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
&
Brett Bode
National Center for Supercomputing Applications
University of Illinois, Urbana-Champaign

Workshop on Parallel Programming on Accelerator Clusters
IEEE Cluster 2009, New Orleans, August 31, 2009

THE Ames Laboratory
Creating Materials & Energy Solutions
U.S. DEPARTMENT OF ENERGY

Outline

- Introduction
- Tesla Processor for Scientific Computing
- CUDA programming model
- NetPIPE – discuss performance results
- GEMM routines from CUBLAS and Intel MKL
- Conclusions & Future work

Introduction

- Commodity clusters are augmented with special purpose computing devices for scientific application acceleration
- Contemporary architectures – GPU, Cell Processor, FPGA etc.
- GPU's are widely deployed due to their high computational density, performance per price ratio, and active programming framework support by vendors
- In our research, the algorithms that are currently targeted for acceleration come from the computational chemistry domain.

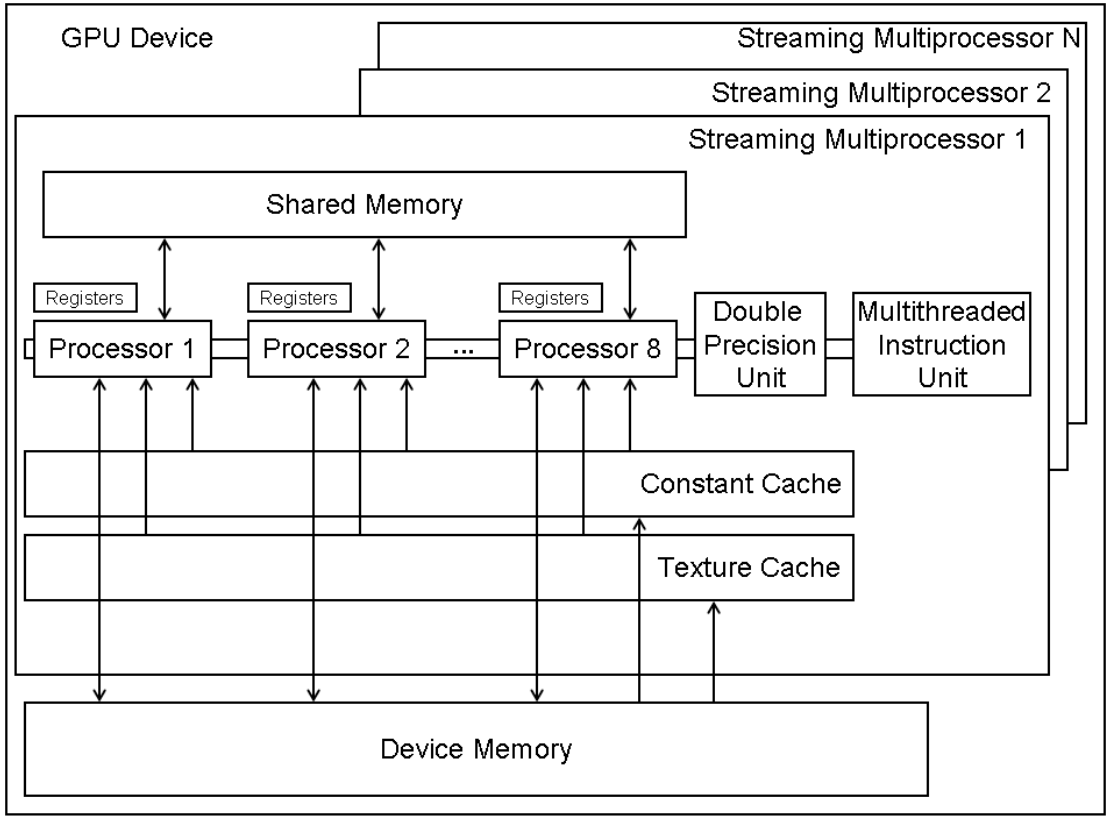
Introduction

- Studies were focused towards understanding communication bottlenecks among the cluster compute nodes
- Understanding bottlenecks in transferring data to the coprocessors might be useful for application developers to achieve a balanced computing
- Computational chemistry algorithms also depend on the matrix-vector and matrix-matrix subroutines.
- In this work we focused on the Level 3 BLAS routines SGEMM/DGEMM and memory transfer throughput and latencies

