Fast Tridiagonal Solvers on the GPU

Yao Zhang
University of California, Davis
yaozhang@ucdavis.edu

Jonathan Cohen
NVIDIA
jocohen@nvidia.com

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

Abstract

We study the performance of three parallel algorithms and their hybrid variants for solving tridiagonal linear systems on a GPU: cyclic reduction (CR), parallel cyclic reduction (PCR) and recursive doubling (RD). We develop an approach to measure, analyze, and optimize the performance of GPU programs in terms of memory access, computation, and control overhead. We find that CR enjoys linear algorithm complexity but suffers from more algorithmic steps and bank conflicts, while PCR and RD have fewer algorithmic steps but do more work each step. To combine the benefits of the basic algorithms, we propose hybrid CR+PCR and CR+RD algorithms, which improve the performance of PCR, RD and CR by 21%, 31% and 61% respectively. Our GPU solvers achieve up to a 28x speedup over a sequential LAPACK solver, and a 12x speedup over a multi-threaded CPU solver.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming–Parallel programming

General Terms Algorithms, Measurement, Performance

Keywords Tridiagonal Linear System, GPGPU, Performance Optimization

1. Introduction

In this work, we study the performance of tridiagonal linear system solvers on a GPU. We use this study to develop an approach to measure, analyze, and optimize the performance of GPU programs. A key observation is that performance is determined by a composition of several factors including global/shared memory access, bank conflicts, computational complexity, and the overhead for synchronization and control. For different algorithms, the weight of each factor's impact on performance is different. This observation motivates a comprehensive performance analysis approach, in which we measure each factor's impact on overall performance, optimize the most critical factors, and choose the right tradeoffs between these factors to improve the performance most effectively. This is in contrast to a traditional bottleneck GPU performance model, in which only the improvement on the bottleneck leads to overall performance increase.

By applying the above analysis approach to optimizing tridiagonal solvers, we are able to achieve a significant speedup with hybrid algorithms that we would not have achieved if we stayed with the basic algorithms and just tried to remove their bottlenecks. We characterize the costs of the different algorithms with respect to computational complexity, number of steps, bank conflicts, vector hardware utilization, and other factors. By using this approach, we are able to find a combination of the basic algorithms that minimizes the total costs. Our hybrid algorithms switch between basic algorithms once the execution reaches a stage where a different approach will be cheaper.

We make the following contributions in this paper. First, we develop an approach to measure, analyze, and optimize the performance of GPU programs. Second, using our analysis approach, we perform a comprehensive study of three parallel tridiagonal algorithms and their variants on a modern GPU. We propose hybrid CR+PCR and CR+RD algorithms and show how they achieve better performance by reducing the number of inefficient algorithmic steps, avoiding bank conflicts, and improving the utilization of vector hardware. The hybrid algorithms improve PCR, RD and CR by 21%, 31% and 61% respectively, and achieve a 12.5x speedup over a multi-threaded CPU solver. Third, we perform experiments to determine the accuracy of various algorithms and give suggestions for improving numerical stability. Fourth, we discuss insights we gained into GPU optimization strategies while developing these algorithms. We find that bank conflicts can have a severe impact on performance for the reduction communication pattern. We also find that each algorithmic step includes a considerable amount of overhead due to the synchronization and control.

Fast solutions to a tridiagonal system of linear equations are critical for many scientific and engineering problems, as well as real-time or interactive applications in computer graphics, video games and animation films [19, 20, 25]. The applications of tridiagonal solvers include alternating direction implicit (ADI) methods [15, 19, 25], spectral Poisson solvers [16], cubic spline approximations, numerical ocean models [13], semi-coarsening for multi-grid solvers [24] and preconditioners for iterative linear solvers [12].

Since the 1960s, a variety of parallel algorithms have been developed for solving tridiagonal systems. Notable among these are cyclic reduction [6, 16], recursive doubling [27], and partition methods [22, 32]. These algorithms have typically targeted vector supercomputers such as the Illiac IV, CDC STAR-100 and Cray-1. As alternatives to the vector programming model, tridiagonal solvers have been studied with a message-passing programming model on the Intel iPSC/1, nCUBE-1, Cray T3E, and a 500-node IBM cluster [3, 10, 15, 28, 29]. Huang et al. and Climent et al. studied tridiagonal solvers with the BSP (bulk synchronous parallel) programming model on Sun workstations and the IBM SP2 [7, 18].

Since 2002, the GPU has evolved from a graphics-specific accelerator to a general-purpose computing vector processor. The GPU is different from older vector computers mainly in two aspects: (1) it has a hierarchical architecture, and (2) it virtualizes all of its processing cores. Today's GPU can provide 1 teraFLOPS of computing power in single precision, which has attracted a great deal of attention from the scientific community [1]. For example,

PPoPP'10, January 9–14, 2010, Bangalore, India.

Volkov and Demmel performed detailed benchmarking for dense linear algebra kernels on a GPU, and showed the great potential of the GPU for scientific computing [30].

The tridiagonal solver was first implemented on a GPU by Kass et al. [19] to perform efficient depth-of-field blurs. Their solver was based on cyclic reduction and was written in a GPU shading language. Sengupta et al. implemented cyclic reduction with CUDA [23], and applied it to real-time shallow water simulation [20, 25]. Volkov and Demmel accelerated the bisection algorithm on a GPU for finding eigenvalues of symmetric tridiagonal matrices [31]. While all previous GPU tridiagonal solvers that we are aware of use cyclic reduction, we identify four other techniques that have superior performance on a current generation GPU, including parallel cyclic reduction, recursive doubling, and their hybrid variants. In this paper, we focus on the problem of solving a large number of small tridiagonal systems in parallel, which maps well to the hierarchical and throughput-oriented architecture of the GPU. This is a common application scenario arising in ADI methods [15, 19, 25], spectral Poisson solvers [16], numerical ocean models [13], and semi-coarsening for multi-grid solvers [24], etc. For both the GPU and CPU solvers, we choose to use single-precision floating point arithmetic, because today's GPU features substantially more single-precision throughput than double-precision, and single precision is generally sufficient for the real-time/interactive computer graphics applications and many of the particular scientific and engineering problems that we cite above. The performance analysis techniques we developed would apply equally well to double-precision solvers and will be more interesting as GPU double-precision performance becomes more competitive with single-precision performance.

The rest of the paper is organized as follows. Section 2 introduces the basics of the CR, PCR, and RD algorithms. Section 3 presents the hybrid algorithms we propose for GPU implementation. Section 4 describes the GPU implementation. Section 5 discusses algorithm complexity, presents our analysis and optimization approach, and shows the experimental results and analysis. Section 6 concludes and describes future work.

