Parallel Programming

Introduction to CUDA C/C++

Part I

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

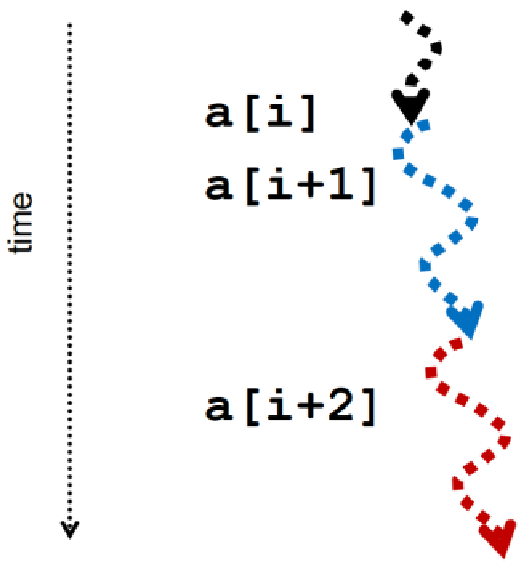
Data parallelism

- 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.

Sequential Execution Model

- One instruction at the time
- Optimizations possible at the machine level

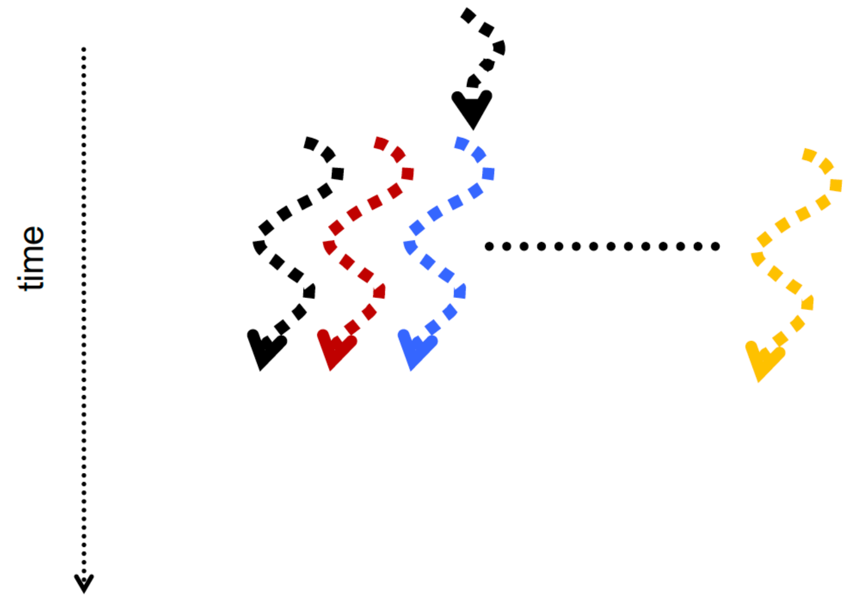
```
int a[N];
for (i = 0; i < N; i++) {
        a[i] = a[i] * 2;
}
```



Parallel Execution Model: SIMD

- SIMD: Single Instruction, Multiple Data
- Some instructions executed concurrently

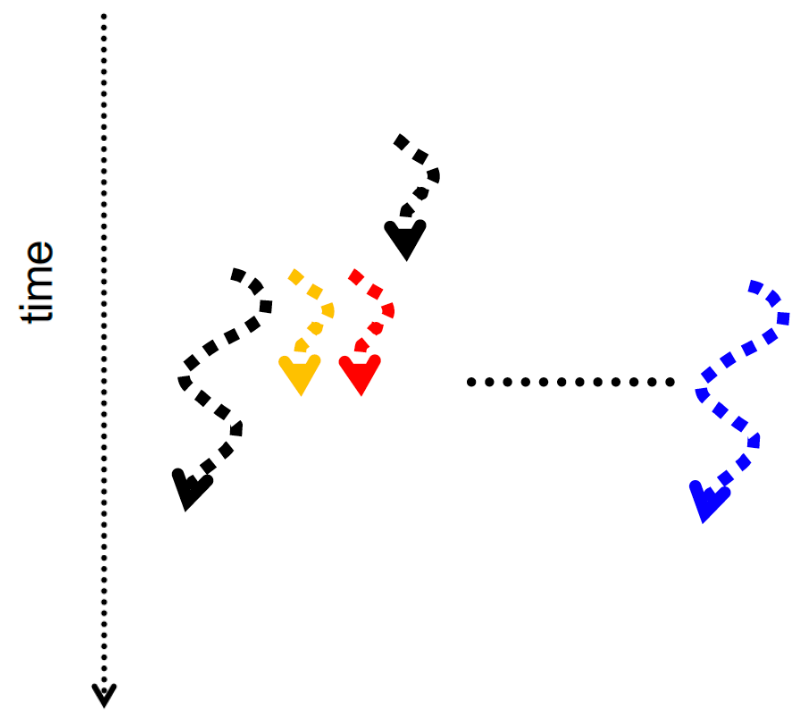
```
int a[N];
for all elements do in parallel{
       a[i] = a[i] * 2;
}
```



