

Accelerating HPC Using GPU's

Glenn Lupton, Don Thulin
High Performance Computing Division
Hewlett-Packard Company
June 13, 2008

1 Introduction

This paper provides an overview of the Nvidia and AMD GPU products and development environments and summarizes performance testing done by HP on one GPU+server configuration. This reflects a snapshot of the products and testing done at the time of this writing.

2 GPU Hardware and Software

Graphics Processing Units (GPU's) designed to accelerate graphics applications are highly parallel processors capable of 100's of Gflops/sec. Competition among the GPU vendors for market share in the PC gaming market has driven technological advancements in graphics cards, and the volume in this market has driven prices down.

These processors can be applied to non-graphical high-performance computing applications. Researchers have been investigating this usage for several years with success in a number of areas. This new market for their technology has been recognized by the main graphics card vendors, Nvidia and AMD/ATI. Both have introduced product lines specifically targeting high-performance computing, sometimes called the GPGPU market for general purpose computing on graphics processing units.

2.1 Nvidia Tesla Products

Nvidia introduced the Tesla product line in 2007. The first Tesla card is called the C870 and is based on their high-end graphics card, the FX 5600, but lacks video outputs and, at \$1,499, is priced at half the \$2,999 FX 5600. Like graphics cards, the Tesla C870 requires a PCI-Express 16x slot. It also draws a lot of power, about 170w, and space-wise it takes up two slots and is full-length. An important limitation of the Nvidia GPU's is that they only support single-precision floating-point arithmetic.



Figure 1
Tesla C870 – 170w, 1.5GB, 518Gflops peak

The slot-speed, space, and power requirements limit the systems that can accommodate a Tesla card. In general, only workstations that can accommodate two high-end graphics cards, such as the HP xw8600, can handle a Tesla C870. This is accomplished by using one of the two graphics slots for the Tesla card. HP also has servers that can accommodate the C870. The HP DL385G5 is a 2U server that has the necessary power and space to host one C870, and the DL785G5 (available May 2008) has 3 PCI-E 16x slots and power for two 225w cards.

Nvidia has also introduced a Tesla product that opens up additional possibilities for connecting Tesla cards to a rack-mount server system. This is the Tesla S870, which is a 1U rack-mount box that contains 4 Tesla C870 cards. It has its own power supply, and connects to a rack-mount server system via cables using two low-profile PCI-E 16x adapter cards. It is priced at \$12,000.



Figure 2
Tesla S870 – Four Tesla C870's in 1U – 800w



Figure 3
Adapter card for Tesla S870 – low-profile PCI-E 16x Gen2

This is a particularly good match for the HP DL160G5, which contains two PCI-E 16x Gen2 slots. When equipped with two quad-core processors, the DL160G5 teams 8 cores with 4 GPU's. This combination will be discussed in some detail later.

The Nvidia GPGPU programming environment is called CUDA (Compute Unified Device Architecture). It includes:

- nvcc C compiler
- CUDA FFT and BLAS libraries
- profiler
- documentation including many examples

The CUDA programming environment can be used with many Nvidia graphics cards, as well as the Tesla cards.

2.2 AMD FireStream

AMD made FireStream GPGPU boards available for early adopters and developers in 2007. In November 2007, they announced the FireStream 9170, intended for production use, with general availability expected in Q2 2008. The board is priced at \$1,999, draws less than 100w, but is similar in size to the Tesla C870. Unlike the Nvidia GPU's, the FireStream supports double-precision, but at an estimated 102Gflops peak compared to the single-precision performance of 500Gflops peak.



Figure 4
AMD FireStream 9170 – 100w, 2GB, 500Gflops peak

While its power requirements are less demanding, the slot and space requirements require it to be hosted in the same HP systems as the Nvidia C870. The HP DL385G5, a dual-processor AMD Opteron server, has the necessary space and power to support this card.

Early FireStream boards were programmed using AMD's CTM (Close to Metal) technology. This was too low level and code was tied to specific chips. This has been replaced with a CAL (Compute Abstraction Layer), which is still a low-level interface but code should be compatible with future GPU's.

The high-level interface to FireStream is via the Brook+ compiler, an AMD implementation of the open source Brook compiler developed at Stanford University that extends C for stream computing. They also plan an implementation of ACML (AMD Core Math Library) including BLAS, LAPACK, and FFT routines. At this writing, testing of the FireStream 9170 in an HP DL385G5 server is just commencing.

2.3 Rapidmind

An alternative to the vendor-supplied programming environments, RapidMind provides extensions to C++ and support libraries that allow the same source code to be optimized to multi-core, GPU, or Cell environments.

3 Accelerator Performance

Of course the interesting question is: how much faster can one make applications run by using GPU's? As one should expect, the answer is: it depends. It depends on the nature of the application itself, and what you using as the basis for comparison, for example, code that is single-core or multi-core, or optimized or not. Given some approximation of expected performance, one can evaluate other considerations such as the effort to port, price/performance, performance/watt, and performance/space.

3.1 Nvidia Tesla S870 with HP DL160G5

This section summarizes performance testing performed on the combination of the Nvidia Tesla S870 and HP DL160G5 server.

3.1.1 Configuration

The configuration tested is:

- DL160G5 – 1U Intel server with one of the two processors installed. This server contains one 2.66GHz E5430 quad-core processor and 16GB of memory.
- Tesla S870 – 1U Nvidia box with four C870 Tesla cards.

The Tesla S870 connects to the DL160 via two interface cards installed in the DL160's two 16x Gen2 slots. The interface cards are cabled to the S870.

The DL160 is running Red Hat EL 5.1. CUDA 1.1 is installed.

3.1.2 Benchmarks

Four benchmarks have been ported to the Nvidia Tesla S870 so far:

1. matmatmul – matrix matrix multiply
2. FFT – One and two dimensional FFT benchmark
3. bandwidthTest – tests bandwidth for writing to and reading from the card
4. Monte Carlo Black-Scholes

3.1.3 matmatmul – matrix matrix multiply

This benchmark computes $C = A*B + C$, where A, B, and C are matrices. It is run for a range of dimensions from 100 to 10000. In all cases the matrices are dimensioned as square matrices. Thus, it is testing a subset of the functionality of the BLAS SGEMM and DGEMM routines.

