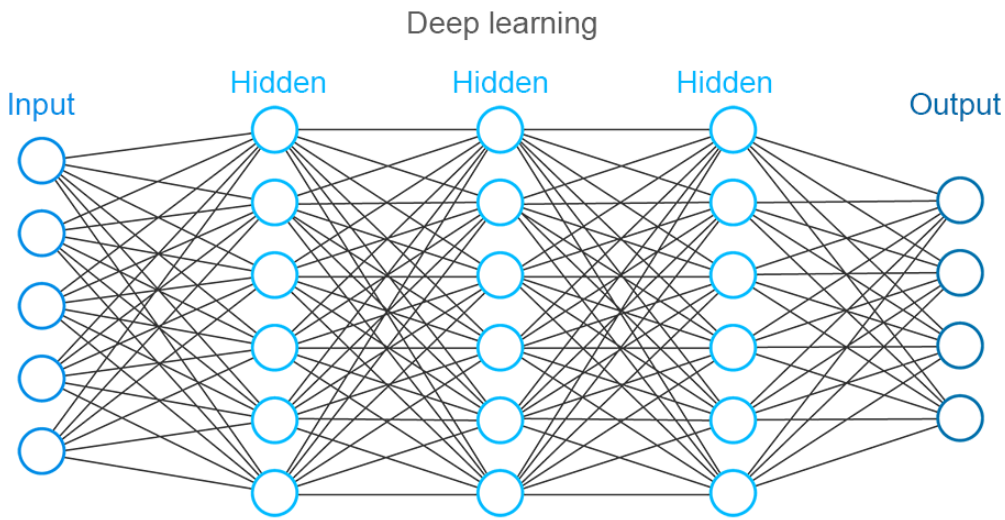
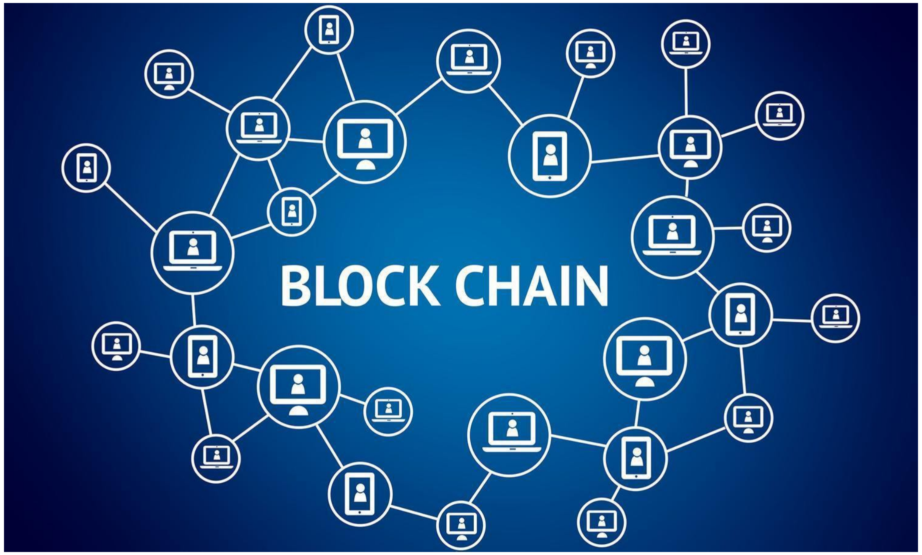
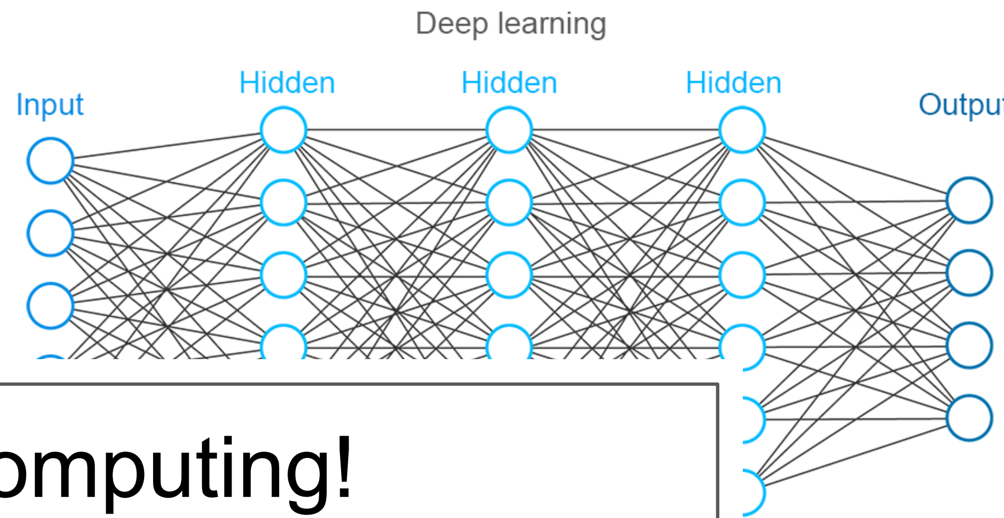
Parallel Programming Language CUDA (C extension)*

