Accelerating the MilkyWay@Home volunteer computing project with GPUs

Travis Desell[1], Anthony Waters[1], Malik Magdon-Ismail[1], Boleslaw K. Szymanski[1], Carlos A. Varela[1], Matthew Newby[2], Heidi Newberg[2], Andreas Przystawik[3], and David Anderson[4]

[1] Department of Computer Science,
Rensselaer Polytechnic Institute, Troy NY 12180, USA
[2] Department of Physics, Applied Physics and Astronomy,
Rensselaer Polytechnic Institute, Troy NY 12180, USA
[3] Institut für Physik, Universität Rostock, 18051 Rostock, Germany
[4] U.C. Berkeley Space Sciences Laboratory,
University of California, Berkeley, Berkeley CA 94720, USA

Abstract. General-Purpose computing on Graphics Processing Units (GPGPU) is an emerging field of research which allows software developers to utilize the significant amount of computing resources GPUs provide for a wider range of applications. While traditional high performance computing environments such as clusters, grids and supercomputers require significant architectural modifications to incorporate GPUs, volunteer computing grids already have these resources available as most personal computers have GPUs available for recreational use. Additionally, volunteer computing grids are gradually upgraded by the volunteers as they upgrade their hardware, whereas clusters, grids and supercomputers are typically upgraded only when replaced by newer hardware. As such, MilkyWay@Home's volunteer computing system is an excellent testbed for measuring the potential of large scale distributed GPGPU computing across a large number of heterogeneous GPUs. This work discusses the implementation and optimization of the MilkyWay@Home client application for both Nvidia and ATI GPUs. A 17 times speedup was achieved for double-precision calculations on a Nvidia GeForce GTX 285 card, and a 109 times speedup for double-precision calculations on an ATI HD5870 card, compared to the CPU version running on one core of a 3.0 GHz AMD Phenom(tm)II X4 940. Using single-precision calculations was also evaluated which further increased performance 6.2 times for ATI card, and 7.8 times on the Nvidia card but with some loss of accuracy. Modifications to the BOINC infrastructure which enable GPU discovery and utilization are also discussed. The resulting software enabled MilkyWay@Home to use GPU applications for a significant increase in computing power, at the time of this publication approximately 216 teraflops, which would place the combined power of these GPUs between the 11th and 12th fastest supercomputers in the world[5].

---

[5] http://www.top500.org/list/2009/06/100

1 Introduction

General-Purpose computing on Graphics Processing Units (GPGPU) is an emerging field of research which allows software developers to utilize the significant amount of computing resources GPUs provide for wide range of applications. While traditional high performance computing environments such as cluster, grids and supercomputers require significant architectural modifications to incorporate GPUs, volunteer computing grids already have these resources available as most personal computers have GPUs available for more recreational use. Additionally, volunteer computing grids are gradually upgraded by the volunteers as they upgrade their hardware, whereas clusters, grids and supercomputers are typically just replaced by newer hardware. As such, MilkyWay@Home's volunteer computing system is an excellent testbed for testing and measuring the potential of large scale distributed GPGPU computing across a large number of heterogeneous GPUs.

There are two main challenges in effectively utilizing GPUs in a volunteer computing setting. The first is developing efficient applications that appropriately utilize the GPU hardware. Additionally, the MilkWay@Home application requires a high degree of floating point accuracy to compute correct results. Only recent generations of GPUs provide the required double precision support, but with varying degrees of adherence to floating point standards. Second, MilkyWay@Home uses the Berkeley Open Infrastructure for Network Computing (BOINC), which provides an easily extensible framework for developing volunteer computing projects [1]. However, new support needed to be added to track not only what GPUs are available on volunteered hosts, but also what driver versions they are using so appropriately compiled applications can be sent to those clients.

This paper discusses the implementation and optimization of the Milky-Way@Home client application for both Nvidia and ATI GPUs. A 17 times speedup was achieved for double precision calculations on a Nvidia GeForce GTX 285 card, and a 109 times speedup for double precision calculations on an ATI HD5870 card, compared to the CPU version running on one core of a 3.0 GHz AMD Phenom(tm)II X4 940. Performing single precision calculations was also evaluated, and the methods presented improved accuracy from 5 to 8 significant digits for the final results. This compares to 16 significant digits with double precision, but on the same hardware, using single precision further increased performance 6.2 times for ATI, and 7.8 times for the Nvidia card. Utilizing these GPU applications on MilkyWay@Home has provided an immense amount of computing power, at the time of this publication approximately 216 teraflops.

