Optimizing and Auto-Tuning Iterative Stencil Loops for GPUs with the In-Plane Method

Wai Teng Tang*, Wen Jun Tan*, Ratna Krishnamoorthy†, Yi Wen Wong†,
Shyh-hao Kuo‡, Rick Siow Mong Goh‡, Stephen John Turner* and Weng-Fai Wong†
*School of Computer Engineering, Nanyang Technological University, Singapore
†Department of Computer Science, School of Computing, National University of Singapore
‡Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore
Email: wttang@ntu.edu.sg

***Abstract*—Stencils represent an important class of computations that are used in many scientific disciplines. Increasingly, many of the stencil computations in scientific applications are being offloaded to GPUs to improve running times. Since a large part of the simulation time is spent inside the stencil kernels, optimizing the kernel is therefore important in the context of achieving greater computation efficiencies and reducing simulation time. In this work, we proposed a novel *in-plane* method for stencil computations on GPUs and compared its performance with the conventional method implemented in the Nvidia SDK. We also implemented an auto-tuning framework for our method to select the optimal parameters for different GPU architectures. A performance model was developed for our proposed method, and is used to speed up the auto-tuning process. Our results show that a speedup of nearly 2× can be achieved compared to Nvidia's implementation.**

***Keywords*-GPU, stencil, auto-tuning**

I. Introduction

Iterative stencil computations form the core of many scientific simulations in a number of diverse disciplines, such as computational fluid dynamics, heat diffusion, electromagnetics, as well as image processing. In many of these applications, the bulk of the simulation time is typically spent in the stencil kernel. It is therefore important to optimize and tune the stencil kernel in order to reduce the overall simulation time, as well as scale the simulation to larger problem sizes.

In a stencil computation, the stencil kernel iteratively sweeps over a large spatial grid many times, and computes for each grid point, a value from the data in its neighboring grid points. Owing to the data parallelism that is inherent in a stencil kernel, a homogeneous many-core architecture such as the GPU is an ideal platform for targeting such computations. Apart from the well-structured and uniform memory accesses within the stencil kernel, the absence of communication between grid elements in the lattice during the computation also makes the GPU suitable for stencil applications. Furthermore, programming models such as CUDA and OpenCL [1], [2] make it easier for scientists to exploit the *Single Instruction Multiple Thread* (SIMT) paradigm that is so well suited for the class of stencil computations. In fact, there has been a number of works describing the use of the GPU for various types of stencil computations (for example, see [3]–[7]).

Although a considerable performance increase can be achieved simply by directly porting a stencil kernel over to the GPU using CUDA, even greater gains can be achieved by carefully tuning and optimizing the code for the underlying hardware architecture. Many different optimization techniques for stencil computations have been proposed, including array padding, prefetching, and various tiling or blocking mechanisms to improve memory bandwidth usage [8]–[16]. These works have reported substantial performance gains compared to a kernel that is not optimized for the underlying GPU hardware.

The key to good performance in optimized stencil kernels is the reduction of memory bandwidth usage by the exploitation of locality of reference that is present in the stencil computations. Owing to the significant overlap of input data that is used in calculating each adjacent grid point, data reuse can be achieved either by relying on hardware managed cache (such as the L1 and L2 caches on the Nvidia Fermi architecture) or through judicious use of registers and shared memory in a GPU. The latter has been shown to yield better performance than the former [17].

In this paper, we will explore different data access strategies to improve the performance of stencil kernels. In particular, we propose a novel algorithm for computing stencil operations based on an *in-plane* method as opposed to the more commonly used *forward-plane* method. One of the main difficulties of stencil computations is the loading of boundary data elements known as the *halo* regions. With the appropriate tuning, we show that our in-plane method is able to obtain optimal parameters on different GPU platforms, and is able to achieve better performance than prior methods.

The main contributions of this paper are as follows: (i) an *in-plane* method is proposed to improve the performance of stencil kernels, (ii) a performance model is developed for the in-plane method and is validated with experimental results, and (iii) performance speedups are demonstrated using actual application kernels.

The rest of the paper is organized as follows. Section II discusses prior work on optimizing stencil computations. In section III, we provide details of our stencil algorithm and discuss their variants in relation to prior methods. Section IV lists our hardware setup and evaluates the performance of

our proposed method. Section V applies our method on real-world application stencils and performs benchmarks on them. In section VI, we develop a performance model for our method that can be used to accelerate the auto-tuning process. Finally, we conclude our paper in section VII.

II. Related Work

Many previous approaches have focused on different levels of blocking to improve cache locality on both multicore CPU and GPU architectures. Datta *et al.* studied and evaluated several optimizations such as array padding, multi-level blocking, loop unrolling and reordering for stencil computations on a wide variety of hardware architectures [11]. In particular, they advocate a decomposition strategy that divides the full grid into thread blocks and register blocks to avoid last level cache misses and to exploit data level parallelism. Their auto-tuning approach then selects the optimal parameters for each architecture. On the Nvidia GTX280 graphics processor with 16×4 computation threads, they showed a sustained performance of 36.5 GFlop/s, or a 3.6× speedup over their naïve CUDA code using 3D blocking.

Another way to improve data reuse and reduce memory traffic is the use of cache oblivious stencil algorithms [12], [13]. Cache oblivious algorithms aim to use the available hardware caches effectively through recursive decomposition of the input data and its associated computations, without having to know in advance parameters such as the cache size and the memory hierarchies [18]. Frigo and Strumpen [13] demonstrated examples of cache oblivious stencil computations using the heat diffusion equation and the 2-dimensional Lattice-Boltzmann Magneto-Hydro-Dynamics, a variant of the Lattice-Boltzmann Method. However, Kamil and coworkers [12] showed that implicit tiling by cache oblivious stencils performed worse than the explicitly tiled and cache-aware approach on the Cell processor.

Apart from blocking in the spatial dimension, some of the previous publications have also looked into temporal blocking as a way to increase data reuse opportunities [14], [16], [19]–[21]. Since choosing the optimal trapezoid (i.e. blocking in both spatial and time dimensions) configuration depends on each stencil kernel, Meng [16] proposed an automated framework to perform parameter selection. In the work by Nguyen *et al.* [14], a 3.5-D blocking method was described which combines 1-D temporal blocking with 2.5-D spatial blocking, instead of 3D spatial blocking. Unlike the full 3D spatial blocking method, their method reduces the overhead and memory bandwidth usage that is required due to less accesses of the overlapped halo regions.

Since many of the blocking parameters are dependent on various factors related to the underlying hardware architecture, such as the number of registers and limitations in cache sizes, a number of code generation and auto-tuning frameworks have been proposed for stencil computations to select the optimal blocking factors for each level of blocking [17], [22]–[24]. For small search spaces, an exhaustive search was used to determine the best run-time parameters [23], whereas for a larger search space, methods like dynamic programming or stochastic search can be used [17].

