Technical Report TR-2012-04

SPIKE::GPU - A GPU-based Banded Linear System Solver

Ang Li, Andrew Seidl, Dan Negrut

November 15, 2012

Abstract

The SPIKE algorithm [1, 2] is an efficient generic divide-and-conquer algorithm for solving banded systems. With partitioning the matrices into several blocks, sufficient concurrency can be exploited on GPUs. We implemented SPIKE::GPU, a solver which exploits the truncated SPIKE as pre-conditioner and then the pre-conditioned result is refined with BiCGStab. Our current results show that SPIKE::GPU can perform more than two times as fast as the banded linear system solver in Intel's Math Kernel Library (MKL) and up to three times if the kernel is manually tuned.

Contents

1 Introduction

The SPIKE algorithm, as outlined in [1, 2], is an efficient generic divide-and-conquer algorithm for solving banded linear systems. The idea of SPIKE, which dates back to 1978, involves the following stages: (a) Pre-processing: (a1) partitioning of the original system on different processors; (a2) factorization of each diagonal block and extraction of a reduced system of much smaller size; and (b) Post-processing: (b1) solving the reduced system, and (b2) retrieving the overall solution. In this section, we will introduce the problem we intend to solve (subsection 1.1); how partition and factorization is done for matrix $\mathbf{A}$ (subsection 1.2); how a reduced system of much smaller size is achieved (subsection 1.3), and how the preconditioned result is refined (subsection 1.4).

1.1 Problem Description

Given an $N \times N$ matrix $\mathbf{A}$ with half-bandwidth K and an N-dimension Right-Hand-Side(RHS) $\mathbf{b}$, we are asked to find an N-dimension vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$. Note that vector $\mathbf{b}$ and $\mathbf{x}$ can be generalized to $N \times M$ matrices.

For our current implementation, a constraint is that the matrix $\mathbf{A}$ is diagonal dominant.[1] The diagonal dominance property allows us to avoid pivoting during the LU-UL factorization. For matrices which do not conform to this property, we plan to use SPIKE::GPU in conjunction with spectral reordering technique that gets the matrix to be diagonal-dominant in our future work.

1.2 Partition and Factorization

For a specified partition size (denoted as $psize$), the matrix $\mathbf{A}$ is virtually partitioned into p parts. Thus the dimension N, the partition number p, and partition size $psize$ own the relation $p = \lfloor \frac{N}{psize} \rfloor$. Note that $psize$ should conform to the constraint $2K \leq psize \leq \frac{N}{2}$. For $psize$ smaller than $2K$, the SPIKE algorithm fails; for $psize$ larger than $\frac{N}{2}$, the whole matrix $\mathbf{A}$ is made up of a single partition. According to this partition, the matrix $\mathbf{A}$ is written in the form of the product of two matrices $\mathbf{D}$ and $\mathbf{S}$, in which $\mathbf{D}$ is block-diagonal and $\mathbf{S}$ is called the "spike"matrix (shown in figure 1). The "spikes" $\mathbf{W}_i$'s $(1 \leq i \leq p-1)$ and $\mathbf{V}_j$'s $(2 \leq i \leq p)$ are computed by solving the equations:

$$\mathbf{A}_1 \mathbf{W}_1 = \begin{bmatrix} 0 \\ \mathbf{C}_1 \end{bmatrix}$$

$$\mathbf{A}_i \begin{bmatrix} \mathbf{V}_i & \mathbf{W}_i \end{bmatrix} = \begin{bmatrix} \mathbf{B}_i & 0 \\ 0 & 0 \\ 0 & \mathbf{C}_i \end{bmatrix} (2 \leq i \leq p-1)$$

$$\mathbf{A}_p \mathbf{V}_p = \begin{bmatrix} \mathbf{B}_p \\ 0 \end{bmatrix}$$

This requires LU factorization of all $\mathbf{A}_i$ $(1 \leq i \leq p)$. Note that all factorizations of $\mathbf{A}_i$'s and equation solutions for $\mathbf{V}_i$'s and $\mathbf{W}_i$'s are independent. As such, they can be executed concurrently by launching p thread blocks.

1.3 System Reduction

After partitioning and factorization, the original problem is reduced to solving the following two linear systems: $\mathbf{Dg} = \mathbf{b}$ and $\mathbf{Sx} = \mathbf{g}$. As matrix $\mathbf{D}$ is block-diagonal and all $\mathbf{A}_i$'s $(1 \leq i \leq p)$ have been LU-factorized, solving the first equation requires only a forward elimination followed by a backward substitution carried out p times independently to solve $\mathbf{D}_i \mathbf{g}_i = \mathbf{b}_i$ $(1 \leq i \leq p)$.

[1] If a matrix $\{a_{ij}\}_{n \times n}$ owns the property $|a_{ii}| \geq d \cdot \sum_{j \neq i} |a_{ij}|$ for a certain $d \geq 1$, then matrix $\mathbf{A}$ owns the property of diagonal dominance, d is called the degree of diagonal dominance.

