Introduction to CUDA/GPU

Dan Mazur, Pier-Luc St-Onge
guillimin@calculquebec.ca
November 26, 2015
Partners and sponsors
Outline

- Login and Setup
- Overview of GPGPU Computing
- OpenACC directives
- Introduction to CUDA-C
  - Simple CUDA programs
  - Global memory
  - Blocks and threads
  - Syntax for common tasks
  - Error Checking
  - Shared Memory and Thread Cooperation
- Big data case study: K-means clustering
Login and Setup

• Your class number is indicated on the password slip given to you by the instructors
  – Username: class###, e.g. class01
    
    ssh class##@guillimin.hpc.mcgill.ca
  – enter password

• (Optional) Start a screen session and note lg-1r**-n**::
  
  screen -S $USER
  screen -d -R $USER  # In case connection dropped

• Create an interactive job on a K20 node:
  
  qsub -I -l nodes=1:ppn=1:gpus=1 -l walltime=7:00:00
  # Wait until your job starts on a K20 node
  module add CUDA_Toolkit/7.5
Recall: Pointers in C

What is the output?

```c
int x = 1;
int *ip;
ip = &x;
*ip = 0;
printf("x=%d\n",x);
```

- A) x=1
- B) x=0
- C) x=0x7fff9575c05f
- D) Error or segmentation fault
- E) Not sure
What is the output?

```
int x = 1;
int *ip;
ip = &x;
*ip = 0;
printf("x=%d\n",x);
```

- A) x=1
- B) x=0
- C) x=0x7fff9575c05f
- D) Error or segmentation fault
- E) Not sure
What is a GPU?
What is a GPU?

• A device for handling computationally expensive hot spots in your code (accelerator, co-processor)

• Large number of low-powered, but low cost (computational overhead, power, monetary cost) processors

• Incredible computing speeds (teraflops) through massive parallelism (1000s of parallel threads or more)

• Heterogeneous computing: CPU and GPU work together on the problem
GPUs Under the Hood

ALU = Arithmetic and Logic Unit - The workhorse
Nvidia GPUs on Guillimin

- **Nodes**
  - 50 with Dual Intel Sandy Bridge EP E5-2670 (8-core, 2.6 GHz, 20MB Cache, 115W)
    - 25 with 64 GB RAM + 25 with 128 GB RAM
  - 8 with Dual Intel Ivy Bridge EP E5-2650v2 (8-core, 2.6 GHz, 20MB Cache, 115W)
    - 8 with 64 GB RAM

- **Cards**
  - 2 x nVidia Tesla Kepler K20 cards per node
  - Peak SP FP: 3.52 TFLOPs, Peak DP FP: 1.17 TFLOPs
  - 5GB memory
What was the ASCI Red?

- 1997, first teraflop supercomputer, same compute power as single K20 GPU
- 4,510 nodes (9298 processors), total 1,212 GB of RAM, 12.5 TB of disk storage
- 850 kW vs. 144 W for Nvidia K20
Notes:

- Chart denotes theoretical maximum values. Actual performance is application dependent.
- The K20 GPU has 13 streaming multiprocessors (SMXs) with 2496 CUDA cores, not directly comparable to x86 cores.
- The K20 GPU and Xeon Phi have GDDR5 memory, the Sandy Bridge has DDR3 memory.
Matrix multiplication results
SE10P is a Xeon Phi Coprocessor with slightly higher specifications than the 5110P
**Embarrassingly parallel financial monte-carlo**

Iterative financial monte-carlo with regression across all paths

“Tesla GPU” is a K20X, which has slightly higher specifications than the K20

Source: xcelerit blog, Sept. 4, 2013
(http://blog.xcelerit.com/intel-xeon-phi-vs-nvidia-tesla-gpu/)
How can accelerators help you do science?

- Two ways of thinking about speedup from parallelism:
  - 1: Compute a fixed-size problem faster
    - Ahmdal's law describes diminishing returns from adding more processors
  - 2: Choose larger problems in the time you have
    - Gustafson's law: Problem size can often scale linearly with number of processors
GPUs in Theoretical Astrophysics
Worldline Numerics

• What is the non-local Casimir interaction between magnetic flux tubes?