In many of the above stencil kernels, either a full 3D or a 2.5-D spatial blocking method was used to improve data reuse and reduce memory bandwidth usage. These methods are based on the *forward-plane* method described in [10].

III. Optimizing Stencil Kernels

A. Stencil Kernels

Stencil kernels constitute a major component in partial differential equation (PDE) solvers where iterative finite difference methods are employed to solve PDEs. In an iterative stencil computation, each stencil element in a grid is swept through and a new value is calculated using the values of its neighboring elements from the previous time-step.

A typical Jacobi iterative stencil computation makes use of two data grids, an input data grid, in, and an output grid, out. At a particular time step, the out grid is updated with values calculated from the nearest neighbors of the corresponding grid element from the in data grid. At the next time step, the roles of the grids are swapped, now with the in grid calculated from data from the out grid. This is usually implemented by swapping the pointers to the grids before calling the computational kernel. The iterations then continue until a particular stop criteria has been met. Figure 1 shows the pseudo-code of this iterative process. $initial$ is the initial input data that is passed to the procedure and ComputeKernel(·) denotes a call to the main computational kernel with references to the input and output grid storage locations as parameters.

For our paper, we focused on three-dimensional nearest-neighbor stencil kernels defined by

$$\begin{aligned} out_{i,j,k} &= c_0 * in_{i,j,k} + \\ &\sum_{m=1}^{m=r} c_m * (in_{i\pm m,j,k} + in_{i,j\pm m,k} + in_{i,j,k\pm m}). \end{aligned} \tag{1}$$

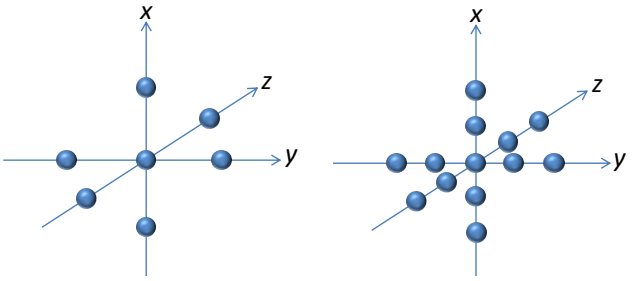
The radius of a given kernel, r, defines the extent of a computation cell, which is given by $(2r+1) \times (2r+1) \times (2r+1)$. A Jacobi stencil of radius r (also known as a $2r$-order stencil) makes use of $6r + 1$ nearest neighbors within its extent for computation. For each element in the grid, a total of $6r + 2$ memory references are required, including a write operation (and excluding loading of the coefficients). A total of $7r + 1$ floating point operations are used to calculate each grid element.

```
1: procedure ITERSTENCILLOOP(initial)
2:     in ← initial
3:     for t = 1 until stop criteria do
4:         ComputeKernel(in, out)
5:         Swap(in, out)
6:     end for
7:     return in
8: end procedure
```

Fig. 1: Iterative stencil computation loop.

TABLE I: List of stencil kernels and their specifications.

Stencil Order	Extent	Memory Accesses/Elem.	Flops/Elem.
2	3×3×3	8	8
4	5×5×5	14	15
6	7×7×7	20	22
8	9×9×9	26	29
10	11×11×11	32	36
12	13×13×13	38	43



(a) 2nd order stencil (b) 4th order stencil

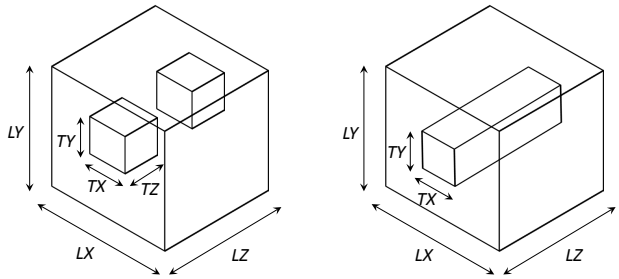
Fig. 2: Stencil kernels

Table I gives the specifications of 2nd to 12th order stencils, while Fig. 2 illustrates the first two stencil kernels, and their neighboring grid points that are needed for computing each grid element.

B. 3D vs 2.5-D Blocking

For exploiting thread-level parallelism, 3D or 2.5-D blocking methods can be used to tile the input grid for stencil computations on a GPU as depicted schematically in Fig. 3. Let the grid or lattice size be $LX \times LY \times LZ$. In the full 3D blocking method, the grid is decomposed into blocks of size $TX \times TY \times TZ$, with each block to be assigned and processed by a corresponding thread block in the GPU. For each block, a data volume containing $(TX+2r) \times (TY+2r) \times (TZ+2r)$ elements need to be accessed, including the additional halo regions in all six faces of the block. Before performing the computations, each data block is usually loaded onto fast on-chip memory in order to improve data reuse. The ratio of extra halo elements loaded to the actual number of elements computed is given by $(1+2r/TX) \times (1+2r/TY) \times (1+2r/TZ)$.

The main principle behind the 2.5-D blocking method is that redundant loads of the halo regions in the z-direction can be reduced since the whole block of data need not be loaded at the same time during computation of the elements inside the block. Instead, for computing points within a single z-plane, only $2r+1$ xy-planes (e.g. 3 and 5 planes for 2nd and 4th order stencils, respectively) are required to be made available at any single instance of the computation. Hence, streaming loads can be used to bring the required data on-line, and the data can be discarded as soon as it is no longer required. Overall, less bandwidth is needed for this method in comparison to the full 3D method where the z-halos are loaded multiple times as the blocks are traversed successively in the z-direction. The



(a) 3-D blocking (b) 2.5-D blocking

Fig. 3: Different spatial blocking methods. Grid is swept in the z direction.

amount of bandwidth required by 2.5-D blocking is reduced by a factor of $(1+2r/TZ)^{-1}$. As an example, a 4th and 8th order stencil using the 2.5-D method will see reductions in bandwidth of 11% and 25% compared to the 3D method if the block size is 32 in all dimensions. In addition, the flops/word ratios will be ∼12 and ∼19, respectively.

The 2.5-D blocking method is employed in the Nvidia Cuda SDK sample [25] (we will refer to the Nvidia stencil code as *nvstencil*). In order to maximize reuse and reduce data fetches from the global memory, shared memory is used as a buffer for inter-thread communication. The buffer is used to store elements from the currently loaded z-plane that are used to compute the output. Each thread also stores a local copy of the $2r+1$ data elements in the range $[z-r, z+r]$ in registers. As the threads coherently traverse down the z-axis, the registers are moved and updated in a pipelining fashion. In this way, only $2r+1$ planes are resident in the registers while the previous points can be discarded.