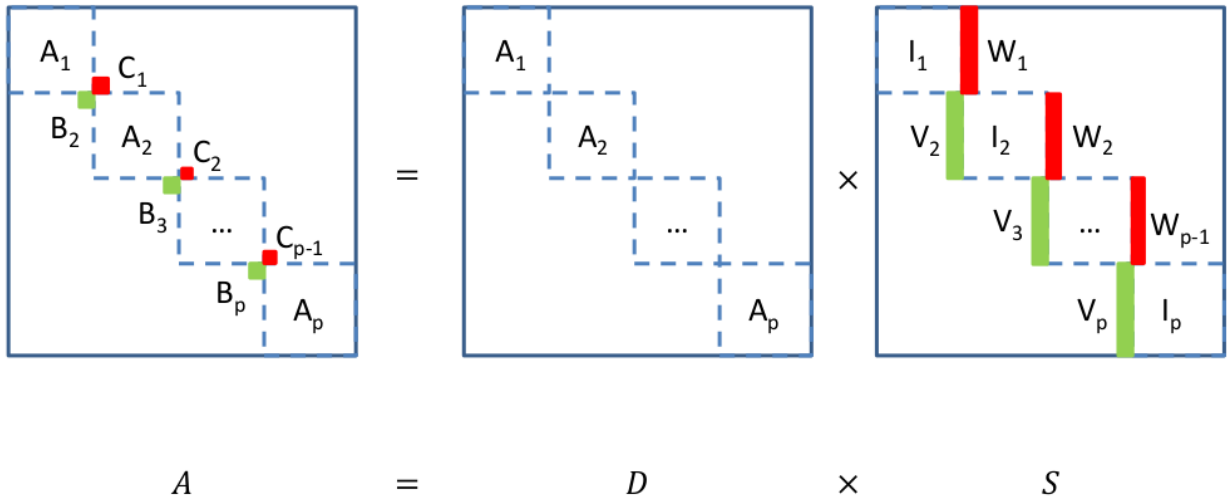

Figure 1: this figure shows how a block-dense matrix $\mathbf{A}$ can be factorized into a block-diagonal matrix $\mathbf{D}$ and a spike matrix $\mathbf{S}$

Two distinct strategies can be pursued for solving $\mathbf{Sx} = \mathbf{g}$. For convenience, we will first introduce some notations. All $\mathbf{V}_i$'s and $\mathbf{W}_j$'s can be expressed in the form

$\mathbf{V}_i = \begin{bmatrix} \mathbf{V}_i^t \\ \mathbf{V}_i' \\ \mathbf{V}_i^b \end{bmatrix}, \mathbf{W}_i = \begin{bmatrix} \mathbf{W}_i^t \\ \mathbf{W}_i' \\ \mathbf{W}_i^b \end{bmatrix}$, where 't' and 'b' represent the $K \times K$ top and bottom matrices, respectively.

Similarly, for all partitions of the RHS $\mathbf{b}_i$'s and unknown $\mathbf{x}_i$'s,

$$\mathbf{b}_i = \begin{bmatrix} \mathbf{b}_i^t \\ \mathbf{b}_i' \\ \mathbf{b}_i^b \end{bmatrix}, \text{where } 1 \leq i \leq p$$

$$\mathbf{x}_i = \begin{bmatrix} \mathbf{x}_i^t \\ \mathbf{x}_i' \\ \mathbf{x}_i^b \end{bmatrix}, \text{where } 1 \leq i \leq p$$

Additionally, the following notation is used:

$$\mathbf{R}_i = \begin{bmatrix} \mathbf{I} & \mathbf{W}_i^b \\ \mathbf{V}_{i+1}^t & \mathbf{I} \end{bmatrix}, \text{where } 1 \leq i \leq p-1$$

$$\mathbf{M}_i = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{W}_{i+1}^t \end{bmatrix}, \text{where } 1 \leq i \leq p-2$$

$$\mathbf{N}_i = \begin{bmatrix} \mathbf{V}_{i+1}^b & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \text{where } 1 \leq i \leq p-2$$

$$\hat{\mathbf{x}}_i = \begin{bmatrix} \mathbf{x}_i^b \\ \mathbf{x}_{i+1}^t \end{bmatrix} \text{ and } \hat{\mathbf{b}}_i = \begin{bmatrix} \mathbf{b}_i^b \\ \mathbf{b}_{i+1}^t \end{bmatrix}, \text{where } 1 \leq i \leq p-1$$

$$\tilde{\mathbf{M}}_{WV} = \begin{bmatrix} \mathbf{R}_1 & \mathbf{M}_1 & & & & \\ \mathbf{N}_1 & \mathbf{R}_2 & \mathbf{M}_2 & & & \\ & \mathbf{N}_2 & \ddots & \ddots & & \\ & & \ddots & \mathbf{R}_{p-3} & \mathbf{M}_{p-3} & \\ & & & \mathbf{N}_{p-3} & \mathbf{R}_{p-2} & \mathbf{M}_{p-2} \\ & & & & \mathbf{N}_{p-2} & \mathbf{R}_{p-1} \end{bmatrix}$$

For the recursive SPIKE, the problem is reduced to solving

$$\tilde{\mathbf{M}}_{WV} \times \begin{bmatrix} \hat{\mathbf{x}}_1 \\ \hat{\mathbf{x}}_2 \\ \vdots \\ \hat{\mathbf{x}}_{p-1} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{g}}_1 \\ \hat{\mathbf{g}}_2 \\ \vdots \\ \hat{\mathbf{g}}_{p-1} \end{bmatrix}$$

and all $\hat{\mathbf{x}}_i$'s are subsequently used to compute $\mathbf{x}'_i$. The method to retrieve solution from $\hat{\mathbf{x}}_i$'s is omitted here as we will not use recursive SPIKE algorithm. For more detail, readers can reference to Heyn and Negrut's work in 2011 [3].

