Intro to GPU Programming
CS 540 – High Performance Computing

The CPU

The "Central Processing Unit"
Traditionally, applications use CPU for primary calculations

- General-purpose capabilities
- Established technology
- Usually equipped with 8 or less powerful cores
- Optimal for concurrent processes but not large scale parallel computations

Wikimedia commons: **Intel_CPU_i7_640_Prescott_bottom.jpg**

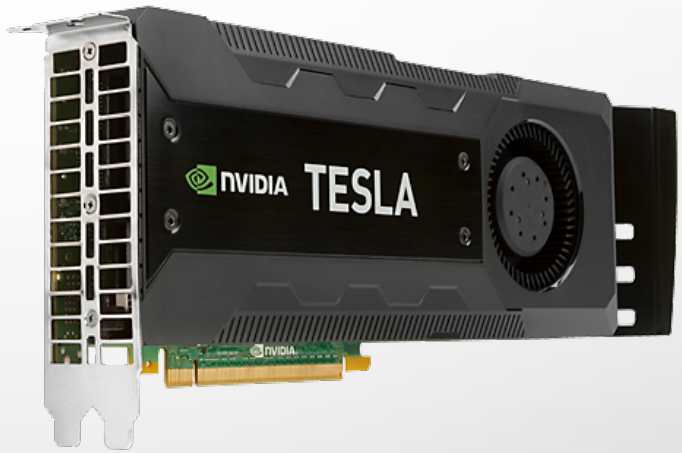
The CPU cont.

- Primary focus is latency
  - amount of time it takes to perform a single instruction
- Clock speed is measured in GHz
- Extremely efficient at performing complex operations
- Very efficient at handling single streams of data
- Relatively expensive to build
- Complex architecture

The GPU

The "Graphics Processing Unit"
Relatively new technology designed for parallelizable problems

- Initially created specifically for graphics
- Became more capable of general computations



Wikimedia commons: **Nvidia_Tesla_K40.jpg**

The GPU cont.

- Lower clock speed compared to CPU's
  - Clock speeds range from high MHz to Low GHz
- Primary focus of a GPU is throughput
  - Accomplish more work in a specific amount of time
- Hardware architecture is less complex
  - Less expensive to manufacture
  - Cheap to build
- Extremely efficient at performing simple instructions
- Very good at handling multiple streams of data
- Thousands of GPU's cores per silicon chip
  - Spawn thousands of threads

- How do we leverage the hardware?

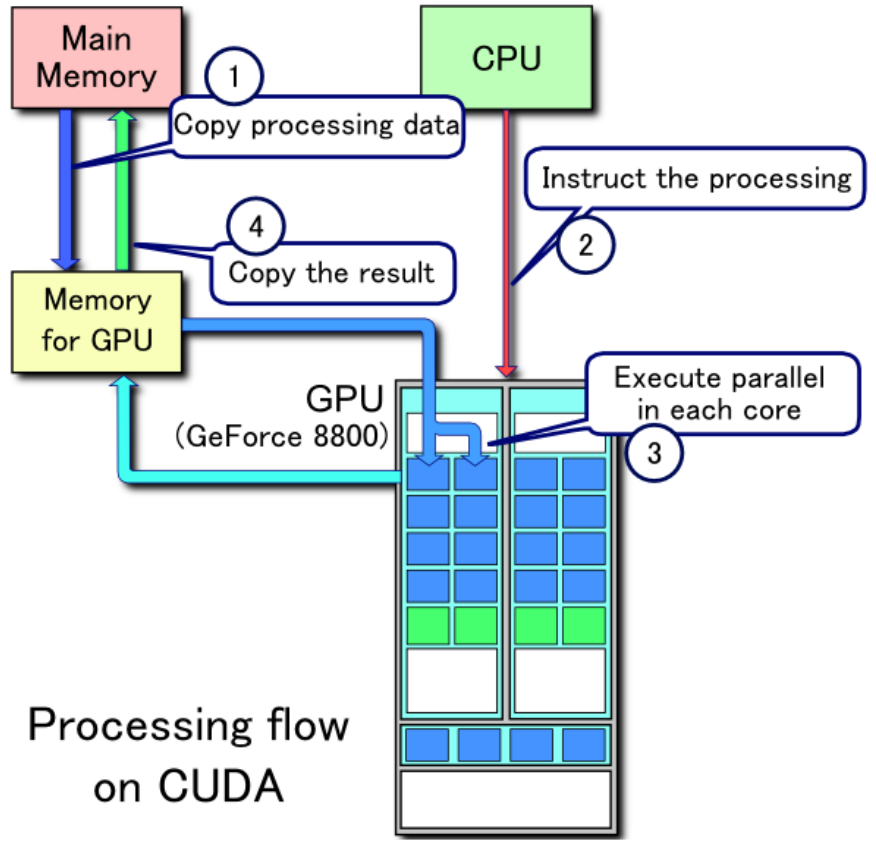
CUDA

- Parallel computing platform developed by NVIDIA
- Compute Unified Device Architecture
- Allows developers to create applications for use in a parallel computing environment.
- C/C++ derived
  - Syntax similar if not identical to C/C++
- Many other languages support CUDA extensions
  - Python, Fortran, Java, Ruby, etc.

