GPU as a General Purpose Computing Resource

Qihang Huang†, Zhiyi Huang†, Paul Werstein†, and Martin Purvis‡
†Department of Computer Science
‡Department of Information Science
University of Otago, Dunedin, New Zealand
Email:{tim, hzy, werstein}@cs.otago.ac.nz, mpurvis@infoscience.otago.ac.nz

Abstract

In the last few years, GPUs (Graphics Processing Units) have made rapid development. Their ever-increasing computing power and decreasing cost have attracted attention from both industry and academia. In addition to graphics applications, researchers are interested in using them for general purpose computing. Recently, NVIDIA released a new computing architecture, CUDA (Compute Unified Device Architecture), for its GeForce 8 series, Quadro FX, and Tesla GPU products. This new architecture can change fundamentally the way in which GPUs are used. In this paper, we study the programmability of CUDA and its GeForce 8 GPU and compare its performance with general purpose processors, in order to investigate its suitability for general purpose computation.

Keywords: GPU, GPGPU, CUDA programming, Massively Parallel Computing Resource, Sun UltraSPARC T1, Intel Pentium 4

1 Introduction

Due to physical limitations, the clock speed of CPUs has come to a maximum limit. However, the Moore's Law still holds, which means there still exists the ability to pack more transistors on a chip. The recent trend in the microprocessor industry is to put more cores (processors) into a single chip. A GPU is a good example of specialized massively parallel processors with over a hundred of cores in the latest products. Yet a GPU is known to be notoriously hard to program. In June 2007, NVIDIA released the CUDA programming guide, v1.0, for its G80 GPUs [1]. The CUDA programming model offers programmers a straightforward C-like interface instead of mapping computation using the traditional graphic API, which makes it significantly easier to utilize a GPU.

Naturally, one could ask questions like "Is a GPU now ready for general purpose computing?", "How is its performance compared to the traditional CPU and the new multi-core processor?" These questions are of importance, as we could, based on their answers, determine whether it is appropriate to combine GPU computing with cluster computing or even Grid computing, e.g., to form GPU cluster or GPU Grid. However, to the best of our knowledge, there is little work published regarding performance comparisons among the latest GPUs, traditional CPUs, and the latest multi-core processors.

This paper attempts to address partially the above questions from the aspects of performance and programmability. The rest of the paper is organized as follows: Section 2 briefly presents the traditional GPU programming and the newly released CUDA programming model, and discusses its programmability issues. Section 3 evaluates the performance of a GPU in comparison with general-purpose processors including UltraSPARC T1 [2] and Intel Pentium 4. Conclusions and future work are suggested in Section 4.

2 CUDA programming

CUDA has changed significantly GPU programming compared to traditional GPU programming. In order to better understand the advantages of CUDA programming, we first briefly introduce traditional GPU programming.

2.1 Earlier GPU programming

Due to the huge demand in the game market, 3D computer graphics cards are becoming cheaper while providing greater processing power. This trend has drawn attention regarding how to apply this computing power to non-graphical applications. Significant work has been done in the development of GPGPU (General-Purpose computing on Graphics Processing Unit) [3].

Yet there are some issues that hinder its development:

- The hardware architecture exposed to programmers is complex.

  In the earlier GPUs, only part of the graphics pipeline was programmable. That is, only the vertex processor

and the fragment processor were programmable (The program for this part is called shader program). In order to trigger the fragment processor, all steps before the fragment processor in the rendering pipeline starting from the vertex processor have to be executed [4], which means programmers need to have a good understanding of the whole rendering pipeline.

- The GPU is programmed via a graphics API, which is not convenient for general purpose computing.

  Although there exists a number of C-like high-level shading languages to program a GPU, they all need to wrap the actual computation with the graphics API. Thus they have a strong computer graphics terminology. This imposes a steep learning curve for novice or even experienced programmers.

- The GPU memory does not support write access in a general way.

  The GPU cannot write its DRAM in the same way a CPU does, though it can read its DRAM in the same way a CPU. This shortcoming removes significant programming flexibility which is readily available on CPUs [1].

