Cholesky Decomposition and Linear Programming on a GPU[*]

Jin Hyuk Jung [†],

Scholarly Paper

Directed by Dianne P. O'Leary [‡]

Abstract

Rapid evolution of GPUs in performance, architecture, and programmability provides general and scientific computational potential beyond their primary purpose, graphics processing. In this work we present an efficient algorithm for solving symmetric and positive definite linear systems using the GPU. Using the decomposition algorithm and other basic building blocks for linear algebra on the GPU, we demonstrate a GPU-powered linear program solver based on a Primal-Dual Interior-Point Method.

Keywords: GPGPU, Cholesky, Matrix Decomposition, Linear Programming, Interior Point Method

1 Introduction

Strong competition in the gaming industry is driving graphics processing hardware development. ATI's Radeon and NVIDIA's GeForce series, the dominant products in the market, offer highly programmable parallel engines for processing graphics problems. Due to competition and the rapidly growing game market, GPUs are now cheap parallel machines available to any users.

Although GPUs formerly were much slower than CPUs and had very limited programmability, now they show superior performance in some classes of applications and show much faster evolution speed than Moore's law predictions for CPUs [Har04]. For example, NVIDIA's latest graphics hardware GeForce 7800 GTX shows sustained performance of 165GFLOPS (300GFLOPS at peak) compared to a 24.6 GFLOPS theoretical peak for a 3GHz Intel dual-core Pentium 4 [Lue05]. Recent support for single pre-

[*]This work was supported in part by the US Department of Energy under Grant DEFG0204ER25655.

[†]Department of Computer Science, University of Maryland, College Park, MD 20742 jjung@cs.umd.edu

[‡]Deparment of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 oleary@cs.umd.edu

cision floating point numbers and long programs gave rise to a new horizon of GPUs' applications. Researchers have been applying GPUs to general computations including evolutionary algorithms [WWF05], linear algebra [KW03], fluid dynamics [FQKYS04], FFT [MA03], and others [OLG+05].

GPUs have a different programming model from CPUs and their own high level programming languages, namely Cg (C for graphics) [Mar], HLSL (High Level Shading Language) [MIC03], and GLSL (OpenGL Shading Language) [Ros04]. Only assembly languages had been available before those high level languages were introduced. Although those high level languages are easier to use than the assembly languages, knowledge of esoteric graphics APIs, DirectX [MIC05] and OpenGL [WNDS05] for instance, is required to load, bind and invoke kernels, to manage streams, and to organize data. BrookGPU hides those difficulties and makes computations as a sequence of kernel calls on streams [BFH+04]. As a result, researchers don't need to perform those abstruse tasks associated with graphics and can concentrate on the development of kernels themselves.

Contributions: We present a new algorithm to decompose symmetric and positive definite dense matrices through a set of kernel calls with minimum copying operations to maximize performance. Using our algorithm and other BLAS kernels, we demonstrate how to build the GPU-powered primal-dual interior-point method with minimum feedback to the CPU. We use:

- Triangular domain updating to exploit the symmetric structure.
- Texture coordinate mapping and index swizzling for efficient texture fetching.

2 Background

2.1 Linear Algebra on GPUs

Only recently have GPUs been used for linear algebra, including programs for matrix multiplication [JH03], an iterative sparse system solver [BFGS03], a direct dense system solver [GGHM05], and others [OLG+05]. Our work to implement a direct solver for symmetric positive definite systems is an extension of those efforts.

2.2 GPU Architecture

This section briefly explains the architectural features of the GPU. A functional block diagram of a GPU is presented in Figure 1. GPUs are equipped with programmable *vertex* and *fragment processors*. General purpose GPU applications use fragment processors more frequently than vertex processors, because GPUs have more fragment processors than vertex processors and drawing a triangle or a rectangle is usually enough to map computational concepts.

The array is the most heavily used data structure to represent vectors and matrices. The counterpart

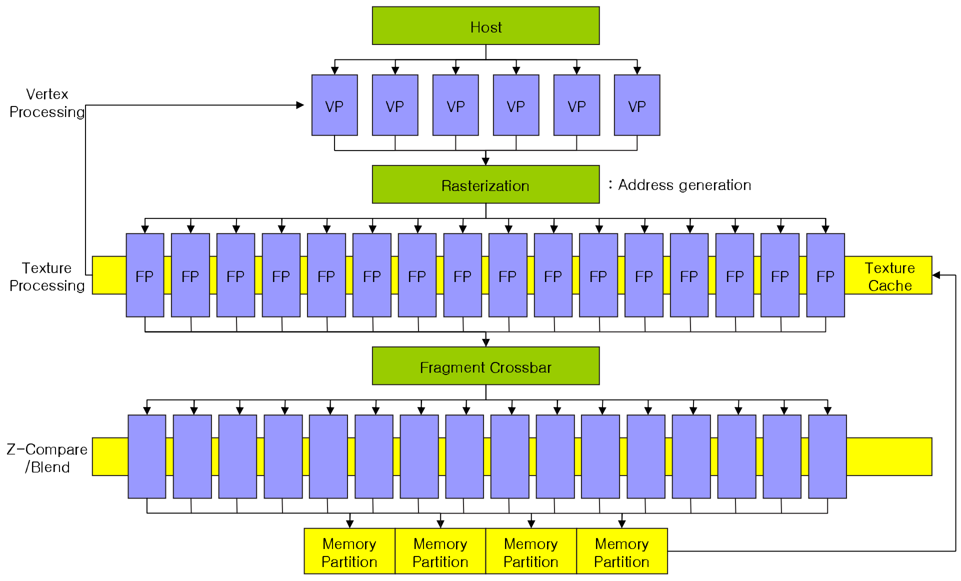


Fig. 1: GeForce 6 series architecture. (Source: GPU Gems 2, 2004) [KW05]

in the GPU is the *texture*, also known as the *stream* in the streaming model perspective. Textures were originally used for patterning of geometries, but modern GPUs are capable of rendering computational results directly to textures, which, in turn, can be fed back into the GPUs as new input streams without being copied back from the framebuffer.

A *kernel* or a *GPU fragment program* is a set of GPU instructions, which are applied to all elements of a stream. Many elements up to the number of fragment processors in the GPU can be processed in parallel as they would be in a SIMD machine. For example, the GeForce 6800 Ultra has 16 fragment processors. In addition, the instruction-level parallelism allows up to 4 arithmetic operations to be performed simultaneously in a fragment processor.

Most computations involve a set of kernel calls. A kernel must process the entire stream as specified in the command issued by the CPU. A subsequent kernel must wait until the previous kernel finishes. As the kernel calls are asynchronous to the CPU, the CPU can keep computing and issuing subsequent kernel calls, in turn, managed by the GPU driver.

As illustrated in Figure 2, a kernel is initiated by drawing a shape, usually a triangle or a quadrilateral in GPGPU(General Purpose GPU) computations. The shape is then transformed to a stream of fragments by the vertex processors and the rasterizer. The fragments are different from pixels in that they don't have colors, which are assigned at the fragment processors. The rasterizer is capable of interpolating texture coordinates linearly for each fragment from the coordinates assigned at each vertex of the shape. These interpolated coordinates are passed to fragment processors as inputs, which are usually used as indexes to fetch textures. Iterator streams provided in BrookGPU use this capability and are also mainly used for index generation.

