The Pennsylvania State University
The Graduate School

ADAPTING SPARSE TRIANGULAR SOLUTION TO GPU

A Thesis in
Computer Science and Engineering
by
Bradley Suchoski

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2012

The thesis of Bradley Suchoski was reviewed and approved* by the following:

Padma Raghavan
Distinguished Professor of Computer Science and Engineering
Thesis Advisor

Kamesh Madduri
Assistant Professor of Computer Science and Engineering

Lee D. Coraor
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Trends in recent years have shown that high performance computing systems are increasingly including GPUs to increase floating point calculation throughput for scientific applications. Many such applications, such as partial differential equation solvers, involve solving linear systems of equations in which the systems being solved are sparse. These applications often use preconditioned iterative solvers, such as preconditioned conjugate gradients (PCG), in which sparse triangular solution is currently a major bottleneck despite recent advancements. In this paper we show that graph coloring can be effective in exposing large amounts of fine-grained parallelism in the sparse triangular solve kernel. A simple performance model is developed to predict the effects of the parallelism exposed by our algorithm which was evaluated on a Nvidia Tesla M2090 GPU compared to the most recent Nvidia method. Tests show that our coloring algorithm drastically increases the parallelism exposed compared to the Nvidia algorithm, resulting in an average speedup of 5.41 relative to Nvidia's code. Our results also indicate that our algorithm will be more scalable than the Nvidia algorithm as the number of cores per chip continues to increase in the future, or if applied to more than one GPU.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to express my sincere gratitude to my thesis advisor Dr. Padma Raghavan for all of her inspiration, motivation, support, and guidance during my time as a Master's student and during this thesis. It was only through her immense knowledge and enthusiasm that I even decided to research this topic, and that this thesis was even possible. I can't imagine having a better advisor, and I truly don't believe I could have done this without her.

I would also like to thank my other committee member, Dr. Kamesh Madduri, for his insightful comments and questions.

I extend my thanks to my lab and class mates, Caleb Severn and Manu Shantharam, for helping with both research and writing. We spend many sleepless nights working to meet deadlines, and although stressful at the time, looking back it was actually somewhat fun.

Finally, I would like to thank my family. To my parents Ed Suchoski and Sue Suchoski for always believing in and supporting me throughout my entire life and studies. Also to my girlfriend, Tina Rementer, for putting up with an uncountable number of endless nights spent waiting to simply spend time with me, only to end up falling asleep before I finished work for the night.

Chapter 1

Introduction

Trends in high performance computing (HPC) indicate a strong presence of hybrid computers, comprising traditional general-purpose processors and highly parallel accelerators such as graphic processing units (GPUs). For example, three out of the top five supercomputers currently employ hybrid nodes [1]. Consequently, there is increasing need to adapt to hybrid architectures the large-scale scientific applications that are typical of the workloads on such high performance computing systems.

Many scientific applications solve an underlying sparse linear system $Ax = b$, through a preconditioned iterative method, such as conjugate gradients with incomplete Cholesky preconditioners (PCG). The computational time of such methods is mainly dominated by the time for: (i) sparse matrix vector multiplication, and (ii) sparse triangular system solution for preconditioner application. Substantial work has been done to speed up sparse matrix vector multiplication on GPUs [2, 3, 4]. More recently, Naumov [5] proposed a dependency graph based "level set" technique to speed up a sparse triangular solution kernel on GPU architectures. However, as shown in Figure 1.1 we observe that it still remains the dominant cost at 90% of the cost per iteration of PCG on average for matrices listed in Table 4.1 (for level set scheme in Nvidia's cuSPARSE and cuBLAS library routines [6]). In this paper, we consider the acceleration of sparse triangular solution on GPUs in the context of iterative methods such as PCG.

This work was supported in part through instrumentation funded by the National Science Foundation through grant OCI0821527

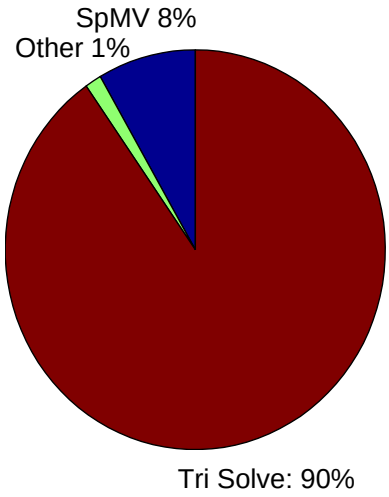


Figure 1.1 – Pie chart showing the percentage of time taken by different operations in one iteration of PCG using Naumov's level set (LS) algorithm.

Several researchers have considered the parallelization of sparse triangular solution on earlier generation multiple-instruction-multiple-data and single-instruction-multiple-data multiprocessors through coloring and special forms of inversion [7, 8, 9, 10]. We expect that the coloring approach first proposed by Schreiber and Tang [7] will be particularly effective in extracting large levels of fine-grained parallelism. Such high degrees of parallelism are latent in the structure of the sparse coefficient matrix and dependent on a symmetric permutation of its rows and columns. The contributions of this paper center around the testing of this conjecture, and developing performance models that capture the effects of enhanced parallelism. It also includes results on observed speed-ups of our method relative to the level set method by proposed by Naumov [5] and implemented in Nvidia's CuSparse library.

Much of the text from this chapter as well as the remaining chapters was written by the author, Raghavan, Severn, and Shantharam in concurrence with [11] and was taken verbatim from that work. The remainder of this paper is organized as follows. Chapter 2 presents a brief overview of the GPU architecture and sparse triangular solution, followed by an overview of earlier related research. Chapter 3 discusses how coloring can reduce dependencies during sparse triangular solution and reveal large groups of disjoint and independent operations that can be mapped

well to execution on GPUs. It also develops performance models that predict execution time reductions from the enhanced fine-grained parallelism. Chapter 4 contains an empirical evaluation, including a discussion of how the performance models match the observed results. Chapter 5 contains brief concluding remarks and directions for further research.

Chapter 2

Background and Related Research

This Chapter provides an overview of Nvidia's Fermi architechture, Nvidia's latest GPU architecture, and the sparse triangular solution kernel. Following that is a brief discussion of recent research related to adapting sparse triangular solution to GPU.

2.1 GPU Architecture

The threads on a GPU are organized into a single centralized hierarchy. At the lowest level, groups of 32 threads referred to as a warp share a single instruction pipeline, and so all threads in a warp run in lock-step. If the control flow branches within a warp, the operations are serialized until execution returns from the branch. In the Fermi architecture, a maximum of 48 of these warps (1536 threads) form a block. Threads in a block have access to up to 48KB of shared memory and can cooperate with one another using barrier synchronization. A centralized server dispatches blocks of threads to 16 streaming multiprocessors (SMP). A SMP executes a single warp of 32 threads concurrently. No guarantees are made by the programming model as to the total number of blocks scheduled in parallel, but the block size must be explicitly configured for a kernel. In Nvidia's Fermi architecture, up to 16 warps (512 threads) execute concurrently, and up to 24K threads can be scheduled in parallel. The key to performance in many cases is keeping many concurrent threads busy. Figure 2.1 shows the Fermi architechture with 16 SMPs (in green) each containing 32 cores.

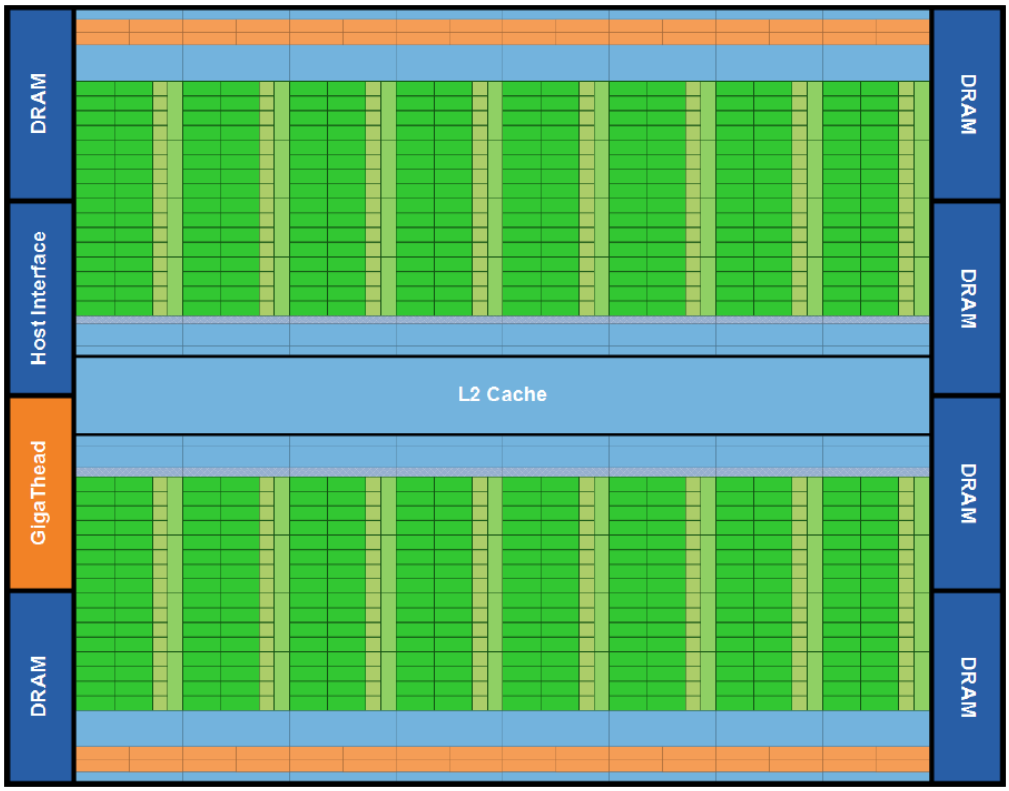


Figure 2.1 – Image taken from the Fermi whitepaper [12] shows GPU architecture with 16 SMPs.

