PARTANS

An autotuning framework for stencil computation on multi-GPU systems

PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems

THIBAUT LUTZ, CHRISTIAN FENSCH, and MURRAY COLE, University of Edinburgh

GPGPUs are a powerful and energy-efficient solution for many problems. For higher performance or larger problems, it is necessary to distribute the problem across multiple GPUs, increasing the already high programming complexity.

In this article, we focus on abstracting the complexity of multi-GPU programming for stencil computation. We show that the best strategy depends not only on the stencil operator, problem size, and GPU, but also on the PCI express layout. This adds nonuniform characteristics to a seemingly homogeneous setup, causing up to 23% performance loss. We address this issue with an autotuner that optimizes the distribution across multiple GPUs.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *Design Studies*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: GPGPU, multi GPU, optimization, stencil computation

ACM Reference Format:
Lutz, T., Fensch, C., and Cole, M. 2013. PARTANS: An autotuning framework for stencil computation on multi-GPU systems. ACM Trans. Architec. Code Optim. 9, 4, Article 59 (January 2013), 24 pages.
DOI = 10.1145/2400682.2400718 http://doi.acm.org/10.1145/2400682.2400718

1. INTRODUCTION

GPGPUs have emerged as powerful and energy-efficient compute devices for many kinds of problems. To improve the overall compute power of a system, vendors have started to offer systems with multiple GPUs, such as the TYAN FT72B7015 (8 GPUs) or the Dell PowerEdge C410x (16 GPUs). These systems reduce the computation time or allow the computation of larger problem sizes. However, exploiting the full compute capability of such systems efficiently presents a difficult programming challenge. Ensuring efficiency in a single GPU system is already complicated due to the complex memory model. Doing so for multiple GPUs, with data distribution and synchronization decisions critically impacting performance, requires additional effort of the programmer. This makes automation an attractive option.

In this article, we focus on the abstraction and optimization of multi-GPU programming for a particular class of compute pattern: *stencil computation*. Stencil computation is a computation pattern on an n-dimensional volume, where each point is updated (usually iteratively) as a function of its neighboring elements. This pattern is found in many application domains, such a quantum physics, weather prediction, and image processing. We present a framework that automatically distributes and optimizes such

This work is supported by the Engineering and Physical Sciences Research Council (EPSRC). This is a new article, not an extension of a conference paper.
Authors' addresses: T. Lutz (corresponding author), C. Fensch, M. Cole, School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom; email: {thibaut.lutz, c.fensch}@ed.ac.uk.

DOI 10.1145/2400682.2400718 http://doi.acm.org/10.1145/2400682.2400718

computations across multiple GPUs. We achieve a high level of abstraction, completely hiding the underlying GPU programming and optimization, leaving the user to focus only on the problem-specific implementation.

We evaluate our framework using an extensive range of stencil computations on different GPUs (NVidia GTX 590 and AMD Radeon HD 5970 dual GPU graphics cards) and different motherboard chipsets (Intel P67 and Intel X79). We extend the applicability of our results by additionally experimenting with synthesized application variants, in which the granularity of the local GPU computations is under our control. This enables us to investigate the impact of future improvements in single GPU performance and optimization.

The contributions of this article are as follows.

—We perform an exhaustive evaluation of the large optimization space. This enables us to understand the importance and interaction of various application and machine characteristics, and to measure the performance and overheads of our optimization strategies against those of an idealized oracle. Our investigation of the optimization space shows that the best strategy depends not only on characteristics of the stencils employed in an application, the problem size, and the GPU specification, but also on the detailed configuration of the connection mechanism (in our systems, PCI express). This aspect of our work is completely novel. We discover that the PCI configuration adds nonuniform characteristics to a system that otherwise seems homogeneous.
—We develop a family of autotuning heuristics, which tune the number and configuration of GPUs, data distribution, and exchange strategies, in ways that are sensitive to the characteristics of the underlying communication configuration. No previous work has attempted to account for these factors systematically. Our heuristics achieve a very high proportion of the optimal possible performance, often to within 5%. Our analysis carefully examines the contributions made by both offline and online tuning components.

The rest of this article is structured as follows: Section 2 gives a concise overview of GPGPU technology, stencil computation, and PCI express communication. In Section 3, we present our framework and the issues involved in optimizing stencil computations on multiple GPUs. Section 4 lists our objectives, while Section 5 gives an overview of the selected applications and architectures. In Section 6, we present our analysis of the optimization space and describe our autotuning mechanism. Section 7 presents related work and we conclude in Section 8.

2. BACKGROUND

2.1. GPGPU Technology

Advances in graphics processing units (GPUs) have provided graphics programmers with increasing flexibility in algorithm design. This flexibility has increased to such an extent that it gave birth to the field of General-Purpose computation on Graphics Processing Units (GPGPU). Originally, each GPU vendor developed its own propriety programming interfaces for its products (e.g., C for CUDA [NVIDIA 2012] or the ATI Stream SDK [AMD 2012]). Later, the OpenCL standard has been proposed to provide a unified programming interface [Khronos Group 2008]. It is the combination of a programming language (very similar to C99) and a runtime environment. Similar to C, OpenCL provides quite a low-level programming model [Khronos Group 2011].

OpenCL is not designed as a general-purpose programming language, but focuses on compute-intensive or data parallel kernels. The OpenCL C programming language is used to describe these kernels. Due to the vast difference in supported hardware platforms and the fact that the actual hardware platform is only known at runtime,

OpenCL uses a runtime compilation approach. The kernels are then dispatched by the runtime system to an OpenCL device, where the kernel executes asynchronously from the host program.

One particular interesting feature of OpenCL is its support for dependencies between tasks. A task in OpenCL can either be a compute operation, or a data transfer between the host and the GPU. For each task, OpenCL allows to specify tasks that need to be completed before the task can be started. In particular, in a multi-GPU system a task can depend on a task that has been issued to a different GPU. This greatly simplifies the programming effort that is required to synchronize execution on multiple GPUs. It also simplifies the overlapping of communication and computation, as the GPU will execute as many tasks in parallel as its hardware is capable of, provided that these tasks are ready for execution.

2.2. PCI Express

The PCI Express bus (PCIe) is currently the de facto standard to connect expansion cards (such as GPUs) to the CPU and memory in a system. However unlike its predecessor PCI and as its name seems to imply, PCIe is not a shared parallel bus architecture. Instead, it uses a star shaped topology with point-to-point links that connect devices to the PCIe root complex [PCI-SIG 2010]. Each link is composed from 1 to 32 full duplex PCIe lanes, determining the bandwidth of the link or slot. The number of lanes is usually given as a factor when one refers to the slot, e.g., PCIe x8 slot or PCIe x16 slot. However, this is not the only information required to reason about the available bandwidth to communicate with a GPU. Most systems have multiple PCIe root complexes, for example one in the chipset and one in the integrated I/O hub of processor. These hubs are then linked by other means, for example QPI (Quick Path Interconnect) or DMI (Direct Media Interface). Finally, motherboard and graphics card manufactures might use PCIe multiplexers or switches to increase the number of devices that can be physically connected to one PCIe link. All these factors introduce nonuniformities into the system. Thus, a quad GPU system with four identical graphics cards might not behave as homogeneously as one could expect. An example of such a complex system is the TYAN FT72B7015: it utilizes two Intel 5520 Southbridge chips each with an I/O hub that supports two PCIe x16 links. These links are then doubled using PCIe switches to support up to 8 GPUs in the system.

As another example, one of our experimental systems uses an ASUS Maximus IV motherboard with Intel's P67 chipset. For a dual GPU configuration, the board supports two configurations: two PCIe x8 links that connect directly to I/O hub in the processor; or, two PCIe x16 links that connect via an Nvidia NF200 PCIe multiplexer. During our research, we found it impossible to get any detailed information about the capabilities of these PCIe switches. Information is only available to OEMs after signing nondisclosure agreements.

In this article, we will show that while these switches are transparent to the user from a logical point of view, they cannot be ignored when one is aiming for optimum system performance.

2.3. Stencil Computation

Stencil computations arise naturally in a wide range of application areas, predominantly within computational science and image processing. They form a constrained subclass of the geometric decomposition pattern recognized by Keutzer et al. [2010] and Mattson et al. [2004]. Stencil computations share characteristics that constrain both the data involved and the control flows that may be applied. With respect to data, the primary data structure is multidimensional and regular, possibly with toroidal wrap-around. We call this data structure the *volume*. Each point of the volume stores

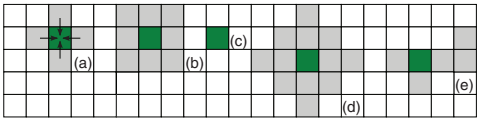


Fig. 1. A fundamental property of stencil computation is that data is drawn into the origin of the stencil, as shown by the arrows in (a). Thus, the only element that is written to is the element at the origin of the stencil (shown in green/dark gray). (a)–(e) show possible stencil shapes.

data in a number of arbitrary strongly typed *fields*. With respect to control flow, processing of the volume takes the form of a sequence of applications of *stencil operations*. A stencil operation updates the values of each point as a function of its value and the values at nearby points. The points considered "nearby" are defined by a *stencil shape*, which is uniform across all points for which it is well-defined. Figure 1 illustrates a selection of stencil shapes. The computation performed at each point is uniform and position independent (except perhaps at boundaries) but may be data dependent. The computation at a point is not permitted to update values at any other points. As shown in Figure 1(a), the computation is only allowed to draw in data from its neighbors.