Parallel Execution Model: SPMD

- Single program, multiple data
- Code is identical across all threads
- Execution path may differ

```
int a[N];
for all elements do in parallel{
    if (a[i] > threshold)
       a[i] = a[i] * 2;
}
```



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;
}
```

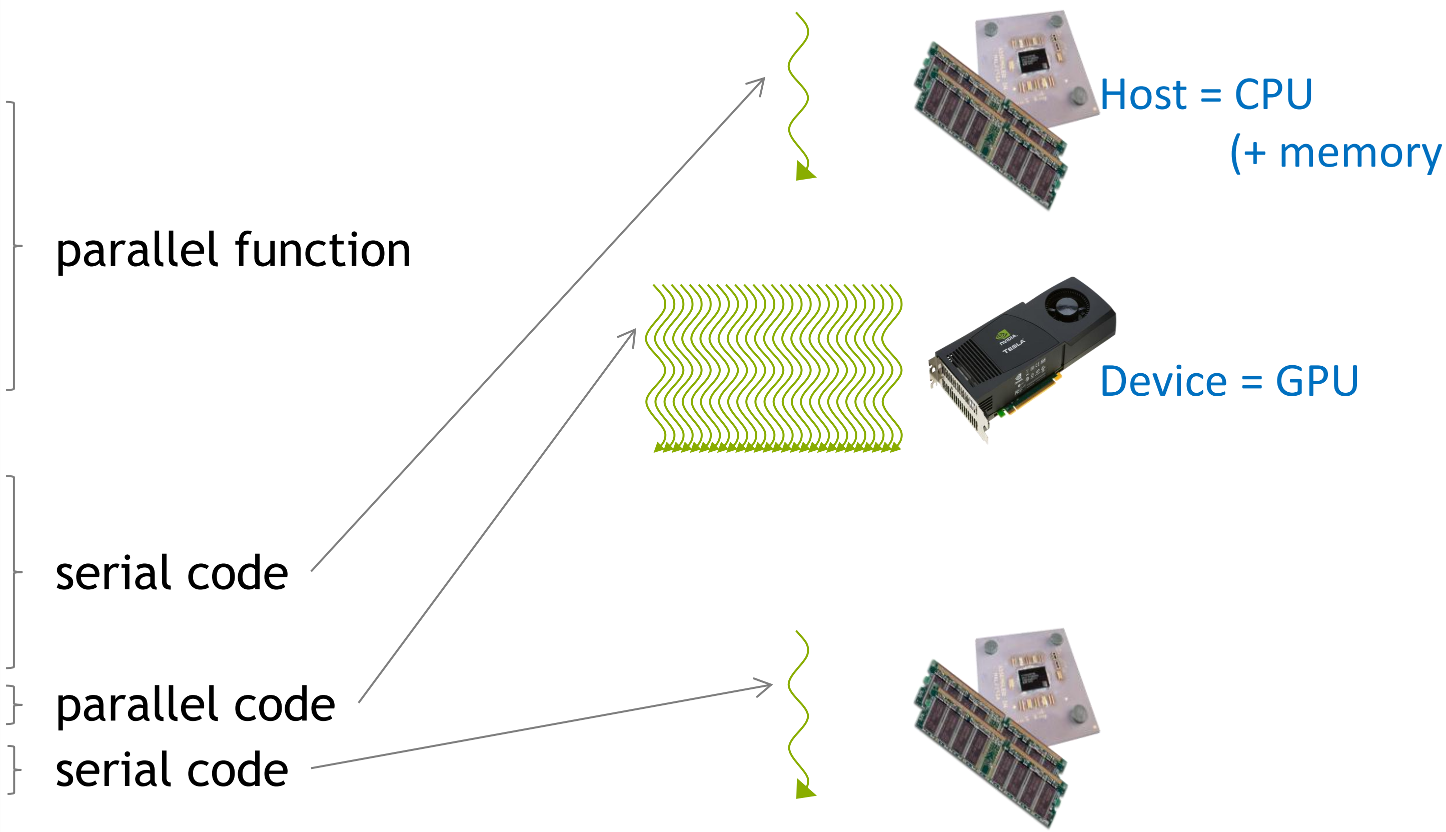
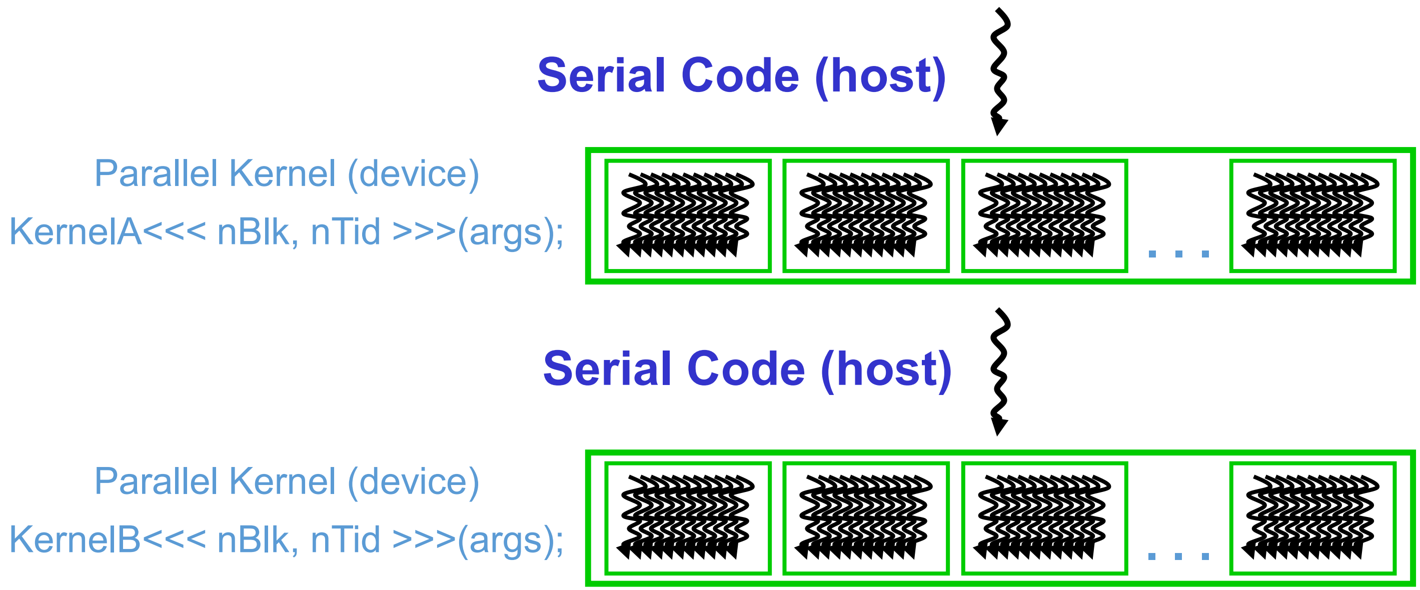


Image source: NVIDIA. CUDA C/C++ Basics

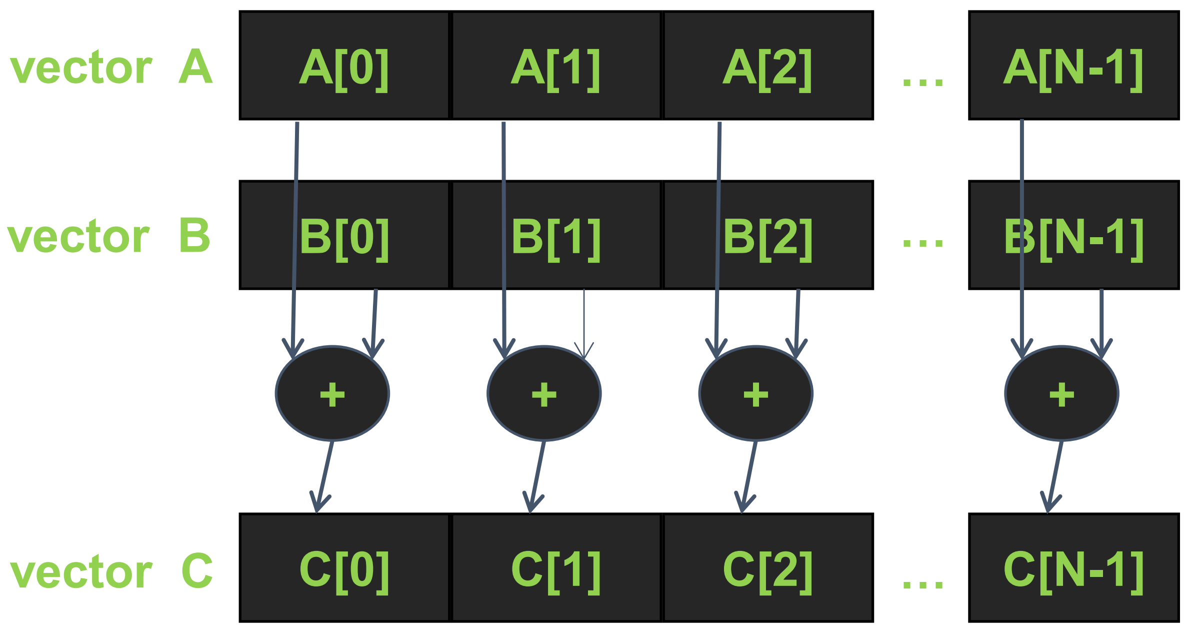
CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code
    - All threads in grid run the same kernel code
    - Each threads has an index that it use to compute memory address and make control decisions



Data Parallelism - Vector Addition

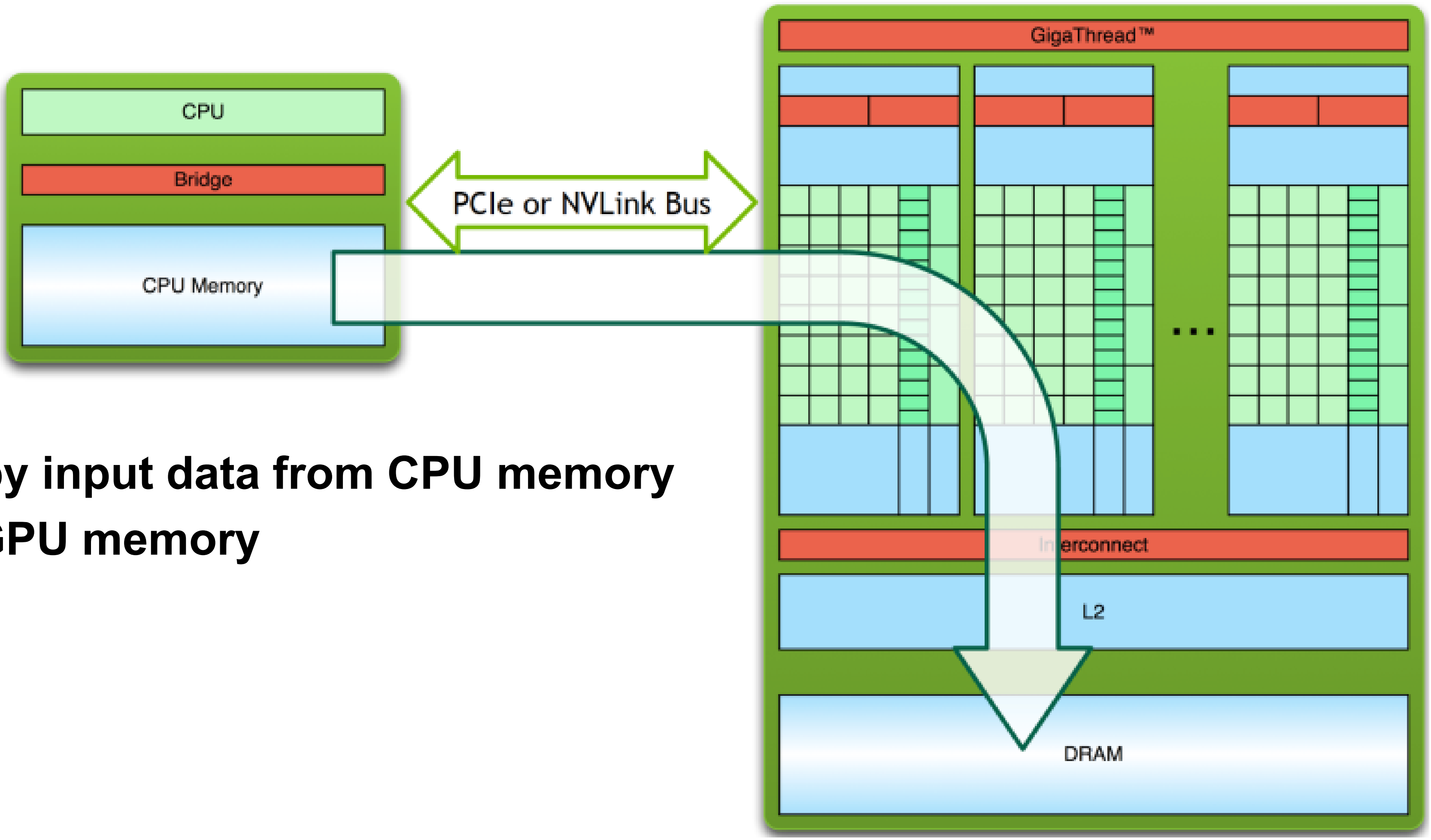
- Adding 2 vectors sequentially using host
- Adding 2 vectors in parallel using device: each thread on device are responsible for computing an element in the sum vector, and all these threads run in parallel



