Parallel Programming

Introduction to CUDA C/C++

Part I

Phạm Trọng Nghĩa

ptnghia@fit.hcmus.edu.vn

Data parallelism

- Question: Why modern software applications run slowly?
- Answer: too much data to process
  - Image-processing apps: million to trillions of pixels
  - Molecular dynamics apps: Thousands to billions of atoms
- Organizing the computation around the data such that we can execute the resulting independent computations in parallel to complete the overall job faster—often much faster.

CUDA C/C++: is extended-C/C++, allows us to write a program running on both CPU (sequential parts) and GPU (massively parallel parts)

```
#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;
}
```

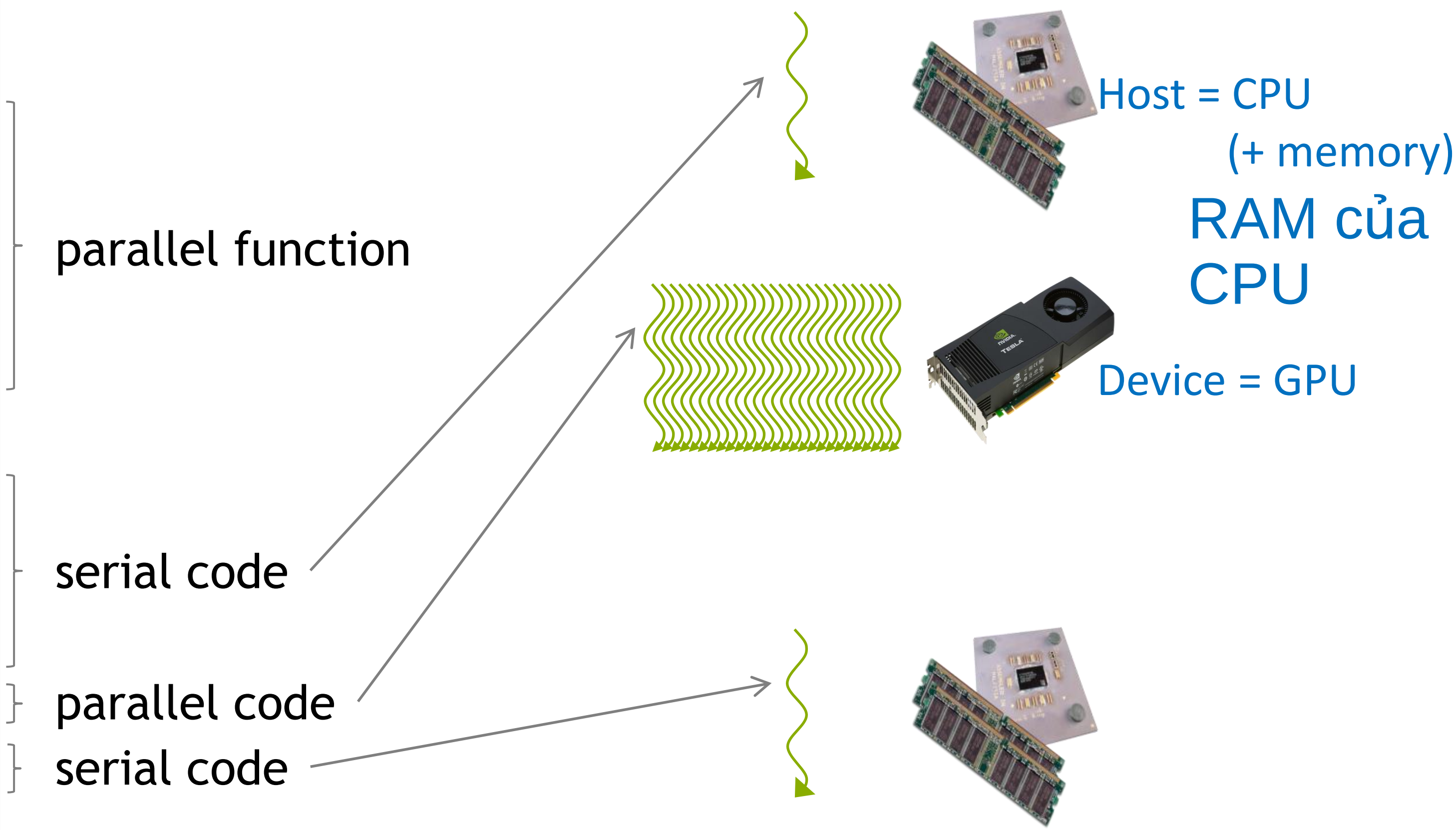


Image source: NVIDIA. CUDA C/C++ Basics

A simple CUDA program: adding 2 vectors

- Adding 2 vectors sequentially using host
- Adding 2 vectors in parallel using device: each thread on device are responsible for computing an element in the sum vector, and all these threads run in parallel
- Who win?

