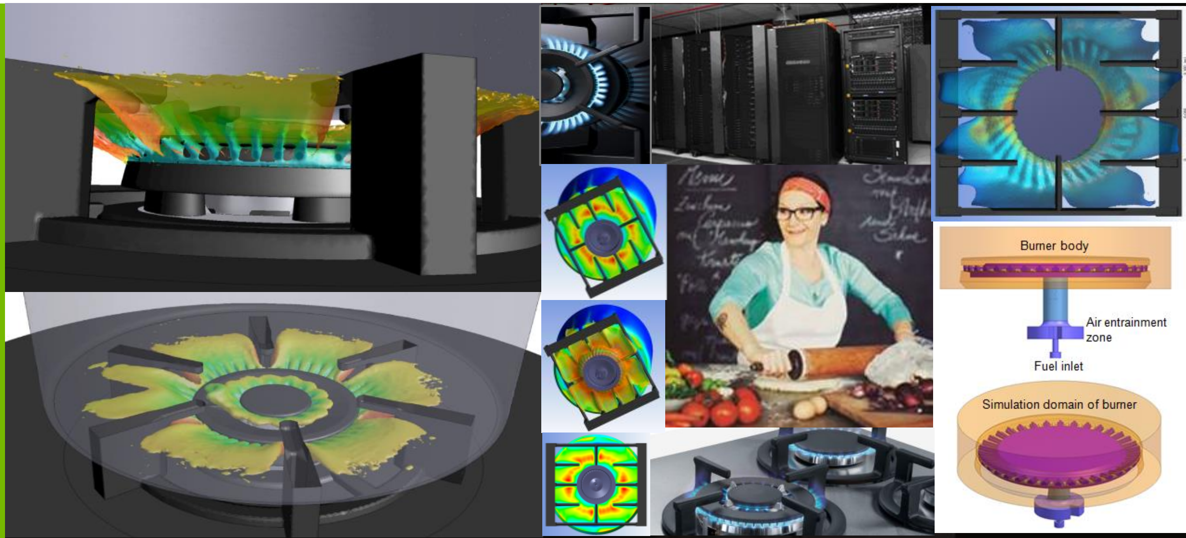
Introduction to GPU Computing

SAN/ Simulations
Adriana Cavada

Product Division Cooking - Europe
Predevelopment Gas Cooktops
REU/OP-CGDCPG
Santander, Spain

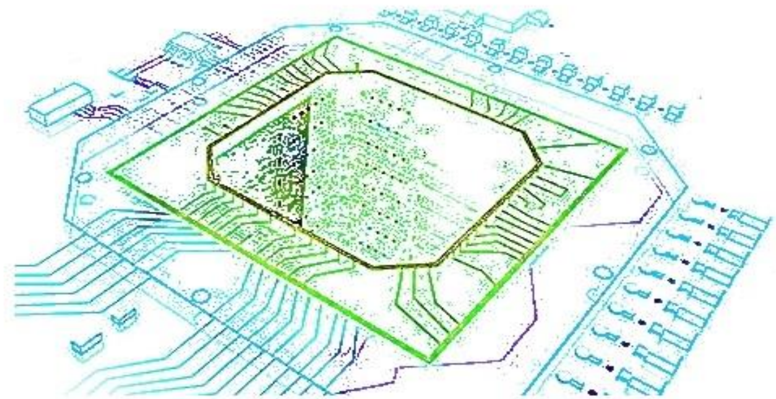
June, 2016



WE.LIVE.COOKING.

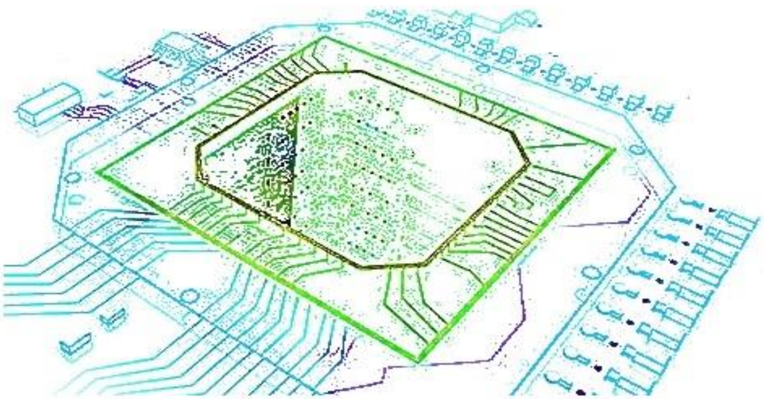
What is GPU?

GPU = Graphics Processor Unit



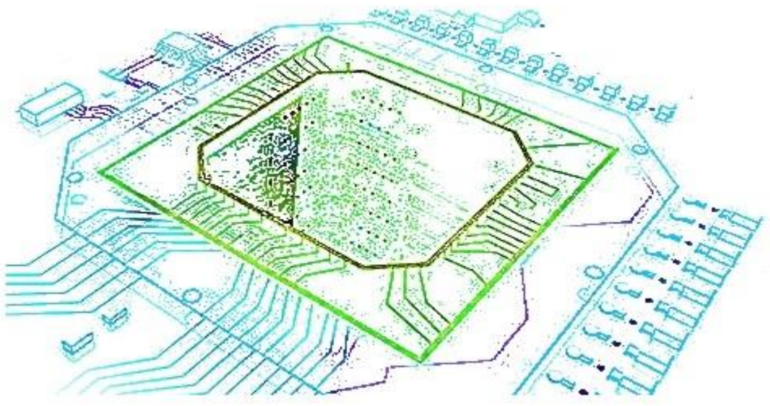
Computer chip mainly designed for image processing and real-time rendering

GPU = Graphics Processor Unit



Parallel processor mainly designed for
image processing and real-time rendering

GPU = Graphics Processor Unit

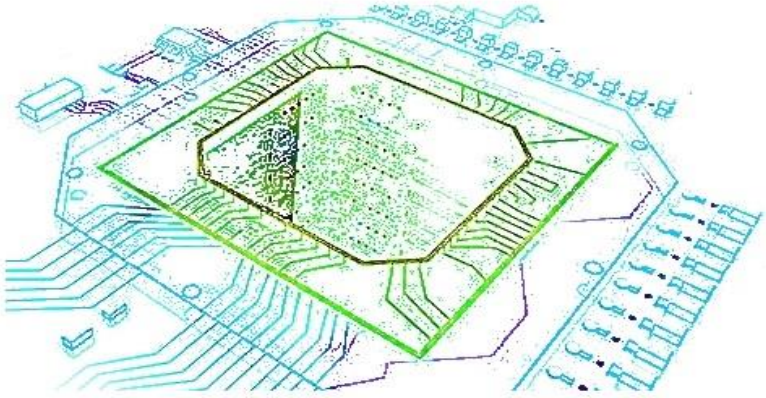


Parallel processor mainly designed for image processing and real-time rendering

"THE WORLD'S FIRST GPU"

NVIDIA GeForce 256
(1999)