However, the *nvstencil* baseline code suffers from low bandwidth utilization efficiency due to poor distribution of memory access among threads during loading of the halo regions. Whereas interior threads access at most a single memory location, threads within the perimeter of r issue two load instructions for the halo regions. In the worst case, threads from the top-left area of a thread block attempt to load from four different memory locations by separately issuing four memory load instructions. This is illustrated in Fig. 4 for a 2nd order stencil. Such a loading pattern is inefficient because,

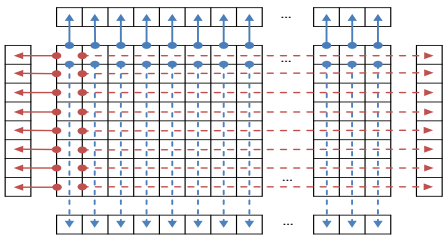
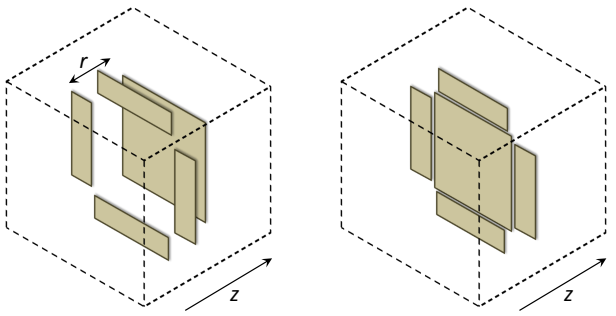


Fig. 4: Distribution of threads accessing the halo regions in *nvstencil*.



(a) Forward-plane loading (b) In-plane loading

Fig. 5: Forward and in-plane loading methods. Arrows indicate direction of sweep.

firstly, non-coalesced memory accesses of the left and right halos will increase the number of memory bus transactions as well as reduce bandwidth usage efficiency. Secondly, many of the threads which are loading the interior elements will be idle most of the time.

C. In-Plane Loading

In this section, we propose another memory access method that is different from the one employed by *nvstencil*. *nvstencil* uses the *forward-plane* loading method to fetch data from the grid. This can be visualized in Fig. 5a, where the plane in which input elements are loaded is located a distance r (stencil radius) away from the plane containing the halos and the output elements. In a typical forward-plane method, the interior elements are loaded first followed by the halo elements. The forward-plane method computes for each point (i, j, k) when $z = k$,

$$out^F_{i,j,k}|_{z=k} = c_0 * in_{i,j,k} + \sum_{m=1}^{m=r} c_m * (in_{i\pm m,j,k} + in_{i,j\pm m,k} + in_{i,j,k\pm m}). \quad (2)$$

Note that r forward planes are required to calculate the output in Eqn. (2).

In the *in-plane* method that we propose, the plane in which the interior elements are read is made to coincide with the plane containing the halos as shown in Fig. 5b. This opens up a few possibilities in which the memory loading patterns can be varied, which will be discussed subsequently. The main difference between the two methods is that the former fetches all neighbor values for calculating each output element, while our method performs incremental updates of the output element from partial reads of the neighbor data.

A formal description of our method is as follows. Let $out^I_{i,j,k}|_{z=k}$ denote the value computed using the in-plane method for the point (i, j, k) when $z = k$. We define

$$out^I_{i,j,k}|_{z=k} = c_0 * in_{i,j,k} + \sum_{m=1}^{m=r} c_m * (in_{i\pm m,j,k} + in_{i,j\pm m,k} + in_{i,j,k-m}). \quad (3)$$

TABLE II: Difference in the number of operations per grid point between in-plane method and *nvstencil*.

Stencil Order	Data Refs.	Flops (*in-plane*)	Flops (*nvstencil*)
2	8	9	8
4	14	17	15
6	20	25	22
8	26	33	29
10	32	41	36
12	38	49	43

Using Eqns. (2) and (3),

$$out^F_{i,j,k}|_{z=k} = out^I_{i,j,k}|_{z=k} + \sum_{m=1}^{m=r} c_m * in_{i,j,k+m} \quad (4)$$

By substituting $out^F_{i,j,k}|_{z=k}$ with $out^I_{i,j,k}|_{z=k+r}$ in Eqn. (4), we can obtain for $p = 1$ to r, a recurrence relation

$$out^I_{i,j,k}|_{z=k+p} = out^I_{i,j,k}|_{z=k+p-1} + c_p * in_{i,j,k+p}. \quad (5)$$

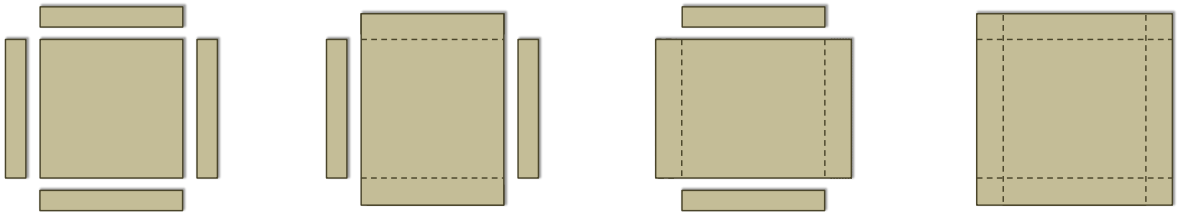
Equation (4) shows that at $z = k$, the computed value for the in-plane method is partial and not yet complete. By pipelining (with depth $p = r$) and aggregating the output as shown in Eqn. (5), the computed value for $out^I_{i,j,k}$ is complete when $z = k + r$.

In our implementation, the write operation at each z-plane is essentially delayed until the next r planes have been loaded. As such, a total of r output elements are cached in registers. After each plane is loaded, the previous output elements are updated and shifted further into the pipeline. Only the last element in the pipeline is written to memory. This procedure is summarized in the following steps.

1) At $z = k$, load the next plane at $in_{i,j,k}$ into shared buffer
2) Compute partial stencil output with Eqn. (3), using the data in the shared buffer
3) Update r previous outputs which were queued in registers with the current point using Eqn. (5)
4) Shift out and write $out^I_{i,j,k-r}$ to global memory
5) Shift in current $out^I_{i,j,k}$ into queue
6) Repeat steps 1 to 5 until the whole z-axis has been traversed

With in-plane loading, the number of flops per stencil element is increased while the number of data references remain the same. Due to the incremental updating of the output element, we now need for every element, a total of $8r + 1$ floating point operations. Table II summarizes the number of data references and flops required per element for the in-plane loading method for various stencil orders. Note that since the z values are also cached in registers as in the 2.5-D blocking method, the bandwidth required by the in-plane method is similar to that of *nvstencil*.

1) Memory Loading Variants: There are a few ways in which the grid data can be loaded using the in-plane method. These variants are shown in Fig. 6. The first *classical* method is very similar to how *nvstencil* loads data from the grid. It is not efficient due to separate memory load instructions that have



(a) Classical (b) Vertical (c) Horizontal (d) Full-slice

Fig. 6: Different variants of the in-plane loading method.

to be issued for the interior points as well as for the boundary halos. Hence, we leave this variant out in our evaluations.