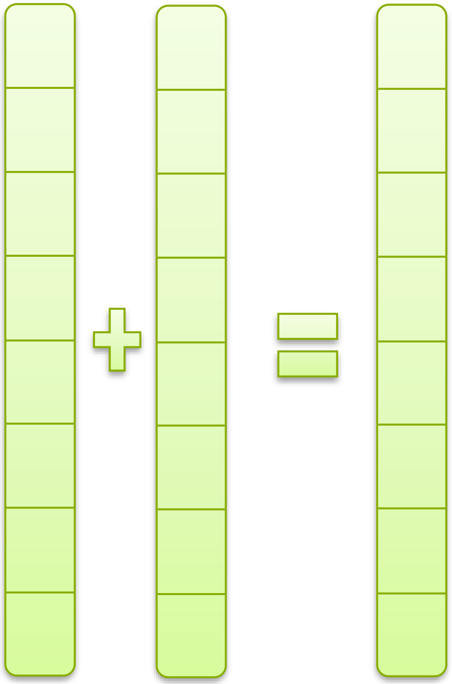
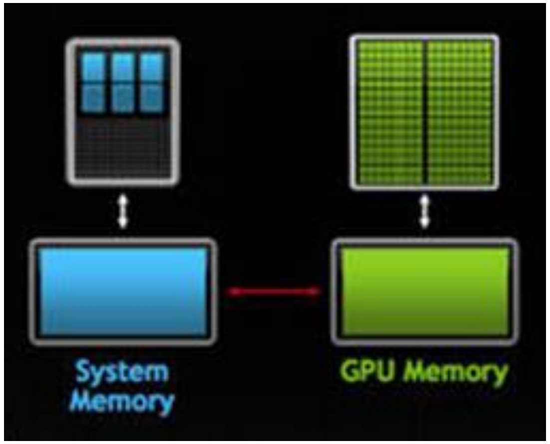


Image source: NVIDIA. CUDA C/C++ Basics

```c
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;
}
```

```c
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
…
```

Image source: Mark Harris. Unified Memory in CUDA 6

```
// 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
...
```

CUDA API do NVIDIA cung cấp
d_*: memory on device

```
// 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
...
```

cờ xác định hướng copy

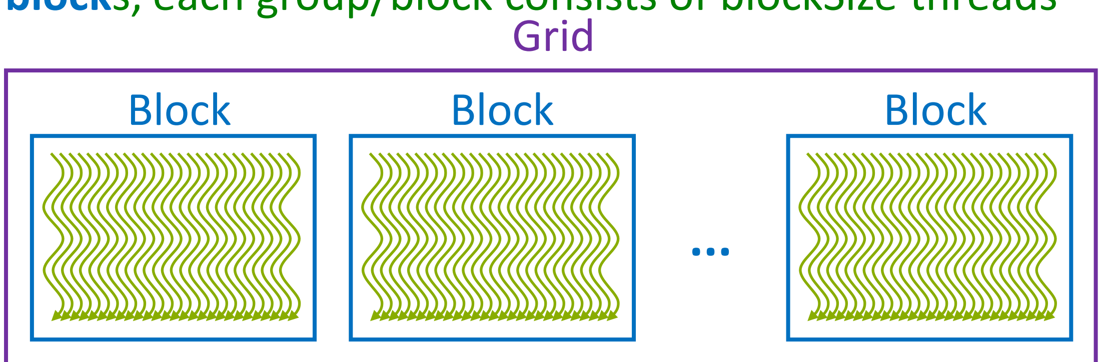
```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// 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);
```

```
// 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); // For simplicity, you can temporarily view blockSize as a number
dim3 gridSize((n - 1) / blockSize.x + 1); // Similarity, view gridSize as a number
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
```

