A Scalable, Numerically Stable, High-performance Tridiagonal Solver using GPUs

Li-Wen Chang, John A. Stratton, Hee-Seok Kim, and Wen-Mei W. Hwu
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
{lchang20, stratton, kim868, w-hwu}@illinois.edu

***Abstract*—In this paper, we present a scalable, numerically stable, high-performance tridiagonal solver. The solver is based on the SPIKE algorithm for partitioning a large matrix into small independent matrices, which can be solved in parallel. For each small matrix, our solver applies a general 1-by-1 or 2-by-2 diagonal pivoting algorithm, which is also known to be numerically stable. Our paper makes two major contributions. First, our solver is the first numerically stable tridiagonal solver for GPUs. Our solver provides comparable quality of stable solutions to Intel MKL and Matlab, at speed comparable to the GPU tridiagonal solvers in existing packages like CUSPARSE. It is also scalable to multiple GPUs and CPUs. Second, we present and analyze two key optimization strategies for our solver: a high-throughput data layout transformation for memory efficiency, and a dynamic tiling approach for reducing the memory access footprint caused by branch divergence.**

***Index Terms*—GPU Computing, GPGPU, Tridiagonal Solver, SPIKE**

I. Introduction

The tridiagonal solver is a very important component in a wide range of engineering and scientific applications, including computer graphics [1][2], fluid dynamics [3][4], Poisson solvers [5], preconditioners for iterative solvers [6], cubic spline interpolation [7], and semi-coarsening [8] for multi-grid methods. Many of these applications require solving several independent tridiagonal linear systems, with the number of systems varying greatly between different applications. For example, a multi-dimensional problem tends to have a large number of independent systems, while a one-dimensional problem often results in a small number of independent systems. In this paper, we are particularly interested in solving only one or a few large systems for several reasons. First, a set of independent systems can always be expressed as a single large system (by uniting those matrices into a large matrix with some zero sub-diagonal elements), but not vice-versa. A parallel solver that performs well even for a single system is therefore the most generally applicable, and can even handle a set of irregularly sized systems with no additional complexity. Second, a single system is the most difficult case for a parallel solver implementation, because the problem has no inherent independence to exploit from disjoint systems. Finally, even in applications where there are several huge systems to solve, the limited memory size of the graphics processing unit (GPU) encourages a solution where the GPU processes a large portion of a single system at a time. Assuming that the systems are all large enough to require partitioning, the amount of communication and transfer overhead will grow with the number of times each system is partitioned. The number of partitions will grow directly in proportion to the number of unrelated systems being solved simultaneously on the same device. All else being equal, we should then prefer to send large integrated systems to the GPU for minimal transfer overhead. Internally, the GPU solver is then free to perform further partitioning as necessary to facilitate parallel execution, which does not require external communication.

Recently, high-performance GPU architectures have been widely used for scientific computation due to their high computational throughput and large memory bandwidth. However, only few GPU-based tridiagonal solvers can efficiently solve a small number of large systems. Davidson et al. [9] and Kim et al. [10] both proposed solvers using a hybrid of parallel cyclic reduction (PCR) [11] and the Thomas algorithm [12] with intelligent decision mechanisms for switching from PCR to Thomas to balance parallelism and computational complexity. NVIDIA released a tridiagonal solver using a hybrid of cyclic reduction (CR) [11] and PCR algorithms in CUSPARSE library [13]. Several other GPU-based tridiagonal solvers are also notable, although they do not directly support use cases with a small number of large systems. Sengupta et al. [2], Göddeke et al. [8], and Davidson et al. [14] proposed tridiagonal solvers for GPUs by using CR, while Egloff [15][16] developed a PCR-based solver. Sakharnykh implemented thread-level parallel Thomas algorithm [3] and introduced a hybrid of PCR-Thomas algorithm [4]. Zhang et al. [17][18] proposed a hybrid technique among the Thomas algorithm, CR, PCR, and recursive doubling (RD) [19].

In general, most previous GPU-based tridiagonal solvers fell short in solving a small number of large systems due to one or both of the following two reasons. The first is that their parallel execution is based on the inherent massive parallelism of a large number of simultaneous systems. The second reason is that they require that the data of each system fit into the high bandwidth of scratchpad memory. Since these scratchpad memories are of modest size, these solvers fail when each system is large. These solvers may work well in some applications, but cannot be used as a general library. Most importantly, to the best of our knowledge, all existing

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

GPU-based solvers exhibit numerical instability. CR, PCR, RD, and the Thomas algorithm are all known to be numerically unstable for some distributions of matrix values, which we will demonstrate in our results. Tridiagonal solver algorithms known to be numerically stable, such as diagonal pivoting or Gaussian elimination with partial pivoting, are less naturally suited to GPUs. None are inherently data-parallel in each step, like CR or PCR. Even if we were to assume a large number of independent systems, numerically stable algorithms tend to rely heavily on data-dependent control flow, which can cause divergence and reduce GPU performance significantly.

This paper introduces several contributions to the field of tridiagonal matrix solvers, including:

- A numerically stable GPU tridiagonal solver based on the SPIKE algorithm and diagonal pivoting
- High-performance data marshaling techniques to interface the SPIKE decomposition with the thread-parallel solver
- A dynamic tiling scheme for controlling inherent divergence in the diagonal pivoting solver

These contributions together are used to produce the first numerically stable tridiagonal solver library that can utilize both the CPU and GPU computational elements of a distributed-memory cluster.

We first describe the first numerically stable tridiagonal solver for GPUs, based on the SPIKE partitioning algorithm [20][21] and 1-by-1 or 2-by-2 diagonal pivoting [22] for solving each partition. The SPIKE algorithm was designed to decompose a banded matrix into several smaller matrices that can be solved independently, and is widely used on CPU-based parallel computers. The solution is completed by a thread-parallel solver for the many partitioned systems. The standard input and output data formats for a tridiagonal solver (e.g. those of LAPACK's `gtsv` function) are incongruent with good memory access patterns for GPU memory bandwidth for a typical thread-parallel solver for partitions of the tridiagonal system. A fast GPU data layout transformation method [23][24] is applied to ensure the various kernels each get the most bandwidth possible from the GPU memory system.

Next, as mentioned previously, a thread-parallel diagonal pivoting kernel has heavily data-dependent control flow, resulting in thread divergence and widely scattered memory accesses that render GPU caches nearly useless. Initial performance results revealed a slowdown of up to 9.6x compared to existing solvers based on a parallel Thomas algorithm. Although such divergence can never be eliminated, we introduce a dynamic tiling technique that can keep divergence under control so the caches of modern GPUs deliver much more benefit, resulting in competitive performance.