GPU = Graphics Processor Unit

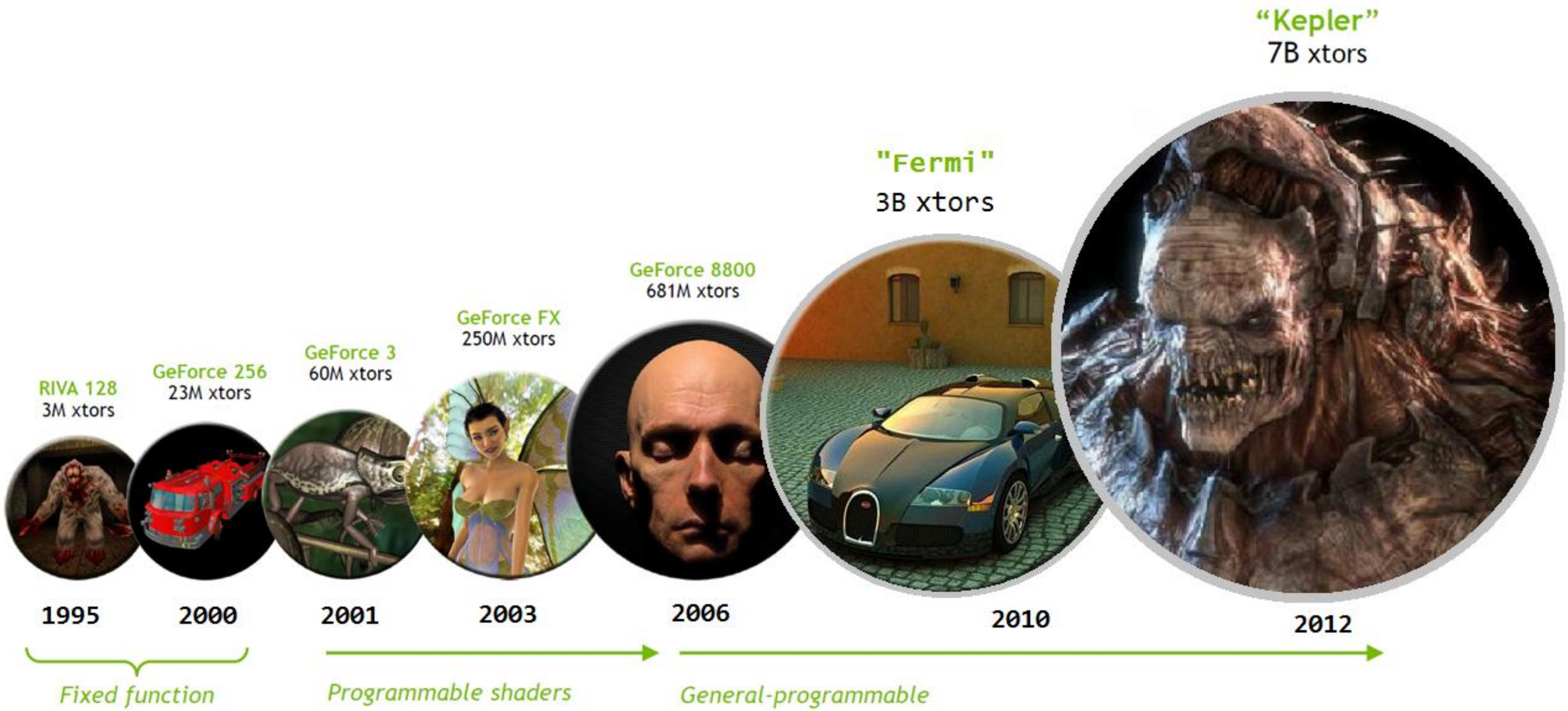


Parallel processor mainly designed for image processing and real-time rendering

"THE WORLD'S FIRST GPU"

NVIDIA GeForce 256
(1999)

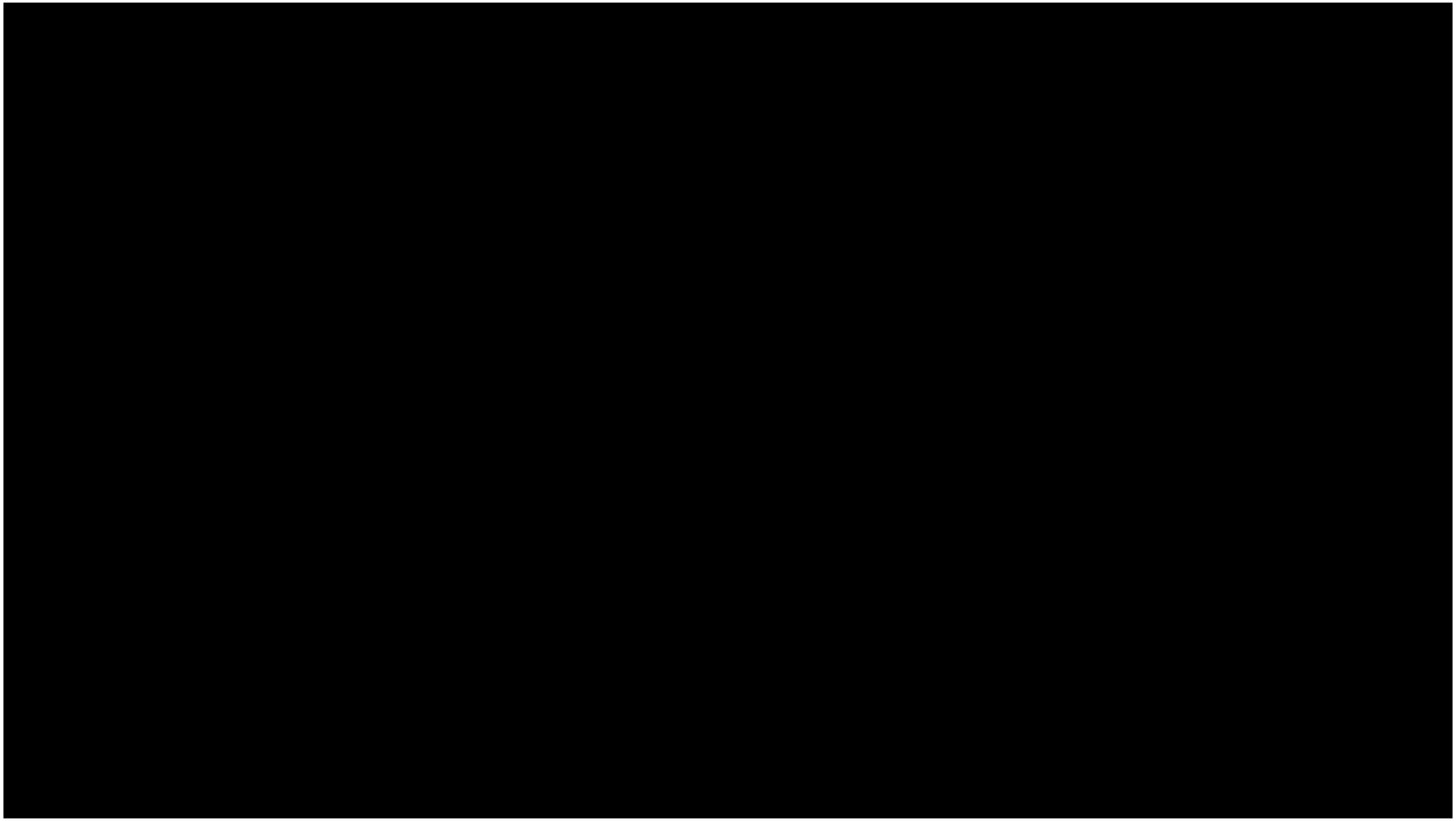
Evolution



Ref: NVIDIA Corporation, *GPU Hardware and Programming Models*

CPU vs GPU

CPU vs GPU



The Mythbusters, Adam Savage and Jamie Hyneman demonstrate the power of GPUs vs CPUs - NVIDIA Corporation

What’s better?

What’s better?

What's better?

Deliver a package as soon as possible?

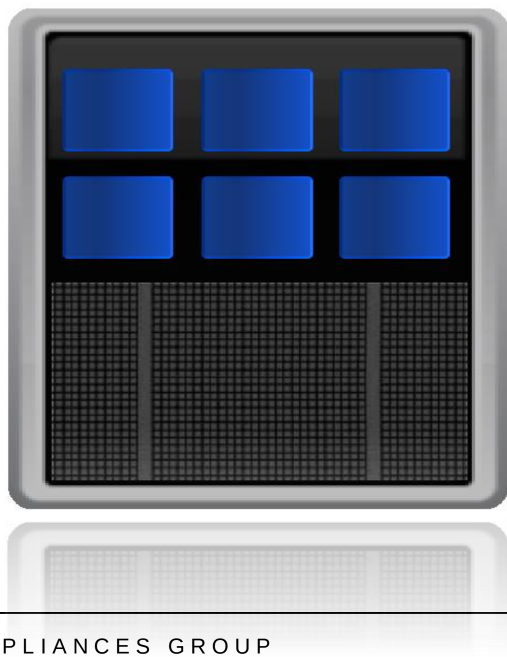
Or deliver many packages within a reasonable timescale?

