Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers

Naoya Maruyama
Tokyo Institute of Technology
JST, CREST
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan
naoya@matsulab.is.titech.ac.jp

Tatsuo Nomura *
Google, Inc.
6-10-1 Roppongi, Minato-ku,
Tokyo, Japan
tatsuo@google.com

Kento Sato
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan
kent@matsulab.is.titech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology
JST, CREST & NII
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan
matsu@is.titech.ac.jp

ABSTRACT

This paper proposes a compiler-based programming framework that automatically translates user-written structured grid code into scalable parallel implementation code for GPU-equipped clusters. To enable such automatic translations, we design a small set of declarative constructs that allow the user to express stencil computations in a portable and implicitly parallel manner. Our framework translates the user-written code into actual implementation code in CUDA for GPU acceleration and MPI for node-level parallelization with automatic optimizations such as computation and communication overlapping. We demonstrate the feasibility of such automatic translations by implementing several structured grid applications in our framework. Experimental results on the TSUBAME2.0 GPU-based supercomputer show that the performance is comparable as hand-written code and good strong and weak scalability up to 256 GPUs.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming LanguagesLanguage Constructs and Features; D.1.3 [**Software**]: Programming TechniquesConcurrent Programming[Parallel programming]

General Terms

Languages

*Part of this work has been performed when the author was at Tokyo Institute of Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC11 November 12-18, 2011, Seattle, Washington, USA
Copyright 2011 ACM 978-1-4503-0771-0/11/11 ...$10.00.

Keywords

Domain Specific Languages, Application Framework, High Perforamnce Computing

1. INTRODUCTION

Heterogeneous computing with both conventional CPUs and vector-oriented GPU accelerators is becoming common because of superior performance as well as power efficiency. The peak performance of latest NVIDIA GPUs can be as high as 515 GFLOPS per chip, which is faster than the latest CPUs by several factors, allowing significant performance boost in compute-bound applications such as N-body problems [13]. GPU's memory bandwidth is also much greater than conventional CPU memory, reaching over 100 GB/s in properly aligned memory accesses, making it possible to achieve significant speedups in memory-bound applications such as computational fluid dynamics [23, 27].

Programming such heterogeneous systems, however, is a notoriously difficult task. The reason is two-fold. First, most of the existing programming models for such systems only provide low level platform-specific abstractions. The lack of high-level unified programming models forces the programmer to learn multiple distinctive models for parallel computing, e.g., message passing for distributed memory machines and GPU-centric models for accelerators, often resulting in ad hoc hybrid programming models. Since parallel programming even with a single model is known to be difficult and error prone, exploiting the potential performance advantage with hybrid models is thus a highly difficult task. Second, in the current heterogeneous architecture, data movements often involve complex performance considerations such as locality optimizations for keeping data close to processor cores and overlapping of communications and computations. While these techniques have long been well known and studied on parallel platforms, realizing them on complex heterogeneous systems further increases the programmer burden. As a result, further scaling performance with a large num-

ber of GPUs remains to be challenging even for highly skilled experts.

To solve the problem and improve programmer productivity, we envision a high level programming model that provides a uniform application programming interface for heterogeneous systems. While low-level interfaces are indeed essential when the maximum programming flexibility is required, such a case should not be common but exceptional, and simplifying programming even with limited flexibilities and small performance cost should be highly important to allow the adoption of heterogeneous systems for a wider range of application programmers.

This paper presents our high-level programming framework called *Physis* that is specialized to stencil computations with regular multidimensional Cartesian grids. In stencil computations each grid point is repeatedly updated by only using neighbor points, exhibiting regular spatial locality. Such a computation pattern, called "structured grids" [2], frequently appears in numerical simulation codes for solving partial differential equations. The performance of stencil applications is often determined by memory system performance since the typical byte-per-flop ratio in such code is higher than the ratio of today's processor and memory systems, including GPUs. Therefore, optimizing data movements is the most important to improve performance of such applications. Typical such optimizations for the GPU include latency hiding by scheduling a large number of concurrent threads, data alignment to allow coalesced memory accesses, and locality optimizations by thread blocking [24]. In addition to these optimizations, more coding effort is required to scale well with a large number of GPUs, such as communication and computation overlapping. GPU performance scalability is especially important for applications using a large amount of data, since a single GPU is equipped only with a few giga bytes of memory.

In the Physis framework, we design its programming model such that architecture neutrality can be realized on various parallel platforms, with a particular focus on GPU-based heterogeneous supercomputers. It provides portable and declarative constructs for describing stencil computations, such as creating multidimensional grids, data copying to and from them, and applying stencils over them. Global view memory model and implicit parallelism are adopted to realize high productivity as well as architecture neutrality. The declarative programming interface at the same time allows for static compilation techniques to automatically parallelize stencil computations over distributed memory environments with optimizations such as compute-communication overlapping. While it is beyond the scope of this paper, the framework is designed to allow for further advanced software techniques to be applied transparently for the application programmer, such as model-based and experimental performance tuning, resiliency through error checking and scalable and fast checkpointing [12, 18].

We describe an implementation of the framework based on the standard C language.[1] We introduce a small set of custom data types and intrinsics for stencil computations into C as an embedded DSL [10]. Those custom extensions are translated to platform native code, such as CUDA for GPU and MPI for message passing. Programs written in the Physis DSL can also be automatically translated to parallel code using MPI with the overlapping optimization for better scalability.

To evaluate our framework, we implement several stencil applications in the Physis DSL and evaluate its performance using the TSUBAME2.0 supercomputer at Tokyo Tech, which is the fourth fastest machine at the Nov 2011 list of Top500. We present results of performance studies using up to 256 NVIDIA Fermi GPUs, and demonstrate that our framework can achieve performance comparable to hand-written versions with good strong and weak scalability. To evaluate the productivity, we also compare lines of code in our DSL with native programming languages, and show that it is comparable to sequential code, indicating a similar level of productivity as sequential programming.

2. BACKGROUND

Stencil is one of the most fundamental computational patterns in numerical algorithms. Many of fluid dynamics phenomena can be described by partial differential equations (PDEs) over multi-dimensional Cartesian grids, such as weather and seismic waves, and their simulations can be implemented with stencil-based numerical algorithms such as finite-difference methods.

