

Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs

Jiayuan Meng, Kevin Skadron
Department of Computer Science
University of Virginia

ABSTRACT

Iterative stencil loops (ISLs) are used in many applications and tiling is a well-known technique to localize their computation. When ISLs are tiled across a parallel architecture, there are usually halo regions that need to be updated and exchanged among different processing elements (PEs). In addition, synchronization is often used to signal the completion of halo exchanges. Both communication and synchronization may incur significant overhead on parallel architectures with shared memory. This is especially true in the case of graphics processors (GPUs), which do not preserve the state of the per-core L1 storage across global synchronizations. To reduce these overheads, ghost zones can be created to replicate stencil operations, reducing communication and synchronization costs at the expense of redundantly computing some values on multiple PEs. However, the selection of the optimal ghost zone size depends on the characteristics of both the architecture and the application, and it has only been studied for message-passing systems in a grid environment. To automate this process on shared memory systems, we establish a performance model using NVIDIA’s Tesla architecture as a case study and propose a framework that uses the performance model to automatically select the ghost zone size that performs best and generate appropriate code. The modeling is validated by four diverse ISL applications, for which the predicted ghost zone configurations are able to achieve a speedup no less than 98% of the optimal speedup.

1. INTRODUCTION

Iterative stencil loops (ISL) [21] are widely used in image processing, data mining, and physical simulations. ISLs usually operate on multi-dimensional arrays, with each element computed as a function of some neighboring elements. These neighbors comprise the *stencil*. Multiple iterations across the array are usually required to achieve convergence and/or to simulate multiple time steps. Tiling [16, 26] is often used to partition the stencil loops among multiple processing elements (PEs) for parallel execution, and we refer to a workload partition as a *tile* in this paper. Similar tiling techniques also help localize computation to optimize cache hit rate for an individual processor [11].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS’09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...\$5.00.

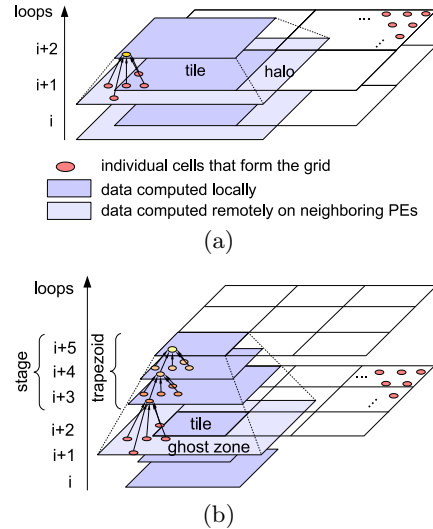


Figure 1: (a) Iterative stencil loops and halo regions. (b) Ghost zones help reduce inter-loop communication.

Tiling across multiple PEs introduces a problem because stencils along the boundary of a tile must obtain values that were computed remotely on other PEs, as shown in Figure 1(a). This means that ISL algorithms may spend considerable time stalled due to inter-loop communication and synchronization delays to exchange these *halo* regions. Instead of incurring this overhead after every iteration, a tile can be enlarged to include a *ghost zone*. This ghost zone enlarges the tile with a perimeter overlapping neighboring tiles by multiple halo regions, as shown in Figure 1(b). The overlap allows each PE to generate its halo regions locally [29] for a number of iterations proportional to the size of the ghost zone. As Figure 1(b) demonstrates, ghost zones group loops into *stages*, where each stage operates on overlapping stacks of tiles, which we refer to as *trapezoids*. Trapezoids still produce non-overlapping data at the end, and their height reflects the ghost zone size.

Ghost zones pose a tradeoff between the cost of redundant computation and the reduction in communication and synchronization among PEs. This tradeoff remains poorly understood. Despite the ghost zone’s potential benefit, an improper selection of the ghost zone size may negatively impact the overall performance. Previously, optimization techniques for ghost zones have only been proposed in message-passing based grid environment [29]. These techniques no longer fit for modern chip multiprocessors (CMPs) for two

reasons. First, communication in CMPs is usually based on shared memory and its latency model is different from that of message-passing systems. Secondly, the optimal ghost zone size is commonly one on a grid environment [1], allowing for a reasonably good initial guess for adaptive methods. However, this assumption may not hold on some shared memory CMPs, as demonstrated by our experimental results later in the paper. As a result, the overhead of the adaptive selection method may even undermine performance. This paper presents the first technique that we know of to automatically select the optimal ghost zone size for ISL applications executing on a shared-memory multicore chip multi-processor (CMP). It is based on an analytical model for optimizing performance in the presence of this tradeoff between the costs of communication and synchronization versus the costs of redundant computation.

As a case study for exploring these tradeoffs in manycore CMPs, we base our analysis on the architectural considerations posed by NVIDIA’s Tesla GPU architecture [24]. We choose GPUs because they contain so many cores and some extra complexity regarding the L1 store and global synchronization. In particular, the use of larger ghost zones is especially valuable in the Tesla architecture, because unlike the grid environment where a tile’s data may persist in its PE’s local memory, data in all PEs’ local memory has to be flushed to the globally shared memory and reloaded again after the inter-loop communication. Our performance model and its optimizations are validated by four diverse CUDA applications consisting of dynamic programming, an ordinary differential equation (ODE) solver, a partial differential equation (PDE) solver, and a cellular automaton. The optimized ghost zone sizes are able to achieve a speedup no less than 98% of the optimal configuration. Our performance model for local-store based memory systems can be extended for cache hierarchies given appropriate memory modeling such as that proposed by Kamil et al. [17].

Our performance model can adapt to various ISL applications. In particular, we find that ghost zones benefit more for ISLs with narrower halo widths, lower computation/communication ratios, and stencils operating on lower-dimensional neighborhood. Moreover, although the Tesla architecture limits the size of thread blocks, our performance model predicts that the speedup from ghost zones tends to grow with larger tiles or thread blocks.

Finally, we propose a framework template to automate the implementation of the ghost zone technique for ISL programs in CUDA. It uses explicit code annotation and implicitly transforms the code to that with ghost zones. It then performs a one-time profiling for the target application with a small input size. The measurement is then used to estimate the optimal ghost zone configuration, which is valid across different input sizes.

In short, the main contributions of this work are a method for deriving an ISL’s performance as a function of its ghost zone, a gradient descent optimizer for optimizing the ghost zone size, and a method for the programmer to briefly annotate conventional ISL code to automate finding and implementing the optimal ghost zone.

