Review

• “Secret” behind GPU performance: simple cores but a large number of them; even more threads can exist live on the hardware (10k–20k threads live).

• Important performance bottleneck: data traffic between memory and register files

• Programming on a GPU:
  • Warp = 32 threads
  • Block = group of threads running on one SM
  • SM runs an integer number of thread blocks
  • Grid = set of blocks representing the entire data set
Code overview

```c
int* d_output;
cudaMalloc(&d_output, sizeof(int) * N);
kernelfor<1, N>>>(d_output);
vector<int> h_output(N);
cudaMemcpy(&h_output[0], d_output, sizeof(int) * N,
            cudaMemcpyDeviceToHost);
for(int i = 0; i < N; ++i) {
    printf("Entry %3d, written by thread %2d\n",
            h_output[i], i);
}
cudaFree(d_output);
```
Function to execute

```
kernel<<<1, N>>>(d_output);
```

Input variables

Number of threads to use
Device kernels

```c
__device__ __host__
int f(int i) {
    return i*i;
}

__global__
void kernel(int* out) {
    out[threadIdx.x] = f(threadIdx.x);
}
```
**global** host device

What are these mysterious keywords?

**__global__** kernel will be
• Executed on the device
• Callable from the host

**__host__** kernel will be
• Executed on the host
• Callable from the host

**__device__** kernel will be
• Executed on the device
• Callable from the device only
./firstProgram 1
./firstProgram 32
./firstProgram 1024
./firstProgram 1025

• Fails at 1025!
• Blocks and grid required for “real” runs.
You can use `__global__` and `__host__` together.

Yes, the function gets compiled...

No, these options are incompatible.

Start the presentation to activate live content.
Thread hierarchy

Two levels of hierarchy:

• Thread block: a block is loaded entirely on an SM. This means that a block of threads cannot require more resources than available, for example to store register variables.

• Grid of blocks: for large problems, we can use multiple blocks. A very large number of blocks can be allocated so that we can execute a kernel over a very large data set.
Problem decomposition

Decompose your problem into:

• Small blocks so that data can fit in SM.
  • Typically the number of threads per block is around 256–512.

• Create a grid of blocks so that you can process the entire data.

• Hardware limitations for GPUs on Certainty:

  Maximum threads per block: 1024
  Maximum dimension 0 of block: 1024
  Maximum dimension 1 of block: 1024
  Maximum dimension 2 of block: 64
  Maximum dimension 0 of grid: 65535
  Maximum dimension 1 of grid: 65535
  Maximum dimension 2 of grid: 65535
Block execution

The device will start by loading data on each SM to execute the blocks.

- At least one block must fit on an SM, e.g., there should be enough memory for register variables.
- The SM will load up as many blocks as possible until it runs out of resources.
- Then it will start executing the blocks, that is all threads will execute the kernel.
- Once all threads in a block are done with the kernel, another block gets loaded in the SM.
- And so on until the device runs out of blocks to execute.
Dimensions

Blocks and grids can be 1D, 2D or 3D. Their dimensions is declared using:

\[
\text{dim3 threadsPerBlock}(N_x); \\
\text{dim3 numBlocks}(M_x);
\]

\[
\text{dim3 threadsPerBlock}(N_x, N_y); \\
\text{dim3 numBlocks}(M_x, M_y);
\]

\[
\text{dim3 threadsPerBlock}(N_x, N_y, N_z); \\
\text{dim3 numBlocks}(M_x, M_y, M_z);
\]

Thread index example:
\[
\text{int col = blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}; \\
\text{int row = blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y};
\]
__global__
void Add(int n, int* a, int* b, int* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if(i < n && j < n) {
        c[n*i + j] = a[n*i + j] + b[n*i + j];
    }
}

/* Dimension along y = n_thread; dimension along x and z = 1 */
dim3 threads_per_block(1, n_thread);

int blocks_per_grid_x = n;
int blocks_per_grid_y = (n + n_thread - 1) / n_thread;
dim3 num_blocks(blocks_per_grid_x, blocks_per_grid_y);

/* Run calculation on GPU */
Initialize<<<num_blocks, threads_per_block>>>(n, d_a, d_b);
Add<<<num_blocks, threads_per_block>>>(n, d_a, d_b, d_c);
I'd use a grid of blocks if I have a simple line `x[i]` to compute with?

- Yes
- No

It depends.

Start the presentation to activate live content.