There are several different types of stencil computation. In the simplest case, a single stencil is applied to the volume once, e.g., edge detection in image processing. Some applications use a composition of several stencils operations that are applied in sequence, e.g., a sequence of image processing filters. More complex applications apply a stencil (or a sequence of stencils) repeatedly. This is either done for a predefined number of steps or until a certain convergence criteria is met. The first case is often associated with simulations that run for a number of time steps. The second case is common for iterative solvers, e.g., for partial differential equations.

2.4. Multiple GPU Computation

Multi-GPU systems are becoming increasingly popular. Innovations include the development of scan-line interleave (SLI) technologies, the development of graphics cards with several GPU chips on a single printed circuit board (PCB), and the integration of GPUs into processor chips used in addition of an external graphics card. Motherboard designs have also evolved to accommodate more PCIe slots to host several graphics cards. PCIe extension devices allow users to increase this number even further. These innovations put more and more strain on parts of the system that were not considered to be a bottleneck in the past. Multiple cards plugged on the same slots through an extension or multiple GPUs on a single PCB all compete for communication over the same bus.

Using more than one GPU for computation brings several benefits. It increases the throughput of the system and can enable computation to be performed faster as long as the distribution overheads are kept low. Furthermore, whereas memory available on a single device is severely limited, which in turn limits the input size of the application, using multiple devices allows much larger problem sizes to be computed. A simple scheme to process a large problem size works by splitting the input into chunks that are processed sequentially on the GPU. However, this approach includes a significant data transfer overhead as for each chunk the input has to be copied to the GPU and updated buffers copied back. By using multiple GPUs, it becomes possible for each chunk to reside on a GPU throughout the whole computation.

3. THE PARTANS FRAMEWORK

We have designed a framework for stencil computation that abstracts away all of the low-level details that are required for multiple GPU programming. We have implemented our framework as a template library in C++. Using C++ provides easy access

```
1  PARTANS_STRUCT(Tuple,
2      float f1; float f2;
3  );
4
5  typedef PARTANS_FIELDS((Tuple,  a),
6                         (Tuple,  b),
7                         (float,  err)
8  ) TupFields;
9
10 Volume<TupFields> *volume =
11     new Volume<TupFields>(_x, _y);
12
13 volume->get<TupFields::a>(0,0)->f1=0.5;
14
15 Tuple *t = volume->get<TupFields::b>(0,0);
```

Fig. 2. Example of volume declaration and interaction. Lines 1 to 3 define a structure, which will be visible on both the host program and the OpenCL source code. Lines 5 to 8 define the type of each element of the volume. In this case each element has two Tuples and a float. Lines 10 and 11 create a two-dimensional volume of dimensions _x and _y. Line 13 and 15 show examples of element accesses, which are type safe.

to the OpenCL runtime, which is provided in C, and enforces type safety, as opposed to the `void*` style used in the OpenCL runtime library. In addition, the template mechanism turns many customizations into compile-time decisions, as opposed to runtime overhead.

In this section, we describe the high-level API, give an overview of the internal implementation strategy, and present an overview of the tuning possibilities exposed by this design.

3.1. API Concepts

The main classes used by the user are *Volume* and *Stencil*. The *Volume* class provides a homogeneous Cartesian *n*-dimensional space. Each point in the space stores a heterogeneous collection of elements called *fields*. A field can be a primitive data type or a complex structure. The user can interact with the volume by setting and getting the value of a particular field at a given coordinate. Our framework guarantees type coherency between the host and the device by making these accesses type safe.

Figure 2 shows the declaration of a volume. In this case, the user defines a structure using a macro that guarantees the structure will be declared in both the host C++ code and the OpenCL C code.

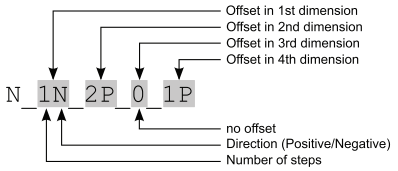
In line 5–8, the user specifies the volume composition. This defines how many attributes each point in the volume has and their types. An optional name can be associated to a field to increase readability of the code. Here, each point of the volume will be a composition of two user-defined structures and a float.

In lines 10 and 11, the volume is created using the field descriptor and information about the dimensions. Note that the user is not required to specify memory layout, the number of cuts, or the locality of the elements. Our framework will automatically choose a memory layout for the volume (for instance array of structure or structure of array) and a volume decomposition strategy depending on configuration parameters and number of devices available.

Stencil objects capture the operations that are applied across *Volumes*, as shown in Figure 3. A stencil consists of a simple element function written in OpenCL C, shown in Figure 3(a). We provide special macros to the programmer that are used to access neighboring points relative to the current position. Figure 3(b) explains these macros. Figure 3(c) shows the final definition of a stencil operation. In addition to the OpenCL C element function, the user also has to specify the fields that are read or updated by the stencil element function. Note that a stencil can either just be a single function, or (as in the example in Figure 3(c)) a sequential composition of functors. Composite stencil objects simplify the expression of complex computations involving several computation phases, each with a particular sequence of functors.

```
void mult(__global const Tuple a[],
          __global       Tuple b[],
          __global       float err[]){
    err[ORIGIN] = b[ORIGIN];
    b[ORIGIN].f1 =
        (a[N_1P_0].f1+a[N_1N_0].f1+
         a[N_0_1P].f1+a[N_0_1N].f1-
         b[ORIGIN].f1)/4.;
    err[ORIGIN] -= b[ORIGIN];
}
```

(a) Stencil Operator



(b) Neighbor macro definition

```
StencilWrapper wrapper(volume);

Stencil<TupFields> stencil =
     wrapper.newCompositeStencil();

stencil.addStencil(
         cl_code, // element function
         "mult",  // entry point
         // list of readonly fields
         { Tuple::a },
         // updated fields
         { Tuple::b, Tuple::err });

stencil.addStencil(
         cl_code,
         "mult",
         { Tuple::b },
         { Tuple::a, Tuple::err });

// apply 100 iterations
stencil(100);
```

(c) Stencil object usage

Fig. 3. Example of a simple stencil computation that performs a Jacobi operator on one Tuple and stores the variation in the third field. The stencil operator, shown in (a), can be defined directly in the host code using a special macro or in a standalone text file. It uses special macros described in (b) to access the neighbors.

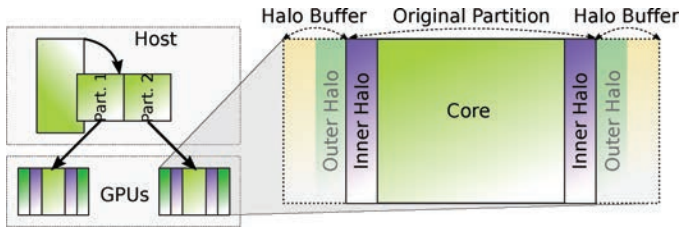


Fig. 4. Example of decomposition for a 2D volume. The volume is transformed if necessary, then cut into smaller chunk. Each chunk is mapped to a GPU buffer, which is large enough to accommodate copies of the boundary regions of the neighboring partitions, called outer halos. Their counterpart, the inner halos, represent the neighbor dependencies. The halo buffers allow halos to dynamically change size within the preallocated buffers.

3.2. Internal implementation strategy

Figure 4 shows how a volume is transformed and partitioned to be mapped to several GPUs. In order to distribute a stencil computation across multiple GPUs, the volume is split into partitions, which are then assigned to different devices. However, applying the stencil function to elements that are at the border of the cut requires access to elements that belong to a different partition. As this partition is stored on a different GPU, accessing these elements directly is not advisable. Instead, it is common practice to create a copy of these elements at the border of each partition, called an *outer halo*. The elements in each partition that are used as the outer halo of another partition are called the *inner halo*. As the size of the inner and outer halo is the same, we simply refer to it as *halo size*.

In a very naïve implementation, the outer halo only contains the minimum number of elements required to apply the stencil function. After the stencil function has been applied, the outer halos need to be updated from the inner halos. As this approach requires a significant number of data communications, it is generally a better idea to allocate more elements in the halo than necessary. This allows the stencil function to be applied multiple times before the outer halo needs to be updated. Figure 5 illustrates this process. Each iteration of the stencil consumes part of the outer halo, as the edges

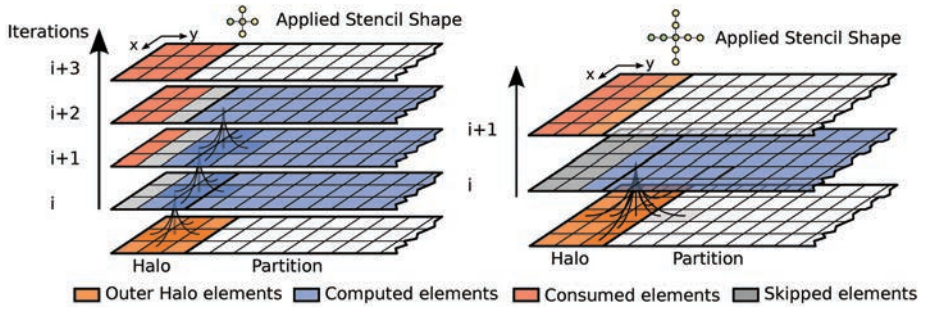


Fig. 5. Illustration of halo consumption. Each iteration consumes as much halo as the size of the stencil shape. In the first case a stencil of size one is applied on a partition with a halo size three, so three iterations can be computed. In the second case, the stencil shape is of size two. Thus, only one iteration can be computed, leaving some unconsumed halo, but not enough for another iteration.

cannot be correctly computed. This data must be renewed when it runs out or becomes too small to satisfy dependencies for the current operator. Furthermore, this introduces redundant computation, because they effectively belong to another partition, which is computing them as well. As a consequence, there is a trade-off between communication cost (which, for a GPU, consists of both the time to read and write these regions plus the latency introduced by initiating these operations) and computation redundancy.

