Int J Parallel Prog (2011) 39:115–142
DOI 10.1007/s10766-010-0142-5

A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations

Jiayuan Meng · Kevin Skadron

Received: 30 January 2010 / Accepted: 26 May 2010 / Published online: 30 June 2010

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 distributed environments. 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 95% of the optimal speedup.

Keywords Ghost zone · Halo · Performance model · Iterative stencil loops · GPU · Tiling

J. Meng (✉) · K. Skadron
Department of Computer Science, University of Virginia, Charlottesville, VA, USA
e-mail: jm6dg@virginia.edu

K. Skadron
e-mail: skadron@cs.virginia.edu

1 Introduction

Iterative stencil loops (ISL) [24] 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 [19,29] 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 [13].

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 Fig. 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 Fig. 1(b). The overlap allows each PE to generate its halo regions locally [32] for a number of iterations proportional to the size of the ghost zone. As Fig. 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 distributed environments [32]. 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 distributed environments [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

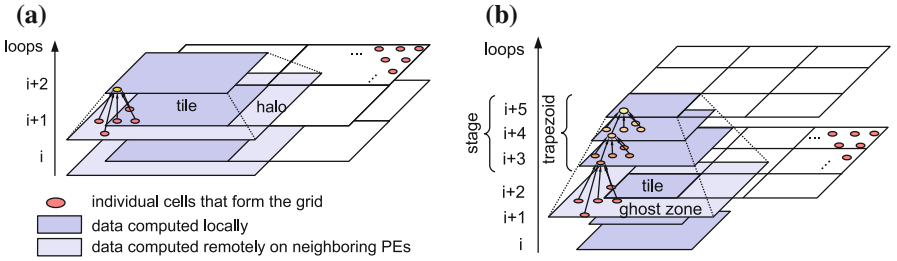

Fig. 1 **a** Iterative stencil loops and halo regions. **b** Ghost zones help reduce inter-loop communication

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 multiprocessor (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 [27]. We choose GPUs because they contain so many cores and some extra complexity regarding the L1 storage and global synchronization. In particular, the use of larger ghost zones is especially valuable in the Tesla architecture because global synchronization is conventionally performed by restarting threads; unlike in the distributed 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 synchronization. Even though the memory fence is later introduced into GPU programming that allows global synchronization without restarting threads, its overhead increases along with the input size, as we will show in Sect. 4.5.

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 95% 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. [20].

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 an analytical 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 [19,21,28,29,33]. In some circumstances, compilers are able to tile stencil iterations to localize computation

or/and exploit parallelism [3,13,35]. Some attempt to reduce redundant computation for specific stencil operations [10]. On the other hand, APIs such as OpenMP are able to tile stencil loops at run-time and execute the tiles in parallel [8]. Renganarayana et al. explored the best combination of tiling strategies that optimizes both cache locality and parallelism [30]. Researchers have also investigated automatic tuning for tiling stencil computations [9,24]. Specifically, posynomials have been widely used in tile size selection [31]. However, these techniques do not consider the ghost zone technique that reduces the inter-tile communication. Although loop fusion [25] and time skewing [36] 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 synchronizes 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 [12]. Krishnamoorthy et al. proposed overlapped tiling that employs ghost zones with time skewing and they studied its effect in reducing the communication volume [22]. 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 [32], and they conclude the optimal ghost zone size is usually one in distributed environments. 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 distributed environments 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 Sect. 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 [4,23], which are used to wisely distribute *pre-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 [16]. To

study the performance of 3-D stencil computations on modern cache-based memory systems, another performance model is proposed by Kamil et al. [20] 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. [5]. The same technique can be used in other existing CUDA applications ranging from fluid dynamics [14] to image processing [37].

Another automatic tuning framework for CUDA programs is CUDA-lite [34]. 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.

We extend our prior work [26] from several perspectives. First, we introduce artificial factors to compensate for the effects in latency hiding and bank conflicts. Second, we study the use of memory fences in global synchronization and extend the analytical model to predict its performance. Third, more sensitivity studies show that varying the applications' input size rarely affects its optimal ghost zone size. Fourth, we simplify the model and show it is insensitive to memory latency. Finally, we measured the power consumption resulted from the ghost zone technique.

3 Ghost Zones on GPUs

We show an example of ISLs and illustrate how ghost zones are optimized in distributed environments. 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 Distributed Environments

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 distributed 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.