We define a stencil as an operation applied over points of multi-dimensional grids of data. More formally, let V be a two-dimensional grid defined over (i, j) coordinates. A general form of 2-D stencils can be written as follows:

$$V^{n+1}(i,j) = f(V^n(i+\alpha, j+\beta) | \alpha, \beta \in C)$$

where $V^N(i,j)$ is the value of the grid at (i,j) at time n, C is a set of constant integral values, and f is a function that takes a set of points at time n and yields the updated value of V at time $n+1$. Stencils are typically defined as an operator that performs computation for each point using its neighbor points. For example, a 5-point diffusion stencil can be written as

$$\begin{aligned} V^{n+1}(i,j) = 1/5(&V^n(i,j) + V^n(i+1,j) + V^n(i-1,j) \\ &+ V^n(i,j+1) + V^n(i,j-1)) \end{aligned}$$

While the basic form of stencils has simple and regular computation characteristics, and therefore relatively straightforward to parallelize, its actual applications to real-world problems can be more complex. One of the major complexities is due to the irregularities of boundary conditions. Although stencils are uniformly defined over the entire grid, it is common that boundary points require separate computations than interior points in order to reflect simulation settings accurately. There are some commonly used conditions such as Dirichlet and Neumann boundary conditions, but applications may use custom conditions to improve simulation accuracy for the problem at hand.

Optimizing data movements in stencils is highly important, yet introduces significant complexities in simulation code. There has been significant research devoted to such optimizations because of low arithmetic complexities and high memory I/O costs on modern microprocessors, including blocking of hardware and software caches [8, 9, 23, 30]. Datta et al. explored stencil optimizations, including cache blocking, memory affinity, and data padding [9]. Nguyen et al. further improved performance by employing both spatial and temporal blocking [19]. Kim proposed use of Mehrstellen discretization, which trades off computation costs

[1] The source code is freely available at `http://github.com/naoyam/physis`.

for faster convergence. This can be effective for stencils, since they are often memory bound and thus idle processing cores can be essentially exploited for free [16].

Since many of important science and engineering problems require large-scale simulations, their implementation techniques on distributed-memory machines have also been a long-term active research topic [25, 27, 29]. One of the most well-known optimizations for such environments is to hide the cost of boundary exchange communications by overlapping them with stencil computations. Typical parallelization on clusters of machines decomposes grids into a disjoint set of subdomains, where each of them is computed by one node. Since stencil operations require neighbor points, however, each node must exchange boundary points with other compute nodes responsible for neighbor subdomains. This communication requirement can introduce significant overheads, especially when a large number of nodes are involved, and thus reducing communication cost is highly important to improve application performance scalability.

3. HIGH-LEVEL FRAMEWORK FOR STENCIL COMPUTATIONS

We design a high-level programming framework that provides a highly productive programming environment for stencil computations. The framework consists of a domain-specific language and platform-specific runtimes. The DSL allows for declarative and flexible descriptions of stencils in an architecture-neutral way, which is then translated to architecture-specific code by source-to-source translators. The framework runtime encapsulates architecture-specific data management tasks and provides a uniform interface of virtual shared memory for multidimensional grids. The rest of this section discusses our major design goals of the framework.

3.1 Design Goals

Automatic parallelization.

We design the Physis DSL amenable to compiler-based automatic parallelization on distributed-memory parallel environments. Although automatic parallelization has been an active research topic for the past decades, it has not been widely successful in practice for general-purpose languages, especially on distributed memory environments, since effectively exploiting data localities available in applications is a complex and difficult task [15]. In contrast, our framework is limited to a small set of domain-specific computations, but by doing so we eliminate the difficulties of automatic parallelization in conventional general-purpose languages, and realize implicit parallelism on a variety of parallel platforms.

Embedded DSL rather than external DSL.

Inventing a completely new language for a given problem domain, i.e., an external DSL approach [10], potentially allows for a maximally optimized language design. In practice, however, being dissimilar to existing familiar languages may hinder adoption by a wide body of application programmers. We design our DSL as a small set of extensions on existing general-purpose languages, i.e., an embedded DSL [10]. We choose C as the base language in our current design and implementation since it is one of the most commonly used languages in high performance computing.

Declarative and expressive programming model.

In order to improve productivity, we maximize programming abstraction by adopting a declarative programming model that allows for less manual programming than imperative models. For example, in the Physis DSL the programmer expresses how each grid element is computed, but it is determined by the framework how the whole grid is processed with the user-specified computation; the overall computation may be performed sequentially or in parallel depending on target environments of the framework. Having too much abstraction, however, can be too restricted to implement real-world scientific simulations. For example, some stencils may be applied only to a part of a whole grid, such as boundary regions. Although we attempt to keep the language extensions minimal, we adopt additional domain-specific constructs if they can further improve productivity of programmers and performance of final implementation codes.

4. PROGRAMMING MODEL

The Physis DSL extends the standard C with several new data types and intrinsics for stencil computations. The user is required to use the extensions to express stencil-based applications, which are then translated to actual implementation code by the Physis translator.

4.1 Runtime Initialization and Finalization

The user program must first initialize the Physis runtime environment by `PSInit` before any use of Physis extensions, which can then be destroyed by `PSFinalize`:

```
void PSInit(int *argc, char ***argv,
   int num_dimensions, size_t max_dim_i,...)
void PSFinalize()
```

The first two parameters of `PSInit` are assumed to be pointers to the command-line argument number and pointers, as in `MPI_Init` of MPI. The additional parameter specifies properties of the *global domain*, where multidimensional grids can be created. The `num_dimensions` parameter specifies the number of dimensions of the global domain and the rest of parameters specify the maximum size of each dimension. The number of the additional parameters must be the same as the number of dimensions.

4.2 Using Multidimensional Grids

4.2.1 Grid Data Types

Physis supports multidimensional Cartesian grids of floating-point values (either `float` or `double`). Grids of structs are not currently supported; they can be represented by using multiple separate grids of floats or doubles.

To represent multidimensional grids, we introduce several new data types named based on its dimensionality and element type, e.g., `PSGrid3DFloat` for 3-D grids of `float` values and `PSGrid2DDouble` for 2-D grids of `double` values. The type does not expose its internal structure, but rather works as an opaque handle to actual implementation, which may differ depending on translation targets.

Since many of the Physis intrinsics are overloaded with respect to the grid types, below we simply use `PSGrid` to specify different grid types when not ambiguous.