2. RELATED WORK

Tiling is a well-known technique to optimize ISL applications [26, 16, 30, 18, 25]. In some circumstances, compilers are able to tile stencil iterations to localize computation or/and exploit parallelism [11, 32]. On the other hand, APIs such as OpenMP are able to tile stencil loops at run-time and execute the tiles in parallel [7]. Renganarayana et al. explored the best combination of tiling strategies that optimizes both cache locality and parallelism [27]. Researchers have also investigated automatic tuning for tiling stencil

computations [21, 8]. Specifically, posynomials have been widely used in tile size selection [28]. However, these techniques do not consider the ghost zone technique that reduces the inter-tile communication. Although loop fusion [23] and time skewing [33] are able to generate tiles that can execute concurrently with improved locality, they cannot eliminate the communication between concurrent tiles if more than one stencil loops are fused into one tile. This enforces bulk-synchronous systems, such as NVIDIA’s Tesla architecture, to frequently synchronize computation among different PEs, which eventually penalizes performance.

Ghost zones are based on tiling and they reduce communication further by replicating computation, whose purpose is to replicate and distribute data to where it is consumed [10]. Krishnamoorthy et al. proposed overlapped tiling that employs ghost zones with time skewing and they studied its effect in reducing the communication volume [19]. However, their static analysis does not consider latencies at run-time and therefore the optimal ghost zone size cannot be determined to balance the benefit of reduced communication and the overhead of redundant computation.

Ripeanu et al. constructed a performance model that can predict the optimal ghost zone size [29], and they conclude the optimal ghost zone size is usually one in grid environment. However, the performance model is based on message-passing and it does not model shared memory systems. Moreover, their technique is not able to make case-by-case optimizations — it predicts the time spent in parallel computation using time measurement of the sequential execution. This obscures the benefit of optimization — an even longer sequential execution is required for every different input size even it is the same application running on the same platform.

Alternatively, Allen et al. proposed adaptive selection of ghost zone sizes which sets the ghost zone size to be one initially and increases or decreases it dynamically according to run-time performance measurement [1]. The technique works fine in grid environment because the initial guess of the ghost zone size is usually correct or close to the optimal. However, our experiments on NVIDIA’s Tesla architecture show that the optimal ghost zone size varies significantly for different applications or even different tile sizes. Therefore an inaccurate initial guess may lead to long adaptation overhead or even performance degradation, as demonstrated in Section 5.4. Moreover, the implementation of the adaptive technique is application-specific and it requires nontrivial programming effort.

The concept of computation replication involved in ghost zones is related to data replication and distribution in the context of distributed memory systems [3, 20], which are used to wisely distribute *existing* data across processor memories. Communication-free partitioning has been proposed for multiprocessors as a compiling technique based on hyperplane, however, it only covers a narrow class of stencil loops [14]. To study the performance of 3-D stencil computations on modern cache-based memory systems, another performance model is proposed by Kamil et al. [17] which is used to analyze the effect of cache blocking optimizations. Their model does not consider ghost zone optimizations.

An implementation of ghost zones in CUDA programming is described by Che et al. [4]. The same technique can be used in other existing CUDA applications ranging from fluid dynamics [12] to image processing [34].

Another automatic tuning framework for CUDA programs is CUDA-lite [31]. It uses code annotation to help programmers select what memory units to use and transfer data among different memory units. While CUDA-lite performs general optimizations and generates code with good performance, it does not consider the trapezoid technique which serves as an ISL-specific optimization.

3. GHOST ZONES ON GPUS

We show an example of ISLs and illustrate how ghost zones are optimized in a grid environment. We then introduce CUDA programming and NVIDIA’s Tesla architecture and show how ghost zones are implemented in a different system platform.

3.1 Ghost zones in Grid Environment

Listing 1: Simplified HotSpot code as an example of iterative stencil loops

```

/* A and B are two 2-D ROWS x COLS arrays      *
 * B points to the array with values produced *
 * previously, and A points to the array with *
 * values to be computed.                    */
float **A, **B;
/* iteratively update the array                */
for k = 0 : num_iterations
  /* in each iteration, array elements are *
   * updated with a stencil in parallel */
  for_all i = 0 : ROWS-1 and j = 0 : COLS-1
    /* define indices of the stencil and *
     * handle boundary conditions by *
     * clamping overflow indices to *
     * array boundaries                */
    top = max(i-1, 0);
    bottom = min(i+1, ROWS-1);
    left = max(j-1, 0);
    right = min(j+1, COLS-1);
    /* compute the new value using the *
     * stencil (neighborhood elements *
     * produced in the previous iteration) */
    A[i][j] = B[i][j] + B[top][j] \
              + B[bottom][j] + B[i][left] \
              + B[i][right];
  swap(A, B);

```

Listing 1 shows a simple example of ISLs without ghost zones. A 2-D array is updated iteratively and in each loop, values are computed using stencils that include data elements in the upper, lower, left and right positions. Computing boundary data, however, may require values outside of the array range. In this case, the values’ array indices are clamped to the boundary of the array dimensions. When parallelized in a grid environment, each tile has to exchange with its neighboring tiles the halo regions, which is comprised of one row or column in each direction. Using ghost zones, multiple rows and columns are fetched and they are used to compute halo regions locally for subsequent loops. For example, if we wish to compute an $N \times N$ tile for two consecutive loops without communicating among PEs, each PE should start with a $(N + 4) \times (N + 4)$ data cells that overlaps each neighbor by $2N$ cells. At the end of one iteration it will have computed locally (and redundantly) the halo region that would normally need to be fetched from its neighbors, and the outermost cells will be invalid. The remaining $(N + 2) \times (N + 2)$ valid cells are used for the next iteration, producing a result with $N \times N$ valid cells.

3.2 CUDA and the Tesla Architecture

To study the effect of ghost zones on large-scale shared memory CMPs, we program several ISL applications in CUDA, a new language and development environment from NVIDIA that allows execution of general purpose applications with thousands of data-parallel threads on NVIDIA’s Tesla architecture. CUDA abstracts the GPU hardware using a few simple abstractions [24]. As Figure 2 shows, the hardware model is comprised of several streaming multiprocessors (SMs), all sharing the same device memory (the global memory on the GPU card). Each of these SMs consists of

a set of scalar processing elements (SPs) operating in SIMD lockstep fashion as an array processor. In the Tesla architecture, each SM consists of 8 SPs, but CUDA treats the SIMD width or “warp size” as 32. Each warp of 32 threads is therefore quad-pumped onto the 8 SPs. Each SM is also deeply multithreaded, supporting at least 512 concurrent threads, with fine-grained, zero-cycle context-switching among warps to hide memory latency and other sources of stalls.