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app.
Yes, but it’s not recommended

No, it won’t happen

Start the presentation to activate live content
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
Compiler options and other tricks

- `-g`  debug on the host
- `-G`  debug on the device (see documentation, CUDA-gdb, Nsight Eclipse Edition)
- `-pg` profiling info for use with gprof
- `-xcompiler` (options for underlying gcc compiler)

More on profiling: `nvprof`
Profiling

- The visual profiler (nvvp) is the simplest way to profile your code. You get a graphical interface that helps navigate the information.
- You can also use a command line tool: nvprof.
- This can also be used to collect information on a cluster to be visualized using nvvp on your local computer.

- Run
  nvprof ./addMatrices -n 4000
NVPROF is profiling process 9229, command: ./addMatrices -n 4000

Dimensions of matrix: 4000 x 4000
Dimension of thread block along y: 512
Dimension of grid: 4000 x 8

All tests have passed; calculation is correct.
Profiling application: ./addMatrices -n 4000

Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>82.99%</td>
<td>17.713ms</td>
<td>1</td>
<td>17.713ms</td>
<td>17.713ms</td>
<td>17.713ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>9.09%</td>
<td>1.9394ms</td>
<td>1</td>
<td>1.9394ms</td>
<td>1.9394ms</td>
<td>1.9394ms</td>
<td>Add(int, int*, int*, int*)</td>
</tr>
<tr>
<td>7.92%</td>
<td>1.6907ms</td>
<td>1</td>
<td>1.6907ms</td>
<td>1.6907ms</td>
<td>1.6907ms</td>
<td>Initialize(int, int*, int*)</td>
</tr>
</tbody>
</table>

API calls:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>80.16%</td>
<td>104.98ms</td>
<td>3</td>
<td>34.993ms</td>
<td>281.33us</td>
<td>104.37ms</td>
<td>cudaMalloc</td>
</tr>
<tr>
<td>13.77%</td>
<td>18.035ms</td>
<td>1</td>
<td>18.035ms</td>
<td>18.035ms</td>
<td>18.035ms</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>2.79%</td>
<td>3.6503ms</td>
<td>2</td>
<td>1.8251ms</td>
<td>1.7031ms</td>
<td>1.9472ms</td>
<td>cudaMemcpySynchronize</td>
</tr>
<tr>
<td>2.67%</td>
<td>3.4989ms</td>
<td>332</td>
<td>10.538us</td>
<td>157ns</td>
<td>395.13us</td>
<td>cudaMemcpyGetAttribute</td>
</tr>
<tr>
<td>0.31%</td>
<td>403.71us</td>
<td>4</td>
<td>100.93us</td>
<td>100.75us</td>
<td>101.32us</td>
<td>cudaMemcpyGetTotalMem</td>
</tr>
<tr>
<td>0.25%</td>
<td>331.28us</td>
<td>4</td>
<td>82.819us</td>
<td>79.806us</td>
<td>88.013us</td>
<td>cudaMemcpyGetName</td>
</tr>
<tr>
<td>0.03%</td>
<td>40.453us</td>
<td>2</td>
<td>20.226us</td>
<td>12.109us</td>
<td>28.344us</td>
<td>cudaMemcpyLaunch</td>
</tr>
<tr>
<td>0.00%</td>
<td>5.8130us</td>
<td>7</td>
<td>830ns</td>
<td>196ns</td>
<td>4.1910us</td>
<td>cudaMemcpySetupArgument</td>
</tr>
<tr>
<td>0.00%</td>
<td>3.1480us</td>
<td>2</td>
<td>1.5740us</td>
<td>329ns</td>
<td>2.8190us</td>
<td>cudaMemcpyGetLastError</td>
</tr>
<tr>
<td>0.00%</td>
<td>2.4130us</td>
<td>2</td>
<td>1.2060us</td>
<td>274ns</td>
<td>2.1390us</td>
<td>cudaMemcpyGetCount</td>
</tr>
<tr>
<td>0.00%</td>
<td>2.2930us</td>
<td>2</td>
<td>1.1460us</td>
<td>361ns</td>
<td>1.9320us</td>
<td>cudaMemcpyConfigureCall</td>
</tr>
<tr>
<td>0.00%</td>
<td>1.6810us</td>
<td>8</td>
<td>210ns</td>
<td>165ns</td>
<td>297ns</td>
<td>cudaMemcpyGet</td>
</tr>
</tbody>
</table>
cuda-memcheck

• Several tools available to verify your code.
• 4 tools accessible with *cuda-memcheck* command:
  memcheck: Memory access checking
  initcheck: Global memory initialization checking
  racecheck: Shared memory hazard checking
  synccheck: Synchronization checking;
    __syncthreads(), not on SM 2.x
• Usage example:
cuda-memcheck  ./memcheck_demo
cuda-memcheck --leak-check full  ./memcheck_demo
GPU Optimization!
Top factors to consider