The paper is organized as follows. Related volunteer computing projects and their progress in using GPGPU is discussed in Section 2. ATI and CUDA optimizations, along with strategies for improving their accuracy are presented in Section 3. Section 4 describes how BOINC has been extended to support GPU applications and how to use these exensions. The paper concludes with conclusions and future work in 5.

2   Related Work

Recently there has been an increase in the amount of volunteer computing projects porting their applications for use on GPUs. Some of these projects include Folding@Home [2], GPUGRID [3], Einstein@Home [4], and Aqua@Home [5]. The GPUs are chosen because of their availability and large increase in computational power over conventional CPUs [6]. However, the increase in computational power is not guaranteed. For example, Aqua@Home's GPU application implemented on Nvidia hardware does not perform as well as the CPU version [7]. Also, in order to reach peak performance levels it is necessary to design the algorithms to access data in specialized ways and limit the usage of branch type instructions [8].

Folding@home pioneered the use of GPUs in a distributed computing project to accelerate the speed of their calculations. The application was first programmed for ATI hardware using the Microsoft DirectX 9.0c API and the Pixel Shader 3.0 specification [6]. The Folding@Home GPU client was over 25 times faster than an optimized CPU version, with outdated hardware in today's standards. To achieve these performance levels they made use of the large amount of parallel execution units present on the GPU through the use of several optimizations [6]. To get a better idea of the impact the GPUs had on performance they stated that in 2006 there were 150,000 Windows computers performing 145 TFlops of work, while there were 550 GPUs producing 34 TFlops of work. Based on their numbers that leads to one GPU producing around 56 times more TFlops per 1 TFlop generated on a CPU.

Since then ATI and Nvidia started to offer tools for programming GPUs without being restricted by a graphics API. Both now use a subset of the C language augmented with extensions designed to ease the handling of data parallel tasks. Nvidia calls this programming environment CUDA (Compute Unified Device Architecture) [9], while ATI's version is named Brook+ [10]. This had a significant impact on both the adoption rate as well as the performance of Folding@home on GPUs. In 2009 they reported only a moderate increase of the performance delivered by 280,000 Windows computers to approximately 275 TFlops. But now 27,000 GPUs from ATI as well as Nvidia provide more than 3 PFlops of computing power [11], almost a hundred-fold increase compared to only 2.5 years before.

Like Folding@Home, GPUGRID recently started to utilize GPUs for performing molecular dynamics calculations [8, 12]. Both projects experienced significant speed ups by performing the calculation on the GPU, in particular Folding@Home stated between 100x and 700x performance increase comparing a Nvidia GTX 280 to a CPU with single-precision calculations.

3   Optimizing Milkyway@Home for GPUs

The MilkyWay@Home application attempts to discover various structures existing in the Milky Way galaxy and their spatial distribution [13]. This requires a

probabilistic algorithm for locating geometric objects in spatial databases [14]. The data being analyzed is from the Sloan Digital Sky Survey [15], which has recently released 15.7 terabytes of images (fits), 26.8 terabytes of data products in fit formats, 18 terabytes of catalogs in the SQL database and 3.3 terabytes of spectra (fits). The observed density of stars is drawn from a mixture distribution parameterized by the fraction of local sub-structure stars in the data compared to a smooth global background population, the parameters that specify the position and distribution of the sub-structure, and the parameters of the smooth background. Specifically this is a probability density function (PDF) that calculates the chance of obtaining the observed star distribution after repeated independent sampling from the total set of stars:

$$\mathcal{L}(\boldsymbol{Q}) = \prod_{i=1}^{N} PDF(l_i, b_i, g_i | \boldsymbol{Q}) \tag{1}$$

where i denotes the ith of N stars (l and b are galactic coordinates and g is the magnitude), and $\boldsymbol{Q}$ represents the parameters in the model. Such probabilistic framework gives a natural sampling algorithm for separating sub-structure from background [13].

