10-405

Using RNNs and CNNs

Catchup from Monday's lecture

Putting together Deep Learners

encoder/decoder

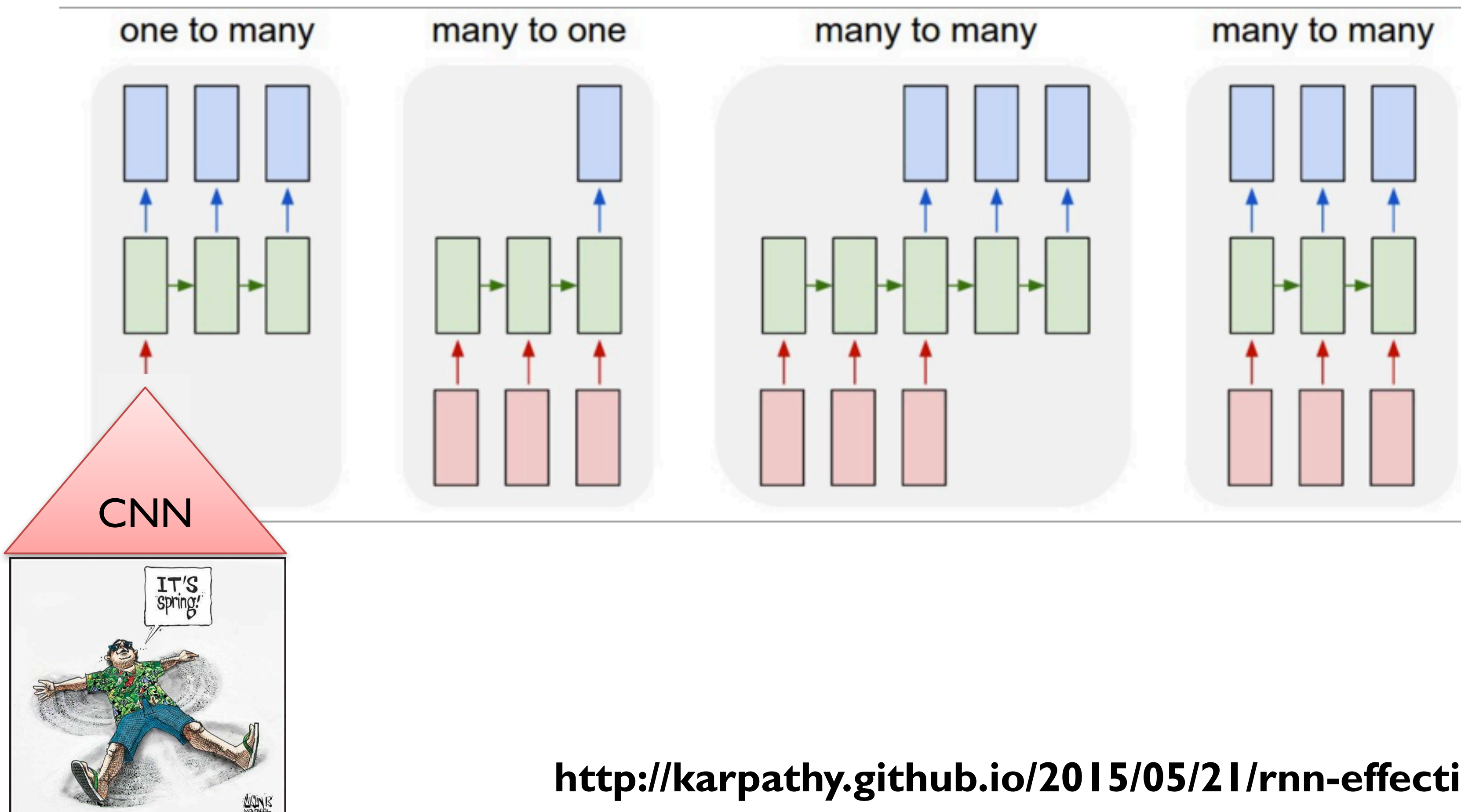
seq2seq

image captioning

sequence classification

translation

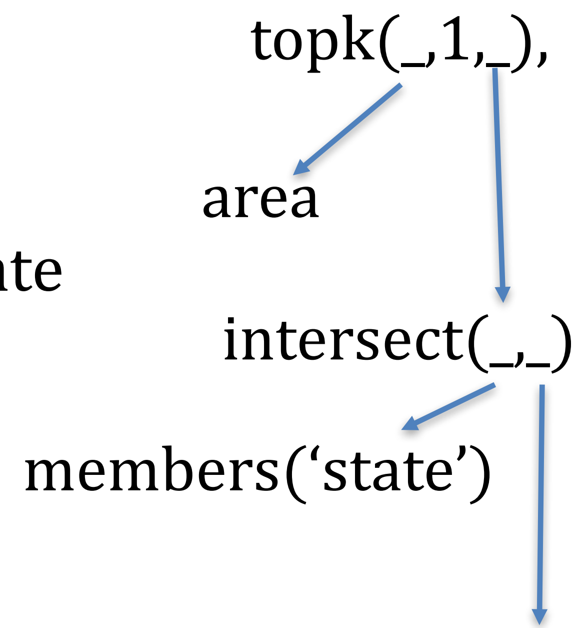
named entity recognition



http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Other examples of seq2seq applications

- ”Code captioning” (Yang et al, NIPS 2016):
  - Input: Java class implementation
  - Output: class-level comments
- Semantic parsing (Yih et al, ACL 2016)
  - Input: NL question “What’s the US state bordering Maryland?”
  - Output: code like “topk(area,1,intersect(members(‘state’), sharesBorder(named(‘Maryland’))”



  - Might emit seq of *operations that add to a tree* instead of tokens: topk(_,1,_), area, intersect(_,_), members(‘state’), ….

Example: reasoning about entailment

A large annotated corpus for learning natural language inference

Samuel R. Bowman[*†]
sbowman@stanford.edu

Gabor Angeli[†‡]
angeli@stanford.edu

Christopher Potts[*]
cgpotts@stanford.edu

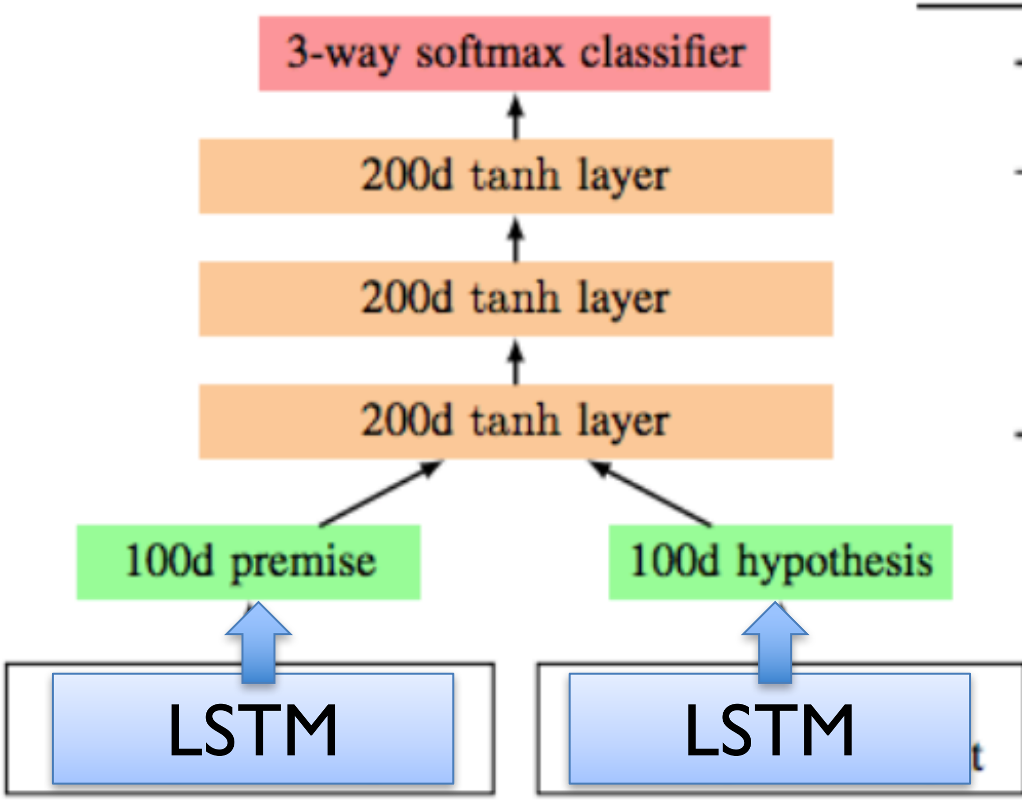
Christopher D. Manning[*†‡]
manning@stanford.edu

A man inspects the uniform of a figure in some East Asian country.	**contradiction** C C C C C	The man is sleeping
An older and younger man smiling.	**neutral** N N E N N	Two men are smiling and laughing at the cats playing on the floor.
A black race car starts up in front of a crowd of people.	**contradiction** C C C C C	A man is driving down a lonely road.
A soccer game with multiple males playing.	**entailment** E E E E E	Some men are playing a sport.
A smiling costumed woman is holding an umbrella.	**neutral** N N E C N	A happy woman in a fairy costume holds an umbrella.

RNNs for entailment

Sentence model	Train	Test
100d Sum of words	79.3	75.3
100d RNN	73.1	72.2
100d LSTM RNN	84.8	**77.6**

System	SNLI
Edit Distance Based	71.9
Classifier Based	72.2
+ Lexical Resources	**75.0**



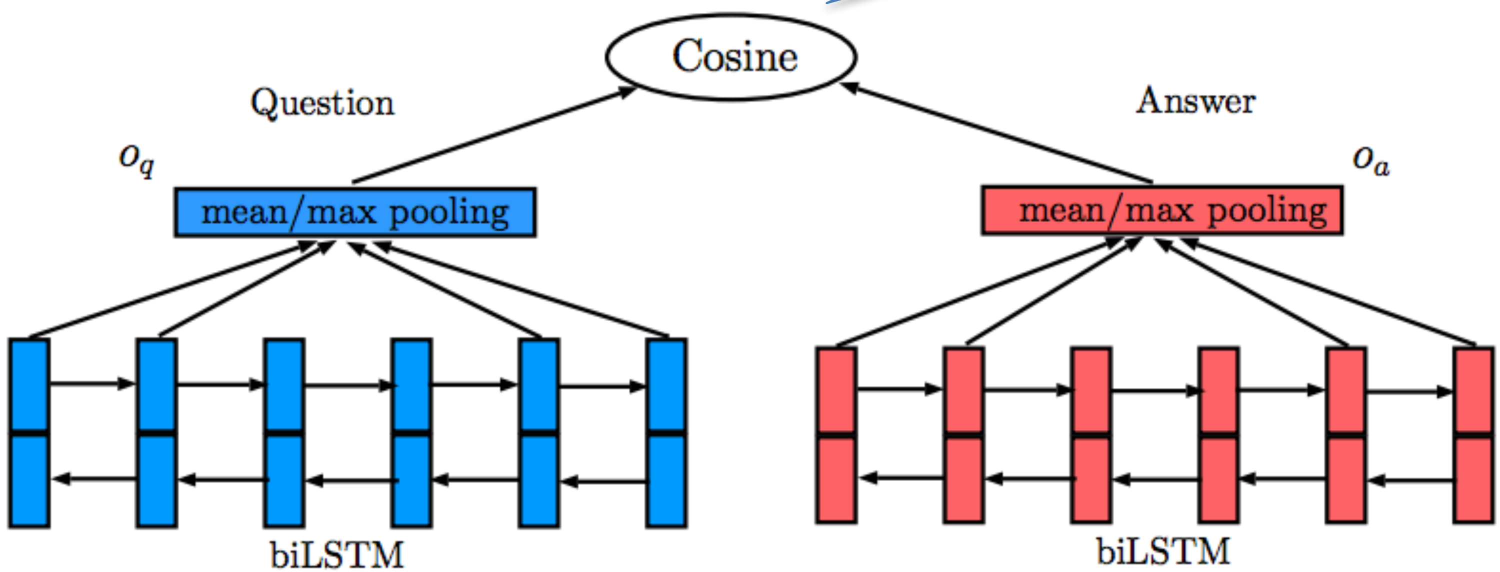
LSTM here is a macro – it's expanded out to build a larger computation graph

