CFD2007, 15th Annual Conference of the
CFD Society of Canada, Toronto, May 2007

Implementation of an Efficient Conjugate Gradient Algorithm for Poisson Solutions on Graphics Processors

S. Menon and J. B. Perot

Department of Mechanical and Industrial Engineering, University of Massachusetts
Amherst, MA - 01003

Email: *sandeepm@engin.umass.edu*

ABSTRACT

Programmable graphics processors have achieved the distinction of being very efficient and cost-effective in terms of floating-point capacity, thereby making it an attractive option for scientific computing. In this paper, we discuss the implementation of the Conjugate Gradient iterative solver on a graphics processor. A Poisson equation is solved with the graphics processor on an unstructured three-dimensional mesh and compared on a standard CPU implementation. The implementation has also been extended to solve the Navier-Stokes equations using the Fractional Step method. Using graphics processors as math coprocessors will greatly benefit applications, such as fluid-flow solvers, which require efficient hardware and software to solve large sparse systems.

1. INTRODUCTION

Today, Graphics Processing Units (GPUs) provide the most cost-effective means of floating-point computational capacity, driven primarily by the multi-million dollar gaming industry. More recently, the capability to program these units for specialized tasks has provided increased versatility and therefore, researchers have explored this architecture for more general-purpose use. With the advent of 32-bit floating-point capabilities on more recent architectures, the feasibility for scientific computation has become quite apparent. Moreover, the current trend of hardware for graphics processors is accelerating at ever-increasing rates, easily out-performing conventional cache-based processors for applications with high arithmetic intensity [1].

The Single Instruction Multiple Data (SIMD) nature of almost all graphics applications is the primary reason why *streaming* architectures, like the one implemented on GPUs, are ideally suited for electronic games and multimedia. Most scientific applications operate on large arrays of variables, in a fashion that involves very minimal reuse of data – and are also well suited to the SIMD paradigm.

Graphics hardware also benefits from the use of improved memory bandwidth technology, which is a critical factor for scientific applications that involve large sets of data. For instance, the sequential memory access bandwidth of a GeForce 6-series GPU (which costs about $50) is about 20 GB/sec, as opposed to 6 GB/sec for a Pentium 4. Since scientific calculations are almost all memory bound this results in a direct improvement in algorithm performance. A similar comparison can be made for random memory accesses (see Fig. 1).

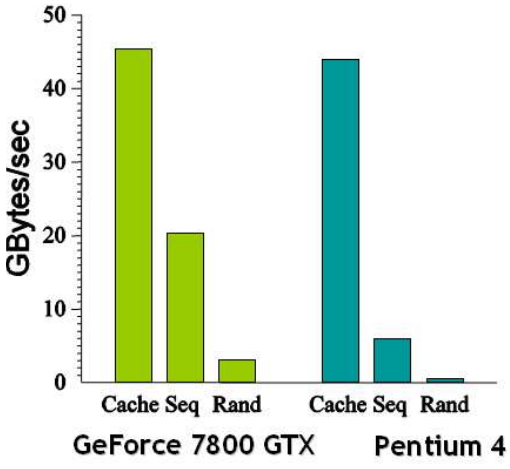

Figure 1: Memory bandwidth comparison

The conjugate gradient algorithm is a common iterative method for solving large sparse matrix systems that exhibit the property of being symmetric and positive-definite. Such systems frequently arise in the solution of discretized linear and non-linear partial differential equations such as the Poisson equation. They also form a large portion of the CPU cost of numerous incompressible flow solvers, since the solution for pressure is basically a Poisson

equation to ensure continuity. The algorithm primarily consists of three operations that must be highly efficient for the solution to be competitive in terms of CPU cost – reduction operations like a vector dot-product, the axpy operation (defined as $y = y + \alpha.x$), and the sparse-matrix multiply operation.

A practical example would be the solution to the steady-state homogenous heat-diffusion equation, given by:

$$\nabla.\mathbf{q} = 0, \text{ where } \mathbf{q} = -k\nabla T$$

We implement this on an unstructured tetrahedral mesh using a classical finite-volume method, as well as a node-based Discrete Calculus approach [2].