• The energy is given by a path integral over all possible paths a virtual electron-positron pair can take through the flux tubes

• Approximate functional integrals with an average over an ensemble of representative worldlines
GPU Worldline Numerics

• Monte Carlo => error bars shrink with sqrt(N)
  - Want to compute thousands of paths or more
• Virtual pairs can interact with many flux tubes
  - Each path is a numerically intensive integral
• GPUs allow thousands of paths to be computed in parallel
Other Applications of GPUs

• Hundreds of scientific applications have GPU accelerated versions

• http://www.nvidia.ca/object/gpu-applications.html

• Chemistry, biology, physics, math, machine learning, weather and climate, CFD, finance, ...

• ABAQUS, Amber, GROMACS, LAMMPS, MATLAB, Theano
Ways to use GPUs

- Accelerated application (AMBER, GROMACS, LAMMPS, ABAQUS, MATLAB)
- Libraries (CUBLAS, CUFFT, Thrust)
- Programming directives (OpenACC)
- GPU programming (CUDA-C, CUDA Fortran, pyCUDA, OpenCL)

Increasing effort
Focus For Today

- No - GPU Accelerated Applications and Libraries
  - Incredibly useful for research, easy to use
  - Wont teach you much about GPUs

- Yes - Explicit GPU Programming
  - Will teach you about GPUs
  - Some depth in one language will help you to understand libraries, applications, directives, and other GPU languages
  - We will focus mainly on CUDA-C
GPU Programming Standards

- CUDA
  - Nvidia proprietary standard
  - Dependant on Nvidia hardware and software
  - Mature toolkit (debugging, profiling, etc.)

- OpenCL
  - Open Standard
  - Similar programming model to CUDA

- OpenMP 4.0 accelerator offloading
  - Open Standard
  - Higher-level, Pragma based

- OpenACC
  - Open Standard
  - Higher-level, Pragma based
Workshop Files

• Please copy the workshop files to your home directory

```bash
cp -R /software/workshop/gpu/* ~/
```

• Contains:
  - Code for the exercises
  - An example submission script
  - Solutions to the exercises
  - Some misc gpu codes to explore
Scheduling GPU Jobs

- Workshop jobs run on a single CPU core + single GPU device
- Submission to Guillimin should specify gpus=1 or gpus=2 (per node)

```bash
qsub -I -l nodes=1:ppn=1:gpus=1 \  
   -l walltime=00:10:00

qsub ./subScript.sh
```

- Example: `subTrivial.sh`
Exercise 1: OpenACC

- OpenACC lets you easily offload segments of code to the GPU
- Uses pragmas, similar to OpenMP
- Our PGI OpenACC license is available through the `pgi` module (`module add pgi/14.9`)
  - `gcc` is also working on OpenACC support (early stages of development)
Exercise 1: OpenACC

• Directives: The second easiest way to accelerate your code next to libraries

• Compile and run the C program matrix_mul.c:

```bash
module add pgi/14.9
pgcc -fast -o matrix_mul matrix_mul.c
./matrix_mul
```

Multiplies two 4000x4000 float matrices using the CPU only. Takes about 20 seconds on our Sandy Bridge processors.
Exercise 1: OpenACC

- We can more than double the speed of this code with a single line and a special compiler

```
#pragma acc kernels copyin(a,b) copy(c)
```

- Use OpenACC
- Execute on the GPU
- Copy matrices a and b to device
- Copy matrix c to device at beginning and from device at end
Exercise 1: OpenACC

- When the serial calculation has finished:
  - Do: `cp matrix_mul.c matrix_mul_acc.c`
  - Insert pragma immediately before the matrix multiplication for loops in `matrix_mul_acc.c`

  ```
  #pragma acc kernels copyin(a,b) copy(c)
  ```

  ```
  pgcc -acc -fast -Minfo \ 
  -o matrix_mul_acc matrix_mul_acc.c
  ./matrix_mul_acc
  ```
OpenACC

- My result: 20.62s in serial, 2.77s OpenACC (2 K20s)
- Speedup of 7.4X
- What happened?
  - Compiler looked at our code and automatically parallelized it on the gpu device using advice from the pragma
  - Similar to OpenMP
- Speedup should increase with matrix size until device memory size becomes an issue
OpenACC

• OpenACC is very versatile and has many more features.

