CS 179: GPU Computing

Recitation 1 - 4/1/16

Recap

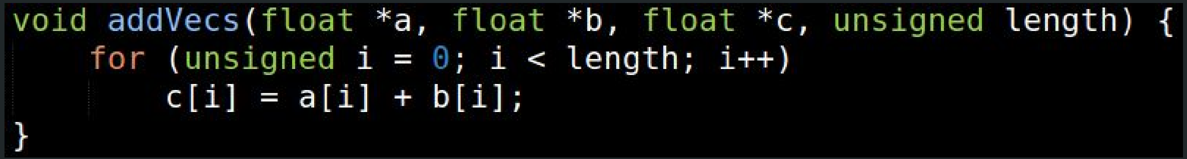
- Device (GPU) runs CUDA kernel defined in .cu and .cuh files
    - C++ code with a few extensions
    - Compiled with proprietary NVCC compiler
    - Kernel defines the behavior of each GPU thread
- Program control flow managed by host (CPU)
    - Uses CUDA API calls to allocate GPU memory, and copy input data from host RAM to device RAM
    - In charge of calling kernel - (almost) like any other function
    - Must also copy output data back from device to host
    - Executable is ultimately C++ program compiled by G++
        - Doesn’t treat object files (.o) produced by NVCC any differently

Recap

- GPU hardware abstraction consists of a *grid* of *blocks* of *threads*
  - Grid and blocks can have up to three dimensions
  - Each block assigned to an independent *streaming multiprocessor* (SM)
  - SM divides blocks into *warps* of 32 threads
  - All threads in a warp execute the same instruction concurrently
  - *Warp divergence* occurs when threads must wait to execute different instructions
    - GPUs are slow - waiting adds up fast!
- *Parallelizable* problems can be broken into independent components
  - Want to assign one thread per “thing that needs to get done”
  - Even better if threads in a warp don’t diverge

Parallelizable Problems

- Most obvious example is adding two linear arrays
  - CPU code:

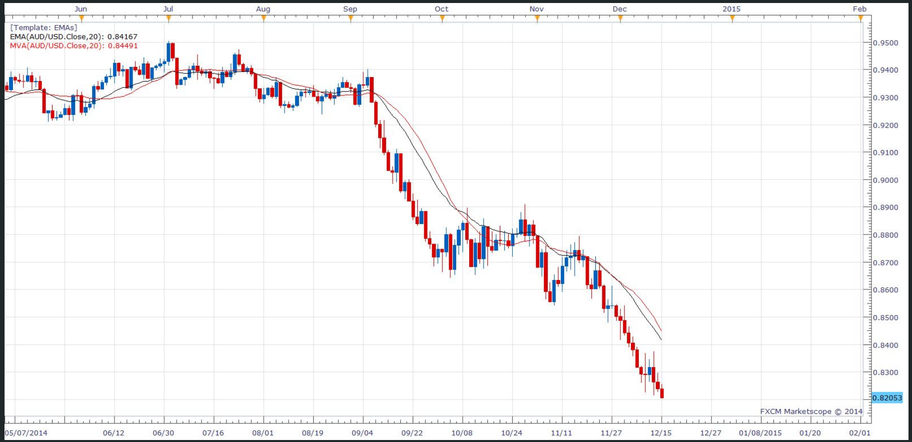


```
void addVecs(float *a, float *b, float *c, unsigned length) {
    for (unsigned i = 0; i < length; i++)
        c[i] = a[i] + b[i];
}
```

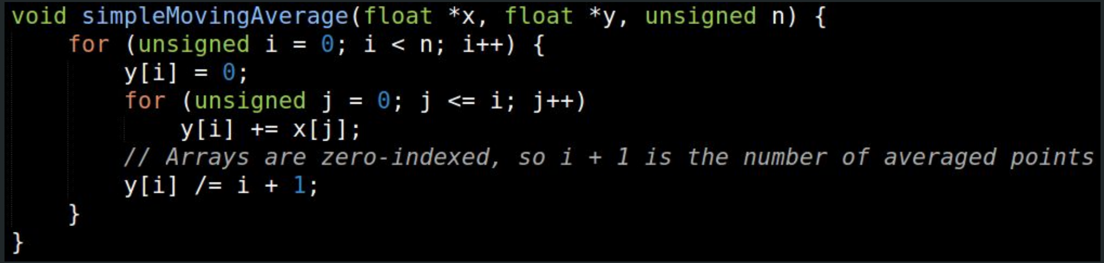
  - Need to allocate **a**, **b**, **c** and populate **a**, **b** beforehand
    - (But you should know how to do that)
- Why is this parallelizable?
  - For **i ≠ j**, operations for **c[i]** and **c[j]** don't have any interdependence
    - Could happen in any order
  - Thus we can do them all at the same time (!) with the right hardware