CPU vs GPU

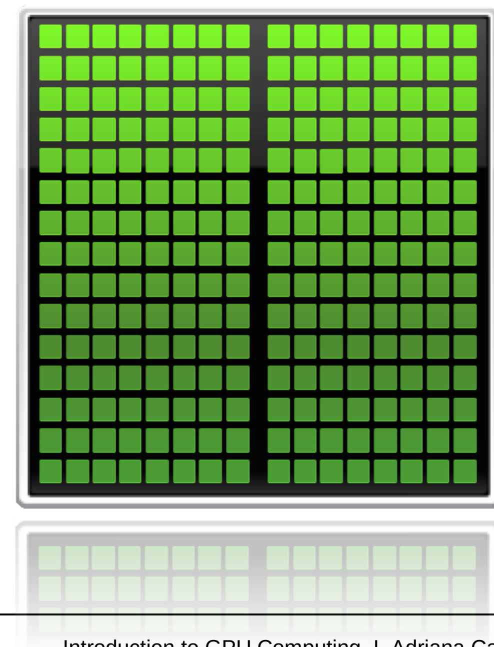
Compute a job as fast as possible

Compute many jobs within a reasonable timeframe

CPU



GPU

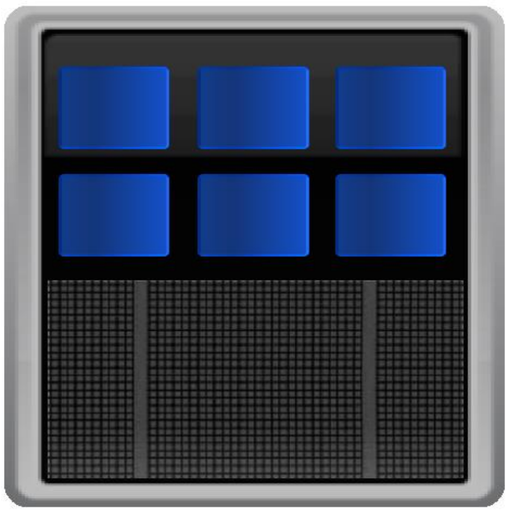


CPU vs GPU

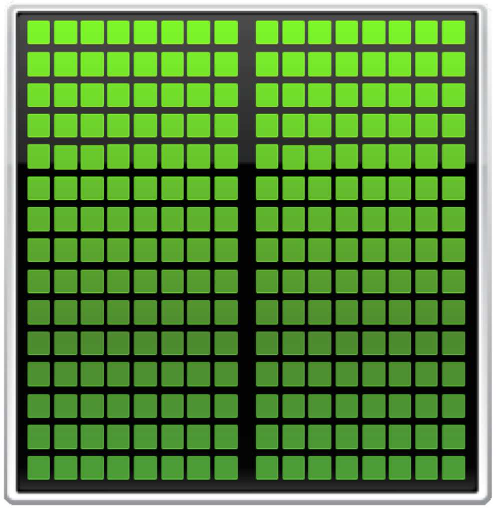
Low Latency

High Throughput

CPU



GPU

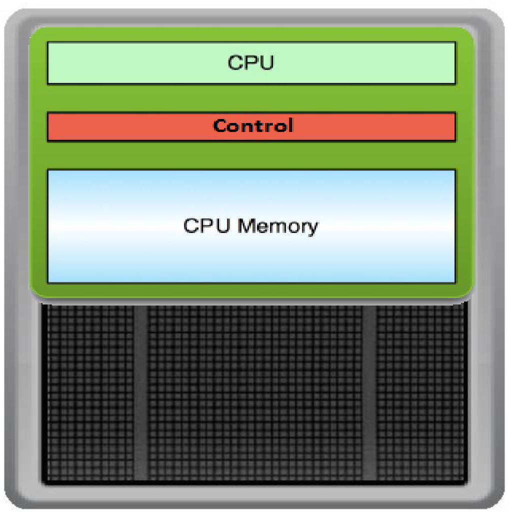


CPU vs GPU

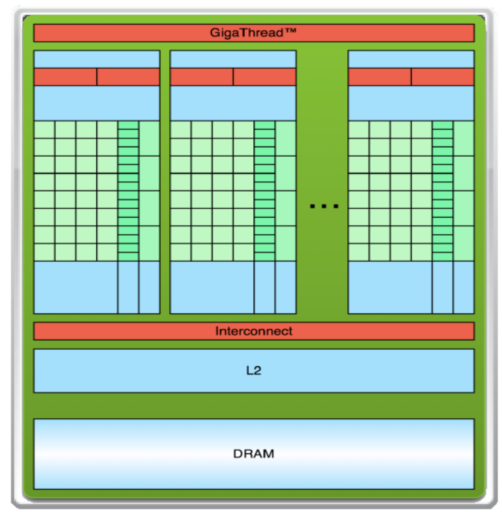
Low Latency

High Throughput

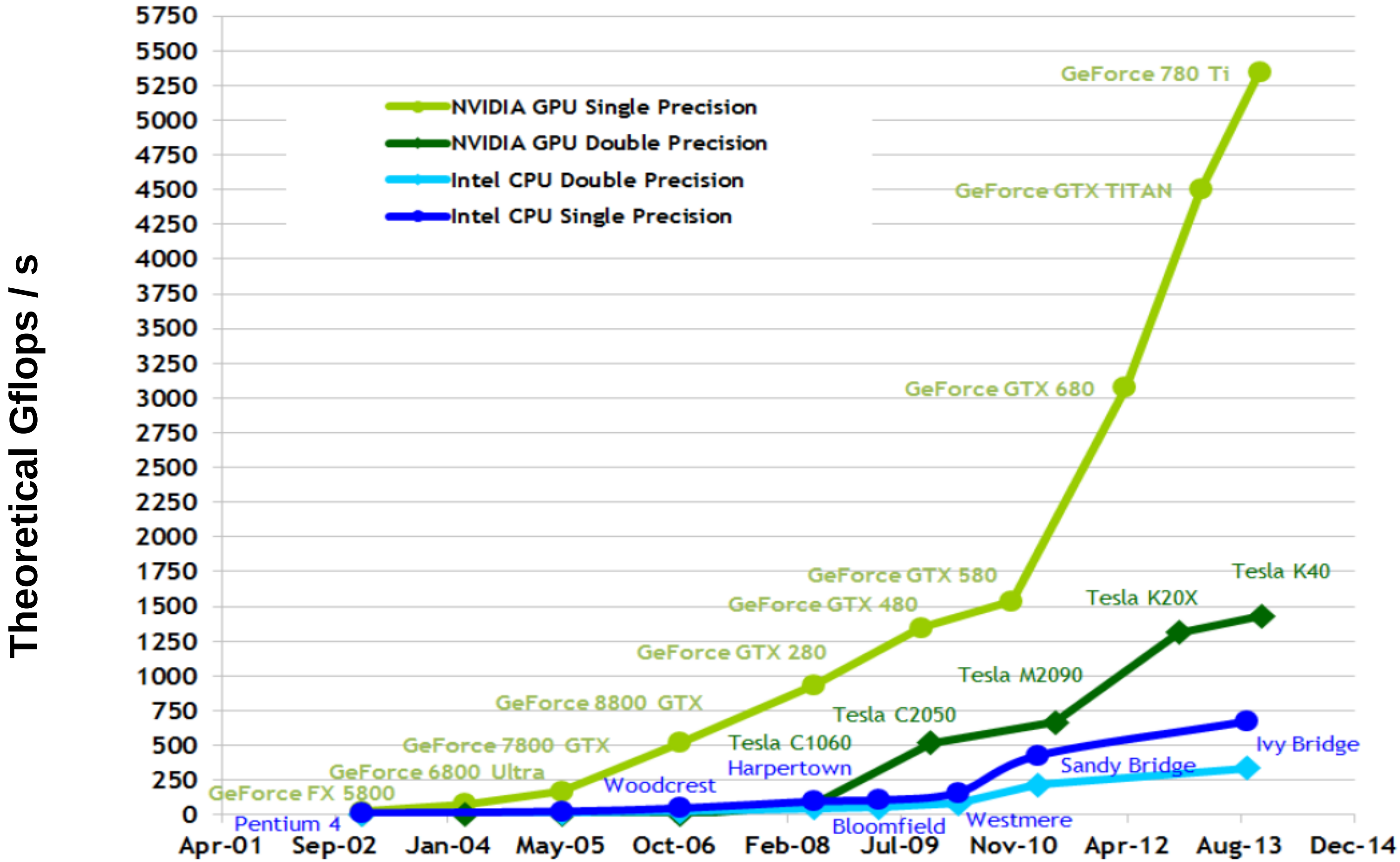
CPU



GPU

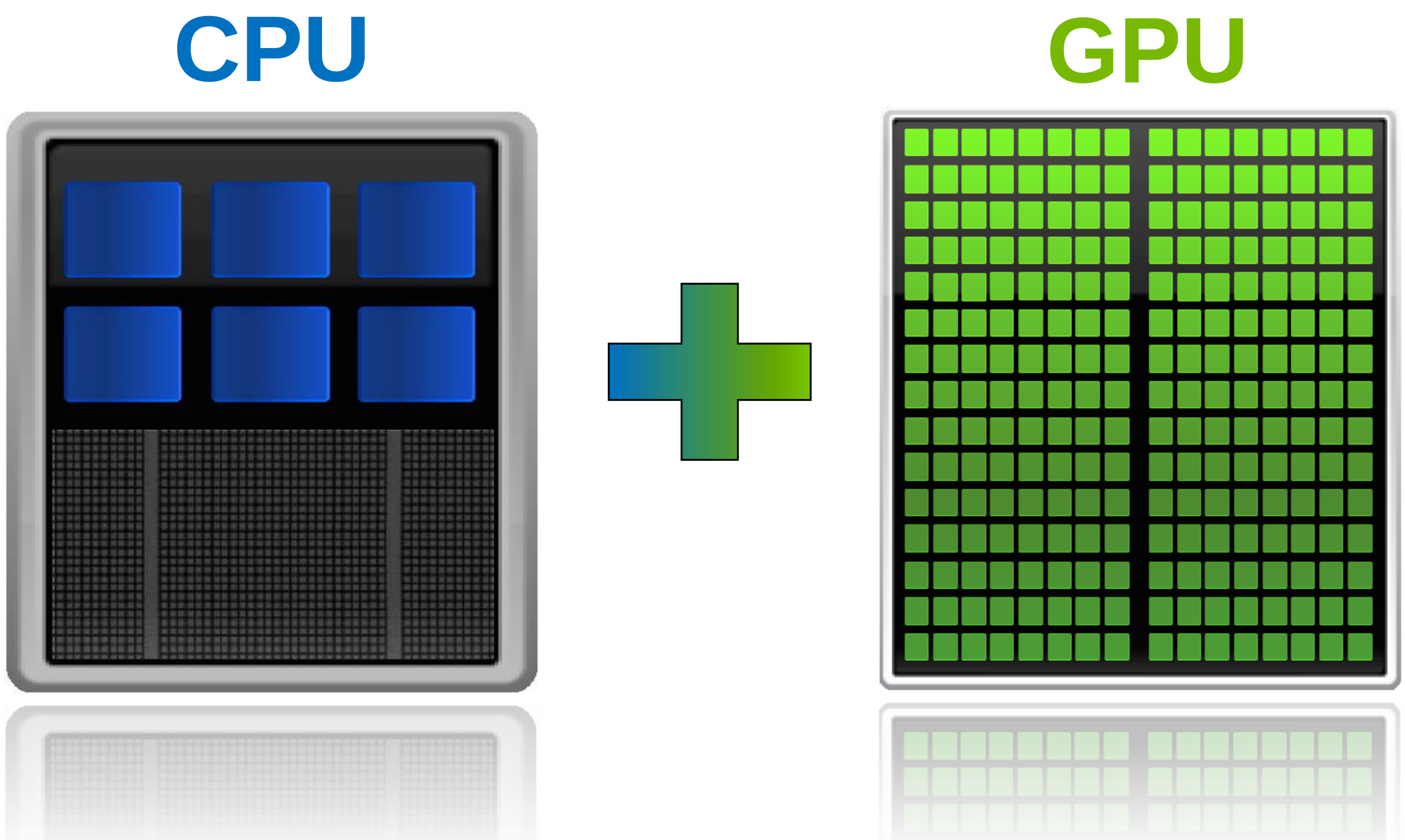


Floating Point Operations

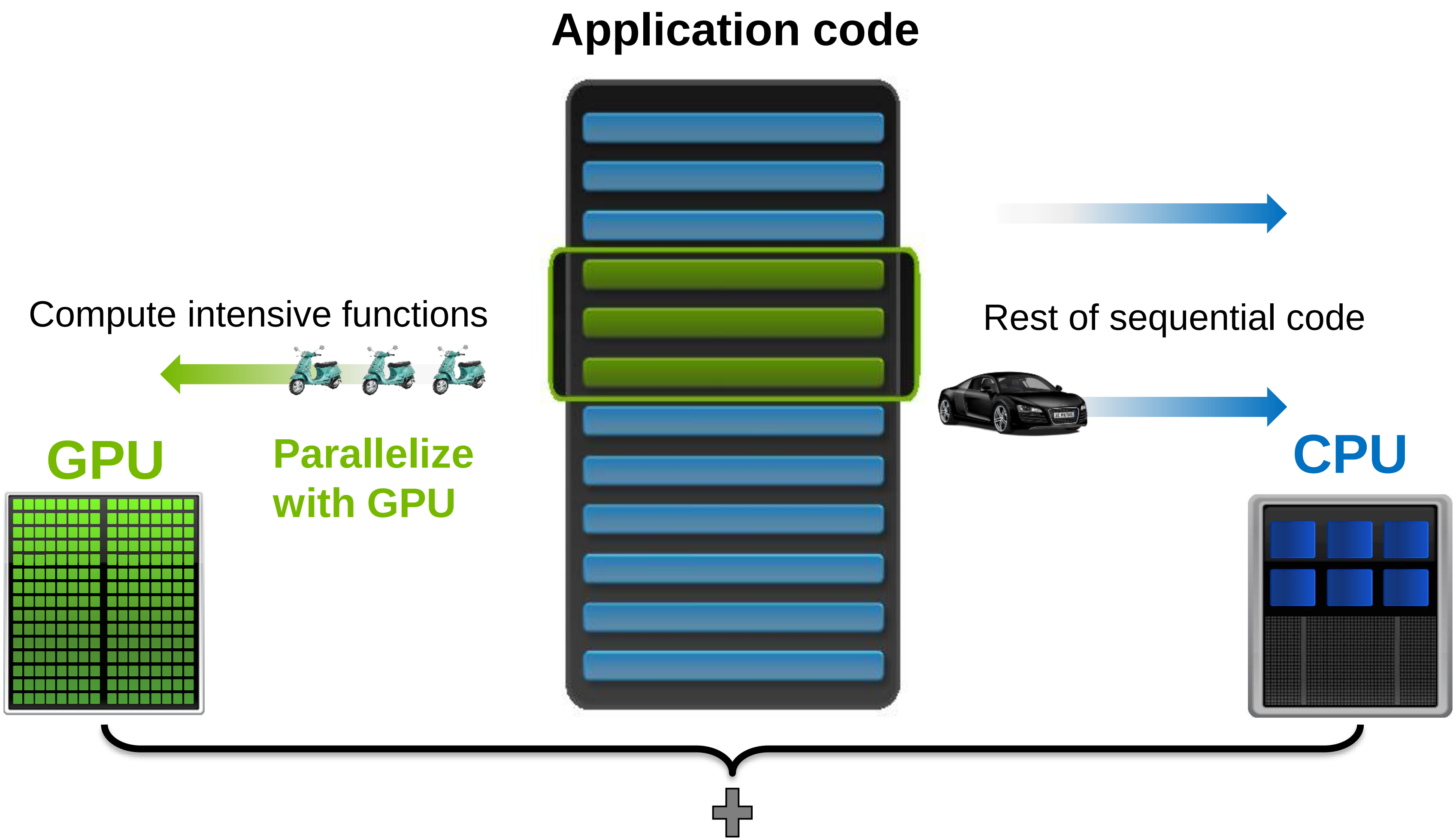


Ref: NVIDIA CUDA C Programming Guide

Better adding tan choosing



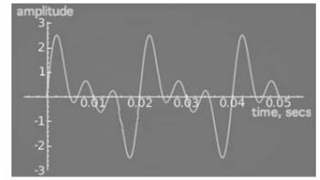
Adding GPUs


Application code