In CUDA programming, a *kernel function* implements a parallel loop by mapping the function across all points in the array. In general, a separate thread is created for each point, generating thousands or millions of fine-grained threads. The threads are further grouped into a grid of *thread blocks*, where each thread block consists of at most 512 threads, and each thread block is assigned to a single SM and executes without preemption. Because the number of thread blocks may exceed (often drastically) the number of SMs, thread blocks are mapped onto SMs as preceding thread blocks finish. This allows the same program and grid to run on GPU of different sizes or generations.

The CUDA virtual machine specifies that the order of execution of thread blocks within a single kernel call is undefined. This means that communication between thread blocks is not allowed within a single kernel call. Communication among thread blocks can only occur through the device memory, and the relaxed memory consistency model means that a global synchronization is required to guarantee the completion of these memory operations. However, the threads within a single thread block are guaranteed to run on the same SM and share a 16 KB software controlled local store or scratchpad. This has gone by various names in the NVIDIA literature but the best name appears to be *per-block shared memory* or PBSM. Data must be explicitly loaded into the PBSM or stored to the device memory.

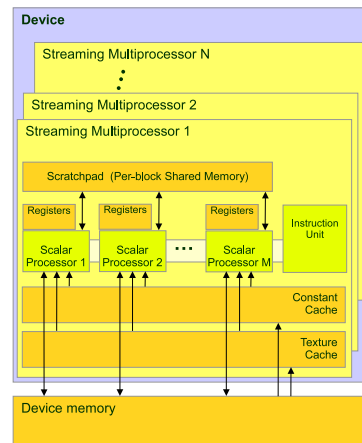


Figure 2: CUDA’s shared memory architecture. Courtesy of NVIDIA.

3.3 Implementing Ghost zones in CUDA

Without ghost zones, a thread block in CUDA can only compute one stencil loop because gathering data produced by another thread block requires the kernel function to store the computed data to the device memory, restart itself, and reload the data again. Different from the case in grid environment, where each tile only has to fetch halo regions, all data in PBSM is flushed and all has to be reloaded again. Moreover, thread blocks often contend for the device memory bandwidth. Therefore, the penalty of inter-loop communication is especially large. Using ghost zones, a thread

block is able to compute an entire trapezoid that spans several loops without inter-loop communication. At least three alternative implementations are possible.

First, as the tile size decreases loop after loop in each trapezoid, only the stencil operations within the valid tile are performed. However, the changing boundary-testing increases the amount of computation and leads to more control-flow divergence within warps, which undermines SIMD performance.

Alternatively, a trapezoid can be computed as if its tiles do not shrink along with the loops. At the end, only those elements that fall within the boundary of the shrunk tile are committed to the device memory. This method avoids frequent boundary-testing at the expense of unnecessary stencil operations performed outside the shrunk tiles. Nevertheless, experiments show that this method performs best among all, and we base our study upon this method although we can model other methods equally well.

Finally, the Tesla architecture imposes a limit on the thread block size, which in turn limits the size of a tile if one thread operates on one data element. To allow larger tiles for larger trapezoids, a CUDA program can be coded in a way that one thread computes multiple data elements. However, this complicates the code significantly and experiments show that the increased number of instructions cancels out the benefit of ghost zones and this method performs the worst among all.

4. MODELING METHODOLOGY

We build a performance model in order to analyze the benefits and limitations of ghost zones used in CUDA. It is established as a series of a multivariate equations and it can demonstrate the sensitivity of different variables. The model is validated using four diverse ISL programs with various input data.

4.1 Performance Modeling for Trapezoids on CUDA

The performance modeling has to adapt to application-specific configurations including the shape of input data and halo regions. For stencil operations over a D -dimensional array, we denote its length in the i^{th} dimension as $DataLength_i$. The width of the halo region is defined as the number of neighborhood elements to gather along the i^{th} dimension of the stencil, and is denoted by $HaloWidth_i$, which is usually the length of the stencil minus one. In the case of code in Listing 1, $HaloWidth$ in both dimensions are set to two. The halo width, together with the number of loops within each stage and the thread block size, determines the trapezoid’s slope, height (h), and the size of the tile that it starts with, respectively. We simplify the model by assuming the common case where the thread block is chosen to be isotropic and its length is constant in all dimensions, denoted as blk_Len . The width of the ghost zone is determined by the trapezoid height as $HaloWidth_i \times h$. We use the average cycles per loop (CPL) as the metric of ISL performance. Since a trapezoid spans across h loops which form a stage, the CPL can be calculated as the cycles per stage (CPS) divided by the trapezoid’s height.

$$CPL = \frac{CPS}{h} \quad (1)$$

CPS is comprised of cycles spent in the computation of all trapezoids in one stage plus the global synchronization overhead ($GlbSync$). Trapezoids are executed in the form of thread blocks whose execution do not interfere with each other except for device memory accesses; when multiple memory requests are issued in bulks by multiple thread

blocks, the requests are queued in the device memory and a thread block may have to wait for requests from other thread blocks to complete before it continues. Therefore, the latency in the device memory accesses ($MemAcc$) needs to consider the joint effect of memory requests from all thread blocks, rather than be regarded as part of the parallel execution. Let CPT (computing cycles per trapezoid) be the number of cycles for an SM to compute a single trapezoid assuming instantaneous device memory accesses, T be the number of trapezoids in each stage, and M be the number of multiprocessors, we have:

$$CPS = GlbSync + MemAcc + CPT \times \frac{T}{M} \quad (2)$$

Where the number of trapezoids can be approximated by dividing the total number of data elements with the size of non-overlapping tiles with which trapezoids end.

$$T = \frac{\prod_{i=0}^{D-1} DataLength_i}{\prod_{i=0}^{D-1} (blk_Len - HaloWidth_i \times h)} \quad (3)$$

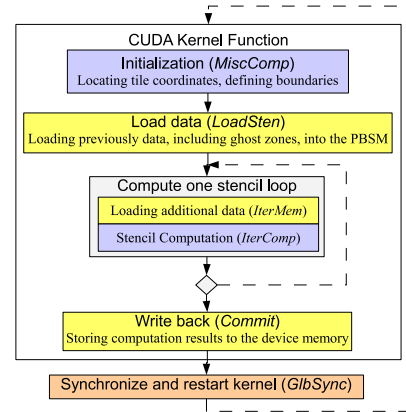


Figure 3: Abstraction of CUDA implementation for ISLs with ghost zones.