After that, we further extend our GPU-based solver to support multi-GPUs and heterogeneous clusters (CPUs+GPUs) by applying MPI (Message Passing Interface) [25], OpenMP [26] and working with corresponding CPU-based implementations (Intel MKL [27] `gtsv` in this paper). We present three evaluations in our experimental results: numerical stability test, performance test for single GPU, and scalability test for multi-GPUs and clusters. All experiments in this paper are evaluated in double precision, and it is easy to support other precisions. In the numerical stability test, we compare our GPU-based results with CUSPARSE for GPU, and Intel MKL, Intel SPIKE [28], and Matlab [29] for CPUs. In the single GPU performance test, we compare our GPU-based methods with only CUSPARSE, and use Intel MKL as the reference. In the scalability test, we show performance scaling for both our GPU-based solver and our heterogeneous solver, using Intel SPIKE as the reference on a compute cluster with CPU+GPU nodes. Table I summarizes the libraries we evaluate. ✓ means good performance or supported, whereas ✗ means poor performance or not supported.

TABLE I
SUMMARY FOR TRIDIAGONAL SOLVERS

Solvers	Numerical Stability	High CPU Performance	High GPU Performance	Cluster scalability
CUSPARSE (`gtsv`)	✗	✗	✓	✗[a]
MKL (`gtsv`)	✓	✓	✗	✗
Intel SPIKE	✓	✓	✗	✓
Matlab (backslash)	✓	✗	✗	✗
Our GPU solver	✓	✗	✓	✓
Our heterogeneous MPI solver	✓	✓	✓	✓

[a]CUSPARSE currently supports only a single-GPU solver, but the Cyclic Reduction algorithm it uses could be extended to multiple GPUs.

In the following sections, we use NVIDIA CUDA terminology [30], although our methods could just as easily apply to OpenCL [31]. Individual functions executed on the GPU device are called "kernel" functions, written in a single-program multiple-data (SPMD) form. Each instance of the SPMD function is executed by a GPU "thread". Groups of such threads, called "thread blocks", are guaranteed to execute concurrently on the same processor. Within each group, subgroups of threads called "warps" are executed in lockstep, evaluating one instruction for all threads in the warp at once. We will introduce other architecture and programming model constructs throughout the paper as necessary.

II. SPIKE ALGORITHM FOR SOLVING LARGE TRIDIAGONAL SYSTEMS

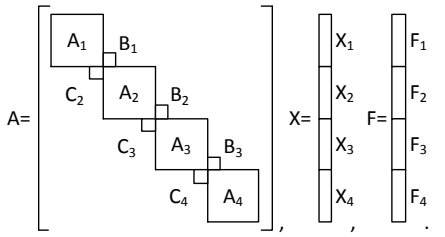


Fig. 1. Data partitioning for SPIKE Algorithm

The SPIKE algorithm decomposes a single system of equations described by a banded matrix into a set of disjoint systems that can be solved independently. A tridiagonal system is a special case of a banded system, allowing us to use the SPIKE algorithm to partition the problem. This section describes a specialized implementation of the SPIKE algorithm for tridiagonal matrices using GPUs. In algebraic terms, we are solving for X in Eq 1.

$$AX = F, \tag{1}$$

where A is a tridiagonal matrix, and X and F are vectors. The matrix A can be partitioned in several square diagonal blocks A_i and a collection of off-diagonal single elements B_i and C_i, as shown in Figure 1. For this particular example, we are partitioning the matrix A into four blocks. Similarly, X and F can also have corresponding partitions, each partition of which has a length equal to the side-length of the tile A_i. We can algebraically isolate the block-diagonal components of A by defining matrices D and S such that

$$A = DS. \tag{2}$$

D is simply the isolated diagonal blocks of A, while S has the structure shown in Figure 2, with values constrained by Eq 3 and Eq 4.

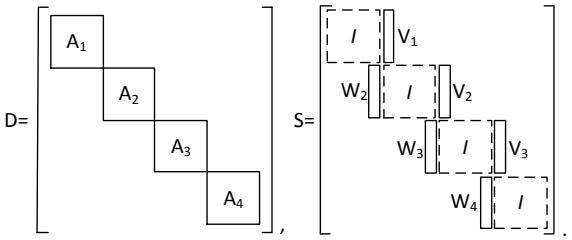


Fig. 2. Matrices D and S

$$A_i V_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ B_i \end{bmatrix} \tag{3}$$

$$A_i W_i = \begin{bmatrix} C_i \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4}$$

Using this formulation, we can substitute Eq 2 into Eq 1 and reassociate the operations, resulting in Eq 5 and Eq 6.

$$SX = Y \tag{5}$$

$$DY = F \tag{6}$$

Since D is block-diagonal, the tiles of the system are independent, so Eq 6 can be restated as a the collection of independent systems as shown in Eq 7.

$$A_i Y_i = F_i \tag{7}$$

In Eq 7, each A_i and F_i are known, allowing us to solve for each Y_i. Similarly, in Eq 3 and Eq 4, A_i, B_i and C_i are known, allowing us to solve for V_i and W_i. The solutions of these three equations can be computed in a procedure and all partitions can be computed in parallel.

Having computed all components of Y and S, we can solve for the final solution vector X using Eq 5. The particular structure of the matrix S, makes the solution the Eq 5 simpler than it may appear at first. Specifically, most of the values can be computed simply by back-substituting the solution to the smaller system defined in Eq 8.

$$\widehat{S}\widehat{X} = \widehat{Y}, \tag{8}$$

$$\widehat{S} = \begin{bmatrix} 1 & 0 & V_1^{(t)} & & & & & \\ 0 & 1 & V_1^{(b)} & & & & & \\ & W_2^{(t)} & 1 & 0 & V_2^{(t)} & & & \\ & W_2^{(b)} & 0 & 1 & V_2^{(b)} & & & \\ & & & W_3^{(t)} & 1 & 0 & V_3^{(t)} & \\ & & & W_3^{(b)} & 0 & 1 & V_3^{(b)} & \\ & & & & & W_4^{(t)} & 1 & 0 \\ & & & & & W_4^{(b)} & 0 & 1 \end{bmatrix},$$

$$\widehat{X} = \begin{bmatrix} X_1^{(t)} \\ X_1^{(b)} \\ X_2^{(t)} \\ X_2^{(b)} \\ X_3^{(t)} \\ X_3^{(b)} \\ X_4^{(t)} \\ X_4^{(b)} \end{bmatrix}, \text{ and } \widehat{Y} = \begin{bmatrix} Y_1^{(t)} \\ Y_1^{(b)} \\ Y_2^{(t)} \\ Y_2^{(b)} \\ Y_3^{(t)} \\ Y_3^{(b)} \\ Y_4^{(t)} \\ Y_4^{(b)} \end{bmatrix}. \tag{9}$$

Here, the superscripts (t) and (b) mean the most top and bottom single elements, respectively, of V, W, X, and Y. $\widehat{X}$ and $\widehat{Y}$ can also be defined with top and bottom elements. After $\widehat{X}$ is solved, the remaining unknown values of the vector X (i.e. the middle elements of each partition) can be computed by applying a backward substitution for each partition. The substitutions for each partition are independent from each other, and can be computed in parallel.