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 overflown 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);
```

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 [27]. As Fig. 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

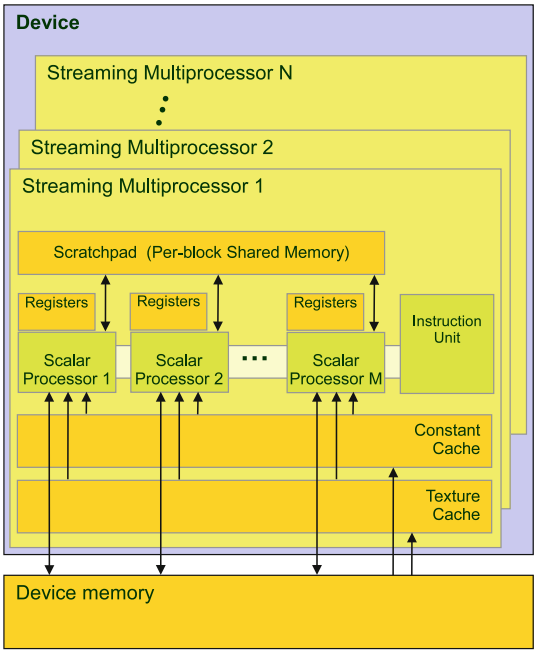

Fig. 2 CUDA's shared memory architecture. Courtesy of NVIDIA

number of SMs, thread blocks are mapped onto SMs as preceding thread blocks finish. This allows the same program and grid to run on GPUs 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.

3.3 Implementing Ghost Zones in CUDA

Without ghost zones nor memory fences, 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 distributed environments where each tile only has to fetch halo regions, all data in PBSM is flushed and all has to be reloaded again. Even after CUDA 2.2 introduced the memory fence that allows global synchronization without restarting kernels, the overhead in such synchronization increases along with the input size, as we will show in Sect. 4.5. 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 boundary of the valid tile has to be calculated for each iteration, which increases the amount of computation and leads to more control-flow divergence within warps. It eventually 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.

$$CPL = \frac{CPS}{h} \tag{1}$$

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 the number of trapezoids in each stage, and M be the number of multiprocessors, we have:

$$CPS = GlbSync + MemAcc + CPT \times \frac{T}{M} \tag{2}$$

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.

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

Furthermore, ISLs, when implemented in CUDA, usually take several steps described in Fig. 3. By default, the global synchronization between stages are implemented by terminating and restarting the kernel. 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 thread. This mainly involves the computation for setting up the thread coordination and handling boundary conditions.

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

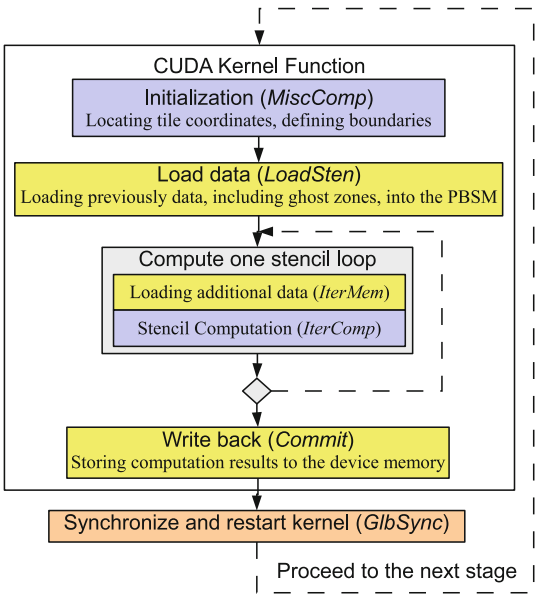

Fig. 3 Abstraction of CUDA implementation for ISLs with ghost zones

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 [6]. The number of cycles to process one memory request with p bytes is

$$CyclesPerReq(p) = \frac{p}{MemBandwidth \div ClockRate} \tag{4}$$

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

$$ConcurBlks = BlksPerSM \times M \tag{5}$$

With $ConcurBlks$ thread blocks executing in parallel, $\frac{T}{ConcurBlks}$ passes 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:

$$passes = \frac{T}{ConcurBlks} \tag{6}$$

$$MemCycles(n) = passes \times \left[UncontendedLat + \alpha \times \frac{n \times CyclesPerReq(x \times CoalesceDegree)}{passes \times CoalesceDegree} \right] = passes \times UncontendedLat + \alpha \times \frac{n \times x \times ClockRate}{MemBandwidth} \quad (7)$$

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.

[IJPP] Besides the peak bandwidth, the memory overhead is also affected by bank conflicts, which depend on the execution of particular applications. Therefore, to compensate for the effect on the memory access overhead, we introduce an artificial factor, $\alpha = \sigma^{D-1}$, where σ is empirically set to 5.0 to best approximate the experimental results. We assume accessing higher dimensional arrays leads to more bank conflicts and lower memory throughput. Further experiments show that the overall performance trend predicted by our analytical performance model is not sensitive to the choice of σ.

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 (Sect. 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:

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

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

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

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.

[IJPP] In addition, the performance model also has to take into account the effect of latency hiding and bank conflicts. Latency hiding overlaps computation with memory accesses; therefore the overall execution time is not the direct sum of cycles spent in memory accesses and computation. We define the effectiveness of latency hiding, β, as the number of instructions that overlap with memory accesses divided by the total number of instructions. Due to the lack of measurement tools, we approximate the effect of latency hiding by assuming half of the instructions overlaps with memory accesses; therefore β equals 0.5. With regard to the overall execution time, the effect of latency hiding is the same as that of reducing the number of warp instructions by a factor of β.

On the other hand, a bank conflict serializes SIMD memory instructions and it has the same latency effect as increasing the number of instructions by a factor of $BankConflictRate$, which equals the number of bank conflicts divided by the number of warp instructions, both can be obtained from the CUDA Profiler [7].