If matrix $\mathbf{A}$ is diagonal-dominant, some approximations can be made to simplify the overall SPIKE solution methodology. It was shown in [4] that the magnitudes of $\mathbf{V}_i$'s decay top to bottom and those of $\mathbf{W}_i$'s decay bottom to top. Thus in the so-called "truncated SPIKE" algorithm, the matrix $\tilde{\mathbf{M}}_{\mathbf{WV}}$ is approximated by discarding all $\mathbf{M}_i$'s and $\mathbf{N}_i$'s to get a new matrix

$$\mathbf{M}_{WV} = \begin{bmatrix} \mathbf{R}_1 & & & \\ & \mathbf{R}_2 & & \\ & & \ddots & \\ & & & \mathbf{R}_{p-1} \end{bmatrix}$$

Note that $\mathbf{M}_{WV}$ is strictly a block-diagonal matrix, it can be solved fast by launching $(p-1)$ kernels. Note that instead of calculating an accurate $\mathbf{W}_i$'s and $\mathbf{V}_i$'s by solving the p linear systems mentioned in subsection 1.2, we can approximate these matrices by solving even smaller systems. The details will be covered in subsection 2.2.

After achieving all $\mathbf{x}_i^t$'s and $\mathbf{x}_i^b$'s, the RHS $\mathbf{b}$ is purified from the contributions of tying blocks $\mathbf{B}_i$'s and $\mathbf{C}_i$'s. Then we solve the resulting block diagonal system using previously computed LU-UL factorizations, as shown below

$$\mathbf{A}_1\mathbf{x}_1 = \mathbf{b}_1 - \begin{bmatrix} \mathbf{0} \\ \mathbf{I}_{k\times k} \end{bmatrix}_{p\times k} \mathbf{C}_1\mathbf{x}_2^t$$

$$\mathbf{A}_i\mathbf{x}_i = \mathbf{b}_i - \begin{bmatrix} \mathbf{0} \\ \mathbf{I}_{k\times k} \end{bmatrix}_{p\times k} \mathbf{C}_i\mathbf{x}_{i+1}^t - \begin{bmatrix} \mathbf{I}_{k\times k} \\ \mathbf{0} \end{bmatrix}_{p\times k} \mathbf{B}_i\mathbf{x}_{i-1}^b, \text{where } 2 \le i \le p-1$$

$$\mathbf{A}_p\mathbf{x}_p = \mathbf{b}_p - \begin{bmatrix} \mathbf{I}_{k\times k} \\ \mathbf{0} \end{bmatrix}_{p\times k} \mathbf{B}_p\mathbf{x}_{p-1}^b$$

1.4 Refinement

The solution obtained with the truncated SPIKE algorithm is not accurate. This is due to (1) in truncated SPIKE algorithm, we ignore the contributions of the matrices $\begin{bmatrix} \mathbf{W}_i^t \\ \mathbf{W'}_i \\ \mathbf{0} \end{bmatrix}$ and $\begin{bmatrix} \mathbf{0} \\ \mathbf{V'}_i \\ \mathbf{V}_i^b \end{bmatrix}$ Table 1 lists the average relative residual[2] for various diagonal dominance factors. The error becomes large at small values of d.

[2]Relative residual is defined as $r = \frac{||\mathbf{Ax}-\mathbf{b}||_\infty}{||\mathbf{b}||_\infty}$

d	Ave. relative residual(%)
1	35
10	1.2
100	0.2
1,000	0.02
10,000	0.002

Table 1: Average relative residual for various degree of diagonal dominance d.

Our results are refined with BiCGStab. It only requires six to seven iterations to make the results converge to an acceptable degree (we set the thredshold of relative residual to be 10^{-8}). Moreover, it is straightforward to implement as it only requires matrix-vector multiplication, reduction and saxpy operations.

Algorithm 1 illustrates the process of BiCGStab. BiCGStab introduces a newly defined residual $\mathbf{r}$. This new residual is checked and refined. In each iteration, the algorithm checks whether the current new residual $\mathbf{r}_i$ converges to a given positive value ϵ. If it does, the algorithm terminates and current $\mathbf{x}_i$ is returned as the solution, otherwise $\mathbf{r}_i$ is corrected. By introducing an intermediate vector $\mathbf{s}_i$, BiCGStab always selects delicately a constant ω_i such that the new residual $\mathbf{r}_{i+1}$ is minimized. For many problems, especially when the preconditioned solution $\mathbf{x}_0$ is close to the real solution $\mathbf{x}$, the BiCGStab converges rather smoothly and often faster than BiCG and CGS. The cost is each iteration step in BiCGStab is slightly more expensive than that in BiCG and CGS. Sometimes when ω_i becomes close to zero, BiCGStab suffers from the problem of stagnation or breakdown. For addressing this problem, Sleijpen and Fokkema proposed more robust algorithms BiCGStab(l) [6]. We did some simulation to find that BiCGStab usually stags when the relative residual is to the order of 10^{-14}. However, we consider a relative residual with value 10^{-8} or 10^{-9} is rather small so that we won't bother pursue a smaller residual. Thus in our work, we simply implemented BiCGStab.

