An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU

Andrew Davidson
University of California, Davis
aaldavidson@ucdavis.edu

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

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

***Abstract*—We present a multi-stage method for solving large tridiagonal systems on the GPU. Previously large tridiagonal systems cannot be efficiently solved due to the limitation of on-chip shared memory size. We tackle this problem by splitting the systems into smaller ones and then solving them on-chip. The multi-stage characteristic of our method, together with various workloads and GPUs of different capabilities, obligates an auto-tuning strategy to carefully select the switch points between computation stages. In particular, we show two ways to effectively prune the tuning space and thus avoid an impractical exhaustive search: (1) apply algorithmic knowledge to decouple tuning parameters, and (2) estimate search starting points based on GPU architecture parameters. We demonstrate that auto-tuning is a powerful tool that improves the performance by up to 5x, saves 17% and 32% of execution time on average respectively over static and dynamic tuning, and enables our multi-stage solver to outperform the Intel MKL tridiagonal solver on many parallel tridiagonal systems by 6–11x.**

***Keywords*-GPU Computing, Auto-Tuning Algorithms, Tridiagonal systems**

I. Introduction

Tridiagonal linear systems arise in many scientific and engineering problems such as alternating direction implicit (ADI) methods [9], spectral Poisson solvers [10], cubic spline approximations, numerical ocean models [8], and preconditioners for iterative linear solvers [6]. A typical application may require solving up to hundreds or thousands of tridiagonal systems, which takes the majority of the application's total computation time. As the massively parallel GPU has evolved from a graphics-specific accelerator to a general-purpose computing device [14], recent studies have shown that GPU-accelerated tridiagonal solvers outperform CPU solvers by over an order of magnitude [4, 5, 16, 17]. Göddeke et al. developed a bank-conflict-free GPU implementation of CR, and employed it as a line relaxation smoother in a multigrid solver [5]. Egloff developed a GPU-based PCR solver to solve one-dimensional PDEs with finite difference schemes [4]. Sakharnykh simulated 3D viscid incompressible fluid on the GPU with the ADI method, which requires solving thousands of tridiagonal systems in parallel [16]. Zhang et al. designed a hybrid solver that combines cyclic reduction and its variant to balance the step and work efficiency [17].

However, a major problem not addressed by any previous work is the efficient solution of large tridiagonal systems that exceed the size of on-chip storage ("shared memory"). While it is straightforward to simply use global memory for all memory references in these solvers, the performance loss over an implementation that effectively uses shared memory is substantial (Egloff estimates a 60% performance degradation [4]). In this work, we tackle this problem by using a multi-stage solver. We first use two splitting stages to reduce large systems to smaller ones, and then solve them in shared memory using a two-stage hybrid tridiagonal algorithm.

One of the key challenges in making such a strategy efficient is choosing the switch points. For any given processor and particular workload, we can manually select appropriate switch points. However, the recent proliferation of manycore processors—several generations of CUDA-capable GPUs and, perhaps more importantly, a wide range of OpenCL-capable processors from many vendors—motivates a more automated and general strategy that will automatically select switch points for any workload and processor. In this work, we propose and implement an auto-tuning strategy that selects these switch points between computation stages.

Several recent studies have autotuned performance by empirical search [2, 3, 12, 13, 15]. Since various performance factors can form a very large optimization space, an exhaustive search for the optimal configuration is particularly time-consuming and impractical. Ryoo et al. [15] developed a technique which effectively prunes the search space by capturing the first-order performance effects. Liu et al. [12] use a greedy algorithm (hill climbing) to accelerate the search. Choi et al. [13] and Meng et al. [3] build a performance model to guide the tuning process respectively for sparse matrix-vector multiply and iterative stencil loops applications. Baskaran et al. [2] also use a model-driven search to determine the levels of loop unrolling and tiling. PetaBricks [1] developed by Ansel et al. uses a compiler-based method where programmers write a parameterized auto-tuned style program for multi-core CPU code.

Compared to the previous studies, our auto-tuning work differs in strategy as we combine two tuning-space pruning strategies. First, our knowledge of the algorithm and stages allows us to decouple the various tunable parameters from each other. While tuning all possibilities for both stages would be prohibitively slow, decoupling the parameters results in a much faster tuning procedure. Our method also

uses machine query parameters as a starting point in our search space to help guide our tuning search. However, we note a fully model-driven tuning strategy is impractical, as certain necessary device parameters cannot be queried, and certain parameter tradeoffs are difficult to model.

We make two major contributions with this work. First, we design a GPU-based multi-stage hybrid tridiagonal solver that can efficiently solve tridiagonal systems of any size and number, as long as they fit into global memory, while still exploiting shared communication when possible. Previous GPU tridiagonal solvers could only solve systems that fit directly into shared memory [17]. As workloads (size of systems, number of systems) and architectures vary, the optimal parameters and switch points will also change. Therefore, our second contribution is our self-tuning implementation of the tridiagonal solver. We believe that the strategy we advocate in this paper—a multi-stage solver that spans system sizes from small to large, with stage switch points determined through auto-tuning—will be applicable not only for tridiagonal solvers but also for a large class of divide-and-conquer problems such as fast Fourier Transforms (FFT) and quicksort.

Our paper is organized as follows: In Section II, we describe the machine model we use with this work. Section III will describe the stages of our tridiagonal solver and the tradeoffs between them. In Section IV we present three parameter selection strategies we tested for our solver, including the self-tuning strategy previously mentioned. Section V contains our results, Section VI our discussion, and Section VII our conclusions.

