2018 21st International Conference of Computer and Information Technology (ICCIT), 21-23 December, 2018

A System for Performance Porting of Iterative Structured Grid Applications in HPC Environments

Ryan Marshall
Department of Computer Science
Tennessee Technological University
Cookeville, TN, USA
rmarshall42@students.tntech.edu

Sheikh K. Ghafoor
Department of Computer Science
Tennessee Technological University
Cookeville, TN, USA
sghafoor@tntech.edu

Md. Bulbul Sharif
Department of Computer Science
Tennessee Technological University
Cookeville, TN, USA
msharif42@students.tntech.edu

***Abstract*—This paper presents a system for the efficient implementation of cellular automata problems on heterogeneous HPC platforms, with separation of the numerical models developed by domain scientists from the low-level details that are typically handled by experienced programmers familiar with MPI+X programming models. We describe a framework designed to reduce development and maintenance time required for new applications, which can also be used to port existing applications to a new HPC environment. We demonstrate how to use the framework to implement cellular automata problems while highlighting the various differences in performance between framework-assisted and native implementations. The evaluation of the framework shows that the performance of application developed using the frame work is comparable to the same application optimized for a specific architecture.**

***Index Terms*—heterogeneous computing, cellular automata, iterative stencil, structured grids, high performance computing**

I. Introduction

Many scientific applications are used to simulate physical processes, including soil erosion, flood inundation, wildfire behavior and atmospheric turbulence. Some applications can be grouped according to their computational patterns. In particular, the group of cellular automata applications that operate on a 2D mesh as iterative stencil computations are important because they can be used to model a range of physical systems studied by scientific researchers all over the world. These applications contain several computational and structural invariants that can be separated from the problem-specific elements and reused for any problem that can be expressed as cellular automata. With the principles of cellular automata in mind, we designed a framework to add a layer of portability, allowing applications to run efficiently on heterogeneous HPC systems using any combination of MPI, OpenMP, OpenCL and CUDA technologies.

Such a framework would be beneficial to a significant portion of the scientific community, as it inherently separates the core algorithms from the parallelization and distribution so that researchers can focus on their domain-specific solutions without having to manage the details of the hardware architecture and execution environment. The framework supports a code base that can perform efficiently on heterogeneous architectures (comparably to a hand-optimized implementation), and allows users to develop applications that are portable enough to run on multiple systems without having to modify their problem-specific code whenever they wish to migrate it to another system. The contributions of this paper include:

- descriptions of a generalized system and proposed framework for cellular automata problems in heterogeneous HPC environments,
- some preliminary performance evaluations of an implementation of the framework in comparison to optimized, native versions.

II. Representative Problem: A 2D Flood Simulator

The flood simulator used for this experiment was developed in [1]–[3], and uses a staggered grid computational stencil to define the domain with the water depth at the center of the cell and x and y directional velocities on the cell edges. The following equations related to Manning's measure for surface roughness [4] are computed repeatedly over time:

$$S_{fx} = n^2(u_{ij})\sqrt{\frac{u^2{}_{ij} + \bar{v}_{ij}^2}{h_{ij} + h_{(i+1)j}}}, \tag{1}$$

$$S_{fy} = n^2(v_{ij})\sqrt{\frac{v^2{}_{ij} + \bar{u}_{ij}^2}{h_{ij} + h_{i(j+1)}}}, \tag{2}$$

where h is the water depth, u is the grid of velocities in the x-direction, v is the grid of velocities in the y-direction, $S_f x$ is the friction slope in the x-direction, $S_f y$ is the friction slope in the y-direction, n is the Manning coefficient, and:

$$\bar{u}_{ij} = \frac{u_{ij} + u_{i(j-1)} + u_{(i-1)(j-1)} + u_{(i-1)j}}{4}, \tag{3}$$

$$\bar{v}_{ij} = \frac{v_{ij} + v_{i(j+1)} + v_{(i+1)(j+1)} + v_{(i+1)j}}{4}. \tag{4}$$

For purposes of this study, we will use a constant n of 0.035. The model uses the upwind finite difference numerical scheme that solves the governing set of shallow water equations as given in [5], which yields non-oscillatory solutions through numerical diffusion [6]. Temporal dependencies for each iteration include the interpolated flow rate that is derived from an input file, where the interpolation function:

$$h(t_r, f_r, s) = f_r + \frac{f_{(r-1)} - f_r}{(s - t_r)(t_{(r-1)} - t_r)} \tag{5}$$

is applied using hydrograph time t and flow f at row r, and current simulation time s. Spatial dependencies for a grid cell include specific cell neighbors of the corresponding cell from the previous time step. By examining Equations 1-4, we can identify the specific dependencies. This model can be implemented as a cellular automata problem, and the new design will have the ability to execute on a range of heterogeneous platforms without requiring modifications to the source code.