4.2.2 Creating and Deleting Grids

Grids of type PSGridFloat3D can be created and destructed with intrinsics PSGridFloat3DNew and PSGridFree, as defined as follows:

```
PSGrid3DFloat PSGrid3DFloatNew(
    size_t dimx, size_t dimy, size_t dimz,
    enum PS_GRID_ATTRIBUTE attr)
void PSGridFree(PSGrid g)
```

The first three parameters of PSGrid3DFloatNew specify the size of each of the three dimensions. Similarly, intrinsics for creating double-type grids and 1-D and 2-D grids are provided. The size of each dimension of grids can be retrieved by intrinsic PSGridDim. Parameter attr is an optional parameter to specify a set of attributes. Currently the only supported attribute is PS_GRID_PERIODIC, which designates that the grid to be created can be accessed with the periodic boundary condition.

4.2.3 Grid Reads and Writes

Grids can be accessed both in bulk and point-wise ways. Bulk reads and writes are:

```
void PSGridCopyin(PSGrid g, const void *src)
void PSGridCopyout(PSGrid g, void *dst)
```

PSGridCopyin copies the continuous chunk of memory pointed by the second parameter into the given grid, while PSGridCopyout copies the grid element values into the memory pointed by the second parameter. The size of data copy is determined by the element type and size of the given grid. Physis assumes the column-major order storage is used in multidimensional grids.

Each point of grids can be accessed using the following three intrinsics:

```
// For 3-dimensional type-T grids
T PSGridGet(PSGrid g,
            size_t i, size_t j, size_t k)
void PSGridSet(PSGrid g,
               size_t i, size_t j, size_t k, T v)
void PSGridEmit(PSGrid g, T v)
```

The set of size_t parameters specify the indices of a point within the given grid, so the number of index parameters depend on the dimensionality of the grid (e.g., three for 3-D grids). The return type of PSGridGet and the v parameter of PSGridSet and PSGridEmit have the same type as the element type of the grid, which is either float or double.

PSGridGet returns the value of the specified point, while PSGridSet writes a new value to the specified point. PSGridEmit performs similarly to PSGridSet, but does not accept the index parameters, and is solely used in stencil functions as described below.

4.3 Writing Stencils

4.3.1 Stencil Functions

Stencils in Physis are expressed as *stencil functions*, which are standard C functions with several restrictions. Stencil functions represent a scalar computation of each point in grids. At runtime, stencil functions may be executed sequentially or in parallel. Figure 1 illustrates a 9-point stencil function for 2-D grids.

There are five restrictions on stencil functions. First, the function parameters must begin with const int parameters, which represent the coordinate of the point where this function is applied, followed by any number of additional parameters, including grids and other scalar values. Non-scalar parameters other than grids are not allowed in stencil functions. The return type of stencil functions must be void.

Second, function calls within stencil functions must be either 1) calls to intrinsics PSGridGet, PSGridEmit, or PSGridDim, 2) calls to builtin math functions such as sqrt and sin, or 3) nested calls to stencil functions. The available math functions depend on a particular target platform, since we simply redirect such calls to platform-native builtin functions. The user can often assume the availability of the standard libc math functions since most of them are supported in CUDA. Stencil functions can also use nested stencil functions, which are subject of the same set of restrictions, except for the function parameter and return type requirement. Nested stencil functions are analogous to non-global device functions in CUDA.

Third, the stencil index arguments of PSStencilGet must match the pattern of x + c, where x must be one of the index parameters of the stencil function and c be an integral immediate value. For example, PSGridGet(g1, x, y, z) in Figure 1 is accepted by our translator, but not PSGridGet(g1, x + t, y, z), where t is not an immediate value but a given parameter. Furthermore, the order of index parameters appearing in PSStencilGet must match the order of the parameters of the stencil function. For example, PSGridGet(g1, z, y, x) is not legal in Physis. This restriction allows us to assume that data dependencies between stencil points can be statically identified and that at runtime they can be efficiently resolved by neighbor data exchanges.

Fourth, in stencil functions, aliases of grid variables must be unambiguously analyzable. Since our current translator supports only a very simple alias analysis, each grid variable must follow the form of static single assignments. Taking the address of a grid variable is also not allowed in stencil functions.

Finally, a stencil function may be executed in parallel with an arbitrary order, so the programmer must not assume any read-after-write dependency among different stencil points within a function. Such dependency can only be enforced between different invocations of stencil functions.

These restrictions are to enforce regular neighbor data accesses patterns in stencil functions, and to allow for static generation of efficient parallel code. Some of them could be relaxed and complimented by runtime analysis and code generation. For example, we could allow for arbitrary index arguments in PSGridGet and let the runtime dynamically resolve data dependencies. However, in general, the cost of such operations is difficult to eliminate by static compilation approaches like ours. Since our current framework prioritizes efficiency of generated code over flexibility, the translator does not accept code that violates the above restrictions. Other standard C constructs such as branches and loops can be used as usual.

4.3.2 Applying Stencils to Grids

Stencil functions can be applied to grids by using two declarative intrinsics: PSStencilMap and PSStencilRun. Figure 2 illustrates how these intrinsics can be used to invoke

```
void diffusion(const int x, const int y,
        PSGrid2DFloat g1, PSGrid2DFloat g2, float t) {
  float v = PSGridGet(g1, x, y)
      + PSGridGet(g1, x+1, y) + PSGridGet(g1, x-1, y)
      + PSGridGet(g1, x, y+1) + PSGridGet(g1, x, y-1)
      + PSGridGet(g1, x+1, y+1) + PSGridGet(g1, x+1, y-1)
      + PSGridGet(g1, x-1, y+1) + PSGridGet(g1, x-1, y-1);
  PSGridEmit(g2, v / 9.0 * t);
  return;
}
```

Figure 1: Example 9-point stencil function

the diffusion stencil of Figure 1 on 3-D grids.

`PSStencilMap` creates an object of `PSStencil`, which encapsulates a given stencil function with its parameters bound to actual arguments. It is analogous to closures in functional programming.

```
PSStencil PSStencilMap(StencilFunctionType stencil,
  PSDomain3D dom, ...)
```

The `stencil` parameter must be the name of a stencil function. We do not support specifying functions with function pointers and other indirect references, and only an actual name with its definition in the same compilation unit is accepted so that the translator can determine the actual stencil function at translation time.

The type of the second parameter, `PSDomain3D`, is a new data type to specify a 3-D rectangular range, which can be instantiated by the following intrinsic:

```
PSDomain3D PSDomain3DNew(
  size_t x_offset, size_t x_end,
  size_t y_offset, size_t y_end,
  size_t z_offset, size_t z_end)
```

Similar to grid types, Physis defines 1-D and 2-D variants of the domain types (i.e., `PSDomain1D` and `PSDomain2D`). The domain object can be used to restrict the region of grid points where the stencil function is applied. For example, when interior and boundary points have different computations to update their values, it can be implemented by creating different stencil functions for interior and boundary regions, and by selectively mapping them to the corresponding regions using domain objects.

