Auto-Generation and Auto-Tuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters*

Yongpeng Zhang and Frank Mueller

Abstract—This paper develops and evaluates search and optimization techniques for auto-tuning 3D stencil (nearest-neighbor) computations on GPUs. Observations indicate that parameter tuning is necessary for heterogeneous GPUs to achieve optimal performance with respect to a search space. Our proposed framework takes a most concise specification of stencil behavior from the user as a single formula, auto-generates tunable code from it, systematically searches for the best configuration and generates the code with optimal parameter configurations for different GPUs. This auto-tuning approach guarantees adaptive performance for different generations of GPUs while greatly enhancing programmer productivity. Experimental results show that the delivered floating point performance is very close to previous handcrafted work and outperforms other auto-tuned stencil codes by a large margin. Furthermore, heterogeneous GPU clusters are shown to exhibit the highest performance for dissimilar tuning parameters leveraging proportional partitioning relative to single-GPU performance.

Index Terms—Accelerators, GPGPU programming, Stencil Codes, GPU clusters

1 Introduction

Main-stream microprocessor design no longer delivers performance boosts by increasing the processor clock frequency due to power and thermal constraints. Nonetheless, advances in semiconductor fabrication still allow the transistor density to increase at the rate of Moore's law. This has resulted in the proliferation of many-core parallel architectures and accelerators, among which GPUs quickly established themselves as suitable for applications that exploit fine-grained data-parallelism.

Still, software development for parallel architectures turns out to be more difficult than that for uni-processors in terms of obtaining high performance, even when aided by new programming models such as CUDA [1] and OpenCL [4]. Programmers spend substantial time and effort to understand the underlying architecture to best utilize all resources. This can become a daunting task since performance is affected by a multitude of architectural features. Even worse, architectural difference between generations of the same hardware line may require a diversity of optimization strategies with sometimes opposite optimal set-points. Programmers may have to explore many (if not all) combinations of optimization options / parameters to determine the best configuration for a particular hardware. This poses a great challenge since programmer productivity is adversely affected by lengthy tuning efforts. Simply re-profiling and re-writing the program upon each hardware upgrade is neither desirable nor feasible over time.

Current compilers for general-purpose languages struggle to balance portability, performance and programmability. Domain-specific languages (DSLs), in contrast, offer a promising solution at the expense of sacrificing language generality [2]. DSLs have restricted expressiveness aimed at a particular domain. It is precisely this domain-specific knowledge that allows the DSL-compiler to attain performance comparable to hand-coded domain implementations. In contrast, general-purpose languages are inherently limited in their optimization scope in exchange for assuring correctness and good overall (but not best) performance on average for a wide range of applications. Examples of well-known DSLs are HTML for web pages, Matlab for scientific computation and SQL for database queries.

This work focuses on providing a portable source-to-source auto-generation and auto-tuning framework for iterative 3D Jacobi stencil computations on different GPUs. We generate stencil code for NVIDIA's GPUs via CUDA, yet the same principles apply for GPUs of other vendor and comparable programming models, e.g., OpenCL [4].

Stencil (nearest-neighbor) computations are widely used in scientific computing, including structured grids as well as implicit and explicit partial differential equation (PDE) solvers in domains ranging from thermo/fluid dynamics over climate modeling to electromagnetics among others. An iterative explicit stencil computation is comprised of computation-intensive kernel. At each discrete timestep, all stencil points are updated according to values of their spatial neighbors from a previous timestep. On one hand, the uniform and communication-free behavior is well suited for the SIMT (single instruction multiple threads) paradigm advocated by state-of-the-art GPUs. On the other hand, an efficient GPU implementation is sensitive to neighbors accessing patterns across different stencils. One key characteristic of most stencil computations is the overlap in input values to update multiple neighboring points. Exploiting

This work was supported in part by NSF grants 1058779, 0958311, 0937908, and DOE grant DE-FG02-08ER25837.
Authors' address: Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206.

this property is crucial to achieve competitive performance on GPUs. One common GPU technique is to use the on-chip *shared memory* (shared by a warp/block of threads) as an intermediate storage space for overlapped input values. Instead of letting each thread fetching all inputs from off-chip global memory, all inputs are first cooperatively loaded to shared memory before they are referenced when computing a new stencil value. This is beneficial even in more recent generations of cache-enabled GPUs since this shared memory is orders of magnitude faster than global memory. It is crucial to determine is how many threads should be grouped together in one block: Increasing the block size increases shared memory data reuse but may also deteriorate the GPU's occupancy rate of processing units [1].

There are many other factors that affect the performance. For example, how many stencil points should a thread work on? The larger the number, the more instruction-level optimizations can be applied by a compiler. But the less data-parallelism is exposed, the higher risk is for not fully utilizing a GPU's processing units. Also, is mapping inputs to texture memory faster? Our experiments show that the answer varies from case to case. Overall, there is no universal, optimal configuration for all types of stencil computations on different GPU models. Therefore, auto-tuning is not only desirable but also necessary to improve performance in this particular domain.

This heterogeneity across different generations/models of GPU exerts more challenges to programmers working on a cluster with hybrid models of GPUs. Such clusters become increasingly common due to the variety of GPUs on the shelf and incremental hardware upgrades. We further study the strategies to make the best use of all available GPU resources on homogeneous/heterogeneous GPU clusters with optionally dissimilar parameters per distinct GPU type.

This work falls into the area of implicitly parallel programming models [5]. Our model relies on a compiler to generate highly efficient parallel code without requiring much interaction with the programmer.

The contributions of this paper are:

- We abstract a wide variety of stencil computations into a set of domain-specific specifications. This allows the end-user to customize specific problems without having to consider the underlying architecture.
- We thoroughly summarize optimization techniques for stencil problems in previous literature and extract three sets of key parameters that affect the performance: (1) Block sizes that determine the shared-memory usage per block; (2) block dimensions that affect the number of registers consumed by each thread and (3) whether or not to map a subset of the input into texture memory.
- We develop an auto-generation and auto-tuning framework, i.e., we translate stencil specifications into executable code that is subsequently auto-tuned to the optimal configuration within a parameterized search space for each target GPU.
- We apply auto-generation and auto-tuning as a means for parameter optimization to GPU clusters and generate MPI program with identical parameters per GPU in homogeneous GPU cluster and with potentially dissimilar parameters per distinct GPU for heterogeneous GPU clusters.
- We show that heterogeneous GPU clusters exhibit superior weak scaling when leveraging *proportional partitioning* of the data space relative to single-GPU performance.
- Experimental results show competitive performance to manual tuning and demonstrate the superiority and necessity for auto-tuning to combining performance with correctness.

2 Design Overview

The stencil computation considered in this work allows point-wise updates according to a sequence of the following equation over a 3D rectangular domain:

$$\begin{aligned} out([i][j][k]) &= \sum_m w_m * in[i \pm I_m][j \pm J_m][k \pm K_m] \\ &+ \sum_l w_l[i][j][k] * in[i \pm I_l][j \pm J_l][k \pm K_l] \\ &+ \sum_n w_n * in_n \end{aligned} \quad (1)$$

The three dimensional addressing in the parenthesis on the left hand side is optional. If absent, we assume the result (out in this case) is an intermediate result that will be used later in another instruction on the right hand side as an input in_n. The first two parts on the right hand side characterize the stencil behavior. The center point and a number of neighboring points in the input grid (in) are weighted by either scalar constants (w_m) or elements in grid variables ($w_l[i][j][k]$) at the same location as the output. Offsets ($I_m/J_m/K_m$ and $I_l/J_l/K_l$) that constrain how the input grid is accessed are all constant. We call their maxima the halo margins of three dimensions ($halo\_i = \max\{I_{m/l}\}$, $halo\_j = \max\{J_{m/l}\}$ and $halo\_k = \max\{K_{m/l}\}$). To ensure that the access pattern is legal (non-negative indexing) for marginal elements in the input grid in, we assume both input and output grids (in and out) are enlarged by twice the halo margins on each associated dimension.

We differentiate w_ls and in in (1) and call them *array parameters* and *array input*, respectively. *Array parameters* are restricted by their access pattern: they can only be accessed at the same position as the output element. The *array input* can be accessed with various constant offsets ($i/j/k$s) on each dimension. We assume there is only one array input, but there can be zero or multiple array parameters.

Given the stencil specification that contains only a list of instructions in the format of Eq. 1, our auto-tuning framework generates a header file and an implementation file that can be either included in user code or compiled into libraries.

