Electronic Transactions on Numerical Analysis.
Volume 28, pp. 174-189, 2008.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

IMPLEMENTING AN INTERIOR POINT METHOD FOR LINEAR PROGRAMS ON A CPU-GPU SYSTEM*

JIN HYUK JUNG† AND DIANNE P. O'LEARY‡

In memory of Gene Golub

Abstract. Graphics processing units (GPUs), present in every laptop and desktop computer, are potentially powerful computational engines for solving numerical problems. We present a mixed precision CPU-GPU algorithm for solving linear programming problems using interior point methods. This algorithm, based on the rectangular-packed matrix storage scheme of Gunnels and Gustavson, uses the GPU for computationally intensive tasks such as matrix assembly, Cholesky factorization, and forward and back substitution. Comparisons with a CPU implementation demonstrate that we can improve performance by using the GPU for sufficiently large problems. Since GPU architectures and programming languages are rapidly evolving, we expect that GPUs will be an increasingly attractive tool for matrix computation in the future.

Key words. GPGPU, Cholesky factorization, matrix decomposition, forward and back substitution, linear programming, interior point method, rectangular packed format

AMS subject classifications. 90C05, 90C51, 15A23, 68W10

1. Introduction.

Hidden inside your desktop or laptop computer is a very powerful parallel processor, the graphics processing unit (GPU). This hardware is dedicated to rendering images on your screen, and its design was driven by the demands of the gaming industry. This single-instruction-multiple-data (SIMD) processor has its own memory, and the host CPU issues instructions and data to it through a data bus such as PCIe (Peripheral Component Interconnect Express). A typical GPU is found in a graphics card in a peripheral expansion slot, or perhaps integrated into the memory controller hub, also known as the north-bridge, which controls high-speed devices; see [7] for more detail. ATI's Radeon and NVIDIA's GeForce series, the dominant products in the market, offer inexpensive but very powerful GPUs.

Originally, GPUs were much slower than CPUs and had very limited programmability. Now they show superior performance on some applications, and their speed is increasing at a rate faster than Moore's law predictions for CPUs [11]. For example, NVIDIA's graphics hardware GeForce 7800 GTX shows sustained performance of 165 GFLOPS (300 GFLOPS at peak) compared to a 24.6 GFLOPS theoretical peak for a 3GHz Intel Pentium D (dual-core processor) [10]. Originally, GPUs worked in half-precision or less, but recent support for single precision floating point numbers and potentially double precision makes them much more attractive for numerical computation. In addition, newer GPUs have the capacity to store longer programs, making complicated algorithms possible. Researchers have applied GPUs to general computations including evolutionary algorithms [27], fluid dynamics [3], FFT [18], and others [22].

Recently GPUs have been used for linear algebra [9], including programs for matrix multiplication [6], an iterative sparse system solver [1], a direct dense system solver [4], and others [22]. Our work to implement a direct solver for normal equations [8] is an extension of those efforts. Parallel Cholesky factorization for sparse matrices on shared memory multiprocessors was considered by Ng and Peyton [19]. Such methods requires full scatter

*Received March 2, 2007. Accepted for publication January 10, 2008. Recommended by M. Overton. 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, USA (jjung@cs.umd.edu).
‡Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA (oleary@cs.umd.edu).

ETNA
Kent State University
etna@mcs.kent.edu

operation, saving a computational result to a desired location. In addition it requires support for threads and synchronization among threads. These features had not been supported until the GeForce 8 Series and CUDA (Compute Unified Device Architecture) were recently released [21].

In this paper we consider how use of the GPU can improve the performance of interior point methods (IPMs) for solving linear programming problems. We begin in section 2 with a brief overview of GPU architecture and programming. Section 3 presents the linear programming problem and the IPM and discusses how the work can be partitioned between the CPU and the GPU. Timing results are presented in section 4 and conclusions in section 5.

2. GPU hardware and software. In this section we briefly describe the architecture and programming of GPUs, concluding with an example of how two matrices might be added.

2.1. GPU architecture. A functional block diagram of a GPU (GeForce 6 and 7 Series) is presented in Figure 2.1[1]. The purpose of the GPU is rendering realistic two- or three-dimensional scenes on two-dimensional displays. A scene is assembled from streams of vertices that specify polygons. The *vertex processors* manipulate each vertex depending on its attributes, which include positions, colors, and normal vectors. Polygons are then tessellated into triangles. Since current displays are two-dimensional and cannot directly show vector graphics, triangles are projected onto two-dimensional screen space and then transformed or rasterized by the *rasterizer* into *fragments*. To make the scenes realistic, *texture mapping* is performed by *fragment processors*, which color or shade the fragments using *textures* specified by a bitmap pattern. Each fundamental element of a texture is referred to as a *texel*.

A vertex in three-dimensions is represented as a four-dimensional vector (x, y, z, w) representing homogeneous coordinates in a three-dimensional projective space. Using these coordinates, a three-dimensional affine transformation can be represented by a linear transformation. A pixel's color is also represented as a four-dimensional vector (r, g, b, a) where r, g, b, and a denote red, green, blue, and alpha (opacity), respectively. Both the vertex and fragment processors are capable of processing four-dimensional vectors very efficiently.

A texture is the counterpart of an array on a CPU and can be used to represent vectors and matrices. The texture is frequently referred to as the *stream* in the streaming model perspective. For typical graphics applications, a bitmap is stored in a texture, but, for general computation, numerical values are stored. The outputs or pixels generated by the fragment processors are stored in *frame-buffer* memory which holds scenes to be displayed. Current GPUs are also capable of *render-to-texture* for 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 frame-buffer.

A computational *kernel* or a *GPU fragment program* is a set of GPU instructions which are initiated by a host CPU and applied to every element of a stream of fragments. Every fragment processor runs the same instruction at each cycle, in parallel. In addition, instruction-level parallelism allows up to 4 arithmetic operations to be performed simultaneously in a fragment processor.

Most computations involve a series of kernel calls. A *single-pass* algorithm uses a single rasterization process, while a *multi-pass* algorithm is composed of multiple rasterization processes. A kernel is initiated with a stream of vertices issued by the host CPU. Since the shape of a matrix or a vector is rectangular, kernels for typical linear algebra operations are initiated by drawing a rectangle with four vertices. A kernel processes the entire stream of fragments

[1]Beginning with the GeForce 8 Series, GPUs have unified processors and different stages of the rendering pipeline. The new pipeline stage is very flexible and compatible with the previous version; see [21] for more details.

