Performance Analysis of Memory Transfers and GEMM Subroutines on NVIDIA Tesla GPU Cluster

Veerendra Allada, Troy Benjegerdes
Electrical and Computer Engineering, Ames Laboratory
Iowa State University
Ames, IA, USA
allada@iastate.edu, troy@scl.ameslab.gov

Brett Bode
National Center for Supercomputing Applications
Urbana-Champaign, IL, USA
bbode@ncsa.uiuc.edu

***Abstract*—Commodity clusters augmented with application accelerators are evolving as competitive high performance computing systems. The Graphical Processing Unit (GPU) with a very high arithmetic density and performance per price ratio is a good platform for the scientific application acceleration. In addition to the interconnect bottlenecks among the cluster compute nodes, the cost of memory copies between the host and the GPU device have to be carefully amortized to improve the overall efficiency of the application. Scientific applications also rely on efficient implementation of the Basic Linear Algebra Subroutines (BLAS), among which the General Matrix Multiply (GEMM) is considered as the workhorse subroutine. In this paper, we study the performance of the memory copies and GEMM subroutines that are crucial to port the computational chemistry algorithms to the GPU clusters. To that end, a benchmark based on the NetPIPE [1] framework is developed to evaluate the latency and bandwidth of the memory copies between the host and the GPU device. The performance of the single and double precision GEMM subroutines from the NVIDIA CUBLAS 2.0 library are studied. The results have been compared with that of the BLAS routines from the Intel Math Kernel Library (MKL) to understand the computational trade-offs. The test bed is a Intel Xeon cluster equipped with NVIDIA Tesla GPUs.**

***Index Terms*—Performance, GPU cluster, NetPIPE, CUDA, CUBLAS, Tesla, Math Kernel Library**

I. Introduction

Applications that exhibit high amounts of data level parallelism show significant amount of performance speedups on Single Instruction Multiple Data (SIMD) architectures. Vector processors, such as the Cray X-MP are one of the earliest such architectures that pioneered in scientific computing. Though vector architectures had semantic advantages like reduced number of instructions per program (loop execution in single instruction), better mechanisms for branch handling [2], etc., expensive high speed on-chip memory and design costs limited their use to scientific computing. Many of the concepts revived as short-vector instructions/SIMD extensions in general purpose computing. Notable among them are the MMX, Streaming SIMD extensions (SSE) and Altivec. Computational demands from the real-time graphics and gaming applications opened new avenues for specialized hardware graphics accelerators. The earliest 2-D and 3-D hardware graphics accelerators had dedicated logic (shader cores) for different operations in the graphics pipeline (vertex, triangle, pixel and rendering units). Because of the unbalanced graphics workloads, programmable shader cores were designed to unify the different graphics operations into a single architecture. Hence the programmable shader cores opened a new domain for general purpose computing.

While the motivation for unified graphics architecture came from the nature of the graphics algorithms, the need for having a programming model for general purpose computing came from the potential speed ups that could result from reusing the same graphics hardware for non-graphics application acceleration. Developing non-graphics applications via graphics API's such as the OpenGL and DirectX was merely a hack with a limited exposure to hardware resources and also reduced portability. Brook for GPU [3], a compiler and runtime implementation of the Brook stream programming language made an earlier attempt to provide general purpose computing on modern graphics hardware. To enable flexible programmable graphics and general purpose computing, NVIDIA came up with a hardware/software architecture called as the Compute Unified Device Architecture (CUDA) [4]. A set of development tools and compiler were released within this new framework. With an increasing interest to develop non-graphics algorithms for the GPU hardware, this field is rapidly progressing under the umbrella General Purpose Computing on Graphical Processing Units) (GPGPU) [5], [6]. AMD (ATI Technologies), another major player in developing the GPU hardware and standards designed a programming model called Brook+ that is based on the Brook GPU [3] for the FireStream GPU series . There are also sustained efforts towards the development of novel GPGPU technologies for future such as Intel Larrabee architecture [7] and Open Computing Language framework [8].

The higher performance per price ratio and the general purpose computing model using the CUDA toolkit has placed the NVIDIA GPUs abreast to the contemporary High Performance Computing (HPC) technologies. Among the NVIDIA CUDA enabled GPUS, the Tesla series is specifically designed for the scientific computing domain. Clusters equipped with Tesla GPUs might not be readily usable by many scientific codes which have been traditionally parallelized to execute on a homogeneous system of compute nodes. The cost of the data transfers between the host and co-processor (GPU) memory spaces, cache effects, sustained floating point operation (flop)

rate, hardware speed and software overheads are some of the important factors that need to be well understood to redesign and develop efficient parallel algorithms that can utilize the hardware resources on the modern heterogeneous GPU clusters. These insights would also in a a broader scope be useful to develop a unified framework under which a comparative analysis can be made among clusters deployed with other types of application accelerators such as PowerXCell 8i [9] and Field Programmable Gate Arrays (FPGA) [10].