Jiwon Seo

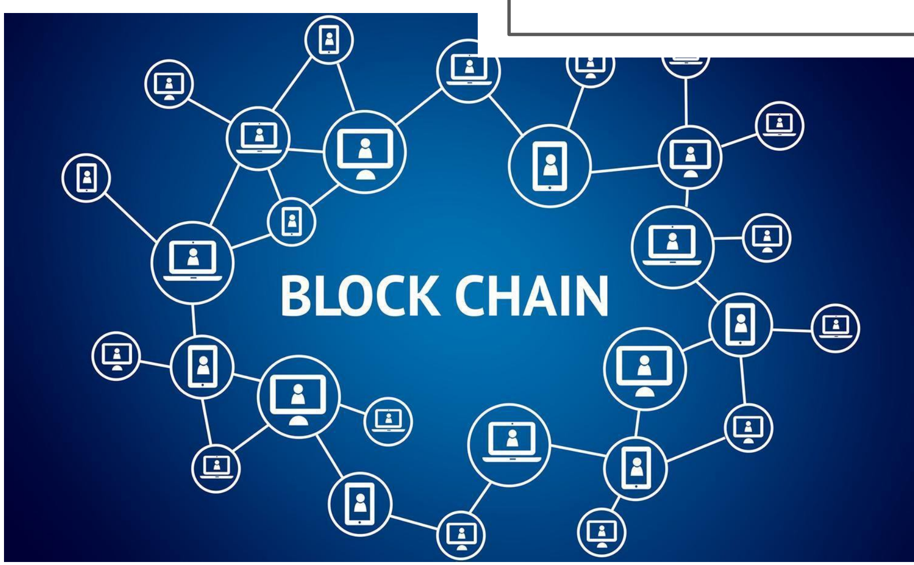
*Slides are based on Caltech CS179 course


Deep learning
Input
Hidden
Hidden
Hidden
Output

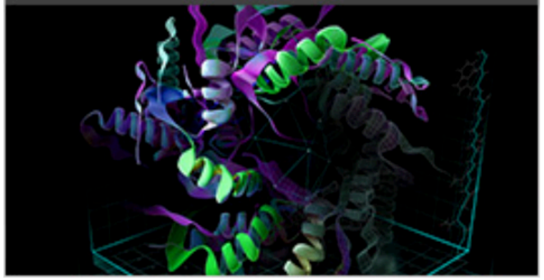

BLOCK CHAIN



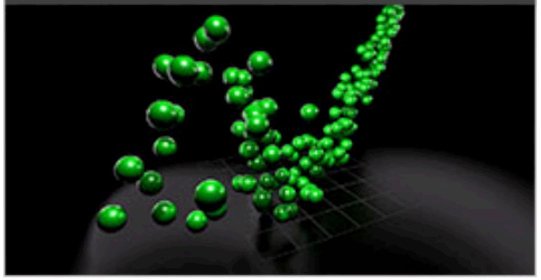
GPU Computing!