ETNA
Kent State University
etna@mcs.kent.edu

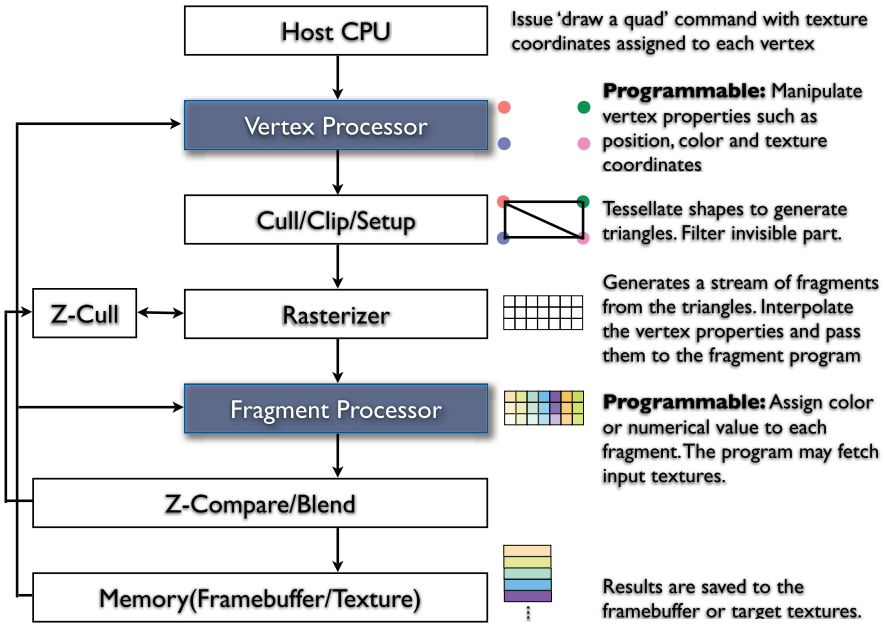


Fig. 2.1: GPU pipeline for NVIDIA GeForce 6 and 7 Series. The vertex and fragment processors are the highly parallel and programmable components in a GPU.

generated from the stream of vertices before a subsequent kernel is initiated. Kernel calls are managed by the GPU driver, so the CPU can compute and issue kernel calls asynchronously.

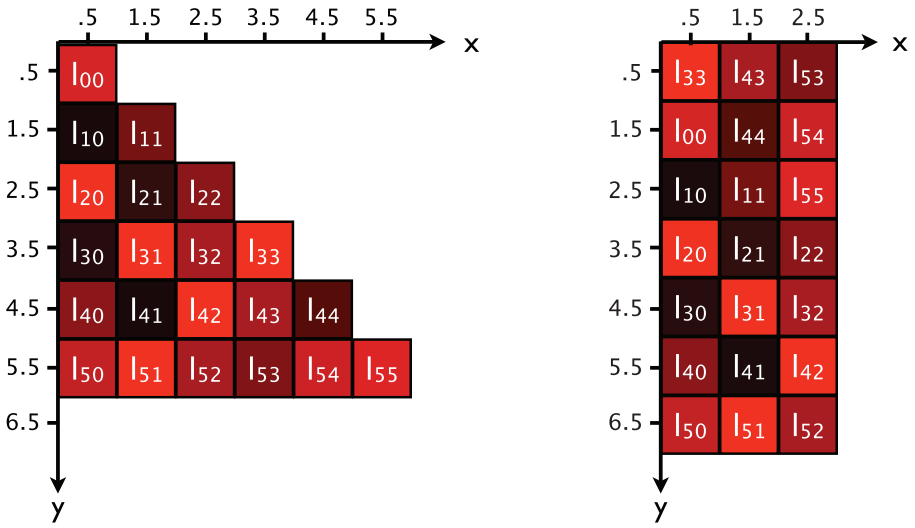
The architecture of the GPU is not much different from that of the ILLIAC IV, a machine from the mid-1970s. This machine had 4 control units (CUs) and 256 processing elements (PEs) [13]. The PEs synchronously executed commands from the CUs. Unlike typical GPUs, up to four PEs could communicate with each other.

A more recent GPU, the GeForce 8800 GTX, has a set of MPs (multiprocessors) each of which has multiple SPs (single processors) [21]. Moreover, each MP supports threaded computing. SPs in a single MP share memory and execute the same instruction at a particular cycle. Different MPs can independently execute different instructions. GPUs are evolving to look more and more like general-purpose parallel machines.

2.2. GPU programming. The core of GPU programming is the kernel. Kernels are written in specialized shading languages such as C for graphics (Cg) [14], high level shader language (HLSL) [16], and OpenGL shading language (GLSL) [24]. Shapes are drawn through a graphics application programming interface (API). Open graphics library (OpenGL) [28] is one of the most widely used APIs in various platforms including Windows and Linux. DirectX [17] is widely used for developing applications for Windows. In our work we use Cg and OpenGL on a GeForce 7 Series GPU.

To make programming easier, Buck et al. introduced BrookGPU [2] which provides abstraction for kernels and simplifies implementation and invocation of kernels. With BrookGPU, drawing a shape is replaced by invoking a kernel just as we would invoke a function written in the C programming language [23]. In addition, BrookGPU offers a convenient invocation of a *parallel reduction* operation such as computing the minimum, maximum or arithmetic sum

ETNA
Kent State University
etna@mcs.kent.edu



(a) Storage of a lower triangular matrix **L** in a texture. When we fetch an element of a texture, we point to its center. For instance, the element l_{42} is stored at $(2.5, 4.5)$.

(b) Storage of a lower triangular matrix **L** in a texture using a packed storage scheme of Gunnels and Gustavson. The 6×6 matrix is stored in a 3×7 texture, with the entries arranged as indicated.

Fig. 2.2: Lower triangular matrices stored in textures, with values stored as intensities of red. The same 6×6 matrices are stored with different format. For the packed format shown in (b), we transpose and move the lower triangular submatrix at the bottom right of (a) to the unused upper left corner. For subsequent figures, we use various colors for better visualization.

of a stream, which abstracts $O(\log n)$ passes of a multi-pass rendering algorithm. Despite those convenient features, we cannot use BrookGPU because it does not support triangular rasterization, which is key to exploiting the structure of symmetric or triangular matrices.

Recently NVIDIA introduced CUDA [21], a development framework for general purpose applications on the GeForce 8 Series[2]. It provides CUBLAS, the BLAS library working on GPUs. CUDA does not support triangular rasterization, which was critical to the performance of the algorithms we discuss below, but spawning multiple threads and having each of them identify its target location could be used to replace triangular rasterization [12].

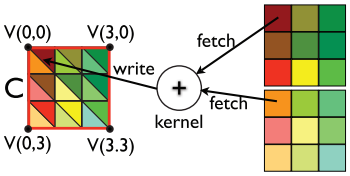
2.3. An example of a GPU algorithm. Given these powerful fragment processors, how might they be used for computational linear algebra? We illustrate the ideas on a simple algorithm, adding two matrices.

We choose to store a matrix as a two-dimensional texture with the numeric values stored as intensities of red.[3] Figures 2.2a and 2.2b illustrate this storage scheme. General matrices are simply arranged with columns along the x-axis and rows along the y-axis as described in Figure 2.2a. Lower triangular or symmetric matrices can be stored in a compact form as

[2] At the time of our development CUDA was not available.

[3] Storing four numerical values as red, green, blue and alpha in a single texel using a four channel texture may increase storage capacity and may improve performance, but we choose the single channel texture for easy implementation. See http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/oldstuff/PerformanceTuning.pdf for further discussion of the trade-offs.

ETNA
Kent State University
etna@mcs.kent.edu



(a) To compute $\mathbf{C} = \mathbf{A} + \mathbf{B}$, the kernel fetches and adds entries from **A** and **B** and stores the result in the target texture **C**. Each entry of **C** in the figure is color-coded to indicate the elements of **A** and **B** that contribute to its value.

```
float main(    uniform samplerRECT A,
               uniform samplerRECT B,
               float2 index : WPOS    ) : COLOR {
    return texRECT(A, index.xy)+texRECT(B, index.xy);
}
```

(b) A kernel for adding two matrices, written in Cg [14].

```
for (x = 0; x < 3; x++)
  for (y = 0; y < 3; y++)
    C[y][x] = A[y][x]+B[y][x];
```

≡ Drawing a square

≡ Addition fragment program

(c) The effect of the kernel is to perform the operations in this nested loop, where the loop indices are specified by specifying the vertices of a square.