2. Preliminaries

We wish to solve a system of n linear equations of the form $Ax = d$, where A is a tridiagonal matrix

$$A = \begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & 0 & & \\ & a_3 & b_3 & c_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & c_{n-1} & \\ & 0 & & & a_n & b_n & \end{pmatrix}.$$

The classic algorithm to solve such a system is the Thomas algorithm, which is Gaussian elimination in the tridiagonal matrix case. The algorithm has two phases, forward elimination and backward substitution. In the first phase, we eliminate the lower diagonal by

$$c'_1 = \frac{c_1}{b_1}, c'_i = \frac{c_i}{b_i - c'_{i-1}a_i}, i = 2, 3, \ldots, n-1$$

$$d'_1 = \frac{d_1}{b_1}, d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}, i = 2, 3, \ldots, n-1$$

The second phase solves all unknowns from last to first:

$$x_n = d'_n, x_i = d'_i - c'_i x_{i+1}.$$

The algorithm is simple, but inherently serial and takes $2n$ computation steps, because the calculation of c'_i, d'_i, and x_i depends on the result of the immediately preceding calculation of c'_{i-1}, d'_{i-1}, and x_{i+1}. In the following subsections, we introduce three parallel algorithms for solving a tridiagonal system. In general, these methods trade more work for fewer steps and are thus better suited for parallel implementation.

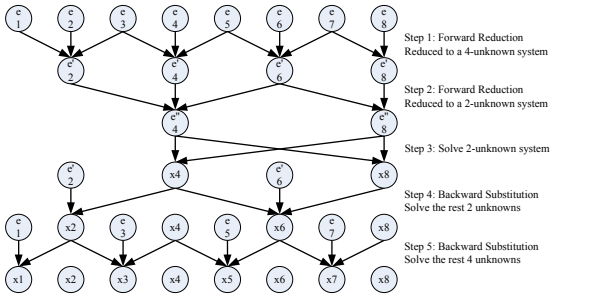

Figure 1. Communication pattern for CR in the 8-unknown case, showing the dataflow between each equation, labeled $e1$ to $e8$. Letters e' and e'' stand for updated equation.

2.1 Cyclic Reduction (CR)

Cyclic reduction was proposed by Hockney [16] and is the only previously implemented algorithm on a GPU. CR consists of two phases, forward reduction and backward substitution. The forward reduction phase successively reduces a system to a smaller system with half the number of unknowns, until a system of 2 unknowns is reached. The backward substitution phase successively determines the other half of the unknowns using the previously solved values.

In each step of forward reduction, we update all even-indexed equations in parallel with equation i of the current system as a linear combination of equations $i, i+1$ and $i-1$, so that we derive a system of only even-indexed unknowns. Equation i has the form $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$. The updated values of a_i, b_i, c_i and d_i are

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2$$

$$c'_i = -c_{i+1}k_2, d'_i = d_i - d_{i-1}k_1 - d_{i+1}k_2$$

$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}$$

In each step of backward substitution, we solve all odd-indexed unknowns x_i in parallel by substituting the already solved x_{i-1} and x_{i+1} values to equation i,

$$x_i = \frac{d'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}.$$

Note that for the sake of simplicity, in the above description, we disregard the special treatment of the last equation and the first unknown respectively in the two algorithm phases. Also, we solve a 2-unknown system between the two algorithm phases. Figure 1 shows the communication pattern of the algorithm for an 8-unknown system.

Both the parallel CR algorithm and the serial Thomas algorithm perform a number of operations that are linear in the number of unknowns. The Thomas algorithm performs $8n$ operations while CR performs $17n$ operations. However, on a parallel computer with n processors, CR requires $2\log_2 n - 1$ steps while the Thomas algorithm requires $2n$ steps.

2.2 Parallel Cyclic Reduction (PCR)

PCR [17] is a variant of CR. In contrast to CR, PCR only has the forward reduction phase. Although the reduction mechanism and formula are the same as those of CR, in each reduction step, PCR reduces each of the current systems to two systems of half size. For

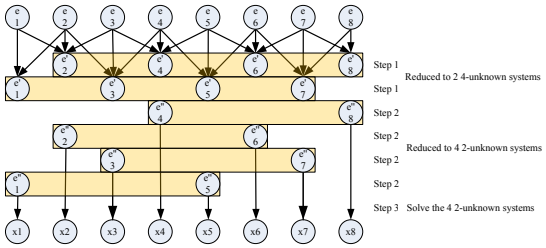

Figure 2. Communication pattern for PCR in the 8-unknown case, showing the dataflow between each equation, labeled $e1$ to $e8$. Letters e' and e'' stand for updated equations. Equations in a yellow rectangle form a system. We omit the arrows in step 2 for clarity.

example, for an 8-unknown system, we reduce it to two 4-unknown systems in step 1 (see Figure 2), then further reduce the two 4-unknown systems to four 2-unknown systems in step 2, and finally solve the four systems in step 3. PCR takes $12n\log_2 n$ operations and $\log_2 n$ steps to finish. PCR requires fewer algorithmic steps than CR but does asymptotically more work per step.

2.3 Recursive Doubling (RD)

Stone [27] proposed the RD algorithm. Egecioglu et al. [10] reformulated the algorithm in the scan (or prefix sum) form. Scan is a parallel primitive originally developed for vector machines [5, 14] that can be efficiently implemented on a GPU [8, 25]. A variety of algorithms have been developed to perform a scan. We choose the algorithm by Hillis and Steele [14] because we need a step-efficient algorithm, as will be explained in Section 3.

The basic idea of RD is to express unknowns as the multiplication of a chain of matrices that can be evaluated in parallel using the scan primitive. Figure 3 shows the communication pattern of RD. We first build matrices B_i, $1 \leq i \leq n$,

$$B_i = \begin{pmatrix} -\frac{b_i}{c_i} & -\frac{a_i}{c_i} & \frac{d_i}{c_i} \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Next we perform a scan operation on B_i, the output of which is a n-component vector with each component as the multiplication of a matrix chain,

$$\begin{aligned} &\text{scan}(B_1, B_1, \ldots, \ldots, B_{n-1}, B_n) \\ &\quad = [B_1 \;\; B_2B_1 \;\; \ldots \;\; \ldots \;\; B_{n-1}\ldots B_2B_1 \;\; B_n\ldots B_2B_1] \\ &\quad = [C_1 \;\; \ldots \;\; \ldots \;\; C_{n-1} \;\; C_n]. \end{aligned}$$

Then we have $x_1 = -\frac{c_{13}}{c_{11}}$, where

$$C_n = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ 0 & 0 & 1 \end{pmatrix}.$$

Finally, all unknowns can be evaluated as, $[X_2 \;\; X_3 \;\; \ldots \;\; X_n]$
$= X_1[C_1 \;\; C_2 \;\; \ldots \;\; C_n]$
$= [X_1C_1 \;\; X_1C_2 \;\; \ldots \;\; X_1C_n]$ where $X_i = \begin{pmatrix} x_i \\ x_{i-1} \\ 1 \end{pmatrix}$.