Example: question answering

LSTM-BASED DEEP LEARNING MODELS FOR NON-FACTOID ANSWER SELECTION

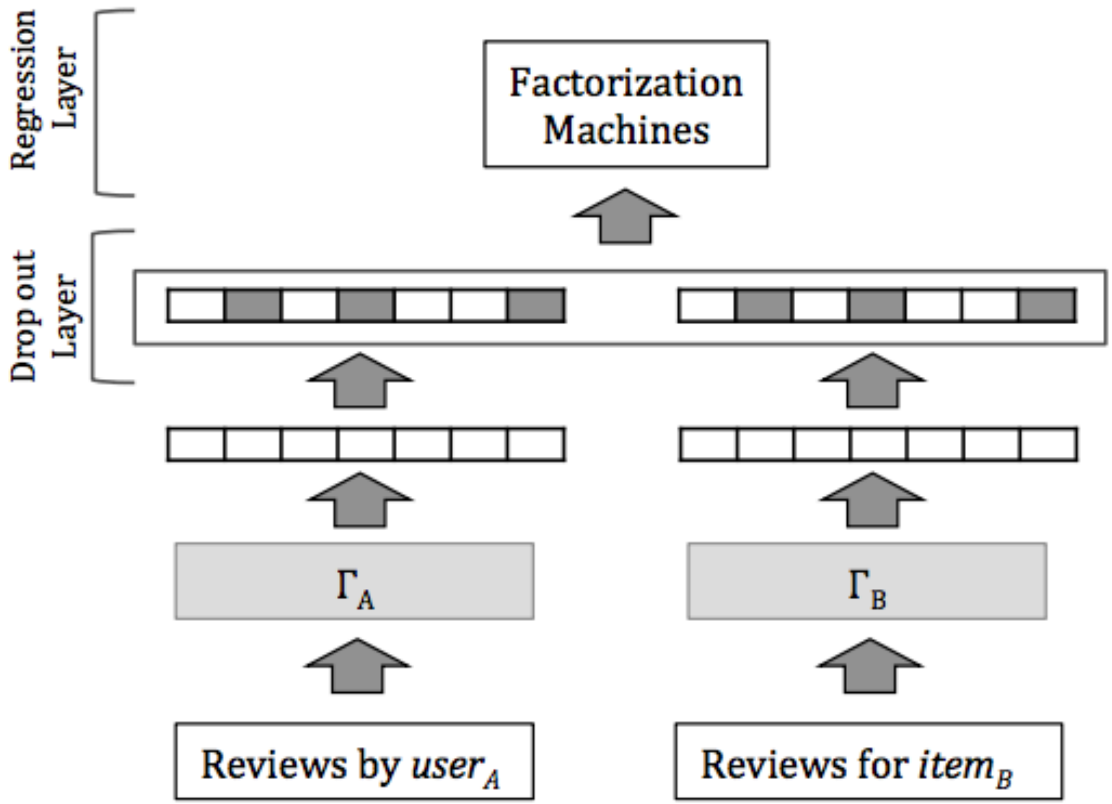
Ming Tan, Cicero dos Santos, Bing Xiang & Bowen Zhou
IBM Watson Core Technologies
Yorktown Heights, NY, USA
{mingtan,cicerons,bingxia,zhou}@us.ibm.com

Common trick: train network to make representations similar/dissimilar, not to classify



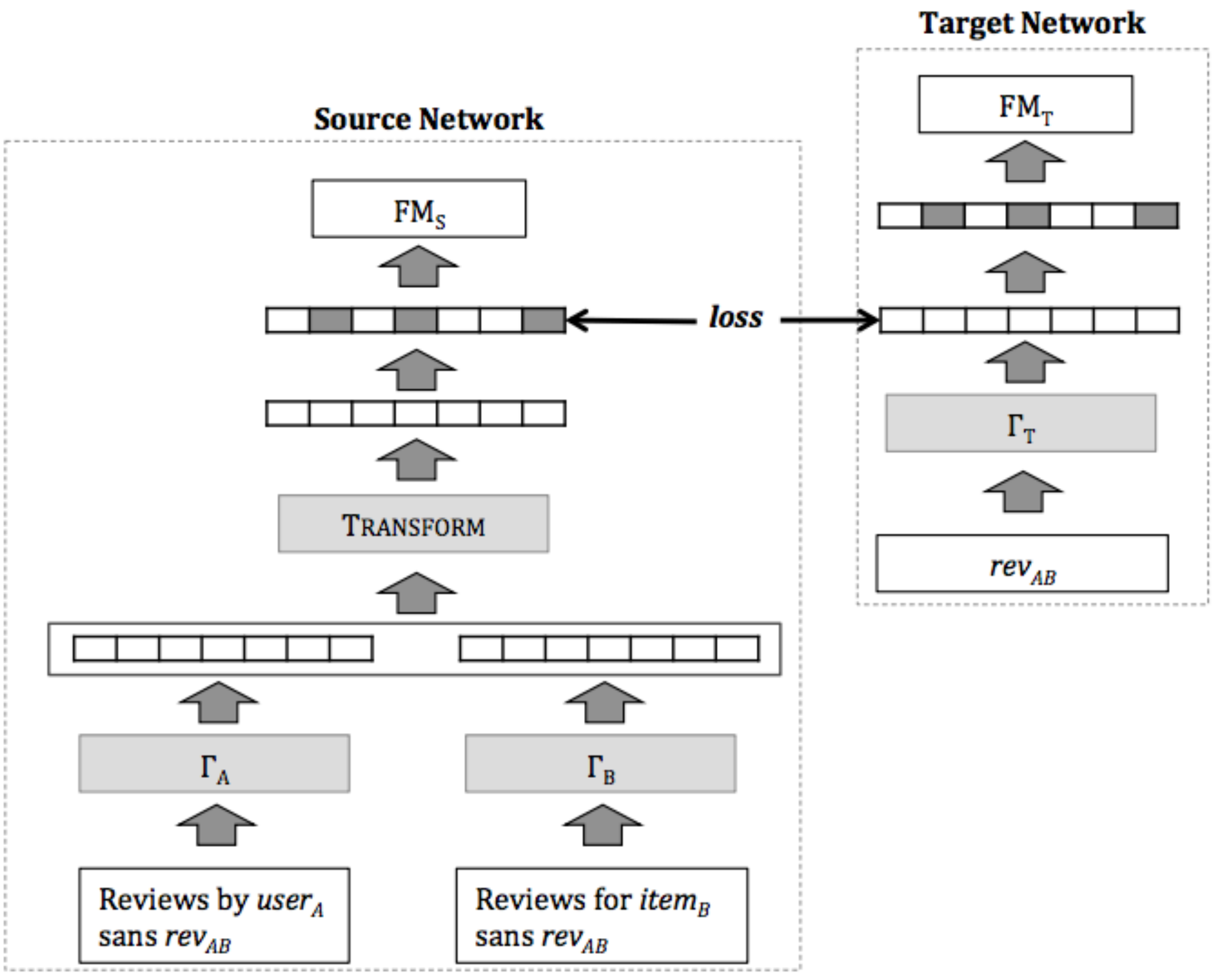
Example: recommendation

Rose Catherine & Cohen, RecSys 2017



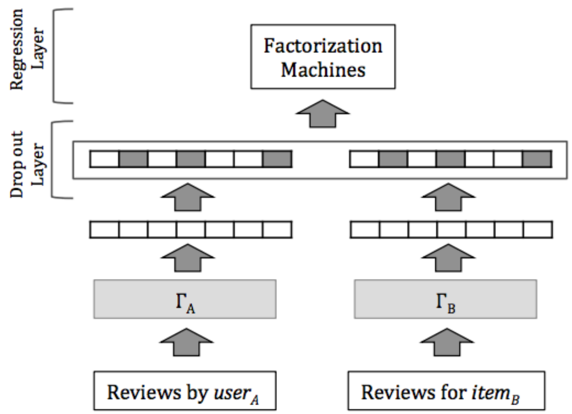
Example: recommendation

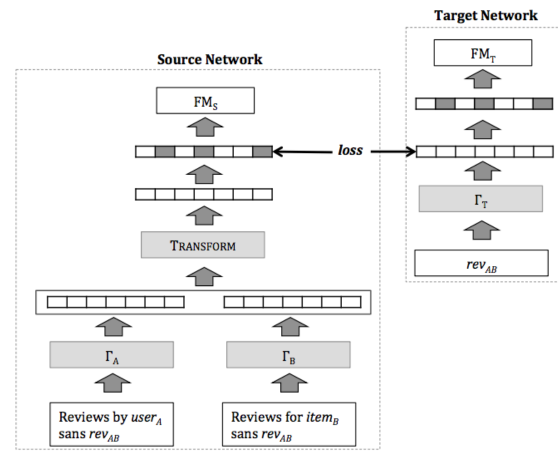
Rose Catherine & Cohen, RecSys 2017



Example: recommendation

Rose Catherine & Cohen, RecSys 2017



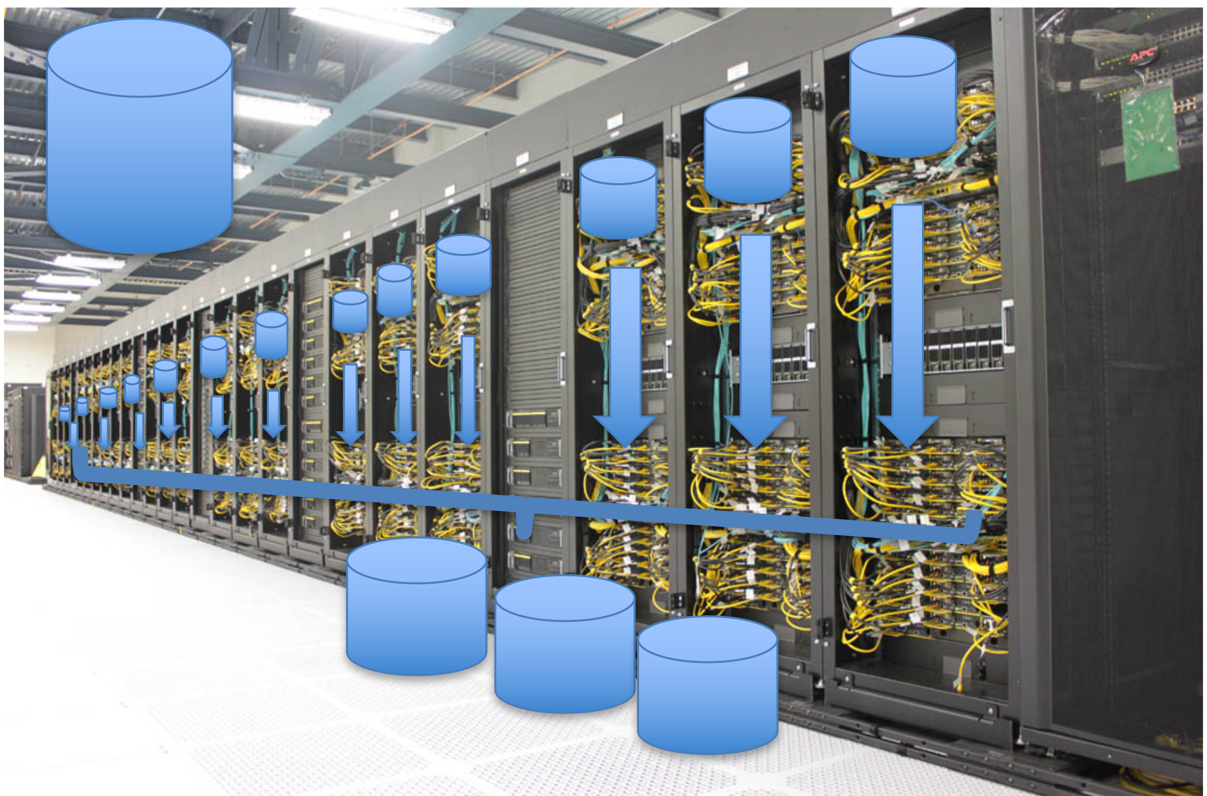


Dataset	DeepCoNN + Test Reviews	MF	DeepCoNN	DeepCoNN-rev$_{AB}$	TransNet	TransNet-Ext
Yelp17	1.2106	1.8661	1.8984	1.7045	1.6387	1.5913
AZ-Elec	0.9791	1.8898	1.9704	2.0774	1.8380	1.7781
AZ-CSJ	0.7747	1.5212	1.5487	1.7044	1.4487	1.4780
AZ-Mov	0.9392	1.4324	1.3611	1.5276	1.3599	1.2691

Big ML and GPUs

Parallel computing with map-reduce:

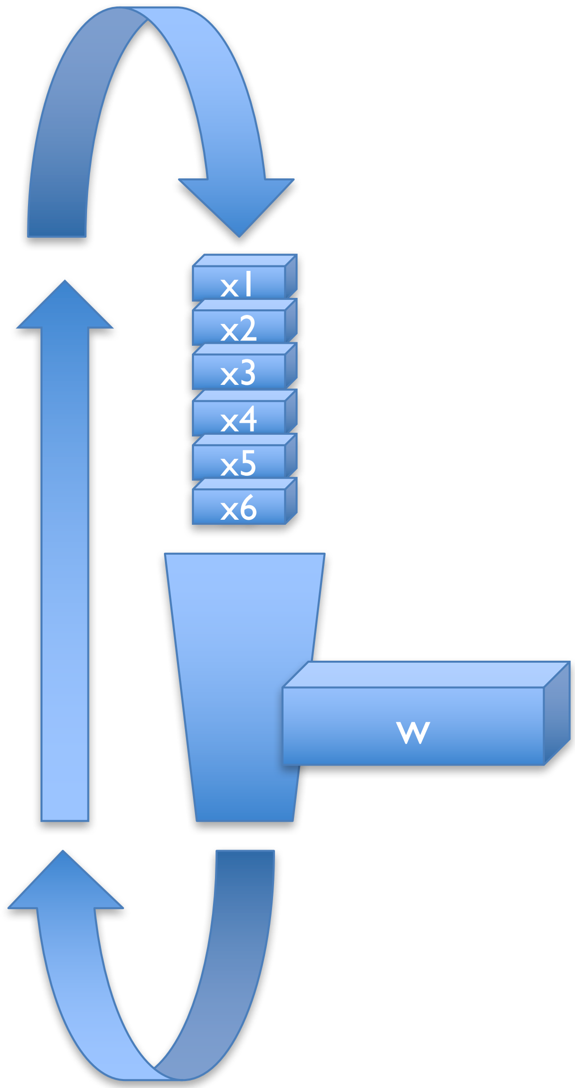
- Stream-and-sort in parallel
- Enormous datasets
- Tasks are i/o bound
- Many unreliable processors
  - which are basically commodity PCs
- Parallelize with mapreduce
  - loosely coupled, heavy-weight jobs
  - communicate via network/disk
- Don’t iterate (typically)