2. IMPLEMENTATION

Although the hardware is well-suited to the afore-mentioned application, implementation of the algorithm involves mapping it to an unusual programming model that is obviously tailor-made for the graphics model. This requires a fairly thorough understanding of the underlying hardware and its limitations. While this paradigm is explained herein, we have implemented a subset of the BLAS routines on the GPU in a manner that could be used by any scientific programmer.

2.1 The Graphics Pipeline

Graphics processors are designed with intention of accelerating the process of drawing three-dimensional geometric primitives to the screen in the form of projected two-dimensional images. The modern graphics pipeline involves several stages through which this is achieved:

The vertex processing stage: Geometric primitives are defined by vertex data, which must be appropriately transformed to the two-dimensional screen by applying an appropriate transformation matrix. Since vertex data is frequently manipulated for sophisticated effects, this stage of the pipeline is fully programmable.

The rasterization stage: This is a non-programmable stage that essentially fills transformed geometric primitives with 'fragments', which are essentially pixels that do not contain any color information.

The fragment processing stage: Fragments that are generated by the rasterization stage are passed into the programmable fragment processor, which applies appropriate coloring/lighting calculations to determine the final pixel color-value. These pixels are then sent to the frame buffer for display. Due to large amount of pixel-processing steps required in graphics applications, modern graphics processors contain a large number of fragment processing units, which form the primary workhorse of the pipeline. For this reason, our scientific calculations primarily use the fragment processing units.

2.2 Mapping algorithms to the GPU

Graphics processors use *textures* to store data in memory. Textures are usually two-dimensional bitmaps that are wrapped onto polygon faces to achieve realistic effects. In this case, they form an analogy to arrays in conventional memory layouts. However, this also means that appropriate transformations must be applied to one-dimensional CPU arrays to map them onto two-dimensional GPU equivalents. One- and three-dimensional texture layouts are also available on the GPU, but since the two-dimensional layout is closely related to the pattern of fragments generated by the rasterizer, it is much more efficient. Therefore, each scientific array is now laid out as a rectangle and defined by a width and a height rather than a single-dimensional length.

While a one-dimensional array of 12 items should probably be laid out as a 3x4 or 4x3 two-dimensional array, the mapping of a one-dimensional array of 13 items into a 2D equivalent is less obvious. In such cases, padding the array with extra elements is necessary. Consideration must also be given to the nature of the layout, which the GPU requires to be either square or rectangular. Square layouts are additionally required to a power-of-two length in each dimension, which means that the amount of padding would be too large for larger array lengths. Since the padded elements are also processed by the GPU, the overall performance would also drop in proportion to the amount of padding. Rectangular layouts, however, are less restrictive in this regard.

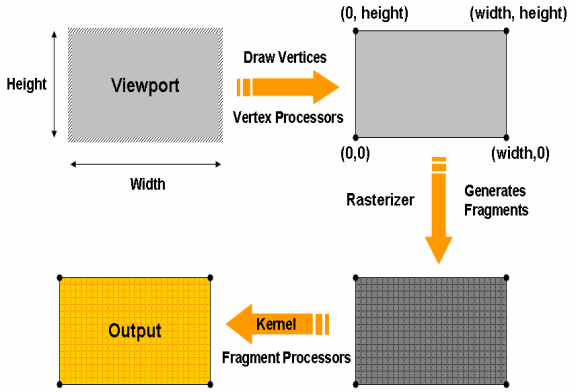

Figure 2: Mapping algorithms to the graphics pipeline

The computational kernels of the algorithm are programs that are written for the vertex and fragment processors that perform an operation identically on all entities being processed. By defining a viewport with dimensions equal to that of the array and drawing an equally sized rectangular region on the screen, the rasterizer generates a fragment for each element in the output array and applies the kernel programs to them. The results of the computation can then be redirected to another array which is attached to the frame buffer. In doing so, these computed results can be used as inputs to a subsequent operation, thereby establishing a feedback mechanism (this is represented in Figure 2).

2.2.1 Reduction

Reduction operations operate on large streams of data to produce a single result. Examples include sum, vector dot-product, min and max. Since GPUs are optimized for read-only / write-only operations in memory, such operations involve acting on adjacent data for each fragment and rendering to another rectangular array of half its length in each dimension. In doing so, each element in the output array corresponds to the local sum of four elements in the input array; which is also beneficial because it helps to minimize errors due to round-off. This process is then repeated in subsequent passes until a small size results. After the array is reduced, one possible approach is to padded the reduced array with zeros to the next appropriate power-of-two dimension, and finally reduced to a single value, as shown in Fig. 3. This is done to ensure reduction operations on arbitrarily- sized rectangular textures.