2.2 Memory characteristics

With so many threads working, bandwidth to GPU memory is a major concern. The memory pipeline is organized around a streaming cache that works by buffering requests for memory, rather than the content of that memory. The interface to the DRAM is 384 bits wide, so bandwidth is maximized when many buffered requests for individual memory locations can be coalesced and issued as a single request for an entire contiguous 384 bit (48 byte) block of memory. However, the Fermi architecture introduced a gradual coalescing model, where the hardware now detects the number of memory blocks accessed by a warp and issues the corresponding number of block reads. Because of this more gradual degradation of memory bandwidth, aggregate patterns of memory access, such as total number of words requested, have become more representative of performance impacts.

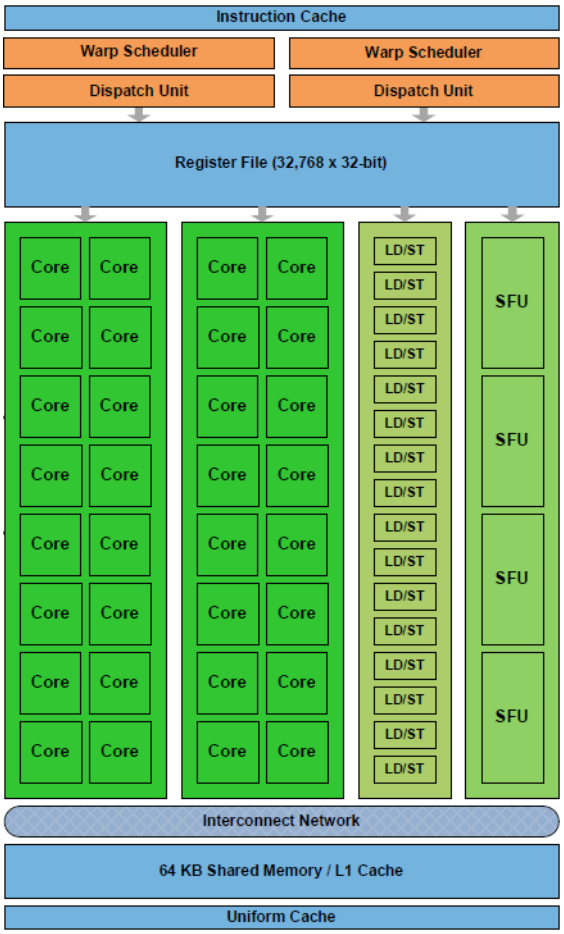


Figure 2.2 – Image taken from the Fermi whitepaper [12] shows streaming multiprocessor (SMP) architecture.

The Fermi architechture is also the first generation of GPU's to introduce a true cache hierarchy. In previous generations of GPU's, each SMP had a 16 KB block of shared memory that could accessed by any thread in a block. Often times this shared memory was utilized as a user managed cache to buffer multiple requests to the same location in DRAM by first explicitly loading data into shared memory, performing work on the copy, then writing back to DRAM. As shown in Figure 2.2 however, each SMP in the Fermi architechture shares a 64 KB block of memory that can be configured as either 48 KB of shared memory plus 16 KB of L1 cache, or 16 KB of shared memory plus 48 KB of L1 cache. Figure 2.1 shows that all SMP then share a L2 cache. This architecture automatically provides all of the

performance benefits of traditional cache hierarchy with minimum extra effort on the part of the programmer.

2.3 Parallel Sparse Triangular Solution

The triangular solution kernel solves a linear system of the form $\mathbf{L}*\mathbf{x} = \mathbf{b}$ where $\mathbf{L}$ is a lower triangular matrix, or $\mathbf{U}*\mathbf{x} = \mathbf{b}$ where $\mathbf{U}$ is upper triangular, and $\mathbf{b}$ and $\mathbf{x}$ are known and unknown vectors respectively. We will refer to $\mathbf{b}$ as the right hand side vector and $\mathbf{x}$ as the solution vector. In the case where $\mathbf{L}$ or $\mathbf{U}$ is dense, the triangular solve process seems to be inherently sequential. To solve for unknown $\mathbf{x}_i$ we must first solve for every previous unknown on which $\mathbf{x}_i$ depends on, that is $\{\mathbf{x}_j : j < i\}$ during forward substitution and $\{\mathbf{x}_j : j > i\}$ during backward substitution. The only opportunity for parallelization is during the dot product when removing these dependencies.

A naive implementation of sparse triangular solve will suffer from the same problems as its dense counterpart. In the sparse case though, these problems will be exacerbated by the fact that the number of nonzeros per row in matrices arising from finite difference and finite element meshes will typically be much lower than the available hardware parallel capacity, thus limiting the amount of parallelism available in the dot products and therefore the hardware utilization. Upon closer inspection however, we will find that there is much parallelism available in these problems. For example, Figure 2.3 shows a lower triangular system, and the graph on the right represents the dependencies among the solution elements during the triangular solve process using forward substitution. In the example, the solution element $\mathbf{x}_5$ depends on the already solved values $\mathbf{x}_2$ and $\mathbf{x}_4$, which in turn may depend on other elements. A similar graph can be created for upper triangular matrices using backward substitution. From this graph we can clearly see that the solution of an element does not always depend on all of its predecessors, for example $\mathbf{x}_7$ does not depend on $\mathbf{x}_2$, and so managing these dependencies is key for effective parallelization.

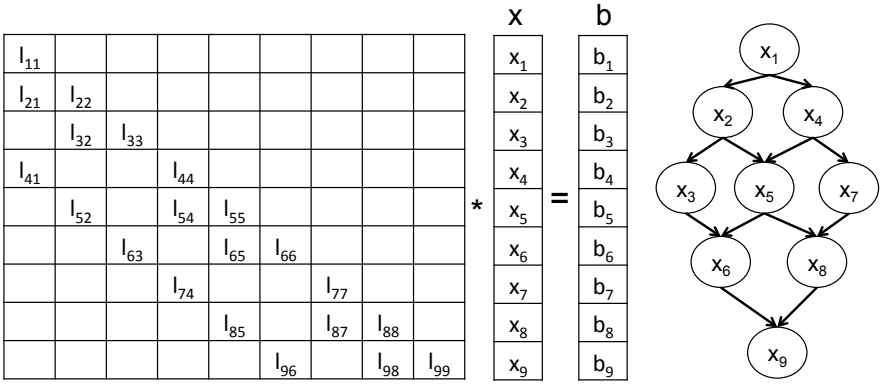


Figure 2.3 – A typical sparse triangular system. The graph on the right represents the dependence among the solution elements.

2.4 Related Research

In regard to the problem of adapting sparse triangular solution to GPUs, the only work we know of is a recent algorithm developed by Naumov [5]. It is a two-phase algorithm, where the first phase involves analysis that performs a modified breadth first search (BFS) of the dependency graph based on the matrix sparsity pattern, starting with $\mathbf{x}_1$ in forward substitution and $\mathbf{x}_N$ in backward substitution. At each step of the BFS, the algorithm visits all neighboring nodes that have 0 unvisited dependencies and groups them together into a level. Within each level, because the elements have 0 unvisited dependencies when they are visited they cannot have any dependencies among one another. It is in this way that the algorithm finds independent rows that can be solved in parallel and groups them into levels. As an example, the graph in Figure 2.3 would contain the 5 levels $L_1 = \{\mathbf{x}_1\}$, $L_2 = \{\mathbf{x}_2, \mathbf{x}_4\}$, $L_3 = \{\mathbf{x}_3, \mathbf{x}_5, \mathbf{x}_7\}$, $L_4 = \{\mathbf{x}_6, \mathbf{x}_8\}$, and $L_5 = \{\mathbf{x}_9\}$.

The second phase of the algorithm corresponds to solving the triangular system using the constructed levels. Because every element in a set L_i is independent, all of its elements can be solved in parallel without any synchronization once all elements in all previous sets L_j for $j < i$ have been solved. So given k levels, the algorithm sequentially iterates over the k levels, solving for the elements in each level in parallel, and then synchronizing before continuing on to the next level. Henceforth, we refer to this approach as the level set (LS) algorithm.

Several parallel sparse triangular solution schemes have been proposed for distributed memory multiprocessors and early SIMD architectures, like the Connection Machine. Schreiber and Tang [7] were the first to propose the use of a coloring of the matrix's graph, followed by a reordering of the matrix by numbering rows and columns in each color contiguously. Similar to the level set algorithm, each parallel step corresponds to solving unknowns within the same color independently. There are as many such steps as the total number of colors. Jones et al. [8] show that graph coloring heuristics are effective in exposing near optimal parallel steps in the sparse triangular solution process. For matrices arising from finite elements models, they show that increase in parallelism due to coloring more than compensates for any increase in the number of iterations required for the convergence of PCG.

We see the coloring approach as the key for reducing dependencies and extracting large amounts of fine-grained parallelism for adapting sparse triangular solution to GPUs. The remainder of this paper discusses in detail the application of graph coloring to the triangular solve process, and analyses its performance relative to the level set (LS) algorithm.

Chapter 3

Adapting Sparse Triangular Solution to GPUs

In this section, we discuss how graph coloring can be used to extract fine-grained parallelism from sparse triangular solution by reducing the dependence among solution elements. As mentioned in Chapter 2, the effectiveness of the coloring approach has been demonstrated earlier for multiprocessors by Schreiber and Tang [7] and Jones and Plassmann [8]. This paper now discusses its application for speeding up sparse triangular solution on GPUs. The contributions in this section include the following:

- An illustrative example on the impact of coloring.
- An analysis of the relative merits of the color set (CS) and level set (LS) methods for achieving speed-ups on GPUs for a model 5-point finite-difference grid, a problem representative of the broader class of sparse matrices.
- A simple performance model to predict execution times on a GPU in terms of a few parameters related to hardware and method attributes.

3.1 Graph Coloring and Fine-Grain Concurrency

Similar to the level set (LS) algorithm, we consider a two-phase "color set" algorithm (CS), where the first phase concerns determining a coloring rather than a breadth first search, performing the corresponding matrix reordering, and setting up the other data structures for performing triangular solution in the second phase. For a given model problem, we compare how this scheme alters the dependencies of thread-level tasks that operate at the level of a row when compared to Naumov's level set (LS) scheme [5].