2.2 CUDA programming

The GeForce 8 series GPUs and the CUDA programming model are NVIDIA's answer to the issues mentioned in Section 2.1. The new hardware architecture exposed to programmers is shown in Figure 1. Let us take the GTX 8800 as an example. Logically it has 16 multiprocessors (MP). Each MP contains 8 streaming processors (SP), which are a set of SIMD (Single Instruction Multiple Data) processors with 16KB on-chip shared memory (mainly used as a software managed cache) and 8192 registers. All the MPs have access to the DRAM (768MB) of the GPU.

The parallelism is achieved by issuing a number of threads to run on the GPU concurrently. These concurrent threads are organized as a two-dimension grid of thread blocks, with each thread block as a two- or three-dimension thread array. Thread blocks are executed on the GPU by executing one or more blocks on each multiprocessor using time slicing. The actual scheduling of the threads works as follows: it splits a block into warps, each of which contains the same number of threads grouped by consecutive thread ID; and then each warp is executed on a MP in a SIMD fashion. Time slicing is used to switch among warps. All threads execute the same program (known as a kernel in CUDA terminology). Task partition is done through a block index (blockIdx) and a thread index (threadIdx).

To utilize the GPU, CUDA provides an extended C-like programming language. A function in the source code could

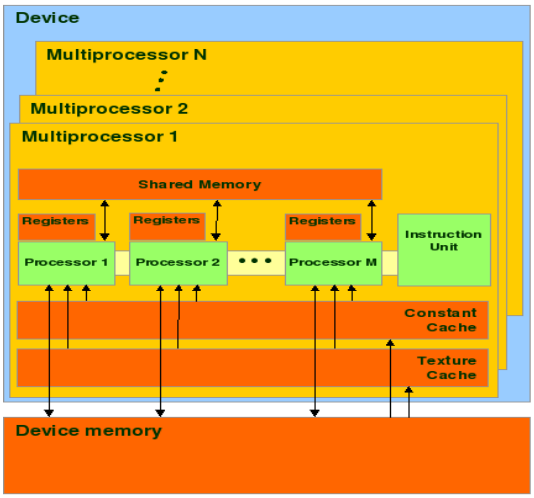

Figure 1: Hardware Model (from [1])

be declared as __host__, __global__, and __device__, denoting respectively that the function will be run in the CPU (the host), run in the GPU but called from a host function, or run in the GPU but called from a global/device function. Any call to a __global__ function is said to issue a kernel to run on the GPU and must first specify the dimension of the grid and the blocks that will be used to execute the function on GPU. The execution model is shown in Figure 2.

A sample program is as follows:

```
__device__ float compute(int idx) {
  //do some computation
  ......
}

__global__ void kernelFunc(float *var){
  // this is a kernel function called
  // from host and executed on GPU
  int i;
  i=blockIdx.x*blockDim.x+threadIdx.x;
  var[i]=compute(i);
}

int main(void) {
  float *var;//array allocated on CPU
  float *varGPU;//array allocated
                //on GPU
  //init var
  ......
  //allocate varGPU, init it using var
  cudaMalloc((void**)&varGPU, \
             sizeof(float)*size);
  cudaMemcpy(varGPU, var, \
             cudaMemcpyHostToDevice);
  //execution specification
```

```
    kernelFunc<<<96,192>>>(varGPU);
    //free varGPU
    cudaFree(varGPU);
    return 0;
}
```

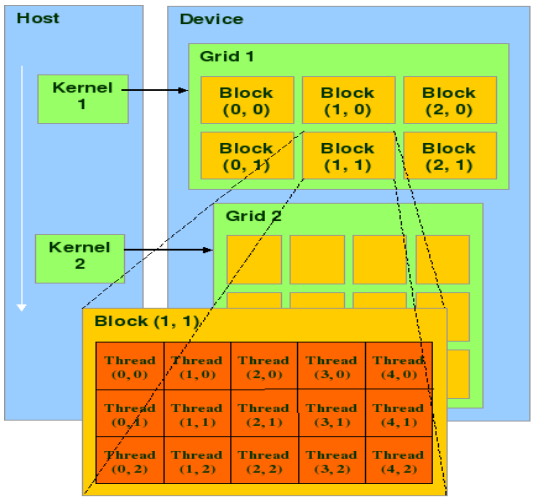

Figure 2: Execution Model (from [1])