NVIDIA TESLA for Scientific Computing

Features:

- Scalable array of streaming multiprocessors (SM)
- Eight scalar processors (SP) per SM
- Two special functional units and a single double precision unit
- Fast on chip shared memory
- 4 GB of GPU device memory



NVIDIA TESLA HARDWARE ARCHITECTURE

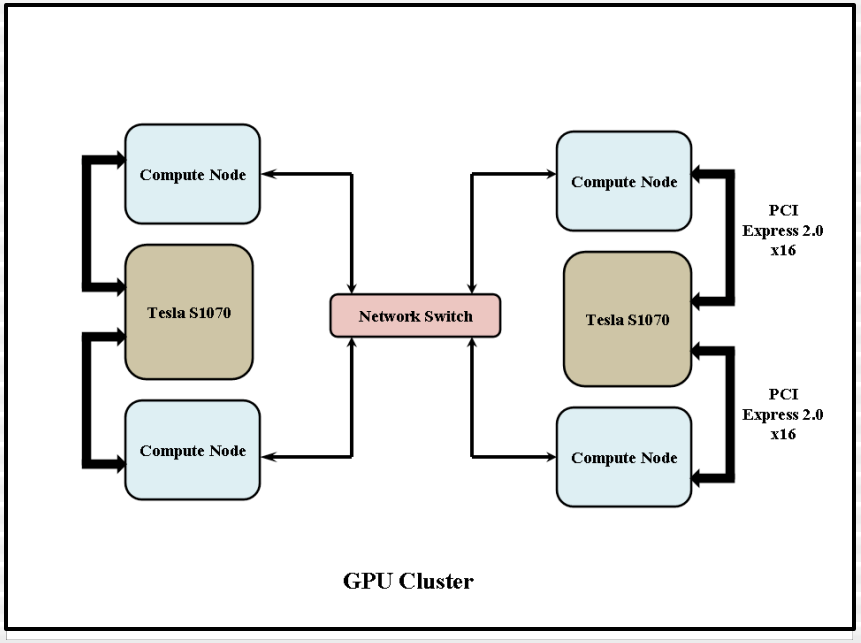
Reference: NVIDIA CUDA Programmers Guide

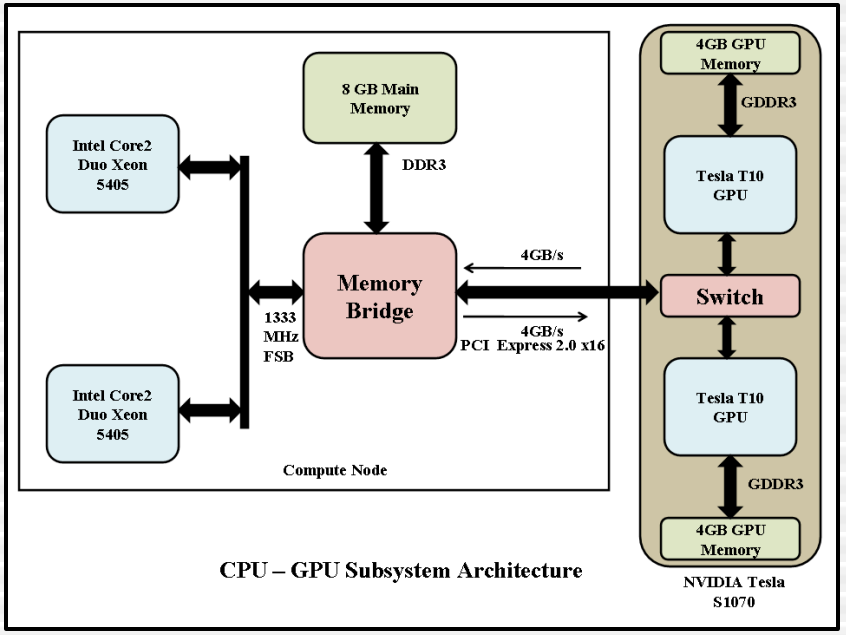
CUDA Programming Model

Salient Features

- GPU is viewed as a highly fined grained multithreaded compute device
- Capable of executing many threads in parallel using a Single Instruction Multi-Threaded (SIMT) model
- Performance benefits due to overlapping computing and memory fetches among the threads
- Designed with a minimal extensions to C/C++ programming language
- Two programming APIs – CUDA high level run-time API and low level driver API