Consider PCG with a sparse symmetric positive definite matrix A. Preconditioner applications using incomplete factors of A typically involve an upper and lower triangular solution with matrices that have zero-nonzero structure derived from A. For ease of discussion, it is assumed that the structure is the same as that of A, i.e., a zero fill incomplete Cholesky factor, although these methods would apply broadly to all forms of preconditioning with incomplete factors. For an $N \times N$ matrix A, its graph, $G(A) = (V, E)$ is undirected with N vertices representing rows/columns with an edge (i, j) between vertices i and j if and only if $A_{i,j} \neq 0$. This graph representation can be used as a model of the computations in a specific sparse kernel. Now, a common approach for enhancing the performance of the kernel involves finding an appropriate renumbering of the vertices of the graph through an ordering π, followed by a symmetric permutation of the sparse matrix according to π. For adapting sparse triangular solution to GPUs, we seek a renumbering of vertices in $G(A)$ that can expose large independent sets of rows that can be mapped to GPU threads for fully parallel execution.

For a given numbering of rows and columns in A, i.e., the original order in which the matrix was formed, paths in G(A) represent dependencies during sparse triangular solution. Without loss of generality, assume the graph of the matrix has a single connected component and consider lower triangular solution with one or more GPU threads responsible for computing each component of the solution vector. Now a vertex, j, corresponds to computing the solution component j, which is possible only if its immediate predecessors $p(j) = \{i : i < j \text{ and } (i, j) \in E \text{ of } G(A)\}$ have been computed. This holds recursively, and so the dependencies

can be determined using graph traversals starting with vertex 1. The level set scheme [5] uses this property to group components into sets; all components in a set can be computed upon in parallel once components in predecessor sets have been calculated.

Consider now a coloring of the vertices of $G(A)$, by which two vertices that share an edge will be placed in different colors. Assume the coloring results in k colors, with $C(i), i = 1, \cdots, k$ indicating the set of vertices in color i such that $|C(1)| \geq |C(2)|, \cdots, |C(k)|$, i.e., the colors are numbered in non-increasing order of the set size and thus the number of vertices in the color. Consider a renumbering of vertices π such that vertices in each color are ordered contiguously starting with those in $C(1)$ and ending with the ones in $C(k)$. After a corresponding permutation of the matrix, observe that a component in $C(i)$ can have predecessors only in one or more of the earlier $i-1$ lower number colors. Therefore the solution of all components in $C(i)$ can occur independently and in parallel once components in all colors numbered $1, \cdots, (i-1)$ have been computed. Thus triangular solution can be staged as k major steps starting with $C(1)$ such that components in each color are independent. This allows one or more GPU threads to be mapped to each row involving the calculation of a single solution component; all rows in a color can be worked upon concurrently leading to large amounts of fine-grain, thread-scale parallelism. This is our color set (CS) algorithm.

As an example, Figure 3.1 shows matrix, dependency graphs, and the lower triangular preconditioner of a small 3×3 finite difference grid in the original order and after a 2-color ordering. Observe that there are five levels of dependencies with the original order; within each level only 1-3 solution components can be computed in parallel, for example, $\{x_3, x_5, x_7\}$ can be computed in parallel given $\{x_1, x_2, x_4\}$. With the color set ordering, there are only two levels of dependencies, containing 4 and 5 components that can be computed in parallel.

Consider a model five-point $K \times K$ finite-difference grid problem that is representative of the broader class of sparse linear systems arising from finite difference and finite element formulations [13]. The sparse matrix A has dimension $N = K^2$ with 5 nonzeroes along most rows. Its graph $G(A) = (V, E)$ is structurally the same as the grid. Consider the matrix in its natural ordering $\pi_{natural}$, as illustrated in Figure 3.1 (a), with the vertex at the bottom left corner numbered 1, followed

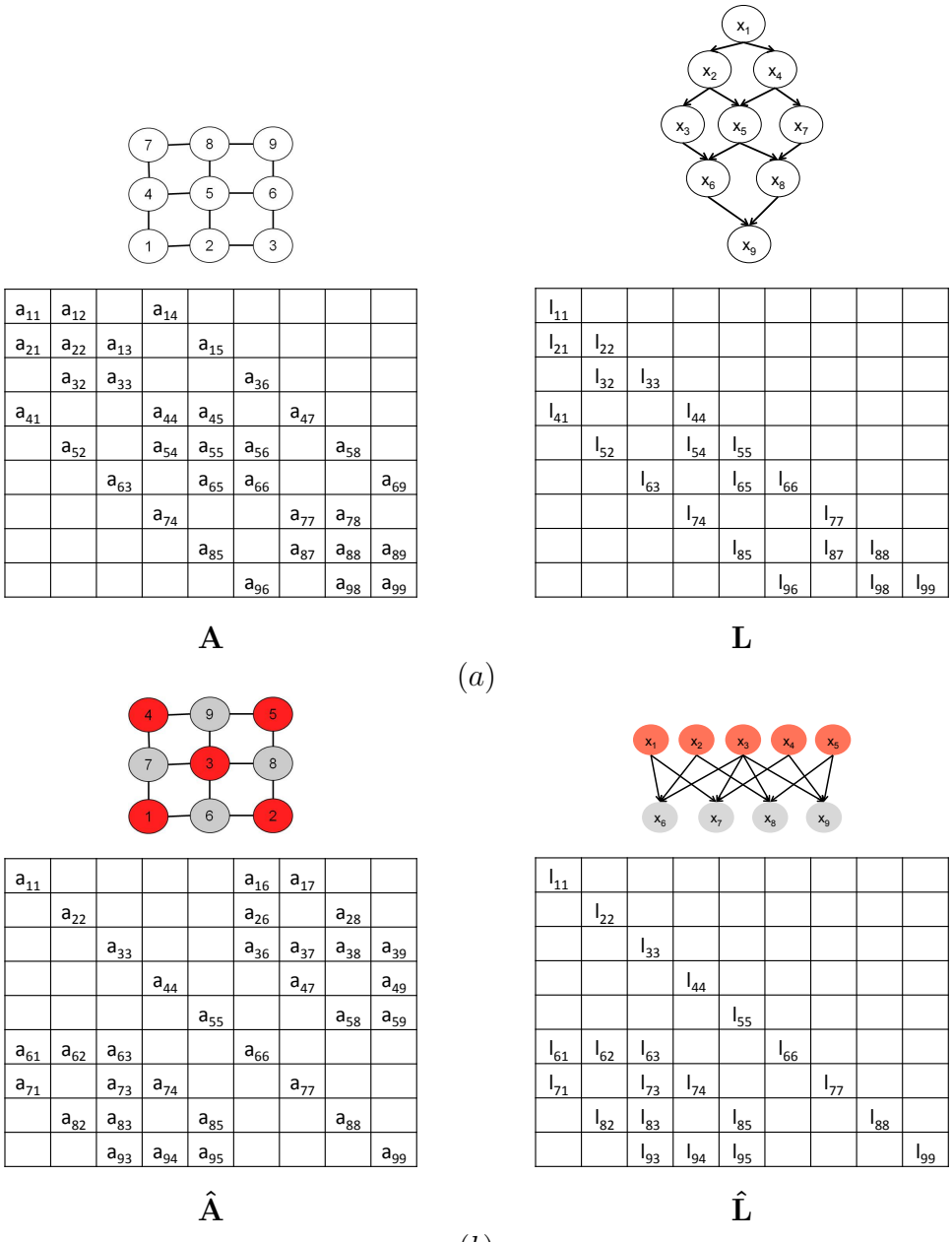


Figure 3.1 – An example of a sparse matrix corresponding to a model 3x3 finite-difference grid where (a) shows the graph, matrix, preconditioner and dependency graph for triangular solution using the level set (LS) scheme, and (b) shows corresponding objects for a 2-coloring using the color set (CS) scheme.

by consecutive numbers assigned to the bottom row ending with the bottom right corner numbered K. The vertices in the row above get numbers in the range $K+1$

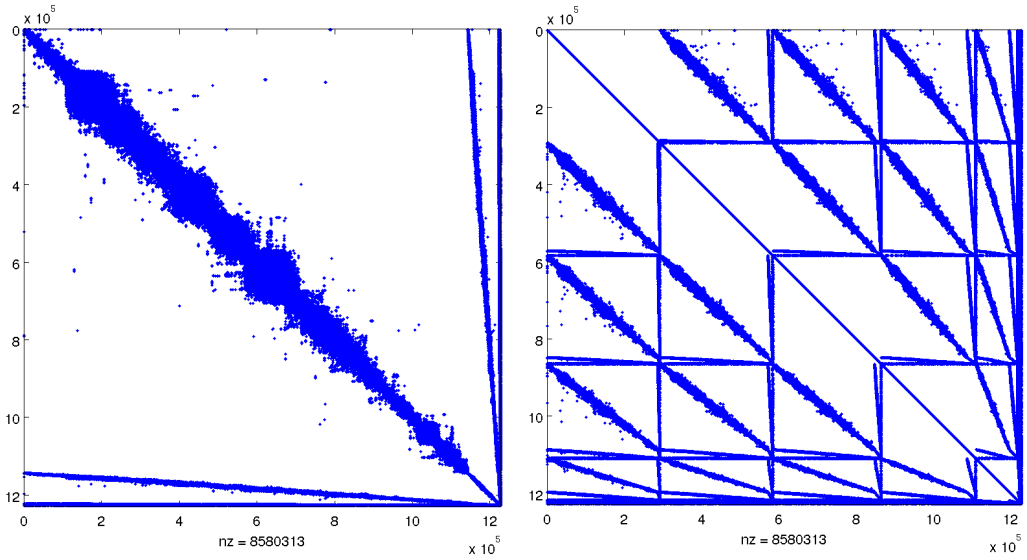
(for the vertex on the left edge) through $2K$ (for the vertex at the right edge). This process is repeated until the vertex at the top right corner is numbered $K^2 = N$. Consider the matrix reordered using 2-coloring π_{color}, starting with red for vertex at the bottom left corner, alternating colors along the bottom grid row, starting with black for the leftmost vertex in the row above and continuing to complete the "red-black" coloring.

Theorem 1. Consider the parallel sparse triangular solution of the model five-point finite-difference grid with a coefficient matrix of dimension N. With a natural numbering $\pi_{natural}$, the solution process with have some $c_1\sqrt{N}$ parallel steps, with each parallel step having at most some $c_2\sqrt{N}$ rows that can be processed concurrently, where c_1 and c_2 are small constants. With a 2-color ordering π_{color}, the solution process with have 2 parallel steps, with each parallel step having $\frac{N}{2}$ rows that can be processed concurrently.

