A Scalable Tridiagonal Solver for GPUs

Hee-Seok Kim, Shengzhao Wu, Li-wen Chang and Wen-mei W. Hwu
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
{*kim868,wu14,lchang20,w-hwu*}@*illinois.edu*

***Abstract*—We present the design and evaluation of a scalable tridiagonal solver targeted for GPU architectures. We observed that two distinct steps are required to solve a large tridiagonal system in parallel: 1) breaking down a problem into multiple subproblems each of which is independent of other, and 2) solving the subproblems using an efficient algorithm. We propose a hybrid method of tiled parallel cyclic reduction(tiled PCR) and thread-level parallel Thomas algorithm(p-Thomas). Algorithm transition from tiled PCR to p-Thomas is determined by input system size and hardware capability in order to achieve optimal performance. The proposed method is *scalable* as it can cope with various input system sizes by properly adjusting algorithm trasition point. Our method on a NVidia GTX480 shows up to 8.3x and 49x speedups over multithreaded and sequential MKL implementations on a 3.33GHz Intel i7 975 in double precision, respectively.**

***Keywords*-GPU Computing, GPGPU, Tridiagonal solver, Tridiagonal systems**

I. Introduction

The tridiagonal solver is an important core tool in wide range of engineering and scientific applications. Some applications of tridiagonal solvers include computer graphics [1][2][3], fluid dynamics [2][4][5], Poisson solvers [6], preconditioner in iterative solvers [7], cubic spline calculation [8] and semi-coarsening for multi-grid method [9][10].

Recent technological evolution of GPUs has lifted many scientific and engineering applications to a level that was only possible with room-sized supercomputers in the past. The enormous performance improvements are largely due to the inherent parallelism and regular memory access patterns of the scientific applications that GPUs can efficiently deal with. Specifically, GPUs can run many threads in parallel that fulfill high computation demand, and provide large memory bandwidth to serve parallel memory access requests.

Accelerating tridiagonal solvers with parallel execution, however, brings significant challenges as it requires breaking inherent sequential dependencies of classic Thomas algorithm [11]. Also, low computation-to-memory ratio of the problem leads us to develop a desired memory layout for efficient parallel execution. Moreover, the parallel implementation needs to keep computational complexity of the algorithm low enough to be competitive against sequential counterparts when input size grows.

While the Thomas algorithm is widely used on sequential machines, several parallel algorithms have been studied. Notable among these parallel algorithms are cyclic reduction(CR) [12], parallel cyclic reduction(PCR) [12] and recursive doubling (RD) [13]. Sengupta et al. [3] first implemented CR on the GPU for water simulation. Göddeke et al. [10] proposed a bank-conflicts-free CR implementation on the GPU for a smoother in multi-grid. Egloff [14][15] developed a PCR implementation to solve large size tridiagonal systems for finite difference PDE solvers. Sakharnykh used p-Thomas algorithm [4] and a hybrid of PCR-Thomas [5] for fluid simulation. Yao Zhang et al. [16][17] proposed a hybrid technique among Thomas, CR, PCR and RD along with a comprehensive performance analysis among various algorithms and hybrid techniques. CR is further optimized with register packing by Davidson et al. [18]. A hybrid of PCR-Thomas, first proposed by Sakharnykh [5] and also studied by [17], is similar to our work but both approaches can only solve small sized systems as their methods store an entire input system in shared memory. As a result, the limited capacity of shared memory considerably limits their availability for real use. Davidson et al. [19] also proposed a scalable PCR-Thomas hybrid to handle large systems. While their work has the most similarity to our method, there are key implementation techniques and decisions that result in different performance that is scrutinized later in this paper.

In this paper, we propose a robust and scalable parallel tridiagonal solver that uses GPUs effectively. Our method is a hybrid of the tiled PCR and p-Thomas algorithms. It implements divide-and-conquer approach as tiled PCR breaks down a system into multiple independent systems, then p-Thomas takes over the systems and runs in parallel. Here, tiled PCR plays an important role in two ways. First, it transforms a system into multiple independent systems so that the original problem becomes more manageable for GPU to work on. Second, it partitions a large system into small chunks, or *tiles*, which efficiently overcomes the size limitation of shared memory without incurring redundant data loads. The hybrid algorithm determines at runtime the degree of parallelism that tiled PCR should provide to match parallelism given in the GPU hardware. The proposed method not only performs efficiently for large size systems but also covers arbitrary number of systems.

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.41

Our experiments show that the proposed method on a NVidia GTX480 can achieve up to 8.3x and 49x speedups over multithreaded and sequential MKL implementation on an 3.33GHz Intel i7 975 with double precision arithmetic. With single precision, we provide up to 12.9x and 82.5x speedups over the CPU implementations.

The rest of the paper is organized as follows: Section II describes previous and background material about tridiagonal solvers and GPU architectures that is relevant to this paper. Section III describes the proposed algorithm in detail. Section IV follows to provide the results. Section V compares our method to the work by Davidson et al. [19] and discusses the benefit of the proposed method. Section VI concludes this paper.

II. Preliminaries

A. Tridiagonal solver

A tridiagonal matrix solver is an algorithm to find a solution of $Ax = d$, where A is an n-by-n tridiagonal matrix and d is an n-element vector as shown in Eq 1.

$$A = \begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & 0 & \\ & \ddots & \ddots & \ddots & \\ & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} d = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{1}$$

1) Thomas Algorithm: The Thomas algorithm is a special case of Gaussian elimination for a tridiagonal matrix. The algorithm has two phases: forward reduction and backward substitution. In the forward reduction phase, the lower diagonal is eliminated by the main diagonal sequentially as shown in the following equations.

$$c_1^{'} = \frac{c_1}{b_1},\ c_i^{'} = \frac{c_i}{b_i - c_{i-1}^{'} a_i},\ i = 2,3,\dots,n-1 \tag{2}$$

$$d_1^{'} = \frac{d_1}{b_1},\ d_i^{'} = \frac{d_i - d_{i-1}^{'} a_i}{b_i - c_{i-1}^{'} a_i},\ i = 2,3,\dots,n \tag{3}$$

The backward substitution uses the upper and the main diagonal to solve all remaining unknowns from the last row to the first row as formulated below.

$$x_n = d_n^{'},\ x_i = d_i^{'} - c_i^{'} x_{i+1},\ i = n-1, n-2, \dots, 1 \tag{4}$$

The required number of elimination steps of Thomas algorithm is $2n-1$ and the computational complexity is $O(n)$.