By identifying and quantifying the stellar substructure and the smooth portion of the Milky Way's spheroid, it will be possible to test models for the formation of our galaxy, and by example the process of galaxy formation in general. In particular, we would like to know how many merger events contributed to the build up of the spheroid, what the sizes of the merged galaxies were, and at what time in the history of the Milky Way the merger events occurred. Models for tidal disruption of merger events that build up the spheroid of the Milky Way can be matched with individual, quantified spatial substructures to constrain the Galaxy's gravitational potential. Since the gravitational potential is dominated by dark matter, this technique will also teach us about the spatial distribution of dark matter in the Milky Way.

This section continues by first describing how precision for single-precision GPU calculation could be improved in Section 3.1 and then with specific optimizations used to greatly improve the performance of the GPU applications in Section 3.2.

3.1 Conserving Precision on Different Platforms

A general concern using heterogeneous systems for computation is to ensure the consistency of the results. This is even more important when completely different architectures like GPUs from different vendors as well as CPUs are utilized. While most floating point operations on current GPUs closely follow the precision requirements of the the IEEE 754 standard [16] there are some notable exceptions. Furthermore because of the parallel nature of the execution on GPUs it is necessary to pay attention to the order in which the individual work items are calculated. Generally, no specific order is guaranteed without serializing. In our case we can easily work around this issue as the results of

the individual work items of the time consuming parts of the calculation are basically just added up. The commonly employed solution is the well known Kahan method [17], which reduces the rounding errors and effectively guards the result against different summation orders.

We have done extensive testing to determine the reliability and reproducibility of the results between different architectures using single-precision as well as double-precision. Using single-precision on GPUs appears especially tempting because of the wider range of supported products and the vastly higher performance. While the peak performance difference between single-precision and double-precision on most CPUs is virtually nonexistent for legacy code and reaches only a factor of two when using *Single Instruction Multiple Data* (SIMD) extensions, it ranges from a factor of five (ATI) to a factor of eight (Nvidia) for GPUs. Furthermore GPUs support only basic double-precision operations in hardware, divisions or square roots for instance are computed iteratively with sequences of instructions. It was not until the latest GPU generations of both manufacturers that support for the instruction to perform double-precision fused multiply-add was included. With this instruction it is possible to obtain correctly rounded results for the already mentioned division and square root operations. Generally one has to evaluate the stability of the algorithm against small deviations in the handling of certain operations on different platforms. In our case the non-adherence to the IEEE standard for division and square root operations does not affect the final results. On the contrary, it enabled the use of low level optimizations which are described in the following section.

Table 1 shows the obtained precision and performance of the different implementations of our algorithm. The CPU and GPU implementations using double-precision operations all deliver the exact same results for typical parameter ranges. The single-precision variants give a sizable speedup while sacrificing some accuracy. But this loss can be minimized by taking care of the rounding errors occurring when combining the results of the individual work items. The resulting precision is in the expected range and actually slightly better than the resolution of the single-precision format ($2^{-24} \approx 6 \cdot 10^{-8}$). The reason is that the Kahan summation method effectively increases the precision of the stored intermediate values and the deviations between single and double-precision partly average out. Furthermore this allows to use CPU as well as GPUs for the same work units as all platforms calculate to the same precision.

3.2 Performance Optimizations

The time consuming part of our algorithm is the evaluation of the PDF for a three-dimensional wedge of the sky. The convolution with the probability distribution of the distance of a star for a given observed magnitude makes this effectively a four dimensional problem, which can be conveniently mapped to parallel architectures like GPUs. To use the hardware as efficiently as possibly we employed a range of optimizations. Changes on the algorithmic level were applied to all platforms, i.e. CPU as well as GPU applications use the same

Platform	precision	measured speedup	theoretical peak
3.0 GHz CPU core, DP, SSE3	0	1	1 (12 GFlop/s)
Nvidia GTX285, DP opt.	$< 1 \cdot 10^{-16}$	17	7.4 (89 GFlop/s)
ATI HD4870, DP	$< 1 \cdot 10^{-16}$	34	20 (240 GFlop/s)
ATI HD4870, DP opt.	$< 1 \cdot 10^{-16}$	48	20 (240 GFlop/s)
ATI HD5870, DP opt.	$< 1 \cdot 10^{-16}$	109	45 (544 GFlop/s)
NV GTX285, SP, naïve sum	$\sim 2 \cdot 10^{-6}$	130	89 (1063 GFlop/s)
Nvidia GTX285, SP	$\sim 1 \cdot 10^{-8}$	130	89 (1063 GFlop/s)
ATI HD4870, SP	$\sim 1 \cdot 10^{-8}$	299	100 (1200 GFlop/s)