Streaming SGD:

- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly

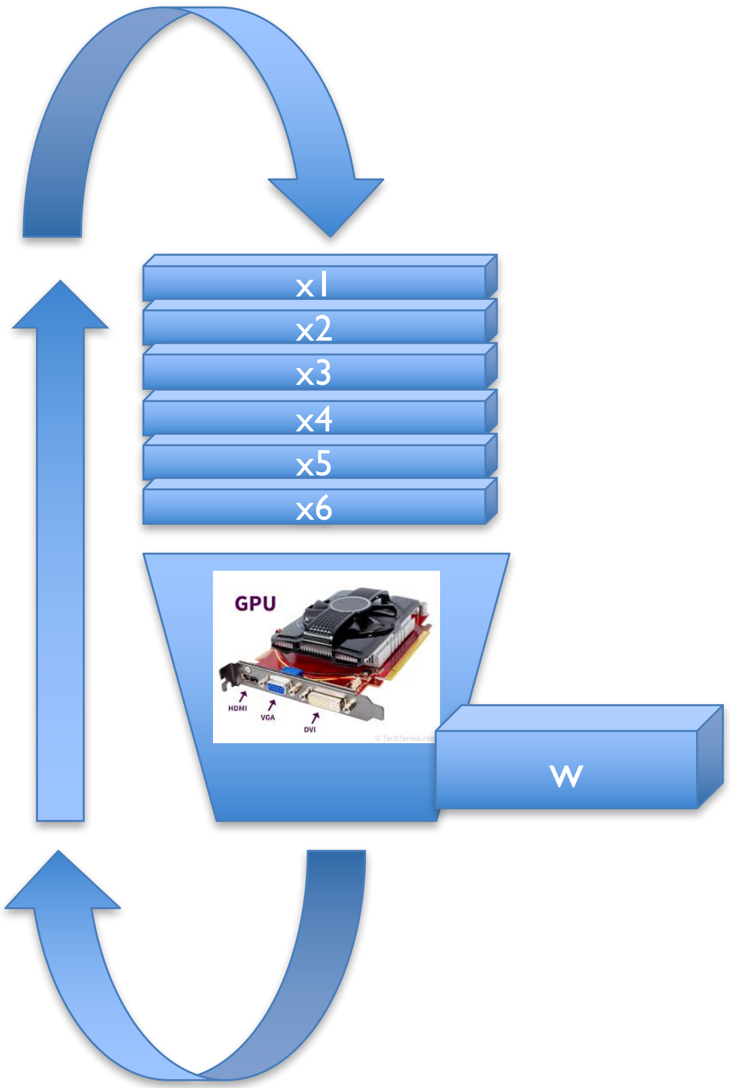


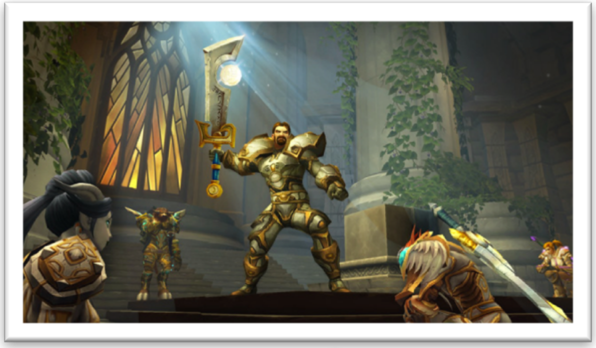
Streaming SGD:

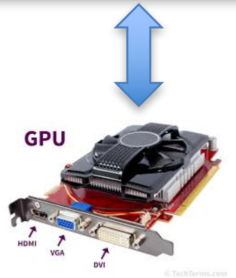
- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly

- You can speed it up by making the tasks in the stream bigger and doing them in **parallel**
- A GPU is a good way of doing that





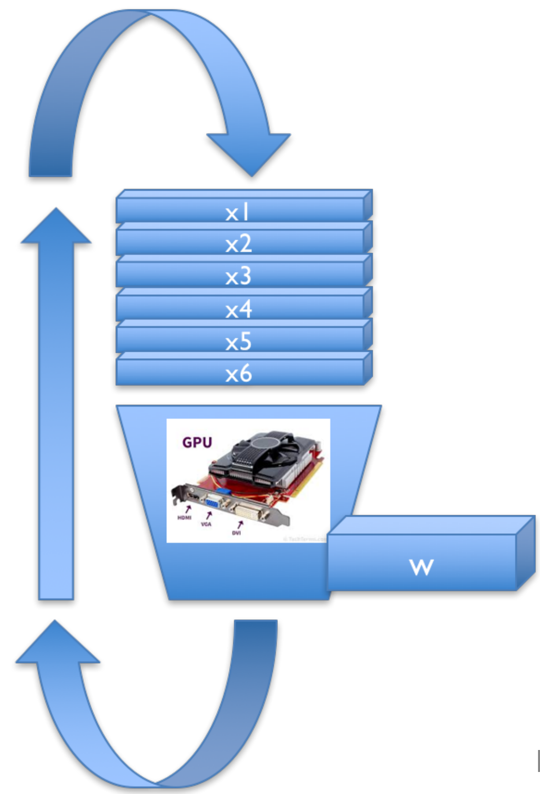


Parallel computing with map-reduce:

- Stream-and-sort in parallel
- Enormous datasets
- Tasks are i/o bound
- Many unreliable processors
    - which are basically commodity PCs
- Parallelize with mapreduce
    - loosely coupled, heavy-weight jobs
    - communicate via network/disk
- Don't iterate (typically)

Parallel ML computing with GPUS:

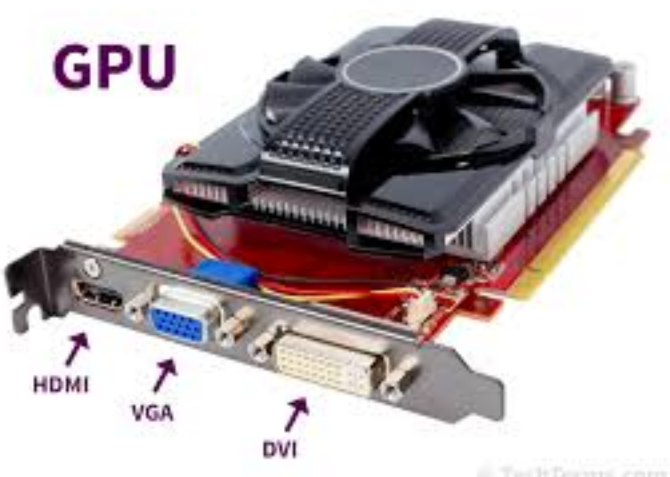
- Iterative streaming ML in parallel
- Big-but-not-too-big datasets
- Tasks are compute bound
- Many fast-but-simple processors
- Replace streaming operations with medium-sized computations that can be done in parallel
- Usually iterate many times



WHAT ARE GPUS?

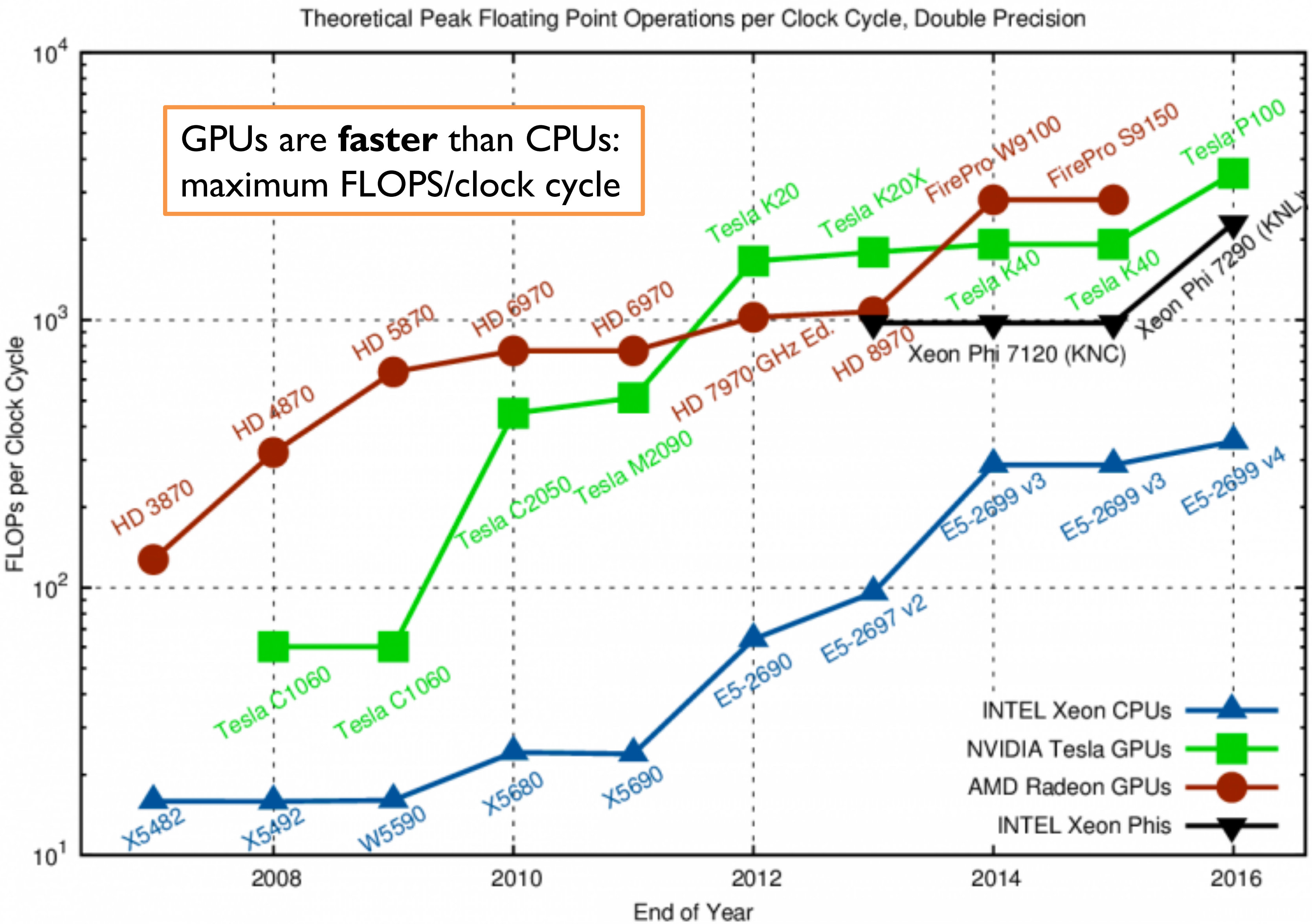
What is a GPU?

A **graphics processing unit** (**GPU**) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. [wikipedia]

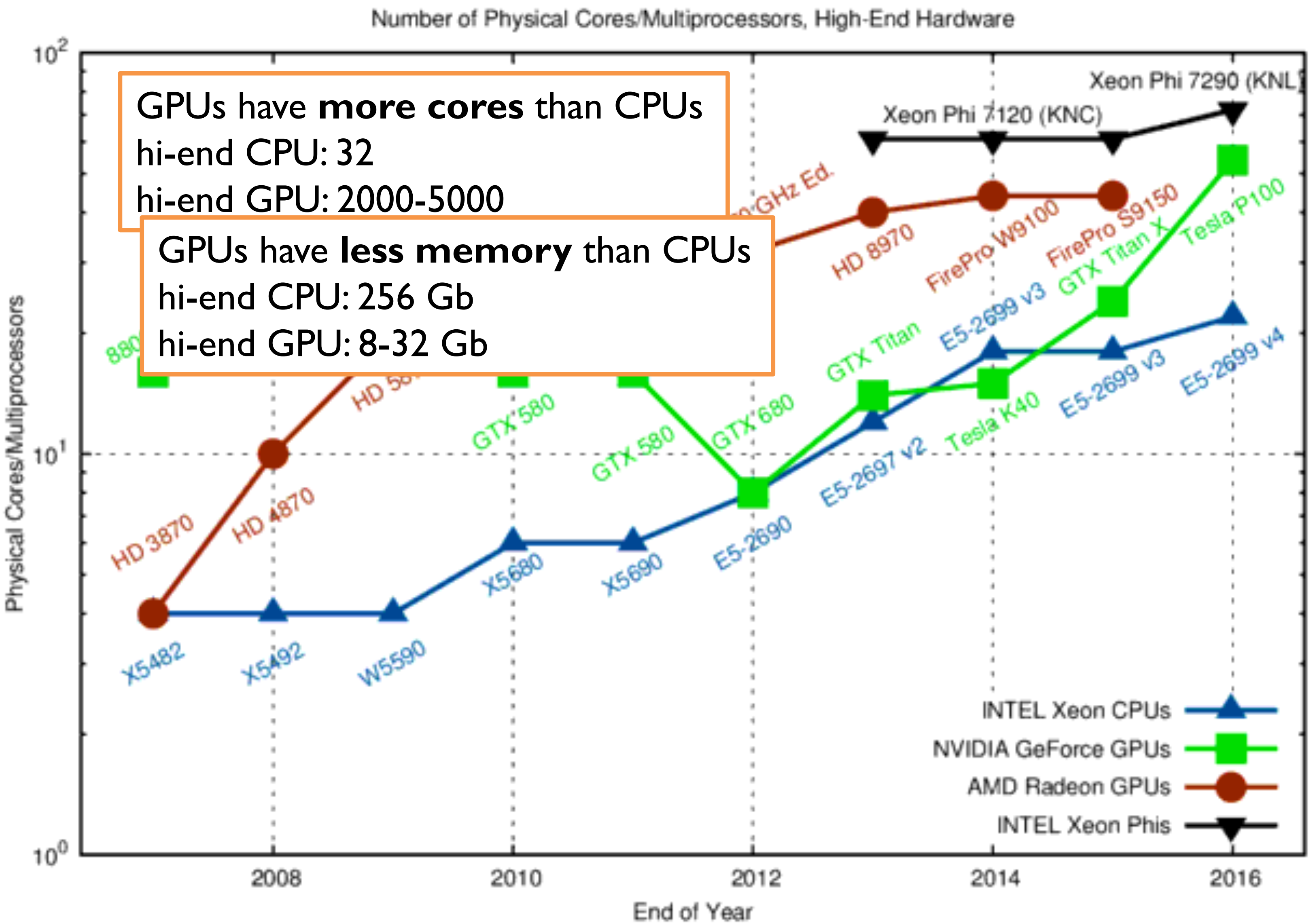


The term GPU was popularized by Nvidia in 1999, who marketed the GeForce 256 as "the world's first …Graphics Processing Unit." It was presented as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines".[3]