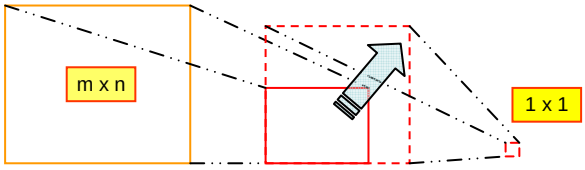

Figure 3: Reduction on arbitrary size rectangular textures with the padding approach

Another approach for the final reduction step is to read the reduced array back to the CPU and complete the summation there, since the CPU is relatively more efficient for smaller sets of data. This proves to be a more efficient approach.

Padding the array with zeros does not interfere with reduction operations like the sum and the dot-product, but is an issue when operations like a min or a max is considered; since zero might mistakenly be reported as the min / max value of the array. Thus, for min / max operations, the array is padded with the first element of the array.

2.2.2 Axpy ($y = y + \alpha.x$)

The add-and-multiply operation is implemented on the GPU with relative ease, since both the 'y' and 'x' vector-fields are sized equally in both dimensions, and a 1:1 correspondence exists for all fragments. The operation involves a sequentially accessed memory-fetch from either array, followed by a write operation to the output attached to the frame buffer.

2.2.3 Sparse-Matrix Multiply

The sparse-matrix multiply in the conjugate gradient loop is decomposed into multiple sparse matrix operations that act on variables located either on faces or cells in the unstructured tetrahedral mesh for the finite volume discretization scheme, or at nodes for the node-based discrete calculus approach. In the case of a finite volume technique, the discretized form of the steady-state homogenous heat-diffusion equation is given as:

$$\mathbf{D}\ (-k\,\mathbf{G}\,T_C) = 0$$

where **G** and **D** are the discrete gradient and divergence operators, respectively.

The mesh-connectivity information is maintained in separate lists that establish the relationship between various entities (for example, cell-face, cell-node, or face-edge connectivity, etc).

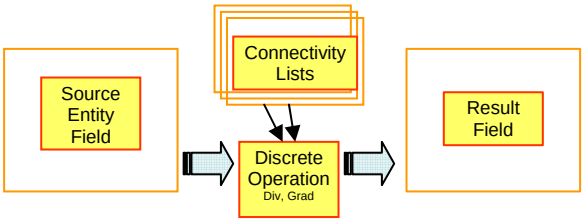

Figure 4: Sparse-matrix multiply operations

The discrete operator is represented as a fragment program that performs random-access memory-fetches from a source-field based on indices (represented as 2D texture coordinates) using the appropriate connectivity-list, and writes to the corresponding output fragment in the resultant field. In a cell-based finite-volume discretization for example, a discrete gradient operator takes values located at cell centers to compute the flux at cell-faces. A discrete divergence operator does the opposite – it takes values located at cell-faces to compute the divergence of the variable at the cell-centers.

A graphics processor can efficiently handle "gather" operations (`a = x[i]`), whereas an indirect write to memory, i.e., a "scatter" operation (`a[i] = x`) is not natively supported. The discrete divergence

operation is inherently a scatter operation when implemented on the CPU, since it shares the same mesh-connectivity structure as the gradient. On the graphics processor however, it must be reformulated as a gather operation and therefore requires a new connectivity structure to be calculated (once).

The node-based discretization approach assumes a control volume which surrounds each node in the mesh, called dual-mesh cells. For more details on the node-based discrete-calculus approach, refer to reference [2]. An interpolation operator takes vector values located at cell-centers to compute an interpolated flux located at faces, while the complementary integration operator computes an integrated vector value at cell-centers from flux values at faces. Similar approaches can be taken to construct other operators such as the discrete curl, for instance.

2.2.4 Boundary conditions

Incorporating boundary conditions in the solver is a slightly more involved procedure. Boundary conditions fall under the category of a scatter operation, since the specification of a Dirichlet or Neumann condition involves writing to specific locations in memory.