Algorithm 1 The BiCGStab algorithm

Require: $\epsilon > 0$
  $\hat{\mathbf{r}}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
  $\rho_0 = \alpha = \omega_0 = 1$
  $\mathbf{v}_0 = \mathbf{p}_0 = \mathbf{0}$
  $i = 0$
  while $\frac{||\mathbf{r}_i||_\infty}{||\mathbf{b}||_\infty} \geq \epsilon$ **do**
    $i = i + 1$
    $\rho_i = (\hat{\mathbf{r}}_0, \mathbf{r}_i)$
    $\beta = (\frac{\rho_i}{\rho_{i-1}})(\frac{\alpha}{\omega_{i-1}})$
    $\mathbf{p}_i = \mathbf{r}_{i-1} + \beta(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$
    $\mathbf{v}_i = \mathbf{A}\mathbf{p}_i$
    $\alpha = \frac{\rho_i}{(\hat{\mathbf{r}}_0, \mathbf{v}_i)}$
    $\mathbf{s} = \mathbf{r}_{i-1} - \alpha\mathbf{v}_i$
    $\mathbf{t} = \mathbf{A}\mathbf{s}$
    $\omega_i = \frac{(\mathbf{t}, \mathbf{s})}{(\mathbf{t}, \mathbf{t})}$
    $\mathbf{x}_i = \mathbf{x}_{i-1} + \alpha\mathbf{p}_i + \omega_i\mathbf{s}$
    $\mathbf{r}_i = \mathbf{s} - \omega_i\mathbf{t}$
  end while

2 Implementation Details

In this section, we will introduce the implementation details of SPIKE::GPU. This section consists of matrix storage in subsection 2.1, window-sliding method for LU-UL factorization, forward elimination and backward substitution in subsection 2.2, methods to hide memory latency in subsection 2.3 and the way we tuned the SPIKE::GPU kernel in subsection 2.5.

2.1 Matrix Storage

Three copies of matrix $\mathbf{A}$ are made on device, one in row-major order and the other two in column-major order. The copies in column-major order are used in the truncated SPIKE algorithm. A matrix with dimension $N = 7$ and half-bandwidth $K = 3$ is shown below.

$$\begin{bmatrix} * & * & * & a_{11} & a_{21} & a_{31} & a_{41} \\ * & * & a_{12} & a_{22} & a_{32} & a_{42} & a_{52} \\ * & a_{13} & a_{23} & a_{33} & a_{43} & a_{53} & a_{63} \\ a_{14} & a_{24} & a_{34} & a_{44} & a_{54} & a_{64} & a_{74} \\ a_{25} & a_{35} & a_{45} & a_{55} & a_{65} & a_{75} & * \\ a_{36} & a_{46} & a_{56} & a_{66} & a_{76} & * & * \\ a_{47} & a_{57} & a_{67} & a_{77} & * & * & * \end{bmatrix}$$

This method of storage requires all diagonal elements to be stored in the K-th column. All other elements are distributed columnwise accordingly.

For the row-major order copy, which is used in BiCGStab, the matrix $\mathbf{A}$ is stored in the following way.

$$\begin{bmatrix} * & * & * & a_{11} & a_{12} & a_{13} & a_{14} \\ * & * & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ * & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \\ a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & * \\ a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & * & * \\ a_{74} & a_{75} & a_{76} & a_{77} & * & * & * \end{bmatrix}$$

We maintain the matrix $\mathbf{A}$ with two different storage formats because we intend to avoid memory divergence. In the preconditioning stage, both the LU-UL factorization and forward elimination/backward substitution require accesses of a column of the matrix $\mathbf{A}$. Column-major order introduces less memory divergence and is thus superior. In the refinement stage, on the other hand, matrix-vector multiplication includes accesses of a row of the matrix $\mathbf{A}$. Row-major order storage will be the better choice than column-major order storage. Despite the extra space cost to store another copy of $\mathbf{A}$ and the extra time cost to transfer the data from one copy to another copy of $\mathbf{A}$, after simulating our kernel on various N's and K's, we observe that the saved time can pay off this cost.

2.2 Window-sliding Method

As described in section 1, for LU-UL factorization and forward elimination/backward substitution, the matrix $\mathbf{A}$ is divided into p partitions, which can be handled simultaneously with p thread blocks. In each partition, nonetheless, both the factorization and elimination/substitution consist of $(psize - 1)$ dependent time steps. This implies synchronization inside a thread block is inavoidable. To guarantee synchronization, instead of repeating $(psize - 1)$ kernel launches, each completing a single time step, we launch the kernel once and synchronize explicitly by calling the routine *__syncthreads*(). Figure 2 displays how LU is done for for each partition of the matrix $\mathbf{A}$.[3] For each time step, a single thread updates a fixed number of entries of matrix

[3]For the convenience of explanation, all matrices in this section are illustrated in a row-major "normal" storage format