2) Cyclic Reduction: CR, also called odd-even reduction, is a special two-way elimination for tridiagonal matrix. It also consists of two phases: forward reduction and backward substitution. In the forward reduction, adjacent equations in the tridiagonal system are used to eliminate alternating unknowns and the resulting new equations form a smaller system with a half number of unknowns. The process continues recursively until two unknowns are left.

$$\begin{array}{c} e_1 \\ e_2 \\ e_3 \\ e_4 \end{array}\begin{bmatrix} b_1 & c_1 & & \\ a_2 & b_2 & c_2 & \\ & a_3 & b_3 & c_3 \\ & & a_4 & b_4 \end{bmatrix} \rightarrow \begin{array}{c} e_1 \\ e_2' \\ e_3 \\ e_4' \end{array}\begin{bmatrix} b_1 & c_1 & & \\ 0 & b_2' & 0 & c_2' \\ & a_3 & b_3 & c_3 \\ & a_4' & 0 & b_4' \end{bmatrix} \rightarrow \begin{bmatrix} b_2' & c_2' \\ a_4' & b_4' \end{bmatrix}$$

Figure 1. An example of forward reduction in CR algorithm

Figure 1 shows one step of forward reduction phase using a 4x4 matrix. In the row e_2, a_2 and c_2 are eliminated by Eqs. 5-6 with e_1, e_2 and e_3, substituting e_2 with e_2'. The same goes for the row e_4 where a_4 is eliminated using e_3 and e_4. After that, e_2' and e_4' can be compacted by removing columns with all zeros, yielding a smaller matrix with half number of unknowns as shown in the final matrix.

The backward elimination can substitute remaining unknowns with the solutions of the smaller systems after successive forward reduction. For instance, e_1 and e_3 in the previous example can be solved once e_2' and e_4' are solved, which can be done easily when the matrix size is 2x2. The equations involved in this stage are shown in Eqs. 5-7.

$$a_i^{'} = -a_{i-1}k_1, b_i^{'} = b_i - c_{i-1}k_1 - a_{i+1}k_2 \tag{5}$$

$$c_i^{'} = -c_{i+1}k_2, d_i^{'} = d_i - d_{i-1}k_1 - d_{i+1}k_2 \tag{6}$$

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

$$x_i = \frac{d_i^{'} - a_i^{'} x_{i-1} - c_i^{'} x_{i+1}}{b_i^{'}} \tag{7}$$

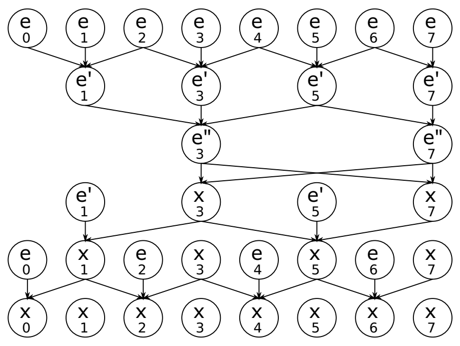


Figure 2. Access pattern of CR with 8-element system

Figure 2 shows the access pattern of CR for an 8-element system. Calculating to the last step of forward reduction reveals tree-shaped dependency pattern from the rows of the input system. The computational complexity of CR is $O(n)$ and the required number of elimination steps is $2\log n + 1$.

3) Parallel Cyclic Reduction: PCR is a modification of CR but it only has a forward reduction phase in contrast to CR. While CR performs forward reduction on only odd or even rows in each step, PCR performs reduction on both odd rows and even rows. A system is transformed into two smaller systems, each of which has half the number of unknowns in PCR.

$$\begin{array}{c} e_1\\ e_2\\ e_3\\ e_4\end{array}\begin{bmatrix} b_1 & c_1 & & \\ a_2 & b_2 & c_2 & \\ & a_3 & b_3 & c_3 \\ & & a_4 & b_4\end{bmatrix} \to \begin{array}{c} e'_1\\ e'_2\\ e'_3\\ e'_4\end{array}\begin{bmatrix} b'_1 & 0 & c'_1 & \\ 0 & b'_2 & 0 & c'_2 \\ a'_3 & 0 & b'_3 & 0 \\ & a'_4 & 0 & b'_4\end{bmatrix} \to \frac{\begin{bmatrix} b'_1 & c'_1\\ a'_3 & b'_3\end{bmatrix}}{\begin{bmatrix} b'_2 & c'_2 \\ a'_4 & b'_4\end{bmatrix}}$$

Figure 3. An example of forward reduction phase of PCR algorithm

Figure 3 shows one step of forward reduction of PCR algorithm. While the forward reduction in CR transforms e_2 and e_4 in the previous example, PCR does another forward reduction for e_1 and e_3, which results in producing two small matrices as shown in the figure. The computational complexity of PCR is $O(n\log n)$ and the required number of elimination steps is $\log n + 1$.

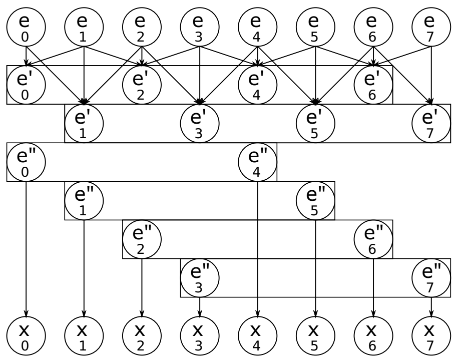


Figure 4. Access pattern of PCR with 8-element system

Figure 4 illustrates the data access pattern of PCR algorithm, showing multiple tree-shaped dependency toward the final solution from the input rows. In this particular example, two boxes from the top represent two systems after one PCR step, each of which is 4-element system. Then two boxes go through one more PCR step to produce four 2-element systems, depicted as four boxes from the bottom. Arrows for the second PCR step is omitted due to complexity of drawing, though it can easily be inferred such as e''_0 consumes e'_0 and e'_2, e''_4 requires e'_2, e'_4 and e'_6, and so on. Two-element systems then can be directly solved and each box produces solutions completing the whole solution.

