additional CPU functions/GPU kernels for corner accesses. Furthermore, information about the stencil is essential for performing domain-specific code optimizations.

Panda stores array descriptors in a table and uses it to count the number of read-only references to each array. When it has finished tallying the references, Panda subsequently sorts the arrays in the order of most-to-least-frequently accessed. A description of the stencil is then stored as a `Stencil` object, which can be used by other modules for transformation purposes. Stencil description is typically adopted by domain-specific languages (DSLs) [23,47] to deal with this problem. However, while DSLs typically require the user to explicitly define the stencil, the Panda compiler is capable of detecting it automatically, like several existing tools [3,12,42,8].

At this stage the Panda source-to-source compiler has sufficient information about the stencil to perform the necessary transformation, which is spearheaded by the respective *Generator* modules.

3.2 MPI Code Generation

Although the *Partitioner* module breaks the global domain into smaller cuboids, it does not generate the MPI function calls necessary for inter-node communication. This responsibility is delegated to the *MPI Manager* by the *Transformer* module.

The main objective of *MPI Manager* is to generate non-blocking asynchronous MPI calls to realize inter-node communication that overlaps halo boundary exchange with computation. However, before the exchange takes place, the respective boundaries must be computed and stored in dedicated send buffers (*packing*). The send buffers are then passed to the `MPI_Isend` function that communicates the content of the send buffer to a receiving neighbor. Data received by a neighboring subdomain is stored in a receive buffer before it is *unpacked*. Additionally, an `MPI_Waitall` is also inserted to ensure that associated MPI requests have completed before the unpacking starts.

3.3 Communication Optimizations

The Panda compiler performs two communication optimizations in order to improve the application performance.

1. All data movement between a host CPU and its device GPU is performed by the `cudaMemcpyAsync` function, which guarantees that the intra-node data movement between the CPU and the GPU happens in the background, thus having the possibility of being overlapped with computation.
2. In the context of MPI+CUDA+OpenMP code generation, Panda creates separate MPI requests for the CPU and the GPU that are used by the designated MPI function calls. By introducing separate MPI requests, we decouple CPU and GPU MPI requests from each other, thus in effect creating two independent communication channels. The benefit of this approach is that the GPU does not need to wait on the CPU's messages to arrive (or vice versa) before it can start unpacking its received data.

3.4 MPI+CUDA Code Generation

For computation of the interior points, Panda generates CUDA kernels based on the pipelined wavefront technique [35], but does not perform register blocking nor loop unrolling. This implementation decision is for simplifying the actual code generation. However, in future work we will investigate auto tuning of cache and register blocking and other optimizations, such as loop unrolling for CPUs [45,28] and warp specalization for GPUs [22]. Listing 3 displays the generated CUDA kernel for computing the interior points.

```
__global__ void ComputeInteriorPoints(
  double *__restrict__ const u_old,
  double *u_new, int nsdx, int nsdy, int nsdz, double kC0,
  double kC1, int offset) {
    unsigned int i = 1+threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int j = 1+threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int k_start = 1+blockIdx.z * offset;
    unsigned int k_stop = k_start + offset;

    if (k_stop > (nsdz+2)−1) { k_stop = (nsdz+2)−2; }

    if (i > 1 && i < (nsdx+2)−2 && j > 1 && j < (nsdy+2)−2)
      for (int k = k_start; k < k_stop; k++) {
        int idx = i + j*(nsdx+2) + k*(nsdx+2)*(nsdy+2);
        u_new[idx] = kC1 * u_old[idx]
        + (kC0 * u_old[idx−1] + u_old[idx+1]
        + u_old[idx−(nsdx+2)] + u_old[idx+(nsdx+2)]
        + u_old[idx−(nsdx+2)*(nsdy+2)]
        + u_old[idx+(nsdx+2)*(nsdy+2)]);
}}
```

Listing 3 Auto-generated CUDA kernel for computing interior points using a 7-point stencil. The code has been formatted for brevity.

One important assumption of Panda is that all stencil-compute loops (i.e. triple loop nests) require inter-node MPI communication. Since we wish to overlap communication with computation, the inter-subdomain halo boundaries are computed separately from the interior points for every identified stencil-compute loop nest. Panda thus generates unique halo boundary functions for every stencil-compute loop nest.

When generating the kernels for computing the different halo boundaries, Panda assumes that a subdomain is a box and thus has six sides (up to six MPI neighbors). Specifically, Panda first enumerates the six different sides, and then iterates over them using the stencil analysis process described in Section 3.1. At the same time, Panda is able to distinguish stencil-compute loops from non-stencil-compute loops.

Each halo boundary, which is a 2D plane, is handled by a double loop nest. The Panda compiler simplifies this part of CPU code generation by performing a deep copy of the original triple loop nest, while removing one loop layer that is not needed for a specific halo boundary. The benefit of such a deep copy technique is that we automatically obtain the loop range (condition statement) of the for-loops.

