Fundamentals of CUDA C/C++

Introduction

CUDA is a parallel computing platform developed by NVIDIA that allows general-purpose computing on GPUs (GPGPU). It is widely used in the fields related to high-performance computing, such as machine learning and computer simulation.

Typically, the parallelism and performance on GPUs are much higher than on CPUs. Let's look at the FLOPs of some latest CPU and GPU chips. With less than double prize, NVIDIA A100 offers nearly 26 times better performance than today's most powerful CPU.

Chips	Performance (TFLOPs)	Price in 2024
AMD Ryzen™ Threadripper™ PRO 7995WX	12.16	US$ 10,000
NVIDIA Tesla A100	312	US$ 18,000

In this article, we will discuss the basic methodology on CUDA programming.

Before You Start

NVIDIA offers a command line tool `nvidia-smi` (system management interface) to show the information of the GPU installed on the computer. Don't worry if you do not have a NVIDIA GPU, simply turn to Google Colab for a free GPU environment. The rest of the article will be done on Google Colab.

After we connect to a GPU runtime on Colab, we can now run `nvidia-smi` in the interactive environment to see the information of the GPU installed. Note that we are in a Python environment and `!` before system commands tells the environment to run it in a Linux shell.

```
!nvidia-smi
```

```
Fri Jul  5 03:05:50 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   66C    P8              10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Great! We now have a Tesla T4 GPU! Make sure you have seen similar outputs and we will now move on to our next topic.

CUDA Programming Basics

The prefix of a file in the CUDA language is `.cu`, and a CUDA file can be compiled along with C/C++/Fortran easily with the help of `CMake`. However, it is not today's topic and we will simply stop here.

Our First CUDA Program

Before we talk about CUDA, let's first look at a C program that we are already familiar with.

```
#include <stdio.h>

void printHelloWorld() {
  printf("Hello, world!\n");
}

int main() {
  printHelloWorld();
}
```

Straightforward, right? After rename the file as `first.cu`, we can compile it with CUDA compiler, just like what we usually do with a C compiler.

```
!nvcc first.cu -o first -run
```

`nvcc` is the CUDA compiler, `-o` specified the name of the output executable file, `-run` executes the binary file right after the compilation process finishes. Now we can see the output from the program.

```
[4]  !nvcc ./first.cu -o first -run

     Hello, world!
```

The output is expected. However, it is nothing different from what we usually do in C, since what we have written is exactly a C program. Now we will refactor the program so that it runs on GPU. In the context of CUDA programming, CPU is usually called `host` and GPU `device`. A function that runs on device is called **kernel function**. The return value of a kernel function **must** be `void`. To enable some function to run on device, we must add some special qualifier before the function. Here we list common qualifiers and explained their meanings.

Qualifiers	Meanings
__global__	The function is executed on device, called by host.
__device__	The function is executed on device, called by device.
__host__	The function is executed on host. This is the default choice when we omit the qualifier.

In this article, tasks are assigned by the host to the device, so we will only use the `__global__` qualifier and default qualifier (no qualifier).

To launch a kernel function (don't forget what is a kernel), we use triple angle brackets to specify the configuration of the kernel function. Let's look at an example. The kernel function is defined here,

```
__global__ void printHelloWorld() {
  printf("Hello, world!\n");
}
```

and called here.

```
int main() {
  printHelloWorld<<<1, 2>>>();
}
```

The configuration of the kernel will be explained later. Note that a kernel function is asynchronous, which is to mean that the host won't wait for the kernel function to finish. Rather, the host will continue to execute the codes below. Function `cudaDeviceSynchronize()` let host wait for **all** the kernels to finish. Let's put it together and look at the results. The complete CUDA program should look at this:

```
#include <stdio.h>

__global__ void printHelloWorld() {
  printf("Hello, world!\n");
}

int main() {
  printHelloWorld<<<1, 2>>>();
  cudaDeviceSynchronize();
}
```

Now we compile and run the program