Only the single precision version of matmatmul was run, since Tesla does not support double precision. The Nvidia CUDA SDK includes a cublas library that provides a subset of the blas library functions. The cublasSgemm function was used to implement this benchmark on Tesla. The cublas library includes functions to copy data from the host to the card and copy results back from the card to the host, and these were also used for all tests. Thus, no board-side code needs to be written or compiled to use these cublas functions, but C code needs to be written to make the CUDA and cublas calls to:

- Select which Tesla card to use
- Copy input data from the host to the card
- Calculate the result by calling cublasSgemm
- Copy the results back to the host

This sequence is straightforward to code in C and does not require any knowledge of optimizing board-side code.

In addition to the cublasSgemm included in the CUDA 1.1 SDK, we tested a tuned version of a sgemm sent to us by Nvidia. Comments in the code say that it was written at

UC Berkeley. Note that sources for the cublas library are available from the Nvidia website. This version runs faster than the cublasSgemm function. It implements a subset of sgemm options and only accepts array-size parameters that are certain multiples of powers of 2. While all multiples of 64 can be used, some array dimensions can be multiples of 4 or 16. Some results for this function are included.

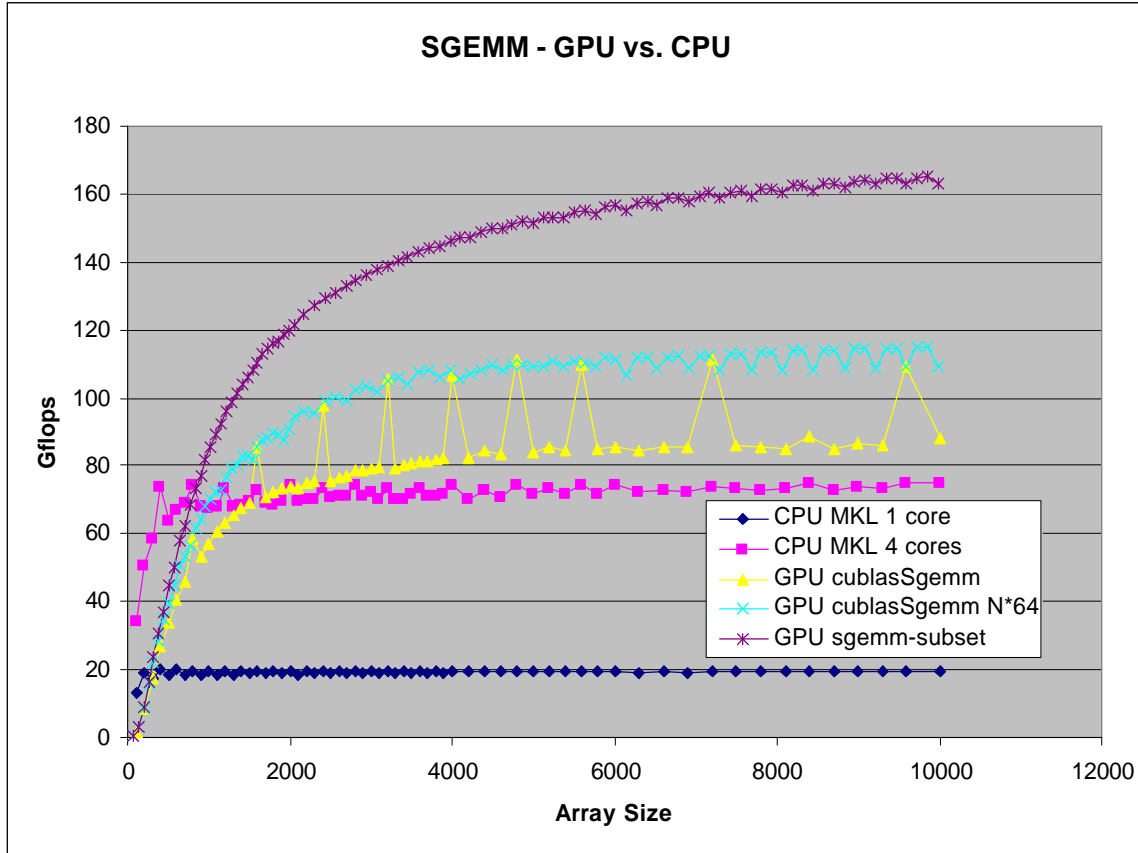


Figure 5

Figure 5 shows the Gigaflops/second measured for a single-precision matrix-matrix-multiply executed on the DL160G6 with one 2.66GHz E5430 quad-core processor and on a Tesla C870 in a Telsa S870.

The bottom two curves show the performance measured using SGEMM in the Intel MKL library with one or four cores of the processor. The MKL library implementation of SGEMM should be considered highly optimized code that has been tuned carefully by experts. This implementation also allows SGEMM to use multiple threads under the control of the environment variable OMP_NUM_THREADS, so it is code that has already been modified to take advantage of multiple cores. The next line shows the performance using cublasSgemm from the CUDA 1.1 library. This measurement of Gflops/second includes the time needed to copy data to and from the GPU card. Note that there are some significant peaks in the results at certain array sizes. The next line (cublasSgemm N*64) uses the same code but uses only array sizes that are multiple of 64.

The top line shows the performance of the CUDA SGEMM that has been tuned for a subset of SGEMM arguments.