It is straightforward to accommodate the deep copied for-loops when generating CUDA kernels, by simply modifying the for-loops to iterate over the respective mesh points that are assigned to one CUDA thread. This technique is better known as grid-stride loops [21]. As Listing 4 shows, the generated halo boundary loop nest in a CUDA kernel is very similar to a regular CPU double loop nest.

```
__global__ void ComputePackEast(
  double* u_new, double* __restrict__ const u_old,
  double* d_send_buffer, int nsdx, int nsdy, int nsdz,
  double kC0, double kC1) {

  int z = 1+threadIdx.y + blockIdx.y * blockDim.y;
  int y = 1+threadIdx.x + blockIdx.x * blockDim.x;

  for (int k = z; k < (nsdz+2)−1; k += blockDim.y * gridDim.y)
    for (int j = y; j < (nsdy+2)−1); j += blockDim.x * gridDim.x)
        int idx = (nsdx) + j*(nsdx+2) + k*(nsdx+2)*(nsdy+2);
        int idx2d = (k−1) * nsdy + j − 1;

        u_new[idx] = kC1 * u_old[idx]
        + (kC0 * u_old[idx−1] + u_old[idx+1]
        + u_old[idx−(nsdx+2)] + u_old[idx+(nsdx+2)]
        + u_old[idx−(nsdx+2)*(nsdy+2)]
        + u_old[idx+(nsdx+2)*(nsdy+2)]);

        d_send_buffer[idx2d] = u_new_sd[idx];
}
```

Listing 4 Auto-generated CUDA kernel for computing a halo boundary (the xy-plane in the "east"). The code has been formatted for brevity.

The Panda compiler takes advantage of the Kepler architecture's read-only cache [26] to further improve the performance. The read-only cache is a 48 kB on-chip memory that can be used to cache data that is known to be read-only during the lifetime of a kernel. Data to be placed in the read-only cache must be flagged with the `const` and `__restrict__` keywords. Thus, upon CUDA kernel generation, the *MPI+CUDA Generator* module will use information from the `Stencil` object to identify read-only arrays. These arrays are then automatically flagged with the `const` and `__restrict__` keywords.

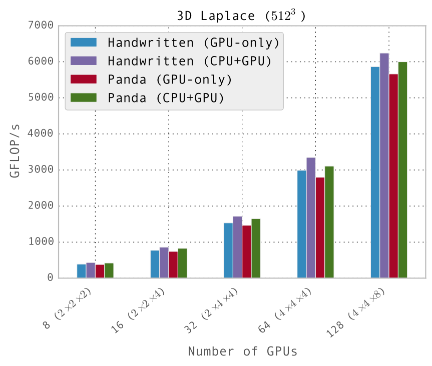
Choosing a good CUDA thread block size might impact the performance of a kernel. Our solution is to generate three variables `block_x`, `block_y` and `block_z`, one per dimension. Each variable (having a default value) is then connected to the command-line interface, allowing the user to experiment with different block configurations at runtime (as opposed to compile time). However, auto-tuning to determine optimal block sizes remains as future work.

3.5 MPI+CUDA+OpenMP Code Generation

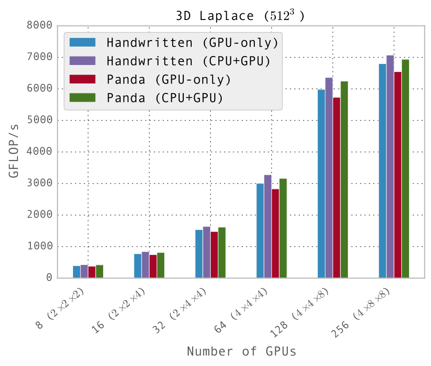
So far, much of the attention in accelerator-based computing has been put on the accelerators, while the CPU's role has mostly been serving as a host for the accelerator. However, as CPUs have gradually become more powerful, researchers have started to study how CPUs and GPUs can be cleverly combined for further performance gains. In our scenario, the trick is to properly divide the computational workload between the CPU and the GPU, so that the CPU can aid the GPU in sharing the computational costs.

One programming strategy is based on the "nested" implementation strategy, as described by Sourouri et al. [38], where a hybrid MPI+CUDA+OpenMP programming model is used to realize concurrent CPU+GPU computations. The principal idea for the strategy is to overlap computation with communication using OpenMP's nested parallelism capability to generate two independent groups of threads. The first thread group handles the CUDA, MPI communication and computation of the halo boundary points on the CPU using OpenMP threads. The second thread group computes the interior points on the CPU.

Generating MPI+CUDA+OpenMP code requires only incremental changes to the MPI+CUDA code. The main difference is that the generated **MPI+CUDA** code is augmented with additional CPU code annotated with OpenMP directives. Moreover, an additional 1D subdomain partitioning along the z axis is applied to divide the computational workload between the CPU and the GPU in every subdomain. Code generation is realized in three passes. First, Panda generates pure MPI code for performing communication and com-

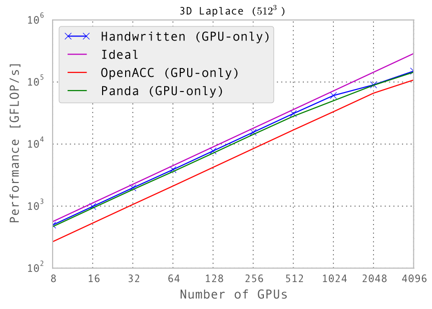


(a) Weak scaling on Wilkes using one GPU per node

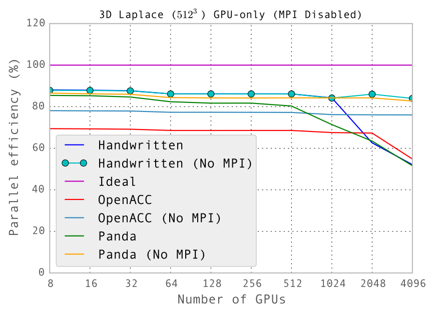


(b) Weak scaling using two GPUs per node

Fig. 2 Weak scaling results on the Wilkes cluster, each MPI process is responsible for 512^3 mesh points.