hàm ceiling

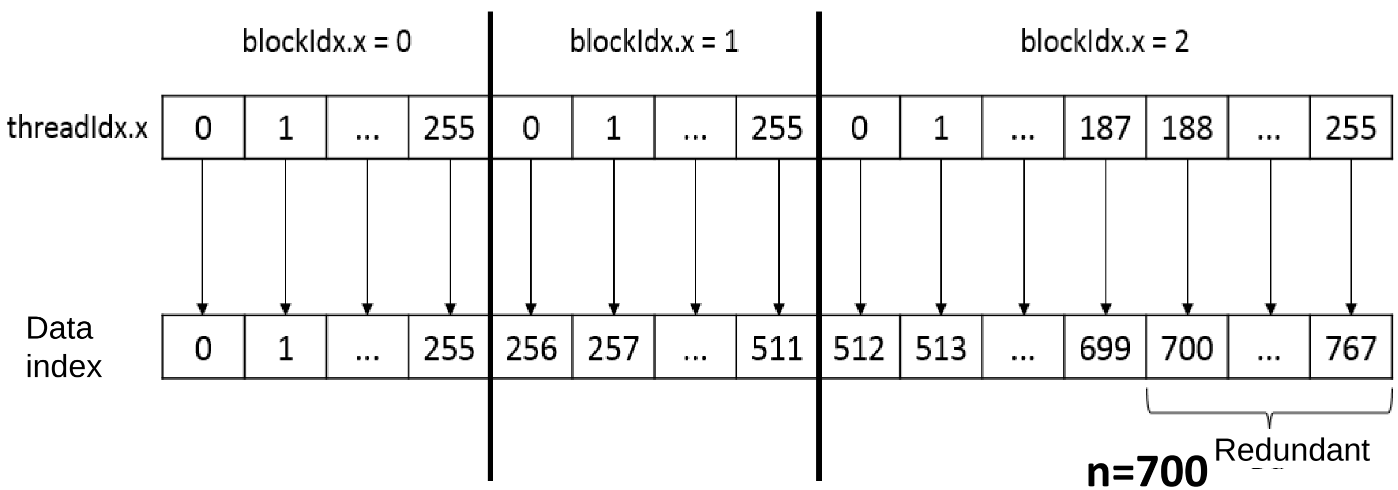
This command creates on device a bunch of threads (called **grid**) executing the addVecOnDevice function in parallel; these threads are organized into gridSize groups or **block**s, each group/block consists of blockSize threads



```
...
// 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);
...
```

Kernel functions must return "void"

```
__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];
}
```



