GPU-Accelerated Preconditioned Iterative Linear Solvers [*]

Ruipeng Li [†]　　Yousef Saad[†]

Abstract

This work is an overview of our preliminary experience in developing high-performance iterative linear solver accelerated by GPU co-processors. Our goal is to illustrate the advantages and difficulties encountered when deploying GPU technology to perform sparse linear algebra computations. Techniques for speeding up sparse matrix-vector product (SpMV) kernels and finding suitable preconditioning methods are discussed. Our experiments with an NVIDIA TESLA C1060 show that for unstructured matrices SpMV kernels can be up to 10 times faster on the GPU than on the host Intel Xeon E5504 Processor. Overall performance of the GPU-accelerated Incomplete Cholesky (IC) factorization preconditioned CG method can outperform its CPU counterpart by a much smaller factor, up to 3, and GPU-accelerated Incomplete LU (ILU) factorization preconditioned GMRES method can achieve a speedup nearing 4. However, with better suited preconditioning techniques for GPUs, this performance can be significantly improved.

1 Introduction

Emerging many-core platforms yield enormous raw processing power, in the form of massive SIMD parallelism. Chip designers are increasing processing capabilities by building multiple processing cores in a single chip. The number of cores that can fit into a chip is increasing at a fast pace and as a result Moore's Law has been given a new interpretation: it is the number of cores that doubles every 18 months [1]. The NVIDIA GPU (Graphics Processing Unit) is one of several available coprocessors that feature a large number of cores. Traditionally, GPUs have been especially designed to handle computations for computer graphics in real-time. Today, they are increasingly being exploited as general-purpose attached processors to speed-up computations in image processing, physical simulations, data mining, linear algebra, etc.

CUDA (Compute Unified Device Architecture) is a parallel language for NVIDIA GPUs, which supports developers to programming on GPU in C/C++ with NVIDIA extensions. Wrappers for Python, Fortran, Java and MATLAB are also available. This paper explores various aspects of sparse linear algebra computations on GPUs. Specifically, we investigate appropriate preconditioning methods for high-performance iterative linear solvers accelerated by GPUs.

The remainder of this report is organized as follows: Section 2 gives a brief introduction of GPU architectures, the CUDA programming model, and related work. Section 3 describes the sparse matrix-vector product (SpMV) kernel on GPUs; Section 4 describes sparse triangular solves and the preconditioned iterative methods, CG and GMRES are discussed in Section 5; Section 6 reports on experimental results and analyses. Finally, concluding remarks are stated in Section 7.

[*]This work is supported by DOE under grant DE-FG 08ER 25841 and by the Minnesota Supercomputer Institute.

[†]Department of Computer Science & Engineering; University of Minnesota; Department of Computer Science & Engineering; Minneapolis, MN 55455, USA.

2 GPU Architecture and CUDA

2.1 GPU Architecture

A GPU is built as a scalable array of multi-threaded *Streaming Multiprocessors* (SMs), each of which consists of multiple *Scalar Processor* (SP) cores. To manage hundreds of threads, the multiprocessors employs a *Single Instruction Multiple Threads* (SIMT) model. Each thread is mapped into one SP core and executes independently with its own instruction address and register state [18].

The NVIDIA GPU platform has various memory hierarchies. The types of memory can be classified as follows: (1) off-chip global memory, (2) off-chip local memory, (3) on-chip shared memory, (4) read-only cached off-chip constant memory and texture memory and (5) registers. The effective bandwidth of each type of memory depends significantly on the access pattern. Global memory is relatively large but has a much higher latency (from 400 to 800 cycles for the NVIDIA GPUs) compared with the on-chip shared memory (1 clock cycle if there are no *bank conflicts*). Global memory is not cached, so it is important to follow the right access pattern to achieve good memory bandwidth.

Threads are organized in *warps*. A warp is defined as a group of 32 threads of consecutive thread IDs. A *half-warp* is either the first or second half of a warp. The most efficient way to use the *global memory* bandwidth is to coalesce the simultaneous memory accesses by threads in a half-warp into a single memory transaction. For NVIDIA GPU devices with 'compute capability' 1.2 and higher, memory access by a half-warp is coalesced as soon as the words accessed by all threads lie in the same segment of size equal to 128 bytes if all threads access 32-bit or 64-bit words. Local memory accesses are always coalesced. Alignment and coalescing techniques are explained in details in [18]. Since it is on chip, the *shared memory* is much faster than the global memory but we also need to pay attention to the problems of *bank conflict*. The size of shared memory is 16K per SM. Cached constant memory and texture memory are also beneficial for accessing reused data and data with 2D spatial locality.

2.2 CUDA Programming

Programming NVIDIA GPUs for general-purpose computing is supported by the NVIDIA CUDA (Compute Unified Device Architecture) environment. CUDA programs on the host (CPU) invoke a *kernel grid* which runs on the device (GPU). The same parallel kernel is executed by many threads. These threads are organized into thread blocks. Thread blocks are distributed to SMs and split into warps scheduled by SIMT units. All threads in the same thread block share the same shared memory of size 16 KB and can synchronize themselves by a barrier. Threads in a warp execute one common instruction at a time. This is referred to as warp-level synchronization [18]. Full efficiency is achieved when all 32 threads of a warp follow the same execution path. Branch divergence causes serial execution.

2.3 Related Work

The potential of GPUs for parallel sparse matrix computations was recognized early on when GPU programming required shader languages [6]. After the advent of NVIDIA CUDA, GPUs have drawn much more attention for sparse linear algebra and the solution of sparse linear systems. Existing implementations of the GPU Sparse matrix-vector product (SpMV) include CUDPP (CUDA Data Parallel Primitives) library [24], NVIDIA CUSPARSE library [5] and the IBM SpMV library [4].

In [4, 5], different formats of sparse matrices are studied for producing high performance SpMV kernels on GPUs.

A GPU-accelerated Jacobi preconditioned CG method is studied in [12]. In [3], the CG method with incomplete Poisson preconditioning is proposed for the Poisson problem on a multi-GPU platform. The Block-ILU preconditioned GMRES method is studied for solving sparse linear systems on the NVIDIA Tesla GPUs in [27]. SpMV kernels on GPUs and the Block-ILU preconditioned GMRES method are used in [16, 25] for solving large sparse linear systems in black-oil and compositional flow simulations. A GPU implementation of deflated PCG method for bubbly flow computations is discussed in [13].

For dense linear algebra computations, the Matrix Algebra for GPU and Multicore Architectures (MAGMA) project [2] using hybrid multicore-multiGPU system aims to develop a dense linear algebra library similar to LAPACK. They reported speeds up to 375/75 GFLOPS in single/double precision for the `GEMM` matrix-matrix operation, and 280 GFLOPS in single precision for solving a linear system by the L/U factorization using multiple GPUs. Vasily Volkov and James Demmel [26] reported that they were able to approach the peak performance of the G80 series GPUs for dense matrix-matrix multiplications.

3 Sparse Matrix-Vector Product

The sparse matrix-vector product (SpMV) is one of the major components of any sparse matrix computations. However, the SpMV kernel which accounts for a big part of the cost of sparse iterative linear solvers, has difficulty in reaching a significant percentage of peak floating point throughput. It yields only a small fraction of the machine peak performance due to its indirect and irregular memory accesses [4]. Recently, several research groups have reported their implementations on high-performance sparse matrix-vector multiplication kernels on GPUs [10, 5, 4]. NVIDIA researchers have demonstrated that their SpMV kernels can reach 15 GFLOPS and 10 GFLOPS in single/double precision for unstructured mesh matrices [5]. This section, will discuss implementation of SpMV GPU kernels using several different formats.

For a matrix-vector product, $y = Ax$, data reuse can be applied to the vector x which is read-only. Therefore, a good strategy is to place the vector x in the cached texture memory and access it by texture fetching [18]. As reported in [5], caching improves the throughput by an average of 30% and 25% in single and double precision respectively.

3.1 SpMV in CSR Format

The compressed sparse row (CSR) format is probably the most widely used general-purpose sparse matrix storage format [20]. In the CSR format, the sparse matrix is specified by three arrays: one real/complex data array A which contains the non-zero elements row by row; one index array JA contains the column index of each non-zero element in A; and one index array IA contains the beginning position of each row. To parallelize the SpMV for a matrix in the CSR format, a simple scheme called *scalar* CSR kernel [5] is to assign each row to one thread. The computations of all threads are independent. This data parallelism and the storage in CSR format are shown in Figure 1. As shown in this figure, matrix rows are distributed to three threads. Each thread works on one partition of the matrix rows. The main drawback of this scheme is that threads within a half-warp access noncontiguous memory addresses, pointed to by the row pointers in Figure 1. As mentioned in Section 2.1, this significantly reduces the chance that memory transactions of a half-warp threads can be coalesced.