(a) MPI+CUDA weak scaling on Titan



(b) MPI+CUDA weak scaling on Titan with MPI disabled

Fig. 3 Weak scaling results on the Titan supercomputer, each MPI process is responsible for 512^3 mesh points.

putation on the CPU. Next, MPI+CUDA code is generated, and in the final pass the two codes are stitched together.

As a number of studies [43,13,6,38,14] have already shown, one of the most challenging aspect of CPU+GPU codes is related to assigning work to the different processing units of a heterogeneous node. Because CPU+GPU codes are extremely sensitive to the workload ratio, Panda's CLI will auto-generate command-line arguments at the start of the translated code so that the user can specify the CPUs workload ratio.

4 Experimental Results

This section investigates the performance of the GPU-only and CPU+GPU code generated by Panda. The two auto-generated code versions are compared against the corresponding handwritten implementations for two cases of stencil

	Titan (Cray XK7)	Wilkes
CPU	Opteron 6274	Xeon E5-2630v2
Clock frequency	2.2 GHz	2.6 GHz
# cores	16	6
# sockets	1	2
L3$ per chip	16 MB	15 MB
Theoretical DP	142 GFLOP/s	249.6 GFLOP/s
Theoretical BW	70.4 GB/s	119.4 GB/s
STREAM	31.9 GB/s	72.95 GB/s
Compiler	cce 8.1.0.144	icc 15.0.5.223
Accelerator	Tesla K20X	Tesla K20c
# GPUs per node	1	2
Theoretical DP	1310 GFLOP/s	1170 GFLOP/s
Theoretical BW	250 GB/s	208 GB/s
STREAM	180 GB/s	151 GB/s
Compiler	nvcc 6.5	nvcc 6.5

Table 1 Experimental platform overview.

computation: the well-known 7-point 3D Laplacian stencil benchmark and a real-world 3D application in cardiac modeling.

Two hardware platforms have been used for our study. The Wilkes cluster at University of Cambridge is the former No. 2 system on the Green500 list [40], and consists of 128 compute nodes. Each Wilkes node is equipped with two 6-core Intel Xeon E5-2630v2 "Ivy Bridge" CPUs and two Tesla K20c GPUs. The second platform is the Titan supercomputer, currently ranked the second fastest supercomputer on the TOP500 list [41]. Each Titan node is equipped with a single 16-core AMD Opteron 6274 CPU and a Tesla K20X GPU. A complete overview of the two GPU clusters are detailed in Table 1. Under the weak scaling experiments, the problem size for each MPI process was fixed at 512^3 for both the 3D Laplacian stencil benchmark and the Cardiac elctrophysiology simulator. The global problem size for the strong scale experiment was $512 \times 512 \times 1024$. All experiments were conducted with double precision.

4.1 3D Laplacian Stencil Benchmark

As the first numerical case, let us consider the simplest diffusion equation, $\partial u/\partial t = \nabla^2 u$, to be discretized by finite differences combined with explicit time stepping. The resulting 3D numerical scheme straightforwardly computes a new time level of u by applying a standard 7-point stencil over the previous time level of u. That is, the computation involved in each time step is the same as the well-known 7-point 3D Laplacian stencil, as shown in Listing 1. Moreover, this simple benchmark application assumes that u remains constant on the entire physical boundary. Hence, during the whole time-stepping procedure, no computation is needed on any of the physical boundary points.

For this 3D benchmark, both of our handwritten implementations use a highly optimized single-GPU kernel that can realize 78% of the realistic memory bandwidth on a K20 GPU, which is measured by the STREAM Triad memory benchmark [24]. More precisely, the handwritten CUDA kernel is based on the technique presented by Su et al. [39], which combines plane sweeping along the z axis with chunking along the y axis. In comparison, the Panda auto-generated GPU kernel can achieve about 72% of the realistic memory bandwidth. The performance difference is due to the fact that the handwritten GPU kernel is more aggressively optimized with register blocking along the z dimension, which is not adopted automatically by the Panda compiler.