```
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;
}
```

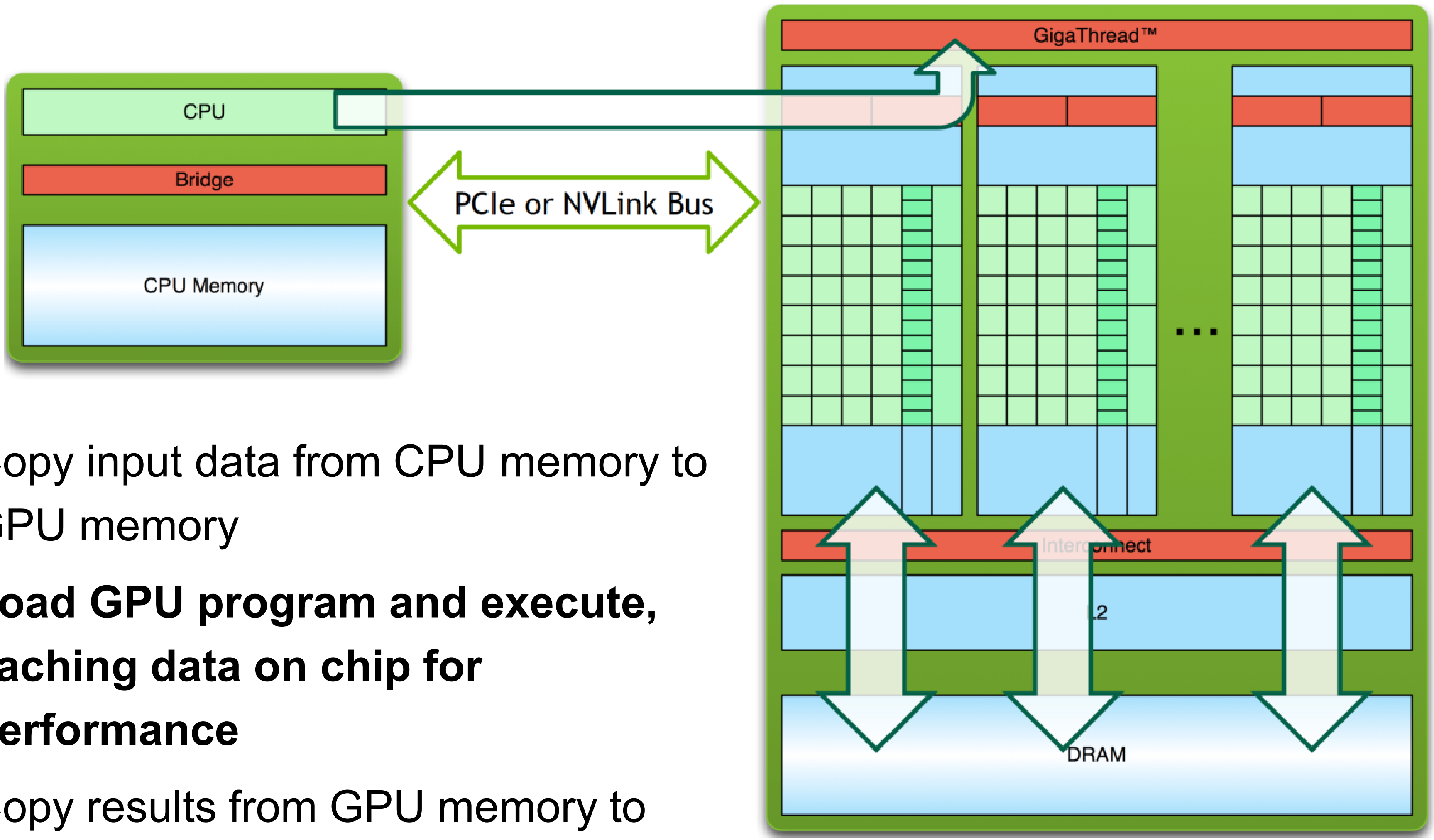
```
void addVecOnHost(float* in1, float* in2, float* out, int n)
{
   for (int i = 0; i < n; i++)
      out[i] = in1[i] + in2[i];
}
```

Simple Processing Flow



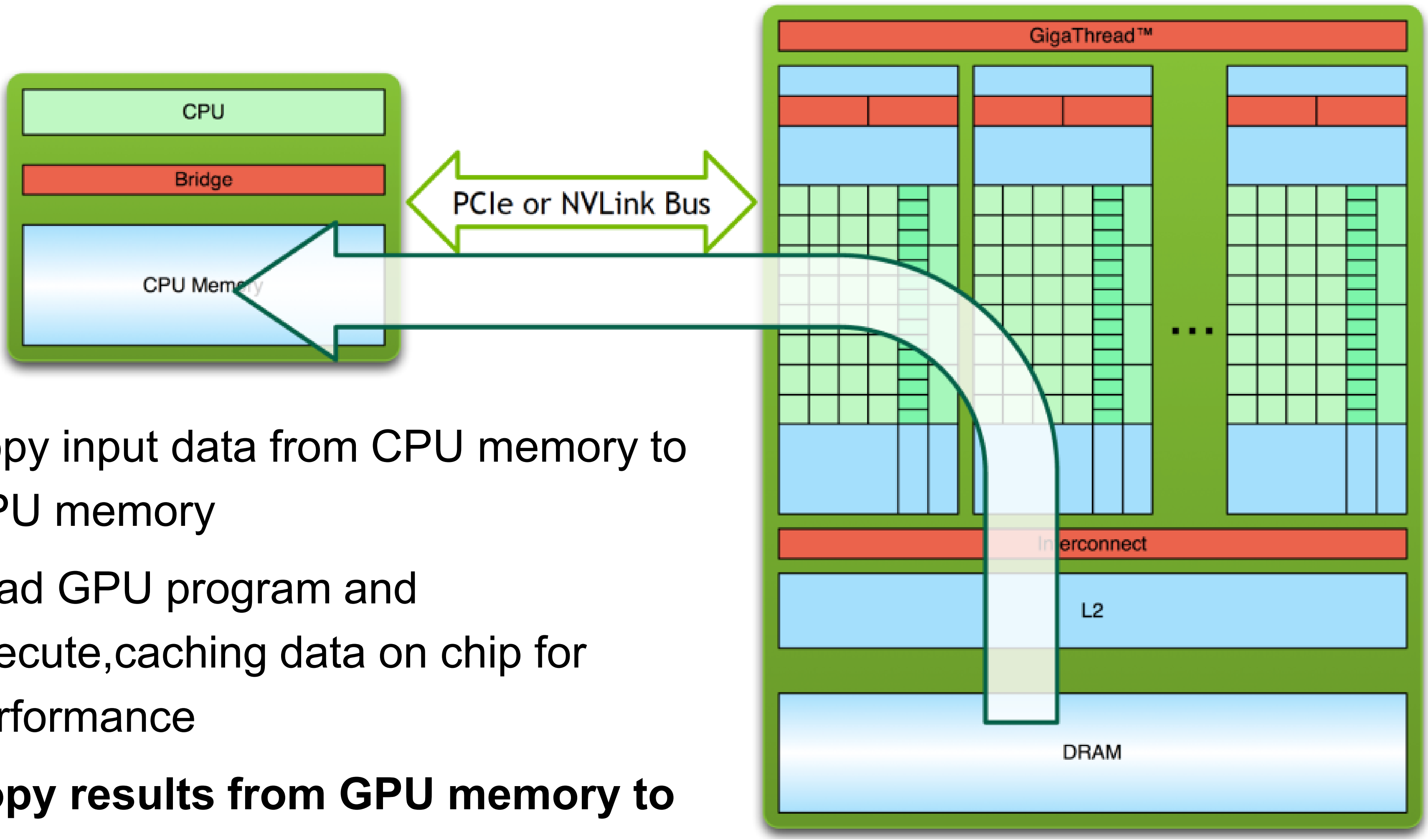
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



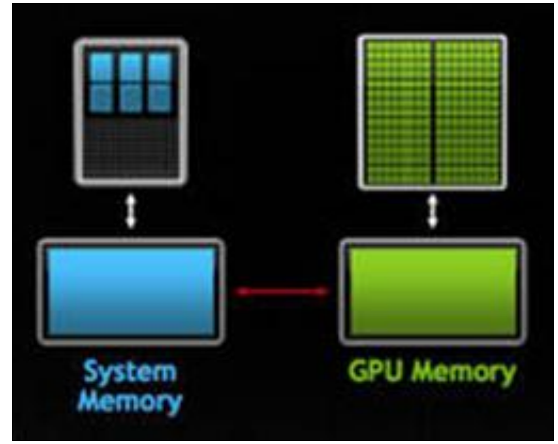
1. Copy input data from CPU memory to GPU memory
2. **Load GPU program and execute, caching data on chip for performance**
3. Copy results from GPU memory to CPU memoryPCI Bus

Simple Processing Flow



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 PCI Bus**