In this example, *kernelFunc* will be executed on the GPU as a grid of 96 thread blocks, with 192 threads in each thread block. If it is run on an 8800 GTX card (with 16 MPs), 6 thread blocks will be run on each MP concurrently. If it is run on a 8800 GTS card (with 12 MPs), 8 thread blocks will be run on each MP concurrently. However since all thread blocks running on the same MP share its registers and shared memory, the actual number running concurrently on a MP might be less than the aforementioned number according to the actual usage of the shared resources in the kernel.

A thread that executes on the GPU has access to the following memory spaces:

- *Register* which can be read and written by its thread.
- *Local memory* which can be read and written by its thread.
- *Shared memory* which can be read and written by the threads in the same block.
- *Global memory* which can be read and written by the threads in the same grid.
- *Constant memory* which is read only by the threads in the same grid.
- *Texture memory* which is read only by the threads in the same grid.

Only the constant memory and the texture memory are cached. Both the local memory and the global memory reside on the DRAM of the GPU. The only difference is whether or not it is accessible to other threads. In terms of access latency, the register, shared memory, and cache of the constant memory and texture memory are of the same magnitude. However access to the rest of memory spaces such as the global memory takes hundreds of times longer. The host (CPU) only has access to the global memory, the constant memory, and the texture memory.

The memory model is shown in Figure 3.

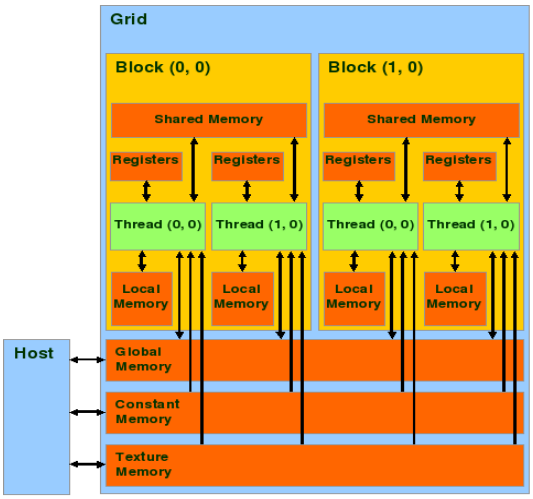

Figure 3: Memory Model (from [5])

2.3 Programmability Issues

We can see from Section 2.2 that the extended C programming interface of CUDA has removed the burden of programmers learning complex shader programming. Also it has a straghtforward way of multi-threaded programming. These features mean simpler code porting from CPU to GPU and between GPUs.

However, even with this great improvement of GPU programmability, CUDA still has some issues. Firstly, the global memory is not cached. This is because more transistors are devoted to data processing than control logic such as data caching. In order to compensate, CUDA encourages programmers to use the on-chip shared memory as manually controlled cache. Clearly, it is not a trivial job for programmers. Often programmers have to alter the algorithm structure to fit into the limited shared memory. One important consequence of this sort of tweak is poor scalability. If the size of the shared memory changes in upcoming GPUs, code porting becomes a problem. The authors believe that hiding shared memory from programmers will make CUDA programming much more convenient. Given the current de-

sign of the GPU hardware, we think a software solution should be investigated to hide the shared memory from the programmers. One way of doing so is to adopt the concept of View-Oriented Parallel Programming (VOPP) [6] into CUDA.

In VOPP, shared data are partitioned into *views* that are non-overlapping memory blocks. Views can be dynamically allocated or statically defined by programmers. Each view is identified by its view id. A view manager is adopted to manage view allocation and destruction. Before a view is accessed, a view primitive like *acquire_view* is called so that the view manager will know which data will be accessed and can place the data in an optimal location such as shared memory. In this way, programmers do not need to distinguish between shared memory and global memory. The compiler and the view manager can optimize the access of the views according to the data access pattern of the program by placing them either in shared memory or in global memory. The relation between global memory and shared memory in the GPU resembles the relation between RAM and cache in a CPU. Therefore, the view-based memory consistency protocol [7] used for VOPP can be readily used for memory consistency of views between global memory and shared memory in the GPU.