Our step-efficient version of RD takes $20n\log_2 n$ operations and $\log_2 n + 2$ steps to finish.

3. Hybrid Algorithms

We will use CR, PCR, and RD as basic building blocks from which to build more complex and efficient algorithms. We start

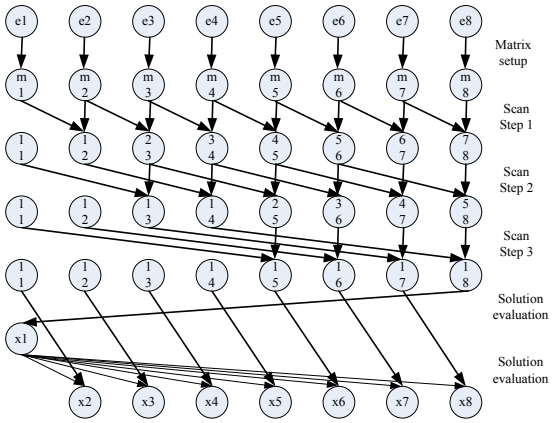

Figure 3. Communication pattern for RD in the 8-unknown case, showing the dataflow between each equation, labeled $e1$ to $e8$. Letter m stands for matrix. During the scan phase, a pair of two numbers (a, b) in a circle stands for $\prod_{i=a}^{b} m_i$.

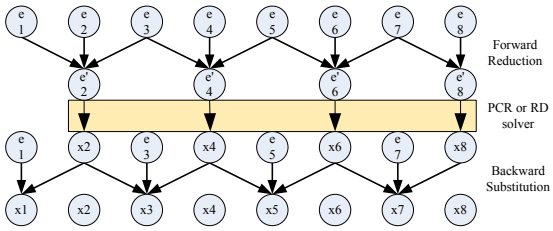

Figure 4. Hybrid algorithms for an 8-known system. Letter e stands for equation, and e' stands for updated equation.

with these three algorithms because they have fine-grained parallel structures, which are suitable for GPU implementation where a large number of lightweight threads are available. Other parallel approaches, such as the sub-structuring method [32] and two-way Gaussian elimination [15], are coarse-grained methods that map larger amounts of work per thread. These methods would be more suitable to a multi-core CPU.

With respect to computational complexity, CR is the best algorithm because it is $O(n)$, while PCR and RD are $O(n \log_2 n)$. However, CR suffers from a lack of parallelism at the end of the forward reduction phase and at the beginning of the backward substitution phase (see Figure 1); on the other hand, although PCR and RD have fewer algorithmic steps, they always have more parallelism through all steps (see Figure 2 and 3). These two basic observations lead us to develop hybrid methods on a GPU. The hybrid methods improve CR by switching to PCR or RD to reduce inefficient steps when there is not enough parallelism to keep a GPU busy. Sengupta et al. use a similar idea to combine a step-efficient scan algorithm with a work-efficient one for implementation on a GPU [26].

The hybrid algorithms first reduce the system to a certain size using the forward reduction phase of CR, then solve the reduced (intermediate) system with the PCR/RD algorithm. Finally, they substitute the solved unknowns back into the original systems using the backward substitution phase of CR. Figure 4 shows the hybrid method schematically. In this example, we switch to PCR or RD before the forward reduction has reached a 2-unknown system, as opposed to the CR case in Figure 1. PCR and RD enable us to finish the inefficient middle steps more quickly, because they have fewer algorithmic steps than CR. In section 5, we perform

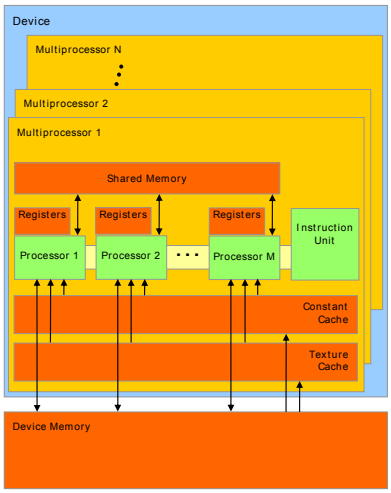

Figure 5. GPU architecture (from CUDA Programming Guide [2])

a quantitative analysis on the complexity of the basic algorithms, hybrid CR+PCR, and CR+RD algorithms on the GPU.

Note that while the algorithms are performed on NVIDIA hardware in this work, the motivation behind the hybrid algorithms is to balance work-efficiency against step-efficiency, which will be an issue on any vector architecture. We are unaware of any research on hybrid solvers that combines CR with PCR/RD. Other hybrid approaches have been studied in a CPU cluster environment. Johnsson and Ho studied hybrid solvers using sub-structuring method, Gaussian elimination and cyclic reduction with the goal of minimizing overall computation and communication costs [15]. Sun and Zhang proposed a two-level hybrid method with a first-level solver based on the Sherman-Morrison modification formula and an application-dependent second-level solver [29].

4. GPU Implementation

We implement the five tridiagonal solvers on an NVIDIA GTX 280 GPU using the CUDA programming model [23]. GTX 280 is a hierarchical shared-memory vector architecture as shown in Figure 5. It is made up of 30 multiprocessors, and each multiprocessor contains 8 thread processors and 16 KB of on-chip shared memory. Shared memory locations are mapped to one of 16 banks with sequential 32-bit words going to sequential banks. Therefore, if multiple threads in a warp (described below) access memory locations 16 words apart, the memory accesses will serialize. See the programming guide [2] for more details.

CUDA virtualizes multiprocessors as blocks and processors as threads, which enables programmers to run thousands of threads and blocks across different generations of GPUs regardless of the number of physical processors. A key concept of CUDA programming model is the *warp*. A warp is a group of 32 threads that execute in lockstep in a SIMD fashion. Because the GPU architecture shares a single instruction unit for all threads in a warp, a warp is the smallest unit of work a GPU issues. This means if we have less than 32-way parallelism, we will still have 32 threads running, some of which will not do useful work. This observation is particularly important in reduction pattern algorithms such as tridiagonal solvers which have stages of less than 32-way parallelism.

To solve hundreds of tridiagonal systems simultaneously, we map the solvers to the GPU's two-level hierarchical architecture with systems mapped to blocks and equations mapped to threads. The number of systems we solve is far larger than the number of multiprocessors, so that all multiprocessors are fully utilized and the context switching between systems helps hide the memory latency. The total storage consists of five arrays: three for the matrix diagonals, one for the right-hand side, and one for the solution vector. These five arrays store the data of all systems continuously, with the data of the first system stored at the beginning of the arrays, followed by the second system, the third system, and so on. For each system, we load the three diagonals and right-hand side from global memory to shared memory, solve the system, and store the solution back to global memory. Therefore global memory communication only occurs at the beginning and end of all algorithms. With current hardware, systems of more than 512 equations would exceed the size of shared memory. Our solvers do support this case at a cost of roughly 3x performance degradation by using global memory only. While small systems (up to 512x512) that run at interactive rates are most common in the graphics community, we could possibly better accelerate larger systems with several first steps (as in CR) that reduces the system to fit into shared memory. We anticipate the most common use of the solvers we have developed is as a component of a larger computation on a GPU, which amortizes the cost of any CPU-GPU data transfers (see section 5.2).