II. Machine Model

The GPU delivers high performance because it features a large number of parallel computing resources. Nickolls and Dally give a good overview of modern GPUs and their programming model [14]; for the purposes of this paper, we assume the following simplified machine model.

The GPU features many parallel *processors* (also known as cores or, in NVIDIA's terminology, "streaming multiprocessors"), each of which has numerous parallel *thread processors* that run parallel threads in lockstep. The number of processors is large enough that it is undesirable to leave any of them idle; code that runs on only a single processor is unlikely to be efficient. Each processor has a fixed, small amount of local *shared memory* that is shared among its thread processors. Problems whose data is all stored in shared memory are substantially faster than problems that must read data from global memory. When accessing global memory, a processor that reads contiguous chunks of memory (a *coalesced* memory access) will sustain significantly higher bandwidth than a processor that does not.

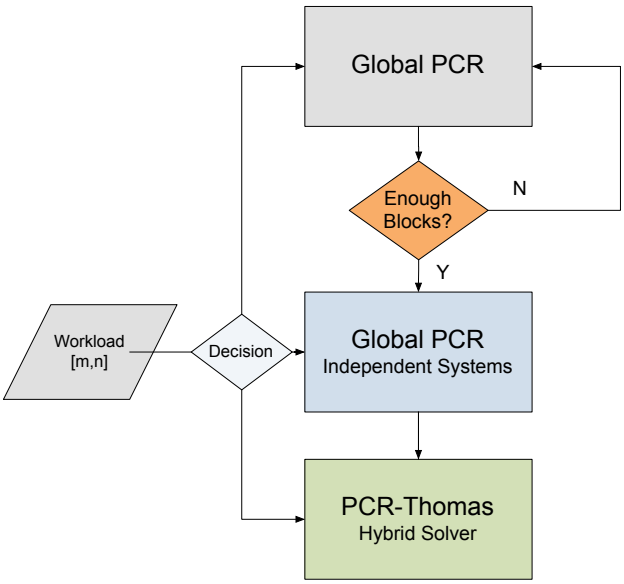


Figure 1: Workflow of our general solver. Depending on the number of systems and system size (m,n), our method selects the best algorithm to split up these systems. First we have an inter-block global split stage (more parallelism, with the cost of additional overhead) until there are enough independent systems that it makes sense for each Global PCR splitting block to work independently. This splitting continues until each subsystem is small enough to be solved using our hybrid PCR-Thomas kernel.

III. Method Description

Our method is composed of four stages assembled into three kernels, each of which maps best to a particular workload. Since each stage is workload-specific, our method takes into account differences in machine-specific parameters, and automatically selects and switches between methods. We elaborate on these parameters and our selection heuristic in Sections III-D and IV.

We construct our method with a bottom-up approach, beginning with an optimized base kernel that can solve a single small system within shared memory. This base kernel is a highly-optimized hybrid PCR (parallel cyclic reduction [11])-Thomas solver (described in Section III-A). It utilizes shared memory without bank conflicts and can solve many systems that fit into shared memory quickly and in parallel. As systems become larger than shared memory, we turn to a second stage (Stage 2). This kernel assigns each system to a processor, where we utilize PCR to quickly split up the system such that each sub-system is small enough to be solved by the baseline PCR-Thomas kernel. However, since each processor works independently on a system, if the workload is only one or a few systems, the machine will be under-utilized. Therefore, we develop another stage

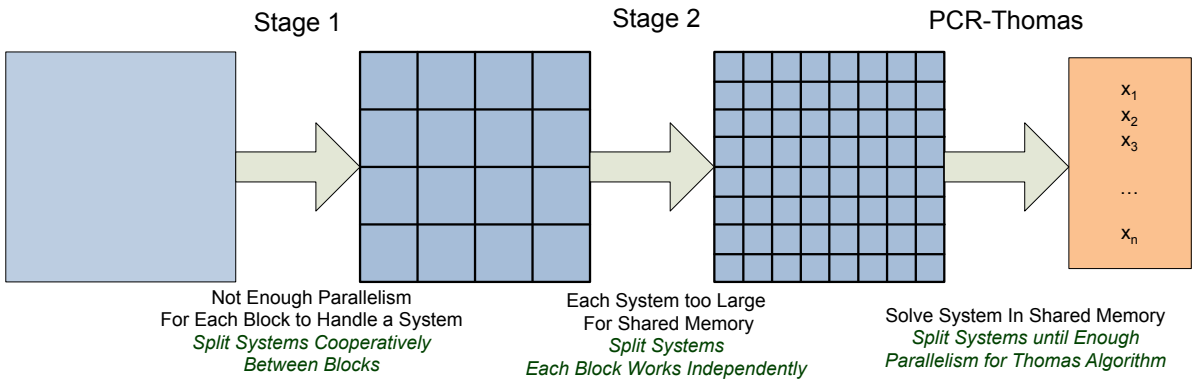


Figure 2: Each stage is optimized to handle a type of workload, and then pass the resulting subsystems to the next stage.

on top of this where processors cooperatively split systems (Stage 1).

In order to implement this system, we must choose switch points between these stages, and within each stage itself. The switch points represent a tradeoff between maximizing parallelism and minimizing total work. We discuss these switch points and tradeoffs in Section III-D, and how to select the best alternatives and parameters by using autotuning in Section IV. Figure 1 shows the workflow methodology of our tridiagonal implementation, and the following subsections discuss each of the three kernels in detail.

