Giới thiệu môn học

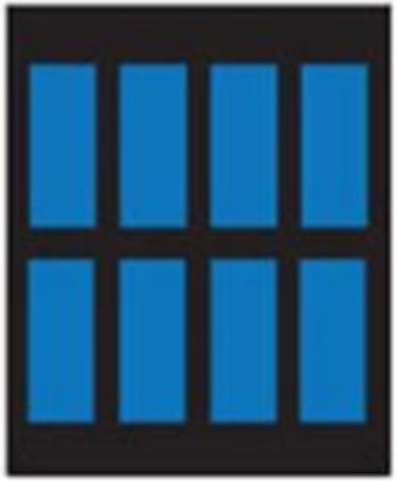
Trần Trung Kiên
ttkien@fit.hcmus.edu.vn

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

CPU vs GPU

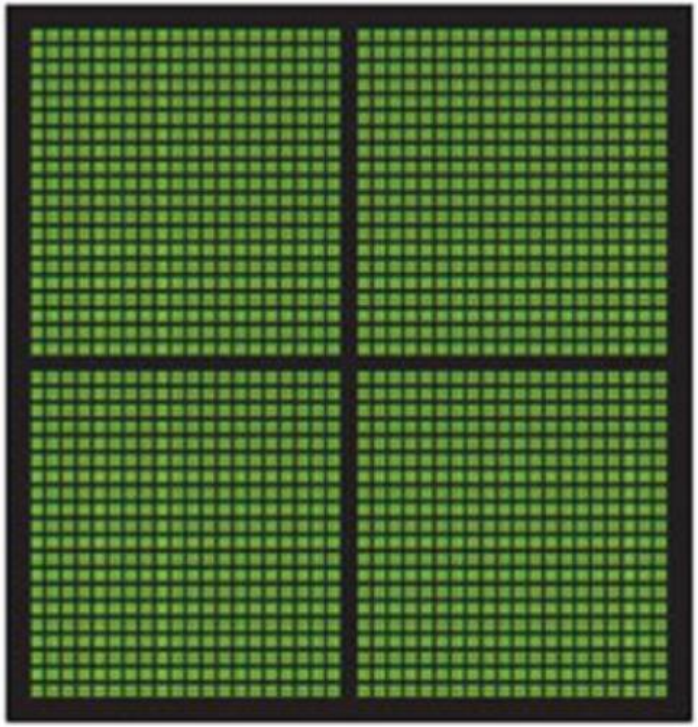
CPU

Có một vài core, mỗi core mạnh và phức tạp



GPU

Có rất rất nhiều core, mỗi core yếu và đơn giản



Video minh họa

Nguồn ảnh: http://www.nvidia.com/object/what-is-gpu-computing.html

CPU

Có một vài core, mỗi core mạnh và phức tạp

Tập trung tối ưu hóa độ trễ (latency); độ trễ = thời gian hoàn thành một công việc

GPU

Có rất rất nhiều core, mỗi core yếu và đơn giản

Tập trung tối ưu hóa băng thông (throughput); băng thông = số lượng công việc hoàn thành trong một đơn vị thời gian

VD: công việc là đưa một người từ địa điểm A đến địa điểm B cách nhau 4500 km

Xe hơi: 2 người, 200 km/h
Độ trễ = ? h
Băng thông = ? người/h

Xe bus: 40 người, 50 km/h
Độ trễ = ? h
Băng thông = ? người/h

CPU

Có một vài core, mỗi core mạnh và phức tạp

Tập trung tối ưu hóa độ trễ (latency); độ trễ = thời gian hoàn thành một công việc

GPU

Có rất rất nhiều core, mỗi core yếu và đơn giản

Tập trung tối ưu hóa băng thông (throughput); băng thông = số lượng công việc hoàn thành trong một đơn vị thời gian