B. GPU Architecture

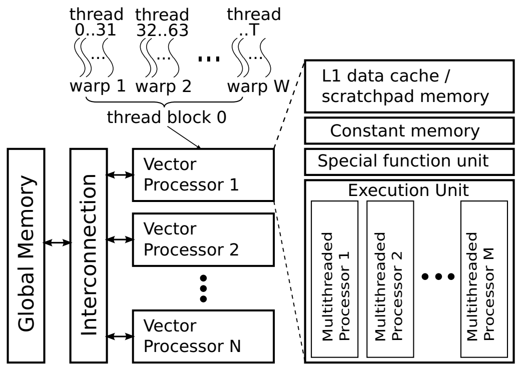


Figure 5. An overview of the GPU Architecture

As shown in Figure 5, modern GPUs typically consist of vector processors each of which contains several multithreaded execution units, one or more special function units, on-chip L1 data cache and/or scratchpad memory, and read-only memory. GPU architectures directly reflect a set of GPU programming primitives. In NVIDIA's CUDA programming model, for instance, a kernel function is the main venue for expressing parallel computation, which runs simultaneously by many *threads*. Multiple threads are grouped into a *thread block*. All of the threads in a thread block will run on a Streaming Multiprocessor(SM) which is an instance of the vector processor. Programmers can use fast synchronization and shared memory among threads within a thread block. All thread blocks together form a *grid*. OpenCL's programming model also has similar mapping between programming primitives to architecture components.

III. Proposed method

Our method is a multi-stage algorithm of tiled PCR and p-Thomas. The algorithm starts with tiled PCR and moves on to p-Thomas. The algorithm switching point is determined by the size of input system and underlaying hardware, as efficiency of both algorithms differs. Figure 6 illustrates how one 8-element system is solved using the proposed method, with a focus on the data access pattern as we have shown previously in Figure 2 and Figure 4. At first, PCR breaks down the input system into two 4-element systems. In this particular example, after one step of PCR, p-Thomas takes over the two systems where two threads solve two matrices in parallel.

The rest of this section describes each part of the proposed method: tiled PCR, p-Thomas and the algorithm switching logic between two implementations.

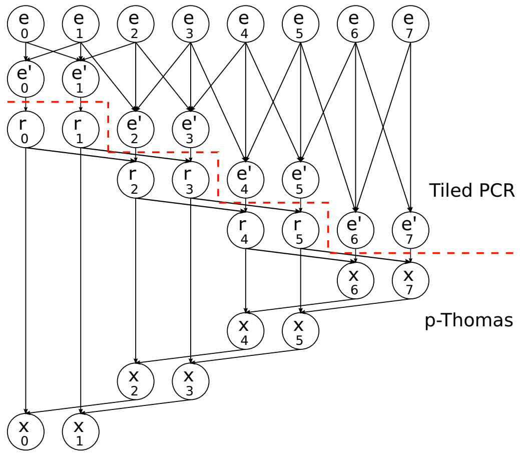
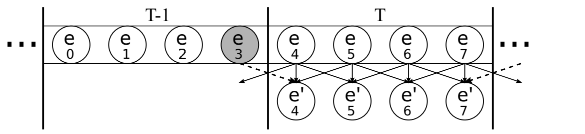


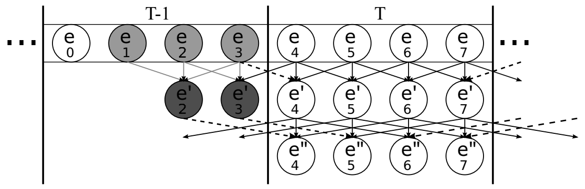
Figure 6. An example of the proposed method with one PCR step for one 8-element system

A. Tiled PCR

The tiled PCR is proposed as a variant of *incomplete* PCR in a sense that it stops breaking down the systems before the algorithm reaches the smallest possible matrices. Unlike previous implementations on GPUs, our tiled PCR works with small fixed-sized arrays, called *tiles*, to handle a large system in which multiple tiles are processed concurrently. With tiling, instead of loading the whole system, a large system is loaded chunk by chunk into a tile which is allocated in shared memory.



(a) 1-step PCR



(b) 2-step PCR

Figure 7. Redundancy of naive tiling of PCR

One major performance limiting factor of naive tiling of PCR is halo elements due to dependency spanning a across tile boundary as shown in Figure 7, illustrating tile T being processed. In (a), processing one-step PCR for tile T requires loading e_3 from tile $T-1$, depicted as grey circles, which becomes redundant as e_3 is also loaded when processing tile $T-1$. The redundancy grows quickly with larger k as shown in (b), where two-step PCR is performed for tile T. In this case, e_1 to e_3 must be brought in from memory to make all dependencies available in order to produce e''_4 and e''_5. Moreover, e'_2 and e'_3, depicted as dark grey circles, need to be calculated which reveals another type of redundancy as they will also be computed while tile $T-1$ is processed. In general, when performing k-step PCR, the number of redundant memory accesses per tile boundary, denoted as $f(k)$, is formulated as

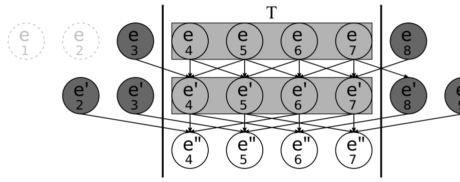
$$f(k) = \sum_{i=0}^{k-1} 2^i, \tag{8}$$

and subsequent number of elimination steps, denoted as $g(k)$, can be formulated as

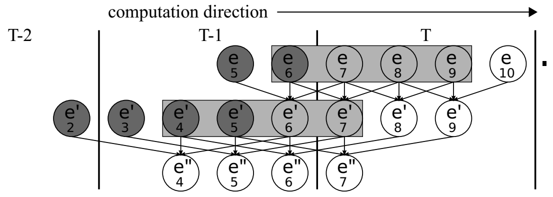
$$g(k) = k \cdot f(k) - \sum_{i=0}^{k} f(i), \tag{9}$$

, both of which grow exponentially as k grows.