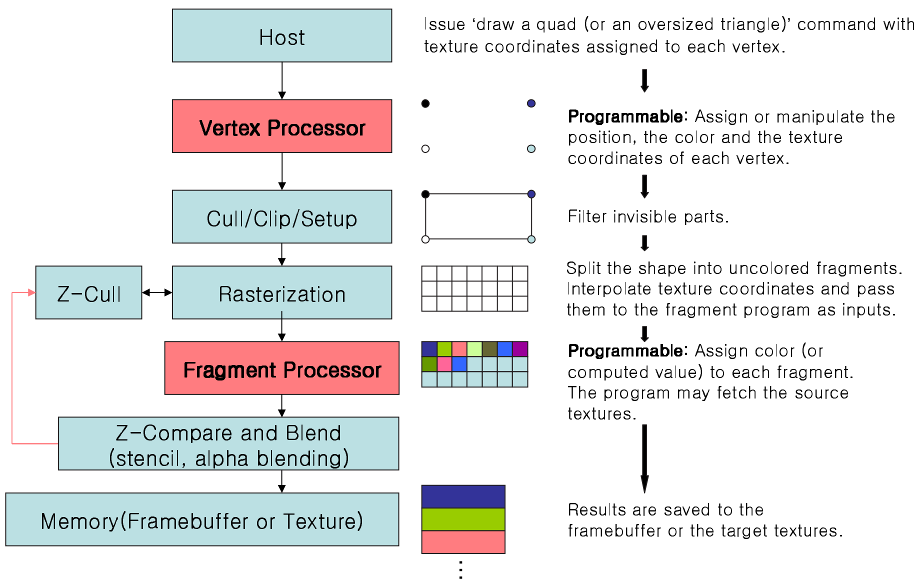

Host
Vertex Processor
Cull/Clip/Setup
Z-Cull
Rasterization
Fragment Processor
Z-Compare and Blend (stencil, alpha blending)
Memory(Framebuffer or Texture)
Issue 'draw a quad (or an oversized triangle)' command with texture coordinates assigned to each vertex.
Programmable: Assign or manipulate the position, the color and the texture coordinates of each vertex.
Filter invisible parts.
Split the shape into uncolored fragments. Interpolate texture coordinates and pass them to the fragment program as inputs.
Programmable: Assign color (or computed value) to each fragment. The program may fetch the source textures.
Results are saved to the framebuffer or the target textures.

Fig. 2: GPU pipeline. The vertex and fragment processors are the highly programmable components in a GPU, but BrookGPU exposes fragment processors only.

To represent a matrix, a 2 dimensional, single channel texture or stream of the same size is created. In specifying the domain of a output or input stream, x-coordinates correspond to column indexes, and y-coordinates to row indexes. So the index ordering is exactly in reverse to that for a matrix.

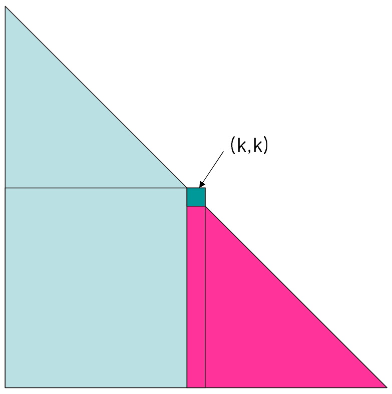
3 Cholesky decomposition on a GPU

3.1 Cholesky decomposition

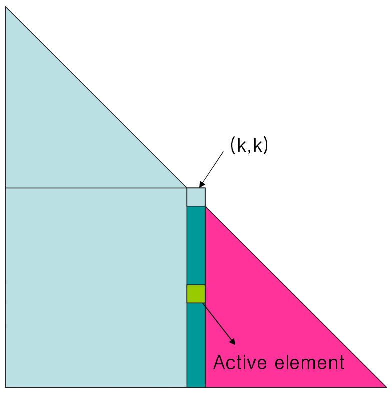
A system of linear equations, $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, where $\boldsymbol{A}$ is a large, dense $n \times n$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are column vectors of size n, can be efficiently solved using a decomposition technique, LU for instance. If the matrix is symmetric and positive definite, Cholesky decomposition is the most efficient in solving the system [GL96]. The Cholesky algorithm using vectorized notation can be stated as

Algorithm 1 Vectorized Cholesky decomposition

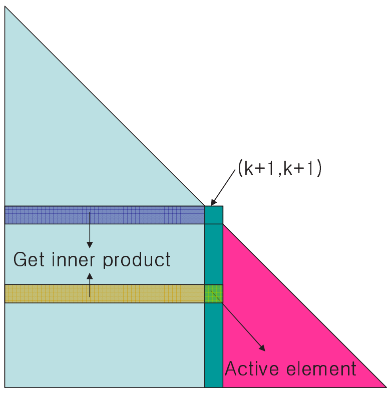
```
for k = 1 to n-1 do
    A(k, k) = sqrt(A(k, k))                          // Square rooting
    A(k+1:n, k) = A(k+1:n, k)/A(k, k)                // Normalization
    for j = k+1 to n do
        A(j, k+1) = A(j, k+1) - $A(j, 1:k)^T \times A(k+1, 1:k)$    // Inner product subtraction
    end for
end for
A(n, n) = sqrt(A(n, n))
```



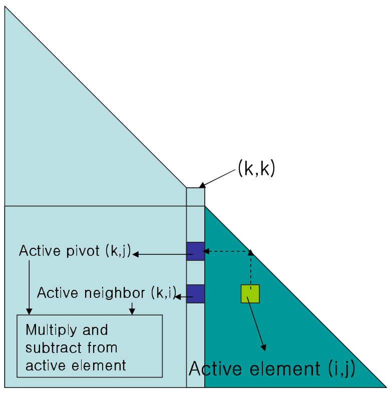
(a) Compute square root of the diagonal element at the kth column and row



(b) Divide each element in the kth column below the diagonal by the diagonal element



(c) **Inner product subtraction** process for inner product updating: Compute the inner product of the two sub-rows, the pivot and the active row, and then subtract the result from the active element



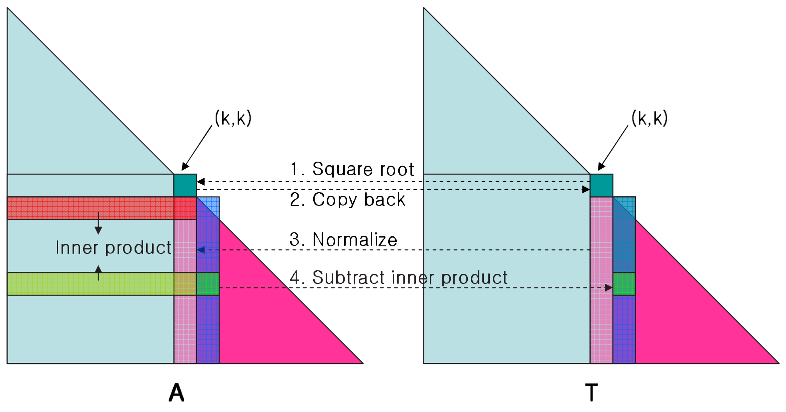
(d) **Submatrix update** process for outer product updating: Compute the product of the two elements, active pivot and active neighbor, and then subtract the result from the active element

Fig. 3: kth iteration of Cholesky decomposition

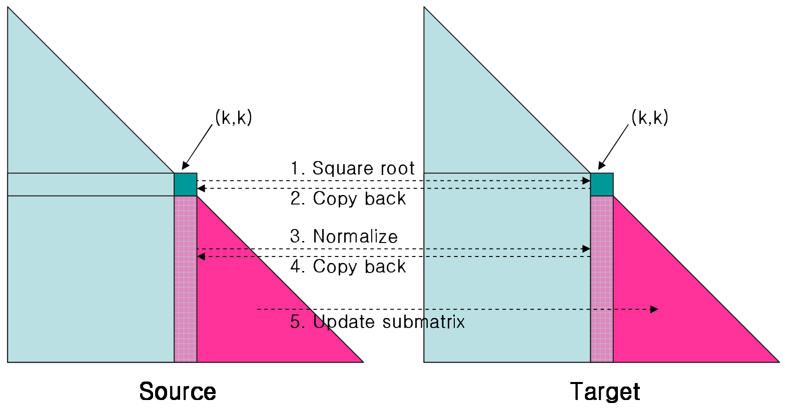
For $n-1$ iterations, 3 routines (square rooting, normalizing, and subtracting inner product or updating submatrix) are done as depicted in Figure 3. Only the routine for square rooting is performed at the last iteration. Each routine can be well matched to a GPU kernel as described in Figure 3a, 3b, and 3c or 3d.