GPUs are **faster** than CPUs:
maximum FLOPS/clock cycle



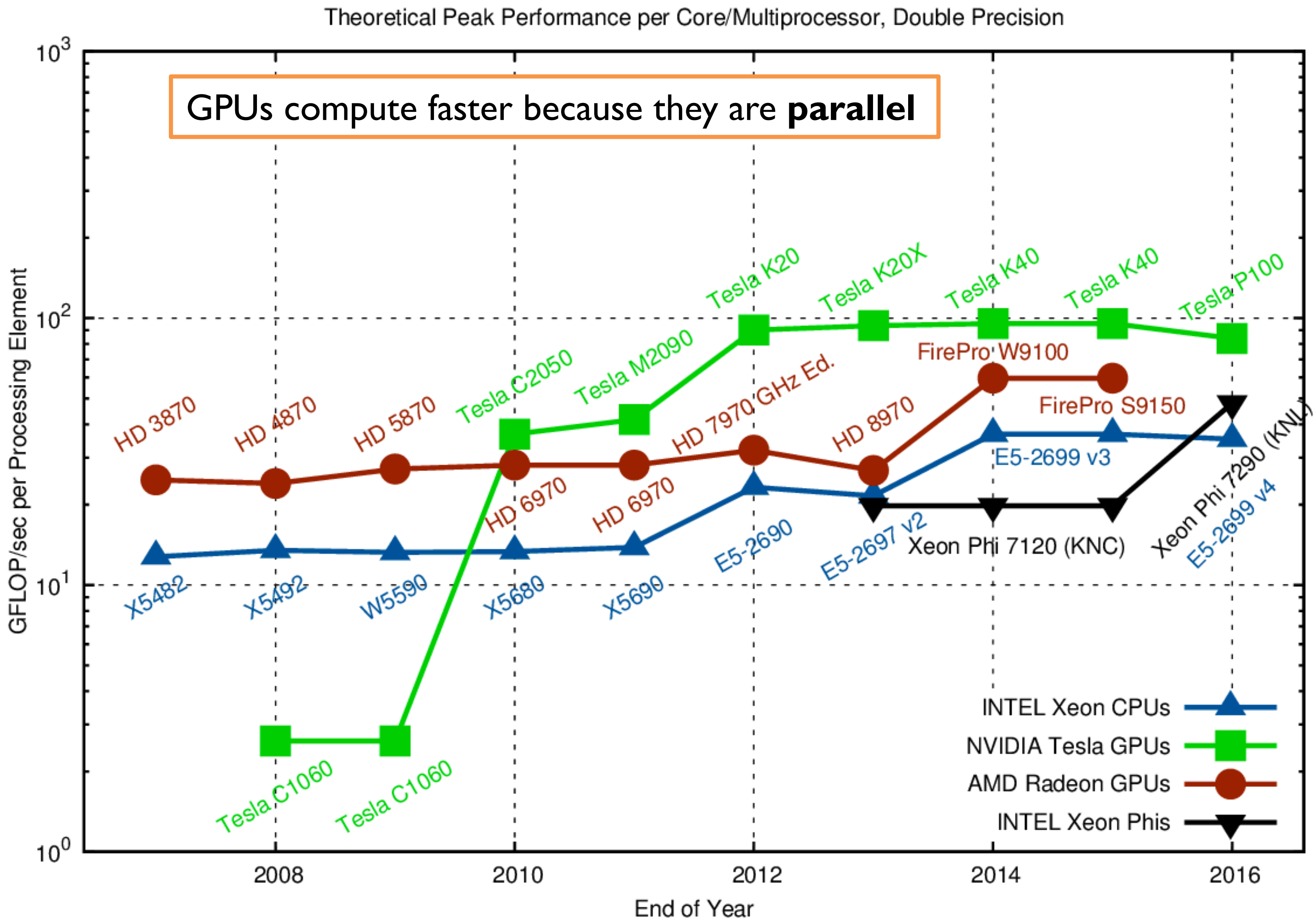
https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/



GPUs have **more cores** than CPUs
hi-end CPU: 32
hi-end GPU: 2000-5000

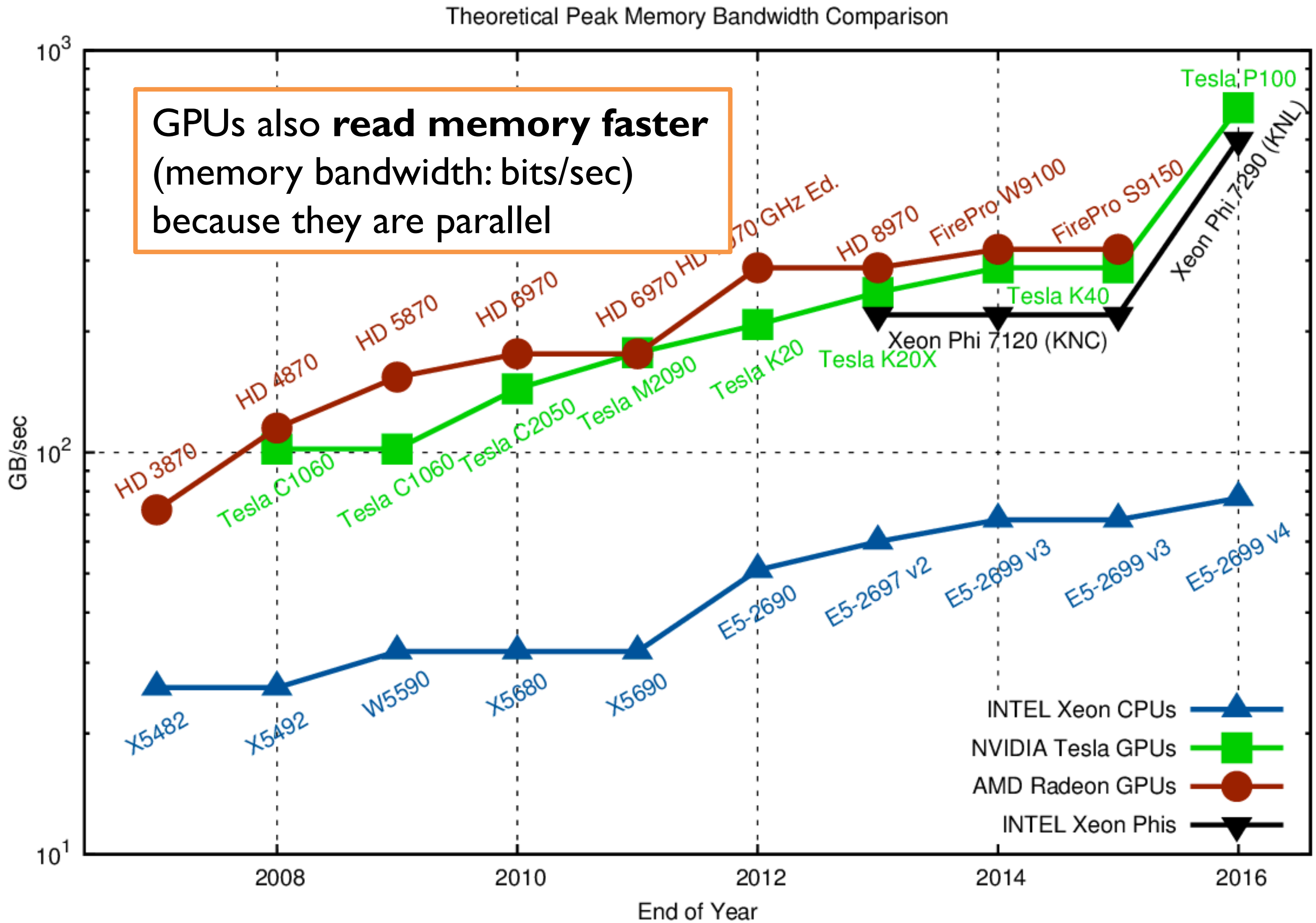
GPUs have **less memory** than CPUs
hi-end CPU: 256 Gb
hi-end GPU: 8-32 Gb

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

GPUs also **read memory faster** (memory bandwidth: bits/sec) because they are parallel

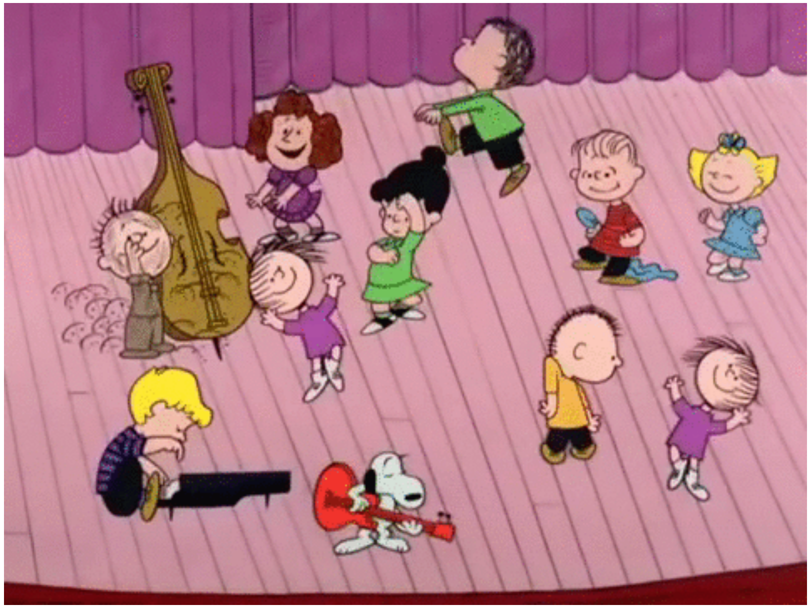


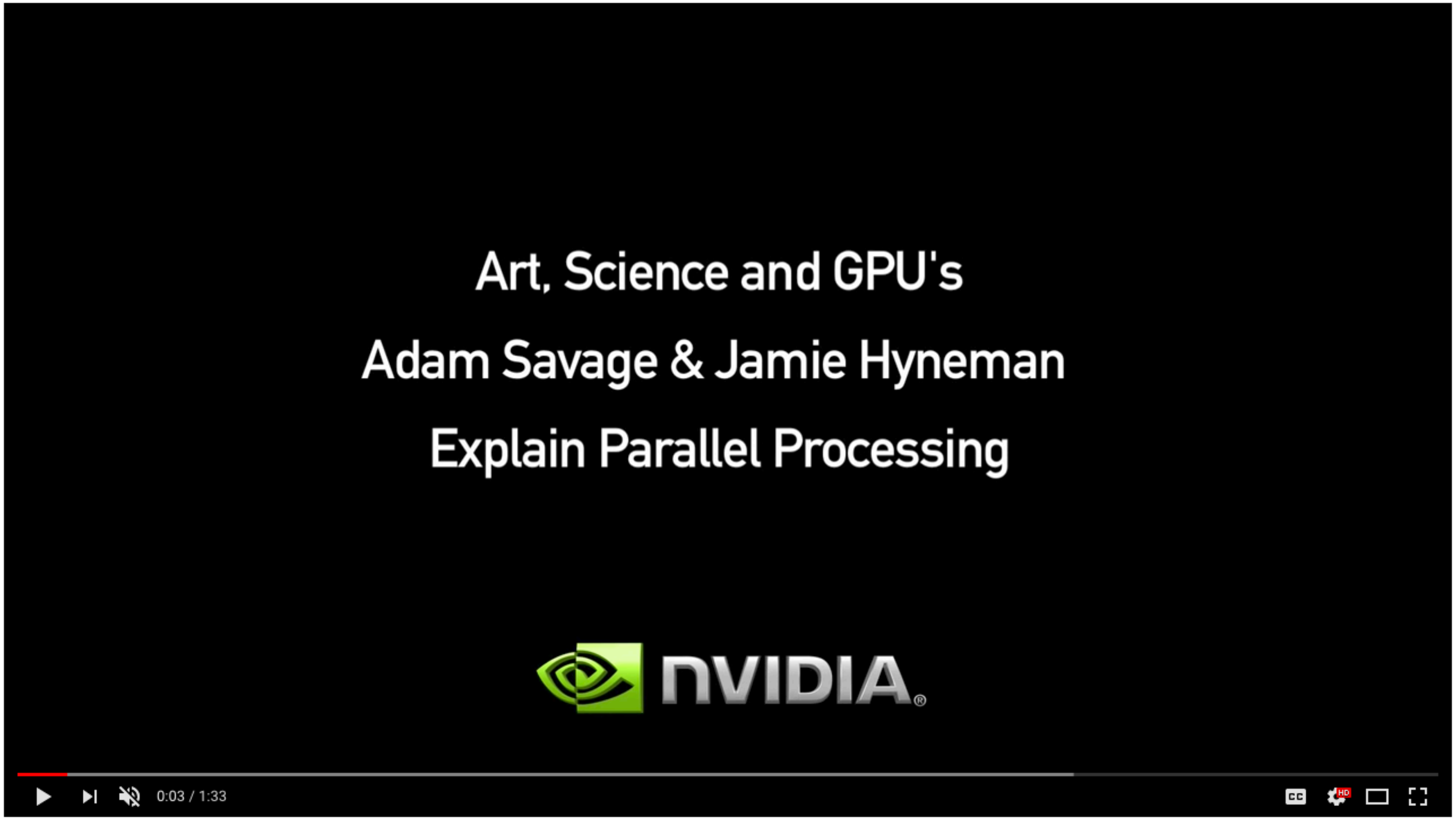
https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Summary of GPUs vs CPUs