Compute intensive functions
Rest of sequential code
GPU
Parallelize
with GPU
CPU

How starting?

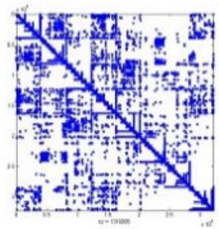
What is CUDA?

- Parallel computing platform
- Programming model
- CUDA uses GPUs for general purpose
- C, C++, Fortran…
- A lot of libraries

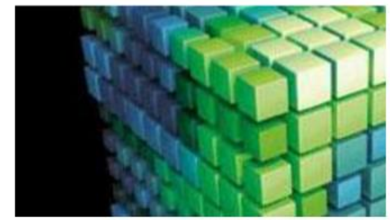


cuFFT

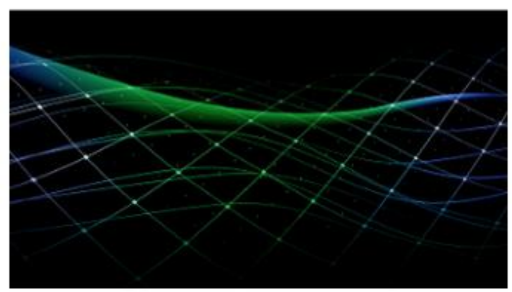
cuDNN



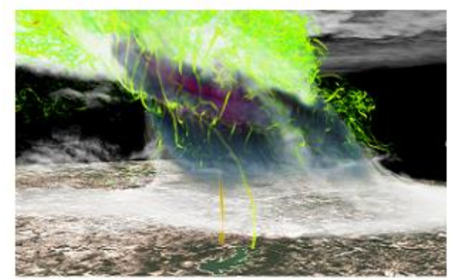
cuSPARSE



cuBLAS



cuSOLVER

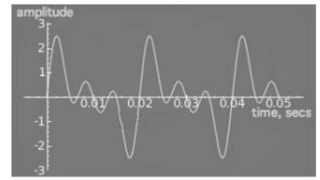


IndeX Framework

B/S/H/

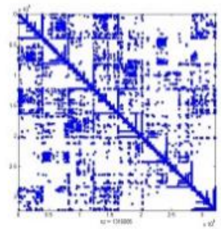
What is CUDA?

- Parallel computing platform
- Programming model
- CUDA uses GPUs for general purpose
- C, C++, Fortran…
- A lot of libraries