• Getting started guide:

/software/CentOS-6/compilers/pgi149/linux86-64/14.9/doc/openAcc_gs.pdf
First CUDA Code: trivial.cu

```c
#include <stdio.h>

__global__ void foo()
{
}

int main()
{
foo<<<1,1>>>();
printf("CUDA error: %s\n", cudaGetErrorString(cudaGetLastError()));
return 0;
}
```

- **Kernel (this one does nothing)**
- **Kernel call (compare to function call in C)**
  ```c
  <<<1,1>>> indicates 1 block of 1 thread(s)
  ```
- **CUDA built-in functions**
  (checks for errors, converts to string)
- **The rest is regular C code**
Exercise 2: CUDA pre-requisites

#include <stdio.h>

__global__ void foo()
{
}

int main()
{
    foo<<<1,1>>>();
    printf("CUDA error: %s\n", cudaGetErrorString(cudaGetLastError()));
    return 0;
}

Compile and run trivial.cu:

nvcc trivial.cu -o trivial

./trivial
trivial.out: CUDA error: no error
NVCC Warning

• **nvcc** warning: The 'compute_10' and 'sm_10' architectures are deprecated, and may be removed in a future release.

• This warning is harmless. By default, **nvcc** compiles for older CUDA architectures.

• Add the `-arch=compute_35` option to **nvcc** to compile specifically for the K20 architecture (compute capability 3.5)
cudaSetDevice(dev);

cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, dev);

sprintf(msg, " Total amount of global memory: \\
%.0f MBytes \n", \\
(float)deviceProp.totalGlobalMem / 1048576.0f);
Exercise 3: CUDA Samples

• Copy the CUDA samples directory to your home directory (warning: ~225MB)

    cp -R /software/CentOS-6/tools/cuda-7.5/NVIDIA_CUDA-7.5_Samples CUDA_Samples

• Contains:
  - Example programs (simple, utilities, graphics, imaging, finance, simulations, advanced, CUDA libraries)
  - Documentation
  - Tools
Exercise 3: CUDA Samples

● Run the deviceQuery program

```bash
cd CUDA_Samples/1_Utilsities/deviceQuery
make
./deviceQuery
cd -
```
Simple CUDA Program

1. Serial CPU code
2. Allocate Memory on Device
3. Copy data from host to device
4. Perform parallel computations on data
5. Copy results from device to host
6. Free device memory
Simple CUDA Kernel

Perform parallel computations on data

“Single Instruction, Multiple Data” (SIMD) or “Single Instruction, Multiple Thread” (SIMT) model
Simple CUDA Program

Allocate Memory on Device

Copy data from host to device

Perform parallel computations on data

Copy results from device to host

Free device memory

cudaMalloc()

cudaMemcpy()

MyKernel<<<nB, nT>>>()

cudaMemcpy()

cudaFree()
CUDA-C Reference Sheet

- Allocating/freeing memory
- Copying Data
- Kernels
- Error Checking
- Timing
- Unified Memory (CUDA 6, CC >= 3.0)
What is the output?

```c
#include <stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data_h, *data_d;
    cudaMalloc( (void**)&data_d, sizeof(int));
    data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice);
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h, data_d, sizeof(int), cudaMemcpyDeviceToHost);
    printf("data: %d\n", *data_h);
    free(data_h); cudaFree(data_d);
    return 0;
}
```

Please see addnumbers.cu (but don't compile or run!)
What is the output?

```c
#include<stdio.h>

__global__ void add2(int *a) {
    *a = *a + 2;
}

int main( void ) {
    int *data_h, *data_d;
    cudaMemcpy( (void**)&data_d, sizeof(int));
    data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice );
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h,  data_d, sizeof(int), cudaMemcpyDeviceToHost );
    printf("data: %d\n", *data_h);
    free(data_h); cudaFree(data_d);
    return 0;
}
```

- A) data: 5
- B) data: 7
- C) Error or Segmentation fault
- D) Will not compile
- E) None of the above
What is the output?

```c
#include<stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main(void)
{
    int *data_h, *data_d;
    cudaMalloc((void**)&data_d, sizeof(int));
    data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    cudaMemcpy(data_d, data_h, sizeof(int), cudaMemcpyHostToDevice);
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h, data_d, sizeof(int), cudaMemcpyDeviceToHost);
    printf("data: %d\n", *data_h);
    free(data_h);
    cudaFree(data_d);
    return 0;
}
```

- A) data: 5
- B) data: 7
- C) Error or Segmentation fault
- D) Will not compile
- E) None of the above
What is the output?

```c
#include<stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data_h, *data_d;
    cudaMalloc( (void**)&data_d, sizeof(int));
data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    //cudaMemcpy( data_d, data_h, sizeof(int),
    //    cudaMemcpyHostToDevice );
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h,  data_d, sizeof(int),
    cudaMemcpyDeviceToHost );
    printf("data: %d\n", *data_h);
    free(data_h); cudaFree(data_d);
    return 0;
}
```