The best halo size, which optimizes the trade-off between communication cost and redundant computation, is a delicate balance that might evolve during the course of program execution. For example, some computations have several distinct compute phases involving radically different operators, leading to a situation in which the best halo size is different for each phase.

To allow more plasticity for the size of halo regions, the allocated size is as large as possible, up to full device duplication if the memory available on the devices allows it. This enables the size of the halo regions to be changed easily and avoids a new buffer creation and a copy when the halo size changes. This large buffer is referred to as *buffer region*. The *outer halo* is part of this *buffer region* and its size can vary from the minimal size defined by the halo shape to the size of the entire buffer.

Device coordination is achieved using a task graph generated by the framework. The host program typically keeps well ahead of the GPUs by generating only the task graph without waiting for completion of the operations. This allows dependencies to be computed in advance, feeding the operations to the GPUs as soon as the dependencies are satisfied, thus keeping it as busy as possible.

Full synchronization between the host and the devices only happens when the user wants to interact with individual volume elements between iterations. Otherwise the state of the host buffers and the various regions of interest are managed by a set of flags updated by the task graph, keeping track notably of the current unconsumed halo size or if the updated version of a field resides on the GPU or the host.

3.3. Optimization Space

Volume decomposition introduces a lot of communication, which should be reduced as much as possible to shift the balance between swap frequency and redundant computations.

In order to reduce the transfer time, a first set of transformations is transparently performed during the volume decomposition. The volume can be rotated, either to allow the area of the cut to be minimized by aligning the largest dimension as the first dimension, or to decrease data dependencies in the case of irregular shaped stencil

operators. Each partition can also be transformed independently before being mapped to the device in order to optimize data layout on a more local scale.

The second important optimization concerns the halo size. As noted above, the best choice can vary dynamically at execution time. Using the *halo region*, a search for the optimal halo size can be performed dynamically for each phase, without the overhead of reallocating and initializing buffers. We make halo size uniform across all partitions to guarantee that they will synchronize after the same number of iterations.

Finally, tweaking the graph allows different swapping strategies, giving more or less priority to computing the halo regions before the core, at the expense of generating a more complicated graph. The naive scheme consists of computing the whole coherent region, composed of the partition plus the valid halo regions, until we consume the halo entirely and pausing computation during the swap. A more advanced scheme computes only the halo regions and their inner counterpart and starts the swapping process while the core of the partition, which now is free of dependency, can be computed in parallel to amortize communication.

4. GOALS OF THIS WORK

The area of autotuning stencil computations on GPU-enabled systems is wide and challenging. In this work we investigate the specific issues raised by the presence of multiple GPUs within the same node, and of the impact of the system communication structure upon the optimization space. Our goals are as follows.

—We explore the communication/redundant computation trade-off for a range of applications and evaluate the impact of application characteristics on this tuning parameter.
—We understand, by experimental exploration, the shape of the optimization space generated by these factors for representative stencil applications. In order to address this rigorously, we generate synthetic, highly parameterizable kernels, informed by our analysis of a range of real examples. This makes our work robust in the face of future, and essentially orthogonal, improvements in the areas of single GPU autotuning and hardware performance, and also with respect to future variation in multi-GPU communication/computation ratios.
—We devise an autotuning heuristic, informed by our initial search space exploration, which is capable of selecting high-performing settings for the various tuning parameters relevant to our selected factors. Specifically, the heuristic will automatically determine the number of GPUs to use, the configuration of these GPUs with respect to the underlying system architecture, and the halo size.

The search space is already complicated and too large to be fully explored in a naive fashion. However the potential gain is important, so tuning mechanisms should be portable and adaptive. Furthermore, GPU setups are getting increasingly heterogeneous: some are on the same chip as the CPU, some share a single card, and future devices might communicate in a peer to peer fashion, which means that this optimization space for multiple device synchronization and coordination will explode.

5. SETUP

5.1. Benchmarks

Real-world stencils involve highly diverse dimensionalities, shapes, and volumes. As our focus is on communication optimization, we chose a set of six applications which all apply stencils iteratively, meaning that halo swapping is necessary to complete the computation.

Table I. Summary of Benchmark Characteristics

	Dimensions	Stencil points	Fields	Reads	Writes	Flops/Point
Game of Life	2D	9	2	9	1	n/a
Reverse Edge	2D	5	3	5	1	5
Hyperthermia	3D	6	11	17	1	16
Himeno	3D	27	15	25	2	33
Swim	2D	9	13	28	11	63
Tricubic	3D	64	5	64	1	132

Note: Game Of Life uses only bit masking on integer type.

The net impact of communication optimizations on a whole application evaluation is affected by the performance of the computational phases between synchronizations. Faster execution of computation magnifies the relative importance of fast communication. In the context of GPU implementation of stencils, faster computation phases could arise from local computational optimizations, or simply from larger or more powerful GPUs. For example, speedups of orders of magnitude have been reported in the literature using single GPU optimization strategies, and compute power of GPUs currently increases at a faster rate than their communication capabilities[1].

In order to extend the value of our analysis against such developments we extended our benchmark suite with virtual variants, replacing the actual computation by synthetic work for which we can accurately control the granularity. This enables us to explore a fuller space, with kernel execution time ranging from that of the original application, to faster versions that may emerge as a result of the preceding trends.

Our benchmark suite consists of the following programs.

—*Game of life* is a cellular automaton implementing Conway's Game of Life. Each iteration represents a generation. The application uses double buffering, meaning that each element of the volume is composed of two chars.
—*ReverseEdge* is an image processing application using a Jacobi operator. A one-way function, edge detect, is applied on an image, and a Jacobi operator is used to approximate the original image.
—*Swim* is a fluid dynamics kernel used for weather prediction. We adapted it from the SPEC OMP 2001 benchmark suite. The fields are composed of complex structures and there are three different operators involved in the computation.
—*Himeno* is a fluid dynamics application using a Jacobi-variant converging stencil, which uses complex fields to represent the main operator of the Poisson equation used in mechanical engineering and theoretical physics.
—*Hyperthermia* is a simulation of the temperature diffusion in human bodies during hyperthermia cancer treatment.
—*Tricubic* is a 64-point stencil used in numerical analysis for tri-cubic interpolation.

Table I presents the stencil characteristics of each application. The operators are defined by the dimensionality of the volume, the stencil shape, the number of fields (total, read from and updated) and the number of floating-point operations per point. The boundary condition has been set to wrap-around for all the benchmarks, which forces halo consumption on both sides of the partition. A simpler boundary policy, like Dirichlet boundaries, effectively halves the communication in the case of two partitions, as there is only a single dependency instead of two.

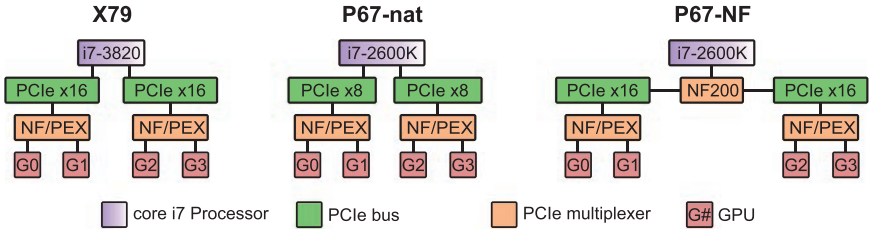


Fig. 6. Overview of our evaluation systems. The P67 system offers two different layouts for dual GPU configuration: either two PCIe x8 in native mode or two PCIe x16 via a NF200 multiplexer. Both the Nvidia GTX 590 and AMD Radeon 5970 are dual GPUs graphics cards that include a PCIe multiplexer on their PCBs.

5.2. Architectures

In this section we give a description of the hardware used to carry out our experiments. We are mainly interested in the PCIe layout rather than the compute power characteristics of the GPUs. This layout is defined by the motherboard chipset and the graphics cards design.

We evaluate three different systems. Our first system uses an Intel core i7-3820 CPU, which supports two native PCIe x16 slots, and the Intel X79 motherboard chipset. The system runs Linux with a 3.1.10 kernel and we refer to this system as *X79*. The second system uses an Intel core i7-2600K CPU, which supports two native PCIe x8 slots, and the Intel P67 motherboard chipset. The system runs Linux with a 2.6.37.6 kernel and we refer to this system as *P67-nat*. The third system is identical to the second system, except that we use 2 PCIe x16 slots that are connected to the core i7 processor via an Nvidia NF200 PCIe multiplexer. We refer to this system as *P67-NF*.

The tested graphics cards are two dual GPU cards. The AMD Radeon 5970 has two GPUs using a TeraScale 2 architecture sharing a single PCIe slot through a PEX multiplexer. We use the AMD APP SDK 2.6 for our experiments. The Nvidia GTX 590 also has two GPUs based on the Fermi architecture and uses an Nvidia NF200 multiplexer. We use the OpenCL runtime provided by CUDA SDK 4.1.28.

Figure 6 shows the three different configuration types used in this article. The bottom multiplexers are found on the PCB of the Nvidia and AMD graphics cards. In total, we have six different setups to evaluate. We did not overclock any processor or graphics card in our experiments.

When distributing data to two GPUs, our systems offer two possible configurations. The data can be assigned to GPUs on the same PCB (e.g., G0 and G1 in Figure 6). We refer to this configuration as *Single Card*. Alternatively, the data can be assigned to GPUs on different PCBs (e.g., G0 and G2). We refer to this configuration as *Dual Cards*. Unlike CUDA, OpenCL currently does not support direct device-to-device copy, thus all the communication is centralized and has to go through the host memory.