Tesla GPU Cluster





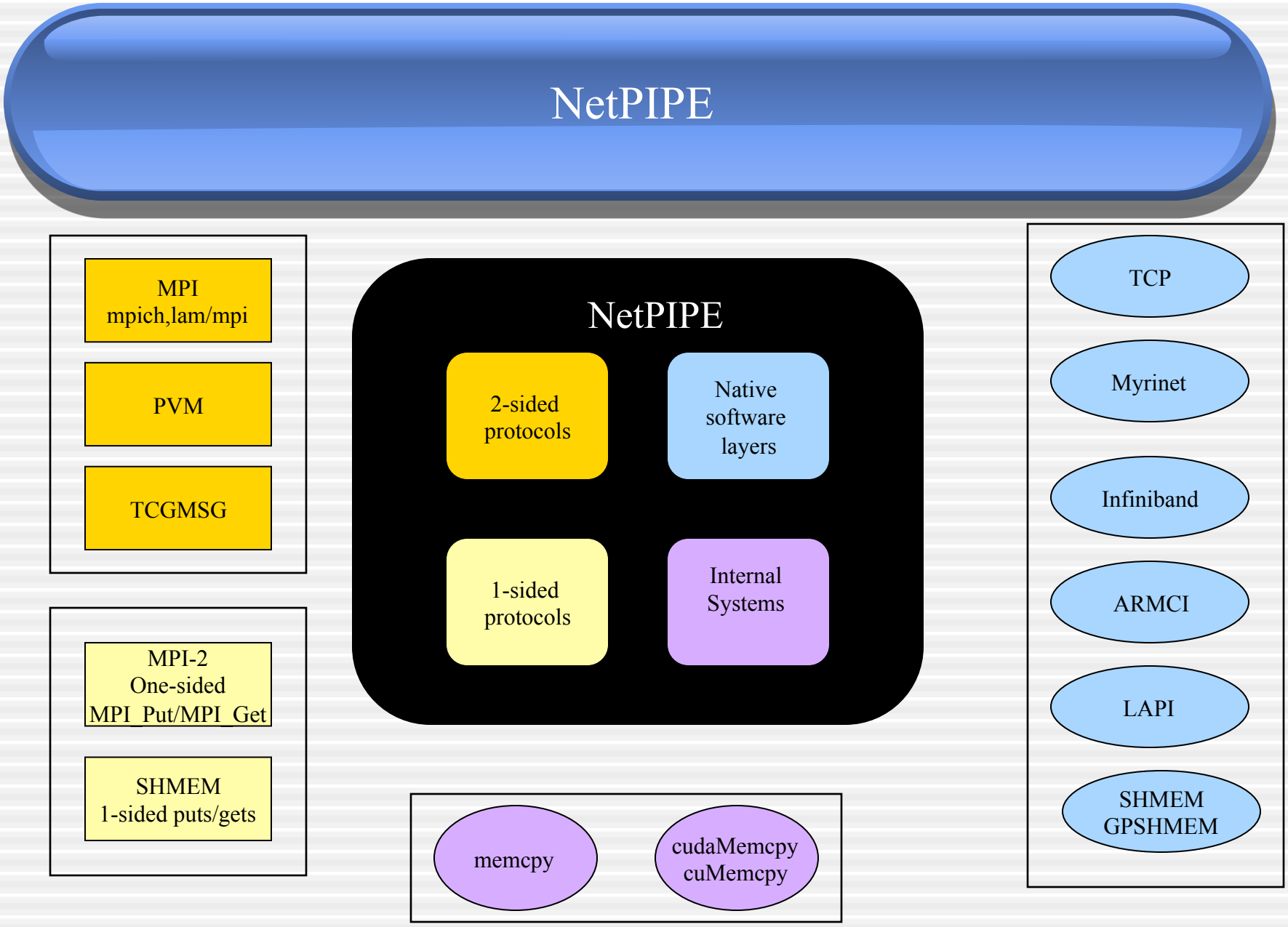
- Tesla S1070 system – 4 Tesla T10 processors each with 4GB device memory
- Compute node – Dual Intel Core2 Quad Xeon 5405, 8 GB of main memory
- Each compute node is connected to two T10 GPU's via PCI Express 2.0 link

Network Protocol Independent Performance Emulator (NetPIPE)

- Network Protocol Independent Performance Emulator
- Micro benchmark that measures the latency and throughput of memory transfer, based on the principles of HINT benchmark
- Neither fix the problem size nor the execution time
- An integrated framework to measure and compare the performances of various interconnects, useful to determine the bottlenecks amount the cluster compute nodes
- Extended the framework to measure the performance of the memory transfers between the host and GPU.

