An Improved Implementation of Preconditioned Conjugate Gradient Method on GPU

Yechen Gui
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
Email: liaoran919@yahoo.com.cn

Guijuan Zhang
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
Email: gj.zhang@siat.ac.cn

Abstract—An improved implementation of the Preconditioned Conjugate Gradient method on GPU using CUDA (Compute Unified Device Architecture) is proposed. It aims to solving the Poisson equation arising in liquid animation with high efficiency. We consider the features of the linear system obtained from the Poisson equation and propose an optimization method to solve it. First, a novel storage format called mDIA (modified diagonal storage format) is presented to improve the efficiency of the Sparse Matrix-Vector product (SpMV) operation. Second, a parallel Jacobi iterative method is proposed when using the Incomplete Cholesky preconditioner to explore inherent parallelism. Third, CUDA streams are also introduced to overlap computations among separate streams. The proposed optimization technique is embedded into our GPU based PCG algorithm. Results on Geforce G100 show that our SpMV kernel yields an improvement of nearly 100% for large sparse matrix with more than 30,000 rows. Also, a speedup of more than 7 is obtained for PCG method, making the real-time physics engine possible.

Index Terms—CUDA, PCG, Incomplete Cholesky preconditioner, SpMV, Poisson equation

I. INTRODUCTION

In the past few years, Graphics Processing Unit (GPU) has evolved into a unified powerful many-core processor. The modern GPUs are well suited for compute-intensive tasks and massively parallel computation (e.g., solving matrix problems [1][2]). As one of the most common and important matrix problems, solving the large-scale linear system can be significantly accelerated if corresponding algorithms can be mapped well to the structure of the GPU and be accord with SIMD (Single Instruction, Multiple Data) pattern.

In this paper, we focus on the problem of solving the Poisson equation. The equation arises in many applications such as computational fluid dynamics, electrostatics, magnetostatics, etc. Numerical solution of the Poisson equation leads to a large sparse linear system. It is usually solved by iterative methods such as the best-known conjugate gradient (CG) method instead of direct methods (e.g., Gaussian elimination). The CG method can be easily implemented to solve linear systems that have a symmetric, definite positive (SPD) matrix [3]. However, it is often used with a suitable preconditioner in order to achieve high convergence rates in large scale applications. A CG algorithm with a preconditioner is called preconditioned conjugate gradient algorithm (PCG) and it has been proven to be efficient and robust in a wide range of applications [4].

Our goal is to solve Poisson equation efficiently by applying PCG algorithm on the Nvidia GPU architecture using CUDA [5]. Since the SpMV routine is the bottleneck of PCG algorithm that consumes nearly 80% of the total time, we present a novel storage format called mDIA storage format to optimize it. Moreover, we parallelize the traditional Jacobi iterative method to solve the lower Cholesky triangular equation when using the Incomplete Cholesky (IC) preconditioner. In addition, to effectively overlap the computation, CUDA streams are also adopted in this paper. Results show that our method obtains a speedup of 7 for PCG algorithm on Geforce G100.

The paper is organized as follows. The next section introduces the background of our method. The related work on GPU based PCG methods are reviewed first, and then we give a brief introduction of our linear system generated from Poisson equation. GPU architecture and our optimization algorithm based on GPU are presented in Section 3. In this section, the optimization techniques are discussed in detail. Section 4 shows experimental results followed by conclusions in Section 5.

II. BACKGROUND

2.1. Related Work

Jeff Bolz et.al [6] was the first to implement CG method on GPU using shader language and the speedup was about 1.5x. He also showed the feasibility of using the Compressed Row Storage Format for SpMV routine. After the advent of NVIDIA CUDA, GPU based iterative methods have been widely used to solve the sparse linear systems [7][8][9]. For example, Georgescu et.al [7] discussed how CG method could be aligned to the GPU architecture. They also discussed the problem with precision and applied different preconditioners to
accelerate convergence. In particular, they stated that for double precision calculations, problem having condition number less than 10 may converge and give a speedup also. In 2009, Buatois et al. [10] introduced their framework CNC for solving general sparse linear systems on GPU with a Jacobi-preconditioned CG method. Their method achieved a speedup of 3.2. However, they warned that GPU is only able to provide comparable accuracy because as the iterations increase, the precision drops in comparison to CPU. They also exploited some of the techniques like register blocking to optimize their algorithm. In [11], the CG method with Incomplete Poisson preconditoning was implemented on a multi-GPU platform. It mainly focused on overlapping communication between different GPUs by interactively exchanging boundary stream and inner stream. Their results showed that the performance can grow proportionally to the problem size and showed a good scalability. In work published by A. Asgarsi [9], the author parallelized a Chebyshev polynomial preconditioner to improve the performance of PCG method based on GPU.