In summary, SPIKE algebraically isolates tiles of the matrix A such that a parallel solver can solve many partitions of the larger system independently. This leads us to the computational implementation of the SPIKE algorithm shown in Figure 3.

A. Our SPIKE implementation

The solver begins by creating the partitions of the input matrix and vector, preparing for the parallel solver step. The parallel partitioned solver for the collection of independent systems then computes both the solutions to the partitions of the original system as well as the W and V components of the spike matrix S. The parallel solver constitutes the majority of the computation in the SPIKE algorithm, and we explain our implementation in detail in Section III. Then a solver for the spike system is invoked, first solving the reduced spike system, followed by the computation of all remaining values in the

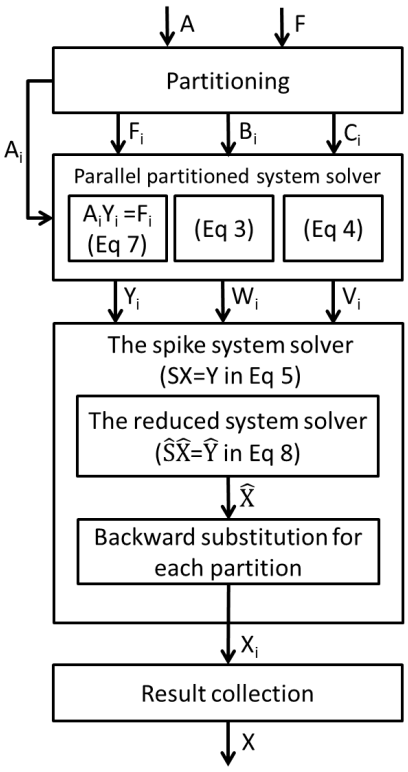


Fig. 3. The SPIKE Algorithm

Partitioning:

- Data partitioning for distributed memory
- Data marshaling (Section III-A) for GPUs

Parallel partitioned system solver:

- Invoke a tridiagonal solver on each compute unit
  - GPU uses diagonal pivoting (Section III)
  - CPU uses Intel MKL with OpenMP

The spike system solver:
The reduced system solver:

- A local reduced system solver on each compute unit
- Collect the reduced systems for distributed memory
- A global reduced system solver for all compute units
- Distribute the reduced solution for distributed memory

Backward substitution:

- Invoke backward substitution on each compute unit

Result collection:

- Data marshaling (Section III-A) for GPUs
- Result collection for distributed memory

Fig. 4. Our SPIKE Implementation

solution X through backward substitution. Finally, the results from all the different partitions are collected and merged into the expected output format and returned. Figure 4 summarizes the functionality of each block in the SPIKE implementation.

In the parallel partitioned system solver step, any solver can be used to solve the independent systems. In our experimental results, we provide both our thread-parallel diagonal pivoting solver (Section III) and a parallel Thomas solver [3][10] for GPUs, and use MKL `gtsv` routine with OpenMP for multi-threaded CPU. In either solver, the thread with Index i solves for Y_i, V_i, and W_i, which all depend on the matrix partition A_i but with different result vectors. The parallel solver step requires the most computation time out of all the steps, and was therefore the focus of more optimization efforts. Section III will describe the most important optimizations, data layout transformation and dynamic tiling, for the diagonal pivoting solver. Data layout transformation was also applied for the Thomas algorithm solver, but dynamic tiling was irrelevant because the Thomas implementation has no thread divergence.

In the spike system solver step, solving the reduced spike system in Eq 8 only occupies less than 5% of the overall solver runtime in our final library. As such, we will only briefly describe its implementation here, and refer the reader to the work of Pollizi et al. [20] for more details. If the number of partitions is small enough, the reduced spike system can be easily solved by collecting all top and bottom elements from partitions, and using a direct solver (a sequential method in most cases) in one node. We apply this method, called the explicit SPIKE method, in the global communication step any time our solver is using more than one node or compute device (CPU or GPU). However, if the number of partitions and the size of the parallel computing system is large, the sequential direct solver may dominate the overall solver's runtime. Since our GPU implementation creates thousands of partitions, we employ a recursive SPIKE method on the GPU (specifically, an "on-the-fly" recursive SPIKE algorithm [20]) to again break the spike system into many partitions that can be solved in parallel. At the cluster level, a recursive SPIKE algorithm permits each node to solve locally the partition of the spike system generated locally, minimizing communication.

In the partitioning step, data is prepared and sent to corresponding computing resources. Certain steps may not be necessary depending on the origin of input data. For instance, data may be read from the file system and copied to one or more GPUs on one or more nodes, or may be already resident on the GPU devices as a result of previous computation in a larger application. The transmission of data to GPUs can be handled explicitly with the baseline language support, or implicitly with supplemental libraries [32]. Data marshaling kernels for data layout transformation (Section III-A) are applied at the partitioning step as well. Similarly, in the result collection step, to return data in the original layout, a reverse data marshaling kernel for GPUs and data transfer among nodes are needed.

III. A Stable Tridiagonal Solver with Diagonal Pivoting

Solving the large number of independent systems is the most computationally demanding part of the SPIKE algorithm. We implement a parallel solver with a generalized diagonal pivoting method to achieve a numerically stable result. The diagonal pivoting method for general and asymmetric tridiagonal systems was proposed by Erway et al. [22]. Unlike Gaussian elimination with partial pivoting, it allows pivoting without row interchanges, which can consume precious GPU memory bandwidth.

An asymmetric and nonsingular tridiagonal matrix A can be always defined as

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & \vdots \\ 0 & a_3 & b_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_n & b_n \end{bmatrix} \tag{10}$$

Then A can be decomposed to LBM^T by applying Eq 11 recursively, where L and M are unit lower triangular and B is block diagonal with 1-by-1 or 2-by-2 blocks. The factorization can be defined as follows:

$$\begin{aligned} A &= \begin{bmatrix} B_d & T_{12}^T \\ T_{21} & T_{22} \end{bmatrix} \\ &= \begin{bmatrix} I_d & 0 \\ T_{21}B_d^{-1} & I_{n-d} \end{bmatrix} \begin{bmatrix} B_d & 0 \\ 0 & A_s \end{bmatrix} \begin{bmatrix} I_d & B_d^{-1}T_{12}^T \\ 0 & I_{n-d} \end{bmatrix} \end{aligned} \tag{11}$$

where B_d is a 1-by-1 or 2-by-2 block, and

$$\begin{aligned} A_s &= T_{22} - T_{21}B_d^{-1}T_{12}^T \\ &= \begin{cases} T_{22} - \frac{a_2c_1}{b_1}e_1^{(n-1)}e_1^{(n-1)T} & \text{if d=1} \\ T_{22} - \frac{a_3b_1c_2}{\Delta}e_1^{(n-2)}e_1^{(n-2)T} & \text{if d=2,} \end{cases} \end{aligned} \tag{12}$$

where $\Delta = b_1b_2 - a_2c_1$ and $e_1^{(k)}$ is the first column of the k-by-k identical matrix. Since the Schur complement A_s is also tridiagonal, we can define the factorization recursively, and eventually, form LBM^T.