When using all the available GPUs, there is a choice to be made on data placement. Figure 7 illustrates the two strategies: blocking placement or circular placement. Blocking placement first assigns partitions to one dual GPU, and then moves on to the next one. We refer to this configuration as *Blocking*. Circular placement assigns partitions in an alternating way. We refer to this configuration as *Circular*. The figure also shows the difference between the dependencies. The arrows in this case indicate the dependencies between the partitions, not the communication.

[1]In the same time that PCIe bandwidth has doubled (PCIe 2.0 to PCIe 3.0), the average compute power of GPUs has tripled.

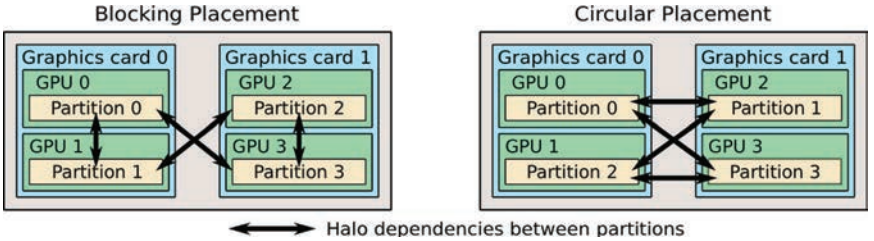


Fig. 7. Different partition dispatching strategies for a four GPU setup. Currently, OpenCL does not support direct device-to-device communication, so the arrows represent only semantic dependencies, but all communication has to go through the host via the main PCIe buses.

5.3. Experimental Methodology

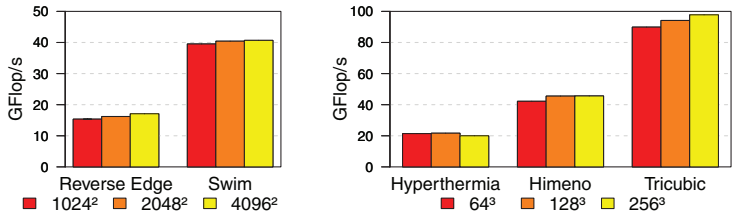
In all experiments, we measure application runtime using wall-clock timers in the host program. We time only the main stencil loop for 10,000 iterations and use this to compute average time for one iteration. Because the GPU measurements can be noisy, due to frequency scaling or shared resource utilization, we use multiple sampling techniques to ensure the stability of the results. For each sample, the main iterative stencil loop is applied eight times in a row with the same number of iterations. If the variance of the interquartile range is below a threshold, the sample is considered successful and the truncated mean of the set is stored. When exploring a parameter range, the space is sampled in random order until five samples are accumulated for each point. This guarantees sampling of similar parameters will be scattered in time to reduce temporal variation of the GPU performance due to external parameters. The final result is the median value of the five interquartile means.

6. RESULTS

We now present the results of our extensive experimental exploration of the implementation and autotuning space.

6.1. Overview

This brief summary highlights findings which are discussed in greater detail in the subsequent sections. We begin in Section 6.2 by investigating the absolute performance obtained by our single GPU implementation which acts as a baseline for our subsequent speedup results. We show that this is competitive with previously published work on similar applications and devices. For example, our simple implementation obtains around 70% of the performance of the highly tuned PATUS system. Turning to our multi-GPU implementations, Section 6.3 shows that the relationship between halo size and performance has a regular bitonic form across all cases, with an identifiable sweet spot, whose precise location varies with application and system. Similarly, we find that problem size has a relatively simple relationship to performance (larger problems have greater potential), while halo shapes which go beyond trivial nearest neighbor can produce less intuitive effects. Finding the correct settings can result in improvements in speedup of the order of 50%, compared to other points which are quite close in the optimization space. In Section 6.4, we explore the impact of the underlying communication technology on otherwise homogeneous systems. For situations in which a subset of devices are selected, we find variations in speed up of up to 33% in regions of the search space at or close to the optimal settings for halo size, and we find that the correct selection of devices can vary with problem granularity with a performance improvement of up to 13% at stake. For experiments in which all four available GPUs are used, we find that optimal allocation of partitions to devices is dependent upon halo size



(a) 2D Stencils (b) 3D Stencils

Fig. 8. Absolute single GPU performance on an Nvidia GTX 590 in GFlop/s for different volume sizes. The graph also shows 99% confidence intervals, however these are so small that they are almost invisible.

and compute granularity, with discrepancies of up to 21% and 13% in favor of *Blocking* and *Circular* respectively, at different points in the space. These findings all serve to emphasize the challenge involved in the full autotuning problem. In Section 6.5, we evaluate our autotuning strategies. These behave impressively, typically obtaining over 90% of the performance improvements found by an exhaustive search of the space.

6.2. Single GPU performance

Figure 8 presents the raw performance for each application using a single GPU on the GTX 590 card. These results are comparable to Phillips and Fatica [2010], who investigated implementation and optimization of the Himeno benchmark on Nvidia Tesla C1060, and Christen et al. [2011] who implemented Hyperthermia and Tricubic benchmarks using their own PATUS framework on an Nvidia Tesla C2050. In direct comparison to the Tesla C2050[2], our naïve implementation achieves about 70% of the performance of the highly optimized PATUS implementation. We did not try to close this gap any further, as the focus of this article is multiple GPU optimizations.

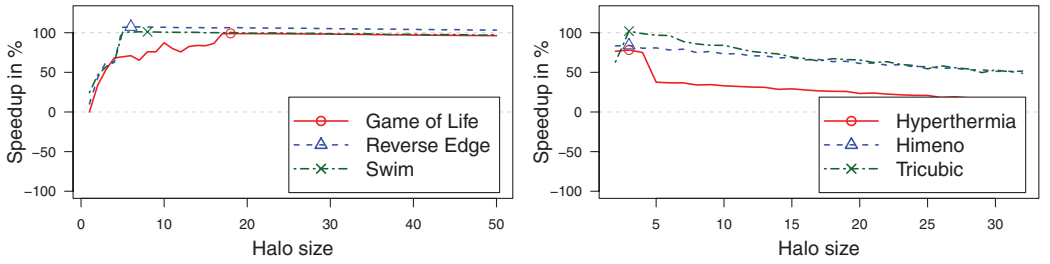
The figure also shows that raw performance for each application is not constant across volume sizes. In order to point out trends and application-independent observations, we use speedup over single GPU performance instead of raw performance in our evaluations.

6.3. Halo Size Impact

In Section 3.2, we described how varying the halo size affects the balance between data communication and redundant computation. We now explore which parameters have an impact on the optimal halo size. In this section, we use the P67-NF system with one GTX 590 dual graphics card. We report the speedup of this system compared to one that only uses a single GPU of the GTX 590 card.

Figure 9 shows the impact of the halo size for several applications in two- and three-dimensional space. Both spaces have the same number of grid points (2^{24}) spanning a square or cube, respectively. We observe for all curves two distinct stages: an ascending and a descending phase. When the halo size is small, the swapping frequency is increased and the latency induced by initiating the swap plus the communication cost itself cannot be hidden by the computation of the core. As the halo size increases, the swapping frequency decreases, leading to less overhead and thus an increase in performance. With increasing halo size, the amount of computation also increases to compute the elements in the outer halo, leading to a steady slowdown. This effect is more noticeable for 3D applications (see Figure 9(b)), because each increment in halo size adds an additional plane that needs computed, as opposed to a vector for 2D problems. The sweet point between the two phases is the optimal halo size. We have

[2]The Tesla uses the same GF110 graphics processor as the GTX 590, however it is clocked about 5% slower.



(a) 2D Stencils - problem size: 4096 × 4096 (b) 3D Stencils - problem size: 256 × 256 × 256

Fig. 9. Impact of halo size on performance. The problem size is in all programs is 2^{24} grid points and is distributed across 2 GPUs. On each graph, we mark the optimal halo size that obtains the best performance.

marked this spot in the graphs. In general, we notice that the dimensionality of the application affects the basic shape of the halo performance curve and location of the best halo. The benchmark *Game of Life* has an optimal halo much larger than the other 2D applications. This benchmark uses 8-bit characters as its main volume elements, requiring much less space and allowing faster exchange.

In our second investigation, we evaluate the impact of the problem size on the optimal halo size. Figure 10 shows that the optimization space becomes more complex as the input size varies. The speedup is around 100% for large problem sizes in all applications. However for smaller problem sizes, the scalability is not as good. The smallest problem size (1024 × 1024) for 2D applications only has a speedup of 27% on average. The curve only shows the increasing phase, indicating that the bottleneck is still communication despite very large halo regions. 3D applications show an even weaker scalability for small problem sizes, as communication is even more expensive. Furthermore, for a cubic input of size 64, the maximum halo possible is too small for the communication to be amortized.

Tricubic shows an interesting effect of the stencil shape. All other applications have stencil shapes that access only the nearest neighbor. However, *Tricubic* also accesses neighboring points with a distance of two. Thus increasing the halo size from an even number to an odd number results in an increase of redundant computation, but still requires the data exchange to happen at the same frequency as before. This pattern is particularly visible for problem sizes of 64^3 and 128^3.