Table 1. Comparison of the performance and precision of different platforms, SP and DP indicate single-precision and double-precision calculations respectively. The low level optimizations used in some DP versions are described in the text. The speedup is given relative to the double-precision CPU implementation running on a single core of a 3.0 GHz AMD PhenomII X4 940 using already vectorized code. The entry in the precision column quantifies the typical relative deviation to the results of the CPU implementation.

algorithm to calculate the results. Here we concentrate on the specific optimizations done for GPUs. As the architectures of ATI as well as Nvidia GPUs share common design principles, most of the optimizations apply to both platforms. Because of the distinct features of the programming environments of both vendors often the same optimization has to be implemented in a different way. Figure 1 depicts the various optimizations that were done along with their effect on the performance for the case of the CUDA implementation.

The first set of optimizations aims at the efficient use of the memory controller and the caches. As recurring expensive expressions are precalculated and stored in lookup tables, the access to these data structures can be accelerated by the use of the texture caches in GPUs increasing the available bandwidth as well as reducing the latency in comparison to accesses to the global memory. This is particularly easy for ATI GPUs as all data structures are allocated as textures by default in Brook+. The hardware tiles and interleaves the data in a transparent way to make use of the spatial locality by coalescing memory accesses and improving the cache hit rate.

For Nvidia GPUs the coalescence of memory accesses requires some manual intervention. The data inside the different data structures has to be reordered to be accessed consecutively by the global thread id. Initially the implementation made use of the GPU's constant memory instead of textures in order to store the precalculated values associated with the algorithm. While constant memory is really stored in the GPU's global memory space it is cached by a dedicated constant cache after the first access [9]. Nevertheless the use of texture memory offers some advantages exploiting the spatial locality in texture references [9]. The first reason is that the textures accesses are cached by a two level cache system which simply offers more space and therefore higher hit rates. The other

reason is the organization in cache lines, i.e. when one value is read from the memory, the successive values which are most likely needed by a neighboring thread or the next iteration in the same thread are also fetched into the cache. This lowers the average latency for the accesses.

For our algorithm the ratio of calculations to the number of memory accesses is high. Depending on the problem one has between 5 and 12 floating point operations per byte fetched from the lookup tables. The calculations can be arranged to take advantage of spatial locality within the caches even when the full lookup tables are larger than the first level caches on CPUs or the texture caches on GPUs. Therefore memory fetches are certainly not a serious bottleneck by themselves. Nevertheless one has a limited number of threads available to hide the access latencies. Therefore it is necessary to look into both, improving the efficiency of memory accesses and minimizing the latency as well as maximizing the number of threads present on the GPU to attain a high occupancy of the execution units. The latter usually requires the reduction of the used hardware registers, as all threads on a SIMD engine in a GPU share the same register file. In the case of the CUDA version, the amount of shared memory usage had to be reduced. This reduction is possible by changing the number of active threads per block, this had a negligible effect on performance. The baseline CUDA version minimized the register usage, but as our algorithm is clearly compute bound no excessive amount of threads is necessary to hide the latencies. Furthermore newer GPUs raised the amount of available registers and lessened the importance of this optimization strategy. It was therefore beneficial to employ the same strategy as for CPUs and ATI GPUs and combine calculations wherever possible in order to minimize the necessary operations. This results in using more registers but allowed for a slight performance increase and is labeled as *Combine Terms* in Figure 1.

