Toward Harnessing DOACROSS Parallelism for Multi-GPGPUs

Peng Di, Qing Wan, Xuemeng Zhang, Hui Wu and Jingling Xue
Programming Languages and Compilers Group
School of Computer Science and Engineering
UNSW, Sydney, NSW 2052, Australia

***Abstract*—To exploit the full potential of GPGPUs for general-purpose computing, DOACR parallelism abundant in scientific and engineering applications must be harnessed. However, the presence of cross-iteration data dependences in DOACR loops poses an obstacle to execute their computations concurrently using a massive number of fine-grained threads. This work focuses on iterative PDE solvers rich in DOACR parallelism to identify optimization principles and strategies that allow their efficient mapping to GPGPUs. Our main finding is that certain DOACR loops can be accelerated further on GPGPUs if they are algorithmically restructured (by a domain expert) to be more amendable to GPGPU parallelization, judiciously optimized (by the compiler), and carefully tuned by a performance-tuning tool.**

We substantiate this finding with a case study by presenting a new parallel SSOR method that admits more efficient data-parallel SIMD execution than red-black SOR on GPGPUs. Our solution is obtained non-conventionally, by starting from a K-layer SSOR method and then parallelizing it by applying a non-dependence-preserving scheme consisting of a new domain decomposition technique followed by a generalized loop tiling. Despite its relatively slower convergence, our new method outperforms red-black SOR by making a better balance between data reuse and parallelism and by trading off convergence rate for SIMD parallelism. Our experimental results highlight the importance of synergy between domain experts, compiler optimizations and performance tuning in maximizing the performance of applications, particularly PDE-based DOACR loops, on GPGPUs.

***Keywords*-GPGPU, DOACR Parallelism, Loop Tiling, SOR**

I. Introduction

GPGPUs have recently emerged as powerful data-parallel co-processors for general-purpose computing as they provide the tremendous memory bandwidth and computation power at commodity prices. The NVIDIA CUDA programming model facilitates programming of general-purpose applications on modern GPGPUs such as the NVIDIA GeForce 8 Series. Unfortunately, the level of effort and expertise required to maximize application performance on such GPGPUs is still quite high. Some recent research efforts focus on automatic transformation of loops into kernels [2], compiler optimizations [2, 10], higher-level programming models (than CUDA) [10], cost models [6, 16], performance tuning [11, 8]. At this stage, the code parallelized in their benchmarks consists of almost exclusively DOALL loops. Despite the simplicity of DOALL loops, the research community is still gathering the experience and knowledge needed to establish principles and strategies that allow efficient mapping of such computations to GPGPUs. One major challenge is that the underlying architectural constraints and threading model interact in a fairly complex way, making the optimization space discontinuous and the performance of an application hard to predict.

Many scientific and engineering applications exhibit abundant DOACR parallelism. To exploit the full potential of GPGPUs for general-purpose computing, we also need to investigate how to map DOACR (i.e., DOACROSS) loops efficiently to GPGPUs. There are very few prior studies here. Some loop optimizations being developed [2, 10] can tackle DOACR loops generically to a limited extent. There is also a recent work on parallelizing 3D PDEs [13]. However, these techniques are constrained by cross-iteration dependences in DOACR loops. As a result, the presence of such parallelism-inhibiting dependences makes it fundamentally difficult to create a massive number of fine-grained threads on GPGPUs to execute the computations in a DOACR loop concurrently on a massive number of processor cores.

In this work, we focus on iterative PDE solvers to establish optimization principles and strategies for their efficient mapping to GPGPUs. We have chosen to parallelize the 2D SSOR (Symmetric Successive-Over Relaxation) method on GPGPUs for two reasons. First, this method represents one of the most important iterative solvers for large systems of linear equations with a massive amount of data parallelism to be harnessed. Second, the underlying loop nest exhibits representative cross-iteration dependences with both temporal and spatial data reuse to be captured.

We describe our experience in parallelizing SSOR as a case study for GPGPUs by summarizing our contributions below.

- We present a new parallel SSOR method that admits efficient data-parallel SIMD execution in GPGPUs. We have obtained this solution in a non-conventional manner. The starting point is a K-layer SSOR solver that performs K forward SOR sweeps and K backward SOR sweeps alternately. This sequential method is then parallelized by applying a non-dependence-preserving scheme consisting of a new domain decomposition technique followed by a generalized loop tiling to the two sweep directions alternately.
- Despite its relatively slower convergence, our new method outperforms red-black SOR on both one and multiple GPGPUs by making a better balance between data reuse and parallelism and by trading-off convergence rate for SIMD parallelism. Our method will perform even better on some future GPGPU architectures that may allow inter-kernel reuse to be exploited.
- Our experimental results demonstrate that certain DOACR loops can be accelerated further on GPGPUs if they are algorithmically restructured to be more amendable to GPGPU parallelization, judiciously optimized, and carefully tuned by a performance-tuning tool. This highlights the importance of synergy between domain experts, compiler optimizations and performance tuning in maximizing the performance of applications, particularly PDE-based DOACR loops, on GPGPUs.

The rest of this paper is organized as follows. Section II gives a brief review of GPGPUs and CUDA. Section III introduces a multi-layer 2D SSOR iterative solver, with a particular emphasis on understanding its cross-iteration data dependences. Section IV describes our new parallel SOR method. Sections V and VI present and analyze our experimental results on single and multiple GPGPUs, respectively. The performance advantages of our method over red-black SOR are validated and discussed. Section VII reviews further some related work. Section VIII concludes the paper.

II. GPU Architecture

This work uses an NVIDIA Tesla S1070-400 GPU computing system as the basis for its study. The S1070 computing system consists of four Tesla C1060 GPUs to form a large set of processor cores. Each C1060 GPU has 30 streaming multiprocessors (SMs) with each SM containing eight streaming processors (SPs) or processor cores, running at 1.3GHz. Each SP can perform one FMAD (two ops) and one FMUL (one op) for three single-precision FLOPs per cycle. With 240 SPs in total, C1060 has a single-precision peak performance of 936 GFLOPS ($30\text{SMs} \times 8\text{SPs} \times (2+1) \times 1.3\text{GHz}$). With four times as many SPs, S1070 can deliver 3744 GFLOPS ($936\text{GFLOPS} \times 4\text{GPUs}$) of single-precision peak performance.

Every C1060 GPU has 102 GB/s bandwidth to its 4GB off-chip, global memory (called *device memory*). This amount of bandwidth can be easily saturated with computational resources supporting nearly 936 GFLOPS of performance. In addition, a global memory access has very high latency (400 – 600 cycles). As a result, several on-chip memories are available to exploit data reuse so as to lessen an application's demand for off-chip memory bandwidth and reduce expensive off-chip memory traffic. In particular, each SM has a user/compiler-managed 16KB shared memory for data reuse or sharing among threads and 16,384 32-bit registers partitioned among threads. For read-only data, the constant and texture cache memories can significantly reduce memory latency. For the experiments done for SSOR in this work, these cache memories are not used.

In S1070, data exchange among its four C1060 GPUs (i.e., inter-GPU communication) is accomplished through the host.

In the NVIDIA CUDA programming model [14], a GPU works as a co-processor with a host by executing data-parallel kernel functions. A user program is compiled by the NVIDIA compiler into host code and kernel code. The host code transfers data to and from the GPU's device memory via API calls and initiates the execution of each kernel by performing a function call.

GPUs architectures allow a large number of fine-grained threads to cooperate in solving large-scale applications. In CUDA, threads are organized hierarchically into three levels. Each kernel creates its own single grid. A grid is divided into many thread blocks. Each thread block is assigned to a single SM for the duration of its execution. Threads in the same thread block can cooperate by barrier-synchronizing their memory accesses and can share data through the shared memory. Threads are otherwise independent, and synchronization across thread blocks is safely accomplished only by terminating the kernel. Finally, threads within a thread block are organized into warps of 32 threads. Each warp executes in a SIMD fashion, issuing in four cycles on the eight SPs of an SM. When a warp running in an SM stalls, the SM can quickly switch to a ready warp in the same thread block or a ready warp in some other thread block assigned to the same SM.