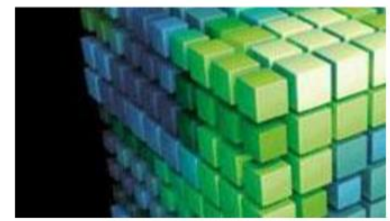


cuFFT

cuDNN

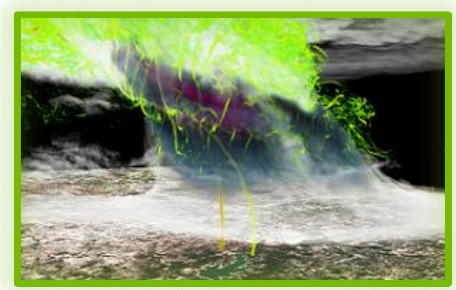


cuSPARSE



cuBLAS

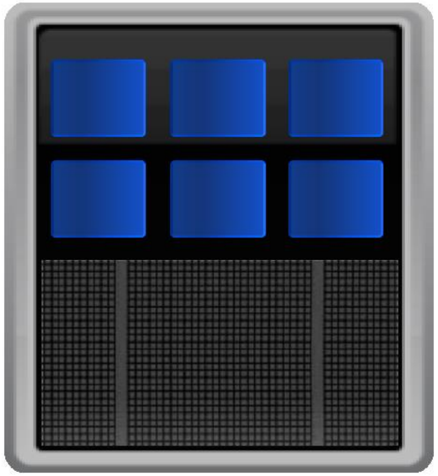
cuSOLVER



IndeX Framework

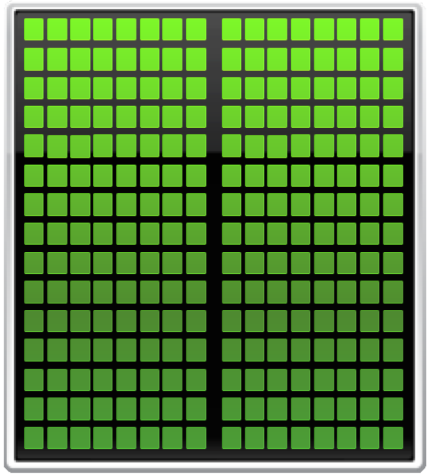
Terminology

Host



CPU + host memory

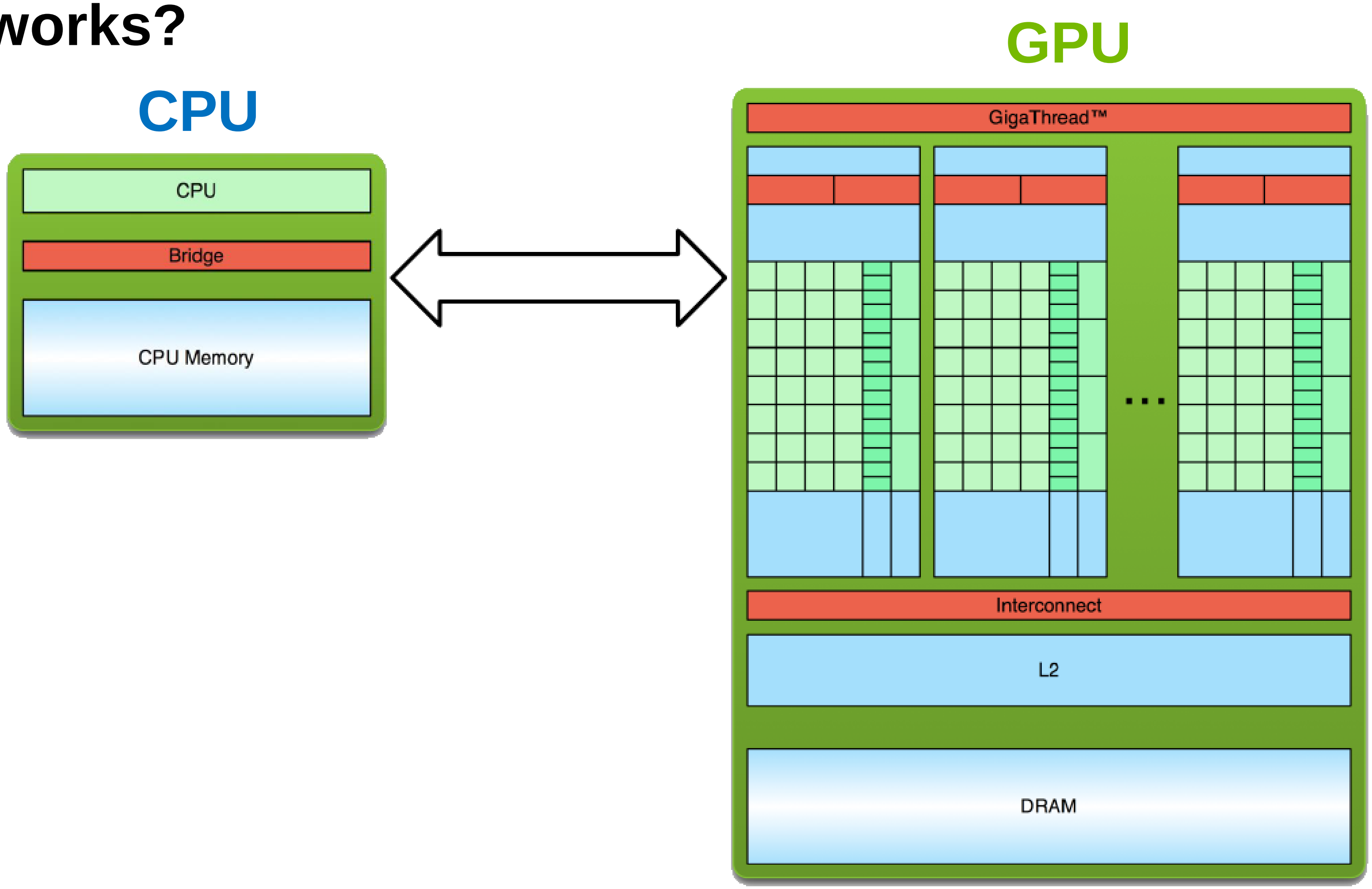
Device



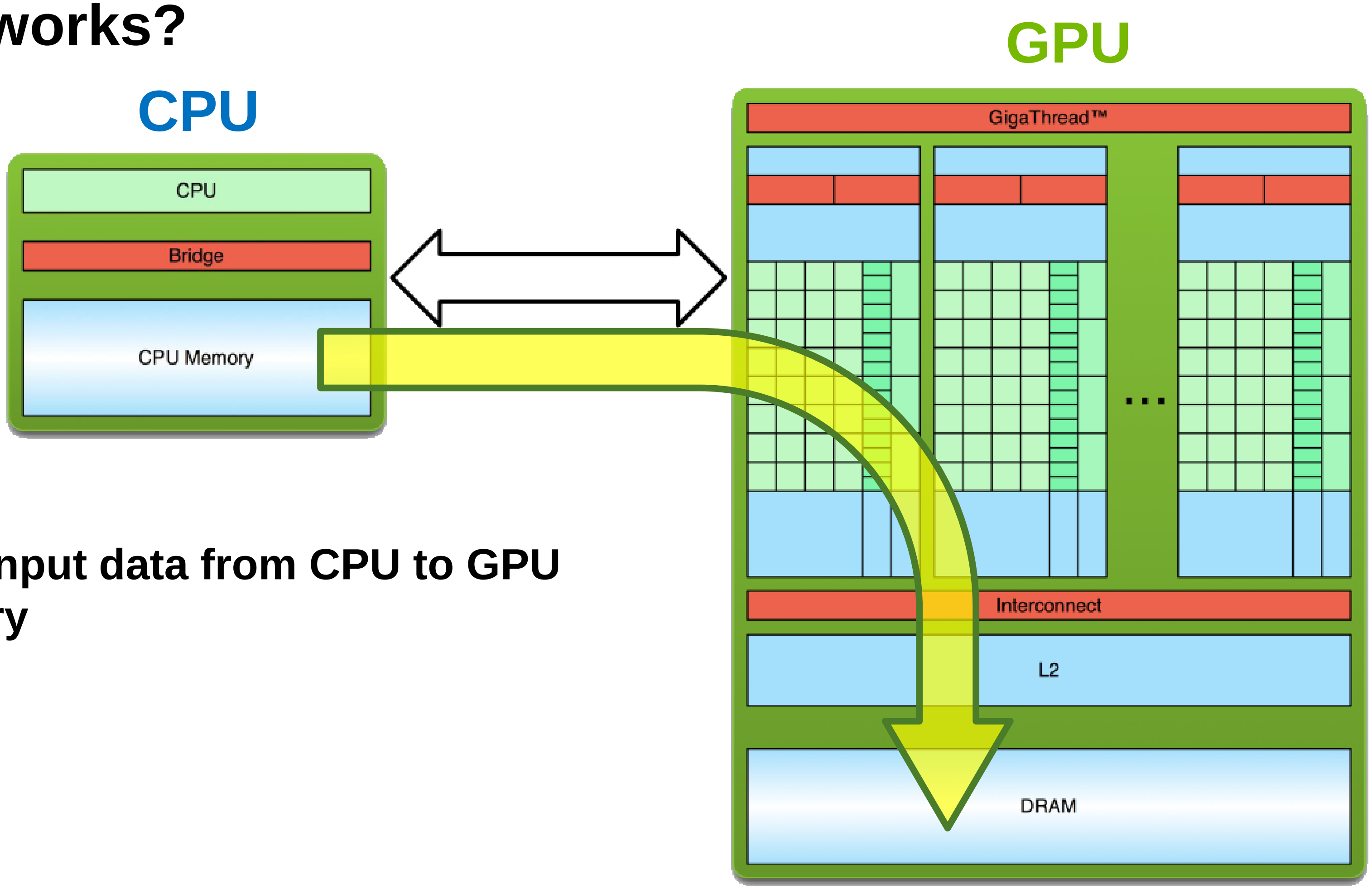
GPU + device memory

How it works?

How it works?