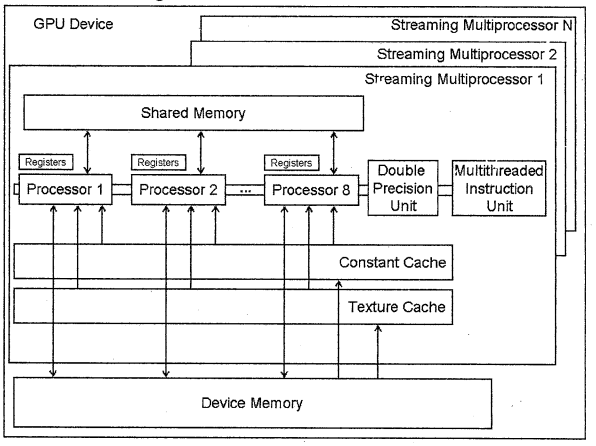
An important issue that pops up in the co-processor computing model is the overhead involved in transferring data to the device memory space vs. the actual computation time. This may be exacerbated in scientific algorithms where data has to be fetched across iterations from a storage device. Many scientific applications also depend on fine-tuned numerical libraries such as the Basic Linear Algebra Subroutines (BLAS). In this paper, we present the results of the the latency and throughput of the memory copies between the host and the GPU device and Single, Double precision General Matrix Multiplication (SGEMM/DGEMM) subroutines provided by the CUBLAS library. The latency-throughput test is based on the Network Protocol Independent Performance Emulator (NetPIPE) [1] benchmark. Memory transfers between the paged/page-locked (pinned) buffers on the host and the device are studied. The performance results of the GEMM routines from the CUBLAS library are compared to that of the routines from the threaded implementation of the Intel Math Kernel Library (MKL). All the tests are run on an Intel Xeon cluster equipped with NVIDIA Tesla S1070 GPUs.

The remaining paper is organized as follows. Section 2 provides an overview of the GPU technologies that are mostly relevant to the HPC domain - the Tesla S1070 compute system, CUDA programming model and the CUBLAS library. Section 3 explains the NetPIPE benchmark mechanism and reports the performance results of the memory transfers. Section 4 compares the performance results of the CUBLAS SGEMM/DGEMM subroutines and the Intel MKL BLAS routines. The subsequent sections give the related work, future work and conclusions.

A. NVIDIA Tesla for Scientific Computing

The unified GPU hardware architecture [11] is built around a scalable array of Streaming Multiprocessors (SM). Each SM consists of eight scalar Streaming Processors (SP), two special functional units, multi-threaded instruction unit (MT-IU) and on-chip shared memory. The NVIDIA Tesla computing cards/system for HPC differs from the graphics counterparts (Quadro and GeForce) in terms of the processor clocks, memory configuration, and computing features. The earlier generation Tesla GPUs (C870 graphics card and S870 graphics system) offered single precision support. The latest feature in the Tesla processor series is the introduction of double precision floating point support in the hardware (Tesla T10 processor and its derivatives) that make them amenable to scientific applications that rely on the higher numerical precision. The architecture of the Tesla hardware is shown in Figure 1

Fig. 1. Tesla Hardware Architecture



Each Tesla T10 computing processor has 4GB of dedicated memory and 240 SPs. Table I summarizes the key differences between the Tesla S870 and S1070 compute systems. Either of these processors can be programmed using the CUDA framework, as explained in the next section.

TABLE I
COMPARISON OF TESLA S870 AND S1070 COMPUTE SYSTEMS

Features	Tesla S870	Tesla S1070
No. of GPUs	4	4
No. of SP cores/GPU	128	240
Processor frequency	1.35 GHz	1.296 GHz
Numerical precision	IEEE 754 single	IEEE 754 single and double
Memory per GPU	1.5GB	4 GB
Peak memory bandwidth	76.8 GB/sec	102 GB/sec
Memory interface	384-bit, 800MHz GDDR3	512-bit, 800MHz GDDR3
System interface	Two PCIe x16 Gen1	Two PCIe x16 Gen2

B. CUDA Programming Model

One of the main objectives of the CUDA programming environment is to develop scalable and efficient parallel programs [4]. In this model, the GPU is viewed as a highly multi-threaded compute device capable of executing many threads in parallel. The threads execute a sequence of instructions in a single-instruction multi-threaded fashion (SIMT). Compute intensive parts of the application are isolated into functions (called in the NVIDIA terminology as the *kernel*) and compiled into the instruction set architecture of the GPU device. The CUDA programming interface is designed with a minimal set of extensions to the C/C++ language [12]. To manage the device contexts, memory, and data transfers between the host and device, the CUDA framework provides two types of

application programming interfaces (API's). They are the low-level CUDA driver API and the high-level CUDA runtime API. The low-level API offers a better level of control, is language independent, but is harder to program. The CUDA runtime API is easy to program and provides a subset of the C standard library and built-in vector types.