As for the CG algorithm, nearly 80% of the total time is consumed by SpMV routine. It yields only a small fraction of the machine peak performance due to its indirect and irregular memory access. Therefore, there exists a large amount of work focusing on speeding up SpMV routine. Typical methods often use CSR (Compressed sparse row) format, COO (the coordinate) format and the DIA (diagonal) storage formats to mitigate the irregularity [12]. In a recent study by Nathan Bell [13], a hybrid method that used the modified ELL-COO format to store the sparse matrix delivered high throughput. However, it relied on an additional sweep operation to find out the number of nonzero elements in the matrix.

2.2. The Linear System Derived from the Poisson Equation

The Poisson equation in liquid animation is a second-order PDE as shown in equation (1). It is used to compute the pressure \( p \).

\[
\nabla^2 p = \frac{\rho}{\Delta t} \tag{1}
\]

Note that in the Poisson equation (1), \( \rho \) is the pressure, \( \rho \) is the density and \( \Delta t \) is time step. It can be further transformed into equation (2) according to finite difference method.

\[
\begin{aligned}
\Delta p_{i,j,k} &= \Delta^2 p_{i,j,k} - \Delta p_{i+1,j,k} - \Delta p_{i-1,j,k} + 6 \Delta p_{i,j,k+1} - \Delta p_{i,j,k-1} - \Delta p_{i,j,k+1} \\
&= \Delta^2 p_{i,j,k} - \Delta p_{i+1,j,k} - \Delta p_{i-1,j,k} + \Delta p_{i,j,k+1} - \Delta p_{i,j,k-1} - \Delta p_{i,j,k+1}
\end{aligned} \tag{2}
\]

In equation (2), \( \Delta x \) is space interval, \( u = (u,v,w) \) is the velocity field. Let

\[
b = -\Delta^2 \left( \frac{\Delta x}{\Delta x} \left( u_{i,j,k} - u_{i,j,k+1} + v_{i,j,k} - v_{i,j,k+1} + w_{i,j,k} - w_{i,j,k+1} \right) \right) \tag{3}
\]

Equation (2) can be converted into \( Ap = b \), and our goal is to solve the unknown vector \( p \).

It can be proved that \( A \) is a sparse, positive-definite and symmetry matrix. In addition, we also explore some other features of matrix \( A \) in order to design more efficient algorithms. See the left side of equation (2), each row of \( A \) has no more than 7 nonzero elements. In this row, the diagonal element is a nonzero integer while the other nonzero elements equal to -1. Other rows also have similar structures. All these features will be considered in our new algorithm. Details will be given in the following sections.

2.3. The PCG Algorithm

Consider

\[
Ax = b, \quad \tag{4}
\]

where \( x \) is an unknown vector, \( b \) is a known vector, \( A \) is a known SPD matrix. According to PCG algorithm, equation (4) can be written as

\[
M^{-1}Ax = M^{-1}b, \quad \tag{5}
\]

where matrix \( M \) is a preconditioner [4].

Given the inputs \( A \), \( b \), a starting vector \( x \), a preconditioner \( M \), a maximum number of iterations \( k_{\text{max}} \) and a error tolerance \( \text{err} \), the PCG algorithm can be described in fig. 1. In this figure, a set of \( a_0, a_1, a_2 \ldots a_n \) are constructed by the conjugation of the orthogonal search directions \( 0, 1, 2, \ldots n \) respectively. Then in the \( k \) th iteration step, \( x_k \) takes exactly one step of the length \( h_k \) along the direction \( a_k \). If the convergence conditions \( \text{err} < \varepsilon \) and \( k < \text{max}_k \) are met, the iterative process is terminated. Note that in our method, we set \( x_0 = 0 \).