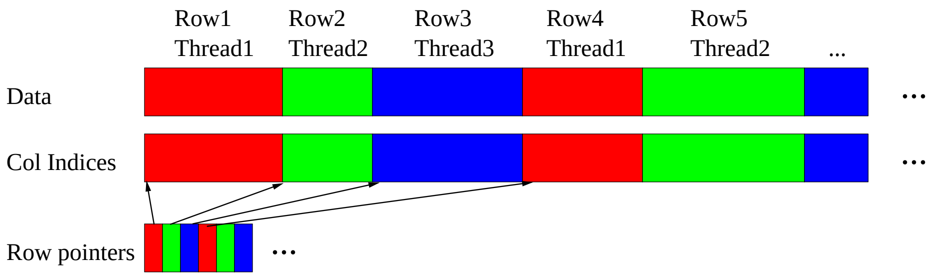


Figure 1: Scalar CSR SpMV Kernel

A better approach, termed *vector* CSR kernel, is to assign a half-warp (16 threads) instead of only one thread to each row. In this approach, the first half-warp works on row one, the second works on row two, and so on. Since all threads within a half-warp access non-zero elements of some row, these accesses are more likely to belong to the same memory segment. So the chance of coalescing should be higher. But this approach incurs another problem related to computing vector dot products in parallel (dot product of a matrix row vector with the vector x). To solve this problem, each thread saves its partial result into the *shared memory* and a parallel reduction is used to sum up all partial results [4, 5]. Moreover, since a warp executes one common instruction at a time, threads within a warp are implicitly synchronized and so the synchronization in the half-warp parallel reduction can be omitted for better performance [18]. In addition, full efficiency of the *vector* CSR kernel requires at least 16 non-zeros per row in A. Our *vector* CSR kernel is slightly different from the implementation in [5] in which a warp of threads are assigned to each row.

3.2 SpMV in JAD Format

The JAD (JAgged Diagonal) format can be viewed as a generalization of the Ellpack-Itpack format which removes the assumption on the fixed-length rows [22]. To build the JAD structure, we first sort the rows by non-increasing order of the number of non-zeros per row. Then the first JAD consists of the first element of each row; the second JAD consists of the second element, etc. The number of JADs is the largest number of non-zeros per row. An example is shown below.

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{pmatrix}, \quad PA = \begin{pmatrix} 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{pmatrix}$$

Here, A is the original matrix and PA is the reordered matrix. Then the JAD format of A is shown in Table 1. The JAD data structure also consist of three arrays: one real/complex data array DJ contains the non-zero elements; one index array JDIAG contains the column index of each non-zero; and one index array IDIAG contains the beginning position of each JAD.

Like in the *scalar* CSR kernel, only one thread works on each matrix row to exploit fine-grained parallelism. Note that since matrix data and indices are stored in the JAD format, the JAD kernel does not suffer from the performance drawback of the *scalar* CSR kernel: consecutive threads can access contiguous memory addresses, which follows the suggested memory access pattern and can improve the memory bandwidth by coalescing memory transactions. This memory access pattern

DJ	3	6	1	9	11	4	7	2	10	12	5	8
JDIAG	1	2	1	3	4	2	3	3	4	5	4	5
IDIAG	1	6	11	13								

Table 1: The JAD Format of Matrix A

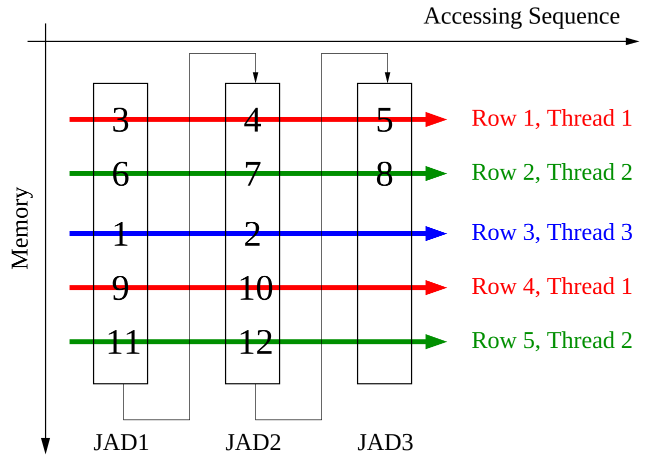


Figure 2: JAD Kernel Memory Access Pattern

is shown in Figure 2.

3.3 SpMV in DIA Format

The DIA (DIAgonal) format is a special format for diagonally structured matrices in which nonzero values are restricted to lie in a small number of diagonals. This format is efficient for memory bandwidth because there is no double indexing as was the case in the CSR or the JAD formats. Specifically, the column index of an element can be calculated by its row index plus the offset value of that diagonal. But this format can also potentially waste storage since zeros are padded for non-full diagonals. The DIA format of the matrix A shown above is illustrated in Figure 2. As in the JAD SpMV kernel, one thread works on each matrix row and consecutive threads can access contiguous memory addresses if the 2D data array DIAG is stored in column major form. It is important to note that the DIA format is not a general-purpose format and patterns of many sparse matrices are not suitable for the DIA format.

$$\text{DIAG} = \begin{array}{|c|c|c|}\hline * & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline 9 & 10 & * \\ \hline 11 & 12 & * \\ \hline\end{array}\;, \text{IOFF} = \begin{array}{|c|c|c|}\hline -1 & 0 & 2 \\ \hline\end{array}$$

Table 2: DIA Format of Matrix A

4 Sparse Triangular Solve

In the row version of the standard forward and backward substitution algorithms for solving triangular systems, the outer loop of the substitution for each unknown is sequential. The inner loop is a vector dot product of a sparse row vector of the triangular matrix and the dense right-hand-side vector b. In the column oriented versions, the inner loop is a saxpy operation of a sparse column vector and the vector b. These dot product and saxpy operations can be split and parallelized but this approach is not efficient in general since the length of the vector involved is usually short.

4.1 Level Scheduling

Better parallelism can be achieved by *level scheduling*, which exploits topological sorting [22]. The idea is to group unknowns $x(i)$ into different levels so that all unknowns within the same level can be computed simultaneously. For forward substitution, the level of unknown $x(i)$, denoted by $l(i)$ can be computed as follow after initializing $l(j) = 0$ for all j,

$$l\,(i) = 1 + \max_{j} \{l\,(j)\}, \text{ for all } j \text{ such that } L_{ij} \neq 0\ .$$

If *nlev* is the number of levels, the permutation array q records the ordering of all unknowns, level by level and $level\,(i)$ points to the beginning of the ith level in array q, then forward substitution with level scheduling is shown in Algorithm 1. The inequality, $1 \leq nlev \leq n$ is satisfied, where n is the size of the system. The best case when $nlev = 1$, corresponds to the situation when all unknowns can be computed simultaneously (e.g., L is a diagonal matrix). In the worst case, when $nlev = n$, the forward substitution is completely sequential. The analogous back substitution is handled similarly.

Algorithm 1 Forward Substitution with Level Scheduling

```
1: for $lev = 1 : nlev$ do
2:   $j_1 = level\,(lev)$
3:   $j_2 = level\,(lev + 1) - 1$
4:   for $k = j_1 : j_2$ do
5:     $i = q\,(k)$
6:     for $j = ial\,(i) : ial\,(i + 1) - 1$ do
7:       $x(i) \leftarrow x(i) - al(j) \times x(jal(j))$
8:     end for
9:   end for
10: end for
```

4.2 GPU Implementation

The implementation of the row version of forward/backward substitution kernel is similar to the *vector* CSR format SpMV kernel. With *level scheduling*, the equations are partitioned into levels. Starting from the first level, each half-warp picks one unknown of the current level to solve and the vector dot product is split and summed up by parallel reduction of the partial results as was done in the *vector* CSR format SpMV kernel. Synchronization among half-warps is needed after each level is finished. The performance of parallel triangular solve on GPUs significantly depends on the number of levels of triangular matrices. For some cases, some reordering technique like the Multiple Minimal Degree (MMD) ordering [11], may help reduce the number of levels so as to increase parallelism. However, the performance of the sparse triangular solve on GPUs is usually far lower than that of sparse matrix-vector products and typically even lower than the sequential implementation on CPUs.