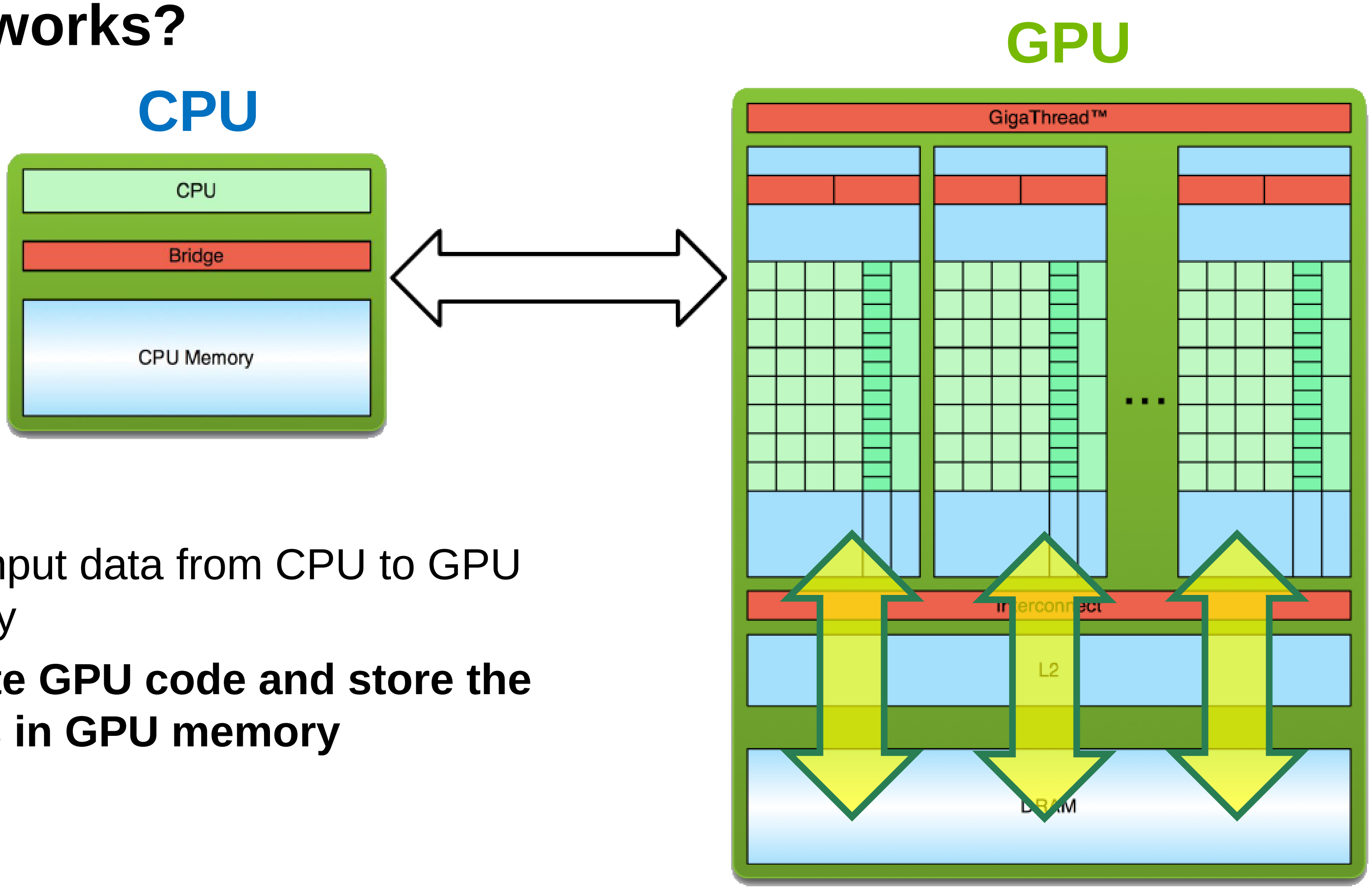

CPU
CPU
Bridge
CPU Memory
GPU
GigaThread™
Interconnect
L2
DRAM

How it works?



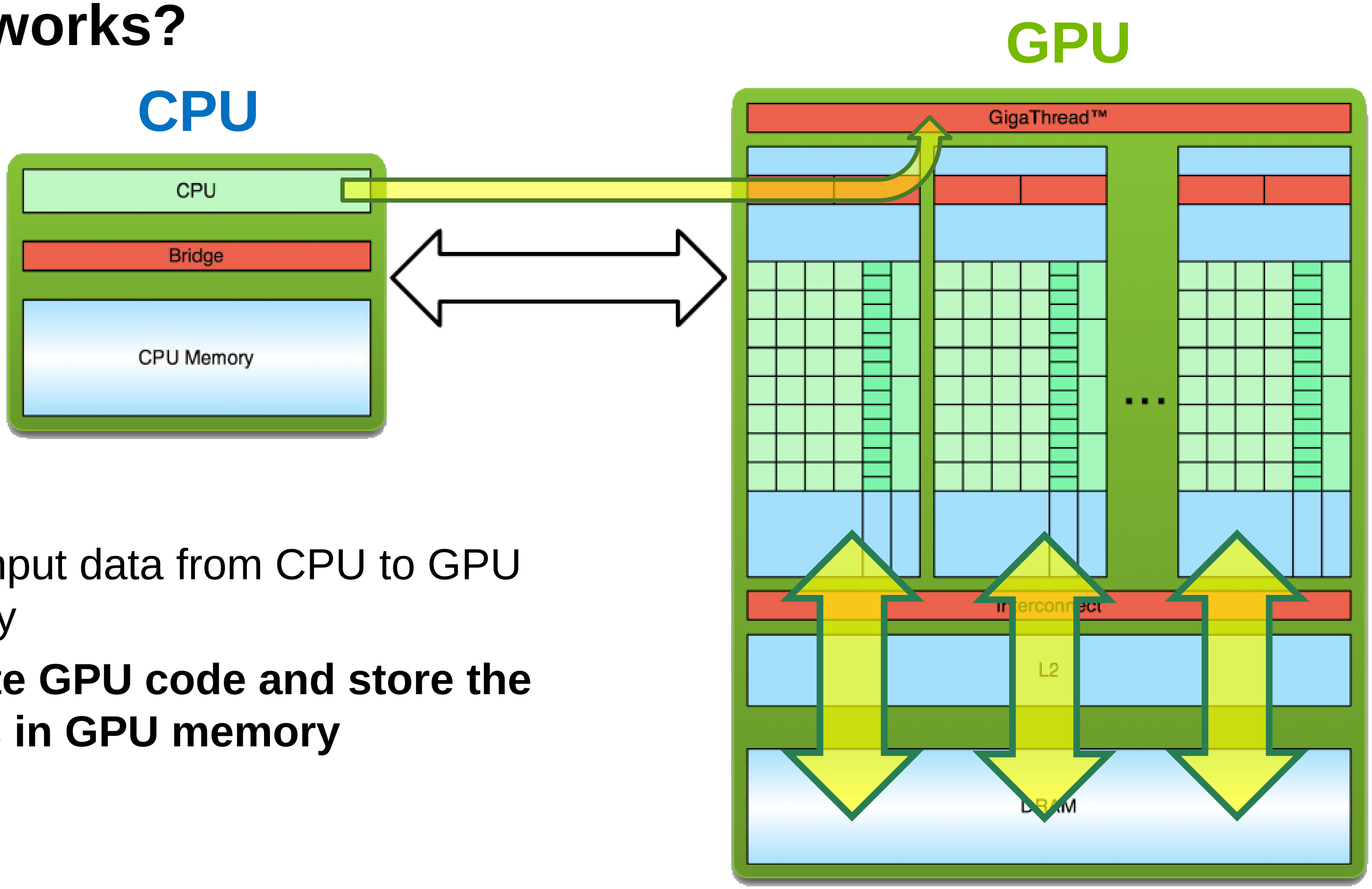
1. **Copy input data from CPU to GPU memory**

How it works?