- A) data: 5
- B) data: 7
- C) Error or Segmentation fault
- D) Will not compile
- E) None of the above
#include<stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data_h, *data_d;
    cudaMalloc( (void**)&data_d, sizeof(int));
    data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    //cudaMemcpy( data_d, data_h, sizeof(int),
    //            cudaMemcpyHostToDevice );
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h,  data_d, sizeof(int),
                cudaMemcpyDeviceToHost );

    printf("data: %d\n", *data_h);
    free(data_h); cudaFree(data_d);
    return 0;
}
What is the output?

```c
#include<stdio.h>
__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data_h, *data_d;
    // cudaMalloc( (void**)&data_d, sizeof(int));
    data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice );
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h,  data_d, sizeof(int), cudaMemcpyDeviceToHost);
    printf("data: %d\n", *data_d);
    free(data_h);
    cudaFree(data_d);
    return 0;
}
```

- A) data: 5
- B) data: 7
- C) Error or Segmentation fault
- D) Will not compile
- E) None of the above
#include<stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data_h, *data_d;
    cudaMalloc( (void**)&data_d, sizeof(int));
    data_h = (int *)malloc(sizeof(int));

    *data_h = 5;
    cudaMemcpy( data_d, data_h, sizeof(int), cudaMemcpyHostToDevice );
    add2<<<1,1>>>(data_d);
    cudaMemcpy(data_h, data_d, sizeof(int), cudaMemcpyDeviceToHost );
    printf("data: %d\n", *data_d);
    free(data_h); cudaFree(data_d);
    return 0;
}

- A) data: 5
- B) data: 7
- C) Error or Segmentation fault
- D) Will not compile
- E) None of the above

Explanation: Segfault...data_d points to device memory, which is not accessible to host code (code without a __device__ or __global__ prefix).
Important Points about Memory

- Device memory is different than host memory
  - Allocated with `cudaMalloc()`
  - Freed with `cudaFree()`
  - Device memory not accessible to host code
  - Host memory not accessible to device code
- Data is copied to and from the device using `cudaMemcpy()`
- Use _d and _h suffixes on variable names
Unified Memory

- Unified memory is new in CUDA 6
  - CUDA runtime manages memory transfers
  - Easier to use complex data structures
  - Allows pass by value and pass by reference
  - Note: only supported by compute capability >= 3.0 on 64-bit Linux or Windows

```c
int *data;
cudaMallocManaged(&data, sizeof(int));
*data = 3;
printf("data = %d\n", *data);
myKernel<<<1,1>>>(data);
cudaDeviceSynchronize();
printf("data = %d\n", *data);
cudaFree(data);
```
Unified Memory

- Here is the unified memory version of addnumbers.cu
  - See: addnumbers_unified.cu
- One data pointer instead of two
- No explicit cudaMemcpy calls

```
#include<stdio.h>

__global__ void add2(int *a)
{
    *a = *a + 2;
}

int main( void )
{
    int *data;
    cudaMallocManaged(&data, sizeof(int));
    *data = 5;
    add2<<<1,1>>>(data);
    cudaDeviceSynchronize();
    printf("data: %d\n", *data);
    cudaFree(data);
    return 0;
}
```
Using more than one thread

- Programmer defines the number of threads per block and the number of blocks.
- Kernel is aware of the thread ID within the block, the block ID, and the block size.

```
MyKernel<<<3, 6>>>()
```
Which expression(s) give(s) a unique index number (idx) for each thread?

block ID 0 1 2
thread ID 0 1 2 3 4 5 0 1 2 3 4 5

block size = 6

A) \( \text{idx} = (\text{thread ID}) + (\text{block ID}) \)
B) \( \text{idx} = (\text{thread ID}) \times (\text{block ID}) \)
C) \( \text{idx} = (\text{thread ID}) \times (\text{block size}) + (\text{block ID}) \)
D) \( \text{idx} = (\text{thread ID}) + (\text{block size}) \times (\text{block ID}) \)
E) C and D
Which expression(s) give(s) a unique index number (idx) for each thread?

