Accelerating Protein Structure Recovery Using Graphics Processing Units

Bryson R. Payne[1], G. Scott Owen[2], and Irene Weber[2]

[1] Georgia College & State University, Department of ISCM, Milledgeville, GA 31061
bryson.payne@gcsu.edu
[2] Georgia State University, Department of Computer Science, Atlanta, GA 30303
owen@siggraph.org, iweber@gsu.edu

Abstract. Graphics processing units (GPUs) have evolved to become powerful, programmable vector processing units. Furthermore, the maximum processing power of current generation GPUs is technically superior to that of current generation CPUs (central processing units), and that power is doubling approximately every nine months, about twice the rate of Moore's law. This research represents the first successful application of GPU vector processing to an existing scientific computing software package, specifically an application for computing the tertiary (3D) geometric structures of protein molecules from x-ray crystallography data. A framework for applying GPU parallel processing to other computational tasks is developed and discussed, and an example of the benefits of taking advantage of the visualization potential of newer GPUs in scientific computing is presented.

1 Introduction

The graphics processing units of the past four years have increased the capabilities of previous generations by a factor of two every six to nine months. Programmable graphics processing units (GPUs) are commonly included as hardware components in new computer workstations, including those workstations used for scientific computing. The current generation of GPUs is roughly equivalent to or greater than the processing power of current CPUs (central processing units), but that power is rarely used to its full capabilities. Numerous advances have been made recently in applying the GPU to non-graphical parallel matrix processing tasks, such as the Fast Fourier Transform (FFT) [6]. This research seeks to apply the vector processing power of a GPU to automatically computing the 3D geometry of proteins from x-ray crystallography data in an existing software package.

Our primary goal is to apply distributed GPU-CPU computation to automated tertiary structure fitting in protein crystallography, a process that normally takes several hours or even days to execute on a sequential CPU. We proposed to use existing software, ARP/wARP [11], that is already fully-featured and widely used in the industry rather than producing a competing package for several reasons. First, change is difficult to engender in any field, especially where the learning curve for effective utilization of new software is steep. Second, the software packages in existence are already well-suited to the needs of researchers in protein

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 451 – 459, 2005.

crystallography. Producing a complete product solely for the purpose of demonstrating GPU acceleration would have required hundreds of thousands of lines of programming to match the capabilities of packages already widely accepted and used.

Our secondary goal is to produce a reusable, portable framework for applying GPU computation to scientific computing problems in other fields. As GPUs gain acceptance as parallel vector processing units, a generalized framework for taking advantage of available GPU vector processing power in other scientific computing applications is needed. By documenting and examining the steps taken to add GPU parallel computation to an existing scientific computing package, it is hoped that future researchers will be able to reapply that framework to other existing and new software packages in other fields.

2 Literature Review

The first programmable consumer GPUs became available less than four years ago. The first generation of programmable GPUs was not well-suited to general-purpose computation for several reasons. First, they allowed access to only selected portions of the graphics pipeline and had no easily accessible off-screen rendering capabilities. Second, they had to be programmed in GPU-specific assembly code, with no standardization across manufacturers. Third, their limited accuracy of 8 bits-per-pixel combined with their slower clock speeds and memory accesses, as well as smaller memory sizes, compared to CPUs made them unattractive to general-purpose computing researchers who needed fast 32-bit floating point operations as a minimum point of entry.

By late 2002, however, a C-like programming language for the GPU, named Cg [5], had been developed for cross-platform GPU programming. Cg contained constructs for looping and conditional branching, required for most general-purpose computing, but GPU hardware took another two years to catch up to the capabilities provided for in the Cg language. Only in the NVIDIA GeForce FX6800 series GPU, released in late 2004, was it first possible to take advantage of true conditional branching on the GPU, as well as handle loops or programs that consisted of more than 1024 total instructions per pixel [8]. These advances enabled the research in this paper, and the previous work [9] upon which it is based.