The overall effect of latency hiding and bank conflicts on computation cycles is as if the number of instructions is scaled with a factor of τ, which equals $(1 - \beta) \times (1 + BankConflictRate)$. After all, the computation cycles for a thread block can be determined as

$$CompCycles = StageInstsPerWarp \times \tau \times ActiveWarpsPerBlock \times CPI \tag{11}$$

where $StageInstsPerWarp$ is the number of instructions executed by an individual warp during a trapezoid stage, and $ActiveWarpsPerBlock$ is the number of warps that have one or more running threads not suspended by branch divergence. While $ActiveWarpsPerBlock$ can be deduced from the tile size, $StageInstsPerWarp$ has to be synthesized for different trapezoid heights. We breakdown $StageInstsPerWarp$ into two additive parts: those that are performed once for each trapezoid ($StageInstsPerWarp_{MC}$), which correspond to $MiscComp$, and those that are performed iteratively in all stencil loops ($StageInstsPerWarp_{IC}$), which correspond to $IterComp$. $StageInstsPerWarp_{MC}$ and $StageInstsPerWarp_{IC}$ are estimated separately as:

$$StageInstsPerWarp_{MC} = IterInstsPerWarp_{MC} \tag{12}$$

$$StageInstsPerWarp_{IC} = h \times IterInstsPerWarp_{IC} \tag{13}$$

where $IterInstsPerWarp$ stands for the number of instructions executed by an individual warp during a conventional iteration. Its two components are $IterInstsPerWarp_{MC}$, which corresponds to $MiscComp$, and $IterInstsPerWarp_{IC}$, which corresponds to $IterComp$. With a trapezoid of height h, computation involved in

$IterComp$ is repeated for h iterations, therefore the number of instructions per iteration is multiplied by h to estimate the number of instructions per stage in Eq. 13. We then use the CUDA Profiler [7] to estimate $IterInstsPerWarp_{MC}$ and $IterInstsPerWarp_{IC}$ for each application using 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 per iteration executed by all the warps on the same multiprocessor. Each type of $InstsPerSM$ relates to their corresponding $IterInstsPerWarp$ as:

$$IterInstsPerWarp = \frac{InstsPerSM}{WarpsPerSM} = InstsPerSM \div \frac{N}{WarpSize \times M} \tag{14}$$

$InstsPerSM_{MC}$ and $InstsPerSM_{IC}$ are measured separately by commenting out different sections of code and rerun the CUDA Profiler. Note that the resulting values of $IterInstsPerWarp_{MC}$ and $IterInstsPerWarp_{IC}$ are independent of the input size and the block size; once obtained, they can be used for arbitrary inputs and block sizes as long as the underlying architecture stays the same.

In Eq. 11, $ActiveWarpsPerBlock$ 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 Eq. 11, we obtain the computation cycles for a thread block as the sum of two terms:

$$MiscComp = \frac{InstsPerSM_{MC} \times M}{N} \times blk\_len^D \times CPI \tag{15}$$

$$IterComp = h \times \frac{InstsPerSM_{IC} \times M}{N} \times \prod_{j=0}^{D-1}(blk\_len - HaloWidth_j) \times CPI \tag{16}$$

4.4 Global Synchronization

To exchange the halo region, a thread block needs only synchronize with its neighbors. In CUDA, the conventional way to perform global synchronization is to terminate and restart the kernel function. 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 Eq. 5. The time spent for restarting other thread 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.

$$GlbSync_{KnlRestart} = 3350 \tag{17}$$

4.5 Extension: Modeling Memory Fence

[IJPP] While the global synchronization is traditionally implemented by restarting a CUDA kernel, the introduction of memory fences since CUDA 2.2 brings an alternative. With memory fences, a single CUDA kernel can persist across stages until all iterations are finished, as described in Fig. 4. This leads to two changes in our performance model. First, $MiscComp$ — the time spent in setting up coordinates and handling boundary conditions — only needs to be counted once for all iterations. Given a large number of iterations, the cycles spent in $MiscComp$ can be negligible and it is no longer a component of CPT. Secondly, the modeling of the overheads in global synchronization has to be adjusted.

We apply the following method to implement global synchronization using memory fences. After synchronizing threads in the same thread block at the end of each stage and using memory fences to make sure their device memory accesses have committed, the first thread in each thread block atomically increments a counter in the device memory. The thread then uses a spinlock which repeatedly reads the counter's value until all thread blocks complete the memory fence. Eventually, it allows the belonging thread block to proceed to the next stage after the counter tells that all thread blocks have committed their partial results into the device memory. We use a different counter for *each* stage. Otherwise, if the same counter is reused across all stages, it has to be reset at the end of each stage after all thread blocks release their spinlock, which requires another round of atomic operations and memory fences.

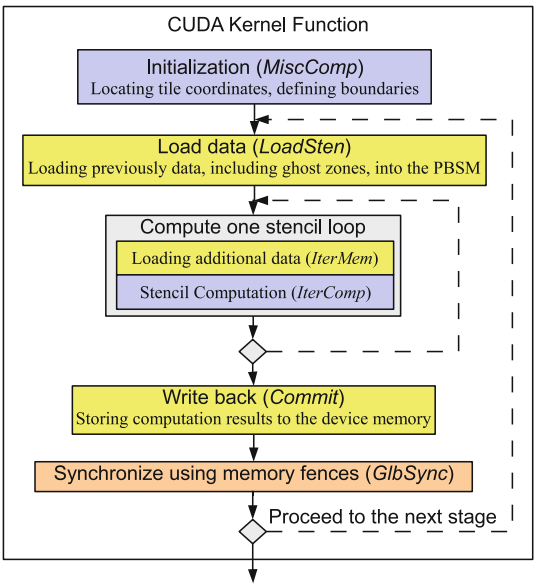

Fig. 4 Abstraction of CUDA implementation for ISLs with ghost zones