As the division and square root are particularly costly on GPUs in double-precision, we looked for possible shortcuts compared to the compiler generated instruction sequences for these operations. Generally the Newton iteration method is used for both. After the exponent extraction which reduces the argument range the single-precision reciprocal or square root instruction is used to get the first estimate. In our case we have some prior knowledge about the range of values we operate with. As we need only the longer mantissa but not the extended exponent range of the double-precision format one can simply spare the exponent extraction. To get correctly rounded results, one needs one final iteration step using a fused multiply add operation [18]. Without this step the last bit of the mantissa may deviate. But as this step doesn't make a difference on GPUs not supporting a fused multiply add instruction without intermediate rounding and it is not the limiting factor in our calculations, we omitted it as well. Together this reduced the amount of instructions in the innermost loops by up to 20 percent. While one cannot guarantee that the results after these changes exactly match the original versions in all cases, we found no deviations in our tests (see Table 1). A plot of how the changes affected the performance of the Nvidia implementation can be seen in Figure 1, labeled as *Sqrt* and *Div*.
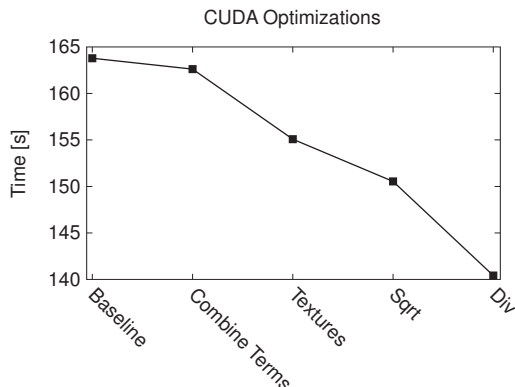
Fig. 1. Optimizations for the double-precision CUDA version and the effect on the calculation time for one evaluation on a Nvidia GTX285 GPU. See the text for a description of the changes.

We also explored the feasibility of some further low level optimizations for ATI. As the execution units of current ATI GPUs are organized in groups which are fed with bundles of independent instructions (resembling VLIW architectures), it is sometimes difficult for the compiler to maximize the utilization of the units. The Brook+ compiler generates code in a pseudo assembly *intermediate language* (IL), which is just-in-time compiled and optimized for the specific GPU in the system at runtime [10]. The IL code is easily accessible. The innermost loop of the time consuming kernels was tuned for an optimal instruction pairing to increase the utilization resulting in an additional 20 percent speedup compared to the Brook+ generated IL.

As a result we have implemented the algorithm with a comparable level of optimization on different platforms. The general strategies are very similar. Only the VLIW-like organization of the execution units of ATI GPUs requires some additional attention to obtain optimal performance. This is even more important for single-precision applications. In our case it was straightforward to combine four threads into a single one by using the appropriate vector data types. On the other hand the memory layout and handling required more effort for Nvidia GPUs.

Generally, the performance of the different implementations roughly follow the theoretical peak performances of the corresponding platforms. As our algorithm is significantly compute-bound it scales very well on massively parallel architectures. In fact, even when neglecting the low level optimized version the resource utilization on GPUs is better than on CPUs. This can be explained by the latency hiding features of GPUs which are designed to mask the execution latencies of the individual instructions as well as memory latencies. This may be even more significant for single-precision when the GPUs can use their fast hardware instructions for division and square root operations which have a

high latency on CPUs. In contrast GPUs are able to continuously operate close to their maximum instruction throughput on our algorithm. Comparing ATI and Nvidia GPUs, both arrive at about the same performance relative to their theoretical peak throughput.

4 Using GPUs on BOINC

BOINC is a platform for volunteer computing. It is used by about 50 projects, including Milkyway@home, SETI@home, IBM World Community Grid, Einstein@home, Rosetta@home, and Climateprediction.net. BOINC allows volunteers to participate in multiple projects. To do so, they download and run a client program (available for all common operating systems) and attach the client to the desired projects. Each attachment has a volunteer-specified resource share indicating the fraction of the computer's available resources that should be allocated to the project.
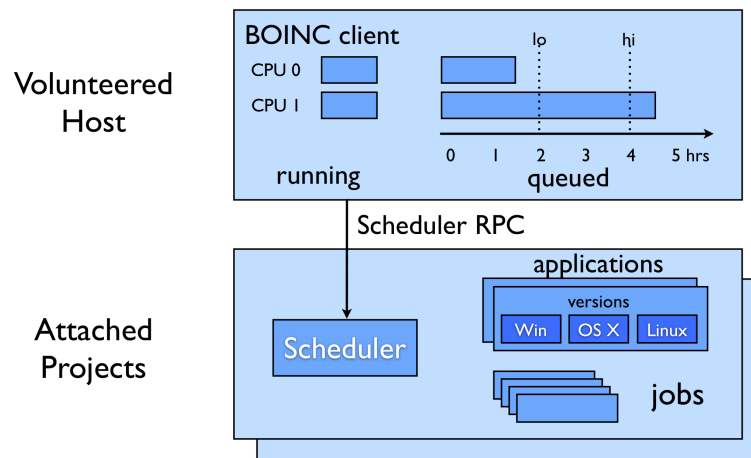

Fig. 2. The basic operation of BOINC.

