

Exploiting Structure of Symmetric or Triangular Matrices on a GPU

Jin Hyuk Jung* Dianne P. O'Leary†

January 2008

Abstract

Matrix computations are expensive, and GPUs have the potential to deliver results at reduced cost by exploiting parallel computation. We focus on dense matrices of the form $\mathbf{AD}^2\mathbf{A}^T$, where \mathbf{A} is an $m \times n$ matrix ($m \leq n$) and \mathbf{D} is an $n \times n$ diagonal matrix. Many important numerical problems require solving linear systems of equations involving matrices of this form. These problems include normal equations approaches to solving linear least squares and weighted linear least squares problems, and interior point algorithms for linear and nonlinear programming problems. We develop in this work efficient GPU algorithms for forming and factoring $\mathbf{AD}^2\mathbf{A}^T$ by exploiting the triangular rasterization capabilities of the GPU.

This report summarizes work from 2005 to 2007 and was supported in part by the US Department of Energy under Grant DEFG0204ER25655 and by the National Science Foundation under Grant CCF 05 14213.

Keywords: GPGPU, general purpose graphics processing units, symmetric matrix, triangular matrix, rectangular packed format, matrix computation, factorization, decomposition, weighted least squares.

*Department of Computer Science, University of Maryland, College Park, MD 20742. jjung@cs.umd.edu, salbang+csr@gmail.com

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. oleary@cs.umd.edu

1 Introduction

Given time series data, b_0, \dots, b_{n-1} measured at times t_0, \dots, t_{n-1} , we may want to fit a model to the data in order to reduce the effects of noise. We choose a set of basis functions $\phi_0(t), \dots, \phi_{m-1}(t)$ that can describe the behavior of the system and build a model

$$u(t) = \sum_{j=0}^{m-1} x_j \phi_j(t). \quad (1)$$

To find the coefficients x_0, \dots, x_{m-1} , we solve a *weighted least squares* problem involving a matrix \mathbf{A} , with entries computed by evaluating the basis functions at the measured times, and a diagonal matrix \mathbf{D} , with entries determined by the uncertainties in the measurements. We solve the problem by forming the matrix $\mathbf{AD}^2\mathbf{A}^T$ and then computing a factorization of this matrix [5]. The cost is proportional to nm^2 . In data fitting problems, m may be on the order of tens or hundreds, but n can be quite large.

Many other important computational problems also involve forming and factoring a matrix of the form $\mathbf{AD}^2\mathbf{A}^T$. For example, interior point algorithms for linear and nonlinear programming solve a sequence of weighted least squares problems [12, 10]. In these applications, m and n can be thousands or more.

Because of the expense of solving these problems, the massive parallelism of the GPU architecture is very attractive [8, 3, 1, 9], and we develop in this work efficient GPU algorithms for forming and factoring $\mathbf{AD}^2\mathbf{A}^T$.

In section 2, we briefly review the weighted least squares problem and how it is solved. Since the matrix $\mathbf{AD}^2\mathbf{A}^T$ is symmetric, only its lower triangular part is needed, and this reduces computation by 50%. We discuss in section 3 how the matrix can be assembled on a GPU using triangular rasterization. Then we present a GPU algorithm based on a rectangular packed storage form. For packed storage, we store a lower triangular matrix by moving the submatrix at the bottom right, rotated by 180 degrees, to the unused upper right corner of the array, thus reducing the required storage. Section 4 explains how factorization can be performed in either full or packed storage. Results in section 5 demonstrate that by rasterizing two triangles simultaneously, we can assemble and factor the packed matrix as fast as the non-packed matrix. Section 6 gives our conclusions and implications for design of GPU languages.

For consistency with GPU notation, we number elements in matrices and vectors starting with 0 instead of 1.

2 Weighted Least Squares

Given the data (t_k, b_k) ($k = 0, \dots, n-1$), and the model (1), we want the model to match the observed data as well as possible:

$$b_k \approx \sum_{j=0}^{m-1} x_j \phi_j(t_k).$$

To find good coefficients, we could think of summing the squares of the differences between the values b_k and the model prediction. Let \mathbf{A} be the $m \times n$ matrix with entries $a_{jk} = \phi_j(t_k)$. If we form a vector \mathbf{b} from the values b_k and a vector \mathbf{x} from the coefficients x_j , then this sum of squares can be written as

$$\sum_{k=0}^{n-1} \left(b_k - \sum_{j=0}^{m-1} x_j \phi_j(t_k) \right)^2 \equiv \|\mathbf{b} - \mathbf{A}^T \mathbf{x}\|^2.$$

The terms in the summation should be weighted by how sure we are about the data; we multiply the k -th term by d_k^2 , where we assume that the error in the observation b_k has mean 0 and standard deviation d_k^{-1} . Then our problem becomes

$$\begin{aligned} & \min_{\mathbf{x}} \sum_{k=0}^{n-1} d_k^2 \left(b_k - \sum_{j=0}^{m-1} x_j \phi_j(t_k) \right)^2 \\ & \equiv \min_{\mathbf{x}} \|\mathbf{D}(\mathbf{b} - \mathbf{A}^T \mathbf{x})\|^2, \end{aligned} \quad (2)$$

where \mathbf{D} is a diagonal matrix with entries d_k .

To solve this problem, we can set the gradient of $\|\mathbf{D}(\mathbf{b} - \mathbf{A}^T \mathbf{x})\|^2$ to zero, obtaining the linear system of equations

$$\mathbf{A} \mathbf{D}^2 \mathbf{A}^T \mathbf{x} = \mathbf{A} \mathbf{D}^2 \mathbf{b}. \quad (3)$$

If \mathbf{A} has full rank (i.e., the basis functions ϕ_j are linearly independent and the data points t_k are distinct), then this system has a unique solution.