Excerpts of the generated code are depicted in Figure 12 (see Appendix document). We encapsulate all static and run-time stencil information into the data structure named

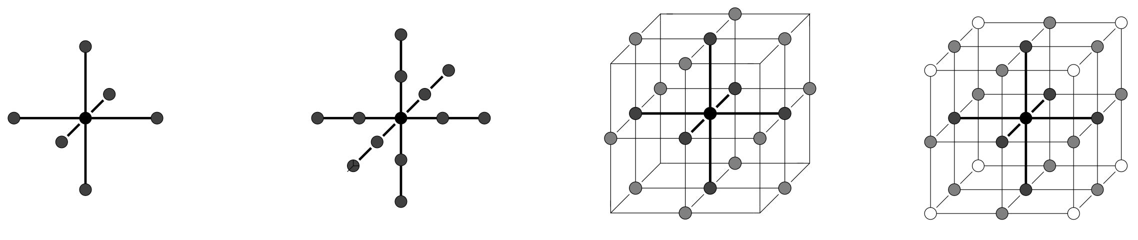

(a) 7-Point (b) 13-Point (c) 19-Point (d) 27-Point

Fig. 1. Stencil Examples

StencilConfig. The two major APIs are stencilIteration() and stencilIteration_mpi(). One performs single GPU calculations, the other is for multiple-node GPUs (GPU clusters) computations with node-to-node MPI message passing. The sample user code shows a typical usage of the APIs. The same code is also used in our auto-tuning engine to measure the wall-clock time for 100 iterations.

We call a stencil calculation an N-point stencil where N is the total number of input points used to calculate one output point and an order-M stencil where M is the maximum over all $halo_i/j/k$s ([8]). In this paper, we choose four types of stencil computations as benchmarks (see Figure 1).

- 7-Point Stencil (Figure 1(a)): Each element in the output grid is updated by the same position in the input grid and 6 neighbors offset by 1 on each direction. The grid point and 6 neighbors are scaled by α and β, respectively, before they are added to generate the output. Both α and β are constants. There are 8 floating-point operations for each point (6 adds and 2 multiplies).
- 13-Point Stencil (Figure 1(b)): The access pattern resembles the 7-point stencil except that the maximal distance to the neighbors extends to 2, making it an order-2 stencil. There are 15 floating-point operations at each point (12 adds and 3 multiplies).
- 19-Point Stencil (Figure 1(c)): This is also called the Himeno benchmark, the behavior of which is detailed elsewhere [9]. We use the same specification (Table I in [9]), except for ignoring the last line of residual calculation. All the weights in this benchmark are array parameters, making it a very cache-unfriendly benchmark. The total number of floating-point operations is 32 and there are 14 memory accesses per point.
- 27-Point Stencil (Figure 1(d)): Each grid point computation involves all points in a $3\times3\times3$ cube surrounding the center grid point. The 4 edge points, 8 corner points and 12 face neighbor points are multiplied by different constants. The number of operations is 30 with 4 multiplies and 26 adds.

Table 4 (see Appendix document) summaries the specifications and properties of the four stencils above.

Our framework has a few limitations. For example, we assume there is only one array input ($in[i][j][k]$) and one array output ($out[i][j][k]$) in the list of instructions in the format of Eq. 1. It is suitable for expressing Laplacian stencils ([6]). But it is straightforward to extend our framework to support Divergence or Gradient stencils that have multiple array inputs or multiple array outputs. There are also stencils that depend on multiple time steps. They can be treated as Divergence stencils. Customized code can be inserted by the user via manual change to the two templates (see Figure 13 in the Appendix document). The auto-tuning effort would proceed as before, even if additional user upcalls/outlined code was added to these templates.

2.1 Domain Specification and Framework

The formulation of a stencil is trivial in our framework as users provides a file specifying an equation according to the format of Eq. 1 plus parameters, such as the size of each dimension and data type (float or double). Table 4 (see Appendix document) shows that each stencil can be expressed by no more than a few lines of code. In contrast to hand-written CUDA kernels, which usually are hundreds of lines of code, this is a considerable improvement in terms of productivity. The conciseness of the user code also eliminates the errors introduced by the traditional transformation of a specification to an implementation in an imperative language by programmers, i.e., our automated transformation directly from the specification into imperative code reduces the chance to introduce programming errors and thus improves overall correctness.

The internal work flow of the framework is depicted in Figure 2. The parser analyzes the specification code in terms of Eq. 1 and extracts stencil features. These include halo margins ($halo\_i/j/k$), input/output array names, scalar or array parameters (ws) and the number of floating-point operations per stencil. The parser also detects different input sharing patterns between neighboring stencil outputs. If two neighboring outputs on the same Z plane share any input points on a different Z plane, we say that the stencil has corner accesses. 7-point and 13-point stencils are corner access free because relative to the center point at [i][j][k], there is only one input point on the $[k\pm1]$ and $[k\pm2]$ planes. 19-point and 27-point stencils have corner accesses because they need to access 5 and 9 points on the $[k\pm1]$ planes, respectively.

The code generator takes those feature parameters and chooses from two different template files depending on whether the stencil has corner accesses or not. The auto-tuning engine mainly operates on a single-node level, where optimized parameters are determined based on run-time profiling. It also ensures the correctness of the transformation via a set of unit tests. The same optimized parameters are used on multiple nodes to generate GPU cluster code with MPI support.

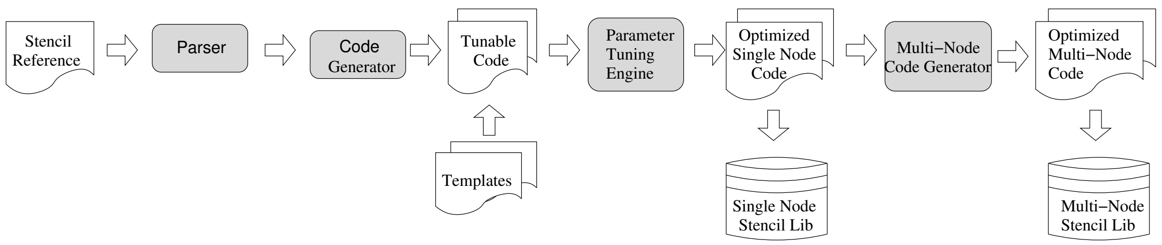


Fig. 2. System work flow: A user-defined specification is parsed to generate tunable code based on a template. The code is passed to an auto-tuning system to find the best parameter configuration for a single GPU (also for GPU clusters with MPI)

2.2 Domain Kernel Template

The design of the template kernel file is affected by the strategy to break the 3D rectangular space into thread blocks in CUDA. In related work, the 3D $X \times Y \times Z$ space was divided into smaller cuboids of size $x \times y \times z$ ([3], [7]). Each of them was mapped to a thread block of the same size. Recently, a 2.5D decomposition method was proposed ([9], [8]). It decomposes the 3D stencil space over the two most frequently changed dimensions (X and Y). Stencils of size $x \times y \times Z$ are assigned to a thread block, which contains only a plane of $x \times y$ threads. Inside the kernel, threads sweep over the Z axis and cooperatively process one plane at a time.

The benefits of the second method are three-fold: (1) It reduces the pressure on shared memory usage. In 3D decomposition, each block maintains a small block of size $(x+2\times halo\_i)\times(y+2\times halo\_j)\times(z+2\times halo\_k)$ in shared memory. The 2.5D method only needs a blocks size of $(x+2\times halo\_i)\times(y+2\times halo\_j)\times(1+2\times halo\_k)$. While sweeping through the z-axis, the planes can be shifted and reused as the work on z-axis is progressed. If the stencil does not have corner accesses, such as 7-point and 13-point stencils, we can further reduce the shared-memory usage to $(x + 2 \times halo\_i) \times (y + 2 \times halo\_j)$ while keeping the other parameters in registers. (2) The 3D decomposition method consumes more memory bandwidth on the Z axis because halo regions on Z are loaded twice on different blocks along the Z axis. (3) The 2.5D decomposition method tends to allocate more stencil points per thread (Z points per thread instead of z points). This is an optimization technique also known as thread fusion. For a large enough problem size, *i.e.*, $(X \times Y)$ generates enough threads, this helps to amortize other overheads, such as initial setup code in the kernel.

In our design, we adopt the block partition strategy in the 2.5D blocking method, *i.e.*, stencil space is partitioned into columns, each of which is assigned to a CUDA block (Figure 3(a)). The cross section of each column, shown in Figure 3(b), is of size $BlockSize.x \times BlockSize.y$. In the CUDA kernel, stencil points are processed along the Z axis. We further unroll over both X and Y dimensions to use $BlockDim.x \times BlockDim.y$ threads per kernel block (see Figure 3(c)). We ensure $BlockSize.x/y$ are divisible by $BlockDim.x/y$. Previous work only exploits the unrolling factor at most over the Y dimension. Our experiments illustrate that unrolling over both dimensions can be beneficial (see Section 4).