- A) \( \text{idx} = (\text{thread ID}) + (\text{block ID}) \)
- B) \( \text{idx} = (\text{thread ID}) \times (\text{block ID}) \)
- C) \( \text{idx} = (\text{thread ID}) \times (\text{block size}) + (\text{block ID}) \)
- D) \( \text{idx} = (\text{thread ID}) + (\text{block size}) \times (\text{block ID}) \)
- E) C and D
**Block and Thread Syntax**

<table>
<thead>
<tr>
<th>Thread ID within a block</th>
<th>threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block ID</td>
<td>blockIdx.x</td>
</tr>
<tr>
<td>Block size</td>
<td>blockDim.x</td>
</tr>
</tbody>
</table>

\[
\text{idx} = (\text{thread ID}) + (\text{block size})\times(\text{block ID}) \\
\text{idx} = \text{threadIdx}.x + \text{blockDim}.x\times\text{blockIdx}.x
\]

**Q:** Why the \( .x \)?

**A:** CUDA allows for (up to) 3-dimensional threadIdx's and blockIdx's: threadIdx.[x, y, z], blockIdx.[x, y, z]
Exercise 4: CUDA Syntax

- Fill in the missing CUDA commands in the file matrixmul.cu
- Refer to your reference sheet for examples
- Choose a difficulty level depending on your experience level with C:
  - matrixmul.cu
  - matrixmul_med.cu
  - matrixmul_adv.cu

(Optional) See CUDA fortran implementation, matrix_mul.cuf

```
pgfortran -fast -o matrix_mul matrix_mul.cuf
```

- The .cuf extension or the -Mcuda compiler option indicate CUDA Fortran
Exercise 5: Write a kernel

• In the program integers.c, host code fills in an array with consecutive integers

• Modify this program so that the array is filled in parallel in a CUDA kernel function

• Don't forget:
  – The CUDA version should have extension .cu
    
    ```
    cp integers.c integers.cu
    ```
  – You may either use unified memory, or create separate host and device pointers
  – Copy results from the device to the host

• integers_adv.c: no hint in the source code!
Exercise 6: Error Checking

- Compile and run `errorcheck.cu`. Notice that no errors are reported to the terminal window.
- Actually, there is more than one error in the program.
- Please use the error checking information on your CUDA reference sheet to add error checking to the program.
  - Advanced: Minimize the amount of added code and make the resulting code look clean
- Run the program and observe the CUDA errors present.
  - Solve any errors you discover and check that the output is 'data = 7' with no CUDA errors reported.
  - Discover the errors though CUDAs error reporting instead of through code inspection.
Exercise 7: Dot Product

- In small groups, plan a kernel function for computing a vector dot product (10 minutes, max)

\[
C = a_0 b_0 + a_1 b_1 + a_2 b_2 + \ldots + a_{N-1} b_{N-1}
\]

- Pseudocode is fine

- Assume vector length is a power of two

```c
__global__ void dot(<nBlocks, nThreads>>(a_d, b_d, c_d);

__global__ void dot( float *a, float *b, float *c)
{
    ...
}
```
One possible solution:

```c
__global__ void dot( float *a, float *b, float *c)
{
    if(threadIdx.x + blockIdx.x*blockDim.x == 0)
    {
        for(int i = 0; i < N; i++)
        {
            *c += a[i]*b[i];
        }
    }
}
```

A 'correct' solution, but makes NO use of parallelism!
What problems do we encounter?

- How can we make good use of the threads and parallelism?
- How should we store intermediate values?
- How can we perform the sum?
Dot Product Strategy

- Compute products in parallel
- Store the products
- Compute the sum using “parallel reduction”

Note: parallel reduction requires some cooperation between threads
Which memory type is the best choice for the cache?

- Want to store an array of $a_i \times b_i$ that we can add through parallel reduction using many threads

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Bandwidth</th>
<th>Accessible to...</th>
<th>Notes</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Global Memory</td>
<td>Slow, high latency</td>
<td>All threads</td>
<td>same as cudaMalloc(), ~4.8GB</td>
<td><strong>device</strong> float data[N];</td>
</tr>
<tr>
<td>B</td>
<td>Constant Memory</td>
<td>Slow, cached</td>
<td>All Threads</td>
<td>read-only</td>
<td><strong>constant</strong> float data[N];</td>
</tr>
<tr>
<td>C</td>
<td>Shared Memory</td>
<td>150x faster than global</td>
<td>Threads in same block</td>
<td>Lifetime of block, ~50KB/block</td>
<td><strong>shared</strong> float data[N];</td>
</tr>
<tr>
<td>D</td>
<td>Register Memory</td>
<td>Even faster, no latency</td>
<td>Local thread only</td>
<td>very limited resource, lifetime of thread</td>
<td>float data[N];</td>
</tr>
</tbody>
</table>
Which memory type is the best choice for the cache?