Experiments show that the overhead in such synchronization correlates to the number of thread blocks. By measuring the execution time of a kernel function that performs nothing but global synchronization with memory fences, and by varying the number of thread blocks, we obtain the empirical function using linear curve fitting with 95% confidence: $GlbSync_{MemFence} = 210.3 \times \mathrm{T} + \delta$, where δ is negligible. In reality, the latency in global synchronization can overlap with computation from other thread blocks. To compensate for such effect, we introduce an artificial factor μ and assume about half of the time spent in global synchronization overlaps with computation ($\mu = 0.5$). Therefore, we have

$$GlbSync_{MemFence} = \mu \times 210.3 \times T \tag{18}$$

The number of kernel instructions used for implementing the memory fence is not included in the profiled instruction count of $InstsPerSM$. In fact, both $InstsPerSM_{IC}$ and $InstsPerSM_{MC}$ stay the same as that in the implementation without memory fences. Nevertheless, $InstsPerSM_{MC}$ is no longer needed because the overheads in $MiscComp$ is negligible in the implementation with memory fences.

4.6 Extensibility to Other Platforms

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) [15]. 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 [20] 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 [6]. We then use the performance model to select the optimal trapezoid height and evaluate its accuracy. [IJPP]

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.

Table 1 Architectural parameters for GeForce GTX 280

Parameter	Symbol	Value
GPU clock rate	$ClockRate$	1.3 GHz
(Assumed) device memory access latency	$UncontendedLat$	300 cycles
Coalesce width	–	16
Concurrent blocks per SM	$BlksPerSM$	8
Warp size	–	32
Number of SPs per SM	–	8
Number of SMs	M	30
Average CPI	CPI	4
Memory bandwidth	$MemBandwidth$	141.7 GBytes/sec
Maximum number of threads per block	–	512
Maximum memory pitch	–	262144 bytes

Parameters not directly used in our performance model is marked by the symbol of "–". Although the warp size is not directly used in the equations, it is accounted for in the measurement of $InstPerSM$, which counts instructions based on warp execution; with constant workload, varying the warp size will also vary the profiled value of $InstPerSM$. The coalesce width is implicitly reflected as well in the memory bandwidth; a smaller coalesce width would undermine the effective memory bandwidth. Note that in GTX 280, memory requests from a half-warp are *always* coalesced

5.1 Benchmarks

Our benchmark suite contains four distinct ISL applications programmed in CUDA. They have different dimensionality, 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* [17] 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 [11] 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.

Table 2 Benchmark parameters used in our performance modeling

	PathFinder	HotSpot	Poisson	Cell
Stencil dimensionality	1	2	2	3
Stencil size	3	3×3	3×3	$3 \times 3 \times 3$
Halo width	2	2×2	2×2	$2 \times 2 \times 2$
$NumStencilArrays$	1	2	1	1
$NumElemPerOp$	1	0	0	0
Profiling input size (N)	100,000	500×500	500×500	$60 \times 60 \times 60$
$InstsPerSM_{MC}$	1,998	13,488	12,825	71,603
$InstsPerSM_{IC}$	1,859	16,645	12,474	2,20,521
$BankConflictRate0$	0	0	1.1	

5.2 Model Validation

We verify our performance model by varying the configurations and input sizes for 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.

Figure 5 compares the performance generated from actual experiments and that predicted from our analytical model. In this illustration, global synchronization is carried out by restarting kernel functions. We increase the size of the thread block size of PathFinder, HotSpot, Poisson and Cell from 64 to 512, 10×10 to 22×22, 10×10 to 22×22, and $6 \times 6 \times 6$ to $8 \times 8 \times 8$, respectively. The performance model captures the overall scaling trend for all benchmarks except for Cell, where performance is predicted to improve with a larger thread block size but it actually degrades with a block size of $8 \times 8 \times 8$. Further characterizations using the CUDA Profiler show this unexpected trend is due to a large number of bank conflicts with this particular thread block size. However, run-time dynamics of bank conflicts are not modeled faithfully in our analytical model.

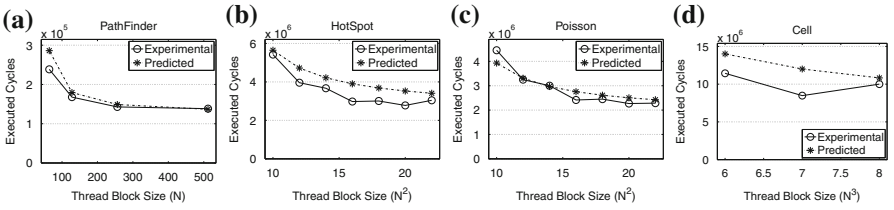

Fig. 5 Model verification by scaling the thread block size of **a** PathFinder, **b** HotSpot, **c** Poisson, and **d** Cell. The trapezoid height is set to 16 for PathFinder, two for HotSpot and Poisson, and one for Cell. The input size is set to 1,000,000 for PathFinder, $2,000 \times 2,000$ for HotSpot and Poisson, and $100 \times 100 \times 100$ for Cell. The examples shown here perform global synchronization by restarting kernels; trends are similar with memory fences. The performance degradation in Cell with a block size of $8 \times 8 \times 8$ results from bank conflicts

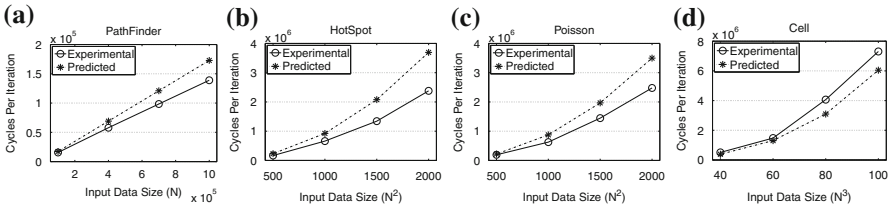