III. Framework Design

The proposed framework targets cellular automata problems that have the following characteristics:

1) Each cell in the domain is capable of storing at least one state value from a finite (but possibly very large) set of states.
2) A local transition function can be defined for each cell in the domain. The resulting output of the function will be the new state of the cell.
3) Each cell belongs to a neighborhood (stencil) of cells that can be accessed through local memory.
4) The domain state at time t transitions into a new state at time $t+1$ after each cell finishes execution of its transition function.
5) The evolutions begin and end at times t, z, respectively, where $t <= z$ and a finite number of time steps can be derived between t and z.

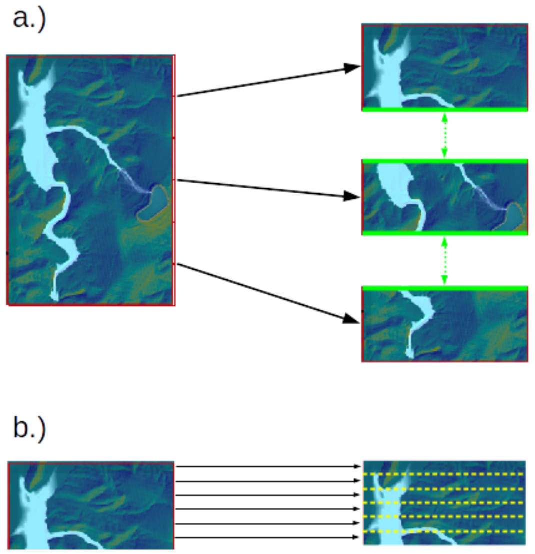

Fig. 1. Partitioning of a flood simulator domain by a.) processes, where edge data communication occurs between process neighbors, and each process by b.) threads in shared memory space.

Generally, these problems can be ported to HPC environments by assigning a cell or small group of cells to a thread and executing the transition function for each cell in each thread, as illustrated in Figure 1b. Large domains can be decomposed and assigned to separate processes that may operate on separate compute nodes; each process executes a smaller version of the overall problem. Because each cell needs read access to every cell value in its neighborhood, cells that reside on the boundaries of a subdomain may need data from cells in an adjacent subdomain. Thus, a series of exchange operations can take place at the beginning of the simulation and at the end of each time step, where each process sends its edge data to its logical neighbors, as shown in Figure 1a.

The framework is designed according to the identifying characteristics of the targeted problem class, recognizing the need for concurrency and domain decomposition. The foundational layer of the framework is realized through a set of elements and their interactions. Control flow begins within a root process

TABLE I
Interface

Name	Description
ID	The unique identifier of the Interface.
Name	The name of the Interface.
Methods	User-defined name, scope, arguments and return value of the Interface's methods.

from which an arbitrary number of child processes can be launched. For design purposes, the root process is called the Launcher, and its child processes are called Components.

1) Launcher: The Launcher is the entry point for all applications built with the framework. The responsibilities of the Launcher are to launch and modify Components, establish the necessary interactions between Components and send signals to Component processes when needed.

2) Component: The encapsulation of related data and communication directives is done via a Component. The characteristics of each Component are defined by the user; they can range from identical to distinct, from redundant to flexible. Each Component is capable of communicating with other Components over an Interface, a set of user-defined permissions and directives.

3) Interface: By default, a Component is equipped with a control Interface (shown in Table II), giving it the ability to install additional, more specialized Interfaces.