Fine-grained tiling in an effort to achieve more parallelism thus will be challenged by a massive amount of redundant memory access and computation as shown previously. Since each tile boundary introduces redundancy, one way to mitigate such redundancy is to use larger tiles. However, it prohibits exploiting parallelism causing the benefit of parallelization to quickly diminish.



(a) 2-step PCR using cached dependency from both sides



(b) 2-step PCR using cached dependency from lefthand side

Figure 8. Dependency caching for tiled PCR

Another way to reduce the redundancy, which we propose, is caching dependent values that would otherwise be redundant, as shown in Figure 8. The figure illustrates a tile being computed to produce e''_4 to e''_7 with cached dependencies in which a grey box represents a set of elements being an object of PCR operation, and dark circles indicate cached results. The caching reduces expensive memory access and elimination costs that are necessary to produce immediate dependency. For instance, in (a), loading e_1 and e_2 as well as one elimination step to generate e'_2 can be avoided if e'_2 is cached from processing tile $T-1$. Implementing the idea in (a) is impractical, however, as getting the cache contents for both side could also bring about the redundancy as we discussed before. We modified the scheme so that all dependencies are cached and available from previously computed tiles, by processing through tiles sequentially as shown in (b). While conceptually the same as (a), in this figure e_5 and e_6 are cached from processing tile $T-1$ and used for the first step of PCR. After the PCR operation, e_9 and e_{10} substitute the contents of the cache so that they can be used the same way for a following tile. Likewise, upon completion of the second PCR step, e'_6 to e'_9 take place of e'_2 to e'_5. The same logic works for further steps of PCR with different cache sizes. Assuming k-step PCR, the overall cache size requirement is $2 \cdot f(k)$ and it remains constant for the same k. As long as the cache has the required size, several tiles can be processed without redundancy.This leads us to a two-level approach where the first level is the unit of parallel execution and each of the tiles has multiple sub tiles that are processed sequentially using caching. The tile size in this approach is no longer bound to shared memory size and as a result the method is adaptable as the sizes of tile and sub tile can be determined by other significant factors such as available degree of parallelism. Therefore, exploiting more parallelism becomes possible compared to the naive approach that only increases tile size.

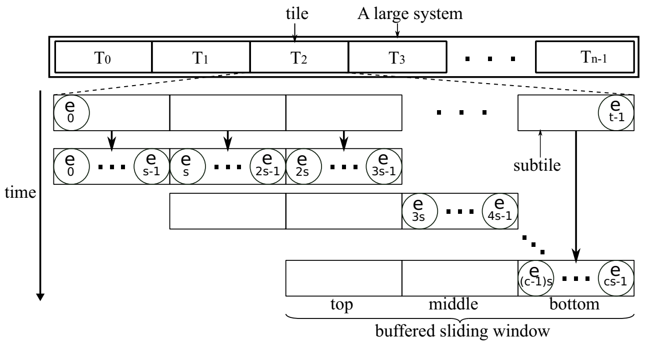


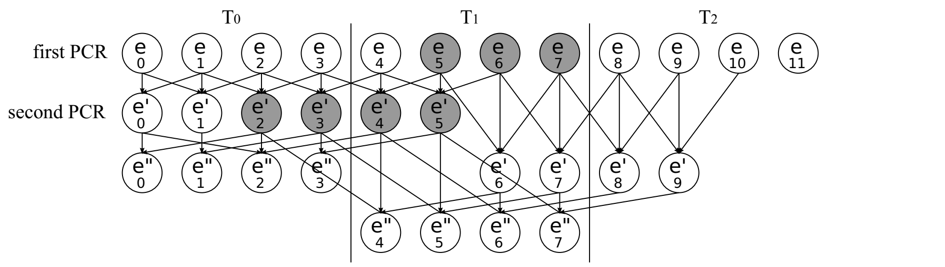
Figure 9. Illustration of the buffered sliding window

We used a technique called *buffered sliding window* for using shared memory to process a tile efficiently as shown in Figure 9. While different tiles are processed in parallel, each tile manages the buffered sliding window and processes sub tiles with the cached dependency. The buffered sliding window is composed of three parts: top and bottom buffers, each of which has sub tile size, and a middle buffer whose size is twice of that of a sub tile. The top buffer, in which the input elements have gone through full PCR steps, caches elements from the middle buffer for the last step of PCR. The middle buffer mostly interacts with the elements in the bottom buffer by providing dependency to them at the same time referring them. In the bottom buffer, the input elements are just loaded from global memory and ready to be processed. A single memory object is allocated on shared memory in order to host these buffers, though logically segmented as such. Having one big memory block over separate blocks for cache contents is desired as it allows the PCR elimination kernel to work across logical buffer boundaries.