2.1 The Bioinformatics Problem

X-ray crystallography is the most commonly used method for determining the 3D structure of proteins in bioinformatics. The 3D structure of a protein is valuable because it determines many of the protein's properties [4], making structure information useful in drug discovery research as well as many other fields in the biosciences. ARP/wARP [11] is the most popular, and most accurate, software package for automatically determining the structure of proteins from x-ray crystallography data [1]. ARP/wARP is used by researchers in our own Bioinformatics Lab so this program was the focus of our efforts.

ARP/wARP makes use of an iterative structure refinement process by means of a program called Refmac [7]. In run-time analysis, Refmac consumed as much as 83% of the run time of a typical 3D structure computation under ARP/wARP, and the source code for Refmac is freely available. Refmac was, therefore, selected as the test bed for our GPU acceleration research, as well as the subject of our investigation into visualization advantages of current GPUs in scientific computing.

3 Implementation

We first desired a proof of concept to test whether the stated superior performance of GPUs was possible to achieve in situations well-suited to the GPU. Our first implementation steps were to demonstrate that the GPU was at least as fast as the CPU at performing 2D convolutions like averaging filters, edge detection, and the Gaussian smoothing filters before implementing more complex algorithms for scientific computing. Straightforward algorithms for convolutions on the GPU and on the CPU were developed for comparison across a wide variety of matrix and image sizes (32x32 to 4096x4096).

3.1 Building GPU Code for ARP/wARP

In a trial run on a 247-residue protein molecule, ARP/wARP was able to fit 238 of the 247 peptides to a model with 98% connectivity in 18 hours on a 2.8 GHz CPU, using 1200 iterations of refinement. As previously mentioned, it was determined through run-time analysis that the most time-consuming process in ARP/wARP for our purposes is the Refmac refinement step, which uses an open-source molecular refinement program [7]. Therefore, we focused our attention on applying distributed GPU-CPU processing to the Refmac algorithms.

Run-time profiling with the GNU compiler tool `gprof` yielded two subprograms in Refmac that consumed over 30% of the total processing time of the program, `indens` and `prot_shrink`. Due to the fact that `indens` could take up to 25% of the total runtime, while `prot_shrink` accounted for 10% or less of the runtime across a sample set of three proteins of varying sizes, attention was given to `indens` first. We set out to translate `indens` from its native FORTRAN for the CPU to Cg on the GPU.

3.2 Toward a Framework

Integrating GPU-CPU distributed computation with existing bioinformatics software introduces two significant hurdles: commingling C and Cg code for the GPU with the native FORTRAN of Refmac, and re-mapping 1D, 2D, and 3D matrix computation from the CPU to 2D texture calculations on the GPU. Our goal with respect to these challenges is to produce a reusable framework for adding GPU acceleration of matrix computation to existing computational tools across any field of scientific computing. We also hope to demonstrate the advantages of adding 3D visualization, which the GPU handles optimally, to general purpose and scientific computing software packages.

4 Experimental Results

The GPU proved to be much faster at straightforward convolutions like Gaussian smoothing, averaging filters, and edge detection. In the case of a 3x3 convolution filter, the GPU (an NVIDIA GeForce FX6800) was from 10 to 90 times faster at high resolutions (2048x2048) than the CPU (a 2.8 GHz Intel Pentium 4), even after communication between main memory and the GPU was accounted for (see Figure 1).

Most image processing packages do not perform straightforward convolution algorithms, however. On the CPU, it is usually several times faster to use the Fast Fourier Transform (FFT) and perform a matrix multiplication, which is much faster on the CPU than a convolution, then use the inverse Fourier Transform to reacquire the resulting image. The GPU is still a full order of magnitude faster than the CPU even after Fourier optimization, which raises the possibility that Fourier processing for convolutions could become obsolete.