Therefore, we can solve our weighted least squares problem (2) by forming the matrix $\mathbf{A} \mathbf{D}^2 \mathbf{A}^T$ and then solving the linear system of equations. The most efficient method for solving this requires computing a lower triangular matrix \mathbf{L} so that $\mathbf{L} \mathbf{L}^T = \mathbf{A} \mathbf{D}^2 \mathbf{A}^T$. The matrix \mathbf{L} is called the Cholesky factor of $\mathbf{A} \mathbf{D}^2 \mathbf{A}^T$, and its computation requires approximately $m^3/3$ operations.

We propose to form $\mathbf{A} \mathbf{D}^2 \mathbf{A}^T$ and factor it on a GPU. This might limit us to single precision. If we require a double precision answer \mathbf{x} , then it might be necessary to perform mixed precision iterative refinement, as explained in Algorithm 1. Note that we form the initial \mathbf{x}_{sgl}^0 by forward and back substitution, solving $\mathbf{L} \mathbf{y} = \mathbf{A} \mathbf{D}^2 \mathbf{b}$ from the first to the last equation, and then solving $\mathbf{L}^T \mathbf{x}_{sgl}^0 = \mathbf{y}$ from the last to the first equation.

Algorithm 1 Mixed-precision iterative refinement [2, 5]

```

 $\mathbf{x}_{dbl}^0 = \text{double}(\mathbf{x}_{sgl}^0);$ 
 $k = 0;$ 
repeat
   $\mathbf{r}^k = \mathbf{AD}^2\mathbf{b} - \mathbf{AD}^2\mathbf{A}^T\mathbf{x}_{dbl}^k;$  //  $\mathbf{A}, \mathbf{D}$  and  $\mathbf{b}$  in double precision
  Solve  $\mathbf{LL}^T\mathbf{x}_{sgl} = \text{single}(\mathbf{r}^k);$  //  $\mathbf{L}$  in single precision
   $\mathbf{x}_{dbl}^{k+1} = \mathbf{x}_{dbl}^k + \text{double}(\mathbf{x}_{sgl});$ 
   $k = k + 1$ 
until  $\|\mathbf{r}^k\|_2 / \|\mathbf{x}_{dbl}^{k+1}\| \leq \epsilon$  or  $k \geq \text{iteration\_limit}$ 

```

3 Assembling Symmetric Matrices

In this section we discuss how $\mathbf{AD}^2\mathbf{A}^T$ can be formed efficiently.

3.1 Full format on a CPU

The full or non-packed format stores an $m \times m$ matrix in an $m \times m$ array or texture as illustrated in Fig. 1. The figure also shows how we store other matrices and vectors used for assembling $\mathbf{AD}^2\mathbf{A}^T$.

Denote the k -th column of \mathbf{A} by \mathbf{a}_k . We observe that

$$\mathbf{C} \equiv \mathbf{AD}^2\mathbf{A}^T = \sum_{k=0}^{n-1} d_k^2 \mathbf{a}_k \mathbf{a}_k^T.$$

Therefore, we can initialize the array \mathbf{C} to $\mathbf{0}$ and, at step k , add in $\mathbf{b}\mathbf{a}_k^T$, where $\mathbf{b} = d_k^2 \mathbf{a}_k$.

Assembling the matrix $\mathbf{AD}^2\mathbf{A}^T$ without considering its structure wastes computational resources, because its upper and lower triangular parts are exactly the same. To avoid redundancy we may omit computing the upper triangular part and copy the lower triangular part if necessary. The algorithm consists of two main routines, column scaling and outer product addition. At the k -th iteration, we scale \mathbf{a}_k by d_k^2 , and store it in a temporary vector \mathbf{b} . Then we add the outer product of \mathbf{b} and \mathbf{a}_k to \mathbf{C} . Considering the redundancy, we may write a CPU version that computes only the lower triangular matrix as in Algorithm 2.

3.2 Full format on a GPU

We can match the two routines, column scaling and outer product addition, to GPU kernels as described in Kernels 1 and 2 and Fig. 2. The key to saving computational time lies in how we arrange the two nested **for** loops (iterating with i and j) for the outer product addition. We design the loops so that they iterate over the lower triangular matrix. We can implement this simple strategy using the triangular rasterization on a GPU. Drawing an isosceles right triangle