- less total memory
- more cores and more parallelism

- Multicore CPUs are mostly multiple-instruction multiple-data (MIMD)
- GPUs are mostly single-instruction multiple-data (SIMD)





https://www.youtube.com/watch?v=-P28LKWTzrI

HOW DO YOU USE A GPU?

http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

Using GPUs for ML

- Programming parallel machines is complicated
- To use the parallelism of a GPU in an ML algorithm we almost always use **matrix algebra** as an abstraction layer – i.e. *vectorize*

```
def logisticRegression(....):
    ....
    for X,Y in ....:
      Z = W.matrix_multiply(X)
        P = logistic(Z)
        W = W + learning_rate * (P – Y) * X
        .....
```

Using GPUs for ML


```
def logisticRegression(....):
  ....
  for X,Y in ....:
    Z = W.matrix_multiply(X)
    P = logistic(Z)
    W = W + learning_rate * (P – Y) * X
    .....
```

Using GPUs for ML

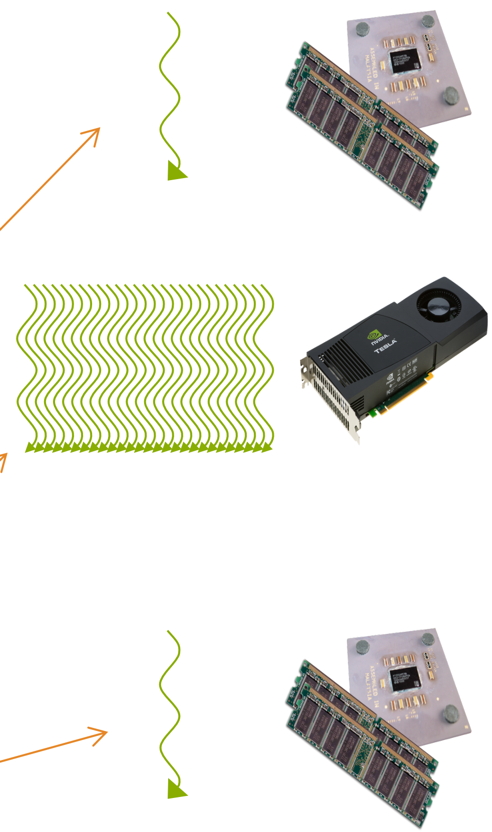
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;            // host copies of a, b, c
        int *d_in, *d_out;        // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

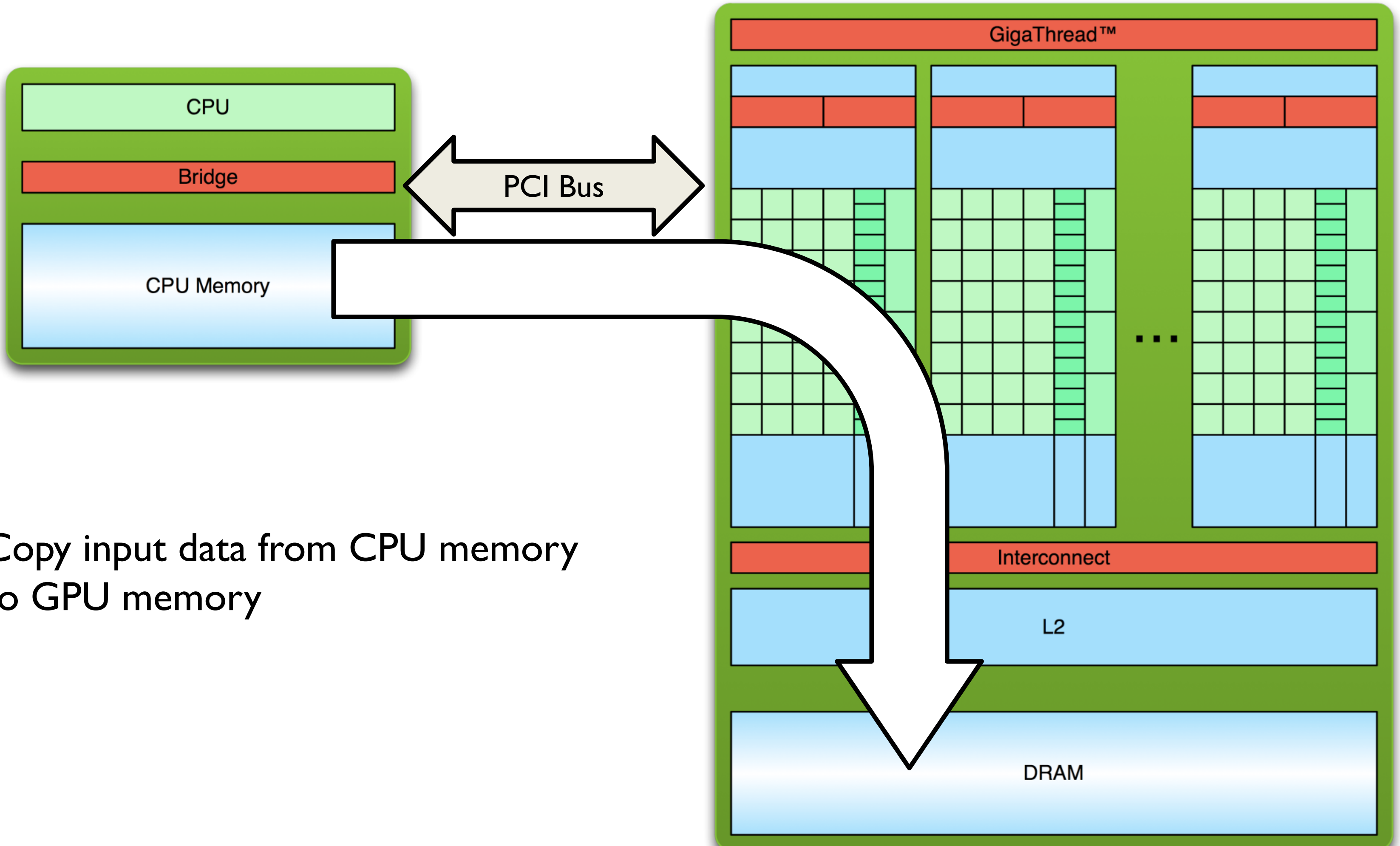
serial code

parallel code

serial code

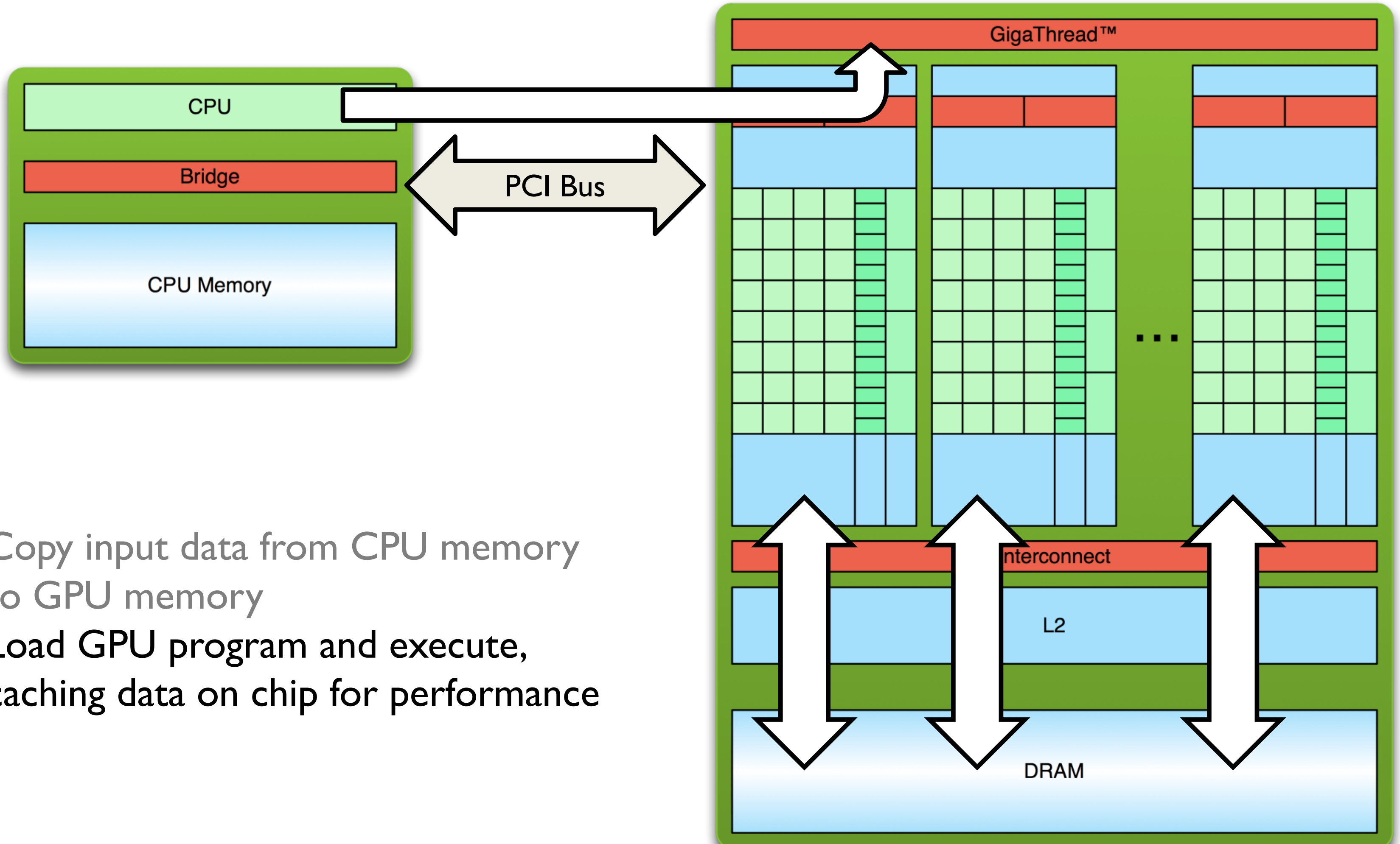


Simple Processing Flow



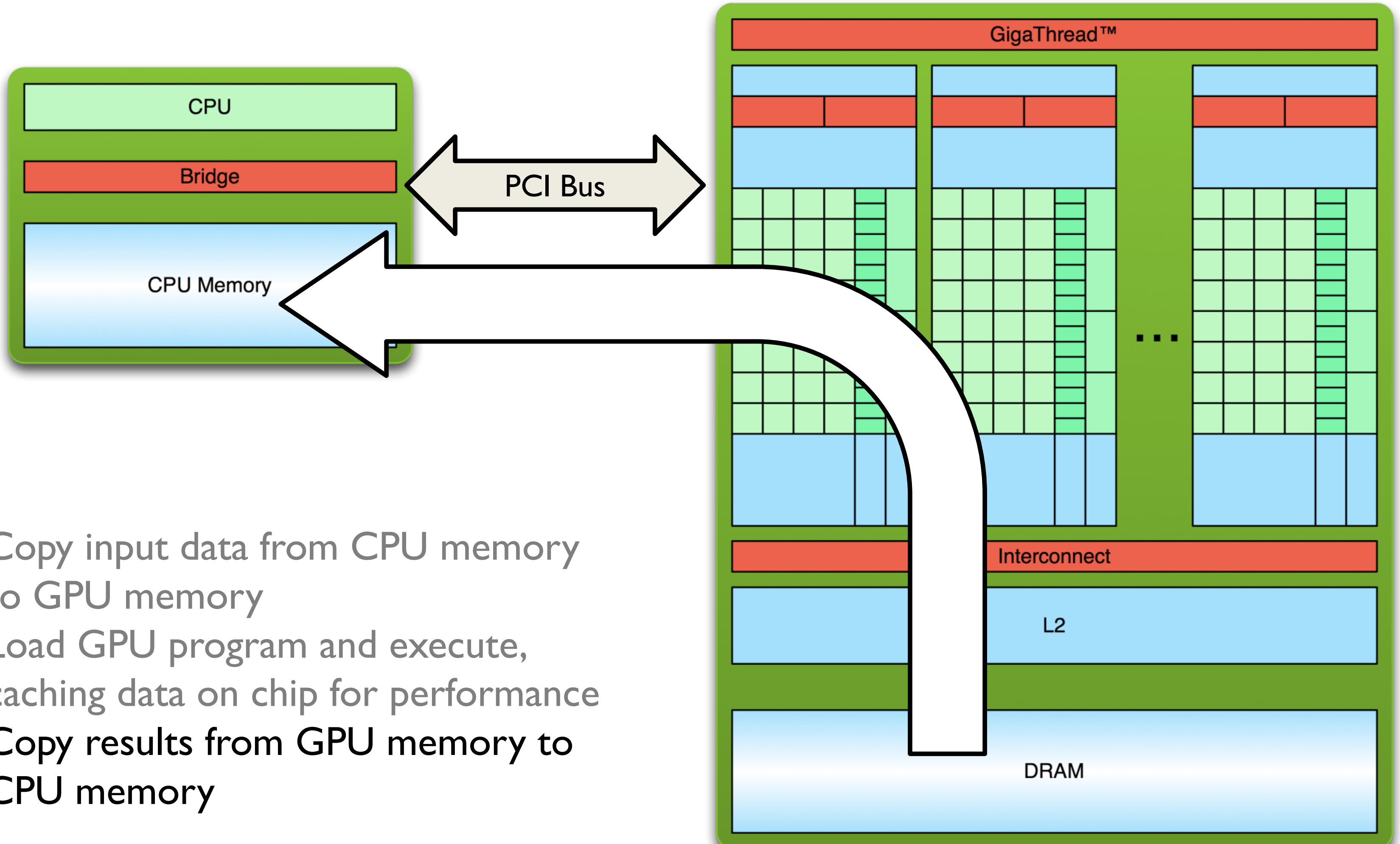
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

SOME EXAMPLE CODE

Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Output:

```
$ nvcc
hello_world.
cu
$ a.out
Hello World!
$
```

- Standard C that runs on the host (the CPU)
- NVIDIA compiler (nvcc)
- We can also write code for the device (the GPU)

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

- Two new things here...