A. When all threads in a thread block have done their work, the "green" window moves bottom-right by one unit to the next time step. As this process resembles sweeping a window, it is thus named the window-sliding method. Similarly, UL factorization is completed. The difference lies in that the window should slide up-left in UL factorization. Note that as UL is solely exploited in approximating $\mathbf{W}_i$'s $(1 \leq i \leq p-1)$and $\mathbf{V}_j$'s $(2 \leq j \leq p)$, the LU results of the top-left $K \times K$ matrices in each partition of **A** can be approximated by UL-factorizing the top-left $l \times l$ $(l > K$ and $l << psize)$matrices in each partition. For our case, we pick $l = 2K$ so that the execution time of UL compared to LU is ignorable. For forward elimination/backward substitution, the square window for matrix **A** is degenerated to a "tall" and "thin" sliding bar due to the fact that in each time step only a column of elements require updates. An additional sliding bar exists to update the RHS vector **b** and it also moves one unit per time step.

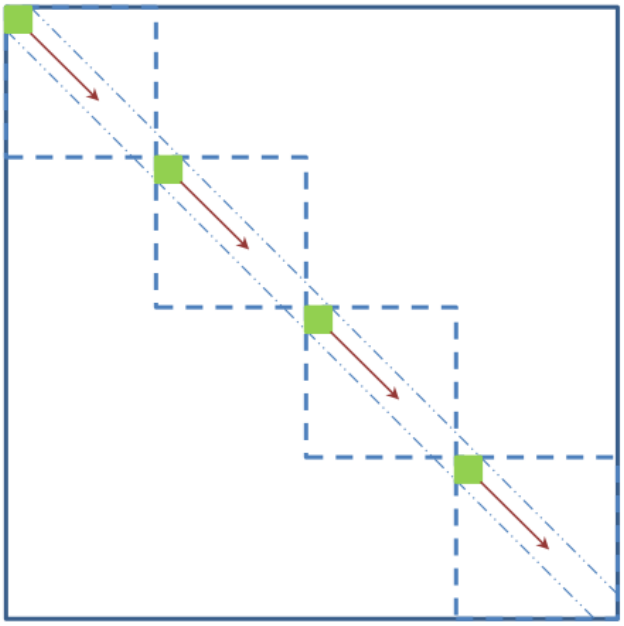

Figure 2: this figure shows the window-sliding method for LU factorization of matrix **A**

For our current implementation, no shared memory is applied to any kernel (LU-UL factorization or forward elimination/backward substitution). A tentative method to apply shared memory is:[4] (1) we allocate space for $(K+1) \times (K+1)$ elements as a circular buffer and this buffer will keep all values inside the window in the current time step; (2) before the first time step, set the window's position to the top-left corner and load the values in the window to shared memory; (3) before window slides, dump all values which will run out of the window back to global memory and read in new values which come into the window; (4) in the last K steps, no loads are needed as the threads come to the boundary of the partition. This method, nonetheless, suffers from memory divergence in step (3) because elements in the north bound and south bound are mapped to discrete addresses. A technique to address this problem is (1) for loads, pre-fetch elements for several time steps and (2) for stores, delay global memory dump and buffer elements to be written for several time steps. We will make this attempt in our future implementation.

2.3 Hide Memory Latency

As the computations of $\mathbf{W}_i$'s $(1 \leq i \leq p-1)$ and $\mathbf{V}_j$'s $(2 \leq j \leq p)$ are rather independent, we managed to hide memory latency in the preconditioning stage by streaming the two processes. We made an observation of

[4]Here we use LU factorization to demonstrate the method

what needed to be done for both streams and then pipelined them in an appropriate way. Table 2 illustrates how memory latency can be hidden by delicate arrangement of operations in calculating W's and V's. Till the purification of RHS, the work is divided into two streams and both of them have perfect pipelined pattern "Compute - Data Transfer - Compute - Data Transfer". Eventually the cost of almost all these data transfers can be hidden by computations. With streaming applied, the performance of our SPIKE::GPU kernel is improved by about 20%.

Stream 1	LU Fact.	prep. to solve $\mathbf{V}$'s	Solve $\mathbf{V}$'s	Copy $\mathbf{V}$'s to $\mathbf{M}_{WV}$	-
Stream 2	Copy $\mathbf{b}$	UL Fact.	prep. to solve $\mathbf{W}$'s	Solve $\mathbf{W}$'s	Copy $\mathbf{W}$'s to $\mathbf{M}_{WV}$

Table 2: By overlapping the computation and transferring data in the processes of calculating $\mathbf{W}$'s and $\mathbf{V}$'s, memory latency is hidden. Operations listed in the same column imply they are supposed to occur simultaneously.

2.4 Mixed Precision Strategy

The performance of single precision arithmetic is superior to that of double precision. As described in [7], on CPUs the difference in performance is usually only a factor of two, whereas on GPUs the difference can be as much as an order of magnitude. Thus, strategic use of precision in a GPU calculation is vitally important to obtaining high performance.