```
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
...
```

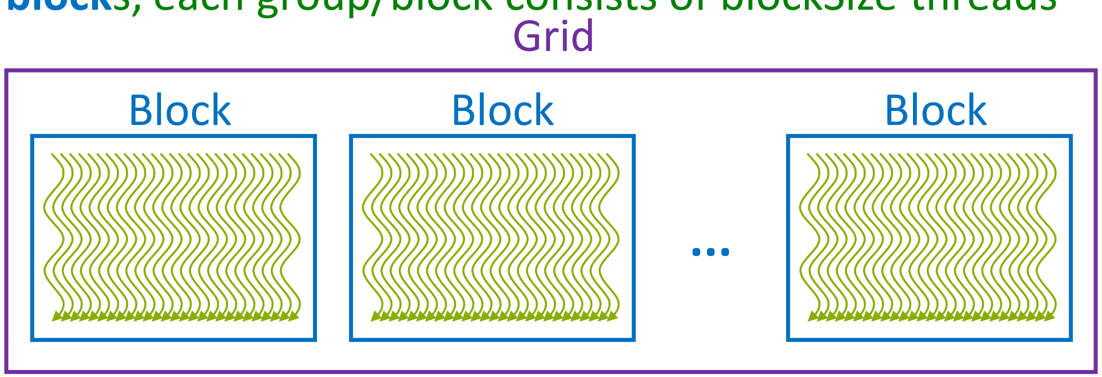
```
// Host allocates memories on device
float *d_in1, *d_in2, *d_out;
cudaMalloc(&d_in1,  n * sizeof(float));
cudaMalloc(&d_in2,  n * sizeof(float));
cudaMalloc(&d_out,  n * sizeof(float));