While maximizing data reuse through the shared memory helps to improve the performance of a kernel, shared memory bank conflicts should be minimized. In addition, reducing the latency in accessing data from global memory is crucial for good performance. Due to a hardware optimization known as *global memory access coalescing*, accesses from adjacent threads in a half-warp to adjacent locations are coalesced into a single contiguous aligned memory access. Thus, the significant performance benefits due to coalesced accesses should be leveraged by compiler optimizations.

Resource	Limit
Number of Threads per Block	512
Number of Active Threads per SM	1024
Number of Active Blocks per SM	8
Shared Memory per SM	16KB
Number of 32-bit Registers per SM	16,384

Table I
CUDA constraints on S1070 and C1060.

Table I lists some architectural constraints imposed on a user program [14]. Due to their complex interactions, it can be difficult to accurately predict the effects of compiler optimizations on the performance of a kernel. Unpublished details about GPU architectures further exacerbate the problem. There is often a tradeoff between the performance of individual threads and the TLP (thread-level parallelism) among all threads [2, 10, 16].

III. The 2D SSOR Iterative Solver

Partial Differential Equations (PDEs) are widely used in scientific and engineering applications. Iterative methods are faster than direct methods in solving a large system of linear equations and are thus often used. Three well-known iterative methods are Jacobi, Gauss-Seidel and Successive-Over-Relaxation (SOR).

Many applications involve boundary value problems that require solving diffusion equations. Consider a 2D case:

$$\Delta u \quad = \quad \frac{\partial^2 u}{\partial i^2} + \frac{\partial^2 u}{\partial j^2} \tag{1}$$

where $\Omega = [0,1] \times [0,1] \in \mathbb{R}^2$ is bounded with $\partial\Omega$ as its boundary. The domain Ω is divided with the step sizes $1/(N_1+1)$ and $1/(N_2+1)$ along the i and j axes, respectively. By using $u_{i,j}$ to denote the finite difference approximation of u at grid point (i,j), we obtain the following five-point approximation of (1):

$$4u_{i,j} - u_{i-1,j} - u_{i,j-1} - u_{i+1,j} - u_{i,j+1} \quad = \quad 0 \tag{2}$$

where $i = 1, \ldots, N_1$ and $j = 1, \ldots, N_2$. The boundary condition is set to be $\partial\Omega = 0$ in the normal manner.

Such a system of equations is often solved using an iterative solver. The Jacobi method updates all grid points at an iteration, say, k using their previous values obtained at iteration $k-1$:

$$u_{i,j}^{k} \quad = \quad \frac{1}{4}(u_{i-1,j}^{k-1} + u_{i,j-1}^{k-1} + u_{i+1,j}^{k-1} + u_{i,j+1}^{k-1}) \tag{3}$$

For the SOR method, the computation of $u_{i.j}^{k}$ uses the values of $u_{i-1,j}^{k}$ and $u_{i,j-1}^{k}$ that have already been computed at iteration k and the old values of $u_{i,j}^{k-1}$, $u_{i+1,j}^{k-1}$ and $u_{i,j+1}^{k-1}$ from iteration $k-1$:

$$u_{i,j}^{k} = (1-\omega)u_{i,j}^{k-1} + \frac{\omega}{4}(u_{i-1,j}^{k} + u_{i,j-1}^{k} + u_{i+1,j}^{k-1} + u_{i,j+1}^{k-1}) \tag{4}$$

The Gauss-Seidel method is a special case of SOR when $\omega = 1$.

A 2D iterative solver is typically implemented using a 3D loop nest, where the inner two loops i and j enumerate all grid points

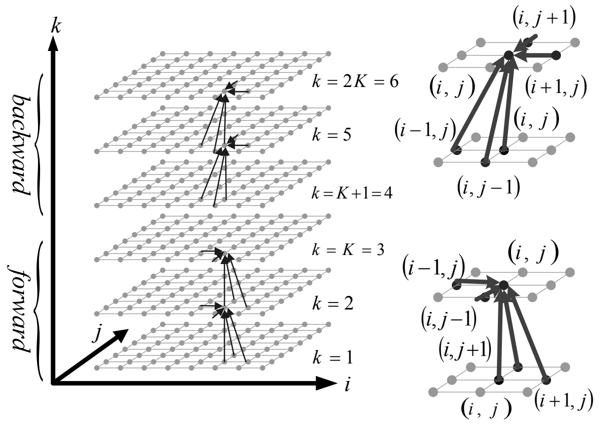

Figure 1. A sequential multi-layer symmetric five-point SOR method (MLSSOR) ($K = 3$). The five dependences at each point in the forward and backward sweeps are directly derived from (4) and (5), respectively.

in the $i - j$ plane, i.e., domain and the outermost loop k performs multiple sweeps, i.e., iterations across the $i - j$ plane. The set of all points (k, i, j) is known as the *iteration space* of the loop nest.

The Jacobi method is inherently parallel since all points can be computed at the same time. However, it is often not used due to its slow convergence and high memory usage. The SOR and Gauss-Seidel methods are known to be inherently sequential in their original forms. With an appropriate choice of the relaxation factor ω, SOR converges faster than Gauss-Seidel.

The symmetric SOR, or SSOR, combines two SOR sweeps together in such a way that the resulting iteration matrix is similar to a symmetric matrix. In other words, SSOR is a forward sweep performed using (4) followed by a backward sweep using:

$$u^k_{i,j} = (1-\omega)u^{k-1}_{i,j} + \frac{\omega}{4}(u^{k-1}_{i-1,j} + u^{k-1}_{i,j-1} + u^k_{i+1,j} + u^k_{i,j+1}) \qquad (5)$$

In this paper, we consider to apply (4) and (5) alternately as illustrated in Figure 1, resulting in what is referred to here as a multi-layer SSOR method. There are K layers since every sweeping direction is repeated K times and the method is symmetric due to the five-point stencil discretization used. In the *forward sweep*, (4) is applied for K iterations, starting at the left and bottom boundaries of the domain and moving towards the right and top boundaries at each iteration. In the *backward sweep*, (5) is applied also K times with the sweeping direction being reversed.

The multi-layer SSOR method, MLSSOR, which is guaranteed to converge [17], enables us to develop a new data-reuse-effective and data-parallel implementation for GPU architectures below.

IV. A Parallel MLSSOR Method for GPGPUs

Several parallel versions of SOR, including red-black SOR (denoted RBSOR) [3], multi-color SOR [1, 12] and block-parallel SOR [22], have been proposed mostly for distributed memory machines. Tang and Xue [19] presented a method for tiling SOR by applying skewing and tiling for distributed memory machines. Goumas et al. [4] later continued this line of investigation by focusing on the parallelization of 2D iteration spaces that result from the discretization of PDEs. In addition, Huang, et al. [7] introduced a code tiling technique for improving the cache performance of PDE solvers for uniprocessors. Michelle et al. [18] presented a parallel Gauss-Seidel method by applying a full sparse tiling technique to improve the cache locality of a program for uniprocessors and shared-memory machines. Wallin et al. [21] considered to temporally tile Gauss-Seidel with pipelining techniques to improve parallelism on shared memory machines.

Due to its simplicity and good performance, RBSOR has often been a popular choice not only for distributed memory machines, but recently, also for GPUs [25]. RBSOR divides a domain of grid points into a chessboard of red and black points. Due to the absence of data dependences between red and black points, the Jacobi method (using SOR) is applied to update the points of one color simultaneously using the previous values computed at the points of the other color. This high degree of fine-grained parallelism makes RBSOR amenable to data-parallel execution on GPUs. On the other hand, RBSOR does not respect the data dependences in the original SOR (and SSOR), resulting in some slightly slower convergence rates under some inputs. In addition, RBSOR exhibits less data reuse (due to red-black ordering) and may suffer from high inter-GPU communication overhead.