The *vertical* and *horizontal* loading patterns are symmetrical in nature. In the former, we group the top and bottom halos with the interior points and load them together. In the latter, the left and right halos are loaded together with the interior elements. The advantage of these loading patterns is that by removing the separate halo loading in one axis, they improve memory coalecsing and also reduce the number of bus transactions at the same time.

This can be further extended into the *full-slice* pattern in Fig. 6d, where the halo regions in both axes are merged with the interior data. Thus, coalescing of the halo loads is improved at the expense of redundant loads at the four corners of the tile ($4r^2$ redundant data elements). Note that the extra loads depend only on the radius of the stencil, and not on the block size. These three methods will be evaluated and compared with the original *nvstencil* method in section IV.

2) Memory-Level Parallelism: To improve the efficiency of memory throughput, we make use of memory-level parallelism to reduce the number of load instructions. There are altogether $(TX \times TY)$ threads in a single block. With the *full-slice* pattern, we need to load $(TX + 2r) \times (TY + 2r)$ elements. Therefore there will be more than 1 load for each thread. By using memory-level parallelism, we can perform loads of two- or four-element wide vectors. If we are loading using four-element wide vectors, we can reduce the number of load instructions by a factor of 4.

In order to use vector loads, memory alignment for the grid data is important. Two-element vector loads require the grid data to be aligned to 8 bytes while four-element vector loads require alignment to be 16 bytes. For the *horizontal* loading pattern, the top and bottom halos and the center area need to be aligned, as vector loads are used in both the halo and center regions. The *vertical* loading pattern only requires the center region to be aligned, as the left and right halos are loaded separately. The *full-slice* pattern requires the whole block to be aligned. A warp-based assignment method for memory loads is then implemented so that the loading of different data regions is aligned to warp-size. This avoids control flow divergence within a warp. This method also differs from the conventional way of loading the interior data elements followed by the halo regions, which causes idle threads within a warp.

3) Register Tiling: To improve the instruction-level parallelism of our algorithm, we increased the area of the grid that each thread block computes for in both dimensions. RX and RY denotes the scaling factor in the x and y directions. Excluding the halos, a total of $(TX * RX) \times (TY * RY)$ elements are read, computed, and output by a thread block of size $TX \times TY$. To achieve coalescing of the memory locations written by the threads, the indices of the output elements are strided according to the number of threads in each of the x and y directions.

IV. Evaluation

A. Experimental Setup

For our experiments, we used GPUs from Nvidia and programmed the stencil applications using the CUDA programming model. The bulk-synchronous programming model offered by CUDA allows exploitation of thread level parallelism through the use of cooperative thread arrays (CTA) or thread blocks within each Streaming Multiprocessor (SM). The basic unit of execution within each SM is a warp, which is a collection of 32 threads executed in lockstep and operating on different data.

We performed the stencil experiments on the GeForce GTX580, GeForce GTX680 and Tesla C2070 cards as shown in Table III. The GTX580 and Tesla C2070 cards are based on the Fermi architecture while the GTX680 is based on the newer Kepler architecture. We included the GTX680 card so as to evaluate the effectiveness of the in-plane method on the new architecture. The GeForce GTX580 and Tesla C2070 cards contain 512 and 448 cores respectively, which are organized into groups of 32 cores, thereby providing 16 and 14 SMs for each of the cards. Each SM has access to 32K on-chip registers and 48KB of shared memory. In the GeForce GTX680 card based on Kepler, the 1536 cores are organized into 8 units,

TABLE III: Specifications of GPUs used in experiments.

GPU Model	Peak Memory Bandwidth	Peak SP Performance	Peak DP Performance
GeForce GTX580	192.4 GB/s	1581 GFlop/s	198 GFlop/s
GeForce GTX680	192.3 GB/s	3090 GFlop/s	129 GFlop/s
Tesla C2070	144 GB/s	1030 GFlop/s	515 GFlop/s

each of which is called a Next-generation Streaming Multiprocessor (SMX). Each SMX comprises 192 cores which can access up to 65536 registers and 48KB of shared memory.

GTX680 has the highest single-precision (SP) floating point performance. However, the double-precision (DP) floating point performance of GTX580 and GTX680 are 1/8th and 1/24th, respectively, of their SP performance. The GTX580 and GTX680 cards have comparable pin bandwidths, both of which are higher than the Tesla C2070. We also measured the throughput achievable on each GPU and obtained 161 GB/s on GTX580, 150 GB/s on GTX680 and 117.5 GB/s on Tesla C2070. The bandwidths achieved are typically around 75% to 85% of the pin bandwidths.

B. Comparison of In-plane Loading Variants

We first evaluate and compare the performance of the in-plane variants using a $512 \times 512 \times 256$ test grid for all stencil orders. The Nvidia stencil *nvstencil* (2.5-D forward-loading method) is used as the baseline for our comparison. Our experimental results are shown in Fig. 7. We used the same test harness for experimenting with all the kernel variations, including *nvstencil*. The output of each kernel is verified to be consistent with the result from the CPU-computed stencil output. We first tuned each variant for the optimal thread block size ($TX \times TY$) without register blocking and obtained the highest measurements for each memory access method.

For the in-plane method, results in Fig. 7 show that the vertical and horizontal loading patterns gave a benefit over *nvstencil* for some cases, while giving none in others. *horizontal* outperformed *nvstencil* in almost all cases, whereas *vertical* fared worse, especially for high order stencils. For instance, it experienced significant slowdowns for 10th and 12th order stencils across all GPUs. In short, their performance varied over different stencil orders, as well as different platforms.

Of the in-plane variants, we observe that the full-slice loading method consistently performed the best for all the stencil kernels and GPU platforms. The speedups achieved over *nvstencil* ranged from $\sim$1.2$\times$ to $\sim$1.4$\times$. The highest speedup of $>$1.4$\times$ was typically achieved for the 2nd order stencil.