Non-Parallelizable Problems

- Potentially harder to recognize
- Consider computing a moving average
  - Input array **x** of **n** data points
  - Output array **y** of **n** averages
  - Two well-known options:
    - Simple
    - Exponential



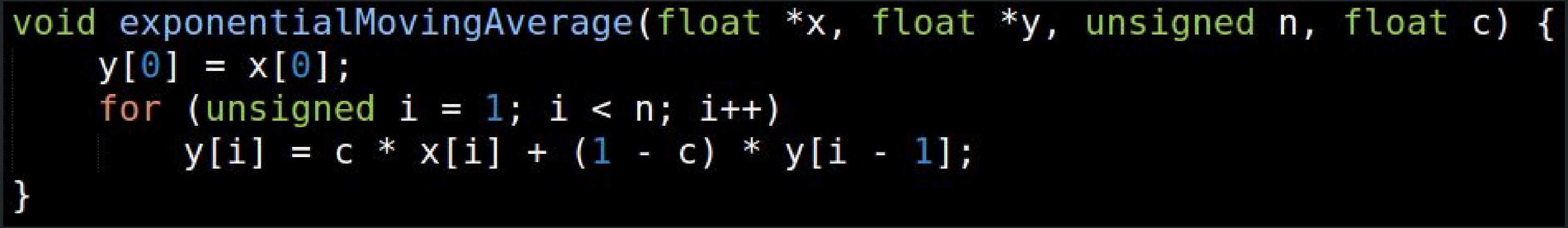
- Simple method just weights all data points so far equally
  - CPU code:
  - Parallelizable? Yes!
    - **y[i]** values separate



```
void simpleMovingAverage(float *x, float *y, unsigned n) {
    for (unsigned i = 0; i < n; i++) {
        y[i] = 0;
        for (unsigned j = 0; j <= i; j++)
            y[i] += x[j];
        // Arrays are zero-indexed, so i + 1 is the number of averaged points
        y[i] /= i + 1;
    }
}
```

Non-Parallelizable Problems

- What about an exponential moving average?
  - Uses a recurrence relation to decay point weight by a factor of $\mathbf{0 < 1 - c < 1}$
    - Specifically, $\mathbf{y[i] = c \cdot x[i] + (1 - c) \cdot y[i - 1]}$
    - Thus $\mathbf{y[n] = c \cdot (x[n] + (1 - c) \cdot x[n - 1] + \ldots + (1 - c)^{n - 1} \cdot x[1]) + (1 - c)^{n} \cdot x[0]}$
  - CPU code:



```c
void exponentialMovingAverage(float *x, float *y, unsigned n, float c) {
    y[0] = x[0];
    for (unsigned i = 1; i < n; i++)
        y[i] = c * x[i] + (1 - c) * y[i - 1];
}
```

  - Parallelizable? Nope
    - Need to know **y[i]** before calculating **y[i + 1]**

What Have We Learned?

- Not all problems are parallelizable
    - Even similar-looking ones
- Harnessing the GPU’s power requires algorithms whose computations can be done at the same time
    - “Parallel execution”
    - Opposite would be “serial execution,” CPU-style
- Output elements should probably not rely on one another
    - Would require multiple kernel calls to compute otherwise
        - Different blocks of threads can’t wait for each other, more on that later in the course
    - In addition to all the extra instructions, there’s a lot of overhead

Assignment 1: Small-Kernel Convolution

- First assignment involves manipulating an input signal
    - In particular, a WAV (.wav) audio file
    - We provide an example to test with
    - Using audio will require libsndfile
        - Installation instructions included in assignment
        - Code also includes an option to use random data instead
- C++ and CUDA files provided, your job to fill in TODOs
    - Code already includes CPU implementation of desired algorithm
    - Your job is to write the equivalent CUDA kernel to parallelize it
    - You're also in charge of memory allocation and host-device data transfers
- Conceptually straightforward, goal is familiarity with integrating CUDA into C++

Some Background on Signals

- A *system* takes input signal(s), produces output signal(s)
- A *signal* can be a continuous or discrete stream of data
  - Typically characterized by amplitude
  - E.g. continuous acoustic input to a microphone
- A continuous signal can also be *discretized*
  - Simply sample it at discrete intervals
    - Ideally periodic in nature
  - E.g. voltage waveform microphone output
- We will consider discrete signals
  - Assignment uses two-channel audio