5 Preconditioned Iterative Methods

Linear systems are often derived from discretization of partial differential equations by finite difference method, finite element method or finite volume method. Matrices from these applications are usually large and sparse, which means that a large number of the entries of A are zeros. The Conjugate Gradient (CG) and the Generalized Minimal RESidual (GMRES) methods, see, e.g., [22], are extensively used Krylov subspace methods for solving symmetric positive definite (SPD)/ non-symmetric linear systems iteratively. These are termed accelerators.

A preconditioner is any form of implicit or explicit modification of an original system that make it easier to solve by a given accelerator [22]. The right preconditioned linear system takes the form:

$$AM^{-1}u = b, u = Mx.$$

The matrix M is the preconditioning matrix. It should be close to A, i.e., $M \approx A$ and the preconditioning operation $M^{-1}v$ should be easy to apply for an arbitrary vector v.

The main computational components in the preconditioned CG or GMRES methods are: (1) Matrix-vector product; (2) Preconditioning operation; (3) Level-1 BLAS vector operations. Preconditioned iterative methods can be speeded-up by the GPU SpMV kernels and the CUBLAS library which is an implementation of BLAS library on top of the NVIDIA CUDA driver [17].

The convergence rate of iterative methods is highly dependent on the quality of the preconditioner used. Computational complexities of different types of preconditioning operations are different. Moreover, their performance varies significantly on different computing platforms. In the following sections, several types of preconditioners along with their GPU implementations will be discussed in detail.

5.1 ILU/IC Preconditioners

The Incomplete LU (ILU) factorization process for a sparse matrix A computes a sparse lower triangular matrix L and a sparse upper triangular matrix U such that the residual matrix $R = LU - A$ satisfies certain conditions [22]. Since the preconditioning matrix $M = LU$ and $LU \approx A$, the preconditioning operation is a back substitution followed by a forward substitution, i.e., $u = M^{-1}v = U^{-1}L^{-1}v$. If no fill-in is allowed in ILU process, we obtain the so called ILU(0) preconditioner. The zero pattern of L and U is exactly the same as that of A. The accuracy of this ILU(0) factorization preconditioning may be not sufficient in many cases. The more accurate ILU(k) factorization allows more fill-ins by using the notion of level-fill-in. ILUT approach uses an alternative method to drop fill-ins. Instead of levels, it drops entries based on the numerical values of the fill-in elements. If the value is smaller than some threshold, it is zeroed out. Details are discussed in [22]. When A is SPD, the incomplete Cholesky (IC) factorization is used as a preconditioner for the CG method. However, the incomplete Cholesky factorization for an SPD matrix does not always exist. The modified incomplete Cholesky (MIC) factorization discussed in [19] is used, which exists for any SPD matrix.

As mentioned in Section 4.2, the performance of a triangular solve on GPUs deteriorates severely when the number of level increases. Triangular systems obtained from ILU or IC factorizations

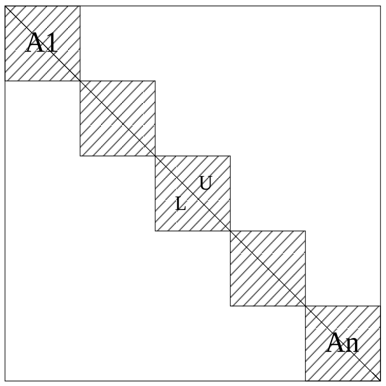


Figure 3: Block Jacobi+ILU Preconditioner

which allow many fill-ins usually have a large number of levels. Although MMD reordering can help increase parallelism in some cases, in general, triangular solves on GPUs are still slower than on the CPU due to their sequential nature. Therefore, the triangular solves which are slow on the GPU can be executed on the CPU for better performance. This implies transferring data (the preconditioned vector) between CPU and GPU in each iteration step. In CUDA, page-locked/mapped host memory can be used to this portion of data to achieve higher memory bandwidth [18]. Although this violates one of the performance guidelines in [18], '*minimize data transfer between the host and the device...even if that means running kernels with low parallelism computations*', this hybrid CPU/GPU computation still exhibits a higher performance than performing the triangular solves on GPUs.

To obtain a higher speed-up from the GPU, parallel preconditioners that are better suited for GPU parallelism must be developed and implemented. The Block Jacobi preconditioners, multi-color SSOR/ILU(0) preconditioners and polynomial preconditioners discussed in the following sections try to achieve high performance in this way.

5.2 Block Jacobi Preconditioners

The simplest parallel preconditioner might be the block Jacobi technique which breaks up the system into independent local sub-systems. These subsystems correspond to the diagonal blocks in the reordered matrix illustrated in Figure 3. The global solution is obtained by adding solutions of all local systems and these local systems can be solved approximately by ILU factorizations. The setup and application phases of the preconditioner can be performed locally without data dependency. Therefore, one CUDA thread block is assigned to each local block and no global synchronization or communication is needed. Threads within each thread block can cooperate to solve the local system and save the local results to the global output vector.

Although block Jacobi preconditioner with a large number of blocks can provide high parallelism, a well-known drawback is that it usually results in a much larger number of iterations to converge. The benefits of increased parallelism may be outweighed by the increased number of iterations.

5.3 Multi-Color SSOR/ILU(0) Preconditioners

A graph G is p-colorable if its vertices can be colored in p colors such that no two adjacent vertices have the same color. A simple greedy algorithm known as sequential coloring algorithm can provide an adequate coloring in a number of colors upper bounded by $\Delta(G) + 1$, where $\Delta(G)$ is the maximum degree of vertices of G.

Suppose a graph is colored by p colors and its vertices are reordered into p blocks by colors. Then its reordered adjacency matrix A has the following structure:

$$\begin{bmatrix} D_1 & & & & \\ & D_2 & & -F & \\ & & \ddots & & \\ & -E & & D_{p-1} & \\ & & & & D_p \end{bmatrix},$$

where the p diagonal blocks are diagonal matrices since no two vertices of the same color are adjacent. If a pointer array *ptr* is defined as *ptr* (i) points to the first row of ith color block, then the MC-SOR (multi-color SOR) sweep will take the form shown in Algorithm 2.

Algorithm 2 Multi-color SOR Sweep

```
1: for $col = 1, \dots, p$ do
2:     $n_1 = ptr(col)$
3:     $n_2 = ptr(col + 1) - 1$
4:     $r(n_1 : n_2) = \omega * (rhs\,(n_1 : n_2) - A(n_1 : n_2, :) * x)$
5:     $x(n_1 : n_2) = x(n_1 : n_2) + D_i^{-1} * r(n_1 : n_2)$
6: end for
```

In Algorithm 2, the loop in line 1 is from the first color to the last one. For each color, a scaling by the diagonal matrix D_i and a matrix-by-vector product $A(n_1 : n_2, :) * x$ are performed. This loop is sequential. However, the scaling and matrix-by-vector product are easy to parallelize. A multi-color SOR sweep can be executed efficiently on GPUs when the number of colors is small, since the number of required matrix-by-vector products is small and the size of the sub-matrix for each color is large. The Symmetric SOR (SSOR) sweep consists of the SOR sweep followed by a backward SOR sweep, i.e, from the last color to the first one. The MC-SSOR(k) multisweep, in which k SSOR sweeps are performed instead of just one, can be used as a preconditioner.

The MC-ILU(0) (multi-color ILU(0)) preconditioner is another preconditioner which achieves good parallelism in preconditioning operations with multi-color reordering. The L/U factors obtained from ILU(0) factorization of multi-color reordered matrices have the structure shown in Figure 4. It is not hard to see that the triangular systems in this form can be solved by a sequence of matrix-by-vector products and diagonal scalings. The parallelism achieved is identical to that of the MC-SSOR preconditioning. However, as is well-known [22], since many more elements are dropped in the multi-color version of ILU(0) than in version with the original ordering, the number of iterations to achieve convergence is usually larger.

For both MC-SSOR(k) and MC-ILU(0), the parallelism is of order of n/p, where p is the number of colors. When p is large, a possible solution to achieve good performance in the SOR sweep and the triangular solve is to sparsify the graph. This sparsification can be carried out as follows: First execute the greedy algorithm to generate a coloring with p colors. Since p is upper bounded by $\Delta(G)+1$, then take all nodes whose degree is larger than or equal to $p-1$, and remove their edges with relatively small weight. This process can be repeated a small number of times until a small enough p is obtained. This sparsification is usually effective in reducing the number of colors.