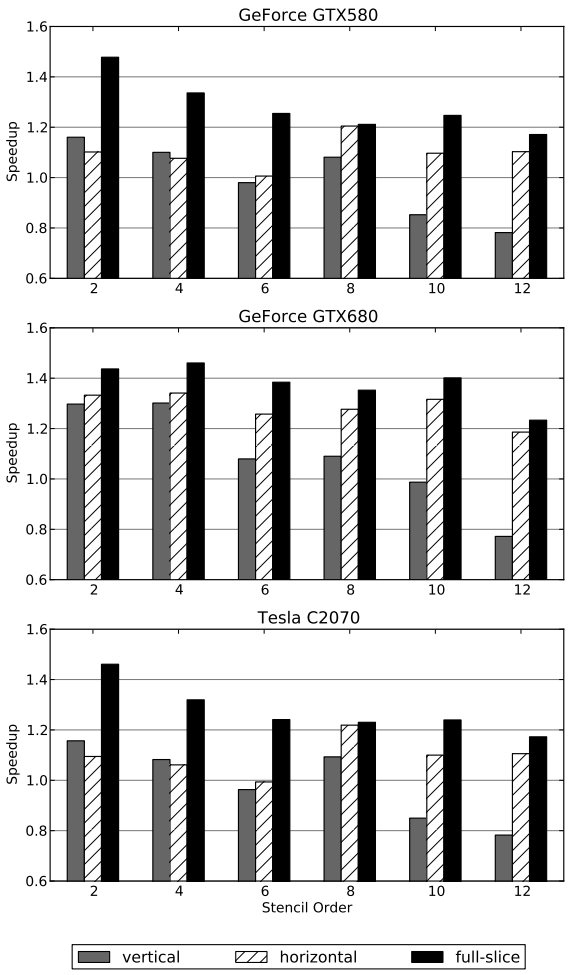


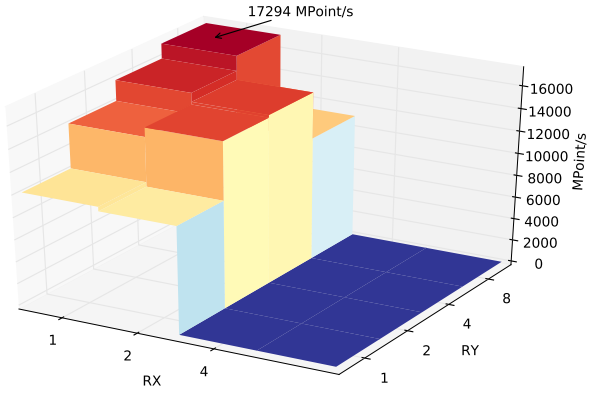
Fig. 7: Speedup of in-plane variants over *nvstencil* with thread blocking only.

C. Auto-Tuning For All Blocking Levels

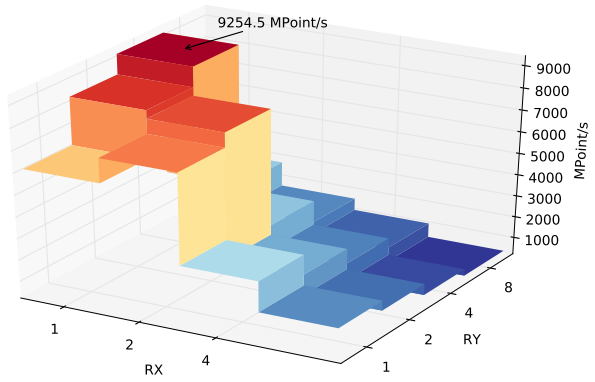
We have also implemented register-level blocking to improve the instruction parallelism of the stencil kernels using the full-slice method. In order to automatically discover the

TABLE IV: Summary of auto-tuning results for the full-slice in-plane method with both thread and register blocking.

	Stencil Order	GeForce GTX580			GeForce GTX680			Tesla C2070		
		Optimal Param.	MPoint/s	Speedup	Optimal Param.	MPoint/s	Speedup	Optimal Param.	MPoint/s	Speedup
SP	2	(256, 1, 1, 8)	17294.0	1.70	(256, 4, 1, 4)	16181.6	1.96	(256, 1, 1, 4)	10761.2	1.65
	4	(32, 2, 2, 4)	14348.6	1.82	(64, 4, 2, 4)	13163.1	1.81	(32, 2, 2, 4)	8994.0	1.77
	6	(32, 8, 2, 2)	10944.2	1.66	(128, 4, 1, 4)	10632.1	1.71	(32, 4, 1, 4)	6965.9	1.65
	8	(32, 4, 1, 4)	9254.5	1.64	(64, 4, 1, 4)	9904.7	1.76	(32, 4, 1, 4)	5949.9	1.66
	10	(32, 8, 1, 2)	7183.9	1.38	(32, 8, 1, 2)	7488.7	1.66	(32, 8, 1, 2)	4550.8	1.39
	12	(32, 8, 1, 2)	6503.6	1.34	(32, 8, 1, 2)	6421.8	1.42	(32, 8, 1, 2)	4130.8	1.34
DP	2	(128, 1, 1, 4)	7206.9	1.35	(64, 2, 1, 4)	6411.6	1.44	(128, 1, 1, 4)	4975.9	1.31
	4	(32, 4, 1, 4)	4858.8	1.30	(64, 4, 2, 4)	4285.0	1.16	(32, 4, 1, 4)	3692.7	1.28
	6	(32, 4, 1, 2)	3432.2	1.16	(128, 4, 1, 4)	3005.8	1.13	(64, 4, 1, 2)	2764.3	1.29
	8	(32, 4, 1, 2)	2788.7	1.12	(64, 4, 1, 4)	2406.4	1.13	(64, 4, 1, 2)	2381.5	1.23
	10	(16, 8, 1, 1)	2388.9	1.15	(32, 8, 1, 2)	1911.0	1.06	(16, 16, 1, 1)	1889.9	1.13
	12	(16, 8, 1, 1)	2029.3	1.05	(32, 8, 1, 2)	1607.8	1.05	(16, 16, 1, 1)	1735.5	1.17



(a) 2nd order stencil, ($TX = 256$, $TY = 1$, $RX = 1$, $RY = 8$)



(b) 8th order stencil, ($TX = 32$, $TY = 4$, $RX = 1$, $RY = 4$)

Fig. 8: Examples of auto-tuning on GeForce GTX580.

optimal parameters for different blocking levels (TX, TY, RX, RY) in our algorithm, we implemented an auto-tuning engine. Selecting the correct parameters is of paramount importance to achieving good performance on the GPU. On one hand, increasing the outer thread block size ($TX \times TY$) relative to the inner register block ($RX \times RY$) increases thread parallelism at the expense of instruction level parallelism. On the other hand, increasing the inner block size relative to the outer block increases the working set per thread at the expense of utilizing more registers, a scarce resource, which invariably leads to lower occupancy or causing spill-overs to global memory. Hence, it is important to balance the trade-offs between the two relative factors to achieve the best performance.

The auto-tuning engine employs an exhaustive search over the parameter space. To constrain the search space, we used the following set of criteria: (i) setting TX to a multiple of a half-warp in order to help with memory coalescing; TY has no such constraint, (ii) ensuring $TX \times TY$ is within the thread limit for each architecture, (iii) making sure the buffer used is within the shared memory limit for each architecture, and (iv) ensuring $TY \times RY$ divides the vertical grid size.