The workaround in this case would be to use the vertex-processor for the scatter operation. By drawing a vertex to a specific coordinate location in the framebuffer and specifying the boundary value along with it, the effect of a scatter operation is achieved. However, as individual points have to be drawn using OpenGL library calls from the CPU; this proves to be a major bottleneck especially for geometry which involves a large ratio of boundary to interior entities. This is alleviated to a small extent if the boundary coordinates and their associated values are pre-computed on the CPU and stored in a vertex buffer, which resides on GPU memory. Unless the boundary conditions are time-varying, this proves to be a fairly efficient approach. Nevertheless, it is only a fraction of the efficiency achieved by the data-streaming paradigm. For details on this implementation, refer [3].

2.2.5 Navier-Stokes implementation

The conjugate gradient algorithm described above has also been used to implement an unsteady, incompressible Navier-Stokes solver on the graphics processor using the classical Fractional-Step approach. This method was first introduced independently by Chorin [4] and Temam [5] as a practical approach to the solution of incompressible fluid-flow. The approach has a few major drawbacks, including the fact that it exhibits poor temporal accuracy (first order accurate). However, we will focus on hardware efficiency rather than accuracy at this point, since other methods which overcome this difficulty exist, and can be easily incorporated [6].

As pointed out by Perot [7], the discretized incompressible Navier Stokes equations can be viewed as a block LU decomposition in the form:

$$\begin{bmatrix} A & G \\ D & 0 \end{bmatrix} \begin{pmatrix} v^{n+1} \\ p^{n+1} \end{pmatrix} = \begin{pmatrix} r^n \\ 0 \end{pmatrix} + \begin{pmatrix} b.c's \\ b.c's \end{pmatrix}$$

where G and D are the discrete gradient and divergence operators mentioned earlier, and A is a sub-matrix whose structure depends on the form of temporal and spatial discretization. A typical structure for *A* (such as the one incorporated here), is to treat diffusion implicitly for stability and an explicit advection term along with a temporal term if the flow is unsteady. This system can be decomposed further to yield the fractional step method:

$$\begin{bmatrix} A & 0 \\ D & -DA^{-1}G \end{bmatrix} \begin{bmatrix} I & A^{-1}G \\ 0 & I \end{bmatrix} \begin{pmatrix} v^{n+1} \\ p^{n+1} \end{pmatrix} = \begin{pmatrix} r^n \\ 0 \end{pmatrix} + \begin{pmatrix} b.c's \\ b.c's \end{pmatrix}$$

This leads to the estimation of an intermediate velocity v^* which is non-divergence free, and must be corrected for pressure at time *n* by solving a Poisson equation.

The chosen domain in this case is a three-dimensional driven-cavity flow with typical conditions of no-slip and zero pressure-gradient normal to the walls.

3. RESULTS

The algorithms were tested with an nVidia 6600GT graphics card, and benchmarked against an AMD Athlon64 running at 1.81 GHz and a Front-Side Bus of 400 MHz. OpenGL was used as the underlying graphics-API, as an interface between the application and the underlying hardware.

Figure 5 compares the performance on both processors for a sum-reduction operation on a large array of data, showing GFlops as a function of problem-size. The plot exhibits typical behavior, with the GPU (triangle-symbols) outperforming the CPU (square-symbols) on substantial problem-sizes, by a factor of about 350%.

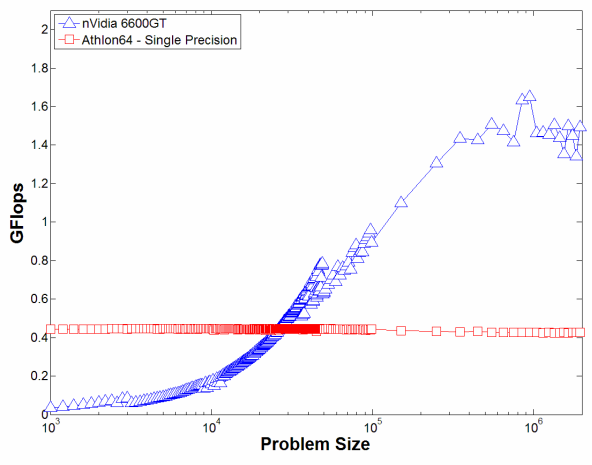

Figure 5: Sum-reduction comparison

Figure 6a and 6b shows a similar plot for a vector dot-product and axpy operation respectively. The dot-product also achieves a speed-up of about 350%, while the axpy operation achieves a speed-up of about 500% on problem sizes of interest. For a nVidia GeForce 6600GT, the sequential memory-access bandwidth is specified as 16GB/sec. One half of this bandwidth is for reading and the other half for writing. This translates into 2 GigaWords / sec (single precision) read and/or write. The `sum` and `axpy` operations confirm that the GPU can operate at close to this peak performance. For the reduction operations, peak performance requires vector lengths of 100k or larger. The simpler operations like the `axpy` can perform at peak speeds for array sizes down to as small as 10k.

