CUDA Programming

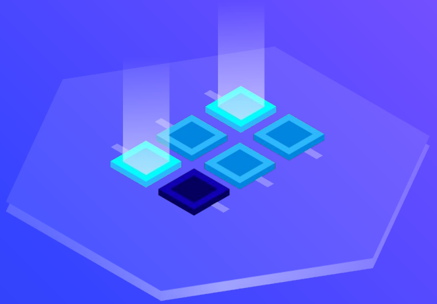
Outline

- What is CUDA?
- When to use CUDA?
- Principles of CUDA
- How CUDA works?

What is Cuda ?

What is Cuda ?

- CUDA = Cuda Architecture + Cuda Programing Model
- Developed by NVIDIA
- Parallel computing on graphics processing units (GPUs)



CUDA

Programming language based on C++

Software development kit

Massively parallel hardware designed

> “CUDA language is based on C++ with a few additional keywords and concepts, which makes it fairly easy for non-GPU programmers to pick up

```
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

History of CUDA

Compute Unified Device Architecture

1989

Pixel Machine

1999

GPU was invented

purely fixed-function devices
GeForce 256

2007

CUDA

Compute Unified Device Architecture
By NVIDIA

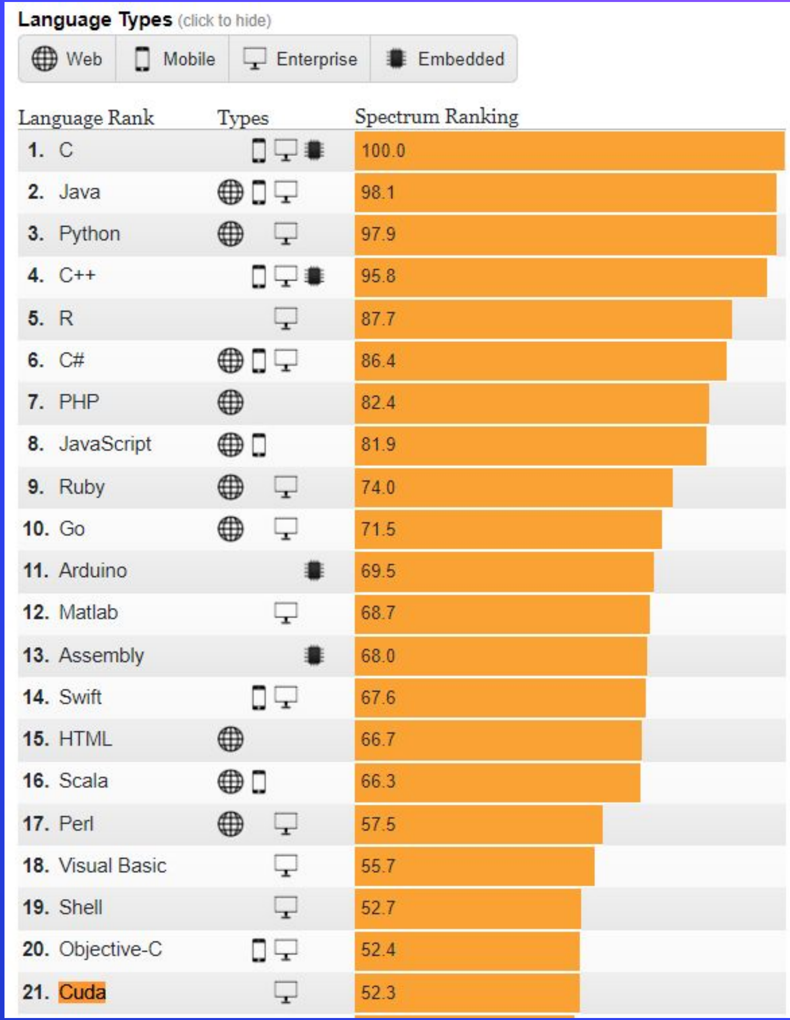
2009

OpenCL

CUDA competitor
Apple and the Khronos Group
Not limited to Intel/AMD CPUs with Nvidia GPUs

Popularit
y

The Top Programming Languages 2016



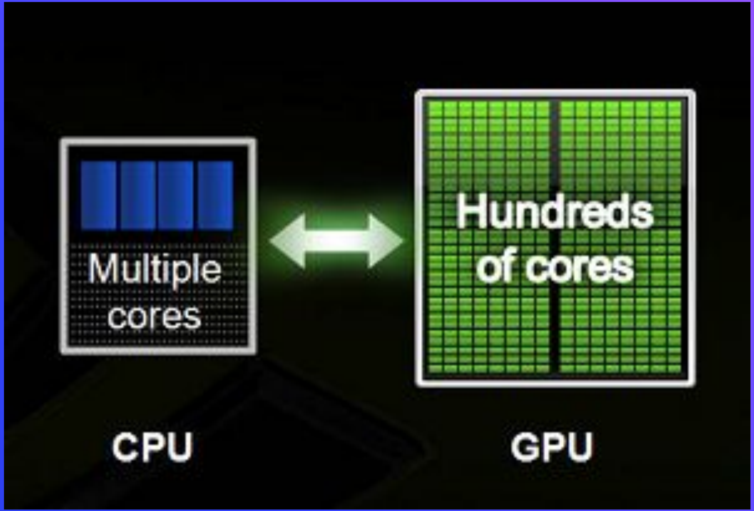
When to use Cuda ?

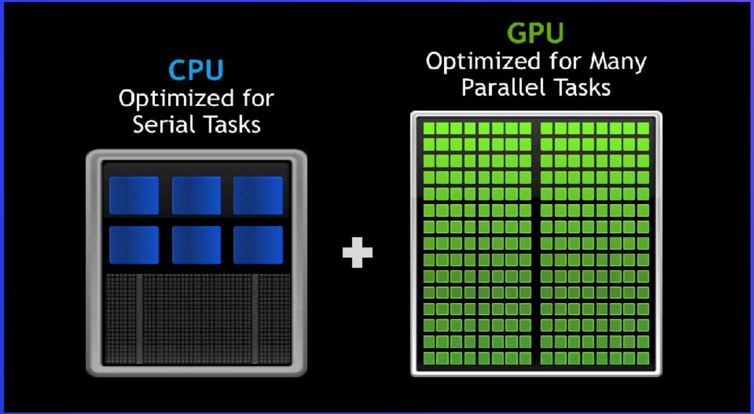
When to use Cuda ?

- Lots of data
- Lots of computations


Benefits of Cuda ?

- Harnesses the power of the GPU by using parallel processing;