In CR, we start with a total thread number equal to half the number of equations, reduce the number of active threads by half after each step of forward reduction, and double the number of active threads before each step of backward substitution (see Figure 1). We always use contiguously ordered threads as active threads so that we do not have unnecessary divergent branches within a SIMD unit. In PCR, the number of active threads is constant and equal to the number of equations across all steps (see Figure 2). In RD, the number of active threads is set to the number of equations initially, and then gradually reduced to half during the major algorithm phase scan (see Figure 3). As with CR, we always use a contiguous chunk of threads as active threads so that we do not have divergent branches in this case either.

In all three solvers, we keep data in-place during the entire solution as shown in Figure 1, 2 and 3. This means for the CR and PCR solvers, we store the reduced systems (updated equations) at the same memory location; for the RD solver, we perform scan in-place. The advantage of an in-place approach is that we save shared memory space so that we can fit multiple blocks running simultaneously on one multiprocessor. The disadvantage is, for the CR solver, we have more and more bank conflicts towards the end of forward reduction and at the beginning of backward substitution. However, in-place PCR and RD do not suffer from bank conflicts. For the hybrid solvers, we copy the data from the intermediate system to another five arrays in shared memory. The copy takes little time and extra storage space for the intermediate system, but makes the solver more modular, because we can directly plug the PCR or RD solver into the intermediate system.

Although all parallel algorithms are general for any system size, our solvers only handle a power-of-two system size, which makes thread numbering and address calculation simpler. In the RD solver, the 3x3 matrices on which we perform scan are special matrices, which enable us to only store the first two rows of matrices and save several floating point operations.

5. Performance Evaluation

In this section, we first summarize the algorithms' complexity on the GPU, then we present the performance results and analysis for each algorithm, and finally we compare the accuracy of all CPU and GPU solvers.

5.1 Characterizing Algorithm Complexity

Table 1 summarizes the computation and communication cost of all algorithms. For all solvers, the global memory communication happens only twice for reading input data and writing output results, as

we have discussed in section 4, and most communication happens between the processing cores and shared memory. As shown in Table 1, CR has the least work (shared memory accesses and arithmetic operations) but the most steps, whereas PCR and RD have fewer steps but more work. This motivates an approach that takes advantage of the best parts of both: doing the least work when there is sufficient parallelism at least of warp size, but then switching to performing fewer steps when there is not enough parallelism to fill the machine. The switch is actually even more beneficial because there are bank conflicts in the CR solver and shared memory access dominates the execution time. Because of bank conflicts, we find that the width at which it is best to switch is actually far larger than the warp size. We will discuss this issue further in section 5.3.

5.2 Performance Results

We test the performance using a 2.5 GHz Intel Core 2 Q9300 quad-core CPU, a GTX 280 graphics card with 1 GB video memory, CUDA 2.0 and the CentOS 5 Linux operating system. The GTX 280 card is an instance of the GT200 architecture with 30 multiprocessors, each with 8 thread processors.

We measure performance using runtime, instead of FLOPS, because different algorithms use different FLOP counts. An analysis of performance in terms of FLOPS for each algorithm can be found in Section 5.3. Figure 6 shows the performance comparison of five GPU solvers for two cases, with and without the time for the CPU-GPU data transfer over the PCI-Express bus. The problem sizes we choose range from 64 64-unknown systems to 512 512-unknown systems. The hybrid solvers outperform other solvers clearly for the problem size 512x512, but perform worse than RD and PCR for the 64x64 and 128x128 cases. This is because for a smaller system size, hybrid solvers introduce extra CR steps to PCR or RD, which offsets the benefits of solving a smaller intermediate system. We will discuss this in detail in section 5.3.4.

If the CPU-GPU data transfer time is included, since the CPU-GPU data transfer dominates the entire solution time by 90–95%, all solvers have similar performance as shown in Figure 6 (right). Whether or not we should include the cost of the CPU-GPU data transfer depends on the application scenario. Transfer time can be ignored if the GPU solver is used locally as a part of a GPU program (for example, GPU fluid simulation [25] and depth-of-field effects [19]). It should be considered if the GPU solver is used as an accelerator for a part of a CPU program.

Figure 7 shows the performance comparison between the best GPU solver and several CPU solvers. The CPU solvers include a Gaussian elimination tridiagonal solver without pivoting (GE), a multi-threaded GE solver (MT), and a Gaussian elimination tridiagonal solver with pivoting (GEP). The MT solver is an OpenMP implementation developed by us with multiple threads solving multiple systems simultaneously. We use four threads for the MT solver with each thread running on one CPU core. We note that the problem size needs to be large for the MT solver to outperform a single-threaded solver. The GEP solver is from LAPACK [4].

With and without the CPU-GPU data transfer time, the best speedups we get are 17.2x and 1.5x respectively. If the GPU solver is used as an accelerator for a part of CPU program, the PCI-Express bus is certainly the bottleneck. However, the trend is that the GPU and CPU are becoming increasingly coupled and ultimately may be integrated on the same chip in the future. Notice that when the problem size increases by 4 times from size 64x64 to 128x128 (or from 128x128 to 256x256), the runtime favorably increases far less than 4 times. This is because the GPU prefers large amounts of parallelism to take advantage of its parallel hardware resources and ample bandwidth. The relative performance on the 512x512 problem size is not as high as the 256x256 problem size because the system size is too large to fit multiple blocks running simultaneously on a GPU multiprocessor, which hurts the performance. Running multiple blocks simultaneously enables the GPU to switch between blocks, overlap the computation and data transfer, and thus improve the hardware utilization. The number of concurrent blocks depends on the GPU hardware resources (register count, shared memory size, and maximum number of active warps, etc) and a program's requirements on these resources per block.

5.3 Performance Analysis

We analyzed each GPU solver by measuring the time spent on each part of the algorithm. We find that the overall execution time depends primarily on the number of algorithmic steps rather than the amount of work per step. In other words, for this problem on the GPU, step-efficient algorithms are preferable to work-efficient algorithms. We use a differential method to measure the time for each part of the algorithm. We first comment out the whole code, then uncomment it incrementally in program order and measure execution time. Finally, we calculate the time difference between all neighboring timing results. For every algorithmic step in a loop, we exit the loop early at that step to measure the time spent until that step. Commenting out part of the code does not affect the number of concurrent blocks, because we have the same shared memory usage and the number of concurrent blocks is limited by the shared memory size rather than register usage in our case.