In this section, we describe a new parallel multi-layer SSOR algorithm, also denoted MLSSOR (as we will refer to its sequential version as the sequential MLSSOR henceforth), for GPU architectures in order to strike a better balance between fine-grained parallelism and data reuse than RBSOR. Like RBSOR, MLSSOR is developed using a non-dependence-preserving parallelization scheme as follows. First, a new domain decomposition technique is applied to enable simultaneous point updating using fine-grained threads (Section IV-A). Second, a generalized loop tiling, which tiles the two sweep directions alternately, is applied so that the resulting tiled code exhibits the same degree of intra-kernel data reuse but better inter-kernel data reuse than traditional loop tiling (Section IV-B). Although existing NVIDIA GPU architectures cannot exploit inter-kernel data reuse, other stream processors such as Imagine [9] and AMD GPUs can. Despite this, applying the generalized loop tiling to NVIDIA GPUs is still beneficial as it improves the convergence rate as discussed in Section V. Third, a new tile scheduling scheme is introduced to ensure that all SPs in one GPGPU can start executing their subdomains at the same time, resulting in significantly improved SIMD parallelism at the expense of some slightly slower convergence rates than the sequential SSOR under some inputs (Section IV-C). Finally, the overall communication cost is kept to a minimum by overlapping computation and communication on multiple GPUs (Section IV-D).

A. Domain Decomposition

Traditionally, the domain of an SOR solver is partitioned disjointly so that a processor computes all the points in its allotted subdomain in every SOR iteration. So the domain is meant to be the mesh, i.e., the $i - j$ plane for a 2D SOR solver. In MLSSOR, however, the sub-mesh allocated to an SP changes as the iteration proceeds, causing adjacent sub-meshes to overlap at their boundaries. To avoid any confusion, the domain of an MLSSOR loop nest is meant to be its 3D iteration space. As a result, domain decomposition divides the iteration space into 3D rectangular boxes (i.e., parallelepipeds).

The two sweep directions are partitioned as shown in Figure 2. We describe only the technique used for a K-layer forward sweep

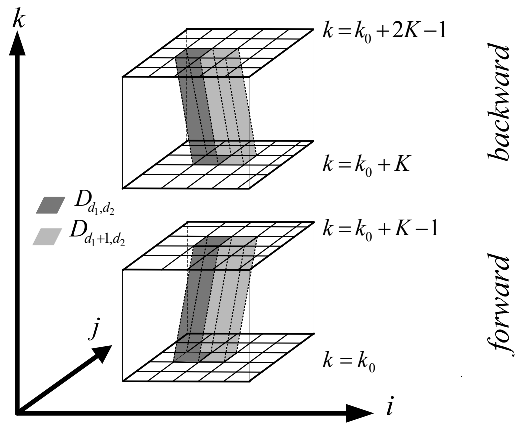

Figure 2. Domain decomposition for two alternate sweeps.

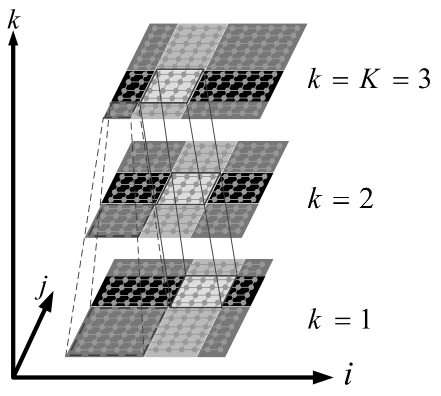

Figure 3. Tiling of the subdomain D_{d_1,d_2} in the forward sweep shown in Figure 2 into $m \times m = 3 \times 3 = 9$ tiles of height $K = 3$. The middle tile is full while all the rest are border tiles.

since it is mirrored by a backward sweep. There are two reasons behind this somewhat unconventional partitioning approach. First, together with our generalized loop tiling, this partitioning approach allows different subdomains to be executed in parallel with the inter-subdomain communication kept to a minimum, as discussed in Section IV-C. Second, as is clear in Section V, better data reuse and convergence rate can be obtained.

Let a 2D mesh of size $N_1 \times N_2$ be partitioned across a 2D mesh of size $P_1 \times P_2$. For simplicity, it is assumed that P_1 divides N_1 and P_2 divides N_2. Consider a K-layer forward sweep starting from $k = k_f$ and ending at $k = k_f + K - 1$. It is partitioned into $P_1 \times P_2$ blocks so that its intersection with k_f layer is divided into $P_1 \times P_2$ rectangles of size $N_1/P_1 \times N_2/P_2$. This is achieved with the following subdomain cutting planes across the K-layers:

$$\begin{aligned} i &= s \times \tfrac{N_1}{P_1} + k - k_f + 1, \quad s = 1, \ldots, P_1 - 1 \\ j &= t \times \tfrac{N_2}{P_2} + k - k_f + 1, \quad t = 1, \ldots, P_2 - 1 \end{aligned} \tag{6}$$

whose normals are $(1, -1, 0)$ and $(1, 0, -1)$, respectively.

All non-border subdomains are 3D rectangles (parallelepipeds) of size $K \times \frac{N_1}{P_1} \times \frac{N_2}{P_2}$. The cutting hyperplanes near the borders of the mesh are so chosen that all subdomains (border or non-border) have roughly the same number of grid points. Let D_{d_1,d_2} be a non-border subdomain located at (d_1, d_2), where $0 \leqslant d_1 < P_1$ and $0 \leqslant d_2 < P_2$. Let $D^k_{d_1,d_2}$ be its k-th layer. Then

$$D^k_{d_1,d_2} = D^{k-1}_{d_1,d_2} + (1,1,1) \tag{7}$$

where $D^{k-1}_{d_1,d_2} + (1,1,1) = \{(k,i,j) + (1,1,1) \mid (k,i,j) \in D^{k-1}_{d_1,d_2}\}$. In this case, every layer $D^k_{d_1,d_2}$ in the subdomain D_{d_1,d_2} is a *translate* of the layer $D^{k-1}_{d_1,d_2}$ below, i.e., drifts away from the coordinate origin, along $(1,1,1)$ as shown in Figure 2.

B. Generalized Loop Tiling

We tile our sequential MLSSOR by using a generalized loop tiling transformation. As can be observed from (4) and (5) and is also illustrated in Figure 1, there exists temporal reuse across all three dimensions in the 3D iteration space of the MLSSOR loop nest. Specifically, each grid point is accessed five times during a sweep across the $i - j$ plane, once by itself and four times by its neighbours, and also accessed multiple times during multiple sweeps. To capture such temporal reuse, all three dimensions must be tiled. Due to the existence of data dependencies in both forward and backward sweeps (Figure 1), it is illegal to simply tile its iteration space by using rectangular boxes.

Figure 3 illustrates the loop tiling being applied to the subdomain D_{d_1,d_2} depicted for a forward sweep in Figure 2. As illustrated in Figure 1, the sweeping direction used for updating the K layers in a forward sweep is reversed in a backward sweep. Thus, the four *non-self* dependencies are also reversed. This results in loop tiling being applied to two sweeping directions alternately, a generalization of traditional loop tiling [23, 24] that tiles the entire iteration space uniformly. In addition, all border tiles are chosen to have different sizes so that they have all the same amount of work. This ensures load balancing among fine-grained threads.

For reasons of symmetry, we explain only how to tile a K-layer subdomain obtained for a forward sweep, where (4) is repeated K times across the $i - j$ plane. In general, a K-layer subdomain is divided into $m \times m$ tiles identified by their tile indices. Let T_{t_1,t_2} be the tile located at (t_1, t_2). Let $K \times M \times M$ be the size of a full tile. Let (k_0, i_0, j_0) be the lexicographically largest point of the bottom-left tile in the subdomain. The subdomain is divided into $m \times m$ tiles by using the following hyperplanes:

$$\begin{aligned} k_0 &\leqslant k < k_0 + K \\ i &= (s-1) \times M + k_0 - k + i_0 + 1, \quad s = 1, \ldots, m-1 \\ j &= (t-1) \times M + k_0 - k + j_0 + 1, \quad t = 1, \ldots, m-1 \end{aligned} \tag{8}$$

whose normals are $(1, 1, 0)$ and $(1, 0, 1)$, respectively.

The main reason for tiling a subdomain this way is to ensure that the subdomains can be executed in parallel as discussed shortly.

There are $(m-2) \times (m-2)$ full (i.e., non-border) rectangular tiles of size $K \times M \times M$ in the center and $m \times m - (m-2) \times (m-2) = 4(m-1)$ border tiles. In Figure 3, only the one in the center is full while all the rest are border tiles.