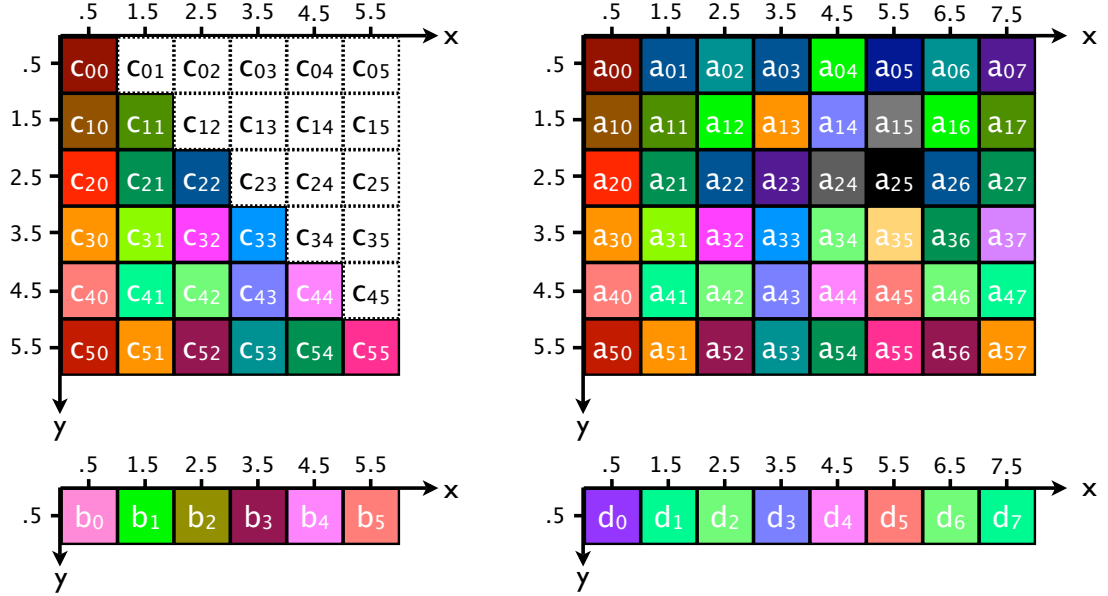


Figure 1: This figure illustrates how we store matrices and vectors for $m = 6$ and $n = 8$. We use C for storing the result of AD^2A^T , b for temporary storage, and d for storing the diagonal of D^2 . The white colored entries of C store 0.

Algorithm 2 A CPU algorithm to form AD^2A^T

```

// Indices are in (row, column) order
// Input d contains the diagonal elements of the scaling matrix D2
// We store a scaled column in b
C = zeros(m,m);
for k = 0 to n-1 do
    // Column scaling
    for i = 0 to m-1 do
        b(i) = d(k)*A(i,k);
    end for
    // Outer product addition
    for j = 0 to m-1 do
        for i = j to m-1 do
            C(i, j) = C(i, j) + A(i, k)×b(j);
        end for
    end for
end for

```

can initiate the outer product addition kernel to work on the lower triangular matrix as illustrated in Fig. 2(b).

Kernel 1 GPU kernel for column scaling

```
float main( uniform samplerRECT A : TEXUNIT0,
            uniform samplerRECT d : TEXUNIT1,
            float3 index          : TEXCOORD0 ) : COLOR {
    return texRECT(A, index.yx) * texRECT(d, index.yz);
}
```

Kernel 2 GPU kernel for outer product addition

```
float main( uniform samplerRECT C : TEXUNIT0,
            uniform samplerRECT A : TEXUNIT1,
            uniform samplerRECT b : TEXUNIT2,
            float2 C_index        : WPOS,
            float2 A_index        : TEXCOORD0,
            float2 b_index        : TEXCOORD1 ) : COLOR {
    return texRECT(C, C_index.xy)
        + texRECT(A, A_index.xy) * texRECT(b, b_index.xy);
}
```

To generate texture coordinates for fetching the input textures \mathbf{A} and \mathbf{b} in the outer product addition kernel, we should attach texture coordinates to each vertex of the triangle. The matrix \mathbf{A} is stored in an $n \times m$ texture and the temporary vector \mathbf{b} , which stores a scaled column of \mathbf{A} , is stored in an $m \times 1$ texture with *width* \times *height* ordering. At the k -th iteration, we need two sets of texture coordinates, $(k + 0.5, j)$ for fetching \mathbf{A} and $(i, 0.5)$ for \mathbf{b} , to process a fragment at (i, j) (with (x, y) coordinates ordering). The y coordinate of $(k + 0.5, j)$ and the x coordinate of $(i, 0.5)$ are synchronized with the fragment position. So for a vertex at (x, y) , we attach two sets of texture coordinates $(k + 0.5, y)$ for \mathbf{A} and $(x, 0.5)$ for \mathbf{b} . Then the rasterizer will linearly interpolate the required coordinates for fetching the inputs.

Implementing the column scaling operation is easier than the outer product addition. We first set the viewport size as $m \times 1$ so that the size of a pixel matches the size of a texel. Of course, we should change the viewport size to $m \times m$ when we perform the outer product addition. To make the column scaling kernel operate on the temporary vector \mathbf{b} , we draw a rectangle with vertices at $(0, 0)$, $(0, 1)$, $(m, 1)$ and $(m, 0)$. In processing a fragment at $(i, 0.5)$ of \mathbf{b} , the kernel needs to fetch the entries at $(k + 0.5, i)$ of \mathbf{A} and at $(k + 0.5, 0.5)$ of \mathbf{d} . The y coordinate of the texture coordinates for fetching \mathbf{A} is synchronized with the x coordinate of the fragment position. So for a vertex at (x, y) , we attach a set of texture coordinates $(x, k + 0.5, 0.5)$, and use the y and x coordinates of the interpolated coordinate triad for fetching \mathbf{A} and y and z for \mathbf{d} . By combining the two routines, we can write an algorithm for assembling $\mathbf{A}\mathbf{D}^2\mathbf{A}^T$ on a GPU as described in Algorithm 3.