`PSStencilRun` executes `PSStencil` objects in a batch manner, as defined as follows:

```
void PSStencilRun(PSStencil s1, PSStencil s2,
  ..., int iter)
```

It accepts any number of `PSStencil` objects, and executes them in the given order for `iter` times. Each stencil function may be executed in parallel, exhibiting implicit parallelism.

The combination of `PSStencilMap` and `PSStencilRun` forms a natural unit of code translation. First, it allows for efficient implementations on distributed environments, where we use RPC-based controls that cause global synchronizations between the master node and every other compute nodes. A `PSStencilRun` call is translated to a single RPC request from the master node to the compute nodes, and the compute nodes execute the stencil objects for the specified number of times with no further RPC communications. Alternatively, we could design a DSL without `PSStencilRun` and simply

```
PSInit(&argc, &argv, 3, NX, NY, NZ);
PSGrid3DFloat g1 = PSGrid3DFloatNew(NX, NY, NZ);
PSGrid3DFloat g2 = PSGrid3DFloatNew(NX, NY, NZ);
// initial_data is a pointer to input data
PSGridCopyin(g1, initial_data);
PSDomain3D d = PSDomain3DNew(0, NX, 0, NY, 0, NZ);
PSStencilRun(PSStencilMap(diffusion, d, g1, g2, 0.5),
             PSStencilMap(diffusion, d, g2, g1, 0.5),
             10);
// result is a pointer to hold result  data
PSCopyout(g1, result);
PSFinalize();
```

Figure 2: Example code to declare grids and run the diffusion functions

use `PSStencilMap` to execute a given stencil function without creating closure-like objects. A straightforward translation, however, would yield one RPC for each stencil object, resulting in more frequent RPCs than with `PSStencilRun`.

Second, the combination can be a unit of further code optimizations. We plan to study such optimizations as fusing and reordering of stencil functions. Explicit grouping of stencil functions would simplify applying such aggressive optimizations to user code at the translation time.

5. FRAMEWORK IMPLEMENTATION

Our framework implementation consists of a source-to-source translator and runtime components for each target platform. As translation targets, we currently generate C for CPU execution and CUDA for GPU execution. In addition, for platforms involving multiple distributed compute resources, we generate message-passing parallel code using MPI. This paper first presents our baseline implementation of the translator and its runtime for GPU clusters, followed by a series of optimizations.

5.1 Runtime

The runtime for GPU clusters implements a virtual shared-memory space for multidimensional grids. It is initialized with the number of dimensions and the size of each dimension, which defines the global domain. The runtime uniformly decomposes the global domain over all the processes as instructed by a user-controllable parameter. For example, when 64 processes are used for 3-D domains, the user can specify decompositions as, e.g., 4x4x4, 1x8x8, or 1x1x64. Each process is assigned its offset and size for the subdomain that the process is responsible for.

One of the runtime processes, which is the rank-0 process in MPI, runs as the master process and accepts requests on grid operations. The master process issues RPCs to other processes to implement accepted RPC requests, and also performs its own computation. The RPC-based master-client parallel execution is chosen to maintain the semantics that functions other than stencils are executed sequentially only by one process.

The runtime provides APIs to implement the DSL intrinsics, including creating and deleting grids, bulk and point-wise data accesses, and executing stencil functions. Newly created grids are automatically decomposed over all the processes based on the global domain decomposition defined at the initialization time. The RPC master instructs the clients to create a subgrid that overlaps with its subdomain. In ad-

dition to the subgrid container, we also allocate buffers for the boundary data of neighbor processes. Note that since our runtime uses GPU accelerators, grids are allocated on GPU memory rather than host CPU memory.

Grid data can be accessed either by RPCs from the master process or by direct array accesses from local processes. The RPCs include bulk and point-wise RPC APIs; copyins and copyouts basically perform scatter and gather over all processes, and point-wise data accesses are implemented as a trivial point-to-point message exchange between the master and the client responsible for the specified coordinate. The runtime also allows for each local process to obtain a pointer to the continuous chunk of memory for its subgrid data, so the grid reads and writes within stencil functions can be translated to direct array accesses.

The RPC API also includes a function to execute a set of stencils for a given number of times. The master accepts a list of `PSStencil` objects and iteration count, and broadcasts them to the client processes. Once they become available, all processes, including the master, start execution of the functions designated by the `PSStencil` objects with the given arguments. Note that the functions must be ordinary C functions—offloading stencils to GPU is done by the translator, which generates offloading stubs for stencil functions. Once this RPC is initiated, all processes run with no interference with one another until completion except for boundary data exchange.

Boundary data exchange is implemented also using the runtime API. It is not an RPC but rather a collective operation where all processes call the same function with the same set of arguments, including the identifier of the grid, its halo forward and backward widths for each dimension, and a boolean value indicating whether the diagonal points need to be exchanged. Data exchanges are performed only between neighbor processes; however, for grids with the periodic boundary condition, the runtime at each end of domain communicates with the processes with the other end of domain to allow wrap-around data accesses.

We use the CUDA API and our own kernels to copy boundaries between host and GPU memory, and MPI for inter-process communication. More specifically, for boundaries that are accessible with unit stride, we simply use `cudaMemcpy` to copy from GPU memory to host page-locked memory. For non-unit stride cases, in order to avoid issuing `cudaMemcpy` multiple times, we map a chunk of host memory to GPU memory space by `cudaAllocHost` and invoke a CUDA kernel that reads boundaries with non-unit stride and directly writes the data into host memory through the mapped memory addresses. Once boundary data are available on host memory, we exchange them using MPI point-to-point communication routines. Note that since the page-locked memory for CUDA GPU memory copies may not be used as MPI send/recv buffers when Infiniband is used, the runtime manages data copies between CUDA and MPI buffers automatically.

As described above, the runtime hides many of the details involved in executing stencil applications on GPU-based heterogeneous clusters. While our current implementation only supports CUDA, we do not see any significant technical difficulties to implement the same interface on other accelerator APIs such as OpenCL. Also, we will extend our runtime as the underlying software and hardware evolves. For example, the CUDA version 4, which will be released soon, supports direct data transfers between node-local GPUs, potentially reducing the pressure to host memory traffic, and even GPU-NIC direct data transfers may become possible in future GPU and network cards. We will optimize our runtime by exploiting such new features so that the user program can transparently benefit from them.

