CUDA Programming for Nvidia GPU

Ivy Peng
Assistant Professor in Computer Science
Scalable Parallel System (ScaLab)
Department of Computer Science, KTH

High-Level Programming Interfaces

- **OpenMP**: compiler directives and library for accelerators
- **OpenACC**: compiler directives and library for NVIDIA GPUs

Compiler + runtime library

- **Thrust**: C++ template library resembling C++ STL.
- **OpenCV**: Computer vision library using GPU
- **CUDA-based libraries for math**: cuBLAS, cuFFT, cuDNN, …
- **TensorFlow**

Libraries atop CUDA

Low-Level Programming GPUs

- OpenCL (Open Computing Language): based on C, not only for GPUs but also for other “accelerators” (DSP, FPGA, …) and integrated GPUs.
- **CUDA (compute unified device architecture)**: extension to C language. Only for **NVIDIA GPUs, most mature** programming environments
- Heterogeneous-Computing Interface for Portability (**HIP**) for AMD GPUs
  - C++ dialect designed to ease conversion of CUDA applications to portable C++ code.

CUDA

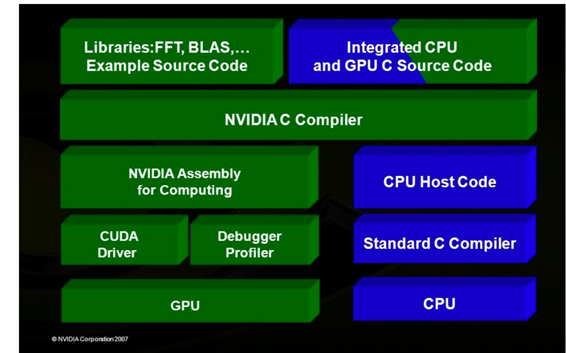
CUDA (Compute Unified Device Architecture) is **NVIDIA**’s program development environment:

- based on **C/C++** with some extensions
    - FORTRAN support provided by compiler from PGI
- **Indexing math** and **synchronization** are the main conceptual difficulties

CUDA Components

Installing CUDA on a system, there are **3 components**:

1. **Driver** low-level software that controls the graphics card
2. **Toolkit**
   - **`nvcc` compiler**
   - Tracing tools
   - profiling and debugging tools
   - several libraries for math, deep learning libraries
3. **SDK**
   - lots of demonstration examples
   - some error-checking utilities



CUDA Programming

Terminology:

- **host** = CPU and its memory (host memory)
- **device** = GPU and its memory (device memory)

Programming in 3 steps:

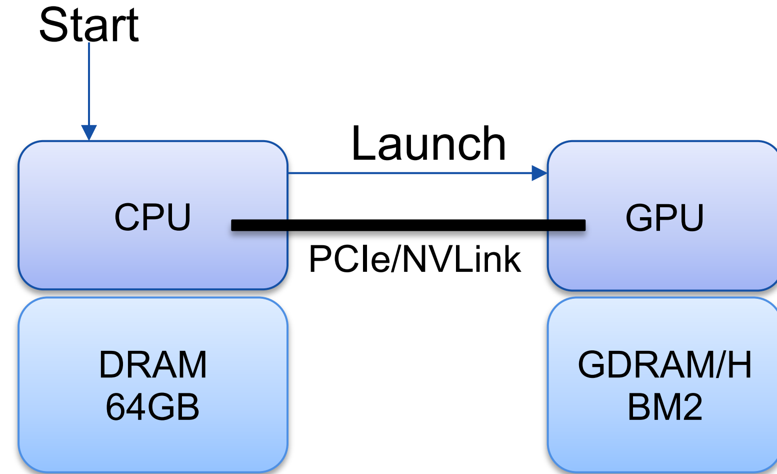
- Define where (host or device) to launch a tasks
- Define data exchange between host and device
- Define computation tasks

example.cu

```
__global__ void kernel_1<<<BPG, TPB>>>(arg1, arg2)
{……}

__device__ void kernel_2(arg1, arg2)
{……}

__host__ void kernel_3(arg1, arg2)
{……}

void kernel_3(arg1)
{…
  cudaMemcpy(…);
…}
```

CUDA Parallelism Model

Launching a kernel on the GPU from the CPU to create **a computational grid of threads**



How to declare a function called by host but executed on device?