Algorithm 3 Assembling $\mathbf{AD}^2\mathbf{A}^T$ in the full format on a GPU

```

Create two textures,  $\mathbf{C}_S$  and  $\mathbf{C}_T$ , of size  $m \times m$ ;
Create a texture  $\mathbf{b}$  of size  $m \times 1$ ;
Bind  $\mathbf{C}_S$  as the target;
Clear the target buffer;
Bind  $\mathbf{C}_T$  as the target;
Clear the target buffer;
for  $k = 0$  to  $n-1$  do
  // Scale the  $k$ -th column of  $\mathbf{A}$ 
  Load 'column scaling' kernel;
  Bind  $\mathbf{b}$  as the target;
  Bind  $\mathbf{A}$  and  $\mathbf{d}$  as the input textures;
  Set viewport size as  $m \times 1$ ;
  Draw a rectangle covering the texture  $\mathbf{b}$ ;
  // Add the  $k$ -th outer product to  $\mathbf{C}_T$ 
  Load 'outer product addition' kernel;
  Bind  $\mathbf{C}_T$  as the target;
  Bind  $\mathbf{C}_S$ ,  $\mathbf{A}$  and  $\mathbf{b}$  as the input textures;
  Set viewport size as  $m \times m$ ;
  Draw a triangle covering the lower triangular part of  $\mathbf{C}_T$ ;
  Swap  $\mathbf{C}_T$  and  $\mathbf{C}_S$ ;
end for
  
```

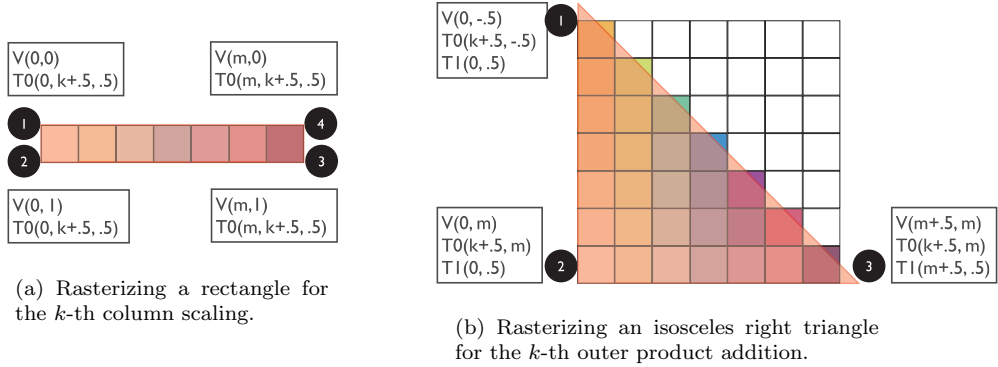


Figure 2: k -th iteration of the matrix multiplication for computing $\mathbf{AD}^2\mathbf{A}^T$. Coordinates are in (x, y, z) order. T_i represents the i -th texture coordinates attached to a vertex V . For fixed texture coordinates, we use an offset of 0.5 to point to the center of texels. The rasterizer will generate the 0.5 offset for interpolated coordinates.

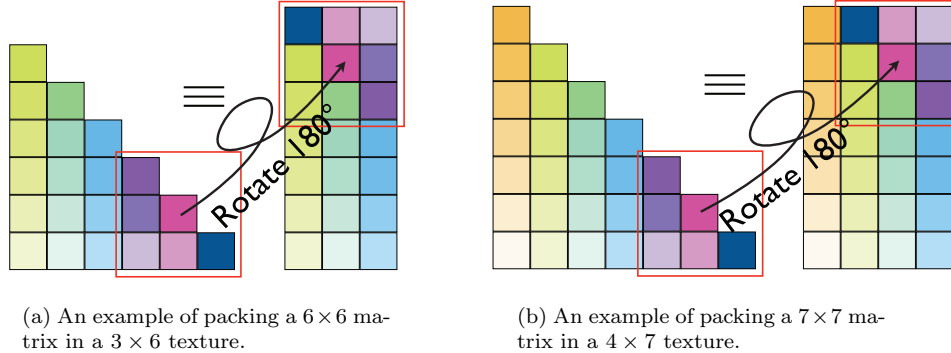


Figure 3: In the packed format, we save storage by storing the right lower triangular submatrix in the upper corner by rotating it.

3.3 Rectangular packed format

In addition to saving computational resources, we can save storage space by exploiting the matrix structure. Since our GPU algorithm computes only the lower triangular matrix, the upper triangular part is not used at all. Gunnels and Gustavson explored a rectangular packed format requiring half the storage space [6]. In our work, we store the rotated right lower triangular submatrix in the upper left corner as illustrated in Fig. 3. Packing a symmetric or triangular matrix results in a $w \times h$ texture, where $w = \lceil m/2 \rceil$ and $h = (m + \text{mod}(m+1, 2))$ represent width and height, respectively.

The key to assembling the matrix in the packed format is the triangular rasterization. The column scaling is not different from section 3.2. For the outer product addition, we need to draw two separate triangles, because the input access pattern of the fragments in the lower trapezoid is different from the upper triangle. So one triangle, with vertices 1 through 3, covers the trapezoid and the other, with vertices 4 through 6, covers the upper triangular part.

We can use the same kernels by attaching different texture coordinates T_0 and T_1 to each vertex V of the triangle covering the upper part. We attach texture coordinates for the vertices of the upper part as we would for their original positions in the full format. The texture coordinates attached to vertices 1 through 3 are exactly the same as those used for the full format. We summarize how we issue vertices with attached texture coordinates in Fig. 4.

4 Cholesky Decomposition

We can factor an $m \times m$ symmetric positive definite matrix \mathbf{C} as $\mathbf{C} = \mathbf{L}\mathbf{L}^T$, where \mathbf{L} is a lower triangular matrix and is usually referred to as the Cholesky factor.