Gflops	Processor	Description
2.1	CPU	Simple implementation
19.6	CPU	Optimized library (MKL)
74.6	CPU	Optimized library – 4 threads
82.2	GPU	cublasSgemm
105.5	GPU	cublasSgemm n*64
147.2	GPU	Sgemm-subset n*64
181.0	GPU	Sgemm-subset n*64, no I/O
205.4	GPU	Sgemm-subset n*64, no I/O, 32b (64b Linux CUDA compiler fix needed)

Table 1
SGEMM Gflops for ~4Kx4K Arrays

Table 1 shows some values from the chart in figure 5 for an array size of approximately 4Kx4K. It also includes the performance of a simple implementation of the SGEMM operation that has not been optimized, and some GPU performance measurements that do not include I/O, that is, the time to write arrays A, B, and C to the card and read back the results in C. The bottom table entry shows a measurement from a 32-bit Linux system, which performs better than the 64-bit system; this is due to a compiler problem which is expected to be fixed. Note the wide variations in performance for both the CPU and GPU. This results in an even wider range of speedups that can be reported as shown in Table 2 below. Depending on what you choose to compare, speed-ups can range from about 1x to about 100x

Speed-up - GPU vs. CPU						
		GPU Gflops				
		82.2	105.5	147.2	181.0	205.4
CPU Gflops	2.1	39.1x	50.2x	70.1x	86.2x	97.8x
	19.6	4.2x	5.4x	7.5x	9.2x	10.5x
	74.6	1.1x	1.4x	2.0x	2.4x	2.8x

Table 2
SGEMM Speed-up for ~4Kx4K Arrays

What speed-up value is reasonable? If you are starting with an application that was highly tuned for single-core performance, then you could consider your initial state to be comparable to the 19.6Gflops CPU value. Recoding to use the GPU might get you 4x to

10x improvement, while recoding to use 4 cores would get less than 4x. While further improvement in the CPU Gflops might be possible using 2 CPU's, CPU performance per core is not expected to improve much over the next year, but GPU performance is expected to improve significantly.

Of course, these speedups would only apply to applications that share characteristics with SGEMM or rely on SGEMM.

3.1.4 Bandwidth Tests

This benchmark is an HP test tool used to evaluate the data transfer characteristics of multiple GPGPUs on different system platforms. It is based on the original "bandwidthTest" example contained in the NVIDIA CUDA SDK, and has been modified and extended to support multiple GPGPUs simultaneously. There are four basic transfer types provided, device to host, host to device, device to device, and read after write. The transfer type, length of transfer, host memory buffer type (pinned or paged) can be selected on a per GPGPU basis, and any of the available GPGPUs may be selected in a test, providing a wide variety of test cases that can be performed.

The bandwidth tests whose results are shown below were conducted using the DL160G5 system platform configured with an NVIDIA S870 Tesla server. The S870 is connected to the DL160 via two PCIe GEN2 x16 interfaces, each interface carries data from 2 GPGPUs. Both PCIe GEN1 and PCIe GEN2 results are measured; this was done by disabling and enabling GEN2 mode in the DL160G5 in the BIOS. It is important to note that while the DL160 and the PCIe bridges in the S870 and interfaces are PCIe GEN2 capable, the current Tesla boards are only PCIe GEN1.

Bandwidth tests were conducted using data transfer sizes from 1,000,000 to 100,000,000 bytes in increments of 1,000,000 bytes. The transfer rate of a 50,000,000 byte transfer was selected to display in the graphs. Transfer types of host to device and device to host, using both pinned and paged host memory buffers are presented. The results are presented as an aggregate transfer rate, that is to say that it is sum of the transfer rates of all the GPGPU's used in a particular test, with 1, 2, 3, or 4 GPGPU's operating simultaneously.

The software used to conduct these tests consists of NVIDIA CUDA SDK and TOOLKIT version 1.1, NVIDIA Driver for Linux with CUDA (171.05) and Redhat Linux 5.1.