Fig. 2.3: Adding two matrices on a GPU.

illustrated in Figure 2.2b. To access an entry in a texture, we use coordinates, just as we use indices to specify an entry of an array in a CPU program. Unfortunately, x-coordinates in a texture correspond to column indices, while y-coordinates indicate row indices, so the index ordering is exactly opposite to that for an array.

As described in Figure 2.1, a kernel is initiated by drawing a shape, usually a quadrilateral. The shape is then transformed to a stream of fragments (of size equal to the number of pixels in the shape) by the rasterizer. Fragments up to the number of processors can be processed simultaneously. The coordinates and position of each fragment are passed to fragment processors as inputs. Then, each fragment processor computes a color or a numerical value for the fragment. Letting the rasterizer divide the shape into fragments is faster than specifying fragments explicitly. The *swizzle operation* is a convenient feature of GPUs; when fetching an entry of a texture, the coordinates of a multidimensional variable can be permuted at no cost. This can be used, for example, to form a matrix transpose, by specifying $b.yx$ instead of b.

A kernel specifies the operation to be performed on each element that it processes, and the elements are specified by vertices passed to the kernel. For example, as depicted in Figure 2.3a, to perform 3×3 matrix-matrix addition, we issue four vertices to specify the texture **C** designated as the target of the rendering. Figure 2.3c gives a CPU-equivalent of the GPU kernel specified in Figure 2.3b. After the vertex processors process "per vertex" operations (nothing in this example), the rasterizer initiates a stream of nine fragments and passes each linearly interpolated vertex property set to a fragment processor. Then the fragment processors run the kernel simultaneously. Each fragment processor fetches and adds values from input textures and stores the result of the addition in the target texture **C**.

In Figure 2.3b, the input parameter `index` specifies the position of the fragment. The attribute `WPOS` indicates that it is an interpolated position. For a matrix entry at the first row and the second column, the interpolated position of the corresponding fragment is $(x, y) = (1.5, 0.5)$. We may use other semantics, `TEXCOORD0` and `TEXCOORD1` for instance, to have the kernel receive other interpolated vertex properties, as explained later. The attribute `COLOR` denotes that the return value of the kernel `main` represents color. The keyword `texRECT` is used for fetching an element of the input texture. The keyword `float2` means the declared variable consists of two single precision values. See [14] for more details of the Cg language.

This introduction to GPUs should be enough to understand the algorithms presented later.

ETNA
Kent State University
etna@mcs.kent.edu

3. Interior point methods for linear programming using a GPU. Linear programming is the problem of minimizing a linear objective function subject to a set of linear constraints, either equalities or inequalities. The standard form is

$$\min_{\mathbf{x}} \ \mathbf{c}^T\mathbf{x} \tag{3.1}$$

$$s.t. \ \mathbf{Ax} = \mathbf{b}, \tag{3.2}$$

$$\mathbf{x} \geq \mathbf{0}, \tag{3.3}$$

where $\mathbf{c}$ and $\mathbf{x}$ are real vectors of size n, $\mathbf{b}$ is a real vector of size m, and $\mathbf{A}$ is an $m \times n$ real matrix with rank $m \leq n$. The dual problem, involving the Lagrange multipliers $\boldsymbol{\lambda}$ for the nonnegativity constraints, is specified by

$$\max_{\boldsymbol{\lambda}} \ \mathbf{b}^T\boldsymbol{\lambda} \tag{3.4}$$

$$s.t. \ \mathbf{A}^T\boldsymbol{\lambda} + \mathbf{s} = \mathbf{c}, \tag{3.5}$$

$$\mathbf{s} \geq \mathbf{0}, \tag{3.6}$$

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

A primal-dual interior point method (IPM) [29] is a standard approach to solving the linear programming problem (3.1)-(3.3) Solving the linear programming problem is equivalent to finding a solution to the KKT (Karush-Kuhn-Tucker) conditions:

$$\mathbf{A}^T\boldsymbol{\lambda} + \mathbf{s} = \mathbf{c}, \tag{3.7}$$

$$\mathbf{Ax} = \mathbf{b}, \tag{3.8}$$

$$x_i s_i = 0, \ \ i = 1, 2, ..., n, \tag{3.9}$$

$$\mathbf{x} \geq \mathbf{0}, \ \mathbf{s} \geq \mathbf{0}. \tag{3.10}$$

The IPM solves this system of equations using a variant of Newton's method. The search direction at each iteration is obtained by solving either the perturbed KKT conditions,

$$\begin{bmatrix} \mathbf{0} & \mathbf{A} & \mathbf{0} \\ \mathbf{A}^T & \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{S} & \mathbf{X} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{\lambda} \\ \Delta\mathbf{x} \\ \Delta\mathbf{s} \end{bmatrix} = \begin{bmatrix} -\mathbf{r}_b \\ -\mathbf{r}_c \\ -\mathbf{r}_{xs} \end{bmatrix}, \tag{3.11}$$

or, equivalently, the normal equations,

$$\mathbf{AD}^2\mathbf{A}^T\Delta\boldsymbol{\lambda} = -\mathbf{r}_b + \mathbf{A}(\mathbf{S}^{-1}\mathbf{X}\mathbf{r}_c + \mathbf{S}^{-1}\mathbf{r}_{xs}), \tag{3.12}$$

$$\Delta\mathbf{s} = -\mathbf{r}_c - \mathbf{A}^T\Delta\boldsymbol{\lambda}, \tag{3.13}$$

$$\Delta\mathbf{x} = -\mathbf{S}^{-1}(\mathbf{r}_{xs} + \mathbf{X}\Delta\mathbf{s}), \tag{3.14}$$

where $\mathbf{D}^2 = \mathbf{S}^{-1}\mathbf{X}$; $\mathbf{r}_b = \mathbf{Ax} - \mathbf{b}$; $\mathbf{r}_c = \mathbf{A}^T\boldsymbol{\lambda} + \mathbf{s} - \mathbf{c}$; $\mathbf{e} = (1, ..., 1)^T$; and $\mathbf{X}$ and $\mathbf{S}$ are diagonal matrices with entries $\mathbf{x}$ and $\mathbf{s}$. The vector $\mathbf{r}_{xs}$ has two definitions: $\mathbf{r}_{xs} = \mathbf{XSe}$ for the affine-scaling step used as a predictor, and $\mathbf{r}_{xs} = \mathbf{XSe} - \sigma\mu\mathbf{e} + \Delta\mathbf{X}^{\text{aff}}\Delta\mathbf{S}^{\text{aff}}\mathbf{e}$ for the combined predictor-corrector step that is actually used to update $\boldsymbol{\lambda}$, $\mathbf{x}$, and $\mathbf{s}$ [29]. Here σ is a *centering parameter* and $\mu = \mathbf{x}^T\mathbf{s}/n$ is the *complementarity measure*. The affine-scaling direction is the pure Newton direction for (3.7)-(3.9), while the corrector step attempts to maintain distance from the nonnegativity constraints.

