Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 398438, 12 pages
http://dx.doi.org/10.1155/2013/398438

Research Article

Efficient CUDA Polynomial Preconditioned Conjugate Gradient Solver for Finite Element Computation of Elasticity Problems

Jianfei Zhang[1] and Lei Zhang[2]

[1] *College of Mechanics and Materials, Hohai University, 1 Xikang Road, Nanjing 210098, China*
[2] *China Institute of Water Resources and Hydropower Research, Beijing 100044, China*

Correspondence should be addressed to Jianfei Zhang; zhjf77@gmail.com

Received 19 July 2013; Accepted 6 September 2013

Academic Editor: Song Cen

Graphics processing unit (GPU) has obtained great success in scientific computations for its tremendous computational horsepower and very high memory bandwidth. This paper discusses the efficient way to implement polynomial preconditioned conjugate gradient solver for the finite element computation of elasticity on NVIDIA GPUs using compute unified device architecture (CUDA). Sliced block ELLPACK (SBELL) format is introduced to store sparse matrix arising from finite element discretization of elasticity with fewer padding zeros than traditional ELLPACK-based formats. Polynomial preconditioning methods have been investigated both in convergence and running time. From the overall performance, the least-squares (L-S) polynomial method is chosen as a preconditioner in PCG solver to finite element equations derived from elasticity for its best results on different example meshes. In the PCG solver, mixed precision algorithm is used not only to reduce the overall computational, storage requirements and bandwidth but to make full use of the capacity of the GPU devices. With SBELL format and mixed precision algorithm, the GPU-based L-S preconditioned CG can get a speedup of about 7–9 to CPU-implementation.

1. Introduction

Recently, the graphics processing unit (GPU) has evolved from a fixed-function special-purpose processor into a highly parallel, multithreaded, many core processor with tremendous computational horsepower and very high memory bandwidth. This makes them the ideal processor to accelerate data parallel applications including graphic and nongraphic problems. However, programming GPUs for general computation was a great challenge in the past. Early efforts to exploit the GPU for nongraphical applications used shading languages [1, 2], such as DirectX, OpenGL, to port various data parallel algorithms to the GPUs, which limited accessibility to the tremendous capability of GPUs for developers from various fields. The advent of compute unified device architecture (CUDA) makes NVIDIA GPUs fully programmable and greatly facilitates the general-purpose applications targeted at GPUs. So far, these applications have ranged from fluid dynamics [3] and molecular dynamics [4] to biomechanics [5], surgical simulation [6], and earthquake modeling [7].

Elasticity is a general problem in solid mechanics. It is therefore fundamental to the practice of mechanical, civil, structural, and aeronautical engineering and also directly relevant to other branches of engineering and applied science. Finite element method is one of the important numerical techniques to solve elasticity problems. As the finite element discretization of elasticity results in a sparse and symmetric, positive definite linear system of equations, preconditioned conjugate gradient (PCG) has become an important iterative solver. To solve large-scale problems, many parallel PCG algorithms and programs have been developed on multi-CPU parallel computers [8–10] and GPUs [11, 12]. However, some commonly used preconditioners for sequential computers have limited parallelism, for example, ILU and SSOR [13]. Therefore, alternative techniques need be developed to specifically target parallel environments. The polynomial preconditioners [13, 14] are the simple and effective methods for efficient parallel iterative solvers. Only matrix-vector products are required to carry out this preconditioning.

In this paper, we first focus on the efficient sparse matrix-vector product (SpMV), which is the major component of the CG iteration and polynomial preconditioning. We propose a sparse representation called sliced block ELLPACK (SBELL), which is a GPU-friendly variant of the storage format of ELLPACK (ITPACK) [15] and specially designed for the iterative solution of finite element equations arising from elasticity. Based on this new format, SBELL, an efficient CUDA SpMV kernel is implemented on NVIDIA GPUs. Then, an extensive performance evaluation of this new approach has been carried out based on a representative set of test matrices. Secondly, this SpMV kernel is assembled into a polynomial preconditioned CG solver. Several polynomial preconditioners are implemented and evaluated. To increase the performance of PCG kernel, mixed precision technique is exploited, that is low precision operations for inner preconditioning and high precision for outer CG iteration.

The remainder of this paper is organized as follows. Section 2 reviews the aspects related to GPU computing, FEM and PCG. Section 3 introduces the proposed format for computation of SpMV on GPUs. Section 4 presents the mixed precision polynomial preconditioned CG. In Section 5, the performance measured on an NVIDIA Geforce GT430 with a set of representative sparse matrices belonging to diverse finite element equations derived from elasticity problems is presented. Finally, Section 6 summarizes the main conclusions.

2. Backgrounds

2.1. GPU Computing. GPU computing is a new way of GPU programming. It signified broader application support, wider programming language support, and a clear separation from the early "GPGPU" model of programming. This new technique emerged with the advent of NVIDIA G80 unified graphics and compute architecture and CUDA [16]. Typical NVIDIA Fermi-architecture GPUs are based on a scalable array of graphics processing clusters (GPCs), streaming multiprocessors (SMs), and memory controllers. Each SM is a highly parallel multiprocessor supporting up to 32 warps at any given time. Different products were launching with different configurations of GPCs, SMs, and memory controllers to address different applications.

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It provides a software environment that allows developers to code algorithms for execution on the GPUs using C and other programming languages. The CUDA programming model [17] assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This means the kernels execute on a GPU and the rest of the C program executes on a CPU. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Programs manage the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime, which includes device memory allocation and deallocation as well as data transfer between host and device memory.

The core of CUDA parallel programming model is three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization, that are simply exposed to the programmer as a minimal set of language extensions. These abstractions guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

A thread block is a set of concurrently executing threads that can cooperate through barrier synchronization and shared memory. These threads are organized as an array of one-dimension, two-dimension, or three-dimension blocks and each of them has a thread ID within its thread block. Blocks are similarly organized into a one-dimensional, two-dimensional, or three-dimensional grid and each of them has a block ID within its grid. Thread blocks execute independently in any order, in parallel or in series. This independence allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write a code that scales with the number of cores. Threads within a block can communicate with each other by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads, the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages and persistent across kernel launches by the same application.

