GPU Computing is everywhere!

http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system

http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1
GPU in a HPC system

http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system

Host system has its DRAM on the main board

http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1
GPU in a HPC system

http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1
Why do we need a GPU????

Difference between a CPU and GPU

• Both cater to different needs → Low Latency or High Throughput?
Difference between Multicore CPU and Manycore GPU

HPC Machines in CS Labs

<table>
<thead>
<tr>
<th>Name</th>
<th>GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>bentley</td>
<td>GF119 [NVS 315] and GM204 [GeForce GTX 980]</td>
</tr>
<tr>
<td>bugatti</td>
<td>GF119 [NVS 315] and GM204 [GeForce GTX 980]</td>
</tr>
<tr>
<td>ferrari</td>
<td>GF119 [NVS 315] and GM204 [GeForce GTX 980]</td>
</tr>
<tr>
<td>jaguar</td>
<td>GF119 [NVS 315] and GM204 [GeForce GTX 980]</td>
</tr>
<tr>
<td>lamborghini</td>
<td>GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]</td>
</tr>
<tr>
<td>lotus</td>
<td>GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]</td>
</tr>
<tr>
<td>maserati</td>
<td>GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]</td>
</tr>
<tr>
<td>porsche</td>
<td>GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]</td>
</tr>
<tr>
<td>raspberries</td>
<td>G96GL [Quadro FX 580]</td>
</tr>
</tbody>
</table>
### Features of the Graphics card **GeForce GTX 980**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Architecture</td>
<td>Maxwell</td>
</tr>
<tr>
<td>GPU Name</td>
<td>GM204</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>2048</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1126 MHz</td>
</tr>
<tr>
<td>VRAM</td>
<td>4 GB GDDR5</td>
</tr>
<tr>
<td>Memory Bus</td>
<td>256-bit</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>7.0 GHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>224.0 GB/s</td>
</tr>
<tr>
<td>Power Connectors</td>
<td>Two 6-Pin</td>
</tr>
</tbody>
</table>

### Processing Flow:

[Diagram showing the processing flow between CPU, GPU Memory, and DRAM]
Processing Flow (contd.):
GPU Architecture

1. **Global Memory:**
   - Accessible by both GPU and CPU
   - Analogous to RAM in a CPU server

2. **Streaming Multiprocessors (SMs):**
   - Perform the actual computations
   - Each SM has its own:
     - Control units, registers, execution pipelines, caches
GPU Architecture — Fermi: Streaming Multiprocessor

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem+ L1 cache
- 32K 32-bit registers
- Register File

GPU Architecture — Fermi: CUDA Core Floating

**Core**
- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit
Memory System-- Architecture

Shared memory(L1)
  • User-managed scratch-pad
  • Hardware will not evict until threads overwrite
  • 16 or 48KB / SM (64KB total is split between Shared and L1)
  • Aggregate bandwidth per GPU: 1.03 TB/s

ECC Protection can be enabled.

CUDA Programming
Abstractions
Shivani Dave
CUDA Application structure:

Serial code executes in a host (CPU) thread

Parallel code executes in many device (GPU) threads across multiple processing elements
Threads are grouped into blocks

- A kernel is executed as a grid of blocks of threads

Kernel Execution

- Each thread is executed by a core CUDA
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time in an asynchronous way
Memory Model

Device Code:
- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant

Host Code:
- Transfer data to/from per-grid Global and Constant memories

<table>
<thead>
<tr>
<th>Threads</th>
<th>Registers and Local Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block of Threads</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>All the blocks</td>
<td>Global Memory</td>
</tr>
</tbody>
</table>

Memory Model (contd …)

- Global memory is the slowest memory on the GPU
- Coalescing improves memory performance; it occurs when multiple (row major order) consecutive threads (IDs) read / write consecutive data items from / to global memory
- 16 (half a warp) global array elements are accessed at once: coalescing produces vectorized reads / writes that are much faster than element wise reads / writes.
- A warp in CUDA, is a group of 32 threads, which is the minimum size of the data processed in SIMD fashion by a CUDA multiprocessor.
Sequential C code -> Vector Addition

```c
#include<stdio.h>
void Addition_Vector(int *a, int *b, int *c)
{
    int i=0;
    for(i=0;i<25;i++)
        c[i]=a[i]+b[i];
}

int main()
{
    int a[25],b[25],c[25],i;
    for(i=0;i<25;i++)
    { 
        a[i]=1;
        b[i]=i+1;
    }
    Addition_Vector(a,b,c);
    for(i=0;i<25;i++)
        printf("\n%d",c[i]);
}
```

