Parallel Programming
Lecture 6
Intro to CUDA

Erich Elsen

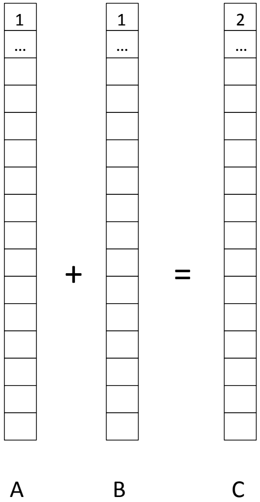
Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2013

1. GPU Threading Model
2. Separation of Memory Spaces
   - `cudaMalloc`
   - `cudaFree`
   - `cudaMemcpy`
3. Creation of First CUDA program
4. Grids and Blocks
5. Compilation and Debugging

Threading Model

Serial Code

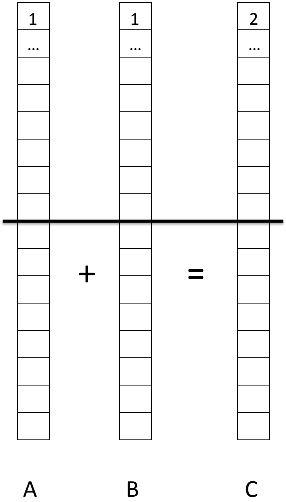


Thread 0

```
for (int i = 0; i < 1000; ++i)
  c[i] = a[i] + b[i];
```

Threading Model

Dual Core Parallelism - 2 Threads



Thread 0

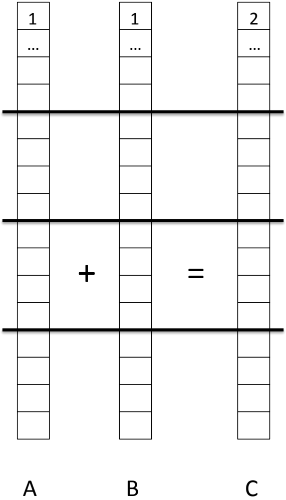
```
for (int i = 0; i < 500; ++i)
  c[i] = a[i] + b[i];
```

Thread 1

```
for (int i = 500; i < 1000; ++i)
  c[i] = a[i] + b[i];
```

Threading Model

Quad Core Parallelism - 4 Threads



Thread 0

```
for (int i = 0; i < 250; ++i)
  c[i] = a[i] + b[i];
```

Thread 1

```
for (int i = 250; i < 500; ++i)
  c[i] = a[i] + b[i];
```

Thread 2

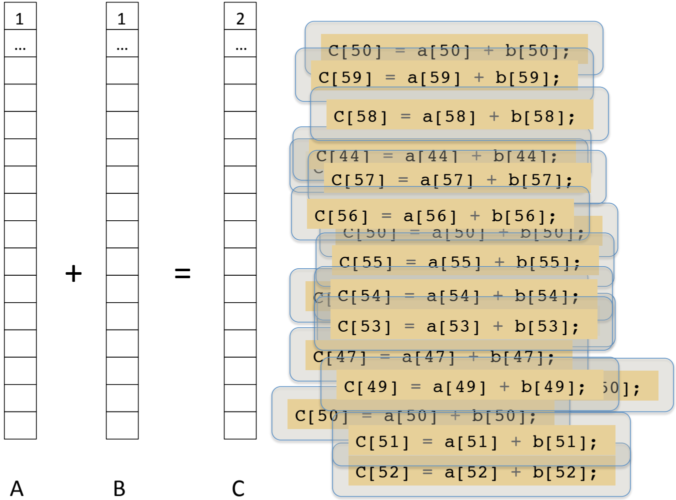
```
for (int i = 500; i < 750; ++i)
  c[i] = a[i] + b[i];
```