- Want to store an array of $a_i \times b_i$ that we can add through parallel reduction using many threads

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Bandwidth</th>
<th>Accessible to...</th>
<th>Notes</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Slow, high latency</td>
<td>All threads</td>
<td>same as cudaMalloc(), ~4.8GB</td>
<td><strong>device</strong> float data[N];</td>
</tr>
<tr>
<td>A</td>
<td>Global Memory</td>
<td>Slow, high latency</td>
<td>All threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Constant Memory</td>
<td>Slow, cached</td>
<td>All Threads</td>
<td>read-only</td>
<td><strong>constant</strong> float data[N];</td>
</tr>
<tr>
<td>C</td>
<td>Shared Memory</td>
<td>150x faster than global</td>
<td>Threads in same block</td>
<td>Lifetime of block, ~50KB/block</td>
<td><strong>shared</strong> float data[N];</td>
</tr>
<tr>
<td>D</td>
<td>Register Memory</td>
<td>Even faster, no latency</td>
<td>Local thread only</td>
<td>very limited resource, lifetime of thread</td>
<td>float data[N];</td>
</tr>
</tbody>
</table>
Let's use Shared Memory

Pros:

• 150x faster than global memory
• Allows cooperation between threads (necessary for parallel reduction)

Cons:

• Only ~50KB/block
• Only allows cooperation within a block (partial parallel reduction)
• Lifetime of the block
Thread Race
Thread Race
Thread Race
Thread Race
Thread Race

Thread 2 thinks we are done, but thread 1 still hasn't cached it's product!

Our algorithm has 'race conditions'
__syncthreads()
\_syncthreads()
__syncthreads()
__syncthreads()
__syncthreads()
__syncthreads()
Where should we syncthreads?

```c
__global__ void dot( float *a, float *b, float *c )
{
    declarations (including cache)
    Compute products 1
    Store products into cache 2
    while (parallel reduction){
        if (first half of cache){
            add values from second half of cache 3
        } //end if
    } //end while
} //end kernel
```

`__syncthreads()` ensures that no threads in a block advance until they all reach the barrier

- A) 1, 4
- B) 2, 3
- C) 2, 4
- D) 2, 3, 4
- E) 1, 2, 3
Where should we syncthreads?

```c
__global__ void dot( float *a, float *b, float *c )
{
    declarations (including cache)
    Compute products
    Store products into cache
    while (parallel reduction){
        if (first half of cache){
            add values from second half of cache
        } //end if
    } //end while
} //end kernel
```

__syncthreads() ensures that no threads in a block advance until they all reach the barrier

- A) 1, 4
- B) 2, 3
- C) 2, 4
- D) 2, 3, 4
- E) 1, 2, 3
Which is the correct way to store products in the cache?

```c
__global__ void dot( float *a, float *b, float *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    __shared__ float cache[threadsPerGrid];
    cache[tid] = a[tid] * b[tid];
    c[blockIdx.x] = parallelReduceSum(cache);
}
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Code</th>
</tr>
</thead>
</table>
| A      | ```
    __shared__ float cache[threadsPerGrid];
    cache[tid] = a[tid] * b[tid];
``` |
| B      | ```
    __shared__ float cache[threadsPerBlock];
    cache[threadIdx.x] = a[tid] * b[tid];
``` |
| C      | ```
    __shared__ float cache[threadsPerBlock];
    cache[blockIdx.x] = a[tid] * b[tid];
``` |
| D      | ```
    __shared__ float cache[blocksPerGrid];
    cache[threadIdx.x] = a[tid] * b[tid];
``` |
| E      | ```
    __shared__ float cache[blocksPerGrid];
    cache[blockIdx.x] = a[tid] * b[tid];
``` |
Which is the correct way to store products in the cache?

```c
__global__ void dot(float *a, float *b, float *c) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    __shared__ float cache[threadsPerGrid];
    cache[tid] = a[tid] * b[tid];
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| A | __shared__ float cache[threadsPerGrid];
    |   cache[tid] = a[tid] * b[tid]; |
| B | __shared__ float cache[threadsPerBlock];
    |   cache[threadIdx.x] = a[tid] * b[tid]; |
| C | __shared__ float cache[threadsPerBlock];
    |   cache[blockIdx.x] = a[tid] * b[tid]; |
| D | __shared__ float cache[blocksPerGrid];
    |   cache[threadIdx.x] = a[tid] * b[tid]; |
| E | __shared__ float cache[blocksPerGrid];
    |   cache[blockIdx.x] = a[tid] * b[tid]; |