We also measure the execution time for global memory access, shared memory access, and computation. Since the global memory access only occurs at the beginning and end of the program, we use the same differential method to measure its time. To estimate shared memory access time, we replace all shared memory accesses with register accesses, and calculate the shared memory access time as the time difference between this program and the original program. The estimated time might be smaller than the actual time because shared memory accesses could be overlapped with computation. We estimate computation time as the total time minus global memory and shared memory access time. Control and synchronization overhead is included in the computation time. Bank conflicts in shared memory access turn out to have a significant impact on performance.

5.3.1 Cyclic Reduction

Recall that cyclic reduction has two phases, forward reduction and backward substitution. Each phase consists of $\log_2 n - 1$ steps where n is the system size. Figure 8 shows the absolute time and percentage time for all solution phases as well as their internal steps. Forward reduction takes about twice as much time as backward substitution, since it requires more computations and memory accesses.

At first glance, the timing for the steps in forward reduction seems quite surprising. Since we reduce the amount of work by half each step, we expect the time for each step to decrease as the algorithm progresses. However, the measured step time does not decrease but rather increases. Further investigation reveals that this is caused by bank conflicts. Since the shared memory access stride is doubled in each step, there are more and more bank conflicts. One method to avoid bank conflicts is to store the even-indexed and odd-indexed equations of all reduced systems separately, at the cost of extra shared memory usage and more complicated addressing[1].

Figure 9 shows the time for each of the 8 steps in the forward reduction phase. For comparison, we measured the time for the same program modified to enforce a shared memory access stride of one so that it is bank-conflict-free. This results in an incorrect

[1] Independently from our work, Göddeke and Strzodka proposed the same technique, and showed that it achieves similar performance as our hybrid CR+PCR solver, at the cost of 50% more shared memory usage [11].

Table 1. Complexity comparison of algorithms. n is the system size. m is the intermediate system size. We assume n and m are powers of 2.

Algorithm	Shared memory accesses	Arithmetic operations	Algorithmic Steps	Global memory accesses
CR	$23n$	$17n$, of which $3n$ are div	$2\log_2 n - 1$	$5n$
PCR	$16n \log_2 n$	$12n \log_2 n$, of which $2n \log_2 n$ are div	$\log_2 n$	$5n$
RD	$32n \log_2 n$	$20n \log_2 n$; no div in major step scan	$\log_2 n + 2$	$5n$
CR+PCR	$23(n-m) + 16m \log_2 m$	$17(n-m) + 12m \log_2 m$	$2\log_2 n - \log_2 m - 1$	$5n$
CR+RD	$23(n-m) + 32m \log_2 m$	$17(n-m) + 20m \log_2 m$	$2\log_2 n - \log_2 m + 1$	$5n$

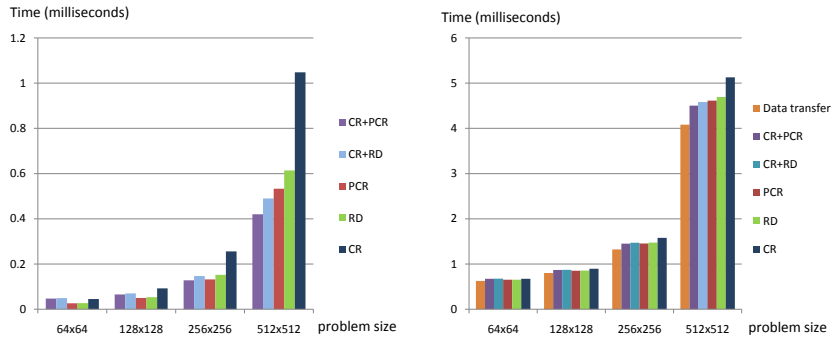

Figure 6. Performance comparison of five GPU solvers. Left: without the CPU-GPU data transfer time. Right: with the CPU-GPU data transfer time.

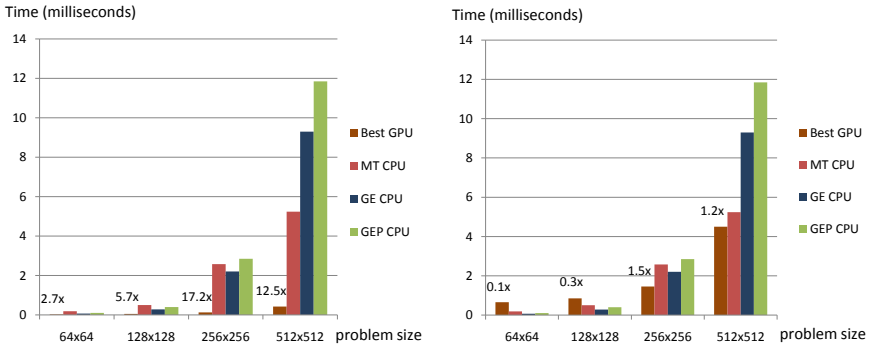

Figure 7. Performance comparison between the GPU and CPU solvers. The speedup numbers are comparisons between the best GPU and the best CPU performance. Left: without the CPU-GPU data transfer time. Right: with the CPU-GPU data transfer time.

algorithm, but is for timing comparison only. Figure 9 shows that the overall penalty can be as high as 4.8x in the case of 16-way bank conflicts. This factor includes the penalty for shared memory access together with the solver computation. The penalty for only shared memory access should be even higher, as will be shown in section 5.3.2. For the bank-conflict-free case, once the algorithm reaches 32 active threads, the time per step remains almost constant for two reasons: (1) a warp is the smallest unit of work on the GPU, and (2) a large portion of the total step time is taken by the overhead of synchronization and loop control. For the with-bank-conflicts case, after the algorithm reaches 32 active threads, the time per step continues to decrease, because all conflicting accesses to shared memory serialize and thus each step runs faster as fewer threads access shared memory.

Figure 10 shows the time breakdown for global/shared memory access and computation. Shared memory accesses dominate the total execution time due to bank conflicts. The global memory accesses are contiguous (coalesced) and reach a bandwidth of 48.5 GB/s. We found that the global memory to register bandwidth could be 25–50% higher than the global memory to shared memory bandwidth, but in our case, we need the latter. Shared memory accesses suffer from bank conflicts and reach a bandwidth of 33 GB/s (this does not mean shared memory does not help, because the stride access pattern is bad for global memory access too). The computation rate is 15.5 GFLOPS, far below the theoretical peak, mainly because of low vector hardware utilization during the end of forward reduction and the beginning of backward substitution, as well as other factors including a large number of expensive division operations, synchronizations, and control logic.