NetPIPE

- For each buffer size of c bytes, three measurements are taken, c-p, c, c+p bytes, where p is the user defined perturbation value
- Timing measurements are done using the gettimeofday() system call
- NetPIPE Throughput plot
  - x-axis, packet size in Bytes in logarithmic scale
  - y-axis, throughput in Gigabits/sec
- NetPIPE Latency plot
  - x-axis, packet size in Bytes in logarithmic scale
  - y-axix, latency in seconds in logarithmic scale


NetPIPE
MPI
mpich,lam/mpi
PVM
TCGMSG
MPI-2
One-sided
MPI_Put/MPI_Get
SHMEM
1-sided puts/gets
NetPIPE
2-sided
protocols
Native
software
layers
1-sided
protocols
Internal
Systems
memcpy
cudaMemcpy
cuMemcpy
TCP
Myrinet
Infiniband
ARMCI
LAPI
SHMEM
GPSHMEM

NetPIPE cudaMemcpy

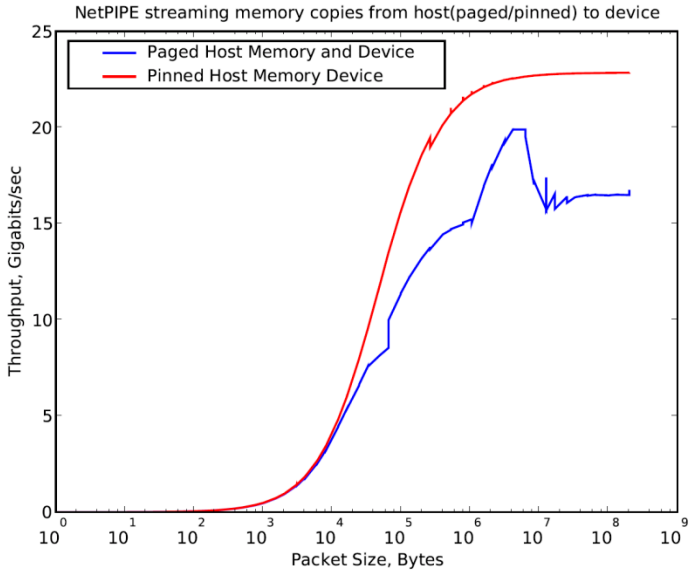
- CUDA provides two types of APIs for managing contexts, memory and data transfers
  - CUDA run-time API and the driver API
  - NetPIPE available for both APIs – NPcudaMemcpy, NPcuMemcpy
- Host memory allocation
  - paged memory – via malloc() system call
  - pinned (page-locked) memory – using the CUDA API
- Memory on the device is allocated as a linear array
- Three types of tests were performed
  - one-sided copies between host and the device
  - roundtrip copies between host and the device
  - device to device memory copies