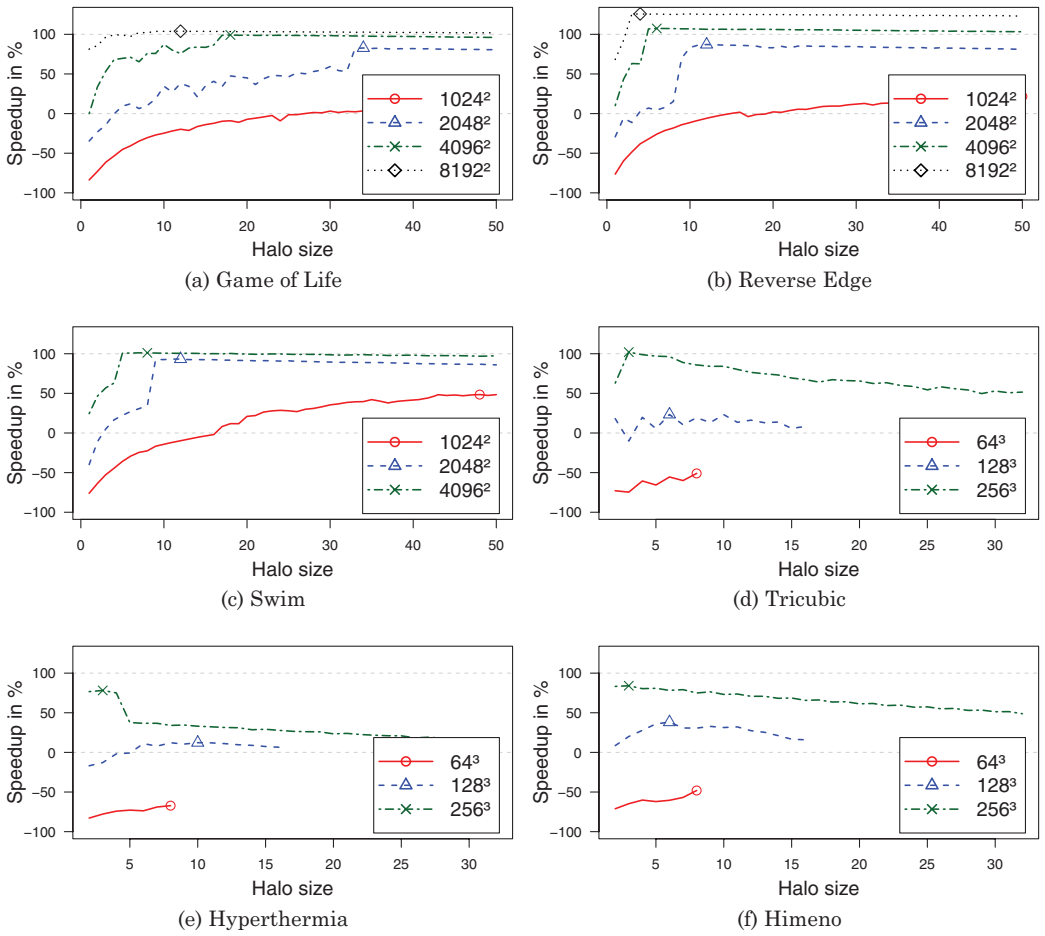
Figure 11 shows the impact of the hardware performance on both the halo size and the scalability for a selected range of applications. The overall performance of the Radeon card is worse in terms of compute power and bandwidth. The sweet spot that balances computation and communication is harder to find. Only approximating it leads to significant performance degradation.

6.4. Data Placement and PCIe Layout

In this section, we experimentally explore the influence of the mapping of volume partitions to GPUs in seemingly homogeneous systems. We demonstrate that the underlying PCIe layout has an influence on the performance in various aspects, including the optimal halo size and scalability. Therefore, it is an important tuning parameter.

We will first explore the impact of the device choice for applications that use a subset of available GPUs and then measure the impact of data locality when using all available resources.

6.4.1. Dual GPU Exploration. Utilizing all the GPUs available is not necessarily the optimal solution. For example, the problem size may be too small to be decomposed

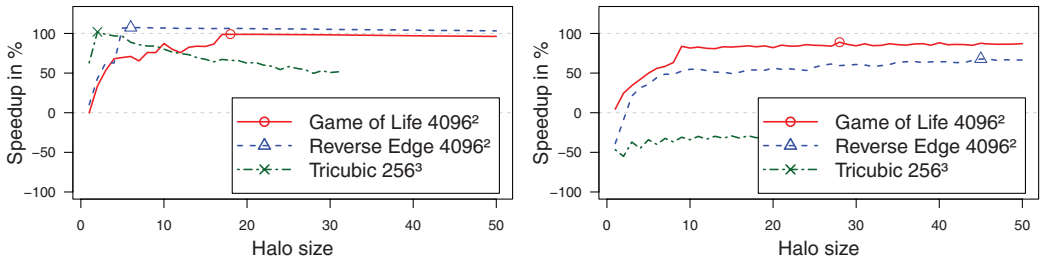


(a) Game of Life (b) Reverse Edge

(c) Swim (d) Tricubic

(e) Hyperthermia (f) Himeno

Fig. 10. Impact of the problem size on the optimal halo size. The legend shows the number of grid points in the total volume. On each graph, we mark the optimal halo size that obtains the best performance. For the 3D application, the small volumes are too small to investigate larger halos.



(a) Nvidia GTX 590 (b) AMD Radeon 5970

Fig. 11. Comparison of hardware performance between Nvidia GTX 590 and AMD Radeon 5970. The Radeon does not have enough memory to support a larger halo size than 32 for *Tricubic*.

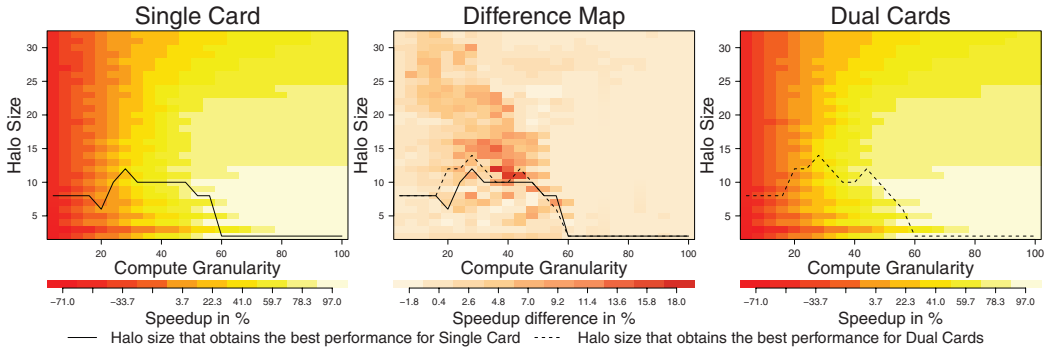


Fig. 12. Comparison of PCIe layouts. The same synthetic kernel is run using two GPUs in a four GPU system. The left graph shows the speedup achieved using the *Single Card* configuration when varying the halo size and compute granularity. The right graph shows the speedup obtained using the *Dual Cards* configuration. The middle heatmap represents the difference of speedup between the second and first configuration.

into many parts, or the communication costs involved may make applications unable to scale efficiently to more than two GPUs.

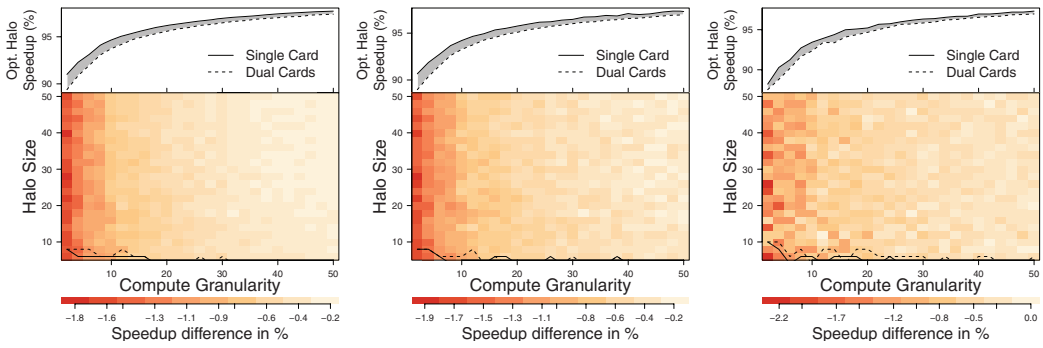
In this section, we explore the performance impact of choosing different device subsets in setups providing four GPUs split over two cards, as described in Section 5.2. In order to explore the space fully, and because the ratio between compute and communication performance of future GPUs is hard to predict, we explore it by artificially shrinking the time of the compute operation from the time taken by our naïve implementation to virtually nothing. This allows us to investigate the space generated by applications which have had their single GPU kernels more highly optimized or which are running on more powerful hardware. As in the previous section, we explore the impact of the halo size for each compute granularity to see the impact of the compute time on the communication overhead.

Figure 12 shows the direct comparison of two different configurations running on the same system (*P67-NF* with two dual GPU GTX 590 graphics cards), but using different PCIe layouts. The left graph shows the *Single Card* configuration, while the right graph shows the *Dual Cards* configuration. In both cases, two PCIe multiplexers (one on the motherboard and one on the graphics card) need to be traversed in order to communicate with a GPU. Both plots show the *Himeno* synthetic kernel with an input size of 256^3 grid points. The lines plotted on each of these graphs represent the optimal halo size, which is the halo size giving the highest speedup for a given granularity.

The middle graph shows the performance difference of speedup between the two configurations. This plot shows us if a performance difference exists somewhere in the full space. The optimal halo lines are identical to the ones on the left and right plot. They illustrate whether the optimal halo size traverses regions with high differences in performance. In this case, choosing the wrong halo size has significant impact on the obtained speedup.