We applied a simple mixed-precision strategy: using single-precision operations in preconditioning stage and double-precision operations in BiCGStab. In the preconditioning stage, we apply various approximations in calculation for a higher performance. As the solution obtained in this approximated algorithm is not accurate by nature, we aggresively go one more step to keep using single-precision operations all the way during this stage. During BiCGStab in contrast, precision is pursued. Thus double precision arithmetics are used in BiCGStab. We compare the execution time between mixed-precision strategy and double-precision-only strategy. Mixed-precision strategy is observed a $15\% - 25\%$ speedup to double-precision-only strategy.

2.5 Kernel Tuning

Partition size has great impact on the kernel performance. An ill-tuned example is displayed in figure 3. The only difference between the configurations of figure 3(a) and figure 3(b) is the matrix dimension N: N in figure 3(b) is one less than N in figure 3(a). In our current implementation, the total number of partition p is $\lfloor \frac{N}{psize} \rfloor$. This implies that the last thread block will always take no less work pressure than other $\lfloor \frac{N}{psize} \rfloor - 1$ blocks. The worst case is when $N = (p+1) \cdot psize - 1$ and the partition size for the last block is almost twice as large as those of other blocks. The last thread block is thus always the bottleneck. We simulated these two configurations with randomly generated matrices $\mathbf{A}$ with degree of diagonal dominance 1.0. The result is the second configuration has a performance degradation of 64.5% even if the dimension of matrix $\mathbf{A}$ is one less. It is arguable that the extra work of the last thread block in the second configuration contributes to additional timing cost.

From this example we can see the importance of kernel tuning. We did a simple case study of how a manually-tuned kernel is superior to an ill-tuned kernel. We vary the value of N between $2,000$ and $100,000$. For ill-tuned configurations, $psize$ is staticly assigned to $min(2048, N)$. The result is shown in table 3. We can see from table 3 that for small N's, especially N with value no greater than $20,000$, the tuning is extremely critical to performance.

Our current implementation does not include autotuning strategies. The kernel can only be tuned manually by letting the user specify the value of $psize$. This method, however, is not practical in that (1) it is not obvious where the optimal $psize$ lies and (2) the users may even be of zero knowledge of SPIKE algorithm. We have proposed a hybrid autotuning method in which we consider both software and hardware factors. We first gave a heuristic to restrict the range where optimal $psize$ can lie in. Then we sample a

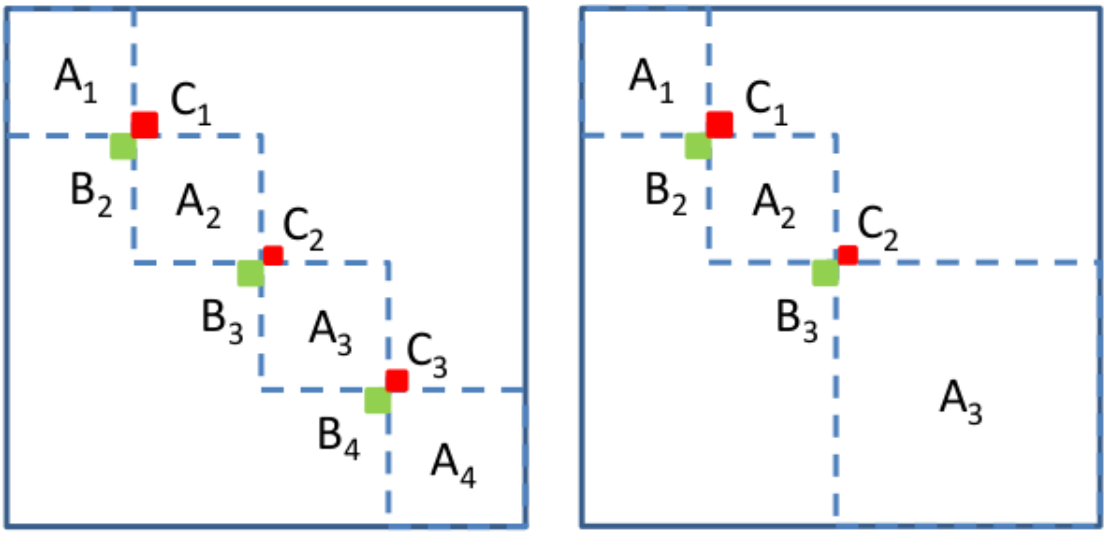

Figure 3: The performance is affected seriously with ill-tuned *psize*. Subfigure (a) illustrates the partition of configuration $N = 8,192$, $K = 32$ and $psize = 2,048$. Subfigure (b) illustrates the partition of configuration $N = 8,191$, $K = 32$ and $psize = 2,048$.

series of *psize*'s to achieve the performance. We do a regression on these points and the optimal *psize* can be retrieved on the curve. This autotuning method will be implemented soon. After the simulation is done, the optimal *psize* will be hardcoded in our library.

N	Exec. Time tuned by hand	Exec. Time ill-tuned	speedup(%)
2,000	4.03	5.69	41.2
4,000	5.56	11.73	111.0
8,000	7.41	20.16	172.6
10,000	8.99	21.07	134.4
20,000	14.24	24.72	73.6
40,000	24.70	32.10	30.0
80,000	42.34	44.53	5.2
100,000	51.86	54.83	5.7

Table 3: The performance comparison between a manually-tuned kernel and an ill-tuned kernel. For all configurations, half-bandwidth K shares the value 32. All timing results are in milliseconds.