Figure 8 shows an example of the tuning that is performed by the auto-tuner. For the purpose of demonstration, since the auto-tuner searches over a four-dimensional parameter space, we illustrate the search in terms of a performance surface over the RX and RY parameters. The optimal TX and TY parameters are fixed to allow a 3D surface to be plotted. For points which did not satisfy the above search constraints, we set them to zero. Figure 8 shows the performance surfaces for the 2nd and 8th order kernels on GeForce GTX580. For the 2nd order stencil, the peak performance of 17294 MPoint/s occurs with blocking parameters (256, 1, 1, 8). For the 8th order stencil, the best performing point has blocking parameters (32, 4, 1, 4).

Table IV summarizes, for all kernels and platforms, the performance of the in-plane full-slice loading method with the chosen optimal blocking parameters (TX, TY, RX, RY)

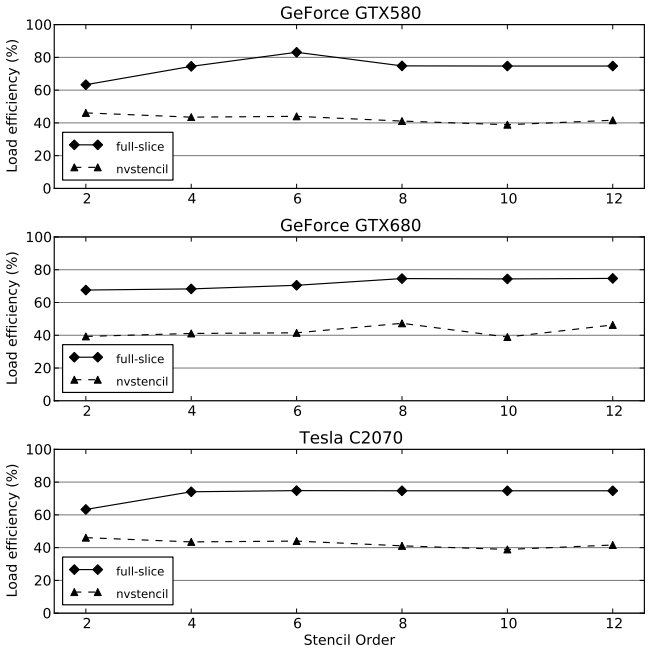


Fig. 9: A comparison of global memory load efficiency.

from the auto-tuning process. The table also lists the best speedup in relation to *nvstencil* that can be achieved using the full-slice method on the various platforms. The results demonstrate the importance and effectiveness of the auto-tuner in exploring and picking the best performing combinations of blocking factors. The set of optimal parameters differs for different stencil orders, and on different GPU hardware. We observe that the speedup of our method over the conventional forward-plane method in *nvstencil* reaches up to $\sim 2\times$ for the 2nd order stencil on GeForce GTX680.

From Table IV, we find that the speedup generally decreases as the order of the stencil is increased. The highest speedups were obtained for either the 2nd or 4th order stencils, while the

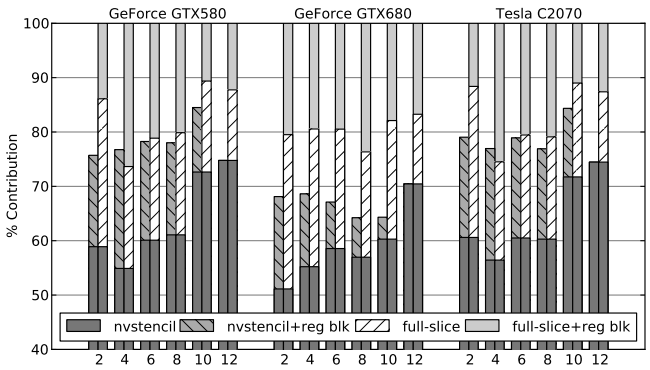


Fig. 10: Breakdown of contributions to performance gain.

12th order stencil registered the lowest speedup. The reason for this is that for the in-plane full-slice method, extraneous data elements in the four corners of each plane are loaded together with the interior elements (compare Fig. 6(d) with Fig. 6(a)). The number of extra data elements loaded increases at a rate of $4r^2$ as the radius of the stencil increases, contributing to the lower speedup achieved for higher stencil orders. For the DP case, only marginal speedup is achieved for high order stencils on GTX580 and GTX680 due to the poorer DP capability of these cards. For Tesla C2070, we find that with the full-slice method, speedups can be achieved for up to 32nd order for SP stencils, and up to 16th order for DP stencils.

In Fig. 9, we plot the global memory load efficiency attained for all stencil orders on the three GPUs. The load efficiency is a measure of the bandwidth requested as a percentage of the effective bandwidth used, and can be used as an indication of how well the memory requests are being coalesced on a GPU. From the graph, we can see that the load efficiency of the full-plane method is higher than *nvstencil* for all stencil orders, implying that better coalescing is achieved by the full-slice method.

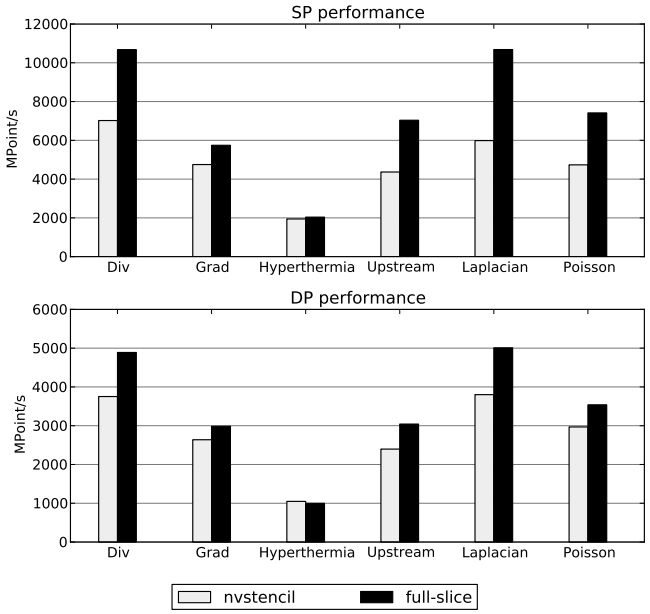
D. Breakdown of Speedup Factors

In Fig. 10, we show a breakdown of the factors that contributed to the performance gain of the kernels, with *nvstencil* as the baseline. We compared three cases: (i) *nvstencil* with register blocking, (ii) full-slice without register blocking, and (iii) full-slice with register blocking. All of them were tuned to obtain the best results. In all cases, we found that the full-slice method with register blocking performed the best across all GPUs.

On average, we found that the full-slice method with register blocking contributed 36% of the performance gain over *nvstencil* on GTX580 and Tesla C2070, and 42% on GTX680. The contributions of register blocking on the full-slice method is about 18%. On the other hand, *nvstencil* with register blocking contributed only around 11%. In a few rare cases, *nvstencil with* register blocking can perform better than full-slice *without* register blocking. However, by including register blocking with the full-slice method, a much higher overall performance can be achieved. On average, about half of the performance gain is contributed by the full-slice method and the other half is contributed by using register blocking on the full-slice method.