The square root and the normalization kernels read and update only the k^{th} column below the diagonal. The inner product subtraction kernel reads the lower left submatrix of $\boldsymbol{A}$ to update the $k+1^{st}$ column below the diagonal. The submatrix update kernel reads the k^{th} column to update lower triangular region starting from the $k+1^{st}$ column. Thus there is no read-after-write(RAW) hazard in the parallel architecture. This may inspire one to bind one stream as both the source and the target at the same time.

However, the result of reading from and writing to the same stream simultaneously is not defined.



(a) Inner product form: Only one additional copy for one element at each iteration is needed to avoid simultaneous reading and writing.



(b) Outer product form: Two additional copies for a column are needed to avoid simultaneous reading and writing.

Fig. 4

Although the GeForce 6800 architecture works correctly with carefully written code using simultaneous reading and writing due to data independency, future architectures with more parallelism may produce incorrect results. To work around this we use a temporary storage of the size equal to $\boldsymbol{A}$ to pingpong the intermediate result as shown in Figure 4. As the calls to the GPU kernels are asynchronous to the CPU, it can continue issuing kernel calls while the GPU is working, i.e., the kernel calls are pipelined by the GPU driver.

3.2 Triangular update and index swizzle

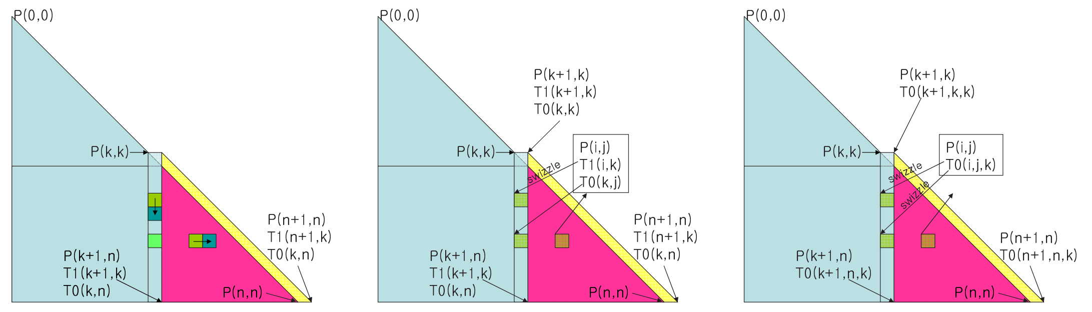
Since BrookGPU doesn't support designating a triangular domain as an output target, a column-wise multipass update is used to mimic triangular update, but this approach cannot fully utilize the parallel architecture because each pass has several processors unused. However, singlepass update over a triangular domain can be implemented with OpenGL API as seen in Figure 5. An oversized triangle, the

Algorithm 2 Cholesky decomposition on the GPU - Outer product form

```
// Note that the 2 dimensional stream or texture's coordinates start from (0,0)
Create two stream S and T of size equal to A.
Copy A to S.

for $k = 0...n-2$, do
  Bind T as the target.
  Run square root kernel on $(k, k)$.

  Bind S as the target.
  Run copy kernel to copy the square rooted result on the diagonal of T back to S.

  Bind T as the target.
  Run normalization kernel over the domain covering the $k^{th}$ column below the diagonal, from $(k+1, k)$
  to $(n-1, k)$.

  Bind S as the target.
  Run copy kernel to copy the normalized column of T back to S.

  Bind T as the target.
  Run submatrix update kernel over the domain covering submatrix $(k+1, k+1)$ to $(n-1, n-1)$.

  Swap S and T, i.e., swap the pointers to texture ID of S and T.
end for

Bind T as the target.
Run square root kernel on $(n-1, n-1)$.

// Now the lower triangular of T is the result of Cholesky decomposition
```

vertices of which are at $(k+1,k)$, $(k+1,n)$, and $(n+1,n)$, is drawn to cover the lower triangular matrix.

Two sets of texture coordinates are assigned to the vertices to generate indexes for the active pivot and the active neighbor. In fact, the index pairs passed to the fragment processor are same as in the LU algorithm [GGHM05]. However, the index pair for the pivot is different, since our Cholesky decomposition doesn't update the upper triangular part. The index pair needs to be swapped, which can be handled by a swizzle operator at no cost. The swizzle operator reorders the coordinates of a multidimensional variable. For example, $a.xxyz$ produces a 4 dimensional value $(a.x, a.x, a.y, a.z)$ and $b.yx$ reverses the order of the coordinates of the 2 dimensional value b. This technique reduces the number of instructions from 6 to 4. In addition, it is observed that the two index pairs share an invariant index k. Thus, it can be put at a z coordinate which is invariant to the x and y coordinates. Thus the pairs can be packed into, interpolated from, and restored (using swizzle) from a single 3D texture coordinate. This reduces the rasterizer's work [KW05].



(a) The pivot moves vertically, when the active element moves horizontally.

(b) The interpolated index pair for the pivot is swizzled to compensate the different movement.

(c) Only one interpolated index triad is needed if a 3D texture coordinate is applied to each vertex.

Fig. 5: kth iteration of Cholesky decomposition: An oversized triangle is drawn to cover sub-lower triangular part of the matrix. Texture coordinates T_0 (and T_1) are assigned to the vertices $P(k+1,k)$, $P(k+1,n)$, and $P(n+1,n)$ to get the indexes for the active pivot and neighbor elements. The interpolated index is then swizzled to point the entries at $P(k,i)$ and $P(k,j)$. The swizzle operator handles this at the same time as it fetches the texture. No temporary register is necessary to form the swizzled indexes.

4 Interior-Point Method on a GPU

4.1 Linear Programming

Linear programming is the problem of optimizing a linear objective function subject to satisfying a set of linear constraints, either equalities or inequalities. The standard form is

$$\min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} \tag{4.1}$$

$$s.t. \quad \boldsymbol{A}\mathbf{x} = \mathbf{b}, \tag{4.2}$$

$$\mathbf{x} \geq \mathbf{0}, \tag{4.3}$$

where $\mathbf{c}$ and $\mathbf{x}$ are real vectors of size n, $\mathbf{b}$ is a real vector of size m, and $\boldsymbol{A}$ is an $m \times n$ real matrix with full row rank, with $m \leq n$. Any inequalities can be transformed to equalities by introducing slack variables.