Usually solving the normal equations is preferred to solving the KKT system, because the matrix for the normal equations is much smaller. Moreover the matrix is symmetric and positive definite, and thus we can use Cholesky factorization, which is faster than LU and requires no pivoting.

In the following sections we discuss how the components of the IPM can be implemented on a GPU.

ETNA
Kent State University
etna@mcs.kent.edu

3.1. Matrix assembly and Cholesky decomposition on a GPU. In [8] we discuss assembling and factoring the matrix $\mathbf{AD}^2\mathbf{A}^T$ on a GPU, so we give only a brief overview in this section.

Gunnels and Gustavson [5] proposed *rectangular-packed format* for symmetric or triangular matrices, saving half the storage space by transposing and moving the lower triangular submatrix at the bottom right to the unused upper left corner, as shown in Figures 2.2a and 2.2b. Storing an $m \times m$ matrix in packed format results in a $w \times h$ texture, where $w = \lceil m/2 \rceil$ and $h = m + \mathrm{mod}(m+1, 2)$.

In order to implement GPU algorithms based on the rectangular-packed format, we need to generate interpolated indices for fetching input textures, using the rasterizer to minimize instruction count [4]. We use $V(x, y)$ to represent a vertex and $T\#(x, y)$ to denote texture coordinates. Since the access pattern of the lower trapezoid is different from that of the upper triangle, we consider two cases.

We assemble the matrix $\mathbf{AD}^2\mathbf{A}^T$ by taking the sum of n outer products. The kth of these involves the kth column of $\mathbf{A}$, scaled by d_{kk}^2, multiplied by the transpose of the kth column of $\mathbf{A}$. We store the kth scaled column of $\mathbf{A}$ in a temporary texture $\mathbf{b}$. Assigning texture coordinates for the triangle covering the lower trapezoid is the same as for the full format. The access pattern of a fragment in the upper triangle is illustrated in Figure 3.1.

In factoring the matrix, we use the outer-product version of the Cholesky algorithm. At step k ($k = 0, \ldots, m-2$), we update elements in columns $j = k+1, \ldots, m-1$ and rows $i = j, \ldots, m-1$ by

$$\ell_{ij} = \ell_{ij} - \ell_{ik}\ell_{jk}.$$

We draw two triangles to initiate the outer product subtraction kernel in steps $k = 0, \ldots, w-1$, as explained in [8]. Attaching texture coordinates to the triangle covering the lower trapezoid is not much different from doing so for a matrix in the full format. To obtain texture coordinates attached to the triangle covering the upper triangular submatrix, we imagine fetching inputs at the original position of an active fragment as illustrated in Figure 3.2a. In steps $k = w, \ldots, m-2$, all required entries are in the same submatrix where the active fragment is, as illustrated in Figure 3.2b, so attaching texture coordinates is straightforward.

3.2. Forward and back substitution on a GPU. Once we compute the Cholesky factor of the matrix, we then solve the system of equations through forward and back substitution.

One option is to transfer the Cholesky factor to the CPU memory and perform the computation there. For reference, we list in Algorithm 1 a CPU version of forward substitution to solve $\mathbf{Ly} = \mathbf{f}$ where $\mathbf{L}$ is a lower triangular matrix. This algorithm needs to be modified for rectangular-packed storage. In this case we partition the system as

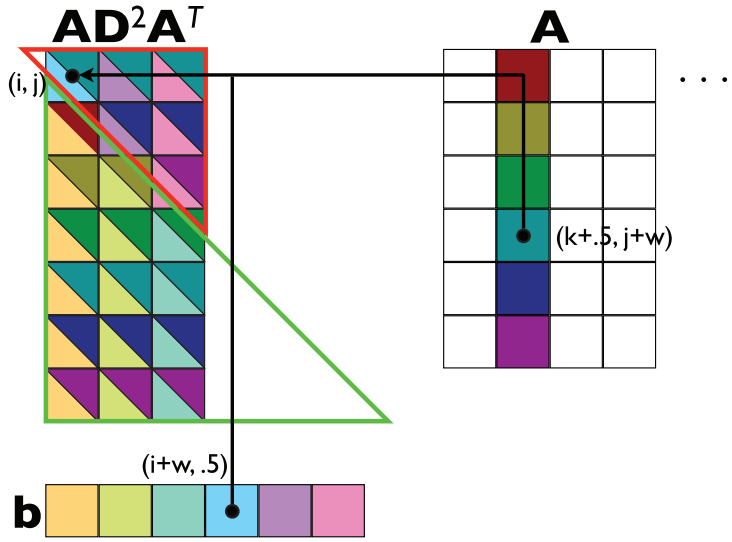
$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix},$$

Remembering that $\mathbf{L}_{11}$ and $\mathbf{L}_{21}$ are stored in the lower trapezoid and $\mathbf{L}_{22}$ is stored in the upper triangle of our texture, as illustrated in Figure 2.2, it is a simple exercise to rewrite Algorithm 1 to access the proper entries. Back substitution is similar.

We can avoid the expensive transfer of the $\mathbf{L}$ factor from the GPU to the CPU by performing forward and back substitution on the GPU. Then we only need to transfer the resulting vector $\mathbf{x}$ of size $m \times 1$. However, we will see in section 4.1 that this approach is slower than transferring the Cholesky factor to the CPU memory and performing forward and back substitutions using the CPU.

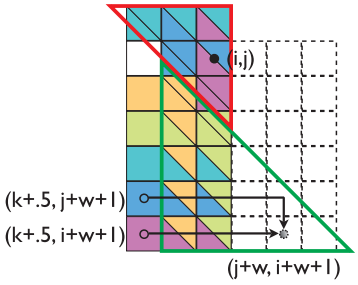
Referring to Algorithm 1, we need kernels for two operations: division and sub-column subtraction. The inner loop disappears, replaced by specifying the vertices in the calls to the

ETNA
Kent State University
etna@mcs.kent.edu

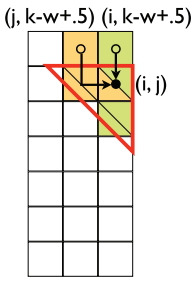


(a)

Fig. 3.1: Forming the matrix $\mathbf{AD}^2\mathbf{A}^T$ on a GPU using outer product updates. At the kth step, we add the outer product of the vector **b** (which stores d_{kk} times the kth column of **A**) with the kth column of **A**. Updates within both the large green and the smaller red triangles are formed in parallel by one kernel call initiated by drawing the two triangles at the same time. The two colors for each element in $\mathbf{AD}^2\mathbf{A}^T$ identify the vector elements whose product forms its update. Note that we take advantage of the relation between **b** and **A** so that we can use the same kernel in both the green and red triangle. A fragment located in the upper triangle with texture coordinates (i, j) receives a contribution of a texel of **A** at $(k + .5, j + w)$ multiplied by a texel of **b** at $(i + w, .5)$. So for a vertex at $V(x, y)$, we attach texture coordinates $T0(k + .5, y + w)$ and $T1(x + w, .5)$.



(a) To understand the access pattern for an active fragment (i, j) located in the upper triangle, it helps to remember its position before packing: $(j+w, i+w+1)$. This figure shows an update when $k < w$.



(b) When $k \geq w$, the entries that generate the update are in the upper triangular submatrix.