Furthermore, ISLs, when implemented in CUDA, usually take several steps described in Figure 3. As it shows, latencies spent in device memory accesses ($MemAcc$) are additively composed of three parts namely:

- *LoadSten*: cycles for loading the ghost zone into the PBSM to compute the first tile in the trapezoid.
- *IterMem*: cycles for other device memory accesses involved in each stencil operation for all the loops.
- *Commit*: cycles for storing data back to the device memory.

CPT , the cycles spent in a trapezoid’s parallel computation, is additively comprised of two parts as well:

- *IterComp*: cycles for computation involved in a trapezoid’s stencil loops.
- *MiscComp*: cycles for other computation that is performed only once in each trapezoid.

We now discuss the modeling of these components individually. The measuring of $GlbSync$ is described as well.

4.2 Memory Transfers

Due to the SIMD nature of the execution, threads from half a warp coalesce their memory accesses into memory requests of 16 words ($CoalesceDegree = 16$). Since concurrent thread blocks execute the same code, they tend to issue their memory requests in rapid succession and the requests are likely to arrive at the device memory in bulks. Therefore, our analytical model assumes that all memory requests from concurrent blocks are queued up. The device memory is optimized for bandwidth: for the GeForce GTX 280 model, it has eight 64-bit channels and can reach a peak bandwidth of 141.7 GBytes/sec [5]. The number of cycles to process one memory request with x bytes is

$$CyclesPerReq(x) = \frac{x}{MemBandwidth \div ClockRate} \quad (4)$$

Because the Tesla architecture allows each SM to have up to eight concurrent thread blocks ($BlksPerSM = 8$), the total number of concurrent thread blocks is

$$ConcurBlks = BlksPerSM \times M \quad (5)$$

With $ConcurBlks$ thread blocks executing in parallel, $\frac{T}{ConcurBlks}$ stages necessary for T thread blocks to complete their memory accesses. To estimate the number of cycles spent to access n data elements of x bytes in the device memory, we have:

$$stages = \frac{T}{ConcurBlks} \quad (6)$$

$$MemCycles(n) = stages \times [UncontendedLat + \alpha \times \frac{n \times CyclesPerReq(x \times CoalesceDegree)}{stages \times CoalesceDegree}] \quad (7)$$

where $UncontendedLat$ is the number of cycles needed for a memory request to travel to and from the device memory. It is assumed to be 300 cycles in this paper, however, later studies show its value does not impact the predicted performance as significantly as the memory bandwidth does, given large data sets. Besides the peak bandwidth, the memory overhead is also affected by pipelining whose depth is not publicly available, and bank conflicts, which depends on the execution of particular applications. Therefore, we introduce an artificial factor, $\alpha = \frac{D}{PipelineDepth}$, that compensates for their effect on the memory access overhead. We assume accessing higher dimensional arrays leads to more bank conflicts and lower memory throughput. Moreover, the memory throughput increases with more pipeline depth, which is assumed to be four according to the depth of the prefetch buffer in DDR2 [22].

For a trapezoid over a D -dimensional thread block with a size of blk_len^D , it typically loads blk_len^D data elements including the ghost zone. Usually, only one array is gathered for stencil operations ($NumStencilArrays = 1$), although in some rare cases multiple arrays are loaded. After loading the array(s), each tile processes

$\prod_{i=1}^D (blk_len - HaloWidth_i)$ elements. Zero or more data elements ($NumElemPerOp \geq 0$) can be loaded from the device memory for each stencil operation, whose overhead is include in the model as $IterMem$. Because our implementation computes a trapezoid by performing the same number of stencil operations in each loop and only committing the valid values at the end (Section 3.3), the number of additional elements to load in each loop remains constant. Finally, the number of elements for each trapezoid to store to the device memory is $\prod_{i=0}^{D-1} (blk_len - HaloWidth_i \times h)$. We summarize these components as:

$$LoadSten = NumStencilArrays \times MemCycles(T \times blk_len^D) \quad (8)$$

$$Commit = MemCycles(T \times \prod_{i=0}^{D-1} (blk_len - HaloWidth_i \times h)) \quad (9)$$

$$IterMem = NumElemPerOp \times h \times MemCycles(T \times \prod_{i=0}^{D-1} (blk_len - HaloWidth_i)) \quad (10)$$

4.3 Computation

We estimate the number of instructions to predict the number of computation cycles that a thread block spends other than accessing the device memory. Because threads are executed in SIMD, instruction counts are based on warp execution (not thread execution!). We set the cycles-per-instruction (CPI) to four because in Tesla, a single instruction is executed by a warp of 32 threads distributed across eight SPs, each takes four pipeline stages to complete the same instruction from four threads. Reads and writes to the PBSM are treated the same as other computation because accessing the PBSM usually takes the same time as accessing registers. After all, the computation cycles for a thread block can be determined as

$$CompCycles = NumWarpInsts \times ActiveWarpsPerBlock \times CPI \quad (11)$$

where $NumWarpInsts$ is the number of instructions executed by each warp and $ActiveWarpsPerBlock$ is the number of warps that has one or more running threads not suspended by branch divergence. While $ActiveWarpsPerBlock$ can be deduced from the tile size, $NumWarpInsts$ has to be synthesized for different trapezoid heights. We categorize $NumWarpInsts$ into two parts: those that are performed once for each trapezoid ($NumWarpInsts_{MC}$), which account for $MiscComp$, and those that are performed iteratively in all stencil loops ($NumWarpInsts_{IC} \times h$), which account for $IterComp$. We use the CUDA Profiler [6] to count $NumWarpInsts_{MC}$ and $NumWarpInsts_{IC}$ separately when processing a small data set sized N with no ghost zones applied ($h = 1$). The recorded numbers, $InstsPerSM_{MC}$ and $InstsPerSM_{IC}$, respectively, are the numbers of instructions executed by all the warps on the same multiprocessor, which relates to $NumWarpInsts$ as:

$$NumWarpInsts = \frac{InstsPerSM}{WarpsPerSM} = InstsPerSM \div \frac{N}{WarpSize \times M} \quad (12)$$

$ActiveWarpsPerBlock$ is an integer value where a warp without all threads active should still be counted as one. We approximate it with floating point numbers so that it can be represented as the number of active threads divided by the the warp size. It is therefore estimated as $\frac{blk_len^D}{WarpSize}$ for $MiscComp$ and $\frac{\prod_{j=0}^{D-1} (blk_len - HaloWidth_j)}{WarpSize}$ for $IterComp$. Substituting these expressions into Equation 11, we have

$$MiscComp(n) = \frac{InstsPerSM_{MC} \times M}{N} \times blk_len^D \times CPI \quad (13)$$

$$\begin{aligned}
IterComp(n) &= h \times \frac{InstsPerSM_{IC} \times M}{N} \\
&\times \prod_{j=0}^{D-1} (blk_Len - HaloWidth_j) \times CPI \quad (14)
\end{aligned}$$

Note that computation involved in *IterComp* is repeated h times and therefore its number of cycles is multiplied by h .