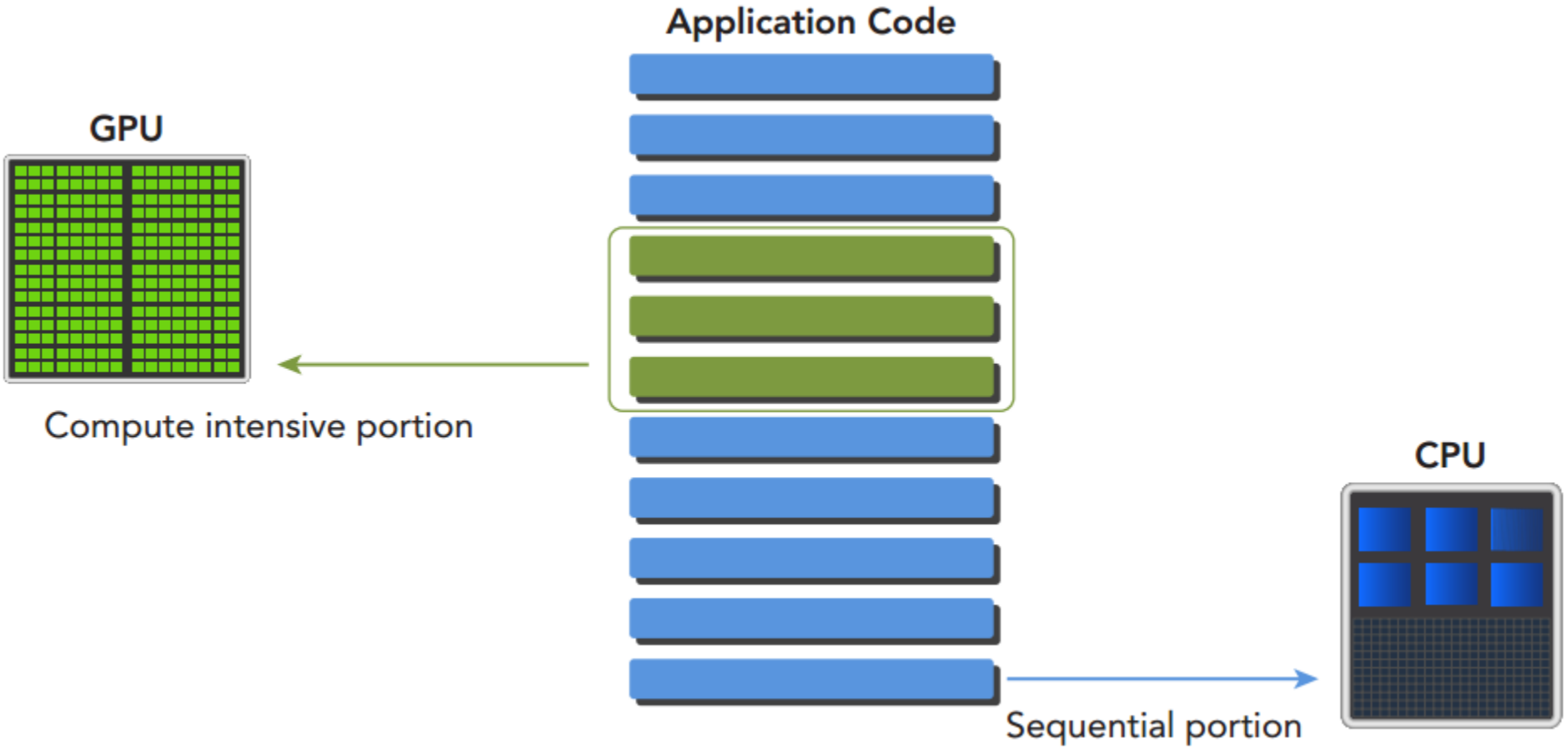
VD: công việc là đưa một người từ địa điểm A đến địa điểm B cách nhau 4500 km

Xe hơi: 2 người, 200 km/h
Độ trễ = 4500/200 = **22.5** h
Băng thông=2/22.5=**0.09** người/h

Xe bus: 40 người, 50 km/h
Độ trễ = 4500/50 = **90** h
Băng thông=40/90=**0.44** người/h

Vậy xe nào tốt hơn?