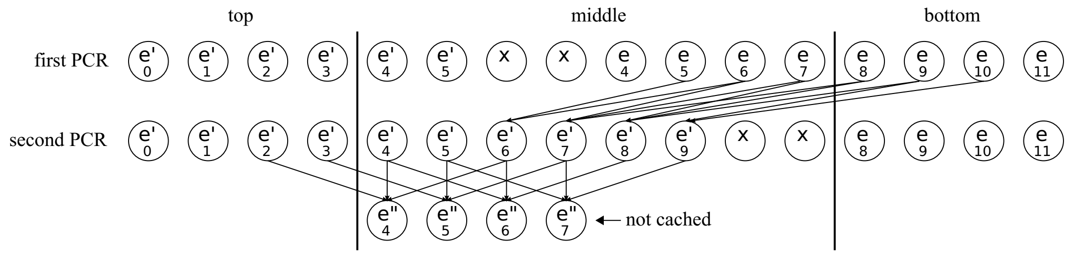
Figure 10 illustrates an example of how the buffered sliding window is used to generate e''_4 to e''_7 out of two-step PCR. In (a), dependency to generate the output is shown, where cached immediate dependency is depicted as grey circles. After lead-in stage to fill the cache, e_5 to e_7 are located at the end of the middle buffer and the first PCR step can work with e_5 to e_{10}. Its output, e'_6 to e'_9, are written back right next to e'_5 so that the next PCR step can also work on linear address space. The last PCR step, the second PCR in this particular case, can be done using e'_2 to e'_9 where the top buffer contributes by providing second half of its contents, e'_2 and e'_3. Note that the final output, e''_4 to e''_7, are not cached as they are not used as dependency for later computations. In (c), it shows the contents of the sliding buffered window ready to process the next sub tile and this is simply done by shifting the contents of the buffers by the size of sub tile.

The buffered sliding window implements several performance optimizations. The overall capacity of the cache, the top and middle buffers, is $3 \cdot f(k)$ which is larger than $2 \cdot f(k)$, the minimum requirement as we discussed before. There are several cases the added margin helps to solve critical performance limiting factors as follows. First, an issue with coalesced memory access is revealed from Figure 8 (b), where the output is not aligned with tile boundary which could break the coalescing rule provided that each thread is assigned to produce one output. The buffered sliding window properly deals with such situation by shifting input or output within the margin to get them aligned. In this particular case, it can be solved by shifting the computation boundary by caching e_5 as shown in Figure 10 (a). Second, the increased capacity allows the cache layout to have padding, denoted as x, which enables efficient cache management when combined with an offset to the start address of where the output is written to. For instance, the output of the first PCR overwrites the middle buffer starting from the padding, and later e'_6 and e'_7 will become the top buffer and e'_8 and e'_9 will take place of e'_4 and e'_5 by shifting the buffer contents to the left. Actual cost devoted to the cache management is thus copying buffers from one to another upon completion of full PCR operation for a sub tile.

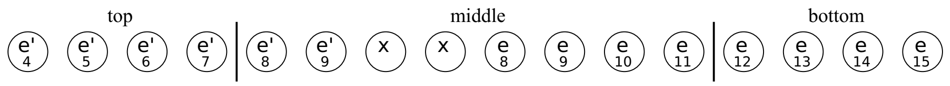
Table I shows properties of the buffered sliding window assuming k steps PCR. A GPU thread block is mapped to perform PCR steps on each tile in which it iterates k steps PCR for each of sub tiles. As shown in the Figure 10, an important optimization is to achieve the maximum parallelism within a thread block in which all the threads perform full PCR steps together until the final step, which is why the number of threads per thread block equals to 2^k. In the previous example where two step PCR is assumed, a thread block is composed of four threads where each thread performs two elimination steps, and the size of a sub tile is four, assuming each thread generates one output per a sub



(a) Dependency to generate e''4 to e''7 from three sub tiles



(b) Lead-in of buffered sliding window with two sub tiles(T0, T1) and processing T2 with that



(c) After processing T2 is done and shifted by a sub tile

Figure 10. An example of the buffered sliding window used to produce e''_4 to $''_7$

Table I
PROPERTIES OF THE BUFFERED SLIDING WINDOW

number of PCR steps	k
size of sub tile	$c2^k$, where $c \geq 1$
size of intermediate results cache	$3 \cdot \sum_{i=0}^{k-1} 2^i \leq 3 \cdot 2^k$
number of threads per thread block	2^k
number of elimination steps per thread	ck
number of elimination steps per sub tile	$ck2^k$

tile, when c equals to 1.

The proposed tiled PCR implementation provides several advantages as follows. First, the tiled PCR has a smaller footprint than the previous PCR approaches. The tiles can be easily accommodated by shared memory that comes with most GPUs available today. Compared to previous works, it enables higher occupancy and as such larger number of thread blocks can be scheduled per SM. It also overcomes the size limitation of shared memory from which conventional PCR implementations on GPUs suffer. Second, with higher occupancy and finer grained memory accesses from global memory, tiled PCR exhibits better memory latency hiding resulting in less idle time of GPU execution units. Section C shows two strategies for increasing the number of sliding windows and thus increasing the probability of having at least one bottom buffer ready while others are being loaded. In contrast, in coarse-grained tiling [16][17][19], a significant cost of synchronization to manage shared memory has to be imposed from a large number of threads in a thread block. Third, maintaining all the required intermediate values in the buffered sliding window minimizes redundant global memory accesses. In terms of the number of global memory access, compared to the coarse-grained tiling, our method is no worse than loading the whole system to shared memory at once. Compared to naive fine-grained tiling, our method can remove a huge amount of redundant global memory accesses attributed to loading halo elements as well as redundant elimination steps as we cache and reuse intermediate values. Finally, tiled PCR can run with a system of arbitrary size due to the tiling technique. The ability to keep the number of PCR steps under control expands the portability of our method to virtually all GPUs.

B. Thread-level Parallel Thomas Algorithm

This stage solves multiple independent systems using Thomas algorithm such that each thread solves a different system. One important concern in this stage is the memory bandwidth due to low memory-to-computation ratio. In order to best utilize the memory bandwidth of GPUs, memory accesses to global memory should be coalesced. This can be achieved when we interleave multiple systems in global memory. Fortunately, PCR naturally produces interleaved results which is perfect match with p-Thomas algorithm. As shown in Figure 6, the interleaving enables consecutive memory accesses for threads in a thread block which results in coalesced memory accesses.