- Running thousands of simultaneous reads instead of single, dual, or quad reads on the CPU.



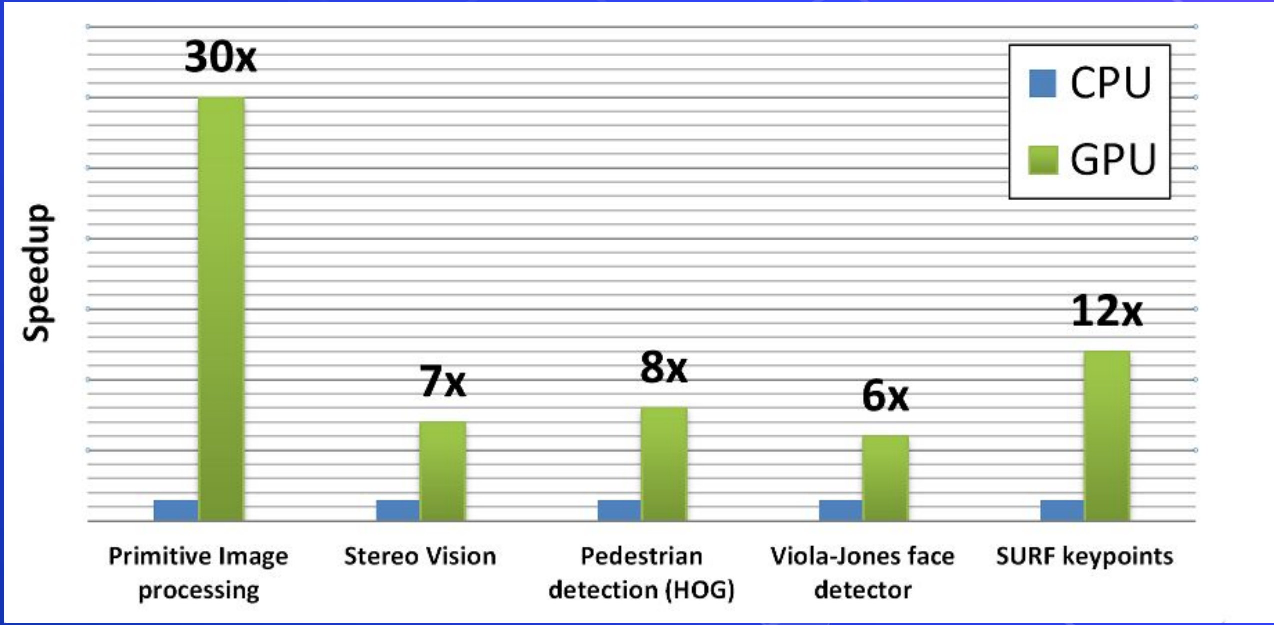


Benefits of Cuda ?

- C/C++ is widely used, easy to learn how to program for CUDA.
- Most of graphics cards of NVIDIA support CUDA.

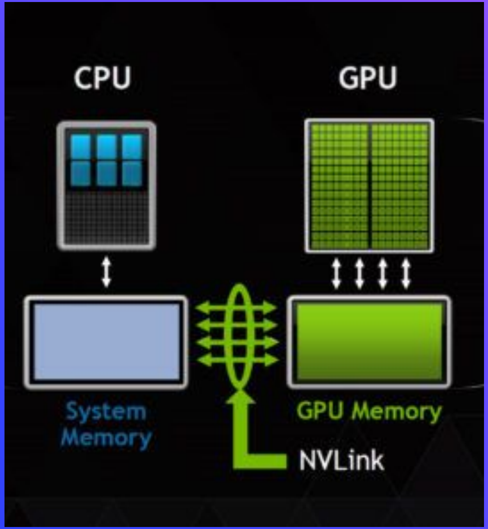
5x to 200x

Huge increase in processing power over conventional CPU processing. Early reports suggest speed increases of 5x to 200x over CPU processing speed.



Limitations of CUDA

Over traditional general purpose computation on GPUs



- Latency between the CPU and GPU

  Copying between host and device memory may incur a performance hit due to system bus bandwidth

- A single process must run spread across multiple memory spaces

  Threads should be running in groups of at least 32 for best performance

- CUDA-enabled GPUs are only available from NVIDIA

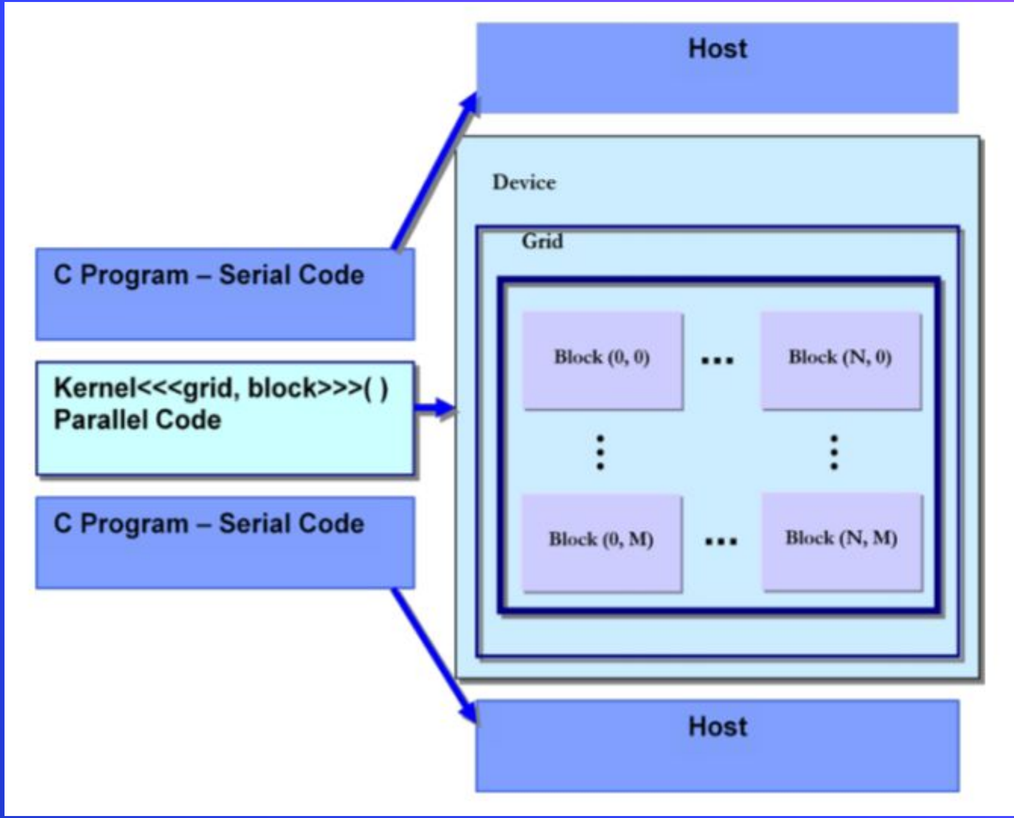
  Unlike OpenCL

Principles of CUDA

Parallel structure

Paradigm

- Parallel programming
- Combination of serial and parallel executions



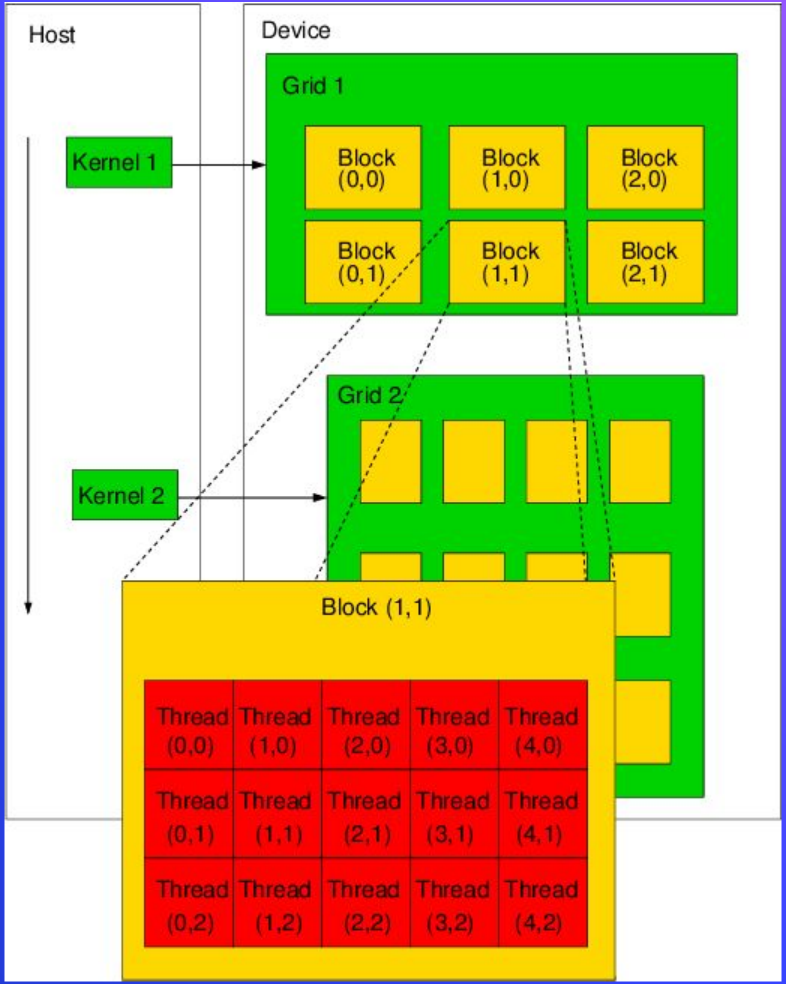
Heterogeneous Architecture

Programming Model

- Device = GPU
- Host = CPU
- Kernel = Functions run on Device

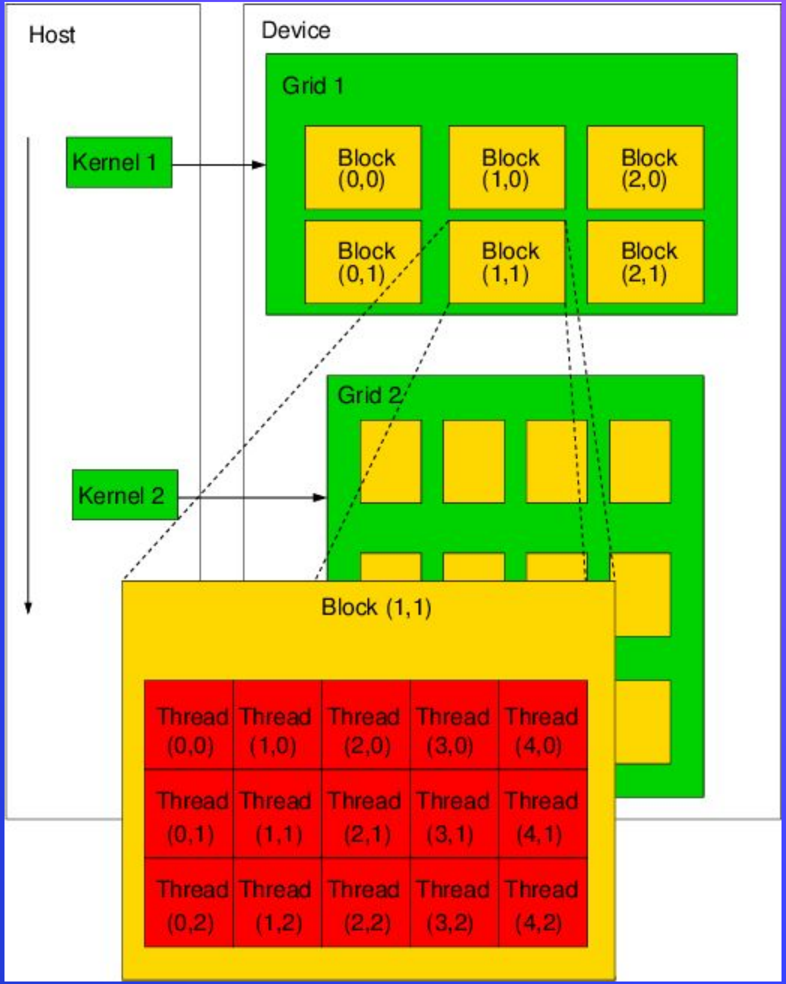
More about CUDA kernel

specific functions in CUDA, A kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads



Programming Model

- 512, 1024 or 2048 threads in one block
- All blocks define a grid
- All block execute same program (kernel)
- Blocks are independent
- Only one kernel at a time



Programming Model

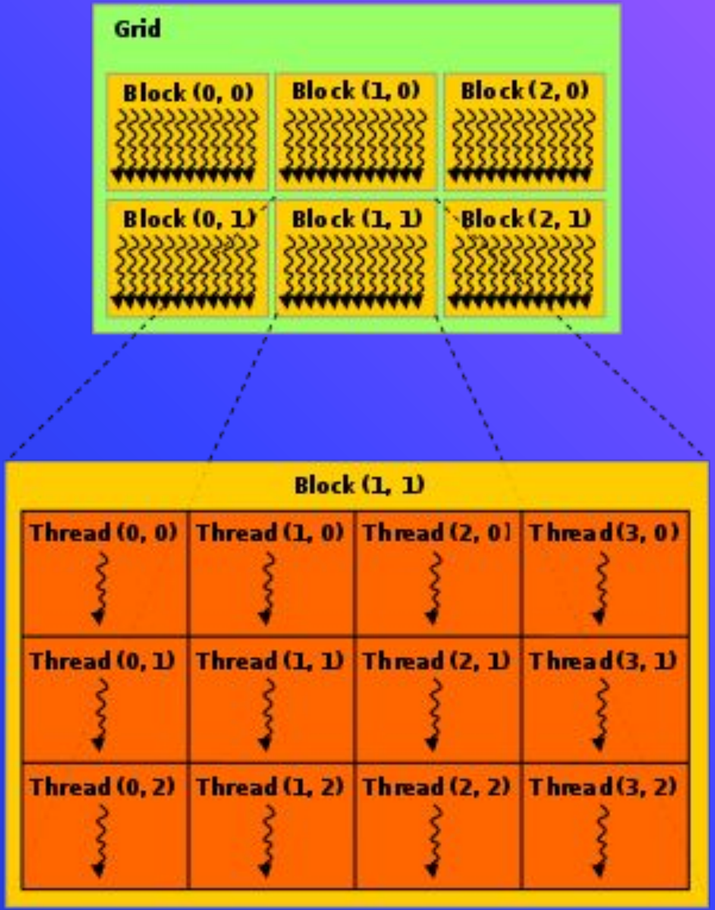
<table>
<tr><th rowspan="2">Technical specifications</th><th colspan="5">Compute capability (version)</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th></tr>
<tr><td>Maximum x- or y- dimensions of a grid of thread blocks</td><td colspan="5">65535</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="4">512</td><td>1024</td></tr>
</table>

In our GPU using for experiment

Max of block = 65535
Max of threads each block = 1024 (Compute Capability: 3.5)

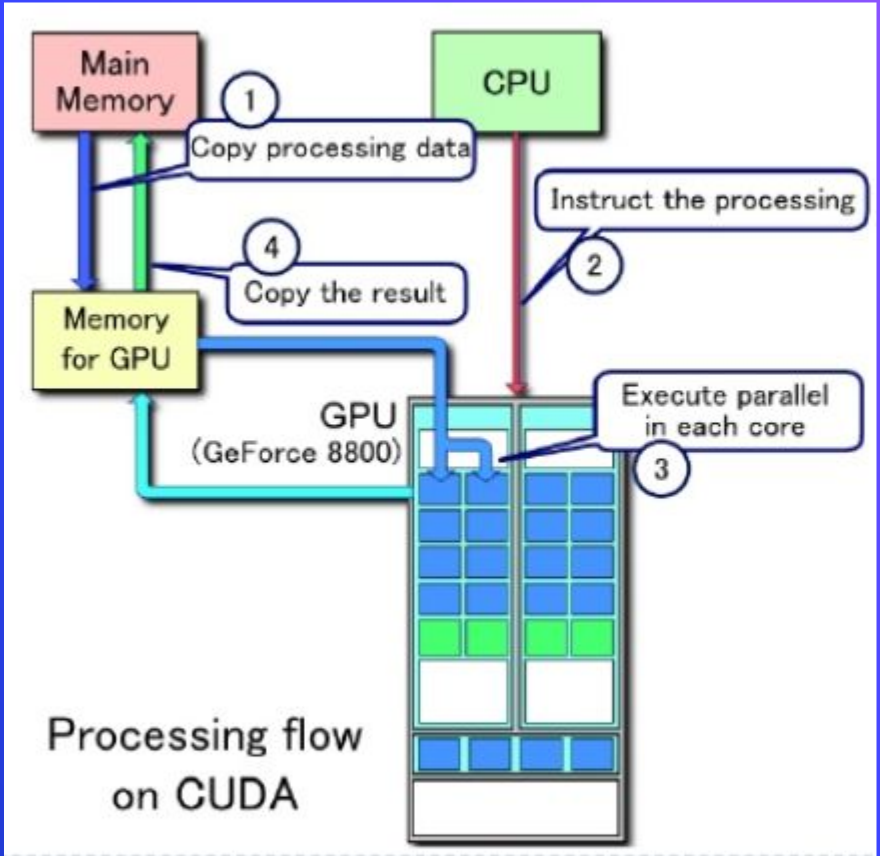
CUDA Thread

- **Thread Cooperation**
  - Share memory > powerful feature of CUDA
  - All threads run same code
- **CUDA threads vs CPU threads**
  - CUDA thread is lightweight
  - CUDA use 1000s threads, CPU single, dual, or quad



Processing Flow

1. Copy data from Main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy result GPU mem to Main mem



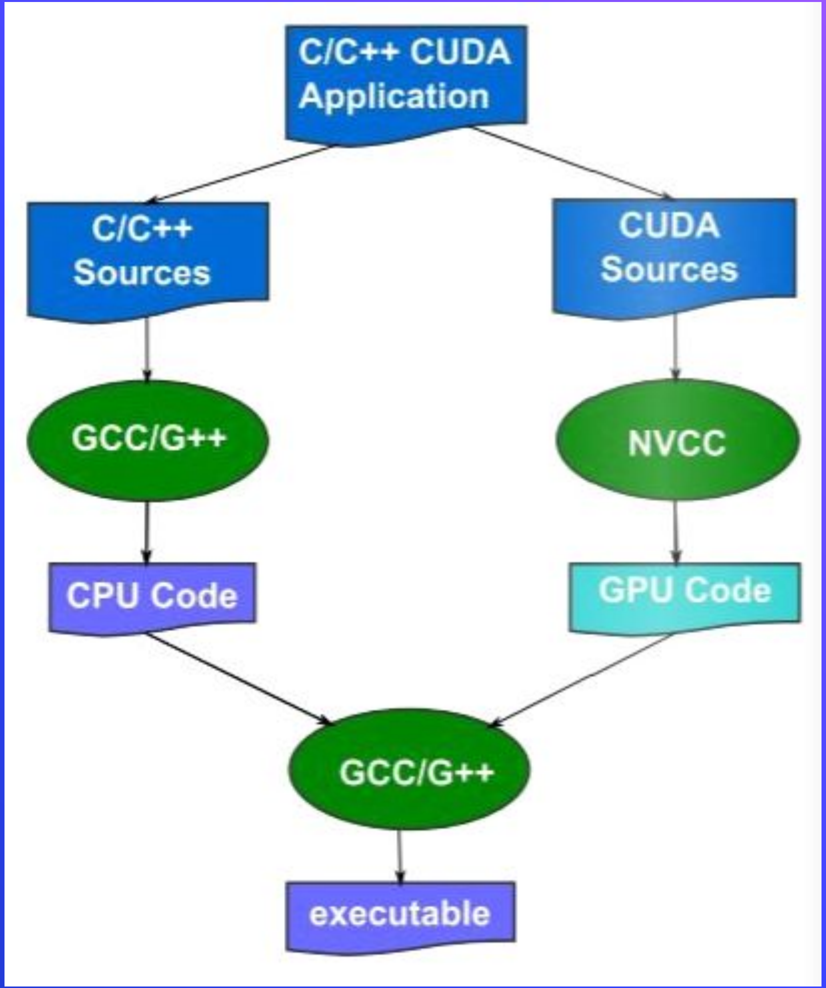
Processing Flow

- Copy data from Main memory to GPU mem
- GPU execute parallel
- Copy result GPU mem to Main mem