A. Base Kernel: A Hybrid Solver for Small Systems

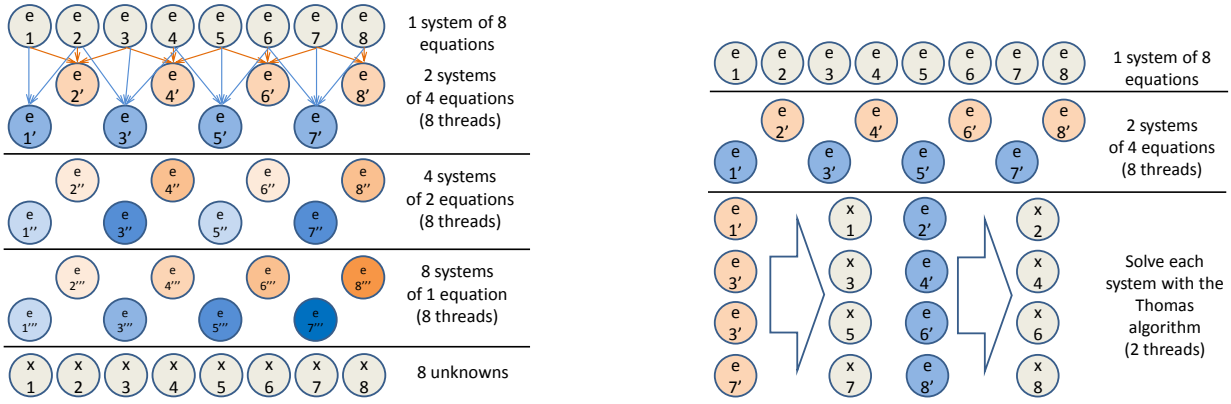
Our base kernel is a hybrid PCR-Thomas solver that fetches each system into shared memory before solving. In previous work, Zhang et al. [17] explored three different parallel algorithms for tridiagonal solvers on GPUs, including PCR, and showed that a hybrid solver between cyclic-reduction (CR) and PCR had the best performance. Independently and in parallel with our work, Sakharnykh [16] recently also proposed a hybrid PCR-Thomas algorithm with a different formulation. His implementation first splits the systems to smaller ones, and then solves each system with a CUDA thread. However, there are two drawbacks of this implementation: (1) It cannot use shared memory since each thread handles an independent system and thus all systems within a processor exceed the shared memory capacity, and (2) this method is only good at solving a large number of small systems, because the major parallelism exploited is at the thread level. We use a multi-stage method to overcome these two drawbacks. We first split the systems small enough to fit into shared memory, then instead of mapping each split system to a thread, we map it to a processor and solve it using a PCR-Thomas solver. This multi-stage method allows us to use shared memory and solve large systems as well as small ones.

Figure 3 illustrates the communication pattern and parallelism for a direct PCR solver and a PCR-Thomas solver (with no strided access). PCR requires only $\log(n)$ stages to solve n equations, but each stage requires $O(n)$ work. The Thomas algorithm is serial, requiring only $O(n)$ total work, but with $O(n)$ steps. Our hybrid algorithm uses PCR to divide the system into smaller parallel subsystems and then uses the Thomas algorithm to solve each smaller system serially within a single thread. Compared to Zhang et al.'s best (CR-PCR) hybrid algorithm, our work has similar performance for single-precision systems and better performance for double-precision systems; our primary advantage is leveraging the superior work efficiency of the Thomas algorithm. We use our hybrid solver to solve small systems.

Adapting this method to solve subsystems of a larger system requires more complex memory accesses. We can do so in one of two ways. First, we could specify a stride size to directly load such subsystems into shared memory. This strided load causes an initial penalty due to non-coalesced read accesses. However, since we store this initial load in shared memory, we are able to re-use this memory and quickly reach a solution. Alternatively, we could create another variation of this kernel that maintains coalesced read access, yet loses some shared memory communication within a processor. When a thread needs to access shared memory outside the range of loaded data, it must go to global memory for that data. Since all subsystems being solved are independent, we can guarantee no global-memory hazards. Choosing the highest-performing alternative is workload-dependent, so we make this decision automatically with the aid of our self-tuner.

B. Stage 2: Splitting Systems Up

Our PCR-Thomas baseline stage only solves systems that can fit within shared memory. For larger systems, our strategy is to split them into smaller systems that can then be solved with our baseline solver. PCR is well suited for this, as each stage splits work in half. Therefore if an input workload (system) is n times larger than the largest chunk



(a) Parallel cyclic reduction (PCR)

(b) PCR-Thomas

Figure 3: PCR and PCR-Thomas algorithms for solving an 8-equation system.

that can fit into shared memory, we require at least $\log(n)$ PCR stages in order for each subsystem to fit into memory.

C. Stage 1: Cooperative Splitting

The previous stage uses one processor to create smaller systems that can be distributed to multiple processors. However, when solving a small number of large systems, we may have insufficient independent systems to keep all processors busy. For these cases, we create a cooperative PCR-based splitting system that uses many processors to split a single system. Invoking this method only performs one split at a time, due to dependencies, and therefore incurs an extra penalty per split due to this synchronization. However, if there are only a few systems and we can split them enough such that Stage 2 can fully occupy the global memory bandwidth, we can obtain a significant speedup. The tradeoff here is the waste in leaving processors idle vs. the additional cost of the synchronization. We thus conclude that we only want to use this method until we spawn enough systems to fill the machine. An example of this type of tradeoff is illustrated in Figure 2. Again, Section III-D and IV will discuss our parameter selection strategy to find this switch point.