2.2. Finite Element Method for Elasticity. The displacement-based finite element method introduces an approximation for the displacement field in terms of shape functions and uses a weak formulation of the equations of equilibrium, strain-displacement relations, and constitutive relation to arrive at the linear system

$$\mathbf{Ku} = \mathbf{f}, \tag{1}$$

where $\mathbf{u}$ is the vector of unknown nodal displacements, $\mathbf{f}$ is the vector of nodal forces, and $\mathbf{K}$ is the structure stiffness matrix, given by

$$\mathbf{K} = \int_{\Omega} \mathbf{B}^{\mathbf{T}}\mathbf{DB}d\Omega, \tag{2}$$

with integration over the problem domain Ω. For simplicity, we consider a 2D plane stress problem discretized with elements having nodes without rotational freedoms and linear

elastic material behavior. Then, the components of the right-hand side of (2) are

$$\mathbf{B} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x} & 0 & \frac{\partial \phi_2}{\partial x} & 0 & \cdots & \frac{\partial \phi_M}{\partial x} & 0 \\ 0 & \frac{\partial \phi_1}{\partial y} & 0 & \frac{\partial \phi_2}{\partial y} & \cdots & 0 & \frac{\partial \phi_M}{\partial y} \\ \frac{\partial \phi_1}{\partial y} & \frac{\partial \phi_1}{\partial x} & \frac{\partial \phi_2}{\partial y} & \frac{\partial \phi_2}{\partial x} & \cdots & \frac{\partial \phi_M}{\partial y} & \frac{\partial \phi_M}{\partial x} \end{bmatrix}, \quad (3)$$

where $\phi_1, \phi_2 \ldots \phi_M$ are shape functions associated with the M nodes in the mesh and elasticity matrix

$$\mathbf{D} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{(1-\nu)}{2} \end{bmatrix}, \quad (4)$$

where E is Young's modulus and ν is Poisson's ratio [18].

The structure stiffness matrix in (2) relates to the whole problem domain and is of dimension $n \times n$, where n is the total number of degrees of freedom (DOF) ($n = 2 \times M$ for our 2D problem). For linear elasticity, the structure stiffness matrix is sparse and symmetric positive definite (SPD). The sparse nature of the finite element matrix is a consequence of local support of the basis functions. The symmetry is present because the Galerkin procedure by which the weak form is generated is self-adjoint; that is, the basis and test functions are the same. The Galerkin method is equivalent to minimization of potential energy, and the nonnegativity of the strain energy in the domain leads to positive definiteness of the structure stiffness matrix. In addition, since each node has multiple DOFs, the structures stiffness matrix can be organized into a blocked matrix if the DOFs associated with each node numbered consecutively. For 2D and 3D problems, the block size is 2 and 3, respectively.

2.3. Preconditioned Conjugate Gradient. The conjugate gradient algorithm is one of the best known iterative techniques for solving sparse symmetric positive definite linear systems. As an iterative solver, lack of robustness is a widely recognized weakness of CG. This drawback can be improved by using preconditioning.

Consider a linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is symmetric and positive definite. It is assumed that a preconditioner $\mathbf{M}$ is available and also Symmetric Positive Definite. A mathematically equal preconditioned system is obtained as follows:

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}. \quad (5)$$

Replacing the usual Euclidean inner product in the Conjugate Gradient algorithm by the M-inner product, the following PCG algorithm is obtained [13].

In general, the reliability of iterative techniques, when dealing with various applications, depends much more on the quality of the preconditioner than on the particular iterative solver used. Incomplete LU factorization (ILU) based preconditioners are effective for single processor computational

$\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$, $\mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0$, $\mathbf{p}_0 = \mathbf{z}_0$
for $j = 0, 1, \ldots$, until convergence
  $\alpha_j = (\mathbf{r}_j, \mathbf{z}_j)/(\mathbf{Ap}_j, \mathbf{p}_j)$
  $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{Ap}_j$
  $\mathbf{z}_{j+1} = \mathbf{M}^{-1}\mathbf{r}_{j+1}$
  $\beta_j = (\mathbf{r}_{j+1}, \mathbf{z}_{j+1})/(\mathbf{r}_j, \mathbf{z}_j)$
  $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j$
end for

Algorithm 1: Preconditioned conjugate gradient.

framework. However, for parallel framework, its use is mainly limited as it incurs high parallel communication overhead similar to that of direct solvers. Jacobi preconditioners are commonly used preconditioners for parallel formulations but they are usually not efficient since it requires many iterations to converge. To enhance performance, these preconditioners can themselves be accelerated by polynomial iterations, i.e., polynomial preconditioning. The main advantages of polynomial preconditioning are its simplicity and flexibility. Only matrix-vector products are required to carry out this preconditioning.

In polynomial preconditioning the matrix $\mathbf{M}$ is defined by

$$\mathbf{M}^{-1} = P_m(\mathbf{A}), \quad (6)$$

where $P_m(\mathbf{A})$ is a polynomial in $\mathbf{A}$ with a degree of no more than m.

Thus, the polynomial preconditioned system is given by

$$P_m(\mathbf{A})\,\mathbf{Ax} = P_m(\mathbf{A})\,\mathbf{b}. \quad (7)$$

The most commonly used polynomial preconditioners for SPD linear systems are constructed using Neumann-series, least-squares, and minimax polynomials. Among these, the Neumann preconditioner is the simplest, cheapest (i.e., least construction cost) and the most stable method. It can be constructed as

$$P_m(\mathbf{A}) = \omega\left(1 + \mathbf{G} + \mathbf{G}^2 + \cdots + \mathbf{G}^m\right), \quad (8)$$

where $\mathbf{G} = (\mathbf{I} - \omega\mathbf{A})$ and scalar ω are adjusted so that $\rho(\mathbf{G}) < 1$.

The least-squares and minimax polynomial preconditioners are the two optimal methods, each of which is derived from a specific optimal approximation. See [13] for full details of these polynomial preconditioners.