```
cudaMemcpy( d_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( d_c, c, SIZE*sizeof(int), cudaMemcpyHostToDevice );

VectorAdd<<< 1, SIZE >>>(d_a, d_b, d_c, SIZE);

cudaMemcpy( c, d_c, SIZE*sizeof(int), cudaMemcpyDeviceToHost );
```

CUDA compiler

- GCC/G++ Compiler for Host code
- NVCC Compiler for device code
- GCC/G++ for linking



Syntax

Identifier

RULES FOR NAMING IDENTIFIER

- Case sensitive
- Cannot start with a digit
- Underscore can be used as first character
- Other special characters are not allowed
- Cannot use keywords as identifier.

Followsym: [a-zA-Z0-9_$]

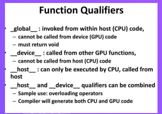
Identifier: [a-zA-Z]{followsym}* | {_{followsym}}+

Specification

- Function Qualifiers
- Function Call
- CUDA Built-in device variable
- Variable Qualifiers

Function Qualifiers

Example


Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {

    int i = threadIdx.x;

    __shared__ bool sharedArr[100];

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

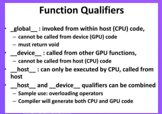
> CUDA language is based on C++ with a few additional keywords and concepts, which makes it fairly easy for non-GPU programmers to pick up

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

```c
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Function Qualifiers

- __global__ :
  - invoked from within CPU code
  - can not be called from GPU code
  - must return void
- __device__ :
  - called from other GPU functions
  - can not be called from CPU code
- __host__ :
  - can only be executed by CPU
  - Called from host



Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

Function Call

Example

Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

```
for (int i = 0; i < SIZE; i++) {
        a[i] = i;
        b[i] = i;
}

add<<<1,1>>>(a,b,c,SIZE);

cudaDeviceSynchronize();

cudaFree(a);
cudaFree(b);
cudaFree(c);
```

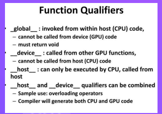
Function Call

```
kernel_function<<<M,N>>>(list of arguments);
```

M is number of block
N is number of threads each block (size of block)
*M*N is number of times that function is executed.*

CUDA Built-in device variable

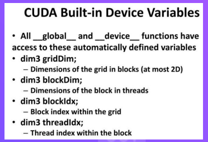
Example


Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

```
__global__
void add(int *a, int *b, int *c, int n) {

    int i = threadIdx.x;

    __shared__ bool sharedArr[100];

    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

int main() {
    int *a, *b, *c;
```

CUDA Built-in device variable


CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
- dim3 gridDim;
  - Dimensions of the grid in blocks (at most 2D)
- dim3 blockDim;
  - Dimensions of the block in threads
- dim3 blockIdx;
  - Block index within the grid
- dim3 threadIdx;
  - Thread index within the block

★ dim3 gridDim : Dimensions of the grid in blocks

★ dim3 blockDim : Dimensions of the block in threads

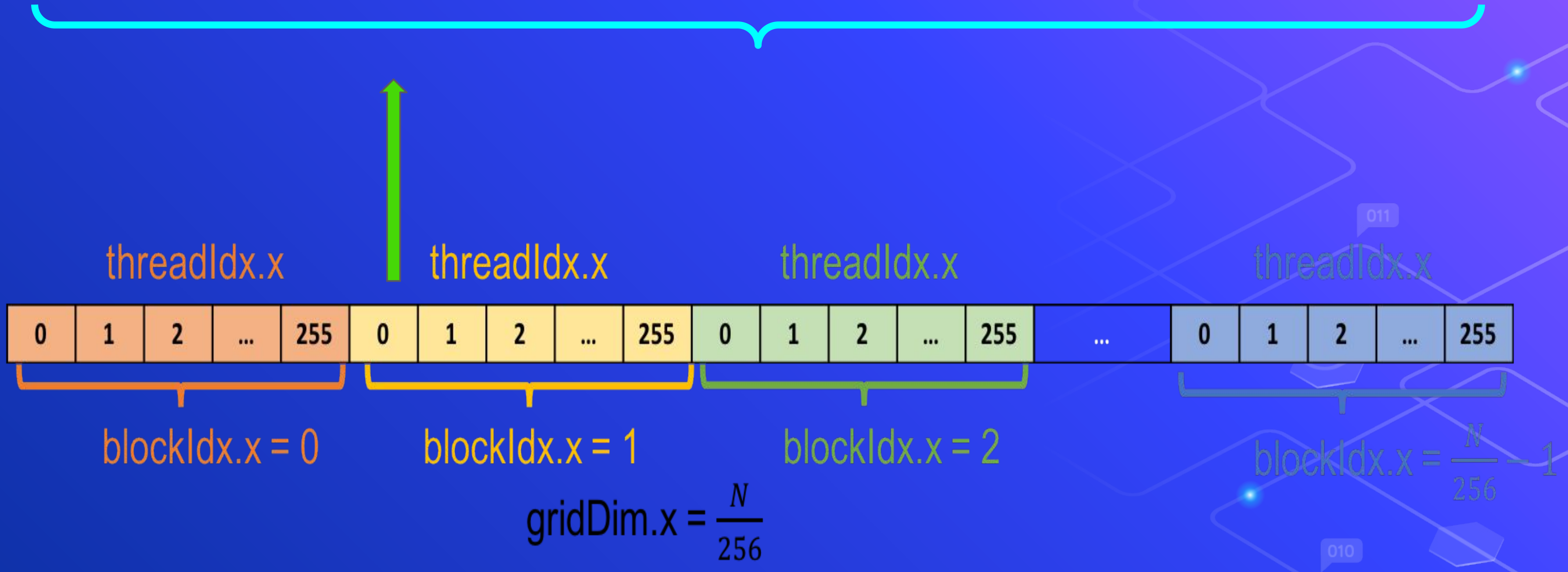
★ dim3 blockIdx : Block index within the grid

★ dim3 threadIdx : Thread index within the block

The thread index i is calculated by the following formula :

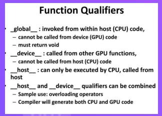
$$i = blockIdx.x * blockDim.x + threadIdx.x$$

index = blockIdx.x * blockDim.x + threadIdx.x = 1 * 256 + 0 = 256



Variable Qualifiers

Example

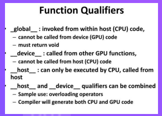

Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

```
__global__
void add(int *a, int *b, int *c, int n) {

    int i = threadIdx.x;

    __shared__ bool sharedArr[100];

    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

int main() {
    int *a, *b, *c;
```

Variable Qualifiers

- __shared__ : accessible by all threads in the same block.
- __device__ : accessible by all threads.

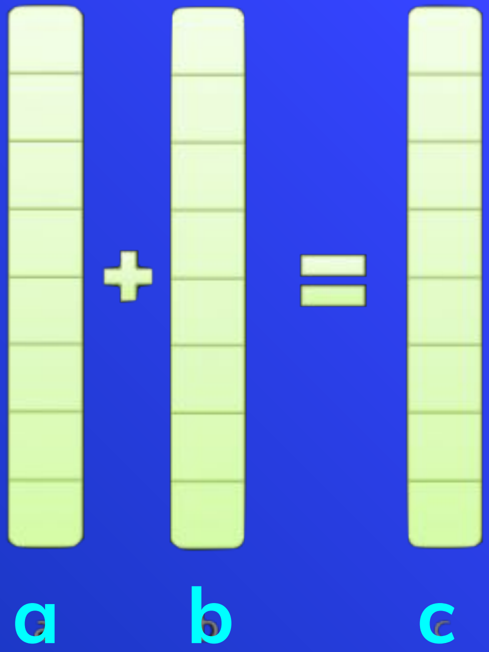


Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

How to use?

Example: **Adding** two vector **a** and **b** resulted in vector **c** with a, b and c have the same size.