To compute \( M^{-1}r_k \) at each iteration step, we choose IC preconditioner in our method to improve the convergence rate [4]. An IC preconditioner can be obtained by factoring a matrix \( A \) into the form \( LL^T \) where \( L \) is a lower Cholesky triangular matrix. \( L \) is restricted to have the same pattern of nonzero elements as \( A \) and other elements of \( L \) are thrown away. Therefore, \( M \) equals \( LL^T \) and \( z_k \leftarrow M^{-1}r_k \) can be converted into \( (LL^T)z_k = r_k \). As a result, \( z_k \) can be directly computed by forward and then backward substitutions.

III. IMPLEMENTATION ON GPU

3.1 GPU Architecture and CUDA

© 2012 ACADEMY PUBLISHER
The new generation of GPU adopts the unified shader architecture CUDA and promise up to 900 Gflops (single precision) of computational power.

A GPU can be seen as a SIMD processor. What this means is that there are an army of processors executing the same instructions in parallel independently. Take fig. 2 for example, a GPU has a scalable array of multithreaded Streaming Multiprocessors (SPs). Each multiprocessor creates, manages, schedules, and executes groups of 32 parallel threads which are called warps. Individual threads composing a warp start together at the same program address, but have their own instruction address counter and register state. Therefore they are free to branch and execute independently.

On GPU chip, each multiprocessor has a set of memories associated with it. They are: on-chip shared memory, global memory, read-only constant cache, and read-only texture cache. Among these memories, global memory is the biggest in size but with highest access latency. On the other hand, Shared memory, constant cache as well as the texture cache resides on chip and can be accessed more efficiently. Note that shared memory is only visible to one block and threads of other block cannot access the data stored in it.

The scalable characteristic of modern GPU provides coarse-grained and the fine-grained data parallelism. They guide the programmer to partition the problem into sub-problems that can be solved independently in parallel by blocks of threads. Moreover, each sub-problem can be divided into finer pieces that can be solved cooperatively in parallel by all threads within the block. Fig. 3 gives an example. In this figure, kernels are launched on GPU device and executed by multiple equally-shaped thread blocks.

3.2 Overview

Algorithm 1 shows our framework for PCG implementation on GPU. In this framework, we use two kernels to complete the computation involved in the for loop. They are the SpMV kernel that computes matrix-vector multiplication such as $A h_{k-1}$, the preconditioning kernel that computes $M^{-1} r_k$.

Besides the above kernels, other operations such as dot products among vectors can be done efficiently by cublas library [14] because they all belong to level-1 BLAS (Basic Linear Algebra Subprograms) functions.

$$
\begin{align*}
  r_0 & \leftarrow b - Ax_0 \\
  z_0 & \leftarrow M^{-1} r_0 \\
  h_0 & \leftarrow z_0 \\
  err_{old} & \leftarrow< r_0, z_0 > \\
  err_{new} & \leftarrow err_{old} \\
  While \ (err_{new} < \epsilon || k < k_{\text{max}}) do \\
  & \quad \alpha_k \leftarrow \frac{err_{new}}{< h_{k-1}, Ah_{k-1} >} \\
  & \quad x_k \leftarrow x_{k-1} + \alpha h_{k-1} \\
  & \quad r_k \leftarrow r_{k-1} - \alpha Ah_{k-1} \\
  & \quad z_k \leftarrow M^{-1} r_k \\
  & \quad err_{old} \leftarrow err_{new} \\
  & \quad err_{new} \leftarrow< r_k, z_k > \\
  & \quad \beta \leftarrow \frac{err_{new}}{err_{old}} \\
  & \quad h_k \leftarrow z_k + \beta h_{k-1}
\end{align*}
$$

Figure 1. PCG algorithm.

![Figure 2. GPU architecture.](image2)

![Figure 3. Serial code executes on the host while parallel code executes on the device.](image3)
In order to improve the efficiency of our GPU based PCG algorithm, we focus on two most expensive kernels here: the SpMV kernel and the preconditioning kernel. In addition, to get a higher level of concurrency, we use a technique called streams [15] to run independent kernels asynchronously so as to overlap the computation.