Table I lists the attributes of an Interface. To access resources of another Component, the Interface must be used. The example below can be used to install a new Interface to a Component:

```
InterfaceProperties: {
 id: NULL,
 name: "newIface",
 methods[{
   name: "methodOne",
   scope: PUBLIC,
   args: [
     ("id", INTEGER, 123),
     ("dir", STRING, "/tmp"),
     ("maxval", FLOAT, 0.01)
   ],
   return: ("rval", FLOAT, 0.0)
 }, {
   name: "methodTwo",
   scope: PRIVATE,
   args: [
     ("rank", INTEGER, 0),
     ("infile", STRING, "/tmp/file.in")
   ],
   return: ()
 }]
}
```

TABLE II
Control Interface

Name	Description
launchComponent	Create and launch a new Component.
addInterface	Adds a new Interface to a Component. Return value is the Interface ID
addDatastore	Associate a Datastore with one or more Interfaces.
linkInterface	Links two Components via their respective Interfaces.
unlink	break an existing link between two Components, or disassociate a Datastore from an Interface.

4) Datastore: Components can access one or more Datastores to store, modify and retrieve data. There are two types of Datastores: *cached*, which is loaded into local memory and *linked*, where a file handle is maintained. A Datastore must be associated with at least one Component, and can attach to one or more Interfaces within a Component. A Datastore has access control mechanisms, where the user can define access by authorized interfaces only. Concurrent access is controlled via write locks.

The object representation of a Datastore called `topogrid`, associated with Interfaces 1 and 2 can be given as such:

```
DatastoreProperties: {
 datastoreID: 0,
 datastoreType: LINKED,
 datastoreName: "topogrid",
 datastorePerms: [(1, 1, 1), (2, 1, 1)]
 datastoreDescriptor: {
   headerBytes: 4096,
   headerFormat: [(8, 128), (12, 256)],
   bodyBytes: 16777216,
   bodyFormat: [(16384, 1024)]
 }
}
```

Table III gives a list of the other values; it is a linked Datastore, where both associated Interfaces have read and write permissions, the header is 8 lines of 128 bytes plus 12 lines of 256 bytes each, and the body is 16384 lines of 1024 bytes each.

A. Framework Operational Layer

An overview of the framework is given in Figure 2, including the operational layer where the Components are named according to their responsibilities. We used four Components to build the layer- Discover, Partition, Compute and Update.

Applications can sometimes contain a large amount of setup code near the entry point. For example, the

TABLE III
Datastore

Name	Description
ID	A unique index
Type	LINKED for disk file, CACHED for in-memory data.
Name	A human readable name.
Perms	Array of triples (id,r,w), where each triple is an interface on the Component identified by 'id', the 'r' and 'w' are boolean values for read, write permissions, respectively.
Descriptor	Information about how to read the data.
HeaderBytes	The number of bytes in the header.
HeaderFormat	A sequence of ordered pairs (n, z), where 'n' is the number of values and 'z' is the size in bytes of each value.

TABLE IV
Discover Component

Element Name	Element Type	Description
Reader	Interface	File reading
Writer	Interface	File writing
Formats	Datastore	File formats for Reader/Writer
Targets	Datastore	List of available compute devices
Stencils	Datastore	List of compute stencils

file reading process can be cumbersome, or the input functions require the filenames to follow certain naming conventions and expect the data to follow a specific format; to read a file in a different format, the user code may require some modification. To eliminate this burden, all operations related to initialization are grouped into a Component called Discover, including gathering the characteristics of the execution environment and input data, querying the system for installed software, and reading configuration files provided by the user that specify the size of the data, length of simulation, number of simulations, output interval and requested device type.

Many of the simulation control variables are either hard-coded or expected as positional arguments to the executable program. To add, remove or define new parameters, the source code would need to be modified, sometimes affecting many different blocks of code throughout the application. After Discover performs its assigned tasks, it will package the data into an internal representation and pass it to another component called Partition, which is responsible for partitioning and mapping operations. Partition performs validation on the data received from Discover and will attempt to launch multiple processes in a manner requested by the user. This validation process is more rigid than Discover, as its input is equivalent to an intermediate language, but also more robust. Partition will generate an error code if validation fails, and control returns to Discover. If a maximum number of attempts fail, the simulator will gracefully exit. Otherwise, each process launched by Partition will be handled by Compute.

The Compute component contains an implementation of the underlying numerical model. Compute can utilize threads, and is typically called on each cell in the domain, with access to the cell's neighborhood in local memory. The Update component is a hook that is executed after each time step of the simulation. The user can supply a configuration file that specifies operations to be taken at the end of each iteration. For example, Update can calculate the real time interval for next generation or time step, determine whether or not to perform output operations, enforce a synchronization point, or generate some aggregate statistics about the data domain based on its current state.