Parallelism and Threads

- To process operations in parallel, the operations must be independent of each other i.e., no data dependencies
- A thread is mapped to a single processor which executes in parallel with the remaining threads.
- A CUDA kernel is executed by a grid (array) of threads with each thread with a unique index id in a specific block.
- A grid is organized as a 2D array of blocks (gridDim.x and gridDim.y)
- Each block is organized as 3D array of threads (blockDim.x, blockDim.y, and blockDim.z)

gridDim = (3,2)
blockDim = (2,2,1)
Thread Allocation

• A thread block can be allocated on any stream multiprocessor and thread blocks must be independent of each other, i.e., cannot communicate with each other at all.
  – pro: now the computation can run on any number of SMs
  – con: this makes programming a GPU harder
• Multiple thread blocks can be scheduled on one multiprocessor, if resources allow it. They still are independent of each other.

Thread Synchronization

• Threads inside one thread block can synchronize – _syncthreads() command
• host can synchronize kernel calls – either explicitly through cudaThreadSynchronize() – or implicitly through memcpy()-s
Parameterizing the thread code

- The programmer’s job is to write the code so that when a collection of threadblocks, each with a collection of threads executes in concert, the collective work done solves the problem.
- But there is only one piece of code (what it does is a function of the grid and thread “coordinates.”
  - GridIdx.x, GridIdx.y
  - threadIdx.x, threadIdx.y, threadIdx.z
- In our first example, there is just one block which has 256 threads in it: all but threadIdx.x are zero, and unused.
- So, we will have all values from 0 to 255
- Analogy with OpenMP/MPI:
  - Dim is line num_threads(), and MPI_Size
  - Ids is like thread_num() and MPI_rank

Multiple Blocks

Multiple block in the Grid.
Each block will have an equal number of threads as defined in the program.
In the above example: blockIdx.x keeps on increasing from 0 to 255 similar to the threadIdx.x. The variable blockDim.x will be 256 which is equal to the number of blocks.
Program Flow

CPU(Host)

Serial Code(Sequential)

Parallel Code(Call to the kernel)

Serial Code(Sequential)

GPU(Device)

Memory

Host's memory

Device's memory

Grid

Blocks 1...n

Threads 1...n

Vector Addition Map

1:1 Mapping

Iteration Space

Range of variable i
0 to 255
0...........255

Data Space

Vectors A, B and C
Function used f(i)

[ ] [ ] [ ].....255

Thread Space

i will be mapped as a
f(Threadid)

[ ] [ ] [ ].....255

There exists a 1:1 mapping between the iterator variable and the ThreadId. There exists only one block each being mapped to a set of 255 threadIds.
Parallel Code ➔ Vector Addition

```c
#include<cuda.h>
void main(){
  int n=256; int b[n]; int a[n],c[n]; //initialize the arrays too
  int *dev_b; int *dev_a, int *dev_c;
  cudaMemcpy((void**)&dev_a, &a, n*sizeof(int));
  cudaMemcpy((void**)&dev_b, &b, n*sizeof(int));
  cudaMemcpy((void**)&dev_c, &c, n*sizeof(int));
  cudaMemcpy(dev_a, &a, n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, &b, n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_c, &c, n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(&c, dev_c, n*sizeof(int), cudaMemcpyDeviceToHost);
}
```

Kernel definition

```c
__global__ void AddIntegers(int *a, int *b, int *c) {
  c[threadIdx.x]=a[threadIdx.x]+b[threadIdx.x];
}
```