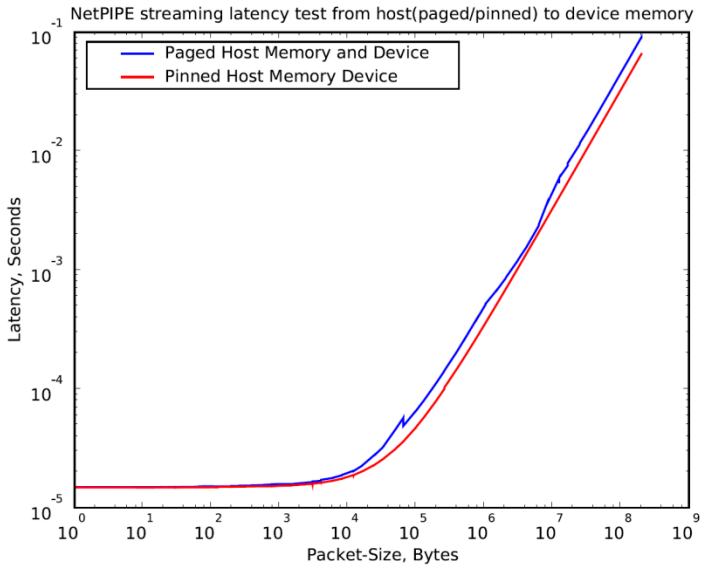
NetPIPE cudaMemcpy Algorithm

```
Input: one-sided/roundtrip
Input: host-device/device-host/device-device
Input: paged/pinned memory
Output: Latency in second, Throughput in Gigabits/sec
/* Begin */
T = MAXTIME
for i = 1 to NTRAILS do
        to = 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)
                else
                        copy buf from host (paged/pinned) to dev
        end
        t1 = time()
        /* keep the minimum value of T */
        T = min(T, t1 – t0)
end
T = T/((1+rt)*NREPEAT)
```

$NREPEAT = target/((bzs2/bsz1)*tlast)$

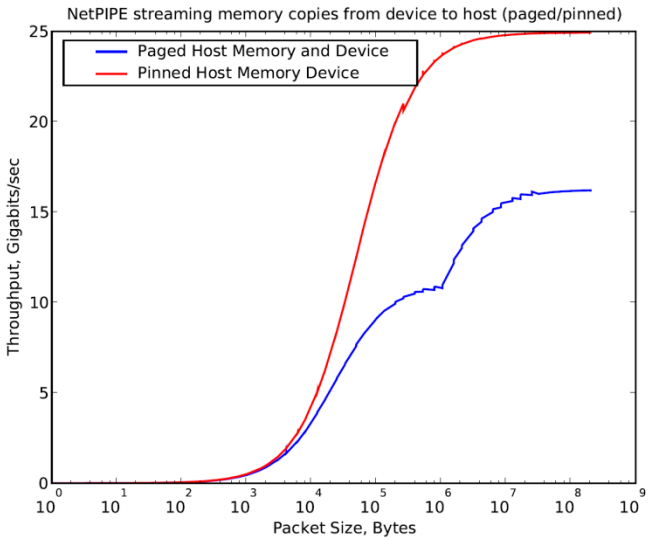
NetPIPE cudaMemcpy (Host to Device)

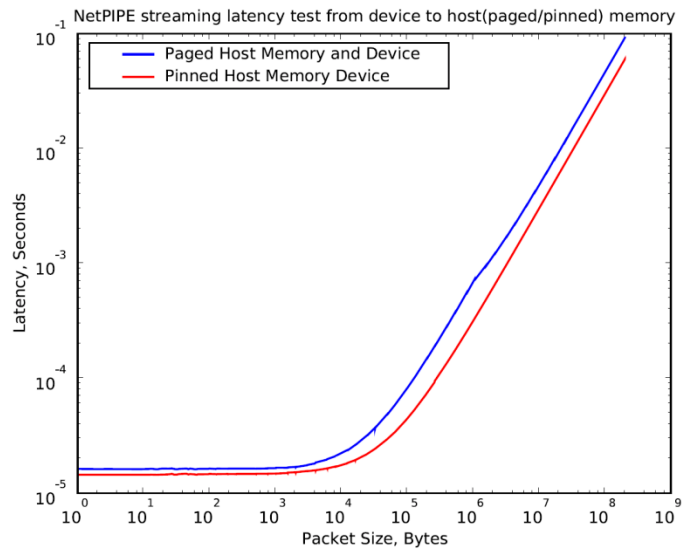




SDK Results		NetPIPE Results	
Size (Bytes)	MB/sec	Size (Bytes)	MB/sec
16855040	2012.5	16777216	2043
33632256	2075	33554432	2087
67186688	2102	67108864	2103