Proof. The proof for $\pi_{natural}$ follows by construction. Observe that once component 1 is computed, one can proceed with the computation for the components above it and to its right on the grid (components $\sqrt{N}+1$ and 2). Proceeding similarly, one can see that component N can be processed after at most $2\sqrt{N}$ levels. Observe that in the intervening levels there are can be at most $\sqrt{N}$ components, corresponding to a vertex along each column of the grid. The proof for π_{color} follows directly from the 2-coloring because components in each color can be executed concurrently. □

For a broader class of matrices from finite difference and finite element schemes, coloring typically leads to a small number of parallel steps. This number is typically independent of the problem size when the graphs have low bounds on vertex degree. Corresponding sets contain large numbers of rows that can be mapped to GPU threads, as indicated in Figure 3.2 for an example. We observe that the coloring need not be close to optimal because the key issue is to reveal independent sets of components that are sufficiently large, i.e., at some multiple of the number of threads that can be executed concurrently by the GPU hardware. We therefore expect relatively simple coloring heuristics will be adequate, thus limiting the overheads of the preprocessing step.

Once the matrix is reordered through coloring, the level set approach could be applied to it to yield further performance benefits. However, that would require the



(a) Original ordering **(b)** Color set ordering

Figure 3.2 – An example matrix `Schmid_thermal2` with 1,228K rows for a steady state thermal problem using an unstructured finite element method [14]. The level set triangular solution with a natural ordering (left) has 1,240 levels with an average parallel workload of $\phi(A, LS) = 990$ rows per step, while coloring produces 7 parallel steps with an average parallel workload of $\phi(A, CS) = 175,435$ rows per step.

additional preprocessing required by the level set scheme for identifying component level dependency information. In our color set implementation, we avoid such dependency analysis overheads by simply processing colors in increasing order of color number (which was determined by color size) and processing all components in a color in parallel. The actual dependency level of a specific component can be lower than its color number, i.e., a component in color i may not depend on components in all earlier colors. However, we expect this simplification will likely not impact performance significantly, especially if our performance model in Equation (3.1) is verified to be predictive for GPUs.

3.2 Performance Impacts of Fine-Grain Concurrency

Current generation GPUs already support large levels of fine-grain, thread-scale concurrency and these levels are likely to grow over the next several years. Consequently, we expect that the amount of available fine-grain thread-scale concurrency in the problem will determine to a large extent the effective execution rate and thus performance. Following this primary effect, we expect that hardware limiters such as bandwidth, per-thread-block and per-thread resources will modulate performance. Finally, additional problem-specific factors, such as nonzero densities per row and hence workload per solution component, may lead to performance trade-offs based on thread to row mappings.

We now focus on how the principal benefits of fine-grain thread-scale concurrency can be modeled to predict performance. In both the level set (LS) and color set schemes (CS), the basic unit of work that can be assigned to a thread concerns solving for a component k and thus processing row k. Now rows in a level or color comprise a *parallel workload* because each row can be processed independently and in parallel. Consider processing the nonzeroes in a row. For each nonzero, after loads for the nonzero value, the column subscript, and the corresponding solution component, there is a floating point multiply and add operation. If multiple threads are assigned to a row, it is presumed that contributions from all threads are added before an update to the right hand side value, thus needing only one load and store of the right hand side value per row. The loads to operation ratio is high, demanding that load latency be masked by having multiple threads in flight. In this regard, we expect little benefit for assigning multiple threads per row, because row nonzero densities are relatively low. On the other hand, there will be immediate benefits if the parallel workload size, i.e., the number of rows in a level or color, is high.

We conjecture that to a first approximation, the size of the *parallel workload* determines performance. Let $T(A, x)$, the time for triangular solution using method x be equal to $nnz(A) \times T_{nnz}(A, x)$ where $nnz(A)$ is the number of nonzeroes in matrix A and $T_{nnz}(A, x)$ is the time to process one nonzero for matrix A using method $x = LS$ or $x = CS$. Define the average parallel workload $\phi(A, x)$, i.e., the

number of rows that can be processed concurrently on average, as $\phi(A, x) = \frac{N}{\delta(A,x)}$ where $\delta(A, x)$ is the number of dependence sets for method x, i.e., the number of levels for LS and the number of colors for CS. Let $N_r(A) = \frac{nnz(A)}{N}$ be the average number of nonzeros per row in A, and P be the hardware capacity per cycle, i.e., the number CUDA cores in the GPU. For a suitable constant α related to the data access latency, we propose the following model.

$$T_{nnz}(A, x) = \begin{cases} \frac{1}{\alpha N_r(A)\phi(A,x)}, & \phi(A, x) \leq P \\ \frac{1}{\alpha N_r(A)P}, & \text{otherwise.} \end{cases} \tag{3.1}$$

In the next section, we empirically evaluate the performance benefits of our CS approach and estimate parameters for the equation above using some data from experiments.

Chapter 4

Experiments and Analyses

In this chapter we report on our empirical evaluation on the relative performance of the color set and level set methods. We report on the test methodology, including test matrices, followed by results for model finite-difference grids and a suite of sparse test matrices representative of practical applications.

4.1 Experimental setup

The test platform consisted of a single node of the Penn State Lion-GA cluster [15], including two 6-core Intel Xeon X5675 CPUs and 4 GB of RAM reserved for the test process. Each node also included three Nvidia Tesla M2090 GPUs. Our tests were run on a single GPU.

Two separate algorithms for triangular solve are analyzed. The first algorithm uses the level set method implemented by Nvidia and the cuSPARSE library [5, 6] for both the analysis and solve phases. We refer to this scheme as the level set (LS) scheme. The second algorithm uses the Boost graph library [16] in the analysis phase to color the graph of the matrix and permute it. The solve phase then uses a custom kernel to perform triangular solve on an entire color without synchronization. We call this the color set (CS) scheme.

The test suite includes 29 real, symmetric, positive definite matrix from the University of Florida Sparse Matrix Collection [14] and a 500×500 model finite difference grid. Properties of these matrices, including derived measures such as the parallel workload are shown in in Table 4.1.

ID	Matrix	N	nnz(A)	$\delta(\mathbf{A}, \mathbf{CS})$	$\phi(\mathbf{A}, \mathbf{CS})$	$\delta(\mathbf{A}, \mathbf{LS})$	$\phi(\mathbf{A}, \mathbf{LS})$	$\mathbf{N_r}(\mathbf{A})$
1	2cubes_sphere	101,492	1,647,264	13	7,807.1	14,330	7.1	16.2
2	thermomech_TK	102,158	711,558	7	14,594.0	323	316.3	7.0
3	thermomech_TC	102,158	711,558	7	14,594.0	323	316.3	7.0
4	x104	108,384	10,167,624	78	1,389.5	4,993	21.7	93.8
5	shipsec8	114,919	6,653,399	54	2,128.1	3,128	36.7	57.9
6	ship_003	121,728	8,086,034	60	2,028.8	4,368	27.9	66.4
7	cfd2	123,440	3,087,898	15	8,229.3	4,358	28.3	25.0
8	boneS01	127,224	6,715,152	36	3,534.0	814	156.3	52.8
9	shipsec1	140,874	7,813,404	48	2,934.9	2,101	67.1	55.5
10	bmw7st_1	141,347	7,339,667	54	2,617.5	708	199.6	51.9
11	Dubcova3	146,689	3,636,649	16	9,168.1	3,267	44.9	24.8
12	bmwcra_1	148,770	10,644,002	40	3,719.3	730	203.8	71.5
13	G2_circuit	150,102	726,674	4	37,525.5	734	204.5	4.8
14	shipsec5	179,860	10,113,096	50	3,597.2	2,689	66.9	56.2
15	thermomech_dM	204,316	1,423,116	7	29,188.0	323	632.6	7.0
16	pwtk	217,918	11,634,424	48	4,540.0	142,169	1.5	53.4
17	hood	220,542	10,768,436	42	5,251.0	605	364.5	48.8
18	BenElechi1	245,874	13,150,496	30	8,195.8	5,755	42.7	53.5
19	offshore	259,789	4,242,673	12	21,649.1	3,453	75.2	16.3
20	msdoor	415,863	20,240,935	42	9,901.5	6,938	59.9	48.7
21	af_x_k101	503,625	17,550,675	15	33,575.0	6,766	74.4	34.8
22	af_shellx	504,855	17,588,875	25	20,194.2	3,726	135.5	34.8
23	parabolic_fem	525,825	3,674,625	5	105,165.0	8	65,728.1	7.0
24	Fault_639	638,802	28,614,564	34	18,788.3	4,463	143.1	44.8
25	apache2	715,176	4,817,870	3	238,392.0	665	1,075.5	6.7
26	ecology2	999,999	4,995,991	2	499,999.5	2,000	500.0	5.0
27	thermal2	1,228,045	8,580,313	7	175,435.0	1,240	990.4	7.0
28	StocF-1465	1,465,137	21,005,389	11	133,194.3	3,005	487.6	14.3
29	G3_circuit	1,585,478	7,660,826	4	396,369.5	2,595	611.0	4.8
30	500 model grid	250,000	1,248,000	2	125,000	999	250.3	5.0

Table 4.1 – Test suite of 30 sparse matrices. The parallel workload measure is shown for the level set (LS) and color set (CS) methods in the columns labeled $\phi(\mathbf{A}, \mathbf{CS})$ and $\phi(\mathbf{A}, \mathbf{LS})$.

In Chapter 3, we had shown that coloring produces large amounts of fine-grain concurrency for the model $K \times K$ grid of dimension $N = K^2$ by reducing the number of dependence sets to 2, unlike level sets on the natural order where the number of dependence sets is proportional to $\sqrt{N}$. To evaluate these effects on actual performance scalability, we consider a series of model grids ranging in size from 400×400 to 1500×1500, as well as the test matrices from Table 4.1 for both LS and CS methods.