Fig. 3.2: Forming a Cholesky factor using the the outer product subtraction kernel when m is even. Colored fragments are processed in parallel.

ETNA
Kent State University
etna@mcs.kent.edu

Algorithm 1 A CPU version of forward substitution

```
// We assume array index starts from 0
// Indices are in (row, column) order
for k = 0 to m-2 do
   // Division
   f(k) = y(k)/L(k,k);
   // Sub-column subtraction
   for i = k+1 to m-1 do
      f(i) = f(i) - f(k)*L(i,k);
   end for
end for
f(m-1) = f(m-1)/L(m-1,m-1);
```

kernels listed in Kernels 1 and 2. We need to keep in mind that the vector **f** is stored in an $m \times 1$ texture in $width \times height$ order[4]. Due to the packing, we need to treat steps 0 to $w-1$ and steps w to $m-2$ differently.

Kernel 1 The GPU kernel for division

```
float main( uniform samplerRECT f : TEXUNIT0,
            uniform samplerRECT L : TEXUNIT1,
            float2 f_index        : WPOS,
            float2 L_index        : TEXCOORD0 ) : COLOR {
    return texRECT(f, f_index)/texRECT(L, L_index);
}
```

(The semantic keywords `TEXUNIT`i, `TEXCOORD`i and `WPOS` represent the i^{th} input texture, the i^{th} interpolated texture coordinates, and the position of the active fragment.)

Kernel 2 The GPU kernel for sub-column subtraction

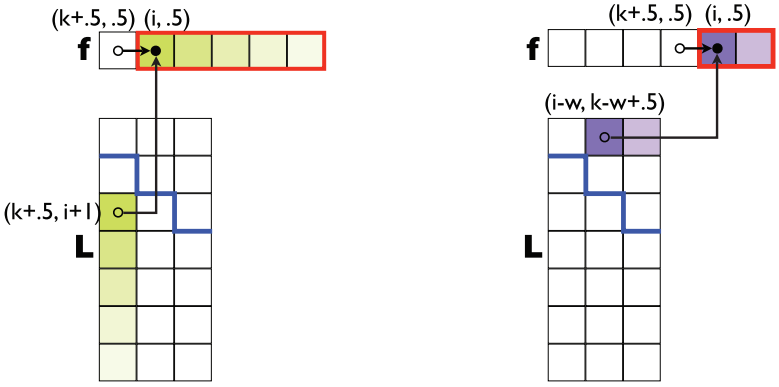
```
float main( uniform samplerRECT f : TEXUNIT0,
            uniform samplerRECT L : TEXUNIT1,
            float2 f_index        : WPOS,
            float2 f_pivot_index  : TEXCOORD0,
            float2 L_index        : TEXCOORD1 ) : COLOR {
    return texRECT(f, f_index).x
           - texRECT(f, f_pivot_index).x * texRECT(L, L_index.yx).x;
}
```

For the k^{th} division operation, we draw a point of size 1×1 at $V(k+.5, .5)$ with a set of attached texture coordinates. Suppose that m is even. Then in the first set of steps we fetch the diagonal entry of **L**, stored in the trapezoid, from position $(k+.5, k+1.5)$. In the second set of steps, the required diagonal entry of **L** is stored in the upper triangle in position $(k-w+.5, k-w+.5)$. Obtaining the attached texture coordinates for odd m is not much different.

For the k^{th} sub-column subtraction, we draw a line of width 1 covering the entries from $k+1$ to $m-1$ of **f**. Texture coordinates $T0$ for fetching **f** are fixed for all active fragments.

[4]This scheme restricts the maximum size of a vector to 4096. Packing a vector in a rectangle texture can remove this restriction [9].

ETNA
Kent State University
etna@mcs.kent.edu



(a) In the first half of the iterations, active fragments fetch elements from the lower trapezoid.

(b) In the second half of the iterations, active fragments fetch elements from the upper triangle.

Fig. 3.3: These figures describe how we access **f** and **L** and attach the texture coordinates to each vertex for the sub-column subtraction when m is even. Colored fragments in **f** are processed in parallel. We used the same color for an active fragment and its corresponding element in **L**.

So attaching $T0$ to vertices is straightforward. To attach the second set of texture coordinates $T1$ for fetching **L**, we need to understand the access pattern of active fragments.

Kernel 3 The GPU kernel for sub-row subtraction

```
float main( uniform samplerRECT f : TEXUNIT0,
            uniform samplerRECT L : TEXUNIT1,
            float2 f_index        : WPOS,
            float2 f_pivot_index  : TEXCOORD0,
            float2 L_index        : TEXCOORD1 ) : COLOR {
    return texRECT(f, f_index).x
           - texRECT(b, f_pivot_index.xy).x * texRECT(L, L_index).x;
}
```

In the first set of steps, as illustrated in Figure 3.3a, an active fragment at $(i, .5)$ needs to fetch a texel of **L** at $(k + .5, i + 1)$. The texture storing **f** is laid out horizontally. Thus, we cannot have the rasterizer interpolate texture coordinates vertically, but we generate interpolated coordinates $T1(i + 1, k + .5)$ by attaching $T1(x + 1, k + .5)$ to a vertex at $V(x, y)$. By swizzling the interpolated texture coordinates `L_index` as in Kernel 2 we can handle the necessary transpose operation.

In the second set of steps, as illustrated in Figure 3.3b, an active fragment at $(i, .5)$ needs to fetch a texel of **L** at $(i - w, k - w + .5)$. We can generate the coordinates by attaching $T1(x-w, k-w+.5)$ to a vertex at $V(x, y)$. No swizzle is necessary, so the kernel, Kernel 3, is slightly different.

By understanding the access pattern, we can in a similar way derive the algorithm for back substitution.

3.3. A CPU-GPU interior point method for linear programming. Algorithm 2 uses a variant of Mehrotra's predictor-corrector (MPC) method from [29] to solve the linear pro-

ETNA
Kent State University
etna@mcs.kent.edu

gramming problem (3.1)-(3.6), performing most of the computation on the GPU.

We stop the iteration when the relative residual and the duality measure are smaller than some small tolerance ϵ:

$$\max\left\{\frac{\max\{\|\mathbf{r}_b\|_\infty, \|\mathbf{r}_c\|_\infty\}}{\max\{\|\mathbf{b}\|_\infty, \|\mathbf{c}\|_\infty, \|\mathbf{A}\|_\infty\}}, \frac{|\mathbf{c}^T\mathbf{x} - \mathbf{b}^T\boldsymbol{\lambda}|}{1 + |\mathbf{c}^T\mathbf{x}|}\right\} \leq \epsilon. \tag{3.15}$$

The coefficient matrix $\mathbf{A}$ is written to the GPU only once, at the beginning. At each iteration, we transfer only a few vectors, including the right-hand side of the normal equation (3.12) and the main diagonal of $\mathbf{D}$. Ideally, matrix assembly, factorization, and forward and back substitution are performed on the GPU; for the remainder of the computation we use MATLAB functions on the CPU. But since our current GPU does not support double precision, we use the CPU for matrix assembly and factorization in later iterations in order to get accurate results[5]. We monitor the quality of the combined predictor-corrector step by testing whether the relative residual norm for (3.12) is too large:

$$r_{\Delta\boldsymbol{\lambda}} = \frac{\|\mathbf{r} - \mathbf{A}\mathbf{D}^2\mathbf{A}^T\Delta\boldsymbol{\lambda}\|}{\|\mathbf{r}\|} \geq \theta_r, \tag{3.16}$$

where $\mathbf{r}$ is the right hand side of (3.12) and θ_r is a threshold parameter. If (3.16) is satisfied, we form and solve the normal equations on the CPU.