- **Full Tiles.** Let $T^k_{t_1,t_2}$ be the set of points in the k-th layer of T_{t_1,t_2}. If T_{t_1,t_2} is a a full tile, then we have:

$$T^k_{t_1,t_2} = T^{k-1}_{t_1,t_2} + (1, -1, -1) \tag{9}$$

Thus, every layer in T_{t_1,t_2} is a *translate* of the layer below along $(1, -1, -1)$, as shown by the middle tile in Figure 3.

- **Border Tiles.** A border tile is a boundary tile that is a hexahedron but not a rectangular box as shown in Figure 3.

C. Parallelization for One GPU

We now explain the rationale behind our unconventional domain decomposition and tiling techniques. Our parallelization strategy is simple. All K-layer sweeps are executed sequentially, bottom-up. Each K-layer sweep is executed concurrently by all SPs in a GPU. In order to enable all SPs to start executing at the same time, the tiles with the same tile index from all subdomains are executed in parallel by the same kernel. However, of the $m \times m$ tiles in a subdomain, the tiles in $\{T_{i,j} \mid 0 \leqslant i, j < m\}$ (e.g., the tiles $T_{1,1}$, $T_{1,2}$, $T_{2,1}$ and $T_{2,2}$ in Figure 3) do not have inter-subdomain dependences and can thus be combined into a larger tile. This avoids unnecessary kernel startup overhead. Section V-A5 applies shared memory reduction to deal with large tiles.

In our implementation, every subdomain in a K-layer sweep is therefore partitioned into 2×2 tiles. There are a total of four kernels executing a K-layer sweep. All tiles with the same tile index (t_1, t_2) from different subdomains form a grid executed by the same kernel, denoted $\mathcal{K}_{t_1,t_2}$. For example, suppose that the two adjacent subdomains D_{d_1,d_2} and D_{d_1+1,d_2} in a forward sweep (highlighted by gray in Figure 2) are of the size $K \times 8 \times 8$. They are each divided into four tiles $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ as illustrated in Figure 4. How their sizes are chosen is discussed below.

The four kernels $\mathcal{K}_{1,1}$, $\mathcal{K}_{1,2}$, $\mathcal{K}_{2,1}$ and $\mathcal{K}_{2,2}$ are executed lexicographically in terms of their kernel indices. The points in the same tile are also executed lexicographically. Like RBSOR, our parallelization scheme does not respect all data dependences in the original SOR. However, the convergence is guaranteed but at somewhat slower rates under some inputs [20, 26]. To (more than) offset a drop in the convergence rate, all subdomains in a K-layer sweep can now be executed in parallel. This represents a good tradeoff for data-parallel GPU computing, one of the key findings worthy being emphasised in this paper.

Below we examine how the data dependences in the original SOR are dealt with and how tile sizes are determined.

Consider the five data dependences depicted in Figure 1 for a K-layer forward sweep. There are two cases depending on whether they are intra- or inter-subdomain dependences:

1) **Intra-Subdomain Dependences.** There are two subcases depending whether these are intra- or inter-tile dependences. Intra-tile dependences are satisfied since the points in a tile are executed lexicographically. Inter-tile dependences are satisfied for the four tiles $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ in a subdomain since their corresponding kernels $\mathcal{K}_{1,1}$, $\mathcal{K}_{1,2}$, $\mathcal{K}_{2,1}$ and $\mathcal{K}_{2,2}$ are executed in that order (i.e., lexicographically in terms of tile indices). For the five dependences illustrated for $\mathcal{T}_{2,1}$ in D_{d_1,d_2} in Figure 4(a), the three from $\mathcal{T}_{1,1}$ to $\mathcal{T}_{2,1}$ are satisfied since $\mathcal{T}_{1,1}$ is computed earlier than $\mathcal{T}_{2,1}$.
2) **Inter-Subdomain Dependences.** There are two subcases:
   a) **Top and Right Border Tiles: $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$.** If a point (k, i, j) in such a border tile of a subdomain depends on $(k-1, i, j)$, $(k-1, i+1, j)$ or $(k-1, i, j+1)$ computed in a bottom or left border tile of an adjacent subdomain (in Case 2(b)), the dependence is satisfied since the dependent point must have already been computed. Such is the case for the three dependences from $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} to $\mathcal{T}_{2,1}$ in D_{d_1,d_2} in Figure 4(b). This explains why the slanted hyperplanes in Figure 3 are used in our loop tiling. In particular, the most up-to-date value of a dependent point is always used. In the sequential SSOR, the value used at grid point $(i+1, j)$ of $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} is computed at $(k-1, i+1, j)$ (along the dependence depicted with an unfilled arrow head). In the parallel version, the value is fetched from $(k, i+1, j)$ (along the dashed dependence).
   b) **Bottom and Left Border Tiles: $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$ and $\mathcal{T}_{2,1}$.** As the opposite of Case 2(a), the situation is reversed except that the dependence from $(k-1, i, j)$ to (k, i, j) is always confined to the same subdomain. If a point (k, i, j) in such a border tile of a subdomain D depends on $(k, i-1, j)$ or $(k, i, j-1)$ computed in a top or right border tile of an adjacent subdomain D', then D' has not been executed yet. In this case, the most up-to-date value $(k-1, i-1, j)$ or $(k-1, i, j-1)$ already computed in D is used instead. The existence of such value is guaranteed due to the use of slanted hyperplanes in domain decomposition shown in Figure 2. This is illustrated in Figure 4(c). Point (k, i, j) of $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} requires the value of $(k, i-1, j)$ of $\mathcal{T}_{2,1}$ in D_{d_1,d_2} (along the dependence depicted with an unfilled arrow head), which is computed after (k, i, j). Thus, the most up-to-date value $(k-1, i-1, j)$ of $\mathcal{T}_{1,1}$ in D_{d_1+1,d_2} (along the dashed dependence) is used.

When a kernel is executed, all inter-kernel dependences are satisfied by fetching the dependent data from global memory.

Our parallel MLSSOR algorithm is guaranteed to converge following a similar line of reasoning as in [20, 26]. However, to accelerate the convergence rate, the bottom-left tile $\mathcal{T}_{1,1}$ is made as large as possible. This ensures that the height K in a K-layer sweep is the largest possible to maximize the chances for SOR to be applied. Therefore, if a subdomain has the size $K \times n \times n$, $\mathcal{T}_{1,1}$ is chosen to have $(n-1) \times (n-1)$ at the bottom layer. This ensures that the largest $K = \lfloor n/2 \rfloor$ is used. The sizes of the other three tiles $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ are then determined accordingly by the tile-cutting hyperplanes given in (8) as illustrated in Figure 4.

D. Parallelization for Multiple GPUs

The basic idea is to partition the mesh of a solver across the multiple GPUs and apply MLSSOR to the sub-mesh allocated to a GPU. With one single GPU, kernels $\mathcal{K}_{1,2}$ and $\mathcal{K}_{2,1}$ do not have inter-kernel data dependences and can thus be combined with somewhat improved data reuse and reduced kernel startup overhead. However, this kernel fusion increases the frequency of inter-GPU communication in multiple GPUs, leading to reduced performance. Thus, the two kernels run sequentially in our experiments.

V. Results and Analysis for One GPU

In this section, we present and analyze the performance results and various tradeoffs that need to be made by a compiler for executing MLSSOR and RBSOR on a single-GPU Tesla C1060. We focus more on MLSSOR and touch on RBSOR briefly. For SSOR, once the data operated by a thread are loaded into a buffer in the shared memory, there are no bank conflicts incurred.