Algorithm 1. Algorithm for our PCG method on GPU

```csharp
// Use cublas() to compute dot products
// and then obtain err
r_k ← b - Ax_k
z_k ← M⁻¹r_k
h_k ← z_k,
errzo ← < r_k, z_k >
errvec ← errzo
while (errzo < e & & k < k_max) do
    // SpMV kernel to calculate A h_k
    kernel_spmv()
    // Use cublasdot()
    α ← errvec / < h_k, A h_k >
    // Use cublasSaspy()
    x_k ← x_k + α h_k
    r_k ← r_k - α A h_k
    // Preconditioning kernel
    z_k ← M⁻¹r_k
    // Device assignment
    errzo ← errvec
    // Use cublasdot()
    errvec ← < r_k, z_k >
    // Use cublasSaspy()
    β ← errvec / errzo
    h_k ← z_k + β h_k
    k++
end while
```

### 3.3 SpMV Kernel

Sparse Matrix-vector multiplication (SpMV) is one of the most fundamental and important operations in sparse matrix computations. It is the dominant cost in many iterative methods for solving large-scale linear systems. Recently, several research groups have reported their implementation on CUDA-compatible GPUs and show that the storage pattern such as CSR, ELL, HYBRID formats can be efficiently accessed by CUDA threads. However, except for the CSR format, all other formats have to fill in zeros to keep the array strictly aligned, thus causing memory waste.

Since the SpMV kernel with CSR format provides important insight for understanding our algorithm, we discuss the implementation of SpMV GPU kernels with CSR and mDIA respectively in the following subsections.

#### 3.3.1 SpMV with CSR format

CSR format is one of the most popular sparse matrix representations. In this format, an N-by-N sparse matrix with K nonzero elements is stored as two arrays: one array `val` holds the K nonzero elements and the other array `col` holds the column indexes of these nonzero elements. What’s more, an additional array `Rowptr` with the length N+1 is used. The first N components of `Rowptr` record the indexes of the first element in each row while the last one denotes the number of nonzero elements in the matrix. Fig. 4 gives an example. Unlike the ELL or DIA, CSR format doesn’t waste any memory space.

```plaintext
Matrix =
[1 3 0 0 0]
[9 7 0 8 2]
[0 6 1 0 0]
[1 0 3 9 0]
[0 0 0 0 2]

val = [1 3 9 7 8 2 6 1 1 3 9 2]

col = [0 1 0 1 3 4 1 2 0 2 3 2]

Rowptr = [0 2 6 8 11 12]
```

Figure 4. CSR format for sparse Matrix.

To parallelize the SpMV operation with CSR format, a scheme called scalar CSR kernel [13] is used. In this kernel, one thread is used to fetch one row of the matrix `A` and then complete the dot product for one component of the result vector. The computations of all threads are independent. The data parallelism as well as the access pattern of scalar CSR kernel is shown in Fig. 5. It gives a simplified example of the allocation of the threads, in which the array data, Col Index, and Rowptr are stored in global memory for the dot product operation.

```plaintext
Row1
Thread1
Data

Row2
Thread2
Col Index

Row3
Thread3

Row4
Thread4

Rowptr

Global Memory
```

Figure 5. Scalar SpMV Kernel with CSR format.

#### 3.3.2 SpMV with mDIA format

(1) Our mDIA storage format

Our matrix has a regular pattern that the elements off the main diagonal are all assigned to a constant integer -1 while the nonzero diagonal elements are also integers. The number of nonzero elements per row varies from 3 to 7 and this irregularity makes ELL or hybrid pattern infeasible, since they will cause large zero fill-ins.

In mDIA format, the constant value is stored in the constant memory. As a result, 1D array `Diag` only needs to store the diagonal elements. Because all diagonal elements in our matrix are nonzero values, the row and
column indexes of them can be easily obtained from $\text{Diag}$. Array $\text{col}$ only needs to record column indexes of the constant value. Besides, Array $\text{Rowptr}$ is used to tell where a new row begins in the array $\text{col}$, similar to array $\text{Rowptr}$ in CSR format. Fig. 6 shows a portion of one matrix and its corresponding storage mechanism. Note that the constant -1 is stored in the constant memory.

Since most of the nonzero elements are off the main diagonal in our matrix and they are stored in the constant memory, the memory usage can be significantly reduced compared with the CSR format. Also, the constant memory, a small high-speed cache residing in the global memory on GPU, enables us to fetch data efficiently [15].