5.2 Translation

Our source-to-source translator builds on top of the ROSE compiler infrastructure [26]. ROSE supports conversion of standard C/C++ code into Abstract Syntax Tree and its analyses and transformations. Once transformations are completed, the final AST can be translated back to source code, which can then be compiled by platform-native compilers.

To use ROSE, we first must make the Physis DSL compatible as an input language to ROSE. While most of the DSL intrinsics are syntactically legal C, and thus can be accepted by ROSE as is, some of the overloaded intrinsics are not, including `PSGridGet`, `PSGridEmit`, `PSGridSet`, `PSStencilMap`, and `PSStencilRun`. We use standard C preprocessor macros to convert such intrinsics to syntactically legal C without any loss of semantics.

Grid variables in Physis are translated to `void*` pointers, which are then passed to appropriate functions of the RPC master. The RPC master then executes the given requests as described above. Most of the Physis intrinsics can be directly supported by the runtime RPC API, including grid creation and deletion, copyins and copyouts, and point-wise data accesses from non-stencil functions. Stencil functions and their invocations by `PSStencilRun` and `PSStencilMap`, however, need significant rewrites as described below.

For each stencil map, we generate its stub function that first performs boundary data exchanges using the runtime API, and then invokes the stencil function given to the map call. We translate the original stencil function to a CUDA global function so that it can be executed on GPU. The CUDA block size is 64 by 4 by default, and is controllable by the user. Each CUDA thread computes all points along with the third dimension (z). The grid writes, i.e., calls to `PSGridEmit` intrinsic, are translated to accesses to the subgrid linear array. For each grid read with `PSGridGet`, we generate additional runtime conditional code that determines whether the read operation accesses its own subgrid or the neighbor halo regions. This is required because we allocate separate buffers for the regions, but can be a significant performance bottleneck since it involves several conditional branches with extra usage of registers. We eliminate it for reads with zero-offset index arguments, i.e., `(x, y, z)` where each of the variables are the index parameters of a stencil function, since in that case the coordinate can be safely assumed to exist locally. Otherwise, we apply an optimization that attempts to eliminate the check from the critical path as described below.

5.3 Optimizations

One of the common optimizations of stencils on large-scale environments is to overlap stencil computations and boundary exchanges so that the communication cost can be hidden within the computation time. This is particularly important on GPU clusters since they involve additional data copies between GPU and CPU memory.

We extend the translator to generate overlapping code by

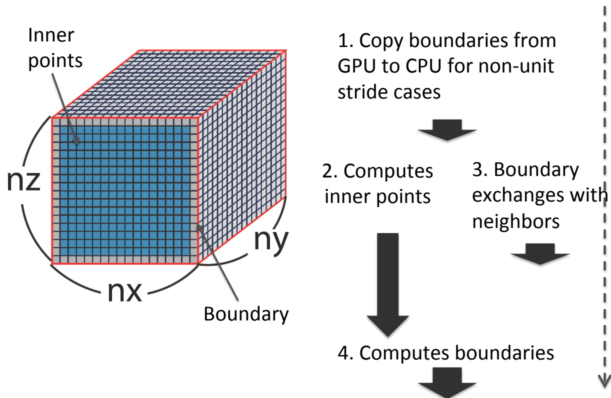

Figure 3: Overlapped stencil computations and neighbor exchanges.

dividing the subdomain of each process into its interior and boundary regions, as illustrated in Figure 3. Since we enforce that grid reads in stencil functions must have static offsets, we can statically identify the width of each halo boundary that needs to be exchanged. Based on the information, we generate separate kernels for the interior and boundary regions and let the kernels and transfers run concurrently as much as possible. For 3-D problems, we generate seven GPU kernels for stencil computations: one for the interior region and six for the six boundary planes. At each iteration, we first launch boundary copy kernels for non-unit stride boundaries, which directly copy the data from GPU memory to mapped host memory. Then, the interior kernel is launched and run asynchronously on GPU. Next, we start asynchronous transfer of the remaining boundary data using `cudaMemcpyAsync`. When both the interior kernel and boundary transfers are finished, the boundary kernels start to run. Since the boundary kernels typically compute only a small fraction of the overall subdomain, each of them may not have enough parallelism itself to fully exploit the GPU. Our multi-stream optimization exploits the parallelism between boundary kernels as well by executing the six boundary kernels in parallel using the concurrent kernel execution feature available in Fermi GPUs.

Another optimization is to eliminate the runtime checking of coordinate locations from interior kernels, since we can safely assume that any of `PSGridGet` in interior kernels does not fall into halo regions. This optimization is not applicable to boundary kernels, since they may access the halo buffers of neighbor subdomains.

Finally, we further optimize address computations of grid points by common subexpression elimination. Stencil functions typically use multiple neighbor grid points, for which address calculations are done individually if no optimization is applied. Instead, we pick one of the accesses as a reference point, for which we compute its address normally, but for the other accesses to the same grid, we replace their address calculations with a mere addition of the reference address and the offset from it. Since the address of reference point must be calculated at a control flow node that dominates all of the reads, we conservatively move it to the beginning of the function. A similar optimization appears to be performed by nvcc compiler v3.2 too; however, PTX output code indicates that this is done only for the case when access indices only differ at the first dimension.

6. EXPERIMENTAL EVALUATION

To evaluate our framework, we used three stencil codes as benchmark programs.

Diffusion: 3-D 7-point stencil.

Himeno: 3-D Jacobi kernel of the Himeno benchmark [14].

Seismic: 3-D seismic wave simulation with 27 stencil functions. The original code is written in Fortran and is developed by Takashi Furumura at University of Tokyo.

The first two codes are relatively small scale, consisting of only one stencil function each, whereas the seismic code consists of 27 stencils with staggered grids. Among the 27 stencils, six are for computing the 2-D surfaces of the 3-D domain, which are implemented with PSDomain objects. All benchmarks use single-precision floating-point data.

We use the TSUBAME2.0 supercomputer at Tokyo Institute of Technology, which consists of 1408 compute nodes. Each node has two Intel Xeon Westmere-EP 2.9GHz CPUs and three NVIDIA M2050 GPUs with 52GB and 3GB of system and GPU memory, running SUSE Linux Enterprise Server 11 SP1. The compute nodes are interconnected by dual QDR Infiniband networks with a full bisection-bandwidth fat-tree topology network. We use CUDA v3.2 for GPU code and gcc/g++ v4.1.2 for CPU code.