A dual is always associated with the primal (4.1)-(4.3). It uses the same data but different variables:

$$\max_{\lambda} \mathbf{b}^T \lambda \tag{4.4}$$

$$s.t. \boldsymbol{A}^T \lambda + \mathbf{s} = \mathbf{c}, \tag{4.5}$$

$$\mathbf{s} \geq \mathbf{0}, \tag{4.6}$$

where λ and $\mathbf{s}$ are real vectors of size m and n, respectively.

4.2 Primal-Dual Interior-Point Method

The Primal-dual interior-point method is one of the most widely used algorithms to solve the linear programming problem (4.1)-(4.3) [Wri97]. Solving the linear programming problem is equivalent to finding a solution to the KKT (Karush-Kuhn-Tucker) conditions:

$$\boldsymbol{A}^T \lambda + \mathbf{s} = \mathbf{c}, \tag{4.7}$$

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}, \tag{4.8}$$

$$x_i s_i = 0, \;\; i = 1, 2, ..., n, \tag{4.9}$$

$$\mathbf{x} \geq \mathbf{0}, \;\; \mathbf{s} \geq \mathbf{0}. \tag{4.10}$$

To solve the KKT conditions, a modified Newton's method is used. The search direction at each iteration is obtained by solving either the perturbed KKT conditions,

$$\begin{bmatrix} \boldsymbol{0} & \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{A}^T & \boldsymbol{0} & \boldsymbol{I} \\ \boldsymbol{0} & \boldsymbol{S} & \boldsymbol{X} \end{bmatrix} \begin{bmatrix} \Delta\lambda \\ \Delta\mathbf{x} \\ \Delta\mathbf{s} \end{bmatrix} = \begin{bmatrix} \mathbf{r}_b \\ \mathbf{r}_c \\ -\mathbf{r}_{xs} \end{bmatrix}, \tag{4.11}$$

or, equivalently, the normal equations,

$$\boldsymbol{A}\boldsymbol{D}^2\boldsymbol{A}^T \Delta\lambda = \mathbf{r}_b + \boldsymbol{A}(\boldsymbol{S}^{-1}\boldsymbol{X}\mathbf{r}_c + \boldsymbol{S}^{-1}\mathbf{r}_{xs}), \tag{4.12}$$

$$\Delta\mathbf{s} = \mathbf{r}_c - \boldsymbol{A}^T \Delta\lambda, \tag{4.13}$$

$$\Delta\mathbf{x} = -\boldsymbol{S}^{-1}(\mathbf{r}_{xs} + \boldsymbol{X}\Delta\mathbf{s}), \tag{4.14}$$

where $\boldsymbol{D}^2 = \boldsymbol{S}^{-1}\boldsymbol{X}$; $\mathbf{r}_b = \mathbf{b} - \boldsymbol{A}\mathbf{x}$; $\mathbf{r}_c = \mathbf{c} - \boldsymbol{A}^T - \mathbf{s}$; $\mathbf{e} = (1, ..., 1)^T$; and $\mathbf{r}_{xs} = \boldsymbol{X}\boldsymbol{S}\mathbf{e}$ and $\mathbf{r}_{xs} = -\sigma\mu\mathbf{e} + \Delta\boldsymbol{X}^{\mathbf{aff}}\Delta\boldsymbol{S}^{\mathbf{aff}}\mathbf{e}$ for predictor and corrector, respectively [Wri97]. The affine-scaling predictor direction is the pure Newton direction for (4.7)-(4.9). The corrector step attempts to reduce the error caused by the linear approximation of the pure Newton direction. Here σ is a *centering parameter*, $\mu = \mathbf{x}^T\mathbf{s}/n$ is the complementarity measure, and $\boldsymbol{X}$ and $\boldsymbol{S}$ are matrices with, respectively vector $\mathbf{x}$ and $\mathbf{s}$ on their diagonals.

Usually solving (4.12)-(4.14) is preferred, because the matrix for the normal equations is much smaller than that for the KKT system. Moreover the matrix is symmetric and positive definite, and thus Cholesky decomposition, which is faster and numerically more stable than LU, can be used. Here a variant of Mehrotra's predictor-corrector (MPC) algorithm from [Wri97] is used. See Algorithm 3. We make use of the following kernels (where multipass algorithms which call several kernels are also referred as kernels for brevity):

- Level 1 - $O(n)$
  - inner product: This uses two kernels, the element-wise vector-vector multiplication and the sum reduction kernel. Multipass.
  - inf_norm: *abs-max* reduction kernel. Multipass.
  - smin: *min* reduction kernel. Multipass.
  - smax: *max* reduction kernel. Multipass.
  - saxpy and its variants: Scalar vector multiplication followed by a vector addition. The variants include element-wise vector-vector multiplication. Singlepass.
  - line search kernels: Specialized kernels to determine step length for the predictor and the corrector. These kernels are used with reduction kernel to compute the step length. Singlepass.
- Level 2 - $O(n^2)$
  - sgemv and its variants: Matrix vector multiplication with optional vector additions. Singlepass(inner product) or multipass(outer product).
  - forward and backward substitution: These kernels solve a system of linear equation where the matrix is lower or upper triangular. Multipass.
- Level 3 - $O(n^3)$
  - aat and adat: Matrix-matrix multiplications to form the matrix of the normal equations. Singlepass(inner product) or multipass(outer product).
  - chol: Cholesky decomposition. Multipass.

Having feedback from the GPU is inevitable as the termination criteria must be determined. This blocks the kernel calls, so the CPU has to wait until the GPU completes all the computations. Calling some of the kernels for solving (4.12)-(4.14) prior to the determination can help the algorithm perform a little better, because they don't depend on the feedback and the CPU can compute the termination criteria while the GPU is working on the search direction. The GPU driver must empty its queue for kernel calls because of the current driver's limitation. Once this limitation is resolved, we can get even more speedup by pre-computing the predictor direction while deciding whether to terminate, because the computation doesn't affect the current point.

5 Analysis

In this section we consider the complexity of the algorithms we have discussed.