On the Wilkes cluster, which has more powerful CPUs than those on the Titan cluster, we compare the auto-generated CPU+GPU (i.e., **MPI+CUDA+OpenMP**) code with the handwritten CPU+GPU counterpart, as well as a comparison between the two GPU-only versions. If only one GPU is used per Wilkes node, the achievable GPU memory bandwidth is about 2× that of the aggregate CPU memory bandwidth. Figure 2(a) displays the measured performance of the four implementations (two handwritten versions versus two Panda auto-generated versions), in the context of using one GPU per node on the Wilkes cluster. The most efficient implementation is the handwritten MPI+CUDA+OpenMP code, followed by the auto-generated MPI+CUDA+OpenMP code. The best CPU workload ratios for the two CPU+GPU codes are 15% for the handwritten version and 8% for the auto-generated version, respectively. This difference in the CPU workload ratio is primarily because that the handwritten version performs a highly efficient 3D cache-blocking [32] technique for computing the interior points on the CPU, and uses non-temporals for computing the halo boundary points. The auto-generated code does not implement these optimizations, which implicitly means that the CPU workload must be smaller. Nevertheless, the auto-generated CPU+GPU code is still capable of outperforming the highly optimized handwritten GPU-only implementation.

Using both GPUs per Wilkes node brings a new challenge, because the number of MPI processes is now two per node (one per CPU socket), effectively reducing the number of CPU cores available per GPU from 12 to 6. This in turns widens the memory bandwidth difference to a factor of 4×, between one GPU and one CPU socket. Consequently, we reduce the CPU's workload ratio from 15% to 10% for the handwritten version, and from 8% to 5% for the auto-generated version. Despite the CPU workload reduction, it is evident from Figure 2(b) that the auto-generated CPU+GPU code is still faster than the hand optimized GPU-only version, in the context of using two GPUs per Wilkes node.

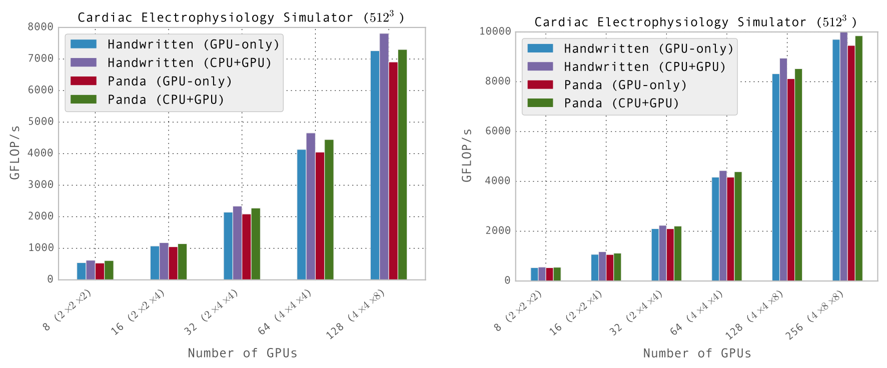
Moving to the Titan platform, Figure 3(a) shows the performance of three GPU-only implementations. That is, in addition to the handwritten version and the Panda auto-generated version, we also adopt a highly optimized OpenACC kernel, which has been kindly reviewed and improved by NVIDIA. The OpenACC implementation makes use of CUDA-aware MPI (not used by the other two implementations), and thus achieves slightly better communication performance on Titan. Despite this advantage of OpenACC, the GPU-only code generated by Panda is able to beat the OpenACC implementation. This is because the Panda translator is domain-specific, thus able to leverage the knowledge of the domain of stencil computations to generate more optimized kernels. The performance of the OpenACC code is largely determined by the generic approach taken by OpenACC, which divides the loop nest into smaller thread blocks, and then executes each thread block in a SIMD fashion on each GPU. Another performance weakness of the OpenACC code arises from a very high register usage that limits the occupancy, thus impeding the performance.

On Titan, the Panda auto-generated code realizes nearly 90% of the performance of the handwritten counterpart. The primary reason why the auto-generated code cannot realize the full performance of the handwritten code is largely because of less efficient compute kernels including the kernels responsible for computing the halo boundary points. Furthermore, the auto-generated code performs unnecessary packing/unpacking of halo boundary data that is actually laid out contiguously in memory.

In Figure 3(b) we have repeated the same weak-scaling study outlined in Figure 3(a), but the MPI calls now are disabled. In other words, there is no inter-node communication overhead. The purpose is to quantify the amount of time spent on communicating, and thereby reveal how well the code is able to hide inter-node communication. As Figure 3(b) shows, the handwritten code does a good job of hiding the MPI communication. It is only when the number of GPUs exceeds 1024 that the MPI communication becomes a decisive bottleneck. The difference between the performance results without inter-node communication and the performance results with communication can help to quantify the impact of inter-node communication. For example, at 2048 GPUs, 23% of the total time of the handwritten code is spent on MPI communication, while 33% is spent on MPI communication when 4096 GPUs are used. Similarly for Panda, MPI communication is well hidden up to 512 GPUs. After 512 GPUs, communication becomes a more pressing issue affecting scalability. At 1024 GPUs, 10% of the time is spent on MPI, 21% at 2048 GPUs, and finally at 4096 GPUs, 31% is spent on communication.