(2) The SpMV kernel

To implement the SpMV kernel with mDIA format on GPU, we assign one thread to compute one component of the result vector. Take thread $i$ for example. It computes the dot product between the $i$-th row of our matrix and the vector. First it fetches the diagonal nonzero element from $\text{diag}[i]$. Then the column indexes of the constant elements from $\text{col}[\text{Rowptr}[i]]$ to $\text{col}[\text{Rowptr}[i+1]]$ are read contiguously. Finally, the dot products operation is executed.

Considering that the vector is reused in the computation of the dot product, we bind it to a 1D texture. This can bring potentially higher bandwidth and can be used to avoid uncoalesced loads from global memory [15]. Other array such as $\text{col}$ are stored in the global memory. Fig. 7 presents the pseudo-code of our implementation.

3.4 Preconditioning Kernel using Jacobi Method

Another important kernel in our PCG algorithm is the preconditioning kernel. According to IC preconditioner, $z_k \leftarrow M^{-1}r_k$ is converted into $(LL^T)z_k = r_k$. Where $L$ is the lower Cholesky triangular matrix. Thus, $z_k$ can be obtained by solving $L(L^Tz_k) = r_k$ in two stages: forward substitution $L^Ty = r_k$ and backward substitution $L^Tz_k = y$.

However, the direct method of backward and forward substitution cannot be used to solve all the components simultaneously on GPU because the computation of the $i$th component of $z_k$ relies on all its previous components.

To get a higher level of parallelism, we use Jacobi method instead of direct method in this paper. Jacobi iterative method is data independent that can be well aligned to SIMD pattern and improves parallelism significantly.

Pseudo-code of our SpMV kernel

```
 Constant__val = -1;
 __global__ void SpMV(float *diag, float *col, float *Rowptr, float *x, )
 {
    int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
    int grid_size = blockDim.x * gridDim.x;
    for(int row = thread_id; row < num_rows; row += grid_size) {
        const int row_beg = Rowptr[row];
        const int row_end = Rowptr[row+1];
        float sum = 0;
        for (int jj = row_beg; jj < row_end; jj++) {
            sum += val*fetch_x(Col[jj], x, UsedTex);
            sum += diag[jj] * fetch_x(row, x, UsedTex);
        }
        y[row] = sum;
    }
 }
```

3.4.1 Jacobi iterative method

Jacobi iterative method is a numerical solution of a system of linear equations with largest absolute values in each row and column dominated by the diagonal element. To solve our lower Cholesky triangular equation $Ly = r_k$, we first decompose the lower Cholesky triangular matrix $L$ into a diagonal matrix $D$ and a lower triangle matrix $R$. Then the system of linear equations becomes

$$(D + R)y = r_k, \quad (6)$$

and finally

$$Dy = r_k - Ry. \quad (7)$$

Therefore, $y$ can be solved iteratively by

$$y_{k+1} = D^{-1}(r_k - Ry_k). \quad (8)$$

Next, $z_k$ can be computed by solving upper Cholesky triangular equation $L^Tz_k = y$ in the same way.

3.4.2 Parallel Jacobi algorithm

Fig. 8 (a) shows the iterative process of our parallel Jacobi algorithm. In this figure, $d_{old}$ stores the result from the previous step and $d_{new}$ is used to update the current computation. The constant vector $d_{const}$ $= D^{-1}r_k$, where the diagonal matrix $D$ is stored in the array $d_{diag}$, $d_{R}$ stores the lower triangular matrix $R$ and $d_{res}$ is used to compute the residue. Note that most
of the operations such as $D^{-1}b$ can be converted to level-1 BLAS operation among the vectors. These operations could be easily parallelized using CUDA. Fig. 8 (b) shows the kernel named VecDiv_kernel for computing $D^{-1}b$ and the kernel named VectorSub_kernel for computing the subtraction between two vectors.

3.5 Parallelism with Streams

CUDA applications can manage concurrency through streams. Different streams, on the other hand, may execute their commands out of order concurrently. Fig. 7 illustrates the GPU time flow for sequential (Fig. 9 (a)) and concurrent (Fig. 9 (b)) kernel executions.

4.1 SpMV Kernel Test

Table 2 shows the performance of our SpMV kernel against the SpMV kernel with CSR format. In this table, GPU Time accounts for the total time consumed for the computation of the two tasks can be overlapped and the GPU resources can be used more effectively.