```c
c[blockIdx.x] = parallelReduceSum(cache);
```
Shared Memory Example

- You will implement the dot product kernel shortly
- First, consider an example: reversing an array
- At first, we will assume there is only one block to avoid algebra problems
Shared Memory Example

Here is the kernel using global memory with no shared memory cache:

```c
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int idx = threadIdx.x;

    int outidx = (blockDim.x - 1 - idx);
    d_out[outidx] = d_in[idx];
}
```
Shared Memory Example

```c
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    __shared__ int s_data[SIZE];

    int idx = threadIdx.x;

    // Load one element per thread and store it
    // *in reversed order* into temporary shared memory
    int outidx = (blockDim.x - 1 - idx);
    s_data[outidx] = d_in[idx];

    // Block until all threads in the block have
    // written their data to shared mem
    __syncthreads();

    // write the data from shared memory in forward order,
    // but to the reversed block offset as before
    d_out[idx] = s_data[idx];
}
```
Shared Memory Example

- This example illustrates using shared memory in one block
- Normally, there are many blocks

Reverse within blocks using shared mem.
and
Reverse the blocks using global mem.
Shared Memory Example

```c
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    __shared__ int s_data[SIZE];

    int inOffset = blockDim.x * blockIdx.x;
    int idx = inOffset + threadIdx.x;

    // Load one element per thread and store it
    // *in reversed order* into temporary shared memory
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[idx];

    // Block until all threads in the block have
    // written their data to shared mem
    __syncthreads();

    // write the data from shared memory to global,
    // reversing the order of the blocks
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int out = outOffset + threadIdx.x;
    d_out[out] = s_data[threadIdx.x];
}
```
DotProduct Pseudocode

```c
__global__ void dotproduct(float *a, float *b, float *c, int N) {
    // Declare shared memory cache

    // Compute global thread index, idx
    // Cache a[idx] * b[idx] into shared memory

    // Synchronize threads

    for (int offset = block_size / 2 ; offset > 0 ; offset /= 2) {
        if (thread_id < offset) {
            cache[thread_id] += cache[offset + thread_id];
        }

        // Synchronize threads
    }

    if (thread_id == 0) {
        // Store reduction result in c[block_id]
    }
}
```
Exercise 8: Implement dot product kernel

• The program `dotproduct.cu` is missing code from the kernel function
  – Please complete the kernel

• Don't forget:
  – Use shared memory for the cache (may start with global and add shared after)
  – Use `__syncthreads()` where necessary
  – Use parallel reduction to sum all products associated with the current block
Choosing number of threads per block

• The resources used per kernel and the limitations of your device determine the concurrency: number of threads that can run simultaneously
  – Threads per block
  – Registers per thread
  – Shared memory per block

• Use the `--ptxas-options=-v nvcc` option to get a report

• Nvidia provides a spreadsheet for planning your kernels to achieve high occupancy
  – CUDA GPU Occupancy Calculator
CUDA Occupancy Calculator

