Programming on GPUs and energy consumption

Sayan Ghosh, Barbara M. Chapman
sghosh4@mail.uh.edu, chapman@cs.uh.edu, Computer Science Dept., University of Houston

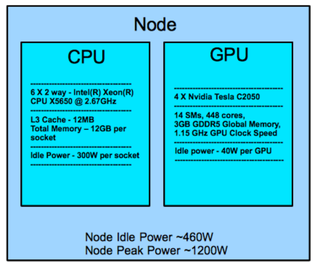
Goals and Motivations

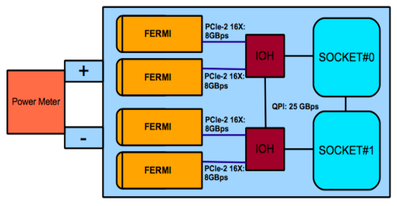
- Many current systems are using GPUs (TianHe-1A (~3Pflops, 7168 GPUs), Future ORNL Titan (20Pflops ?))
- GPUs could make computations fast, but what is energy cost at the node level?
- How is the complexity of an algorithm related to system energy cost?
- Measure performance and power on CPUs and GPUs for the kernels
- Relative CPU vs. GPU measurements
- Single problem mapped across all cores (CPU and GPU) to assist in direct comparison between CPUs and GPUs

Testbed and Measurement Environment

- Power meter Yokogawa WT500: 10Hz, node level
- Accessible via PwrLib (API, developed at PNNL) from application

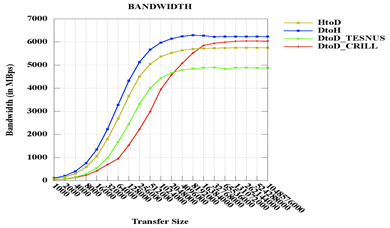
```
double pwr_in_Watts=0.0, wallclock_time_in_secs=0.0;
PNNLinit_power_meter();
PNNLstart_power_meter();
        <Code to profile>
PNNLstop_power_meter();
PNNLget_pwr_time(&pwr_in_Watts, &wallclock_time_in_secs);
```

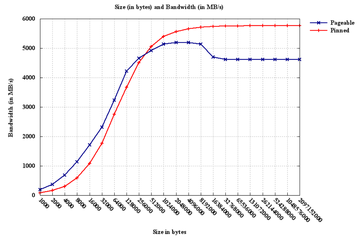


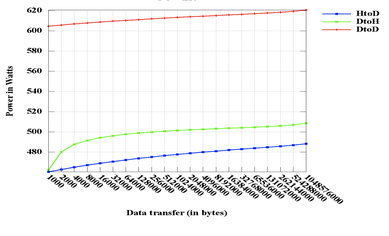


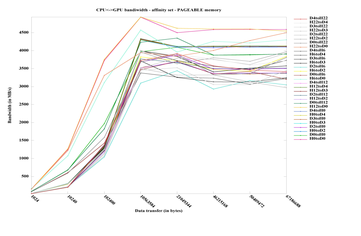
Bandwidth Test

- Data Transfer rate is a prime candidate to judge performance
- Found to have a minimal impact on overall power (PCIe 2 16X power is ~10W, plus PCIe switches have built in Power Management events)
- "cudaMemCpy" function have different code paths for varying cases
- For small data sizes, pinned memory transfer have more overhead than pageable
- GPU-GPU memory copies are slow or not possible when they are in different I/O hubs









Abstract

GPUs are slowly becoming ubiquitous devices in High Performance Computing. Nvidia's newly released version 4.0 of the CUDA API for GPU programming offers multiple ways to program on GPUs and emphasizes on multi-GPU environments which are common in modern day compute clusters. However, the cost of running the computers are getting higher with their progress in FLOP counts, which could be attributed to increased energy consumption and cooling costs. The fastest supercomputer (K-computer from Japan) consumes around 10 MW (only CPUs), versus the recent #1 position holder Tianhe-1A at NUDT, China, is around 4 MW (half the number of GPUs as CPUs). Since the energy (power X time) of a system has an obvious correlation with the user program, hence programming techniques could affect the overall system energy significantly.

This work discusses the various strategies of GPU programming, and tries to explore how the differences in these approaches could lead to an economical solution.