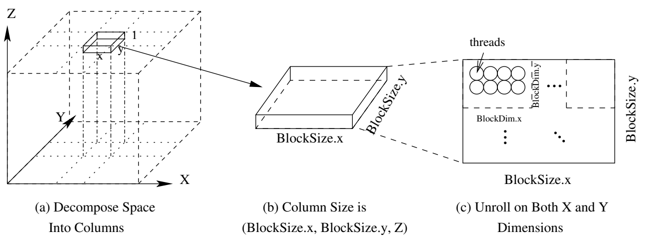

Z
Y
X
x
y
1
BlockSize.x
BlockSize.y
threads
BlockDim.y
BlockDim.x
BlockSize.x
BlockSize.y
(a) Decompose Space Into Columns
(b) Column Size is (BlockSize.x, BlockSize.y, Z)
(c) Unroll on Both X and Y Dimensions

Fig. 3. Stencil space decomposed over X & Y; process one column per thread block; thread code is unrolled.

Our code generator is based on two kernel templates, depending on whether the stencil has corner accesses (Fig. 13(a) in the Appendix document) or not (Fig. 13(b) in the Appendix document), where $halo\_k = 1$ is assumed in these figures. Their most distinct difference is how the shared memory is used. For stencils with corner accesses, all input stencils are first stored in shared memory to calculate the output stencils. The corner-free stencils can be treated as a special case where a plane of stencils does not share inputs other than the points on the same plane. Therefore, only the middle plane is stored in shared memory in this case — all other inputs along the Z axis are stored in register files. This approach, tailored to corner-free stencils, reduces the shared memory footprint. For later GPUs where register access time is less than shared memory, it also speeds up the stencil calculation.

3 GPU-Specific Auto-Tuning

In the following, we describe in detail various optimization techniques used by our implementation. We reason about their effects on performance and consider if they need to be made elastic by promoting them as parameters for auto-tuning.

3.1 Single Node Optimizations

Coalescing Memory Accesses: For NVIDIA GPUs, the latency of global memory references is deeply affected by whether the memory is accessed in coalesced way or not. More recent GPUs support coalesced memory access when memory accesses conducted by threads in one warp can be combined into as few memory transactions as possible [1], where a warp is the basic thread instruction scheduling unit in NVIDIA GPUs. We reinforce the following rules to coalesce most of the memory accesses:

- The size of the most frequently changing dimension (X dimension) for input/output arrays is padded to multiples of 32 stencil elements.
- The origin of the input/output arrays are shifted right by $32 - HALO\_I$ stencil elements relative to the memory pointer obtained from the CUDA malloc function. This guarantees 128-bit alignment. The internal origins of the input/output array thus become 128-bit aligned ensuring coalesced memory accesses for output arrays as long as every thread loads the same row at the same time when operating on a half-warp granularity.
- Parameter arrays are allocated to be the same size as the input/output array, even though only the internal elements are used throughout the stencil calculation. This way, the indices of parameter arrays and parameter input become identical saving registers and extra cycles for address calculations. Similar to the input/output arrays, their origins are also shifted to the right. Reading from the parameter arrays become coalesced as well.

Tuning the Block Size: Choosing the right block size is one of the most important factors to balance the utilization of registers and shared memory. Since we use Z-axis sweeps, our blocks have two parameters that determine the size of each sub-plane assigned to CUDA blocks: BlockSize.x and BlockSize.y (see Figure 3(b)). The optimal blocking size is determined by several seemingly conflicting factors:

- Since accesses to part of the halo margins (two gray columns in Figure 15 in the Appendix document) are non-coalesced memory accesses, we want to limit these as much as possible. This gives us incentive to increase $BlockSize.x$ as much as possible.
- To reduce the redundant loading of halo margins between different blocks, we need to keep the block close to a square shape.
- The shared-memory usage is proportional to $BlockSize.x \times BlockSize.y$. It must not surpass the shared-memory size on-chip.

Our experiments show that the optimal blocking size can be different under different scenarios: On one hand, different GPU models require different sizes for the same stencil problem. On the other hand, the same GPU model requires different blocking sizes for different stencil problems. To obtain the coalesced memory access effects for an input array, our search space for $BlockSize.x$ is a multiple of the half-warp (16, 32, 48, 64). $BlockSize.y$ has no such constraints. So we sweep its value continuously from 2 to 16.

Given a specific column size, it is not necessary to have a one-to-one mapping from the sub-plane to CUDA block threads. One thread can process multiple stencil points via loop unrolling (see Figure 3(c)). The search space for the CUDA block size ($BlockDim.x$ and $BlockDim.y$) is a subset of the block size search space, with the constraint that $BlockSize.x/y$ is integer divisible by $BlockDim.x/y$, thus avoiding branches altogether. The motivation behind this ratio is that a smaller set of threads has a higher efficiency in using registers. This thorough search gives us the opportunity to balance register utilization and shared memory space, two key and scarce resources for stencil implementations on GPUs.

Loading the Input Array Efficiently: An important step in the stencil kernel is to efficiently access the input array. A straightforward but naive implementation is to load it directly from the off-chip global memory while calculating the output point. The obvious drawback is that this does not exploit the data sharing between neighboring threads. The on-chip shared memory serves as an ideal user-controlled scratch pad in this scenario. The problem narrows down to how to efficiently load a larger block of data $((BlockSize.x + 2 * HALO\_I) \times (BlockSize.y + 2 * HALO\_J))$ using a smaller set of computation threads ($BlockDim.x \times BlockDim.y$). We first load the internal region ($BlockSize.x \times BlockSize.y$). Because $BlockDim.x/y$ are divisible by $BlockSize.x/y$, this can be done easily without branches. For marginal regions, we rely on the code generator to map computational threads to elements on the margin region, as shown in Figure 15 (see Appendix document). In the graph, we assume $BlockDim.x/y$ equals to $BlockSize.x/y$, respectively. Each computing thread is sequentially assigned to a point in the margin area. The x and y indices are auto generated as a constant array. The number of points in the margin area is not necessarily divisible by the number of computing threads. In those cases, threads will be responsible for loading more than one marginal points or there will be idle threads that load the upper-left corner point (see the Figure 15(a) in the Appendix document) to avoid diverging branches. Comparing with other approaches, e.g., [9], this method neither requires branches nor issues any unnecessary loads. The only non-coalesced memory loads are issued for the columns on each side of the sub-plane.

Figure 14 (see Appendix document) shows the auto-generated CUDA code for Figure 15(a) (see Appendix document). In this case, BlockDims are the same size as BlockSizes (6×4). Our code generator generates two index arrays (haloThreadmappingX and haloThreadmappingY) for all 24 threads. The upper left corner is assumed to have index (0,0), which is referenced by the last four idle threads. Inside the kernel, the offsets are pre-loaded and used to load the margin regions without any branches.

Using Texture Memory: Mapping the read-only input array into the GPU's texture memory has been shown to improve performance in [9], because texture memory provides cached read-only accesses optimized for spatial locality and it has separate texture addressing unit. But our experiments show that texture references do not result in performance gains for all cases. There is no native texture support for the double precision data type, but we can use the texture fetch for the int2 type and $\_\_hiloint2double$ to convert it to double. Whether or not to use texture memory for the input array is determined by a boolean tuning parameter.

In total, we have five tuning parameters (blockSize.x/y, blockDim.x/y and texture). This gives a few thousand combinations. We chose to exhaustively explore all possibilities of this design space in the auto-tuning engine to find the optimal setting. This does not affect the application runtime since tuning only needs to be conducted once for a particular GPU.

3.2 Multi-Node Auto-Tuning

For GPU clusters, we divide the stencil space along the Cartesian space. Each node is responsible for updating a smaller rectangular 3D space. The tuning parameters determined for a single node are re-used directly for multi-node scenarios. However, the code generator needs to break the single kernel into several smaller ones, each of which only processes a portion of the data set. The objective is to separate the six plane boundaries from the internal region. While the boundaries need to be exchanged between neighboring nodes, the internal regions can be calculated completely in parallel with communication.

Our framework generates MPI calls for inter-node communication. At each iteration, each node performs the following steps:

(1) Kernels copy non-continuous boundaries residing in GPU memory into continuous GPU memory buffers. For stencils with corner accesses, eight corners and 12 edges are also copied into separate buffers. Then, continuous boundaries are transferred from GPU memory to host memory via cudaMemcpy.

(2) An asynchronous kernel updates internal regions.

(3) MPI sends and receives are issued to exchange boundaries. Once boundaries are received, boundaries are copied from host memory to GPU memory. This step can be overlapped with the step (2).