3 Evaluation

In this section, our evaluation environment (subsection 3.1), simulation results (subsection 3.2) and profiling results (subsection 3.3) will be reported.

3.1 Environment

Our code was compiled with NVCC compiler with optimization option $-\mathbf{O}3$. Our kernel was tested on NVIDIA GeForce GTX680 card (Kepler). Kepler was first unveiled in May, 2012 and it was supposed to be the fastest and most power-efficient GPU ever built [8]. With four Graphics Processing Clusters (GPCs)

and eight next-generation Streaming Multiprocessors (SMXs), Kepler has $1,536$ CUDA cores, which is three times that of Fermi. While the GFLOPs of Kepler is almost twice that of Fermi ($3,090$ vs $1,581$), its Thermal Design Power is smaller ($195W$ vs $244W$). We have it in mind that the strong computation power of Kepler has greatly benefited our work.

For comparison with CPU, we compared our kernel with Intel's Math Kernel Library (MKL). Intel® MKL includes a wealth of routines, such as highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions, to accelerate application performance and reduce development time. Among these, we call the routines to do LU factorization and forward elimination/backward substitution. The MKL code was compiled with Intel's compiler ICPC on Intel®, also with optimization option $-\mathbf{O}3$.

3.2 Simulation Results

We did two groups of simulation. We tested the wall clock time as a function of N-curve where $K = 32$ and the wall clock time as a function of K-curve where $N = 400,000$.

Figure 4 shows the execution time as a function of N-curve where half-bandwidth K is fixed to 32. Note that SPIKE::GPU has a performance gain of at least 2.1 and up to 3.1 over MKL's solver if our kernel is carefully tuned.

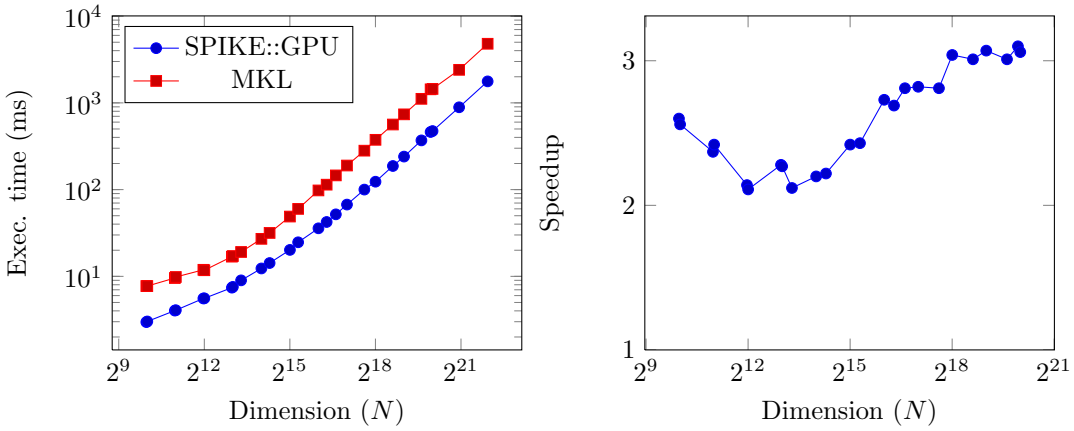

Figure 4: Comparison between SPIKE::GPU and MKL with half-bandwidth $K = 32$. The SPIKE::GPU is manually-tuned.

Figure 5 displays the execution time as a function of K-curve where matrix dimension N is fixed to $400,000$.

The curve is paraphrased in segment. For $K \leq 32$, SPIKE::GPU scales much better than MKL; for $32 < K \leq 64$, though a sudden decline is observed when $K = 33$, SPIKE::GPU still scales much better than MKL; for $64 < K \leq 128$, SPIKE::GPU scales worse than MKL due to the fast growing of LU factorization time cost; for $128 < K \leq 256$, MKL performs irregularly so that it is hard to tell which solver scales better. In all cases, SPIKE::GPU has performance gain larger than 1 compared to MKL.

3.3 Profiling Results

The tool we used to profile our kernel is NVIDIA's Visual Profiler (NVVP). First introduced in the year 2008, NVVP is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. Now it is available as part of CUDA Toolkit.

Figure 6 displays the kernel time distribution. Note that the two largest time consumers are LU factorization in the preconditioning stage and the repetitive calls of matrix-vector multiplication kernel. For