D. Summary & Switch Points

The solver we have just described has four stages, which we now describe from the top down. (1) If there are too few systems to keep all processors and memory controllers busy, we split each system into smaller systems, using multiple processors per system in order to utilize all processors. (2) After we have sufficient systems to keep all processors busy, we continue to split the systems until they fit into shared memory. In this stage, since we already have sufficient system-level parallelism, we are able to split each system independently with different processors, requiring only one kernel call and much less communication overhead. (3) Once subsystems fit into shared memory, we load each system from global memory into shared memory and solve it within a single processor. We continue to split these systems using parallel threads within a processor until we have sufficient parallel systems to allow each thread to solve a subsystem using the Thomas algorithm. (4) Finally, we solve all systems in parallel using the Thomas algorithm.

There are various tradeoffs when switching between stages. We want to enter stage 2 as early as possible, because the splitting in stage 2 is parallel across processors and requires little communication. However, at the same time, we need stage 1 to run long enough to generate sufficient systems to keep all processors and memory controllers busy. Between stage 2 and stage 3, we see similar tradeoffs between parallelism and splitting cost. On one hand, we want to enter stage 3 as early as possible so that we can start solving each systems in shared memory; on the other hand, we prefer that stages 1 and 2 split systems until they are small enough so that multiple systems can fit onto a single processor, which helps increase the latency tolerance and hence throughput on that processor. Between stages 3 and 4, switching earlier is beneficial due to the lower complexity of the Thomas algorithm, but has the cost of reduced parallelism and thus poor vector hardware utilization and latency-hiding.

Choosing the optimal switch points between stages is a difficult problem, since the decisions involve multiple factors including global/shared memory bandwidth, memory coalescing efficiency, thread occupancy, and algorithm complexity. We compare three solutions for switch point selection, which are respectively non-tuned, statically tuned, and dynamically tuned. The non-tuned solution uses default switch points that work for all GPUs. The static tuner guesses switch points based on the device information of a particular GPU. The dynamic tuner searches the best switch points at runtime using the static tuner's guess as a starting point. These methods will be described in the next section.

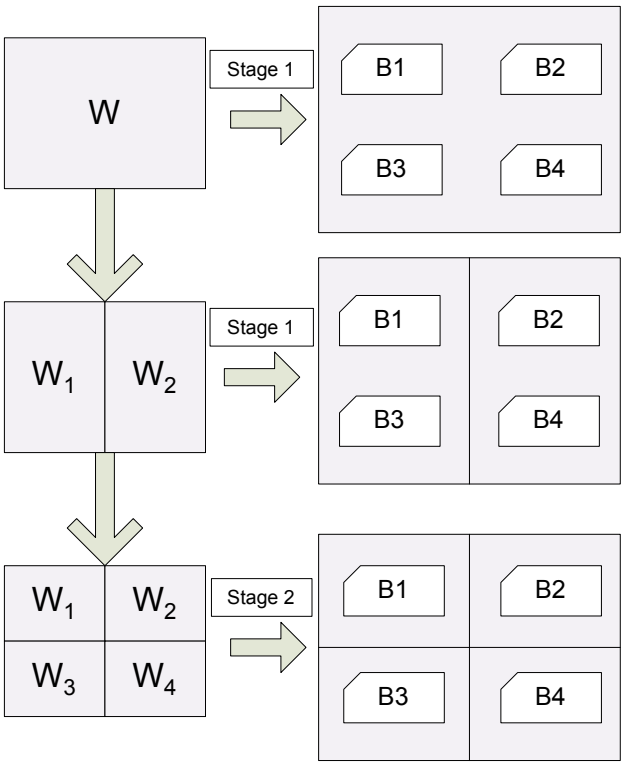


Figure 4: A Workload W needs to be split by CUDA blocks B_1 through B_4. Although stage 1 incurs a higher penalty per split than stage 2, running a number of stage 1 splits allows stage 2 to have higher throughput.

IV. Automatic Self-Tuning

The machine- and workload-dependent switch point parameters discussed in Section III-D are complex and difficult to optimize. To evaluate different ways of setting these parameters, we evaluate three different methods:

- A baseline parameter selection method, which chooses default parameters without any knowledge of the architecture. Using the same parameters for all architectures and workloads is a common approach taken by many codes and libraries.
- A machine-aware selection method. This method queries static machine characteristics and uses them to determine the optimum switch point.
- A self-tuning selection method. This method uses static machine characteristics when available, but also uses micro-benchmarks and searches a tuning space to identify the best choice of parameter switch points between all kernels.

Performance comparisons of these three methods will then reveal the usefulness of a self-tuner with respect to the other two methods.

Later in this section we will discuss in further detail our implementation of each of these parameter selection strategies, but first we will describe our target GPUs, their capabilities, and range of hardware resources.

A. GPU Targets

We tested our tridiagonal solvers and parameter selection methods on three different GPU architectures. All of these architectures are CUDA-capable GPUs chosen from the previous three generations of NVIDIA cards. Therefore our tests cover a wide spectrum of hardware capabilities. Table I lists the devices used in our tests, and some of their differences in capabilities.