4.2 Effects of Fine-Grain Concurrency on Performance

A primary benefit of the color set scheme concerns exposing larger levels of fine-grain concurrency than level set for a given matrix. To measure such fine-grain concurrency, we had defined, in Chapter 3, the average parallel workload $\phi(A, x)$, i.e., the number of rows that can be processed concurrently on average, for method x on matrix A. These parallel workload values are shown in Table 4.1. We observe that $\phi(A, CS)$ values are typically larger than $\phi(A, LS)$ by factors of 100 or more.

We had conjectured that the time per nonzero of the kernel could be modeled using Equation 3.1 in Chapter 3 , which is replicated here.

$$T_{nnz}(A, x) = \begin{cases} \frac{1}{\alpha N_r(A)\phi(A,x)}, & \phi(A, x) \leq P \\ \frac{1}{\alpha N_r(A)P}, & \text{otherwise.} \end{cases}$$

For the Tesla M2090 we set $P = 512$, representing the maximum hardware parallel processing capacity per cycle at 512 CUDA cores. To cancel out the variation of $N_r(A)$, Figure 4.1 shows experimentally collected values for $N_r(A) \times T_{nnz}(A, x)$ plotted vs $\phi(A, x)$ compared to values predicted by the model with $\alpha = 6.448e{-}3$, obtained through a linear least square fit to experimentally observed timings for both CS and LS when $\phi(A, x) < 512$. Observe that this model approximates the major observed trend in the time per nonzero. CS leads to larger values of the average parallel workload and hence the clustering of points representing significantly lower times per nonzero from better utilization of the GPU hardware.

To interpret this value of α, assume that $\min(\phi(A, x), P)$ operations can be

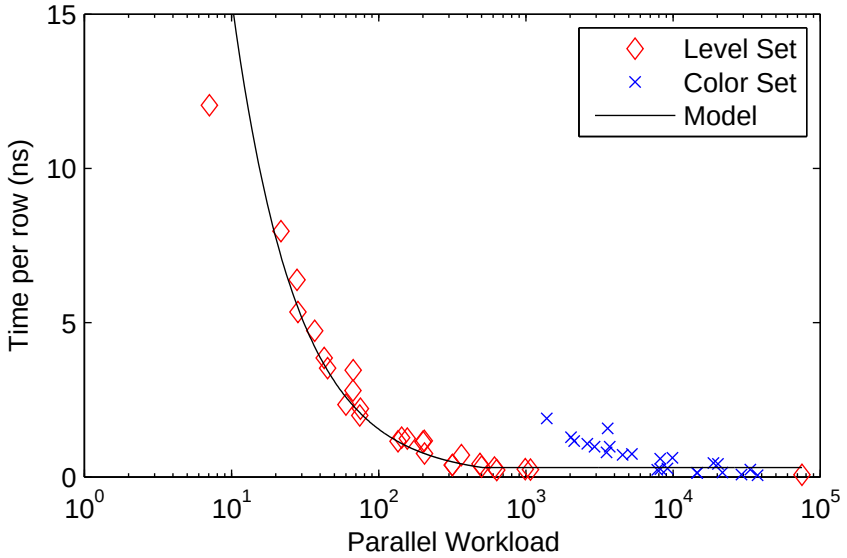


Figure 4.1 – Comparison of measured times for $N_r(A) \times T_{nnz}(A, LS)$ in red, $N_r(A) \times T_{nnz}(A, CS)$ in blue, and $N_r(A) \times T_{nnz}(A, x)$ predicted by the model (3.1) with α fitted to points with $\phi(A, x) < 512$.

completed in 1 cycle once corresponding data have been loaded, and that either bandwidth or instructions in flight are limiting factors that prevent full masking of the latency of data access. For simplicity we ignore the operations related to the solution, right hand side, and rowPtr vectors at they will constitute only a small percentage of the total operations. For each non-zero in A we must load 2 words, the column index and the value. However, because L will only have roughly half as many non-zeros as A, during a triangular solve we must load 1 word per non-zero in A. Then assuming a latency of l cycles to load approximately all $\frac{1}{2}N_r(A)$ nonzeros in each row of L, and using the Tesla M2090's clock rate of 1.3 GHz we can approximate

$$T_{nnz} \approx \frac{1}{\phi(A,x)} \left(\frac{1 \text{ word}}{1 \text{ nz}}\right) \left(\frac{2l \text{ cycles}}{N_r(A) \text{ words}}\right) \left(\frac{1 \text{ ns}}{1.3 \text{ cycles}}\right) = \frac{2l}{1.3 N_r(A) \phi(A,x)} \frac{\text{ns}}{\text{nz}}$$

We can view α as representing an effective latency of $l = \frac{1.3 N_r}{2\alpha} \approx 3399.6$ cycles per row. This interpretation maps to Little's law [17] and a physical meaning that is in-line with hardware specifications, despite the simplifying assumptions [18].

Next, we consider the measured performance for a series of $K \times K$ model

grids using both LS and CS algorithms. Recall that in Chapter 3 we showed that the LS algorithm will have $c_1\sqrt{N}$ parallel steps, which gives an average parallel workload $\phi(A, LS) = \frac{1}{c_1}\sqrt{N} = \frac{1}{c_1}K$. We also showed that the CS algorithm will have only 2 parallel steps regardless of grid size, and an average workload $\phi(A, CS) = \frac{N}{2} = \frac{K^2}{2}$. From this simple observation we can clearly see that for the model grid problem the CS algorithm will be much more scalable than the LS algorithm as hardware parallelism increases. This effect is clearly visible in Figure 4.2, which shows measured $T_{nnz}(A, x)$ for both algorithms on a series of model grid sizes.

Figure 4.2 shows that $T_{nnz}(A, LS)$ is bound by available parallel workload which is not sufficient to utilize the hardware capacity. Consequently, incremental performance benefits continue even as grid sizes increase to values as large as $K = 1500$. $T_{nnz}(A, CS)$, on the other hand, is limited by the parallel processing capacity when $\phi(A, CS) > 512$. This happens for relatively small grid sizes that are greater than or equal to $K = 32$. More importantly, from our model (3.1) we can see that the CS algorithm will be more scalable as the hardware parallel processing capacity P increases, as expected on future generation GPUs. For example, even on a relatively large 400×400 grid, the average parallel workload for level set (LS) of approximately $\phi(A, LS) \approx 200$ is less than the current hardware capacity $P = 512$, thus limiting the performance. However, the color set (CS) algorithm on the same model grid exposes a much larger average parallel workload, at $\phi(A, CS) = 80,000$, to allow the efficient utilization of much larger values of GPU hardware parallelism.

We now consider the impact of coloring relative to the level set method for all matrices from Table 4.1; the structures of 29 of these sparse matrices are more irregular than the structure of the model grid. Figure 4.3 shows the speedup of the CS algorithm relative to LS, i.e., values of $\frac{T(A,LS}{T(A,CS}$) for each matrix A listed in Table 4.1. CS exhibits a speed-up with geometric mean of 5.41 relative to LS, with speed-ups of over 10 for several matrices. These speed-ups are primarily determined by the average parallel workload made available by each scheme. Observe that the number of colors (see Table 4.1) maintains at a small value independent of the matrix size N, much as for the model problem, leading to extremely large values of $\phi(A, CS)$. For LS, the average parallel workload is determined largely by the ordering of the matrix, which can greatly increase the number of levels (Table 4.1)

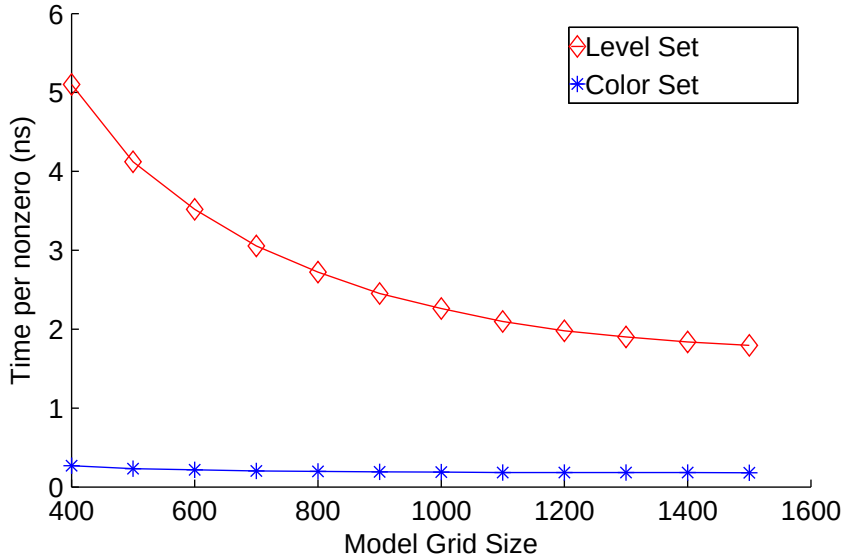


Figure 4.2 – Measured $T_{nnz}(A, x)$ for $K \times K$ model grid matrices.

leading to significantly lower values of $\phi(A, LS)$.

Additional performance benefits beyond those observed by CS could be obtained by applying LS to matrices permuted according to a coloring π_{color}. We refer to this algorithm as color plus level set (CPLS). Figure 4.4 shows the average speedup in this case to be 7.23 relative to LS on $\pi_{natural}$. However, these benefits will come at the combined overheads of the preprocessing required for both CS and LS. Additionally, we conjecture that as our model predicts this increase in performance is not due to any increased parallelism exposed by the extra LS analysis, but rather to optimizations in the implementation of the LS algorithm that have not yet been incorporated into the CS algorithm. If this were the case, then the performance of CS would approach that of CPLS, but without the overhead of the extra LS analysis.