The matrix $\mathbf{A}\mathbf{D}^2\mathbf{A}^T$ can become ill-conditioned in two ways, also making it necessary to use the double precision CPU solver. First, the dual problem may have fewer than m active constraints, which causes more than $n - m$ entries of $\mathbf{D}^2$ to approach zero. To monitor this, we count the number of entries in $\mathbf{D}$ smaller than some small tolerance $\epsilon_d > 0$ and use the CPU if

$$\left|\left\{i : d_{ii}^2 < \epsilon_d \text{ for } i = 1, ..., n\right\}\right| > n - m, \tag{3.17}$$

where d_{ii} is the i^{th} diagonal element of $\mathbf{D}$. Second, some of the primal variables $\mathbf{x}$ may be unbounded, which causes some diagonal entries of $\mathbf{D}^2$ to grow too fast relative to the others [29]. To monitor this, we measure the ratio between the largest d_{ii}^2 and the smallest d_{ii}^2 among diverging entries. So, given parameters θ_a and θ_d, we use the CPU if

$$\max_{d_{ii}^2 > \mu\theta_d}\left(d_{ii}^2\right) / \min_{d_{ii}^2 > \mu\theta_d}\left(d_{ii}^2\right) > \theta_a. \tag{3.18}$$

4. Results. To test our algorithms, we used an NVIDIA GeForce 7800 GTX (24 fragment processors, 580 MHz core clock cycle, 1750 MHz memory clock cycle, 512 MB GDDR3 memory, 256 bit bus) and an Intel Xeon 3.0GHz (1 MB L2 cache, 8GB DDR2 dual channel memory, 400 MHz effective memory clock cycle and 800 MHz FSB). The operating system is Linux Red Hat 3.4.5-2 64bit. We compiled our code using gcc 3.4.5. We implemented and ran the IPM using MATLAB 7.2.0.283 (R2006a) which uses Intel's Math Kernel Library for BLAS and LAPACK function calls. Results in [8] showed that packing does not degrade overall performance for matrix assembly and factorization, and GPU algorithms outperform ATLAS (Automatically Tuned Linear Algebra Software) routines for sufficiently large matrices.

[5] In fact, even the single precision arithmetic on the GPU is not fully compliant with the IEEE standard [20, 21], so it is important to monitor the quality of the results.

ETNA
Kent State University
etna@mcs.kent.edu

Algorithm 2 GPU-Powered Mehrotra's Predictor-Corrector Algorithm

Specify the parameters ϵ, ϵ_d, θ_r, θ_d, and θ_a.
Transfer the coefficient matrix $\mathbf{A}$ in single precision packed format to GPU memory.
Set `useGPU` as true.
Generate an initial point $(\mathbf{x}^0, \boldsymbol{\lambda}^0, \mathbf{s}^0)$ according to [15].
for k = 0,1,2,... **do**
  Set $\mu = \frac{\mathbf{x}^{k\,T}\mathbf{s}^k}{n}$.
  Terminate if the convergence criteria are met or iteration count limit is reached.
  Set `useGPU` as false if any of (3.16), (3.17) and (3.18) is satisfied.
  if `useGPU` **then**
    Transfer the diagonal of the scaling matrix $\mathbf{D}^2$ to GPU memory.
    Compute and factor $\mathbf{AD}^2\mathbf{A}^T$ using the GPU.
    Transfer $\mathbf{r}_{xs}$ for the predictor to GPU memory.
  else
    Compute and factor $\mathbf{AD}^2\mathbf{A}^T$ in double precision non-packed form using the CPU.
  end if
  Use forward and back substitution to solve (3.12) for the predictor step, transferring the resulting $\Delta\boldsymbol{\lambda}^{\text{aff}}$ to CPU memory if `useGPU` is true.
  Use (3.13)-(3.14) to compute $\Delta\mathbf{x}^{\text{aff}}$ and $\Delta\mathbf{s}^{\text{aff}}$.
  Determine the predictor step length:

$$\alpha_{\text{aff}}^{\text{pri}} = \arg\max_{\alpha\in[0,1]} \{\mathbf{x}^k + \alpha\Delta\mathbf{x}^{\text{aff}} \geq 0\},\ \alpha_{\text{aff}}^{\text{dual}} = \arg\max_{\alpha\in[0,1]} \{\mathbf{s}^k + \alpha\Delta\mathbf{s}^{\text{aff}}) \geq 0\}.$$

  Determine the centering parameter:

$$\sigma = (\mu_{\text{aff}}/\mu)^3,\ \text{where}\mu_{\text{aff}} = \frac{(\mathbf{x}^k + \alpha_{\text{aff}}^{\text{pri}}\Delta\mathbf{x}^{\text{aff}})^T(\mathbf{s}^k + \alpha_{\text{aff}}^{\text{dual}}\Delta\mathbf{s}^{\text{aff}})}{n}.$$

  Use forward and back substitution to solve (3.12) for the combined predictor-corrector step, transferring $\mathbf{r}_{xs}$ to GPU memory and transferring the resulting $\Delta\boldsymbol{\lambda}$ to CPU memory if `useGPU` is true.
  Use (3.13)-(3.14) to compute $\Delta\mathbf{x}$ and $\Delta\mathbf{s}$.
  Determine step size parameters, α_k^{pri} and α_k^{dual}:

$$\alpha_k^{pri} = 0.99 \times \arg\max_{\alpha\in[0,1]} \{\mathbf{x}^k + \alpha\Delta\mathbf{x} \geq 0\},$$
$$\alpha_k^{dual} = 0.99 \times \arg\max_{\alpha\in[0,1]} \{\mathbf{s}^k + \alpha\Delta\mathbf{s} \geq 0\}.$$

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

4.1. Forward and back substitution. Figure 4.1 compares our forward and back substitution algorithms with `strsv` of ATLAS 3.6.0 [26]. In contrast to matrix assembly and factorization, the GPU algorithms for forward and backward substitution have no performance advantage over the CPU algorithms. Kernel 1 is inherently a non-parallel process, since it must wait until Kernel 2 and 3 finish. So each iteration cannot start until the previous iteration completes. Notice that the graphs for forward and back substitution on the GPU are almost linear in m, while the arithmetic complexity is quadratic. The host CPU sends a fixed number of vertices (and attached texture coordinates) for each rasterization process, or $O(m)$ vertices in total. Therefore it seems that the latency in initiating GPU kernels dominates the overall time, for the problem sizes tested.