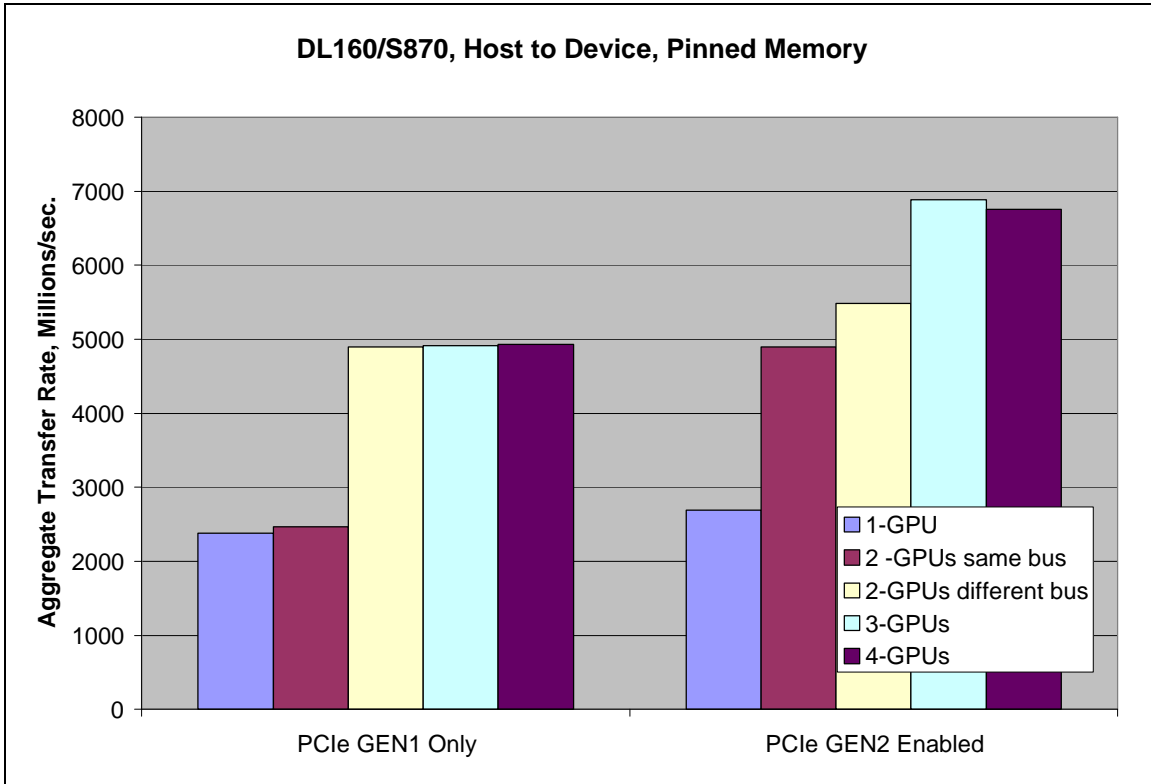


Figure 6

Figure 6 shows the measured transfer rate for 1, 2, 3 and 4 GPGPUs when transferring data from pinned (non-paged) memory host buffers to the GPGPUs. The data presented is for PCIe GEN1 and GEN2 enabled cases. In the case of a single GPGPU, PCIe GEN1 or GEN2 results in only marginal differences. It is when one looks at the 2 GPGPU cases that one sees a significant performance difference. With 2 GPGPUs on a single PCIe GEN2 bus, the performance is nearly twice that of a single GPGPU. In the PCIe GEN1 case, there is little performance gain. When 2 GPGPUs are split across independent PCIe busses, the resultant aggregate transfer rate is approximately twice that of a single GPGPU, regardless of the PCIe bus type. When a 3rd GPGPU is added, the PCIe GEN2 case continues to add to the aggregate transfer rate and the PCIe GEN1 case remains the same. It is at this point where one sees the limits of the system platform. While the results for PCIe GEN2 scale reasonably well for 1, 2 and 3 GPGPUs, the addition of a 4th GPGPU, in either the PCIe Gen1 or GEN2 cases, does not result in any additional aggregate bandwidth.

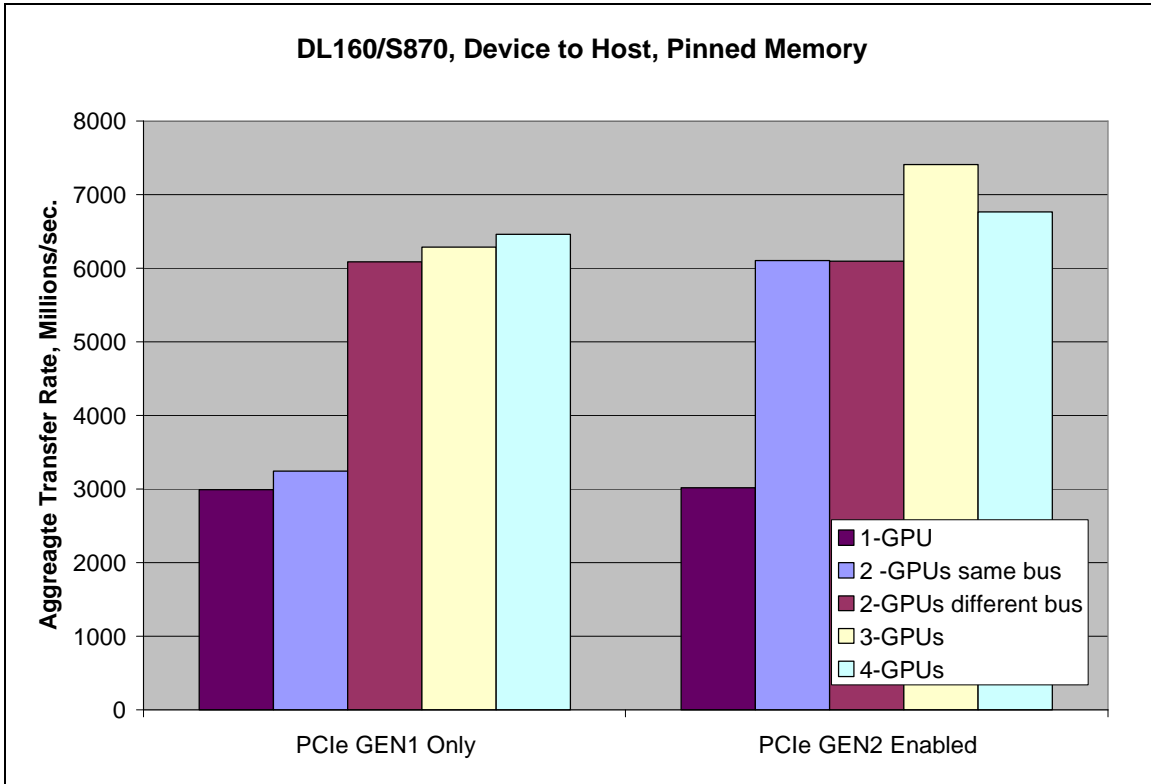


Figure 7

Figure 7 shows the measured transfer rate for 1, 2, 3 and 4 GPGPUs when transferring data from the GPGPUs to a pinned memory host buffers. This is the same test as in, Figure 6, except that the transfer direction is reversed. Overall transfer rates increase when compared to Figure 6 and is the expected result for this system platform. Again, one can see the effect the system platform has on the transfer rates. For PCIe GEN2, 1 and 2 GPGPUs, the transfer rate scales well. Adding a 3rd GPGPU increases the overall bandwidth by about 1/2 of what a single GPGPU is capable of performing. While adding a 4th GPGPU, decreases the overall transfer rate from the 3 GPGPU case.

Overall, these are impressive I/O rates, with transfers from device to host reaching a maximum of about 3GB/sec, showing that these GPU devices and host DL160G5 are utilizing the available PCIe 16x bandwidth.