C. Further Optimizations for GPUs

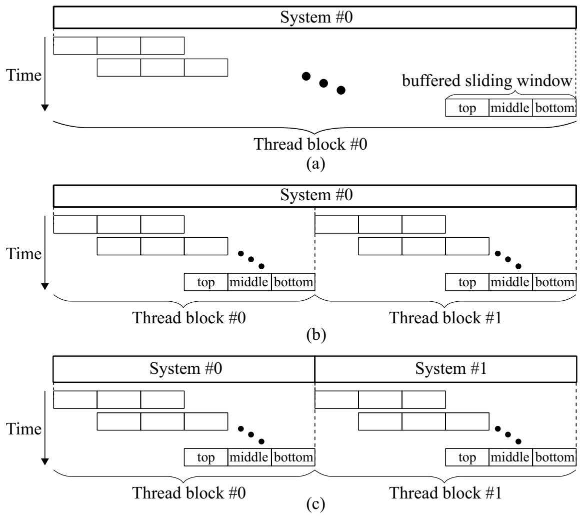


Figure 11. Variants of tiled PCR configuration upon mapping systems onto different number of thread blocks

Several variants of configuration are feasible with the basic idea of tiled PCR and are summarized in Figure 11. In Figure 11(a), one system is assigned to a thread block, and in this case the thread block will need to iterate the single buffered sliding window. When the whole input can fit into shared memory, our method no longer manages the buffered sliding window and it reduces to [16][17]. Another configuration in Figure 11(b) is that one large system is mapped to a group of thread blocks. With this configuration, more than one buffered sliding window would run concurrently. The tradeoff here is that each thread block needs to load overlapping tiles in the boundary of regions, which issues redundant global memory accesses. Note that our method naturally reduces to 'in-shared memory' PCR as [19] when the input system fits shared memory. Lastly, Figure 11(c) shows one thread block processing multiple systems with multiple window by multiplexing several systems. The benefit of this configuration is being able to hide more memory latency with perfectly permutable independent global memory accesses and computations.

Tiled PCR and p-Thomas can be combined into one kernel though they are radically different. The idea is progressively invoking p-Thomas without waiting for tiled PCR kernel to finish processing the whole data. The resulting effect is that p-Thomas can work with partial data that becomes available while tiled PCR kernel executes. In doing so, results out of the PCR step will be directly consumed by combining them with the previous results, which are in registers, using forward reduction in p-Thomas algorithm. Then the updated partial result is stored in the same registers that are used for the previous results, while the previous results are written to global memory. Register tiling is used here not only to provide high bandwidth but also save expensive global memory accesses. Also, the cost due to prior kernel termination and post kernel invocation can be removed.

However, kernel fusion does not always improve performance. In particular, the tiled PCR kernel tends to have lower occupancy than the p-Thomas kernel due to its heavier use of shared memory. Once fused, the number of parallel threads within a thread block is bound to the lower number of the two kernels and in turn the parallelism of p-Thomas might be limited. Thus, kernel fusion should be carefully used when a large number of parallel workload is envisioned.

D. Algorithm Transition from tiled PCR to p-Thomas

Having observed that one single algorithm cannot cope with all combinations of hardware and input sizes, we need to have control over the algorithm to pick the best combination depending on the situation. This is based on a relation between the amount of parallelism the hardware can support and potential workload sizes.

We take the aforementioned criteria and formulate that into functions of computation cost with respect to several significant factors. The cost function tells us how many elimination steps are involved. Here we assume k-step PCR on M independent systems, each of which being a 2^n-element linear system, running on a P-way parallel machine. When M is larger than P, the minimum is when k equals to zero and our hybrid method will directly proceed to p-Thomas algorithm. In a case where M is smaller than P, the minimum would be when k is the maximum possible value such that $2^k \cdot M \leq P$. Table II summarizes the computation cost function of relevant methods with various conditions.

When the input size is 2^n, the number of elimination steps in Thomas is $2 \cdot 2^n - 1$. In Thomas, M means degree of parallelism, which is why for $M \leq P$ the total execution time does not change. When $M > P$, the workload saturates the parallelism so the total amount of workload can be amortized. In PCR, the system is broken down as it continues so the workload can be divided by the number of available parallelism in any case. Our method is combination of the two. $k \cdot 2^n$ indicates elimination cost of PCR and $2 \cdot 2^{n-k} - 1)$ is due to Thomas. With larger degree of parallelism compared to hardware capability, the computation cost can be divided by P. However, P-Thomas still underutilizes the parallelism with the resulting system in case of $2^k \cdot M \leq P$ after PCR.

In practice, the parallelism a GPU provides is not intuitively identifiable as it depends on several factors, some of which are non-linear due to sharing of hardware resources. Also, the closed-form solution cannot be easily expressed and found during runtime. Instead, we present empirical heuristic values that are optimized on NVidia GTX480

Table II
COMPUTATION COST COMPARISON OF THOMAS, PCR AND THE PROPOSED METHOD

algorithm	$M > P$	$M \leq P$
Thomas	$\frac{M}{P}(2 \cdot 2^n - 1)$	$2 \cdot 2^n - 1$
PCR	$\frac{M}{P}(n \cdot 2^n + 1)$	$\frac{M}{P}(n \cdot 2^n + 1)$
k-step tiled PCR + p-Thomas	$\frac{M}{P}\{2(2^n - 2^k) + k \cdot 2^n\}$	$\frac{M}{P} k \cdot 2^n + \frac{M}{P} 2(2^n - 2^k)$, when $2^k \cdot M > P$ $\frac{M}{P} k \cdot 2^n + 2(2^n - 2^k)$, when $2^k \cdot M \leq P$

Table III
HEURISTIC VALUES OF K-STEP FOR A VARIOUS RANGES OF INPUT SIZE ON GTX480

M	k-step	Tile size(2^k)
$M < 16$	8	256
$16 \leq M < 32$	7	128
$32 \leq M < 512$	6	64
$512 \leq M < 1024$	5	32
$1024 \leq M$	0	1

which is shown in Table III. Finding proper values for different situations can be done only once and the effort can be quickly amortized afterwards.

