Lec2: Heterogeneous Data Parallel Computing

2025Spring: COSC4397

Based on Programming Massive Parallel Processors, 4th Ed, Chapter 2

Agenda

- Data parallelism
- CUDA language, SPMD
- Threads, host, device, memory management
- Structure of a CUDA C kernel function

Data Parallelism

- “same compute, on different parts of the data”
- Those computations on different parts of the data can often be done independently
- Many application exhibits rich data parallelism that is amenable to scalable parallel computation
- The most important type of parallelism that GPU exploits
- Examples:
  - Graphics: each pixel could be independently rendered
  - Image processing: blurring
  - Matrix computations: multiplication, decomposition, etc

Programming Model: Data Parallelism

- How to program GPU (general purpose, non-graphics applications) to exploit data parallelism?
  - SIMD instructions? (x86: SSE, AVX, arm: NEON)
  - Vector instructions? (No longer used much; used on Vector machine such as Cray supercomputers)
  - Threads? (most flexible, but high cost? E.g. CPU multi-threading)
  - Loops? (OpenMP annotation? Are iterations independent?)
- Programming model
  - Needs to be simple for productivity
  - Needs to be performant on the underlying execution model

GPU Execution Model

- To be studied later in a separate lecture, but in a nutshell…
  - Multi-core: core in NVIDIA speak is Streaming Multiprocessor (SM)
  - Wide SIMD machine: each SIMD lane is called a CUDA core
  - Pipelined functional units

CUDA/C Extension

- In NVIDIA speak CUDA/C model is Single Thread Multiple Data (STMD), highlighting the role of thread.
- In fact, thread is the main mechanism to express parallelism in CUDA/C
- What's a thread?
  - Software thread vs hardware thread
  - Thread on GPU vs Thread on CPU
  - Thread in CUDA vs Thread in POSIX threads

CUDA/C Structure

- Heterogeneous model (sometimes accelerator model): a program is executed on two different architectures:
  - Host: usually the CPU, likely x64 or ARM
  - Devices: usually discrete GPUs

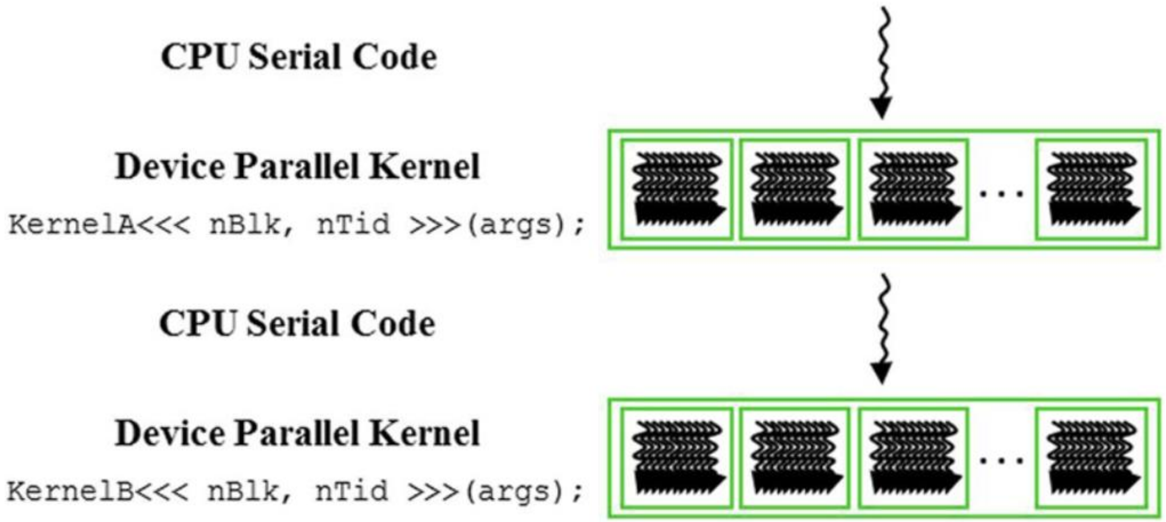


FIGURE 2.3 Execution of a CUDA program.

CUDA/C

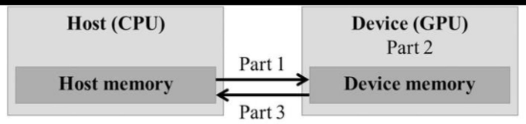
- Kernel: the function executable on GPU
- Launching a kernel (from host):
  - Send code of the kernel to GPU for execution
  - GPU ”spawns” a large number of threads, **each one of those threading** executing the kernel function
  - Those threads are called a **grid** of threads
- Execution on Host and Devices are asynchronous
  - After launching the kernel, the CPU program proceeds without waiting (usually, unless using a synchronous call)
  - To synchronize (host code wait for GPU kernel to finish): explicit call device synchronize

Vector Add Example: Loops

```c
// Compute vector sum C_h = A_h + B_h
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    for (int i = 0; i < n; ++i) {
        C_h[i] = A_h[i] + B_h[i];
    }
}
int main() {
    // Memory allocation for arrays A, B, and C
    // I/O to read A and B, N elements each
    ...
    vecAdd(A, B, C, N);
}
```

FIGURE 2.4 A simple traditional vector addition C code example.