Some of the information in Table I can be queried at runtime; however, most released CUDA programs use default parameters, or at best very simple device queries for sanity checks. Our results show that to ensure maximum performance on new hardware, these parameters must be reevaluated after every major architectural change.

B. Default Parameters

Since the default parameters are machine-oblivious, the default parameters must at least return correct answers (i.e. not crash) for all architectures. For parameters such as the PCR-Thomas switch point within the base kernel, this is not as vital as all devices will perform decently as long as the switch point is large enough that each warp has systems to solve. Though this parameter's performance difference is not the most vital, it will still be sub-optimal as the best switch point is machine-dependent.

A severe disadvantage of default parameters are that they must work on all machines. Therefore we must select a maximum number of systems to solve with PCR-Thomas that can fit on the lowest performing card. This limitation is due to the shared memory size differences between CUDA architectures. With the weakest architecture only able to fit 256 elements at a time, we select this as our switch point to our base kernel. This assumes that a kernel wishes to switch to PCR-Thomas as soon as each block can fit into shared memory, which is not always the optimal strategy.

Next we must decide default parameters for the last switch point (between Stages 1 and 2). The top-level kernel will only be selected for workloads containing a few systems (as discussed in Section III-D). The default parameters wishes to balance two tradeoffs as best as possible. First, we need enough independent systems so that Stage 2 can run at full occupancy, but we also note each call of Stage 1 is more expensive per-split than that of Stage 2. Our default parameter is sixteen systems; Stage 1 does not complete until it has split its input into sixteen independent systems. We chose this default selection as most devices have between four and twenty-four processors. Though this parameter is only partially related to the number of processors, we believe

Name	Global Memory Bandwidth	Shared Memory Size	Number of Processors	Thread Processors per Processor
8800 GTX	57.6 GB/s	16 KB	14	8
GTX 280	141.7 GB/s	16 KB	30	8
GTX 470	133.9 GB/s	48 KB	14	32

Table I: GPU devices used in our tests and benchmarks, from the last three generations of hardware. Each generation span introduces a change not only in the essential compute capabilities, but the memory and processor organization.

it to be a reasonable guess given no knowledge of the target architecture.

C. Machine Query Tuning

Our next parameter selection method relies on device information that can be queried using CUDA's *deviceProperties* class. Some of the information that this class can supply is contained in Table II. With some hardware specific information, we can make better parameter decisions than the default baseline method.

However, the range of information is somewhat limited. For example, the PCR splitting stage, as described in Section III, is not limited only by the number of processors, but is also heavily reliant on the global memory bandwidth. When there are enough independent systems to fully saturate this metric, we wish to switch directly to our Stage 2 kernel. However, information on this bandwidth (which is dependent on the number of memory controllers and the bus width to the global memory) can't be queried, and therefore we must estimate based only on the number of available processors.

Likewise, the switch-point tradeoffs within the hybrid PCR-Thomas algorithm are difficult to model using only machine-queried parameters. We know the number of thread processors per processor and the size of the shared memory, and can also calculate the number of accesses per step and the total number of steps to solve the system. However, we have no knowledge of (1) the number of shared memory banks and (2) the shared memory bandwidth per bank. Without this information, we can only make a reasonable guess given the shared memory size. Therefore we make a guess based on the warp size[1] instead, which is constant across all devices. Therefore when there are 64 independent subsystems per block, we switch to the Thomas method.

D. Self-Tuned Method

Our major contribution in this work is an automatically-tuned optimization method that uses device-queried parameters as starting guide points, and then performs a search on parameters that it identifies as independent. This strategy has two large advantages. First, since we identify ahead of time the parameter switch points that are independent, we reduce the search-space tremendously. As an example, if a parameter $P1$ had 16 possibilities, and $P2$ has 32 possibilities, and we identify $P1$ and $P2$ as independent of each other, then we must test only 16+32=48 possibilities instead of 16×32=512.

In our case we can identify the switch point between stage 1 and stage 2, and the switch point between stage 3 and stage 4 in the PCR-Thomas kernel as independent. The reason we can decouple these is as follows:

- The goal of the switch point between stage one and stage two is to split systems until there is enough parallel work for stage two.
- The goal of the switch point between stage three and stage four is dependent on small systems in shared memory and the size of each small system.
- These dependencies do not interfere with each other, and therefore we can decouple them.

The second advantage of our method is our use of machine queried parameters as a guess point also helps us reduce our tuning space. For most of our parameters, there lies a parameter set that provides a local minimum in a hyperbolic search space. Therefore, by using machine queries to help guide our search, we usually get very close to this local minimum, reducing the time to locate this minimum.

We illustrate this through a parameter we wish to tune. When do we switch to the PCR-Thomas kernel? Recall that the machine-query strategy switches as soon as each subsystem can fit into shared memory. In order to improve upon this, we perform the following series of tests:

- Begin with the machine-query parameter selection for a workload guaranteed to fill the machine (number of systems much greater than the number of processors).
- Benchmark this selection vs. two times the number of systems at half the size. We must tune for the ideal stage-3 to stage-4 switch point for each of these settings, and for the two base PCR-Thomas kernels we coded (uncoalesced with better shared memory access and coalesced with worse shared memory access).
- Continue until you have found the local minimum, and save those results for future runs.
- Repeat this stage increasing the stride count for the two base PCR-Thomas kernels (this simulates solving larger systems), until we know how large systems must be until the uncoalesced version is preferred.