IV. PERFORMANCE EVALUATION

The result we provide in this section was generated using a system with NVidia GTX480 graphics processor with 1.5GB of memory and a 3.33GHz Intel quad-core i7 975 processor with hyper-threading enabled, running Fedora 12 Linux as its operating system. We benchmark our implementation and compare our result with the multithreaded and sequential Intel MKL libraries on the same system, with various combinations of number of systems and system sizes, denoted as M and N, respectively. Our GPU implementation is written in CUDA and compiled using nvcc 3.2, however, it can easily be ported to OpenCL. The CPU implementation we compare against is compiled using the Intel C/C++ compiler(icc). Note that the out of the box tridiagonal solver in Intel MKL does not support multi-threading. Therefore, the CPU implementation becomes multi-threaded only when there are two or more independent systems to be solved ($M \geq 2$).

First, we demonstrate scalability with respect to the number of systems by showing the result from varying M for a selected N. The result is shown in Figure 12. In the CPU implementations, an obvious relation can be found between the execution time and the input size, which is perfectly linear. In contrast, our method shows close results compared to the CPU implementations when M is small, however, as M grows it outperforms demonstrating significant speedups. Our method in double precision achieves up to 8.3x and 49x speedups over multi-threaded and sequential MKL library, respectively, when N is 512.

There are points in the performance trend where the curve of our method changes when M is smaller than 1,024. This is attributed to the decreased number of PCR steps as M changes according to Table III. When M is smaller than 512, the performance is dominated by tiled PCR and even with increased workloads the execution time growth is sub-linear due to underutilizing thread level parallelism. Similarly, a flat region can be found when M is between 512 and 4,096 where additional workload does not actually increase the execution time much. This is due to the imbalance of the parallel workload and the processing power of the GPU. In particular, with smaller number of parallel threads, the long latency of global memory load is mostly exposed and it directly adds to the execution time. An increased number of systems leads to larger number of parallel threads and consequently long latency of memory loads and arithmetic operations can be hidden. Once the latency is completely hidden, which happens M is 4,096 or greater, the execution time increases linearly and our method shows good scalability. Saturating the thread level parallelism of GPUs needs significant amount of independent workloads and this is a strong advocate to our hybrid method in which the frontend excavates parallelism and the backend runs with divided subproblems. With single precision, we achieved 12.9x and 82.5x speedups over the CPU implementations showing similar performance trend, though this is not shown in the graph.

Next, we changed the benchmark such that it tests scalability regarding the input system size. Figure 13 shows the performance results in double precision with various matrix sizes for a few selected input systems. For this experiment, we picked 1, 16, 256 and 2,048 for M, each of which incurs different number of PCR steps. With large M, as expected from the previous experiment, our method performs well against the CPU counterparts, achieving up to 5x and 30x speedups over multi-threaded and sequential MKL library, respectively, when M is 2,048. As M gets smaller, the performance gap between our implementation and the best CPU implementation shrinks. This is because the reduced

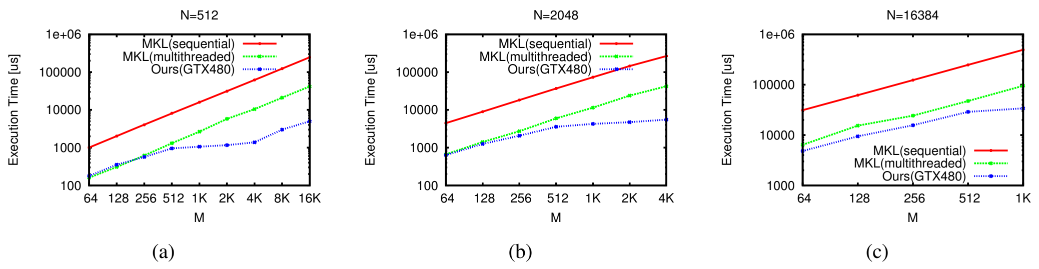


Figure 12. Performance results of varying number of systems with fixed input sizes in double precision

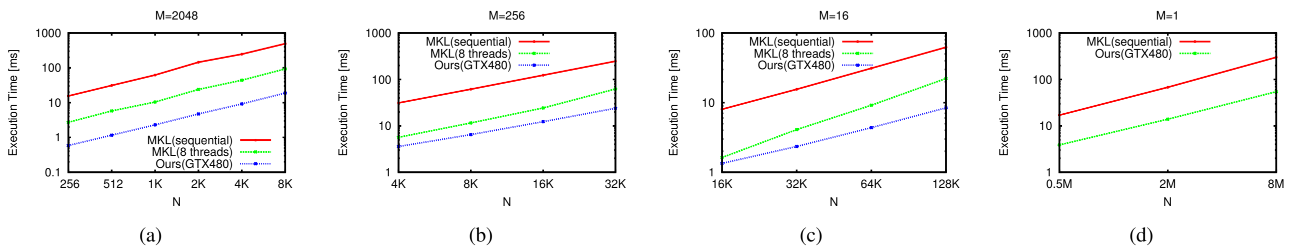


Figure 13. Performance results of varying input sizes on a few fixed number of systems in double precision

parallelism prompts our method to increase its reliance on PCR. For M is 2,048 the kernel runs p-Thomas only. Though it is not shown in the graph, the portion of tiled PCR in total execution time is 6.25% and 36.2% for M is 256 and 16, respectively. Even in the worse case with a test on a single system with a very large input size, our method consistently shows around 5.5x speedup in all test cases. In this case, tiled PCR contributes roughly 55% of the total execution time.