(4) Kernels update stencils on boundaries.

These steps are illustrated in Figure 4.

For GPU clusters, it is important to keep the load balanced across all computing nodes. Our Cartesian partition strategy makes sure every node receives nearly the same amount of data to process for a homogeneous GPU cluster. But if we applied the same equal-space partitioning for heterogeneous GPU clusters, more powerful GPUs would finish the calculation first and then wait for slower GPUs for each stencil iteration. Since we already obtained the GFlops rate of a code in the single node auto-tuning step, we can reuse this information for balancing partition sizes. The idea is to partition the data space into layers and assign them to groups of GPUs, where *each group* consists of GPUs with identical models. We choose the least frequently changed axis (Z-axis) to create this partition layer. All GPU groups use the same partition across the X- and Y-axes. The partition lengths on Z-axis are selected proportional to each GPU model's GFlops capability. Here, we assume that the numbers of each GPU model are integer divisible. This guarantees that planes partition the layers. In effect, the communication pattern is the same as that of a homogeneous GPU cluster. Figure 5 shows how partitioning is performed for a cluster with two types of GPUs. The top GPU group has more computational power. Its GPUs are therefore assigned to a larger data space.

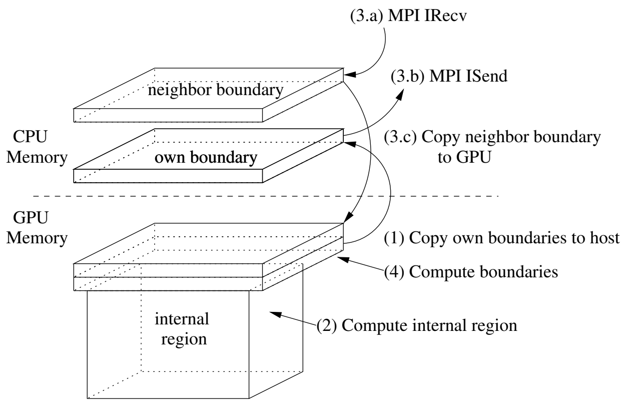


Fig. 4. Steps in multi-node scenario. For clarity, only one boundary plane is shown.

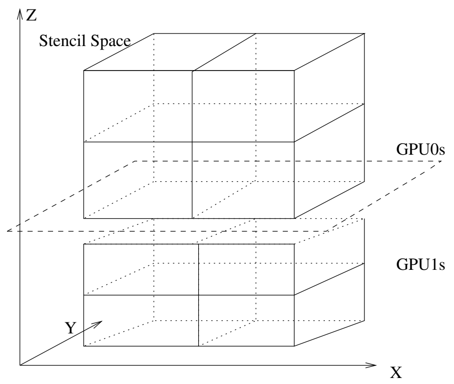


Fig. 5. Partition stencil space across different GPU types (There are two types in the graph). The space along Z-axis is assigned to GPUs according to their GFlops capabilities.

4 Experimental Results

4.1 Experimental Setup

We conducted experiments on single nodes with four NVIDIA GPU models: Geforce GTX 280, Tesla C1060, Tesla C2050 and Geforce GTX 480, spanning two generations of NVIDIA GPUs ranging from consumer-end graphics card to high-performance computing GPUs. Their main specifications are depicted in Table 1. All kernels are compiled under CUDA 4.1 with O3 optimizations. Experiments with Tesla C2050s are conducted with ECC turned off. For Fermi GPUs (Tesla C2050 and GTX 480), we prefer shared memory over L1 cache since the shared memory size is 48 KB (vs. 16 KB in earlier GPUs).

We conducted multi-node experiments on two homogeneous GPU clusters connected by fat-tree QDR Infiniband (36 Gbps). One cluster was comprised of 32 nodes, each with one Tesla C2050, the other had 48 nodes, each with one Geforce GTX480. We then combined the two GPU clusters to form a larger heterogeneous GPU cluster and applied our hybrid partition strategies.

Model	SM Count	Core Count	L1 Cache	Bandwidth(GB/s)	Register File Size	Shared Memory	SP GFlops	DP GFlops
Geforce GTX 280	30	240	N	141.7	16 KB	16KB	933	78
Tesla C1060	30	240	N	102.4	16 KB	16KB	933	78
Tesla C2050	14	448	Y	144	32 KB	16 or 48 KB	1288	515
Geforce GTX 480	15	480	Y	177.4	32 KB	16 or 48 KB	1345	168

TABLE 1
Single Node Experiment Platforms

Model	BlockSize.x	BlockSize.y	BlockDim.x	BlockDim.y	Texture	**SP GFlops**
Geforce GTX 280	64/32/64/16	8/8/3/6	32/32/64/16	8/2/3/2	Y/Y/N/N	76.0/117.0/57.6/94.2
Tesla C1060	64/64/64/32	8/6/6/8	32/64/64/32	8/2/3/2	Y/N/Y/N	57.5/91.8/44.8/95.5
Tesla C2050	64/64/64/64	8/6/3/4	32/64/32/32	8/3/3/4	Y/Y/N/Y	87.3/133.8/64.6/157.6
Geforce GTX 480	64/64/64/64	3/3/3/8	32/32/32/32	3/3/3/4	Y/Y/N/Y	108.2/167.8/77.4/203.7
Model	BlockSize.x	BlockSize.y	BlockDim.x	BlockDim.y	Texture	**DP GFlops**
Geforce GTX 280	16/16/16/16	16/16/6/6	16/16/16/16	4/8/3/3	N/N/Y/N	32.5/35.4/24.0/29.0
Tesla C1060	32/16/32/16	6/16/4/6	32/16/32/16	2/8/2/3	N/N/Y/N	28.8/35.3/22.8/29.3
Tesla C2050	64/32/64/32	8/6/3/6	32/32/64/32	4/2/3/2	Y/Y/N/Y	45.9/66.8/31.8/97.7
Geforce GTX 480	64/32/64/32	6/6/3/4	32/32/64/16	3/2/3/4	Y/Y/N/Y	55.2/77.2/38.7/86.0

TABLE 2
7/13/19/27-Point Stencil Results on Single GPU for Single/Double Precision (SP/DP)

4.2 Single Node Results

Model	**SP GFlops**	**DP GFlops**
Geforce GTX 280	22.6/24.5/23.4/21.5	12.7/13.1/11.7/11.6
Tesla C2050	70.2/67.1/41.5/70.7	30.3/29.5/18.4/30.0

TABLE 3
7/13/19/27-Point Stencil Results for Naive Kernel

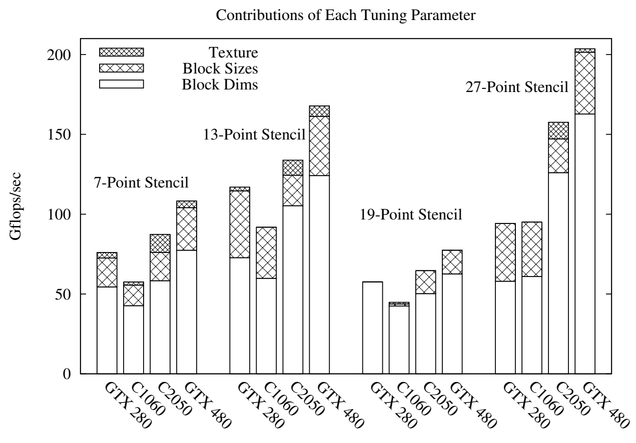
Our single-node auto-tuning engine finds the optimal parameters for all stencil types on each GPU model within the given search space. We constrain the domain size to $256 \times 256 \times 256$. These parameters are shown in Table 2. Each GPU model has different optimal settings for all stencil types, even within the same GPU generation. Almost all models favor large $BlockSize.x$ except for some cases with early generation GPUs. These older GPUs have tighter restrictions on shared memory size, especially for double precision (DP) stencils. Thus, they can only afford smaller $BlockSize.x$ sizes. $BlockSize.y$ is usually less than $BlockSize.x$, except for 7/13-point DP stencils on a GTX 280 and the 13-point DP stencil on a Tesla C1060 because their smaller $BlockSize.x$ (16) allows them to have a larger $BlockSize.y$. Thus, reducing the non-coalesced memory access (increasing $BlockSize.x$) is favored over reducing redundant loads (increasing $BlockSize.y$).

As a comparison, we also generate naive codes directly from the DSL without using any Shared Memory. Their performance on GTX 280 and Tesla C2050, after similar BlockSizes and BlockDims tuning, is shown in Table 3. The GFlops numbers are far inferior to those using Shared Memory. We also removed the padding in the X dimension for both the input and output arrays. This generated a performance loss of roughly 10%.