TABLE V: List of application stencils benchmarked and the number of input and output grids required.

	Div	Grad	Hypertherm.	Upstream	Laplacian	Poisson
In	3	1	10	1	1	2
Out	1	3	1	1	1	1



	Speedup achieved by full-slice method over *nvstencil*					
	Div	Grad	Hypertherm.	Upstream	Laplacian	Poisson
SP	1.52	1.21	1.05	1.61	1.78	1.57
DP	1.30	1.13	0.95	1.27	1.32	1.19

Fig. 11: Performance and speedup of application stencils listed in Table V.

V. Applications

A. Benchmarks

In this section, we apply the in-plane full-slice method on the application stencils listed in Table V and carry out benchmarks to evaluate the effectiveness of our method in comparison to *nvstencil*. The benchmarks also give us an idea of how well the in-plane method will perform under real-world scenarios. Many of the application stencils are taken from ref. [17].

The first two stencils, *Div* and *Grad*, are commonly used differential operators, the 3D discrete divergence operator and the 3D discrete gradient operator. The former maps a vector field to a scalar function and the latter maps a function to a vector field. The third stencil, *Hyperthermia* is used in simulating the distribution of temperature in the human body during hyperthermia cancer treatment, while the next stencil, *Upstream* is found in weather forecast code (see [17]). The

Laplacian stencil is the 3D discrete Laplacian used in many image processing applications. The last stencil, *Poisson* is used to solve the 3D Poisson equation. These stencils differ in the number of input and output grids that are used in computing the result, and these are summarized in Table V.

Figure 11 shows the performance of the stencils when the forward-plane method (*nvstencil*) and the in-plane methods are used. In general, we obtained better performance when the in-plane method is used, especially when the number of input and output grids are not too large. For instance, the *Laplacian* stencil achieved the greatest speedups of close to 1.8× when only one input and one output grids are required. In contrast, we observe that the speedup achieved by *Hyperthermia* is small, and may even slowdown due to the large number of coefficient grids required (9 out of the 11 grids are used for spatially varying coefficients); any improvement due to the in-plane method was offset by the large amount of coefficient data that had to be loaded.

B. Comparison with Previous Work

To put our performance results into perspective, we compare our results with previous work. Nguyen *et al.* [14] reported a performance of 9234 MPoint/s on GTX285 for a 2nd order SP stencil. On a DP stencil, they obtained ∼4600 MPoint/s. Similarly, Datta [11] obtained 36.5 GFlop/s on GTX280. Since the 2nd order SP stencil is bandwidth-limited, if we extrapolate their results to the GTX580 using the theoretical bandwidths, our performance will be about ∼39% higher than their results for the SP case and ∼16% higher for the DP case. In the stencil code generation and auto-tuning framework by Christen [17], ∼30 GFlop/s was obtained on Tesla C2050 for the SP Laplacian stencil. In comparison, we achieved about 96 GFlop/s on Tesla C2070. The Tesla C2050 and Tesla C2070 GPUs have the same specifications except for the amount of DRAM available. In another auto-tuning framework called Physis [26], the authors obtained 67 GFlop/s on Tesla M2050 for a 7-point SP stencil, while we achieved about 97 GFlop/s. Recently, Holewinski [27] achieved 28.7 GFlop/s on GTX580 for the 7-point 3D Jacobi DP stencil. In our case, we obtained ∼65 GFlop/s on GTX580 for the same stencil.

VI. A Model-Based Approach for Auto-Tuning the In-Plane Method

The auto-tuning method used in section IV-C is exhaustive, and requires every configuration in the parameter space to be executed in order to identify the best configuration. This is very time-consuming. In this section, we develop a simple model for the in-plane method that helps to accelerate the auto-tuning process by restricting the parameter space to be searched.

For a given GPU, let *SM* be the number of streaming multiprocessors in a GPU, *BW* be the off-chip global memory bandwidth, and *Smem* and *Reg* be the per-SM shared memory limit and register limit, respectively. The memory bandwidth is divided evenly among the available SMs, each given by $BW_{SM} = BW/SM$. Furthermore, let Blk_{SM} and $Warp_{SM}$ be the maximum number of blocks and the maximum number of warps that can be scheduled on a SM in the GPU. Let K_R and K_S denote the number of registers and amount of shared memory used by a stencil kernel, and $Warp_{Blk}$ be the number of warps in a thread block. Our model is described by the following equations (6)-(14).

$$Blks = \frac{LX \times LY}{(TX \times RX)(TY \times RY)} \tag{6}$$

$$ActBlks = \min(\lfloor \frac{Reg}{K_R} \rfloor, \lfloor \frac{Smem}{K_S} \rfloor, \lfloor \frac{Warp_{SM}}{Warp_{Blk}} \rfloor, Blk_{SM}) \tag{7}$$

$$Stages = \lceil Blks/(SM \times ActBlks) \rceil \tag{8}$$

$$RemBlks = \lceil \frac{Blks - (Stages - 1) \times ActBlks \times SM}{SM} \rceil \tag{9}$$

$$T_m = \frac{Lat}{Clock} + \frac{Bytes_{Blk}}{BW_{SM}} \tag{10}$$

$$T_c = \frac{ActBlks \times Ops \times RX \times RY \times Warp_{Blk}}{Clock} \tag{11}$$

$$T_s = f(ActBlks) \times T_m + ActBlks \times T_c \tag{12}$$

$$T_l = f(RemBlks) \times T_m + RemBlks \times T_c \tag{13}$$

$$Perf = (LX \times LY)/(T_s \times (Stages - 1) + T_l) \tag{14}$$

Equation (6) determines the number of thread blocks required to complete each plane of the grid, while Eqn. (7) determines the maximum number of active blocks that can fit onto a SM at any time given the kernel resource usage and SM limits. The number of stages needed to finish the computation is calculated in Eqn. (8). In each stage, each SM is populated with the maximum number of available blocks that can fit, given the available resources. Equation (9) computes the number of remaining blocks per SM to be scheduled for the last stage. Equation (10) computes the time required for a thread block to access data in the global memory. *Lat* is the global memory access latency in cycles, and $Bytes_{Blk}$ is the total number of bytes read and written for each stencil plane.