If the VOPP API could be implemented in CUDA as a library, a *view* becomes the universal memory concept to programmers. However, we need further investigation in order to determine if an implementation of VOPP will be efficient under the various restrictions of a GPU.

Secondly, CUDA provides an API for threads to communicate within the same thread block, but there is no mechanism for inter-block communication. To fully utilize the GPU computing power, we need to use all MPs. However, since each block can only run on a single MP, we need to generate multiple thread blocks on different MPs which do not provide any communication facility for the thread blocks. For applications where inter-block communication is essential, it is not possible to run them without extra effort. We propose two ways to achieve this inter-block communication. One way is to implement a mutex in the global memory to coordinate two blocks. However hard-coding mutexes is not flexible and becomes error-prone as the number of blocks increases. Another way is to split the kernel into multiple parts and use the host (CPU) as a central place to synchronize the different parts. This can incur significant data transfer between host and GPU. We have adopted the second approach in our N-Body problem and Game of Life problem to be discussed in Section 3.3.

Thirdly, programmers need to be aware of the GPU memory spaces. It is not only the separation between the shared memory and the global memory, but also between the GPU memory and the CPU memory, which causes extra coding for passing the data between different memory spaces.

There are a number of other small inconvenience and limitations. For example, because __device__ functions are inlined into __global__ functions, they cannot be recursive. This could be solved by a standard transformation of the automatic function stack to the manual recursive stack. Also, at the moment floating point calculations only support single precision. However this is expected to change in upcoming GPUs.

3 Performance evaluation

In this section, we evaluate the performance of a GPU against general purpose processors. We first present the test results of memory access for different processors. Then we present performance results of two non-graphical applications: matrix multiplication and N-Body problem running on a GPU, a UltraSPARC T1, and a Pentium 4. Then we summarize the results of our tests. We do not include graphics applications in our performance evaluation since we focus on general purpose computing. GPU performance for graphics applications has been explored in previous work such as [8].

The configurations of the testing platforms are as follows:

- The GPU is a NVIDIA GTX 8800, which has 16 multiprocessors (MPs). Each MP has 8 streaming processors (SPs). Each SP runs at 1.35GHz clock speed. The GPU is installed in an Intel 2.8GHz Pentium PC.

- The Intel Pentium 4 processor has a 2.8GHz clock speed and is configured with 2GB RAM and 2048 KB cache.

- We use a Sun UltraSPARC T1 to compare scalability with the GPU. The T1 processor has 8 cores. Each core runs at 1.0GHz clock speed and has four strands (hardware threads). In total it can run 32 threads simultaneously. The T1 chip is installed in a Sun T2000 server with 16GB RAM. Since there is only one floating point unit (FPU) in the T1, the FPU is a bottleneck for parallel floating point applications. In order to make reasonable comparisons, we make a special adjustment for some test cases and note it accordingly. For the Sun, the three applications are programmed with OpenMP, compiled by *cc* from Sun Studio 11, and run on Solaris 11.

In the following subsections, the terms GPU, Intel, and Sun are used to denote the above testing platforms.

3.1 Characteristics of memory accesses

In order to find out the characteristics of GPU memory such as memory access latency, we have designed three memory test programs. The first test determines the time of memory copy from one array to another array. We test it in four cases: memory copy in the Intel's RAM with cache disabled, memory copy in the GPU's global memory, memory copy in the Intel's RAM with cache enabled, and memory copy in the GPU's shared memory. We use a range of array sizes from 1KB to 7KB. The GPU shared memory has a size of 16KB, but it is also used for passing kernel function arguments, so we cannot use all shared memory for data. In order to hold two arrays on shared memory at the same time, we make the largest size to be 7KB. We use only one thread in this test. The test results are shown in Table 1.