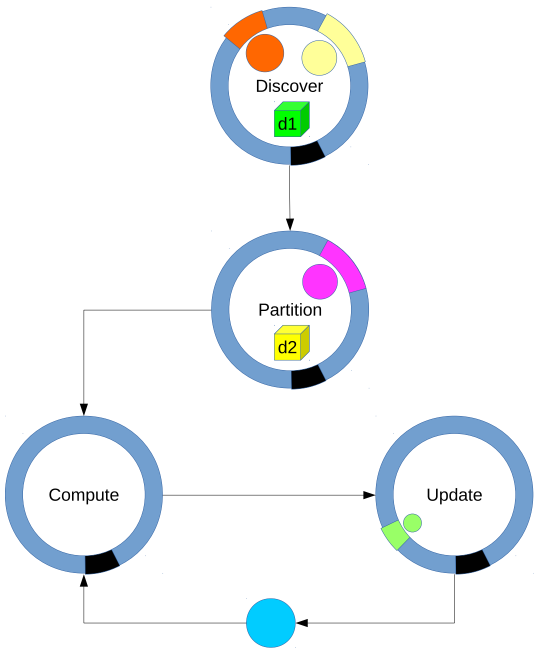

Fig. 2. Overview of the framework, showing its main Components and high-level interactions, their respective Interfaces and Datastores (shown as cubes d1 and d2). The smaller circle at the bottom center is the Launcher, which serves as a synchronization point.

B. Discover

The flood simulator accepts a small range of input files in both ASCII and binary formats. It supports MPI+OpenMP and MPI+CUDA environments, so Discover will check for access to CPU cores and GPU devices as specified by the user. Based on Equations 1-4, a Moore neighborhood is sufficient for local computations.

TABLE V
Partition Component

Element Name	Element Type	Description
SimProps	Interface	Requested simulation parameters
PartMap	Interface	Partition to device mappings
SimData	Datastore	Parameters for all simulations

TABLE VI
Compute Component

Element Name	Element Type	Description
Kernel	Interface	Operations for active kernel
Domain	Datastore	Cell data available to active kernel

C. Partition

After Discover performs its assigned tasks, it will package the data into an internal representation and pass it to another component called Partition, which is responsible for partitioning and mapping operations. Partition performs validation on the data received from Discover and will attempt to launch multiple processes in a manner requested by the user. This validation process is more rigid than Discover, but also more robust. Partition will generate an error code if validation fails, and control returns to Discover. If a maximum number of attempts fail, the simulator will gracefully exit.

D. Compute

The Compute component contains an implementation of the numerical model supplied by the user. Compute can utilize threads, and is typically called on each cell in the domain, with access to the cell's neighborhood in local memory. The flood simulator will execute the kernel using one GPU core per cell when using the GPU, and one CPU core per line when using OpenMP.

E. Update

The Update component is a hook that is executed after each time step of the simulation. The user can supply a configuration file that specifies operations to be taken at the end of each iteration. For example, the output for the flood simulator is specified by a fixed interval over the total iterations, so we add a print function to the list of hooks and a conditional test for Update to perform after each iteration to determine if the interval has been met. If the conditional returns true, Update will call a print function to generate an output file. There is an additional hook required for the flood simulator that executes an implementation of the function given in Equation 5 to compute the flow value for the next iteration.

TABLE VII
Update Component

Element Name	Element Type	Description
Hooks	Interface	Conditional expressions, applied per iteration
Conditions	Datastore	Results of conditionals applied by Hooks

IV. Experimental Setup

The experiments described in this section will compare hand-tuned (HT) implementations to framework-assisted (FA) implementations. The results will show raw performance, computation and communication overheads. Run time in seconds, speedup, and Megacells/sec (Mc/s) will be used to compare the computational efficiency of all the implementations. Higher Mc/s corresponds to higher performances, since more cells are dispensed of in the same time reference window. This measure allows capturing the average speed during the computation and it is particularly useful in case of real domains, where the number of affected cells can significantly be modified along the simulation time. Mc/s (as m) is computed according to the following equation:

$$m = \frac{rki}{t} * 10^6, \tag{6}$$

where r is the number of rows in a 2D grid, k is the number of columns in a 2D grid, i is the total number of iterations, and t is the number of wall seconds it took for the simulation to complete.