We first illustrate the effectiveness of our framework in improving programmer productivity by comparing the lines of code of each benchmark, and show that programs written in Physis can be as compact as sequential code. Next, we evaluate the performance of the three benchmarks and demonstrate the scalability of automatically generated parallel code. All of the performance results are obtained with the CUDA block size fixed at 64x4.

6.1 Comparison of Code Size

Figure 4 compares the lines of code of each benchmark. The baseline is the size of original sequential code, which is 160, 276, and 884 lines, respectively. Note that the original diffusion and Himeno codes are written in C, while the seismic is in Fortran. We can see that for diffusion and Himeno the size of Physis source code is comparable to its sequential version and is considerably less than the manually written MPI version, indicating that Physis successfully maintains a similar level of productivity as sequential programming.

The Physis version of the seismic code is larger than its sequential version by approximately 50%, and is comparable to the MPI version. We suspect that this is because the manual codes are written in Fortran, and believe that Fortran-binding of Physis would be able to achieve the similar level of code size as the Fortran sequential code.

Although smaller code size does not necessarily mean higher programmer productivity, it is still a useful and easy-to-understand metric to compare program complexity and productivity. While our study of code size is still limited to the three sample codes, the above preliminary results present promising results of our framework design.

6.2 Effects of Optimizations

Figure 5 shows the effects of framework optimizations in the diffusion benchmark. We fix the problem size of each

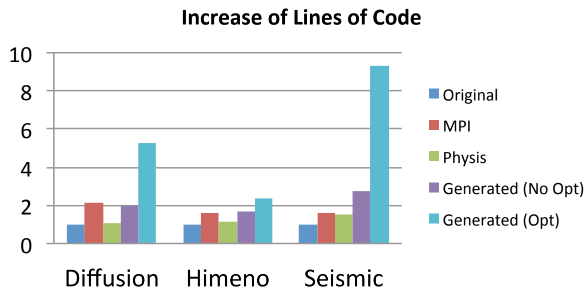

Figure 4: Comparison of lines of code, excluding white spaces and comment lines. Y-axis shows relative increase of code size compared to the corresponding sequential code.

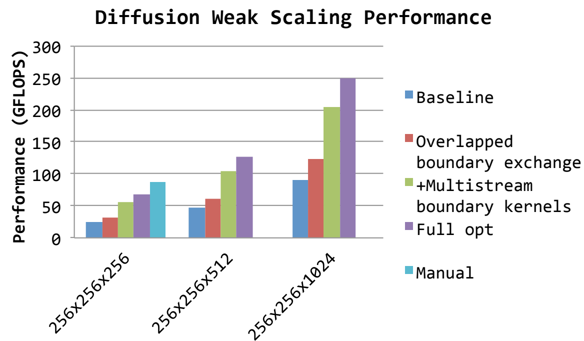

Figure 5: Effects of the performance optimizations on the diffusion stencil. The problem sizes are varied from 256x256x256 to 256x256x512, and to 256x256x1024, for which we use one, two, and four GPUs, respectively.

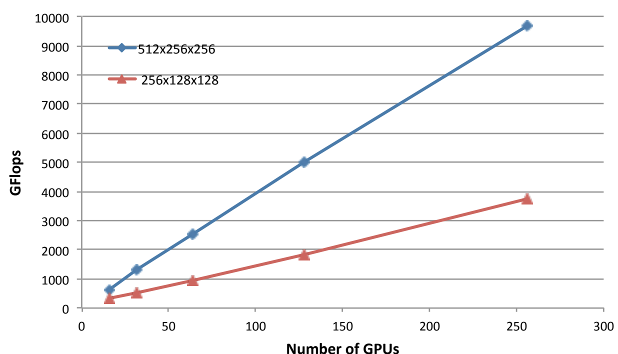

Figure 6: Weak scaling performance of Diffusion benchmark with up to 256 GPUs.

GPU at 256^3 and evaluated the weak scaling performance from one GPU to four GPUs with 1-D decomposition along the z-direction. The baseline is the performance with no optimization, and the rest of the bars show performance improvements by incrementally applying the optimizations. The first optimization is to overlap the boundary data exchange and interior region computations. The second optimization is to decompose the boundary kernel into six kernels, each of which is for one of the 2-D surface, and executes them concurrently using the concurrent kernel execution feature of the Fermi GPU. The full optimization case is to apply all the previous optimizations and the common subexpression elimination on the address computations of grid points. For the one-GPU case, we also show the performance of a manually written CUDA code, which is implemented straightforwardly without optimizations.

We can see that the overlapping optimization realizes modest performance improvements with all the three problem settings, including the single-GPU case. As described in Section 5.3, for each read of grid points, the DSL translator by default generates runtime conditional code that determines whether the coordinate resides in halo regions or not, which can be highly expensive in GPUs. Even the single-GPU case is affected by this overhead since this experimentation uses the same generated source in MPI and CUDA. One side effect of the overlapping optimization is that we can statically determine that the interior kernel does not need such runtime check since the kernel is exclusively used for interior points. Therefore, we can safely remove the overhead from the interior kernel, which in fact resulted in considerable performance improvements. The actual improvements, however, were relatively small because the boundary computation kernel spent nearly the same time as the interior kernel. This is greatly alleviated by the multi-stream optimization, which concurrently executes the six boundary kernels for 3-D grids.

Overall, the auto-generated single GPU code achieved 77% of the performance of the non-optimized manual code. Section 6.5 presents performance comparisons with tuned kernels.

6.3 Weak Scaling Evaluation

Figure 6 shows the results of weak scaling evaluation with the diffusion code. The red and blue lines are the cases where each GPU is assigned a subdomain of 256x128x128 and 512x256x256, respectively. In both cases, the 3-D domain is decomposed only over y and z dimensions. As expected, the larger problem size allowed for better performance and scaling, which is almost linear scaling up to 256 GPUs, although even the smaller case achieved 11.64 times speedup with 256 GPUs compared to the 16-GPU case.

Figure 7 shows the results of weak scaling evaluations with the seismic code, where each GPU computes a subdomain of 256^3 region. Unlike the diffusion case, the problem domain is decomposed over x and y dimensions; in other words, the domain is expanded in the x-y plane with the problem size of each GPU fixed. The decomposition implies that boundary exchanges involve non-unit stride data accesses, thus resulting lower scalability than the diffusion code. The performance of seismic benchmark exhibits significant drop at 64 GPUs and relatively low scalability afterward, which remains to be a subject of more detailed performance analysis.