3. SpMV in the Form of Sliced Block ELLPACK

As we can see from Algorithm 1, the main computational components in PCG are composed of matrix-vector product, preconditioning operation, and vector operations. Since the polynomial preconditioning is actually a serial of matrix-vector products, the efficiency of polynomial PCG solver on NVIDIA GPUs for finite element equations greatly relies on CUDA SpMV kernel. In this section, we consider

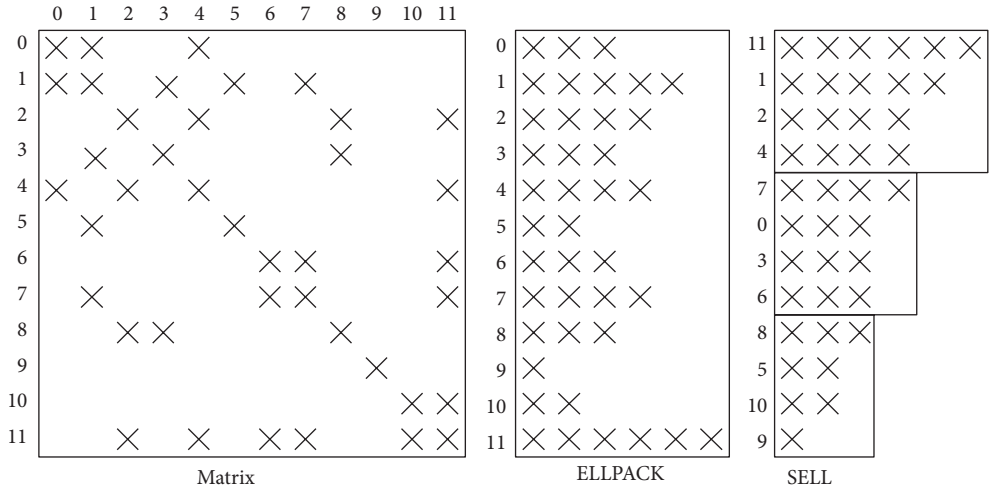

Figure 1: ELLPACK and sliced ELLPACK format of matrix.

the operation $\mathbf{y} = \mathbf{Ax}$, where $\mathbf{A}$ is a sparse matrix and $\mathbf{x}$ and $\mathbf{y}$ are column vectors.

Because the stiffness matrix arising from finite element discretization of elasticity on unstructured mesh is a general sparse SPD matrix, the CSR format [13] is commonly used to store these matrices. This data structure takes three arrays to represent sparse matrix and permits a variable number of nonzeros per row. For a $N \times N$ matrix, the column indices and nonzero entries are stored in the arrays **JA** and **A** of dimension NNz. Array **IA** of dimension $N + 1$ stores the pointers to the beginning of every row in **A** and **JA**, both sorted out by row index. The last entry in **IA** stores NNz, the number of nonzeros in the matrix. For CSR format, a straightforward CUDA implementation is called scalar kernel, which uses one thread per matrix row. The most significant problem of this kernel is that the threads in a warp access **A** and **JA** noncontiguously, which leads to noncoalesced global memory access. An alternative to the scalar method, which we call the vector kernel, assigns one warp to each matrix row. The vector kernel accesses indices and data contiguously and therefore overcomes the principal deficiency of the scalar approach. However, efficient execution of the vector kernel demands that matrix rows contain a number of nonzeros greater than the warp size. That means the performance of the vector kernel is sensitive to matrix row size. Another drawback for both scalar and vector kernels is that the locality of access to vector **x** is not maintained due to the indirect addressing.

ELLPACK [15] was introduced to suit vector computers. This format consists of two dense arrays: array **A** to save the entries and array **JA** to save the column index of every entry. Both arrays are of dimension $N \times MaxNzPerRow$, where N is the number of rows and $MaxNzPerRow$ is the maximum number of nonzeros per row in the matrix. Figure 1 (middle) illustrates the ELLPACK format of matrix, where $N = 12$ and $MaxNzPerRow = 6$. Note that the size of all rows in these compressed arrays **A** and **JA** is the same, because every row with fewer than $MaxNzPerRow$ nonzeros is padded with zeros. Therefore, ELLPACK can be considered

```
spmv_ell_kernel (int N, int MaxNzPerRow, int *JA,
                 float *A, float *x, float *y)
{
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  for (i = tid;  i < N;  i+ = blockDim.x * gridDim.x){
    for (k = 0;  k < MaxNzPerRow; k ++)
      y[i]+ = A[i + N × k] × x[JA[i + N × k]];
  }
}
```

Algorithm 2: SpMV kernel for the ELLPACK format.

as an approach to fit a sparse matrix in a regular data structure similar to a dense matrix. Consequently, this format is appropriate to compute operations with sparse matrices on vector architectures.

Parallelizing SpMV for the ELLPACK format is straightforward: one thread is assigned to each row of the matrix and each thread computes the sparse dot product between the corresponding matrix row and the x vector. As shown in Algorithm 2, if element i of vector **y** is computed by a thread identified by index tid and the arrays store their elements in column-major order, this thread accesses to the elements $\mathbf{A}[i + k \times N]$ and $\mathbf{JA}[i + k \times N]$ with $0 \leq k < MaxNzPerRow$, where k is the column index into the data structures **A** and **JA**. Thanks to the column-major ordering used to store the matrix elements into the data structures, two threads tid and $tid + 1$ can access consecutive memory address; thereby the conditions of coalesced global memory access can be fulfilled. We also can see from the kernel in the form of ELLPACK that every block of threads can complete its computation without synchronization with other blocks. This is because there are no data dependencies in the computation of different elements of **y**, as every thread computes one element of the vector **y**. These two merits, coalesced global

```
spmv_sell_kernel (int N, int S, int T, int *MnzPerRow, int *nnz,
                  int *JA, float *A, float *x, float *y)
{
   int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
   int sid = thread_id/T;
   int tid = thread_id-sid *T;
   int num_sets = (blockDim.x * gridDim.x)/T;
   int nslice = (N + S − 1)/S;
   for (int is = sid; is < nslice; is+ = num_sets){
      for (i = tid;  i < S;  i+ = T){
         int row = is × S + i;
         if (row < N){
            for (k = 0;  k < MnzPerRow[is];  k ++){
               int col = JA[nnz[is] + S × k + i];
               y[i]+ = A[nnz[is] + S × k + i] × x[col]; }
         }
      }
   }
}
```

Algorithm 3: SpMV kernel for the SELL format.

memory access and nonsynchronized executions, improve the performance of SpMV kernels based on ELLPACK.

However, this good performance of ELLPACK only applies when the maximum number of nonzeros per row does not substantially differ from the average. In practice, unstructured meshes do not always meet this requirement. For this case, the percentage of zeros is high in the ELLPACK data structure and there is a relevant amount of padding zeros. This penalty inevitably results in the redundant memory access and arithmetic operations with padding zeros. As described in Algorithm 2, when computing every $u[i]$, with $0 \leq i < N$, the k-loop must iterate until $k = MaxNzPerRow$ is reached in every iteration. Clearly, the ELLPACK format alone is an appropriate choice for representing matrices obtained from structured meshes.