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

Hello World! with Device COde

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll get to the parameters (1,1) soon

Hello World! with Device Code

```
__global__ void mykernel(void){
}

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

Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

- `mykernel()` does nothing at all in this example …. so let’s fix that.

Addition on the Device

- A simple kernel to add two integers (coming up: adding two arrays)

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

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

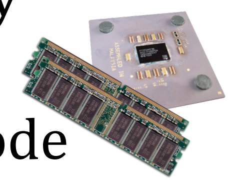
- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code

  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the Device: `add()`

- Returning to our `add()` kernel

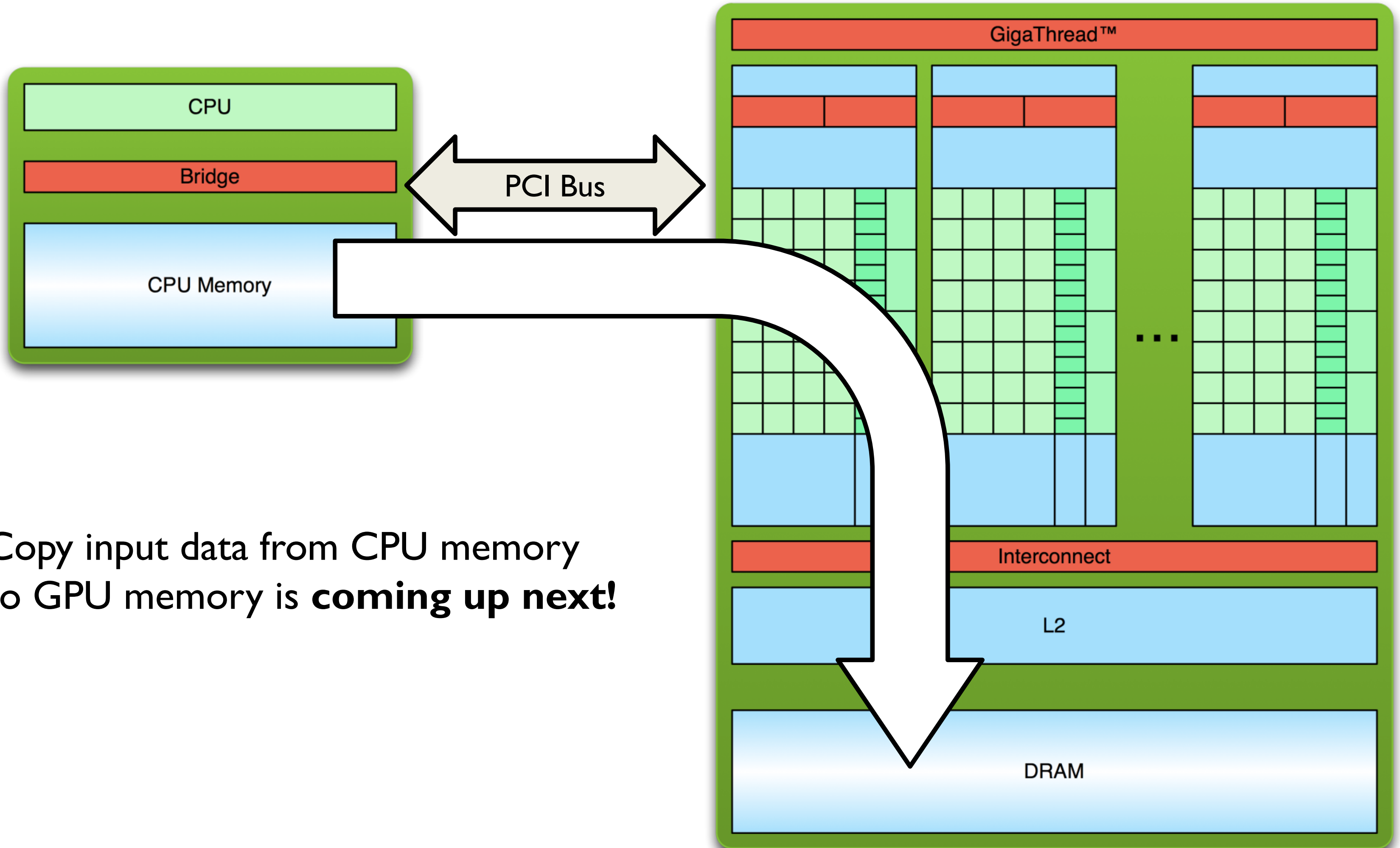
```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- Let's take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
    int a, b, c;                // host copies of a, b, c
    int *d_a, *d_b, *d_c;       // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