By applying the multi-color ordering of the sparsified matrices to the original matrices, we can obtain an approximated multi-color ordering of the original matrices. With this ordering,

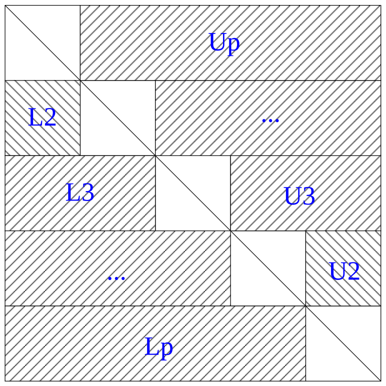


Figure 4: MC-ILU(0) Factorization

all diagonal blocks, D_is are not diagonal matrices anymore but the off-diagonal entries of D_is are small. If these small entries can be ignored, we can use the same algorithms to perform approximate MC-SSOR/ILU(0) preconditioning.

5.4 Least-Squares Polynomial Preconditioners

The polynomial preconditioning matrix M is defined by $M^{-1} = s(A)$, where s is some polynomial. Thus the right preconditioned system can be written as $As(A)x = b$. There is no need to construct the matrices $s(A)$ or $As(A)$ explicitly since for an arbitrary vector v, $As(A)v$ can be computed by a sequence of sparse matrix-vector products.

Consider the inner product on the space $\mathbb{P}_k$ the space of polynomials of degree not exceeding k:

$$\langle p, q\rangle = \int_l^u p(\lambda)q(\lambda)\omega(\lambda)\, d\lambda$$

$p(\lambda)$ and $q(\lambda)$ are polynomials and $\omega(\lambda)$ is some non-negative weight on interval $[l, u]$. The corresponding norm $\sqrt{\langle p, p\rangle}$ is called ω-norm, denoted by $\|p\|_\omega$. Our aim is to find the polynomial s_{k-1} that minimizes $\|1 - \lambda s(\lambda)\|_\omega$ over all polynomials s of degree less than k. This polynomial s_{k-1} is called the *least-squares polynomial* [22].

The explicit formula of $s(k)$ using the Jacobi weights is given and the coefficients of $s(k)$ up to degree 8 is tabulated in [22]. As discussed in [9], the function $\phi(t)$ can be approximated as the least-squares approximation by the polynomial $\phi_k(t)$,

$$\phi(t) \approx \phi_k(t) = \sum_{j=1}^{k} \gamma_j \mathcal{P}_j(t),\ \gamma_j = \langle \phi(t), \mathcal{P}_j(t)\rangle,$$

where $\{\mathcal{P}_j\}$ is a basis of polynomials which is orthonormal for some L_2 inner-product. The orthogonal basis of polynomials is traditionally computed by the well-known Stieltjes procedure, which utilizes a 3-term recurrence [7].

$$\beta_{j+1}\mathcal{P}_{j+1}(t) = t\mathcal{P}_j(t) - \alpha_j\mathcal{P}_j(t) - \beta_j\mathcal{P}_{j-1}(t). \tag{1}$$

Starting with a certain polynomial $\mathcal{S}_0$, the procedure is shown in Algorithm 3. The inner product and polynomial basis should be carefully chosen in order to avoid numerical integration. The basis

Algorithm 3 Stieltjes Procedure

```
1:  $\mathcal{P}_0 \equiv 0$,
2:  $\beta_1 = \|\mathcal{S}_0\|_{\langle\,\rangle}$, $\mathcal{P}_1(t) = {}^{\mathcal{S}_0(t)}/_{\beta_1}$,
3:  $\gamma_1 = \langle \phi(t), \mathcal{P}_1(t) \rangle$
4:  for $j = 1, \ldots, k$ do
5:      $\alpha_j = \langle t\mathcal{P}_j, \mathcal{P}_j \rangle$,
6:      $\mathcal{S}_j(t) = t\mathcal{P}_j(t) - \alpha_j \mathcal{P}_j(t) - \beta_j \mathcal{P}_{j-1}(t)$,
7:      $\beta_{j+1} = \|\mathcal{S}_j\|_{\langle\,\rangle}$,
8:      $\mathcal{P}_{j+1}(t) = {}^{\mathcal{S}_j(t)}/_{\beta_{j+1}}$,
9:      $\gamma_{j+1} = \langle \phi(t), \mathcal{P}_{j+1}(t) \rangle$
10: end for
```

of Chebyshev polynomials was used in [9]. The inner products and the computation of Stieltjes coefficients are also discussed in detailed in [9].

For solving linear systems, we want to approximate $\phi(A)b$, in which $\phi(A) = A^{-1}$. Assume that A is symmetric and $\Lambda(A)$, the spectrum of A, is included in the interval $[l, u]$. Given an initial guess of the solution x_0, the initial residual is $r_0 = b - Ax_0$. Then the solution can be approximated as follows,

$$x = x_0 + A^{-1}r_0 \approx x_0 + \sum_{j=1}^{k} \gamma_j \mathcal{P}_j(A) r_0.$$

Define $\mathcal{P}_j(A)r_0$ to be v_j, then we have $x \approx x_0 + \sum_{j=1}^{k} \gamma_j v_j$. With the 3-term recurrence in (1), it is easy to derive the following recurrence,

$$\beta_{j+1} v_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$$

With the coefficients α_i, β_i and γ_i obtained from the Stieltjes procedure by setting $S_0 = t$, the vector x which approximates $x_0 + A^{-1}r_0$ can be computed by the iteration shown in Algorithm 4.

Algorithm 4 Approximation of $x_0 + A^{-1}r_0$

```
1: $v_0 = 0$, $v_1 = r_0/\beta_1$
2: $x_1 = x_0 + \gamma_1 v_1$
3: for $j = 1, \ldots, k$ do
4:     $v_{j+1} = (Av_j - \alpha_j v_j - \beta_j v_{j-1}) / \beta_{j+1}$
5:     $x_{j+1} = x_j + \gamma_{j+1} v_{j+1}$
6: end for
```

This method can produce a good approximation of A^{-1} assuming that a tight bound of smallest eigenvalue λ_n and the largest eigenvalue λ_1 can be found. As mentioned in [28], the upper bound must be larger than λ_1 but it should not be too large as this would otherwise result in slower convergence. The simplest technique is to use Gershgorin circle theorem but bounds obtained in this way are usually big overestimates of λ_1. An much better estimation can be obtained inexpensively by using a small number of steps of the Lanczos method [15]. A safeguard term is added to 'quasi'-guarantee that the upper bound is larger than $\lambda_1(A)$. It equals the absolute

MATRIX	N	NNZ	NNZ/ROW
sherman3	5,005	20,033	4.0
memplus	17,758	126,150	7.1
msc23052	23,052	588,933	25.6
bcsstk36	23,052	1,143,140	49.6
dubcova2	65,025	547,625	8.4
cfd1	70,656	949,510	13.4
poisson3Db	85,623	2,374,949	27.7
cfd2	123,440	1,605,669	13.0
boneS01	127,224	3,421,188	26.9
majorbasis	160,000	1,750,416	10.9
copcase1	208,340	1,382,902	6.6
af_shell8	504,855	9,046,865	17.9
parafem	525,825	2,100,225	4.0
ecology2	999,999	2,997,995	3.0
lap7pt	1,000,000	6,940,000	6.9
spe10	1,094,421	7,598,799	6.9
thermal2	1,228,045	4,904,179	4.0
atmosmodd	1,270,432	8,814,880	6.9
atmosmodl	1,489,752	10,319,760	6.9

Table 3: Matrices used for experiments

value of the last element of the normalized eigenvector associated with the largest eigenvalue of the tridiagonal matrix. This safeguard is not theoretical but is observed to be generally effective[28]. The computations in preconditioning operation consists of the SpMV and level-1 BLAS vector operations, both of which can be performed efficiently on GPUs. Typically a small number of steps of Lanczos iterations are enough to provide a good estimate of extreme eigenvalues. The Lanczos algorithm can be accelerated by GPUs as well, since the computations it required are also SpMVs and level-1 BLAS vector computations.