• Memory: data movement!
• Occupancy and concurrency:
  “hide latency through concurrency”
• Control flow: branching
Memory

• The number one bottleneck in scientific applications are not flops!
• Flops are plentiful and abundant.
• The problem is data starvation.
• Computing units are idle because they are waiting on data coming from / going to memory.
Memory access

• Similar to a regular processor but with some important differences.
• Caches are used to optimize memory accesses: L1 and L2 caches.
• Cache behavior is complicated and depends on the compute capability of the card (that is the type of GPU you are using).
• Generations (compute capability): 2.x, 3.x, 5.x, and 6.x (no 4.x).
• For this class, we have access to 2.0 hardware.
• We will focus on this for our discussion.
Cache

• There are two main levels of cache: L1 and L2.
• L2 is shared across SMs.
• Each L1 is assigned to only one SM.
• When reading data from global memory, the device can only read entire cache lines.
• There is no such thing as reading a single float from memory.
• To read a single float, you read the entire cache line and “throw out” the data you don’t need.
Cache lines

- **L1**: 128-byte lines = 32 floats.
- **L2**: 32-byte segments = 8 floats.

- Commit these fundamental numbers to memory now!
How is data read from memory then?

• Remember how the hardware works.
• All threads in a warp execute the same instruction at the same time.
• Take 32 threads in a warp.
• Each thread is going to request a memory location.
• Hardware groups these requests into a number of cache line requests.
• Then data is read from memory.
• At the end of the day, what matters is not how much data threads are requesting from memory, it’s how many memory transactions (cache lines) are performed!
Example

```c
__global__ void offsetCopy(float* odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

offset = 0

This is the best case.
All memory accesses are coalesced into a single memory transaction.
One memory transaction

Aligned accesses (sequential/non-sequential)

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>2.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
</tr>
<tr>
<td></td>
<td>1x 32B at 128</td>
</tr>
<tr>
<td></td>
<td>1x 32B at 160</td>
</tr>
<tr>
<td></td>
<td>1x 32B at 192</td>
</tr>
<tr>
<td></td>
<td>1x 32B at 224</td>
</tr>
</tbody>
</table>
__global__ void offsetCopy(float* odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}

offset = 1
Strided access

__global__ void stridedCopy(float* odata, float* idata, int stride) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[stride*xid];
}

- Stride is 2.
- Effective bandwidth of application reduced by 2.
Efficiency of reads

- Perfectly coalesced access: one memory transaction.
  - Bandwidth = Peak Bandwidth
- Misaligned: two memory transactions.
  - Bandwidth = Peak Bandwidth / 2
- Strided access
  - Bandwidth = Peak Bandwidth / \( s \)
    for a stride of \( s \)
- Random access
  - Bandwidth = Peak Bandwidth / 32

```c
__global__ void randomCopy(float* odata, float* idata, int* addr) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[addr[xid]];
}
```
Data type size

• The previous discussion covered the most common case where threads read a 4-byte float or int.
• char: 1 byte. float: 4 bytes. double: 8 bytes.
• CUDA has many built-in data types that are shorter/longer:
  - char1-4, uchar1-4
  - short1-4, ushort1-4
  - int1-4, uint1-4
  - long1-4, ulong1-4
  - longlong1-2, ulonglong1-2
  - float1-4
  - double1-2
• Example: char1, int2, long4, double2
size of a single memory transaction is 128 by

- Only if the memory access is coalesced
  - No, this is only true when there is no offset in the access
- Yes, if the type is float, but not double
  - Yes, when the bandwidth is close to peak
How to calculate the warp ID

- How do you know if 2 threads are in the same warp?
- Answer: look at the thread ID. Divide by 32: you get the warp ID.
- Thread ID is computed from the 1D, 2D or 3D thread index using:

  \[ x + yD_x + zD_xD_y \]

```c
int tID = threadIdx.x
          + threadIdx.y * blockDim.x
          + threadIdx.z * blockDim.x * blockDim.y;
int warpID = tID/32;
```
Matrix transpose

• Let’s put all these concepts into play through a specific example: a matrix transpose.
• It’s all about bandwidth!
• Even for such a simple calculation, there are many optimizations.
template<typename T>
__global__
void simpleTranspose(T* array_in, T* array_out, int n_rows, int n_cols) {
    const int tid = threadIdx.x + blockDim.x * blockIdx.x;

    int col = tid % n_cols;
    int row = tid / n_cols;

    if(col < n_cols && row < n_rows) {
        array_out[col * n_rows + row] = array_in[row * n_cols + col];
    }
}
Memory access pattern

Coalesced reads

Strided writes