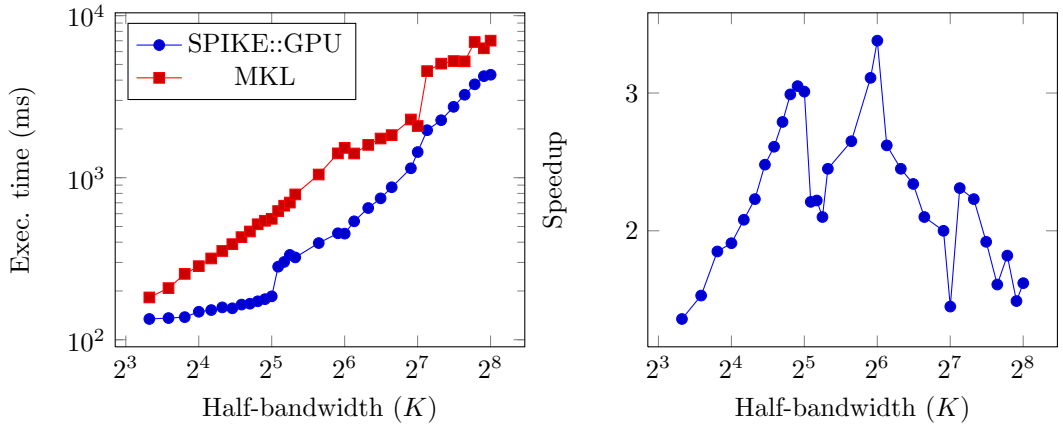

Figure 5: Comparison between SPIKE::GPU and MKL with dimension $N = 400,000$. The SPIKE::GPU is manually-tuned.

further improvement of our kernel, we will seek for more chances to improve the performance of these two kernels. For improving LU factorization, we will try to apply shared memory appropriately. For improving mat-vec multiplication kernel (strictly for the entire BiCGStab), we consider directly calling MKL routines so that there will be no host/device communication in this stage.[5]

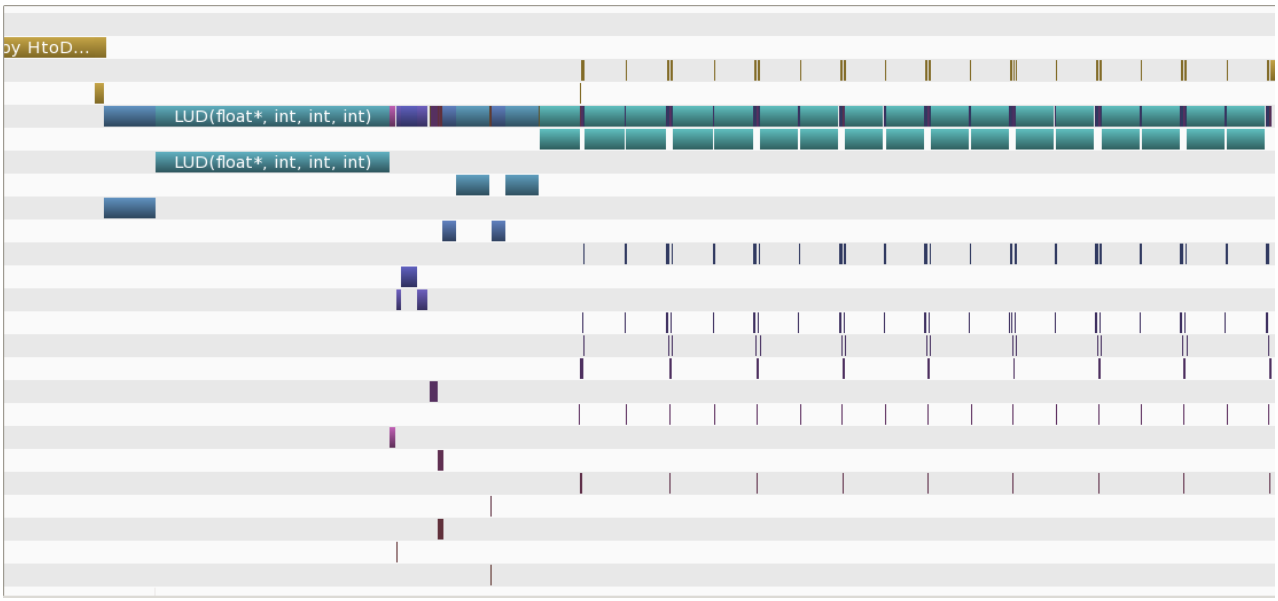

Figure 6: this figure shows the profiling results of execution time for $N = 800,000$, $K = 32$ and $psize = 2,048$

[5]This attempt has been made without payoff. MKL routines are much more time-consuming compared with our GPU kernels.

References

[1] Polizzi, E.; Sameh, A. H. (2006). *A parallel hybrid banded system solver: the SPIKE algorithm.* Parallel Computing **32** (2): 177-194.

[2] Polizzi, E.; Sameh, A. H. (2007). *SPIKE: A parallel environment for solving banded linear systems.* Computers and Fluids **36**: 113-141

[3] Heyn, T.; Negrut, D. (2011). *SPIKE - A Hybrid Algorithm for Large Banded Systems.* TR-2011-01

[4] Demko, S.; Moss, W.F.; Smith P.W. *Decay rates for inverses of band matrices*, Math. Comput. **43** (168) (1984) 491-499

[5] Van der Vorst, H. A. (1992). *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems.* SIAM Journal on Scientific and Statistical Computing **13** (2): 631-644.

[6] G. L. Sleijpen and D. R. Fokkema (1993). *BiCGStab(l) for linear equations involving unsymmetric matrices with complex spectrum.* Electronic Transactions on Numerical Analysis, **1**(11), 2000.

[7] Clark, M. A., Babich, R., Barros, K., Brower, R. C. and Rebbi, C. (2010). *Solving Lattice QCD systems of equations using mixed precision solvers on GPUs.* Computer Physics Communications, **181**(9), 1517-1528.

[8] *NVIDIA GeForce GTX 680 Whitepaper*, in 2012