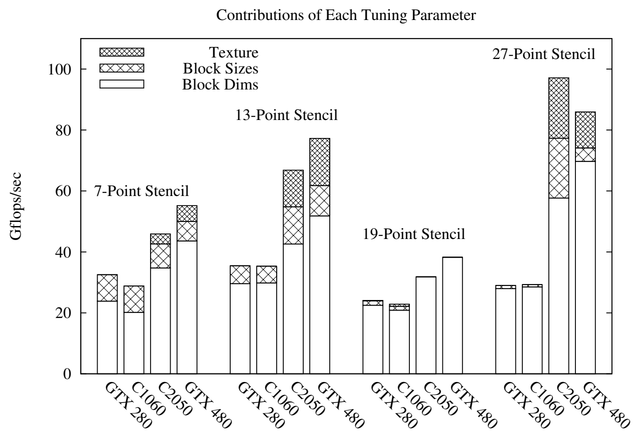
An illustration of each tuning parameter's contribution to performance is given in Figure 6. Here, auto-tuning is comprised of three steps: (1) $BlockSize.x/y$ are set to be equal to $BlockDim.x/y$; (2) $BlockSizes.x/y$ are tuned for better performance; (3) texture mapping is enabled/disabled. The necessity to unroll is confirmed by the fact that $BlockDim.x/y$ sizes are almost always different than $BlockSize.x/y$. The only exception is given by a 19-point DP stencil for Fermi GPUs. In this cases, $BlockSize.y$ is too small to unroll. In addition, Fermi GPUs provide enough registers to support a $BlockDim.x$ of the same size as $BlockSize.x$.

Another interesting observation is that mapping the input array to texture memory does not necessarily result in better performance. This is in part because some stencils are not bandwidth-limited on certain GPUs. For GPUs that have high GFlops capabilities, using texture memory usually helps because memory references are on the critical path (7/13/27-point DP stencils for C2050 and GTX 480). Using texture memory has one overhead though: Texture mapping requires the device memory to start from 128-bit aligned address. But our input/output array base addresses are shifted to non-aligned addresses so that the addresses with offset at $halo\_i$ (base address for internal region) are 128-bit aligned. Thus, an extra offset adjustment calculation is need to enable texture mapping. This extra arithmetic for address computation can negate the benefit of lower latencies for texture memory accesses for some cases.

Of the four GPU models, both Geforce GTX 280 and Tesla C1060 belong to the first generation of CUDA-enabled Nvidia GPUs (computing capability 1.x) while Tesla 2050 and GTX 480 are of the second generation, known as the Fermi architecture. The major difference within a generation is their theoretical memory bandwidth as well as DP performance (for Teslas), which lower models either lack (first generation) or only provide at a lower rate (second generation). Our GFlops rates give us insight to whether a stencil type is bandwidth-limited or computation-limited on a certain GPU. For SP stencils, GFlops rates for Teslas are almost always inferior to that of Geforce models in the same generation, except for the first generation 27-point case. And their ratio is similar to the bandwidth ratio.

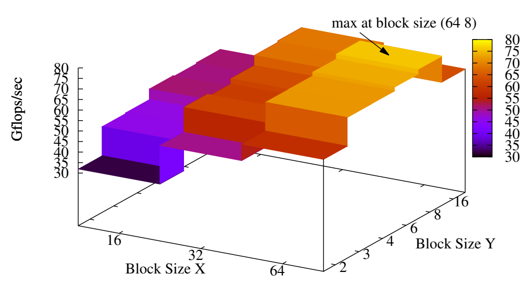


(a) Single Precision (SP) Stencils

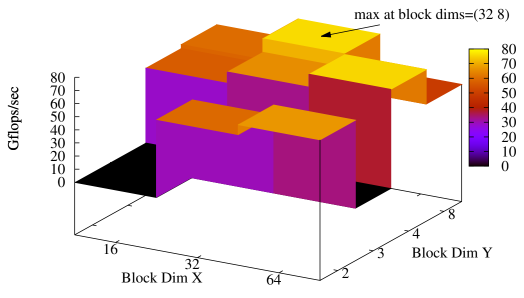


(b) Double Precision (DP) Stencils

Fig. 6. Stencil Tuning Effect Breakups

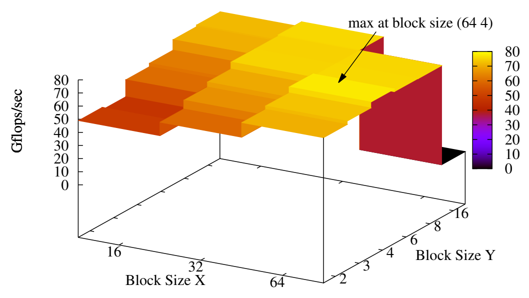


(a) Find Optimal Block Sizes

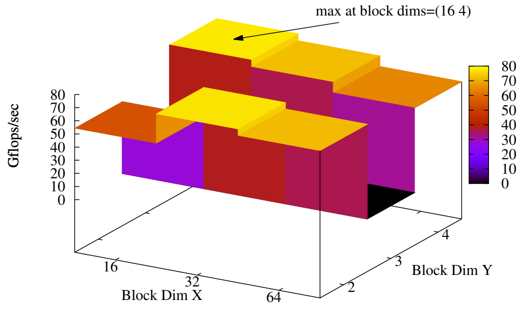


(b) Find Optimal Block Dims, Fixed Block Size at (64, 8)

Fig. 7. GTX 280 7-Point Stencil (SP)



(a) Find Optimal Block Sizes



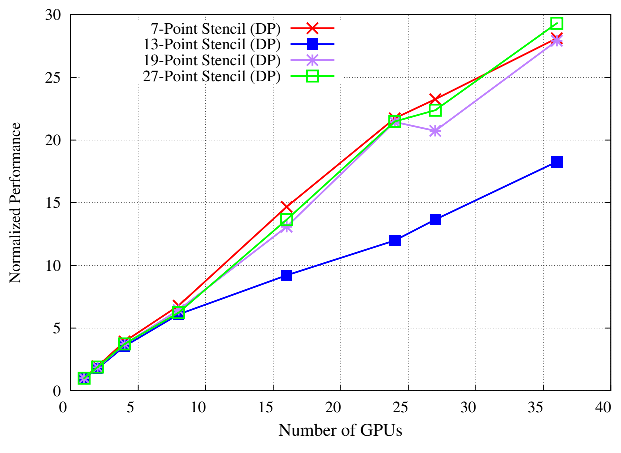
(b) Find Optimal Block Dims, Fixed Block Size at (64, 4)

Fig. 8. C2050 27-Point Stencil (DP)

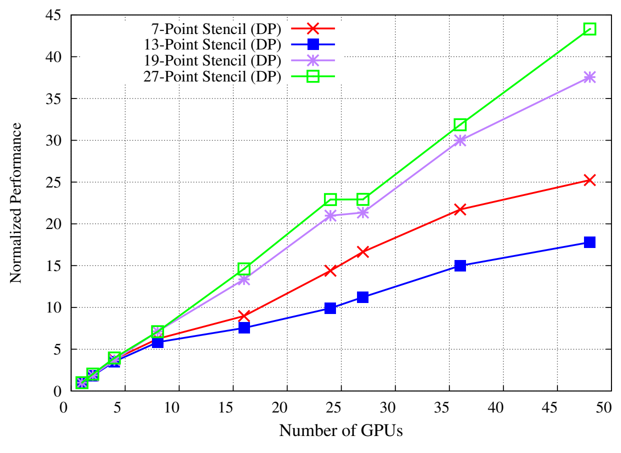
Therefore, Tesla models are bandwidth-limited in almost all SP stencils. In DP stencils, a similar ratio can be found for 7-point stencil for first generation and 7/13/19-point stencils for second generation GPUs. But for other cases, GFlops rates for Tesla models are close to or better than for Geforce models. Therefore, those stencils are computation-bounded for Geforce GPUs.

To demonstrate the effectiveness of the auto-tuning engine, we select two cases and represent performance in GFlops as a surface in a 3D histogram. Figure 7 depicts the single-precision (SP) 7-point stencil on a GTX 280. Figure 8 depicts the DP 27-point stencil on a Tesla C2050. The left diagrams in the figures illustrate how the performance changes while varying $BlockSize.x/y$, assuming the best $BlockDim.x/y$ has been found. The right diagrams in the figures depicts how the performance changes when varying $BlockDim.x/y$ for a fixed $BlockSize.x/y$ overall. The figures demonstrate that each tuning parameter plays an important role in the final performance, neither one of which can be explored independently of the other.