6.4 Strong Scaling Evaluation

Figure 8 shows the strong scaling performance of the diffusion stencil with the problem size fixed at 512x512x4096. We evaluated 1-D, 2-D, and 3-D decompositions using up to 128 GPUs. In the 1-D decomposition, we uniformly decomposed the z-direction by the number of GPUs. In 2-D, we also decomposed the y-direction by two GPUs and again

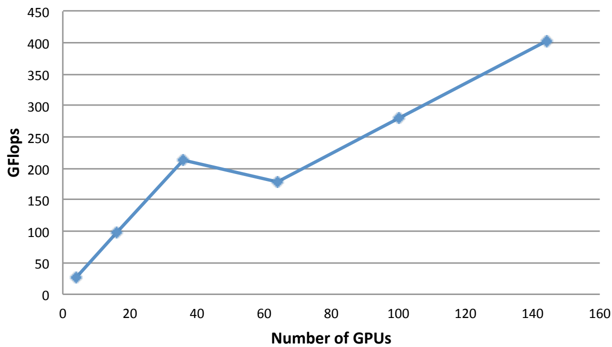

Figure 7: Weak scaling performance of Seismic benchmark with up to 144 GPUs. The problem size of each GPU is fixed at 256x256x256.

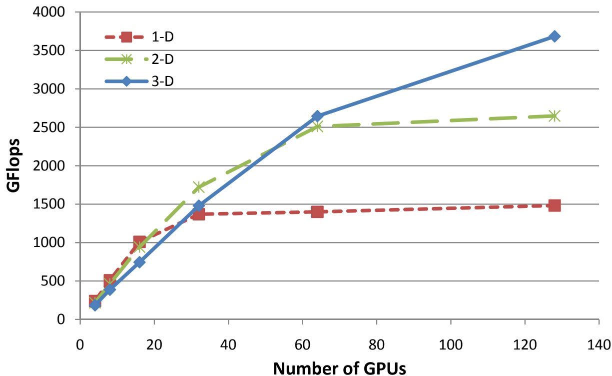

Figure 8: Performance of 512x512x4096 diffusion with three different decompositions.

uniformly decomposed the z-direction with the rest of GPUs. Similarly, in the 3-D decomposition, we decomposed the x-direction into two GPUs in addition to the uniform y- and z- direction decompositions. As expected, the 1-D and 2-D cases performed better with a smaller number of GPUs, but as the number increases, the 3-D version outperformed the other two versions. While it is well known that 3-D decomposition often performs better in large-scale settings, our contribution is to allow application scientists to transparently enjoy such better performance and scalability without investing significant amount of efforts.

Figure 9 shows the performance of the Jacobi kernel of the Himeno benchmark of size XL (1024x512x512). We evaluated 1-D, 2-D, and 3-D decompositions, each of which we used the experimentally obtained best decompositions, such as 4x4x8 for the 3-D decomposition with 128 GPUs. As seen in the graph, 2-D and 3-D decompositions scale well up to 128 GPUs. More specifically, the 2-D decomposition with 16 GPUs achieved 468 GFLOPS, and reached 2224 GFLOPS with 128 GPUs; the 3-D decomposition with 16 GPUs achieved 456 GFLOPS, and reached 2506 GFLOPS with 128 GPUs. The parallel efficiency when the number of GPUs is increased from 16 to 128 is 59.3% with 2-D and 67.5% with 3-D decomposition. While in these experiments we used experimentally obtained best decomposition, automating such manual processes is a subject of our future research.

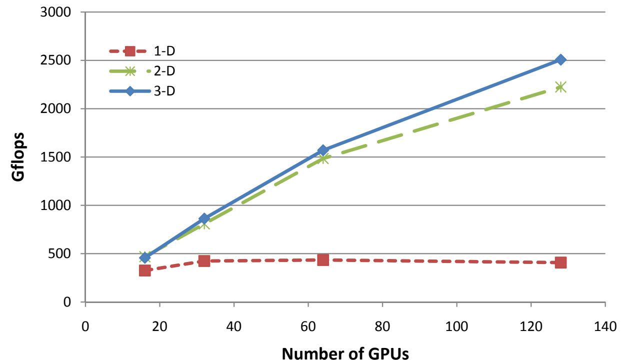

Figure 9: Strong scale performance of the Jacobi kernel of Himeno benchmark of size XL (1024x512x512). Each line shows the performance of 1-D, 2-D, or 3-D uniform decomposition.

6.5 Comparison with Other Implementations

One of the fastest implementations of the 7-point stencil is reported by Nguyen et al. [19], achieving 117 GFLOPS on a GeForce GTX285 GPU for single precision with spatial blocking optimization. On the other hand, Physis performance of the same kernel is 67 GFLOPS on Tesla M2050. Because the kernel is memory bound and also because the memory bandwidth of the two devices are quite similar (159 GB/s and 148 GB/s, respectively), we can see that the performance of our auto-generated version is approximately 60% of hand-tuned code. In addition to spatial blocking, Nguyen et al. reported 220 GFLOPS performance when temporal blocking is also employed. We expect that once it is implemented in our DSL translator, it would also be able to substantially improve the performance of Physis code.

7. LIMITATIONS

Although the current realization of the proposed framework has demonstrated promising results, it is by no means complete. In addition to more sophisticated optimizations and platform support (e.g., OpenCL), there are several technical restrictions and limitations that need to be carefully examined before applying the framework to applications.

Data Models: Not all stencil-based simulations can be expressed in our framework since it specifically focuses on dense multidimensional Cartesian grids. For example, discrete modeling with complex unstructured grids is not possible to implement in this framework. As in Chapel [4], where multiple different types of data models are supported as first-class language citizens, it would be possible to introduce different types of grids into the DSL by extending its translator and support runtime.

Limited Data Accesses: The framework supports limited data access operations on grids. They are most efficiently accessed within stencil functions, which have several restrictions how they are written as described in Section 4.3.1. For example, stencil sizes and directions must be statically determined. This restriction implies that the framework cannot efficiently implement applications with irregular data access patterns, although it significantly simplifies automatic parallelization of our target computation patterns.

Interoperability: The framework lacks well-defined interoperability with existing parallelization methods such as MPI.

External sequential programs can use Physis-generated programs with the standard C function call convention, since the DSL translator simply generates C code with the same function signatures as its input. However, we assume parallel resources are exclusively available to Physis, and thus the generated code may not correctly interoperate with already parallelized code. This is not desirable for certain cases, e.g., incremental porting of existing parallel applications with a large code base. To improve the interoperability, we plan to define interfaces to work with common parallelization methods such as MPI.

