Abstraction of Graphics Hardware Through The Use of Modern Interfaces to Increase Performance of Linear Algebra Routines

M. Woolsey, W. E. Hutchcraft, and R. K. Gordon

Department of Electrical Engineering
University of Mississippi, University, MS 38677, USA
muwoolse@olemiss.edu, eeweh@olemiss.edu, eegordon@olemiss.edu

Abstract – General purpose computation on graphics processing units (GPGPU) is introduced through the application of modern interfaces that abstract graphics hardware. In order to provide an example of these techniques, implementation of an iterative matrix solving algorithm is detailed using two interfaces – Stanford's BrookGPU and Accelerator from Microsoft Research. Performance of the Accelerator implementation is then analyzed.

Keywords: Graphics processing unit, GPGPU, and parallel computing.

I. BACKGROUND

Graphics processing units (GPUs) utilize a parallel pipeline architecture to render graphics onto a 2D screen. A traditional GPU consists of vertex processors, a rasterizer, and pixel processors, as shown in Fig. 1. The vertex processor handles operations such as geometric transformations and per-vertex lighting. The rasterizer converts the vertex data to a 2D array of pixels, and the pixel processors perform texturing and per-pixel lighting operations. Early GPUs employed fixed function pipelines, in which the vertex and pixel processors performed a set of predefined operations. Later GPUs, however, incorporated programmable pipelines, in which programs called shaders can be passed to the vertex and pixel processors. For general purpose applications, most computations are performed within pixel shaders.

The programmable pipelines incorporate many parallel processors that function according to a single instruction, multiple data programming scheme. A pixel operation, for example, can be applied independently to every pixel in a scene. This data-parallel processing capability is the primary draw to general purpose utilization. In addition to parallel processing, graphics hardware also provides a number of useful built-in data types and operations. Because the hardware is optimized to operate on 3D graphics and lighting, data types include multicomponent floating point vectors, and instruction sets contain useful operations such as dot products.

In the latest generation of graphics hardware, both vertex and pixel operations are carried out on processors called unified shaders or simply referred to by their function – stream processors. This architecture results in a more traditional parallel computing environment in which data can be spread over a number of identical processors. The current generation NVidia 8800GT used in this research contains 112 stream processors, with a peak performance of over 500 GFLOPS, and a cost of less than $250.

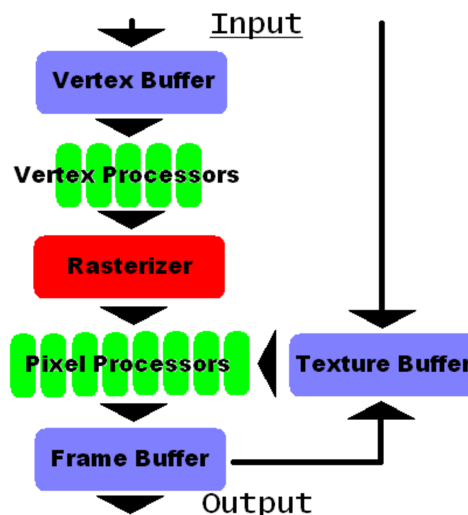

Fig. 1. Traditional GPU architecture.

Processing general purpose data on a GPU, however, has required a creative approach since the hardware is specialized for graphics use. In order for data arrays to be processed within a GPU, they must be stored as textures in graphics memory. In addition, the shader must be explicitly loaded into the GPU memory. By rendering a quadrilateral, data values and pixels can be mapped to one another, and the shader operates on the texture values. The rendering phase may be iterated as necessary, redirecting the shader output as an input texture. The final output may then be read from graphics memory.

II. GPU PROGRAMMING

Until recently, programming graphics hardware required the knowledge of a shader language. These are languages that can be compiled to run on pixel and vertex processors, such as the OpenGL Shading Language (GLSL), Microsoft's High Level Shader Language (HLSL), and NVidia's C for graphics (Cg). GLSL is a cross platform shading language, due to its OpenGL heritage. HLSL is used within the Windows operating system, as it links to DirectX. Cg, however, is unique in that it can target either OpenGL or DirectX as a compiler option. These languages are common in graphics programming, and they require extensive knowledge about the graphics processes as described in the previous section as well as the selected graphics interface – OpenGL or DirectX. While these languages are a great improvement over hardware-specific assembly language, a higher level of graphics hardware abstraction is necessary for general purpose computation.

