CSE 4373/5373 - General Purpose GPU Programming

Heterogeneous Data Parallel Computing

Alex Dillhoff

University of Texas at Arlington

Heterogeneous Data Parallel Computing

- Data parallelism is achieved through independent computations on each sample or groups of samples.
- GPU programs are defined through a **kernel** function.
- The kernel defines what is executed in each independent thread.
- Data must be transferred between the CPU (host) and GPU (device).

CUDA C Programs

CUDA C Programs

A basic CUDA program consists of:

- A **kernel** function defining the work to be performed on each thread.
- Data that is accessible on the **device**.
- Device memory allocation.
- Memory transfer from the **host** to the **device**.
- Execution of the **kernel** from the **host** machine.
- Data transfer from the **device** back to the **host**.
- Memory cleanup.

CUDA C Programs

This process is sequential, but the host does not need to wait for the device to finish before continuing.

As we write more complicated programs, we will take advantage of both task and data parallelism.

Example: Vector Addition

Vector Addition

Hwu et al. refer to vector addition as the ”Hello World” of CUDA programming.

It is an *embarrassingly parallel* problem, meaning that it is trivial to parallelize.

Vector Addition

Given two vectors of the same length, $\mathbf{x}$ and $\mathbf{y}$, the vector addition operation is defined as:

$$\mathbf{z}_i = \mathbf{x}_i + \mathbf{y}_i \quad \forall i \in \{1, \ldots, n\}$$

The vector addition operation is commutative and associative. The operation can be performed in parallel on each element of the vectors.

Vector Addition

A simple implementation in C.

```
void vecAdd(int *x_h, int *y_h, int *z_h, int n) {
    for (int i = 0; i < n; i++) {
        z_h[i] = x_h[i] + y_h[i];
    }
}
```

Vector Addition

One small note about the variable names: it is common to use the suffix `_h` to denote a variable that is allocated on the host (CPU) and `_d` to denote a variable that is allocated on the device (GPU).

In this case, the vector addition operation is performed on the host machine.

Vector Addition

An equivalent implementation in CUDA C++.

```
__global__
void vecAdd(float *x_d, float *y_d, float *z_d, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        z_d[i] = x_d[i] + y_d[i];
    }
}
```

Vector Addition

This kernel executes on a single thread.

The thread index is computed using built-in variables `blockIdx.x`, `blockDim.x`, and `threadIdx.x`.

The details of how these variables are defined are not important right now. The main point is that each kernel is executed on a single thread. For a GPU with thousands of individual threads, this kernel will be executed thousands of times in parallel.

Vector Addition

The `__global__` keyword placed before the function definition indicates that the function can be called from both the host and the device.

Keyword	Description
`__global__`	Executed on the device, callable from the host and device
`__device__`	Executed on the device, callable from the device only
`__host__`	Executed on the host, callable from the host only

Vector Addition

Unless otherwise specified, functions that you define will be executed on the host.

It is not necessary to specify the `__host__` keyword.

If you want the compiler to generate both host and device code, you can use the `__host__ __device__` keyword combination.

Calling the Kernel

To call this kernel, we essentially need two sets of data: one on the host and one on the device.

Assumption: the data is already given on the host as `x_h` and `y_h` with length `n`.

Memory Allocation

The first step is to allocate memory on the device using `cudaMalloc`.

The address given can be printed, but the data cannot be accessed directly from the host.

```
float *x_d, *y_d, *z_d;
cudaMalloc(&x_d, n * sizeof(float));
cudaMalloc(&y_d, n * sizeof(float));
cudaMalloc(&z_d, n * sizeof(float));
```

Memory Allocation

The first argument is a pointer to address of the variable.

Remember that taking the address of a pointer will result in a double pointer.

This is necessary because the function will need to dereference the pointer to store the address to the allocated data.

Memory Allocation

The memory that is allocated on the device is called **global memory**.

It is accessible by all threads on the device.

There is also a small amount of **shared memory** that is accessible by threads within a single block along with a **unified memory** model.

Memory Allocation

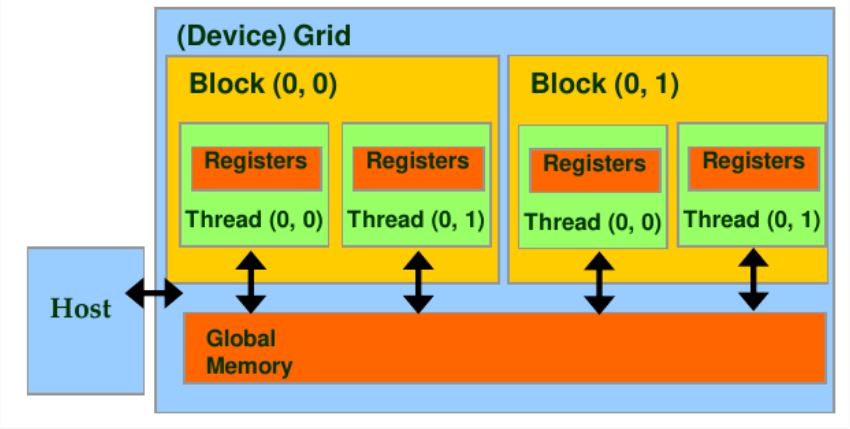

Figure 1: Overview of memory layout (source: NVIDIA DLI).