Thread 3

```
for (int i = 750; i < 1000; ++i)
  c[i] = a[i] + b[i];
```

Threading Model

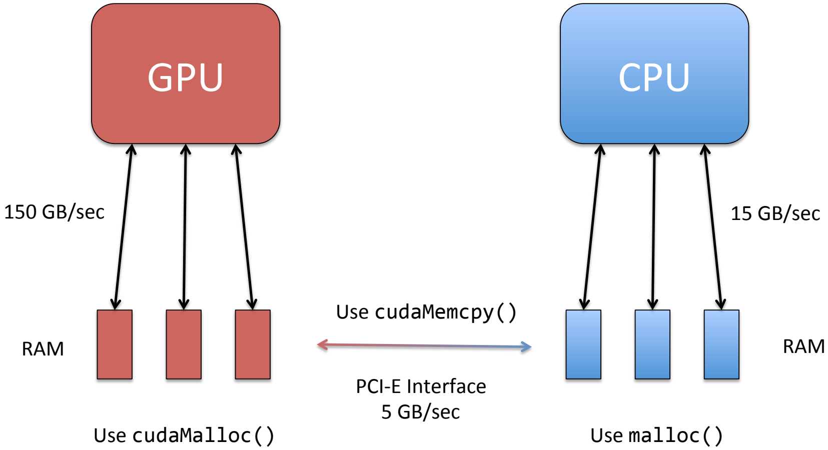
GPU Parallelism - N Threads



Threading Model

- CPU - Number of Threads $\sim$ Physical Cores
- GPU - Number of Threads $\sim$ Amount of Data

Memory Separation



Programming Exercise

First CUDA program:

- Get number of threads from commandline
- Allocate device memory
- Launch Kernel that has each thread save its ID
- Copy the IDs back to the CPU and print them

A few keywords will get you going

- Kernel: a short function that is executed by every thread running on the device (GPU)
- `__global__`: GPU kernel function launched by CPU
- `__global__` functions have the following restrictions:
  - Can only access GPU memory (*)
  - Not recursive
  - Must have `void` return type
  - No static variables
  - No variable number of arguments
- `threadIdx` built-in variable that holds the id of each thread

Memory allocation and Movement

```
cudaMalloc(void **pointer, size_t nbytes);
cudaFree(void *pointer);
cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
```

More about `cudaMemcpy`

- `direction` specifies locations (host or device) of `src` and `dst`
- Blocks CPU thread: returns only after the copy is complete
- Doesn't start copying until previous CUDA calls complete

First CUDA Program

- We'll create it right now!
- Program will be uploaded to github
- Get it with:

```
git clone git://github.com/ekelsen/CME213-LectureExamples.git
```

Introducing nvcc

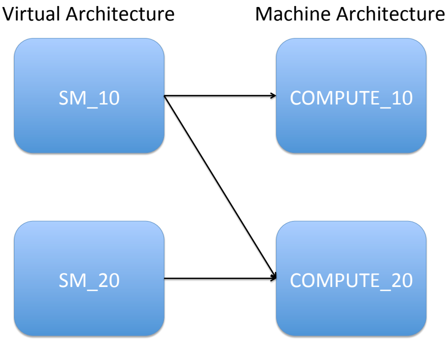
- CUDA files have a .cu extension
- `nvcc` is a complete C++ compiler
  - You can compile (almost) any valid C++ program with it
  - No C++11 or even C++0x support
  - Can also compile CUDA extensions
- Works like you'd expect
  - `nvvc -o test test.cu`
- But one set of very important options!

Virtual Architectures and Machine Code

- Unlike CPUs the GPU instruction set is changing rapidly
- A binary created today won't run on the GPUs of tomorrow
- How to maintain backward compatibility?
- Introduce virtual architectures
- Virtual architecture specifies a virtual instruction set the compiler can target
- The actual instruction set supported by specific architecture is different
- `ptxas` translates virtual ptx code into machine code
- Architectures are forward compatible w.r.t machine code but not backward compatible
    - PTX code for virtual architecture `compute_13` can generate machine code for `sm_20` hardware but not `sm_10`

Practical consequences (for now)

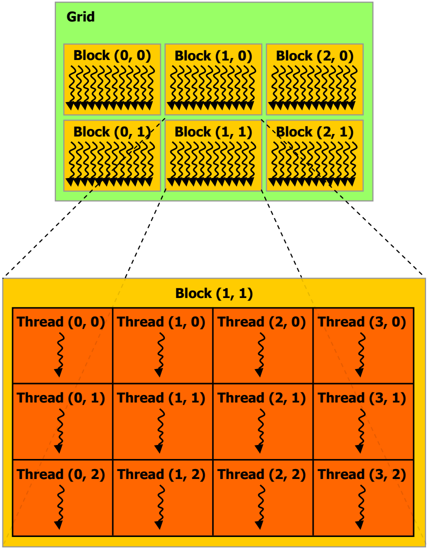
- Default target if none is specified is 1.0
- Cluster GPUs are 2.0
- Later we'll talk about some instructions only available in 2.0+
- `printf`, some atomic operations
- add `-arch=sm_20` option to `nvcc` to generate 2.0 code
- Consequences for timing if JIT is done