6 Experiment Results

The experiments were conducted on a workstation with Intel Xeon E5504 Processor (4M Cache, 2.00 GHz, 8-core) and an NVIDIA TESLA C1060 GPU (240 cores, 1.3 GHz, 4GB memory) running 64-bit Linux system. The CUDA Toolkit and SDK 3.1 are used for programming. The CUDA kernels were compiled by NVIDIA CUDA Compiler (nvcc) with flag -arch sm_13 in order to enable double precision. The CPU program is compiled by g++ compiler using -O3 optimization level.

For the iterative linear solves, the stopping criterion for convergence is that the relative residual be $\leq 10^{-6}$. The maximum number of iterations allowed is $1,000$. The linear systems used are taken from the University of Florida sparse matrix collection [8] and a few cases from reservoir simulations were also used. The size, number of non-zeros and average number of non-zeros per row of each matrix are tabulated in Table 3.

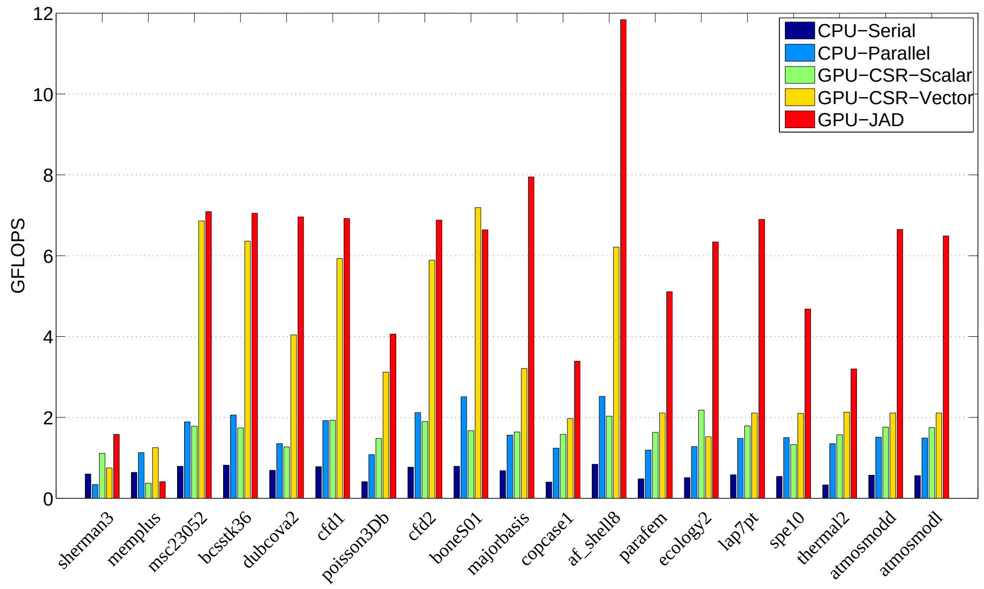


Figure 5: CPU/GPU SpMV Kernels Performance

6.1 SpMV Kernels

We first compare the performance of CPU and GPU SpMV kernels for double precision floating point arithmetic. On the CPU, the SpMV kernel is implemented using the CSR format and parallelized with OpenMP on the multi-core CPU. The GPU SpMV kernel uses two CSR format variants: CSR-scalar and CSR-vector, and a JAD format implementation. Each SpMV kernel is executed 100 times for each matrix and the execution speed is measured in GFLOPS, which is shown in Figure 5. According to the experimental results, the CSR-vector kernel performs better than the CSR-scalar kernel and due to a better memory access pattern, the JAD format kernel yield a higher throughput than the two CSR format kernels. Relative to the sequential CPU SpMV kernel, the speedup of the GPU SpMV kernel can reach a factor of 10.

The time of transferring data between CPU and GPU is not included in calculating the speed. This is reasonable for relatively small matrices, since in the application of iterative solver, the sparse matrices and vectors do not need to be transferred back and forth in each iteration. This transfer takes place only twice: at the beginning for the initial data and at the end when the solution is copied back.

For diagonally structured matrices, the SpMV kernel in DIA format can be much more efficient than the general-purpose sparse matrix format like the CSR or the JAD format since indirect addressing is avoided. The test cases for DIA format SpMV kernel consist of a standard 7-point finite difference discretization of the 3-D Laplacian operator and two matrices from the UF sparse matrix collection of which all non-zeros are in a small number of diagonals. The performance is shown in Figure 6, from which it is clear that the SpMV kernel in DIA format is much faster than the CSR and the JAD formats for these matrices.

6.2 Sparse Triangular Solve on GPUs

Table 4 shows the numbers of levels in level scheduling and the performance of solving triangular systems obtained from ILU factorization on the CPU and the GPU. The performance of the

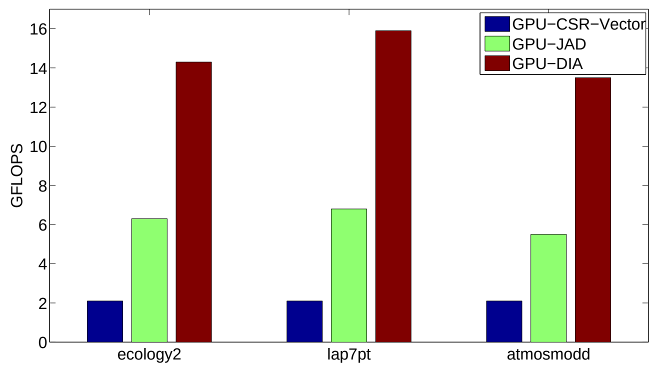


Figure 6: GPU DIA SpMV Kernel Performance

Cases	CPU MFLOPS	GPU-Row MFLOPS	GPU-Col MFLOPS	GPU-Lev	
				#lev	Mflops
parafem	402	3.0	3.9	7	1061
thermal2	268	2.8	3.7	1,239	376
poisson3Db	354	9.5	11.6	136	812
atmosmodd	471	2.8	3.5	352	554
atmosmodl	451	2.7	3.4	432	541

Table 4: Triangular Solve with Level Scheduling

row/column versions of triangular solves on the GPU in which only the dot products or the saxpy operations are parallelized is extremely low, reaching only about 5 MFLOPS whereas it can reach hundreds of MFLOPS on the CPU. For the cases shown in Table 4, by level scheduling, the performance of sparse triangular solves on the GPU can outperform that on the CPU. Table 5 shows that MMD reordering can be used to reduce the number of levels and thus improve performance of sparse triangular solves on the GPU.

Much higher parallelism can be achieved for solving the special triangular systems obtained from ILU(0) factorization of multi-color reordered systems. For matrices which can be colored with a small number of colors, this particular triangular solve can reach a much higher FLOPS rates on GPUs. Table 6 shows the performance of this triangular solves on our GPU and CPU platforms.

Cases	CPU MFLOPS	GPU-Lev		GPU-MMD-Lev	
		#lev	MFLOPS	#lev	Mflops
thermal2	268	1,239	376	32	812
atmosmodd	471	352	554	6	953
atmosmodl	451	432	541	6	949
bcsstk36	627	4,457	64	274	282

Table 5: Triangular Solve with Level Scheduling and Reordering

Cases	GPU-GFLOPS	CPU-GFLOPS
majorbasis	2.28	0.58
atmosmodd	2.62	0.54
poisson3Db	2.19	0.41
atmosmodd	2.52	0.50

Table 6: Triangular Solve of MC-ILU(0) Factors

Cases	Triangular Solve on	Prec Time (sec)	#iters	Iter Time Speedup
parafem	GPU	1.39	333	2.1
thermal2	GPU	2.96	504	3.1
ecology	GPU	1.94	512	1.9
dubcova	CPU	0.39	71	1.3
bcsstk36	CPU	0.61	145	1.3
cfd2	CPU	0.94	212	1.5
boneS01	CPU	4.60	258	1.5
msc23052	CPU	0.56	250	1.4
af_shell8	CPU	8.44	126	1.8
cfd1	CPU	0.78	210	1.5

Table 7: GPU-IC-CG Method

6.3 IC/MIC-CG method

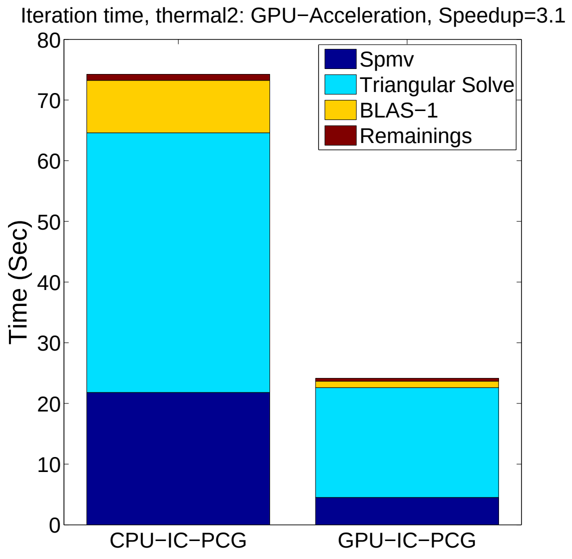
In our implementation of the GPU-accelerated IC/MIC preconditioned CG method, the SpMV and level-1 BLAS operations are performed in parallel on the GPU but the triangular solves of the preconditioning operations are executed on the CPU if they cannot be accelerated using level scheduling on the GPU.