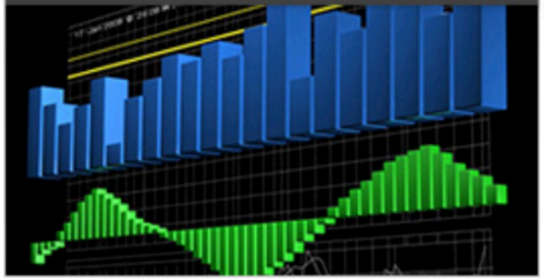
BIOINFORMATICS



COMPUATIONAL CHEMISTRY

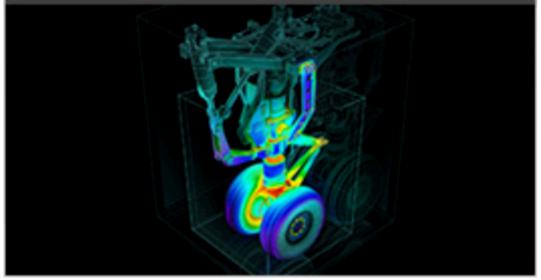


COMPUTATIONAL FINANCE



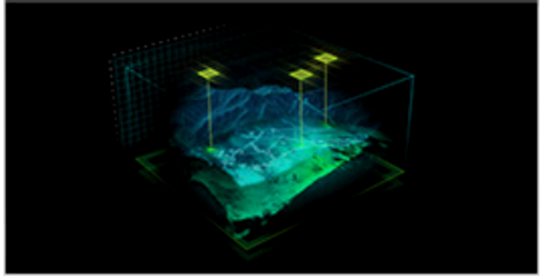
COMPUTATIONAL FLUID DYNAMICS

COMPUTATIONAL STRUCTURAL MECHANICS

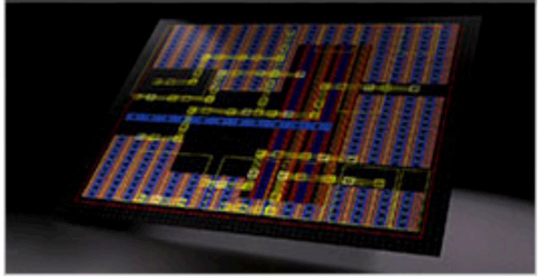


DATA SCIENCE

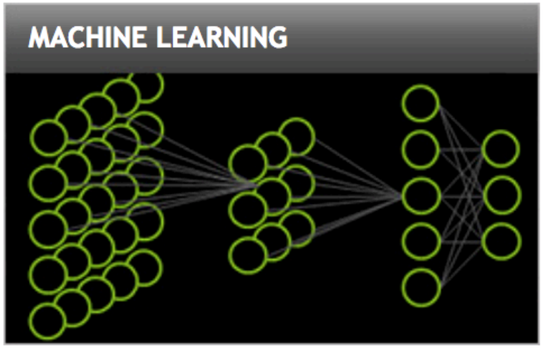
DEFENSE

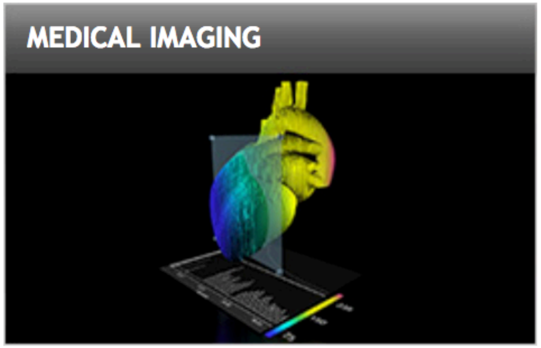


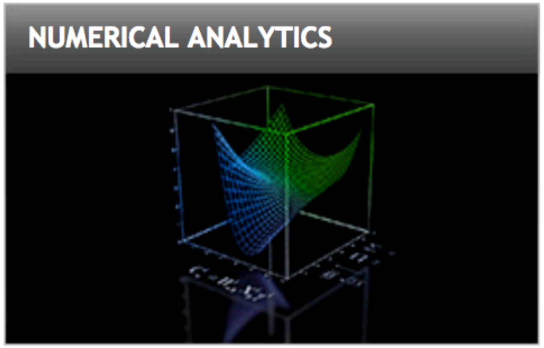
ELECTRIC DESIGN AUTOMATION

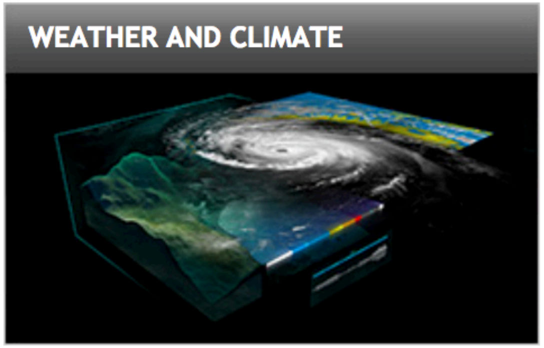


IMAGING & COMPUTER VISION


MACHINE LEARNING


MEDICAL IMAGING


NUMERICAL ANALYTICS


WEATHER AND CLIMATE

Programming/Running Environment

Cloud GPU machines (ChunDoong,천둥)

- Accounts created for you:
  - One account for two students (check course homepage)
  - Limited resource, but enough for your homework
- See course homepage and http://chundoong.snu.ac.kr/ for more

Change your password from the temp one we made

- Use *passwd* command