Algorithm 3 Mehrotra's Predictor-Corrector Algorithm

```
Generate an initial point $(\mathbf{x}^0, \lambda^0, \mathbf{s}^0)$
for k = 0,1,2,... do
  Get primal and dual objective values using inner product kernels.
  Get $\mathbf{r}_c$ and $\mathbf{r}_b$ using sgemv kernels.
  Get norm of $\mathbf{r}_c$ and $\mathbf{r}_b$ using the infinity norm kernel.
  Transfer the primal and dual objective values and the norm of residuals from GPU memory to CPU memory.

  Terminate if the relative duality gap and the residual norms are sufficiently small (6.1)-(6.3).

  Solve (4.12)-(4.14) using level 1, 2 and 3 kernels (including adat; Cholesky; and forward and backward substitution) to obtain $(\Delta\mathbf{x}^{\mathtt{aff}}, \Delta\lambda^{\mathtt{aff}}, \Delta\mathbf{s}^{\mathtt{aff}})$.

  Determine the predictor step length using level 1 kernels (line search and reductions):
    $\alpha_{\text{aff}}^{\text{pri}} = \arg\max_{\alpha\in[0,1]} \{\mathbf{x}^k + \alpha\Delta\mathbf{x}^{\text{aff}} \geq 0\}$
    $\alpha_{\text{aff}}^{\text{dual}} = \arg\max_{\alpha\in[0,1]} \{\mathbf{s}^k + \alpha\Delta\mathbf{s}^{\text{aff}}) \geq 0\}.$

  Determine the centering parameter using level 1 kernels:
    $\sigma = (\mu_{\mathtt{aff}}/\mu)^3,$ where $\mu_{\mathtt{aff}} = (\mathbf{x}^k + \alpha_{\text{aff}}^{\text{pri}}\Delta\mathbf{x}^{\mathtt{aff}})^T(\mathbf{s}^k + \alpha_{\text{aff}}^{\text{dual}}\Delta\mathbf{s}^{\mathtt{aff}})/n.$

  Solve (4.12)-(4.14) using level 1 and 2 kernels (including forward and backward substitution) to obtain $(\Delta\mathbf{x}^{\text{cc}}, \Delta\lambda^{\text{cc}}, \Delta\mathbf{s}^{\text{cc}})$.

  Compute the search direction $(\Delta\mathbf{x}^{\text{k}}, \Delta\lambda^{\text{k}}, \Delta\mathbf{s}^{\text{k}})$ by adding $(\Delta\mathbf{x}^{\mathtt{aff}}, \Delta\lambda^{\mathtt{aff}}, \Delta\mathbf{s}^{\mathtt{aff}})$ and $(\Delta\mathbf{x}^{\text{cc}}, \Delta\lambda^{\text{cc}}, \Delta\mathbf{s}^{\text{cc}})$ using level 1 kernels.

  Determine step size parameters, $\alpha_k^{pri}$ and $\alpha_k^{dual}$ using level 1 kernels (line search and reductions):
    $\alpha_{\max}^{\text{pri}} = \arg\max_{\alpha\in[0,1]} \{\mathbf{x}^k + \alpha\Delta\mathbf{x}^k \geq 0\}$   (4.15)
    $\alpha_{\max}^{\text{dual}} = \arg\max_{\alpha\in[0,1]} \{\mathbf{s}^k + \alpha\Delta\mathbf{s}^k \geq 0\}.$   (4.16)

  Select $\alpha^{\text{pri}} \in [0, \alpha_{\max}^{\text{pri}})$ and $\alpha^{\text{dual}} \in [0, \alpha_{\max}^{\text{dual}})$ according to Mehrotra's heuristic [Wri97] using level 1 kernels.

  Set $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k^{pri}\Delta\mathbf{x}^k$, $(\lambda^{k+1}, \mathbf{s}^{k+1}) = (\lambda^k, \mathbf{s}^k) + \alpha_k^{dual}(\Delta\lambda^k, \Delta\mathbf{s}^k)$ using level 1 kernels.
end for
```

LU [GGHM05]			
Kernel	# of Frags	bytes/frag	Bytes
Copy	$\frac{n(n+1)}{2}$	8	$4n(n+1)$
Normalization	$\frac{n(n+1)}{2}$	12	$6n(n+1)$
Copyback	$\frac{n(n+1)}{2}$	8	$4n(n+1)$
Elimination	$\frac{(n-1)n(2n-1)}{6}$ $\approx \frac{n^3}{3}$	16	$\approx 16\frac{n^3}{3}$
Total			$\approx \frac{16}{3}n^3$

(a) LU decomposition.

Cholesky (inner)			
Kernel	# of Frags	bytes/frag	Bytes
Square root	n	8	$8n$
Copy	$n-1$	8	$8(n-1)$
Normalization	$\frac{(n-1)n}{2}$	12	$6(n-1)n$
Innerproduct subtraction	$n-k$ at kth itr	$4(2k+2)$ at kth itr	$\approx 8\frac{n^3}{3}$
Total			$\approx \frac{8}{3}n^3$

(b) Cholesky decomposition (inner product).

Cholesky (outer/singlepass/square)			
Kernel	# of Frags	bytes/frag	Bytes
Square root	n	8	$8n$
Copy	$n-1$	8	$8(n-1)$
Normalization	$\frac{(n-1)n}{2}$	12	$6(n-1)n$
Copyback	$\frac{(n-1)n}{2}$	8	$4(n-1)n$
Submatrix Update	$\frac{(n-1)n(2n-1)}{6}$ $\approx \frac{n^3}{3}$	16	$\approx 16\frac{n^3}{3}$
Total			$\approx \frac{16}{3}n^3$

(c) Cholesky decomposition (outer product via singlepass-update over square domain).

Cholesky (outer/singlepass/triangular)			
Kernel	# of Frags	bytes/frag	Bytes
Square root	n	8	$8n$
Copy	$n-1$	8	$8(n-1)$
Normalization	$\frac{(n-1)n}{2}$	12	$6(n-1)n$
Copyback	$\frac{(n-1)n}{2}$	8	$4(n-1)n$
Submatrix Update	$\frac{(n-1)n(n+1)}{6}$ $\approx \frac{n^3}{6}$	16	$\approx 16\frac{n^3}{6}$
Total			$\approx \frac{8}{3}n^3$

(d) Cholesky decomposition (outer product via singlepass-update over lower triangular domain). This cannot be implemented in BrookGPU.

Tab. 1: Memory access (reading/writing) counts. A table for the outer product Cholesky decomposition via multipass triangular update is omitted as it's the same as the single pass triangular update version.

5.1 Cholesky decomposition

The square root kernel requires only one *sqrt* instruction with one texture fetch. The normalization kernel requires only one *div* instruction with two texture fetches. The submatrix update kernel requires exactly one *mad* operation with three texture fetches. The inner product subtraction kernel requires k mad instructions with $k+1$ texture fetches, and a dynamic loop to iterate k times. All kernels need one writing operation. Memory access counts and computational complexities of each algorithm are summarized in Tables 1 and 2.

The inner product implementation doesn't employ a cache aware algorithm [JH03] due to the cache inefficiency in texture access [FSH04]. The outer product implementation is cache ignorant [GGHM05], which allows the algorithm to achieve maximum bandwidth.

5.2 Interior Point Method

The problem we are interested in is dense linear programming problem, i.e., we assume that $\boldsymbol{A}$ is dense. At each iteration, most computational resources are consumed in solving the normal equations (4.12). To do this, we form and factor a symmetric positive definite matrix at each iteration. Approximate computational costs for the two operations are as follows:

LU [GGHM05]			
Kernel	# of Frags	Op/frag	Ops
Normalization	$\frac{n(n+1)}{2}$	1	$\frac{n(n+1)}{2}$
Elimination	$\frac{(n-1)n(2n-1)}{6}$ $\approx \frac{n^3}{3}$	1	$\approx \frac{n^3}{3}$
Total			$\approx \frac{n^3}{3}$

(a) LU decomposition.

Cholesky (inner)			
Kernel	# of Frags	Op/frag	Ops
Square root	n	1	n
Normalization	$\frac{(n-1)n}{2}$	1	$\frac{(n-1)n}{2}$
Innerproduct subtraction	$(n-k)$/itr.	k/itr.	$\approx \frac{n^3}{6}$
Total			$\approx \frac{n^3}{6}$

(b) Cholesky decomposition (inner product).

Cholesky (outer/singlepass/square)			
Kernel	# of Frags	Op/frag	Ops
Square root	n	1	n
Normalization	$\frac{(n-1)n}{2}$	1	$\frac{(n-1)n}{2}$
Submatrix Update	$\frac{(n-1)n(2n-1)}{6}$ $\approx \frac{n^3}{3}$	1	$\approx \frac{n^3}{3}$
Total			$\approx \frac{n^3}{3}$

(c) Cholesky decomposition (outer product via singlepass-update over square domain).