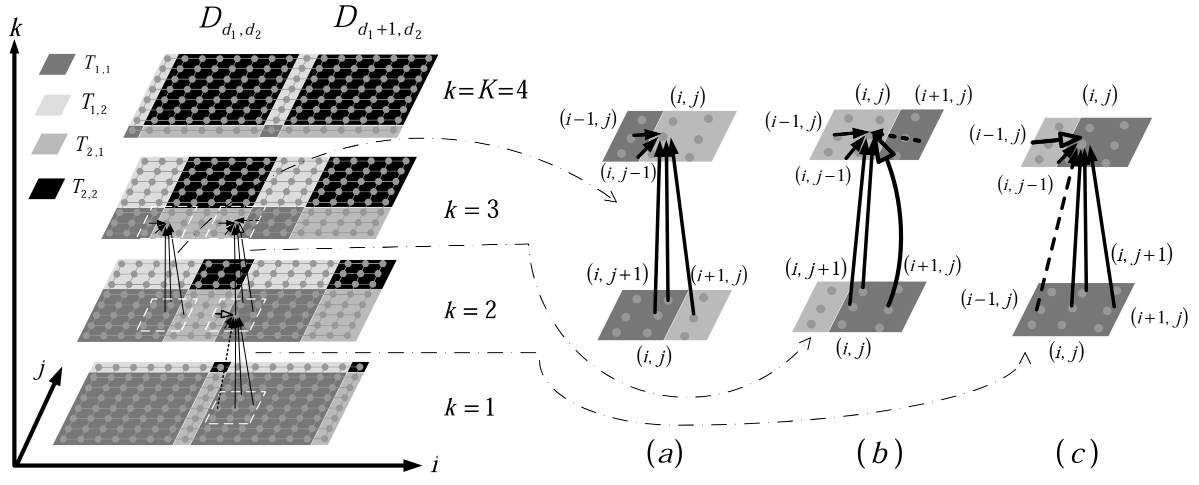

Figure 4. Enforcement of the data dependences illustrated for the two subdomains illustrated in Figures 2 and 3 in a forward sweep. The meanings of the dashed dependence shown in Part (b) (Part(c)) is referred to in Case 2(a) (Case 2(b)) discussed in Section IV-C.

Algorithm		Number of Iterations to Converge		
Input		1	2	3
Tolerance Error		0.001	0.001	0.000001
SSOR		64	67	10214
RBSOR		78	70	10567
MLSSOR	$2 \times 4 \times 4$	112 (139)	105 (120)	12769 (13215)
	$4 \times 8 \times 8$	96 (111)	91 (92)	11463 (12043)
	$8 \times 16 \times 16$	76 (82)	71 (85)	10735 (10997)
	$16 \times 32 \times 32$	71 (81)	64 (70)	10447 (10853)

Table II
CONVERGENCE RATES FOR THREE INPUTS OF SIZE 8192×8192 WITH THE GIVEN TOLERANCE ERRORS SHOWN. FOR MLSSOR, DIFFERENT SUBDOMAIN SIZES LEAD TO DIFFERENT CONVERGENCE RATES. THE RATES INSIDE THE BRACKETS ARE OBTAINED WHEN ONLY THE FORWARD SWEEPING DIRECTION IS USED.

To begin with, Table II compares the convergence rates of SSOR, MLSSOR and RBSOR for three different inputs. The convergence rate of MLSSOR depends on the subdomain size used. Note that MLSSOR is designed to trade off its convergence rate for data parallelism, as demonstrated in this section. MLSSOR can converge more slowly than RBSOR. For all experimental results presented in this section and Section VI, the input data set used in Column 2, i.e., "Input 1" of this table is used.

We are now ready to explain the two reasons for applying our generalized loop tiling to the multi-layer SSOR. First, better convergence is achieved than if a single sweeping direction is used as shown in Table II. Second, inter-kernel data reuse can be exploited in some stream processors even though this is not presently possible for NVIDIA GPUs. As shown in Figure 2, any pair of mirrored subdomains in two adjacent sweeps access the same set of points. We estimate that MLSSOR can achieve about 40% higher performance if such inter-kernel reuse can materialize.

A. MLSSOR

We focus on two different subdomain sizes when $K \times M \times M = 2 \times 4 \times 4$ and $K \times M \times M = 4 \times 8 \times 8$. In each case, K is the largest possible to obtain the fastest convergence as discussed in

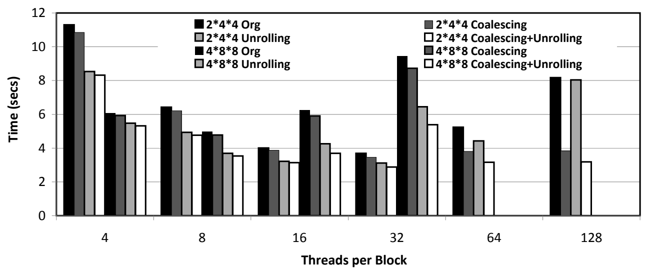

Figure 5. Execution times of MLSSOR. In each case, Org means that neither unrolling nor coalescing is performed.

Section IV-C and the best temporal reuse. For each subdomain size, we consider four different optimizations depending on whether loop unrolling and global memory coalescing are used or not.

We first present the performance results of MLSSOR and then analyze these results. We also discuss various tradeoffs along the way, highlighting the importance of compiler optimizations, performance modeling and performance tuning.

1) Performance: Figure 5 shows the execution times of MLSSOR with respect to varying number of threads per thread block. Some performance bars are missing since in those configurations the 16KB shared memory (cf. Table I) is not big enough to hold the data used by all the threads in a single thread block. We observe that the performance of MLSSOR is sensitive to subdomain size. The effect of any optimization (or combination) on performance is non-linear due to complex interactions among various GPU architectural constraints (cf. Table I).

2) Resource Usage and Performance Estimates: We analyze the results of Figure 5 by making use of the resource usage information for kernel $\mathcal{K}_{1,1}$ from Table III. Memory coalescing does not appear in the table since it is immaterial to the statistics collected. A similar trend is observed if one of the other three kernels is used. For subdomain sizes $2 \times 4 \times 4$ and $4 \times 8 \times 8$, the points/thread values for $\mathcal{K}_{1,1}$ are 3×3 and 7×7, respectively, as shown in Figure 3. For each configuration identified by the

Subdomain Size	Points/ Thread	Threads/ Block	Shared Memory/ Thread Block (bytes)	Registers/Thread		W_{TB}	B_{SM}	#Active Threads	Performance (GFLOPS)		Bandwidth (GB/s)	
				No Unrolling	Unrolling				No Unrolling	Unrolling	No Unrolling	Unrolling
$2 \times 4 \times 4$	3×3	4	404	16	13	0.125	40	160	99.7	82.2	46.9	78.7
		8	724			0.25	22	176				
		16	1364			0.5	12	192				
		32	2644			1	6	192				
		64	5204			2	3	192				
		128	10324			4	1	128				
$4 \times 8 \times 8$	7×7	4	1260	17	59	0.125	13	52	108.1	93.9	26.9	46.6
		8	2412			0.25	6	48				
		16	4716			0.5	3	48				
		32	9324			1	1	32				
		64	Out of Shared Memory									
		128										

Table III
RESOURCE USAGE OF MLSSOR (INSENSITIVE TO COALESCING) FOR KERNEL $\mathcal{K}_{1,1}$ UNDER THE CONFIGURATIONS DEFINED BY THE FIRST THREE COLUMNS (AND ALSO ILLUSTRATED IN FIGURE 5).

first three columns together, a number of statistical data are listed. Columns 4 – 6 are self-explanatory. By compiling CUDA code with the `-cubin` flag, we could get some understanding about on-chip memory usage. In Column 7, W_{TB} is the number of warps in a thread block, which is determined by dividing the number of threads in a thread block by 32. In Column 8, B_{SM} is the number of thread blocks assigned to each SM. It is usually determined by shared memory and register usage (Columns 4 – 6). As indicated in Table I, C1060 has 16KB shared memory and 16,384 registers per SM. Consider the configuration when the subdomain size is $2 \times 4 \times 4$ and threads/block is 8. Every thread block needs 724B shared memory. So the maximum number of simultaneously active blocks in one SM is $16KB \div 724B = 22$ (Column 8). The number of registers required by 22 threads block is $16 \text{ registers/thread} \times 8 \text{ threads/block} \times 22 \text{ active blocks} = 2816$. Since this number is less than 16384, 22 blocks can be assigned to the same SM. Otherwise, B_{SM} is decided by register usage. The bottleneck then shifts from shared memory to registers. In a special case, although there are enough registers and shared memory to execute more blocks, the number of active threads may exceed the maximum value 1024 available per SM (cf. Table I). B_{SM} has to change to satisfy this constraint. For this configuration, the largest number of active threads is 176 only (Column 9).