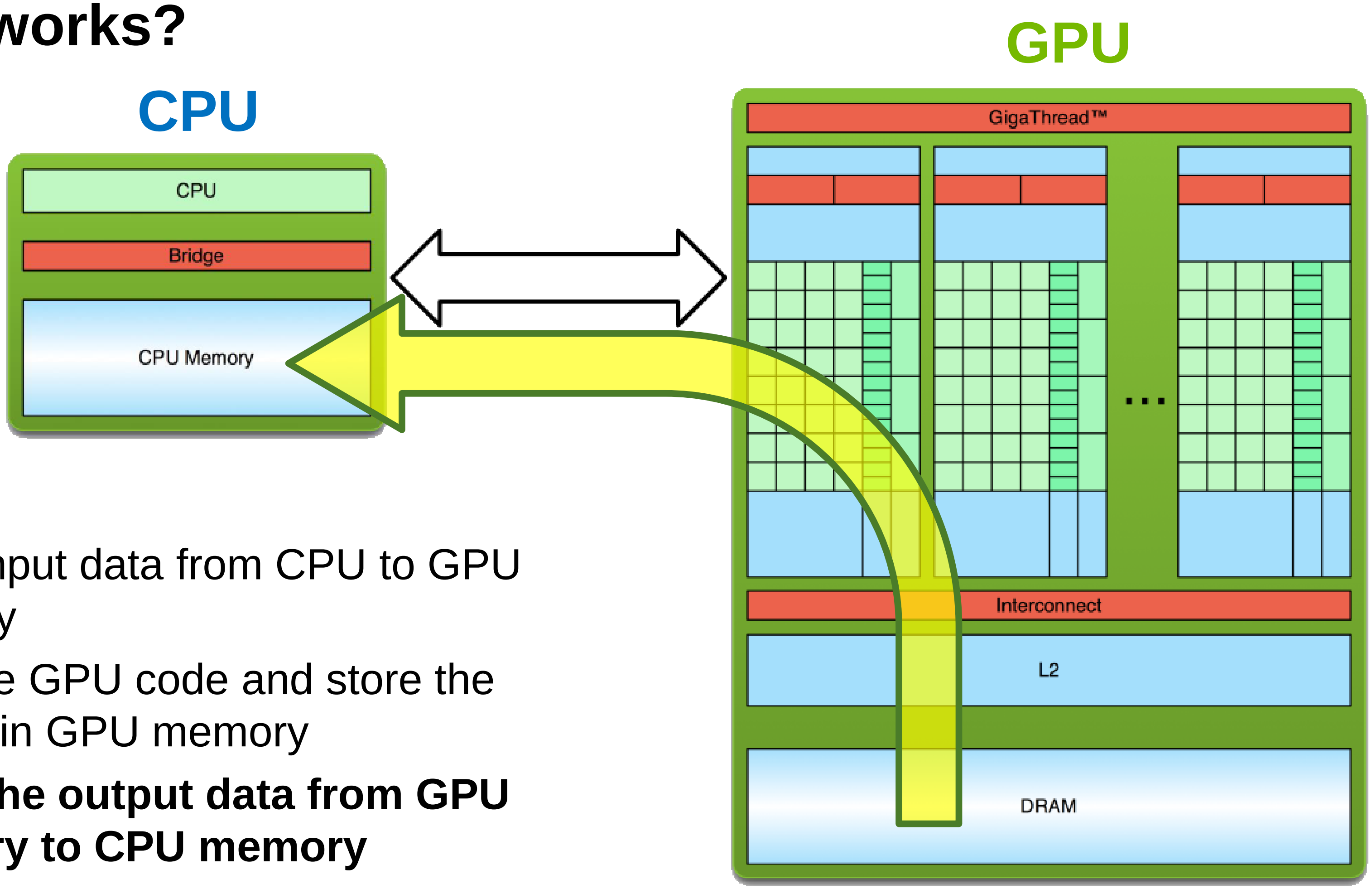
1. Copy input data from CPU to GPU memory
2. **Execute GPU code and store the results in GPU memory**

How it works?



1. Copy input data from CPU to GPU memory
2. **Execute GPU code and store the results in GPU memory**

How it works?



1. Copy input data from CPU to GPU memory
2. Execute GPU code and store the results in GPU memory
3. **Copy the output data from GPU memory to CPU memory**

Hello World!

Hello world!

Standard C code

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

Run on the host

NVIDIA compiler (nvcc) can compile programs with no device

Hello world!

Standard C code

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

C with CUDA

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

Two new elements…

Hello world!

C with CUDA

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

Two new elements…

Hello world!

- Keyword __global__ indicates that the function is a device function. It is called from host code

- NVIDIA compiler separates code into host and device:
  - **Device functions** are processed by nvcc
  - **Host functions** are processed by standard compiler like gcc

C with CUDA

Two new elements…

Hello world!

C with CUDA

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

Two new elements…

Hello world!

- Triple angle brackets are a call from host code to device code
- Also called a "**kernel launch**"
- The kernel *mykernel()* does nothing, but a device function is needed

C with CUDA

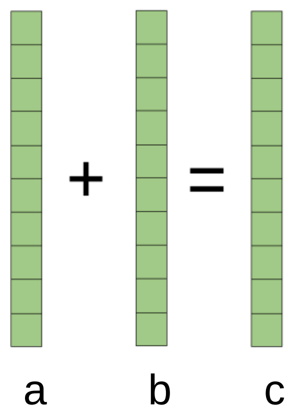
```
__global__ void mykernel(void)
{
}
int main(void)
{
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Two new elements…

but massive parallelism?

We need a more interesting example…

Vector addition



Integer addition

- The kernel to add two integers

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

Integer addition

- The kernel to add two integers

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

- As before the keyword __global__ means that
    - `add()` is a device function
    - `add()` will be called from the host

Integer addition

- The kernel to add two integers

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

- As before the keyword __global__ means that
  - `add()` is a device function
  - `add()` will be called from the host

- We use device pointers `*a`, `*b`, `*c` point to GPU memory

Pointers reminder?

Integer addition

- The kernel to add two integers

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

- As before the keyword __global__ means that
  - `add()` is a device function
  - `add()` will be called from the host

- We use device pointers `*a`, `*b`, `*c` point to GPU memory

Pointers reminder?

- CUDA functions to handle device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

Integer addition

- Let's look at the main() function

```
Host copies of a, b, c
```

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Integer addition

- Let's look at the main() function

```
Host copies of a, b, c
Device copies of a, b, c
```

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Integer addition

- Let’s look at the main() function

```
Host copies of a, b, c
Device copies of a, b, c

Allocate space in
device for a, b, c
```

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Integer addition

- Let's look at the main() function

```
Host copies of a, b, c
Device copies of a, b, c

Allocate space in
device for a, b, c

Inputs
```

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Integer addition

- Let's look at the main() function

    Host copies of a, b, c

    Device copies of a, b, c

    Allocate space in device for a, b, c

    Inputs

    Copy inputs to device

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Integer addition

- Let's look at the main() function

Host copies of a, b, c

Device copies of a, b, c

Allocate space in
device for a, b, c

Inputs

Copy inputs to device

Launch add() kernel on GPU

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Integer addition

- Let's look at the main() function

Host copies of a, b, c

Device copies of a, b, c

Allocate space in
device for a, b, c

Inputs

Copy inputs to device

Launch add() kernel on GPU

Copy result to host

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

but massive parallelism?
vector addition?

Let’s running in parallel…

Moving to parallel

- Instead of executing `add()` once, execute N times in parallel

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

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Moving to parallel

- Instead of executing `add()` once, execute N times in parallel

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

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