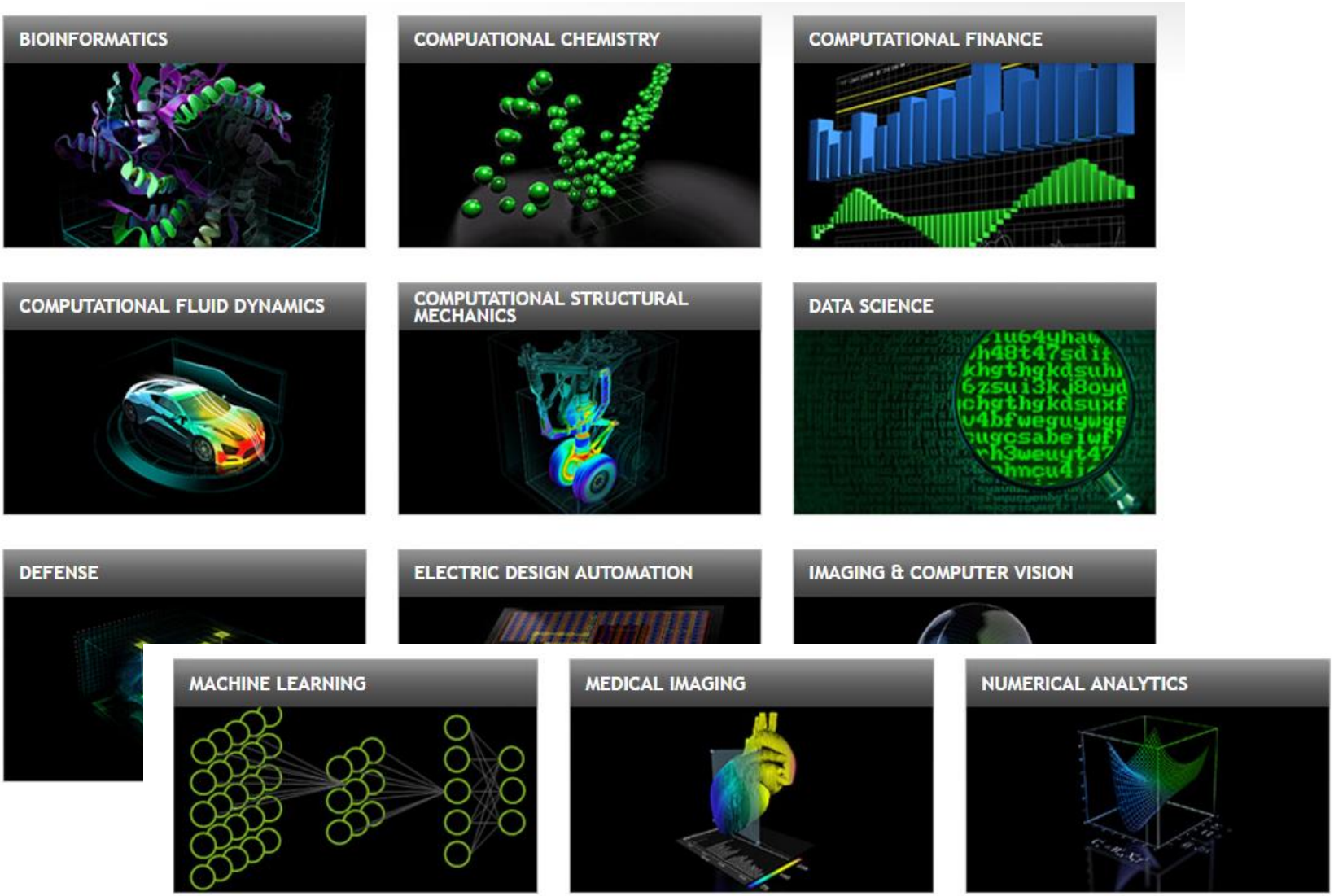
CPU + GPU



CUDA (Compute Unified Device Architecture) **C/C++**: là ngôn ngữ C/C++ được mở rộng, cho phép viết chương trình chạy trên cả CPU và GPU (NVIDIA)

Nguồn ảnh: John Cheng et al. Professional CUDA C Programming. 2014

Các lĩnh vực ứng dụng lập trình song song trên GPU



Nguồn ảnh: http://www.nvidia.com/object/gpu-applications-domain.html

Nội dung môn học:

- Giới thiệu CUDA
- Các dạng tính toán song song thường gặp
- Cách thực thi song song trong CUDA
- Các loại bộ nhớ trong CUDA
- Qui trình tối ưu hóa chương trình CUDA
- Các chủ đề mở rộng (nếu có thời gian)

Sau khi học xong môn học này, SV có thể:

- Cài đặt được chương trình chạy song song trên GPU bằng CUDA
- Vận dụng được cách thực thi song song trong CUDA để tăng tốc chương trình
- Vận dụng được các loại bộ nhớ trong CUDA để tăng tốc chương trình
- Vận dụng được qui trình tối ưu hóa chương trình CUDA
- Vận dụng được kỹ năng làm việc nhóm để hoàn thành các bài tập nhóm trong môn học

Đánh giá môn học

- Các bài tập cá nhân (gồm cả LT & TH) trong suốt quá trình học: 50%
- Đồ án nhóm vấn đáp vào cuối kỳ: 50%

Đánh giá môn học

Nên nhớ mục tiêu chính ở đây là **học, học một cách chân thật**

Bạn có thể thảo luận ý tưởng với bạn khác cũng như là tham khảo các tài liệu, nhưng **code và bài làm phải là của bạn, dựa trên sự hiểu của bạn**

Nếu vi phạm thì sẽ bị 0 điểm cho toàn bộ môn học

Lời khuyên

- Đi học đầy đủ
- Ghi chép bài
- Lấy khó làm niềm vui
- Đơn giản hóa, tập trung vào các việc chính, tránh xa các nguồn gây nhiễu

Các tài liệu tham khảo của môn học

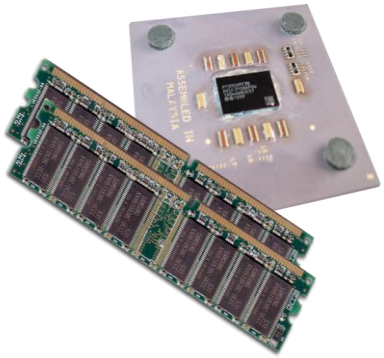
- Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014
- David B. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2016
- Lê Hoài Bắc, Vũ Thanh Hưng, Trần Trung Kiên. *Lập trình song song trên GPU*. NXB KH & KT, 2015
- NVIDIA. *Intro to Parallel Programming*. Udacity
- NVIDIA. *CUDA Toolkit Documentation*

Giới thiệu CUDA C/C++

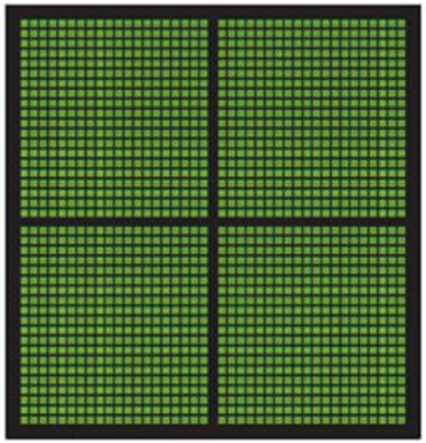
Trần Trung Kiên
ttkien@fit.hcmus.edu.vn

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

CPU vs GPU



CPU
MULTIPLE CORES



GPU
THOUSANDS OF CORES

Tối ưu hóa **độ trễ (latency)**

Tối ưu hóa **băng thông (throughput)**

Nguồn ảnh: http://www.nvidia.com/object/what-is-gpu-computing.html

CUDA (Compute Unified Device Architecture) **C/C++**: là ngôn ngữ C/C++ được mở rộng, cho phép viết chương trình chạy trên CPU (những phần tính toán tuần tự) **+ GPU (những phần tính toán song song)**