Now we have the selection parameters for stage-3-to-stage-4 and stage-2-to-stage-3 independent of the switch point between stage 1 and stage 2. The final parameter we

[1]The warp size in an NVIDIA GPU is the granularity of work run in lockstep; on all NVIDIA GPUs a warp is 32 threads that are run in parallel.

Query Parameter	Description
Global Mem	Total amount of global memory available
Processors	Total number of processors available; each processor has n thread processors where n depends on the architecture
Constant Memory	Total amount of constant memory per block; each constant loaded can be broadcast across MPs
Shared Memory	Total amount of shared memory per processor, which limits both the number of systems being processed concurrently in a processor and the maximum size our PCR-Thomas kernel can solve
Register Memory	The limited number of registers available per block often yields a trade-off between the total number of threads and the number of registers per thread
Grid Dimensions	The API imposes a limit on the software layout of work (specifically the number of blocks per grid)

Table II: A list of queryable CUDA device properties. Our machine-tuning method uses these parameters and the characteristics of the workload to make a best guess on the optimum parameters.

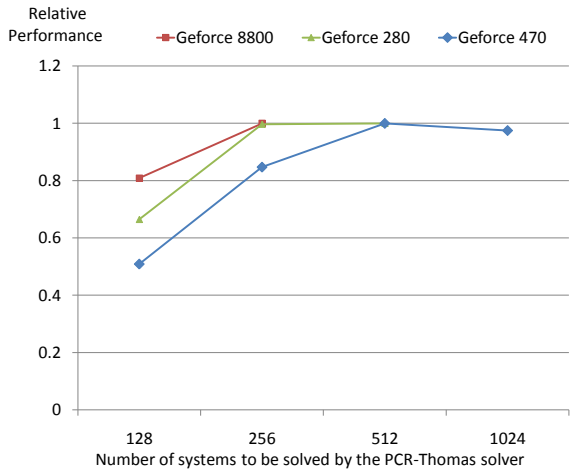


Figure 5: Performance comparison at various switch points from stage 2 to stage 3. The numbers are normalized to the best performance at optimal switch point.

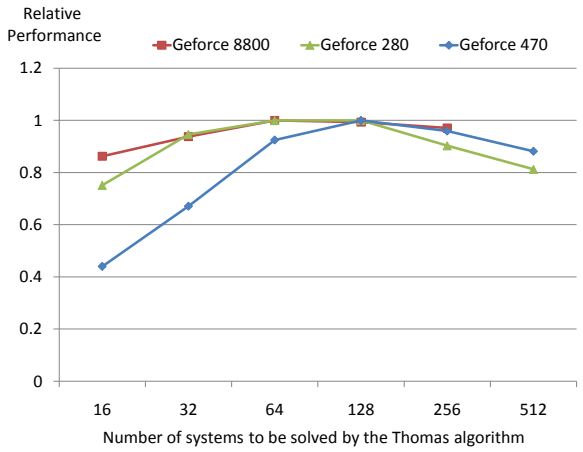


Figure 6: Performance of the PCR-Thomas solver at various switch points. The numbers are normalized to the best performance at an optimal switch point.

must tune we is the total number of independent systems we need to keep stage 2 busy. First, we select a sizable workload for each machine that requires a large amount of splitting (e.g. one system that takes all of global memory). Since we know the splitting endpoint (the stage-2-to-stage-3 switch point), we can start at the machine-guess parameter and iterate over its neighbors until we find a local minimum. We then save this switch point parameter for future runs.

A typical self-tuning run for a particular system and GPU takes less than one minute.

V. Experimental results

Our test platform uses the NVIDIA CUDA 3.1 GPU programming environment running on Microsoft Windows XP, a 3.4 GHz Intel Core i5 dual-core CPU, and three NVIDIA GPUs of different generations: GeForce 8800, 280, and 470. At a high level, static tuning is almost always better than non-tuning and delivers an average 17% runtime improvement over non-tuning; dynamic self-tuning is always better than either static or no tuning and shows a 32% improvement in runtime over no tuning.

Figure 5 compares our performance at various switch points from stage 2 (global splitting) to stage 3 (solving in shared memory). The switching involves a tradeoff between the cost of global memory splitting, shared memory solving, and the amount of parallelism that can hide memory latency. Depending on the register and shared memory resources, the largest systems that can be solved locally on-chip are of sizes 256, 512, and 1024 respectively for the GeForce 8800, 280, and 470. For the GeForce 470, our self-tuning revealed that it is beneficial to split the system one step further from size 1024 to 512 even though 1024 can already fit in shared memory. This is because the GeForce 470 features more thread processors per processor and requires more resident threads on a processor to keep a high occupancy for latency hiding; processors on the 470 can work on two subsystems of size 512 locally at the same time. For the GeForce 280, switching at system sizes 256 and 512 have comparable performance. The GeForce 8800 requires the fewest threads to maintain high occupancy, and therefore prefers a larger system size of 256 instead of 128.