One prime candidate for further optimization is the number of threads assigned per row during the CS solve phase. For simplicity, the implementation in our experiments used a predetermined constant independent of matrix structure. Through experimentation, a value of 4 threads per row was found to result in the largest aggregate speedup of CS over LS for the tests in Table 4.1, however that was not the case for every individual matrix. For example, we see in Figure 4.1 that matrices with low parallel workload $\phi(A, CS)$ of around 10^3 have a slightly

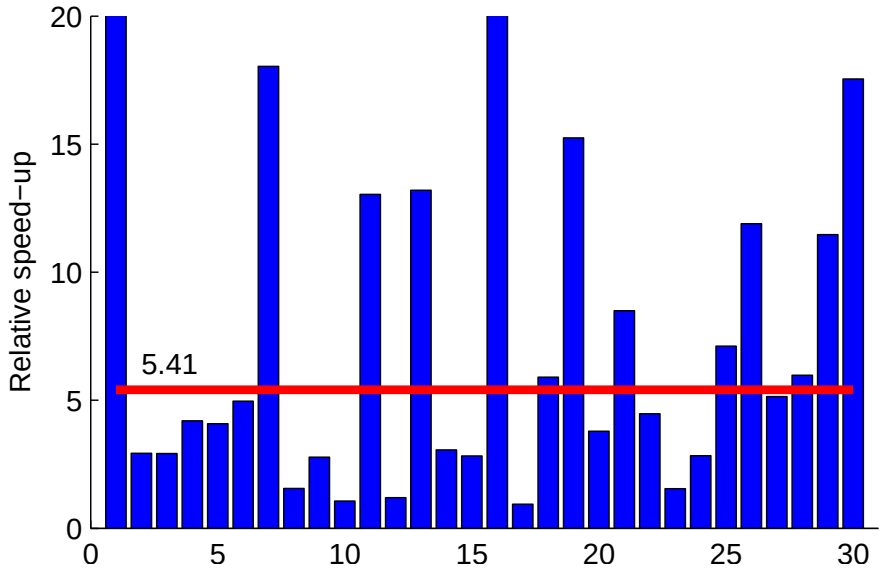


Figure 4.3 – Speedup of CS algorithm relative to LS algorithm, i.e., $\frac{T(A,LS)}{T(A,CS)}$ for each matrix A listed in Table 4.1.

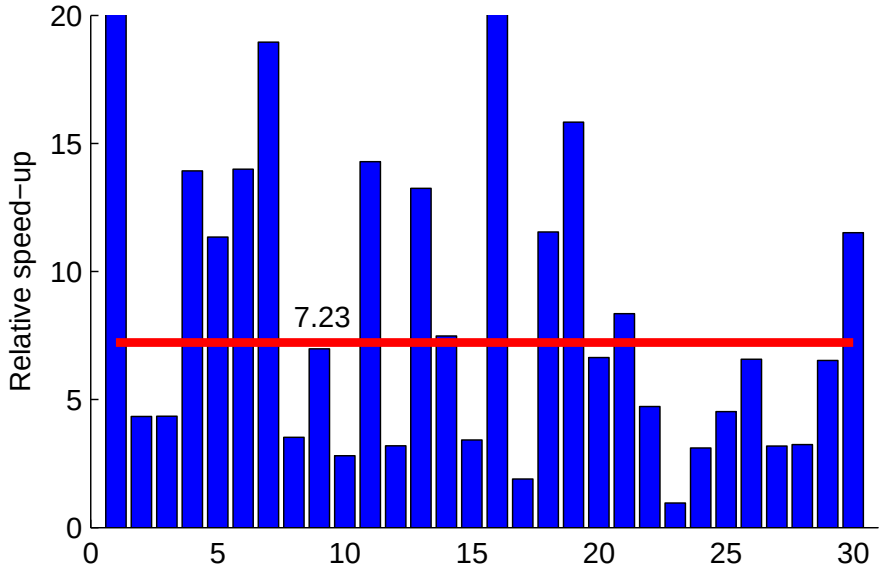


Figure 4.4 – Speedup of LS algorithm applied to π_{color} relative to LS algorithm applied to $\pi_{natural}$ for each matrix A listed in Table 4.1.

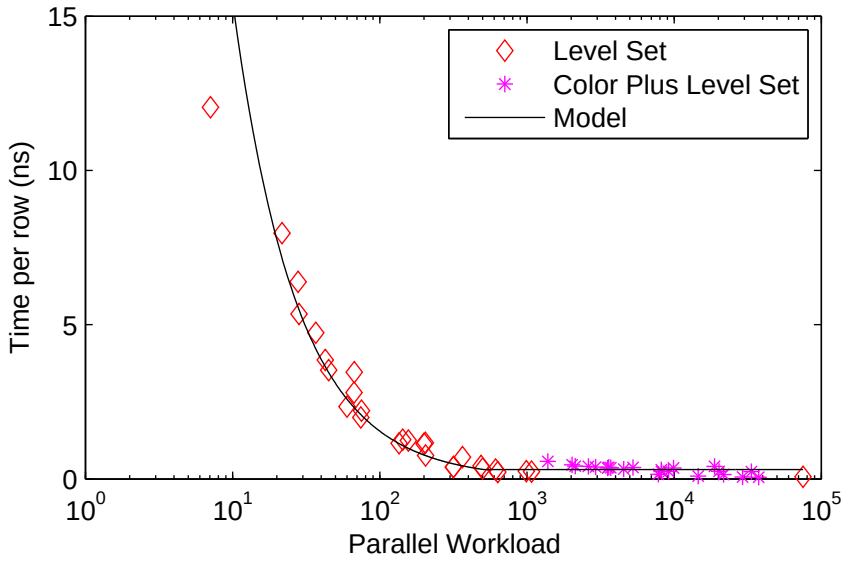


Figure 4.5 – Same model as before with points for the combined color set and level set (CPLS) algorithm added. Notice how the CPLS points closely match the model across the entire domain, rather than just at high parallel workload like the CS algorithm.

higher runtime than the predicted by the model, and that runtime approaches the model as parallel workload increases.

A possible explanation for this observation is that matrices with a large number of nonzeros per row are very dense, which will typically result on a larger number of colors and thus a low parallel workload. To understand how this will affect the CS runtime, lets first consider the best case scenario when L has exactly 4 nonzeros in every row. Using 4 threads per row in this case is optimal because during a single instruction, each warp will read 8 entire rows of A which are stored in the same block of contiguous memory that can be read in the minimum number of loads. As the number of nonzeros per row increases however, warps will be reading only a portion of each row per instruction, resulting in more fragmented memory access pattern and multiple loads to disjoint memory blocks being issued. We expect that dynamically choosing the number of threads per row to be optimal for the entire matrix, or optimal per color, will result improved performance approaching that of the model. The source code for the LS algorithm was not available to us to confirm it, however we suspect that this is what the LS algorithm is doing.

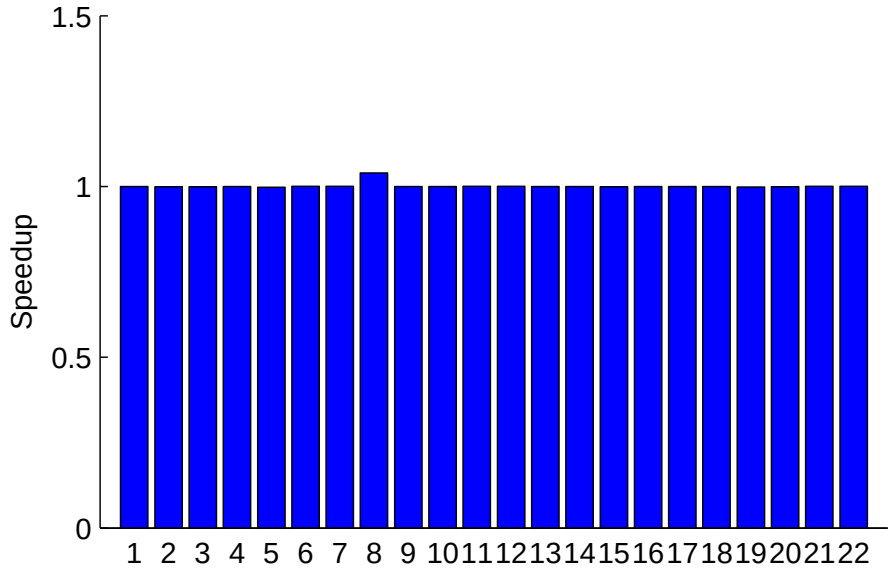


Figure 4.6 – Speed of SpMV using π_{color} relative to $\pi_{natural}$

Figure 4.5 shows experimental data from the combined color plus level set (CPLS) compared to model 3.1, and as we expect the CPLS does more consistently agree with the model across the entire domain. Although we haven't confirmed that the number of threads per row was the limiting factor in the CS tests, this seems to support our hypothesis that it is factors other than increased parallelism exposed by the extra LS analysis that caused the increased performance of CPLS.

4.3 Overheads of Preprocessing Analysis

Benefits of the color set scheme are apparent in Figure 4.3. However, they come at the cost of the preprocessing step which involves finding a coloring. For our implementation we used the Boost graph library [16] to obtain a near optimal coloring on the CPU before transferring the problem to the GPU. Figure 4.7 shows that on average, the measured cost of performing the coloring relative to a single triangular solve on the GPU using the CS algorithm for the 30 test matrices from Table 4.1 was $124 \times T(A, CS)$. Figure 4.8 shows the corresponding overhead for the level set algorithm, again relative to the color set triangular solve time for comparison, was $61 \times T(A, CS)$, thus CS incurs larger overheads than LS.

We expect that in practical applications, the preprocessing costs of CS will be

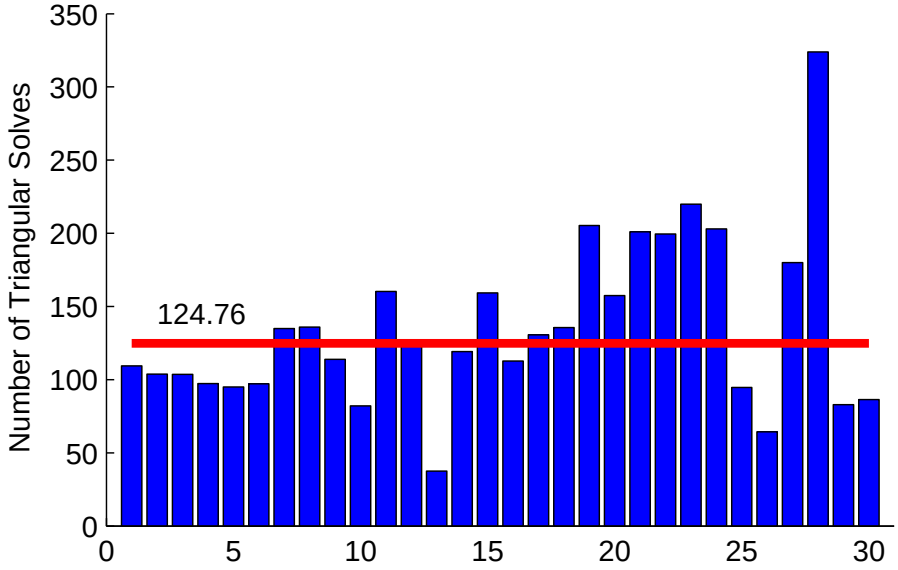


Figure 4.7 – Overhead of performing coloring in CS algorithm relative to $T(A, CS)$ for each matrix A listed in Table 4.1.