```bash
$ nvcc -o dotprod dotprod_soln.cu --ptxas-options=-v
ptxas info : 0 bytes gmem
ptxas info : Compiling entry function '_Z3dotPfS_S_' for 'sm_20'
ptxas info : Function properties for _Z3dotPfS_S_
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 11 registers, 1024 bytes smem, 56 bytes cmem[0]
```

CUDA GPU Occupancy Calculator

### Just follow steps 1, 2, and 3 below! (or click here for help)

1. Select Compute Capability (click): 3.5
2. Select Shared Memory Size Config (bytes): 49152

### 2. Enter your resource usage:
- Registers Per Thread: 256
- Shared Memory Per Block (bytes): 1024

### 3. GPU Occupancy Data is displayed here and in the graphs:
- Active Threads per Multiprocessor: 2048
- Active Warps per Multiprocessor: 64
- Active Thread Blocks per Multiprocessor: 8
- Occupancy of each Multiprocessor: 100%

### Impact of Varying Block Size

- Multiprocessor, Warp Occupancy
- Impact: 48% for Block Size 64
- Threads Per Block: 0 to 1024
Identifying GPU Algorithms

• SIMD Parallelizability
  – Number of concurrent threads (need 1000s)
  – Minimize conditionals and divergences

• Operations performed per datum transferred to device (FLOPs/GB)
  – Data transfer is overhead
  – Keep data on device and reuse it
Which algorithm gives the most GPU performance boost?

Put the following in order from least work per datum to most:

- i) matrix-vector multiplication
- ii) matrix-matrix multiplication
- iii) matrix trace (sum of diagonal elements)

A) i, ii, iii
B) iii, i, ii
C) iii, ii, i
D) i, iii, ii
E) They are all about the same
Which algorithm gives the most GPU performance boost?

Put the following in order from least work per datum to most:

- i) matrix-vector multiplication
- ii) matrix-matrix multiplication
- iii) matrix trace (sum of diagonal elements)

- A) i, ii, iii
- B) iii, i, ii
- C) iii, ii, i
- D) i, iii, ii
- E) They are all about the same
Big data case study: K-means clustering

- GPUs have limited memory resources
  - 5GB global memory
  - 64kB constant memory
  - 48kB shared memory / block

- For data size > global memory
  - Can still achieve performance boost with GPUs
  - Good performance requires advanced techniques to achieve GPU computation + CPU computation + PCIe memory transfers to occur simultaneously
  - This discussion is to show what is possible, not a learning-goal of this workshop

- Case study based on Ren Wu, Bin Zhang, Meichun Hsu, “GPU-Accelerated Large Scale Analytics”, 2009, Hewlett-Packard Laboratories
  - K-means clustering
K-means clustering

- Unsupervised machine learning
- Divide a data set into k different categories based on the features of that data set
- E.g. Clothing manufacturer: based on customer's height and weight data, divide them into 3 or more size categories
K-means clustering

Step 1: Randomly generate K locations (circles)

Step 2: Group data points by proximity to locations

Step 3: Update locations to the centroid of each group

Iterate over steps 2 and 3

Images: I, Weston.pace
“Small data” GPU implementation

- Computational hotspot: computing distances between each cluster centroid and each data point, $O(n*k)$
- Parallelization: Each thread loops over cluster centroids and computes the distances for a single data point
- Memory management:
  - Copy transposed data set to device (Must be transposed to get coalesced reads for memory bandwidth)
  - Copy cluster centroids to constant memory
  - Compute distances and assign clusters in parallel
  - Copy cluster assignments back to host
Big data challenges

• What if data does not fit into global memory?
  – Can the problem be solved in smaller pieces?
  – If so, we must transfer data for each piece over PCIe.
    • How to manage all of the data transfers?
  – When and how to transpose?
CUDA Streams and Data Partitioning

- Suppose we need to process the data in separate chunks because of memory limitations.

- The GPU has a mechanism for task-based parallelism (in addition to the data parallelism we have worked with today).
  - CUDA Streams: A queue of events to be executed in order on a device.

- By using two or more CUDA streams, we can perform the memory copies in parallel with kernel executions.

- Can transpose our chunks directly on the device using a kernel.
# CUDA Streams

## Stream 0
- Copy chunk of data to device (memcpyAsync)
- Transpose chunk of data on device (kernel)
- Compute distances, assign clusters (kernel)
- Copy cluster assignments from device (memcpyAsync)

## Stream 1
- Copy chunk of data to device (memcpyAsync)
- Transpose chunk of data on device (kernel)
- Compute distances, assign clusters (kernel)
- Copy cluster assignments from device (memcpyAsync)
K-means GPU speedup

- Dual quad-core Intel Xeon 5345 2.33 GHz CPUs
- Nvidia GeForce GTX 280 GPU (1GB memory)
- The paper reports for big data problems
  - 200x-400x faster than single-core CPU
  - 20x-40x faster than 8-core CPU version
We learned how to:

- Use OpenACC pragmas to easily accelerate loops
- Allocate and free device memory
- Copy data between host and device
- Implement programs with SIMD parallelism
- Identify and fix errors based on CUDA error messages
- Use shared memory for cooperation between threads
- Compile CUDA programs using nvcc
- Identify whether an algorithm is a good candidate for ‘easy’ performance gains on a GPU
- Recognize some advanced techniques for big data processing
Keep Learning...

- **Documentation:**
  - [http://docs.nvidia.com](http://docs.nvidia.com)

- **Tutorials:**
  - [https://nvidia.qwiklab.com/](https://nvidia.qwiklab.com/) (interactive iPython notebooks on AWS)

- **Examples:**

- **Courses:**
  - [https://www.coursera.org/course/hetero](https://www.coursera.org/course/hetero)

- **Questions:**
  - [http://stackoverflow.com](http://stackoverflow.com)
  - [https://forums.geforce.com/](https://forums.geforce.com/)
  - guillimin@calculquebec.ca
What Questions Do You Have?

guillimin@calculquebec.ca