The dimension of the pivot block B_d (d is 1 or 2) is chosen based on some pivoting criteria [22]. Our implementation uses asymmetric Bunch-Kaufman pivoting [33][22], which chooses a 1-by-1 pivot if the pivoting diagonal entry is sufficiently large relative to adjacent off-diagonal elements, i.e. $|b_1|\,\sigma_1 \geq \kappa|a_2c_1|$, where $\sigma_1 = max\{|a_2|,|a_3|,|b_2|,|c_1|,|c_2|\}$ and $\kappa = \frac{\sqrt{5}-1}{2}$. On the other hand, if the condition is not satisfied, 2-by-2 pivoting is applied.

After factorization, A can be easily solved by applying a forward substitution for lower triangular L, a block-diagonal matrix solver for B, and a backward substitution for upper triangular M^T. Our implementation combines these three solvers into one procedure (Figure 5).

Here, we note two important facts. First, this algorithm has control flow divergence, since the pivoting strategy is chosen dynamically and depends on the data. Second,the algorithm is sequential, since A_s is dynamically calculated and influences later data. Because of the second fact, we apply a thread-level parallel algorithm to let each system be solved by a thread. However, this strategy may cause branch divergence, because of the first fact. An optimization technique called dynamic tiling is proposed to minimize effect of branch divergence in Section III-B.

```
Input:
Lower diagonal array a
Main diagonal array b
Upper diagonal array c
Rhs vector f
Size n

Output:
Vector x

Kernel body:
k=0
while(k<n){
  if(pivoting criteria) {
    //1-by-1 pivoting (d=1)
    gradually solve y in Lz=f and By=z
    form M
    k+=1;
 } else {
    //2-by-2 pivoting (d=2)
    gradually solve y in Lz=f and By=z
    form M
    k+=2;
  }
}
solve x in M^Tx=y;
// This part also contains divergence,
// since the structure of M depends on
// d in each step.
```

Fig. 5. Diagonal pivoting pseudo-code

A. Data Layout Transformation

For the GPU memory system to perform well, the data elements simultaneously accessed by threads in a thread block should be very close together in the memory address space, for good coalescing and cache line usage. However, as we mentioned in Section I, adjacent elements in each diagonal are stored in consecutive locations in the most solver interfaces, such as LAPACK `gtsv`. In that format, simultaneous accesses from threads reading from different partitions will have widely spread addresses. Therefore, in the partitioning step of the SPIKE algorithm, we marshal the data such that Element K of Partition I is adjacent to Element K from partitions I-1 and I+1. This strategy can guarantee better memory efficiency in the kernel execution but pay the cost of data marshaling. Also, output of GPU may require another data marshaling step to move the results back to the layout matching the initial input data before returning the results.

Figure 6 illustrates the relationships between layouts before and after data marshaling. In this example, there are two thread blocks, each containing three threads, each solving a 4-element system. In the initial layout, all coefficients of the same system are placed into consecutive locations, such as b_1, b_2, b_3, and b_4

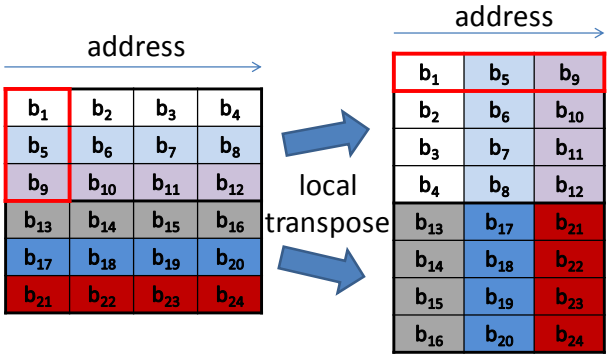


Fig. 6. Illustration for data layout transformation

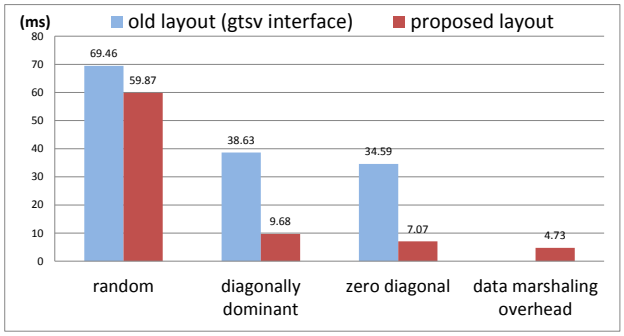
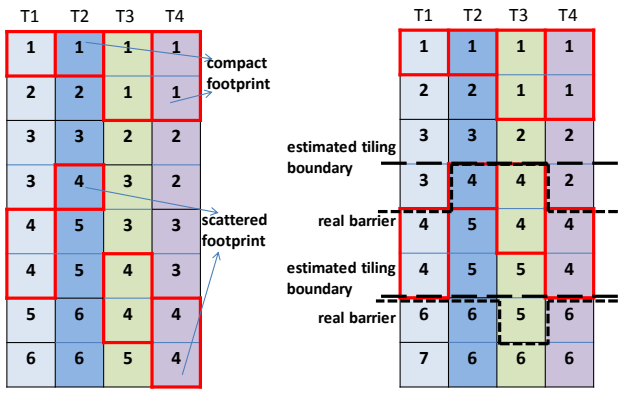


Fig. 7. Performance improvement in parallel partitioned system solver by applying data layout transformation



(a) un-tiled (b) tiled

Fig. 8. Illustration of dynamic tiling

in Figure 6. The threads start by accessing the elements along the left edge, the first elements of their respective partitions, creating a strided access pattern. To create a better access pattern, we perform a local transpose on the data assigned to one thread block. Then, the threads within a thread block accessing element K of their respective partitions will be accessing elements adjacent in the address space. This tiled transpose is analogous to the Array of Structures of Tiled Arrays layout analyzed by Sung et al. [24]; we adapted their general transpose kernel to our specific context.

In Figure 7, we evaluate the performance improvement by comparing three kinds of data sets(8 million row), where the tridiagonal matrix is either randomly generated, strictly column diagonally dominant, or has a zero diagonal with non-zero off-diagonal entries. The random matrix causes strong branch divergence as different pivoting degree are chosen. The strictly column diagonally dominant matrix results in no branch divergence with 1-by-1 diagonal pivoting always selected. The matrix with main-diagonal entries always zero also has no branch divergence, always causing 2-by-2 diagonal pivoting.