Cholesky (outer/singlepass/triangular)			
Kernel	# of Frags	Op/frag	Ops
Square root	n	1	n
Normalization	$\frac{(n-1)n}{2}$	1	$\frac{(n-1)n}{2}$
Submatrix Update	$\frac{(n-1)n(n+1)}{6}$ $\approx \frac{n^3}{6}$	1	$\approx \frac{n^3}{6}$
Total			$\approx \frac{n^3}{6}$

(d) Cholesky decomposition (outer product via singlepass-update over lower triangular domain). This is not implementable with BrookGPU.

Tab. 2: Arithmetic operation(multiplication/division/sqrt) counts

- Forming the matrix : $\frac{m^2 n}{2}$,
- Factoring : $\frac{m^3}{6}$.

In BrookGPU, the actual costs are approximately $m^2 n$ for the matrix formation and $\frac{m^3}{3}$ for the decomposition, because BrookGPU doesn't support triangular domain computation. Usually the problems in this class have many more variables than constraints(i.e., $n \gg m$) which implies that forming the matrix is more expensive than the decomposition.

6 Results

Our algorithms were tested in the following environment:

- NVIDIA GeForce 6800
  - 12 fragment processors
  - 325 MHz core clock cycle
  - 600 MHz memory clock cycle
  - 256 MB DDR memory
- Intel Pentium IV 2.8GHz
  - Hyper Threading
  - 16 KB L1
  - 1 MB L2

  - 512 MB DDR2 dual channel
  - 400 MHz effective memory clock cycle
- Operating system
  - Windows XP with Microsoft VC7 compiler, or
  - Linux Fedora Core 4 with gcc3 compiler

6.1 Cholesky Decomposition

The time spent to transfer data between CPU and GPU is not considered, because our application will form the matrix using GPU kernels and will solve the system using the data stored in the memory of the graphics hardware.

6.1.1 Overhead of copy to texture

Figure 6 presents timings for BrookGPU implementations. The outer product Cholesky algorithm is outperformed by the inner product version when measured on Linux due to the copy-to-texture (CTT) operation, in which the computed results are initially saved to the framebuffer and then copied to the designated texture. BrookGPU uses the old p-buffer, and does not support the render-to-texture(RTT) in Linux [Lef04]. Thus the outer product algorithm requires approximately $(n-k)^2$ more copy operations at each iteration.

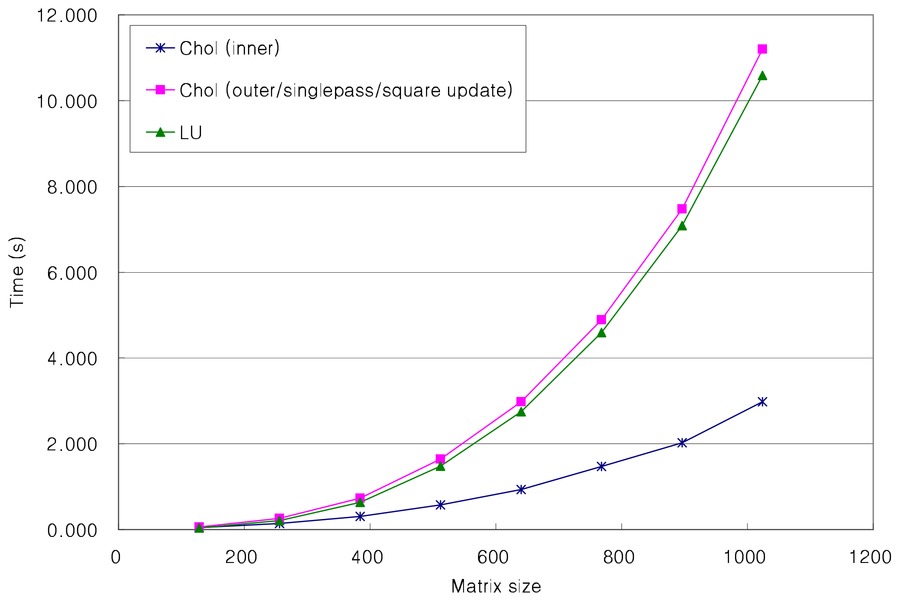


Fig. 6: BrookGPU implementations measured on Linux with OpenGL backend. Average execution time of Cholesky algorithms via inner product and outer product (singlepass square update), and LU decomposition.

6.1.2 Dynamic loop and computed index pairs

In constrast, the outer product Cholesky algorithm outperforms the inner product version on Windows XP, which supports RTT for p-buffers as seen in Figure 7. The inner product subtraction kernel needs the index pairs to be gathered from two separate variables in the loop, which obviously increases the instruction counts. Despite the cache-friendly sequential access pattern, fetching textures in a dynamic loop doesn't benefit from the pattern because GPU architectures are suited to typical graphics applications which usually perform many arithmetic operations per memory access [FSH04].

The inner product Cholesky is less parallel than the outer version because it uses a dynamic loop, and fewer fragments are processed in the inner product subtraction kernel than in the submatrix update kernel. When the number of fragments to be processed is not a multiple of the number of fragment processors, only a few fragment processors are active to serve the dynamic loop at the end. Especially in the later iterations, this is quite expensive. For instance, a single processor works to repeat 1023 times at the last iteration of decomposing a matrix of dimension 1024×1024.

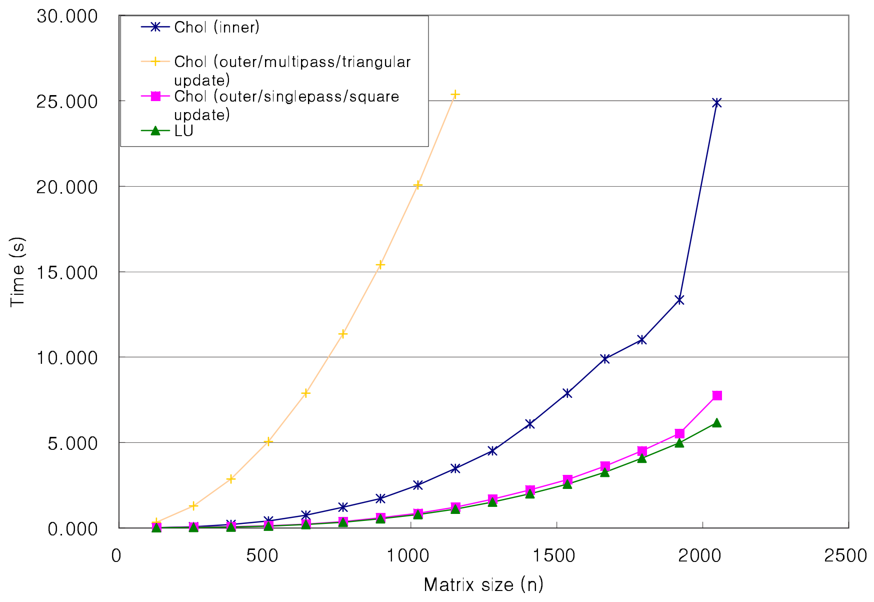


Fig. 7: BrookGPU implementation measured on Windows XP with OpenGL backend. Average execution time of Cholesky algorithms via inner product and outer product (singlepass square update), and LU decomposition.

6.1.3 Speedup by triangular update, index interpolation, and swizzle

The outer product Cholesky algorithm implemented with BrookGPU cannot be faster than LU, because it cannot use the triangular update to take advantage of the symmetric structure. Figure 8 shows that Cholesky with triangular update takes about half the time of LU using OpenGL API.