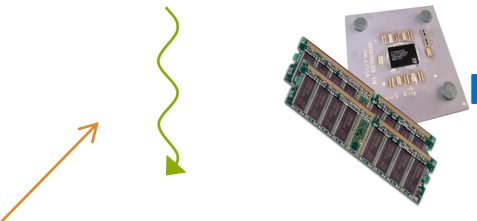
```
#include <iostream>
#include <algorithm>

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

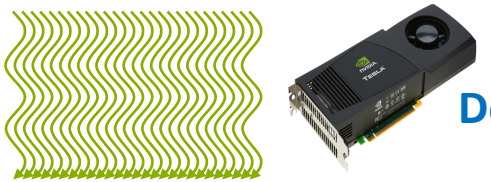
serial code

parallel code

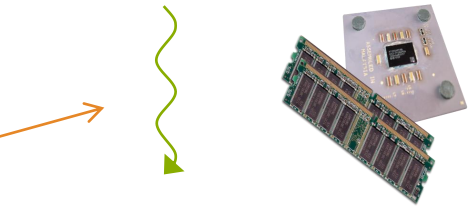
serial code



Host = CPU
(+ bộ nhớ)



Device = GPU



Nguồn ảnh: NVIDIA. CUDA C/C++ Basics

CPU vs GPU

- Host: CPU và bộ nhớ của nó
- Device: GPU và bộ nhớ của nó

Chương trình CUDA đầu tiên

C

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

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

CUDA

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

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

- __global__: hàm sẽ chạy trên device (GPU)
  - Kernel
  - Được gọi từ host (CPU)
- <<<...>>>: cấu hình thực thi

Cài đặt CUDA

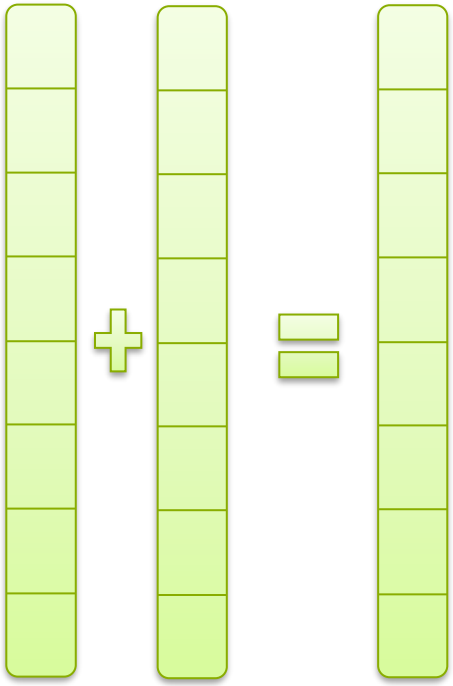
- Đòi hỏi GPU phải hỗ trợ CUDA.
  - https://developer.nvidia.com/cuda-gpus
- Cài CUDA toolkit.
  - https://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html
- Video hướng dẫn cài CUDA trên Windows:
  - https://www.youtube.com/watch?v=cL05xtTocmY

Làm thế nào để chạy CUDA

- Dùng Google Colab
  - https://colab.research.google.com/notebooks/intro.ipynb
- Cấu hình để chạy CUDA trên Google Colab
  - https://medium.com/@harshityadav95/how-to-run-cuda-c-or-c-on-google-colab-or-azure-notebook-ea75a23a5962

Chương trình CUDA đầu tiên: cộng 2 véc-tơ

- Cộng 2 véc-tơ tuần tự bằng host
- Cộng 2 véc-tơ song song bằng device
- Ai thắng?

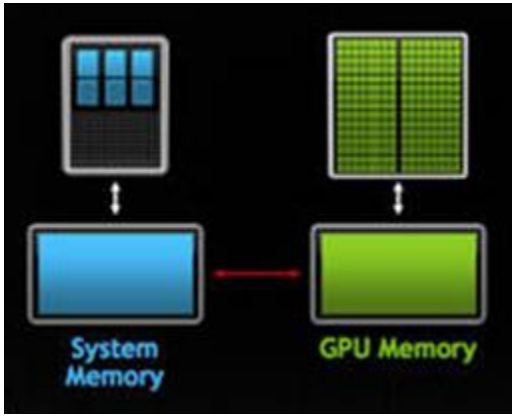


Nguồn ảnh: NVIDIA. CUDA C/C++ Basics

```
int main(int argc, char **argv)
{
   int n; // Vector size
   float *in1, *in2; // Input vectors
   float *out;  // Output vector

   // Input data into n
   …

   // Allocate memories for in1, in2, out
   …

   // Input data into in1, in2
   …

   // Add vectors (on host)
   addVecOnHost(in1, in2, out, n);