The basic operation of BOINC is shown in Figure 2, in which a volunteer host is attached to projects A and B. The BOINC client maintains a queue of jobs and manages their execution; depending on user preferences, it may run jobs when the computer is in use, not in use, or both. When the length of the queue (in terms of estimated runtime) falls below a lower limit, the client selects a project and issues a scheduler RPC to it, requesting enough jobs to reach the upper limit. The lower and upper limits on queue length are chosen to avoid running out of work during periods of disconnection, and to minimize the frequency of scheduler RPCs. The choice of project is based on resource

share: the client maintains a per-project work debt that increases in proportion to resource share, and decreases as work is done. Jobs are requested from the project whose debt is greatest.

Each project has its own server and database. The database stores a set of jobs, each of which is associated with an application. An application can have several versions, one per platform (Window 32-bit, Windows 64-bit, Mac OS X, Linux, etc.). The attributes of a job include its memory and disk requirements, an estimate of its FLOP count, and its deadline. The scheduler RPC's request message describes the host's operating system and version, processor type and benchmark scores, memory size, and available disk space. It requests an amount of work, as measured in expected runtime. Based on this information, the scheduler attempts to find a set of jobs that satisfy the request and that can be executed on the host (i.e., the memory and disk requirements are met, the deadline will probably be met, and a version is available for the host's platform).

This basic framework was extended to handle applications that use GPUs as well as CPUs. The resulting system handles clients that have arbitrarily many GPUs, possibly of different types (currently Nvidia and ATI are supported). GPUs of a given type are assumed to be identical. The volunteer host population may have a wide range of GPU models, driver versions, and memory sizes.

Referring to Figure 2, we now allow an application to have multiple versions per platform. For a given platform (say, Win32) an application might have versions for one CPU, for Nvidia GPU, for ATI GPU, and for multiple CPUs. The resource requirements of a particular version may not be integral: for example, a job might need 0.5 GPUs and 0.1 CPUs, and these fractions may depend on the host.

The notion of resource share is defined as applying to a host's aggregate processing resources, not to the resource types separately. For example, suppose a host has a CPU and a GPU, and the GPU is twice as fast. Suppose that the host is attached to projects A and B with equal resource shares, and that project A has both GPU and CPU applications, while project B has only CPU applications. In this case project A should be allocated 100% of the GPU and 25% of the CPU, while project B should be allocated 75% of the CPU (see Figure 3).

As shown by this example, the client must be able to ask the server for jobs of a particular resource type (CPU or GPU). Thus, we extended the scheduler RPC protocol to include a separate work request (in seconds) for each resource type. In the scheduler RPC reply, jobs are associated with particular versions, hence with particular resource types and usage levels.

Two major parts of the BOINC client have been extended to support GPU applications. First, the job scheduler is now GPU-aware; the client allocates GPUs, and passes command-line parameters to GPU jobs telling them which GPU instance(s) to use. Because GPU memory is physical rather than virtual, preempted GPU jobs must always be removed from memory.

The client maintains separate job queues for each processing resource, and maintains a separate per-project debt for each resource as well. It maintains a
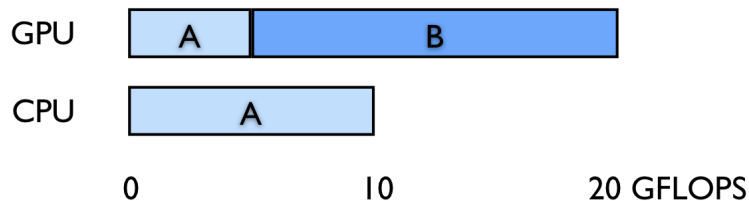
Fig. 3. A project's resource share applies to all processing resources. In this example, projects A and B each get 15 GFLOPS.

dynamic estimate of which projects have jobs for which resources. A project's overall debt is the sum of its debts over all resources, weighted by the average speeds of the resources. The client's work-fetch policy can be summarized as follows: when the queue for some resource falls below its lower limit, the client chooses, from among the projects likely to have jobs for that resource, the one whose overall debt is greatest. It then issues a scheduler RPC to that project, requesting work for the given resource and possibly for others as well. This policy enforces resource shares as described above.