To relieve the inherent drawback of ELLPACK, two variants have been proposed. One is the ELLPACK-R format [19] which consists of two arrays, **A** and **JA** of dimension $N \times MaxNzPerRow$ and an additional integer array called **rl** of dimension N (i.e., the number of rows) with the purpose of storing the actual length of every row, regardless of the number of the zero elements padded. The SpMV based on ELLPACK-R further improves the performance reached by ELLPACK on GPUs due to reduction of useless computation and imbalance of the threads in one warp by the inclusion of the array **rl**. The problem of ELLPACK-R is that too much extra storage is still required to store padding zeros for the matrices obtained from unstructured meshes.

Another variant is the sliced ELLPACK (SELL) format [20], which first permutes the rows of **A** in ascending or descending order; and then, partitions this reordered **A** into slices of S rows and every slice is stored in ELLPACK format. Thus, this format can largely reduce the padding zeros as compared to Figure 1 (middle) with Figure 1 (right), where $S = 4$. The SpMV kernel in this format is described by Algorithm 3, where T threads are assigned to every slice of rows and collaborate in the computation. We suppose T is a multiple of warp size to enhance coalesced memory access. The array **nnz** denotes the beginning index of the slice into **A** and **JA**. As we can see in this Algorithm, the k-loop reaches the maximum value for specific threads into the slice. Then, the runtime of every slice is proportional to the maximum number of nonzeros per row $MnzPerRow[is]$ related to every slice, and it is not necessary that the k-loop for all threads into every warp reaches $MaxNzPerRow$ for all rows. Consequently, the useless iterations are reduced compared with SpMV based on ELLPACK.

For matrices with a natural block structure, blocked formats, for example, blocked CSR (BCSR) [21] and blocked ELLPACK (BELLPACK) [22], have been proposed for SpMV. These proposals compress the sparse matrix by small dense entries blocks with the size is $l \times m$. Figure 2 illustrates the storage of a blocked matrix in the BELLPACK format, where the block size is 2×2. Since only storing one column index per block is needed, the column index storage and transfer can be reduced by up to roughly 2×2. Therefore, the blocked approaches reach better performance than the corresponding nonblocked versions in principle. However, BELLPACK still suffers the same problem as ELLPACK in the case of variable nonzeros across rows.

To fit the finite element stiffness matrix derived from elasticity with the feature of unstructured and blocked sparsity, we here combine the BELLPACK and sliced ELLPACK and propose a sliced block ELLPACK (SBELL) format. Firstly, the sparse blocked matrix is permuted by the number of non-zero blocks of every row in ascending or descending order. Then this permuted matrix is sliced into groups of blocked rows and BELLAPCK is used to store the block entries in each slice. Figure 3 illustrates the SBELL representation of the matrix in Figure 3 (left), where there are 6 blocked rows and slice size

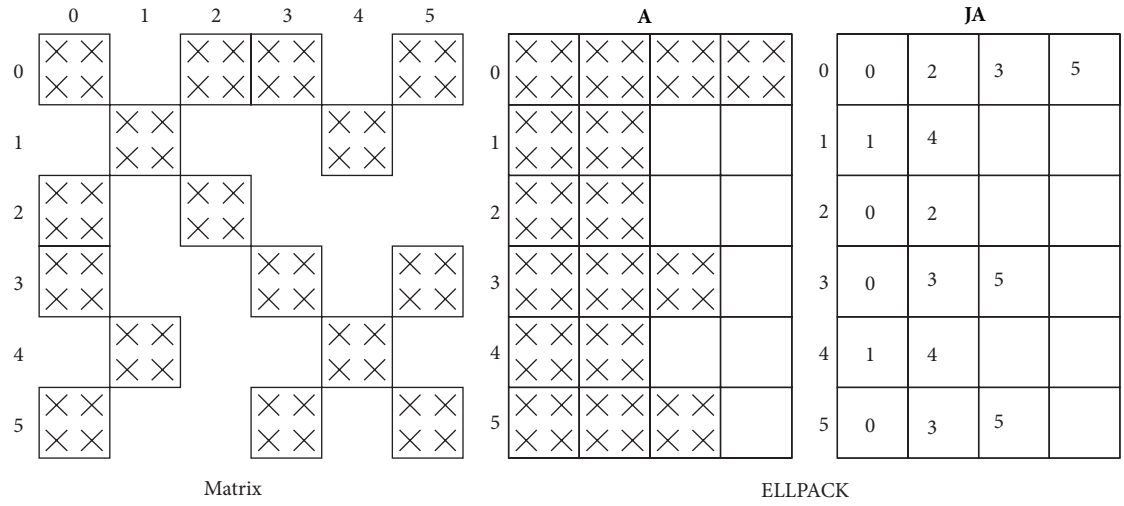

Figure 2: BELLPACK format of matrix.

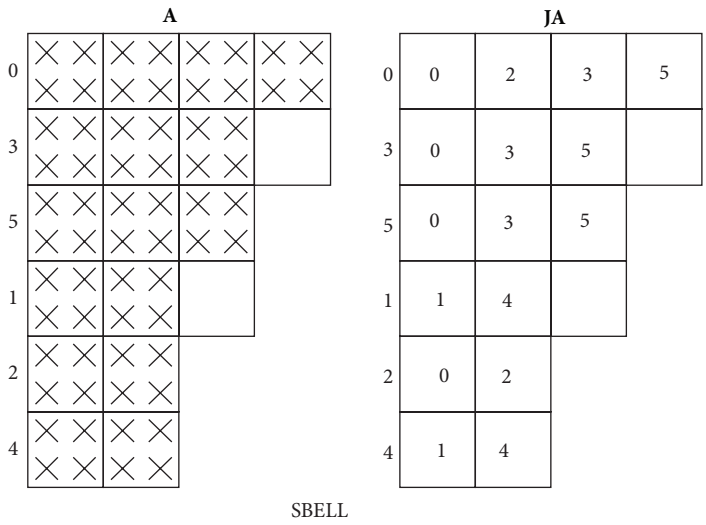

Figure 3: SBELL format of matrix.

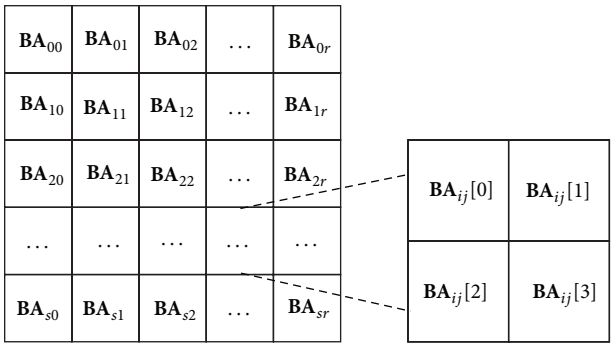

Figure 4: Entries of each slice in the SBELL.

is set to 2. Thanks to permutation and slicing, zero-padding blocks and related useless arithmetic operations are greatly reduced.