The ability to perform general purpose computations on graphics hardware without extensive background in graphics programming is now possible due to the abstraction provided by modern interfaces including Brook for GPUs (BrookGPU) from Stanford University and Accelerator from Microsoft Research. BrookGPU and Accelerator extend C and Microsoft C Sharp (C#) respectively with new syntax and data types allowing data to be transferred to and from the GPU and shaders to be loaded and configured as necessary without complication. An example program will be shown for both, and a brief performance analysis will be conducted on the Accelerator program.

III. JACOBI ALGORITHM

The Jacobi iterative matrix solving algorithm has been written for both the CPU and GPU as a demonstration. The algorithm computes $\bar{u}$ from $\bar{\bar{M}}\,\bar{u} = \bar{f}$, and can be explored by solving for u_1.

$$M_{11}u_1 + M_{12}u_2 + \cdots + M_{1N}u_N = f_1 ,$$

$$u_1 = \frac{f_1 - M_{12}u_2 - \cdots - M_{1N}u_N}{M_{11}} .$$

Let $\bar{\bar{D}}$ be a diagonal matrix that holds only the main diagonal of $\bar{\bar{M}}$. The above expression may be written as,

$$u_1 = \frac{f_1 - (M_{11} - D_{11})u_1 - \cdots - (M_{1N} - D_{1N})u_N}{D_{11}} .$$

The RHS of this equation can be used to update the LHS value, which results in the following iterative matrix equation.

$$\bar{u}^{n+1} = \bar{\bar{D}}^{-1}\bar{f} - \bar{\bar{D}}^{-1}(\bar{\bar{M}} - \bar{\bar{D}})\bar{u}^{n}$$

$\bar{\bar{M}}'$ can be defined as $\bar{\bar{M}} - \bar{\bar{D}}$ for simplicity, and the elements of $\bar{\bar{D}}$ can be stored in the vector $\bar{d}$. This allows the following expression, in which "÷" represents an element-by-element division.

$$\bar{u}^{n+1} = (\bar{f} - \bar{\bar{M}}'\,\bar{u}^{n}) \div \bar{d}$$

The procedure is performed by a matrix-vector multiplication, followed by an element-by-element vector subtraction, and then an element-by-element vector division. The equation is successively evaluated for $\bar{u}^{n+1}$, and at the end of each iteration, the components of $\bar{u}$ are updated simultaneously [1].

The advantage of Jacobi iteration in this demonstration is that the evaluation of $\bar{u}$ components can be performed independently, in parallel between updates. The algorithm may be summarized in the flowchart shown in Fig. 2.

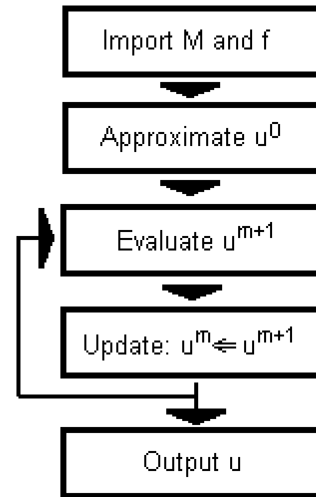

Fig. 2. Iterative algorithm.

In this research, the CPU and GPU Jacobi solvers are applied to a matrix generated by a finite element program. Other numerical methods may utilize a completely different approach. In a finite difference analysis, for example, the only elements contributing to the value of an unknown are its neighbors, so the algorithm only examines a point's neighbors to update its value, rather than an entire matrix row. A performance increase has been demonstrated in finite difference time domain by applying GPU programming in this way [2].

IV. BROOKGPU

BrookGPU was developed at Stanford University as an offshoot of the Brook stream processing language, created for the Merrimac streaming supercomputer. Brook extends C with the concepts of *streams* and *kernels*. A stream is similar in concept to an array, and can contain float values as well as the multicomponent vector types – float2, float3, and float4 as shown in Fig. 3. Data may be passed from an array to a stream using the streamRead operator and vice-versa using streamWrite. Other than these operators, streams may only be accessed in functions called kernels. Within a kernel, stream operations are performed in parallel, and the concept of stream shape must be considered to ensure proper operation [3]. As an example, the following program segment contains the Jacobi iteration using BrookGPU.

The first step in writing a BrookGPU program is to define and initialize the stream. Data arrays declared outside the provided program segment are read into the streams using the StreamRead function. For clarity, all streams use unmodified names, while arrays used by the CPU are given a suffix – i.e. *d* is a GPU stream while *d_array* is stored in system memory. The Jacobi algorithm is divided into four kernels within the main *for* loop. First a multiplication is performed in which each row of matrix *M* is multiplied element-by-element with the row vector *u*. The resulting *temp* matrix is sent to a reduction kernel, in which its rows are summed yielding a column vector. This vector represents the matrix-vector product, which is then transposed to a row vector. The statement *indexof row.yx* returns an index of the current row element with x and y switched, such that row[x,y] = column[y,x]. The final kernel updates by evaluating the statement of row vectors element-by-element. After the iterations are complete, the computed value of *u* is written back to a standard array from the GPU.