Data layout transformation can potentially provide up to 4.89x speedups, because of better memory efficiency. For the input matrices without branch divergence, our new data layout results in perfectly coalesced memory accesses and outperforms the original layout by a factor of 3.99-4.89x. The overheads of data marshaling are also shown in Figure 7. The overhead may be amortized if applications are iterative, or may be partially ignored if users do not care about the order of output. However, it is really application-dependent, so the overhead is counted as a part of our final execution time in our evaluations in Section IV. Even with data marshaling overhead, our new layout still outperforms the original layout in all cases.

B. Dynamic Tiling

On the random matrix, which causes strong branch divergence, our new data layout has only a minimal impact. This is because on each iteration, different threads in a warp consume a different number of input elements based on whether they perform 1-by-1 or 2-by-2 pivoting that iteration. As each thread progresses, choosing 1-by-1 or 2-by-2 pivoting independent of other threads in the warp, the elements required by each thread on a particular iteration become fragmented and uncoalesced. Figure 8(a) shows an illustration where four threads are iterating through elements of their partition in the proposed layout. Elements are marked with the iteration number where the thread assigned to that system consumes that element. On iteration one, the threads touch two "lines" of data. Even if not perfectly coalesced, a cache or other read-combining mechanism will deliver reasonable bandwidth for these accesses. However, as threads consume different numbers of elements on each iteration, by the time they get to iteration four their elements are scattered across five "lines" of data. If the threads get far enough apart, the amount of data the cache needs to hold to avoid evicting lines before all threads consume their elements continues to grow. When the access "footprint" exceeds the cache capacity available to the thread block, memory system performance degrades significantly.

To counteract this problem, we propose a dynamic tiling mechanism, shown in Figure 9, which bounds the size of the access footprint from the threads in a warp. The original while loop is dynamically tiled to a few smaller while loops. A barrier synchronization is put between the smaller while loops to force "fast" threads to wait for "slow" threads. Figure 8(b) illustrates the new footprint after dynamic tiling. At the first barrier synchronization, two fast threads idle one iteration to wait for the other two slow threads. After dynamic tiling, the size of the footprint is controlled by the

(a) Un-tiled

```
k=0
while(k<n) {
  if(condition) {
    1-by-1 pivoting
    k+=1
  } else {
    2-by-2 pivoting
    k+=2
  }
}
```

(b) Tiled

```
k=0
for(i=0; i<T; i++) {
  n_barrier=(i+1)*n/T
  while(k<n_barrier) {
    if(condition) {
      1-by-1 pivoting
      k+=1
    } else {
      2-by-2 pivoting
      k+=2
    }
  }
  barrier for a warp*
}
```

*An implicit barrier synchronization for a warp is automatically generated after the end of a while loop or for loop in CUDA

Fig. 9. Source code difference for dynamic tiling

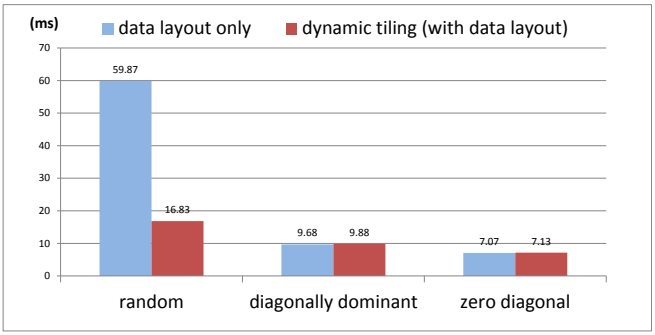


Fig. 10. Performance improvement in parallel partitioned system solver by applying dynamic tiling

tiling parameters, which can be tuned to balance increased computational divergence (from fast threads idling) with the limits on the access footprint size.

Figure 10 shows the performance improvement. Our dynamic tiling strategy provides up to a 3.56x speedup when branch divergence is heavy, and causes minimal overhead when there was no divergence to address. In Figure 11, our performance analysis is further supported by the hardware performance counters from the NVIDIA Visual profiler [34]. The nondivergent datasets exhibited practically perfect memory utilization with or without dynamic tiling, as expected.

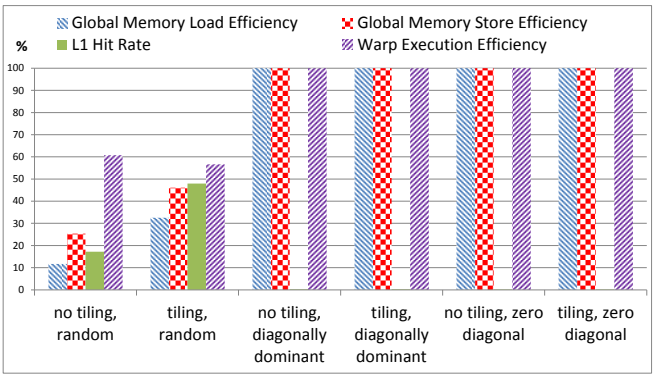


Fig. 11. Profiler counters for dynamic tiling

The L1 hit rate is close to 0% because lines of data are touched only once each, entirely copied to registers in a single instruction. Without dynamic tiling, the random dataset had a global memory load efficiency of only 11.7%, and a global memory store efficiency of only 25.3%. The low hit rate of the L1 cache (17.2%) suggests that the footprint was so large that the cache could not contain it all simultaneously, evicting lines before all the threads could consume their elements from that line. Dynamic tiling improved global memory load efficiency on the random dataset from 11.7% to 32.5%, global memory store efficiency from 25.3% to 46.1%, and L1 cache hit rate from 17.2% to 48.0%, with only a minor decrease in warp execution efficiency. These metrics support our conclusion that dynamic tiling improves performance by more effectively using the hardware cache.

IV. Evaluation

In this section, we evaluate three important features for a high-performance numerical solver: numerical stability, performance, and scalability. Table II lists the specifications of the machines used to collect the data presented here, which we will refer to by name (AC and Forge) for the remainder of the paper.

TABLE II
Machines used for evaluations

Name	Description	Specification
AC	A single node of the NCSA GPU Accelerated Cluster [35]	CPU: 1× Intel Xeon X5680 @3.33GHz Memory: 24GB DDR3 GPU: 2 NVIDIA GTX480 OS: Fedora 12 Compiler: icc 11.1 MKL: 10.2.6 NVIDIA Driver: 295.20 NVCC: 4.1, V0.2.1221 CUSPARSE Version: 4010
Forge	NCSA GPU Forge cluster [36][a]	CPU: 2 AMD Opteron 6128 HE @2.0GHz Memory: 64GB DDR3 GPU: 6 or 8 NVIDIA M2070 OS: Red Hat Enterprise Linux Server 6.1 Compiler: icc 12.0.4 MPI: OpenMPI 1.4.3 MKL: 10.3.4 NVIDIA Driver: 285.05.33 NVCC: 4.0, V0.2.1221 CUSPARSE Version: 4000

[a]The machine specification of Forge is per node.