Memory Transfer

Data is transferred between the host and CPU via `cudaMemcpy`.

The arguments are the **destination pointer**, **source pointer**, **size**, and **direction**.

The direction is an enumerated type that can be one of the following:

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

Grids, Blocks, and Threads

The CUDA programming model is based on a hierarchy of **grids**, **blocks**, and **threads**.

- A **grid** is a collection of **blocks**.
- A **block** is a collection of **threads**.
- The number of **blocks** and **threads** that can be executed in parallel is limited by the hardware.
- The number of **blocks** and **threads** that can be executed in parallel is called the **grid size** and **block size**, respectively.

Grids, Blocks, and Threads

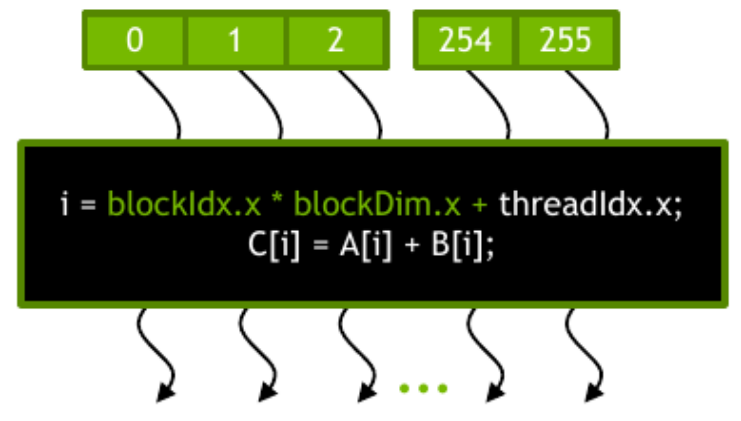

Figure 2: A single block of 256 threads (source: NVIDIA DLI).

Grids, Blocks, and Threads

Threads within each block are executed in parallel and do not interact with threads in other blocks.

For threads within a single block, there is a small amount of shared memory as well as other tools for communication.

Kernel Execution

```
vecAdd<<<ceil(n / 256.0), 256>>>(x_d, y_d, z_d, n);
```

Kernel Execution

Calling the kernel function almost looks like any ordinary function call; the main difference being the inclusion of the <<< and >>> syntax.

These are used to specify the size of the grid and blocks, respectively.

Kernel Execution

In this example, we specified that each block has 256 threads.

We can use that specification to dynamically determine the number of blocks based on the input size.

The number of blocks is computed as the ceiling of the input size divided by the number of threads per block to ensure that there are enough blocks to cover the entire input size.

Kernel Execution

Returning to the kernel function, the thread index is computed using built-in variables `blockIdx.x`, `blockDim.x`, and `threadIdx.x`.

These are defined as **`struct`** variables.

Kernel Execution

Modern GPUs have a 3-dimensional grid, but we only need to worry about the first dimension for now.

The thread index is computed as the product of the block index and the number of threads per block plus the thread index within the block.

Kernel Execution

It is possible to have more threads than there are blocks.

As much as possible, you should try and work with powers of 2 to ensure that the hardware is used as efficiently as possible.

Kernel Execution

In this example, we check to see if the thread index is less than the input size.

If it is, the vector addition operation is performed. Otherwise, the function exits.

It isn't worth worrying about now, but we will see that conditional statements like this can have significant performance implications.

Kernel Execution

The resources available to you are dependent on the device and its compute capability.

This document lists the compute capabilities for various NVIDIA GPUs.

Kernel Execution

One other thing to note is that the kernel function is executed asynchronously.

Once the kernel is launched, the host will continue executing code as normal.

In our example, our host may try to copy the data back before the kernel has finished executing.

Kernel Execution

We can make sure the kernel has finished execution by calling `cudaDeviceSynchronize()`.

This should happen directly before we transfer data back to the host.

Transferring Data Back to the Host

Once the kernel has finished executing on the device, we need to transfer the data back to the host.

```
cudaMemcpy(z_h, z_d, n * sizeof(float), cudaMemcpyDeviceToHost);
```

Cleaning Up

Once we are done with the data on the device, we can free the memory using `cudaFree`.

```
free(x_h);
free(y_h);
free(z_h);
cudaFree(x_d);
cudaFree(y_d);
cudaFree(z_d);
```

Error Checking

Error Checking

The functions we use in the CUDA API return an error code.

We can use this to create robust code that checks for errors and either corrects them or exits gracefully.

Error Checking

```
cudaError_t err = cudaMalloc(&x_d, n * sizeof(float));
if (err != cudaSuccess) {
    fprintf(stderr, "Error: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

A common pattern is to define a macro that checks the result of a CUDA function and exits if there is an error.

Error Checking

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file,
                      int line, bool abort=true) {
   if (code != cudaSuccess) {
      fprintf(stderr,"GPUassert: %s %s %d\n",
                cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}
```