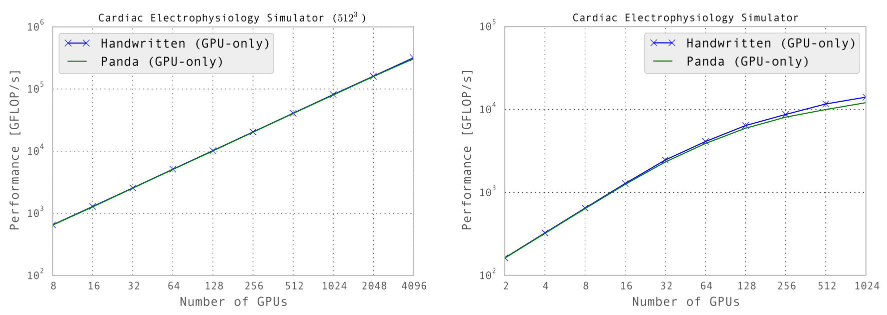
The reason that we only present the GPU-only performance measurements on Titan is that CPU+GPU versions were unsuccessful on Titan. Recall that each Titan node is equipped with a single 16-core AMD Opteron 6274 CPU and a Tesla K20X GPU. The performance difference between the GPU and the CPU, by comparing the realistic memory bandwidth performance, is approximately 5.6×. Closing this performance gap is challenging, especially since the 16 CPU cores share 8 floating point units. Thus, it is not possible to delegate enough threads to the two thread groups responsible for computing the halo boundary and interior points.

The lesson learned from clusters such as Titan is that CPU+GPU codes do not pay off, if the performance gap between the CPU and the GPU is too big. In such a scenario, GPU-only code might be a better alternative. Luckily, Panda is capable of generating both GPU-only and CPU+GPU code. Hence, the user can freely choose the best option that suits a given hardware platform.



(a) Weak scaling on Wilkes using one GPU per node (b) Weak scaling on Wilkes using two GPUs per node

Fig. 4 Weak scaling results on the Wilkes cluster, each MPI process is responsible for 512^3 mesh points.



(a) MPI+CUDA weak scaling on Titan (b) MPI+CUDA strong scaling on Titan

Fig. 5 Weak and strong scaling results on the Titan supercomputer. Under the weak scaling experiment, each MPI process is responsible for 512^3 mesh points, while under the strong scaling experiment, the global problem size is $512 \times 512 \times 1024$.

4.2 Cardiac Electrophysiology Simulator

We have also applied Panda to a real-world 3D cardiac electrophysiology simulator, which simulates the propagation of electrical signals in the cardiac tissue. The purpose of such a simulator is to study complicated cardiac features, such as spiral waves, which may lead to life threatening situations such as ventricular fibrillation.

The mathematical model of concern was derived by Aliev and Panfilov [10]. Without going into details, it suffices to mention that the model consists of a 3D reaction-diffusion equation, coupled with a two-state ordinary differential equation (ODE) system per spatial mesh point. In comparison with the preceding 7-point Laplacian stencil benchmark, the cardiac simulator has additionally implemented an ODE solver, as well as enforcing a homogeneous Neumann condition on the entire physical boundary.

The input serial code of the cardiac simulator to Panda is annotated similarly to Listing 1, with the addition of the **panda boundary** directive in order to deal with the Neumann boundary condition, as outlined in Section 2.2. For comparison, we have also implemented two handwritten versions: GPU-only and CPU+GPU.

Figure 4(a) and Figure 4(b) show the performance results of the cardiac simulator on the Wilkes cluster using both handwritten and auto-generated CPU+GPU and GPU-only implementations. Like the 7-point Laplacian stencil benchmark, the most efficient implementations for the cardiac simulator are the two that involve concurrent CPU+GPU computations.

The performance upper-hand of the handwritten code comes largely from the faster kernels for the halo boundaries, and for computing the PDE and the ODE parts. The computations involve many coefficients, which easily cap the occupancy due to high register usage. The handwritten code makes use of the GPU's constant memory for this purpose. Moreover, it also uses plane sweeping and loop unrolling to achieve high performance. These optimization techniques are not exploited in the kernels generated by Panda.

The performance difference between the handwritten GPU-only code and the Panda GPU-only version is more visible under the strong scaling experiments conducted using up to 1024 GPUs on Titan, as shown in Figure 5(b). We observe that already at 32 GPUs the two performance curves start to diverge. Profiling reveals that there are two reasons for this behavior. The first reason is that Panda does unnecessary halo boundary packing and unpacking on the xy planes. This is avoided by the handwritten code. The second reason is that the Panda generated kernels for the halo boundaries and the interior points are not as fast as the handwritten kernels, which may constitute a bottleneck under strong scaling experiments when each subdomain becomes very small. One side effect of the generalizations that Panda makes is the introduction of additional overheads. However, we believe that these overheads are modest enough that we do not see them as the main obstacle to scalability.

5 Related Work

The number of prior works conducted by other researchers is large. To help the reader, we will categorize the related work into three types: compiler directives, libraries and DSLs.

Compiler directives

A developer friendly approach is to use compiler hints to guide the compiler in generating parallelized code. Thanks to the support from numerous vendors, OpenACC and OpenMP have rapidly established themselves as the *de facto* solutions for directive-based code development. Although capable of delivering acceptable performance [44,19] in a broad range of applications, neither OpenACC nor OpenMP targets an entire cluster. Users are thus left to their own to write code that deals with MPI.