For a blocked matrix in the SBELL with 2×2 blocks and Nb blocked rows, the entries in each slice with a size of s, as shown in Figure 4, are arranged in global memory as Figure 5. When $Sb = 1$ and $Sb = Nb$, SBELL collapses into BCSR and BELLPACK, respectively. As the entries in each slice are stored in column-major, the threads assigned to these slices can access the entries in every block contiguously.

The SpMV kernel is given by Algorithm 4, where T threads are assigned to every slice of Sb blocked rows. Similar as the kernel in the SELL format, the array of **nnzb** denotes the beginning block index of the slice into **BA** and **BJA**, and $MnzPerRow[is]$ gives the maximum number of nonzero blocks per blocked row related to every slice. In this algorithm, every thread performs the dot-product between a blocked row and the vector **x** to compute a block of the vector **y** without synchronizations for no data dependencies in the computation of different blocks. Moreover, threads in each thread set access to consecutive memory address of **BA** and

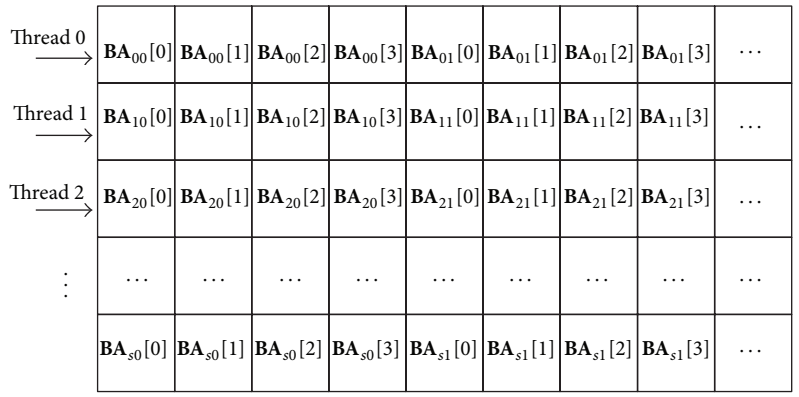

Figure 5: Entries of each slice in global memory.

```
spmv_sbell_kernel (int Nb, int Sb, int T, int *MnzbPerRow, int *nnzb,
                   int *BJA, float *BA, float *x, float *y)
{
  int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
  int sid = thread_id/T;
  int tid = thread_id − sid * T;
  int num_sets = (blockDim.x * gridDim.x)/T;
  int nslice = (Nb + Sb − 1)/Sb;
  for (int is = sid; is < nslice; is+ = num_sets) {
    for (ib = tid;  i < Sb;  ib+ = T) {
      int brow = is × Sb + ib;
      if (brow < Nb) {
        for (int kb = 0;  kb < MnzbPerRow[is]; kb++) {
          int bcol = BJA[nnz[is] + Sb × kb + ib];
          for (i = 0; i < bsize; i + +)
            for (j = 0; j < bsize; j + +)
              int col = bsize * bcol + j;
                  k = i * bsize + j;
              y[brow * bsize + i]+ = BA[nnzb[is] * bsize * bszie + jb * S * bszie * bsize
                                        +k * bsize + brow] × x[col]; }
        }
      }
    }
  }
}
```

Algorithm 4: SpMV kernel for the SBELL format.

BJA; thus the conditions of coalesced global memory access can also be reached.

4. Mixed Precision Polynomial Preconditioned Conjugate Gradient

The purpose of the mixed precision algorithm is to reduce the overall computational and storage requirements by introducing low precision arithmetic. For example, single precision number takes only half the storage of a double precision one. Thus, storage and bandwidth requirements are halved. For the computational demands, this is somewhat more hardware dependent. Most modern CPU architectures obtain twice the performance for single precision execution compared to double precision. On GPU architectures this relation might be more distinct. Taking the NVIDIA Tesla C10760 as an instance, the single precision peak processing power is up to twelve times faster than the double precision's [23]. However, scientific applications do not take advantage of these capabilities, as the low precision operations would mean an unacceptable loss of the result accuracy for many problems.

Actually, for many algorithms we need not perform high precision arithmetic for all intermediate computations to gain highly accurate final results. The knowledge about the error

$k = 0$
$\mathbf{d}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$ compute in high precision
$\mathbf{A}\mathbf{c}_k = \mathbf{d}_k$ solve in low precision
$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{c}_k$ correct in high precision
$k = k + 1$ iterate until convergence in high precision

Algorithm 5: Mixed precision strategy for linear equation solvers.

$\mathbf{A}^L = \mathbf{A}^H$
$\mathbf{r}_0^H = \mathbf{b}^H - \mathbf{A}^H\mathbf{x}_0^H$
$\mathbf{r}_0^L = \mathbf{r}_0^H$
$\mathbf{z}_0^L = P_m(\mathbf{A}^L)\mathbf{r}_0^L$
$\mathbf{z}_0^H = \mathbf{z}_0^L$
$\mathbf{p}_0^H = \mathbf{z}_0^H$
for $j = 0, 1, \ldots,$ until convergence
$\alpha_j^H = (\mathbf{r}_j^H, \mathbf{z}_j^H)/(\mathbf{A}^H\mathbf{p}_j^H, \mathbf{p}_j^H)$
$\mathbf{x}_{j+1}^H = \mathbf{x}_j^H + \alpha_j^H\mathbf{p}_j^H$
$\mathbf{r}_{j+1}^H = \mathbf{r}_j^H - \alpha_j\mathbf{A}^H\mathbf{p}_j^H$
$\mathbf{r}_{j+1}^L = \mathbf{r}_{j+1}^H$
$\mathbf{z}_{j+1}^L = P_m(\mathbf{A}^L)\mathbf{r}_{j+1}^L$
$\mathbf{z}_{j+1}^H = \mathbf{z}_{j+1}^L$
$\beta_j^H = (\mathbf{r}_{j+1}^H, \mathbf{z}_{j+1}^H)/(\mathbf{r}_j^H, \mathbf{z}_j^H)$
$\mathbf{p}_{j+1}^H = \mathbf{z}_{j+1}^H + \beta_j^H\mathbf{p}_j^H$
end for

Algorithm 6: Mixed precision polynomial PCG.

propagation in the algorithm can be used to confine the use of high precision computations to only a few relevant places [24]. This leads to a mixed precision method which mixes using low and high precision computations in different parts of the algorithms. The idea behind the mixed precision method is to exploit the large disparity between single and double precision peak performances on current GPUs and obtain a result of high accuracy.