A BrookGPU source file containing streams and kernels must be converted to C and shader code using the brcc compiler. After C compilation, program execution calls upon the Brook Runtime, which controls implementation of the kernels on the GPU. Compiler switches applied to brcc allow conversion to Cg or HLSL and optimizations for ATI and nVidia hardware. The last official release of BrookGPU is v0.4, released October 15, 2004, so development using modern hardware and drivers may not be efficient. The Jacobi program produced oddly scaled results until graphics drivers were updated. Even after the driver update, however, the scaling problems would return for all but impractically small matrix sizes. For this reason, a performance evaluation will not be conducted on the BrookGPU Jacobi implementation. At the time of writing, a long awaited update (billed as the v0.5 "test release") has been made available for download, but has not been tested in this project.

Main Program Segments:

```
// DEFINE STREAMS
float d<1,n>;                       //row
float f<1,n>;                       //row
float u<1,n>;                       //row
float columnProduct<n,1>;           //column
float rowProduct<1,n>;              //row
float M<n,n>;                       //matrix
float temp<n,n>;                    //matrix

// INITIALIZE STREAMS
streamRead( d, d_array );           //diagonal
streamRead( f, f_array );           //forcing vector
streamRead( M, M_array );           //matrix M
streamRead( u, u_array );           //initial approx.

// RUN JACOBI ITERATIONS
for( i=0; i<iterations; i++ )
{
    mul( M, u, temp );
    sum( temp, columnSum );
    transpose( columnProduct, rowProduct );
    update( d, f, rowProduct, u );
}

streamWrite( u, u_array );      //output u
```

Kernels:

```
kernel void mul( float a<>, float b<>, out float c<> )
        c = a * b;

reduce void sum(  float a<>, reduce float r<> )
        r += a;

kernel void transpose(float column[][], out float
row<>)
        row=column[ indexof  row.yx ];

kernel void update( float diagonal<>, float forcing<>,
                    float product<>, out float new<> )
        new = ( forcing – product ) / diagonal;
```

Fig. 3. Sample from BrookGPU Jacobi program.

V. ACCELERATOR

Accelerator was produced by Microsoft Research, and is designed to abstract hardware such as GPUs and cell processors. The current implementation provides abstraction for GPGPU programming on C#, and correspondence with a developer on the project suggests that a native C++ version may eventually be released. Accelerator provides a ParallelArray class that contains all necessary functions – I/O, element operations, reductions, transformations, and linear algebra. Linear

algebra routines include vector and matrix multiplications. The ParallelArray class also contains several subclasses such as IntParallelArray and Float4ParallelArray which define data types for single and multicomponent parallel data. Unlike BrookGPU, the shaders are not explicitly separated in the form of kernels. Instead, pixel shaders are created from the Accelerator operations automatically. This does not allow as much control over the underlying shaders, but it allows them to be created and optimized by the compiler at runtime [4].

In the Accelerator program of Fig. 4, the GPU must be initialized before additional calls to the ParallelArray class, which has been shortened to PA for simplicity. Four arrays are read into disposable float parallel arrays, which must be disposed at the end of the program. The Jacobi iteration is straightforward: the matrix-vector product is performed using the InnerProduct function of the ParallelArray class, and the new value of u is calculated as in the BrookGPU program. An additional step is required for memory management. Every time PA.Evaluate() is called, a new array is allocated on the GPU. The old array is explicitly disposed with each iteration in order to free memory. At the completion of the iterations, the ToArray function is used to recover the computed value of u, and all other memory allocated on the GPU is freed.

```
// INIT & UPLOAD TO GPU
PA.InitGPU();
DFPA d = new DFPA( d_array );
DFPA f = new DFPA( f_array );
DFPA M = new DFPA( M_array );
DFPA  u = new DFPA( u_array );
DFPA uNew = null;

// BEGIN JACOBI ITERATIONS
for( i=0; i<iterations; i++ )
{
        // MULTIPLY MATRIX BY u
        FPA product = PA.InnerProduct( M, u );
        // UPDATE u
        uNew = PA.Evaluate( ( f - product ) / d );
        u.Dispose();
        u = uNew;
}

// DOWNLOAD RESULT FROM GPU & CLEAN UP
PA.ToArray( u, out u_array );
d.Dispose();
f.Dispose();
M.Dispose();
uNew.Dispose();
PA.UnInit();
```

Fig. 4. Sample from MSR Accelerator Jacobi program.