Vector Add Example: Offloading to Device




```
void vecAdd(float* A, float* B, float* C, int n) {
    int  size = n* sizeof(float);
    float  *d_A *d_B, *d_C;

    // Part 1: Allocate device memory for A, B, and C
    // Copy A and B to device memory
    ...

    // Part 2: Call kernel – to launch a grid of threads
    // to perform the actual vector addition
    ...

    // Part 3: Copy C from the device memory
    // Free device vectors
    ...
}
```

FIGURE 2.5 Outline of a revised vecAdd function that moves the work to a device.

- Host program is the control center.
- Host program typically "offload" heavy computation to Device (GPU)
- Host and device have different memory space: host memory vs device memory (global memory)
- Data needs to be explicitly moved between host and device

Device Global Memory and Data Transfer

- Before calling the kernel, the host program needs to
  - Allocate GPU memory
  - Move necessary input data to the GPU memory
- And launch the GPU kernel…
- Afterwards, the CPU needs to move the output in GPU memory back to host for post-processing

- cudaMalloc()
  - Allocates GPU memory
  - Returns a pointer that **points to GPU memory** on host program
  - Accessing data pointed by GPU memory pointer will segfault
- cudaFree()
  - De-allocate GPU memory space
- cudaMemcpy(dst,src,size,dir)
  - Memory data transfer
  - Src, dst are source and destination (pointers)
  - Dir is the direction: HostToDevice, DeviceToHost

VecAdd with data transfer

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

FIGURE 2.8 A more complete version of vec-Add().

- Pop quiz:
  - Why the &A_d in cudaMalloc(&A_d, size)?

Aside: Error Checking in CUDA code

CUDA Kernel and Thread

- A CUDA kernel is a **function** that is executed by **each of the threads** on GPU
- This is called SPMD, Single Program Multiple Data
- On the right is an example of N thread blocks, each consisting of 256 threads, executing the same code (kernel function)

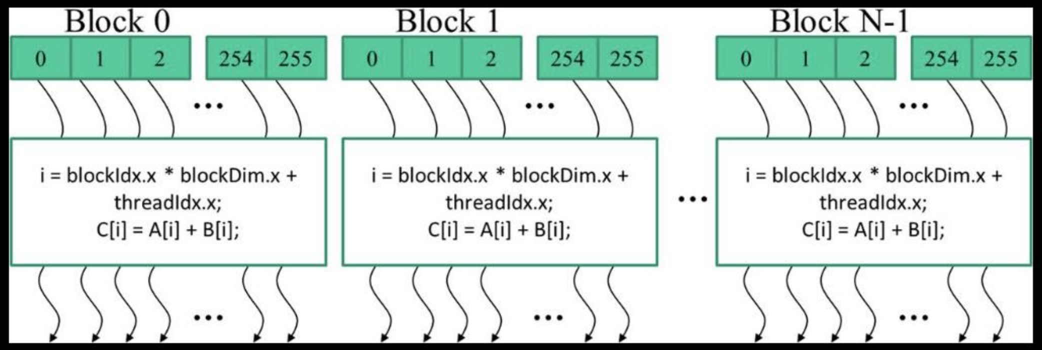


FIGURE 2.9 All threads in a grid execute the same kernel code.

Thread Organization and ID

- Host program **launches** a kernel function, with the following info:
  - The kernel function name
  - Kernel function parameters
  - # of **thread blocks**
  - # of **threads per block**
- Each thread computes one element of C, but which one?
  - Depending on thread ID
  - How to id each thread?

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

FIGURE 2.10 A vector addition kernel function.

Thread Organization and ID

- **Built-in variables:**
  - Which thread block? blockIdx
  - Which thread in the block? threadIdx
  - How many blocks? gridDim
  - How many threads per block blockDim
- Private variable: i
- Branch: if (i<n)

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

FIGURE 2.10 A vector addition kernel function.