C. CUBLAS Library

The BLAS library [13] is a set of subroutines that provide standard building blocks for performing the basic vector and matrix operations. Level 3 BLAS routines handle the matrix-matrix operations and are computationally expensive. An efficient implementation of BLAS is highly desirable, as many scientific applications and other numerical linear algebra software packages such as the LAPACK, ScaLAPACK are build on those routines. CUBLAS [14] is the implementation of the BLAS on the top of NVIDIA CUDA driver for the GPUs. The CUBLAS library calls are used to create matrix and vector objects in the GPU memory space and fill them with data from the host memory. A sequence of CUBLAS calls are executed and the results are copied from the GPU space to the host memory. The CUBLAS API uses a calling convention akin to that of the C programming language and a FORTRAN type column-major order for storing matrix objects. To be compatible with legacy FORTRAN codes, two interfaces (thunking and non-thunking) are provided by the vendor that can be compiled using the standard FORTRAN compilers. The thunking interface can be used directly in an application without any modifications. When using the non-thunking interface, the application has to allocate and deallocate the matrix objects using device pointers and explicitly handle the data movement between the CPU and GPU memory spaces.

D. GPU Cluster

The GPU cluster is configured with compute nodes comprising of the Intel Xeon processors and Tesla S1070 blade server as shown in the Figure 2. The Tesla S1070 blade server has four Tesla T10 GPUs and two PCI express x16 interfaces to the host systems. In our particular configuration, two compute nodes are connected to the S1070.

Each compute node of the cluster has two Core2 Quad processors (Intel(R) Xeon(R) E5405) that are clocked at 3.0 GHz and has 8 GB of main memory. The Core2 Quad processor has 2 X 6MB L2 cache per core pair and a front side bus running at 1333MHz. Each compute node has access to 2 GPUs of the S1070 server. Each Tesla GPU has 4 GB of dedicated device memory connected via a GDDR3 interface. The CPU-GPU subsystem architecture is shown in Figure 3. The BLAS implementation from the Intel Math Kernel Library (MKL - Version No: 10.0.1.014) [15] is used to evaluate the performance of the GEMM subroutines on the CPU. The Intel MKL library provides highly optimized and multi-threaded math routines based on the OpenMP library implementation.

Fig. 2. Cluster of compute nodes connected to the Tesla S1070

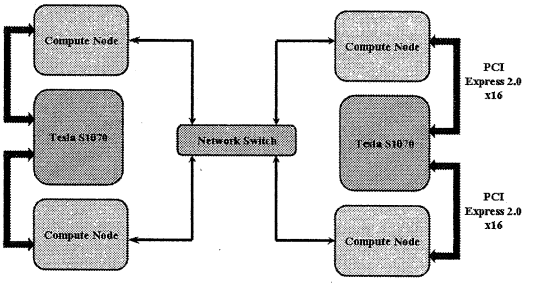
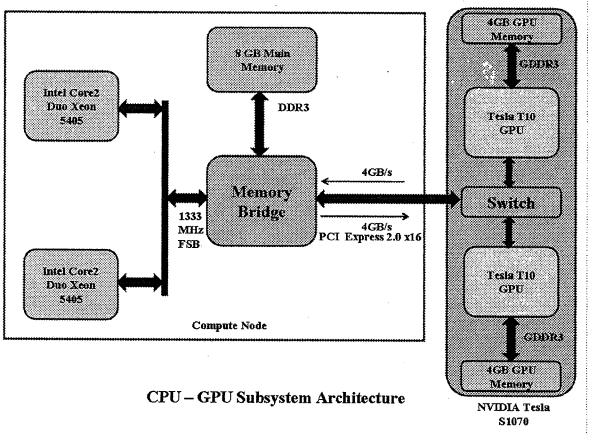


Fig. 3. Compute node sharing two GPUs from Tesla S1070



II. NetPIPE cudaMemcpy

The micro benchmark is based on the Network Protocol Independent Performance Emulator (NetPIPE) [1] framework and measures the latency and throughput of the memory copies between the host and the GPU device. NetPIPE is a variable time benchmark , based on the principles developed in the HINT benchmark by Gustafson et al. [16]. The buffer sizes are increased at regular intervals with slight perturbations, i.e. for each buffer size of c bytes, three measurements are taken for c-p, p and c+p bytes, where p is the user defined perturbation value. To evaluate the memory transfer performance, the NetPIPE benchmark program performs streaming i.e one-sided memory copies between the host and device or vice-versa and a round-trip host to device memory copy in a ping-pong fashion. An additional test to measure the performance of the device to device memory copies is also available. The default test is the round-trip host-to-device memory copy. The latency and throughput of the memory copies are the measured performance metrics. The latency is used to calculate the throughput of the memory copies.