Programming/Running Environment

Cloud GPU machines (for interactive testing)

- Google Colab
- We will give a short tutorial next week

Programming/Running Environment

Alternative: Use your own machine:

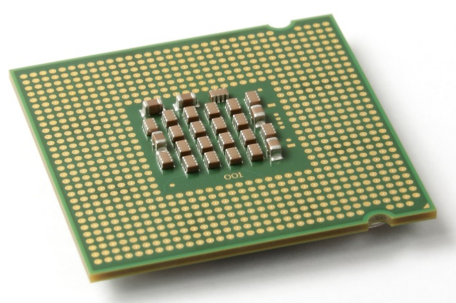
- Must have an NVIDIA CUDA-capable GPU
- Virtual machines won’t work
- You need to install NVIDIA dev tools

The CPU

The “Central Processing Unit”

Traditionally, applications primarily use CPU

- General-purpose capabilities
- Usually equipped with 4~16 powerful cores
- Optimal for concurrent processes but not large scale parallel computations



Wikimedia commons: Intel_CPU_Pentium_4_640_Prescott_bottom.jpg

The GPU

The "Graphics Processing Unit"

Relatively new technology designed for parallelizable problems

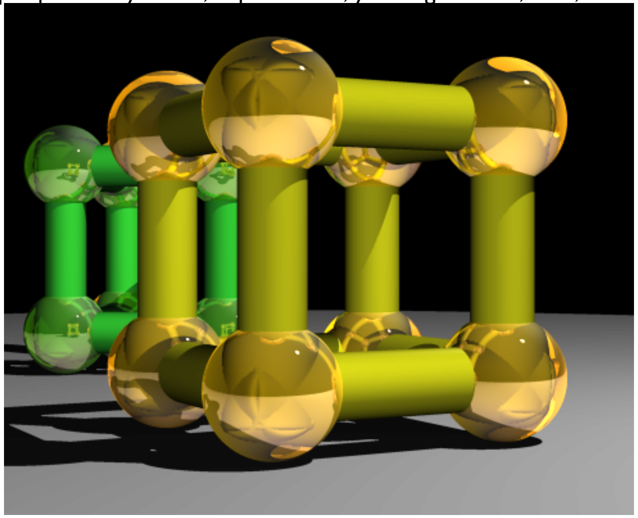
- Initially created specifically for graphics
- Became more capable of general computations
- Particularly well applied in machine learning (deep learning)