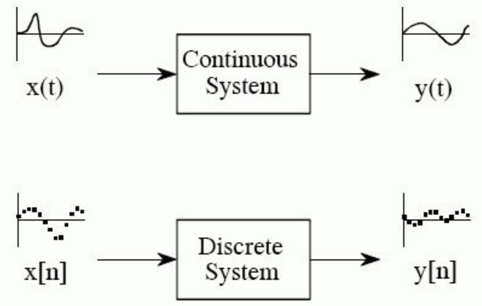


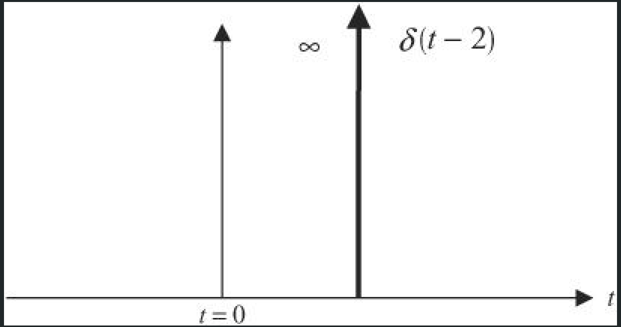
FIGURE 5-1
Terminology for signals and systems. A system is any process that generates an output signal in response to an input signal. Continuous signals are usually represented with parentheses, while discrete signals use brackets. All signals use lower case letters, reserving the upper case for the frequency domain (presented in later chapters). Unless there is a better name available, the input signal is called: $x(t)$ or $x[n]$, while the output is called: $y(t)$ or $y[n]$.

Linear Systems

- Suppose some system takes input signal $\mathbf{x_i[n]}$ and produces output signal $\mathbf{y_i[n]}$
  - We denote this as $\mathbf{x_i[n] \rightarrow y_i[n]}$
- If the system is *linear*, then for constants **a, b** we have:
  - $\mathbf{a \cdot x_1[n] + b \cdot x_2[n] \rightarrow a \cdot y_1[n] + b \cdot y_2[n]}$
- Now suppose we want to pick out a single point in the signal
  - We can do this with a *delta function*, $\boldsymbol{\delta}$
  - If we treat it as a discrete signal, we can define it as:
    - $\boldsymbol{\delta}\,\mathbf{[n - k] = 1}$ if $\mathbf{n = k}$, $\boldsymbol{\delta}\,\mathbf{[n - k] = 0}$ if $\mathbf{n \neq k}$
  - “Zero everywhere with a spike at **k**”
- This definition means that $\mathbf{x[k] = x[n] \cdot \boldsymbol{\delta}\,[n - k]}$
  - **Note:** I was wrong about this in recitation. We use the delta function to pick out the value of signal **x[n]** at constant point **k**.

Linear Systems

- Next we can define a system's response to **δ [n - k]** as **h_k[n]**
  - I.e. **δ [n - k] $\rightarrow$ h_k[n]**
- From linearity we then have **x[n] · δ [n - k] $\rightarrow$ x[n] · h_k[n]**
  - **x[n]** is the input signal, **δ [n - k]** is the delta function signal
    - **Note:** I was wrong about this in recitation; see the previous slide for details.
  - Response at time **k** defined by response to delta function



Time-Invariance

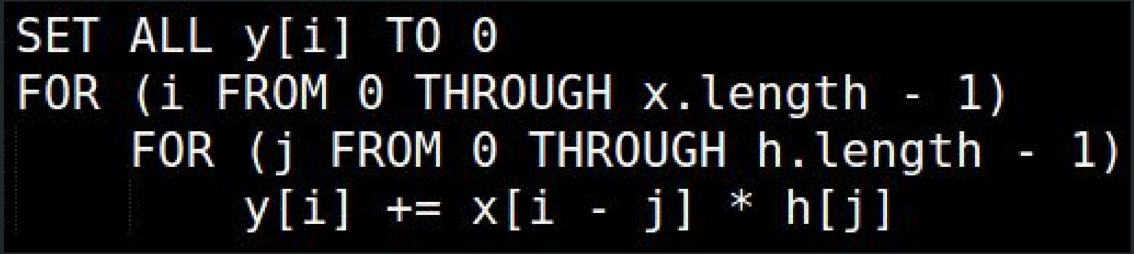
- If a system is *time-invariant*, then it will satisfy:
    - $\mathbf{x[n]} \rightarrow \mathbf{y[n]} \Rightarrow \mathbf{x[n + m]} \rightarrow \mathbf{y[n + m]}$ for integer $\mathbf{m}$
- Thus given $\boldsymbol{\delta}\ \mathbf{[n - k]} \rightarrow \mathbf{h_k[n]}$ and $\boldsymbol{\delta}\ \mathbf{[n - l]} \rightarrow \mathbf{h_l[n]}$, we can say that $\mathbf{h_k[n]}$ and $\mathbf{h_l[n]}$ are "time-shifted" versions of each other
    - Instead of a new response $\mathbf{h_k[n]}$ for each $\mathbf{k}$, we can define $\mathbf{h[n]}$ such that $\boldsymbol{\delta}\ \mathbf{[n]} \rightarrow \mathbf{h[n]}$, and shift $\mathbf{h}$ with $\mathbf{k}$ such that $\boldsymbol{\delta}\ \mathbf{[n - k]} \rightarrow \mathbf{h[n - k]}$
    - By linearity, we then have $\mathbf{x[n]} \cdot \boldsymbol{\delta}\ \mathbf{[n - k]} \rightarrow \mathbf{x[n]} \cdot \mathbf{h_k[n]}$
- This lets us rewrite the system's response $\mathbf{x[n]} \rightarrow \mathbf{y[n]}$:
    - $\mathbf{x[n] = \Sigma\ x[k] \cdot \boldsymbol{\delta}\ [n - k] \rightarrow \Sigma\ x[k] \cdot h_k[n - k] = x[k] \cdot h_k[n] = y[n]}$
        - Output must be equivalent to $\mathbf{y[n]}$ because $\mathbf{x[n]} \rightarrow \mathbf{y[n]}$
    - **Note:** sum is over all $\mathbf{k}$.