A. Environment

For our simulation runs, we used a heterogeneous HPC cluster consisting of 4 nodes, each having Intel Xeon E5-2680 processors in 2 sockets, 10 cores each (with hyper-threading), for a maximum of 160 dedicated CPU processes across all 4 nodes. The heterogeneity is introduced with the GPGPU arrangement of each node, where Node 1 is equipped with a single Nvidia Tesla K40M (K40) and Node 2 with two K20M (K20) GPUs. The GPUs on Nodes 3-4 will not be used for these simulations. Additionally, a desktop machine was used for the portability experiment, equipped with a AMD A-10 6800K APU and two Radeon HD 7750 GPUs.

The applications were written in C, C++, with OpenCL 1.2 and CUDA 9.1 for the NVidia GPU kernels, and were compiled with GNU Compiler Collection version 4.7.2. The MPI implementation was OpenRTE 1.4.5 and the operating system was Debian GNU Linux 7. The software environment for the desktop system uses AMD APP 3.0 on Ubuntu 14.04.5 LTS.

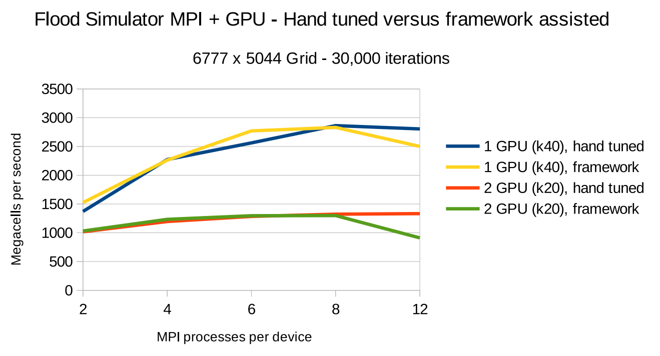

Fig. 3. Comparison of MPI+GPU results for hand-tuned versus framework-assisted versions of a 2D flood simulator using an increasing number of processes and process mappings.

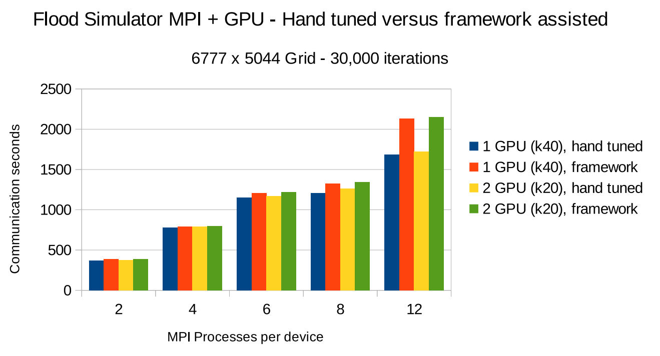

Fig. 4. Comparison of MPI+GPU communication time for four versions of a 2D flood simulator using an increasing number of processes and process mappings.

V. Results and Analysis

Figure 3 shows the results in Mc/s for both the HT and FA simulations. As expected, the single GPU versions perform better than the double GPU versions, due in large part to the communication overhead when exchanging neighborhood data. While the results in general for single GPU and double GPU remain similar between the HT and FA versions up to 8 processes, the FA results drop off for 12 processes. Noting the performance of the original version approaches a constant value when increasing the number of processes per device, the devices experience a communication bottleneck at or around 12 processes, and this effect is more prevalent in the FA version. Communication times are shown in Figure 4. The HT version makes better use of native CUDA functionality that are specific to the execution environment (for example, block and grid sizes are hard-coded for a specific GPU device). Migration to a different environment could decrease performance significantly while performance of the FA version remains relatively unchanged.

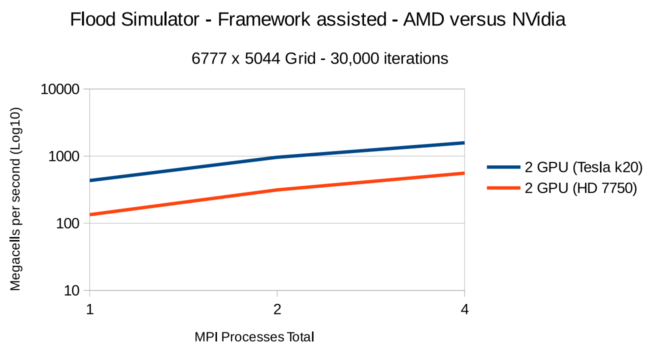

Fig. 5. Performance comparison of a framework assisted 2D flood simulator using MPI+GPU on two different architectures. Where the total processes equal 1, only a single GPU is used; otherwise the processes are equally balanced over 2 GPUs.