Finally, let us look at the last four columns in Table III, which give performance and bandwidth estimates for kernel $\mathcal{K}_{1,1}$. Tesla C1060 is capable of issuing $240\text{SPs} \times 1.3\text{GHz} = 312$ billion operations per second. When all the SPs are fully occupied, which is achievable in an application that has many threads, does not have many synchronization operations, and does not stress memory bandwidth. In this situation, for example, if 10% of a program instruction mix are fused FMAD and FMUL which can be done each GPU cycle, then its single-precision performance can be at most $3 \times 10\%\text{FP} \times 312 = 93.6\text{GFLOPS}$. We can obtain kernel assemble instructions through the `-ptx` flag. In Columns 10 and 11, the GFLOP estimates for $\mathcal{K}_{1,1}$ are given for both unrolled and non-unrolled cases, which will be further discussed in Section V-A3.

Another potential bottleneck is global memory bandwidth. If 5% of code are loads from off-chip memory, required bandwidth is $240\text{SPs} \times 5\%\text{instructions} \times 4\text{B/instruction} \times 1.3\text{GHz} = 62.4\text{GB/s}$. This value is estimated average bandwidth for running threads. It is possible that all threads simultaneously load data, thus the latency of accessing to global memory still exists. However, if this value without any global memory optimization is more than 102 GB/s, which is Tesla C1060's off-chip bandwidth, a lot of time will be spent on waiting for data transport and the bandwidth is likely to be bottleneck. In the last two columns, the bandwidth estimates for $\mathcal{K}_{1,1}$ are given for unrolled and non-unrolled cases.

3) Effects of Unrolling and Coalescing On performance: Full loop unrolling often achieves the best performance for MLSSOR and is thus applied to obtain the results given in Figure 5. Unrolling improves data parallelism by removing branch instructions, and consequently, reduces significantly the dynamic number of instructions executed. With full unrolling, the MLSSOR performance always improves as shown in Figure 5 although the GFLOP estimates have dropped as listed in Table III. In addition, unrolling also affects register usage. In the case of $2 \times 4 \times 4$, the number of registers per thread has dropped from 16 to 13. For $4 \times 8 \times 8$, the register requirement increases noticeably from 17 to 59. Although unrolling usually increases register pressure, the increase is small relative to the total number of registers available per SM (at least for MLSSOR). Thus, full loop unrolling accelerates the MLSSOR performance by reducing the dynamic instruction count executed.

Two strategies for reducing the negative impact of bandwidth on performance are to improve data reuse and reduce global memory access. The bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction. The memory system may be able to combine these into a single memory accessing request. Even if the average required bandwidth in our experiments is less than 102 GB/s and thus not the key bottleneck, the impact of coalescing on performance is still noticeable. As shown Figure 5, coalescing always improves performance. In particular, when $B_{SM} = 1$ in Table III, the performance benefit of coalescing is maximized. In this case, when threads in the unique block assigned to an SM stall on a load instruction, there exists no other blocks that can be scheduled by the SM to overlap computation and communication.

4) Correlating Configurations' Relative Performances: We find from Figure 5 that it is difficult to establish a certain relation between an MLSSOR configuration and its execution time. We

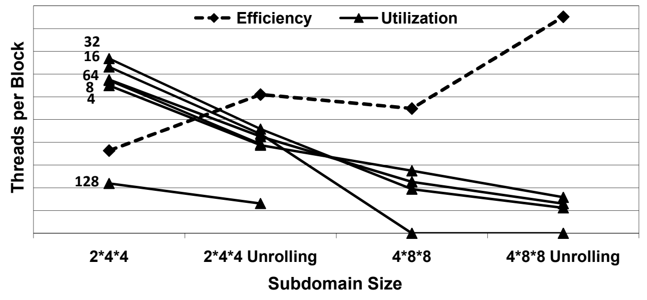

Figure 6. Efficiency and Utilization of MLSSOR (insensitive to coalescing) of kernel $\mathcal{K}_{1,1}$ for all the configurations given in Figure 5.

use the two performance metrics from [16] to provide some rough estimates for the relative performance results of certain configurations. Both are meaningful only if global memory bandwidth is not the performance bottleneck. The Efficiency metric indicates the overall efficiency of a configuration in terms of the total number of instructions that must be executed before the kernel finishes:

$$\mathsf{Efficiency} = \frac{1}{Instr \times Threads} \tag{10}$$

where $Instr$ derived from the PTX code of a kernel estimates the number of dynamic instructions executed per thread and $Threads$ is the number of threads created by the kernel. In this work, this metric also reflects well the impact of data reuse on performance. When the tile size increases, $Threads$ decreases sharply. A thread will have more work to do. In addition, the convergence will be accelerated as shown in Table II, resulting in a better efficiency.

The Utilization metric is about the utilization of the compute resources on a GPU by considering how often a warp may wait and the amount of work available (from other warps) when it does:

$$\mathsf{Utilization} = \frac{Instr}{Regions}\left[\frac{W_{TB}-1}{2} + (B_{SM}-1) \times W_{TB}\right] \tag{11}$$

where $Regions$ is the number of dynamic instruction intervals delimited by *blocking instructions* or the start/end of the kernel. Long latency instructions, such as texture memory operations and synchronization instructions, are considered as blocking instructions. $\frac{Instr}{Regions}$ gives the average number of non-blocking instructions per interval. The quantity within the brackets indicates the number of independent warps in an SM. The first term is the number of other currently executing warps in the same thread block. Dividing by two is for computing average possibility, half warps still need to execute. The second item is the number of warps in other thread blocks assigned to the same SM. When the degree of parallelism is low, the value of Utilization is small.

Figure 6 plots Efficiency and Utilization of $\mathcal{K}_{1,1}$ for all configurations in Figure 5. A solid curve represents Utilization as a function of different configurations for a fixed threads/block as marked. The single dashed curve represents Efficiency for all possible values of threads/block since their efficiency curves are identical.

It should be pointed out that these two metrics are not suitable for the configuration with only 4 threads per block. The full computing capability of an SM consisting of eight SPs is not fully utilized. Otherwise, there are good correlations between the Efficiency and Utilization metrics in Figure 6 and the actual performance results in Figure 5. Below we use these two metrics to analyze the effects of thread granularity and loop unrolling on performance.

First of all, for both unrolled and non-unrolled code, Utilization drops when thread granularity increases. This implies that the number of active and ready threads is cut down because larger threads consume more resources. In limited resource situations, shared memory and register usage affect the throughput of an SM. However, since an increase in Efficiency counteracts a decrease in the throughput caused by reduced Utilization, MLSSOR is not always slower when $4 \times 8 \times 8$ is used, since larger threads, i.e., subdomains lead to faster convergence rates as shown in Table II.

Next, for the same subdomain size with and without unrolling, the improved Efficiency due to unrolling is entirely attributed to a reduction in $Instr$ since $Threads$ remains unchanged. Moreover, unrolling does not alter the number of memory accesses. So $Regions$ should not change remarkably. According to Table III, unrolling usually does not reduce resource usage. As a result, Utilization worsens as $instr$ decreases. However, the gain from improved Efficiency seems to more than offset the loss caused by worsened Utilization here. So unrolling is always beneficial for MLSSOR.

Furthermore, looking at the combined effects of thread granularity and unrolling on performance, MLSSOR has better Efficiency and Utilization under "$2\times4\times4$ Unrolling" than "$4\times8\times8$" (without unrolling). This means that a decrease in $Instr$ due to unrolling affects the two metrics more than an increase in thread granularity, i.e., subdomain size. Consequently, "$2 \times 4 \times 4$ Unrolling" results in better Efficiency than "$4 \times 8 \times 8$", which has better Efficiency than "$2 \times 4 \times 4$". This analysis correlates well with the results given in Figure 5, where MLSSOR runs faster under "$2 \times 4 \times 4$ Unrolling" except for the pathological "4 threads/block" case.

Finally, the weights of Efficiency and Utilization are unpredictable, and the optimal configuration may need to balance both metrics. This is consistent with the observation made in [16], highlighting the importance of using a tuning tool for efficient solution space exploration.