Our auto-tuning engine performs an exhaustive search over all possible permutations. This guarantees a global optimum with respect to the parameter search space. Adaptive search methods could be adopted to prune the search space. However, care must be taken because local optima exist, as seen in the figures. For example, in Figure 7(b), (64,4) is another locally optimal $BlockDim.x/y$ pair. Considering the search space is relatively small (a few thousands combinations), off-line exhaustive search is a feasible approach.

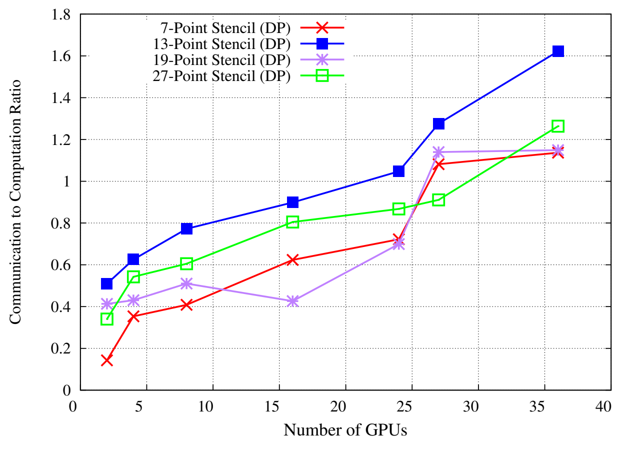


(a) C2050 cluster

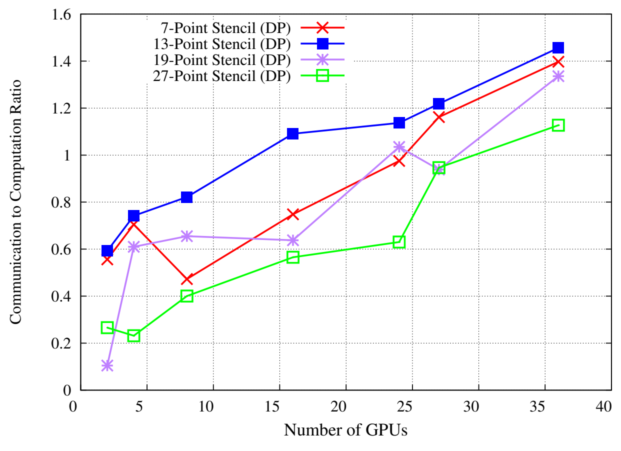


(b) GTX 480 cluster

Fig. 9. Weak Scaling of DP Stencils on GPU Clusters



(a) C2050 cluster



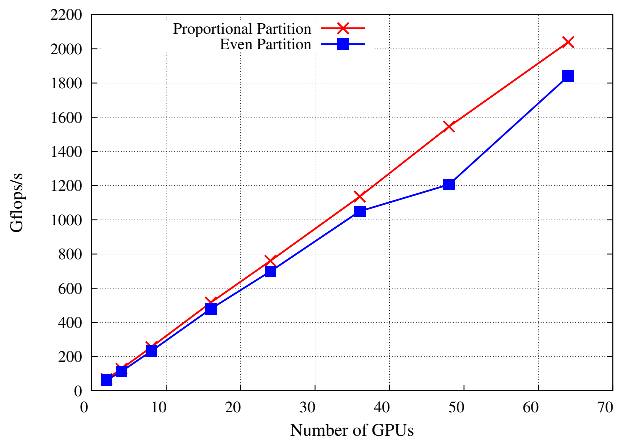
(b) GTX 480 cluster

Fig. 10. Communication to Computation Ratio

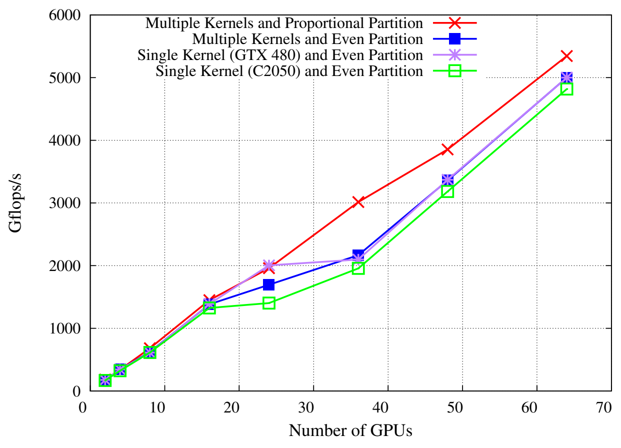
4.3 Multi-Node Results

We study the weak scaling property [14] of our framework in the two GPU clusters. We keep the problem size per GPU constant and increase the stencil size over all three dimensions at roughly the same rate as the increase in number of GPUs. Therefore, the total stencil space is kept as close to a cube as possible. Our partition strategy ensures that the stencil space of each node is shaped as a cube as well but favors the Z partition over Y and X partitions to break ties. Building a cube stencil space represents a worst case scenario because it forces partitions across X and Y axes, which leads to more message traffic. The Y axis of Figure 9 depicts the normalized performance (measured in GFlops) of a single GPU. We also profiled the message passing time (step 2 in Section 3.2) and GPU time (step 3 in Section 3.2) and plotted their ratios in Figure 10. Since step 1 and step 4 take much less time than step 2 and step 3, the implementation parallelizes step 2 and step 3. This ratio can be used to indicate whether our implementation is communication (network bandwidth) bounded or computation bounded. For the C2050 GPU cluster, all three order-1 stencils (7/19/27-point) show better scalability than order-2 stencils. That means the network has become a serious bottleneck impacting the 13-point stencil's performance. The communication to computation ratio reaches to around 1.6 in this case and only around 1.2 for other stencils (Figure 10(a)). For GTX 480 clusters, because the GTX 480 has higher single-node DP GFlops for 7/13/19-point stencils, weak scaling is worse than that on the C2050 cluster. But for 27-point stencils, GTX 480's single-node DP GFlops is less than C2050's DP GFlops. It exerts less pressure on the network. Therefore, scalability is better. Of both scenarios, the 13-point stencil always has the worst scalability among all stencil types. This can be explained by the difference in inter-node message sizes required by different stencils types. The message size is roughly proportional to the size of the stencil order. Therefore, our 13-point stencil is communication-bound in our current cluster configuration.

Some of the curves do not show a noticeable improvement from 24 to 27 GPUs (nodes). The 19-point stencil curve even shows a slight drop. This is because the stencil space is divided into $2 \times 3 \times 4$ and $3 \times 3 \times 3$ partitions in these two cases, respectively. The latter case contains a center node that needs to communicate with all other 26 nodes. This node becomes a hot-spot and reduces the performance. But as we increase the number of GPUs, the curve recovers to the expected slope for weak scaling.



(a) 19-point DP



(b) 27-point DP

Fig. 11. Performance Results on Heterogeneous GPU cluster (The ratio of two GPUs is 1:1 in all experiments)

To demonstrate the effectiveness of auto-tuning and proportional partitioning on heterogeneous GPU clusters, we compare the GFlops in three different setups:

- Multiple Kernels and Proportional Partitioning: We generate separate kernels with the auto-tuned parameters for each GPU type and divide stencil space according to their GFlops capabilities. This is the optimal setup.
- Multiple Kernels and Even Partitioning: We use optimized kernels for each GPU type but evenly divide the stencil space among all GPU types.
- Single Kernel and Even Partitioning: We use just one kernel for all GPU types and evenly divide the stencil space. Since we have two types of GPUs in our cluster, we test two different kernels (for each of the single-GPU optimal parameters), unless they have the same parameter settings.

Figure 11 shows the GFlops/s rate versus the number of GPUs in 19-point and 27-point DP stencil code. Because the optimal parameter setting for both GTX480 and C2050 are the same for 19-point stencil (see Table 2), they can share the same stencil kernels to achieve the best performance. The last two setups become identical in this case. Figure 11(a) demonstrates that proportional partitioning is always superior to even partitioning. The first (optimal) setup produces the best GFlops/s rate among all four curves in Figure 11(b). Because a single C2050 performs better than a GTX480 for a 27-point stencil (97.7 GFlops versus 86 GFlops), GTX480s are on the critical path in the heterogeneous cluster. This explains why kernels optimized for GTX480 outperform kernels optimized for C2050. The performance difference diminishes for larger scales in both figures. This can be explained as follows: The larger the scale, the greater is the communication to computation ratio (the total message size in the system increases but computation is kept constant here) and the less important it becomes to reduce computation time discrepancies.