The drop in performance for the CPU is clearly evident for larger problem sizes, which confirms the observation that it is poorly suited for scientific applications (which tend to involve large data sets)..

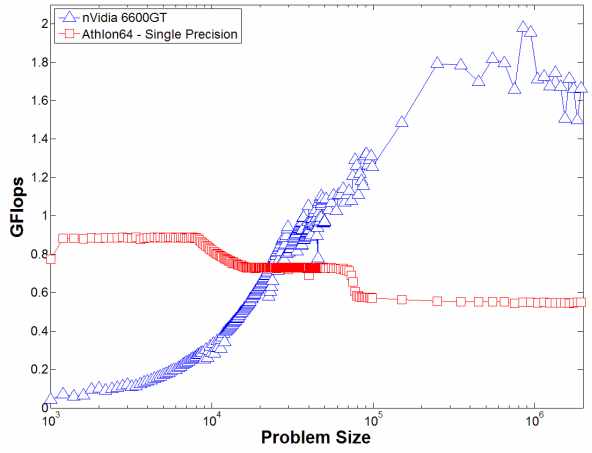

Figure 6a: Vector dot-product comparison

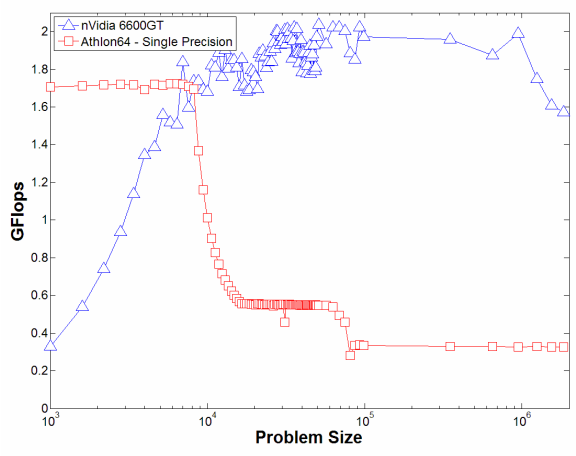

Figure 6b: Axpy comparison

Figure 7a and 7b shows the sparse-matrix-multiply comparison for a gradient and integration operation. The interpolation operator involves two dot-products and an addition operation for each face, as opposed to a single subtraction for the gradient. This would explain the higher performance. Owing to the random texture-fetching nature of the sparse matrix multiplication operation, the GPU performance drops with an increase in problem-size, but the GPU still outperforms the CPU by a factor of about 250%. Random memory accesses on the GPU are roughly 6 times slower than sequential access and so sparse matrix operations are also slower by roughly this factor. This can be alleviated to a small extent by a bandwidth reduction process that re-numbers the connectivity indices; thereby allowing increased memory-accesses within the cache of the processor.