Table 1: Memory Copy (time in μ second)

Platform	Size			
	1KB	2KB	4KB	7KB
Intel(CD)	2.11	3.54	6.67	15.73
GPU(GM)	94.88	189.65	379.53	735.38
Intel(CE)	0.63	1.26	2.49	4.85
GPU(SM)	8.42	16.72	33.34	64.48

In Table1, "CD" stands for cache disabled; "CE" stands for cache enabled; "GM" stands for global memory; "SM" stands for shared memory. To disable cache in the Intel CPU, the privileged instruction "WBINVD" (write back and invalidate cache) is used.

From Table 1, we can see that GPU memory is more than 10 times slower than RAM access in the Intel machine, for both cache (or shared memory) enabled and disabled. To hide the memory latency in the GPU, hundreds of threads are suggested to be used in one MP of GPU. The second test (described in the next paragraph) shows that increasing the number of threads in the GPU can increase the throughput of memory accesses.

The second test determines the throughput of random access of a 1KB array residing in global memory, constant memory, texture memory, and shared memory respectively. The GPU execution configuration varies from 1 block with 1 thread per block, to 512 blocks with 256 threads per block. The test results are shown in Table 2. To reduce extra overhead of the *for* loop, each loop contains 64 memory accesses. This idea is similar to [9], which can keep the loop overhead to around 2%.

Table 2 shows that when the number of threads increases, the throughput of memory accesses is increased in general. Also the shared memory access is at least 30 times faster than global memory.

The last test determines the speed of transfering a block of data in global memory to shared memory and the read speed of global memory and shared memory. This test is made to see one major overhead of implementing VOPP in GPU. This overhead results from requiring a view to be loaded from global memory to shared memory when needed. The results are shown in Table 3. Again, because the 16KB shared memory in GPU is also used for passing kernel function argument, only 15KB is used in the last column.

In Table 3, "G2S" means copying data from the global memory to the shared memory; "GR" means reading data from the global memory; "SR" means reading data from the shared memory. We can see from the table that it is profitable to use shared memory as cache of global memory. However, if the data is only read once, there is no performance gain by copying data from the global memory to the shared memory. Programmers have to keep this in mind in order to use the shared memory for improving performance. If VOPP is implemented, the compiler has to determine this before deciding whether or not to copy a view in the shared memory.

3.2 Matrix multiplication

Matrix multiplication computes the matrix product of A and B and stores the result into C. Since the computation of each element in C is independent, matrix multiplication is a problem with embarrassingly parallelism and thus particularly suitable for a GPU.

We use three problems sizes for our tests. The first problem size (called PS1) is 3072×800 for matrix A and 800×800 for matrix B. The second problem size (called PS2) is 3072×1600 for matrix A and 1600×1600 for matrix B. The third problem size (called PS3) is 3072×3200 for matrix A and 3200×3200 for matrix B. PS1 is used for testing the scalability of the GPU and UltraSPARC T1. PS2 is used for testing the performance of the GPU with differ-

Table 2: Memory access throughput (in GB/s)

Block $\times$ Thread	Memory Type			
	global	constant	texture	shared
1×1	0.32	2.92	0.53	2.73
32×64	3.40	37.60	31.16	73.67
128×128	3.54	38.59	34.71	74.96
256×128	3.73	51.48	38.99	97.91
256×256	3.73	47.49	38.92	109.46
512×256	3.32	51.65	38.24	115.37

Table 3: Transfer Speed and Read Speed (time in μ second)

Type	Size				
	1KB	2KB	4KB	8KB	15KB
G2S	22	45	87	174	327
GR	18	32	59	113	214
SR	1.1	1.6	2.8	5.1	9.4

ent execution configurations. All problem sizes are used to test computing power of the GPU in comparison with the Intel CPU and Sun T1. To make the comparision fair since the T1 only has one FPU, we use long integers in the scalability and computing power tests, and use floating point numbers for the GPU test with different execution configurations.

The scalability result is shown in Figure 4.