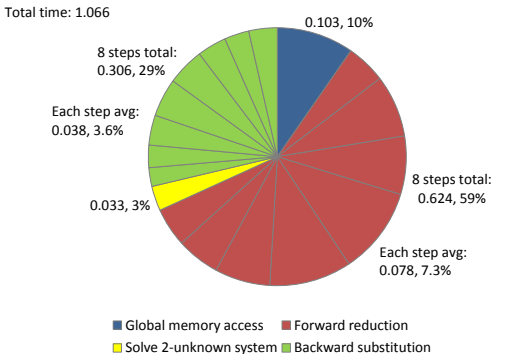

Figure 8. Time breakdown of CR for problem size 512x512. Timings are in milliseconds.

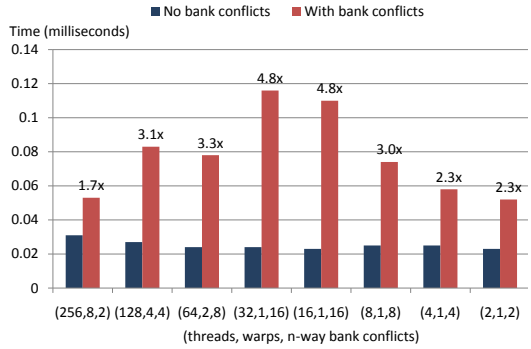

Figure 9. Bank conflicts' impact on performance in forward reduction phase for problem size 512x512.

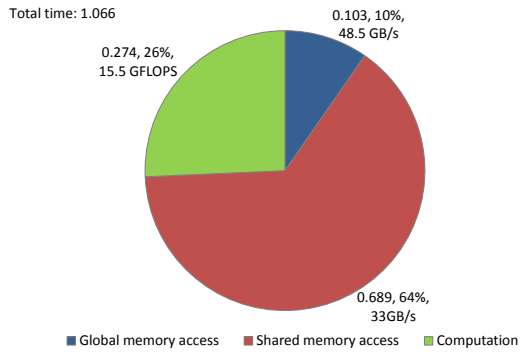

Figure 10. Time breakdown for CR in terms of global/shared memory access and computation for problem size 512x512. Timings are in milliseconds.

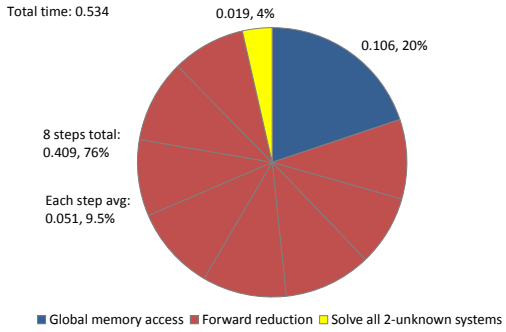

Figure 11. Time breakdown of PCR for problem size 512x512. Timings are in milliseconds.

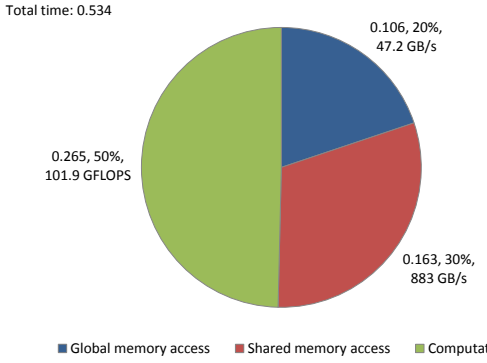

Figure 12. Time breakdown for PCR in terms of global/shared memory access and computation for problem size 512x512. Timings are in milliseconds.

5.3.2 Parallel Cyclic Reduction

PCR takes about half the time as CR, mainly because PCR only has the forward reduction phase. Furthermore, although PCR does more work during each forward reduction step than CR, the average step time is less than that of CR as shown in Figure 8 and 11, because PCR is free of bank conflicts. Only 30% of the total time is spent on shared memory access compared to 64% in the CR case (see Figure 10 and 12). The achieved shared memory bandwidth is 883 GB/s, 26 times the bandwidth achieved in the CR case. The factor of 26 is due to two reasons: the large penalty of bank conflicts in CR, and the low vector load/store utilization when the thread count is less than the load/store width of 16. PCR's 101.9 GFLOPS computation rate is also higher than CR's 15.5 GFLOPS (see Figure 10 and 12) because of PCR's high vector hardware utilization. We should note that the comparison of the computation rate and sustained bandwidth help us understand the interaction between algorithms and GPU architecture and the algorithms' efficiency on GPU, but cannot serve as a measure to compare the performance of algorithms, because different algorithms may have different complexity in memory access and computation. For example, CR achieves a lower computation rate, but it also performs fewer computations than PCR. In comparing the performance of different algorithms, the only thing that matters is the final timing result.

5.3.3 Recursive Doubling

RD takes slightly more time than PCR. The average per-step time of RD is similar to PCR but RD has two more step than PCR. Both RD and PCR are free of bank conflicts. The shared memory access time of RD is 1.6 times that of PCR (see Figure 12 and 14), which

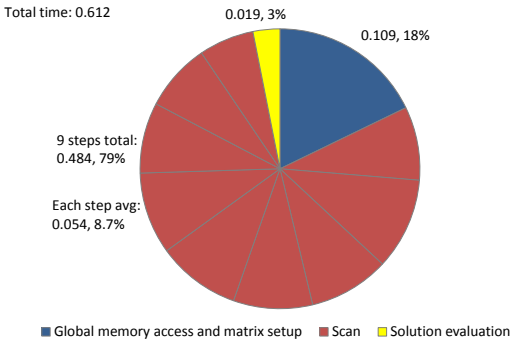

Figure 13. Time breakdown of RD for problem size 512x512. Timings are in milliseconds.

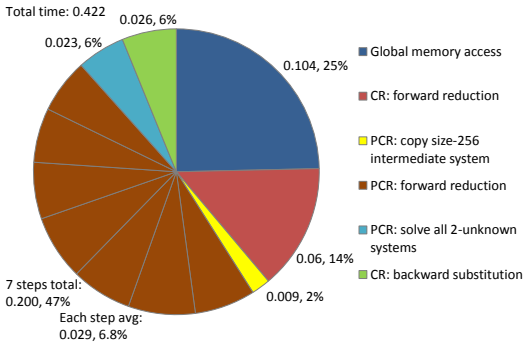

Figure 15. Time breakdown of CR+PCR for problem size 512x512. Timings are in milliseconds.

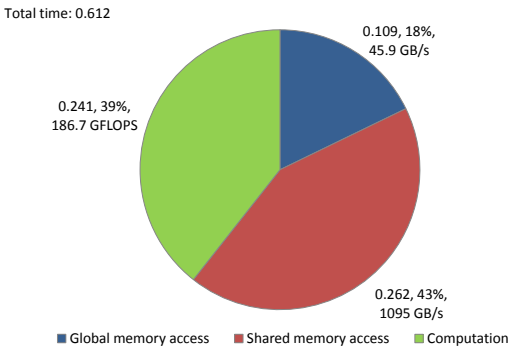

Figure 14. Time breakdown for RD in terms of global/shared memory access and computation for problem size 512x512. Timings are in milliseconds.