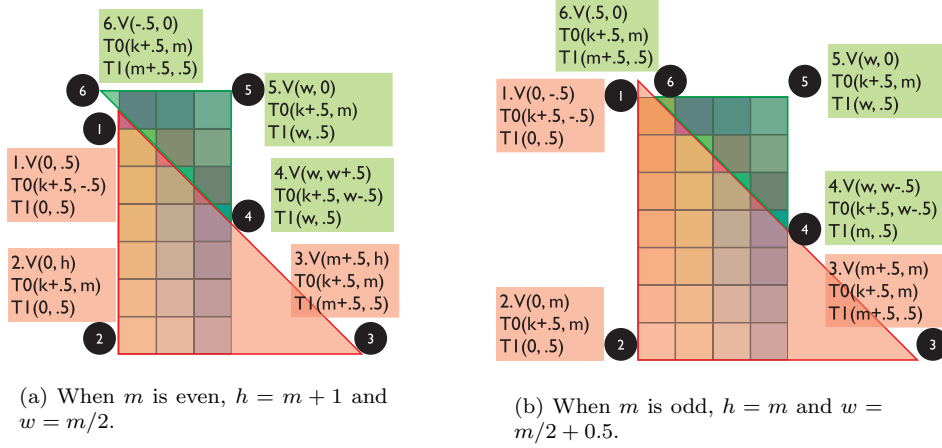


Figure 4: Drawing two isosceles right triangles is the key to assembling the matrix in packed format. Notice that the texture coordinates for the 6th vertex are same as those for the 3rd. Texture coordinates for the 5th vertex would be same as those for a vertex at (w, h) .

4.1 Full format on CPU

A CPU implementation of the Cholesky decomposition consists of three components, described in Algorithm 4. We repeat square rooting the k -th diagonal entry, normalizing the entries below the diagonal, and subtracting the outer product of the normalized column from the entries in the remaining lower triangular submatrix. In the k -th normalization process, we use the *pivot* to indicate the k -th diagonal entry. In the k -th outer product subtraction process, we refer to the entries at (j, k) and (i, k) as the *pivot* and *neighbor* of an active entry at (i, j) , respectively.

4.2 Full format on GPU

We can transform each component of Algorithm 4 to a GPU kernel as described in Kernels 3, 4 and 5.

Kernel 3 GPU kernel for square rooting

```
float main( uniform samplerRECT L : TEXUNIT0,
            float2 index          : WPOS ) : COLOR {
    return sqrt(texRECT(L, index.xy));
}
```

Algorithm 4 A CPU version of Cholesky decomposition

```
// Indices are in (row, column) order
L = lower_triangle(C);
for k = 0 to m-2 do
    // Square rooting
    L(k, k) = sqrt(L(k, k));
    // Normalization
    for i = k+1 to m-1 do
        L(i, k) = L(i, k)/L(k, k);
    end for
    // Outer product subtraction
    for i = k+1 to m-1 do
        for j = k+1 to i do
            L(i, j) = L(i, j) - L(j, k)×L(i, k);
        end for
    end for
end for
L(m-1, m-1) = sqrt(L(m-1, m-1));
```

Kernel 4 GPU kernel for normalization

```
float main( uniform samplerRECT L : TEXUNIT0,
            float2 index          : WPOS,
            float2 pivot_index    : TEXCOORD0 ) : COLOR {
    return texRECT(L, index.xy)
        /texRECT(L, pivot_index.xy);
}
```

Kernel 5 GPU kernel for outer product subtraction

```
float main( uniform samplerRECT L : TEXUNIT0,
            float2 index          : WPOS,
            float2 neighbor_index : TEXCOORD0,
            float2 pivot_index    : TEXCOORD1 ) : COLOR {
    return texRECT(L, index.xy) -
        texRECT(L, neighbor_index.xy)
        *texRECT(L, pivot_index.yx);
}
```

For the outer product subtraction, we draw an oversized triangle whose vertices are at $(k+1, k+0.5)$, $(k+1, m)$ and $(m+0.5, m)$ to cover the lower triangular matrix. As seen in Fig. 5, we attach two sets of texture coordinates to each vertex to generate indices for the active neighbor and pivot. An active fragment at (j, i) will have the interpolated texture coordinates T_0 of $(k+0.5, i)$ and T_1 of $(j, k+0.5)$. As described in Fig. 5(b), the pivot is at $(k+0.5, j)$. So we use the GPU's swizzle operator to rearrange indices for the pivot.

For the square rooting, we simply draw a square to make the kernel operate on the diagonal entry. For the normalization kernel, we draw a rectangle covering the off-diagonal column with attached texture coordinates pointing to the

square rooted diagonal entry. By combining all three rasterization processes, we can write a GPU version of Cholesky decomposition as described in Algorithm 5.

Algorithm 5 Cholesky decomposition on a GPU

```

Create two streams  $\mathbf{L}_S$  and  $\mathbf{L}_T$  of size equal to  $\mathbf{C}$ ;
Copy  $\mathbf{C}$  to  $\mathbf{L}_S$ ;
for  $k = 0$  to  $m - 2$  do
  Load square root kernel;
  Bind  $\mathbf{L}_T$  as the target and  $\mathbf{L}_S$  as an input texture;
  Draw a square covering the entry at  $(k, k)$ ;
  Load copy kernel;
  Bind  $\mathbf{L}_S$  as the target and  $\mathbf{L}_T$  as an input texture;
  Draw a square covering the entry at  $(k, k)$ ;

  Load normalization kernel;
  Bind  $\mathbf{L}_T$  as the target and  $\mathbf{L}_S$  as an input texture;
  Draw a rectangle covering the  $k$ -th column below the diagonal;
  Load copy kernel;
  Bind  $\mathbf{L}_S$  as the target and  $\mathbf{L}_T$  as an input texture;
  Draw a rectangle covering the  $k$ -th column below the diagonal;

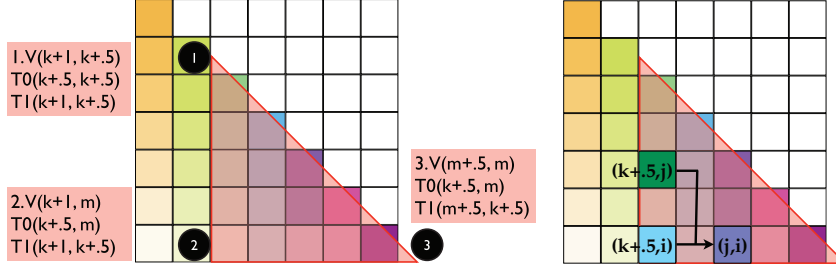
  Load outer product subtraction kernel;
  Bind  $\mathbf{L}_T$  as the target and  $\mathbf{L}_S$  as an input texture.
  Draw an isosceles right triangle covering lower triangular submatrix;

  Swap  $\mathbf{L}_S$  and  $\mathbf{L}_T$ ;
end for
Load square root kernel;
Bind  $\mathbf{L}_T$  as the target and  $\mathbf{L}_S$  as an input texture;
Draw an isosceles right triangle covering the entry at  $(m - 1, m - 1)$ ;

// Now  $\mathbf{L}_T$  is the Cholesky factor of  $\mathbf{C}$ 