// 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);
```

```
// 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);
```

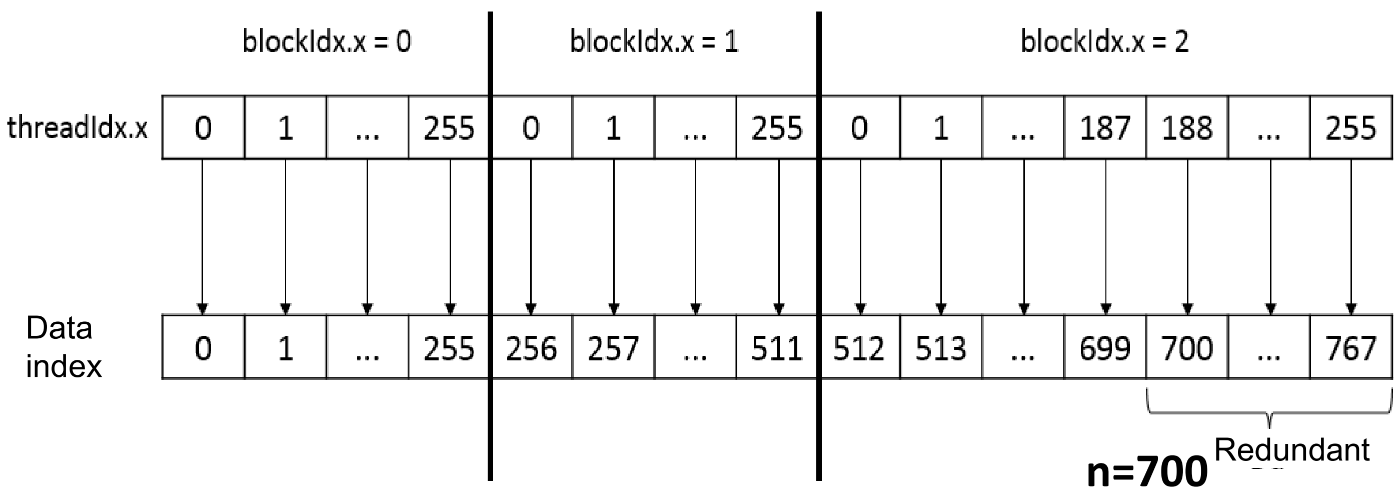
This command creates on device a bunch of threads (called **grid**) executing the addVecOnDevice function in parallel; these threads are organized into gridSize groups or **block**s, each group/block consists of blockSize threads



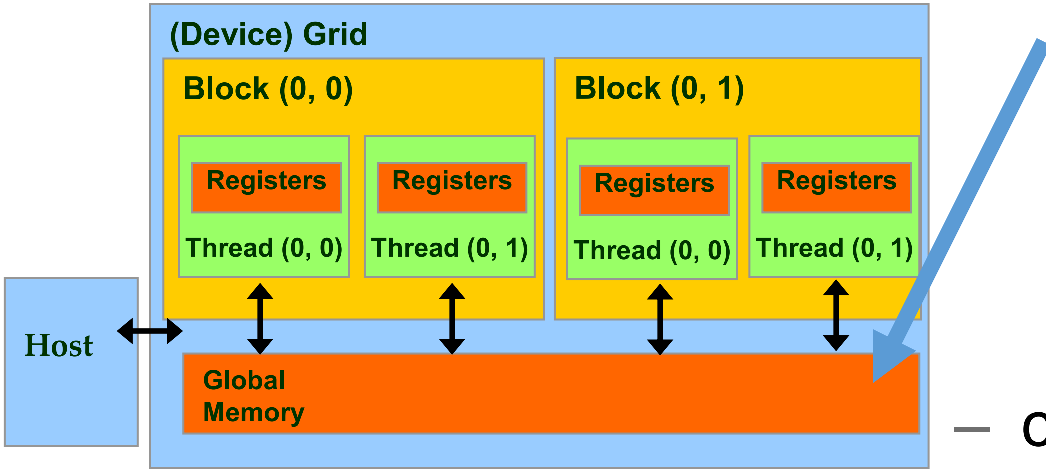
```
...
// Host invokes kernel function to add vectors on device
dim3 blockSize(256);
dim3 gridSize((n - 1) / blockSize.x + 1);
addVecOnDevice<<<gridSize, blockSize>>>(d_in1, d_in2, d_out, n);
...
```

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];
}
```

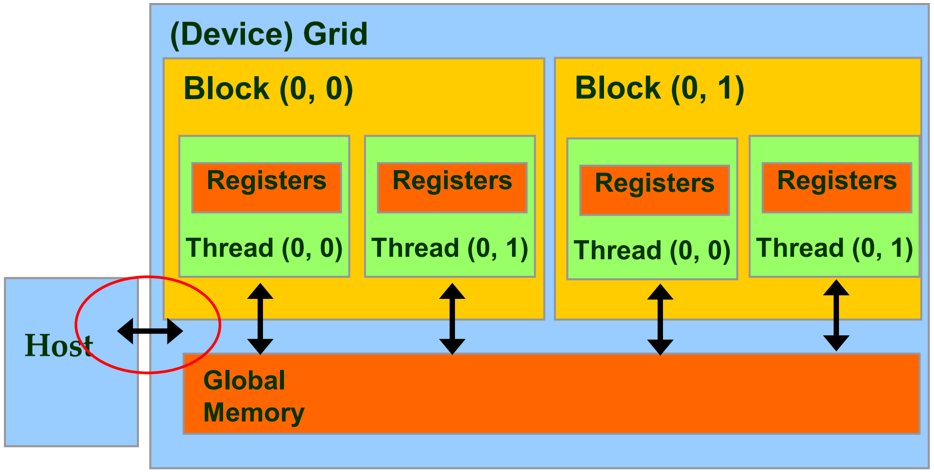


CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device <u>global memory</u>
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - **Type/Direction of transfer**
  - Transfer to device is **synchronous** with respect to the host

cudaMemcpyHostToHost	Host -> Host
cudaMemcpyHostToDevice	Host -> Device
cudaMemcpyDeviceToHost	Device -> Host
cudaMemcpyDeviceToDevice	Device -> Device
cudaMemcpyDefault	Default based unified virtual address space