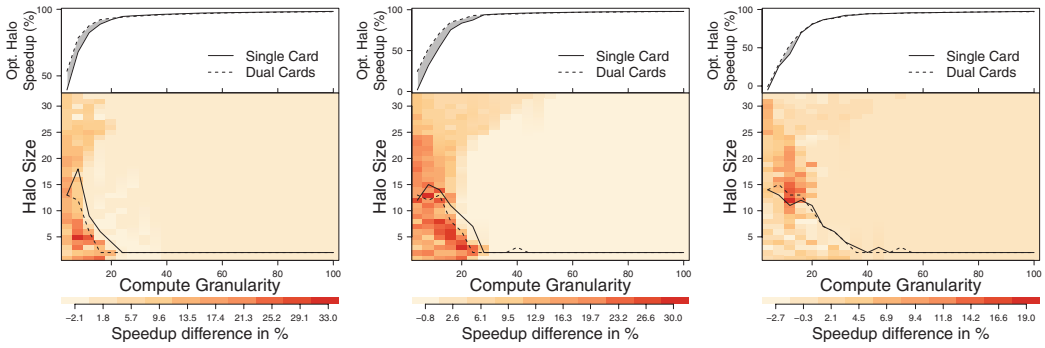
In the rest of this section, we only use the difference map to visualize the difference between the *Single Card* and *Dual Cards* configurations. We summarize the speedup information as a line plot above the difference map (see Figure 13(a) for example), that shows the best obtainable speedup for a given compute granularity. All evaluations are performed using the Nvidia GTX 590 graphics cards, unless stated otherwise. Figures 13 to 15 show the difference maps for different synthetic kernels running on our three evaluation systems.

For the 2D synthetic kernel *Reverse Edge* (see Figure 13), there is very little difference between the *Single Card* and *Dual Cards* configuration. For small compute granulari-



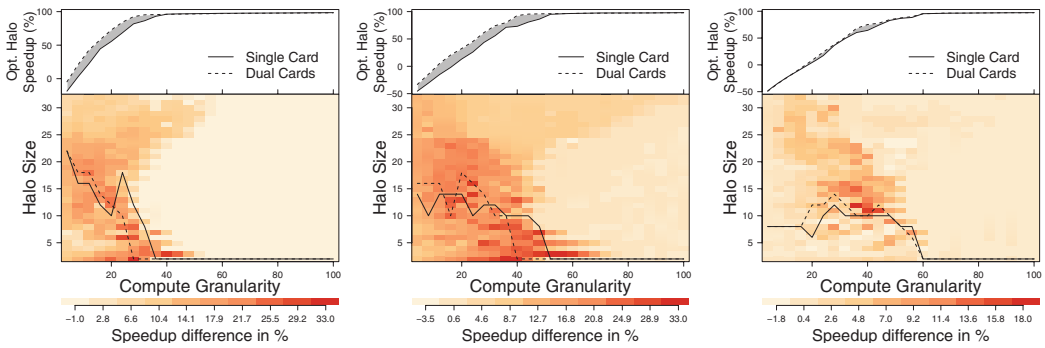
(a) *X79* (b) *P67-nat* (c) *P67-NF*

Fig. 13. Impact of the PCIe layout on different systems using the GTX 590 GPUs for *Reverse Edge* with an input size of 4096^2 grid points.



(a) *X79* (b) *P67-nat* (c) *P67-NF*

Fig. 14. Impact of the PCIe layout on different systems using the GTX 590 GPUs for *Hyperthermia* with an input size of 256^3 grid points.



(a) *X79* (b) *P67-nat* (c) *P67-NF*

Fig. 15. Impact of the PCIe layout on different systems using the GTX 590 GPUs for *Tricubic* with an input size of 256^3 grid points.

ties, the *Single Card* configuration tends to obtain about 1.5% higher speedup than the *Dual Cards* configuration. The similarity between the three systems is caused by the high scalability of the two-dimensional problem. The speedup achieved by using two GPUs over one GPU tends towards 100%, and the lower limit for the entire space is above 90%. We find similar results for the other 2D synthetic kernels.

Figure 14 shows the same experiment for the *Hyperthermia* synthetic kernel. This space is very different. For the *X79* and *P67-nat* systems (see Figures 14(a) and 14(b)), which are both using native PCIe connections, the overhead of the multiplexer is clearly visible for small compute granularities using the *Single Card* configuration. Most of the space shows little difference between the two configurations. This area corresponds to the space where the communication is completely hidden by the computation costs. The communication overhead starts taking over gradually as the granularity decreases. This happens when either the halos are too small and have to be swapped too often to be amortized by the computation, or when they are too large and the swap takes too long to be hidden by the core computation. This explains the round shape of the lightly colored area along the left side.

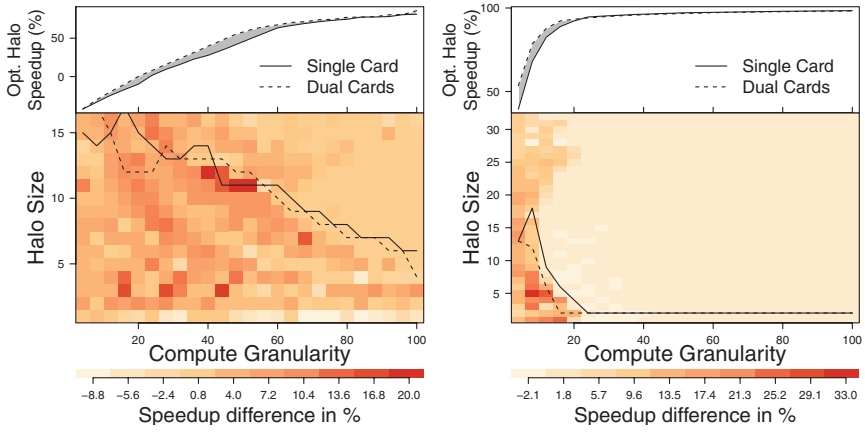
For the two first systems, we can clearly observe the gain of avoiding the multiplexer. For the *Single Card* configuration, communication has to go through the on-GPU multiplexer. In the *Dual Cards* configuration both GPUs are accessed through the on-GPU multiplexer. However, as the second GPU on each card is idle, the overhead is negligible or null. The *Dual Cards* configuration obtains higher speedups for lower compute granularity. This difference is accentuated in the *P67-nat* system that only provides PCIe x8 links, compared to the PCIe x16 links found in the *X79* system.

The multiplexer becomes unavoidable in the *P67-NF* system for both configurations. In total, there are three multiplexers: one on the motherboard and one on each graphics card. For the *Single Card* configuration, one graphics card is not being used but the multiplexer on the other one is used to access both GPUs. For the *Dual Cards* configuration, one GPU on each PCB is idle, but the multiplexer on the motherboard is being used to access both graphics cards. Our experimental exploration shows that in this case using both cards is still beneficial, with a gain of up to 22.69% along the optimal paths.

Figure 15 shows the same experiment for the *Tricubic* synthetic kernel. We observe a similar behavior as for *Hyperthermia*. However, the larger stencil shape used in *Tricubic* results in more communication and increases the difference between the two configurations further.

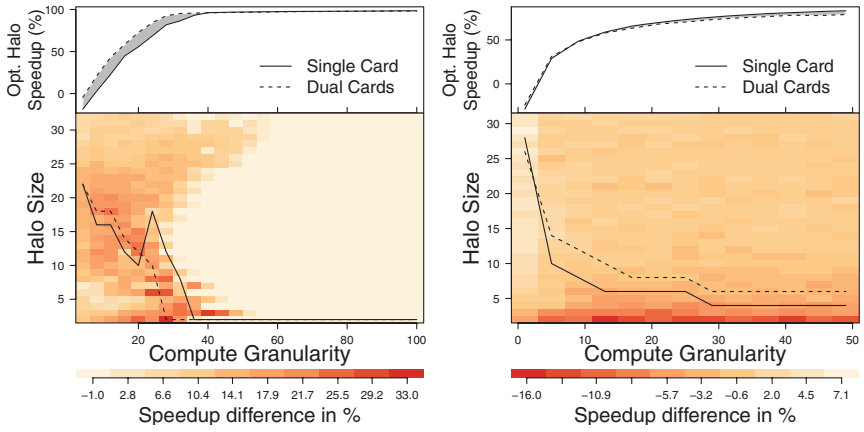
Figure 16 shows the impact of the problem size on the performance difference for *Hyperthermia* running on the *X79* system. As established in Section 6.3, changing the problem size changes the communication pressure. This extra pressure on the smaller volume translates into a performance difference across the entire space, as opposed to no difference for most of the space when using a larger volume.

Figure 17 shows a selected result comparing the performance difference using different GPUs. Figure 17(a) shows the difference map for *Tricubic* running on the *X79* system using the GTX 590 graphics cards. Figure 17(b) shows the difference map for the same setup, except that we replaced the graphics cards with AMD Radeon 5970. Besides the scalability difference, which is caused by the inferior compute performance of the Radeon 5970, we can notice that the best configuration depends on the compute granularity. For most of the presented results, the *Single Card* configuration is the best choice for the Nvidia GTX 590 GPUs. For the Radeon 5970, the *Dual Cards* configuration performs best for small compute granularities, while the *Single Card* configuration performs better for larger compute granularities. Choosing to use a single card leads to a performance improvement of up to 13.2% along the optimal path.



(a) Hyperthermia 128 (b) Hyperthemia 256

Fig. 16. The impact of the PCI layout depends on the problem size. (a) presents the *Hyperthermia* synthetic kernel running on the setup with the X79 chipset and a problem size of 128^3; (b) shows the performance difference for the same experiment and the same setup but a problem size of 256^3.



(a) Nvidia GTX 590 (b) AMD Radeon 5970

Fig. 17. Impact of the PCIe layout on the *X79* system using different graphics cards for the *Tricubic* synthetic kernel with an input size of 256^3 grid points.

6.4.2. Data Placement for Full System Utilization. Figure 18 is similar to Figure 12, but this time all the available GPUs are being used. Both configurations show a similar speedup, up to 3.85x for high compute granularities. However, the maximum observed slowdown also increases compared to just using two GPUs. Having four partitions increases the pressure on the swapping cost, and all the communication involved is harder to amortize by the core computation.