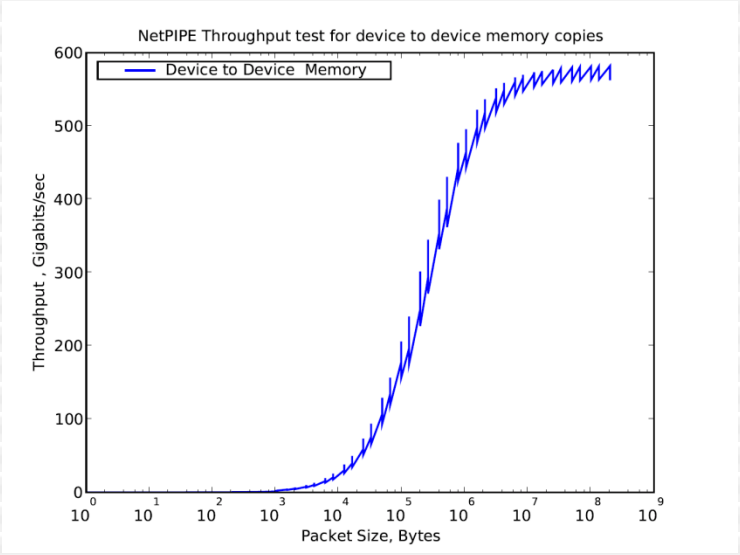
NetPIPE CudaMemcpy (Device to Host)

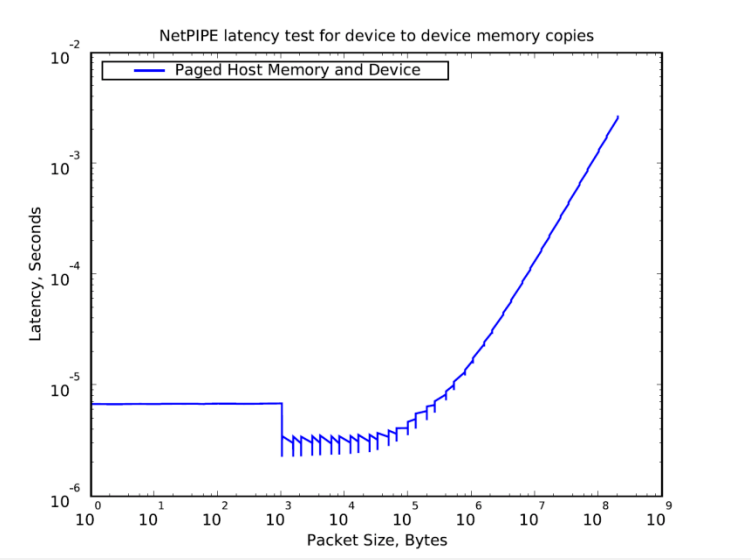




- Host to Device attains maxium around 6MB
- Device to host attains maximum around 1MB
- Latency ratio varies roughly 1 to 3 times
- Device to host throughput is larger than the host to device with pinned buffers

NetPIPE CudaMemcpy (Device to Device)





- 4GB of 512-bit GDDR3 memory
- peak device throughput of 72GB/s for 16MB of buffer sizes
- variations towards perturbations seen for smaller buffer sizes

Basic Linear Algebra Subroutines

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

alpha, beta scalars, A, B, C are matrices of dimension m x k, k x n, m x n respectively

op(X) = X or X’

- CUBLAS is the implementation of BLAS routines using the cuda driver
- Functions to create matrix and vector objects in the GPU memory space and to move data
  from host to the device
- Column major ordering similar to the FORTRAN language