Figure 7(a) and Figure 7(b) show the time breakdown of the IC/MIC PCG method and the GPU-accelerated IC/MIC PCG method for the case *thermal2* and the case *cfd2*. As shown, the running time of the SpMVs and the BLAS-1 operations is reduced by factors of 8 and 10 on average respectively. In the case *thermal2*, the triangular solves can be accelerated by the GPU using level scheduling. As shown in Figure 7(a), the running time of the triangular solves is reduced by a factor of 2.2. So the overall speedup is a factor of 3.1. However, in the case *cfd2*, the triangular solves are performed on the CPU. As shown in Figure 7(b), the running time of the preconditioning operations is slightly increased since a data transfer between the CPU and the GPU is now required. So the overall speedup is a factor of 1.5, which reflects Amdahl's Law, describing the performance improvement when only a fraction of the calculation is parallelized.

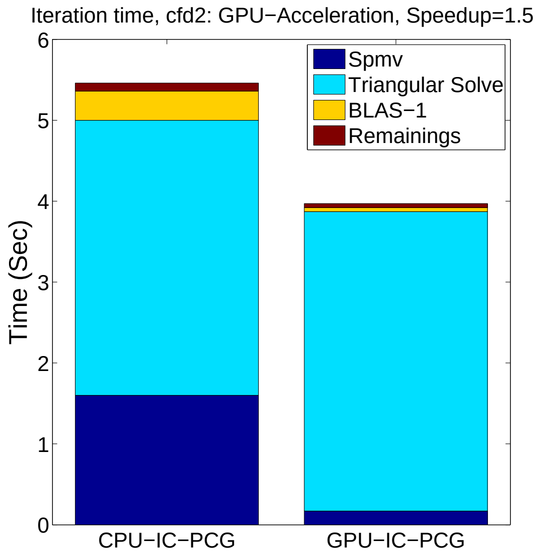
Table 7 shows the performance of all SPD cases using the GPU-accelerated IC/MIC preconditioned CG method. The numbers of iterations to converge, the preconditioner construction time and the speedup of the iteration time compared to the CPU counterpart are tabulated.

6.4 L-S Polynomial-CG method

Table 8 shows the number of iterations, the preconditioner construction time, and the optimal degrees of the least-squares polynomials for all cases. High level fill-in IC/MIC preconditioners can



(a) thermal2



(b) cfd2

Figure 7: Time Breakdown of IC/MIC-CG method

be expensive to build. However, the time of building L-S polynomial preconditioner reduces to a few steps of the Lanczos algorithm. This time is often small. By comparing Table 7 and Table 8, we see that the construction of the least-squares polynomial preconditioners is less expensive than the IC/MIC preconditioner. Figure 8 shows the speedup for all cases in terms of iteration time. As shown, the L-S polynomial preconditioned CG method can reduce the iteration time by up to a factor of 7 compared to the GPU-accelerated IC/MIC preconditioned CG method.

Cases	#Iters	Prec Time (sec)	Polyn Degree
dubcova	17	0.36	10
bcsstk36	66	0.14	60
cfd2	20	0.60	80
boneS01	235	0.37	10
parafem	41	0.61	30
msc23052	98	0.13	40
thermal2	147	2.85	20
af_shell8	68	1.41	20
ecology	323	1.25	20
cfd1	58	0.39	30

Table 8: L-S Polynomial Preconditioner

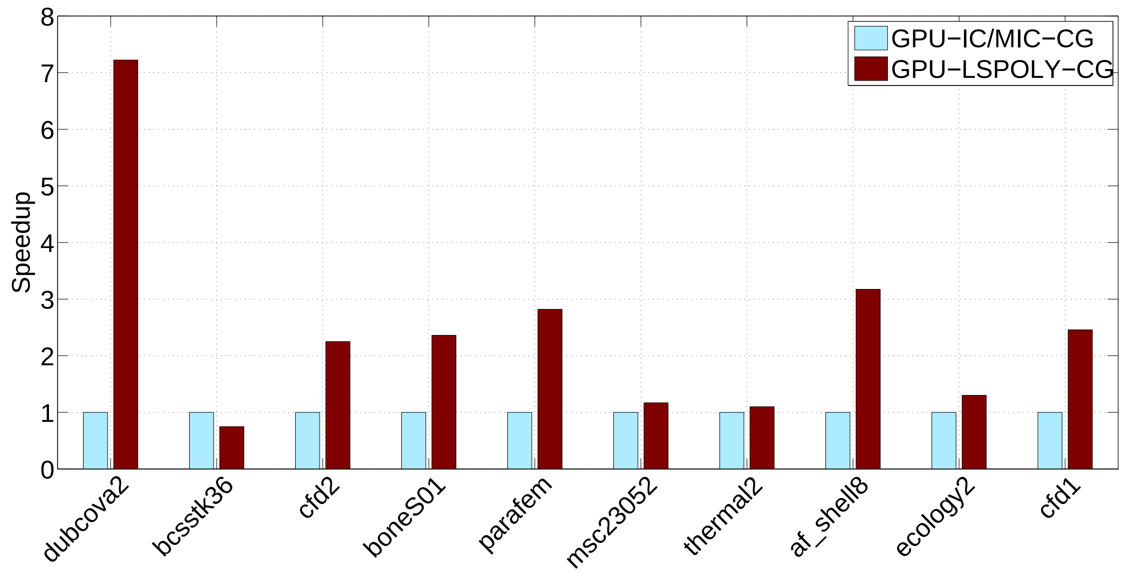


Figure 8: GPU LSPOLY-CG method

6.5 ILUT-GMRES Method

Now we discuss the performance of GPU-accelerated ILUT preconditioned GMRES methods. The ILUT preconditioner is implemented using the dual threshold method suggested in [21]. Similarly to the IC-PCG method, SpMV and level-1 BLAS operations are performed on the GPU whereas the triangular solves are executed on the CPU if they cannot be accelerated by the GPU using level scheduling. The Krylov subspace dimension is set to 40.

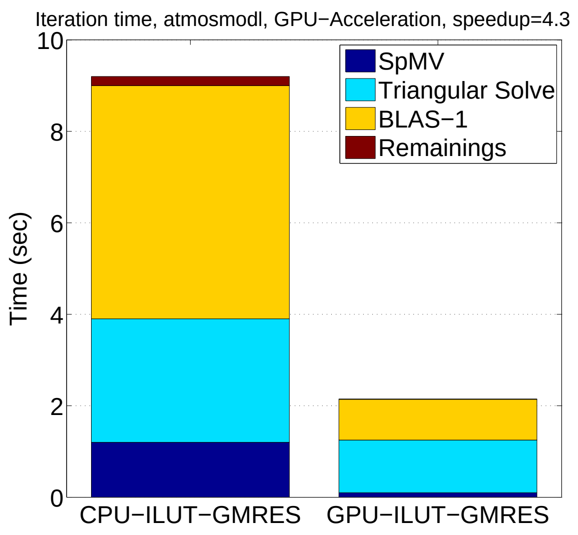
Figures 9(a) and 9(b) show the time breakdown of the ILUT preconditioned GMRES method and the GPU-accelerated ILUT preconditioned GMRES method for the case *atmosmodd* and the case *spe10*. As shown, the running time of the SpMV and BLAS-1 operations is reduced by factors of 12 and 10 on average respectively. In the case *atmosmodd*, the triangular solves can be accelerated by the GPU using level scheduling. As shown in Figure 9(a), the running time of the triangular solves is reduced by a factor of 2.4. So the overall speedup is a factor of 4.3. However, in the case *spe10*, the triangular solves are performed on the CPU. As shown in Figure 9(b), the running time of the preconditioning operation is slightly increased since data transfer between CPU and GPU is now required. So the overall speedup is a factor of 3.4.