GPUs – The Motivation

Raytracing:

```
for all pixels (i,j):
    Calculate ray point and direction in 3d space
    if ray intersects object:
      calculate lighting at closest object
      store color of (i,j)
```

Superquadric Cylinders, exponent 0.1, yellow glass balls, Barr, 1981



More Examples

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];
return C;
```

This operates sequentially… can we do better?

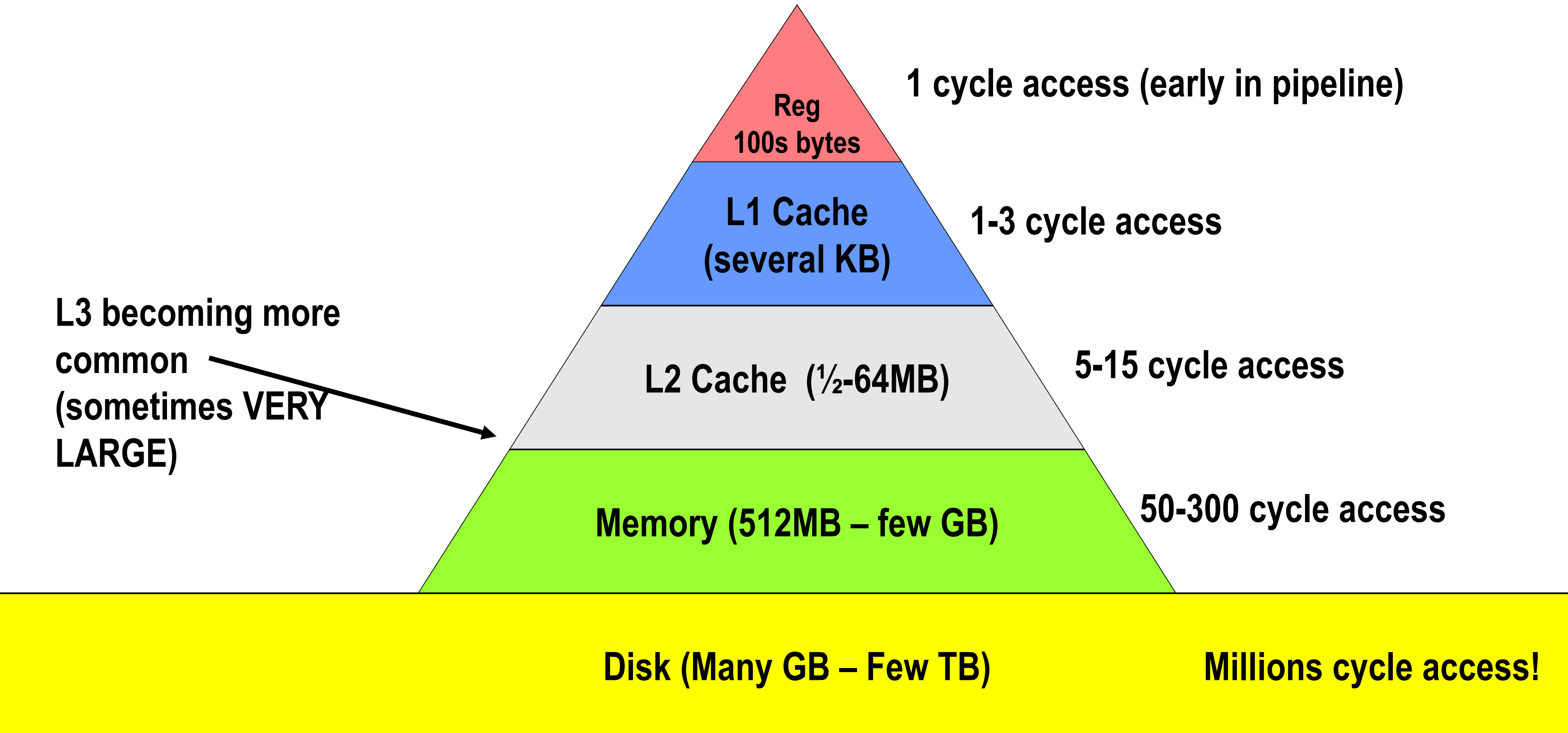
A simple problem…

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