CUDA makes this distinction by prepending one of the following function type qualifiers:

- `__global__` is the **qualifier for kernels** (which can be **called by the host and executed on device**)
- `__host__` functions called from the host and executed on the host (default qualifier, often omitted)
- `__device__` functions are called from **the device and execute on the device** (a function that is called from a kernel needs the `__device__` qualifier)

Question: which qualifier do you have before the function you call **from the GPU** and you want to run **on GPU**:

- __global__
- __host__
- __device__

?

Question: which qualifier do you have before the function you call **from the CPU** and you want to run on **GPU**:

- __global__
- __host__
- __device__

?

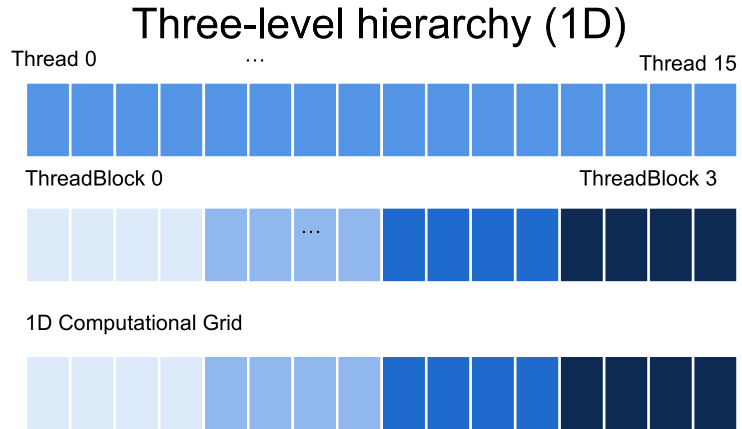
CUDA Parallelism Model

Threads are organized in a three-level hierarchy:

- **Thread**
- Thread**Block** (1D, 2D or 3D)
- ThreadBlock **Grid** (1D, 2D or 3D)

How to determine their values for your problem?

- Start with the total number threads you need
  - E.g., 1D array of N elements -> N threads
- Threads per Block is typically 32, 64,128 or 256
  - Is your problem 1D, 2D, 3D?
  - Now you can calculate Grid




```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

Launch a Kernel in CUDA

Kernel is a kind of **special function executed on the GPU**
Kernel **launch** ≅ regular function call with addition of number of threads

```
aKernel<<<BPG, TPB>>>(arg1, arg2, …)
```

To specify a kernel launch, we start with kernel name (`aKernel`) and end with argument list between `()`

Now for the CUDA extension: we specify the dimensional of the computational grid, the **grid dimensions** and **block dimension** between triple angle brackets (`<<<BPG, TPB>>>`).

Execution Configuration: Tell how Many Threads we Need

`BPG` = number of blocks in the grid

`TPB` = number of threads in the block

Together they constitute the **execution configuration** and specify the **dimensions of the kernel launch**

What it is BPG and TPB in this case?

Thread 0 … Thread 15

ThreadBlock 0 … ThreadBlock 3

1D Computational Grid

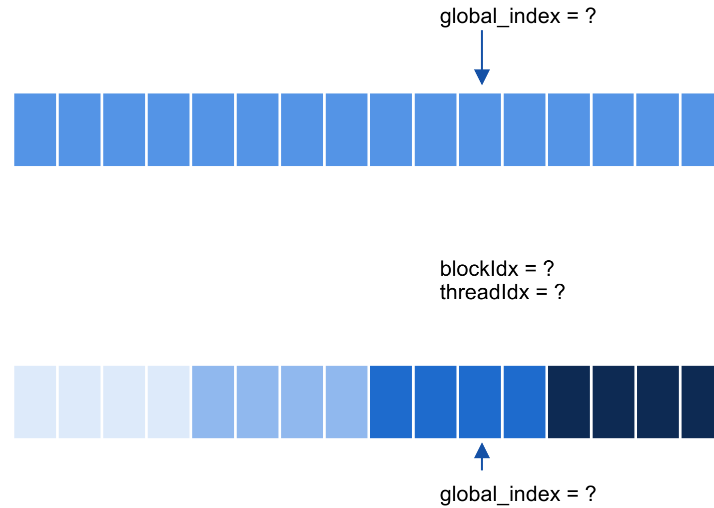
CUDA Built-in Variables

CUDA provides build-in dimension and index variables when in the kernel

- **Dimension variables**
  - `gridDim` = number of blocks in the grid
  - `blockDim` = number of threads in each block
- **Index variables**
  - `blockIdx` = index of the block in the grid
  - `threadIdx` = index of the thread within the block

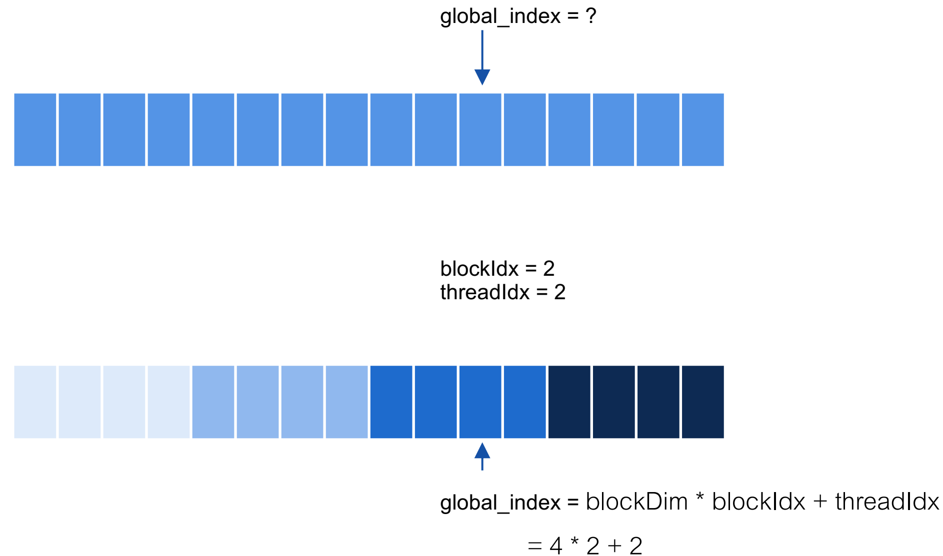
Question: How do I calculate my global thread ID (1D grid)?

Using `threadIdx`, `blockIdx`, and what do I need also?



Question: How do I calculate my global thread ID (1D grid)?

Using `threadIdx`, `blockIdx`, and what do I need also?



Data Transfer between host and device

- Kernels execute on the GPU and **do not**, *in general*, **have access to data stored on the host side**
- **Kernels cannot return a value**, so the return type is always void, and kernel declarations starts as