Even though the full system is utilized, the difference map shows some inequalities between the configurations for lower compute granularity. Over the complete space, *Blocking* is up to 21% faster than *Circular*. At other points in the space, *Circular* is up to 13% faster than *Blocking*. The optimal path for both configurations crosses this space, meaning the best partition mapping decision depends on the compute cost in this case.

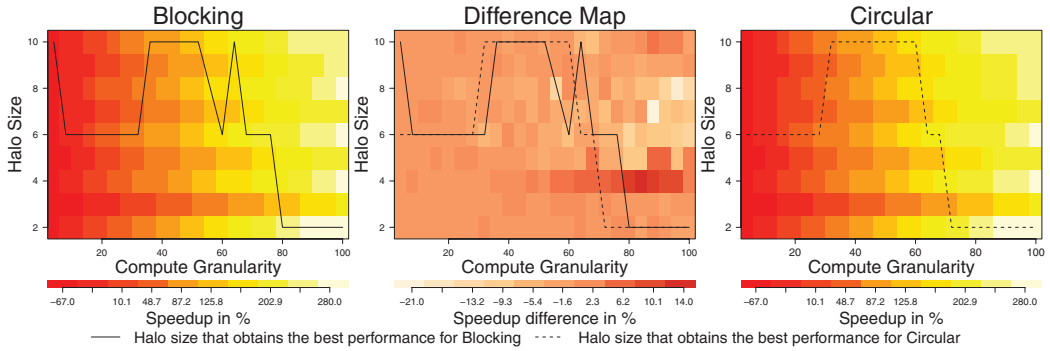


Fig. 18. Data placement impact when using four GPUs. The left-hand side shows speedup when four partitions have one of their dependencies on the same card and the right-hand side shows another placement where partitions have both dependencies on the other graphics card.

6.5. Autotuning

Our results so far confirm the nontrivial nature of the stencil optimization space, and hence the need for an automated approach to its navigation. In this section, we describe and evaluate our autotuning strategies. Our strategies combine offline and online phases.

The first offline decision concerns the *volume orientation*, as explained in Section 3.3. This optimization simply requires examination of the volume and stencil shape. In the case of an asymmetric stencil, the volume is aligned in such a way that the cut is applied to decrease the swapping frequency. Otherwise, the volume is aligned to be cut along the largest dimension, which allows a broader range of halo sizes to be considered and minimizes the cut area.

In the next offline step, we select a swapping strategy from a pool of predefined strategies (presented in Section 3.3). Our experiments so far indicate that our most advanced strategy, which overlaps communication and computation, is optimal in all cases; hence we select and implement it offline. However, our framework can easily switch this into an online decision if the need arises.

The framework then tunes the *GPU selection and partitioning*, which is an offline decision. It determines how many, and which GPUs to use, and in the case of full system utilization, how to assign partitions to GPUs. To achieve this, all stencils used in the application are automatically extracted and profiled independently offline to assess their scalability. The results are combined, weighted by the complexity and usage of each stencil in the case of multi-operator applications, to make an overall decision.

Finally, the *halo size* is adapted online. As the application runs, we vary the halo size and gather performance data. The data is used to refine the halo adjustments. We have experimented with a range of adjustment strategies. The search is always informed by the (application-specific) shape of the stencils to prune part of the search space. This avoids the oscillating performance for applications like *Tricubic* (discussed in Section 6.3).

The simplest is an *exhaustive search*: try all feasible halo sizes in order and eventually pick the best. The second strategy is inspired by the observation that for most applications, performance has a simple bitonic relationship to halo size. The *hill climbing search* increases the halo size linearly, from the minimum, until performance degrades for several consecutive points.

These two strategies are linear searches, resulting in poor performance if the optimal halo is in the middle of the range or at the opposite end. To guarantee reasonable

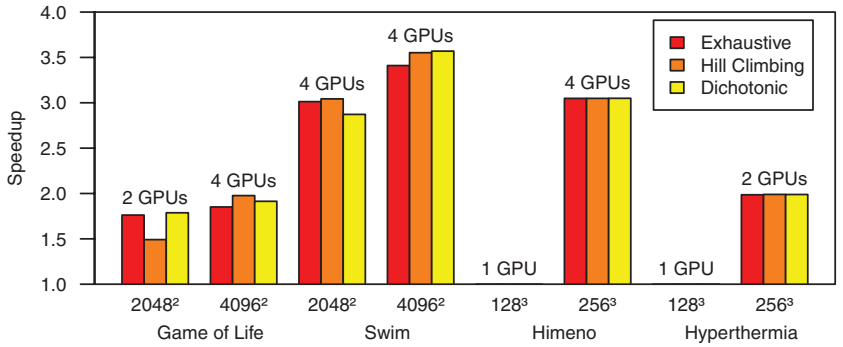


Fig. 19. Speedup over single GPU using autotuning for two input sizes and 100,000 iterations. The autotuner decides how many and which devices to use and performs an online search for the optimal halo size using several search strategies.

convergence time, we also implemented a *dichotomic search*. We sample the average time per iteration at five regularly spaced halo sizes, and recurse within the best interval.

Figure 19 shows the efficiency of the autotuning for eight applications and two input sizes each. Each bar represents the final speedup obtained across the given number of GPUs when accounting for the overhead of the online search phase and the outcome of each search strategy. Each application executes 100,000 iterations. The average speedup for applications running on two GPUs is 1.83x, and on four GPUs is 2.86x. Speedup increases with problem size, which indicates that communication is the limiting factor to scalability. 3D applications with an input size of 128^3 are not worth distributing across several devices, the communication dominates over the gain of parallel computation.

For the online tuning, we observe that the overall performance is similar for the three searches. In most cases, *hill climbing* and the *dichotomic search* perform a little better than the *exhaustive search*. In particular, this is the case for larger problem sizes.

In order to compare the searches more accurately, we break down the performance impact of the halo refinement phase and the outcome of the search in Table II. We evaluate the quality and online performance overhead of our strategies with respect to the performance of an idealized *oracle* strategy. The oracle performance corresponds to the best performance found for the given application across the full optimization space. For each application, input size, and search strategy we report:

—the percentage of oracle performance obtained during the online tuning phase. This figure reflects the overhead introduced by our online scheme and the suboptimal halos with which it experiments.
—the percentage of oracle performance obtained after the online tuning phase. This figure reflects the quality of the tuning outcome.
—the number of iterations to convergence. This captures the minimum number of iterations necessary for the search to converge. To increase the accuracy of the measurement and reduce the noise, each swapping is sampled ten times, making it expensive in terms of iterations for large halos.

The *exhaustive search* guarantees to eventually find the optimal halo size, which gives perfect posttuning performance. It requires a very expensive search phase, but as the number of iterations increases, this search phase is amortized, giving this strategy a good overall performance. However, when the number of iterations is not high enough,

Table II. Performance of Various Search Strategies

	Game Of Life		Swim		Himeno		Hyper	
	2048^2	4096^2	2048^2	4096^2	128^3	256^3	128^3	256^3
Offline Search Parameters								
Number of GPUs	2	4	4	4	1	4	1	2
Data Placement	B/D	C/D	C/D	C/D	n/a	C/D	n/a	B/D
Exhaustive Search Speedup								
Search Phase Perf. (%)	94.43	95.44	90.67	87.26	n/a	91.83	n/a	91.13
Post Search Perf. (%)	100.00	100.00	100.00	100.00	n/a	100.00	n/a	100.00
Iterations to converge	36,550	145,350	18,490	72,250	n/a	550	n/a	2,100
Hill Climbing Search Speedup								
Search Phase Perf. (%)	71.11	94.96	89.12	90.63	n/a	92.41	n/a	98.34
Post Search Perf. (%)	83.19	99.98	100.00	95.35	n/a	100.00	n/a	100.00
Iterations to converge	2,300	2,750	7,280	23,030	n/a	540	n/a	90
Dichotomic Search Speedup								
Search Phase Perf. (%)	97.90	95.53	91.24	87.93	n/a	94.02	n/a	91.67
Post Search Perf. (%)	99.36	96.78	93.69	95.35	n/a	100.00	n/a	100.00
Iterations to converge	4,260	9,400	5,510	8,430	n/a	380	n/a	540

Note: Data Placement Abbreviations: (B) Blocking, (C) Circular, (D) Dual Cards, (S) Single Card
Search Phase Perf.: percentage of oracle performance obtained during the online tuning phase.
Post Search Perf.: percentage of oracle performance obtained after the online tuning phase.

the search does not have time to sample the entire space and the overhead is much higher. This is the case for *Game of Life* with a volume size of 4096, which we allowed to overrun beyond the standard 100,000 iterations in order to fully measure convergence. This effect is increased by an important performance difference across the full range of possible halo sizes, which increases the full search time, as shown in Section 6.3.

Hill climbing converges faster than the exhaustive search in most cases. This makes it more efficient than the first strategy in most cases. For *Hyperthermia* with a volume size of 256, the convergence is achieved twenty times faster. However for *Himeno*, it is almost as expensive as the exhaustive search because the optimal halo is large. Therefore the high-quality result is not surprising in this case. Furthermore, it cannot cope well with bumpy search spaces. For example, with *Game of Life* and a volume of 2048, hill climbing stopped in a local minimum, missing the true optimal halo. Despite a fast convergence, the overall performance converges to the outcome of the search phase, which is in this case worse than performing a complete search.