<table>
<thead>
<tr>
<th>MATRIX</th>
<th>#N</th>
<th>#Nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix_1</td>
<td>8087</td>
<td>49915</td>
</tr>
<tr>
<td>Matrix_2</td>
<td>22028</td>
<td>131,118</td>
</tr>
<tr>
<td>Matrix_3</td>
<td>65043</td>
<td>394,877</td>
</tr>
<tr>
<td>Matrix_4</td>
<td>140,120</td>
<td>876,518</td>
</tr>
<tr>
<td>Matrix_5</td>
<td>209,908</td>
<td>1,325,066</td>
</tr>
<tr>
<td>Matrix_6</td>
<td>274,207</td>
<td>1,962,311</td>
</tr>
<tr>
<td>Matrix_7</td>
<td>304,207</td>
<td>1,962,311</td>
</tr>
</tbody>
</table>

Figure 10. The operations using CUDA streams.

As a result, the computation of the two tasks can be overlapped and the GPU resources can be used more effectively.

IV. RESULTS

We use GeForce G100 to test the performance of our parallel PCG algorithm. GeForce G100 has 8 CUDA cores and the peak performance for single precision is $10.4$ Gflops. The CPU used is AMD 7750 dual-core processors with the core frequency of 2.7GHz. The CPU implementation of PCG is single-threaded.

Table 1 shows some matrices that generated from our Poisson equation. They will be used in our performance test. In this section, we start from testing the performance of our SpMV kernel and then test Jacobi iterative method for solving the lower Cholesky triangular equations, followed by CUDA stream results.

### 4.1.1 SpMV kernels performance

Table 2 shows the performance of our SpMV kernel against the SpMV kernel with CSR format. In this table, **GPU Time** accounts for the total time consumed for the
SpMV kernel in our PCG algorithm and the speedup is the ratio compared with CPU Time. On CPU, the SpMV routine is implemented using CSR format. According to the results, our SpMV kernel runs an average of one time faster than that in CSR format and it offers an average of about 10 times speedup compared with the CPU version.

<table>
<thead>
<tr>
<th>matrix</th>
<th>Timings for spMV CPU routine</th>
<th>Timings for spMV GPU kernel</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSR</td>
<td>mDIA</td>
<td>CSR</td>
</tr>
<tr>
<td>Matrix_1</td>
<td>0.48</td>
<td>0.09</td>
<td>0.05</td>
</tr>
<tr>
<td>Matrix_2</td>
<td>1.34</td>
<td>0.25</td>
<td>0.14</td>
</tr>
<tr>
<td>Matrix_3</td>
<td>5.50</td>
<td>1.00</td>
<td>0.57</td>
</tr>
<tr>
<td>Matrix_4</td>
<td>15.15</td>
<td>2.63</td>
<td>1.52</td>
</tr>
<tr>
<td>Matrix_5</td>
<td>29.67</td>
<td>5.34</td>
<td>2.99</td>
</tr>
<tr>
<td>Matrix_6</td>
<td>44.82</td>
<td>8.03</td>
<td>4.50</td>
</tr>
<tr>
<td>Matrix_7</td>
<td>54.69</td>
<td>9.50</td>
<td>5.41</td>
</tr>
</tbody>
</table>

4.1.2 The performance of CG&PCG algorithm with our SpMV kernel

We embed our SpMV kernel into CG&PCG method. On GPU, the IC preconditioner in PCG method is solved by direct method. Further improvement using Jacobi iterative method will be presented in the next subsection.

We compare our results with CG&PCG methods using CSR format as shown in Table 3. It illustrates that due to our SpMV kernels, the CG algorithm outperforms by nearly 50% when the number of nonzero elements reached 1,962,311.

However, when PCG method is applied, this table shows that our advantage over the CSR-based method has been less obvious; and when the dimension of matrix has reached up to 304,207, the performance improvement is only about 10%. This is due to the sequential nature of the direct method as mentioned before.

4.2 Jacobi Method for IC Preconditioner

In this section, we adopt two variants: the direct method and the Jacobi iterative method to explore GPU performance for PCG algorithm.