Table 9 shows the numbers of iterations to converge and the speedups of the iteration time compared to the CPU counterpart for all non-symmetric cases by using the GPU-accelerated ILUT preconditioned GMRES method.

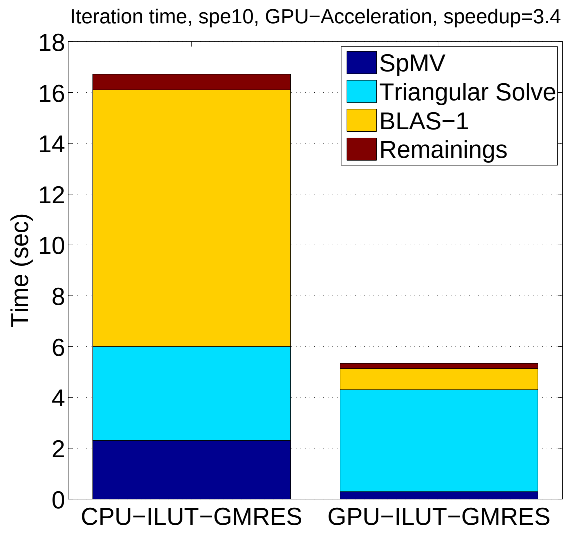
6.6 MC-SSOR/ILU(0)-GMRES Method

For matrices which are colorable with a small number of colors, MC-SSOR preconditioners could be more efficient, since the preconditioning operation is highly parallel and its convergence rate is comparable to that of ILU preconditioners. It is also much less expensive to build. The time of building multi-color SSOR preconditioner is essentially the time of the multi-color reordering. This time is typically small.

Figure 10 shows the numbers of colors, optimal k values of MC-SSOR(k) preconditioner, the



(a) atmosmodl

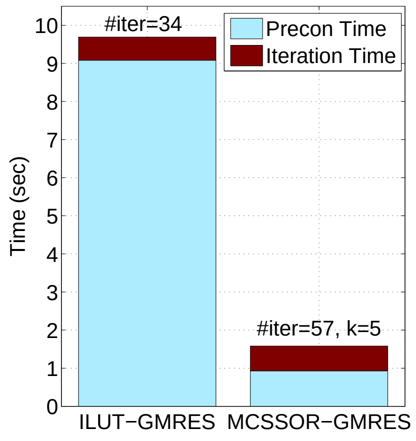


(b) spe10

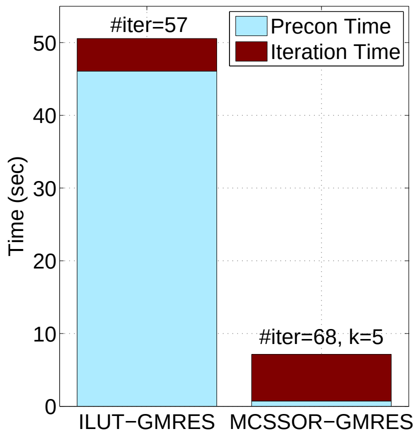
Figure 9: Time Breakdown of ILUT-GMRES method

Cases	Triangular Solve on	#Iters	Iter Time Speedup
atmosmodd	GPU	57	3.2
atmosmodl	GPU	36	4.3
poisson3Db	GPU	34	2.5
majorbasis	CPU	5	2.0
sherman3	CPU	55	3.7
memplus	CPU	23	1.5
copcase1	CPU	10	2.5
spe10	CPU	110	3.4

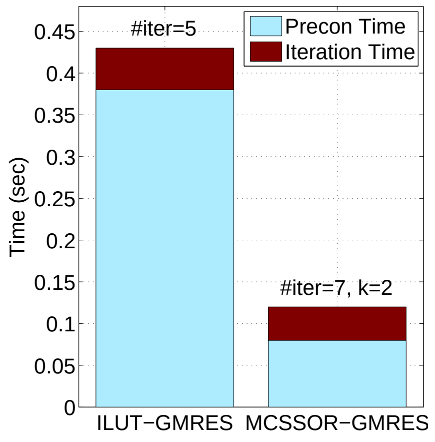
Table 9: GPU-ILUT-GMRES Method



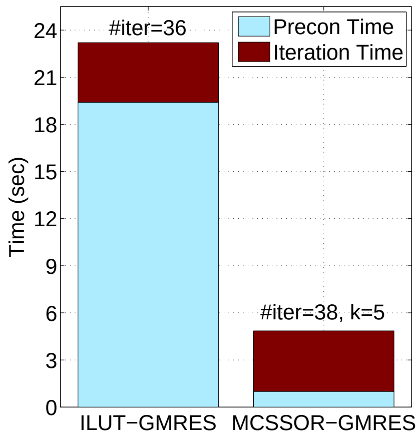
(a) poisson3Db, #col=14



(b) atomsmodd, #col=2



(c) majorbasis, #col=6



(d) atomsmodl, #col=2

Figure 10: ILUT/MC-SSOR-GMRES Method

numbers iterations required, and the total time for solving these linear systems using the GPU-accelerated MC-SSOR preconditioned GMRES method compared to the ILUT preconditioner. Table 10 compares the numbers of iterations and the iteration time required using the MC-ILU(0) preconditioned GMRES method and the standard ILU(0) preconditioned GMRES method on the GPU. Compared to the standard ILU(0) preconditioner, the MC-ILU(0) preconditioned GMRES method needs more iterations to achieve convergence. However, the overall iteration time is reduced.

For matrices which cannot be colored with a small number of colors, the sparsification method discussed in section 5.3 could be invoked to reduce the numbers of colors. This sparsification method is applied to two cases to improve the efficiency of MC-SSOR/ILU(0) preconditioners. The numbers of colors before and after sparsification, numbers of iterations, and iteration time are tabulated in Table 11. As is shown, the numbers of colors are effectively reduced and the performance is improved.

Cases	MC-ILU(0)		ILU(0)	
	#iters	Iter Time	#iters	Iter Time
poisson3Db	96	0.53	96	0.85
atomsmodd	219	5.32	176	7.94
majorbasis	12	0.04	9	0.07
atomsmodl	96	2.66	58	3.00

Table 10: ILU(0)/MC-ILU(0)-GMRES Method

Cases	Precon	Sparsify	#col	#iter	Time
memplus	MC-SSOR	No	22	47	0.23
		Yes	3	32	0.07
	MC-ILU(0)	No	22	104	0.35
		Yes	3	115	0.20
spe10	MC-SSOR	No	340	202	28.14
		Yes	5	200	10.88
	MC-ILU(0)	No	340	559	29.05
		Yes	53	607	22.31

Table 11: MC-SSOR/ILU(0) with Sparsification

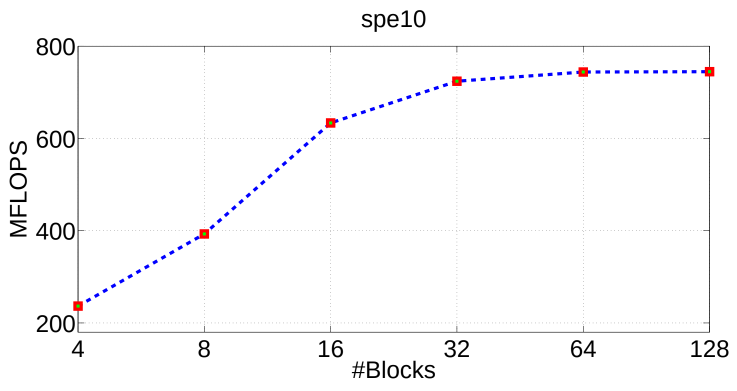
6.7 Block Jacobi-GMRES Method

When using block Jacobi preconditioning, matrices are first partitioned by some graph partitioning algorithm like METIS [14] and reordered. Then ILUT factorization is performed on each diagonal block. The preconditioning operation, the block L/U solves are executed on the GPU. As shown in Figure 11(a), the speed of the Block L/U solve is increased as the number of blocks grows, because more parallelism is available in the computation. However, as shown in Figure 11(b), the number of iterations required increases. The iteration time is shown in Figure 11(c).