The core algorithm used by the NetPIPE benchmark tool

is shown in Algorithm 1. The value of nrepeat is calculated based on the time of the last data transfer as explained in the original NetPIPE paper [1]. This allows the tests to run enough times such that the total time of the runs is greater than the timer resolution. The default target time is 0.25 seconds. Eq. (1) gives the formula to compute the nrepeat. The minimum value of latency is taken after running the test for a fixed number of times (NTRAILS), which by default is set to 7.

$$nrepeat = TARGET/((bsz2/bsz1) * tlast) \quad (1)$$

where tlast is the last transfer time for the buffer of size bsz1 and bsz2 is the buffer size for which the value of nrepeat is calculated.

```
Input: streaming/ping-pong
Input: host-device/device-device
Input: paged/pinned host memory
Output: Latency in seconds
Output: Throughput in Gigabits per second
/* Set the variables based on the input */
RT ← 1 if round-trip, else RT ← 0
Paged ← 1 if paged, else Paged ← 0 if pinned
/* Set T to a large value */
T = MAXTIME
for i = 1 to NTRIALS do
    t0 = Time()
    for j = 1 to NREPEAT do
        if RT then
            copy buf from host (paged/pinned)to dev
            copy buf from dev to host (paged/pinned)
        end
        else
            copy data from host/device to device/host
        end
    end
    t1 = Time()
    /* Keep the minimum value of T */
    T = min(T,t1-t0)
end
T = T/((1+rt)*NREPEAT)
```

Algorithm 1: NetPIPE algorithm for memory copies

The NetPIPE benchmark modules namely the cudaMemcpy and cuMemcpy are developed using the CUDA runtime API and driver API respectively (These APIs are discussed in the next section). The benchmark mechanism is however the same irrespective of the module. The results that are reported in this paper are using the NetPIPE cudaMemcpy module. The output file contains the buffer size, throughput and the transfer time.

A. Evaluation of Memory Latency and Throughput

The memory buffer is allocated as a linear array on the device. The memory buffer on the host can be allocated in one of the following two ways - as a paged memory array using the malloc() system call or as a page-locked (pinned) memory buffer using the CUDA API. The advantage in the later case is that the device to host memory bandwidth is higher due to asynchronous access of the memory using DMA, however at the expense of reduced available system memory for the operating system and other applications.

For the NetPIPE throughput plot, the packet size in Bytes is shown in a logarthmic scale on the x-axis and the throughput achieved in Gigabits/sec on the y-axis. The throughput value in the plot is shown in bits/sec rather than Bytes/sec in order to maintain the consistency with the other NetPIPE modules that traditionally reported the throughput in Megabits/sec. However, the results that are discussed in the text are in GB/s to compare with the physical link speeds that are usually reported in GB/s. The NetPIPE latency plot has the packet size in Bytes on the x-axis and latency in Seconds on the y-axis. Both axes are represented in the logarithmic scale.

As a verification step, the results that are obtained using NetPIPE benchmark are validated against the reference benchmark results provided by the CUDA SDK. The timing measurements by the reference benchmark are done using the CUDA timer functions. The timing function that is used for NetPIPE is based on the gettimeofday() function. The results seem to be consistent to that of the reference benchmark. In all the results that are discussed below, the NetPIPE test is executed up to buffer sizes of 192 MB.

Figures 4 shows the throughput and latency of the streaming or one-way memory copies from the host to the device. With the paged memory buffers on the host, we found that that throughput increased linearly and attains a maximum at 6MB of buffer size. The variation towards perturbation is not seen. For the host pinned memory, throughput reaches the saturation of 2.85 GB/s around buffer size of 8MB.

A comparison of the results from the NetPIPE and the reference benchmark for the streaming host (paged) to device memory copy is shown in Table II

TABLE II
COMPARISON OF NETPIPE RESULTS WITH CUDA SDK BANDWIDTH TEST

SDK Results		NetPIPE Results	
Size (Bytes)	MB/s	Size (Bytes)	MB/s
16855040	2012.5	16777216	2043
33632256	2075	33554432	2087
67186688	2102	67108864	2103

Figure 5 shows the throughput of the one-way memory copies from the device to the host. For transfers between the device and host paged memory, the throughput reaches a maximum value of 2.0 GB/s around 64 MB. The throughput behaviour hence is similar to the host to device copy. It rises linearly and reaches the knee of saturation around 8 MB. With the pinned buffers on the host, the maximum throughput achieved is 3.11 GB/s.The latency plots show relatively constant latency until 10 KB and rises roughly linearly with packet size beyond 1 MB.

Figure 6 shows the ping-pong throughput and latency curves for memory copies between the host paged/pinned buffers and the device. For ping-pong copies between the host paged memory buffers and the device, a maximum throughput of 1.6 GB/s is observed. For the pinned buffer on the host, the throughput is 2.9 GB/s. The PCIe 2.0 bidirectional bandwidth is 8 GB/s and hence the link is not completely saturated. For pinned memory