The usual mixed precision strategy used with linear equation solvers on GPUs is defect correction, whose original form to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ is given by Algorithm 5. The core idea of this algorithm is to split the solution process into a computationally intensive but less precise inner iteration and a computationally simple but precise outer correction loop. To solve the defect equation, an arbitrary iterative solver running in low precision can be employed.

Another view on this scheme of defect correction is to interpret the low precision inner solver as a preconditioner in a high precision iterative method. Therefore, the polynomial preconditioned CG can be treated as the outer CG loop which is nested by an inner polynomial preconditioning. Furthermore, the polynomial preconditioning is a serial of matrix-vector products which is more computationally intensive than the outer CG loop with one matrix-vector product. Thus, we can greatly improve the performance of the polynomial PCG solution on GPUs if the mixed precision method is employed. The mixed precision polynomial PCG is described as Algorithm 6, where superscript ^{H} and ^{L} denote the high precision and low precision. In this algorithm, the inner preconditioning is performed in fast and cheap low precision, and the outer CG loop is in accurate high precision. The conversion between the two precision formats is implemented by duplicating the values into a new array with different precision. Compared to the fully high precision version, the distinct deficiency of the mixed algorithm is the extra memory needed to hold the low precision matrix and vectors.

5. Experimental Results

5.1. Test Platform and Examples. Our numerical experiments were conducted on a platform composed of an Intel Core2 Duo CPU E7400 @ 2.8 GHz and an NVIDIA Geforce GT430 GPU running 64-bit Windows 7 system. The NVIDIA Geforce GT430 card is of compute capability of 2.1 and has 2 multiprocessors, 96 cores, 1 GB of global memory, 64 KB of constant memory, and 48 KB of shared memory per block. The CUDA Toolkit and SDK 4.1 are used for programming.

Table 1 lists the test block SPD matrices from finite element discretization of elasticity problems, including their important characteristics, such as element type, number of unknowns, number of nonzero entries, and so forth. Although all of these matrices are symmetric, they all have been considered as general to compute SpMV. Additionally, Table 2 lists the factors of entries stored with different formats to CSR format, in which the size of slice is set to 32 for SELL and SBELL and the block size of BELL and SBELL is 2 and 3 for 2 dimensional and 3 dimensional problems, respectively. From Table 2, we can see that ELLPACK and BELL work well for structured mesh, for example, quadrilateral element and hexahedral element, but store too much padding zeros for unstructured meshes, for example, triangular element and tetrahedral element. However, SELL and SBELL suit general meshes despite mesh structures.

During the performance profiling, the running time is recorded in the best case among different configurations of threads. For the PCG iterative solve, the stopping criterion for convergence is the relative residual $\leq 1.0^{-7}$.

5.2. SpMV Kernels. A comparative analysis of the performance of different kernels to compute SpMV on NVIDIA GPUs has been carried out in this work. The following formats to store the matrix have been evaluated: CUSPARSE, CSR(vector), ELLPACK, SELL, BELL, and SBELL. We present results for double precision floating point arithmetic. All kernels have been evaluated using the texture memory. Here, the vector $\mathbf{x}$ has been stored, binding to the texture memory for all kernels evaluated, since in the computation of $\mathbf{y} = \mathbf{A}\mathbf{x}$ only the vector v is reused throughout the products with the different rows of the matrix.

Table 1: Examples used for testing.

Name	Element type	Unknowns	Nonzeros	Nonzeros/unknowns
tri	Plane stress 3-node triangular element	1,060,112	14,813,544	13.97
quad	Plane stress 4-node quadrilateral element	321,602	5,769,604	17.94
tet	4-Node tetrahedral element	624,579	27,482,265	44.00
hex	8-Node hexahedral element	499,125	38,976,723	78.09

Table 2: Factors of entries stored with different formats to CSR.

Name	ELLPACK/BELL	SELL	SBELL
tri	1.43	1.00	1.00
quad	1.00	1.00	1.00
tet	1.91	1.00	1.00
hex	1.04	1.00	1.00

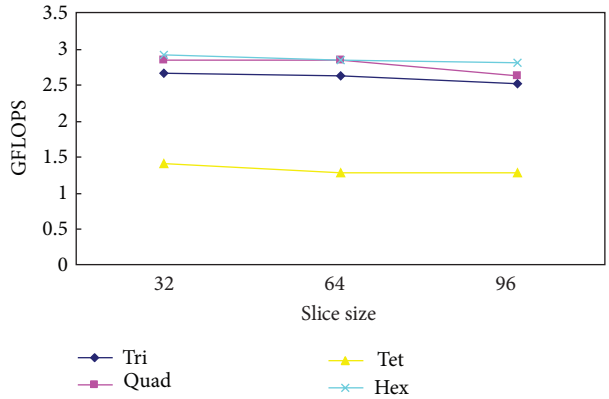

Figure 6: Performance results of SELL with different slice sizes.

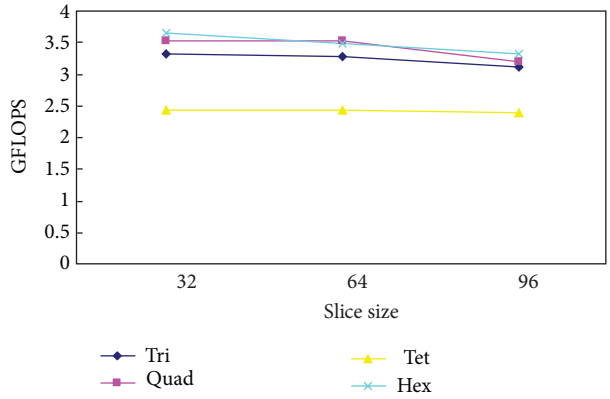

Figure 7: Performance results of SBELL with different slice sizes.