Moving now to the scheduler, deciding whether a GPU job can be handled by a particular client may involve many details of the GPU hardware and driver. This decision is made by an application planning function that is supplied by the project for each of its versions. This function takes as input a description of the host and its GPUs. It returns a) a flag indicating whether the jobs can run the version, and if so b) an estimate of the resource usage (number of CPUs, number of GPUs) and c) estimate of the FLOPs/second the version will achieve on the host.

In sending a job to a client, the scheduler must now decide which of possibly several versions to use. This is done as follows. The scheduler considers all the application's versions for the client's platforms, and calls the respective application planning functions. It skips those that use resources for which no work is being requested, and from the other it selects the one whose FLOPs/second estimate is greatest. This estimate is then used to estimate the job's runtime on the host, and the work request for the resource is decremented by that amount.

5 Summary

This work discusses the implementation and optimization of the Milky-Way@Home client application for both Nvidia and ATI GPUs. A 17 times speedup was achieved for double-precision calculations on a Nvidia GeForce GTX 285 card, and a 109 times speedup for double-precision calculations on an ATI HD5870 card, compared to a vectorized CPU version running on one core of a 3.0 GHz AMD Phenom(tm)II X4 940. Performing single-precision calculations was also evaluated, and the methods presented improved accuracy from 5

to 8 significant digits for the final results. This compares to 16 significant digits with double-precision, but on the same hardware, using single-precision further increased performance 6.2 times for ATI, and 7.8 times faster on the Nvidia card. Utilizing these GPU applications on MilkyWay@Home has provided an immense amount of computing power, at the time of this publication approximately 216 teraflops.

While developing GPGPU applications still requires significant technical knowledge, the process is becoming easier. Additionally, the large amount of computing resources this type of hardware provides makes utilizing GPUs a highly desirable prospect, especially in the area of volunteer computing. As the hardware and software matures, we expect GPGPU applications to become more mainstream. The techniques discussed in this paper can aid in the development of other GPGPU applications and describe how they can be effectively used in a volunteer computing environment.

6 Acknowledgements

References

1. D.P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203. IEEE Computer Society, 2005.
2. V.S. Pande. http://folding.stanford.edu.
3. G. De Fabritiis. http://gpugrid.net.
4. B. Allen. http://einstein.phys.uwm.edu.
5. D-Wave Systems Inc. http://aqua.dwavesys.com.
6. E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V.S. Pande. N-Body simulation on GPUs. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 188, New York, NY, USA, 2006. ACM.
7. D-Wave Systems Inc. http://aqua.dwavesys.com/faq.html.
8. M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, and V.S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30:864–872, 2009.
9. NVIDIA Corporation. NVIDIA CUDA Programming Guide Version 2.3.1.
10. AMD Corporation. ATI Stream Computing User Guide Version 1.4.0.

11. A.L. Beberg, D.L. Ensign, G. Jayachandran, S. Khaliq, and V.S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2009.

12. M.J. Harvey, G. Giupponi, and G. De Fabritiis. ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *Journal of Chemical Theory and Computation*, 5, 2009.

13. Jonathan Purnell, Malik Magdon-Ismail, and Heidi Newberg. A probabilistic approach to finding geometric objects in spatial datasets of the Milky Way. In *Proceedings of the 15th International Symposium on Methodoligies for Intelligent Systems (ISMIS 2005)*, pages 475–484, Saratoga Springs, NY, USA, May 2005. Springer.

14. C. Reina, P. Bradley, and U. Fayyad. Clustering very large databases using mixture models. In *Proc. 15th International Conference on Pattern Recognition*, 2000.

15. J. Adelman-McCarthy *et al.* The 6th Sloan Digital Sky Survey Data Release, http://www.sdss.org/dr6/, July 2007. ApJS, in press, arXiv/0707.3413.

16. IEEE Standard for Binary Floating-Point Arithmetic, 1985. ANSI / IEEE Std. 754-1985.

17. W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

18. M.A. Cornea-Hasegan, R.A. Golliver, and P. Markstein. Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms. In *Proceedings of the 14th IEEE Symposium on Computer Arithemtic*, pages 96–105, 1999.