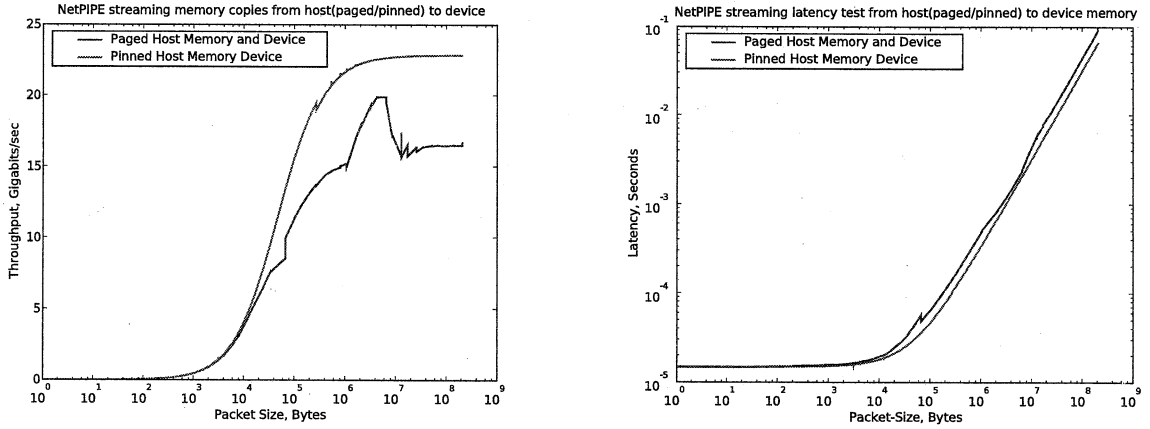


Fig. 4. (a) NetPIPE throughput curve for streaming memory copies from host (paged/pinned) memory to the device (b)NetPIPE latency curve for streaming memory copies from host (paged/pinned) memory to the device

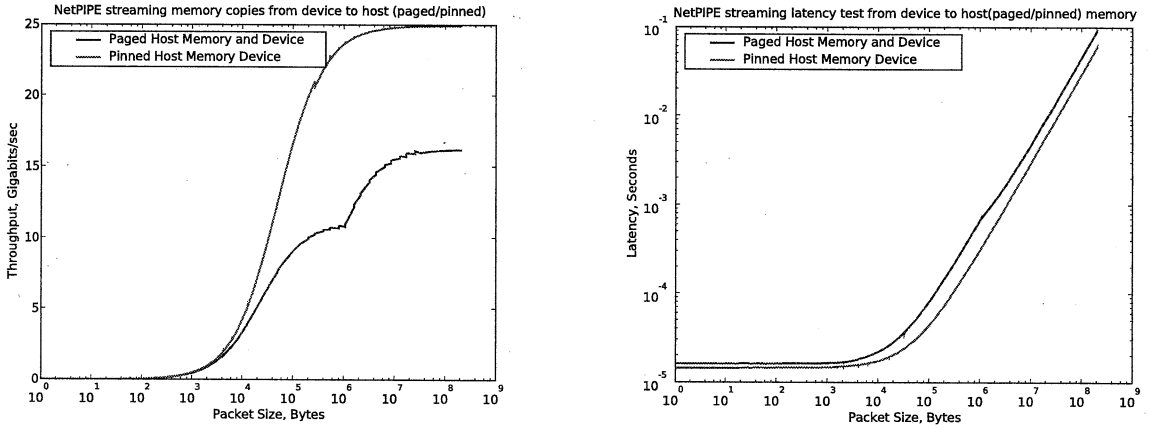


Fig. 5. (a) NetPIPE throughput curve for streaming memory copies from device to host (paged/pinned) memory (b)NetPIPE latency curve for streaming memory copies from device to host (paged/pinned) memory

the curve linearly rises till 8 MB buffer size, beyond which the throughput remains constant. The throughput plot doesn't show any affects due to the perturbations. From the data set we found that the ratio of the average round-trip latencies for paged memory copies to that of pinned memory varies between 1 and 3. This shows that the pinned memory buffers show a considerable advantage for the application.

Each GPU in the S1070 has access to 4GB of dedicated memory. The memory interface is 512-bit GDDR3 and has a peak bandwidth of 102 GB/s. Figure 7 shows the device to device throughput and latency curves. We had achieved a peak device throughput of 72 GB/s around 16 MB of buffer size. As evident from the throughput curves, there are perturbation affects on the throughput for the odd buffer sizes (c - p and c + p bytes). The latency curve remains constant till 1 KB (we presumed some startup overhead) beyond which it rises linearly.

III. Performance of SGEMM and DGEMM

Single/Double Precision General Matrix Multiply (SGEMM/DGEMM) are BLAS3 matrix-matrix subroutines. The DGEMM is considered to be one of the important subroutines that is used in scientific computing and is often tuned by vendors for their own architectures to achieve the best performance. Moreover, the widely accepted LINPACK benchmark [17] that measures the systems floating point performance uses the DGEMM subroutines.

The GEMM function performs one of the operations given by

$$C := alpha * op(A) * op(B) + beta * C \ ,$$

where alpha and beta are double precision scalars and A, B, C are matrices of dimensions m x k, k x n and m x n respectively.