   // Free memories
   …

   return 0;
}
```

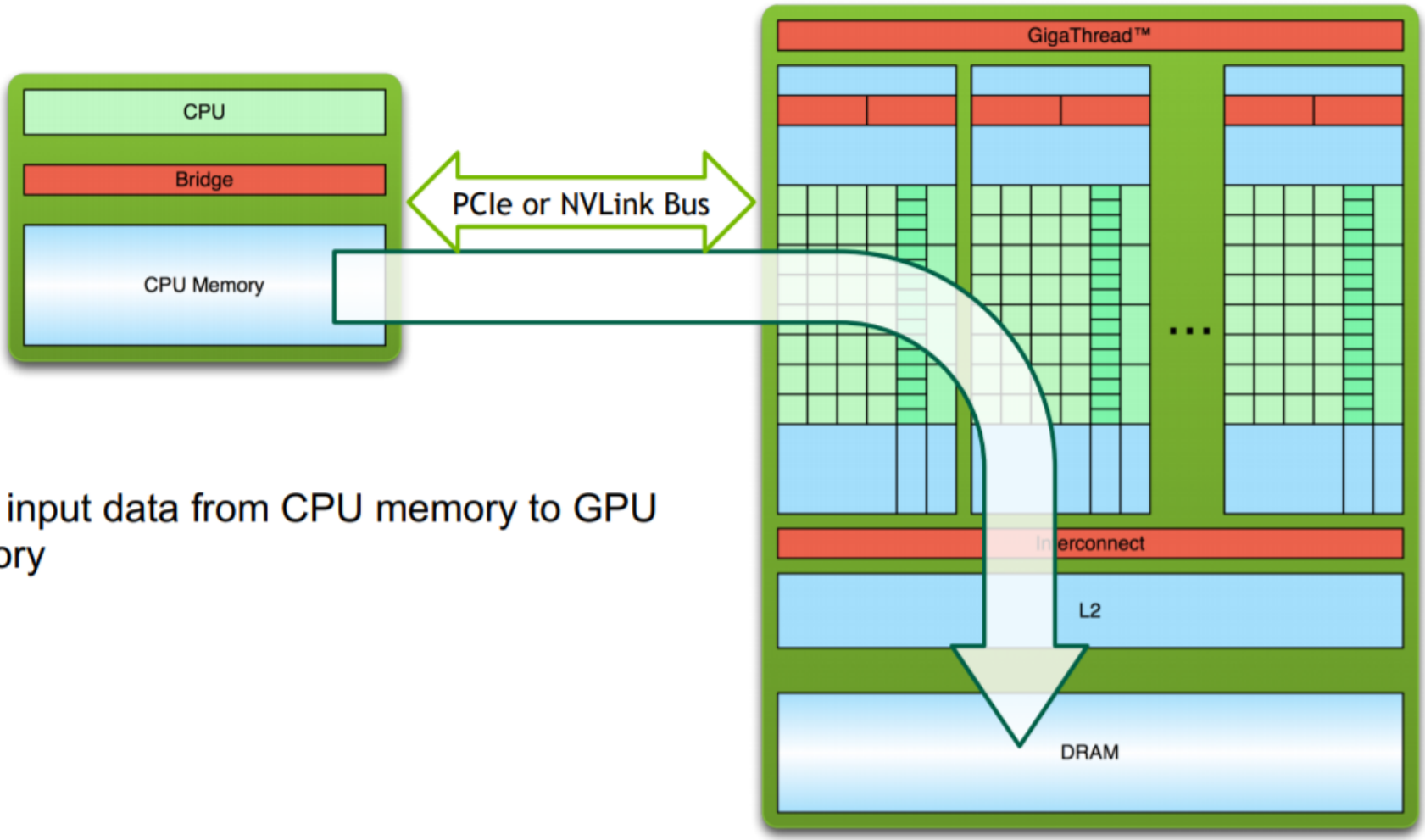
```
void addVecOnHost(float* in1, float* in2, float* out, int n)
{
   for (int i = 0; i < n; i++)
      out[i] = in1[i] + in2[i];
}
```

```
int main(int argc, char **argv)
{
   int n; // Vector size
   float *in1, *in2; // Input vectors
   float *out;  // Output vector

   // Input data into n
   …

   // Allocate memories for in1, in2, out
   …

   // Input data into in1, in2
   …

   // Add vectors (on host)
   addVecOnHost(in1, in2, out, n);