Extending our Program

- Lets extend our program to more threads
- New concepts: Grids, Blocks
  - Grids made of blocks
  - Blocks made of threads
  - `threadIdx` - threadId
  - `blockIdx` - blockId
  - `blockDim` - block dimensions
  - `gridDim` - grid dimensions, not used very often
- Kernel launch syntax - `<<<numberOfBlocks, threadsPerBlock>>>`
- Communication much more expensive between blocks compared to within them
- Add some error checking

Grids and Blocks



Asynchronous kernel launches

- `Kernel<<<gridDim, blockDim>>>(...);`
- returns immediately!
- Useful for overlapping CPU/GPU computation
- Bad for error checking.
- Use `cudaDeviceSynchronize()` to force the CPU to wait for the GPU
- Memcopies and allocations are automatically synchronous

Debugging

- "Invalid configuration argument" means that either your block dimensions or grid dimensions were invalid
- Can be because they were too large or zero in all dimensions
- This is easy to diagnose — just print out your grid/block dimensions before you launch the kernel
- "Unspecified launch failure" usually means that an out of bounds memory access has occurred
- This is the GPU equivalent of a segfault
- Use cuda-memcheck to catch out of bounds memory access.

Debugging

- Add -G to the nvcc command to enable GPU side debugging information
- Add `-lineinfo` to get line information in error messages
- `nvcc -o test test.cu -arch=sm_20 -G -lineinfo`
- `cuda-memcheck ./test`
- Similar to valgrind for the cpu.

```
__global__
void myKernel(int *in) {
  in[threadIdx.x] += 1;
}

int main(void) {
  int *dIn;
  cudaMalloc(&dIn, sizeof(int));

  myKernel<<<1, 128>>>(dIn);
  return 0;
}
```

Debugging

Results of `cuda-memcheck ./test` are:

```
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 4
=========     at 0x00000060 in test.cu:4:myKernel
=========     by thread (32,0,0) in block (0,0,0)
=========     Address 0x00201080 is out of bounds
=========
========= ERROR SUMMARY: 1 error
```

Line number is off by one, but it will give a general idea of where to look.

Debugging

- You can use `printf` from inside a kernel!
- You need to make sure the kernel is compiled for at least device capability 2.0
- That's the `-arch=sm_20` option we pass to `nvcc`

Debugging

Don't do this:

```
__global__
void test(...) {
    const int tid = blockDim.x * blockIdx.x + threadIdx.x;
    //...
    printf("%d %d\n", foo, bar);
    //...
}
```

- Every thread in every block will try and print something - tons of data!
- The default size of the print buffer is 8MB
- The buffer is circular
- Main idea: protect the print statement somehow

Debugging

Print only for one block:

```
if (blockIdx.x == 0)
  printf("%d %\n", foo, bar);
```

or even better for only one thread:

```
const int gtid = threadIdx.x + blockIdx.x * blockDim.x;
if (gtid == 4732)
  printf("%d %d\n", foo, bar);
```

Timing

- CPU timers
  - Be careful of asynchronous calls!
  - Use `cudaDeviceSynchronize` to force CPU/GPU synchronization
  - Otherwise you will time launch overhead, not the kernel itself
- GPU Timers
  - Insert events into the GPU timeline
  - We can wait on specific GPU events
  - Determine time between pairs of events
  - Really powerful when combined with streams
- `nvprof` - commandline profiler can also be useful for aggregate info

Common convention

- Need to launch N total threads and have M threads per block
- M is not a divisor of N
- Calculate number of blocks like this:
- `(N + M - 1) / M`

Apply an unary function to input

- $x \rightarrow f(x)$
- We'll write it now!
- Should handle very large inputs
- Arbitrary type for x
- Arbitrary function f

Vector Types

- 2 and 4 component versions of basic datatypes
- These types are ubiquitous in CUDA
- `int2`, `float4`, ...
- Access components with `.x .y .z .w`

```
struct int2 {
  int x;
  int y;
};
```

Actual declarations have additional alignment specifiers

Next time

- 2D/3D blocks and grids
- Warps and memory coalescing
- Shared memory
- Matrix Transpose