The operation op(X) is defined as

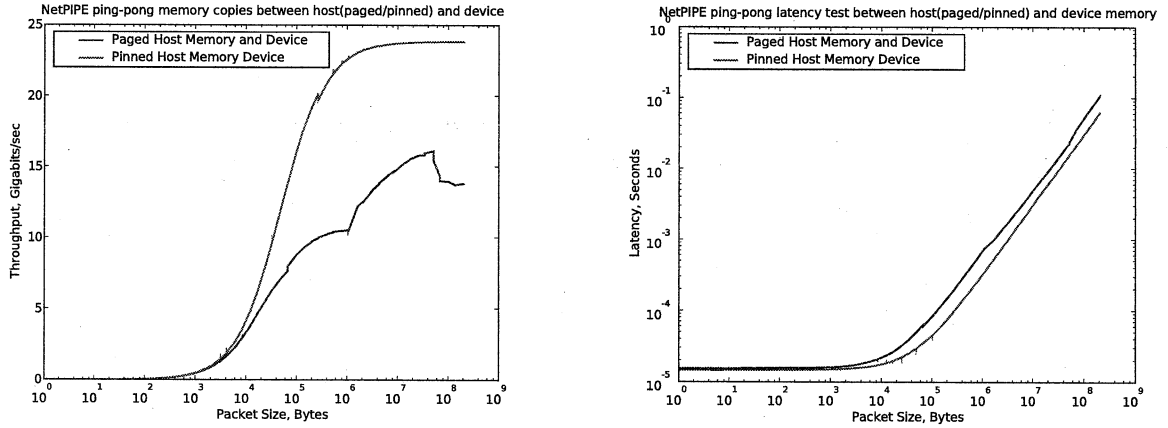


Fig. 6. (a) NetPIPE ping-pong memory copy throughput between the host paged and page-locked(pinned) memories and the device (b)NetPIPE ping-pong memory copy latency between the host and device for paged and page-locked memories

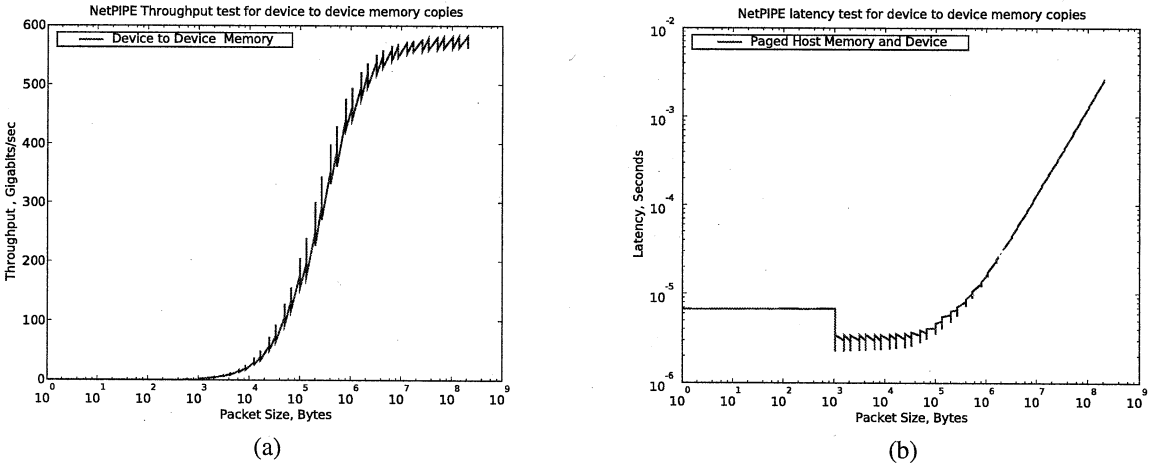


Fig. 7. (a) NetPIPE device to device memory copy throughput curve (b) NetPIPE device to device memory copy latency curve

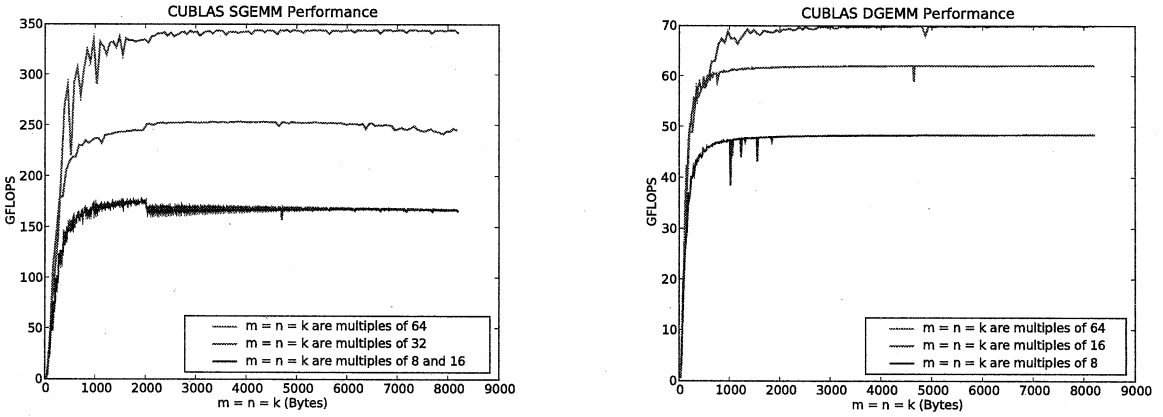
$$op(X) = X \text{ or } op(X) = X'$$

The Tesla T10 processor has 30 SM's with each SM having 8 SPs. Each SM has one double precision floating point unit and a special functional unit. A single double precision multiply and add operation (1 MAD = 2FLOPS) can be performed by the double precision unit. Using the standard convention of measuring theoretical FLOPS as a product of the floating point operations per cycle, clock frequency, and the number of functional units, the double precision theoretical peak for the T10 processor is 30 * 1.3 GHz * 2 FLOPS = 78 GFLOPS. The single-precision theoretical peak is 240 * 1.3 GHz * 3 = 936 GFLOPS.