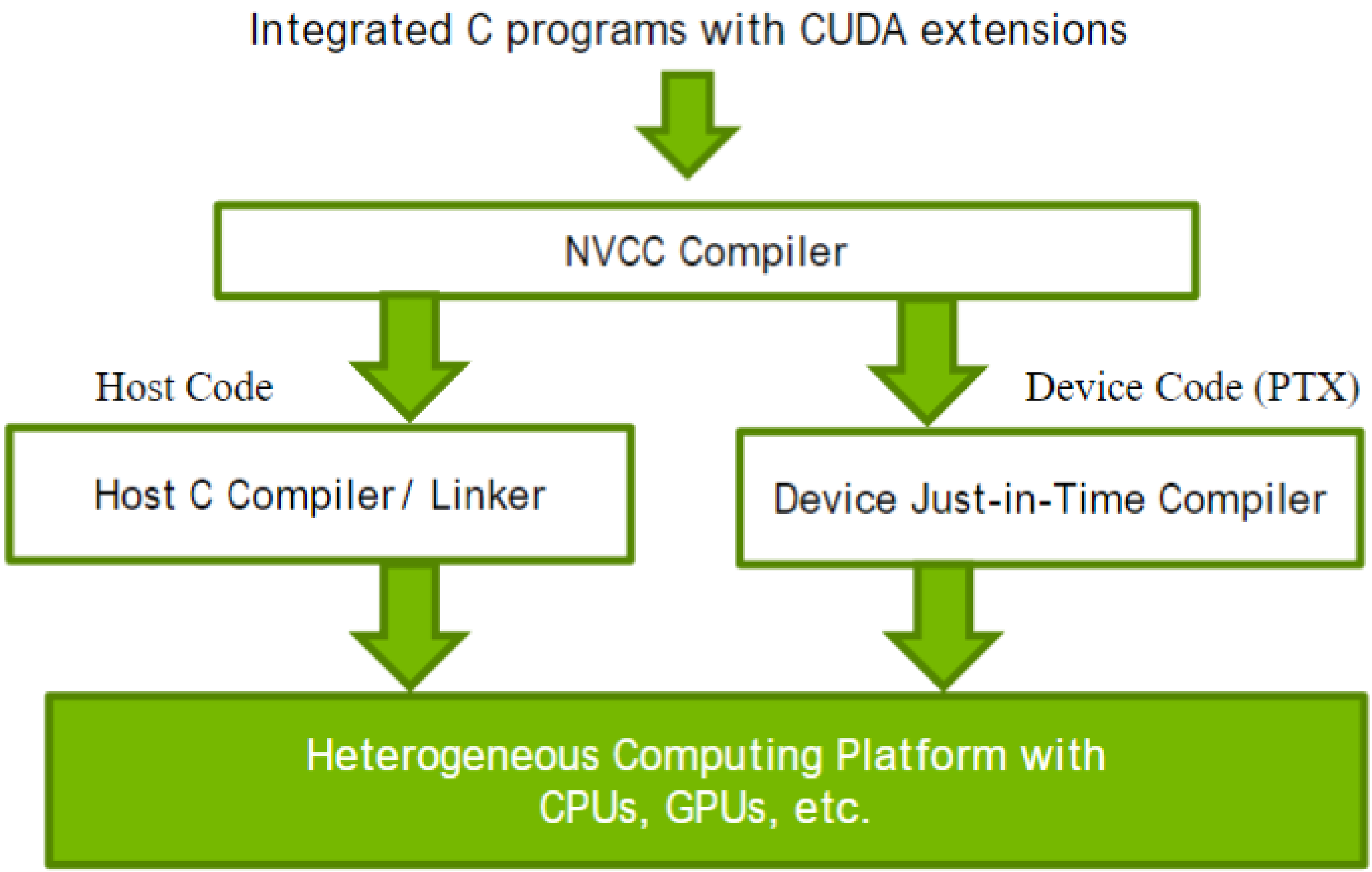
More on CUDA Function Declarations

	Callable from	**Execute on**	**Execute by**
__device__ float DeviceFunc()	device	device	Caller host thread
__global__ void KernelFunc()	host	device	New grid of device thread
__host__ float HostFunc()	host	host	Caller thread device

- __global__ define a kernel function
  - A kernel function must return void
- __device__ and __host__ can be used together
  - Generate two versions of object code for the same function
- __host__ is optional if use alone.

Compiling A CUDA Program

- Use NVCC (NVIDIA C compiler)



Kernel function execution is asynchronous w.r.t host by default

After host calls a kernel function to be executed on device, host will be free to do other works without waiting the kernel to be completed

```
...
// 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?
```

OK, because the cudaMemcpy function forces host to wait until the kernel finishes, only then it starts to copy

Kernel function execution is asynchronous w.r.t host by default

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

Kernel function execution is asynchronous w.r.t host by default

```
...
// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
double start = seconds(); // seconds is my function to get the current time
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
cudaDeviceSynchronize(); // Host waits here until device completes its work
double time = seconds() - start; // ✓
...
```

Error checking when calling CUDA API functions

- It’s possible that an error happens but the CUDA program still run normally and give wrong result

  → don’t know where to fix bug ☹

  → to know where to fix bug, we should always check error when calling CUDA API functions

CUDA API

- For convenience, we can define a macro to check error and wrap it around C

```
#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));
```

Error checking when calling kernel functions?

Read here, “Handling CUDA Errors” section

Experiment: host vs device

- Generate input vectors with random values in [0, 1]
- Compare running time between host (`addVecOnHost` function) and device (`addVecOnDevice` function, block size 512) with different vector sizes
- GPU: GeForce GTX 560 Ti (compute capability 2.1)

Experiment: host vs device

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

Experiment: host vs device

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

Experiment: host vs device

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

Experiment: 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

Experiment: 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			

Experiment: 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

Experiment: 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			

Experiment: 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

Experiment: 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

Reference

- [1] Slides from *Illinois-NVIDIA GPU Teaching Kit*
- [2] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022

THE END