Computing an index pair for the pivot needs 2 more instructions, whereas swizzling an interpolated index pair doesn't increase the instruction count because swizzling and fetching can be processed simultaneously. Using interpolated and swizzled indexes performs 35% better than using a computed pivot index pair gathered from the interpolated indexes of the active neighbor and element. As mentioned in Section 3.2, we used index triads to reduce the rasterizer's work. Since the rasterizer is not the bottleneck, no noticeable difference is observed between the pair and the triad implementations.

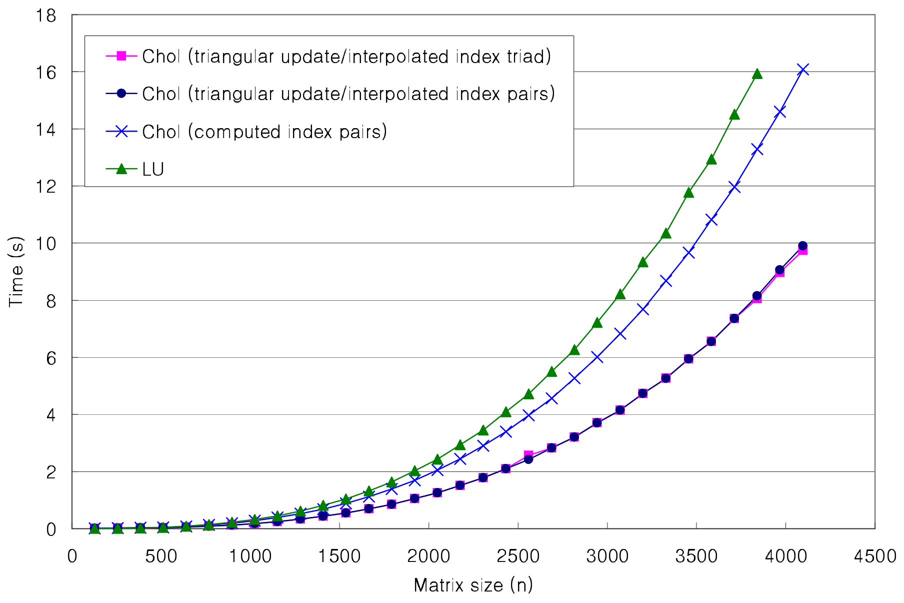


Fig. 8: OpenGL API implementation. Execution time of Cholesky and LU decomposition [GGHM05] measured on Windows XP.

6.1.4 Square root kernel and bandwidth usage

In the square root kernel, only one fragment is shaded so there is no parallelism. Therefore, the Cholesky decomposition (outer product with triangular update) shows less bandwidth usage than LU as seen in Figure 9. When the matrix is sufficiently large, however, Cholesky uses as much bandwidth as LU does. This means that our Cholesky algorithm is also bandwidth-bounded [GGHM05].

6.2 Primal-Dual Interior Point Method

6.2.1 Decomposition and formation

As seen in Figure 10, if the linear programming problem has two times more variables than constraints, then forming the matrix is approximately 6 times as expensive as decomposing it ($m^2n = 2m^3 = 6{\times}m^3/3$) when only rectangular domains can be updated.

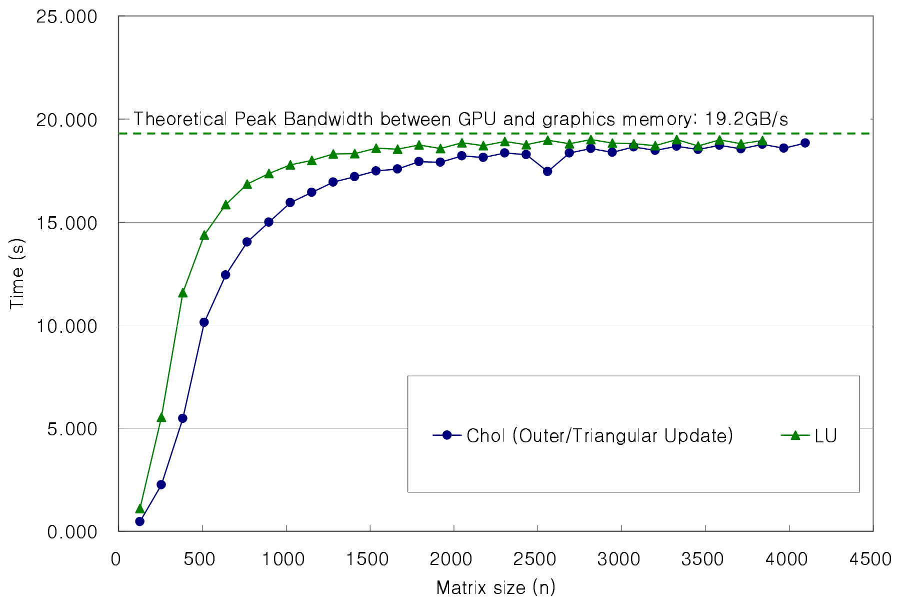


Fig. 9: OpenGL API implementation. Memory bandwidth utilized by Cholesky decomposition (with outer product form with singlepass triangular update) and LU decomposition.

6.2.2 Problem solving

We used termination criteria from [Wri97] stopping when the relative residuals and duality measure are sufficiently small:

$$\frac{\|\mathbf{r}_b\|_\infty}{1+\|\mathbf{b}\|_\infty} \leq \epsilon, \tag{6.1}$$

$$\frac{\|\mathbf{r}_c\|_\infty}{1+\|\mathbf{c}\|_\infty} \leq \epsilon, \tag{6.2}$$

$$\frac{|\mathbf{c}^T\mathbf{x}-\mathbf{b}^T\mathbf{y}|}{1+|\mathbf{c}^T\mathbf{x}|} \leq \epsilon, \tag{6.3}$$

where the termination tolerance ϵ is a small positive number. Usually 10^{-8} is chosen when double precision is used, but it's not easily achievable with single precision. For that reason we used problem specific tolerances. The initial starting point, $(\mathbf{x}, \lambda, \mathbf{s})$, was chosen using the heuristic in [Meh92]. We compared our implementation with a MATLAB version implementing exactly the same algorithm using the dense constraint matrix with single precision. The results are shown in Table 3.

7 Conclusions

We have presented several Cholesky implementations for dense symmetric and positive definite matrices. In the interior point method, matrix formation and decomposition are most avaricious of computational resources. Thus, support for the triangular output domain is essential in order to exploit the symmetric