   // Free memories
   …

   return 0;
}
```



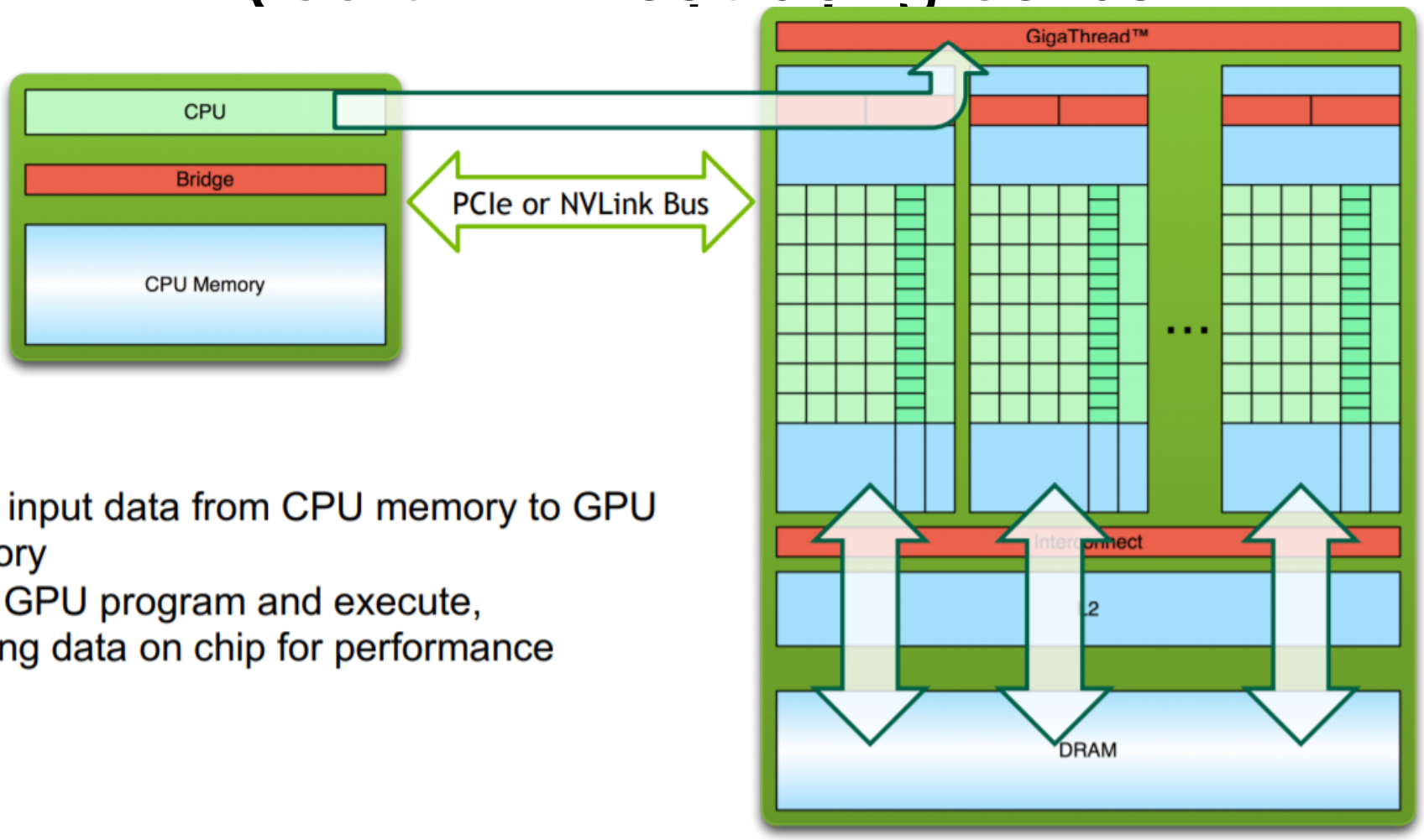

```
// Host allocates memories on device
…

// Host copies data to device memories
…

// Host invokes kernel function to add vectors on device
…

// Host copies result from device memory
…

// Host frees device memories
…
```

Quá trình hoạt động cơ bản



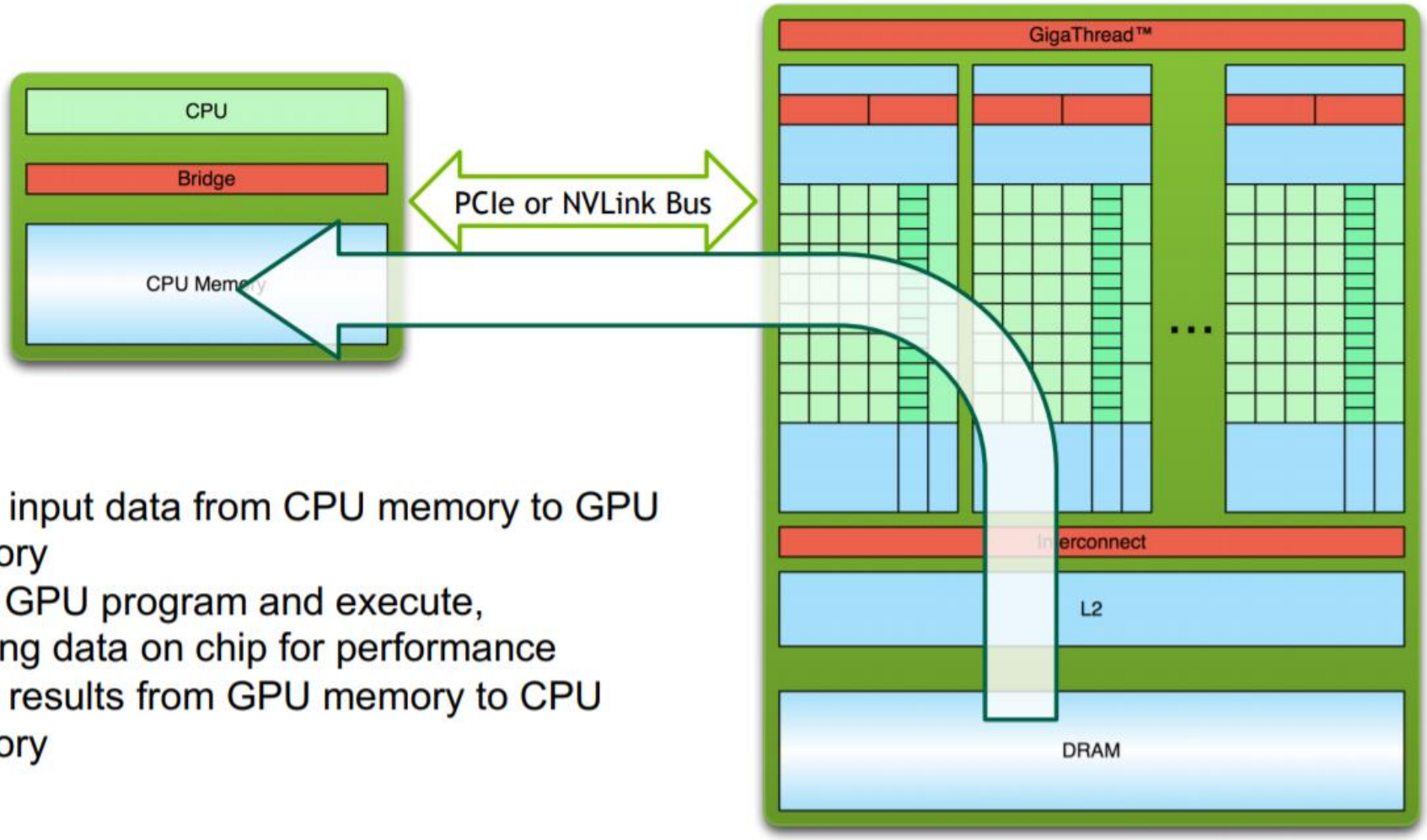
1. Copy input data from CPU memory to GPU memory

Quá trình hoạt động cơ bản



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

Quá trình hoạt động cơ bản



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

Quản lý bộ nhớ

- Bộ nhớ cho host và devide là riêng biệt.
  - Thông thường, ***bộ nhớ devide*** được xử lý bởi devide code và không thể truy xuất được từ host code.
  - Ngược lại cho ***bộ nhớ host***.
- Các hàm xử lý ***bộ nhớ devide***
  - cudaMalloc(), cudaFree(), cudaMemcpy()
  - ***Tương tự,*** malloc(), free(), memcpy()