V. Comparison and Discussion

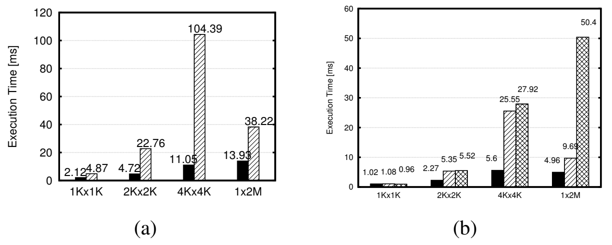


Figure 14. Execution time comparison between Davidson's method and our method. (a) comparison of double precision execution time between the proposed method (left bar) and our implementation of Davidson et al. (right bar). Note that Davidson et al. did not report double precision results (b) comparison of single precision execution time between the proposed method (left bar), our implementation of Davidson et al. (middle bar) and that reported in Davidson et al. (right bar)

We compare the performance results of our method against the work of Davidson et al. [19] as both propose a hybrid of PCR-Thomas. Our method shows better performance than their implementation, showing 2x to 10x speedup for most of the cases, as shown in Figure 14, for the following reasons. First, they divide a large system into coarse-grained tiles so that each chunk maximally occupies shared memory for PCR. As discussed in Section III.A, coarse-grained tiling suffers from large shared memory requirement, fewer concurrent thread blocks, and exposed latency for loading data from global memory. Second, in their work each PCR step is performed in lockstep until the size of reduced input fits in shared memory so that a hybrid of PCR and p-Thomas can work with the reduced input on shared memory. The benefit of this approach is that in each step of PCR the complete immediate dependency is available. However, in order to perform a PCR step for a large input system that is tiled and parallelized on multiple thread blocks, a kernel must be globally synchronized which entails expensive kernel termination and relaunch. Similarly, a moderately divided input system is mapped on a thread block before the hybrid of PCR and p-Thomas runs, where synchronization of threads within a thread block is also required at each step of PCR which hurts the performance.

VI. Conclusions

In this study we proposed a robust tridiagonal matrix solver on GPUs. The proposed method is a hybrid of tiled PCR and p-Thomas algorithm, in which tiled PCR divides a large system into multiple systems and parallel Thomas follows. Sophisticatedly designed tiling and the buffered sliding window in our tiled PCR is the key enabler for it to be used as a parallelism excavating frontend. In particular, the value of tiled PCR is not just being able to process a large system that most of the conventional approaches can not.

It also minimizes redundant global memory access, hides memory access latency due to independent tiling of a large system, and provides desirable memory layout so that the following p-Thomas algorithm can achieve a high rate of coalesced memory access.

We performed experiments with various combinations of different input sizes and number of input systems. Our method shows 8.3x and 49x speedups over multithreaded and sequential CPU implementations, respectively. The results indicate that our method is scalable when either the input size or the number of system changes. The experimental results justified our view of two phase tridiagonal solver which combines efficient frontend to divide a system into multiple parallel sub-systems and a computationally efficient algorithm to deal with multiple independent systems. We also found that determining transition point from tiled PCR to p-Thomas plays very important role to balance the amount of parallel workloads and the capability of the underlying hardware.

The buffered sliding window approach can also be applied to other types of divide-and-conquer type algorithms. Future work includes further developing the approach into a generalized strategy and optimizing other applications using the ideas proposed in this paper.

ACKNOWLEDGMENTS

We are grateful to John D. Owens, Yao Zhang and Andrew Davidson for invaluable feedback which helped us to significantly improve this paper. We also would like to thank the anonymous referees for their many helpful comments and suggestions. The authors acknowledge the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

REFERENCES

[1] M. Kass, A. Lefohn, and J. Owens, "Interactive depth of field using simulated diffusion on a GPU," Tech. Rep. #06-01, Pixar Animation Studios, Jan. 2006. http://graphics.pixar.com/library/DepthOfField.

[2] M. Kass and G. Miller, "Rapid, stable fluid dynamics for computer graphics," in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, (New York, NY, USA), pp. 49–57, ACM, 1990.

[3] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*, pp. 97–106, Aug. 2007.

[4] N. Sakharnykh, "Tridiagonal solvers on the GPU and applications to fluid simulation," NVIDIA GPU Technology Conference, September 2009.

[5] N. Sakharnykh, "Efficient tridiagonal solvers for ADI methods and fluid simulation," NVIDIA GPU Technology Conference, September 2010.

[6] R. W. Hockney, "A fast direct solution of Poisson's equation using Fourier analysis," *J. ACM*, vol. 12, pp. 95–113, January 1965.

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

[8] L. Chang, M. Lo, N. Anssari, K. Hsu, N. Huang, and W. Hwu, "Parallel implementation of multi-dimensional ensemble empirical mode decomposition," *International Conference on Acoustics, Speech, and Signal Processing*, May 2011.

[9] M. Prieto, R. Santiago, D. Espadas, I. M. Llorente, and F. Tirado, "Parallel multigrid for anisotropic elliptic equations," *J. Parallel Distrib. Comput.*, vol. 61, pp. 96–114, January 2001.

[10] D. Göddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 22–32, 2011.

[11] S. D. Conte and C. W. D. Boor, *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill Higher Education, 3rd ed., 1980.

[12] R. W. Hockney and C. R. Jesshope, *Parallel computers : architecture, programming and algorithms / R.W. Hockney, C.R. Jesshope*. Hilger, Bristol :, 1981.

[13] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," *J. ACM*, vol. 20, pp. 27–38, January 1973.

[14] D. Egloff, "High performance finite difference PDE solvers on GPUs." http://download.quantalea.net/fdm_gpu.pdf, Feb. 2010.

[15] D. Egloff, "Pricing financial derivatives with high performance finite dierence solvers on GPUs," in *GPU Computing Gems*, in press.

[16] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, (New York, NY, USA), pp. 127–136, ACM, 2010.

[17] Y. Zhang, J. Cohen, A. A. Davidson, and J. D. Owens, "A hybrid method for solving tridiagonal systems on the GPU," in *GPU Computing Gems*, in press.

[18] A. Davidson and J. D. Owens, "Register packing for cyclic reduction: A case study," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, Mar. 2011.

[19] A. Davidson, Y. Zhang, and J. D. Owens, "An auto-tuned method for solving large tridiagonal systems on the GPU," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, May 2011.