4.4 Global Synchronization

To exchange the halo region, a thread block needs only synchronize with its neighbors. However, in CUDA, the only way to do this is global synchronization that involves terminating a kernel function and restarting a new one. Due to the lack of publicly available technical details, we assume that the exposed overhead of global synchronization is the time to terminate and restart concurrent blocks, whose quantity is calculated in Equation 5. The time spent for other blocks is hidden by the computation of concurrent blocks.

In the case of GTX 280, there are 240 concurrent thread blocks. We therefore measured the time spent in restarting a kernel function with 240 empty thread blocks, which averages at 3350 cycles.

$$GlbSync = 3350 \quad (15)$$

4.5 Extendability

Our performance model focuses on shared memory CMPs with local store based memory system. Therefore, it can be extended to model the Cell Broadband Engine (CBE) [13]. Specifically in CBE, a tile needs not flush its data to the globally shared memory in order to communicate with others. Moreover, the model needs to consider the effect of latency hiding in the light of direct memory access (DMA) operations.

Given an appropriate memory latency model, the performance model can also be generalized to systems with cache hierarchies. Several additional factors need to be modeled including caching efficiency and the effect of prefetching. One candidate cache latency model for ISLs is Kamil et al.'s Stencil Probe [17] which does not take into account the effects of ghost zones yet.

5. EXPERIMENTS

We validate our performance model using four distinct ISL applications that fall in the categories of dynamic programming, ODE and PDE solvers, and cellular automata. We compare their performance predicted by the model with their actual performance on the GeForce GTX 280 graphics card, whose architectural parameters are summarized in Table 1 and are used in our performance model for optimizations. These parameters are retrieved using device query and some are obtained literally through the GTX 280 specifications [5]. We then use the performance model to select the optimal trapezoid height and evaluate its accuracy.

While the major workload is carried out by the GPU, the benchmarks are launched on a host machine with an Intel Core2 Extreme CPU X9770 with a clock rate of 3.2 GHz. The CPU connects to the GPU through NVIDIA's MCP55 PCI bridge. The actual time measurement only includes the computation performed on the GPU.

5.1 Benchmarks

Our benchmark suite contains four distinct ISL applications programmed in CUDA. They have different dimension-

clock rate	1.3 GHz
coalesce width	16
warp size	32
number of SMs	30
concurrent blocks per SM	8
number of SPs per SM	8
SP pipeline depth	4
average CPI	4
memory bandwidth	141.7 GBytes/sec
maximum number of threads per block	512
maximum memory pitch	262144 bytes

Table 1: Architecture parameters used in our performance model.

ality, computation intensities, and memory access intensities.

- *PathFinder* uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, each node picks a neighboring node in the previous row that has the smallest accumulated weight, and adds its own weight to the sum.
- *HotSpot* [15] is a widely used tool to estimate processor temperature. A silicon die is partitioned into functional blocks based on a floorplan, and the simulation solves a ODE iteratively, where new temperature in each block is recalculated based on its neighborhood temperatures in the previous time step.
- *Poisson* numerically solves the poisson equation [9] which is a PDE widely used in electrostatics and fluid dynamics. The solver iterates until convergence, using stencils to calculate the Laplace operator over a grid.
- Cell is a cellular automaton used in Game of Life [2]. In each iteration, each cell, labeled as either live or dead, counts the number of live cells in its neighborhood, and determines whether it will be live or dead in the next time step.

The benchmark-specific parameters used in our performance model are listed in Table 2.

	PathFinder	HotSpot	Poisson	Cell
stencil dimensionality	1	2	2	3
stencil size	3	3 × 3	3 × 3	3 × 3 × 3
halo width	2	2 × 2	2 × 2	2 × 2 × 2
NumStencilArrays	1	2	1	1
NumElemPerOp	1	0	0	0
Profiling Input Size (N)	100,000	500 × 500	500 × 500	60 × 60 × 60
InstsPerSM _{IC} (N)	1998	13488	12825	71603
InstsPerSM _{IC} (N)	1859	16645	12474	220521

Table 2: Benchmark parameters used in our performance modeling.

5.2 Model Validation

We verify our performance model by increasing the input size of each benchmark and comparing the experimental and theoretical results. The performance is measured in CPU cycles and is then normalized to GPU cycles. The measured execution time is then compared with the time predicted by the performance model.

In the experiment shown in Figure 4, we increase the input size of PathFinder, HotSpot, Poisson and Cell from 100,000 to 1,000,000, 500 × 500 to 2000 × 2000, 500 × 500 to 2000 × 2000, and 40 × 40 × 40 to 100 × 100 × 100, respectively. The trapezoid height is set to two uniformly and the thread block size is set to 256 for PathFinder, 20 × 20 for HotSpot, 16 × 16 for Poisson, and 8 × 8 × 8 for Cell. The prediction error is

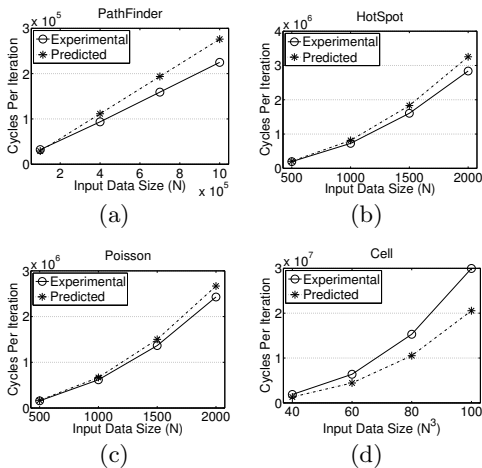


Figure 4: Model verification by scaling the input size of (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. Although the prediction error ranges from 2% to 30%, the performance model captures the overall scaling trend for all benchmarks.

9-22% for PathFinder, 2-12% for HotSpot, 2-8% for Poisson, and 28-30% for Cell. Nevertheless, the performance model captures the overall scaling trend for all benchmarks.