The performance of the cublasSgemm and cublasDgemm subroutines is shown in Figure 8. The results are plotted for square matrices (m = n = k). The size of the matrices are increased in steps of 8 in each dimension (row and column). The GFLOPS are calculated without taking the memory transfer time into account. This gives an indication of the raw performance of the GPU device. The data set obtained for the cublasSgemm showed a particular pattern in terms of the performance. When m, n, k dimensions are exactly divisible by 64 (64 x 64 blocking), the subroutine showed peak performances. With multiples of 8 or 16 it shows the minimal performance and intermediate performance for multiples of 32. A partial dataset for both the precisions is shown in Table III with sizes varying from 8128 to 8180 in increments of 8.

The performance of cublasDgemm also showed a similar pattern with the peak performance for multiples of 64, 32, intermediate performance for multiples of 16 and worst performance for 8. However, when compared to the cublasSgemm, the difference between the highest and the intermediate performance curves is relatively very small.

For the GEMM routines, the ratio of the number of floating



(a) (b)

Fig. 8. (a) cublasSgemm performance for different matrix sizes (b) cublasDgemm performance for different matrix sizes

point operations to the data transfers to the GPU memory space is in the order of $O(\frac{N^3}{N^2})$. For matrices of larger dimensions, the GFLOPS calculated by taking into account the memory transfer times is slightly lesser than that of the GFLOPS obtained from the computations alone. The ratio is 0.95 for the cublasDgemm and 0.87 for the cublasSgemm. Since many scientific applications run GEMM's on matrices with large dimensions the plots shown can be taken as real-time performance metrics.

TABLE III
CUBLAS SGEMM AND DGEMM PERFORMANCE

cublasSgemm		cublasDgemm	
Size (m=n=k)	GFLOPS	Size (m=n=k)	GFLOPS
8128	344.32	8128	69.96
8136	167.08	8126	48.46
8144	167.42	8144	62.11
8152	167.46	8128	48.46
8160	245.67	8126	62.11
8168	167.13	8144	48.46
8172	166.58	8126	62.11
8180	166.08	8144	48.44

A. Performance of SGEMM and DGEMM on CPU

Figure 9 shows the performance scaling of the GEMM operation on the Dual Core2 Quads on the machine. The number of double precision floating point operations per cycle for the Core2 Quad is 4 and the theoretical GLOPS are 4 * 3GHz * 4 cores = 48 GFLOPS. The number of single precision floating point operations per cycle is 8 and has 96 GFLOPS of theoretical performance. The scaling curves show the performance of the MKL GEMM subroutines for 1, 2, 4, 8 threads of execution. The number of threads that can be executed is controlled via OMP_NUM_THREADS environmental variable on the Linux systems. The peak single and double precision GFLOPS when running with 8 threads are 157.8 and 75.82 respectively. The performance scales linearly with the number of threads. Hence, we conclude that the subroutines are well parallelized with good utilization of the processor cores.

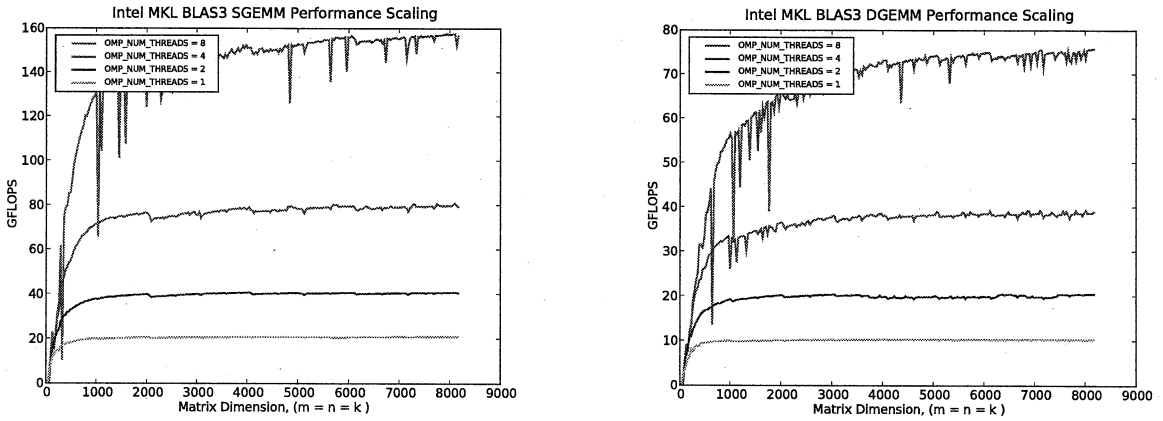
When running with 8 OMP threads, the Thread Affinity Interface is used to bind all the threads to the available physical processing units. The interface is controlled via the KMP_AFFINITY environment variable. This interface is a very powerful mechanism to control the thread execution on Intel multicore processors. A detailed reference is available from the Intel Compiler documentation [18].