```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Moving to parallel

- Instead of executing `add()` once, execute N times in parallel

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

- Each parallel invocation of `add()` is referred to as a **`block`**

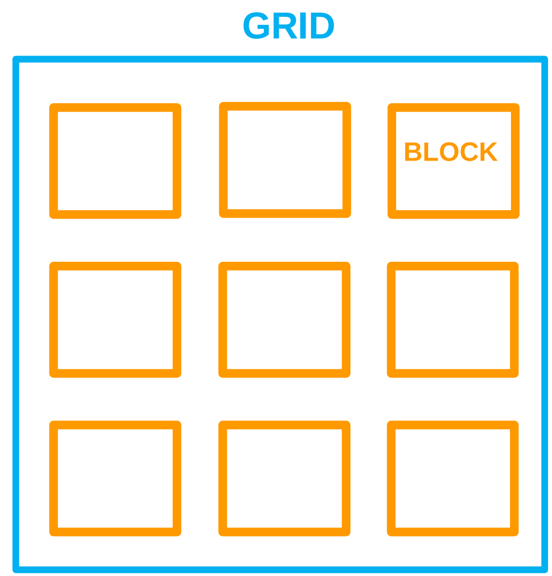
```
int main(void)
{
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Moving to parallel

- Instead of executing `add()` once, execute N times in parallel

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

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

- Each parallel invocation of `add()` is referred to as a **`block`**



Moving to parallel

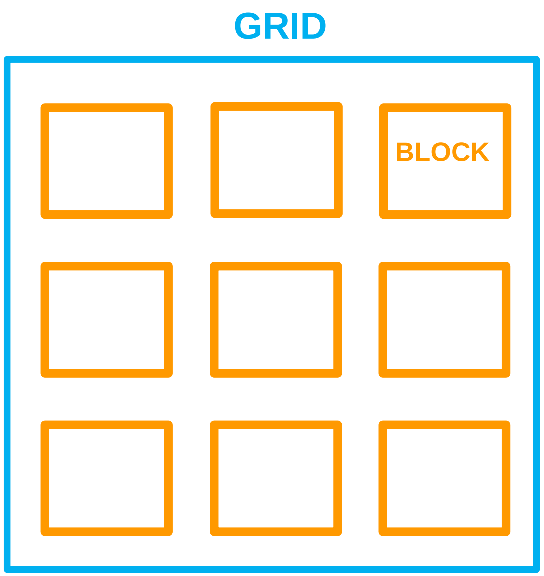
- Instead of executing `add()` once, execute N times in parallel

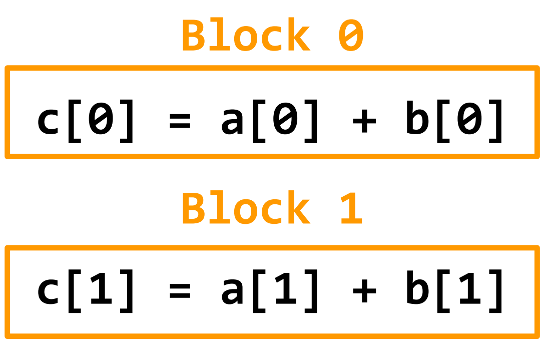
```
add<<<1,1>>>();
```

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

- Each parallel invocation of `add()` is referred to as a **`block`**
  - block indexing using **`blockIdx.x`**

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





Vector addition

Number of blocks

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

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (int *)malloc(size); random_ints(a,N);
    b = (int *)malloc(size); random_ints(b,N);
    c = (int *)malloc(size);

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<N,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Vector addition

Number of blocks

New setup of inputs

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

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (int *)malloc(size); random_ints(a,N);
    b = (int *)malloc(size); random_ints(b,N);
    c = (int *)malloc(size);

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<N,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Vector addition

Number of blocks

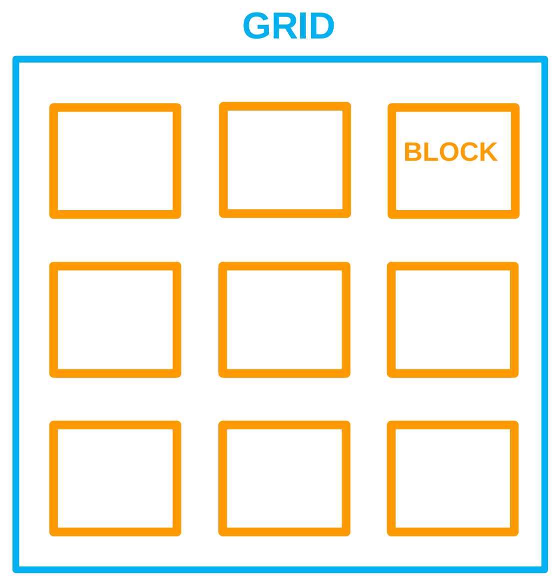
New setup of inputs

Launch kernel in N blocks

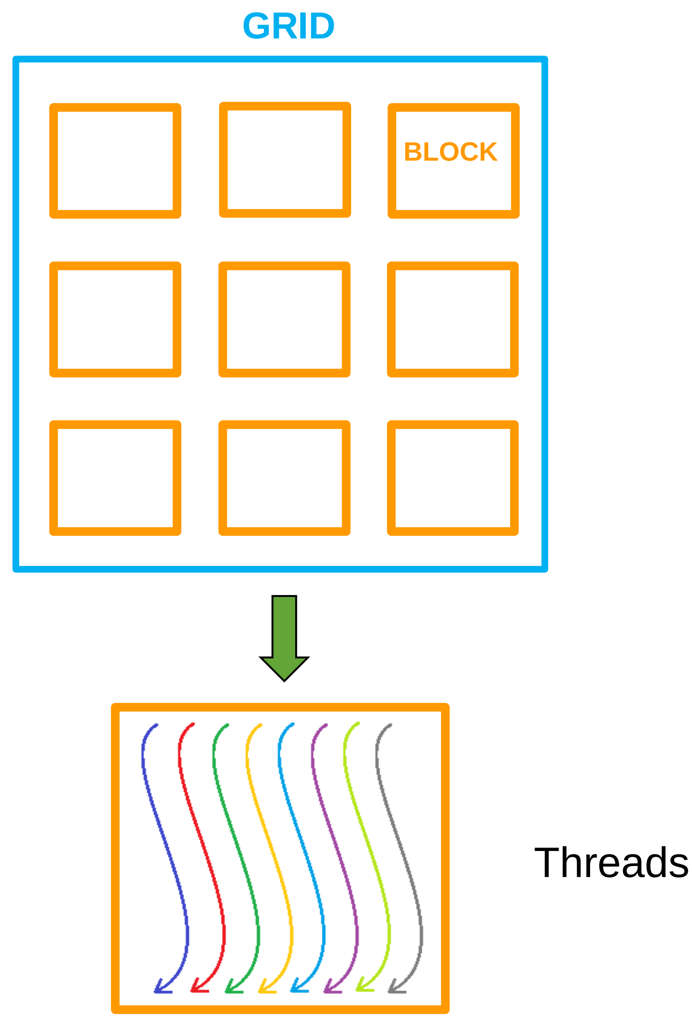
```
#define N 512
int main(void)
{
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = N * sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (int *)malloc(size); random_ints(a,N);
    b = (int *)malloc(size); random_ints(b,N);
    c = (int *)malloc(size);

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<N,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

Introducing threads

Threads



Threads

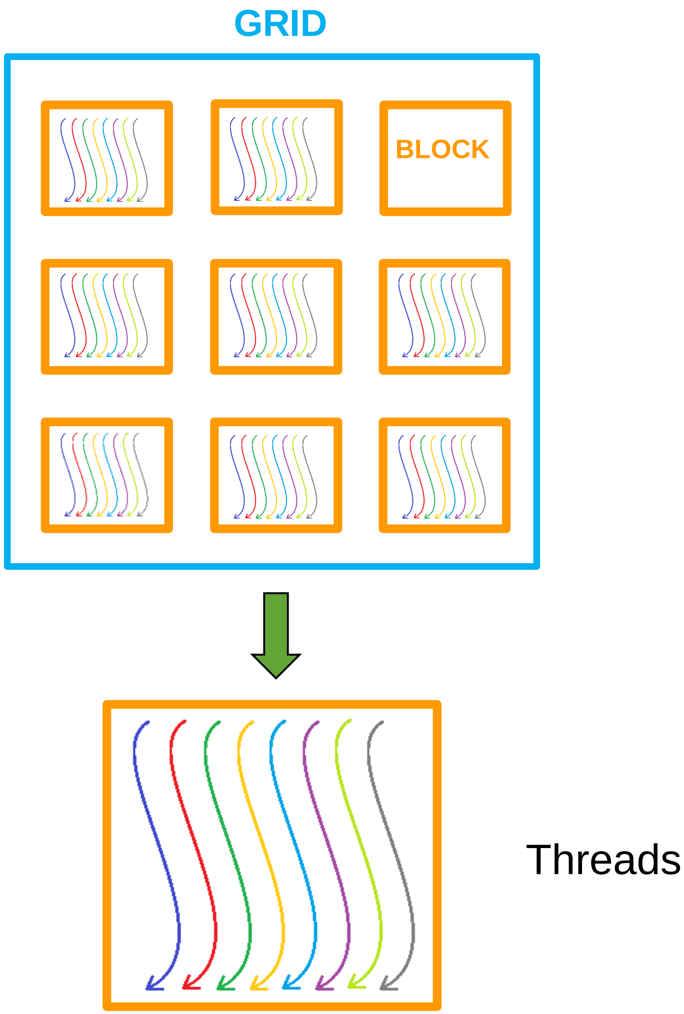


Threads

- Change `add()` to use parallel threads

```
add<<<N,1>>>();
```
Blocks

```
add<<<1,N>>>();
```
Threads



Threads

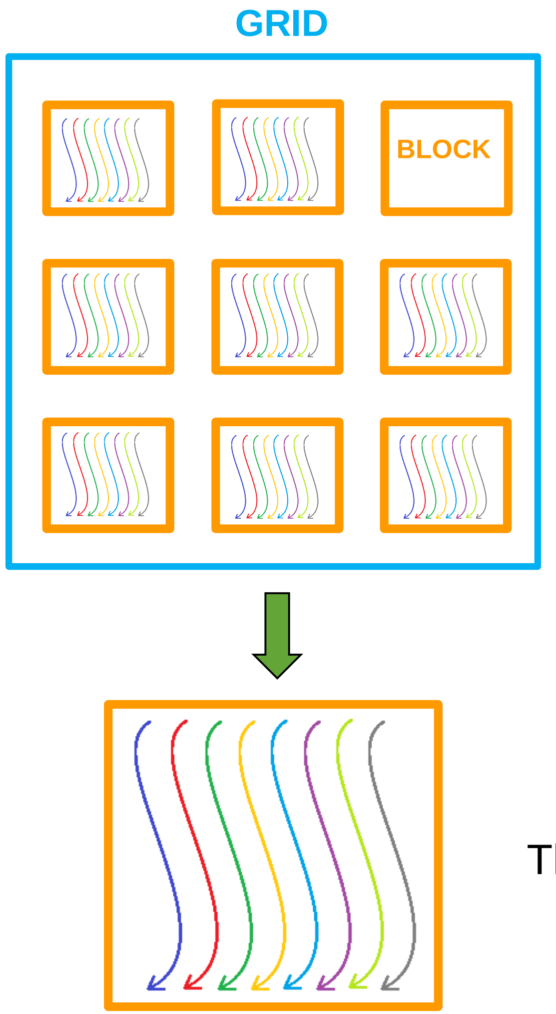
- Change `add()` to use parallel threads

`add<<<N,1>>>();` Blocks

`add<<<1,N>>>();` Threads

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

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



Threads

Review

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - With blocks `<<<N,1>>>`
  - With threads `<<<1,N>>>`

Review

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - With blocks `<<<N,1>>>`
  - With threads `<<<1,N>>>`

Steps in CUDA

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

3 Ways to accelerate applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

Applications

Libraries	OpenACC Directives	Programming Languages
“Drop-in” Acceleration	Easily Accelerate Applications	Maximum Flexibility

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

Questions?