5.3 Sources of Inaccuracy

Our performance modeling may be subject to three sources of inaccuracy:

- *Unpredictable dynamic events.* This includes control-flow divergence, bank-conflicts in both the PSM and the device memory, and the degree of device memory contention. Although the error introduced by control flow divergence is minimized by run-time profiling the number of dynamic instructions, the other two types of errors are intrinsic in shared memory systems and they are difficult to reduce. The degree of bank-conflict in the device memory is estimated using arrays’ dimensionality because higher dimensionality leads to strided accesses that are more likely to incur bank conflicts. Moreover, due to the homogeneity of thread blocks, we assume thread blocks are likely to issue device memory requests close in time and therefore requests are queued and processed in bulks, maximizing the memory bandwidth. Note that the effect of latency hiding using multi-threading is considered in our model — SMs can switch to another concurrent thread block upon device memory accesses and therefore memory requests from all concurrent thread blocks can be issued and queued together.
- *Insufficient technical details.* Due to the lack of information regarding the launching of kernel functions and the depth of memory pipelining, we are not able to accurately estimate the latency for global synchronization and the overhead of queuing due to memory contention. Instead, we measure the latency of global synchronization based on a simplified model, and introduce an artificial factor α to compensate the effect of pipelining.
- *Approximation Error.* To maintain a continuous function for gradient based optimization, we have to calculate *ActiveWarpsPerBlock* as floating point values rather than integers. This error is more significant

for thread blocks narrow in length which lead to more branch divergence on the boundary, such is the case with Cell. In this scenario, a warp with most threads suspended may be counted as a small fraction, while in reality it should be counted as one.

Despite these sources of inaccuracy, we show that the performance modeling reflects the experimental performance scaling and it is sufficient to guide the selection of the optimal ghost zones or trapezoid configuration for ISL applications in CUDA.

5.4 Performance Optimization

5.4.1 Choosing Tile Size

We formulate the *cost-effectiveness* of a trapezoid as the ratio between the size of the tile that it produces at the end and the size of the tile that it starts with. It is represented as

$$CostEffect = \frac{\prod_{i=0}^D (blk_Len - HaloWidth_i \times h)}{\prod_{i=0}^D (blk_Len - HaloWidth_i)} \quad (16)$$

For a given trapezoid height, a larger tile size always increases the cost-effectiveness of trapezoid, reducing the percentage of stencil operations to replicate and achieving better performance, as we will demonstrate in Section 6.3. In our CUDA implementation, the tile size is determined by the thread block size which is set as large as possible within the architectural limit of 512.

5.4.2 Selecting Ghost Zone Size

The optimal ghost zone size is determined by the optimal trapezoid height, which in turn depends on the dynamics among computation and memory access. Our technique estimates the optimal trapezoid height in two steps. First, we comment out part of the application code in order to obtain *InstsPerSM_{MC}* and *InstsPerSM_{IC}* separately using the CUDA Profiler [6]. The run-time profiling is performed once for each ISL algorithm, and it only requires the computation of one stencil loop with a minimum input size big enough to occupy all the SMs. The resulting instruction counts are then used to compute the optimal trapezoid height in Equation 1. While Equation 1 is not a posynomial function, it is convex and we can obtain a unique solution using gradient-based constrained nonlinear optimization. We apply a constraint that sets the upper bound of the trapezoid height so that a tile produced at the end of a thread block has a positive area:

$$\forall i \in [0, D), blk_Len - HaloWidth_i \times h > 0 \quad (17)$$

The run-time profiling and ghost zone optimization needs to be performed only once for each ISL. However, they need to be recalculated if the same application is ported to systems with different settings (e.g. memory bandwidth, number of SMs, etc).

The performance model estimates that the optimal trapezoid height is 14 for PathFinder with a thread block size of 256, 2 for HotSpot and Poisson with a thread block size of 20×20 , and 1 for Cell with a thread block size of 8. The input size involved in the calculation is 1,000,000 for PathFinder, 2000×2000 for HotSpot, 2000×2000 for Poisson, and $100 \times 100 \times 100$ for Cell.

The predictions are compared to experimental results shown in Figure 5 and they turn out to match the results of PathFinder and Cell. Even though experiments show that the optimal performance is achieved at a trapezoid height of three for both HotSpot and Poisson, the performance at the predicted trapezoid height is no worse than 98% of the optimal performance. The resulting speedups compared to performance

without ghost zones are 2.29X, 1.45X, 1.51X, and 1.00X for PathFinder, HotSpot, Poisson and Cell, respectively.

Our estimation of the optimal trapezoid height can also serve as the initial guess for an adaptive selection method. Suppose run-time profiling calculates the average execution time for every two loops and decides whether to increment or decrement the trapezoid height, and there is a total of 30 iterations. With a more accurate initial guess, our adaptive technique achieves speedups of 2.29X, 1.44X, 1.50X, and 0.90X for PathFinder, HotSpot, Poisson and Cell, while an initial guess of one, as proposed in [1], results in speedups of 1.93X, 1.39X, 1.45X and 0.90X, respectively. The slow-down in Cell is because the performance does not benefit from ghost zones, however, the adaptive method still probes a trapezoid height of two.

6. SENSITIVITY STUDY

Using the validated modeling, we are able to further study the performance scaling brought by ghost zones on shared memory architectures. We show how it affects the execution time spent in computation, memory access, and global synchronization. We also investigate what applications and system platforms may benefit more from ghost zones.

6.1 Component Analysis

We transform Equation 2 to the accumulation of six components, each term represents average cycles spent in one component within a stencil loop:

$$CPL = GlbSync' + LoadSten' + Commit' + MiscComp' + IterComp' + IterMem' \quad (18)$$

$$GlbSync' = \frac{GlbSync}{h} \quad (19)$$

$$LoadSten' = \frac{LoadSten}{h} \quad (20)$$

$$Commit' = \frac{Commit}{h} \quad (21)$$

$$MiscComp' = \frac{MiscComp \times T}{h \times M} \quad (22)$$

$$IterComp' = \frac{IterComp \times T}{h \times M} \quad (23)$$

$$IterMem' = \frac{IterMem}{h} \quad (24)$$

Our performance model shows that the execution time is dominated by $IterComp'$, and followed by $LoadSten'$, $MiscComp'$, and $IterMem'$ when applicable. Figure 6 illustrates how each component reacts to the increased trapezoid height or ghost zone size. Each component's execution time is normalized to its time spent with a trapezoid height of one. The figure is drawn using a synthetic benchmark similar to HotSpot but with $NumElemPerOp$ set to one instead of zero to illustrate the scaling curve of $IterMem'$.

Time for $IterComp'$ and $IterMem'$ increases monotonically with taller trapezoids due to increased computation replication. However, time spent in $LoadSten'$, $Commit'$, and $MiscComp'$ decreases first with taller trapezoids due to reduced inter-loop communication and the number of stages. Nevertheless, they eventually increase dramatically due to the exploding number of thread blocks resulted from taller trapezoids which end with tiny non-overlapping tiles.