A. Numerical Stability Evaluation

We test numerical stability of our solver against 16 types of nonsingular tridiagonal matrices of size 512, including ones from Erway et al. [22] and recent literature [37][38]. These matrices are carefully chosen to challenge the robustness and numerical stability of the algorithm. All solvers are tested on AC. Our matrix size is only 512, near the limit of what Matlab would solve with reasonable time and memory. The description and the condition number of each tridiagonal matrix is listed in Table III, while the corresponding relative error of each solver is shown in Table IV. Here, the relative error for a solution $\hat{x}$ is calculated from the following equation:

$$\frac{\|A\hat{x}-F\|_2}{\|F\|_2} \tag{13}$$

TABLE III
MATRIX TYPES USED IN THE NUMERICAL EVALUATION

Matrix type	Condition number	Description
1	4.41E+04	Each matrix entry randomly generated from a uniform distribution on [-1,1] (denoted as `U(-1,1)`)
2	1.00E+00	A Toeplitz matrix, main diagonal is 1e8, off-diagonal elements are from `U(-1,1)`
3	3.52E+02	gallery('lesp',512) in Matlab: eigenvalues which are real and smoothly distributed in the interval approximately [-2*512-3.5, -4.5].
4	2.75E+03	Each matrix entry from `U(-1,1)`, the 256th lower diagonal element is multiplied by 1e-50
5	1.24E+04	Each main diagonal element from `U(-1,1)`, each off-diagonal element chosen with 50% probability either 0 or from `U(-1,1)`
6	1.03E+00	A Toeplitz matrix, main diagonal entries are 64 and off-diagonal entries are from `U(-1,1)`
7	9.00E+00	inv(gallery('kms',512,0.5)) in Matlab: Inverse of a Kac-Murdock-Szego Toeplitz
8	9.87E+14	gallery('randsvd',512,1e15,2,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, 1 small singular value
9	9.97E+14	gallery('randsvd',512,1e15,3,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, geometrically distributed singular values
10	1.29E+15	gallery('randsvd',512,1e15,1,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, 1 large singular value
11	1.01E+15	gallery('randsvd',512,1e15,4,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, arithmetically distributed singular values
12	2.20E+14	Each matrix entry from `U(-1,1)`, the lower diagonal elements are multiplied by 1e-50
13	3.21E+16	gallery('dorr',512,1e-4) in Matlab: An ill-conditioned, diagonally dominant matrix
14	1.14E+67	A Toeplitz matrix, main diagonal is 1e-8, off-diagonal element are from `U(-1,1)`
15	6.02E+24	gallery('clement',512,0) in Matlab: All main diagonal elements are 0; eigenvalues include plus and minus 511, 509, ..., 1
16	7.1E+191	A Toeplitz matrix, main diagonal is 0, off-diagonal element are from `U(-1,1)`

TABLE IV
RELATIVE ERRORS AMONG METHODS

Matrix type	SPIKE-diag_pivot[a]	SPIKE-Thomas[a]	CUSPARSE	MKL	Intel SPIKE[b]	Matlab
1	1.82E-14	1.97E-14	**7.14E-12**	1.88E-14	1.39E-15	1.96E-14
2	1.27E-16	1.27E-16	1.69E-16	1.03E-16	1.02E-16	1.03E-16
3	1.55E-16	1.52E-16	2.57E-16	1.35E-16	1.29E-16	1.35E-16
4	1.37E-14	1.22E-14	**1.39E-12**	3.10E-15	1.69E-15	2.78E-15
5	1.07E-14	1.13E-14	1.82E-14	1.56E-14	4.62E-15	2.93E-14
6	1.05E-16	1.06E-16	1.57E-16	9.34E-17	9.51E-17	9.34E-17
7	2.42E-16	2.46E-16	5.13E-16	2.52E-16	2.55E-16	2.27E-16
8	2.14E-04	2.14E-04	~~**1.50E+10**~~	3.76E-04	2.32E-16	2.14E-04
9	2.32E-05	3.90E-04	~~**1.93E+08**~~	3.15E-05	9.07E-16	1.19E-05
10	4.27E-05	4.83E-05	~~**2.74E+05**~~	3.21E-05	4.72E-16	3.21E-05
11	7.52E-04	**6.59E-02**	~~**4.54E+11**~~	2.99E-04	2.20E-15	2.28E-04
12	5.58E-05	7.95E-05	5.55E-04	2.24E-05	5.52E-05	2.24E-05
13	5.51E-01	5.45E-01	~~**1.12E+16**~~	3.34E-01	3.92E-15	3.08E-01
14	2.86E+49	4.49E+49	2.92E+51	1.77E+48	**3.86E+54**	1.77E+48
15	2.09E+60	~~**Nan**~~	~~**Nan**~~	1.47E+59	~~**Fail**~~	3.69E+58
16	~~**Inf**~~	~~**Nan**~~	~~**Nan**~~	~~**Inf**~~	~~**Fail**~~	4.7E+171

[a]The number of partitions is 64 for a 512-size matrix on a GPU.
[b]The number of partitions is 4 for a 6-core Intel Xeon X5680 CPU .

In Table IV, we use the default Matlab tridiagonal solver as a baseline for numerical accuracy and stability. Bold numbers indicate solution errors 100× larger than the baseline results, while bold and struckthrough highlight solution errors 1 million times worse than the baseline results. Significantly low relative error rates, 100× better than the baseline results, are underlined. Among 16 types of test matrices, outright solver failures can be grouped into three categories: 1) CUSPARSE, Intel SPIKE, and our SPIKE-Thomas implementation fail two particular matrices; 2) our SPIKE-diagonal_pivoting solver and MKL fail one specific matrix; and 3) Matlab produces results with finite errors for all matrices. The accuracy of the valid solutions from the different solvers also varies greatly. With CUSPARSE, 7 matrices result in high relative errors (at least 100× worse than Matlab) and 5 of those 7 are at least 1 million times worse than Matlab. Our SPIKE-Thomas has only one matrix with a finite solution error more than 100× worse that the Matlab solution error. Intel MKL, Matlab, and our SPIKE-diagonal_pivoting have roughly comparable error rates. While the best solver for a particular application will depend on the properties of the tridiagonal system in that application, on average, the Intel SPIKE solver produces the most stable results for a "typical" system, whereas our SPIKE-diagonal_pivoting solver is the best GPU solver overall for numerical stability.

B. Single GPU Performance Evaluation

We evaluate the performance of our GPU-based solver in which we compare against the results of CUSPARSE library on a GPU and MKL library on a CPU. Note that the MKL library does not include a multithreaded version of the `gtsv` function. All the tests are done on AC, where the results of MKL were generated using icc with -O3 optimization, and the results of our SPIKE-based solvers and CUSPARSE are generated using nvcc and gcc with -O3 optimization.