Fig. 6 Model verification by scaling the input size of **a** PathFinder, **b** HotSpot, **c** Poisson, and **d** Cell. The trapezoid height is set to 16 for PathFinder, two for HotSpot and Poisson, and one for Cell. The thread block size is set to 256 for PathFinder, 20×20 for HotSpot, 16×16 for Poisson, and $8 \times 8 \times 8$ for Cell. The performance model captures the overall scaling trend for all benchmarks. The examples shown here perform global synchronization by memory fences; trends are similar with restarting kernels

In the experiment shown in Fig. 6, we increase the input size of PathFinder, HotSpot, Poisson and Cell from $100,000$ to $1,000,000$, 500×500 to $2,000 \times 2,000$, 500×500 to $2,000 \times 2,000$, and $40 \times 40 \times 40$ to $100 \times 100 \times 100$, respectively. In this illustration, global synchronization is performed using memory fences. 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 divergences, bank conflicts in both the PBSM and the device memory, and the degree of device memory contention. Although the error introduced by control flow divergence is reduced by profiling the number of dynamic warp instructions which account for instructions in different branches, the other two types of errors are intrinsic in shared memory systems and they are difficult to prevent. The degree of bank conflicts in the device memory is estimated using arrays' dimensionality because higher dimensionality leads to strided accesses that are more likely to incur bank conflicts. Experiments also reveal that the frequency of bank conflicts also correlates to the thread block size in a nonlinear way. 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. Finally, the effectiveness of latency hiding is also empirically approximated using an artificial factor.
- *Insufficient technical details.* Due to the lack of information regarding the launching of kernel functions, we are not able to accurately estimate the latency for global synchronization. Instead, we measure the latency of global synchronization based on a simplified, empirical model.
- *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 is 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-efficiency* 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

$$CostEffect = \frac{\prod_{i=0}^{D}(blk\_len - HaloWidth_i \times h)}{\prod_{i=0}^{D}(blk\_len - HaloWidth_i)} \tag{19}$$

For a given trapezoid height, a larger tile size always increases the cost-efficiency of trapezoid, reducing the percentage of stencil operations to replicate and achieving better performance, as we will demonstrate in Sect. 6.4. 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 [7]. 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 Eq. 1. While Eq. 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:

$$\forall i \in [0, D), blk\_len - HaloWidth_i \times h > 0 \tag{20}$$

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. number of SMs, warp size, etc).

[IJPP] The performance model then predicts the optimal trapezoid height according to different applications and thread block sizes. In the example of Fig. 7, we configure

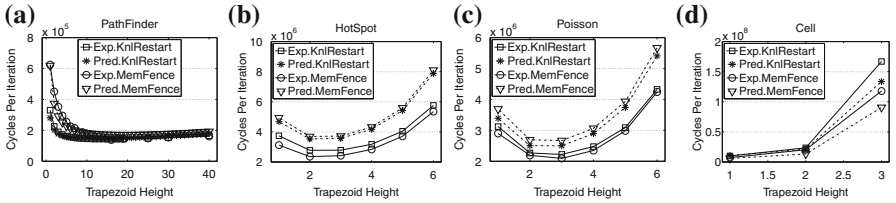

Fig. 7 Evaluating the performance optimization by scaling the trapezoid height of **a** PathFinder, **b** HotSpot, **c** Poisson, and **d** Cell. "Exp" denotes actual performance measurement obtained from experiments, "Pred" denotes predicted performance obtained from the analytical model. Two types of implementations for global synchronization are evaluated; "KnlRestart" corresponds to restarting kernel functions, and "MemFence" corresponds to using the memory fences. The performance at the predicted trapezoid height is no worse than 95% of the optimal performance.

PathFinder with a thread block size of 256, HotSpot and Poisson with a thread block size of 20×20, and Cell with a thread block size of 8. The input size involved 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 Fig. 7. Using the optimal trapezoid height obtained from actual experiments, the speedups compared to performance without ghost zones are 2.32X, 1.35X, 1.40X, 1.0X for PathFinder, HotSpot, Poisson and Cell, respectively; if memory fences are used, the speedups become 4.50X, 1.32X, 1.29X, 1.0X, respectively. The predicted optimal trapezoid heights turn out to exactly match the experimental results except for Poisson and PathFinder. In the case of Poisson implemented with memory fences, the optimal trapezoid height is predicted to be 2 but it is actually 3. In the case of PathFinder, the optimal trapezoid height is 16, and it is predicted to be 19 with memory fences and 12 without memory fences. Nevertheless, the performance at the predicted trapezoid height is no worse than 95% of the optimal performance.

Our estimation of the optimal trapezoid height can also serve as the initial guess for an adaptive selection method, assuming kernels are restarted after each stage to reset the trapezoid height. Suppose run-time profiling calculates the average execution time for every two stages 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, 1.35X, 1.40X, and 0.92X for PathFinder, HotSpot, Poisson and Cell, while an initial guess of one, as proposed in [1], results in speedups of 1.94X, 1.32X, 1.36X and 0.92X, respectively. The slowdown in Cell is because the performance does not benefit from ghost zones, however, the adaptive method still probes a suboptimal trapezoid height of two.