What Have We Learned?

- *Linear time-invariant* systems have some very convenient properties
  - Most importantly, they can be characterized entirely by **h[n]**
  - This allows **y[n]** to be written entirely in terms of the input samples **x[k]** and the delta function response **h[n]**
- Remember:
  - $\mathbf{y[n] = \Sigma\, x[k] \cdot h_k[n - k]}$
  - **x[n]** is the input signal to our system
  - **y[n]** is the output signal, or “impulse response” from our system
  - $\boldsymbol{\delta}$ **[n]** is the delta function signal
  - **h[n]** is the impulse response from our system for $\boldsymbol{\delta}$ **[n]**

Putting It All Together

- Assignment asks you to accelerate convolution of an input signal
    - E.g. input **x[0..99]**, system with **h[0..3]** delta function response
    - For *finite-duration* **h** such as this, computable with $\mathbf{y[n] = \Sigma\, x[k] \cdot h[n - k]}$
    - **y[50]** computation, for example, would be:
        - $\mathbf{y[50] = x[47] \cdot h[3] + x[48] \cdot h[2] + x[49] \cdot h[1] + x[50] \cdot h[0]}$
        - All other **h** terms are **0**
        - Here **y[50]** etc. refer to the signal at that point
- This sum is parallelizable
    - Pseudocode:



```
SET ALL y[i] TO 0
FOR (i FROM 0 THROUGH x.length - 1)
    FOR (j FROM 0 THROUGH h.length - 1)
        y[i] += x[i - j] * h[j]
```

Assignment Details

- All you need to worry about is the kernel and memory operations
- We provide the skeleton and some useful tools
  - CPU implementation - reference this for your GPU version
  - Error checking code for your output
  - Delta function response **h[n]** (default is Gaussian impulse response)
    - **Note:** I was wrong in the recitation, saying that **h[n]** is the response to any function we wish to convolve. Rather, the system is defined such that its response to the delta function is the signal we to convolve.
    - This derivation is a discrete-time version of https://en.wikipedia.org/wiki/LTI_system_theory#Impulse_response_and_convolution. Looking at this will help distinguish when we refer to the signal as a function and when we refer to a specific point in it.

Assignment Details

- Code can be compiled in one of two modes
  - Normal mode (**AUDIO_ON** defined to be **0**)
    - Generates random **x[n]**
    - Can test performance on various input lengths
    - Can run repeated trials by increasing number of channels
  - Audio mode (**AUDIO_ON** defined to be **1)**
    - Reads **x[n]** from input WAV file
    - Generates output WAV from **y[n]**
    - Gaussian **h[n]** is an (imperfect) *low-pass* filter - high frequencies should be attenuated

Debugging Tips

- **printf()** can be useful, but gets messy if all threads print
    - Better to only print from certain threads, though your kernel will diverge
- If you want to check your kernel's output, copy it back to the host
    - More manageable than printing from the kernel and you can write normal C++ to inspect the data
- Use the **gpuErrchk()** macro to check CUDA API calls for errors
    - Example usage: **gpuErrchk(cudaMalloc(&dev_in, length * sizeof (int)));**
    - Prints error info to stderr and exits
- Use small convolution test cases before trying large arrays or the test WAV
    - E.g. 5-element **x[n]**, 3-element **h[n]**

Any Questions?