```

To complete the factorization, we draw m squares, $m - 1$ rectangles, and $m - 1$ triangles to initiate the square rooting, normalization, and outer product subtraction kernel, respectively. Drawing a square covering a single pixel to initiate the square root kernel does not utilize the parallel architecture [7]. As a result our Cholesky algorithm utilizes less memory bandwidth for small matrices than the GPU LU decomposition algorithm of Galoppo et al. [4]. Coping with this trouble requires architectural changes supporting thread level parallelism. With the support, square rooting could be done in conjunction with the outer product subtraction once the outer product subtraction kernel finishes computing the left-most diagonal entry.



(a) We draw an oversized triangle to cover the sub-lower triangular part of the matrix. We attach texture coordinates T_0 to the vertices to get the indexes for the active pivot and neighbor elements.

(b) An active fragment at (j, i) has its active neighbor at $(k + 0.5, j)$ and active pivot at $(k + 0.5, i)$, whereas it has interpolated texture coordinates T_0 of $(k + 0.5, i)$ and T_1 of $(j, k + 0.5)$.

Figure 5: These figures illustrate how we rasterize an oversized triangle for the k -th outer product subtraction of Cholesky decomposition.

4.3 Rectangular packed format

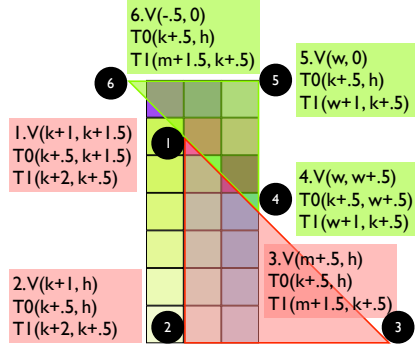
In order to factor a symmetric positive definite matrix in packed format, we draw two triangles for the outer product subtraction in the first half of the iterations as we did in section 3.3. In the second half of the iterations, we draw a single triangle covering the upper triangular part. Since we draw triangles differently, we need to divide the single **for** loop in Algorithm 5 into two, the first of which iterates k from 0 to $w - 1$ and the second of which iterates k from $w - 1$ to 1 (if m is even, as in Fig. 6) or to 2 (if m is odd, as in Fig. 7). We can also use the same kernels that we used for the full format by attaching different texture coordinates to each vertex of the triangles. We summarize how to specify vertices and texture coordinates for the outer product subtraction in Fig. 6 and Fig. 7.

5 Results

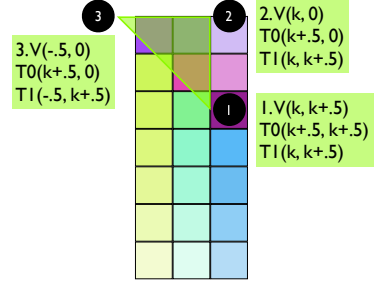
We tested our algorithms using a PC equipped with an Intel Xeon 3.0GHz CPU and a GeForce 7800 GTX GPU attached to 16x PCIe slot, and running on 64bit Red Hat Linux.

5.1 Assembling AD^2A^T

The packed format can save half of the storage space without sacrificing speed. Fig. 8 compares our implementation for packed format with an algorithm made with **saxpy** and **ssyrk** of the ATLAS (Automatically Tuned Linear Algebra

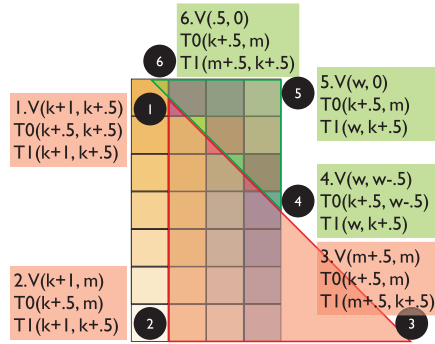


(a) For the first half of the iterations, we iterate k from 0 to $w - 1$. The texture coordinates attached to the 6th vertex are same as those for the 3rd vertex.

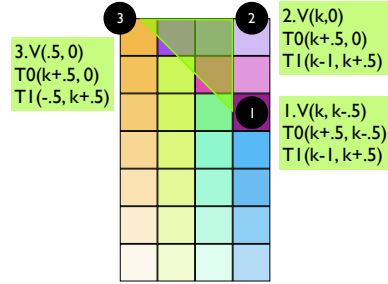


(b) For the second half of the iterations, we decrease k from $w - 1$ to 1.

Figure 6: These figures illustrate how we rasterize the triangles for the k -th outer product subtraction of Cholesky decomposition for the packed format when m is even.