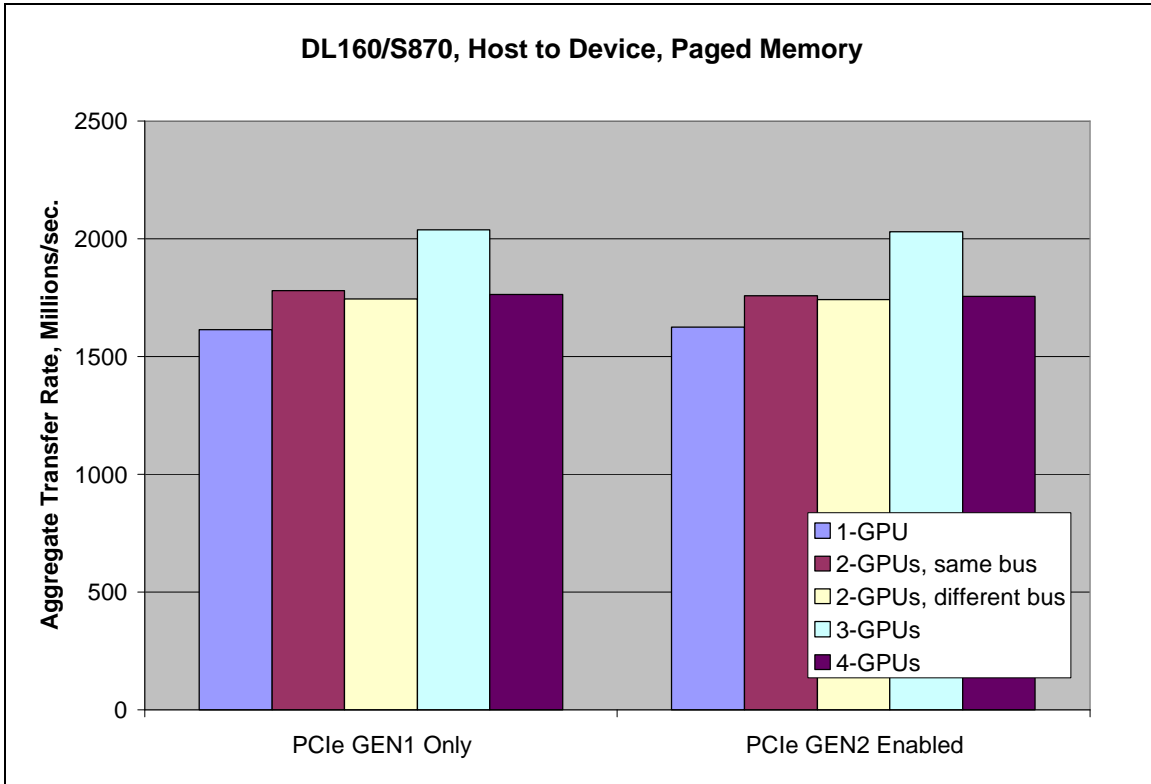


Figure 8

Figure 8 shows the measured transfer rate for 1, 2, 3 and 4 GPGPUs when transferring data from paged memory host buffers to a GPGPU. This is the same test as shown in Figure 6, except that the host memory buffers are allocated from normal paged memory. In comparing Figure 6 and this figure, one sees that the overall transfer rate declines significantly. The primary difference between pinned and paged host memory buffers is that the memory pages allocated for a pinned buffer are locking into memory and not subject to paging. In performing I/O to paged memory buffers, the O/S may be required to break up a large transfer into a series of smaller I/O transfers, whereas with pinned memory the O/S more likely perform a very large transfer as a single operation.

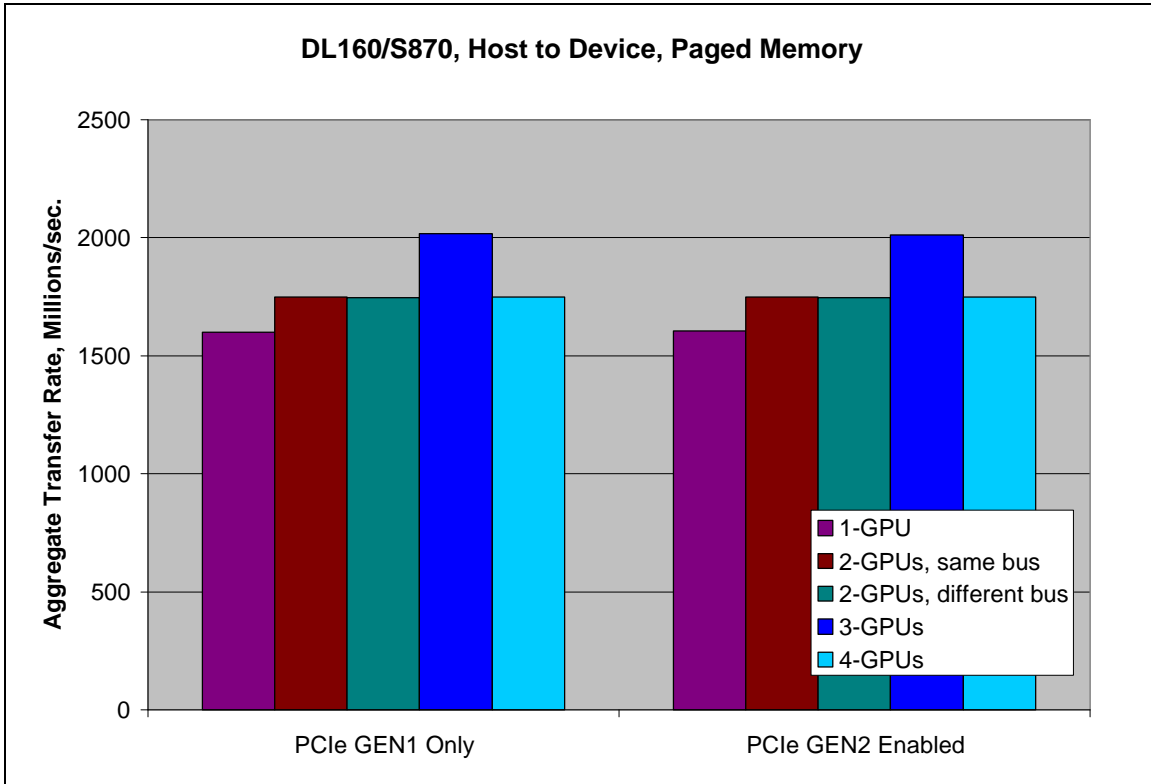


Figure 9

Figure 9 shows the measured transfer rate for 1, 2, 3 and 4 GPGPUs when transferring data from the GPGPUs to paged memory host buffers. This is the same test as shown in Figure 7, except that the host memory buffers are allocated from normal paged memory.