Programming with accelerator requires Microsoft Visual Studio or Visual C# Express (available for download) and naturally targets the DirectX graphics interface. No additional steps are required in the compilation process other than including the accelerator.dll file within the project. A disadvantage, however, is that Accelerator ties to DirectX through the .NET framework requiring a Microsoft language such as C#. Interfacing existing programs to Accelerator involves use of the .NET framework, or by writing to a data file that can be imported by an Accelerator program. The current version of Accelerator – available from research.microsoft.com – is v1.1, last updated July 9, 2007.

VI. ELECTROSTATIC EXAMPLE

As a practical example of electromagnetics computations on GPUs, a simple electrostatic problem domain is studied. Consider a rectangular cross-section. The sides and bottom of this domain are maintained at ground potential, while the top is excited with the positive half-cycle of a unit-sinusoidal potential source. The cross-section is discretized by a triangular mesh [5], and a nodal finite element analysis is performed.

Although the Jacobi program is a proof-of-concept rather than an optimized solver, it has been successfully applied to the matrix equation resulting from the finite element analysis. For the finer mesh of Fig. 5, the RMS error present between the computed and theoretical potentials is 0.0075. The error between the computed CPU and GPU results is negligible, suggesting no significant loss of precision between the two architectures in this case.

VII. PERFORMANCE ANALYSIS

Performance of the GPU (using Accelerator) and CPU Jacobi algorithm implementations was examined. Various mesh densities provided differing numbers of unknowns in order to compute speed factors for varying matrix sizes. The speed factor for this application is defined as the ratio of CPU to GPU processing time. Sufficient parallel computation must be performed in order to overcome the communication and setup penalties of the GPU.

For consistency, the initial approximation is set to 0.5 for each unknown. While no test for convergence is employed in the current version of the program, the number of iterations chosen to be ten times the number of unknowns for each mesh in order to assure convergence without an explicit test. For trials of less than 500 unknowns, the communication and set up time required by the GPU outweighs any performance increase, which can be noticed from the data presented in Figs. 6 and 7. The speed factor increases to 21.1 for the case of 4000 unknowns, beyond which the CPU runs were not feasible

on the test machine.

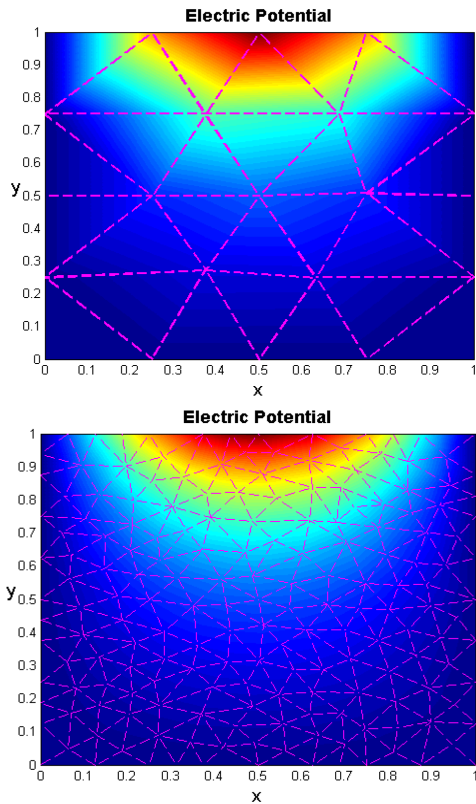

Fig. 5. Electrostatic potential in rectangular tube using different meshes.

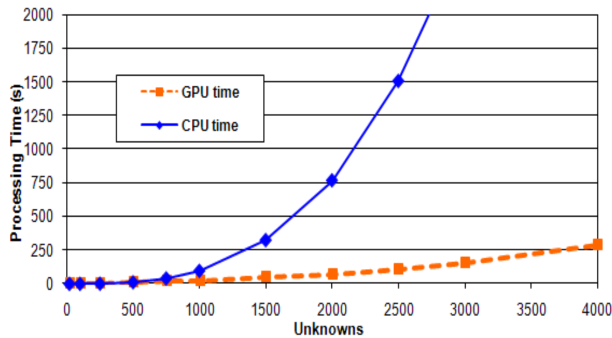

Fig. 6. Processing time for Jacobi iterations. NVidia 8800GT GPU, 2.2GHz Athlon 64 CPU.