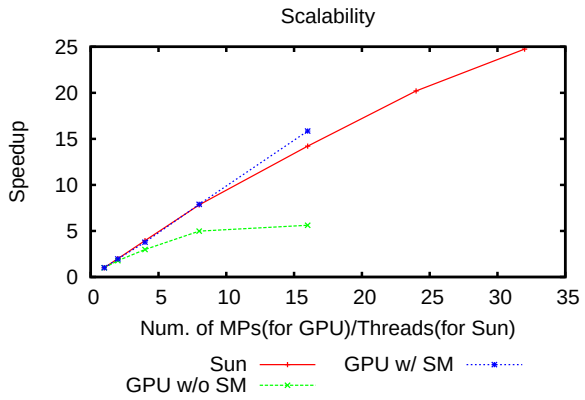

Figure 4: Matrix Multiplication(Problem Size 1)

Figure 4 shows the speedup of matrix multiplication running on the GPU and Sun UltraSPARC T1. It also shows the speedup of the application without using the shared memory (similar to software managed cache) in the GPU in order to show the importance of using the shared memory in GPU applications. We use 256 threads per block in the application for testing the scalability of the GPU.

The figure shows the scalability of GPU is extremely good for matrix multiplication when shared memory is used. Its speedup is almost linearly increasing with the number of MPs. However, when not using the shared memory, the speedup curve starts to become flat when the number of MPs is 8. The scalability of the UltraSPARC T1 is also very good, and the speedup keeps growing all the way up to 32 threads for the OpenMP implementation of matrix multiplication. From this test, we know the GPU has excellent scalability for embarrassingly parallel problems. Its scalability is even better than the UltraSPARC T1, which is a start-of-the-art general purpose multicore processor.

Table 4: Different Execution Configuration of GPU

	64 t/b	192 t/b	256 t/b	256 t/b SM
Time (second)	9.94	6.781	2.593	0.325
GFLOPS	1.57	2.3	6.1	48.3

We tested the performance of GPU with different execution configurations. The result is shown in Table 4. “GFLOPS” means giga floating point operation per second; “t/b” means threads per block (“32 t/b” means 32 threads per block); “SM” means using the shared memory.

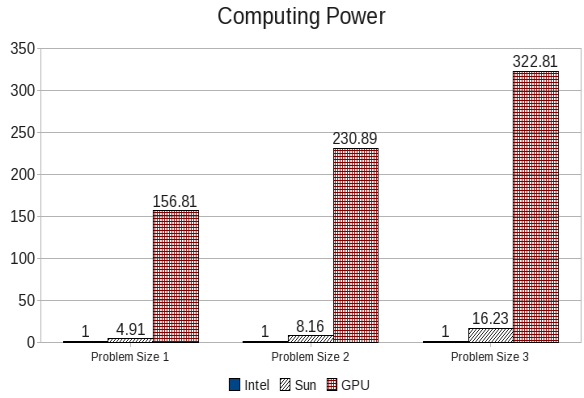

Figure 5: Matrix Multiplication

From the result, we observe that by increasing the number of threads in one block, the GFLOPS of the GPU increases as well. This is because more threads in a block can hide memory access latency. More interesting, there is a big difference (8 times better) when shared memory is used.

In Figure 5, we compare the performance of the GPU against Intel and Sun. In this test, the Sun uses 32 threads and the GPU uses 256 threads and shared memory is used. The numbers on top of the bars represent the computing power. It is based on the Intel CPU, whose computing power is considered to be 1. The computing power of other processors is calculated by dividing the running time on the Intel by the running time on that processor.

Compared with the Pentium 4, the GPU can be 322 times faster for PS3. One interesting result is that when the problem size grows, the GPU and Sun computing power grows. That means the Intel CPU slows down more than the GPU and Sun as the problem size grows. This is because increasing problem size will increase the granularity of the parallel algorithm. This is more advantageous to the multi-core processors.

3.3 N-Body problem

The N-Body problem is concerned with determining the effects of forces between astronomical bodies in space. Its task is to find the positions and movements of bodies in space that are subject to gravitational forces from other bodies using Newtonian laws of physics.