CUDA cont.

- CUDA enabled GPU's can support 1024 threads/block
  - block - represents a group of threads that can be executing serially or in parallel
- CUDA Process Flow
  - Copy data from host (CPU) main memory (RAM) to device (GPU) main memory (on-board RAM)
  - CPU invokes a CUDA kernel
    - CPU is the only component that can invoke a kernel
    - A kernel is a CUDA function
- GPU processes data
  - Thousands of cores/threads in parallel
  - Copy results back to host when complete
- Copy process can be asynchronous

CUDA Process Flow Diagram



Wikipedia: **cuda_process_flow.jpg**

CUDA Example

```
__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

CUDA Example 2

```
#include "../common/book.h”
__device__ int addem( int a, int b ) {
    return a + b;
}

__global__ void add( int a, int b, int *c ) {
    *c = addem( a, b );
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

Another CUDA Example

Add two arrays

- A[] + B[] -> C[]

On the CPU:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
C[i] = A[i] + B[i];
```

- Operates sequentially, but we can do better.

A simple problem

- On the CPU (multi-threaded, pseudocode):

```
(allocate memory for CPU)
Create # of threads equal to number of cores on processor (2-8)
(Indicate portions of A, B, C to each thread...)

In each thread,
For (i from beginning region of thread)
        C[i] <- A[i] + B[i]
//lots of waiting involved for memory reads, writes, ...
Wait for threads to synchronize...
```

- Slightly faster – 2-8x (slightly more with other tricks)

A simple problem cont.

- How many threads? How does performance scale?

- Context switching:
  - process of storing and restoring the state of a process/thread so that execution can be resumed from the same point at a later time.
    - High penalty on the CPU
    - Low penalty on the GPU

A simple problem on the GPU.

- On the GPU:

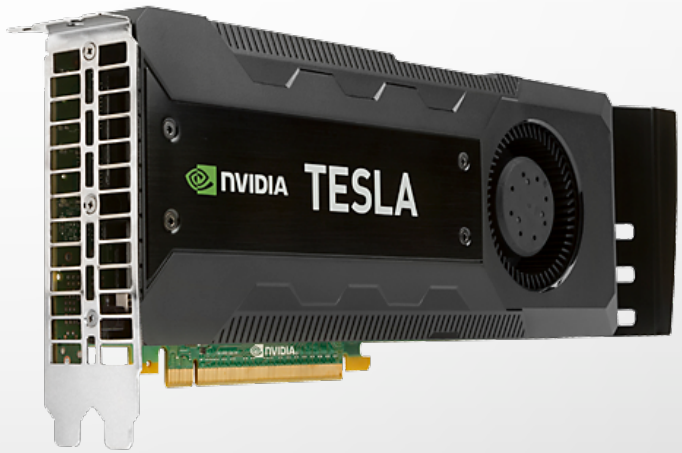
```
(allocate memory for A, B, C on GPU)
Create the “kernel” – each thread will perform one (or a few)
additions
        Specify the following kernel operation:

        For (all i‘s assigned to this thread)
        C[i] <- A[i] + B[i]

Start ~20000 (!) threads
Wait for threads to synchronize...
```

A simple problem on the GPU.

- Parallelism / lots of cores
- Low context switch penalty!
  - We can "mask" performance loss by creating more threads!
- Why the low context switch penalty?



Wikimedia commons: **Nvidia_Tesla_K40.jpg**

CUDA Kernel

- Our "parallel" function
- Simple implementation

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    //Decide an index somehow
    c[index] = a[index] + b[index];
}
```

- Indexing

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

Invoking the Kernel

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Invoking the Kernel cont.

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

The CUDA Compiler, nvcc

- CUDA source files end with the .cu extension
- nvcc is a complete C++ compiler
  - Almost any valid C++ program can be compiled with nvcc
  - No C++ 11
- Works just like the standard GNU gcc/g++ compiler
  - nvcc cuda_hello.cu –o helloworld

References

- CUDA by Example – Jason Sanders et.al.
- http://courses.cms.caltech.edu/
- https://en.wikipedia.org/wiki/Thread_block
- https://developer.nvidia.com/cuda-education
- https://en.wikipedia.org/wiki/Context_switch
- CME 214 Introduction to parallel computing using MPI, openMP and CUDA – Eric Darve, Stanford University