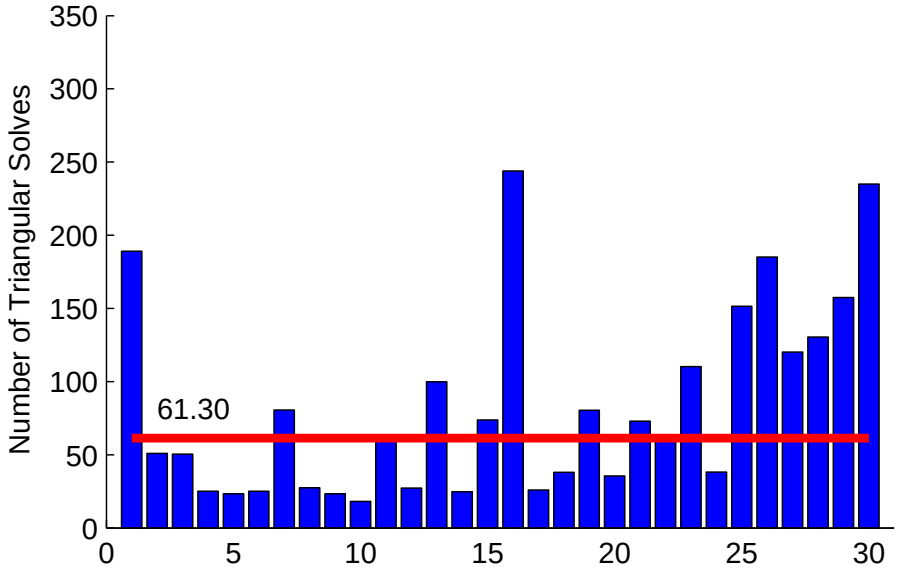


Figure 4.8 – Overhead of performing the breadth first search in LS algorithm relative to $T(A, CS)$ for each matrix A listed in Table 4.1.

amortized over a larger number of triangular solutions, because solvers such as preconditioned conjugated gradients often require several hundred iterations, with each iteration involving two triangular solutions. Additionally, such solvers are

often applied on linear systems where the numerical values in the matrix or the right hand side vector may change without changes in the sparsity structure, thus allowing the preprocessing costs of CS to be amortized over multiple linear system solutions. Finally, in every test matrix the average parallel workload $\phi(A, CS)$ exposed by the near optimal coloring exceeded the maximum parallel capacity P of a single Tesla M2090 by several factors. Consequently, overheads could potentially be reduced by a faster coloring algorithm that may lead to more colors but still exposes large parallel workloads that match or exceed the parallel capacity of the GPU hardware.

Chapter 5

Concluding Remarks

5.1 Summary of Results

In this paper, we have demonstrated how sparse triangular solution can be adapted to GPUs through a matrix ordering derived from a coloring of its graph. Our color set scheme and the level set scheme of Naumov [5] operate as a sequence of parallel steps where each parallel step involves a set of rows that can be operated upon independently and in parallel. The sequencing of these parallel steps manages dependencies. The size of the parallel steps (in terms of the number of rows) is a measure of the fine-grain thread-scale concurrency; larger sets are better suited to utilized the parallel capacity of GPU hardware.

We show that for model grid problems, that are representative of the broad class of systems from finite-element and finite-difference methods, coloring can reduce the number of parallel steps to a small value, independent of the matrix dimension. We also show that, to a first approximation, the throughput achieved per matrix nonzero can be modeled as being in inverse proportion to the average size of the parallel steps, until the latter exceeds the concurrent computational capacity of the hardware.

Our experiments support our analytical results and demonstrate that the color set schemes greatly increases the average parallel workload (i.e., the average number of rows in parallel steps) to values that correspond to a fixed fraction of the nonzeros in a matrix, indepedent of matrix dimension. This leads to greatly enhanced throughput per nonzero on GPUs for our color set method compared to

the level set method. On average, our color set method achieves a speed-up of 5.41 relative to level set, with speed-ups as large as 63 in some cases. Additionally, the level method can be applied to the matrix reordered by coloring to yield an increased average speedup of 7.23, albeit at the expense of the preprocessing required for both methods. We conjecture that this increase is due to optimizations unrelated to the extra parallelism exposed by the additional preprocessing, but have not confirmed this hypothesis.

Finally we have shown that the while applying the colored permutation significantly speeds up the triangular solve kernel, it had almost no effect on the other major operation in PCG, sparse matrix vector multiply kernel, which together with triangular solve constituted 99% of each PCG iteration.

5.2 Future Work

We have shown in this paper how graph coloring can be used to expose more parallelism in the sparse triangular solve kernel, overcoming the most significant limiting factor preventing full utilization of the hardware. However, our analysis indicates that secondary effects are still limiting the performance of our algorithm. We proposed, and data seems to support the conjecture, that the next most important factor limiting performance may be the number of threads per row used during the solve phase. Further tests and analysis need to be done to confirm this however.

In addition to the performance of the solve phase, we have also shown that the overhead of performing the coloring in the analysis phase to be significant at $124 \times T(A, CS)$. Our tests were done using a near optimal coloring provided by the Boost graph library. These colorings resulted in a very high average parallel workload $\phi(A, CS)$, which for every test matrix in Table 4.1 far exceeded the Tesla M2090's parallel hardware capacity of $P = 512$ CUDA cores. A faster but less efficient coloring heuristic may be used to reduce the overhead of preprocessing without having a major impact on overall performance, so long as the average parallel workload remains above the hardware capacity. Also, although the average parallel workload of all the test matrices exceeded the hardware capacity, in most cases the smallest colors in each matrix were below the hardware capacity. A coloring

heuristic that maximizes the minimum color size rather one than minimizing the total number of colors may be more effective for the color set algorithm.

Finally, this paper analysed the performance of a color set algorithm relative to Nvidia's level set algorithm both on GPU. Naumov compared the runtime of the level set algorithm on GPU to Intel's MKL, however no studies have been done comparing CPU implementations of the level set or color set algorithms to their GPU counterparts. As high end GPU's typically consume more power than current CPU's it is important to determine if it is cost effective to offload the triangular solve process to GPU, or if parallel CPU implementations of these algorithms can achieve similar performance but with lower energy consumption.

Appendix

Color Set Analyze Phase Code

```
template< typename T >
void BGL_ordering( CusparseCSRMatrix<T> const &A,
                   size_t *p,
                   int *numColors = NULL,
                   int **colorPtr = NULL )
{
    typedef typename boost::adjacency_list<
        typename boost::listS,
        typename boost::vecS,
        typename boost::undirectedS >
        Graph;
    typedef typename boost::graph_traits<Graph>::vertex_descriptor
```

```
        vertex_descriptor;
typedef typename boost::graph_traits<Graph>::vertices_size_type
        vertices_size_type;
typedef typename boost::property_map<Graph,typename boost::vertex_index_t>::const_type
        vertex_index_map;
typedef std::pair<int,int> Edge;

std::vector< Edge > edge_array( A.m_data->m_nnz - A.m_data->m_nRows );
size_t k = 0;
for( size_t i = 0; i < A.m_data->m_nRows; ++i )
{
    for( int j = A.m_data->m_rowPtr[i]; j < A.m_data->m_rowPtr[i+1]; ++j )
    {
        if( (int)i != A.m_data->m_colInd[j] )
        {
            edge_array[k] = Edge( (int)i, A.m_data->m_colInd[j] );
            ++k;
        }
    }
}

Graph g(&edge_array[0],&edge_array[0]+edge_array.size(),A.m_data->m_nRows);
std::vector<vertices_size_type> color_vec(num_vertices(g));
typename boost::iterator_property_map<vertices_size_type*,vertex_index_map>
    color(&color_vec.front(),boost::get(boost::vertex_index,g));
boost::sequential_vertex_coloring(g,color);
```

```
std::map< int, std::set<int> > colorMap;
for( int i = 0; i < (int)A.m_data->m_nRows; ++i )
{
    colorMap[color[i]].insert( i );
}

if( NULL != colorPtr && NULL != numColors )
{
    *numColors = colorMap.size();
    *colorPtr = (int*)malloc( ( *numColors + 1 ) * sizeof( int ) );
}

int cIdx = 0;
k = 0;
(*colorPtr)[0] = 0;
for(typename std::map< int, std::set<int> >::const_iterator
    it = colorMap.begin(); it != colorMap.end(); ++it )
{
    ++cIdx;
    (*colorPtr)[cIdx] = (*colorPtr)[cIdx-1] + it->second.size();
    for( typename std::set<int>::const_iterator it2 = it->second.begin();
         it2 != it->second.end(); ++it2 )
    {
        p[k] = *it2;
        ++k;
    }
}
```

```
    return;
}
```