5.5 The Choice to Use Memory Fences

[IJPP] As described in Sect. 4.4 and Sect. 4.5, global synchronization can be implemented by either restarting kernel functions or using memory fences. The use of memory fences has two effects. First, thread coordinates and boundary conditions can persist through stages within a single kernel function call, and their corresponding

computation, $MiscComp$, becomes negligible compared to the overall computation in all stages. Secondly, the overhead in global synchronization increases linearly with the number of thread blocks, which may eventually incur a latency longer than that resulted from restarting kernels. According to our performance model, the additional GPU cycles resulting from using memory fences can be estimated as:

$$\begin{aligned} Overhead_{MemFence} &= GlbSync_{MemFence} - GlbSync_{KnlRestart} \\ &\quad - MiscComp \times T \div M \qquad (21) \\ &= \mu \times 210.3 \times T - 3350 \\ &\quad - T \times \frac{InstsPerSM_{MC}}{N} \times blk\_len^{D} \times CPI \qquad (22) \\ &= T \times \delta - 3350 \qquad (23) \end{aligned}$$

Given a particular architecture, a particular application, and a chosen block size, Eq. 23 shows that the overhead of memory fences only correlates to the number of thread blocks, T. As T increases along with the input size or the trapezoid height, eventually the factor δ determines whether using memory fences is beneficial. A positive δ predicts that using memory fences incurs overhead and thus restarting kernels is preferred. A negative δ predicts that using memory fences is beneficial. The absolute value of δ represents the confidence of the prediction. For the configurations used for Fig. 7, the δ values are 87 for PathFinder, 21 for HotSpot, 25 for Poisson, and -464 for Cell. This prediction is reflected in our observation in Fig. 7 where memory fences benefit the performance of Cell and hurts the performance of PathFinder. However, due to the approximate in modeling global synchronization, our model is not able to precisely predict whether using memory fences is beneficial to performance, especially when the confidence is relatively low. HotSpot and Poisson are examples. The model predicts that their performance will suffer from the memory fence, but they benefit instead. Nevertheless, such misprediction only occurs when the performance with memory fences is similar to that with restarting kernels.

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 accesses, and global synchronization. We also investigate what applications and system platforms may benefit more from using ghost zones.

6.1 Component Analysis

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

$$\begin{aligned} CPL = &\ GlbSync^{*} + LoadSten^{*} + Commit^{*} \\ &+ MiscComp^{*} + IterComp^{*} + IterMem^{*} \qquad (24) \end{aligned}$$

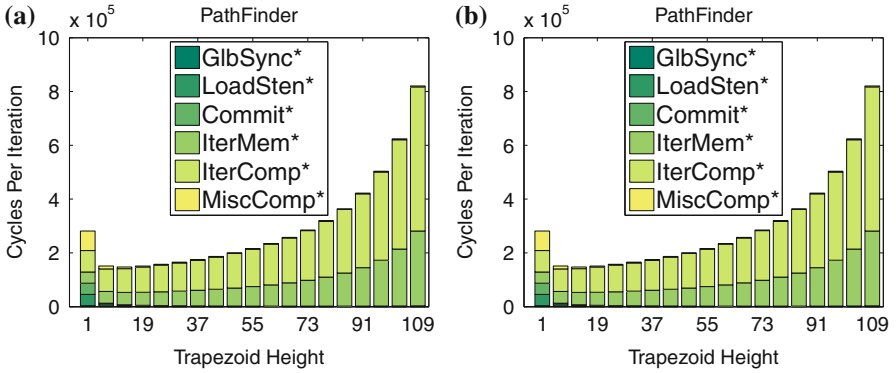

Fig. 8 The estimated performance breakdown for PathFinder obtained from our analytical model. Data is based on an input size of 1,000,000 and a block size of 256. **a** Global synchronization by restarting kernels **b** Global synchronization with memory fences

$$GlbSync^* = \frac{GlbSync}{h} \tag{25}$$

$$LoadSten^* = \frac{LoadSten}{h} \tag{26}$$

$$Commit^* = \frac{Commit}{h} \tag{27}$$

$$MiscComp^* = \frac{MiscComp \times T}{h \times M} \tag{28}$$

$$IterComp^* = \frac{IterComp \times T}{h \times M} \tag{29}$$

$$IterMem^* = \frac{IterMem}{h} \tag{30}$$

We use PathFinder as an example to show the estimated performance breakdown obtained from our analytical model. As shown in Fig. 8, all components contribute significantly to the overall performance except for $GlbSync^*$ in the case of restarting kernels and $MiscComp^*$ in the case of memory fences. As the trapezoid height increases, time spent in $LoadSten^*$ and $Commit^*$ drops while $IterMem^*$ and $IterComp^*$ increase gradually. Eventually, $IterMem^*$ and $IterComp^*$ dominate the execution time. Note that $IterMem^*$ only exists for those applications that access additional device memory objects during trapezoid computation.

Figure 9 illustrates in more detail about 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 $LoadSten^*$, $Commit^*$, $MiscComp^*$, and $GlbSync^*_{MemFence}$ decreases first with taller trapezoids

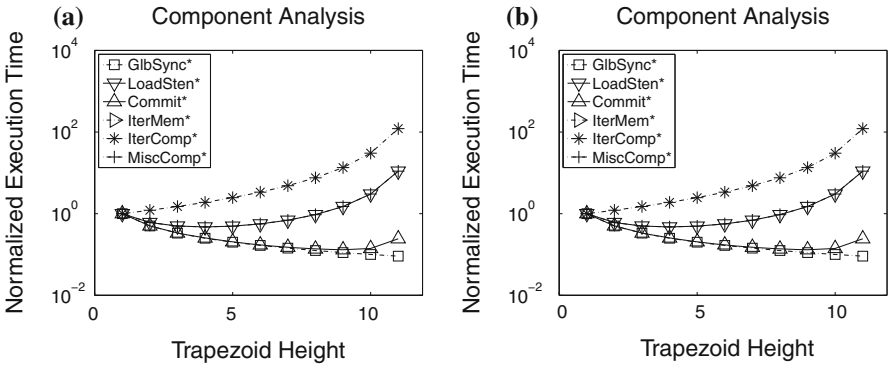

Fig. 9 The effect of the trapezoid's height on each component with the execution time for each component normalized to the case where a trapezoid's height is one. **a** Global synchronization by restarting kernels **b** Global synchronization with memory fences

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, Eqs. 25–30 can be regarded as functions of h, and their derivatives are all monotonically increasing, as can be seen from Fig. 9. As a result, the overall objective function is convex and it has a unique minimum.

6.2 Insensitive Factors