Figure 12 shows the performance for two kinds of matrices, a random matrix, which is similar to the type 1 from the stability evaluation but with a larger size (8 million), and a strictly column diagonally dominant matrix. In our evaluation,

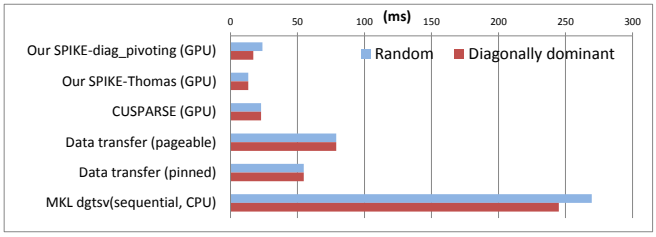


Fig. 12. Single GPU performance comparison among GPU-based solvers for an 8M-size matrix

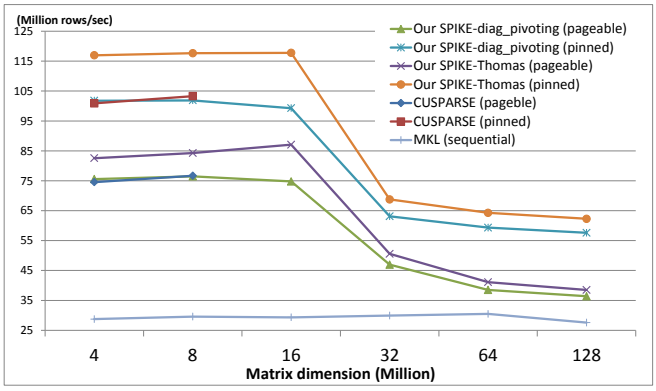


Fig. 13. Single GPU performance comparison for large matrices

the random matrix is used to estimate robustness of our SPIKE-diagonal pivoting, because it causes very strong branch divergence in our method. For the GPU-based solvers, the time we list is the kernel execution time on a single GPU without counting time of data transfer, which is independent of the GPU solver itself and listed separately for both pageable and pinned host memory. In the case of the random matrix, our method performs with comparable execution time (less than 5% difference) to that of CUSPARSE, while in terms of precision, our SPIKE-diagonal_pivoting has a much smaller error rate. Compared to MKL, our method can get 3.20× and 12.61× speedups with and without considering data transfer for GPU, respectively. Our SPIKE-Thomas implementation, which is numerically stable in many cases but cannot be guaranteed, outperforms CUSPARSE by a factor of 1.72×, and compared to MKL, it shows 3.60× and 22.70× speedups with and without data transfer, respectively. CUSPARSE and SPIKE-Thomas have no data-dependent conditions in their implementation, and so perform identically for either test matrix. However, for a random matrix, the tridiagonal solver in CUSPARSE and our SPIKE-Thomas implementation may have stability issues, which may compute solutions that are either invalid or with a higher error rate than is tolerable.

If we assume the input matrix is strictly column diagonally dominant (from prior knowledge or a matrix test before running the solver), both CUSPARSE and our SPIKE-Thomas implementation are stable and should produce solutions with reasonable error. Meanwhile, our SPIKE-diagonal_pivoting would not suffer from branch divergence, and MKL can also run slightly faster without row interchange. In this case, our SPIKE-diagonal_pivoting shows 1.35×, 3.13×, 16.06× speedups over CUSPARSE, MKL with and without considering data transfer for GPU, respectively.

For a very large system, the whole data may be too big to fit into the memory size on single GPU. For example, `dgtsv` in CUSPARSE cannot support a double precision matrix whose dimension is larger than 16776960 rows on GTX 480, because it runs out of memory space. In our GPU-based solver, data can be partitioned and solved chunk-by-chunk on a single GPU. By using this strategy, our solver can work well for a matrix whose data size is larger than GPU memory, though the data transfer among CPUs and GPUs grows slightly faster than the problem size itself. Figure 13 shows the final performance (including all data transfer for the GPU-based solvers) of our SPIKE-diagonal_pivoting, our SPIKE-Thomas, CUSPARSE, and Intel MKL for solving a random matrix. Note that CUSPARSE fails when the input matrix size is larger than 8M. Let us be especially clear that for these results, we are comparing libraries on a single system, not processors within the system; the Intel MKL solver uses only a single core of the six-core CPU. In terms of performance, our numerically stable solver can achieve up to 3.54× speedups over Intel MKL. When the size of required memory is larger than GPU memory capacity (1.5 GB for GTX 480), its speedups decrease but are still up to 1.95× faster than Intel MKL.

C. Scalability Evaluation

We evaluate our GPU-based algorithms for multiple GPUs on NCSA Forge. First, our GPU-based solver for a random matrix is evaluated on 1, 2, 4, 8 and 16 GPUs. The results of 1, 2, and 4 GPUs use on single node, while the results of 8 and 16 GPUs are generated on multiple nodes.

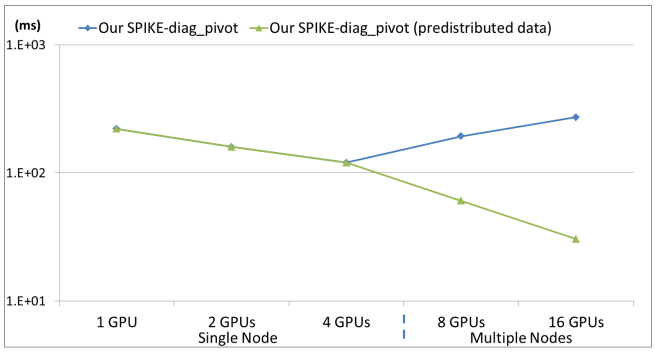


Fig. 14. Multi-GPU scalability evaluation for a 16M-size problem

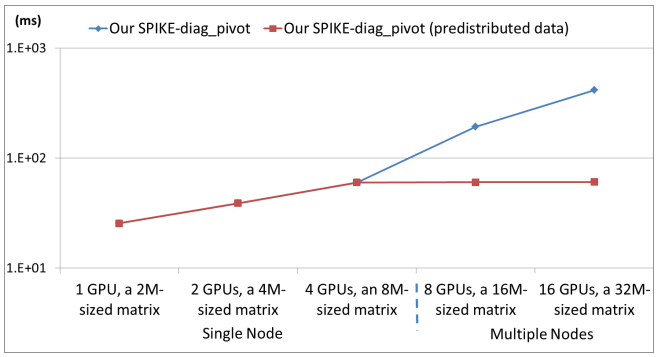
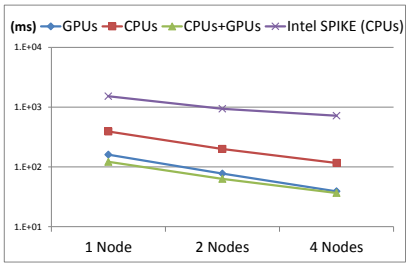
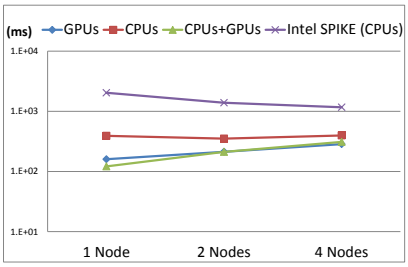