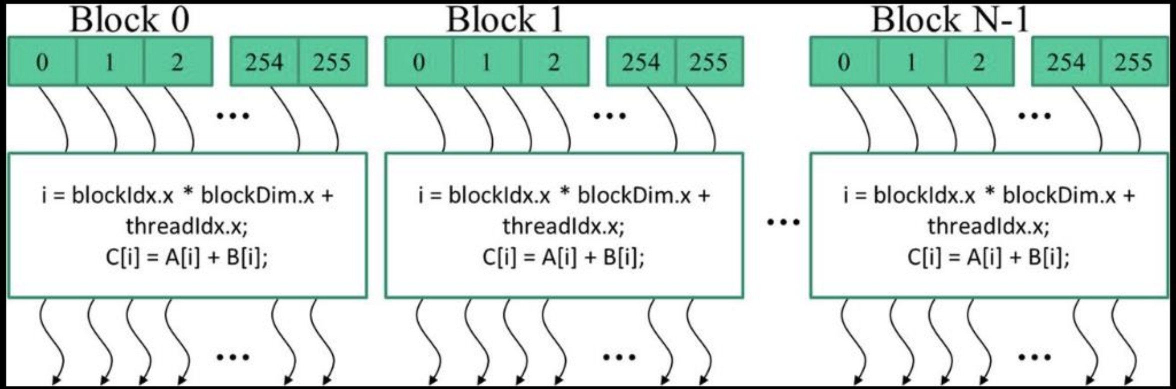


FIGURE 2.9 All threads in a grid execute the same kernel code.

CUDA Qualifiers

Qualifier Keyword	Callable From	Executed On	Executed By
__host__ (default)	Host	Host	Caller host thread
__global__	Host (or Device)	Device	New grid of device threads
__device__	Device	Device	Caller device thread

FIGURE 2.11 CUDA C keywords for function declaration.

Launching a kernel

- Launching a kernel from host program is like calling a function, but with a funny <<<gridDims, blockDims>>>
- Specifies how many threads to launch:
  - gridDims: number of blocks
  - blockDims: number of threads per block
  - total threads = gridDims* blockDims
- Kernel launch is asynchronous

```
void vecAdd(float* A, float* B, float* C, int n) {
    float *A_d, *B_d, *C_d;
    int size = n * sizeof(float);

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

FIGURE 2.13 A complete version of the host code in the vecAdd function.

Compilation & Execution

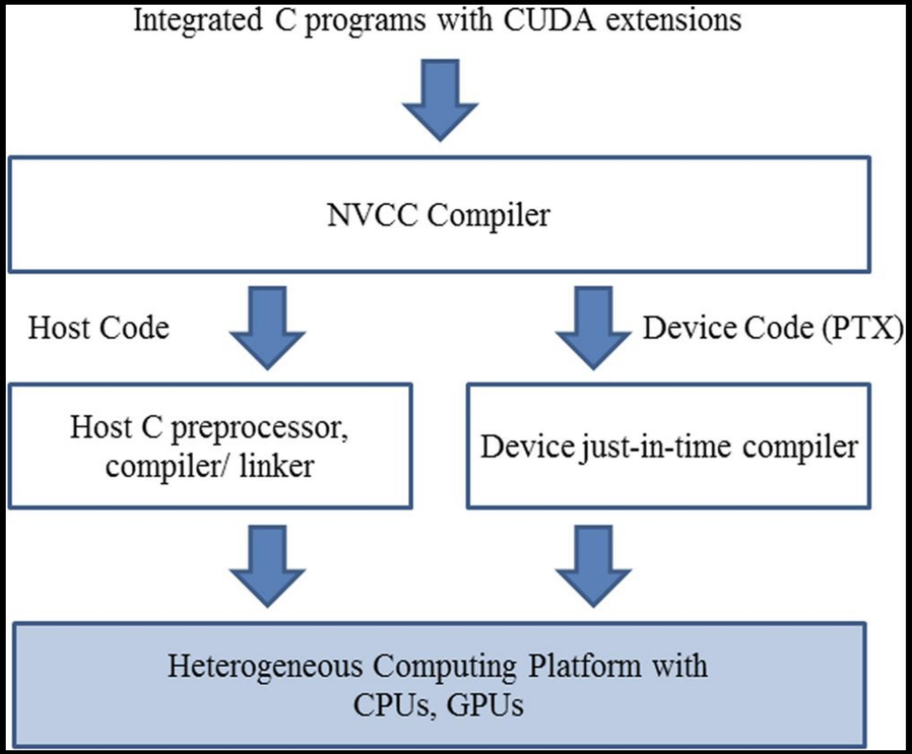


FIGURE 2.14 Overview of the compilation process of a CUDA C program.

- You can put both host code and kernel/device code in a single file, (*.cu)
- NVCC will compile the *.cu into a combined object file or executable.
- Can handle the object/executable files as if they are single architecture.

CUDA Driver and Runtime

Pop Quiz

- 1. If we want to use each thread in a grid to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to the data index (i)?

(A) i=threadIdx.x + threadIdx.y;

(B) i=blockIdx.x + threadIdx.x;

(C) i=blockIdx.x*blockDim.x + threadIdx.x;

(D) i=blockIdx.x * threadIdx.x;"

- "2. Assume that we want to use each thread to calculate two adjacent elements of a vector addition. What would be the expression for mapping the thread/block indices to the data index (i) of the first element to be processed by a thread?

(A) i=blockIdx.x*blockDim.x + threadIdx.x +2;

(B) i=blockIdx.x*threadIdx.x*2;

(C) i=(blockIdx.x*blockDim.x + threadIdx.x)*2;

(D) i=blockIdx.x*blockDim.x*2 + threadIdx.x;"