Our work is closely related to [29,30,16,42], which all use compiler directives to automatically offload computation to a single accelerator. OpenACC [29] is known to provide good performance on Nvidia GPUs, while OpenMP [30] is known to deliver particularly good performance on CPUs and Xeon Phi co-processors. Mint [42] by Unat et al. is a domain-specific translator for stencil methods by transforming serial stencil C/C++ code to CUDA code. OpenMPC [16] by Lee and Eigenmann provides an extension of OpenMP, so that code annotated with OpenMP directives is translated to CUDA code. OpenMPC also includes an auto-tuner for performance tuning. Like OpenACC and OpenMP, both Mint and OpenMPC target only a single accelerator.

OpenMP-D [4] by Basumallik and Eigenmann provides a set of custom directives that extends OpenMP for translating OpenMP code to MPI code. Similar to OpenMPC, OpenMP-D takes a generic approach, and is not restricted to stencil computations. Dathathri et al. [6] have developed a compiler for auto-generation of regular computation on structured grid for heterogeneous CPU-GPU clusters. OpenCL is chosen as the programming model to generate code for both CPUs and GPUs. The compiler by Dathathri et al. can also generate CPU+GPU code using an asymmetric work distribution similar to ours. The authors however are not able to make their CPU+GPU code scale beyond a single node, believing that the CPU is the bottleneck. Ravishankar et al. [33] have developed a compiler framework of code generation for mixed irregular/regular computations targeting homogeneous distributed memory systems. Our Panda compiler framework shares several similarities with the work by Ravishankar et al., such as static analyses of partitionable loops, use of compiler directives to annotate distributed data structures, etc. However, one important distinction is that our tool is capable of targeting GPU-enhanced clusters in addition to homogeneous CPU clusters.

Libraries

PARTANS [20] by Lutz et al. provides a C++ template library to ease the burden of OpenCL programming of stencil code targeting multiple GPUs. The authors have also developed an extensive auto-tuner for performance optimizations. PARTANS supports multiple GPUs per node, but its scalability is limited because the library only supports 1D domain decomposition. Shimokawabe et al. [36] have developed a C++ library for performing large-scale weather forecast simulations on the TSUBAME 2.5 supercomputer. The library of Shimokawabe et al. supports various domain decompositions, and also takes advantage of multiple GPUs on the same node using GPUDirect v2 (peer-to-peer) memcopies for fast intra-node data transfers. Both PARTANS and the framework of Shimokawabe et al., however, lack the ability to perform pure CPU or concurrent CPU+GPU computations. Furthermore, users with sequential implementations must rewrite their code in order to take advantage of these libraries.

DSLs

DSLs constitute a compromise by giving up some of the language generality for performance. Since a DSL is restricted to a particular application domain, it can leverage on this knowledge to deliver excellent performance. Contrary to a directive-based approach, DSLs require considerable effort in code development. A similar investment in code *redevelopment* is also required if the user has an existing parallel implementation.

The DSLs that lie quite close to Panda are [5,11,31,23,13,47,9]. PATUS [5] is a CPU-GPU stencil code generation and auto-tuning framework developed by Christensen et al. PATUS depends on user-provided description files, because it lacks a stencil analyzer that can automatically recognize stencil shapes. Code generation for different architectures is explicitly defined in a machine architecture description file. Holewinski et al. [11] have developed a single-GPU stencil code generator using overlapped tiles in OpenCL. Neither PATUS nor the work by Holewinski et al. generates code for concurrent CPU+GPU execution.

The Halide [31] DSL represents a compiler and auto-tuner framework by Ragan-Kelley et al. It generates stencil code for 2D image processing on CPUs, GPUs, and CPUs+GPUs. Like PATUS and the framework developed by Holewinski et al., Halide targets CPUs and manycore processors within a single node.

Physis [23] by Maruyama et al. is an embedded DSL that targets large-scale GPU clusters. A dedicated compiler translates input code that is implemented in the Physis DSL into MPI+CUDA code, which overlaps inter-node data transfers with computation. However, Physis cannot generate heterogeneous CPU+GPU code. The SnuCL [13] framework by Kim et al. can run a wide range of OpenCL applications on GPU clusters. SnuCL abstracts the processing units, such as CPUs and GPUs, across an entire cluster to make it appear as a single processing unit on a single machine. Applications transformed by SnuCL are capable of concurrent CPU+GPU computations, but due to the workload distribution strategy adopted by SnuCL, the performance benefit of this approach is limited. Another limitation is that SnuCL does not take serial code as input, only parallel OpenCL code. The auto-generation and auto-tuning stencil framework [47] by Zhang and Mueller generates high-quality stencil code that can be executed on GPU clusters. The framework however cannot generate pure MPI or CPU+GPU code nor can it handle physical boundary conditions.

STELLA [9] is a recent DSL/library that targets atmospheric stencil codes discretized on structured grids. Like the library developed by Shimokawabe et al. [36], STELLA is particularly optimized for a specific weather prediction and regional climate model called COSMO. Similar to Panda, STELLA is able to handle physical boundary conditions, but unlike Panda, it is not able to produce codes that can perform concurrent CPU+GPU computations.