B. Performance Comparison of CPU and GPU

The DGEMM subroutine from the CUBLAS library achieves a peak of 69 GFLOPS than that of the single Core2 Quad processor which is close to 40 GFLOPS. However, there is a very close match in performance (~76GFLOPS) when the MKL DGEMM call is executed using 8 threads on the compute node (dual Quads). However the cublasSgemm performance largely surpasses the MKL sgemm call. This brings up an important issue for the application developers to perform computations in single precision whenever the precision effects do not impact the application behaviour. The overall performance of the application might be significantly improved by using this mixed approach on GPUs. Since many scientific applications run iteratively, data would be available in the GPU memory space and hence may show good performance for matrices of lower dimensions, where the transfer time might have affected otherwise.

IV. RELATED WORK

The use of GPUs as a co-processor in cluster computing was studied by Fan et al. [19] for simulating the airborne contaminants using the lattice Boltzmann model from the Computational Fluid Dynamics (CFD). Since CUDA framework was not available at that time, they used the graphics API for accelerating the application. However, the scaling curves might behave in a similar way with the CUDA framework.



(a) (b)

Fig. 9. (a) Scaling of Intel MKL SGEMM subroutine using multiple threads (b) Scaling of Intel MKL DGEMM subroutine using multiple threads

Barrachina et al. [20] evaluated the performance of the single precision CUBLAS routines and proposed hybrid algorithms with computations split across the CPU and the GPU. Ryoo et al. [21] studied performances on the Geforce 8800 GTX architectures and compared various parameters like concurrent active threads, speedup achieved, % of cpu-gpu transfer time, % of GPU execution time for a suite of application kernels developed using the CUDA framework. After analyzing some of the classical optimization principles like loop unrolling and shared memory buffering for improving the access patterns of the global memory, they observed that compute-intensive kernels that have low global memory accesses showed good speedups over the CPU. The most dated evaluation of the GPU for linear algebra subroutines and memory performance is done by Vasily Volkov et al. [22]. For larger matrices, the performance speedups were shown to be attaining 80 to 90% of the the theoretical peak. The NVIDIA CUDA SDK also provided a reference bandwidth benchmark program that helped us validate the results. Unlike the reference benchmark, NetPIPE is a variable time benchmark that can take into account a user-defined perturbation value and runs the test for a wide range of buffer sizes.

V. Conclusions and Future Work

In this paper, we studied the performance of the SGEMM/DGEMM subroutines from the CUBLAS 2.0 library on the latest NVIDIA Tesla compute system. Also presented are the overheads in memory copies between the device and host. During the development of NetPIPE cudaMemcpy module the results were validated against the reference benchmark results provided by the CUDA SDK. Based on the results in this paper, we conclude that the cudaMemcpy module of NetPIPE is accurate and can be used to measure data movement performance between the host and the GPU device. As a future work, we would like to integrate this module with the NetPIPE Infiniband module to incorporate memory copies to remote GPU devices using the Remote Direct Memory Access (RDMA) features of the Infiniband. We also intend to extend the NetPIPE framework to incorporate the performance of the asynchronous memory copies. These findings may be useful for GPU cluster application developers and system administrators likewise.

VI. Acknowledgements

This manuscript has been authored by Iowa State University of Science and Technology under Contract No. DE-AC02-07CH11358 with the U.S. Department of Energy. The authors wish to thank the Air Force Office of Scientific Research and NVIDIA Corporation for providing funds and equipment to build the GPU cluster. The authors also wish to thank Professor Mark S Gordon for his support and Andrey Asadchev for his useful insights during the CUBLAS and MKL performance analysis and other CUDA related discussions. We also acknowledge Mark Klein at the Scalable Computing Laboratory for his efforts in setting up the GPU cluster and other timely support.

References

[1] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "Netpipe: A network protocol independent performace evaluator," in *In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004, pp. 777–786.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[5] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x

[7] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[8] http://www.khronos.org/opencl/.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[10] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13–13, 2007.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[12] NVIDIA, *CUDA Programming Guide 2.2*, 2009.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[14] NVIDIA, *cuda CUBLAS Library 2.1*, 2008.

[15] Intel, *Intel Math Kernel Library for Linux* OS*, 2009.

[16] J. Gustafson and Q. Snell, "Hint: A new way to measure computer performance," *Hawaii International Conference on System Sciences*, vol. 0, p. 392, 1995.

[17] J. J. Dongarra, "the linpack benchmark : an explanation," in *Proceedings of the 1st International Conference on Supercomputing*. New York, NY, USA: Springer-Verlag New York, Inc., 1988, pp. 456–474.

[18] Intel, thread Affinity Interface. [Online]. Available: http://software.intel.com/en-us/intel-compilers/

[19] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47.

[20] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Orti, "Evaluation and tuning of the level 3 cublas for graphics processors," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, April 2008.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 73–82.

[22] V. Volkov and J. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC*, 2008, p. 31.