ETNA
Kent State University
etna@mcs.kent.edu

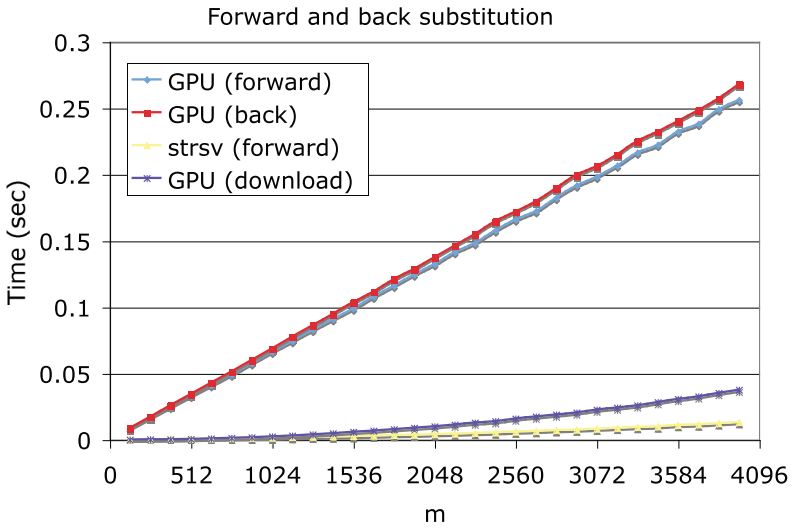


Fig. 4.1: Timing result for forward and back substitution.

The combined time required for moving the packed Cholesky factor to the CPU and performing `strsv` is much less than that for the GPU algorithms. Thus, transferring the factor to the CPU memory and doing forward and back substitution using the CPU results in better performance in the IPM, unless the CPU can be performing other useful work while the GPU is computing.

4.2. Interior point method. We set the termination tolerance parameter ϵ to 10^{-8}. Other parameters are set as follows:

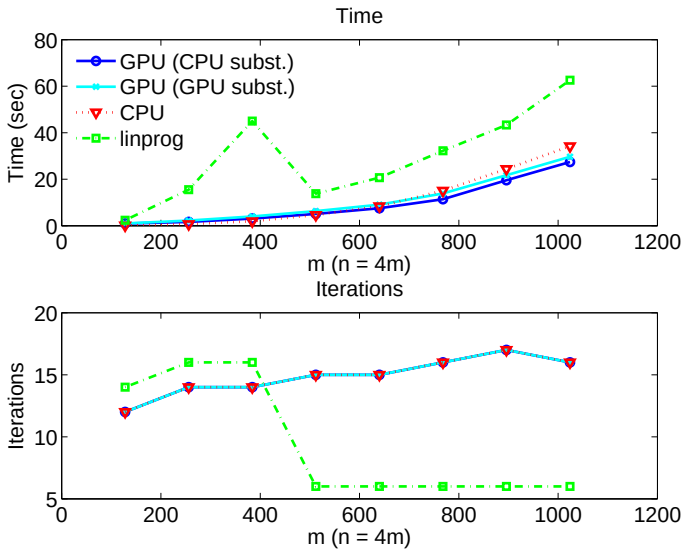
$$\theta_r = 10^{-2}, \epsilon_d = 10^{-4}, \theta_d = 10^3, \text{ and } \theta_a = 10^5.$$

We used the packed version for matrix assembly and factorization. We implemented the two options for the substitution: transferring the Cholesky factor to the CPU, or using the GPU to solve the triangular systems. We compared these two options with our full double precision MATLAB implementation[6] without using the GPU for solving the normal equations (3.12) and with MATLAB's `linprog` function. The results are shown in Table 4.1 and in Figure 4.2.

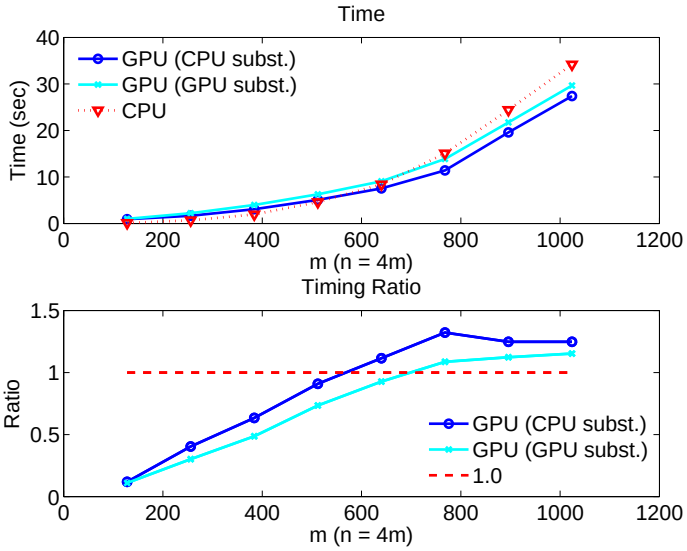
The NETLIB problems are not large enough to gain a performance advantage using the GPU. As illustrated in Table 4.1, the full double precision CPU version usually needs fewer iterations to terminate than the GPU versions. We generated random problems with each constraint in the dual tangent to the unit sphere as described in [25]. Results are summarized in Figure 4.2. Our algorithms using the GPU are slower for small problems but faster than the full double precision CPU version for $m > 640$. In solving small problems, data transfer cost and communication latency prevent the solver from achieving good performance.

[6]It is also possible to implement a CPU version of a hybrid single and double precision IPM, but MATLAB 7.2 running on 64bit Linux has a bug in interfacing with single precision BLAS routines. This bug prevented us from forming the normal equation matrix in single precision; see http://www.mathworks.com/support/bugreports/details.html?rp=268001. This bug is fixed in MATLAB 7.4.

ETNA
Kent State University
etna@mcs.kent.edu



(a) Our algorithms are compared with MATLAB's `linprog`.



(b) Timing result for `linprog` is eliminated to magnify gap between the combined CPU-GPU solvers and the CPU only solver. In the bottom figure, we plot the ratio of the time for the CPU solver to that for the GPU-powered solvers. Values greater than 1 indicate a performance advantage for the GPU solver.

Fig. 4.2: We measured running time and iteration count of Algorithm 2 on random problems. For sufficiently large problems, using the combined CPU-GPU solver yields better performance. The horizontal axis represents m, where we set $n = 4m$. GPU in the label means that the GPU is used for assembling and factoring matrices in Algorithm 2, whereas CPU means that the GPU is not used at all.

ETNA
Kent State University
etna@mcs.kent.edu

Table 4.1: We measured the running time and iteration count of Algorithm 2 on NETLIB problems. The iteration count in parentheses represents the number of iterations at which the GPU is used for assembling and factoring the matrix for the normal equations. We used two versions of the GPU algorithm. The one labeled (GPU subst.) uses the GPU to solve the triangular systems, and the other one, labeled (CPU subst.) uses the CPU. None of these problems is sufficiently large to get performance gain through using a GPU.

Problem	Size	GPU (GPU subst.)		GPU (CPU subst.)		CPU	
		Iterations	Time (s)	Iterations	Time (s)	Iterations	Time (s)
afiro	27×51	9 (7)	0.19	9 (7)	0.21	9	0.01
adlittle	56×138	11 (9)	0.47	11 (9)	0.34	11	0.01
agg2	516×758	20 (15)	6.16	20 (15)	4.07	20	2.68
agg3	516×758	20 (16)	6.35	28 (17)	5.25	20	2.68
bandm	305×472	17 (8)	2.06	17 (8)	1.39	17	0.61
beaconfd	173×295	9 (4)	0.59	9 (4)	0.39	9	0.09
blend	74×114	11 (6)	0.34	11 (6)	0.22	11	0.01
e226	223×472	22 (11)	2.24	21 (11)	1.53	22	0.44
sc50b	50×78	8 (5)	0.21	8 (5)	0.13	8	0.01
sctap1	300×660	15 (12)	3.17	15 (12)	2.16	15	0.65

MATLAB's `linprog` is slower than our algorithms even when it terminates with fewer iterations. It fails to converge to an optimal solution for problems with $m \geq 512$. It uses LIPSOL [30] which always uses a Cholesky-infinity factorization supporting only sparse matrices. This causes overhead in factorization of dense normal equations matrices. Modifying the Cholesky-infinity factorization to support dense matrices would improve the performance.