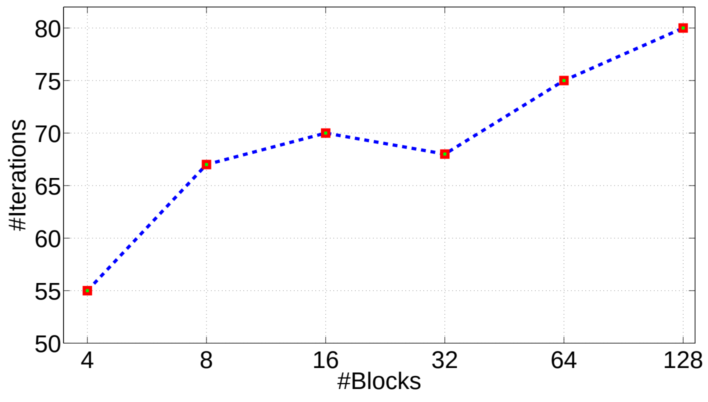
Table 12 shows the performance of PCG methods for solving a 2D/3D Poisson equation. The first row of Table 12 shows the IC/MIC preconditioned CG method running on the CPU. Row 2 to row 7 show the GPU-accelerated PCG methods with different preconditioners. By comparing the rows of Table 12, we can see that the GPU-acceleration in the PCG methods and the performance difference between different preconditioners.

Precon	Lap2d		Lap3d	
	#iters	time(sec)	#iters	time(sec)
CPU-IC	94	2.24	48	4.40
GPU-IC	94	1.14	48	2.14
GPU-LS-POLY	55	0.85	16	0.90
GPU-IC(0)	254	1.91	80	2.04
GPU-MC-IC(0)	442	1.12	129	1.15
GPU-MC-SSOR	309	1.83	104	2.25
GPU-BJ	251	2.27	95	2.73

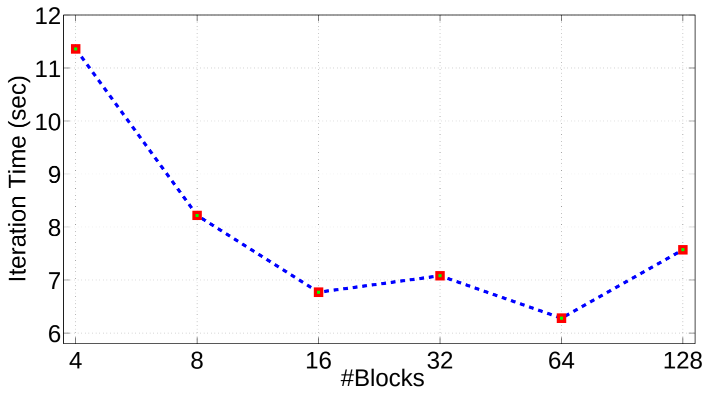
Table 12: Performance Comparison of All Preconditioning



(a) Block L/U Solve Performance



(b) #Iterations



(c) Iteration Time

Figure 11: Block Jacobi ILU-GMRES Method

7 Conclusion

In this work, we have explored high performance sparse matrix-vector product (SpMV) kernels in different formats on current many-core platforms and used them to construct effective iterative linear solvers with several preconditioner options. Since the performance of triangular solve is low on GPUs, this computation can be accomplished by CPUs. By this hybrid CPU/GPU computations, IC preconditioned CG method and ILU preconditioned GMRES method are adapted to a GPU environment and achieve performance gains compared to its CPU counterpart. In order to achieve better GPU-acceleration, a few preconditioners with enhanced parellelism were considered. Among these, the block Jacobi preconditioner is the simplest but it is usually not efficient since it requires many iterations to converge. For matrices which can be colored with a small number of colors, multi-color SSOR/ILU(0) preconditioners could provide a better performance since their preconditioning operations can be executed with high parallelism and they can yield comparable convergence rates to the standard ILU preconditioners. For symmetric linear systems, the Least-Squares polynomial preconditioner can be much more efficient for the PCG method since its preconditioning operation relies on the SpMV kernel.

Based on the experimental results reported here and elsewhere, we observe that, when used as general purpose many-core processors, current GPUs provide a much lower performance advantage for irregular (sparse) computations than they can for more regular (dense) computations. However, when used carefully, GPUs can still be beneficial as co-processors to CPUs to speed-up complex computations. We highlighted a few alternative approaches to standard ones in the arena of iterative sparse linear system solvers.

In our future work we will take a more careful look of the GPU SpMV kernels using CUDA Visual Profiler to find the potential throughput bottleneck. We will also consider the hybrid format SpMV kernel suggested in [5] and explore more efficient preconditioning techniques for GPU-accelerated iterative linear solvers.

References

[1] Anant Agarwal and Markus Levy, *The kill rule for multicore*, DAC '07: Proceedings of the 44th annual Design Automation Conference (New York, NY, USA), ACM, 2007, pp. 750–753.

[2] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov, *Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects*, Journal of Physics: Conference Series **180** (2009), no. 1, 012037.

[3] Marco Ament, Gunter Knittel, Daniel Weiskopf, and Wolfgang Strasser, *A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform*, PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (Washington, DC, USA), IEEE Computer Society, 2010, pp. 583–592.

[4] Muthu Manikandan Baskaran and Rajesh Bordawekar, *Optimizing sparse matrix-vector multiplication on GPUs*, Tech. report, IBM Research, 2008.

[5] Nathan Bell and Michael Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (New York, NY, USA), ACM, 2009, pp. 1–11.

[6] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder, *Sparse matrix solvers on the gpu: conjugate gradients and multigrid*, ACM Trans. Graph. **22** (2003), no. 3, 917–924.

[7] Philip J. Davis, *Interpolation and approximation*, Blaisdell, Waltham, MA, 1963.

[8] Timothy A. Davis, *University of florida sparse matrix collection, na digest*, 1994.

[9] J. Erhel, F. Guyomarc, and Y. Saad, *Least-squares polynomial filters for ill-conditioned linear systems*, Tech. Report umsi-2001-32, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001.

[10] F.Vazquez, E.M.Garzon, J.A.Martinez, and J.J.Fernandez, *The sparse matrix vector product on GPUs*, Tech. report, Dept of Computer Architecture and Electronics, University of Almeria, 2009.

[11] A. George and W. H. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Rev. **31** (1989), no. 1, 1–19.

[12] Serban Georgescu and Hiroshi Okuda, *Conjugate gradients on graphic hardware: Performance & Feasibility*, 2007.

[13] Rohit Gupta, *A GPU implementation of a bubbly flow solver* , Master's thesis, Delft Institute of Applied Mathematics, Delft University of Technology, 2628 BL Delft, The Netherlands, 2009.

[14] George Karypis and Vipin Kumar, *Metis - a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0*, Tech. report, University of Minnesota, Department of Computer Science / Army HPC Research Center, 1998.

[15] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, Journal of Research of the National Bureau of Standards **45** (1950), 255–282.

[16] Ruipeng Li, Hector Klie, Hari Sudan, and Yousef Saad, *Towards realistic reservoir simulations on manycore platforms*, SPE Journal (2010), 1–23, submitted.

[17] NVIDIA, *Cublas library user guide 3.0*, 2010.

[18] ———, *NVIDIA CUDA programming guide 3.0*, 2010.

[19] Yves Robert, *Regular incomplete factorizations of real positive definite matrices*, Linear Algebra and its Applications **48** (1982), 105 – 117.

[20] Y. Saad, *SPARSKIT: A basic tool kit for sparse matrix computations*, Tech. Report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[21] ———, *ILUT: a dual threshold incomplete ILU factorization*, Numerical Linear Algebra with Applications **1** (1994), 387–402.

[22] ———, *Iterative methods for sparse linear systems, 2nd edition*, SIAM, Philadelpha, PA, 2003.

[23] Y. Saad and M. H. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing **7** (1986), 856–869.

[24] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, *Scan primitives for gpu computing*, Graphics Hardware 2007, ACM, August 2007, pp. 97–106.

[25] H. Sudan, H. Klie, R. Li, and Y. Saad, *High performance manycore solvers for reservoir simulation*, 12th European Conference on the Mathematics of Oil Recovery, 2010.

[26] Vasily Volkov and James Demmel, *LU, QR and Cholesky factorizations using vector capabilities of GPUs*, Tech. report, Computer Science Division University of California at Berkeley, 2008.

[27] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan, *Solving sparse linear systems on nvidia tesla gpus*, ICCS '09: Proceedings of the 9th International Conference on Computational Science (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 864–873.

[28] Yunkai Zhou, Yousef Saad, Murilo L. Tiago, and James R. Chelikowsky, *Parallel self-consistent-field calculations via Chebyshev-filtered subspace acceleration*, Phy. rev. E **74** (2006), 066704.