What’s new and exciting about Graham’s GPUs

Sergey Mashchenko
SHARCNET
From Fermi to Pascal

• The Monk GPUs are very dated - they are of Fermi generation, and since then NVIDIA introduced Kepler, Maxwell, and Pascal GPU architectures.
  – Fermi: 2010
  – Kepler: 2012
  – Pascal: 2016
  (Maxwell didn't have any HPC GPUs.)

• The new cluster Graham has 320 of HPC Pascal GPUs, P100. (Cedar at Simon Fraser has 584 P100's.)
## Evolutionary changes

<table>
<thead>
<tr>
<th>Specification</th>
<th>Monk</th>
<th>Graham</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA cores</td>
<td>448</td>
<td>3584</td>
</tr>
<tr>
<td>SP flops</td>
<td>1.03 TFlops</td>
<td>9.3 TFlops</td>
</tr>
<tr>
<td>Device memory</td>
<td>5.2 GB</td>
<td>12 GB</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>148 GB/s</td>
<td>549 GB/s</td>
</tr>
</tbody>
</table>
Revolutionary changes

- **CUDA Dynamic Parallelism (CDP)**: new hard/software feature allowing for dynamic workload generation on GPU (kernels launched from kernels). Makes GPU much more general purpose computing device. First appeared in Kepler GPUs.

- **Hyper-Q**: in previous generations, multiple CPU threads could only access the GPU sequentially (one queue); Kepler / Pascal expand that to 32 parallel queues. This should significantly accelerate mixed MPI/CUDA and OpenMP/CUDA codes, without any code modifications. Also great for GPU farming.
Dynamic Parallelism

- Dynamic parallelism (DP) is available in CUDA 5.0 and later on devices of Compute Capability 3.5 or higher (sm_35 for Kepler; sm_60 for Pascal).
- Under DP, an application can launch a coarse-grained kernel which in turn launches finer-grained kernels to do work where needed.
Dynamic Parallelism

- DP is perfect for adaptive grid codes and codes with recursion.
DP: simple example

- DP allows one to move almost everything to GPU.

```cpp
// On device:
// Second level kernels (multi-threaded):
__global__ void kernel1 (){}
__global__ void kernel2 (){}

// Top level kernel (single-threaded):
__global__ void main_kernel (){
  if (threadIdx.x == 0) {
    // These second level kernels will run sequentially (would need streams for concurrency)
    kernel1<<<Nblocks, Nthreads>>>();
    kernel2<<<Nblocks, Nthreads>>>();
    ...
  }
}

// On host:
int main() {
  main_kernel<<<1,1>>>();
}
```
Amdahl's Law

- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

\[
\text{speedup} = \frac{1}{1 - P}
\]

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup). If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
Amdahl's Law (2)

- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

\[
\text{speedup} = \frac{1}{P \frac{1}{P} + S}\]

where \( P \) = parallel fraction, \( N \) = number of processors and \( S \) = serial fraction.
Amdahl's Law (3)

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at $P = 0.50$, 0.90 and 0.99 (50%, 90% and 99% of the code is parallelizable):

<table>
<thead>
<tr>
<th>N</th>
<th>$P = 0.50$</th>
<th>$P = 0.90$</th>
<th>$P = 0.99$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.82</td>
<td>5.26</td>
<td>9.17</td>
</tr>
<tr>
<td>100</td>
<td>1.98</td>
<td>9.17</td>
<td>50.25</td>
</tr>
<tr>
<td>1000</td>
<td>1.99</td>
<td>9.91</td>
<td>90.99</td>
</tr>
<tr>
<td>10000</td>
<td>1.99</td>
<td>9.91</td>
<td>99.02</td>
</tr>
</tbody>
</table>
Hyper-Q: why is it important?

- GPUs work well when you saturate them with data-parallel threads.
- Graham GPU has 8 times more cores (so need 8x more threads to get saturated) than the Monk GPU.
- From the Amdahl's law, a code which runs well on Monk will likely perform poorly* on Graham.
- Hyper-Q helps to mitigate this, by allowing to share one GPU between different CPU threads.
Live demo of Hyper-Q

• A simple code, primes_HQ, only runs one block of threads per kernel.

• This mimics a realistic code which doesn't have enough of parallelism to saturate a modern GPU.

• Important: Hyper-Q is usually not enabled by default.
Job script for GPU farming

#!/bin/bash
#SBATCH --gres=gpu:1
#SBATCH -t 0-00:30
#SBATCH --mem=4G
#SBATCH -c 16

export CUDA_MPS_LOG_DIRECTORY=$HOME/tmp
nvidia-cuda-mps-control -d

for ((i=0; i<16; i++))
  do
    ./code &>out &
  done
wait
Other new features

- Atomic operations improvements:
  - `atomicAdd` now supports FP64 (integer and float)
  - `atomicMin` and `atomicMax` now support INT64
- Half precision (FP16) at twice speed of FP32
- HBM2 memory: much higher bandwidth, hardware ECC (no memory or efficiency wasted for ECC).

- Quantitative improvements:
  - Grid length (1D): 65,535 -> 2e9
  - 32-bit registers per thread: 63 -> 255
  - Concurrent kernels per device: 16 -> 128
Binary reduction
Kernel for binary summation

__shared__ double sum[BLOCK_SIZE];
...
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim.x; // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1)
{
    int halfPoint = nTotalThreads / 2; // Number of active threads

    if (threadIdx.x < halfPoint)
    {
        int thread2 = threadIdx.x + halfPoint; // the second element index
        sum[threadIdx.x] += sum[thread2]; // Pairwise summation
    }
    __syncthreads();
    nTotalThreads = halfPoint; // Reducing the binary tree size by two}
Binary at the lower level, atomic at the higher level

```c
__shared__ float sum[BLOCK_SIZE];
// Initialize sum[] array here
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim.x; // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1){
    int halfPoint = nTotalThreads / 2; // Number of active threads
    if (threadIdx.x < halfPoint)
    {
        int thread2 = threadIdx.x + halfPoint; // the second element index
        sum[threadIdx.x] += sum[thread2]; // Pairwise summation
    }
    __syncthreads();
    nTotalThreads = halfPoint; // Reducing the binary tree size by two
}
if (threadIdx.x == 0)
    atomicAdd (&xsum, sum[0]); // Atomic reduction
```
FP64 reduction on monk

- Two-level binary reduction:

```c
// Host code
#define BSIZE 1024 // Always use a power of two; can be 32...1024
// Total number of elements to process: 1024 < Ntotal < 1024^2

int Nblocks = (Ntotal+BSIZE-1) / BSIZE;

// Low level (the results should be stored in global device memory):
x_prereduce <<<Nblocks, BSIZE >>> ();

// High level (will read the input from global device memory):
x_reduce <<<1, Nblocks >>> ();
```
Online quiz

Link:  http://www.socrative.com

Room:  CUDADAY2