In summary, the related work reveals the lack of a developer-friendly programming model that can realize high performance on accelerated clusters by auto-generating CPU+GPU code based on serial input code written in a general-purpose programming language such as C. This gap in the compiler toolchain represents a particular obstacle to domain scientists who wish to harness the computational powers of CPU-GPU clusters. The Panda framework is thus an effort to close the gap.

6 Limitations

Panda is a domain-specific compiler that targets 3D stencil computations on regular grids. While some might see domain-specific translation as a restriction, we see the opportunity of carrying out meaningful optimizations that would not be possible in a more generic approach.

At the moment of writing, Panda is not equipped with a dedicated runtime system that can automatically detect the number of CPU cores available on a target system. This means that the default numbers of OpenMP threads needed in the context of MPI+CUDA+OpenMP code generation must be defined as command-line arguments. The lack of such a runtime system makes it difficult for the user to know exactly how many OpenMP threads should be dedicated to the two thread groups. As a rule of the thumb, we recommend that $\frac{2}{3}$ of the spawned OpenMP threads are dedicated to computing the interior points, while the remaining $\frac{1}{3}$ are dedicated to computation of the boundary points.

Finally, to further improve the performance of the CPU+GPU code produced by Panda, it is necessary to generate more optimized CPU code for computations of the interior and boundary points. High-performance CPU code is an important ingredient in CPU+GPU implementations to reduce the computational performance gap between the CPU and the GPU. As numerous works have already shown [34,7,18], techniques such as cache blocking are effective to optimize stencil codes on CPUs.

Other features currently not handled in Panda include I/O and checkpointing, which remain as future work. Users with serial applications that rely on these features must manually modify the generated code.

7 Conclusion

In this paper we have presented the Panda compiler framework, consisting of a directive-based programming model and a source-to-source translator. From annotated serial C code, Panda can automatically generate various forms of parallel code that can efficiently run on GPU-accelerated distributed-memory systems.

We have demonstrated that the MPI-supported GPU-only code generated by Panda can realize 90% of the performance of a highly optimized handwritten counterpart. Moreover, Panda's GPU-only code scales nicely on more than 4000 GPUs on the Titan supercomputer.

With respect to concurrent CPU+GPU computation, coding is notoriously hard due to many fine-grained details. The Panda framework fills the missing gap in automated generation of hybrid MPI+CUDA+OpenMP code for stencil computations. The automatically generated CPU+GPU code from Panda can in many cases outperform handwritten GPU-only code. We thus believe that Panda can satisfy the performance requirements of many domain scientists, so that they can focus on the science instead of tedious programming details. At the same time, Panda generates code with high readability, so advanced users can use Panda as a springboard to quickly generate parallel and hybrid code that can later be manually modified for further performance enhancements.

Future work will mainly address some of Panda's current limitations, such as handling stencils with a wider reach than 7 points. Another topic is periodic physical boundary condition, which in the context of MPI parallelization requires implementing wrap-around communication. We also plan to construct a runtime system to provide a better user experience with respect to adjusting input, such as the CPU workload ratio, to the translated application.

We will also explore better support for future GPU clusters that are equipped with multiple GPUs per node. In the current version of Panda, an MPI process is spawned per GPU. However, a more promising approach is to use only a single MPI process, but adopting multiple CPU threads to control the GPUs.

Currently, the Panda source-to-source compiler is specifically designed for GPU clusters, but we will consider extending Panda with respect to Xeon Phi clusters. Such an extension will involve fine-grained use of OpenMP on Xeon Phis as opposed to using CUDA on GPUs. Our preliminary study suggests that the extension can be implemented in a straightforward manner.

Acknowledgements

This work was supported by the FriNatek program of the Research Council of Norway, through grant No. 214113/F20. The authors thank High Performance Computing Service at the University of Cambridge, UK. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. Ang, J., Barrett, R., Benner, R., Burke, D., Chan, C., Cook, J., Donofrio, D., Hammond, S., Hemmert, K., Kelly, S., Le, H., Leung, V., Resnick, D., Rodrigues, A., Shalf, J., Stark, D., Unat, D., Wright, N.: Abstract machine models and proxy architectures for exascale computing. In: Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing (Co-HPC), pp. 25–32 (2014)
2. Argonne Leadership Computing Facility: Aurora. http://aurora.alcf.anl.gov/ (2015). [Online; accessed 1-June-2015]
3. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA code generation for affine programs. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, pp. 244–263 (2010)
4. Basumallik, A., Eigenmann, R.: Towards automatic translation of OpenMP to MPI. In: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 189–198 (2005)