Color Set Solve Phase Code

```
#include <stdio.h>
#include <cuda_runtime_api.h>
#include <cublas_v2.h>
#include <cusparse.h>

#include <Logger.h>
#include <TriSolveKernels.h>

#ifndef CUDA_BLOCK_SIZE
#define CUDA_BLOCK_SIZE 512
#endif

#ifndef CUDA_THREADS_PER_ROW
#define CUDA_THREADS_PER_ROW 4
#endif

/*!
  \brief Solves one step of the triangular system (D+L)*x=alpha*b or (D+U)*x=alpha*b for x.
         D is a diagonal matrix, L & U are strictly lower & upper triangular
         respectively. The system should be pre analyzed & divided into contiguous
         sections that can be solved in parallel, and this function will solve the
```

```
         range [Ik0,Ik1)

  \param Ik0 Starting row of the range [Ik0,Ik1) to solve
  \param Ik1 Ending row of the range [Ik0,Ik1) to solve
  \param D The elements of the diagonal matrix D
  \param rowPtr The index of the beginning of each row in the CSR matrix UL
  \param colInd The column index of each element in the CSR matrix UL
  \param UL The values of the strictly upper triangular matrix U or strictly lower
            triangular matrix L, stored in CSR format
  \param x The solution vector
  \param alpha a scalar
  \param b The right hand side vector
*/
__global__
void csrBackSubstituteKernel( int Ik0, int Ik1, float *D,
                              int *rowPtr, int *colInd, float *UL,
                              float *x, float alpha, float *b )
{
    int i = Ik0 + blockIdx.x * blockDim.y + threadIdx.y;
    if( i >= Ik1 )
    {
        return;
    }

    __shared__ float sum[CUDA_THREADS_PER_ROW][CUDA_BLOCK_SIZE/CUDA_THREADS_PER_ROW];
    sum[threadIdx.x][threadIdx.y] = 0.0f;
    // compute, in parallel, the different sums needed for the inner products
```

```
    for( int j = rowPtr[i] + threadIdx.x; j < rowPtr[i+1]; j += blockDim.x )
    {
        sum[threadIdx.x][threadIdx.y] += UL[j] * x[colInd[j]];
    }
    __syncthreads();

    // perform the reduction to accumulate the total sums of each row in sum[0][threadIdx.y]
    for( int j = blockDim.x / 2; j > 0; j /= 2 )
    {
        if( threadIdx.x < j )
        {
            sum[threadIdx.x][threadIdx.y] += sum[threadIdx.x+j][threadIdx.y];
        }
        __syncthreads();
    }

    // update x[i]
    if( 0 == threadIdx.x )
    {
        x[i] = ( alpha * b[i] - sum[0][threadIdx.y] ) / D[i];
    }
    return;
}

/*!
  \brief Solves triangular system (D+L)*x=alpha*b or (D+U)*x=alpha*b for x.
         D is a diagonal matrix, L & U are strictly lower & upper triangular
```

```
        respectively. The system should be pre analyzed & divided into contiguous
        sections that can be solved in parallel

  \param colorPtr An array of size N+1 containing the index of the
                  first row in each color
  \param N The number of colors
  \param upperTriangular True if solving (D+U)*x=alpha*b, false otherwise
  \param D The elements of the diagonal matrix D
  \param rowPtr The index of the beginning of each row in the CSR matrix UL
  \param colInd The column index of each element in the CSR matrix UL
  \param UL The values of the strictly upper triangular matrix U or strictly lower
            triangular matrix L, stored in CSR format
  \param x The solution vector
  \param alpha a scalar
  \param b The right hand side vector
*/
void
csrTriSolve( int *colorPtr, int N,
             bool upperTriangular, float *D,
             int *rowPtr, int *colInd, float *UL,
             float *x, float alpha, float *b )
{
    for( int k = 0; k < N; ++k )
    {
        int Ik0 = upperTriangular ? colorPtr[N-k-1] : colorPtr[k];
        int Ik1 = upperTriangular ? colorPtr[N-k] : colorPtr[k+1];
        dim3 blockDim( CUDA_THREADS_PER_ROW, CUDA_BLOCK_SIZE / CUDA_THREADS_PER_ROW );
```

```
        int gridDim = ( Ik1 - Ik0 ) / blockDim.y;
        if( ( Ik1 - Ik0 ) % blockDim.y != 0 )
        {
            ++gridDim;
        }
        csrBackSubstituteKernel<<<gridDim,blockDim>>>( Ik0, Ik1, D, rowPtr, colInd, UL, x, alpha, b );
        cudaThreadSynchronize();
    }

    return;
}

/*!
  \brief Solves one step of the triangular system (D+L)*x=alpha*b or (D+U)*x=alpha*b for x.
         D is a diagonal matrix, L & U are strictly lower & upper triangular
         respectively. The system should be pre analysed & divided into contiguous
         sections that can be solved in parallel, and this function will solve the
         range [Ik0,Ik1)

  \param Ik0 Starting row of the range [Ik0,Ik1) to solve
  \param Ik1 Ending row of the range [Ik0,Ik1) to solve
  \param D The elements of the diagonal matrix D
  \param rowPtr The index of the beginning of each row in the CSR matrix UL
  \param colInd The column index of each element in the CSR matrix UL
  \param UL The values of the strictly upper triangular matrix U or strictly lower
            triangular matrix L, stored in CSR format
  \param x The solution vector
```

```
  \param alpha a scalar
  \param b The right hand side vector
*/
__global__
void csrBackSubstituteKernel( int Ik0, int Ik1, double *D,
                              int *rowPtr, int *colInd, double *UL,
                              double *x, double alpha, double *b )
{
    int i = Ik0 + blockIdx.x * blockDim.y + threadIdx.y;
    if( i >= Ik1 )
    {
        return;
    }

    __shared__ double sum[CUDA_THREADS_PER_ROW][CUDA_BLOCK_SIZE/CUDA_THREADS_PER_ROW];
    sum[threadIdx.x][threadIdx.y] = 0.0f;
    // compute, in parallel, the different sums needed for the inner products
    for( int j = rowPtr[i] + threadIdx.x; j < rowPtr[i+1]; j += blockDim.x )
    {
        sum[threadIdx.x][threadIdx.y] += UL[j] * x[colInd[j]];
    }
    __syncthreads();

    // perform the reduction to accumulate the total sums of each row in sum[0][threadIdx.y]
    for( int j = blockDim.x / 2; j > 0; j /= 2 )
    {
        if( threadIdx.x < j )
```

```
        {
            sum[threadIdx.x][threadIdx.y] += sum[threadIdx.x+j][threadIdx.y];
        }
        __syncthreads();
    }

    // update x[i]
    if( 0 == threadIdx.x )
    {
        x[i] = ( alpha * b[i] - sum[0][threadIdx.y] ) / D[i];
    }
    return;
}

/*!
  \brief Solves triangular system (D+L)*x=alpha*b or (D+U)*x=alpha*b for x.
         D is a diagonal matrix, L & U are strictly lower & upper triangular
         respectively. The system should be pre analyzed & divided into contiguous
         sections that can be solved in parallel

  \param colorPtr An array of size N+1 containing the index of the
                  first row in each color
  \param N The number of colors
  \param upperTriangular True if solving (D+U)*x=alpha*b, false otherwise
  \param D The elements of the diagonal matrix D
  \param rowPtr The index of the beginning of each row in the CSR matrix UL
  \param colInd The column index of each element in the CSR matrix UL
```

```
  \param UL The values of the strictly upper triangular matrix U or strictly lower
            triangular matrix L, stored in CSR format
  \param x The solution vector
  \param alpha a scalar
  \param b The right hand side vector
*/
void
csrTriSolve( int *colorPtr, int N,
             bool upperTriangular, double *D,
             int *rowPtr, int *colInd, double *UL,
             double *x, double alpha, double *b )
{
    for( int k = 0; k < N; ++k )
    {
        int Ik0 = upperTriangular ? colorPtr[N-k-1] : colorPtr[k];
        int Ik1 = upperTriangular ? colorPtr[N-k] : colorPtr[k+1];
        dim3 blockDim( CUDA_THREADS_PER_ROW, CUDA_BLOCK_SIZE / CUDA_THREADS_PER_ROW );
        int gridDim = ( Ik1 - Ik0 ) / blockDim.y;
        if( ( Ik1 - Ik0 ) % blockDim.y != 0 )
        {
            ++gridDim;
        }
        csrBackSubstituteKernel<<<gridDim,blockDim>>>( Ik0, Ik1, D, rowPtr, colInd, UL, x, alpha, b );
        cudaThreadSynchronize();
    }
    return;
}
```

Bibliography

[1] http://top500.org.

[2] Bell, N. and M. Garland (2009) "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *In SC 2009: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*.

[3] Bordawekar, R. and M. M. Baskaran (2008) "Optimizing Sparse Matrix-Vector Multiplication on GPUs," *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*.

[4] Matam, K. K. and K. Kothapalli (2011) "Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU." in *ICPP* (G. R. Gao and Y.-C. Tseng, eds.), IEEE, pp. 612–621.

[5] Naumov, M. (2011), "Incompete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS," .

[6] CUDA TOOLKIT Version 4.0.

[7] Schreiber, R. and W. P. Tang (1982) "Vectorizing the conjugate gradient method," *Symposium on CYBER 205 Applications*.

[8] Jones, M. T. and P. E. Plassman (1994) "Scalable iterative solution of sparse linear systems," *Parallel Computing*, **20**, pp. 753–773.

[9] Raghavan, P. (1998) "Efficient Parallel Triangular Solution Using Selective Inversion," *Parallel Processing Letters*, **8**(1), pp. 29–40.
URL \url{http://www.cse.psu.edu/~raghavan/Papers/solve1.ps}

[10] Alvarado, F. L., A. Pothen, and R. Schreiber (1994) *Highly Parallel Sparse Triangular Solution*, *Tech. rep.*, Old Dominion University, Norfolk, VA, USA.

[11] Suchoski, B., C. Severn, M. Shantharam, and P. Raghavan (2012) "Adapting Sparse Triangular Solution to GPU," *International Workshop on Unconventional Cluster Architectures and Applications*.

[12] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Whitepaper.

[13] Heath, M. T. (1996) *Scientific Computing: An Introductory Survey*, 2nd ed., McGraw-Hill Higher Education.

[14] Davis, T. A. (1994) "University of Florida Sparse Matrix Collection," *NA Digest*, **92**.

[15] Lion-GA Cluster at Penn State.

[16] Siek, J. G., L.-Q. Lee, and A. Lumsdaine (2001) *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*, Addison-Wesley Professional.

[17] Little, J. D. C. and S. C. Graves (2008) "Little's Law," in *Building Intuition: Insights from Basic Operations Management Models and Principles*, Springer Science+Business Media, pp. 81–100.

[18] Wang, P. (2011) "Analysis-Driven Optimizations in CUDA," in *GPU Technology Conference, Asia*.
URL `developer.download.nvidia.com/GTC/PDF/1082\_Wang.pdf`