We're getting ready to do this…



1. Copy input data from CPU memory to GPU memory is **coming up next!**

Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Next: Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
c[3]  = a[3] + b[3];
```

Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let's take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c                   // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

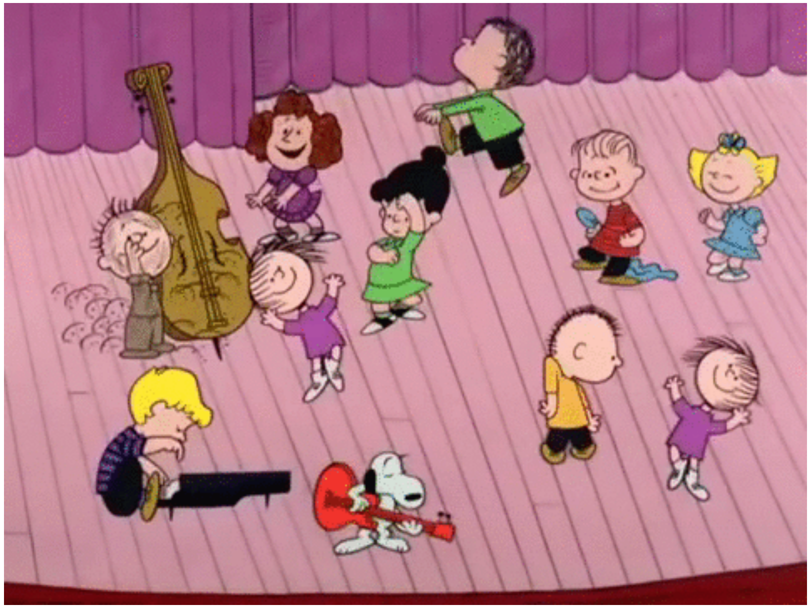
MORE DETAILS ON GPU PROGRAMMING

http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

Summary of GPUs vs CPUs

- less total memory
- more cores and more parallelism

- Multicore CPUs are *mostly* multiple-instruction multiple-data (MIMD)
- GPUs are *mostly* single-instruction multiple-data (SIMD)



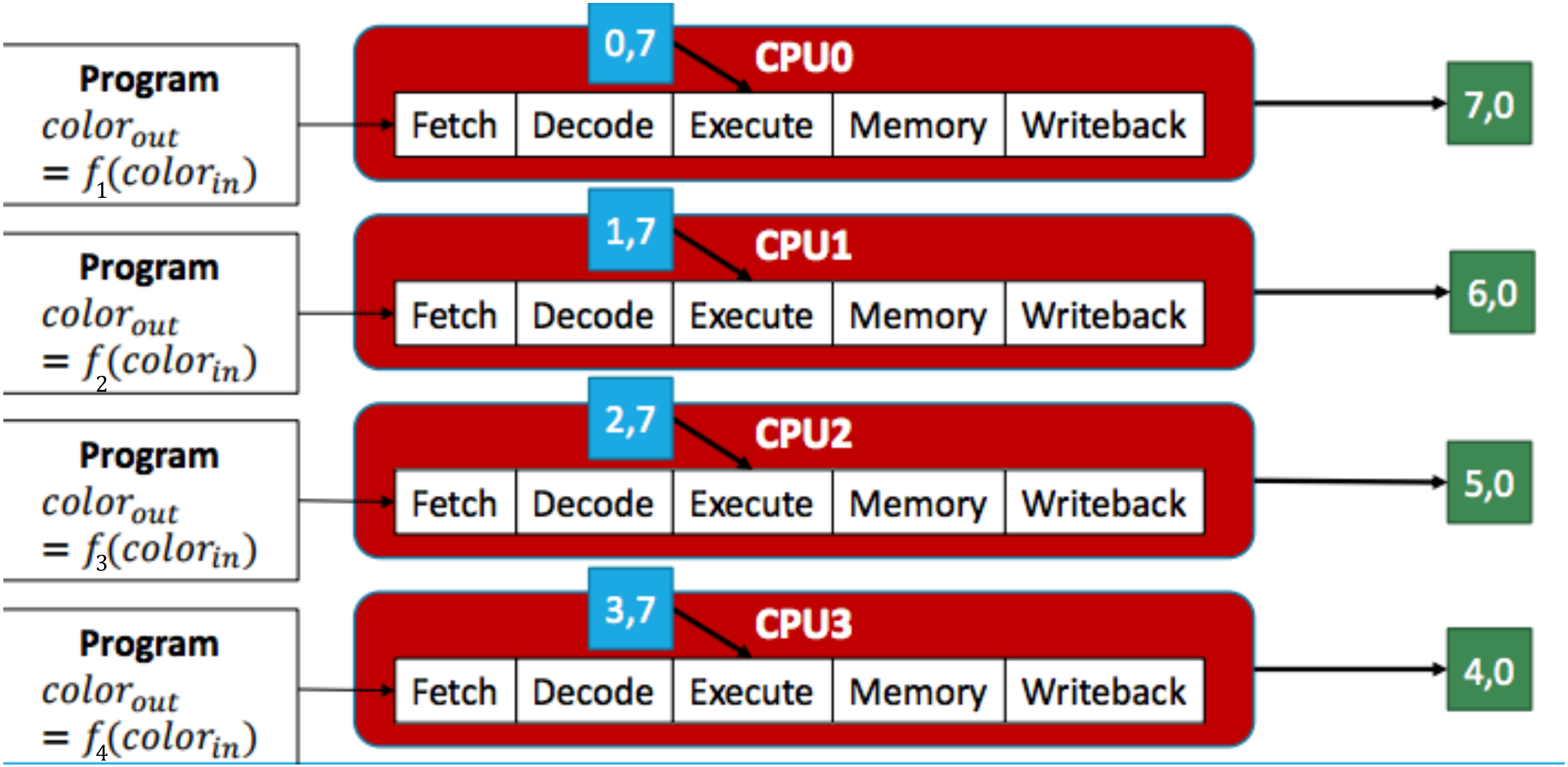
Blocks, Grids, Threads, Warps

- Recall blocks are the things that work in parallel, and blocks are arranged in **grids**

- `c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];`

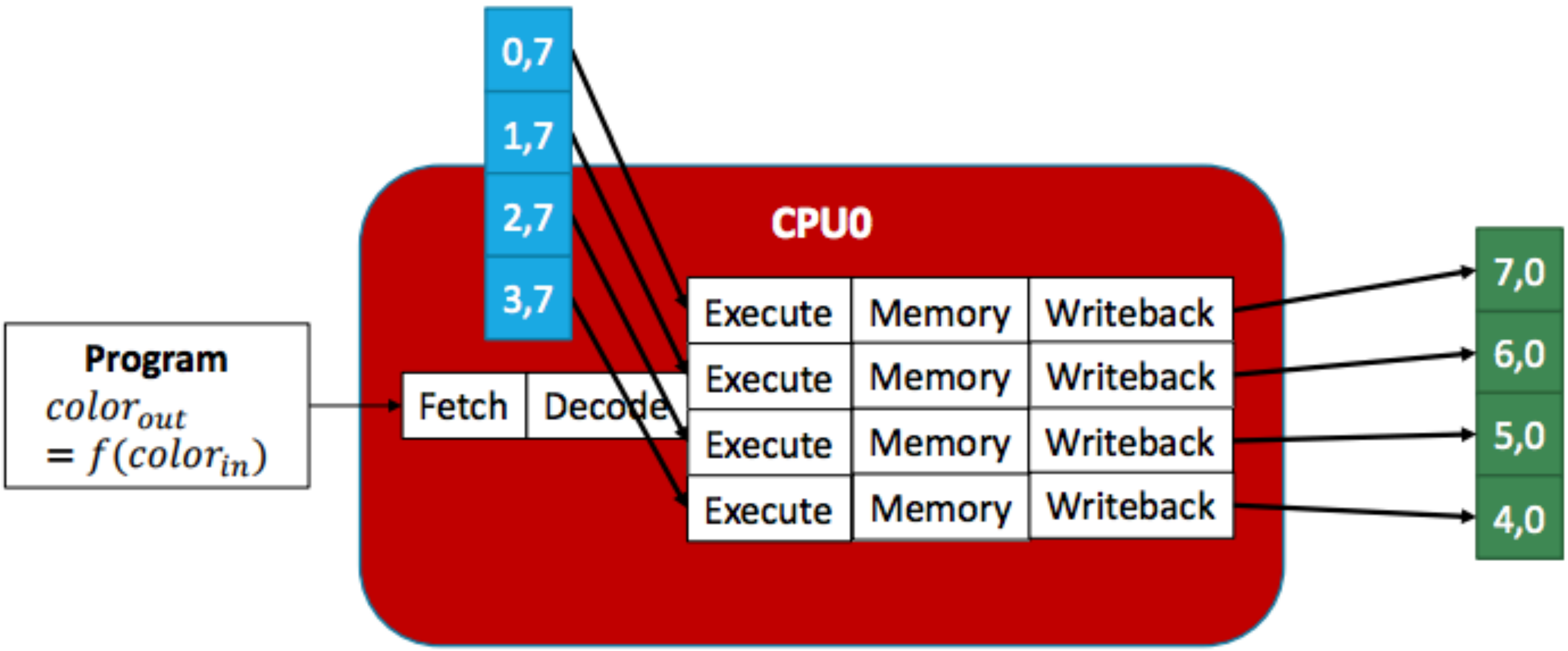
- That would be SIMD (single instruction multiple data)
- It's actually more complicated than that....

MIMD



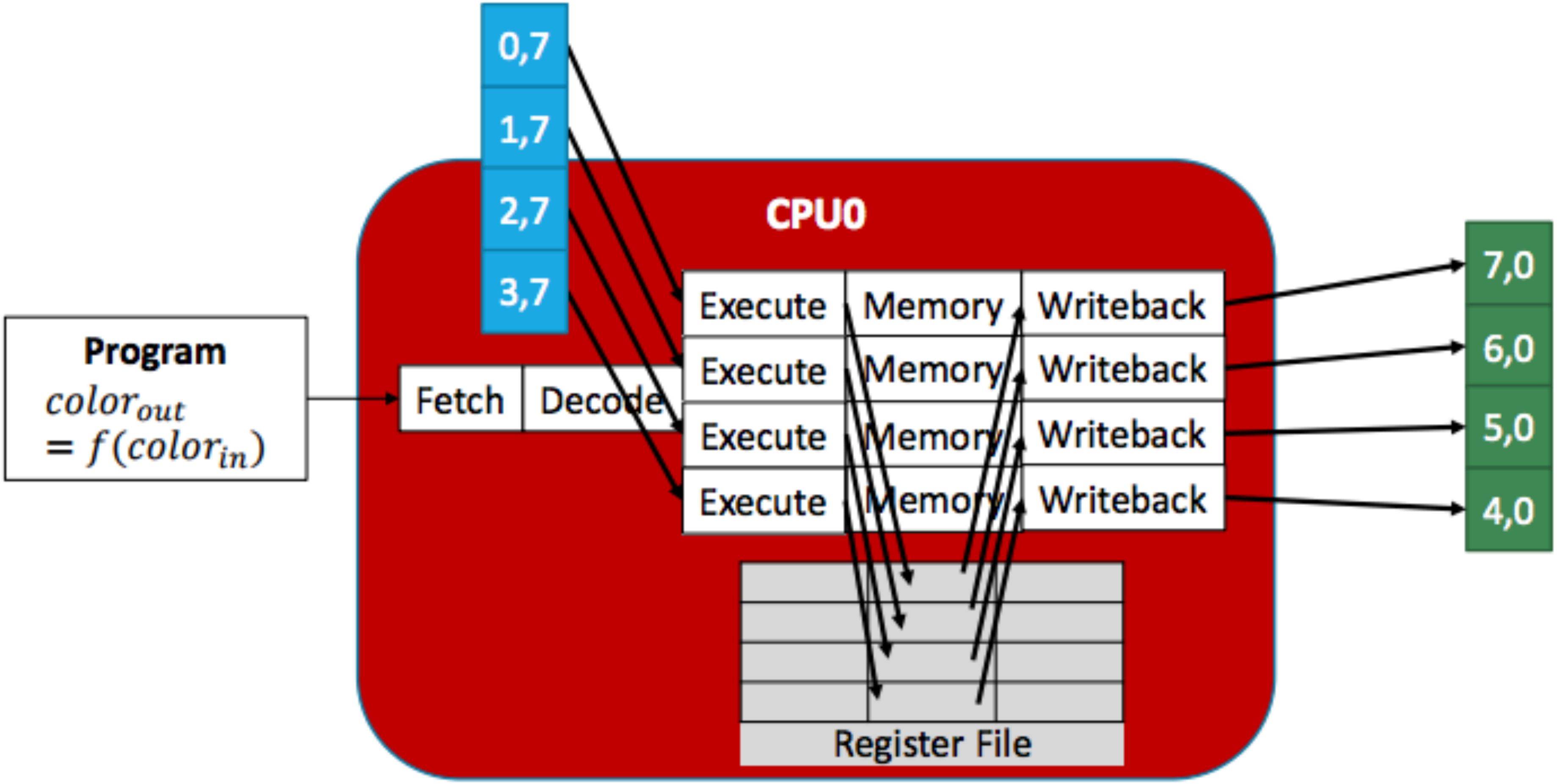
https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

SIMD

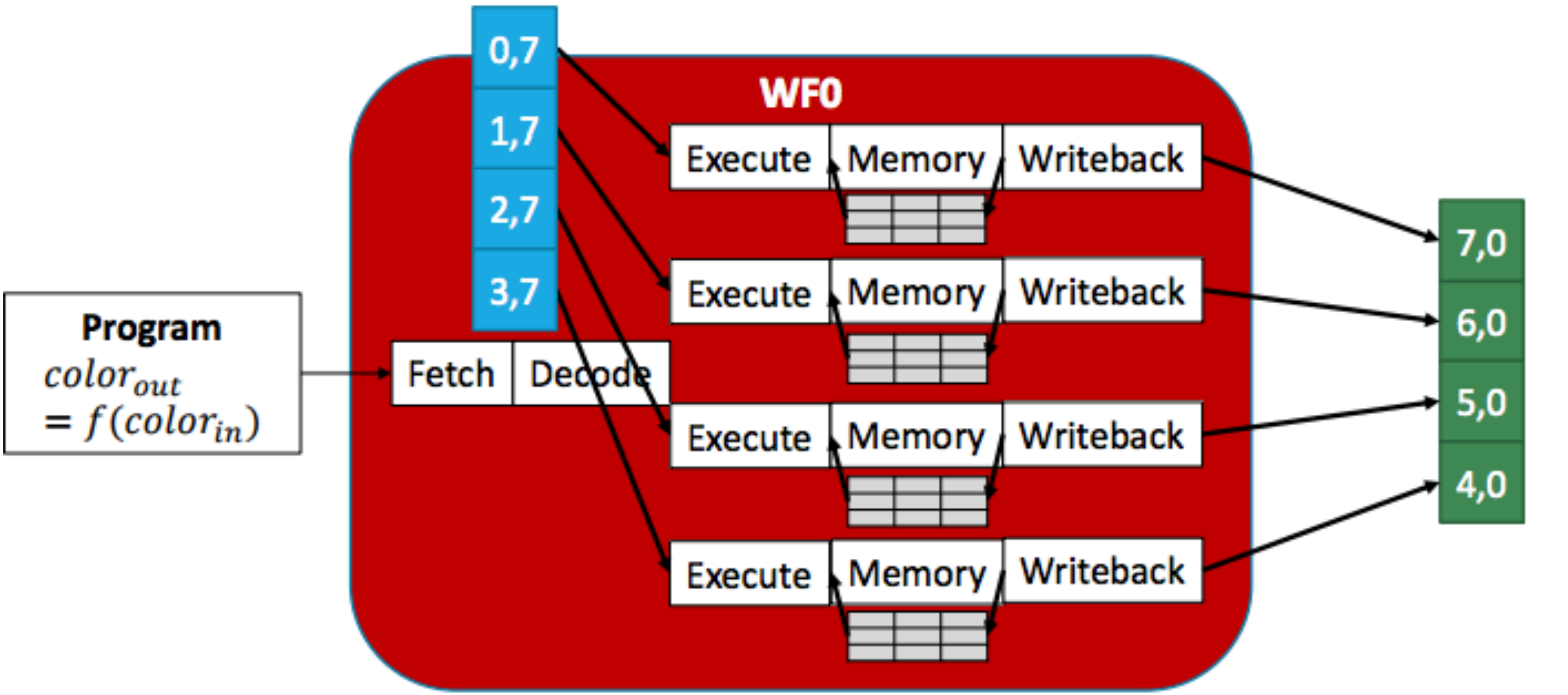


0,7
1,7
2,7
3,7
CPU0
Program
$color_{out} = f(color_{in})$
Fetch
Decode
Execute
Memory
Writeback
Execute
Memory
Writeback
Execute
Memory
Writeback
Execute
Memory
Writeback
7,0
6,0
5,0
4,0

SIMD: Zooming in

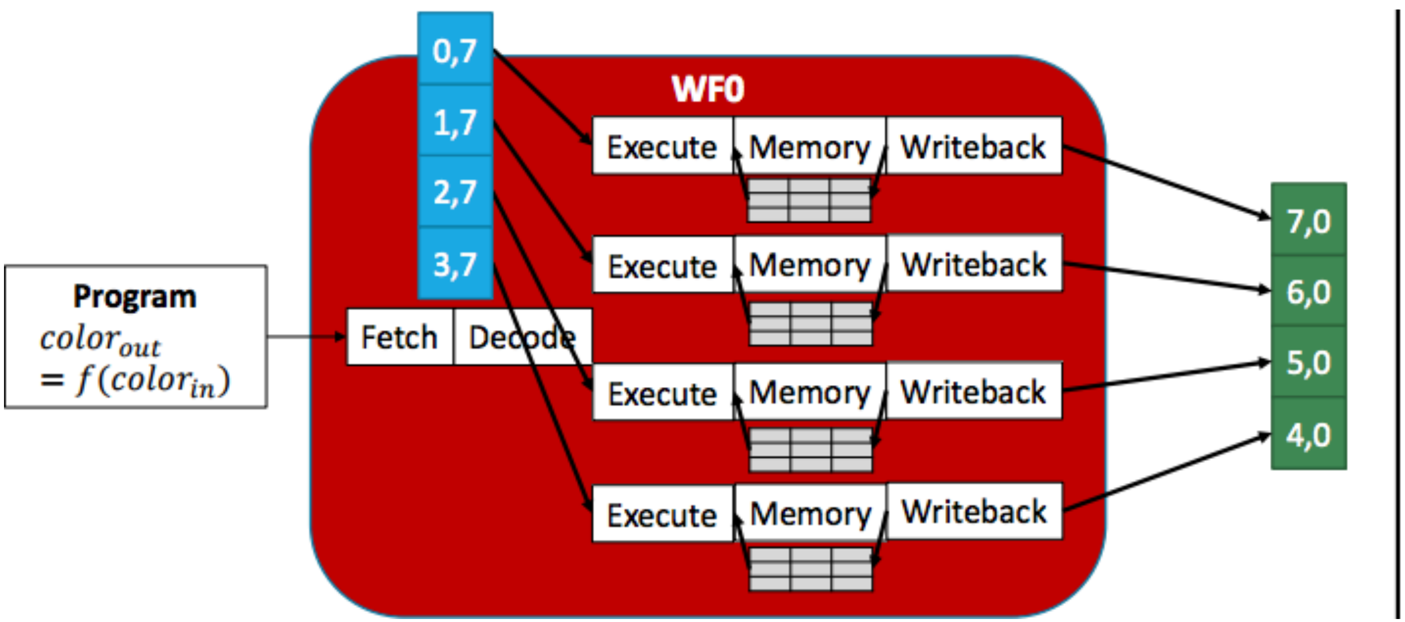


SIMT: Single Instruction Multiple Threads

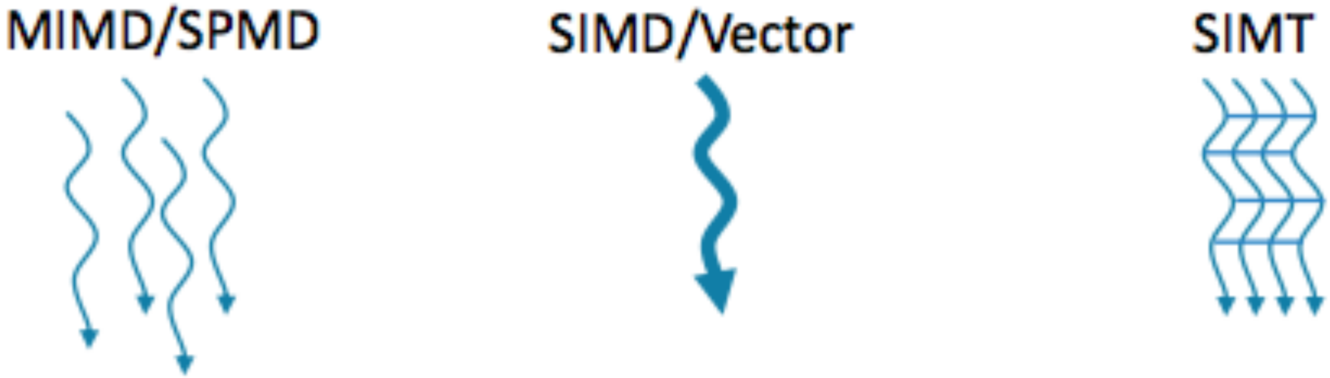


SIMT: Single Instruction Multiple Threads

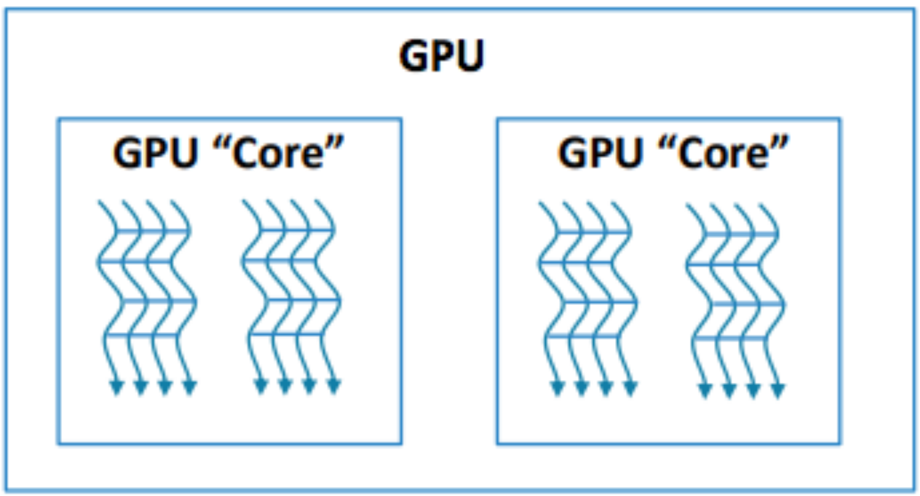
A **thread** can access its own **block id** and also **thread id**. Blocks and threads are in a **grid**, which is 2D or 3D (there's a .x and a .y part)