No Parallel

```
int main() {
	int *a, *b, *c;
	int SIZE = 1000000;

	// Allocation memory in devices for a,b and c
	cudaMallocManaged(&a, SIZE*sizeof(int));
	cudaMallocManaged(&b, SIZE*sizeof(int));
	cudaMallocManaged(&c, SIZE*sizeof(int));

	// Simple initialization
	for (int i = 0; i < SIZE; i++) {
		a[i] = i;
		b[i] = i;
	}

	// Launch kernel on device
	add<<<1,1>>>(a,b,c,SIZE);

	// Free memory
	cudaFree(a);
	cudaFree(b);
	cudaFree(c);

	return 0;
}
```

```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {

	// Iteration to do addition
	for (int i = 0; i < n; i++)
		c[i] = a[i] + b[i];
}
```

- Using only one block with one thread.
- Using a loop to do addition on each index.

No Parallel

```
==12305== NVPROF is profiling process 12305, command: ./add_normal
==12305== Profiling application: ./add_normal
==12305== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min
 GPU activities:  100.00%  266.64ms         1  266.64ms  266.64ms
      API calls:   50.29%  274.20ms         3  91.401ms  831.23us
                   49.00%  267.15ms         1  267.15ms  267.15ms
                    0.42%  2.3071ms         1  2.3071ms  2.3071ms
                    0.19%  1.0519ms         3  350.62us  321.01us
                    0.04%  236.00us        96  2.4580us     191ns
                    0.04%  213.08us         1  213.08us  213.08us
                    0.00%  22.577us         1  22.577us  22.577us
                    0.00%  3.6050us         1  3.6050us  3.6050us
                    0.00%  1.8500us         3     616ns     235ns
                    0.00%  1.2910us         2     645ns     262ns
                    0.00%     339ns         1     339ns     339ns
```

With Parallel

```
int main() {
        int *a, *b, *c;
        int SIZE = 1000000;
        int THREAD_SIZE = 256;

        // Allocation memory in devices for a,b and c
        cudaMallocManaged(&a, SIZE*sizeof(int));
        cudaMallocManaged(&b, SIZE*sizeof(int));
        cudaMallocManaged(&c, SIZE*sizeof(int));

        // Simple initilization
        for (int i = 0; i < SIZE; i++) {
                a[i] = i;
                b[i] = i;
        }

        // Launch kernel function with SIZE/THREAD_SIZE + 1 block
        // Each block has THREAD_SIZE threads
        add<<<SIZE/THREAD_SIZE + 1,THREAD_SIZE>>>(a,b,c,SIZE);

        // Free memory
        cudaFree(a);
        cudaFree(b);
        cudaFree(c);

        return 0;
}
```

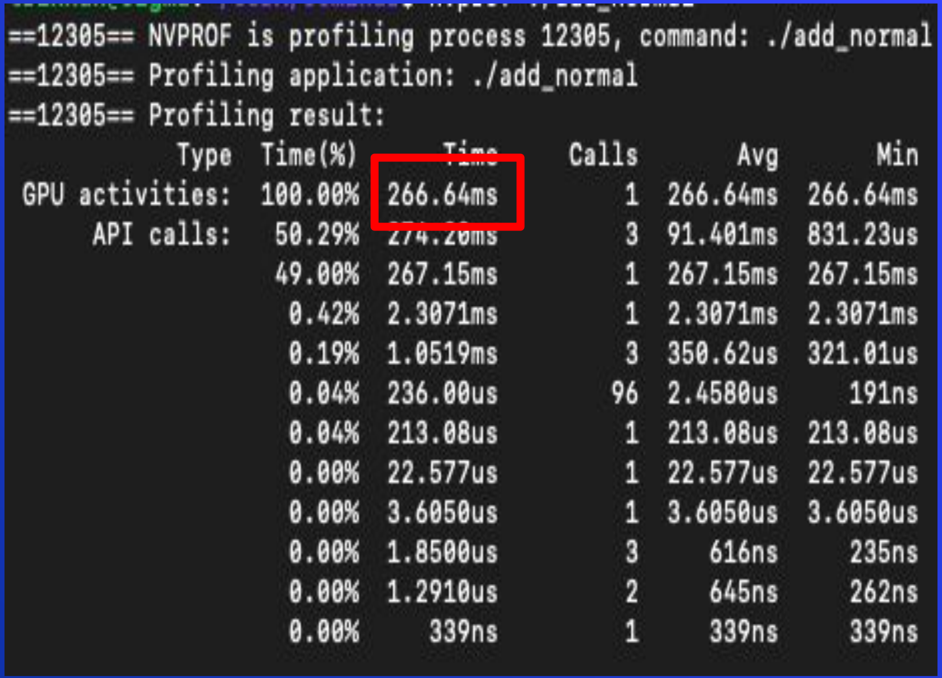
```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {
        // Get thread index
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        if (index < n)
                c[index] = a[index] + b[index];
}
```

- Using only THREAD_SIZE block, each block has SIZE/THREAD_SIZE + 1 threads.
- Each thread we do addition on one index.

With Parallel

```
==11831== NVPROF is profiling process 11831, command: ./add_parallel
==11831== Profiling application: ./add_parallel
==11831== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min
 GPU activities:  100.00%  57.537us         1  57.537us  57.537us
      API calls:   98.42%  262.15ms         3  87.383ms  631.43us
                    0.95%  2.5211ms         1  2.5211ms  2.5211ms
                    0.39%  1.0363ms         3  345.43us  300.18us
                    0.09%  249.13us        96  2.5950us     192ns
                    0.08%  204.57us         1  204.57us  204.57us
                    0.05%  145.72us         1  145.72us  145.72us
                    0.01%  31.471us         1  31.471us  31.471us
                    0.00%  4.1350us         1  4.1350us  4.1350us
                    0.00%  2.2310us         3     743ns     227ns
                    0.00%  1.4950us         2     747ns     402ns
                    0.00%     324ns         1     324ns     324ns
```

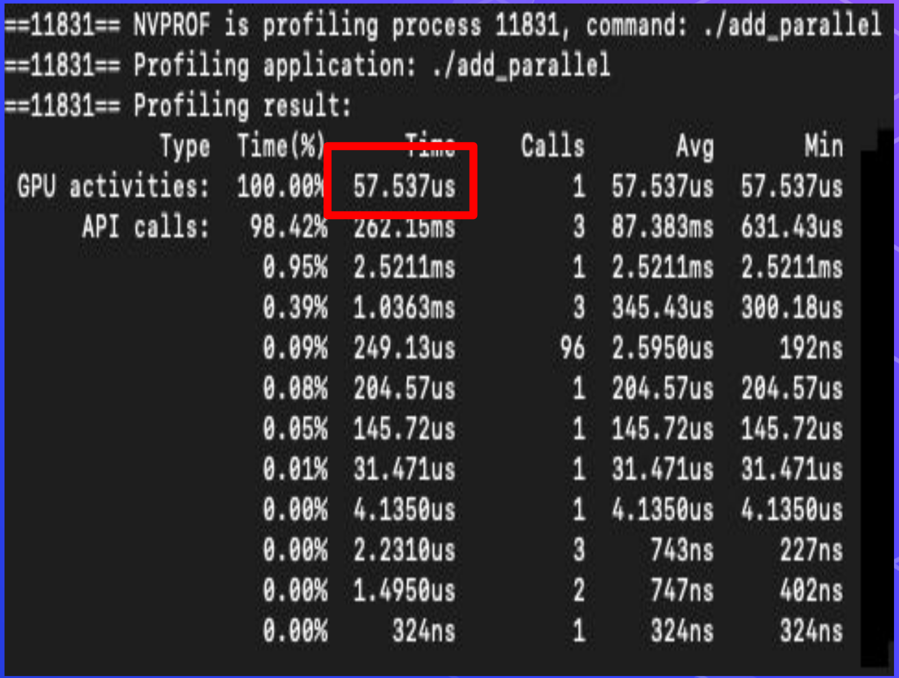
No Parallel



==12305== NVPROF is profiling process 12305, command: ./add_normal
==12305== Profiling application: ./add_normal
==12305== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min
GPU activities:	100.00%	266.64ms	1	266.64ms	266.64ms
API calls:	50.29%	274.20ms	3	91.401ms	831.23us
	49.00%	267.15ms	1	267.15ms	267.15ms
	0.42%	2.3071ms	1	2.3071ms	2.3071ms
	0.19%	1.0519ms	3	350.62us	321.01us
	0.04%	236.00us	96	2.4580us	191ns
	0.04%	213.08us	1	213.08us	213.08us
	0.00%	22.577us	1	22.577us	22.577us
	0.00%	3.6050us	1	3.6050us	3.6050us
	0.00%	1.8500us	3	616ns	235ns
	0.00%	1.2910us	2	645ns	262ns
	0.00%	339ns	1	339ns	339ns