- $O(N^3)$ computations
- $O(N^3)/O(N^2)$ computation to communication ratio

Intel Math Kernel Library (MKL), KMP_AFFINITY

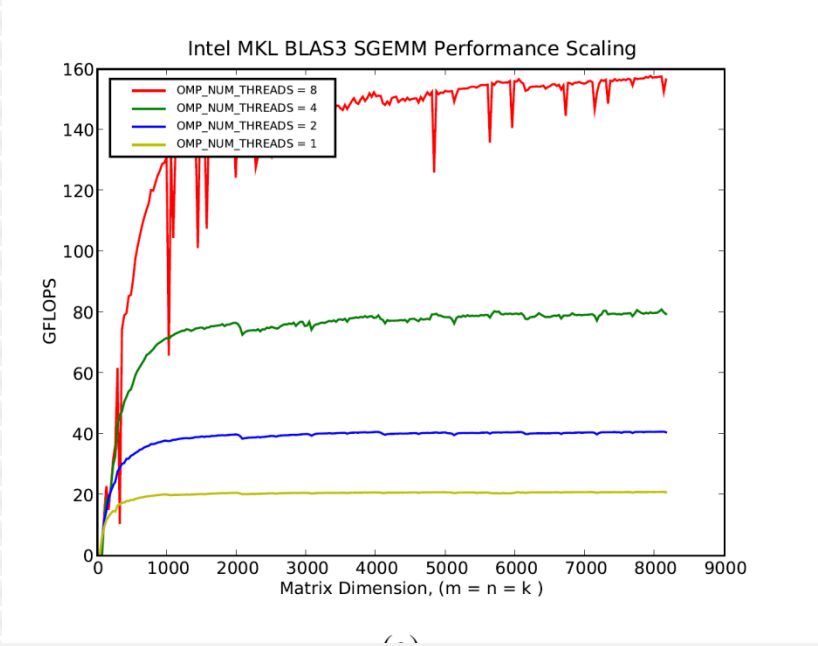
- Intel Math Kernel Library (MKL) provides highly optimized and multi-threaded BLAS routines based on OpenMP run-time library
- 10.0.1 version is used for this study
- Thread Affinity Interface – restricts execution of certain threads to a subset of physical processing units
- Performance varies if threads are not scheduled to all the physical processing units
- High-level affinity interface uses environmental variable KMP_AFFINITY

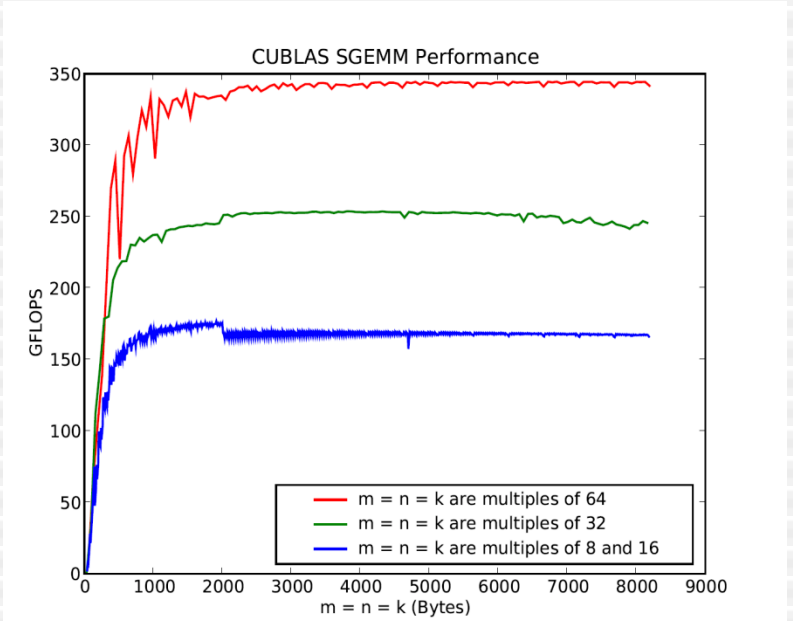
cublasSgemm / cublsasDgemm

- Tesla T10 has 30 SM's with 8 SP's per SM
- 1 MAD = 2 floating point operations
- DP Theo = 30 *1.3 GHz * 2 = 78 GFLOPS
- SP Theo = 240 * 1.3 GHz * 2 = 936 GFLOPS
- CPU DP Theo = 4 * 3 GHz * 4 cores = 48GFLOPS
- CPU SP Theo = 8 * 3 GHz * 4 cores =96GFLOPS

cublasSgemm		**cublasDgemm**	
Size(m = n = k)	**GFLOPS**	**Size (m = n = k)**	**GFLOPS**
8128	344.32	8128	69.96
8136	167.08	8136	48.46
8144	167.42	8144	62.11
8152	167.46	8152	48.46
8160	245.47	8160	62.11
8168	167.13	8168	48.46
8172	166.58	8172	62.11
8180	166.08	8180	48.44