```
1.(allocate memory for C)
2.Create # of threads equal to number of cores on processor
    (around 4, 8, perhaps 16)
    (Indicate portions of A, B, C to each thread...)
...
3. 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...
```

This is slightly faster – 4-16x (slightly more with other tricks)

Memory Hierarchy



These are rough numbers: mileage may vary for latest/greatest

Caches USUALLY made of SRAM

More on Accessing Times

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

From Jeff Dean's Slide (2010)

A simple problem…

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

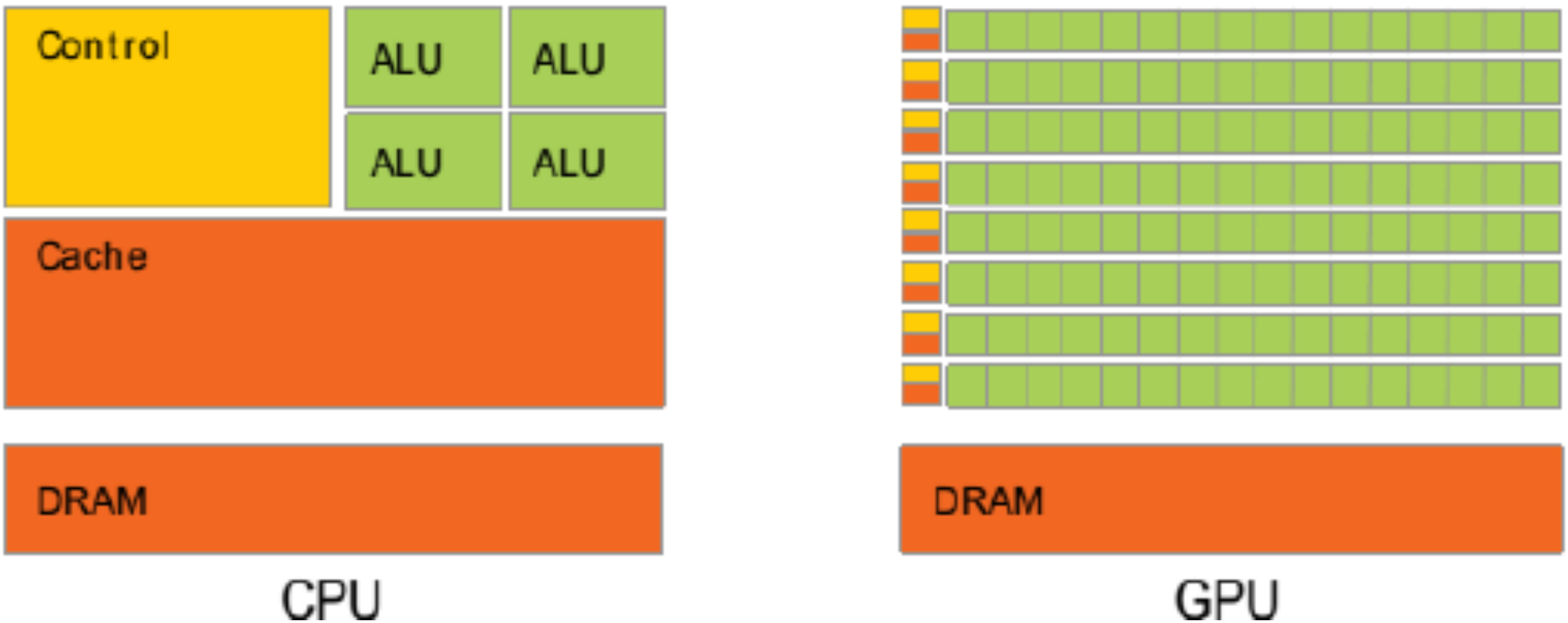
```
1.(allocate memory for C)
2.Create # of threads equal to number of cores on processor
    (around 4, 8, perhaps 16)
    (Indicate portions of A, B, C to each thread...)
...
3. In each thread,
   For (i from beginning region of thread)
       C[i] <- A[i] + B[i]
   // lots of waiting involved for memory reads, writes, ...
      Wait for threads to synchronize...
```

This is slightly faster – 4-16x (slightly more with other tricks)

A simple problem…

- How many threads? How does performance scale?
- Context switching:
  - The action of switching which thread is being processed
  - High penalty on the CPU
  - Not an issue on the GPU

A simple problem…



A simple problem…

- On the GPU:

1. Allocate memory for A, B, C on GPU
2. Create the "kernel" – each thread will perform one (or a few) additions