```
__global__ void aKernel(arg1, arg2, …)
```

  - **How do I get the results from my kernel ??**

Data Transfer between host and device

The CUDA runtime API provides these functions for transferring input data to the device and transferring results back to the host:

- `cudaMalloc()` allocates device memory
- `cudaMemcpy()` transfers data to or from a device
  - `cudaMemcpy(void* dest, void* src, size_t size,cudaMemcpyHostToDevice)` **host mem → GPU mem**
  - `cudaMemcpy(void* dest, void* src, size_t size, cudaMemcpyDeviceToHost)` **GPU mem → host mem**
- `cudaFree()` frees device memory that is no longer in use

Data Transfer between host and device

Question: how I get my result from the kernel?

Data Transfers are Synchronous

By default, **data transfers are synchronous (the function does not return until the data transfer is complete)**, so `cudaMemcpy()` stalls the program execution

- GPU cannot continue to other operations until data transfer is finished, and data transfer is slow.

Synchronous v.s. Asynchous

```
main(){
  ➡syncKernel(arg1, arg2, …)
  ➡asyncKerne2(arg1, arg2, …)
  ➡syncKerne3(arg1, arg2, …)
}
```

Kernel Launching is Asynchronous

- As soon as the kernel is launched, **the CPU returns from the call of kernel without waiting for the completion of the kernel**.
- In practice, the CPU launches the kernel and right away executes what is after the kernel launch without waiting for the kernel to finish

```
main(){
    syncKernel(arg1, arg2, …)
    asyncGPUKerne2(arg1, arg2, …)
    syncKerne3(arg1, arg2, …)
}
```

Asynchronicity might create problems …

Example: a code that launches a kernel (=GPU) to print to screen and then ends.

In such situation, **after starting the GPU threads, control returns to the application and the application exits**.

At application exit, it's ability to send output to the standard output is terminated by the OS → the output generated by the kernel has nowhere to go!

```
int main(){
    syncKernel(arg1, arg2, …)
    asyncGPUKerne2(arg1, arg2)
    return 0;
}
```

Thread Synchronization

Kernels enable multiple computations in parallel, but **they don’t ensure the order of execution** (asynchronous). CUDA provides functions to synchronize :

- **`cudaDeviceSynchronize()`** effectively synchronizes **all threads** in a grid → waits for all the threads in the kernel to complete before proceed.
- `__synchThreads()` synchronizes **threads within a block**

Q & A