5. Conclusions. We have presented a CPU-GPU algorithm for solving linear programming problems using interior point methods. This algorithm uses rectangular-packed matrix storage [5] and uses the GPU for tasks such as matrix assembly, Cholesky factorization, and forward and back substitution. By comparing our implementations with a CPU implementation, we demonstrated that we can improve performance by using the GPU and mixed precision for sufficiently large dense problems. For some sparse problems, techniques such as supernodal multifrontal approaches can be used to create dense submatrices for which a GPU might be used. Since GPU architectures and programming languages are rapidly evolving, we expect that GPUs will be an increasingly attractive tool for matrix computation in the future.

Acknowledgments. We are grateful to the referees for their helpful comments.

REFERENCES

[1] J. BOLZ, I. FARMER, E. GRINSPUN, AND P. SCHRÖODER, *Sparse matrix solvers on the GPU: conjugate gradients and multigrid*, in ACM SIGGRAPH 2003 Papers, ACM, New York, NY, 2003, pp. 917–924.

[2] I. BUCK, T. FOLEY, D. HORN, J. SUGERMAN, K. FATAHALIAN, M. HOUSTON, AND P. HANRAHAN, *Brook for GPUs: stream computing on graphics hardware*, in ACM SIGGRAPH 2004 Papers, ACM, New York, NY, 2004, pp. 777–786.

[3] Z. FAN, F. QIU, A. KAUFMAN, AND S. YOAKUM-STOVER, *GPU cluster for high performance computing*, in Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, IEEE, Washington, DC, 2004, p. 47.

[4] N. GALOPPO, N. K. GOVINDARAJU, M. HENSON, AND D. MANOCHA, *LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware*, in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE, Washington, DC, 2005, p. 3.

ETNA
Kent State University
etna@mcs.kent.edu

[5] J. A. GUNNELS AND F. G. GUSTAVSON, *A new array format for symmetric and triangular matrices*, in PARA'04 Workshop on State-of-the-Art in Scientific Computing, 2004, pp. 247–255.
[6] J. HALL, N. CARR, AND J. HART, *Cache and bandwidth aware matrix multiplication on the GPU*, Tech. Report UIUCDCS-R-2003-2328, University of Illinois at Urbana-Champaign, 2003. http://graphics.cs.uiuc.edu/~jch/papers/UIUCDCS-R-2003-2328.pdf.
[7] INTEL CORP., *Intel 945G express chipset product brief*, 2005. http://www.intel.com/products/chipsets/945g/index.htm.
[8] J. H. JUNG AND D. P. O'LEARY, *Exploiting structure of symmetric or triangular matrices on a GPU*, in Workshop for General Purpose Processing on Graphics Processing Units, Boston, MA, Oct. 2007, Computer Science Department Report CS-TR-4914, Institute for Advanced Computer Studies Report UMIACS-TR-2008-12, Jan. 2008. http://hdl.handle.net/1903/7984.
[9] J. KRÜGER AND R. WESTERMANN, *Linear algebra operators for GPU implementation of numerical algorithms*, in ACM SIGGRAPH 2005 Courses, ACM, New York, NY, 2005, p. 234.
[10] D. LUEBKE, *General-purpose computation on graphics hardware*. SIGGRAPH 2005 GPGPU Course, Aug. 2005. http://www.gpgpu.org/s2005/.
[11] D. LUEBKE, M. HARRIS, J. KRÜGER, T. PURCELL, N. GOVINDARAJU, I. BUCK, C. WOOLLEY, AND A. LEFOHN, *GPGPU: general purpose computation on graphics hardware*, in ACM SIGGRAPH 2004 Course Notes, ACM, New York, NY, 2004, p. 33.
[12] D. LUEBKE, NVIDIA CORP., *Personal communication*, Oct. 2007.
[13] F. T. LUK, *Computing the singular-value decomposition on the ILLIAC IV*, ACM Trans. Math. Software, 6 (1980), pp. 524–539.
[14] W. R. MARK, R. S. GLANVILLE, K. AKELEY, AND M. J. KILGARD, *Cg: a system for programming graphics hardware in a C-like language*, in ACM SIGGRAPH 2003 Papers, ACM, New York, NY, 2003, pp. 896–907.
[15] S. MEHROTRA, *On the implementation of a primal-dual interior point method*, SIAM J. Optim., 2 (1992), pp. 575–601.
[16] MICROSOFT CORP., *High-level shader language*, in DirectX 9.0 Graphics, 2003. http://msdn.microsoft.com/directx.
[17] MICROSOFT CORP., *DirectX 9.0 graphics*, in DirectX 9.0 Graphics, 2005. http://msdn.microsoft.com/directx.
[18] K. MORELAND AND E. ANGEL, *The FFT on a GPU*, in Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Aire-la-Ville, Switzerland, Eurographics Association, 2003, pp. 112–119.
[19] E. NG AND B. W. PEYTON, *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors*, SIAM J. Sci. Comput., 14 (1993), pp. 761–769.
[20] NVIDIA CORP., *Fast texture downloads and readbacks using pixel buffer objects in OpenGL*, Technical Brief, Santa Clara, CA, Aug. 2005.
[21] NVIDIA CORP., *CUDA Programming Guide*, Santa Clara, CA, Feb. 2007.
[22] J. D. OWENS, D. LUEBKE, N. GOVINDARAJU, M. HARRIS, J. KRÜGER, A. E. LEFOHN, AND T. J. PURCELL, *A survey of general-purpose computation on graphics hardware*, Computer Graphics Forum, 26 (2007), pp. 80–113.
[23] D. M. RITCHIE, *The development of the C language*, SIGPLAN Notices, 28 (1993), pp. 201–208.
[24] R. J. ROST, *OpenGL(R) Shading Language*, Addison-Wesley Longman Publishing Co., Redwood City, CA, 2004.
[25] A. L. TITS, P.-A. ABSIL, AND W. P. WOESSNER, *Constraint reduction for linear programs with many inequality constraints*, SIAM J. Optim., 17 (2006), pp. 119–146.
[26] R. C. WHALEY AND A. PETITET, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, 35 (2005), pp. 101–121.
[27] M.-L. WONG, T.-T. WONG, AND K.-L. FOK, *Parallel evolutionary algorithms on graphics processing unit*, in 2005 IEEE Congress on Evolutionary Computation, 2005, pp. 2286–2293.
[28] M. WOO, J. NEIDER, T. DAVIS, AND D. SHREINER, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Longman Publishing Co., Boston, MA, 2005.
[29] S. J. WRIGHT, *Primal-Dual Interior-Point Methods*, SIAM, Philadelphia, PA, 1997.
[30] Y. ZHANG, *Solving large–scale linear programs by interior–point methods under the MATLAB environment*, Tech. Report 96–01, Department of Mathematics and Statistics, University of Maryland Baltimore County, Baltimore, MD, 1996.