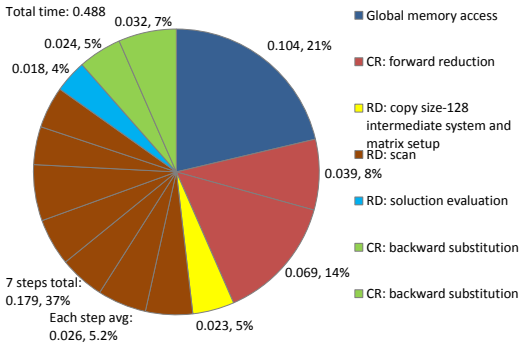

Figure 16. Time breakdown of CR+RD for problem size 512x512. Timings are in milliseconds.

follows from the fact that RD has twice as many shared memory accesses as PCR (see Table 1). RD requires almost twice as many floating point operations as PCR, but almost no divisions (see Table 1), which gives RD a computation time close to that of PCR, but twice the computation rate of PCR (see Figure 12 and 14).

5.3.4 Hybrid CR+PCR

The hybrid CR+PCR solver takes 61% and 21% less time than CR and PCR respectively for the problem size 512x512. There are two reasons CR+PCR outperforms CR. First, because the intermediate system is solved by PCR which only has a forward reduction phase, CR+PCR has fewer steps and thus less control and synchronization overhead. Second, because the intermediate system is a smaller system of size 256 (plus PCR is free of bank conflicts), each step takes less time than a step in CR's forward reduction phase (see Figure 8 and 15). The second reason also explains why CR+PCR outperforms PCR. Although we add one CR forward reduction step and one backward substitution step, the size of the remaining (intermediate) system is reduced by half, and therefore takes almost half of the time per step (see Figure 11 and 15).

As we add more CR steps, we see more overhead per CR step, but with a smaller intermediate system. We experimented with different sizes of intermediate systems and found that for size-512 systems, the hybrid solver performs best with size-256 intermediate systems, as shown in Figure 17. The best switch point of size 256 is far larger than the warp size 32. This is because the switch is beneficial not only in terms of the improved vector hardware utilization and fewer steps to solve the intermediate system, but also results in fewer bank conflicts and fewer total algorithmic steps, and thus less overhead of synchronization and control (the earlier we switch to PCR/RD, the fewer total steps we have; see Table 1).

5.3.5 Hybrid CR+RD

The CR+RD solver is slightly slower than the CR+PCR solver. The principle for improving performance is the same as for the CR+PCR solver, and all the analysis in section 5.3.4 applies. One difference is that the size of the intermediate systems is 128 instead of 256 in the CR+PCR case, due to the limit of shared memory size. Since the intermediate system is smaller, the average time per step is even more reduced, at the cost of extra CR steps.

5.3.6 Summary

So far, we have analyzed and compared all five tridiagonal solvers on the GPU. We used the differential method to measure and analyze the time for each part of the algorithm, as well as the time breakdown for memory access and computation. By doing so, we have revealed the advantages and limitations of each algorithm. We found that bank conflicts and control overhead associated with each step are two factors critical to performance. To further improve the performance, we used hybrid approaches to combine the benefits of CR and PCR/RD. The hybrid solutions have fewer steps than CR, less work per step than PCR/RD, no bank conflicts in solving the intermediate system, and therefore the best performance.

We also found that the theoretical peak computing power is hard to reach due to various factors including low vector hardware utilization, bank conflicts, divisions, synchronizations, and time to access shared memory. We noticed that being able to running multiple CUDA blocks concurrently results in better performance,

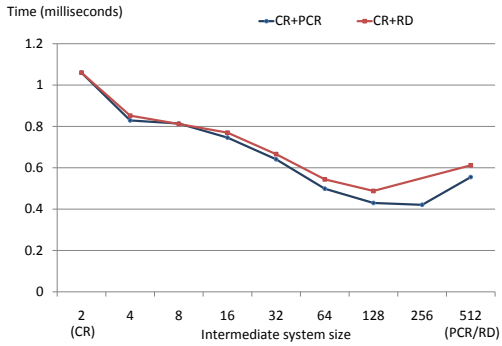

Figure 17. Timings for the hybrid solvers with various intermediate system sizes (problem size: 512x512). Endpoints mark non-hybrid implementations; each point in the middle is a hybrid implementation.

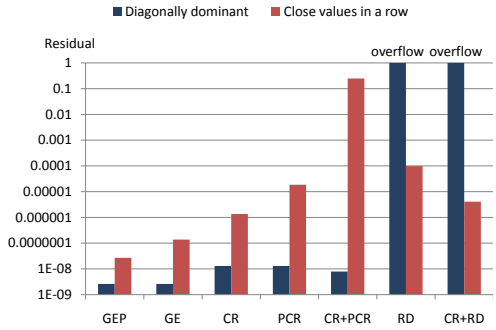

Figure 18. Accuracy comparison of CPU solvers (problem size: 512x512).

because the GPU can switch between blocks for better hardware utilization.

In contrast to a traditional view that a GPU application is either compute-bound or memory-bound, we found that the performance of this particular family of GPU solvers depends on all factors including global/shared memory access, bank conflicts, algorithmic steps, and computational complexity. This motivates a comprehensive performance analysis to reveal the factors that have the most impact on performance, and then redesign the algorithms/implementation correspondingly to minimize the total costs. The right tradeoffs between these factors are important. For example, the hybrid solvers add extra CR steps to PCR/RD, which introduces bank conflicts and control overhead, but reduce the overall amount of work and improve the hardware utilization, which results in better overall performance.

Instead of manually measuring the each factor's impact on overall performance as we have done, we see a future need to develop automatic methodologies and tools to perform performance evaluation and give programmers prioritized tasks for optimizations. We are unaware of any such of research on a GPU. In the multi-core computing domain, Williams et al. developed a model that gives programmers guidance for optimization [33], and we are currently investigating GPU-specific models that would aid in such analysis.

5.4 Accuracy Experiments

The numerical instabilities of algorithms come from multiple sources including round-off errors, arithmetic overflow, and ill-conditioned problems. All GPU solvers in this work do not include pivoting; therefore they might fail for a general tridiagonal matrix. The cyclic reduction algorithm is stable without pivoting for diagonally dominant matrices or symmetric and positive definite matrices [21]. Recursive doubling is stable for diagonally dominant matrices plus other conditions [9].

In our experiments, we found that for the systems of size larger than 64, RD favors matrices with close values in rows (thus not diagonally dominant), otherwise it might overflow. This is because during RD's matrix setup phase, the close values in rows generate matrices with elements close to 1, which help to prevent the overflow of the matrix chain multiplication in RD's scan phase.