5) Shared Memory Reduction: From Table III, we see that the bottleneck of MLSSOR is shared memory. Our case study indicates the importance for future GPU architectures to aggressively exploit both intra- and inter-kernel data reuse in scientific and engineering applications to boost the performance of DOACR loops. Given the 16KB shared memory, the scarce resource must be effectively utilized with some shared memory reduction technique. One solution is to undo the merge of the tiles in $\{T_{i,j} \mid 0 \leqslant i,j < m\}$ into $\mathcal{T}_{1,1}$ so as to execute the smaller tiles in separate kernels. But using smaller tiles leads to smaller tile height K as discussed in Section IV-C, resulting in slower convergence and poorer data reuse. To avoid these problems and to facilitate memory coalescing, a different solution is used. When a row of points in a tile, i.e., $\mathcal{T}_{1,1}$, $\mathcal{T}_{1,2}$, $\mathcal{T}_{2,1}$ or $\mathcal{T}_{2,2}$ are computed, only this row and its two adjacent rows are kept in the shared memory. This solution also sacrifices some temporal reuse but allows memory coalescing to be realized more effectively.

With the shared memory reduction technique being applied, Figures 7 and 8 are now given as the analogues of Figures 5 and 6, respectively. Some observations are in order. First, the performance of MLSSOR drops slightly with shared memory reduction in most but not all configurations. Second, MLSSOR did not compile

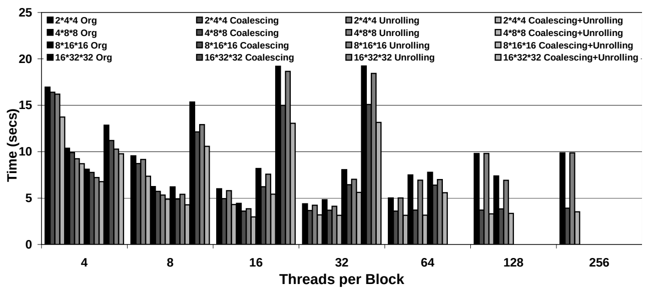

Figure 7. Execution times of MLSSOR with shared memory reduction.

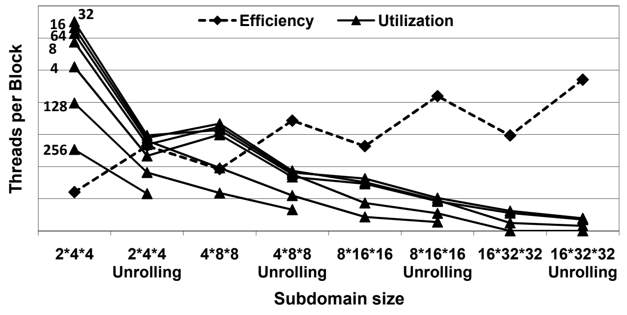

Figure 8. Efficiency and Utilization of MLSSOR (insensitive to coalescing) of kernel $\mathcal{K}_{1,1}$ with shared memory reduction.

before for $4\times8\times8$ when the threads/block is 64 or 128 due to lack of shared memory but compiles now. Third, two larger subdomain sizes $8 \times 16 \times 16$ and $16 \times 32 \times 32$ are now included and can compile except for a few large threads/block values. Finally, an analogue of Table III is omitted due to space limit. For comparison purposes, the new GFLOPS and bandwidth values corresponding to the "$2\times4\times4$" and "$4\times8\times8$" rows in Table III are "120.9 70.5 58.0 123.9" and "124.5 101.3 62.1 101.3", respectively. With memory reduction, the required bandwidths are higher but the GFLOPS values do not change as much at a similar magnitude.

Again there are good correlations between Figure 7 and Figure 8. In particular, there is an important point worth being restated. Although "$4 \times 8 \times 8$" results in lower Efficiency than "$2 \times 4 \times 4$ Unrolling", MLSSOR is a better performer under "$4\times8\times8$" when the threads/block is 8 and 16 due to higher Utilization. When the threads/block increases to 32, the effect of a decrease in Efficiency on performance is larger than that of an increase in Utilization, MLSSOR is slower under "$4 \times 8 \times 8$". However, the situation is different in 128 threads/block. MLSSOR performs better under "$4 \times 8 \times 8$" even though its Efficiency and Utilization values are both lower. As mentioned earlier, the required bandwidth of "$2\times4\times4$ Unrolling" is 123.9 GB/s, which is beyond the maximum 102 GB/s available in Tesla C1060, while the required bandwidth of "$4 \times 8 \times 8$" is 101.3GB/s. Without coalescing, the GPU may stall on waiting for accessing to global memory. As a result, the execution time under "$2 \times 4 \times 4$ Unrolling" is prolonged. With coalescing, however, the overall memory time is reduced. Thus, MLSSOR performs slightly better under "$2 \times 4 \times 4$ Unrolling + Coalescing" than "$4 \times 8 \times 8$".

With shared memory reduction, the number of active threads can

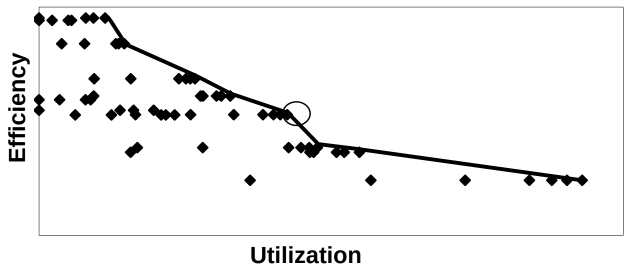

Figure 9. Searching for optimal solutions by performance metrics (illustrated using Efficiency and Utilization for kernel $\mathcal{K}_{1,1}$). The best configuration is highlighted by a circle.

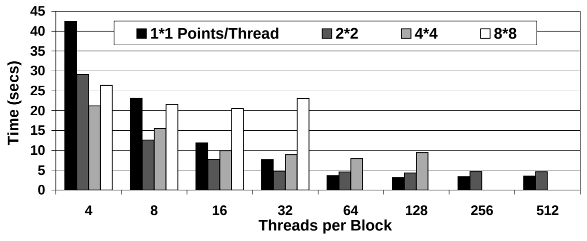

Figure 10. Execution times of RBSOR.

sometimes increase by nearly 87.5%. Thus, finer data partitioning reduces an application's demand for resources and increases its degree of parallelism. However, it may affect negatively other architectural constraints, i.e. by saturating the bandwidth. Again the optimal configuration requires a balanced tradeoff to be made.

6) Summary of Performance Metrics: Figure 9 plots the two metric values for all configurations given in Figures 5 and 7. The maximum metric value along each axis has been normalized to one for comparison purposes. In general, the best performance should come from configurations with both high Efficiency and Utilization although their weights are difficult to valuate [16]. Thus, one desires configurations located towards the upper right corner of the graph. The points connected by the line have higher opportunity to get better performance than others. The circled point for "$4 \times 8 \times 8$ Coalescing+Unrolling" using 16 threads per block with shared memory reduction is the best performer.

B. RBSOR

We have implemented RBSOR taken from the Java Grande benchmark suite for CUDA following [25]. The performance results are displayed in Figure 10. RBSOR exhibits the same performance with 1×1 points/thread when threads/block ranges from 64 to 512. By examining resource usage, we find that the bottleneck in this case is neither shared memory nor registers. With 64 or more threads per block, the number of threads to be launched per SM exceeds 1024 (cf. Table I). Thus, the bottleneck is instruction issue. Therefore, fine-grained parallelism often gives rise to good performance on GPU architectures. But the overall performance can be constrained by architectural constraints, such as the number of active threads allowed, if the data reuse is not adequately exploited.

For RBSOR, the Efficiency and Utilization metrics do not appear to be sufficient in explaining its performance results. We have made an attempt to understand its performance trend through experimentation, analysis and consulting [25]. RBSOR seems to run at its full speed at 1×1 points/thread with 128 threads/block. Both RBSOR and MLSSOR are compared in detail below.

VI. Results and Analysis for Multiple GPUs

We compare MLSSOR and RBSOR on a Tesla S1070 computing system consisting of four GPUs. We evaluate MLSSOR and RBSOR using their configurations giving rise to the best single-GPU performances. These may not be the absolute best for a multi-GPU setting but seem to be a good choice for stencil-based computations. MLSSOR uses a subdomain of size $4 \times 8 \times 8$ with 16 threads/block with shared memory reduction while RBSOR's configuration is 1×1 points/thread with 128 threads/block.