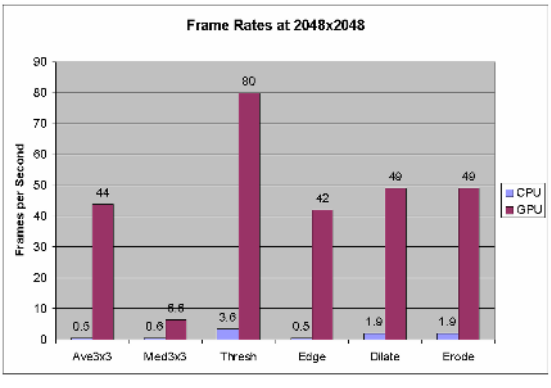

Fig. 1. Comparison of processing speed of various operations on CPU vs. GPU

The GPU is especially well-suited to performing 2D convolutions and morphological masking and filtering operations. Furthermore, programming the GPU version of these algorithms is a straightforward process, allowing the developer to access pixel neighborhoods using a relative indexing paradigm rather than a complicated modular arithmetic scheme for referencing 2D array elements in main memory. Figure 2 shows the GPU code in Cg for performing a 3x3 averaging filter. Figure 3 shows the same algorithm in a straightforward CPU implementation in C.

Notice that the Cg code shows relative texture lookups using vectors like (-1, 1) to denote left one, down one from the current pixel. The C version, on the other hand, is more difficult for several reasons. First, the C version must treat each color component (red, green, and blue, or *RGB*) as separate computations, while the GPU computes all three components simultaneously as three-component vectors of type `float3`. Second, the C method of computing array positions is one-dimensional rather than the 2D relative indexing of Cg. Therefore, non-intuitive modular arithmetic is necessary to resolve the location in main memory of a pixel that is one

unit left and one down from the current pixel. More information on digital image processing on GPUs can be found in a related paper [10].

```
float3 a[9];
a[0] = texRECT(image, texCoord + float2(-1, 1));
a[1] = texRECT(image, texCoord + float2(0 , 1));
a[2] = texRECT(image, texCoord + float2(1 , 1));
a[3] = texRECT(image, texCoord + float2(-1, 0));
a[4] = texRECT(image, texCoord + float2(0 , 0));
a[5] = texRECT(image, texCoord + float2(1 , 0));
a[6] = texRECT(image, texCoord + float2(-1,-1));
a[7] = texRECT(image, texCoord + float2(0 ,-1));
a[8] = texRECT(image, texCoord + float2(1 ,-1));
color = (a[0]+a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]+a[8])/9.0;
```

Fig. 2. A Cg 3x3 averaging filter function

```
int a1,a2,a3,a4,a5,a6,a7,a8,a9;
int wd=TexInfo->bmiHeader.biWidth;
int ht=TexInfo->bmiHeader.biHeight;
int m = wd*ht*3;
for (r=0; r<m;r++)
{
  a1=(r-(wd+1)*3+m)%m;
  a2=(r-(wd)*3+m)%m;
  a3=(r-(wd-1)*3+m)%m;
  a4=(r-3+m)%m;
  a5=r;
  a6=(r+3)%m;
  a7=(r+(wd-1)*3)%m;
  a8=(r+(wd)*3)%m;
  a9=(r+(wd+1)*3)%m;
  TexBits2[r] = (TexBits[a1]+TexBits[a2]+TexBits[a3]+TexBits[a4]+TexBits[a5]+
    TexBits[a6]+ TexBits[a7]+TexBits[a8]+TexBits[a9])/9;
}
```

Fig. 3. A simple subroutine in C for computing the 3x3 averaging filter

4.1 GPU Acceleration of Refmac

The GPU representation of the `indens` algorithm that operates on 1D arrays suffered from the same type of problem as the CPU operating on 2D arrays, only in reverse. Whereas the CPU version was able to use straightforward addressing to loop through the arrays, the GPU had to use more costly modular arithmetic to determine texture coordinates in 2D to correspond to the 1D positions of the array elements being processed. Because of the inefficiency of this additional computation, a speedup on the order of 90 times, as seen in straightforward convolutions, was not possible. However, due to the fact that the original FORTRAN code was not optimized for CPU caching, either, a significant advantage in speed was still afforded to the GPU version of the code.

In run-time analysis of the two algorithms across three proteins of varying sizes, the GPU version was 1.8 to 2.6 times faster than the GPU at the `indens` subprogram than the CPU. Figure 4 below shows the comparison, with the GPU version broken

down into two components: CPU time (pre-processing and memory transfers from the CPU to the GPU and back again) and GPU time (the actual processing time of the algorithm on the GPU).