5 Conclusion

This paper shows that GPU programmability and performance are not mutually exclusive under DSLs. With a DSL specification fed to the front-end, problem descriptions can become very concise and intuitive. Using auto-tuning with run-time profile feedback, optimal tuning points within the parameter search space can be identified. Our framework combines auto-generation and auto-tuning of 3D stencil codes on heterogeneous GPU clusters. We extract a small, selective number of key performance-sensitive parameters and auto-tune them to achieve the best possible performance over a variety of GPUs. Compared to previous work, we manage to keep the programmer's effort to even a lower overhead without significant sacrifice in performance. We also show that heterogeneous GPU clusters exhibit the when leveraging *proportional partitioning* of the data space relative to single-GPU performance.

6 References

[1] NVIDIA Cooperation, CUDA Programming Guide.

[2] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30(2):41–55, Mar. 2010.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, 2008.

[4] http://www.khronos.org/opencl. OpenCL.

[5] W.-m. Hwu, S. Ryoo, S.-z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly Parallel Programming Models for Thousand-core Microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, pages 754–759, 2007.

[6] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *In IEEE Intl Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[7] S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawa. GPU Accelerated Computing from Hype to Mainstream, the Rebirth of Vector Computing. In *Journal of Physics: Conference Series 180*, 2009.

[8] P. Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, 2009.

[9] E. Phillips and M. Fatica. Implementing the Himeno Benchmark with CUDA on GPU Clusters. In *International Parallel and Distributed Processing Symposium(IPDPS)*, Apr 2010.

7 APPENDIX

```
typedef struct {
  int dims[3];
  int iter;
  int haloMargins[2][3];
  ...
  int numNodes; // for multi-node
  int curNode;// for multi-node
} StencilConfig;

// Sample user code
// Also our profiling routine
int main(int argc, char **argv) {
  StencilConfig config;
  config.iter = 0;
  config.dims[0] = 256; ... // more init.
  initStencil(&config);
  while(config.iter < 100) // run 100 iterations
    stencilIteration(_mpi)(&config);

  exitStencil(&config);
}
```

Fig. 12. Example of Auto-Generated Code (Excerpts)

```
// Auto-generated indexes
__const__ __device__ int haloThreadMappingX[4][6] =
     {{1,2,3,4,5,6}, {1,2,3,4,5,6}, {0,0,0,0,7,7},
     {7,7,0,0,0,0}};
__const__ __device__ int haloThreadMappingY[4][6] =
     {{0,0,0,0,0,0}, {5,5,5,5,5,5}, {1,2,3,4,1,2},
     {3,4,0,0,0,0}};


// Only show code that load input array to Shared Memory
__global__ stencil(...) {
    ...
  halo_offsetx = haloThreadMappingX[threadIdx.x][
       threadIdx.y];
  halo_offsety = haloThreadMappingY[threadIdx.x][
       threadIdx.y];
  int myindexX = g_tx + threadIdx.x;
  int myindexY = g_ty + threadIdx.y;
  for (k = HALO_MARGIN_Z; k < zSize + HALO_MARGIN_Z; k++)
       {
         ....
         // load internal
         shArr[threadIdx.y][threadIdx.x] = input[myindexY
              ][myindexX];
         // load margins, no branches
         shArr[halo_offsetx][halo_offsety] = input[
              myindexY+halo_offsety-HALO_MARGIN_Y][
              myindexX+halo_offsetx-HALO_MARGIN_X];
       ...
       }
}
```

Fig. 14. Auto-generated CUDA source code for Fig 15(a)

8 RELATED WORK

Auto-tuning has long been identified as an effective approach to offer portability and productivity. For example, ATLAS [10], OSKI [23] and FFTW [12] are well recognized auto-tuning libraries targeted at general-purpose processors for dense/sparse linear algebra subroutines and FFT kernels in digital signal processing, respectively.

Recent improvements in programmability of GPUs allow auto-tuning to be applied to GPUs as well. Several CUDA implementations for linear algebra subroutines and FFTs with auto-tuning capability already exist [13], [15], [20].

Previous implementations of stencil computations on GPUs can be grouped into three categories in terms of their emphasis: (1) Hand-coded implementations of a particular stencil strive to achieve the best performance possible [8], [19], [9], but some of their optimization techniques do not even generalize to other types of stencils. (2) Ease of programming is chosen as the primary goal over performance. Such works usually contain code generators for various kind of stencils [3], [6], [17], [22]. (3) Other work focuses on a particular parameter and studies how its tuning can affect performance [16], [18].

We conjecture that performance or programmability are not mutually exclusive. The merit of our work is to offer both ease of programming and performance at the same time. By providing a stencil specification front-end, we alleviate the end-user's burden to master architectural details. Near-optimal performance is achieved by extracting necessary parameters and thoroughly auto-tuning them. Even though some of the aforementioned work utilizes certain tuning parameters, such work either relies on ad-hoc hand tuning [22] or the tuning space is limited [3], [6].

We report our results on a wide range of GPUs and stencil types, which allows us to compare our performance directly with a wide range of prior work, both for hand-written and auto-generated codes.

Datta *et al.*'s work on optimizing stencil codes in multi-core architectures including GPUs is one of the early contributions in this area [3]. They showed an unprecedented 36 GFlops for 7-point stencil on a GTX 280 with their highly optimized code. Theirs is 10% faster than our performance (32.5 GFlops). This is mainly due to the difference between the instruction orders in our template file and their hand-tuned kernel code, as we discovered by inspecting their and our codes side-by-side. But interestingly, their best performance is achieved at a block size of 16×16 and unroll factor of 4 over the dimension Y, which is consistent to our findings in our auto-tuning engine. However, this configuration is *only* optimal for a DP 7-point stencil on the GTX 280s. For everything else, the 16×16 block sizes are no longer optimal, as indicated by Table 2.

An efficient and handwritten CUDA implementation on the Himeno benchmark is reported by Philips *et al.* [9]. Their implementation, with an extra two Flops per stencil for residual calculation, achieved 50 GFlops SP on a Tesla C1060. Our auto-generated code achieves 44.8 GFlops on the same platform and is within 5% to theirs if Flops are normalized ($44.8 \times \frac{34}{32} = 47.6$). Their best block sizes are 64×2 for Tesla C1060, while ours is 64×6 with an unrolling factor of 2 over the Y axis. This is because they load the input arrays into shared memory by issuing four branch-free loads aligned at four corners. Choosing

Kernel	Specification	# array params	Flops per stencil	mem. refs per stencil
7-point order-1	$tmp = (u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1}) * beta;$ $u1_{i,j,k} = tmp + alpha * u_{i,j,k};$	0	8	8
13-point order-2	$tmp = coef1 * (u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1});$ $tmp+ = coef2 * (u_{i+2,j,k} + u_{i-2,j,k} + u_{i,j+2,k} + u_{i,j-2,k} + u_{i,j,k+2} + u_{i,j,k-2});$ $u1_{i,j,k} = tmp + coef0 * u_{i,j,k};$	0	15	14
19-point order-1 (himeno)	$s0 = wrk1_{i,j,k} + a0d_{i,j,k} * p_{i,j,k+1} + a1d_{i,j,k} * p_{i,j+1,k+1};$ $s0+ = b0d_{i,j,k} * (p_{i,j+1,k+1} - p_{i,j-1,k+1} - p_{i,j+1,k-1} + p_{i,j-1,k-1}) + a2d_{i,j,k} * p_{i+1,j,k};$ $s0+ = b1d_{i,j,k} * (p_{i+1,j+1,k} - p_{i-1,j+1,k} - p_{i+1,j-1,k} + p_{i-1,j-1,k});$ $s0+ = b2d_{i,j,k} * (p_{i+1,j,k+1} - p_{i-1,j,k+1} - p_{i+1,j,k-1} + p_{i-1,j,k-1}) + c0d_{i,j,k} * p_{i,j,k-1};$ $s0+ = c1d_{i,j,k} * p_{i,j-1,k} + c2d_{i,j,k} * p_{i-1,j,k};$ $ss = (s0 * a3d_{i,j,k} - p_{i,j,k}) * bnd_{i,j,k};$ $wrk2_{i,j,k} = p_{i,j,k} + omega * ss;$	12	32	32
27-point order-1	$b_{i,j,k} = param0 * a_{i,j,k}$ $+param1 * (a_{i-1,j,k} + a_{i+1,j,k} + a_{i,j-1,k} + a_{i,j+1,k} + a_{i,j,k-1} + a_{i,j,k+1})$ $+param2 * (a_{i-1,j-1,k} + a_{i-1,j+1,k} + a_{i+1,j-1,k} + a_{i+1,j+1,k} + a_{i-1,j,k-1} + a_{i-1,j,k+1}$ $+a_{i+1,j,k-1} + a_{i+1,j,k+1} + a_{i,j-1,k-1} + a_{i,j-1,k+1} + a_{i,j+1,k-1} + a_{i,j+1,k+1})$ $+param3 * (a_{i-1,j-1,k-1} + a_{i-1,j-1,k+1} + a_{i-1,j+1,k-1} + a_{i-1,j+1,k+1}$ $+a_{i+1,j-1,k-1} + a_{i+1,j-1,k+1} + a_{i+1,j+1,k-1} + a_{i+1,j+1,k+1});$	0	30	28

TABLE 4
Specifications of Four Stencil Benchmarks. Indices are subscripted to save space.