Moreover, Equations 19 to 24 can be regarded as functions of h , and their derivatives are all monotonically increasing, as can be seen from Figure 6. As a result, the overall objective function is convex and it has a unique minimum.

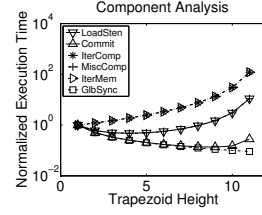


Figure 6: The effect of the trapezoid's height on each component with the execution time for each component normalized to the case where a trapezoid's height is one.

6.2 Software Sensitivity

According to the performance model, we summarize several characteristics that enable an application to benefit from ghost zones.

Stencils operating on lower-dimensional neighborhood. In fact, with the same trapezoid height and the same halo width in each dimension, the ratio of replicated operations grows exponentially with more dimensionality, as can be seen from Equation 8, 14, and 10. This phenomenon is observed in Section 5 where the 1-D PathFinder benefits the most, followed by 2-D HotSpot and Poisson, and the 3-D Cell does not benefit at all.

Narrower halo widths. A wider halo region increases the amount of operations to replicate, therefore it increases $IterComp'$ which usually dominates the overall performance. The peak speedup becomes smaller and it tends to be reached with a shorter trapezoid (Figure 7).

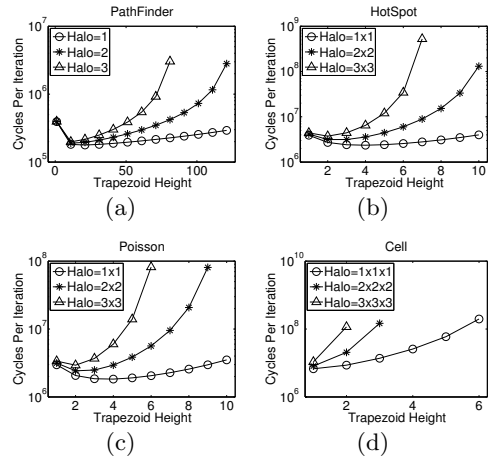


Figure 7: Comparing the effect of various halo widths for (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. Smaller stencils with smaller halo width can benefit more from the trapezoid technique, as demonstrated by our performance modeling by synthesizing four benchmarks with various halo width.

Smaller computation/communication ratio. The penalty for replicating computation-intensive operations may be large enough to obscure ghost zone's savings in communication and synchronization. In this case, peak speedup is likely to be achieved with shorter trapezoids.

In addition, different input sizes hardly affect the relative performance scaling over different trapezoid configurations. This is because it is only reflected in the number

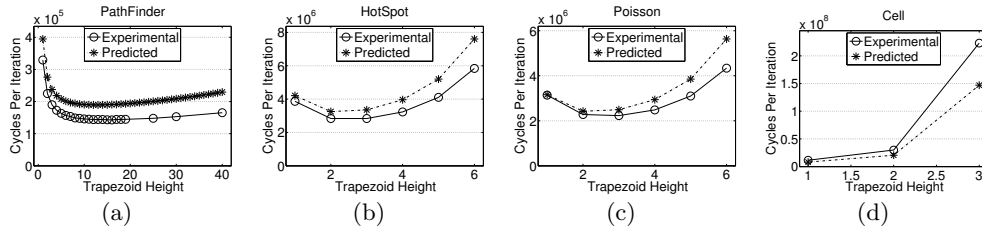


Figure 5: Evaluating the performance optimization by scaling the trapezoid height of (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. The predicted optimal trapezoid height is 14 for PathFinder, 2 for HotSpot and Poisson, and 1 for Cell. While experiments show that the optimal performance is achieved at a trapezoid height of 3 for both HotSpot and Poisson, the performance at the predicted trapezoid height is no worse than 98% of the optimal performance.

of thread blocks and is a linear factor to all components except *GlbSync'*, which contributes little to the overall performance.

6.3 System Sensitivity

Although our performance model is based on the Tesla architecture, it can be easily extended to model other shared memory parallel systems. We investigate how ghost zone's benefits vary across different system platforms.

Larger tile size. As we discussed in Section 5.4.1, by using larger tiles, less computation needs to be replicated and performance can be improved. As Figure 8 shows, a larger thread block size increases the peak speedup and shifts the best configuration towards taller trapezoids. The benefit of ghost zones may be more significant for architectures that easily allow for larger tile sizes, such as CBE.

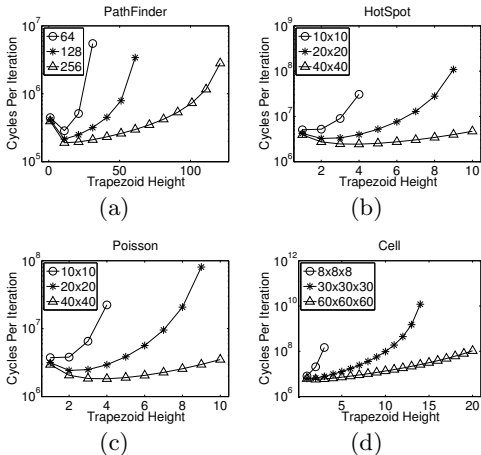


Figure 8: Compare the effect of various block sizes for (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. Code programmed with larger thread blocks can benefit more from ghost zones, as demonstrated using our performance modeling to scale the thread block size beyond the architectural limit.

Longer synchronization latency. One benefit of ghost zones is the elimination of inter-loop synchronization. As Figure 6 shows, *GlbSync'* always decreases with taller trapezoids. Nevertheless, latency in synchronization is not a major contributor to the overall performance on GTX 280.

Longer memory access latency. The savings in inter-loop communication is more evident with longer memory access latency. This can be caused by higher bandwidth demand, higher contention, or higher transferring overhead. For CBE

Listing 2: Code annotation for automatic trapezoid optimization

```
float **A, **B;
for k = 0 : num_iterations with trapezoid.height=[H]
  for_all i = 0 : ROWS-1 and j = 0 : COLS-1
    /* define the array(s) to be loaded from */
    apply trapezoid.obj=[B]
    /* define the dimensionality of the array */
    apply trapezoid.dimension=2
    /* define the halo width in all dimensions
    apply trapezoid.gather[-1,+1][-1,+1]
    top = max(i-1, 0);
    bottom = min(i+1, ROWS-1);
    left = max(j-1, 0);
    right = min(j+1, COLS-1);
    A[i][j] = B[i][j] + B[top][j] \
              + B[bottom][j] + B[i][left] \
              + B[i][right];
  swap(A, B);
```

whose memory bandwidth is not as optimized as Tesla, it is likely that ISL applications benefit more from ghost zones.