With Parallel



==11831== NVPROF is profiling process 11831, command: ./add_parallel
==11831== Profiling application: ./add_parallel
==11831== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min
GPU activities:	100.00%	57.537us	1	57.537us	57.537us
API calls:	98.42%	262.15ms	3	87.383ms	631.43us
	0.95%	2.5211ms	1	2.5211ms	2.5211ms
	0.39%	1.0363ms	3	345.43us	300.18us
	0.09%	249.13us	96	2.5950us	192ns
	0.08%	204.57us	1	204.57us	204.57us
	0.05%	145.72us	1	145.72us	145.72us
	0.01%	31.471us	1	31.471us	31.471us
	0.00%	4.1350us	1	4.1350us	4.1350us
	0.00%	2.2310us	3	743ns	227ns
	0.00%	1.4950us	2	747ns	402ns
	0.00%	324ns	1	324ns	324ns

No Parallel

```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {

	// Interation to addition
	for (int i = 0; i < n; i++)
		c[i] = a[i] + b[i];
}

int main() {
	int *a, *b, *c;
	int SIZE = 1000000;

	// Allocation memory in devies for a,b and c
	cudaMallocManaged(&a, SIZE*sizeof(int));
	cudaMallocManaged(&b, SIZE*sizeof(int));
	cudaMallocManaged(&c, SIZE*sizeof(int));

	// SImple initilization
	for (int i = 0; i < SIZE; i++) {
		a[i] = i;
		b[i] = i;
	}

	// Launch kernel function with one block with one thread
	add<<<1,1>>>(a,b,c,SIZE);

	// Free memory
	cudaFree(a);
	cudaFree(b);
	cudaFree(c);

	return 0;
}
```

With Parallel

```
#include <stdio.h>

__global__
void add(int *a, int *b, int *c, int n) {
	// Get thread index
	int index = threadIdx.x + blockIdx.x * blockDim.x;
	if (index < n)
		c[index] = a[index] + b[index];
}

int main() {
	int *a, *b, *c;
	int SIZE = 1000000;
	int THREAD_SIZE = 256;

	// Allocation memory in devices for a,b and c
	cudaMallocManaged(&a, SIZE*sizeof(int));
	cudaMallocManaged(&b, SIZE*sizeof(int));
	cudaMallocManaged(&c, SIZE*sizeof(int));

	// Simple initilization
	for (int i = 0; i < SIZE; i++) {
		a[i] = i;
		b[i] = i;
	}

	// Launch kernel function with SIZE/THREAD_SIZE + 1 bloc
	// Each block has THREAD_SIZE threads
	add<<<SIZE/THREAD_SIZE + 1,THREAD_SIZE>>>(a,b,c,SIZE);

	// Free memory
	cudaFree(a);
	cudaFree(b);
	cudaFree(c);

	return 0;
}
```

Experiment: runtime: 126x

	Plus	Quadratic
GPU one thread	0.266s	
GPU Parallel	0.000057s	
CPU	0.0072s	

Thanks!

Nguyễn Minh Trí
Đỗ Trí Nhân

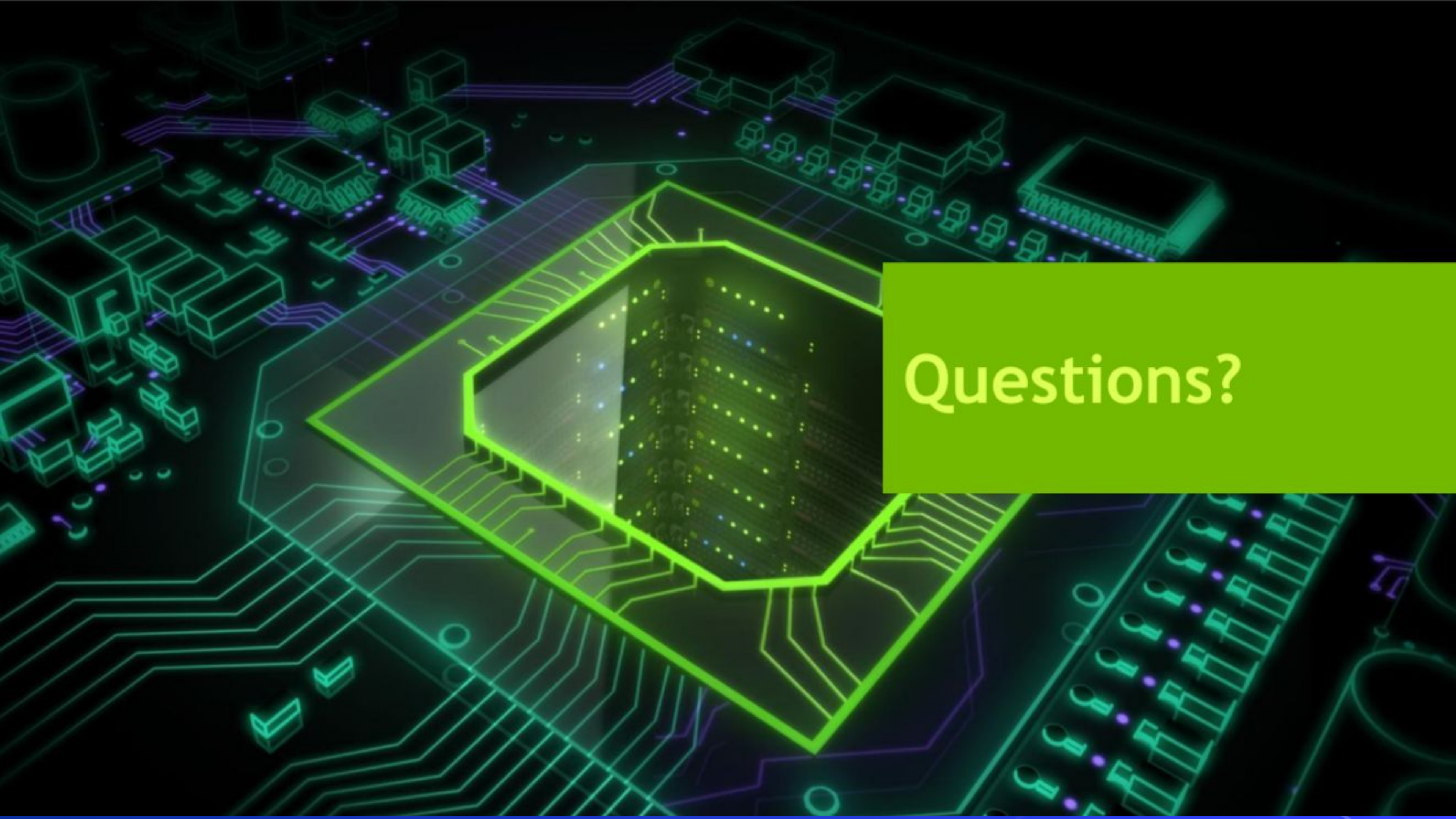
References

Ref

An Introduction to CUDA Programming
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
https://stackoverflow.com/questions/5211746/what-is-cuda-like-what-is-it-for-what-are-the-benefits-and-how-to-start
https://www.tutorialspoint.com/cuda/cuda_key_concepts.htm
https://www.tutorialspoint.com/cuda/cuda_keywords_and_thread_organization.htm
https://developer.nvidia.com/cuda-faq
https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
https://www.sciencedirect.com/science/article/pii/S0167819119301759
https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016#index/2016/1/1/1/1/1/50/1/50/1/50/1/30/1/30/1/30/1/20/1/20/1/5/1/5/1/20/1/100/
https://medium.com/@inaccel/cpu-gpu-fpga-or-tpu-which-one-to-choose-for-my-machine-learning-training-948902f058e0
http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf
http://proceeding.vap.ac.vn/index.php/proceedingvap/article/view/000214/203
https://www.slideserve.com/ryu/cuda-programming
https://www.myzhar.com/blog/tutorials/tutorial-nvidia-gpu-cuda-compute-capability/
https://books.google.com.vn/books?id=CYxjDwAAQBAJ&pg=PA7&lpg=PA7&dq=pytorch+is+wrapper+of+cuda&source=bl&ots=43Smx_h6jw&sig=ACfU3U20lxlu8I4kSDSTS5RGQEFk6QMnuQ&hl=en&sa=X&ved=2ahUKEwiz_tLSqa7pAhXOzIsBHU55DCMQ6AEwD3oECBIQAQ#v=onepage&q=pytorch%20is%20wrapper%20of%20cuda&f=false
http://www.c4learn.com/cplusplus/cpp-variable-naming/
https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
https://tatourian.blog/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/

Main

https://www.yumpu.com/en/document/read/50976433/cuda-parallel-programming-tutorial
http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf
https://www.slideserve.com/ryu/cuda-programming
https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