<table>
<thead>
<tr>
<th>matrix</th>
<th>Timings for CG steps</th>
<th>Timings for future CG and PCG steps</th>
<th>Timings for GPU based method(Seconds)</th>
<th>Timings for PCG based method(Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CG steps</td>
<td>CG steps</td>
<td>GPU method</td>
<td>PCG method</td>
</tr>
<tr>
<td></td>
<td>mDIA</td>
<td>CSR</td>
<td>mDIA</td>
<td>CSR</td>
</tr>
<tr>
<td>Matrix_1</td>
<td>84</td>
<td>0.64</td>
<td>0.68</td>
<td>17</td>
</tr>
<tr>
<td>Matrix_2</td>
<td>90</td>
<td>0.76</td>
<td>0.86</td>
<td>21</td>
</tr>
<tr>
<td>Matrix_3</td>
<td>123</td>
<td>1.40</td>
<td>1.80</td>
<td>23</td>
</tr>
<tr>
<td>Matrix_4</td>
<td>146</td>
<td>2.42</td>
<td>3.69</td>
<td>25</td>
</tr>
<tr>
<td>Matrix_5</td>
<td>192</td>
<td>4.44</td>
<td>6.70</td>
<td>30</td>
</tr>
<tr>
<td>Matrix_6</td>
<td>221</td>
<td>6.41</td>
<td>9.87</td>
<td>34</td>
</tr>
<tr>
<td>Matrix_7</td>
<td>234</td>
<td>7.25</td>
<td>11.53</td>
<td>35</td>
</tr>
</tbody>
</table>

When solving the lower Cholesky triangular equation, Timings for the parallel Jacobi iterative method and the direct method are both accumulated at every iterative step and their final costs are showed in Table 4. From this table, it can be noticed that GPU performance of the Jacobi iterative method has been efficiently improved about 3 times.

4.3 The Speedup of Our GPU based PCG Algorithm

Fig. 11 shows the speedup of our parallel PCG algorithm after using our SpMV kernel, Jacobi iterative method, as well as CUDA streams. It can be seen from the figure that there has been at least 16% increase for our PCG algorithm compared with CSR formats. Furthermore, our PCG algorithm with three optimization techniques proposed obtains an average of 6 times speedup, while the CSR method only offers an average of 4.

V. CONCLUSIONS

In this work, we propose an optimization method for GPU based PCG algorithm. It is designed to solve the Poisson equation arising in liquid animation efficiently. By utilizing optimized SpMV kernel, iterative Jacobi method and CUDA streams, our method improves the efficiency of solving large sparse linear systems significantly. Experimental results also show the effectiveness of our method.

Next we will focus on seeking out the other potential bottleneck of PCG algorithm by CUDA Visual Profiler. Besides, studies on finding more suitable preconditioners for GPU-based PCG algorithm should be taken.

ACKNOWLEDGMENT

The authors wish to thank Prof. Shengzhong Feng. This work was supported in part by the National High-Tech Research and Development Plan of China (863 program) under Grant No.2009AA10A129-2, the Science
and Technology Project of Guangdong. Province under the grant No.2010A090100028, National Natural Science, Foundation of P. R. China under the Grant No.60903116, and Knowledge, Innovation Project of the Chinese Academy of Sciences, No.KGCX2-YW-131, the Science and Technology Project of Shenzhen, under the grant No. JC200903170443A, ZD201006100023A.

VI. REFERENCES


[14] Nvidia, “CUDA Toolkit 4.0 CUBLAS Library” NVIDIA Corporation, Santa Clara, April, 2011


Born in Jiangxi province in China, Yechen Gui is a graduate of Xidian University, where she earned her bachelor degree in 2006, majoring in biomedical engineering. After that, she went to Southern Medical University for her postgraduate studies. During that time, she became fairly interested in parallel algorithms in medical image processing. She implemented parallel ray-casting algorithm as well as CT reconstruction algorithms on GPU using CUDA and published two articles in two different core journals.

After Yechen gained her master degree in 2009, she worked as a research assistant in Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences until now. During this time, she published 4 articles in parallel algorithms on GPU in all, one of which is accepted in the conference of 2010 GPU Solutions to Multi-scale Problems in Science and Engineering (GPU-SMP2010), others of which are all indexed by EI. Now she’s research field is towards computational fluid animations as well as parallel algorithms.