Thus, even though the cards inside the Tesla S870 are GEN1, enabling GEN2 in the DL160G5 resulted in significantly higher aggregate bandwidth when transferring data between GPU card and pinned memory when using two cards on the same bus or when using more than 2 GPU's. The benefits of GEN2 slots on the DL160G5 are expected to be even greater when GEN2 Tesla cards become available.

3.1.5 1D and 2D FFT

These benchmarks compute the discrete Fourier transform (DFT) using the fast Fourier transform (FFT) algorithm in one and two dimensions. The 1D FFT is performed for sizes varying in size from 2^1 to 2^{20} . The 2D FFT is performed for sizes varying from 2^2 to 2^{12} in each dimension.

Only the single precision version of FFT benchmarks was run, since S870 Tesla server does not support double precision floating point. The NVIDIA CUDA SDK includes a cufft library that provides APIs for 1D, 2D, and 3D real and complex FFTs. The cufft library provides a programming interface that is similar to the well known FFTW open source software package.

To use the cufft library, one first creates a plan that defines the size, number of dimensions, and type of FFT to be performed, real to real, complex to complex, etc. As cufft library does not include functions to copy data to/from the GPGPU, the cudaMemcpy function is used for this purpose. Once the data has been moved to the GPGPU, the FFT is performed by calling the cufftExecC2C function. This then performs a complex to complex FFT.

The benchmark consists of a timed loop that performs two FFTs, one forward and one inverse. The results of the inverse transform are compared to the input data to the forward transform and tested for accuracy as they should be same. Unlike Intel's MKL library, the cufft library does not offer the ability to scale the result of an FFT. In order to do the comparison the results of the inverse transform are scaled by the reciprocal of the transform size using a small CUDA kernel written as part of this benchmark. Thus the timed portion of the benchmark looks like this:

- Create the FFT plan
- Copy input data from the host to the Tesla GPGPU
- Calculate the forward FFT
- Calculate the inverse FFT
- Scale the results
- Copy the results from the Tesla GPGPU to the host
- Destroy the FFT plan

In order to provide accurate timing, smaller FFTs may be performed up to a 100 or a 1000 times before the execution time is calculated. Larger transform sizes may be performed a few as 1 or 10 times.

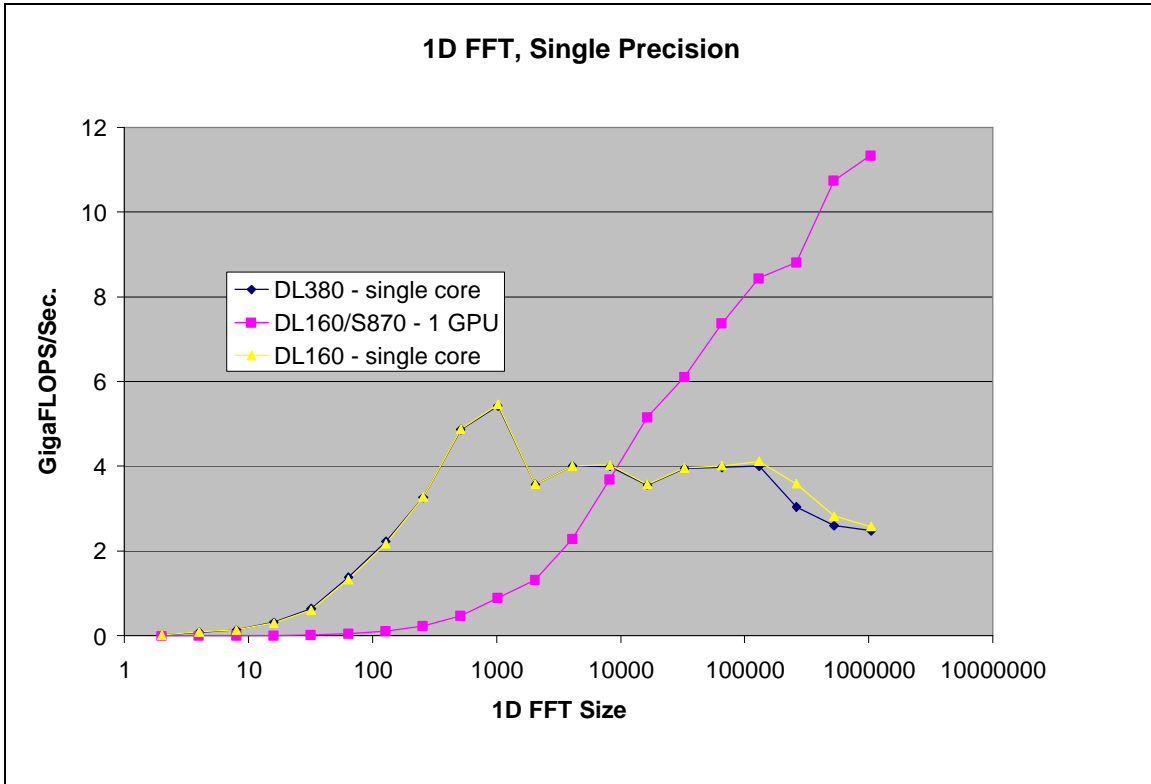


Figure 10

Figure 10 shows the Gigaflops/sec. measured for 1D complex-to-complex forward and inverse FFT. The lines on the graph represent results for a single core DL160, DL380 and a single S870 GPGPU. Until approximately the 8192 transform size, the CPUs outperform the GPGPU, probably the result of having to move the transform data to and from the GPGPU. After that point, the GPGPU results continue increase, whereas the CPU results remain constant and the fall off as the transform size increases. It should be noted that the NVIDIA cufft library allows for batching of 1D FFT transforms, where one is able to provide the data for multiple FFTs in a single transfer to the GPGPU, and then perform all the transforms in a single library function call. The current benchmark is unable to take advantage of these unique capabilities of the library due to its structure and this is likely to provide higher transform rate for smaller 1D FFT transforms.