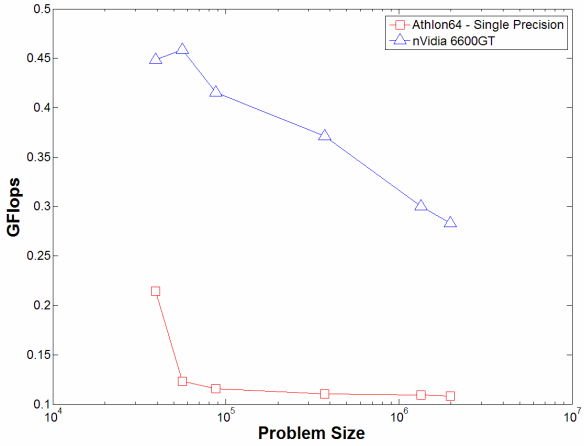

Figure 7a: Gradient operator comparison

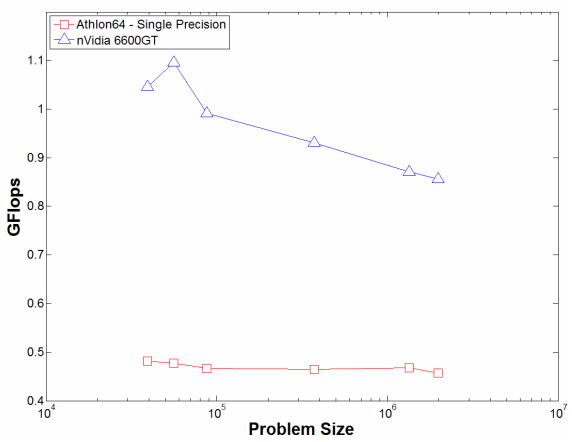

Figure 7b: Integrate operator comparison

When a full scientific algorithm is implemented using these GPU operators the performance is dominated by sparse matrix multiplies. Figure 10 shows the computation-time per CG iteration comparison between both processors for the Conjugate Gradient algorithm for various problem-sizes, using the finite-volume cell-based discretization. The computation is performed on a realistic geometry (a crankshaft) for various mesh-resolutions, shown in Fig. 8. Dirichlet boundary conditions are specified at the ends of the crankshaft, while the rest of the walls are insulated. Typical contours for temperature are shown in Fig. 9.

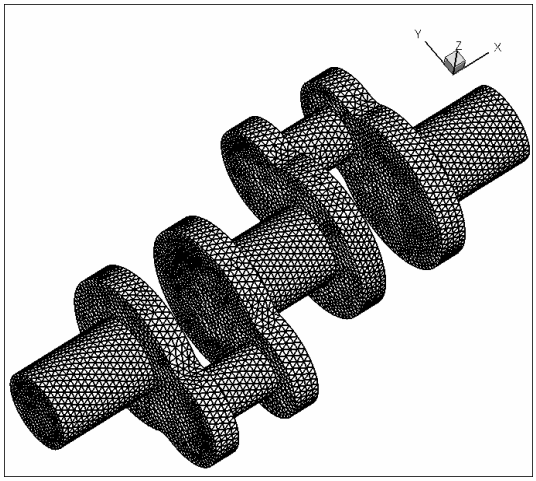

Figure 8: Crankshaft mesh geometry used for the computation

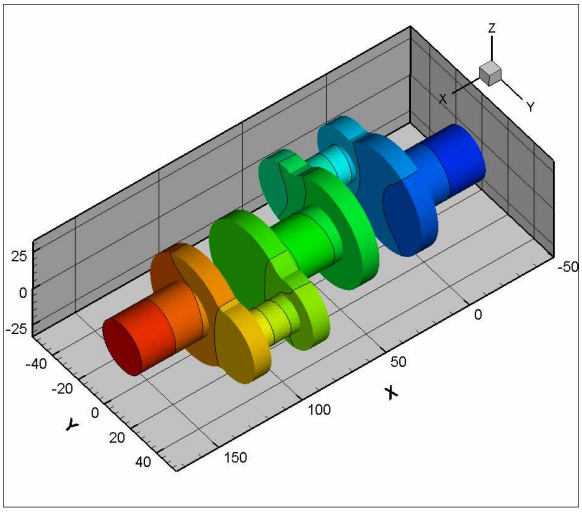

Figure 9: Temperature contours along the crankshaft

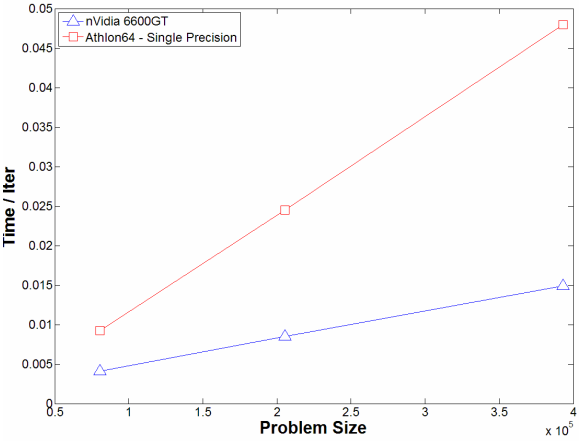

Figure 10: Performance of the Conjugate Gradient algorithm for the Poisson Solution (cell-based finite-volume discretization)

A similar result is obtained for the node-based discrete-calculus approach shown in Fig. 11, where the time per iteration is compared for both the architectures. Clearly, the difference in computation cost is significant for larger problem sizes, with the graphics processor outperforming the CPU by a factor of about 2.