5. Christen, M., Schenk, O., Burkhart, B.: PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, pp. 676–687 (2011)
6. Dathathri, R., Reddy, C., Ramashekar, T., Bondhugula, U.: Generating efficient data movement code for heterogeneous architectures with distributed-memory. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, pp. 375–386 (2013)
7. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 361–366 (2005)
8. Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., Verdoolaege, S.: Hybrid hexagonal/classical tiling for GPUs. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 66:66–66:75 (2014)
9. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: STELLA: A domain-specific tool for structured grid methods in weather and climate models. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 41:1–41:12 (2015)
10. Hanslien, M., Artebrant, R., Tveito, A., Lines, G.T., Cai, X.: Stability of two time-integrators for the Aliev-Panfilov system. International Journal of Numerical Analysis and Modeling **8**, 427–442 (2011)
11. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: Proceedings of the 26th ACM International Conference on Supercomputing, pp. 311–320 (2012)
12. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An auto-tuning framework for parallel multicore stencil computations. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1–12 (2010)
13. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing, pp. 341–352 (2012)
14. Langguth, J., Sourouri, M., Lines, G.T., Baden, S.B., Cai, X.: Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes. Micro, IEEE **35**(4), 6–15 (2015)
15. Lawrence Livermore National Laboratory: ROSE compiler infrastructure. http://rosecompiler.org (2015). [Online; accessed 04-June-2015]
16. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2010)
17. Lee, S., Vetter, J.S.: Early evaluation of directive-based GPU programming models for productive exascale computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 23:1–23:11 (2012)
18. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pp. 451–460 (2010)
19. Levesque, J.M., Sankaran, R., Grout, R.: Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 15:1–15:11 (2012)
20. Lutz, T., Fensch, C., Cole, M.: PARTANS: An autotuning framework for stencil computation on multi-GPU systems. ACM Trans. Archit. Code Optim. **9**(4), 59:1–59:24 (2013)
21. Mark Harris: CUDA pro tip: Write flexible kernels with grid-stride loops. http://goo.gl/b8Vmkh (2015). [Online; accessed 12-November-2015]
22. Maruyama, N., Aoki, T.: Optimizing stencil computations for NVIDIA Kepler GPUs. In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations, pp. 89–95 (2014)

23. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 11:1–11:12 (2011)
24. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (1995)
25. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. **47**(4) (2015)
26. NVIDIA: NVIDIA's next generation CUDA compute architecture: Kepler GK110. http://goo.gl/9ju84x (2013). [Online; accessed 12-November-2015]
27. Oak Ridge Leadership Computing Facility: Summit. https://olcf.ornl.gov/summit/ (2015). [Online; accessed 29-May-2015]
28. Olschanowsky, C., Strout, M.M., Guzik, S., Loffeld, J., Hittinger, J.: A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 793–804 (2014)
29. OpenACC - Directives for Accelerators: The OpenACC Application Program Interface. http://openacc-standard.org (2015). [Online; accessed 23-May-2015]
30. OpenMP Architecture Review Board: OpenMP Application Program Interface. http://openmp.org (2015). [Online; accessed 23-May-2015]
31. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 519–530 (2013)
32. Rahman, S.M.F., Yi, Q., Qasem, A.: Understanding stencil code performance on multicore architectures. In: Proceedings of the 8th ACM International Conference on Computing Frontiers, pp. 30:1–30:10 (2011)
33. Ravishankar, M., Dathathri, R., Elango, V., Pouchet, L.N., Ramanujam, J., Rountev, A., Sadayappan, P.: Distributed memory code generation for mixed irregular/regular computations. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 65–75 (2015)
34. Rivera, G., Tseng, C.W.: Tiling optimizations for 3D scientific computations. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (2000)
35. Schäfer, A., Fey, D.: High performance stencil code algorithms for GPGPUs. In: Proceedings of 2011 International Conference on Computational Sciences (ICCS), vol. 4, pp. 2027–2036 (2011)
36. Shimokawabe, T., Aoki, T., Onodera, N.: High-productivity framework on GPU-rich supercomputers for operational weather prediction code ASUCA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 251–261 (2014)
37. Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., Matsuoka, S.: Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 3:1–3:11 (2011)
38. Sourouri, M., Langguth, J., Spiga, F., Baden, S.B., Cai, X.: CPU+GPU programming of stencil computations for resource-efficient use of GPU clusters. In: Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on, pp. 17–26 (2015)
39. Su, H., Wu, N., Wen, M., Zhang, C., Cai, X.: On the GPU performance of 3D stencil computations implemented in OpenCL. In: Proceedings of the 28th International Supercomputing Conference, vol. 7905, pp. 125–135 (2013)
40. Top500.org: June 2015 — the green500 list. http://www.green500.org/lists/green201506 (2015). [Online; accessed 04-Sept-2015]
41. Top500.org: November 2015 — top500 supercomputer sites. http://top500.org/lists/2015/11/ (2015). [Online; accessed 18-Nov-2015]

42. Unat, D., Cai, X., Baden, S.B.: Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of the International Conference on Supercomputing, pp. 214–224 (2011)
43. Venkatasubramanian, S., Vuduc, R.W.: Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In: Proceedings of the 23rd International Conference on Supercomputing, pp. 244–255 (2009)
44. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC - First Experiences with Real-World Applications. In: Euro-Par 2012 Parallel Processing - 18th International Conference, vol. 7484, pp. 859–870 (2012)
45. Williams, S., Kalamkar, D.D., Singh, A., Deshpande, A.M., Van Straalen, B., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., Oliker, L.: Optimization of geometric multigrid for emerging multi- and manycore processors. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 96:1–96:11 (2012)
46. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
47. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 155–164 (2012)