Minimizing *t* of (*p X t*) = energy

- Tweaks that improve performance of single threaded programs also applicable here (using & instead of %, >> 1 instead of /, et al)
- Using pinned memory to avoid an extra copy – DMA of GPUs
- Avoiding `%Synchronize%` statements if possible
- Using CUDA streams to overlap kernel execution and data transfer – program benefits only if kernel execution time is comparable to memory copy time (eg: transpose) and the number of streams (upto 16 streams are supported, but there are only two copy engines [because PCIe is bidirectional])
- GPU global memory is off-chip and around 100 times slower than faster on-chip shared memory – using shared memory for reductions, and computations on contiguous elements are preferable
- Use `__volatile__` keyword to specify data to be stored on registers (less virtual registers allocated, compiler forced to reuse data)
- If one thread in a warp is stalled, it means the rest 31 threads are waiting – conditional statements could be a performance killer
- Coaleascing global memory – the loop strides should be in multiples of half-warp to minimize transactions
- Avoiding multiple copies – copying data once to the GPU is preferred
- Operations in the same stream will be executed in order and would overlap with operations in other streams. Having separate memory for every stream could avoid costly `sync` statements

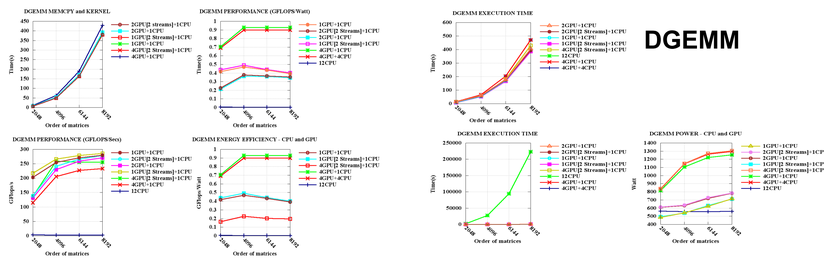
Test Kernels

- Matrix-Matrix Multiply (DGEMM)
  - Uses CUBLAS (v 2.0) on the GPUs
  - GNU Scientific Library - BLAS for CPUs
- Matrix Transpose (naïve)
- Eratosthenes Sieve Prime Number generation
  - Calculate prime numbers within N
  - Trick is to find primes from 0 to sqrt(N) [serial], then from sqrt(N) to N find if any number in that range is divisible by any primes in 0 to sqrt(N) [embarassingly parallel]
  - Low arithemtic intensity, conditional execution – good for CPUs – GPUs have a far greater branch prediction overhead

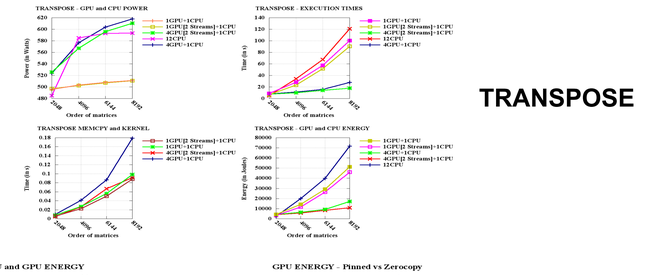
Kernels	1 GPU		2 GPUs		4 GPUs		12 CPUs
	2 streams	No streams	2 streams	No streams	2 streams	No streams	pthreads/omp
DGEMM	Y	Y	Y	Y	Y	Y	Y
Transpose	Y	Y	N	N	Y	Y	Y
Prime Sieve	N	Y	N	N	N	N	Y

Results

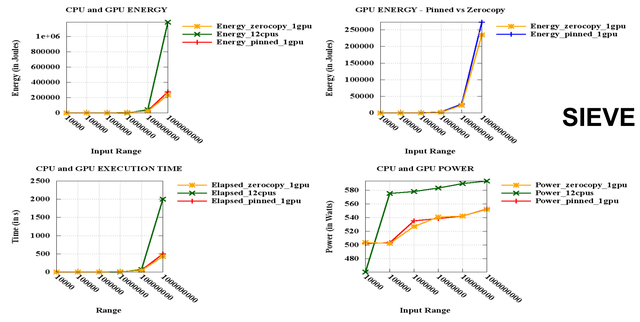
DGEMM



TRANSPOSE



SIEVE



Inferences

- Porting applications to multiple GPUs could increase node power substantially without bringing any benefit to the execution time
- For GPUs in a single node, if data transfer dominates an algorithm, then it does not adds to power significantly
- Optimization is a must to increase energy efficiency
- Algorithms which have an almost equal memory transfer and kernel execution overhead; streams could be used to overlap execution and data transfer. 10% overlap in DGEMM and upto 50% in Transpose observed.
- Bandwidth depends on the layout of GPUs on the node. Eg: GPUs across different I/O hubs cannot participate in peer to peer data copy in some cases
- GPUs have upto 7 GB of memory, moving all data at a shot could avoid overheads w.r.t PCIe transfers
- Conditional statements could cause incorrect behavior and slowness

Acknowledgements

Thanks to Dr. Darren Kerbyson, Dr. Kevin Barker and Dr. Abhinav Vishnu of PNNL for all their guidance and scrutiny. I started this work under the mentorship of them while interning at PNNL during the summer of 2011.
I also would want to thank Sunita Chandrasekaran of HPCTOOLS lab for reviewing the content.