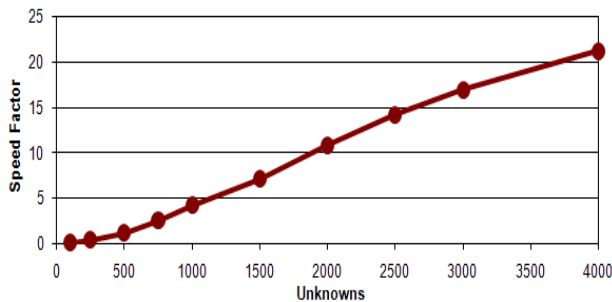

Fig. 7. Speed factor (CPU time / GPU time). NVidia 8800GT GPU, 2.2GHz Athlon 64 CPU.

Figure 8 illustrates the performance increase by generation of graphics technology using a benchmark of 2000 unknowns. The devices used in this graph spanned from a GPU produced in October 2002 (earliest technology compatible with Accelerator) to a current generation one. All other GPU results were produced using an NVidia 8800GT, while CPU results were produced using a single core of a 2.2GHz AMD Athlon 64 processor.

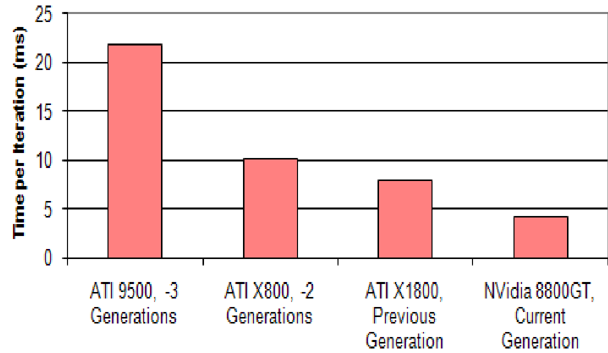

Fig. 8. Accelerator benchmark.

VIII. CONCLUSIONS

General purpose computation on graphics processing units is now available to scientific and engineering programmers through the rise of high level interfaces such as BrookGPU and Accelerator. Through the implementation of the Jacobi algorithm, both BrookGPU and Accelerator syntax and programming issues have been discussed. Performance analysis of the Accelerator program has provided insight on the current power and continuing performance increases available through the use of GPUs.

REFERENCES

[1] E. Kreyszig, *Advanced Engineering Mathematics, 8th edition*, John Wiley and Sons, New York, 1999.

[2] M. J. Inman and A. Z. Elsherbeni, “Programming video cards for computational electromagnetics applications,” *IEEE Antennas and Propagation Magazine*, vol. 47, no. 6, December 2005.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPU's: Stream computing on graphics hardware,” *Transactions on Graphics* 23, August 2004.

[4] D. Tarditi; S. Puri, and J. Oglesby, “Accelerator: using data parallelism to program GPU's for general-purpose uses,” *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems 2006*, San Jose, CA, USA, October 2006.

[5] J. R. Shewchuk, “Triangle: engineering a 2D quality

mesh generator and delaunay triangulator," *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148 Lecture Notes in Computer Science, pp. 203-222, Springer-Verlag, Berlin, May 1996.

Maxwell Woolsey was born in Oxford, Mississippi on October 25, 1981. He earned his B.S. *summa cum laude* in electrical engineering from the University of Mississippi in 2005 and is currently pursuing his M.S. degree in electromagnetics also at the University of Mississippi. His primary focus is the application of FEM techniques to high-frequency electromagnetics problems. Other related interests include audio and microwave circuit design as well as parallel computing.

W. Elliott Hutchcraft was born in Lexington, Kentucky on April 29, 1973. He earned his B.S. in electrical engineering at the University of Mississippi, Oxford, MS in 1996, his M.S. in electrical engineering at the University of Mississippi, Oxford, MS in 1998 and his Ph. D. in electrical engineering at the University of Mississippi, Oxford, MS in 2003. He is an Assistant Professor in the Department of Electrical Engineering at the University of Mississippi in Oxford, Mississippi. Dr. Hutchcraft is a member of Eta Kappa Nu, Sigma Xi, IEEE, Tau Beta Pi, Phi Kappa Phi, and ARFTG.

Richard K. Gordon was born in Birmingham, Alabama on November 26, 1959. He earned his B.S. in physics at Birmingham Southern College, Birmingham, AL in 1983, his M.S. in mathematics at the University of Illinois, Urbana, IL in 1986 and his Ph. D. in electrical engineering at the University of Illinois, Urbana, IL in 1990. He is an Associate Professor in the Department of Electrical Engineering at the University of Mississippi in Oxford, Mississippi. Dr. Gordon is a member of Eta Kappa Nu, Phi Beta Kappa, and Tau Beta Pi.