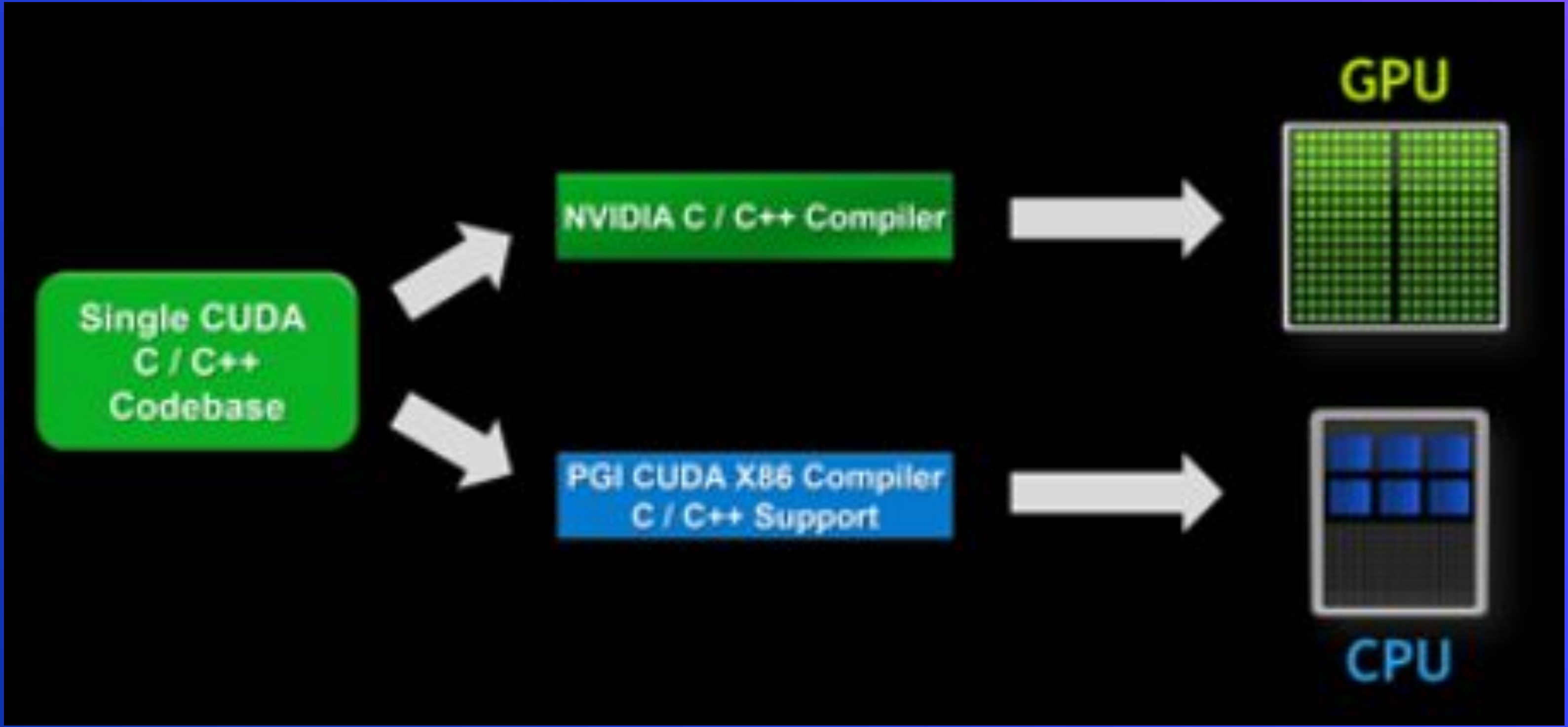


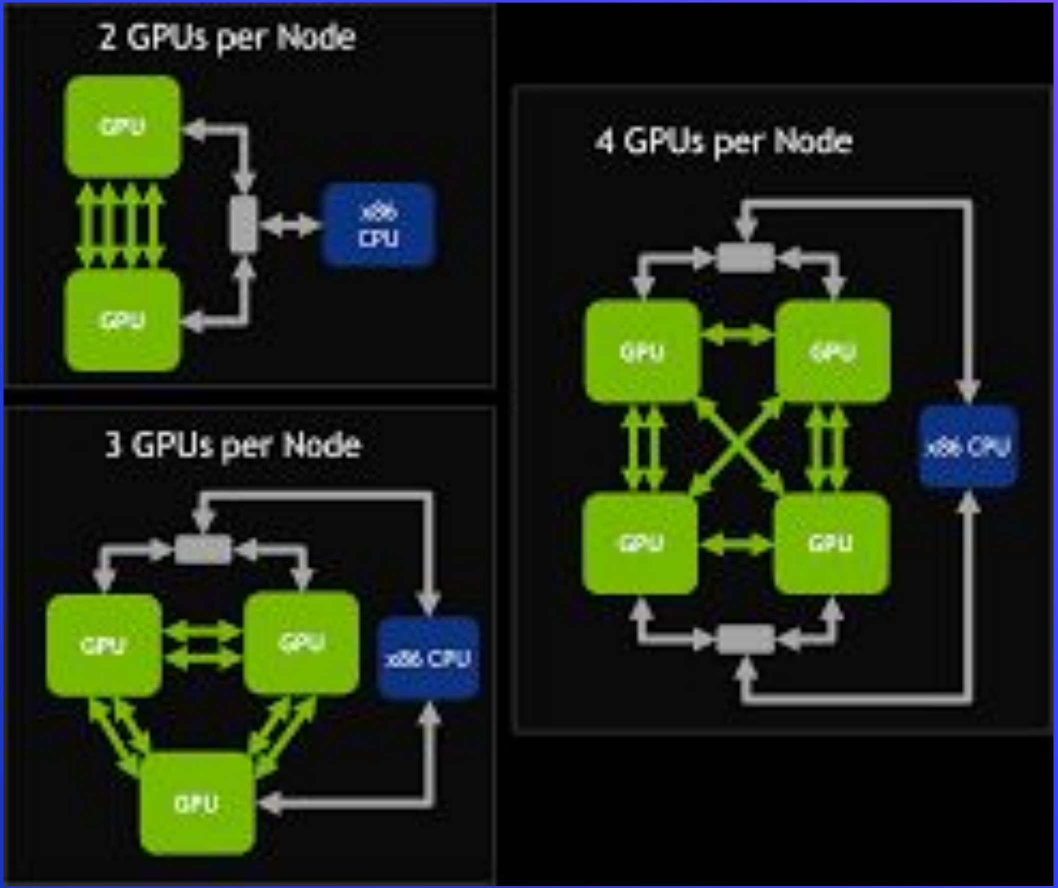
Questions?

Solving 1000 quadratic equations

Future an Advance of Parallel Programming

CUDA compiler - Future (option CPU or GPU)