(a) For the first half of the iterations, we iterate k from 0 to $w - 1$. The texture coordinates attached to the 6th vertex are same as those for the 3rd vertex.



(b) For the second half of the iterations, we decrease k from $w - 1$ to 2.

Figure 7: These figures illustrate how we rasterize the triangles for the k -th outer product subtraction of Cholesky decomposition for the packed format when m is odd.

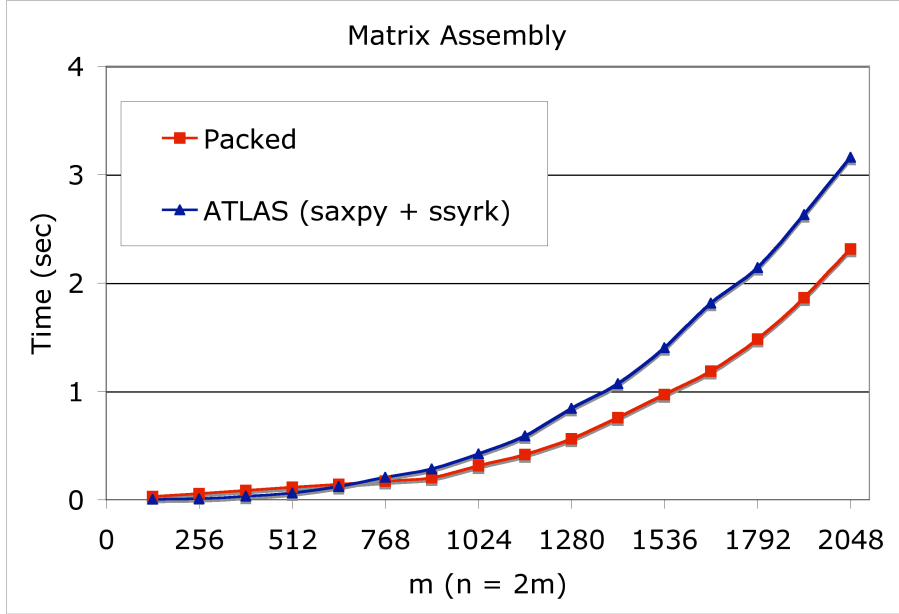


Figure 8: Timing result for assembling $\mathbf{AD}^2\mathbf{A}^T$. Transferring \mathbf{A} and \mathbf{d} to and retrieving the result from the GPU are not considered.

Software) library. As the packed version performs almost the same as the non-packed version, we do not provide the timing result for non-packed format. (For $m=2048$, the non-packed version is only 1% faster than the packed version.) Drawing two triangles only requires three more vertices and three more sets of texture coordinates, and the time is negligible. The number of processed fragments at every iteration determines the performance of our algorithm, and it is exactly the same in both the non-packed and the packed version.

Our GPU implementations outperforms that of ATLAS for m larger than 768, but, due to latency caused by initiating kernels through drawing shapes, GPU implementations are slower than ATLAS for small matrices. Problems of size less than 1024 are considered small by today’s standards.

5.2 Cholesky Decomposition

Triangular rasterization is the key to implementing the algorithms for decomposition. Without the triangular rasterization, we cannot achieve performance gain over LU [7]. We compared our Cholesky algorithm with the LU algorithm written by Galoppo et al. [4]. The LU has wider applicability (since it can be used for nonsymmetric matrices, too) but requires twice as many arithmetic operations. As seen in Fig. 9, our algorithm outperformed the LU algorithm by 83.5% for $m = 3584$. We also compared our algorithms with **spotrf**, the Cholesky decomposition algorithm of ATLAS.

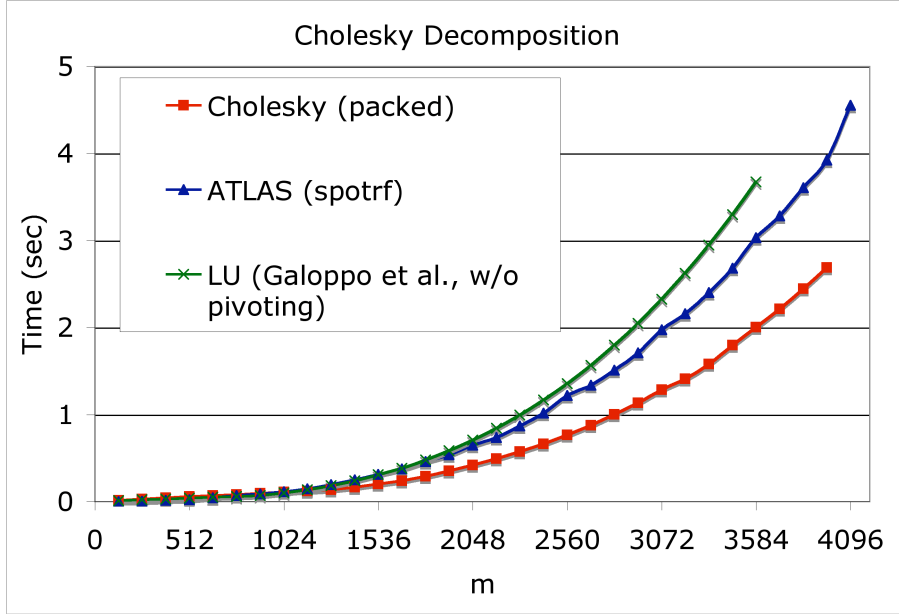


Figure 9: Timing result for Cholesky decomposition.