__global__ lets the compiler know that it is a kernel function and accordingly appropriate actions will be taken

Call to the kernel from the host is made using the below format:
```c
<<<number of blocks, number of threads>>>```

Built-in Variables

- In the kernel a set of built-in variables specifies the grid and block dimensions (Dim) and indices (Idx).
- These can be used to determine the thread ID
  - `gridDim` contains .x and .y grid dimensions (sizes)
  - `blockIdx` contains block indices .x and .y in the grid
  - `blockDim` contains the thread block .x, .y, .z dimensions (sizes)
  - `threadIdx` contains .x, .y and .z thread block indices
- 1D thread block: ID = `threadIdx.x`
- 2D thread block: ID = `threadIdx.x + threadIdx.y*blockDim.x`
- 3D thread block: ID = `threadIdx.x + threadIdx.y*blockDim.x + threadIdx.z*blockDim.x*blockDim.y`

API functions in CUDA

- Device Memory Allocation function
- Host-Device Data/transfer
Device Memory Allocation function

cudaMalloc()
• Similar to Malloc function in C
• It will allocate memory in the GPU’s global memory.
• Two parameters
  1. Address of a pointer to the allocated object
  2. Size of allocated object in terms of bytes

cudaFree()
• Frees object from device global memory
• Parameter - Pointer to freed object

Host-Device Data /transfer

cudaMemcpy()
--Used for memory data transfer
Parameters:
1. Pointer to destination
2. Pointer to source
3. Number of bytes copied
4. Type of transfer
There are 4 types of data transfer:
- Host to Host
- Host to Device
- Device to Host
- Device to Device
All these data transfers are asynchronous.
Programming Examples

2D Grid Block

• Cuda allows us to create 1-D, 2-D, 3-D grid blocks
• 2-D Kernel Launch:
  • Image processing tasks typically impose a regular 2D raster (an image) over the problem domain. Computational Fluid dynamics tasks might be most naturally expressed by partitioning a volume over a 3D grid.
2-D Representation Program

• 2-D mapping

```c
__global__ void kernel(int *array)
{
  int index_x = blockIdx.x * blockDim.x + threadIdx.x;
  int index_y = blockIdx.y * blockDim.y + threadIdx.y;

  // map the two 2D indices to a single linear, 1D index
  int grid_width = gridDim.x * blockDim.x;
  int index = index_x * grid_width + index_y;

  // map the two 2D block indices to a single linear, 1D block index
  int result = blockDim.x * gridDim.x * blockIdx.y;

  // write out the result
  array[index] = result;
}
```

Vector Addition Map

### 2:1 Mapping

- **Iteration Space**
  - Range of variable i
  - Range of variable j
  - 0...255
  - 0...255

- **Data Space**
  - Vectors A(2-D array), C
  - Function used f(i,j)
  - □□□□□□□□□□

- **Thread Space**
  - j and j will be mapped as f(BlockIdThreadId, BlockId)
  - ▲▲▲▲▲▲▲▲▲▲▲▲...16
  - ▲▲▲▲▲▲...16

There exists a 2:1 mapping between the iterator variable i, j and the ThreadId for every BlockId.
There exists 16 blocks each being mapped to a set of 2 values in x and y directions.
Matrix Multiplication

- Consider two Matrices M and N. We need to store the result in P
- Executing on the host:
  - `void MatrixMulOnHost(float* M, float* N, float* P, int Width)`
  - 
    ```
    for (int i = 0; i < Width; ++i) 
      for (int j = 0; j < Width; ++j) {
        double sum = 0;
        for (int k = 0; k < Width; ++k) {
          double a = M[i * Width + k];
          double b = N[k * Width + j];
          sum += a * b;
        }
        P[i * Width + j] = sum;
      }
    ```

Cuda Code:

```c
void MatrixMultiplication(float* M, float* N, float* P, int Width) {
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;

  // Transfer M and N to device memory
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
  cudaMemcpy(VOID, &Md, size, cudaMemcpyHostToDevice);
  cudaMemcpy(VOID, &Nd, size, cudaMemcpyHostToDevice);

  // Allocate P on the device
  cudaMemcpy(VOID, &Pd, size, cudaMemcpyHostToDevice);

  // Kernel invocation code - to be shown later
  ...

  // Transfer P from device to host
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
  freeDeviceKeables();
}
```
Kernel Code

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Md, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // P value stores the P element that is computed by the thread
  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
  {
    float Pdelement = Md[ty * Width + k];
    float Pdelement = Md[k * Width + tx];
    Pvalue = Md * Pdelement;
  }

  // Write the matrix to device memory each thread writes one element
  Md[ty * Width + tx] = Pvalue;
}
```

Programming Example-Jacobi1D.cu
References:

• http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf
• https://class.coursera.org
• Programming Massively Parallel Processors - David B. Kirk and Wen-mei W. Hwu
  • http://docs.nvidia.com/cuda/parallel-thread-execution/
    #axzz3p0 rtlxV
  • https://code.google.com/p/stanford-cs193g-sp2010/wiki/
    TutorialMultidimensionalKernelLaunch