We choose this application because we want to measure how the GPU performs when there is communication between two iterations. After each time interval before continuing with the computation, each thread needs to update the location and velocity of each body. This needs synchronization among all threads. As pointed out in Section 2.3, CUDA does not directly support inter-block communica-

tion, so we need to synchronize the threads via the host, which resembles a master/slave pattern.

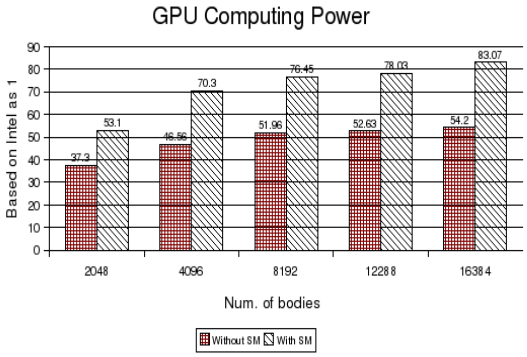

Figure 6: N-Body (floating point number)

Figure 6 shows the computing power of the GPU based on the Intel CPU. We can see that when the problem size (number of bodies) increases, the GPU computing power increases in comparison with the Intel CPU, which is consistent with the matrix multiplication.

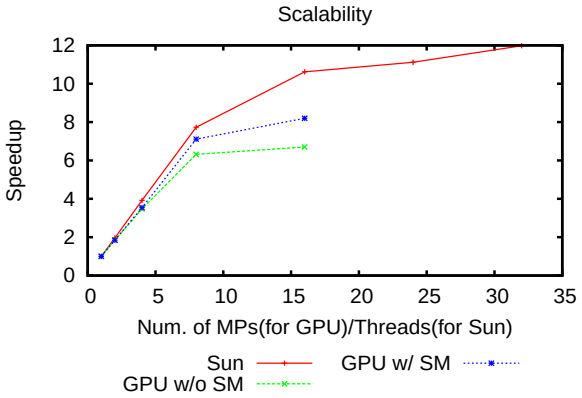

Figure 7: Scalability of N-Body problem

For the next test, we alter the N-body problem to use long integers instead of floating point numbers. For square root operation used to calculate distance between bodies, an integer approximation function is used. Figure 7 shows the scalability of the application on the GPU and the UltraSPARC T1. The number of bodies in the application is 4096. Each block in the GPU has 256 threads. From the figure, we can see that the Sun UltraSPARC T1 has a better scalability than the GPU. This is because the global memory is more often used for information sharing in this application. Figure 8 shows the computing power of Intel, Sun, and GPU. Comparing Figure 6 and Figure 8, we can see that the GPU supports floating point operation better than

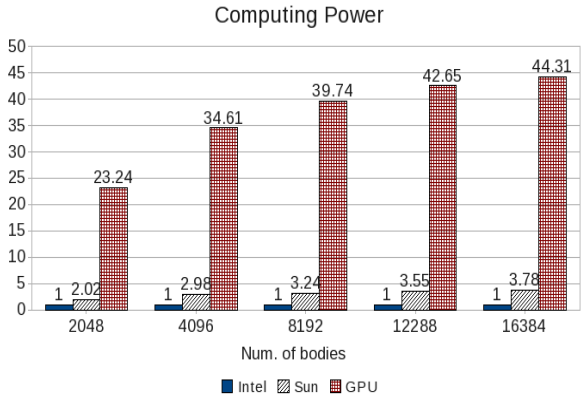

Figure 8: N-Body (long integer number)

integer operation. The reason is because the GPU was originally designed to off-load graphic rendering computation from the CPU, which is floating point intensive, and thus the GPU hardware is optimized for floating point operations.

3.4 Summary

In Section 3.1, we showed that by using multiple threads, the non-cached global memory access in GPU is comparable to the Intel processor. The results from Section 3.2 and 3.3 demonstrate the shared memory in GPU has a much better performance than global memory. There is a significant difference in performance between whether the shared memory is used or not. The GPU computing power is promising with a maximum of 322 times of the Intel CPU for matrix multiplication and a maximum of 83 times for the N-body problem. Even without using the shared memory, the GPU can obtain a 20 times to 50 times the computing power of the Intel CPU.