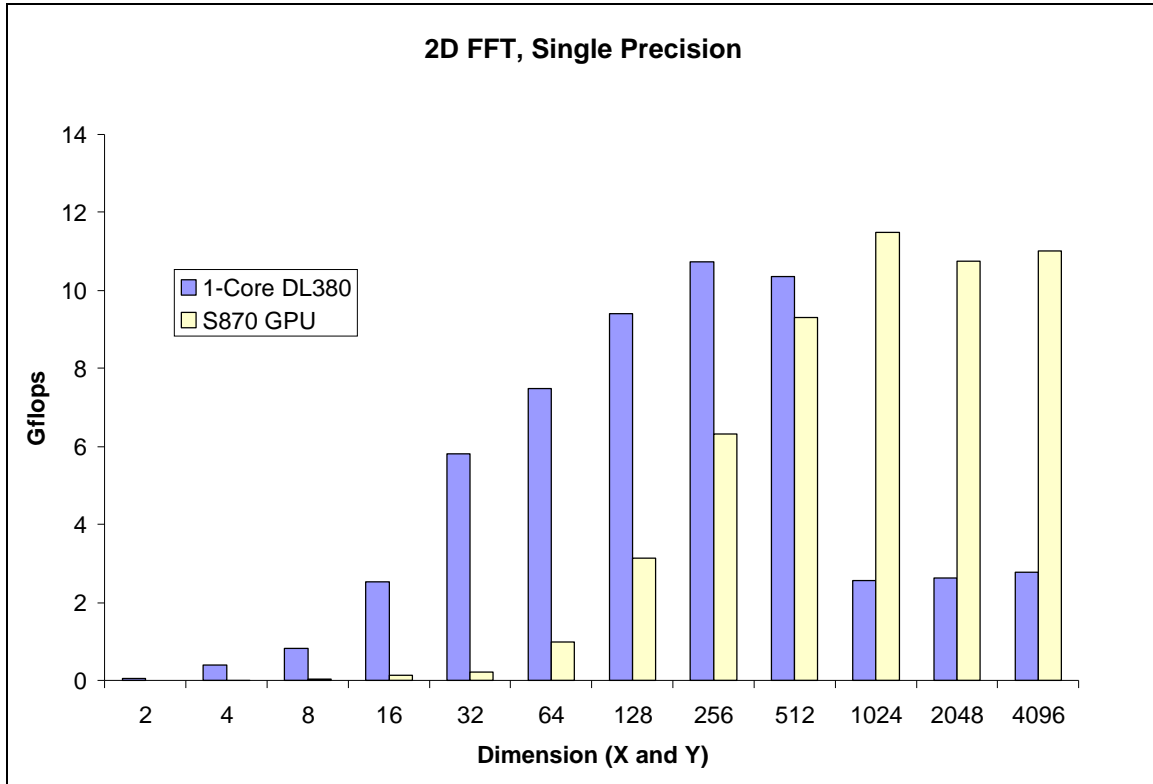


Figure 11

Figure 11 shows the Gigaflops/sec measured results for 2D complex-to-complex forward and inverse FFT. The bars on the graph represent results for a single core DL380 and a single S870 GPGPU. The results shown are only for square transform sizes, this was done for sake of simplicity of the graph. Similar to the 1D results, it shows that the GPGPU can outperform the CPU, but only when the transform size is above a certain value.

These results only show an advantage to FFT on the GPU vs. CPU for large sizes. However, this does not take into consideration that a real application would perform some computations on the GPU between the forward and inverse FFT. It should also be noted that forward transforms can be considerably faster than inverse transforms on the GPGPU. We have measured speeds that are up to 3 times as fast for forward transforms when compared to inverse transforms for larger problem dataset sizes. Some published performance measurements are for forward transforms only, and some do not include the time to copy data to/from the GPU memory. However, the HP benchmark is designed to time the complete set of operations, including both the forward and inverse transforms, which allows the benchmark to perform an accuracy check of the computations when compared to the CPU. In addition, using pinned memory transform arrays would likely provide increased performance; the benchmark is not structured to take advantage of this method of memory allocation.

3.1.6 Monte Carlo Black-Scholes

This benchmark is an evaluation of European Stock Option Pricing using a Monte Carlo simulation of the Black-Scholes equation. The benchmark evaluates only a single stock option, from 1 to 256 million times, each evaluation is referred to as an experiment. An experiment consists of a draw from the random number generator, an evaluation of the equation, and a sum of the option price and sum of the square of the option price. This benchmark was chosen for its very high computational density and minimal amount of input/output data.

The benchmark uses a Hammersley sequence to generate a uniform random number sequence. This is then transformed to a normal distribution using a polar form Box-Muller transform. This method of generating random numbers was chosen for its ability to be implemented in a highly parallel form. In both implementations of the benchmark, for multi-core CPUs and the GPGPU, the same random number sequence is generated.

In the multi-core CPU version of the benchmark parallelism is achieved through the use of OpenMP pragmas and setting the OMP_NUM_THREADS environment variable to specify the number of threads to be used in the simulation. In the GPGPU version, a CUDA kernel was developed to execute on the GPGPU. The GPGPU operates as a coprocessor to the host system and is capable of executing a very high number of threads in parallel. The number of threads used in the simulation is determined at run-time, when the kernel is launched. For this benchmark, 4096 threads were used.

The benchmark as implemented on the NVIDIA GPGPU generates an array of partial sums of the option price and the square of the option price. Each partial sum is generated by a single thread running on the GPGPU. At the end of the simulation, the array is transferred to the host and the CPU generates the final sums and option pricing.

Only a single precision version of the benchmark was run, as the S870 only supports single precision floating point format.