8. RELATED WORK

Recent developments of GPU accelerators for scientific computing have enabled low-cost power-efficient approaches to increase compute performances. However, one of the side effects of this trend is the significant decrease of programmer productivity due to the complexities involved in programming heterogeneous architecture. Although little work has been done for large-scale heterogeneous supercomputers, several recent projects attempted to solve the problem.

Mint is a high-level directive-based framework for stencil computations [28]. It allows for regular loop-based stencil programs to be annotated with its custom directives so that stencil loops can be executed on GPUs. Ypnos is a Haskell-based DSL for stencil computations that is designed so that compiler-based automatic parallelization is possible [22]. Both of them share common objectives with ours, such as automatic parallelization, but so far they are limited to single-GPU platforms, whereas our primary focus is to realize scalable multi-GPU implementations.

Listz is a DSL for unstructured mesh-based simulations [6]. As in our framework, actual implementations of the mesh interface are hidden from the programmer, which allows Listz to perform aggressive domain-specific optimizations. The implementation of the Listz is based on Scala's extensive language features, which facilitate developments of DSLs. Listz could be use to implement structured-grid stencil applications; however, since it targets unstructured meshes, it may not be able to fully exploit the optimization opportunities of structured data.

Designing a new programing language for improving programmer productivity in high performance computing has been pursued by projects such as X10 [7], Chapel [4], and Fortress [1]. X10 and Chapel provide a rich set of parallel programming constructs such as controlling data affinity as in other PGAS languages such as UPC [5] and CoArray Fortran [20]. While these languages are designed to support general purpose programming, Fortress allows for higher level programming by supporting intuitive mathematical notations in program source code. We could also support higher level of notations for expressing simulation code. For example, partial differential equations can be more naturally expressed with mathematical notations than C or Fortran, so extending Physis with such notations would be of more useful for domain scientists.

Programmer productivity may be improved by library-based approaches as well. A library of numerical methods may be able to provide a clear separation between application logic and its implementation. This approach is widely successful in some of common scientific code, such as BLAS for linear algebra [17] and FFTW for fast Fourier transform [11]. Similarly, there have been attempts to provide libraries for the same application domain as ours, such as OpenFOAM [21]. Our DSL-based framework is more general in the sense that it can be used to implement such libraries as an architecture-neutral implicitly parallel programming substrate. Our framework can also be compared with more general software toolkits such as PETSc [3]. While such packages support a much wider variety of scientific applications than our stencil framework, we realize more advanced software-based techniques, including automatic parallelization, architecture-neutral programming, and automatic optimizations.

9. CONCLUSION

In order to improve programmer productivity on large-scale heterogeneous GPU clusters, we have designed and implemented the Physis framework that supports portable programming of stencil computations with structured grids. The C-based DSL represents a high-level declarative programming model for stencil computations. The DSL translator and runtime together realize an efficient implementation of the programming model with optimizations such as automatic overlapping of computations and communications. This paper presented our current framework implementation and evaluations of its productivity and performance. We have shown that our framework successfully generates scalable code for up to 256 GPUs.

We are currently working on the following directions. First, we need to experiment with more applications to fully understand and evaluate the expressiveness of the DSL and its implementation. For example, we are currently working on 3-D Lattice Boltzmann Method. Our preliminary study indicates that LBM can be implemented with the Physis DSL rather straightforwardly. Second, performance evaluations with the current three sample codes are limited to 256 GPUs, although TSUBAME has more than 4000 GPUs. We will evaluate scalability of our implementations with larger scale experiments. Third, we will implement more performance optimizations such as stencil fusing and tuning of CUDA block sizes and domain decompositions. Finally, we plan to extend the backend targets to include other architectures, such as multicore CPUs and other accelerators. A promising approach would be to generate code in OpenCL; however, since best performing optimization strategies would depend on the target architecture, architecture-specific optimizations should be applied for optimal performance. Physis would be able to encapsulate such optimizations within the framework so that the programmer can develop applications in a performance-portable way.

Acknowledgments

We deeply thank Mark Silberstein for valuable discussions on the initial design of the domain-specific approach. We also thank the ROSE project for its development of the highly useful compiler framework, and Prof. Furumura for the seismic simulation code. This project was partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing." We also thank MEXT Grant-in-Aid for Young Scientists (22700047), JST-ANR FP3C, and NVIDIA CUDA Center of Excellence.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The fortress language specification, March 2008.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar. 2008.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] H. P. Z. Bradford L. Chamberlain, David Callahan. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[5] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical report, IDA Center for Computing Sciences, 1999.

[6] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 835–847, 2010.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[8] J. M. Cohen and J. Molemake. A Fast Double Precision CFD Code Using CUDA. In *21st International Conference on Parallel Computational Fluid Dynamics (ParCFD2009)*, 2009.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1–12, 2008.

[10] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[11] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb. 2005.

[12] L. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Low-overhead diskless checkpoint for hybrid computing systems. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1 –10, dec. 2010.

[13] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[14] R. Himeno. Himeno benchmark. `http://accc.riken.jp/HPC_e/himenobmt_e.html`, July 2011.

[15] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, New York, NY, USA, 2007. ACM.

[16] T. Kim. Hardware-aware analysis and optimization of stable fluids. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 99–106, 2008.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, September 1979.

[18] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[19] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, 2010.

[20] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[21] OpenCFD. OpenFoam user guide. `http://www.openfoam.com/docs/`, 2011.

[22] D. A. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: declarative, parallel structured grid programming. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 15–24, 2010.

[23] E. Phillips and M. Fatica. Implementing the Himeno Benchmark with CUDA on GPU Clusters. In *IEEE International Parallel & Distributed Processing Symposium*, pages 1–10, Apr. 2010.

[24] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[25] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS Three-Dimensional Fluid Simulation for Fusion Science with HPF on the Earth Simulator. *SC Conference*, 0:51+, 2002.

[26] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In L. BÃűszÃűrmÃľnyi and P. Schojer, editors, *Modular Programming*

Languages, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Berlin / Heidelberg, 2003.

[27] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 tflops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[28] D. Unat, X. Cai, and S. B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated c. In *Proceedings of the International Conference on Supercomputing (ICS'11)*, ICS '11, pages 214–224, 2011.

[29] S. Venkatasubramanian, R. W. Vuduc, and N. None. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM.

[30] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.