[IJPP] Although the uncontended latency ($UncontendedLat$) and the number of concurrent thread blocks per SM ($BlksPerSM$) is directly used in our performance model to calculate Eq. 7, they are not sensitive to the overall performance. Under the architecture of GTX 280 and assuming a full block size ($blk\_len^D = 512$), Eq. 7 can be re-written as

$$MemCycles(n) = n \times \frac{UncontendedLat}{blk\_len^D \times M \times BlksPerSM} + n \times \frac{\alpha \times x \times ClockRate}{MemBandwidth} \tag{31}$$

$$= 0.0024 \times n + \alpha \times 0.0367 \times n \tag{32}$$

with $\alpha \geq 1$. As a result, the first term involving $UncontendedLat$ and $BlksPerSM$ is at least an order of magnitude smaller than the second term. Therefore, the values of $UncontendedLat$ and $BlksPerSM$ hardly impact the overall performance. In fact, the two additive terms in Eq. 32 each represents the memory access time attributed to latency and bandwidth, respectively. It reflects the fact that the GPU performance is usually bandwidth-bounded rather than latency-bounded.

The choice of the optimal trapezoid height also has little to do with the input size of the application. This is because in Eq. 24, all the terms are linearly related to the input size, n, except for $GlbSync^*_{KnlRestart}$ when global synchronization is implemented

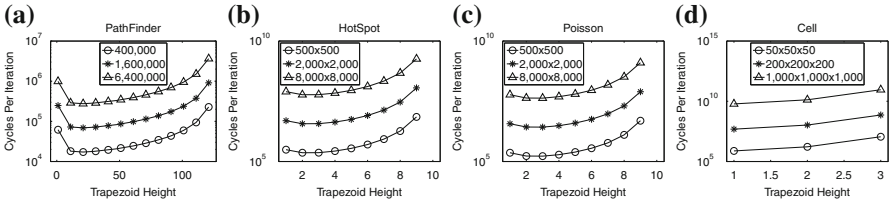

Fig. 10 Compare the effect of various input sizes for **a** PathFinder, **b** HotSpot, **c** Poisson, and **d** Cell. The optimal trapezoid height is independent of the input size. Data is estimated using thread blocks sized 256 for PathFinder, 20×20 for HotSpot and Poisson, and $8 \times 8 \times 8$ for Cell. Global synchronization is implemented with memory fences but the overall trend is almost identical to that implemented by restarting kernels

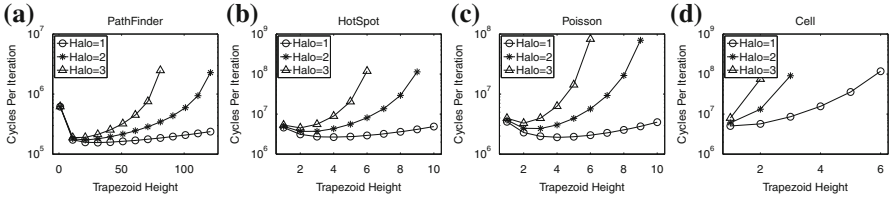

Fig. 11 Comparing the effect of various halo widths for **a** PathFinder, **b** HotSpot, **c** Poisson, and **d** Cell. Global synchronization is implemented with memory fences but the overall trend is almost identical to that implemented by restarting kernels. Smaller stencils with smaller halo width can benefit more from the trapezoid technique, as demonstrated by our performance modeling by synthesizing four benchmarks with various halo width

with restarting kernels. However, $GlbSync^*_{KnlRestart}$ becomes negligible compared to other terms when the input size is sufficiently large, as illustrated in Fig. 8(a). Overall, the optimal trapezoid height remains the same for different input sizes, as illustrated in Fig. 10.

6.3 Application 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 Eq. 8, 16, and 10. This phenomenon is observed in Sect. 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 (Fig. 11).

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.

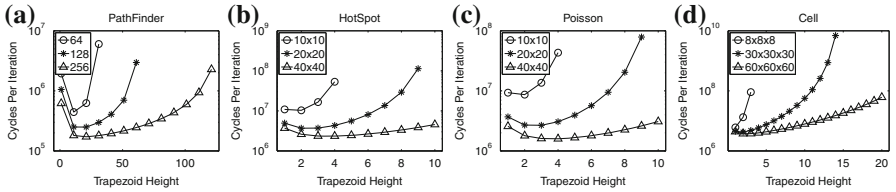

Fig. 12 Compare the effect of various block sizes for **a** PathFinder, **b** HotSpot, **c** Poisson, and **d** Cell. Global synchronization is implemented with memory fences but the overall trend is almost identical to that implemented by restarting kernels. Code programmed with larger thread blocks can benefit more from ghost zones, as demonstrated using our performance modeling to scale the thread block size beyond the architectural limit

6.4 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 adding ghost zones can benefit across different system parameters.

Larger tile size. As we discussed in Sect. 5.4.1, by using larger tiles, less computation needs to be replicated and performance can be improved. As Fig. 12 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. Note that the benchmark of Cell may benefit from the ghost zone technique if the architecture allows larger tile size or thread block size.

Longer synchronization latency. One benefit of ghost zones is the elimination of inter-loop synchronization. As Fig. 9 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 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.

6.5 Energy Consumption

Using ghost zones, the savings in energy consumption are approximately linear with the reduction in execution time. We measure the power consumption using the power meter WattsUp [18]. Because power is measured by plugging the machine's AC power into the WattsUp, the measurements are filtered by the behavior of the power supply. Therefore, the data collected from the WattsUp represents the total power that includes

leakage and dynamic power spent by CPU, GPU, and buses. While the overall power increases around 84 watts on average after CUDA kernels are launched, we do not observe a noticeable change in power by using ghost zones. As a result, it appears that energy can be modeled as linear with the execution time, regardless of whether ghost zones are used. Since ghost zones do improve performance in most cases we studied (Sect. 5.4), they also improve energy efficiency.

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 non-trivial 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 Code annotation for automatic trapezoid optimization