The host is an Intel Xeon Quad-core CPU running at 2.66GHz. The mesh of an SOR solver is distributed block-wise along one dimension to the four GPUs. We need to use a series of CPU threads to schedule and manage the execution of the sub-meshes allocated to the GPUs. In our experiments, four CPU threads are created to run on four CPUs concurrently. Each CPU thread is associated with an individual GPU. It is responsible for distributing the required data in the sub-mesh to the device memory of its associated GPU, scheduling kernel execution on it, and communicating the boundary data of sub-meshes with the other GPUs indirectly via the host.

Figure 11 shows the speedups of MLSSOR over RBSOR. Overall, MLSSOR performs better with increasingly larger problem size and more GPUs. However, the performance increases are not linear. We analyze this phenomenon by separating the inter-GPU communication cost from the computation cost during a program execution. All device-to-device copies are asynchronous. The associated idle times are not stable in different runs of the same program. Thus, the inter-GPU communication time of a program is measured as an average of 10 program runs. Figure 13 shows the inter-GPU communication overhead increases of RBSOR over MLSSOR as the problem size increases on more and more GPUs. Figure 12 replots Figure 11 with the inter-GPU communication costs being annihilated. Now, the computation speedup of MLSSOR over RBSOR increases more smoothly than before as the problem size increases across the multiple GPUs.

Given a problem size, MLSSOR and RBSOR incur about the same amount of inter-GPU communication. The difference lies in the frequency of communication. For MLSSOR, the inter-GPU communication occurs when $\mathcal{K}_{1,2}$ and $\mathcal{K}_{2,2}$ run to completion. Note that if $\mathcal{K}_{1,2}$ and $\mathcal{K}_{2,1}$ were merged (as discussed Section IV-D), the inter-GPU communication would occur when every kernel completes, causing a 50% increase in communication frequency. However, RBSOR communicates four times as many as MLSSOR. As thread granularity increases, the frequency of communication incurred by MLSSOR decreases. However, due to the idle times elapsed in inter-GPU communication, performance fluctuations are expected across the problem sizes.

VII. Related Work

We supplement a brief review of related work earlier by discussing some current work for GPU architectures on languages, compiler optimizations and performance tuning.

CUDA is a popular programming model for the NVIDIA GPUs used by C/C++ programmers. The provided abstractions may sometimes limit the programmability of GPUs. One research direction is to make CUDA kernels more accessible in high-level programming languages [10]. Developing programmer-friendly interfaces for accelerating Java programs with CUDA is also an interesting topic [25]. Furthermore, systematic compiler optimization techniques are useful in generating highly parallelized CUDA code automatically. The latest version of the PGI 9.0 release from the Portland Group includes PGF95 and PGCC accelerator compilers, which are supported on all Intel and AMD x64 processor-based systems with CUDA-enabled NVIDIA GPUs [5]. Recently, Lee et al. [10] introduce an automatic compiler framework to generate CUDA code from OpenMP programs. Baskaran, et al. [2] explore the use of affine loop transformations for GPU parallelization.

Another important research is to develop cost models and tuning tools for estimating and understanding the performances resulting from different optimizations. Due to complex interactions among the GPU architectural constraints, it seems to be difficult to find near-optimal configurations by searching the solution space blindly. Ryoo et al. [16] discuss some useful performance metrics and optimization principles for GPU architectures [15], but they did not concern themselves specifically about DOACR loops. Hong and Kim [6] present an analytical model for estimating the execution time of a program running on NVIDIA GPUs.

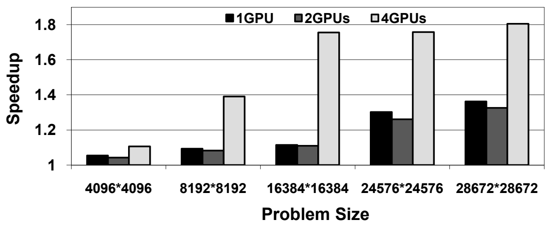

Figure 11. Speedups of MLSSOR over RBSOR for multi-GPUs.

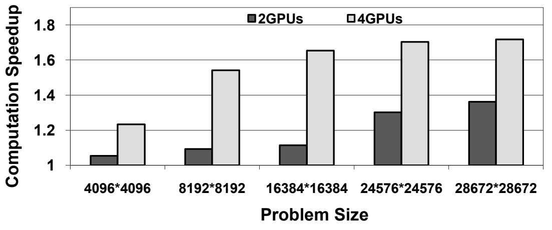

Figure 12. Computation speedups of MLSSOR over RBSOR for multi-GPUs (with zero communication overhead assumed).

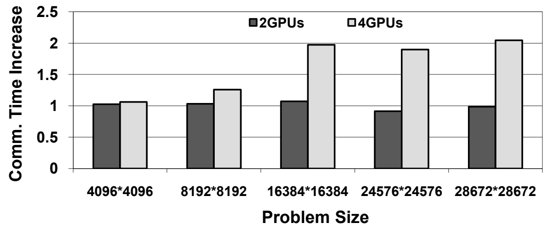

Figure 13. Inter-GPU communication overhead increases of RBSOR over MLSSOR for multi-GPUs.

VIII. Conclusion

DOACR loops are difficult to run efficiently on GPGPUs since their cross-iteration dependences pose a major obstacle to the exploitation of the fine-grained parallelism in these loops. We have described our experience on parallelizing the SSOR method in order to establish optimization principles and strategies for accelerating the performance of DOACR loops on GPGPUs. We have presented a new parallel SSOR solver, which is developed based on a non-dependence-preserving parallelization scheme, and demonstrated its performance advantages over red-black SOR. Our experimental results, validated by detailed analysis, show that many DOACR loops may be potentially accelerated if different algorithms that are more amenable to GPU computing can be developed. The importance of synergy between domain experts, compiler optimizations and performance tuning is highlighted in maximizing application performance.

IX. Acknowledgement

This research is supported by an Australian Research Council Grant (DP0881330).

References

[1] L. Adams and J. Ortega. A multi-color SOR method for parallel computation. In *1982 International Conference on Parallel Processing (ICPP'82)*, pages 53–56, 1982.

[2] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[3] Stephen H. Brill and George F. Pinder. A block red-black SOR method for a two-dimensional parabolic equation using Hermite collocation. *The Mathematics of Finite Elements and Applications*, 1997.

[4] Georgios Goumas, Nikolaos Drosinos, Vasileios Karakasis, and Nectarios Koziris. Coarse-grain parallel execution for 2-dimensional PDE problems. *International Parallel and Distributed Processing Symposium*, 0:381, 2007.

[5] The Portland Group. PGI accelerate compiler, 2009.

[6] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[7] Q. Huang, J. Xue, and X. Vera. Code tiling for improving the cache performance of PDE solvers. In *2003 International Conference on Parallel Processing (ICPP'03)*, pages 615 – 625, 2003.

[8] Changhao Jiang and Marc Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, 2005.

[9] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, page 282, 2002.

[10] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, 2009.

[11] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.

[12] Rami G. Melhem and K. V. S. Ramarao. Multicolor reordering of sparse matrices resulting from irregular grids. *ACM Trans. Math. Softw.*, 14(2):117–138, 1988.

[13] Paulius Micikevicius. 3d finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 2009.

[14] NVIDIA. NVIDIA CUDA programming guide 2.2, 2009.

[15] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[16] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

[17] V. K. Saul'yev. *Integration of Equations of Parabolic Type Equation by the Method of Net*. Pergamon Press, 1964.

[18] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse tiling for stationary iterative methods. *Int. J. High Perform. Comput. Appl.*, 18(1):95–113, 2004.

[19] P. Tang and J. Xue. Generating efficient tiled code for distributed memory machines. *Parallel Computing*, 26(11):1369–1410, 2000.

[20] Rohallah Tavakoli and Parviz Davami. New stable group explicit finite difference method for solution of diffusion equation. *Applied Mathematics and Computation*, 181(2):1379–1386, 2006.

[21] Dan Wallin, Henrik Löf, Erik Hagersten, and Sverker Holmgren. Multigrid and Gauss-Seidel smoothers revisited: parallelization on chip multiprocessors. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 145–155, 2006.

[22] Dexuan Xie. A new block parallel SOR method and its analysis. *SIAM J. Sci. Comput.*, 27(5):1513–1533, 2006.

[23] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[24] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

[25] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par*, pages 887–899, 2009.

[26] Hongkai Zhao. A fast sweeping method for Eikonal equations. *Math. Comp.*, 74:603–627, 2005.