The reported figures represent an arithmetic average of 100 SpMV operations. The number of FLOPs for one sparse matrix-vector multiplication is precisely twice the number of nonzeros in the matrix. Therefore, the speed of floating operation, which is reported in units of GFLOPS, is simply the number of FLOPs of one single matrix-vector product divided by the average running time. The time of transferring data between host and device is not included in calculating

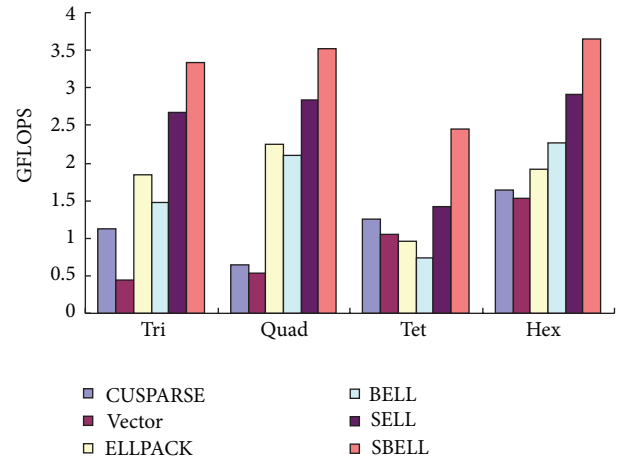

Figure 8: SpMV performance results.

the speed. In the context of iterative solvers, such data transfers can be negligible as they occur only twice: at the beginning and end of iteration, and thus can be amortized over a large number of SpMV operations.

Figures 6 and 7 show the different performance results of SELL and SBELL with different slice sizes. All the slice sizes are set into multiples of 32 to have coalescing data read. From these Figures, we can see the performance declines with the size increasing for triangular, quadrilateral, tetrahedral, and hexahedral meshes. We therefore fix the slice size to 32 in the remaining numerical tests.

Figure 8 reports the SpMV performance results of CUSPARSE, VECTOR, ELLPACK, BELL, SELL, and SBELL kernels. CUSPARSE denotes the kernel using CUDA CUSPARSE library in the storage of CSR and VECTOR is the vector SpMV kernel using 32-thread warp per matrix row for the CSR sparse matrix format. As expected, the SBELL kernel offers the best performance in all the cases. It reaches 3.33, 3.52, 2.44, and 3.64 GFLOPS on triangular, quadrilateral, tetrahedral, and hexahedral meshes, improvements of 24.72%, 23.78%, 72.22%, and 24.86% to SELL. This result is attributable to the decrease of column indices accesses

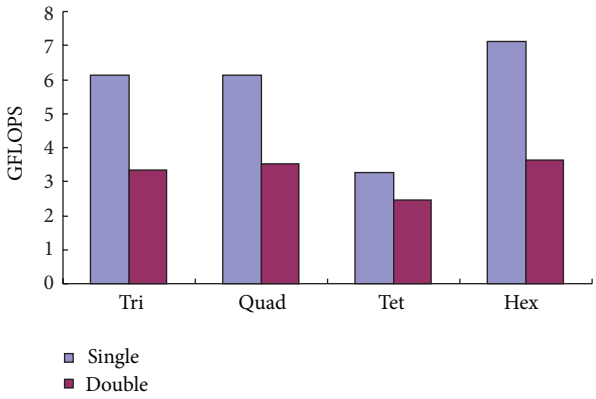

Figure 9: SpMV performance comparison between single and double precision results.

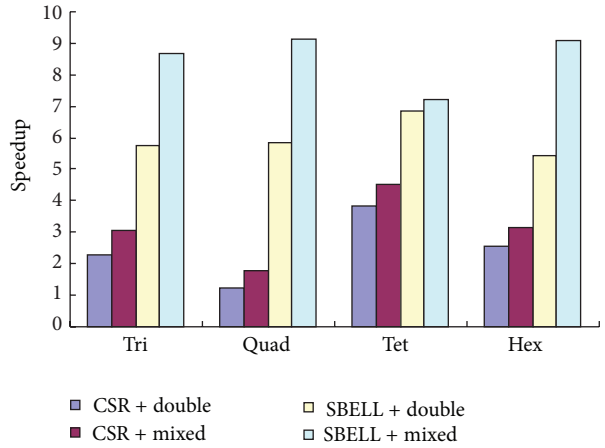

Figure 10: Speedup of GPU-based L-S preconditioned CG.

by block storage. From Figure 8, we also can see that all the ELLPACL-based kernels outpace CSR-based kernels in performance for their regular data structure of ELLPACK. Compared to CUSPARSE, SBELL gets speedups of 2.98, 5.52, 1.94, and 2.22 for all the meshes.

Figure 9 reports SpMV performance comparison between single precision and double precision arithmetic in the format of SBELL. As expected, the single kernels offer the better performance than double kernels in all cases. Double precision performance in the triangular element mesh example is 54.4% that of the single precision result. For the quadrilateral element mesh, double precision performance is 57.3% of the corresponding single precision result. For hexahedral element mesh, the percent is 51.1%. The tetrahedral element mesh retains 74.8% of its single precision performance. These results show that mixed precision algorithms can potentially achieve better performance than double precision versions.

5.3. Polynomial Preconditioners. The performance of the polynomial preconditioned CG algorithm for the solution of a variety of SPD linear systems from finite element approximation of elasticity problems have been compared and analyzed, with different polynomial preconditioners and different orders of polynomial preconditioners being used. The test matrices and their structural characteristics are presented in Table 1.

The least-squares polynomial preconditioner can produce a good approximation of $\mathbf{A}^{-1}$ assuming that a tight bound of smallest eigenvalue λ_n and the largest eigenvalue λ_1 can be found. As mentioned in [12], the upper bound must be larger than λ_1, but it should not be too large as this would otherwise result in slower convergence. This bound can be obtained inexpensively by using a small number of steps of the Lanczos method [25] and a safeguard term [26] added to guarantee that the upper bound is larger than λ_1. In addition, all systems undergo row and column scaling prior to computation of the solution; therefore, good estimates of the spectrums of the scaled systems are also required.

Table 3 lists the convergence iterates for quadrilateral mesh example "quad" by different polynomial methods with different orders of polynomials. Compared to the convergence iterates of 1938 for Jacobi preconditioner, all the methods improve the convergence. The L-S polynomial converges fastest and the Chebyshev shows oscillatory convergence. The number of convergence iterates may be reduced by increasing the value of the degree of the preconditioning polynomial. However, the number of the SpMV operations in each iteration increases accordingly. Table 4 lists the convergence time of different polynomial methods with different orders in the format of CSR on CPU. Compared to the convergence time of 64.92 seconds for Jacobi preconditioner, the running times of Neumann and Chebyshev greatly increase. The L-S has best convergence performance and its convergence time is almost identical to Jacobi with polynomial order set to 6 for all the four meshes listed in Table 5. Consequently, the L-S with order of 6 is chosen as a polynomial preconditioner in the following PCG tests.