The Black-Scholes NVIDIA GPU results shown below were conducted using the DL160G5 system platform configured with an NVIDIA S870 Tesla server. Only one of the four GPU's in the S870 was used. The CPU results are from running the benchmark on a BL460c server blade. The BL460c is a dual-socket quad-core 3.0GHz Intel Xenon based platform. The software used to conduct NVIDIA S870 test consists of NVIDIA CUDA SDK and TOOLKIT version 1.1, NVIDIA Driver for Linux with CUDA (171.05) and Red Hat Enterprise Linux 5.1 64-bit. The software used to conduct the CPU based test consists of the Intel C++ compiler for Linux, Intel Math Kernel Library for Linux and Red Hat Enterprise Linux 4.

Monte Carlo Black-Scholes (Single Precision)

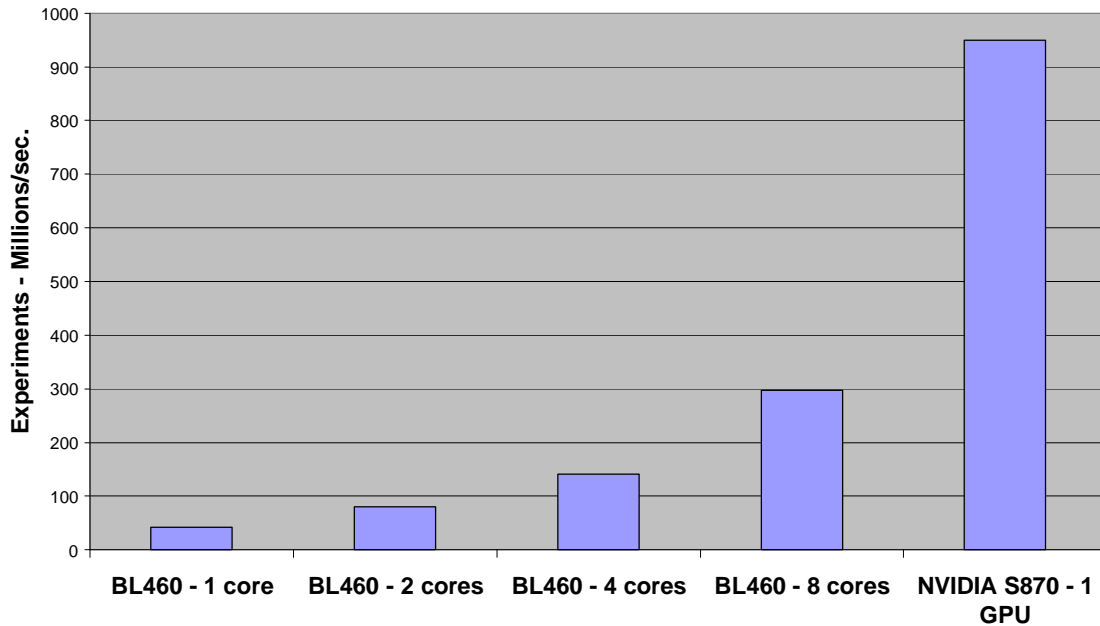


Figure 12

Figure 12 shows the results of the Monte Carlo Black-Scholes simulation in millions of experiments per second. The data points shown are for the maximum results achieved over the course of simulations containing 1 to 256 million experiments. For the CPU results, one sees a near linear scaling over 1 to 8 CPU cores. This is an expected result given the highly parallel nature of the benchmark.

The results for the NVIDIA GPGPU demonstrate the computational capability of the GPGPU, providing a 23X increase when compared to a single CPU core and a 3X increase for an 8-core platform. In addition to the execution time of the benchmark on the GPGPU, results include all of the necessary overhead operations to allocate memory on the GPGPU and host, transfer of parameters and the execution kernel to the GPGPU, transfer of the results from the GPGPU to the host, and final calculations on the host. In this benchmark, the overhead items are essentially fixed for each simulation and have an effect on the results that can be achieved for a given simulation size. For example, in a simulation with 4 million experiments the GPGPU results are 12X that of a single-core CPU result; with 16 million experiments one sees 19X that of a single-core CPU result. Simulations with larger numbers of experiments rapidly approach the maximum observed.

4 Summary

A number of things have been learned thus far in our investigations:

- A GPU can deliver 10x the single-precision Gflops of CPU core, but a wide range of speedups can be stated for a given problem. It is important to describe the conditions of both the GPU and CPU execution of the computation.
- In making comparisons to CPU performance, it is important to note if the GPU performance includes the time to transfer data to and from the GPU board.
- The HP DL160G5 server is a good match for the Tesla S870 given the DL160G5's two PCIe Gen2 16x slots. Its Gen2 also helps deliver more bandwidth, but pinned memory must be used to realize this.
- CUDA BLAS and FFT libraries provide optimized GPU implementations of these functions, and do not require expertise in optimizing code for the GPU. However, applications will likely require some user-written GPU code to be used in combination with calls to these libraries.
- FFT on the GPU outperforms the CPU, but only if the transform size is sufficiently large. Smaller sizes may be a win depending on the other computations to be carried out on the data prior transferring the data back to host memory. Using pinned verses paged memory buffers may also be a win depending on transfer sizes. Also, batching of 1D transforms needs to be considered, these are likely to effective over many transforms. A key to achieving good acceleration is to have a high ratio of computation to data movement.
- Applications that require little data transfer, have long computation times, and are readily adapted to use parallelism such as Monte Carlo Black-Scholes show impressive speed-ups compared to optimized multi-core implementations.
- The addition of additional interfaces to the overall benchmark framework to accommodate the features of accelerators is clearly a beneficial endeavor. For instance, allowing a particular accelerator to allocate memory using a mechanism optimal for that accelerator is likely to provide an improved result.

Future investigations will include an analysis of the effect of competition between GPU's for I/O bandwidth. Comparisons to CPU performance using up to 8 cores will also be done. The results will be extended to include AMD FireStream 9170, which we have just begun to test.