The scalability of the GPU is excellent for embarrassingly parallel problems such as matrix multiplication. However, when the global memory is used intensively for communication between threads in different MPs, the GPU is less scalable than the Sun UltraSPARC T1.

	GPU (GTX 8800)	Intel Pentium 4 2.8GHz
Computing power	332	1
In-use power(Watts)	257	112
PE ratio	140	1
Price (USD)	600	150
CE ratio	83	1

Table 5: GPU vs Intel

Power efficiency is an important factor for choosing computing resource. The power efficiency and cost efficiency of the GPU are given in Table 5 in terms of the Intel CPU.

In the table, we assume the GPU is 332 times more powerful than the Intel CPU, as demonstrated in our test results for the matrix multiplication problem. The power efficiency is calculated using the computing power divided by the in-use power. The PE ratio stands for power efficiency ratio in terms of the Intel PC (excluding the monitor). The GPU PE ratio is calculated using the GPU power efficiency divided by the Intel PC power efficiency.

The cost efficiency is calculated using the computing power divided by the price. The CE ratio stands for cost efficiency ratio in terms of the Intel CPU. The GPU CE ratio is calculated using the GPU cost efficiency divided by the Intel CPU cost efficiency.

From the table, we know that the GPU can be 140 times more power efficient than the Intel PC, and is 83 times more cost efficient than the Intel CPU.

From the above figures, we can conclude that a GPU can be a "greener" and cheaper computing resource.

4 Conclusions and future work

In this paper, we investigated a GPU as a general purpose computing resource based on two fundamental aspects: programmability and performance.

Due to the secretive nature of the GPU industry, some low level information about the GPU is (intentionally) vague. Programmers can only get a high level overview of CUDA programming on NVIDIA GPUs. However, from our initial experience, it is not difficult to learn GPU programming even for novice programmers. But it is certainly not trivial to achieve high performance programming with the GPU. For example, the use of the shared memory could dramatically improve the GPU's performance, but it also considerably increases the programming complexity.

Nevertheless, the GPU has improved significantly both in terms of programmability and performance. It performs particularly well on data parallel applications. Even for non-embarassingly parallel non-graphics applications, it performs reasonably well. Thus we argue that a GPU as a whole is a good candidate for general purpose computing.

Previous work has been done to investigate the possibility of using GPUs for GPU Cluster Computing and GPU Desktop Grids Computing ([10, 11]). But that research was based on older generations of GPUs. With NVIDIA's new GPU hardware and the CUDA programming model, we demonstrated the improved performance and programmability of a GPU. However, further research is needed to improve the GPU programmability for wider general purpose computing.

References

[1] NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 1.0. http://developer.nvidia.com/object/cuda.html, June 2007.

[2] UltraSPARC Architecture 2005 Specification. http://opensparc-t1.sunsource.net/, 2005.

[3] http://www.gpgpu.org.

[4] T. Dokken, T.R. Hagen, and J.M. Hjelmervik. The GPU as a high performance computational resource. In *Proceedings of Spring Conference on Computer Graphics 2005*, pages 21–26, 2005.

[5] Programming Massively Parallel Processors, Course Slides, University of Illinois, Urbana-Champaign. http://courses.ece.uiuc.edu/ece498/al1/index.html, 2007.

[6] Z. Huang and W. Chen. Revisit of View-Oriented Parallel Programming. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 801–810, 2007.

[7] Zhiyi Huang, Martin Purvis, and Paul Werstein. View-Oriented Parallel Programming and View-based Consistency. In *Proceedings of the Fifth International Conference on Parallel and Distributed Computing, Applications and Technologies (LNCS 3320)*, pages 505–518, 2004.

[8] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-D Tree GPU Raytracing. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 167–174, 2007.

[9] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *ATEC'96: Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, pages 23–23, 1996.

[10] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, 2004.

[11] Jian Wang, Aobing Sun, Yongbo Li, and Haitao Liu. Programmable GPUs: New General Computing Resources Available for Desktop Grids. *Grid and Cooperative Computing(GCC)*, 0:46–49, 2006.