In Figure 5, we plotted the Mc/s on a logarithmic scale, since the K20 is nearly an order of magnitude more powerful than the HD 7750 and our interest is mainly in the trend. From this preliminary result we see a common trend in performance increase as the number of processes increase.

VI. Related Works

Several approaches to optimized code generation for stencil problems have been explored [7]–[9], including some C++ libraries.

Thrust is a C++ template library that abstracts and attempts to optimize a group of commonly used CUDA C operations to provide the programmer a way to implement certain parallel algorithms at a high level, benefit from performance optimizations over native code and maintain interoperability with the native CUDA libraries [10]. Thrust uses the Structure of Arrays pattern to ensure regular accesses are able to coalesce. A special iterator is incorporated for the purpose of encapsulation into tuples on a given range via zip function, and is shown to be 2.85 times faster than a comparable Array of Structures implementation.

Kokkos is a C++ library targeted towards many-core devices that offers its users the ability to abstract the parallel dispatch model from user-specific codes, providing a layer over multidimensional arrays that allows the changing of access patterns at compile time [11]. Kokkos supports SIMD operations using `parallel_for` and `parallel_reduce` functors, and gives the user the ability to change layouts from row-major to column-major without modifying the indexing scheme in the user code.

VII. Conclusion

In this paper, we proposed a framework designed for a class of scientific problems that can be represented by cellular automata that can operate portably and efficiently on HPC systems. We have demonstrated the practical application on a multi-node heterogeneous cluster, with performance results comparable to results obtained under a hand tuned version. We also showed that the same framework can operate on different architectures and produce a similar trend in performance. Future work activities in this area include:

- expanding the preliminary experiments to observe more operating environments,
- adding support for adaptive partitioning, where domains can be repartitioned after the initial distribution,
- adding multiscalar processing support for larger datasets,
- adding capabilities to utilize CUDA non-default streams to reduce communication overhead,
- extending support for multiple synchronization points to allow for lookahead and dramatically reduce overall run time,
- preparing a software package for generalized usage within the domain of grid simulations and cellular automata.

References

[1] A. J. Kalyanapu, S. Shankar, E. R. Pardyjak, D. R. Judi, and S. J. Burian, "Assessment of GPU computational enhancement to a 2D flood model," *Environmental Modelling & Software*, vol. 26, no. 8, pp. 1009–1016, aug 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1364815211000582

[2] S. S. Alfred J Kalyanapu, Sheikh K Ghafoor, Ryan J Marshall, Tigstu T Dullo, David R Judi, "Benchmark Exercise for Comparing the Computational Performance of Two-Dimensional Flood Models in CPU, Multi-CPU, and GPU Frameworks," in *World Environmental and Water Resources Congress 2014*, ch. 133, pp. 1322–1331. [Online]. Available: http://ascelibrary.org/doi/abs/10.1061/9780784413548.133

[3] R. Marshall, S. Ghafoor, M. Rogers, A. Kalyanapu, and T. T. Dullo, "Performance evaluation and enhancements of a flood simulator application for heterogeneous hpc environments," *International Journal of Networking and Computing*, vol. 8, no. 2, pp. 387–407, 2018.

[4] R. Manning, J. P. Griffith, T. F. Pigot, and L. F. Vernon-Harcourt, *On the flow of water in open channels and pipes*, 1890.

[5] S. V. Patankar, *Numerical heat transfer and fluid flow*. Taylor & Francis, 1980.

[6] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics*. Springer Berlin, 1996, vol. 3.

[7] T. Muranushi, "Paraiso : An automated tuning framework for explicit solvers of partial differential equations," vol. 5, 04 2012.

[8] N. Maruyama, K. Sato, T. Nomura, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–12.

[9] A. Schäfer and D. Fey, "Libgeodecomp: A grid-enabled library for geometric decomposition codes," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 285–294. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_39

[10] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*. Elsevier, 2011, pp. 359–371.

[11] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Extreme Scaling Workshop (XSW), 2013*. IEEE, 2013, pp. 18–24.