Figure 6 shows the performance of the stage-3 PCR-

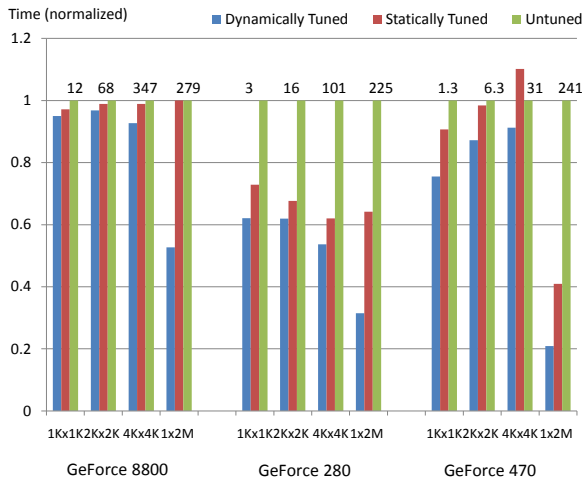


Figure 7: Comparison of non-tuned, statically tuned, and dynamically tuned performance for various workloads and GPUs. 1K×1K stands for 1024 1024-equation systems and 1×2M stands for 1 system of 2 million equations. The timings are normalized to the untuned (or default parameters) execution time. The numbers on top of the columns indicate the non-tuned execution time in milliseconds.

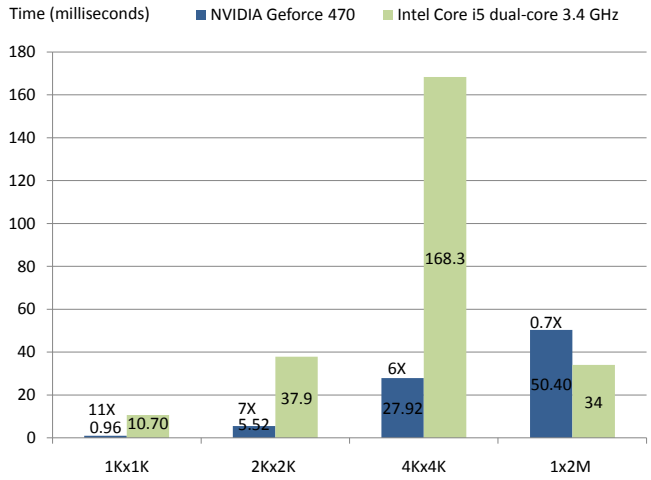


Figure 8: Performance comparison for various workloads between GPU and CPU. The CPU solver is a highly-optimized tridiagonal solver from the Intel MKL library (version 10.2.5.035). When solving many systems, we use a two-threaded implementation on two CPU cores with each thread executing a MKL solver. For solving a single system of 2 million equations, we use a single thread, since the MKL solver is sequential.

Thomas solver at various switch points. At stage 3, each split system is loaded into shared memory and solved by the PCR-Thomas solver. The PCR algorithm splits the single system into many subsystems, then passes these subsystems to the Thomas algorithm that uses a single thread per subsystem. Switching from PCR to the Thomas algorithm earlier means a reduced algorithmic complexity from $O(n \log n)$ to $O(n)$, but at the cost of less parallelism to hide memory latency. Tuning shows that for the GeForce 280 and 470, the best switch point is 128 subsystems, while for the GeForce 8800, the best switch point is 64 subsystems. Because our static tuner will always choose 64 subsystems as the switch point, this result means dynamic tuning will improve the performance further.

Figure 7 summarizes the non-tuned, statically-tuned, and dynamically-tuned performance for various workloads and GPUs. The static tuning outperforms the non-tuned solver by up to 60% with an average runtime decrease of 17%, primarily because it can use machine-specific parameters to allow larger systems to fit into a single GPU processor. In contrast, the non-tuned solver must use a safe third-stage system of size 256 to ensure it runs on all GPUs. Because it is hard to estimate the costs and benefits of various tradeoffs between parallelism, splitting cost, and coalescing efficiency, the static tuner is unable to make the best decision on switch points. For example, switching early from global splitting to stage 3 (solving in shared memory) reduces the splitting work in global memory and enjoys more shared memory communication, but results in a lower processor occupancy at the same time. As previously discussed in Figure 5, dynamic tuning is able to find a better stage-3 system size than the default size chosen by static tuning. As another example, in the stage-3 PCR-Thomas solver, switching early from PCR to Thomas reduces the amount of total work, but may lead to insufficient parallelism to utilize all thread processors. Dynamic tuning is able to choose a better subsystem size for the PCR-Thomas solver than static tuning's default size. The dynamic tuning achieves a maximum of a 5x speedup and an average of 32% against the non-tuned performance, with the largest speedups on the largest systems.

VI. Discussion

In this section we will discuss the implications from our results. We will first discuss the results from each of our parameter selection strategies: default, machine-queried, and self-tuning. Following that we will discuss the scalability of our method by comparing it to a CPU MKL implementation. Finally, we will discuss how our method can be applied on a broader scale for algorithms with similar computational patterns.

A. Parameter Selection

Tuning has a significant impact on performance. Default parameters behave very poorly across systems, because the default parameters are designed for a baseline (least-common-denominator) architecture (in this case the 8800 GTX). This realization has of course motivated tuning

strategies across many areas of GPU computing and beyond, and machine-query tuning can give significant speedups. However, the assumptions made by such a tuning strategy are not optimal, and can even sometimes be detrimental.

For example, Figure 7 shows that using the default parameters outperforms the statically-tuned parameters for a 4096×4096 system. This specific result is because our static tuning assumes the PCR-Thomas kernel should be launched as soon as each system can fit into shared memory, while the default parameters yield more splitting stages that result in higher occupancy per processor and in this case, better performance.