Dichotomic search is the fastest to converge, but gives only an approximation of the optimal halo size. As it recursively samples only a few points and makes a search using the extremities of each range, the result of the search could be quite far from the value of the optimal halo size. However, it does not stop on the first local minima like the hill climbing strategy. For applications with a large input data, this strategy is necessary to navigate through the large parameter space in a small number of iterations. The number of iterations necessary to sample a halo size is relative to its size, as it affects the swapping frequency, which is why noniterative searches are more performant for large ranges. Furthermore, as demonstrated by the synthetic benchmarks, the optimal halo size tends to increase as the compute time decreases, making nonlinear searches like the *dichotomic search* well adapted to explore large parameter spaces with a small overhead at the cost of having only an approximation of the optimal halo.

For the 3D applications, *Himeno* and *Hyperthermia*, there is little difference between the three searches. This is due to the high number of iterations and the relatively small range of possible halo sizes. All three searches perform well and find a point close to the

optimal halo size in a small number of iterations. The overall speedup converges to this value, amortizing the search overhead. For larger problem sizes, the same differences as the 2D applications would arise with a greater magnitude, as the amount of redundant computation increases much faster when the dimensionality of the cut increases.

7. RELATED WORK

Stencil computations have been very extensively studied, both generically and in the context of individual application instances, and across diverse architectural platforms including many-core and cluster, with or without GPU acceleration.

The high-performance computing literature contains countless discussions of stencil-structured codes implemented on message-passing architectures. In early examples, Fox [1984] notes the need for "subroutine calls" to handle the "new form of boundary conditions involving the transfer of information from other nodes" (i.e., halos), while Reed et al. [1987] discusses the relationship between partition shape, stencil structure, and architecture. Optimization techniques can be borrowed from message passing approaches, replacing network communication by PCI communication in our new context. For example, Ripeanu et al. [2001] gives a good discussion of the importance of finding the optimal halo size (referred to as the "ghost zone").

With respect to GPUs, PATUS is a code generation and tuning framework for stencil computations with support for multicore processors and a single GPU [Christen et al. 2011], Chombo [Applied Numerical Algorithms Group, LBNL] focuses mainly on PDE solvers for regular and adaptive meshes running on multicore CPUs while Kamil et al. [2010] deal with autotuning of stencil kernels on a range of multicore and single GPU systems. PADS is a compiler tool which generates CUDA code from stencil applications written in OpenMP [Han et al. 2011] and Mint provides a source to source translator to transform annotated C code in CUDA [Unat et al. 2011]. It is beyond our scope to survey this material exhaustively. We note that our experimentation with granularity tunable synthetic benchmarks helps to keep our work orthogonal to improvements offered by single GPU tuning schemes, whether generic or application specific [Itu et al. 2011; Phillips and Fatica 2010].

Our work is more closely related to approaches that employ multiple GPUs in the solution of a single stencil application. Most previous systems, designed with HPC contexts in mind, treat each GPU as an accelerator associated with a distinct process and node in a cluster. For example, Physis is a compiler-based approach for automatic parallelization of a code written in a domain-specific language into CUDA code and MPI code for internode communication [Maruyama et al. 2011]. SBLOCK is a framework for PDE solvers based on CUDA and MPI and tested with sixty-four GPUs. Phillips and Fatica [2010] investigated implementation and optimization strategies for the Himeno benchmark using CUDA and MPI. They scaled their experiments to sixteen GPUs using a GPU cluster. Our scheme shares some high-level principles with these approaches, including the need for partitioning and correct, optimized maintenance of halos. However, our work is distinguished by the use of multiple GPUs *on the same node*, with resulting interesting effects of the communication substructure. The SkelCL [Steuwer et al. 2012] and SkePU [Dastgeer et al. 2011] projects target single-node, multiple-GPUs configurations from skeleton-structured APIs. While neither supports stencils directly as a single abstraction, stencil computations can be expressed by composition and iteration of simpler data parallel operations.

Kim et al. [2011] provide a generic mechanism through which OpenCL programs developed for a single GPU system can be transparently executed on multiple GPUs. The underlying kernel partition strategy is supported by a dynamic, sampled tracing mechanism (executed on the host CPU) to determine memory footprints and what is in effect a simple virtual memory mechanism, which supports the necessary coherence

across GPU memories between kernels. In contrast to our approach, Kim's system has the strength of being transparently applicable to any existing Open CL program. On the other hand, it requires the programmer to write a full OpenCL program in the first place, where our system asks only for the core of the kernel. Furthermore, our constrained domain permits two classes of optimization not available to Kim's system. Firstly, we can explore optimizations that span multiple kernel invocations (corresponding to multiple iterations of the stencil). For example, we can leave non-halo data in place on GPUs and move only halos between stencil iterations, in a way not accessible to Kim's kernel-by-kernel translation and tracing. Secondly, we can explore skeleton-specific optimizations such as tuning the halo-width and its effect on the computation/communication trade-off.

The impact of PCI configuration on multi-GPU systems performance has been noted previously [Schaa and Kaeli 2009], but no systematic attempts have been made to incorporate such phenomena into a systematic tuning mechanism.

8. CONCLUSION AND FUTURE WORK

We have presented a framework that distributes stencil computations across multiple GPUs and automatically optimizes the distribution in a way that is sensitive to characteristics of the stencils, the problem size, the GPU devices, and the underlying PCIe interconnect. Our approach is the first to take all of these issues into consideration systematically and simultaneously.

We have demonstrated that adaptation to the given PCI express configuration is a significant factor in achieving high performance overall, with a potential performance loss of 23% if this is not correctly addressed. Our autotuning strategies achieve excellent performance, with low overheads, when compared to the behavior of an idealized exhaustive search. We have demonstrated the robustness of our autotuner to likely future improvements in single GPU performance and tuning.

In future we plan to introduce energy use as an optimization cotarget, allowing a trade-off with raw performance when assessing scalability. The structure of our framework will allow this to be performed transparently to the programmer.

We plan to further develop our heuristics, for example to address the issues that arise in the presence of several layers of multiplexers. Looking further ahead, we will consider increasingly heterogeneous platforms. For example, modern processors, such as AMD's Fusion series and Intel's Ivy-Bridge processors, include medium-performance GPU cores. We aim to incorporate such resources into our strategies. We intend to incorporate a layer of cluster parallelism, in order to offer seamless, transparent programming and tuning of stencil applications across the full systems spectrum. Finally, we plan to consider the extension of our approach to other problem patterns, such as wavefront and adaptive mesh refinement.

REFERENCES

AMD. Accelerated parallel processing (APP) SDK (formerly ATI stream). http://developer.amd.com/appsdk

Applied Numerical Algorithms Group, LBNL. CHOMBO - Software for adaptive solutions of partial differential equations. https://commons.lbl.gov/display/chombo/

Christen, M., Schenk, O., and Burkhart, H. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*.

Dastgeer, U., Enmyren, J., and Kessler, C. W. 2011. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering (IWMSE'11)*. 25–32.

Fox, G. C. 1984. Concurrent processing for scientific calculations. In *Proceedings of the 28th IEEE Computer Society International Conference (COMPCON'84)*. 70–73.

HAN, D., XU, S., CHEN, L., AND HUANG, L. 2011. PADS: A pattern-driven stencil compiler-based tool for reuse of optimizations on GPGPUs. In *Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. 308–315.

ITU, L., SUCIU, C., MOLDOVEANU, F., AND POSTELNICU, A. 2011. GPU optimized computation of stencil based algorithms. In *Proceedings of the 10th Roedunet International Conference (RoEduNet)*. 1–6.

KAMIL, S., CHAN, C., OLIKER, L., SHALF, J., AND WILLIAMS, S. 2010. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. 1–12.

KEUTZER, K., MASSINGILL, B., MATTSON, T., AND SANDERS, B. 2010. A design pattern language for engineering (parallel) software: Merging the PLPP and OPL projects. In *Workshop on Parallel Programming Patterns (ParaPLOP)*.

KHRONOS GROUP. 2008. Khronos launches heterogeneous computing initiative. http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/

Khronos Group 2011. *The OpenCL Specification version 1.2* 15 Ed. Khronos Group. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

KIM, J., KIM, H., LEE, J. H., AND LEE, J. 2011. Achieving a single compute device image in opencl for multiple GPUs. In *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming*. 277–288.

MARUYAMA, N., NOMURA, T., SATO, K., AND MATSUOKA, S. 2011. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of the 2011 International Conference fo High Performance Computing, Networking, Storage, and Analysis(SC'11)*. 1–11.

MATTSON, T. G., SANDERS, B. A., AND MASSINGILL, B. L. 2004. *Patterns for Parallel Programming*. Addison-Wesley.

NVIDIA. CUDA Zone. http://developer.nvidia.com/category/zone/cuda-zone.

PCI-SIG 2010. PCI Express®Base Specification 3.0 Ed. PCI-SIG.

PHILLIPS, E. H. AND FATICA, M. 2010. Implementing the himeno benchmark with CUDA on GPU clusters. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. 1–10.

REED, D., ADAMS, L., AND PATRICK, M. 1987. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. Comput. C-36,* 7, 845–858.

RIPEANU, M., IAMNITCHI, A., AND FOSTER, I. 2001. Cactus application: Performance predictions in grid environments. In *Proceedings of European Conference on Parallel Computing (EuroPar)*. Springer, 807–816.

SCHAA, D. AND KAELI, D. 2009. Exploring the multiple-gpu design space. In *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. 1–12.

STEUWER, M., KEGEL, P., AND GORLATCH, S. 2012. Towards high-level programming of multi-GPU systems using the SkelCL library. *In Proceedings of the IEEE* 26^{th} *International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*. 1858–1865.

UNAT, D., CAI, X., AND BADEN, S. B. 2011. Mint: Realizing CUDA performance in 3D stencil methods with annotated c. In *Proceedings of the International Conference on Supercomputing* (ICS'11). 214–224.

Received June 2012; revised October 2012; accepted November 2012