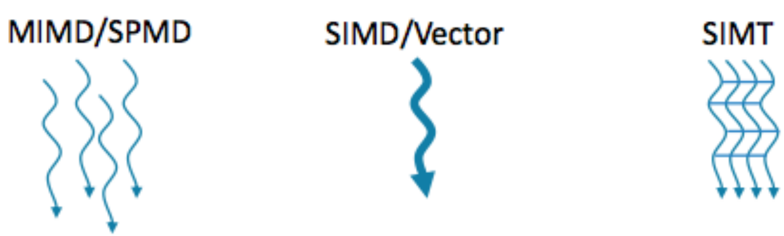
Comparison



What's in a GPU?



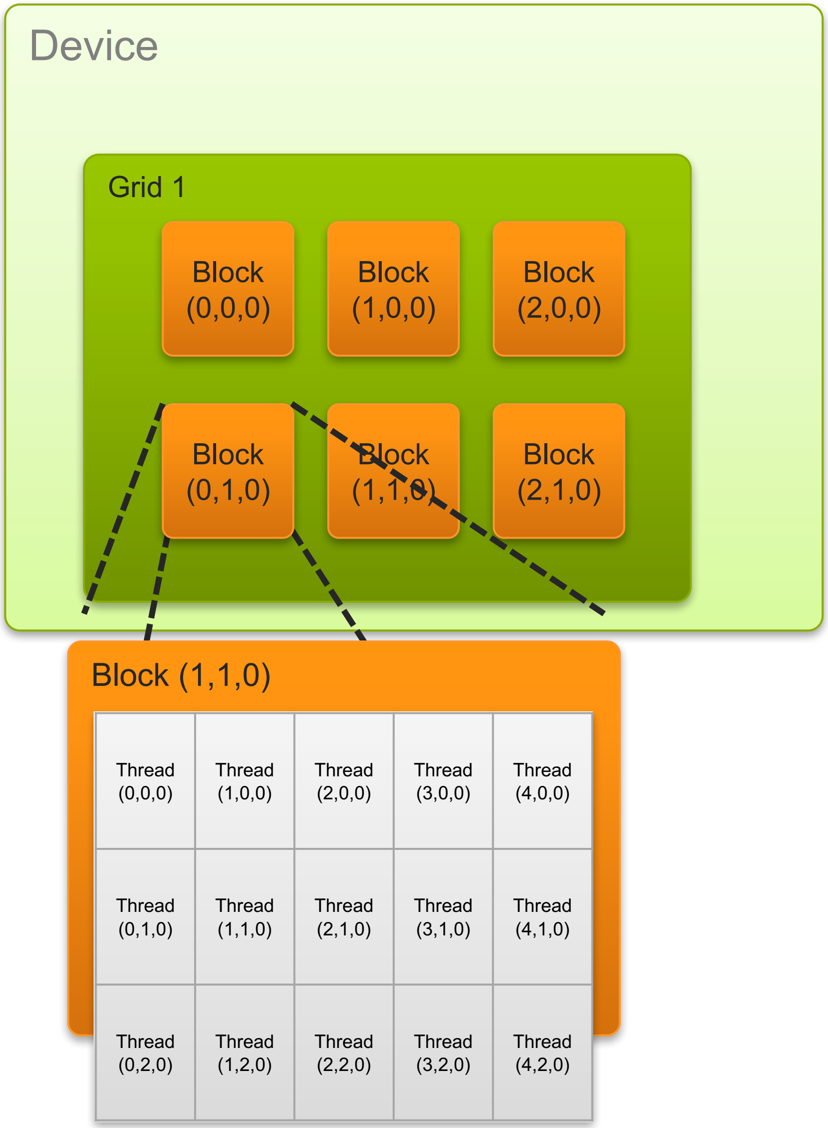
Threads (SIMT, synchronous threads) are grouped into **cores** (which are decoupled, like a MIMD machine)



IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (`x`)

- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



Thread and block parallelism

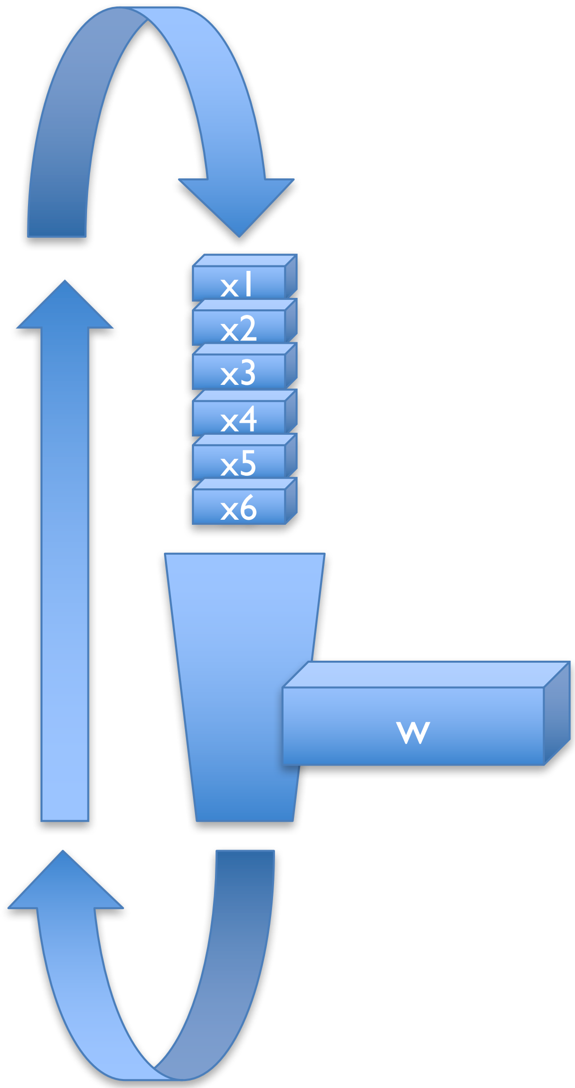
```
tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
bx = cuda.blockIdx.x
by = cuda.blockIdx.y
bw = cuda.blockDim.x
bh = cuda.blockDim.y
x = tx + bx * bw
y = ty + by * bh
array[x, y] = something(x, y)
```

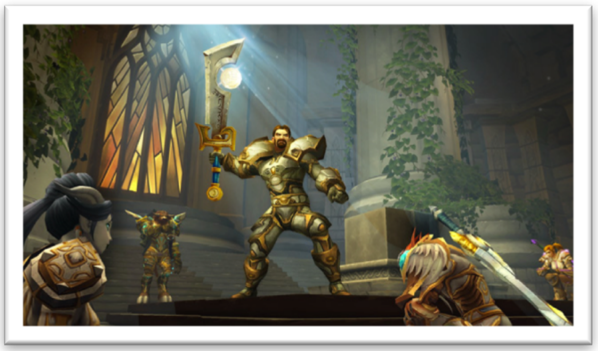
How Do You REALLY Use a GPU?

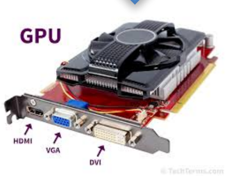
Streaming SGD:

- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly





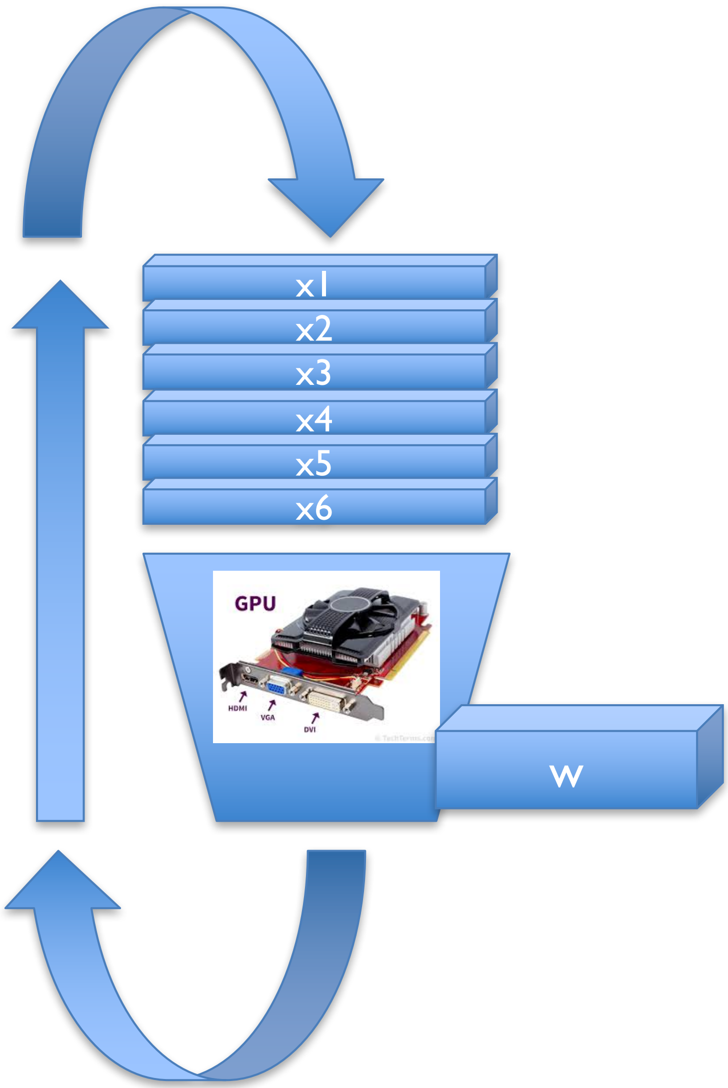


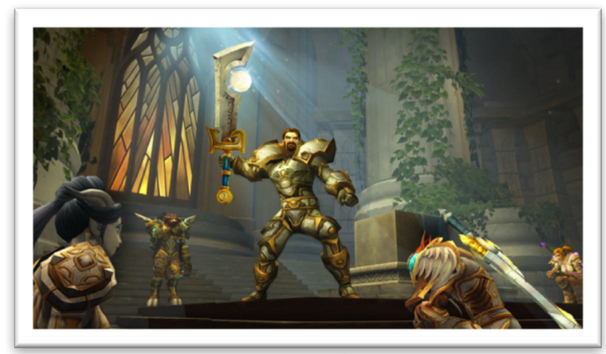
Streaming SGD:

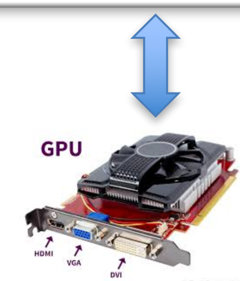
- Iterative
- Sequential
- Fast
- Scale up by bounding memory

- You can handle very large datasets … but slowly

- You can speed it up by making the tasks in the stream bigger and doing them in **parallel**
- A GPU is a good way of doing that







http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

So this will run in parallel on a GPU?

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

No: we're not there yet….

So this will run in parallel on a GPU? Not yet….

http://minpy.readthedocs.io/en/latest/get-started/logistic_regression.html

```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```

Option 1: switch from numpy (old package) to cupy (new GPU-oriented package)

Option 2: switch to a package that will compile to a GPU and also compute the gradients for you (like Theano, Tensorflow, Torch, ...)