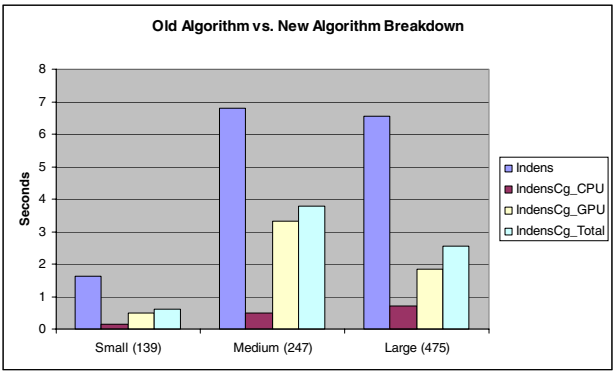

Fig. 4. Comparison of CPU and GPU versions of `indens` algorithm. The GPU version (`indensCg`) is broken down into CPU time (texture transfer, readback, etc.) and GPU processing time

While this demonstrated the significant potential for acceleration of a single algorithm, the 1.8 to 2.6 times speedup in the indens subprogram translated to only one to two hours of time savings on a 16 to 48-hour protein structure recovery. Even if the `prot_shrink` algorithm could yield an optimal 90:1 speedup, due to the fact that it consumes only 8-10% of each Refmac iteration, another hour or two would be the maximum time savings possible in this application. Clearly the acceleration potential of the GPU holds great promise in certain settings, but achieving significant gains in general-purpose applications consisting of hundreds of algorithms and subprograms may be achieved only by rewriting major portions of the code.

4.2 Adding Visualization to Refmac

The area of GPU potential still remaining to be explored was visualization. We had already observed, from poring through log files from ARP/wARP, that the program could resolve a great number of residue positions in a protein rather quickly, then reach a plateau after which few of the remaining residues could be placed no matter how many iterations were provided. However, ARP/wARP has no integrated visualization tool that would enable the researcher to visually inspect a protein for suitability and terminate the structure-building process early.

In the case of the smaller protein sample tested, a protein consisting of 139 amino acids, or residues, 134 of the residues had been correctly placed after only 300 refinement cycles, a plateau that went unsurpassed in the remaining 900 iterations of refinement. In other words, 12.6 hours of the 16.8 hour GPU-accelerated run time for the smaller protein were unnecessary; a 96% complete solution was achieved in only 4.2 hours.

By adding a system call once per Refmac iteration to a simple visualization program, RasMol [2], it was possible to view updated results of the previous iteration every one to four minutes, allowing the researcher to visually inspect the progress of the structure-building process from a different workstation across the room without having to sift through log or other intermediate files. Figure 5 shows some of the simple visualization styles provided by RasMol.

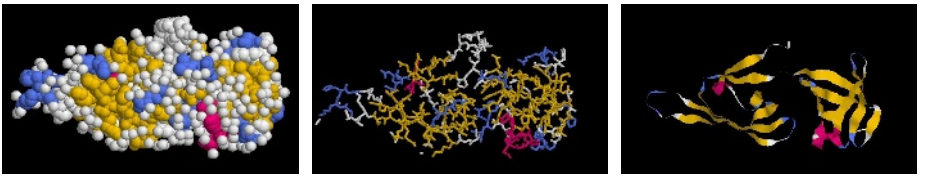

Fig. 5. Three visualization views of a protein with 134 residues in RasMol (space fill, stick, and ribbon views)

By providing this visual information automatically with no additional work on the part of the researcher, it was possible to halt each structure at 90% completion for a total time savings of 45 out of 80 hours of runtime, or 56%. It took only 34 minutes, 8 hours, and 26 hours, respectively, to resolve proteins with 139, 247, and 475 residues to 90% completion, versus a total of 80 hours and 50 minutes to allow them to run to 96% completion. For a researcher skilled at manually placing the remaining residues, the time saved by having automatic visualization built in to the model-building process would have been over 10 times greater than the time saved by accelerating the structure-building process using the GPU.