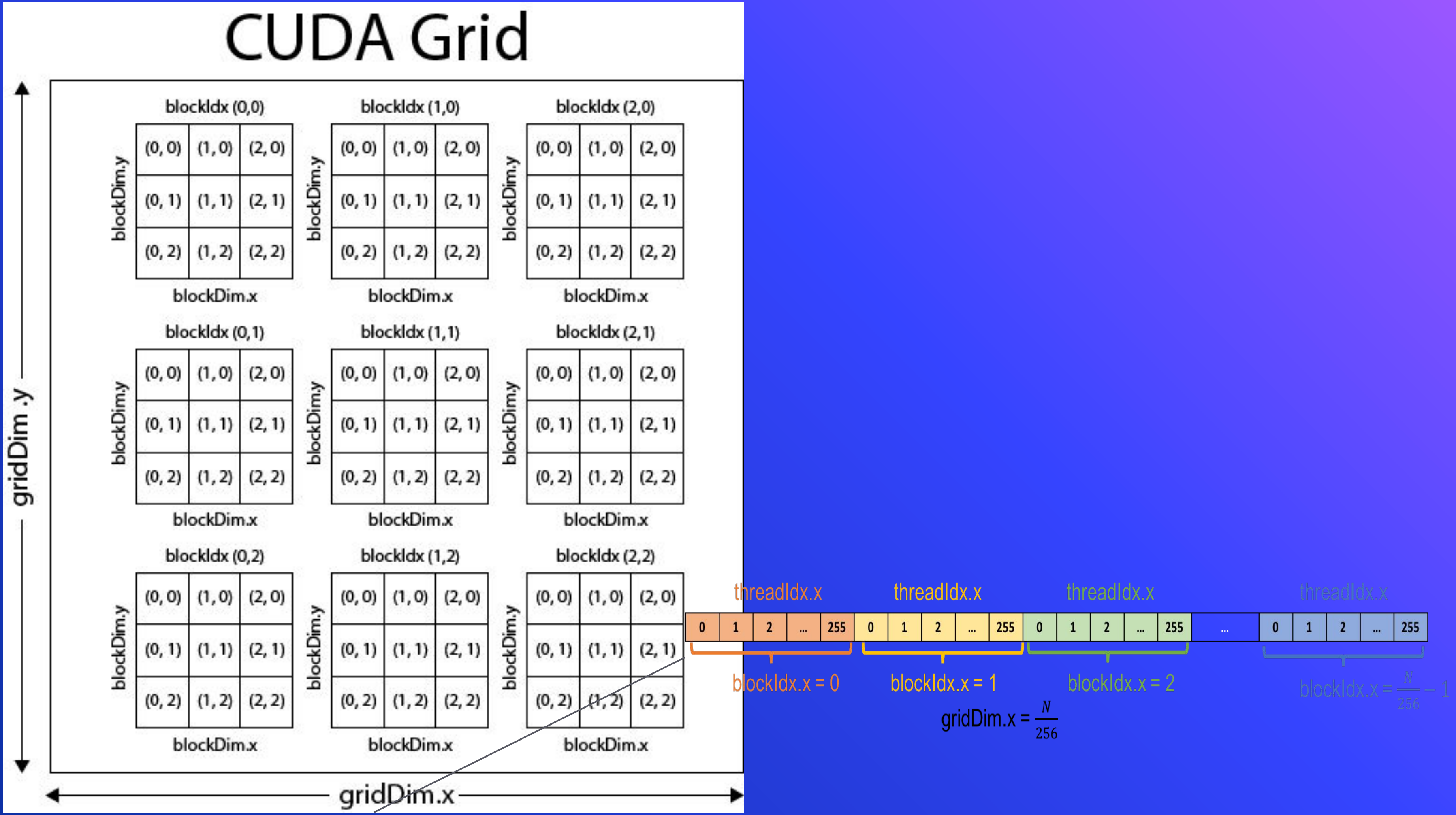



2 GPUs per Node
GPU
GPU
x86
CPU
3 GPUs per Node
GPU
GPU
GPU
x86 CPU
4 GPUs per Node
GPU
GPU
GPU
GPU
x86 CPU

CPU
- Small models
- Small datasets
- Useful for design space exploration

GPU
- Medium-to-large models, datasets
- Image, video processing
- Application on CUDA or OpenCL

TPU
- Matrix computations
- Dense vector processing
- No custom TensorFlow operations


CUDA Grid
gridDim.y
gridDim.x
blockIdx (0,0)
blockIdx (1,0)
blockIdx (2,0)
blockIdx (0,1)
blockIdx (1,1)
blockIdx (2,1)
blockIdx (0,2)
blockIdx (1,2)
blockIdx (2,2)
blockDim.y
blockDim.x
(0, 0) (1, 0) (2, 0)
(0, 1) (1, 1) (2, 1)
(0, 2) (1, 2) (2, 2)
threadIdx.x
0 1 2 ... 255
blockIdx.x = 0
blockIdx.x = 1
blockIdx.x = 2
...
$blockIdx.x = \frac{N}{256} - 1$
$gridDim.x = \frac{N}{256}$

NVIDIA CUDA

Open CL

Sample Code

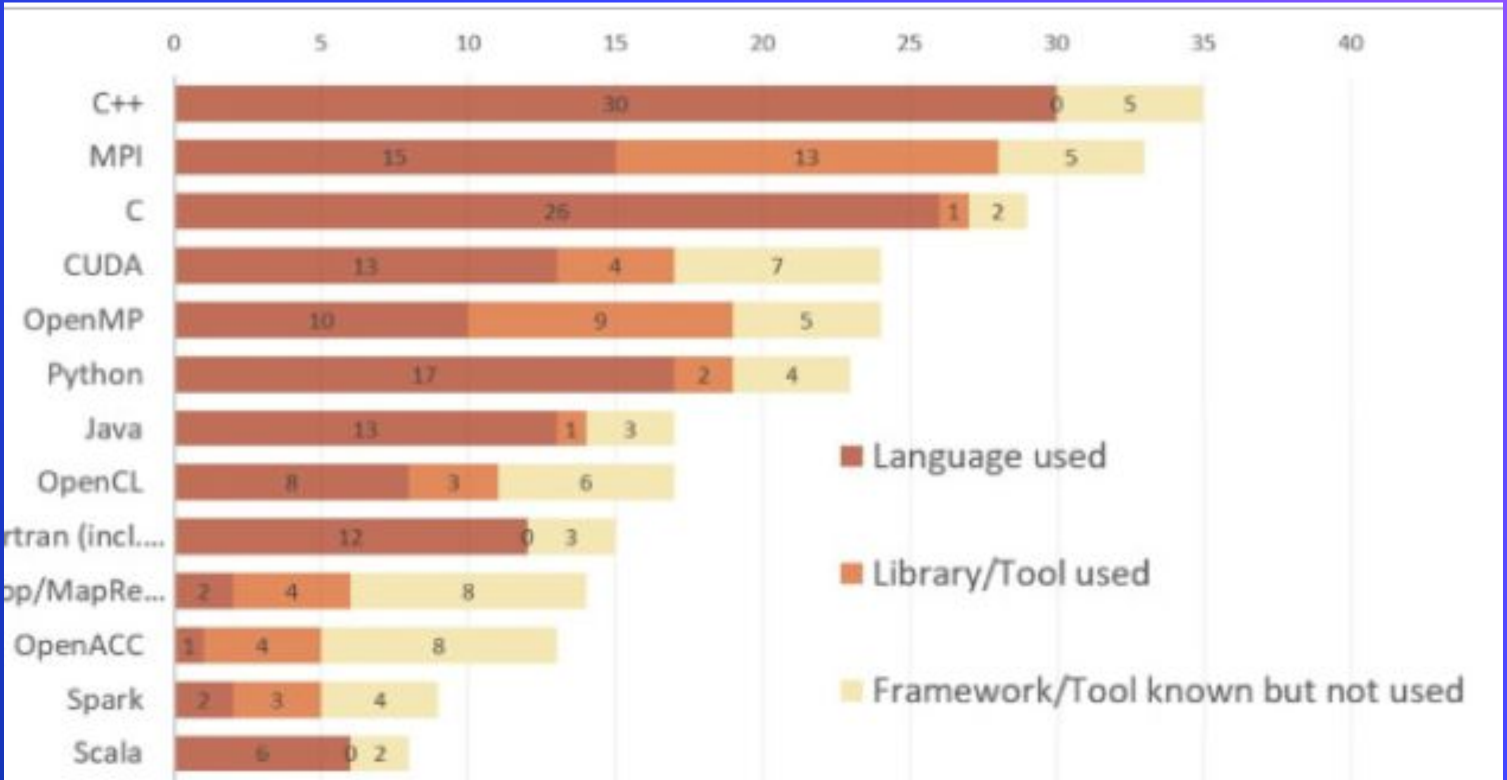
Example: Increment Array Elements

CPU program

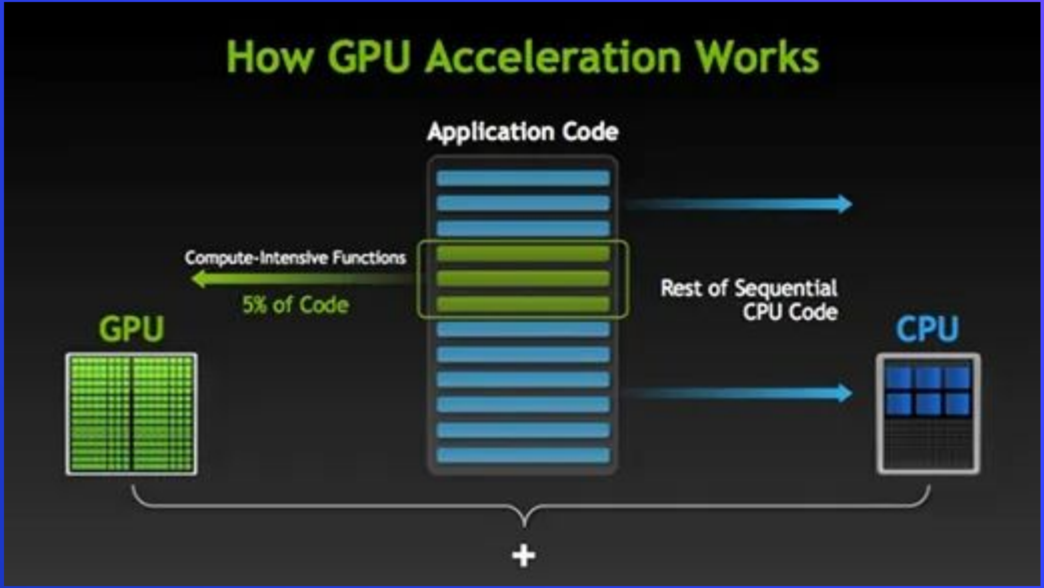
```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


void main()
{
    .....
        increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < N)
            a[idx] = a[idx] + b;
}


void main()
{
    ...
        dim3 dimBlock (blocksize);
        dim3 dimGrid( ceil( N / (float)blocksize) );
        increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```



https://www.sciencedirect.com/science/article/pii/S0167819119301759


How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential
CPU Code
CPU
+