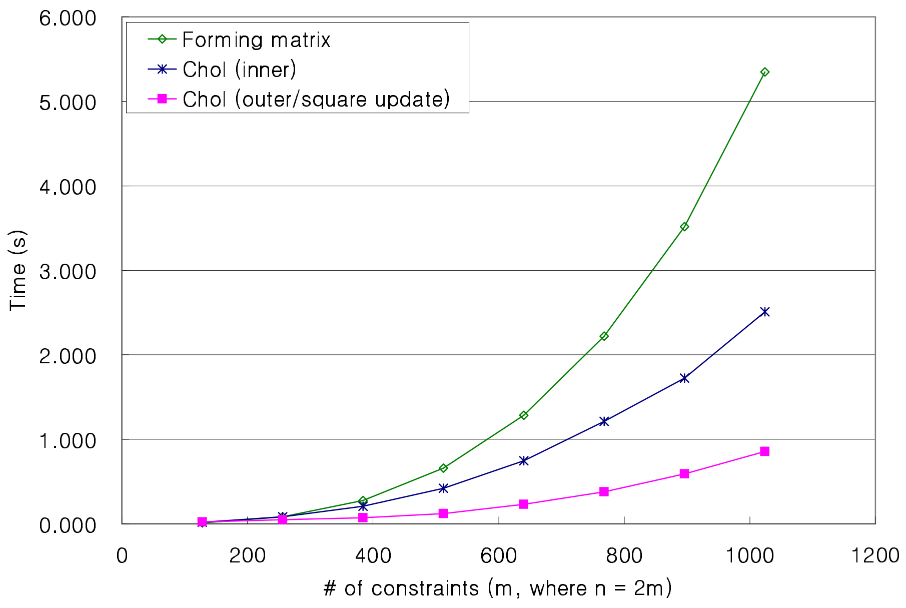


Fig. 10: BrookGPU implementations measured on Windows XP with OpenGL backend. Forming the matrix for the normal equations is usually more expensive than decomposing the matrix. Triangular domain support is essential in achieving speedup in the matrix formation.

structure. In our experiment, Cholesky decomposition can achieve its full performance gain over LU only when the triangular update technique is deployed.

Rendering a triangle is truly a native feature of GPUs. Developers of streaming languages targeting GPUs should seriously consider incorporating the feature in their model. When the feature is enabled, our algorithm for forming the symmetric matrix also uses it.

Languages designed for graphics hardware, including Cg, GLSL, and HLSL, are not adequate for general purpose computations for those who are not familiar with graphics APIs. Specialized languages including BrookGPU, Accelerator [TPO05], and others that hide those peculiarities mitigate the difficulties of using the hardware. Our implementation of the interior point method would not be possible without the aid of such specialized languages.

Due to lack of double precision support, our interior point algorithm cannot converge as closely to an optimal point as a double precision implementation. Using GPUs in the early iterations and CPUs in the later iterations can be helpful because the systems of early iterations are usually not ill-conditioned. Recent study on improving performance while maintaining the double precision accuracy via a combined CPU-GPU solver on a finite element method (FEM) [GST05] could be extended to the interior point method on a combined CPU-GPU system.

Problem	Size	ϵ	GPU Iterations	GPU Time (s) Inner Cholesky	GPU Time (s) Outer Cholesky	MATLAB(MPC) Iterations	MATLAB(MPC) Time (s)
adlittle	57×97	3×10^{-5}	12	0.60	0.62	9	0.10
afiro	28×32	4×10^{-5}	7	0.20	0.20	7	0.03
agg2	517×302	5×10^{-5}	18	22.77	17.14	17	5.02
agg3	517×302	6×10^{-4}	20	25.30	19.02	17	5.05
bandm	306×472	2×10^{-3}	12	5.22	4.45	12	1.12
beaconfd	174×262	3×10^{-4}	7	1.23	1.18	6	0.21
blend	75×83	2×10^{-3}	11	0.70	0.88	8	0.08
e226	224×282	9×10^{-4}	18	4.94	4.39	16	1.19
sc50b	51×48	3×10^{-5}	6	0.28	0.29	6	0.04
sctap1	301×480	5×10^{-4}	15	7.00	6.02	13	1.88

Tab. 3: Average running time of Interior Point Methods on NETLIB problems. The BrookGPU implementation was measured on Windows XP with OpenGL backend. Cholesky decomposition in inner or outer product form is used to solve the normal equation (4.12). Note that the outer product version uses the square update.

8 Acknowledgements

We would like to thank Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha for disclosing their LU implementation. Thanks to the easy stream programming model of BrookGPU, we were able to implement the interior point method running on a GPU without being bothered by the messy graphics APIs.

References

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for GPUs: Stream Computing on Graphics Hardware, 2004.

[FQKYS04] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[FSH04] K. Fatahalian, J. Sugerman, and 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*, pages 133–137, New York, NY, USA, 2004. ACM Press.

[GGHM05] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh 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*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

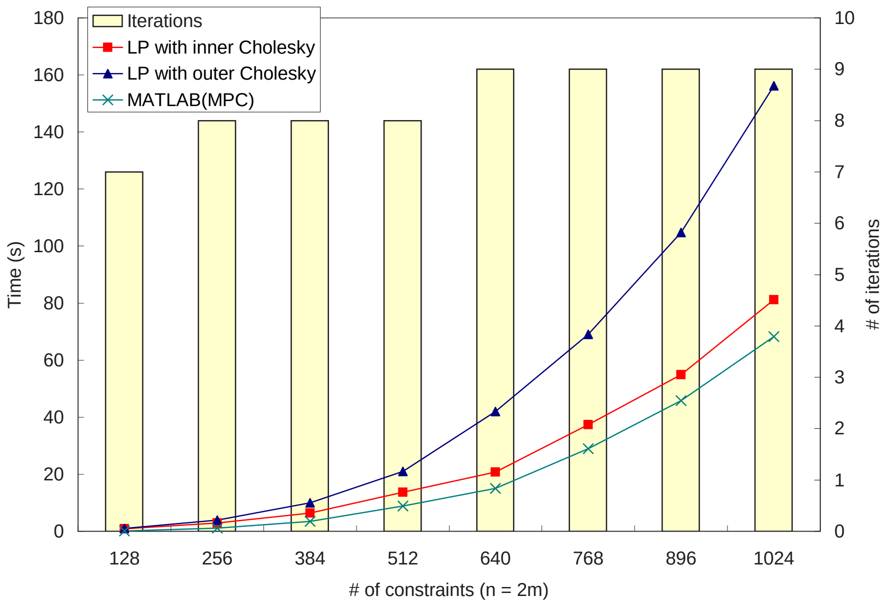


Fig. 11: Brook implementations and MATLAB implementation measured on Linux with OpenGL backend: Average running time of interior point method on random problems.

[GL96] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[GST05] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *Simulationstechnique 18th Symposium in Erlangen, September 2005*, volume Frontiers in Simulation, pages 139–144. SCS Publishing House e.V., 2005. ASIM 2005.

[Har04] Mark J. Harris. GPGPU: Beyond Graphics. In *Programming Graphics Hardware*. Eurographics Tutorial, August 2004.

[JH03] J. Hart. J. Hall, N. Carr. Cache and Bandwidth Aware Matrix Multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, 2003.

[KW03] Jens Krueger and Ruediger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[KW05] Jens Krüger and Rüdiger Westermann. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[Lef04] Aaron Lefohn. GPU Data Formatting and Addressing. In *SIGGRAPH 2004 GPGPU Course*. SIGGRAPH, August 2004.

[Lue05] David Luebke. General-Purpose Computation on Graphics Hardware. SIGGRAPH 2005 GPGPU Course, August 2005.

[MA03] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Mar]

[Meh92] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, November 1992.

[MIC03] MICROSOFT CORP. High-level shader language., 2003.

[MIC05] MICROSOFT CORP. DirectX 9.0 graphics., 2005.

[OLG$^+$05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[Ros04] Randi J. Rost. *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[TPO05] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism. Technical Report MSR-TR-2005-184, December 2005.

[WNDS05] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[Wri97] Stephen J. Wright. *Primal-dual interior-point methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[WWF05] Man-Leung Wong, Tien-Tsin Wong, and Ka-Ling Fok. Parallel Evolutionary Algorithms on Graphics Processing Unit. In *The 2005 IEEE Congress on Evolutionary Computation*, pages 2286– 2293, 2005.