```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// Host copies data to device memories
…

// Host invokes kernel function to add vectors on device
…

// Host copies result from device memory
…

// Host frees device memories
…
```

```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// Host copies data to device memories
cudaMemcpy(d_in1, in1, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_in2, in2, n * sizeof(float), cudaMemcpyHostToDevice);

// Host invokes kernel function to add vectors on device
...

// Host copies result from device memory
...

// Host frees device memories
...
```

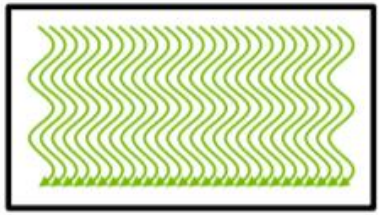
```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// Host copies data to device memories
cudaMemcpy(d_in1, in1, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_in2, in2, n * sizeof(float), cudaMemcpyHostToDevice);

// Host invokes kernel function to add vectors on device
...

// Host copies result from device memory
cudaMemcpy(out, d_out, n * sizeof(float), cudaMemcpyDeviceToHost);

// Host frees device memories
...
```

```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// Host copies data to device memories
cudaMemcpy(d_in1, in1, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_in2, in2, n * sizeof(float), cudaMemcpyHostToDevice);

// Host invokes kernel function to add vectors on device
...

// Host copies result from device memory
cudaMemcpy(out, d_out, n * sizeof(float), cudaMemcpyDeviceToHost);

// Host frees device memories
cudaFree(d_in1);
cudaFree(d_in2);
cudaFree(d_out);
```

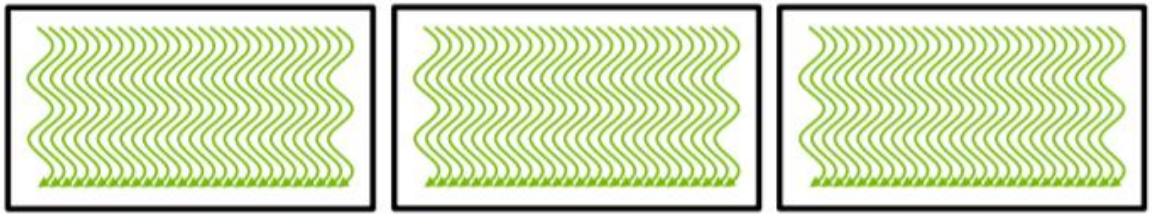
Thread trong CUDA

Thread

Thread Block

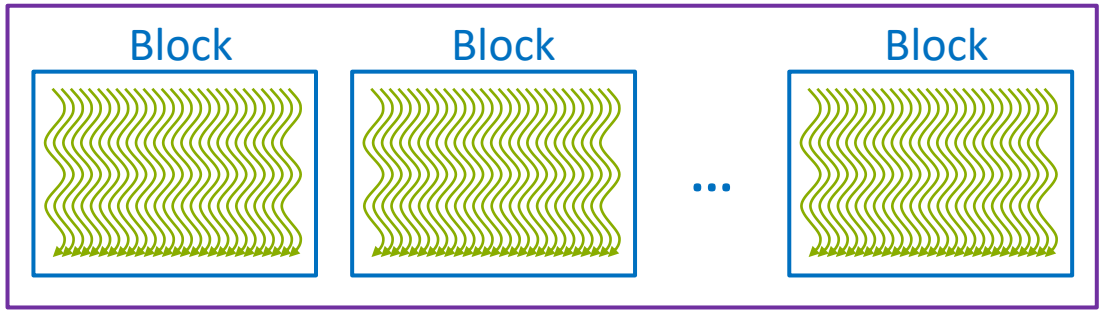


Grid



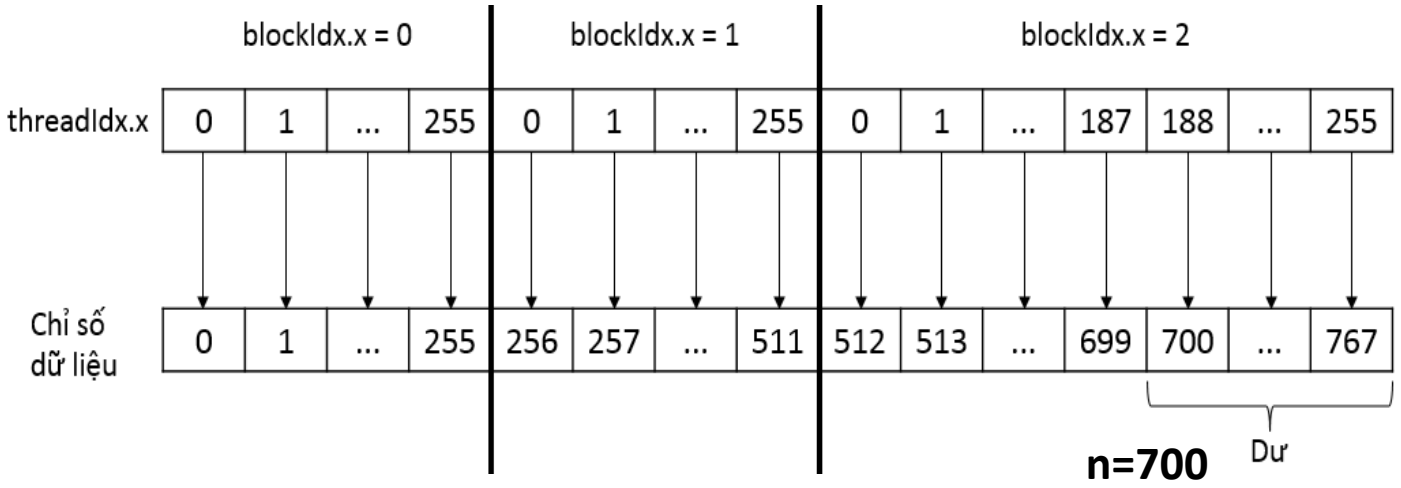
```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// Host copies data to device memories
cudaMemcpy(d_in1, in1, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_in2, in2, n * sizeof(float), cudaMemcpyHostToDevice);

// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
```

Câu lệnh này tạo ra ở device một đống các thread (gọi là một **grid**) cùng thực thi song song hàm addVecOnDevice; các thread này được tổ chức thành các nhóm (**block**) có cùng kích thước



```
...
// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
...
```

```
__global__ void addVecOnDevice(float* in1, float* in2, float* out, int n)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < n)
      out[i] = in1[i] + in2[i];
}
```



BUILT-IN VARIABLES

threadIdx.x: thread index trong một block
blockIdx.x Block index trong grid
blockDim.x : số lượng thread trong một block.
gridDim.x số lượng block trong một grid.
Được tạo tự động
Read only

Tính "không đồng bộ" của hàm kernel

Một tính chất của hàm kernel là **"không đồng bộ" (asynchronous)**: sau khi host gọi hàm kernel ở device, host sẽ được tự do tiếp tục làm các công việc của mình mà không phải chờ hàm kernel ở device thực hiện xong

```
...
// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);

// Host copies result from device memory
cudaMemcpy(out, d_out, n * sizeof(float), cudaMemcpyDeviceToHost); // OK?
```

Tính "không đồng bộ" của hàm kernel

```
…
// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
double start = seconds();
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
double time = seconds() - start; // OK?
…
```

Tính "không đồng bộ" của hàm kernel

```
…
// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
double start = seconds();
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
cudaDeviceSynchronize();
double time = seconds() - start; // OK
…
```

Kiểm lỗi khi gọi các hàm CUDA API

- Nhiều khi có lỗi nhưng chương trình CUDA vẫn chạy bình thường và cho ra kết quả sai!
- Do đó, nên luôn tiến hành kiểm lỗi khi gọi các hàm CUDA API
- Để cho tiện, có để định nghĩa một macro kiểm lỗi:

```
#define CHECK(call)                                                          \
{                                                                            \
   cudaError_t err = call;                                                   \
   if (err != cudaSuccess)                                                   \
   {                                                                         \
      printf("%s in %s at line %d!\n", cudaGetErrorString(err), __FILE__, __LINE__); \
      exit(EXIT_FAILURE);                                                    \
   }                                                                         \
}
```

```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
CHECK(cudaMalloc(&d_in1,  n * sizeof(float)));
CHECK(cudaMalloc(&d_in2,  n * sizeof(float)));
CHECK(cudaMalloc(&d_out,  n * sizeof(float)));

// Host copies data to device memories
CHECK(cudaMemcpy(d_in1, in1, n * sizeof(float), cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_in2, in2, n * sizeof(float), cudaMemcpyHostToDevice));

// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);

// Host copies result from device memory
CHECK(cudaMemcpy(out, d_out, n * sizeof(float), cudaMemcpyDeviceToHost));

// Host frees device memories
CHECK(cudaFree(d_in1));
CHECK(cudaFree(d_in2));
CHECK(cudaFree(d_out));
```

Kiểm lỗi hàm kernel?

Đọc ở đây, mục “Handling CUDA Errors”

Thí nghiệm: host vs device

- Phát sinh ngẫu nhiên giá trị của các véc-tơ đầu vào trong [0, 1]
- So sánh thời gian chạy giữa host (hàm `addVecOnHost`) và device (hàm `addVecOnDevice`, kích thước block là 512) với các kích thước véc-tơ khác nhau
- GPU: GeForce GTX 560 Ti (compute capability 2.1)

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64			

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256			

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256	**0.002**	0.018	0.118

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256	**0.002**	0.018	0.118
1024			

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256	**0.002**	0.018	0.118
1024	**0.006**	0.017	0.347

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256	**0.002**	0.018	0.118
1024	**0.006**	0.017	0.347
4096			

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256	**0.002**	0.018	0.118
1024	**0.006**	0.017	0.347
4096	0.030	**0.017**	1.775

Thí nghiệm: host vs device

Vec size	Host time (ms)	Device time (ms)	Host time / Device time
64	**0.001**	0.040	0.024
256	**0.002**	0.018	0.118
1024	**0.006**	0.017	0.347
4096	0.030	**0.017**	1.775
16384	0.127	**0.017**	7.403
65536	0.516	**0.055**	9.409
262144	1.028	**0.197**	5.220
1048576	3.773	**0.277**	13.619
4194304	13.870	**0.617**	22.479
16777216	55.177	**1.993**	27.683