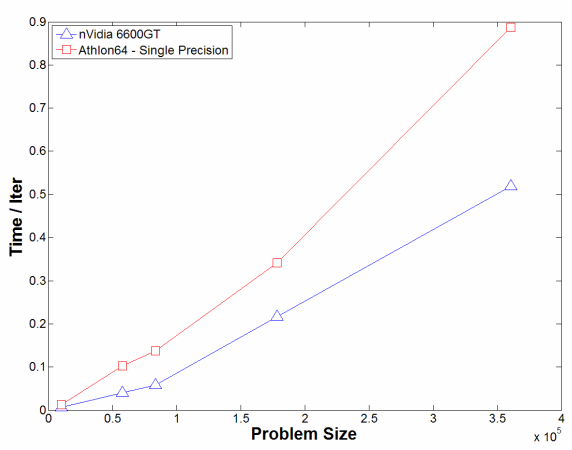

Figure 11: Performance of the Conjugate Gradient algorithm (node-based discretization)

For the solution of the driven-cavity flow, since the diffusion term is treated implicitly, each time-step involves four calls to the conjugate gradient solver – three for the momentum equations and the other for pressure. The performance results for the pressure solver are shown in Fig. 12.

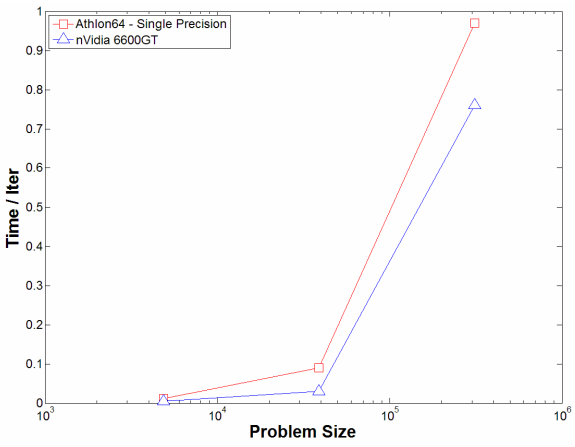

Figure 12: Performance of the Conjugate Gradient algorithm for Navier Stokes

DISCUSSION

The solution cost for these algorithms is dominated by memory accesses and therefore, the performance of the algorithm is largely dictated by the memory bandwidth of the architecture.

The nVidia GeForce 8800 GPU has a rated memory bandwidth of 81GB/sec and so it would be expected to achieve roughly 10 Gflops for basic math operations and 2-5 GFlops for the sparse matrix operations. Since up to four graphics processors can be placed on a single motherboard, this could result in almost an order of magnitude performance increase over currently available commodity CPUs.

Owing to the streaming nature of the hardware, graphics processors have clearly demonstrated the ability to act as very efficient and cost-effective math co-processors for problems involving a large amount of numerical effort.

ACKNOWLEDGEMENTS

Partial financial support for this work was provided by the Office of Naval Research (Grant N00014-01-1-0267), the Air Force Office of Scientific Research (Grant FA9550-04-1-0023) and the National Science Foundation (Grant CTS-0522089).

REFERENCES

[1] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., and Purcell, T. A survey of general-purpose computation on graphics hardware. *Eurographics*, August 2005, pp. 21-55.

[2] Perot, J. B., and Subramanian, V. Discrete Calculus methods for diffusion. J. Comput. Phys., doi: 10.1016/j.jcp.2006.12.022, 2006.

[3] Menon, S. Using alternative hardware for scientific calculations. Masters Thesis, University of Massachusetts, Amherst, 2007.

[4] Chorin, A. J. Numerical solutions of the Navier Stokes equations, *Math. Comput.* **22**, 745 (1968).

[5] Temam, R. On the approximation of the Navier Stokes equations using the projection method, *Arch. Rat. Mech. Anal.* **32**, 377 (1969)

[6] Chang, W., Giraldo, F., and Perot, J. B. Analysis of an Exact Fractional Step method. *J. Comput. Phys.* **180** (1), 2002, pp. 183-199.

[7] Perot, J. B. Analysis of the Fractional Step method. *J. Comput. Phys.* **108** (1), 1993, pp. 51-58