5.4. PCG. In our implementation of the GPU-accelerated polynomial preconditioned CG method, the SpMV and level-1 BLAS operations are performed in parallel on the GPU. The polynomial 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.

Figure 10 shows the speedups of GPU-based L-S preconditioned CG to CPU-implementation on four example meshes. As shown, the double precision version in the form of CSR reaches a speedup of 1.2–3.8 and the mixed precision 1.8–4.5. For the SBELL format, the double precision gets a speedup of 5.5–6.9 and the mixed precision 7.2–9.1. This difference between double precision and mixed precision is significant here, and SBELL is more suitable than CSR for GPU computing in finite element computation of elasticity problems.

Table 3: Convergence iterates of different polynomial methods with different orders.

Polynomial	Order = 2	Order = 3	Order = 4	Order = 5	Order = 6
Neumann	1321	1142	1024	932	862
L-S	898	690	564	477	413
Chebyshev	1176	1370	1356	1391	1354

Table 4: Convergence time of different polynomial methods with different orders (unit: sec).

Polynomial	Order = 2	Order = 3	Order = 4	Order = 5	Order = 6
Neumann	104.81	116.17	129.75	137.19	145.42
L-S	68.75	67.28	66.31	65.56	65.33
Chebyshev	103.75	158.44	193.89	243.17	273.80

Table 5: Convergence time of different polynomial methods on different meshes (unit: sec).

Mesh	Jacobi	L-S (order = 6)
tri	666.88	650.86
quad	64.92	65.33
tet	243.11	327.22
hex	43.98	63.38

6. Conclusions

We have introduced sliced block ELLPACK (SBELL) format to store sparse matrix arising from finite element discretization of elasticity. Based on this sparse representation, a CUDA SpMV kernel on GPU has been implemented. Compared with CUSPARSE library, vector, ELLPACK, BELL, and SELL kernels, the SBELL SpMV kernel gets the best performance. To accelerate convergence of the conjugate gradient iterative method to solve the finite element equations on GPU, polynomial preconditioning methods have been investigated. From the numerical tests on different meshes, polynomial methods are always feasible and generally are of good convergence performance. In addition, the number of convergence iterates may be reduced by increasing the value of the degree of the preconditioning polynomial. Considering the convergence time increasing with the SpMV operations multiplying for high order polynomials, the L-S polynomial method shows the best performance and has been chosen as a preconditioner in PCG solver to finite element equations derived from elasticity. In the PCG solver, mixed precision algorithm is used by introducing single precision arithmetic in computationally intensive preconditioning inner loop and double precision in outer correction loop. This mixed precision implementation not only reduces the overall computational and storage requirements but makes full use of the capacity of the GPU devices. With SBELL format and mixed precision implementation, the GPU-based L-S preconditioned CG can reach a speedup of over 7 to CPU-implementation for different meshes.

In future work, optimizations will be investigated in practical problems. Furthermore, we plan to study the hardware-adaptive algorithms to automatically capture optimal computational performance on GPUs with different configurations.

Conflict of Interests

The authors do not have any conflict of interests with the content of the paper.

Acknowledgments

This work is supported by National Natural Science Foundation of China (no. 51109072).

References

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[2] J. D. Owens, D. Luebke, N. Govindaraju et al., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] E. Elsen, P. LeGresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, vol. 227, no. 24, pp. 10148–10161, 2008.

[4] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[5] H. Courtecuisse, H. Jung, J. Allard, C. Duriez, D. Y. Lee, and S. Cotin, "GPU-based real-time soft tissue deformation with cutting and haptic feedback," *Progress in Biophysics and Molecular Biology*, vol. 103, no. 2-3, pp. 159–168, 2010.

[6] W. Wu and P. A. Heng, "A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting," *Computer Animation and Virtual Worlds*, vol. 15, no. 3-4, pp. 219–227, 2004.

[7] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, no. 5, pp. 451–460, 2009.

[8] M. A. Heroux, P. Vu, and C. Yang, "A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP," *Applied Numerical Mathematics*, vol. 8, no. 2, pp. 93–115, 1991.

[9] A. R. M. Rao, "MPI-based parallel finite element approaches for implicit nonlinear dynamic analysis employing sparse PCG solvers," *Advances in Engineering Software*, vol. 36, no. 3, pp. 181–198, 2005.

[10] Y. Liu, W. Zhou, and Q. Yang, "A distributed memory parallel element-by-element scheme based on Jacobi-conditioned conjugate gradient for 3D finite element analysis," *Finite Elements in Analysis and Design*, vol. 43, no. 6-7, pp. 494–503, 2007.

[11] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *Journal of Computational and Applied Mathematics*, vol. 236, no. 15, pp. 3584–3590, 2012.

[12] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," Tech. Rep. umsi-2010-112, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, Minn, USA, 2010.

[13] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, 2nd edition, 2003.

[14] Y. Liang, R. Kanapady, and K. Tamma, "An efficient parallel finite-element-based domain decomposition iterative technique with polynomial preconditioning," Tech. Rep. TR 05-001, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minn, USA, 2005.

[15] D. R. Kincaid, T. C. Oppe, and D. M. Young, "ITPACKV 2D User's Guide," CNA-232, 1989, http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[16] NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2012, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[17] NVIDIA CUDA C Programming Guide, Version 4.2, 2012, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[18] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The Finite Element Method: Its Basis and Fundamentals*, Butterworth-Heinemann, Oxford, UK, 6th edition, 2005.

[19] F. Vázquez, J. J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[20] A. Monakov, A. Lokhmotov, and A. Avetisyan:, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and Compilers*, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., Lecture Notes in Computer Science, pp. 111–125, Springer, 2010.

[21] V. Karakasis, G. Goumas, and N. Koziris, "Exploring the effect of block shapes on the performance of sparse kernels," in *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS '09)*, pp. 1–8, IEEE Computer Society, Washington, DC, USA, May 2009.

[22] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, pp. 115–125, ACM, January 2010.

[23] http://en.wikipedia.org/wiki/Nvidia_Tesla.

[24] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, 2007.

[25] 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*, vol. 45, pp. 255–282, 1950.

[26] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, "Parallel self-consistent-field calculations via Chebyshev-filtered subspace acceleration," *Physical Review E*, vol. 74, no. 6, part 2, Article ID 066704, 2006.