Dynamic self-tuning, on the other hand, does not suffer from these problems. By first identifying the optimal parameters for our two PCR-Thomas base kernel implementations, and the size of each subsystem (stage-2-to-stage-3 switch point), our self-tuned method is better able to balance the tradeoff between processor occupancy and shared memory communication. Self-tuning also does a much better job than machine-query tuning for solving a single large system. Since the machine-query tuning strategy is unable to obtain specific information regarding the memory controllers and bus width, it has little information to tune with and therefore selects poorly. The result is roughly a 2x difference in performance on each card, sometimes electing to switch too early and sometimes too late.

In general, tuning is necessary for these algorithms to scale in performance across machines. Though guesses through machine queries can help in this regard, we advocate some form of self-tuning. Our self-tuning strategy is powerful because it (1) prunes the tuning space such that these tuning runs are not prohibitively expensive and (2) it can account for more situations than a static query.

B. CPU vs. GPU Timings

We tested a wide range of systems on a GPU with our algorithm. The NVIDIA GTX 470 significantly outperforms the CPU Intel MKL implementation for large numbers of parallel tridiagonal systems. The MKL solver uses a sequential LU decomposition algorithm. For solving many systems, we use OpenMP to parallelize the program at the system level with each system solved by a single CPU thread. Figure 8 shows that increasing the size and count of systems results in a slightly decreasing advantage for the GPU over the CPU. Our speedups range from 6x (4096 parallel systems of 4096 equations) to 11x (1024 parallel systems of 1024 equations).

If we consider the extreme case of 1 enormous system (a single system with 2 million elements), the CPU outperforms the GPU. As PCR is not a work-efficient algorithm, when the workload becomes PCR-dominated (a 2-million-element system must perform 11 stages of PCR splitting in global memory before fitting into shared memory), the speedups of our GPU hybrid method versus the CPU MKL implementation deteriorate.

C. Our Work on a Broader Scale

Our work here concentrates on solving tridiagonal systems that vary in size, all the way from many small systems to one large system. In particular, our strategy of solving one large (or many large) system(s) by breaking them into smaller systems is one of the contributions of this paper. More broadly, we believe this strategy computation is useful beyond tridiagonal systems.

Consider the problem of bottom-up merge sorting, where small chunks of the input are sorted and then these sorted chunks are recursively merged in a tree structure. Hagerup and Rüb's parallel merge algorithm represents this style of sort [7]. An implementation of this algorithm on the GPU faces the same issues as our tridiagonal solver: a shift from solving many independent chunks within a single processor's shared memory to solving many independent chunks that do not fit within shared memory, and a second shift from solving enough chunks to fill the machine to solving fewer, larger chunks that do not fill the machine. In general we believe many divide-and-conquer algorithms (e.g. FFT, dense matrix multiplication, median finding), which often overlap with recursively-specified cache-oblivious algorithms, will benefit from this strategy.

VII. Conclusion

In this work, we presented a tridiagonal solver that can solve a variety of workloads on the GPU. This solver had multiple stages that were best suited for different workloads. This allowed us to efficiently solve much larger systems than previous GPU tridiagonal solvers. Each stage in our system processed subsystems, passing its results on to the next stage until all systems were solved. In order to know when to switch to the next stage, we created three parameter selection techniques: default, static machine-query, and dynamic self-tuned. For this workloads, static tuning noticeably outperforms default tuning, and dynamic self-tuning outperforms both. We also note that on one case, the CPU outperforms the GPU.

We believe these results motivate future use of self-tuning techniques, particularly for multi-stage algorithms that involve multiple switch points (e.g. quicksort on the GPU). However, we note that trade-off penalties and tuning dependencies between algorithms vary widely, so to achieve similar speedups on other algorithms, we feel that designers of these implementations must have a good understanding of their parameter space and how it affects the algorithm. Given the trends in the manycore market, the potential benefits of such implementations will only grow: more generations of GPUs with different performance characteristics coupled with the larger diversity of manycore devices (particularly

OpenCL-capable devices) make performance tuning an increasingly difficult problem that we hope will be addressed with auto-tuning techniques.

The next challenge in this specific application domain is high-performance blocked tridiagonal solvers and optimized banded solvers. More broadly, we would like to extend this auto-tuning strategy to more algorithms in the divide-and-conquer space and extend our techniques to also explore the boundary between GPU and CPU.

Acknowledgements

The authors would like to thank Dominik Göddeke (Institute of Applied Mathematics, Dortmund, Germany) for insightful feedback on earlier versions of this paper. Thanks also to the SciDAC Institute for Ultrascale Visualization, the HP Labs Innovation Research Program, and the National Science Foundation (Awards 0541448, 1017399, and 1032859) for funding, and to NVIDIA for equipment donations.

References

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, Dublin, Ireland, June 2009.

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

[3] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 115–126, January 2010.

[4] Daniel Egloff. High performance finite difference PDE solvers on GPUs. Technical report, QuantAlea GmbH, February 2010. http://download.quantalea.net/fdm_gpu.pdf.

[5] Dominik Göddeke and Robert Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):22–32, January 2011.

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

[7] Torben Hagerup and Christine Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33(4):181–185, December 1989.

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

[9] 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.

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

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

[12] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing (IPDPS '09)*, May 2009.

[13] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 256–265, June 2009.

[14] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March/April 2010.

[15] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain Zee Ueng, Sara S. Baghsorkhi, and Wenmei W. Hwu. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, 68(10):1389–1401, 2008.

[16] Nikolai Sakharnykh. Efficient tridiagonal solvers for adi methods and fluid simulation. In *NVIDIA GPU Technology Conference 2010*, September 2010.

[17] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 127–136, January 2010.