Smaller CPI. Technology is driving towards a smaller CPI — either by improving the instruction-level parallelism (ILP) or increasing the computation bandwidth (e.g. SIMDization). A smaller CPI puts less weight on computation and more on memory accesses, therefore programs can benefit from the ghost zones further.

7. AN AUTOMATED FRAMEWORK TEMPLATE FOR TRAPEZOID OPTIMIZATION

Since the benefit of ghost zones depends on various factors related to both the application and the system platform, it is hard for programmers to find out the best configuration. Moreover, implementing ghost zones for ISL applications involves nontrivial programming efforts and it is often error-prone. We therefore propose a framework template that automatically transform ISL programs in CUDA to that equipped with the optimal trapezoid configuration. The framework is comprised of three parts: code annotation and transformation, one-time dynamic profiling and off-line optimization.

Listing 2 illustrate the proposed code annotation for the pseudocode in Listing 1. The programmer specifies the array to be loaded from and its dimensionality, as well as the halo width of the stencil operations. The framework is then able to transform the code to that equipped with ghost zones. The code transformation also implicitly distinguishes computation involved in stencil loops from the rest for the purpose of profiling.

With the transformed code, the framework then counts

the instructions and estimates the computation intensity using the CUDA Profiler, as described in Section 4.3. Profiling is performed once implicitly and the results are used for calculation in the performance model. Finally, the performance model estimates the optimal trapezoid configuration based on the current system platform — as described in Section 5.4 — and generates appropriate code.

8. CONCLUSIONS AND FUTURE WORK

We establish a performance modeling of ISL applications programmed in CUDA and study the benefits and limitations of ghost zones. The performance modeling based on the Tesla architecture is validated using four distinct ISL applications and it is able to estimate the optimal trapezoid configuration. The trapezoid height selected by our performance model is able to achieve a speedup no less than 98% of the optimal speedup for our benchmarks. Our performance model can be extended and generalized to other shared memory systems. Several application- and architecture-characteristics that can leverage the usage of ghost zones are identified. Finally, we propose a framework template that can automatically incorporate ghost zones to ISL applications in normal CUDA code and optimize it with the selection of trapezoid configurations. An immediate step in our future work will be to port our infrastructure to the OpenCL standard once suitable tools are available. Extensions can be implemented for CMPs with cache-based memory systems. It will also be interesting to study how the benefit from ghost zones is effected by cache prefetching and cache blocking optimizations.

9. ACKNOWLEDGEMENTS

This work was supported in part by SRC grant No. 1607, NSF grant nos. IIS-0612049 and CNS-0615277, a grant from Intel Research, and a professor partnership award from NVIDIA Research.

10. REFERENCES

- [1] G. Allen, T. Damlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In *SC'01*, pages 52–52, New York, NY, USA, 2001.
- [2] M. Alpert. Not just fun and games. April 1999.
- [3] S. Chatterjee, J.R. Gilbert, and R. Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. *SC'93*, pages 420–429, Nov. 1993.
- [4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA, June 2008.
- [5] NVIDIA Corporation. Geforce gtx 280 specifications. 2008.
- [6] NVIDIA Corporation. NVIDIA CUDA visual profiler. June 2008.
- [7] L. Dagum. OpenMP: A proposed industry standard API for shared memory programming, October 1997.
- [8] 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 *SC '08*, pages 1–12, Piscataway, NJ, USA, 2008.
- [9] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, 1998.
- [10] L. Chen Z.-Q. Zhang X.-B. Feng. Redundant computation partition on distributed-memory systems. In *ICA3PP '02: Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, page 252, Washington, DC, USA, 2002.
- [11] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS'05*, pages 361–366, New York, NY, USA, 2005.
- [12] N. Goodnight. CUDA/OpenGL fluid simulation, April 2007.
- [13] M. Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF'06*, New York, NY, USA, 2006.
- [14] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *J. Parallel Distrib. Comput.*, 19(2):90–102, 1993.
- [15] W. Huang, M. R. Stan, K. Skadron, S. Ghosh, K. Sankaranarayanan, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *DAC'04*, 2004.
- [16] W. Jalby and U. Meier. Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system. pages 429–432, 1986.
- [17] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP'05*, pages 36–43, New York, NY, USA, 2005.
- [18] M. Kowarschik, C. Weiß, W. Karl, and U. Rüdte. Cache-aware multigrid methods for solving poisson's equation in two dimensions. *Computing*, 64(4):381–399, 2000.
- [19] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *PLDI '07*, 42(6):235–244, 2007.
- [20] P. Lee. Techniques for compiling programs on distributed memory multicomputers. *Parallel Comput.*, 21:1895–1923, 1995.
- [21] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, 2004.
- [22] J. Lin, H. Zheng, Z. Zhu, Z. Zhang, and H. David. Dram-level prefetching for fully-buffered dimm: Design, performance and power saving. *ISPASS'07*, 2007.
- [23] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *Parallel Distrib. Syst.*, 8:19–28, 1997.
- [24] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.
- [25] K. N. Premnath and J. Abraham. Three-dimensional multi-relaxation time (mrt) lattice-Boltzmann models for multiphase flow. *J. Comput. Phys.*, 224(2):539–559, 2007.
- [26] J. Ramanujam. Tiling of iteration spaces for multicomputers. In *Proc. Int. Conf. Parallel Processing*, pages 179–186, 1990.
- [27] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. Towards optimal multi-level tiling for stencil computations. *IPDPS'07*, pages 1–10, March 2007.
- [28] L. Renganarayana and S. Rajopadhye. Positivity, posynomials and tile size selection. In *SC '08*, pages 1–12, Piscataway, NJ, USA, 2008.
- [29] M. Ripeanu, A. Iamnitchi, and I. Foster. Cactus application: Performance predictions in a grid environment. In *EuroPar'01*, 2001.
- [30] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *SC '00*, page 32, Washington, DC, USA, 2000.
- [31] S.-Z. Ueng, S. Baghsorkhi, M. Lathara, and W. m. Hwu. CUDA-lite: Reducing GPU programming complexity. In *LCPC'08*, 2008.
- [32] D. Wonnacott. Time skewing for parallel computers. In *WLCPC'99*.
- [33] D. Wonnacott. Achieving scalable locality with time skewing. *Int. J. Parallel Program.*, 30(3):181–221, 2002.
- [34] Z. Yang, Y. Zhu, and Y. Pu. Parallel image processing based on CUDA. *Int. Conf. Comput. Sci. and Software Eng.*, 3:198–201, Dec. 2008.