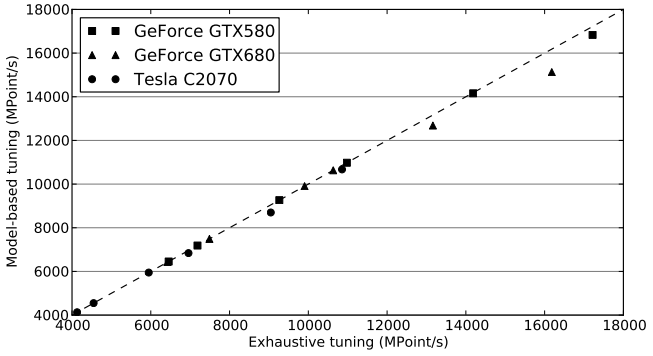


Fig. 12: A comparison of the performance obtained using the model-based auto-tuning approach and the exaustive search method for all stencil orders, with β=5%.

Equation (11) computes the time required for a thread block to finish computing the stencil output. *Ops* denotes the number of flops required for a particular stencil order, and *Clock* is the clock frequency of the GPU. Next, the time taken for each stage to finish its computation is computed using Eqn. (12). For the last stage, Eqn. (13) is used instead. The function $f(arg)$ in Eqns. (12) and (13) is used to model latency hiding during memory accesses, and returns a value between 1 and *arg*. When the total number of warps is $Warp_{SM}$, i.e. at full occupancy, the function assumes perfect latency hiding. On the other hand, if there is only a single warp, then execution becomes serialized. In the current model, $f(arg)$ uses a linear function. Finally, the predicted performance in MPoint/s is computed using Eqn. (14).

There are several sources of inaccuracies with this model. Firstly, the model did not take into account bank conflicts arising from overlapping data accesses by neighboring grid points. Secondly, scheduling overhead of the warps was assumed to be insignificant. Thirdly, cache effects were ignored in the model even though the GPUs employ a small amount of L1 cache for each SM. Nevertheless, these limitations do not prevent the model from being useful. Specifically, we found that the model is helpful in limiting the search space for the auto-tuning process.

The model-based auto-tuning procedure is described as follows. Let M be the total number of configurations in the global parameter space. For every parametric configuration (TX, TY, RX, RY), the performance model (Eqns. (6)-(14)) is used to predict the run-time performance. All the predictions are then ranked in decreasing order starting from the candidate with the best predicted performance. A user-specified cutoff of $\beta\%$ of the total number of configurations is then used to select the top $N = \beta/100M$ candidate configurations from the ranked predictions. Next, all the selected N configurations are executed in order, and the actual run-time performance of each configuration is recorded. Finally, the auto-tuning procedure completes by returning the configuration which has the best actual run-time performance.

Generally, we find that a cutoff β that is 5% of the total parameter space is sufficient to yield a good result. In Fig. 12, we plot and compare the performance of the stencils which have been auto-tuned using the model-based approach with those tuned using the exhaustive search method (for all stencil orders on the GTX580, GTX680 and Tesla C2050 GPUs). β is set to 5%. We can see that, in general, the model-based auto-tuning approach is able to find a configuration that performs close to the best performing configuration obtained using the exhaustive method. The difference in performance obtained using the model-based and exhaustive auto-tuning approach is typically about 2% on average. The largest performance difference between the two approaches is $\sim$6%, and occurs for the GeForce GTX680 GPU. This could be due to architectural differences in the newer Kepler cards which the model does not capture. We intend to improve this aspect in future. Nevertheless, the utility of the model-based auto-tuning approach lies in its ability to yield highly performing configurations while searching only a small fraction of the global parameter space.

VII. Conclusion

Current implementations of stencil kernels on GPUs do not make effective use of the available memory bandwidth due to poor memory coalescing of the halo regions. In this paper, we proposed a novel in-plane loading method that is different from the conventional foward-plane loading method used in the Nvidia SDK, as well as by many other implementations. Our experiments demonstrated speedups of $\sim 2\times$ over *nvstencil*. We also showed that our method can be applied with achievable speedups of close to $\sim 1.8\times$ on real-world application stencils. A model-based tuning approach was developed for the proposed method.

Acknowledgment

This work was supported by the Agency for Science, Technology and Research PSF Grant No. 102-101-0028. We are also grateful to the anonymous reviewers for their comments.

References

[1] *Nvidia CUDA*, http://developer.nvidia.com/cuda/.

[2] *Khronos OpenCL Specification*, http://www.khronos.org/opencl/.

[3] P. Bailey, J. Myre, S. Walsh, D. Lilja, and M. Saar, "Accelerating lattice Boltzmann fluid flow simulations using graphics processors," in *38th International Conference on Parallel Processing (ICPP '09)*, pp. 550 –557.

[4] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," in *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, pp. 244–255.

[5] E. Elsen, P. LeGresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, vol. 227, no. 24, pp. 10 148 – 10 161, 2008.

[6] M. Griebel and P. Zaspel, "A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations," *Computer Science - Research and Development*, vol. 25, pp. 65–73, 2010.

[7] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," in *International Conference on High Performance Computing Simulation (HPCS '09)*, pp. 36 –43.

[8] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *ACM/IEEE Supercomputing Conference*, Nov. 2000, p. 32.

[9] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, Nov. 2004.

[10] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2*, 2009, pp. 79–84.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*, pp. 1–12.

[12] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Proceedings of the 2006 workshop on Memory system performance and correctness (MSPC '06)*, pp. 51–60.

[13] M. Frigo and V. Strumpen, "The memory behavior of cache oblivious stencil computations," *J. Supercomput.*, vol. 39, no. 2, pp. 93–112, Feb. 2007.

[14] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pp. 1–13.

[15] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing gpu memory bandwidth via warp specialization," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pp. 1–11.

[16] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *Proceedings of the 23rd international conference on Supercomputing (ICS '09)*, pp. 256–265.

[17] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS '11)*, pp. 676 –687.

[18] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science*, 1999, pp. 285–297.

[19] J. Habich, T. Zeiser, G. Hager, and G. Wellein, "Enabling temporal blocking for a lattice Boltzmann flow solver through multicore-aware wavefront parallelization," in *21st International Conference on Parallel Computational Fluid Dynamics*, 2009, pp. 178–182.

[20] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, vol. 1, pp. 579–586.

[21] M. Christen, O. Schenk, P. Messmer, E. Neufeld, and H. Burkhart, "Accelerating stencil-based computations by increased temporal locality on modern multi- and many-core architectures," in *High-performance and hardware-aware computing: Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC '08)*, pp. 47–54.

[22] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS '10)*, pp. 1 –12.

[23] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *2012 Symposium on Code Generation and Optimization (CGO '12)*, 2012.

[24] D. Han, S. Xu, L. Chen, and L. Huang, "PADS: A pattern-driven stencil compiler-based tool for reuse of optimizations on GPGPUs," in *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS '11)*, pp. 308 –315.

[25] *Nvidia CUDA SDK stencil sample*, http://developer.nvidia.com/cuda-cc-sdk-code-samples#FDTD3d.

[26] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pp. 1–12.

[27] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM international conference on Supercomputing (ICS '12)*, pp. 311–320.