$BlockSize.y$ as 2, in their case, minimizes redundant memory loads, which is beneficial because SP Himeno is bandwidth limited on the C1060. They also reported near-perfect weak scaling efficiency on up to 16 GPUs. But their system configuration is different from ours: (1) Each node has two GPUs instead of one in our case. Therefore, half of the network messages become memory copies on the same host. (2) The stencil space only grows along the Z axis, eliminating the need to perform Cartesian partitioning. This reduces the multi-node code complexity significantly.

Kamil *et al.* proposed an auto-tuning framework for multi-core architectures [6]. However, they reported only 14 GFlops DP on a 7-point stencil for a GTX 280. This is mainly because their code generator does not take advantage of the fast on-chip shared memory, which is an ideal intermediate storage level to reduce memory load for stencil-like computations.

Nguyen *et al.* have reported by far the fastest implementation of any SP stencil code on single GPU [19]. Their manually-written code for a 7-point stencil achieves 136 GFlops on GTX 285 (a similar platform as GTX 280), a large gain over our reported 76 GFlops. However, their extra speedup comes from saving a large amount of global memory accesses by exploiting data locality on the time domain. This is equivalent to executing several iterations per kernel, a technique also known as increasing the ghost region. Increasing the ghost region leads to less frequent message exchanges but does not reduce the total amount of data transferred in the network because the payload for each message increases as well. It has been shown to be insignificant in multi-node scenarios due to the slower inter-node communication [21]. Therefore, we decided not to include ghost region sizes/update frequencies as a tuning parameter in our code generator and auto-tuning schemes. For DP stencils, their performance is no better than [3] due to limitations in shared memory size of the GTX 285.

Unat *et al.* proposed a compiler framework called Mint using annotated C as the front-end. It converts stencil computation into C code using pragmas with several levels of optimized CUDA code [22]. Our DP performance of a 7-point stencil on the C1060 achieves the same GFlops as their hand-written code (28 GFlops). In contrast, auto-generated Mint code with the highest level optimization achieves only 22 GFlops.

Christen *et al.* [11] and Maruyama *et al.* [17] proposed two DSLs: Patus and Physis. Patus purely depends on the cache on the Fermi architecture without using any shared memory. Therefore, its auto-tuning capability is severely limited. Physis currently lacks any auto-tuning scheme, one has to choose block sizes manually. Both report SP performance inferior to ours.

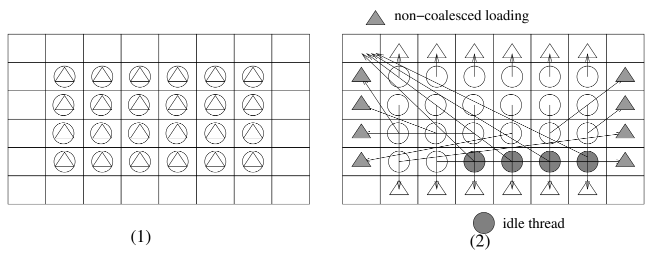
```
#define sizey (BLOCK_Y+halo_j*2)
#define sizex (BLOCK_X+halo_i*2)

template <class T>
__global__ stencil_iteration(...) {
  // Initialization Instructions
  g_tx = blockIdx.x * BLOCK_X;
  g_ty = blockIdx.y * BLOCK_Y;
  ...
  __shared__ T shArr[3][sizey][sizex];
  first = 0; second = 1; third = 2;
  shArr[0][][] = ; // Load first 2 planes
  shArr[1][][] = ;
  if (g_tx + BLOCK_X <= max_dimx && g_ty + BLOCK_Y <=
       max_dimy)
  {  // all unrolls fit in the max X/Y dimension
    for (k=halo_k; k<=zSize; k++) {
      // Load third plane to __shared__
      shArr[2][][] = ;
      __syncthreads();
      { // stencil calc., unrolled, w/o boundary check
        ...
      }
      __syncthreads();
      first = (first+1)%3; // Shift planes
      second = (second+1)%3;
      third = (third+1)%3;
  } } else {
    for (k=halo_k; k<=zSize; k++) {
      // Load third plane to __shared__
      shArr[2][][] = ;
      __syncthreads();
      { // stencil calc., unrolled with boundary check
        if (g_tx + threadIdx.x < max_dimx && g_ty +
             threadIdx.y < max_dimy)
          ...
      }
      __syncthreads();
      first = (first+1)%3; // Shift planes
      second = (second+1)%3;
      third = (third+1)%3;
} } }
```

(a) With Corner Accesses

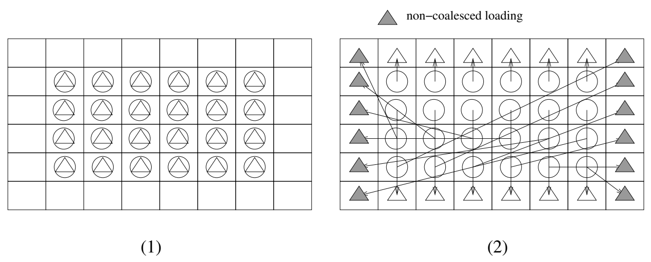
```
#define sizey (BLOCK_Y+halo_j*2)
#define sizex (BLOCK_X+halo_i*2)

template <class T>
__global__ stencil_no_corner(...) {
  // Initialization Instructions
  g_tx = threadIdx.x + blockIdx.x * BLOCK_X;
  g_ty = threadIdx.y + blockIdx.y * BLOCK_Y;
  ...
  __shared__ T shArr[sizey][sizex];
  T middle = ...;  T below = ...; // Load first 2 planes
      to registers
  if (g_tx + BLOCK_X <= max_dimx && g_ty + BLOCK_Y <=
      max_dimy)
  {  // all unrolls fit in the max X/Y dimension
    for (k=halo_k; k<=zSize-halo_k; k++) {
      top = middle; // Shift registers
      middle = below;
      // load third plane to registers
      T below = ...;
      __syncthreads();
      // load middle plane to __shared__
      ...
      __syncthreads();
      { // stencil calc., unrolled, w/o boundary check
        ...

  } } } else {
    for (k=halo_k; k<=zSize-halo_k; k++) {
      top = middle; // Shift registers
      middle = below;
      // load third plane to registers
      T below = ...;
      __syncthreads();
      // load middle plane to __shared__
      ...
      __syncthreads();
      { // stencil calc., unrolled, with boundary check
        if (g_tx + threadIdx.x < max_dimx && g_ty +
             threadIdx.y < max_dimy)
          ...
} } } }
```

(b) Without Corner Accesses

Fig. 13. Stencil Kernel Templates



(a) No Corner Case



(b) With Corner Case

Fig. 15. Load input sub-plane to shared memory. Internal regions are loaded in Step (1). There is a one-to-one mapping between computing threads and internal regions. In Step (2), the mapping is auto-generated by the parameter tuning engine. (A circle denotes a thread. A triangle denotes an array element loaded at the current step.)

9 References for the Appendix

[10] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27:2001, 2000.

[11] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. *In IEEE Intl Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[12] M. Frigo. A Fast Fourier Transform Compiler. *SIGPLAN Not.*, 39:642–655, April 2004.

[13] P. Guo and L. Wang. Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs. *Computational and Information Sciences, International Conference on*, 0:1154–1157, 2010.

[14] J. L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31:532–533, May 1988.

[15] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, pages 884–892, 2009.

[16] Z. Li and Y. Song. Automatic Tiling of Iterative Stencil Loops. *ACM Trans. Program. Lang. Syst.*, 26:975–1028, November 2004.

[17] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. 2011.

[18] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, 2009.

[19] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International*

Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–13, 2010.

[20] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 30:1–30:10, 2009.

[21] M. Ripeanu, A. Iamnitchi, and IanFoster. Cactus Application: Performance Predictions in Grid Environments. In *In proceedings of European Conference on Parallel Computing (EuroPar) 2001*, 2001.

[22] D. Unat, X. Cai, and S. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the 25th International Conference on Supercomputing (ICS'11)*, 2011.

[23] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of Automatically Tuned Sparse Matrix Kernels. In *Institute of Physics Publishing*, 2005.