As for the Cholesky decomposition algorithm, the number of generated fragments at every iteration is exactly the same in both the non-packed and the packed version. For this reason, the algorithm for the rectangular packed format performs as fast as that for the non-packed format. For $m = 3328$, the non-packed version is only 1% faster than the packed. We cannot pack the matrix for $m = 4096$, because the packed matrix would have height 4097 whereas the maximum texture height that the GPU supports is 4096.

5.3 Weighted Least Squares

In our implementation, we used the MATLAB-C interface to make use of MATLAB functions for operations other than the assembly and factorization. CPU operations use double precision. MATLAB uses Intel Math Kernel Library for the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) routines. Table 1 and Table 2 show the timing results and performance gain of our implementation using the GPU over the CPU implementation in solving randomly generated problems of various sizes, with $n = 2m$. We measured the speedup by dividing the CPU timing result by that of GPU. We measured the relative error of \mathbf{x}_{dbl}^k by computing

$$\frac{\|\mathbf{x}_{dbl}^k - \mathbf{x}_{CPU}\|}{\|\mathbf{x}_{CPU}\|}$$

Table 1: Results for randomly generated weighted least squares.

m	Relative Error		Refinement iteration count	Time (s)				Speedup
	\mathbf{x}_{sgl}^0	Refined		GPU		CPU		
				\mathbf{x}_{sgl}^0	Refine	Total		
512	3.11E-04	3.37E-13	4	0.39	0.02	0.41	0.20	0.49
1024	3.10E-04	4.25E-13	4	0.49	0.04	0.53	1.03	1.94
1536	4.61E-04	6.96E-13	4	1.33	0.09	1.42	3.41	2.40
2048	1.01E-03	1.76E-12	5	2.99	0.22	3.21	7.95	2.48

Table 2: Results for randomly generated ill-conditioned weighted least squares.

m	Relative Error		Refinement iteration count	Time (s)				Speedup
	\mathbf{x}_{sgl}^0	Refined		GPU		CPU		
				\mathbf{x}_{sgl}^0	Refine	Total		
512	2.92E-02	1.16E-10	7	0.36	0.02	0.38	0.20	0.53
1024	7.19E-02	2.01E-10	10	0.48	0.10	0.58	1.05	1.81
1536	9.35E-02	2.37E-10	13	1.33	0.30	1.63	3.41	2.09
2048	1.13E-01	3.41E-10	15	3.00	0.66	3.66	7.98	2.18

where \mathbf{x}_{CPU} is the solution to the system of equations (3) computed only by the CPU in double precision. Timing results to get \mathbf{x}_{sgl}^0 include transferring the data \mathbf{A} and \mathbf{d} , and retrieving the Cholesky factor \mathbf{L} . We set the tolerance parameter ϵ to 10^{-8} and the iteration limit to 100 in Algorithm 1. We generated the data matrix \mathbf{A} , the diagonal entries of \mathbf{D}^2 , and the vector \mathbf{b} using uniformly distributed random numbers between 0 and 1. To generate ill-conditioned problems, we set the i -th diagonal element of \mathbf{D}^2 as $10^{-4+8i/(n-1)}$ for $i = 0, \dots, n-1$. Ill-conditioned problems require more refinement iterations. Notice that we obtained more than 100% speedup for m larger than 1536 in both types of problems.

6 Conclusions

We have presented efficient GPU algorithms for forming a dense positive definite matrix $\mathbf{AD}^2\mathbf{A}^T$ and then computing its Cholesky factor. The best algorithms exploit triangular rasterization to minimize storage without increasing computation time. By comparing our implementations with conventional CPU algorithms, we demonstrated the potential of the commodity parallel architecture of a GPU for solving important numerical problems.

We demonstrated performance advantage in solving weighed least squares problems by using a GPU. In such applications, assembling and factoring are most demanding of computational resources. Thus support for triangular rasterization is essential in exploiting the structure of symmetric or triangular

matrices. This provides one reason for developers of streaming languages such as BrookGPU [1] and Accelerator [11] to consider implementing this feature.

References

- [1] I. Buck, T. Foley, D. Horn, J. Sugerma, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, in: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, ACM, New York, NY, USA, 2004.
- [2] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, J. Kurzak, Mixed precision iterative refinement techniques for the solution of dense linear systems, *International Journal of High Performance Computing Applications* 21 (4) (2007) 457–466.
- [3] K. Fatahalian, J. Sugerma, P. Hanrahan, Understanding the efficiency of GPU algorithms for matrix-matrix multiplication, in: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, ACM Press, New York, NY, USA, 2004.
- [4] N. Galoppo, N. K. Govindaraju, M. Henson, D. Manocha, LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware, in: SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2005.
- [5] G. H. Golub, C. F. Van Loan, *Matrix Computations* (3rd ed.), Johns Hopkins University Press, Baltimore, MD, USA, 1996.
- [6] J. A. Gunnels, F. G. Gustavson, A new array format for symmetric and triangular matrices., in: PARA, 2004.
- [7] J. H. Jung, Cholesky decomposition and linear programming on a GPU, *Scholarly Paper* (Jan. 2006).
- [8] J. Krüger, R. Westermann, A GPU framework for solving systems of linear equations, in: M. Pharr (ed.), *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chap. 44, Addison-Wesley, 2005, pp. 703–718.
- [9] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, Cg: a system for programming graphics hardware in a C-like language, in: SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, ACM, New York, NY, USA, 2003.
- [10] Y. Nesterov, A. Nemirovski, *Interior Point Polynomial Algorithms in Convex Programming*, SIAM, 1995.
- [11] D. Tarditi, S. Puri, J. Oglesby, Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism, *Tech. Rep. MSR-TR-2005-184*, Microsoft (Dec. 2005).

- [12] S. J. Wright, Primal-Dual Interior-Point Methods, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.