Performance of SGEMM subroutine

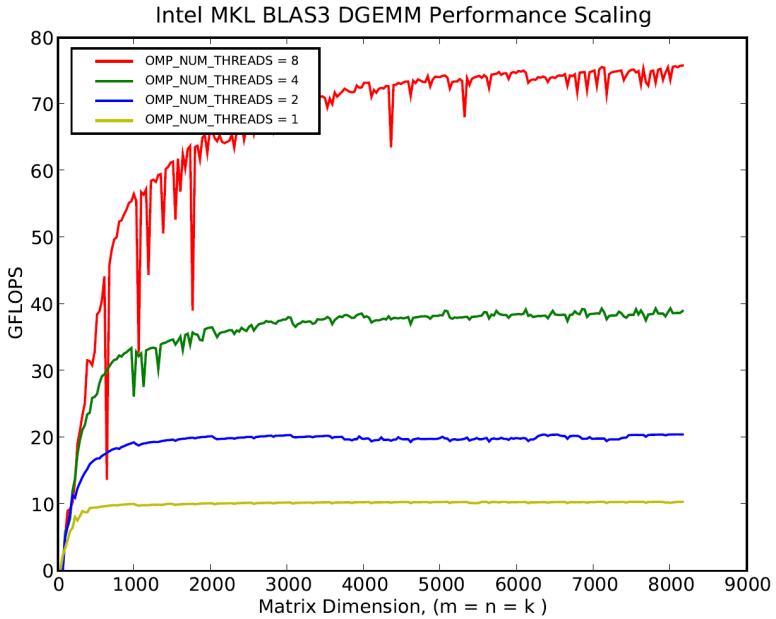


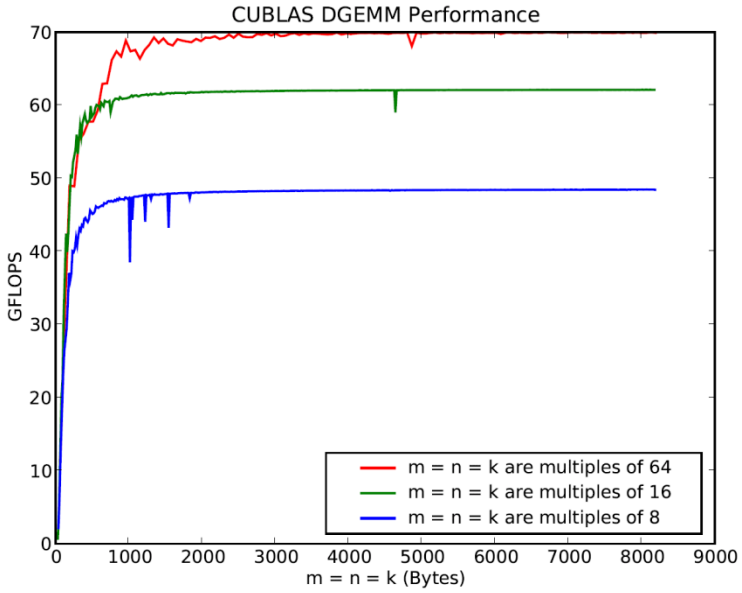


- scaled well with 8 cores, however explicit control of threads was necessary for the
- 157.8 GFLOPS maximum achieved

- Performance segmented based on the matrix size
- 345 GFLOPS

Performance of DGEMM subroutine





- scaled well with 8 cores, however explicit control of threads was necessary using the KMP_AFFINITY
- 75.82 GFLOPS maximum achieved

- Performance segmented over the input matrix size
- 70 GFLOPS maximum achieved

- Single precision performance is maximum, so it should be harnessed as much as possible.
- Techniques for mixed-precision algorithms should be developed to make a judicious choice

Conclusions

- A cudaMemcpy module is developed within the NetPIPE framework to compute the memory transfer overheads
- Results were compared against the bandwidth benchmark program provided with the cuda SDK.
- The single precision cublasSgemm is roughly 3 times faster than the MKL version
- Performance of the cublasDgemm and the MKL dgemm with 8 threads

Future Work

- Study the performance of the asynchronous memory copies
- Copies to different devices by independent threads of the CPU
- Copies to the remote GPU memory via Infiniband one-sided communications (would be useful to extend the distributed data array programming model for the GPU device memory space)
- A NetPIPE port for the OpenCL version.

Acknowledgements

Professor Mark S Gordon

Professor Theresa Windus

Andrey Asadchev, PhD Candidate

Mark Klein, Systems personnel, SCL, Ameslab

NVIDIA Corporation

Questions