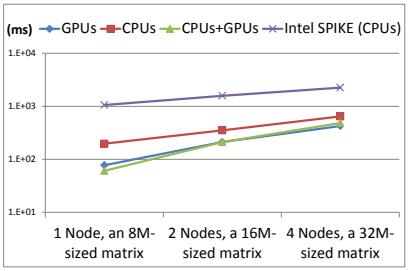
Fig. 15. Multi-GPU scalability evaluation for a varied-size problem(weak scaling results)



(a) Strong scaling, 16M-size matrix, with pre-distributed data



(b) Strong scaling, 16M-size matrix, with data distributing



(c) Weak scaling, varied-size matrices, with data distributing

Fig. 16. Cluster scalability evaluation for CPUs only, GPUs only, and CPUs+GPUs configurations of our library.

In Figure 14, we show strong scaling to solve a 16M-sized random matrix for our solver among multiple GPUs. Within one node, our solver scales quite well to multiple devices. However, it is not perfectly linear scalability, since multiple GPUs may compete for the bandwidth of PCI-E bus. Among GPUs in multiple nodes, our GPU-based solver achieves perfectly linear scalability when the input begins already distributed among the various nodes. However, when the input source and solution reside on a single node, the total runtime is dominated by the MPI data distribution time. Figure 15 evaluates scaling among different sizes. In this case, when the number of GPUs scales, we also scale the problem size. It shows our library does scale well when the matrix size increases.

In Figure 16, we evaluate our heterogeneous MPI solver using CPUs only, GPUs only, or both CPUs and GPUs on the Forge cluster. We compare against Intel SPIKE, which uses the CPUs only, as a reference. Up to 4 nodes are used, and in each node we use 2 GPUs and 4 CPU cores. Figure 16(a) shows good strong scaling to solve a 16M-sized random matrix when the data is predistributed, while Figure 16(b) shows that MPI data distribution still dominates the runtime when the library is invoked from a single node. Figure 16(c) shows weak scaling performance including MPI distribution, which again emphasizes that the cost of distributing partitions of data over MPI grows with the number of nodes on our cluster, and grows fastest for the heterogeneous solver, where the number of used compute devices per node is greatest.

V. Conclusions and Future Work

Previous tridiagonal libraries were relatively context-specific, only applicable for certain kinds of matrices or on certain kinds of processors. Invoking a sequential CPU library will only utilize a single CPU core. Previous GPU solvers required matrix systems with limited sizes or specific data properties to compute a valid solution. The Intel SPIKE library expresses all parallelism as MPI ranks, which can be less efficient on a multicore, shared-memory system.

With this work, we provide a high-performance tridiagonal solver library that is much more broadly applicable. It provides numerical stability comparable with other general-purpose solvers, and can utilize any or all CPU and GPU resources on shared- or distributed-memory systems. It can distribute work to multiple compute resources when invoked from a single process, and also provides a direct interface to distributed programs. Even when better solvers for various compute elements arise, our library's infrastructure provides the tools necessary to distribute work to each solver within a unified interface.

Opportunities for future work include OpenCL implementations of the algorithms and optimizations described here for even broader applicability. In addition, our dynamic tiling optimization relies on the presence of a hardware cache to improve memory system performance. Future work should continue to explore potential numerically stable algorithms and optimizations for GPUs that lack a hardware cache.

The full source code of our library with an open license agreement is released at http://impact.crhc.illinois.edu.

Acknowledgments

We would like to thank the anonymous referees for their many helpful comments and suggestions. The work is partly supported by the FCRP Gigascale Systems Research Center, the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515), and the UIUC CUDA Center of Excellence.

References

[1] A. Lefohn, U. C. Davis, J. Owens, and U. C. Davis, "Interactive depth of field using simulated diffusion," tech. rep., 2006.

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

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

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

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

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

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

[8] 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.

[9] 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.

[10] H.-S. Kim, S. Wu, L.-W. Chang, and W.-M. Hwu, "A scalable tridiagonal solver for gpus," in *Parallel Processing (ICPP), 2011 International Conference on*, pp. 444 –453, sept. 2011.

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

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

[13] NVIDIA Corporation, *CUDA CUSPARSE Library*, Jan. 2012.

[14] 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.

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

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

[17] 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.

[18] 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*, vol. 2, Morgan Kaufmann, Aug. 2011.

[19] 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.

[20] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: The SPIKE algorithm," *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006.

[21] E. Polizzi and A. Sameh, "SPIKE: A parallel environment for solving banded linear systems," *Computers and Fluids*, vol. 36, no. 1, pp. 113–120, 2007.

[22] J. B. Erway, R. F. Marcia, and J. Tyson, "Generalized diagonal pivoting methods for tridiagonal systems without interchanges," *IAENG International Journal of Applied Mathematics*, vol. 4, no. 40, pp. 269–275, 2010.

[23] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 513–522, ACM, 2010.

[24] I.-J. Sung, G. D. Liu, and W.-M. W. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *IEEE Conference on Innovative Parallel Computing (In Press)*, May 2012.

[25] MPI Forum, "Message Passing Interface (MPI) Forum Home Page." http://www.mpi-forum.org/.

[26] OpenMP, "The OpenMP API specification for parallel programming." http://openmp.org.

[27] Intel, "Math Kernel Library." http://developer.intel.com/software/products/mkl/.

[28] Intel, "Intel adaptive spike-based solver." http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/.

[29] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[30] NVIDIA Corporation, *CUDA Programming Guide 4.1*, Nov. 2011.

[31] K. Group, "OpenCL." http://www.khronos.org/opencl/.

[32] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-M. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, (New York, NY, USA), pp. 347–358, ACM, 2010.

[33] J. R. Bunch and L. Kaufman, "Some stable methods for calculating inertia and solving symmetric linear systems," *Math. Comp.*, pp. 63–179, January 1977.

[34] NVIDIA Corporation, *COMPUTE VISUAL PROFILER*, Jan. 2012.

[35] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-M. Hwu, "GPU clusters for high-performance computing," *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–8, 2009.

[36] National Center for Supercomputing Applications at the University of Illinois, "Dell nvidia linux cluster forge." http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/DellNVIDIACluster/.

[37] I. S. Dhillon, "Reliable computation of the condition number of a tridiagonal matrix in o(n) time," *SIAM J. Matrix Anal. Appl.*, vol. 19, pp. 776–796, July 1998.

[38] G. I. Hargreaves, "Computing the condition number of tridiagonal and diagonal-plus-semiseparable matrices in linear time," *SIAM J. Matrix Anal. Appl.*, vol. 27, pp. 801–820, July 2005.