4.3 Developing a Framework

There were five main steps in the implementation used here that could be replicated for use in any existing or new software development in order to make use of GPU acceleration. First, run-time analysis, with a tool like `gprof`, is used to determine which subroutines consume the majority of processing time. Second, the source code of each high-use subroutine is examined for suitability for GPU implementation (small number of large vectors or matrices in the computation, etc.). Third, the selected algorithms are mapped to a 2D texture-rendering problem on a per-pixel basis, which is the MIMD-processing model of the GPU. Fourth, the source code is modified to include calls to the new GPU versions of the algorithms selected for acceleration. Finally, run-time testing is used to determine the speedup, if any, and adjustments are made, if needed, by revisiting steps two through four as needed.

While these steps are not trivial, they represent a model for future application of GPU vector processing to existing and new scientific and general-purpose computing packages. Possibilities for simplifying the framework in certain domains are given in the following section.

5 Conclusions and Future Work

This research presents the first successful application of GPU parallel vector processing to an existing scientific computing software package. In addition to implementing GPU-based acceleration of the given software, a visualization component was added to the process, resulting in a total time savings on the order of 60% over the CPU-only version with no visualization included.

In convolution-based operations from digital image processing (DIP), the GPU showed a speedup of up to 90:1 over the same implementation on the CPU, and a full order of magnitude of improvement over the FFT-optimized version of the algorithms on the CPU. Because 2D DIP is well-suited to GPU acceleration, a fruitful area for future research would be developing an image processing API at the same level of abstraction as the Brook language [3] for stream processing. An API or multi-step compiler system like Brook designed specifically for DIP operations could be a significant step toward general-purpose utilization of the GPU as a parallel vector processor.

While there exists significant potential for accelerating scientific computing by using GPUs for parallel vector computation, three main obstacles still exist. First, translating algorithms from the original implementations to 2D, GPU versions is not a trivial process. Second, not all vector processing algorithms are good candidates for GPU acceleration, including those with excessive branching and conditional logic. Finally, differences in graphics hardware and drivers still pose a problem in implementing the same algorithm across different platforms. A significant move toward standardization in programming langages for GPUs will be necessary if GPUs are to be used for scientific computing on a consistent basis.

References

1. Badger, J.: An evaluation of automated model-building procedures for protein crystallography, *Acta Crystallographica*. International Union of Crystallography (2003) 823-827.
2. Bernstein, H.J., Sayle, R.: RasMol Molecular Graphics Visualization Tool. Available online at http://openrasmol.com. (2000).
3. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware, *Proceedings of ACM SIGGRAPH 2004*. ACM Press, (2004) 777-786.
4. Lunney, E.A.: Computing in Drug Discovery: The Design Phase, *Computing in Science & Engineering 3(5)*. IEEE, (2001) 105-108.
5. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard, M.J.: Cg: a system for programming graphics hardware in a C-like language, *ACM Transactions on Graphics 22(3)*, pages 896-907. ACM Press, July 2003.
6. Moreland, K., Angel, E.: The FFT on a GPU, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association (2003) 112-119.
7. Murshudov, G.N., Vagin, A.A., Dodson, E.J.: Refinement of Macromolecular Structures by the Maximum-Likelihood Method. *Acta Crystallographica*. International Union of Crystallography, (1997) 240-255.

8. NVIDIA Corporation: GeForce 6800 Product Overview. Available online at http://www.nvidia.com/object/IO_12464.html. (2004).
9. Payne, B.R.:Accelerating Scientific Computation in Bioinformatics by Using Graphics Processing Units as Parallel Vector Processors. (Doctoral dissertation, Georgia State University, 2004). *Dissertation Abstracts International* (UMI. No. pending)
10. Payne, B.R., Owen, G.S., Belkasim, S.O.: Digital Image Processing on GPUs. Submitted to the Fourth International Workshop on Computer Graphics and Geometric Modeling, CGGM'2005. Emory University, Atlanta, USA, May 22-25, 2005.
11. Perrakis A., Morris R., Lamzin V.S.: Automated protein model building combined with iterative structure refinement. *Nature Struct. Biol.*, (1999) 458-463.