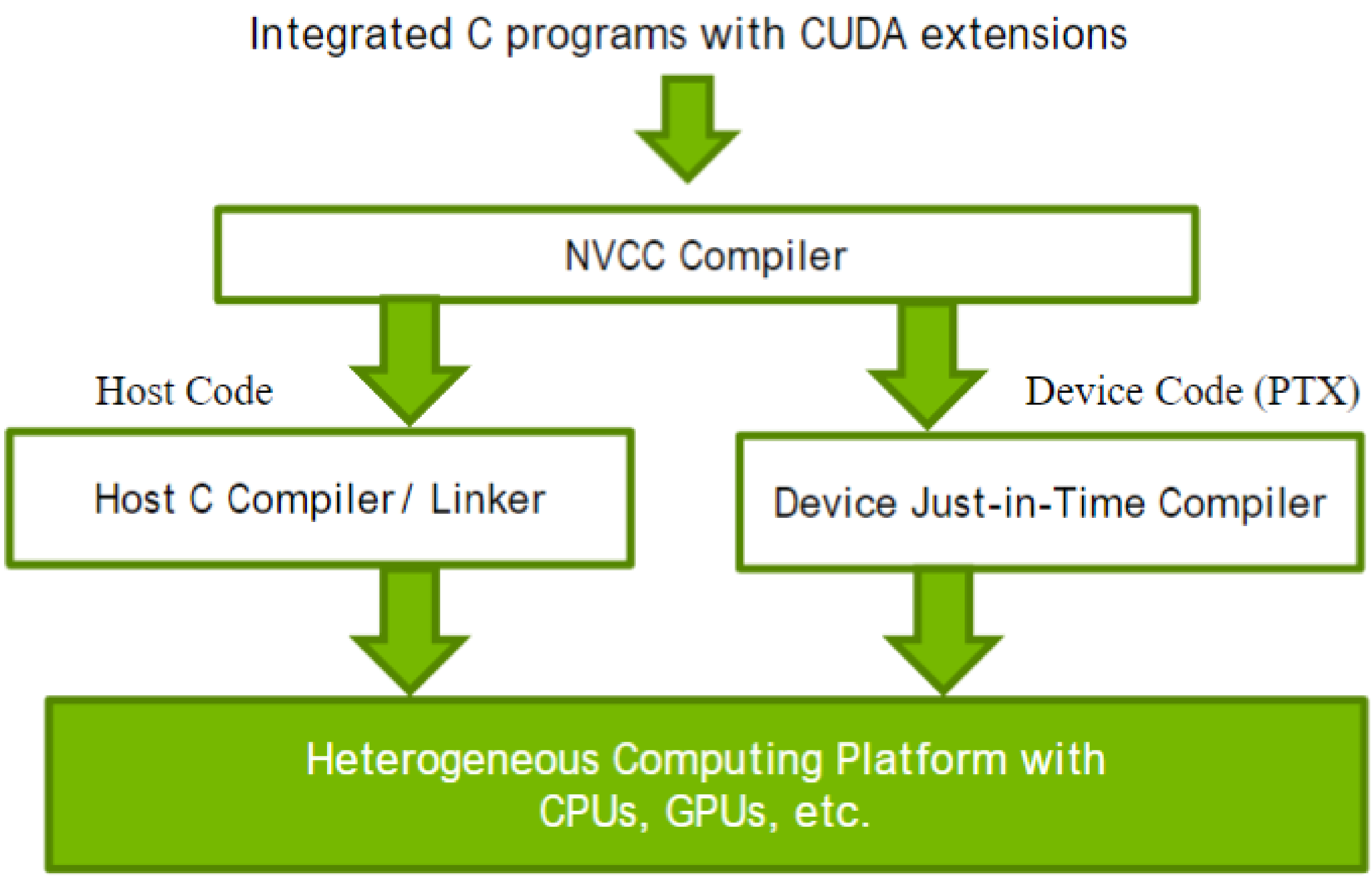
More on CUDA Function Declarations

	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

- For convenience, we can define a macro to check error and wrap it around CUDA API function calls

```
#define CHECK(call)\
{\
        const cudaError_t error = call;\
        if (error != cudaSuccess)\
        {\
                fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);\
                fprintf(stderr, "code: %d, reason: %s\n", error,\
                                cudaGetErrorString(error));\
                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));
```

Handling kernel errors

- Handling kernel errors is a bit more complicated because kernels execute asynchronously with respect to the host.
- The CUDA runtime maintains an error variable that is overwritten each time an error occurs.
- The function cudaPeekAtLastError() returns the value of this variable
- The function cudaGetLastError() returns the value of this variable and also resets it to cudaSuccess.
- Read here, “Handling CUDA Errors” section, for more info

Handling kernel errors

```
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaError_t errSync  = cudaGetLastError();
cudaError_t errAsync = cudaDeviceSynchronize();
if (errSync != cudaSuccess)
       printf("Sync kernel error: %s\n",
cudaGetErrorString(errSync));
if (errAsync != cudaSuccess)
       printf("Async kernel error: %s\n",
cudaGetErrorString(errAsync));
```

- Synchronous error:
  - Invalid execution configuration parameters, e.g. too many threads per thread block
  - errSync returned by cudaGetLastError().
- Asynchronous errors:
  - On the device after control is returned to the host
  - Out-of-bounds memory accesses

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
- [3] https://www.eecs.umich.edu/courses/eecs498-APP

THE END