We compare the accuracy of the CPU and GPU solvers by checking the residual of the solution, i.e., $\|Ax-b\|$. We use two sets of experiments. The first set uses diagonally dominant matrices that arise from fluid simulation [20], and the second set uses random matrices with close values in all rows. All CPU and GPU solvers use single-precision floating point arithmetic.

Figure 18 shows the results of the accuracy experiments. For the diagonally dominant case, the GPU solvers CR, PCR and CR+PCR all show good accuracy, while RD and PCR+RD suffer from arithmetic overflow. One remedy for overflow is to scale the results of matrix chain multiplication if large numbers are detected, but this method introduces a considerable amount of control overhead. For matrices with close values in rows, since the matrices could be non-diagonally-dominant, the CR, PCR and CR+PCR solvers all achieve worse accuracy; RD and CR+RD do not achieve good accuracy either but survive from overflow. GEP always has the best accuracy because it has pivoting. One future work to improve accuracy for the GPU solvers is to incorporate pivoting into these parallel algorithms.

6. Conclusion

In conclusion, we have studied five tridiagonal solvers that run on a GPU based on three algorithms, CR, PCR and RD. We develop an approach to measure, analyze and improve the performance of GPU programs in terms of memory access, computation and control overhead. By applying this approach to our GPU tridiagonal solvers, we show that hybrid algorithms have the best performance, and why. For solving 512 512-unknown systems, the hybrid solvers achieve a 12.5x speedup over the multi-threaded CPU solver, and a 28x speedup over the LAPACK solver. We recommend that the GPU solvers should be used as a part of a larger GPU application to amortize the PCI-Express transfer cost. Our cyclic reduction solvers are within an order of magnitude of the error of the LAPACK solver for diagonally dominant matrices, while recursive doubling solvers are prone to arithmetic overflow.

We performed detailed benchmarks and analysis to understand the interaction between the GPU architecture and the algorithms' memory access pattern, step count, and computational complexity. We found that an algorithm's performance is not bottlenecked by a particular source, but limited by a composition of multiple factors including global/shared memory access, bank conflicts, and computation, synchronization and control overhead. We also showed how hybrid solutions can achieve better performance by optimizing these factors.

We see several future research directions: (1) generalize the solvers for block tridiagonal matrices, (2) incorporate a pivoting strategy to GPU-based tridiagonal solvers for numerical stability, and (3) develop tools that can automatically measure various algorithm characteristics' impact on performance, and thus help programmers to optimize their GPU applications.

Acknowledgments

Thanks to Dominik Göddeke and Rajesh Bordawekar for their helpful comments on our paper draft. Thanks to Shubho Sengupta for the discussions on efficient GPU implementation of the recursive doubling algorithm. Thanks also to our funding agencies, the HP Labs Innovation Research Program, the National Science Foundation (Award 0541448), and the SciDAC Institute for Ultrascale Visualization, and to NVIDIA for equipment donations.

References

[1] General-purpose computation using graphics hardware. `http://www.gpgpu.org/`.

[2] NVIDIA CUDA compute unified device architecture, programming guide, 2009. Version 2.0.

[3] S. Allmann, T. Rauber, and G. Runger. Cyclic reduction on distributed shared memory machines. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 290–297, 2001.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 2–11. IEEE Computer Society Press, 1990.

[5] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[6] B. L. Buzbee, G. H. Golub, and C. W. Nielson. On direct methods for solving Poisson's equations. *SIAM Journal on Numerical Analysis*, 7(4):627–656, 1970.

[7] J.-J. Climent, C. Perea, L. Tortosa, and A. Zamora. An overlapped two-way method for solving tridiagonal linear systems in a bsp computer. *Applied Mathematics and Computation*, 161(2):475–500, 2005.

[8] Y. Dotsenko, N. K. Govindaraju, P.-P. J. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 205–213. ACM, June 2008.

[9] P. Dubois and G. Rodrigue. An analysis of the recursive doubling algorithm. In D. Kuck, D. Lawrie, and A. Sameh, editors, *High Speed Computer and Algorithm Organization*, pages 299–305. Academic Press, New York, NY, 1977.

[10] Ö. Eğecioğlu, C. K. Koc, and A. J. Laub. A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors. *Journal of Computational and Applied Mathematics*, 27:95–108, 1989.

[11] D. Göddeke and R. Strzodka. Accurate mixed-precision GPU-multigrid solvers on anisotropic grids. Submitted to *IEEE Transactions on Parallel and Distributed Systems, Special Issue: High Performance Computing with Accelerators*.

[12] A. Greenbaum. *Iterative Methods for Solving Linear Systems.* SIAM, Philadelphia, 1997.

[13] G. R. Halliwell. Evaluation of vertical coordinate and vertical mixing algorithms in the HYbrid-Coordinate Ocean Model (HYCOM). *Ocean Modelling*, 7:285–322, 2004.

[14] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[15] C. T. Ho and S. L. Johnsson. Optimizing tridiagonal solvers for alternating direction methods on boolean cube multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 11(3):563–592, 1990.

[16] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, 12(1):95–113, Jan. 1965.

[17] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger, Bristol, 1981.

[18] Y. Huang and W. F. McColl. Two-way BSP algorithm for tridiagonal systems. *Future Generation Computer Systems*, 13:337–347, Mar. 1998.

[19] M. Kass, A. Lefohn, and J. D. Owens. Interactive depth of field using simulated diffusion. Technical Report 06-01, Pixar Animation Studios, Jan. 2006.

[20] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 49–57, Aug. 1990.

[21] J. J. Lambiotte and R. G. Voigt. The solution of tridiagonal linear systems on the CDC STAR-100 computer. *ACM Trans. Math. Software*, 1(4):308–329, 1975.

[22] S. M. Müller and D. Sheerer. A method to parallelize tridiagonal solvers. *Parallel Computing*, 17:181–188, 1991.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue: Tomorrow's Computing Today*, 6(2):40–53, Mar. 2008.

[24] M. Prieto, R. Santiago, D. Espadas, I. M. Llorente, and F. Tirado. Parallel multigrid for anisotropic elliptic equations. *J. Parallel Distrib. Comput.*, 61(1):96–114, 2001.

[25] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, Aug. 2007.

[26] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, May 2006.

[27] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, Jan. 1973.

[28] X.-H. Sun, H. Zhang, and L. M. Ni. Efficient tridiagonal solvers on multicomputers. *IEEE Transactions on Computers*, C-41(3):286–296, Mar. 1992.

[29] X.-H. Sun and W. Zhang. A parallel two-level hybrid method for tridiagonal systems and its application to fast Poisson solvers. *IEEE Transactions on Parallel and Distributed Systems*, PDS-15(2):97–106, Feb. 2004.

[30] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, page 31 (11pp), Nov. 2008.

[31] V. Volkov and J. W. Demmel. Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. LAPACK Working Note 197, Department of Computer Science, University of Tennessee, Knoxville, Jan. 2008.

[32] H. H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Software*, 7:170–183, 1981.

[33] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.