```
morekeywords
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);
```

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 transforms 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 Sect. 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 Sect. 5.4—and generates appropriate code.

8 Conclusions and Future Work

We establish a performance model for 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 95% 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.

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.

References

1. Allen, G., Dramlitsch, T., Foster, I., Karonis, N.T., Ripeanu, M., Seidel, E., Toonen, B.: Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In: SC'01, pp. 52–52 (2001)
2. Alpert, M.: Not just fun and games. April (1999)
3. Bromley, M., Heller, S., McNerney, T., Steele, G.L. Jr.: Fortran at ten gigaflops: the connection machine convolution compiler. PLDI '91 **26**(6), 145–156 (1991)
4. Chatterjee, S., Gilbert, J.R., Schreiber, R.: Mobile and replicated alignment of arrays in data-parallel programs. In: SC'93, pp. 420–429 November (1993)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general purpose applications on graphics processors using CUDA, June (2008)
6. NVIDIA Corporation. Geforce gtx 280 specifications. (2008)
7. NVIDIA Corporation. NVIDIA CUDA visual profiler. June (2008)
8. Dagum, L.: OpenMP: a proposed industry standard API for shared memory programming, October (1997)
9. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC '08. 1–12 (2008)
10. Deitz, S.J., Chamberlain, B.L., Snyder, L.: Eliminating redundancies in sum-of-product array computations. In: ICS '01, pp. 65–77 (2001)
11. Evans, L.C.: Partial differential equations. Am. Math. Soc. (1998)
12. Chen, L., Zhang, Z.-Q., Feng, X.-B.: Redundant computation partition on distributed-memory systems. In: ICA3PP '02, pp. 252 (2002)
13. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: ICS'05, pp. 361–366 (2005)
14. Goodnight, N.: CUDA/OpenGL fluid simulation. April (2007)
15. Gschwind, M.: Chip multiprocessing and the cell broadband engine. In: CF'06 (2006)

16. Huang, C.-H., Sadayappan, P.: Communication-free hyperplane partitioning of nested loops. J. Parallel Distrib. Comput. **19**(2), 90–102 (1993)
17. Huang, W., Stan, M.R., Skadron, K., Ghosh, S., Sankaranarayanan, K., Velusamy, S.: Compact thermal modeling for temperature-aware design. In: DAC'04. (2004)
18. Electronic Educational Devices Inc. Watts up? electricity meter operator's manual. (2002)
19. Jalby, W., Meier, U.: Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system, pp. 429–432 (1986)
20. Kamil, S., Husbands, P., Oliker, L., Shalf, J., Yelick, K.: Impact of modern memory subsystems on cache optimizations for stencil computations. In: MSP'05, pp. 36–43 (2005)
21. Kowarschik, M., Weiß, C., Karl, W., Rüde, U.: Cache-aware multigrid methods for solving poisson's equation in two dimensions. Computing **64**(4), 381–399 (2000)
22. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Effective automatic parallelization of stencil computations. PLDI '07 **42**(6), 235–244 (2007)
23. Lee, P.: Techniques for compiling programs on distributed memory multicomputers. Parallel Comput. **21**, 1895–1923 (1995)
24. Li, Z., Song, Y.: Automatic tiling of iterative stencil loops. ACM Trans. Program. Lang. Syst. **26**(6), 975–1028 (2004)
25. Manjikian, N., Abdelrahman, T.S.: Fusion of loops for parallelism and locality. Parallel Distrib. Syst. **8**, 19–28 (1997)
26. Meng, J., Skadron, K.: Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In: ICS '09, pp. 256–265 (2009)
27. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
28. Premnath, K.N., Abraham, J.: Three-dimensional multi-relaxation time (mrt) lattice-Boltzmann models for multiphase flow. J. Comput. Phys. **224**(2), 539–559 (2007)
29. Ramanujam, J.: Tiling of iteration spaces for multicomputers. In: Proceedings International Conference Parallel Processing, pp. 179–186. (1990)
30. Renganarayana, L., Harthikote-Matha, M., Dewri, R., Rajopadhye, S.: Towards optimal multi-level tiling for stencil computations. IPDPS'07, pp. 1–10, March (2007)
31. Renganarayana, L., Rajopadhye, S.: Positivity, posynomials and tile size selection. In: SC '08, pp. 1–12 (2008)
32. Ripeanu, M., Iamnitchi, A., Foster, I.: Cactus application: Performance predictions in a grid environment. In: EuroPar'01. (2001)
33. Rivera G., Tseng, C.-W.: Tiling optimizations for 3D scientific computations. In: SC '00, p. 32 (2000)
34. Ueng, S.-Z., Baghsorkhi, S., Lathara, M., Hwu, W.m.: CUDA-lite: Reducing GPU programming complexity. In: LCPC'08. (2008)
35. Wonnacott, D.: Time skewing for parallel computers. In: WLCPC'99, pp. 477–480 (1999)
36. Wonnacott, D.: Achieving scalable locality with time skewing. Int. J. Parallel Program. **30**(3), 181–221 (2002)
37. Yang, Z., Zhu, Y., Pu, Y.: Parallel image processing based on CUDA. Int. Conf. Comput. Sci. Software Eng. **3**, 198–201 (2008)