```
   Specify the following kernel operation:
     For all i's (indices) assigned to this thread:
         C[i] <- A[i] + B[i]
```

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

GPU: Strengths Revealed

- Emphasis on parallelism means we have lots of cores
- This allows us to run many threads simultaneously with no context switches
- In addition, GPU's context switching overhead is cheaper

GPU Programming – First Try

Based on https://devblogs.nvidia.com/even-easier-introduction-cuda/

GPU Programming – First Try

CPU Code first:

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];
```

GPU Programming – First Try

```
// initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  // Run add function on 1M elements on the CPU
  add(N, x, y);
  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;}
```

GPU Programming – First Try

CUDA Programs

1. CUDA Programming Interfaces (Library)
2. Extended Syntax

GPU Programming – First Try

STEP 1. Add function needs to run on GPU:

```
// CUDA Kernel function to add the elements of two arrays
on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

GPU Programming – First Try

STEP 2. Memory allocation on GPU:

```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

GPU Programming – First Try

STEP 3. Launch add() kernel on GPU

```
add<<<1, 1>>>(N, x, y);
```

GPU Programming – First Try

STEP 3a. Launch add() kernel on GPU

```
add<<<1, 1>>>(N, x, y);
```

GPU Programming – First Try

STEP 3b. Wait until add() kernel finishes

```
add<<<1, 1>>>(N, x, y);

cudaDeviceSynchronize();
```

GPU Programming – First Try

Compile with NVCC and run!

```
$ nvcc add.cu -o add_cuda
$ ./add_cuda
Max error: 0.00000
```

GPU Programming – First Try

Compile with NVCC and run!

```
$ nvcc add.cu -o add_cuda
$ ./add_cuda
Max error: 0.00000
```

Profile with nvprof

```
$ nvprof ./add_cuda
```

```
$ nvprof ./add_cuda
==3355== NVPROF is profiling process 3355, command: ./add_cuda
Max error: 0
==3355== Profiling application: ./add_cuda
==3355== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  463.25ms         1  463.25ms  463.25ms  463.25ms  add(int, float*, float*)
...
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

In add() kernel, 256 threads do the same computation

```
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

Let 256 threads do different computation

```
__global__
void add(int n, float *x, float *y) {
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

From 1 Thread to 256 Threads

Launching add() kernel with 256 threads

```
add<<<1, 256>>>(N, x, y);
```

nvprof for profiling

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	2.7107ms	1	2.7107ms	2.7107ms	2.7107ms	add(int, float*, float*)

463ms → 2.7ms : 200X speedup!

From 1 Block to multiple Blocks

First parameter for kernel launch

```
add<<<1, 256>>>(N, x, y);
```

Number of thread blocks

From 1 Block to multiple Blocks

First parameter for kernel launch

```
add<<<1, 256>>>(N, x, y);
```

→ Multiple thread blocks

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

From 1 Block to multiple Blocks

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

Need to update kernel function

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

From 1 Block to multiple Blocks

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

nvprof for profiling

Time(%)	Time	Calls	Avg	Min	Max
100.00%	94.015us	1	94.015us	94.015us	94.015us

463ms → 2.7ms → 94us : 4000X speedup!