```
!nvcc first.cu -o first -run
```

And we get the results

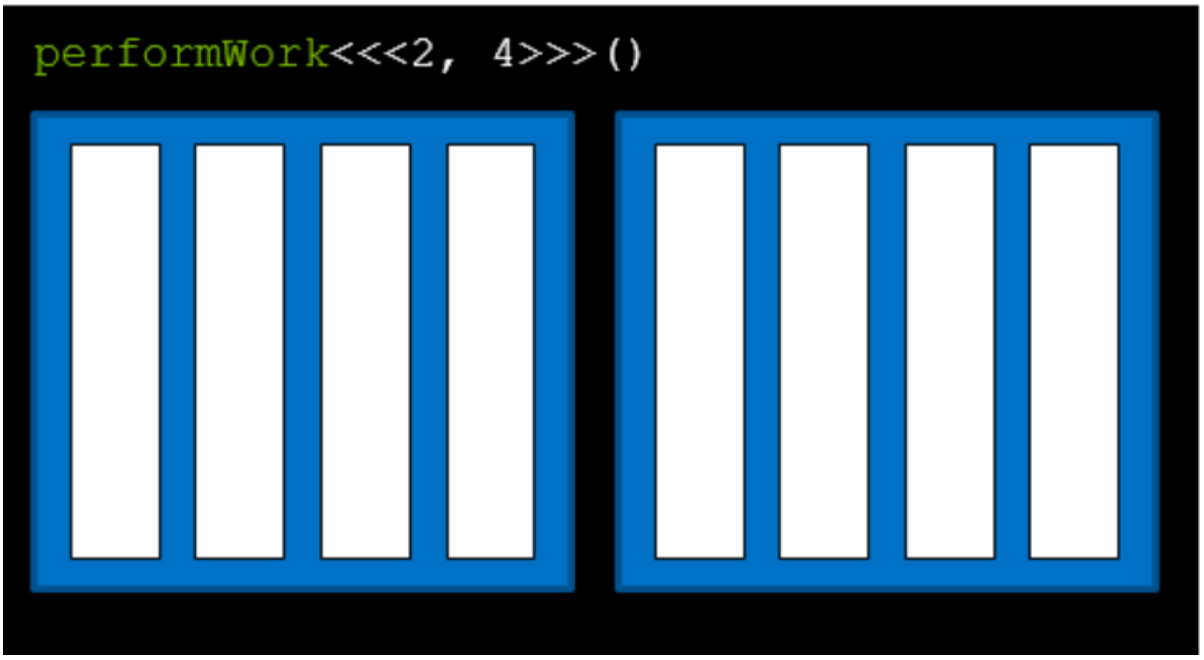
```
[5]  !nvcc ./first.cu -o first -run

     Hello, world!
     Hello, world!
```

which shows our program are truly running on GPU!

CUDA Thread Hierarchy

Kernel Configuration

In the past section, we haven't really talked about what does the parameters in the triple angle brackets actually denote. Let's look at the CUDA thread hierarchy.



CUDA follows a grid-block-thread hierarchy. The overall structure is called **grid**, which is in black and contains blocks. The first parameter denotes **number of blocks** and the second parameter denotes **threads per block**. **Blocks** are painted blue and **threads** white in the slide. So, kernel function `performWork<<<2, 4>>>` actually says, the host assigns work to 2 blocks, with 4 threads each. Now you can explain why we have seen two echoes in our first CUDA program.

A thread can get its **block index** and **thread index within the block** by variables `blockIdx.x` and `threadIdx.x`, which are available directly in the definition of a kernel function.

Let's look at an example,

```
#include <stdio.h>

__global__ void threadInBlock() {
  printf("Thread %d from block %d\n", threadIdx.x, blockIdx.x);
}

int main() {
  threadInBlock<<<2, 4>>>();
  cudaDeviceSynchronize();
}
```

which gives the output:

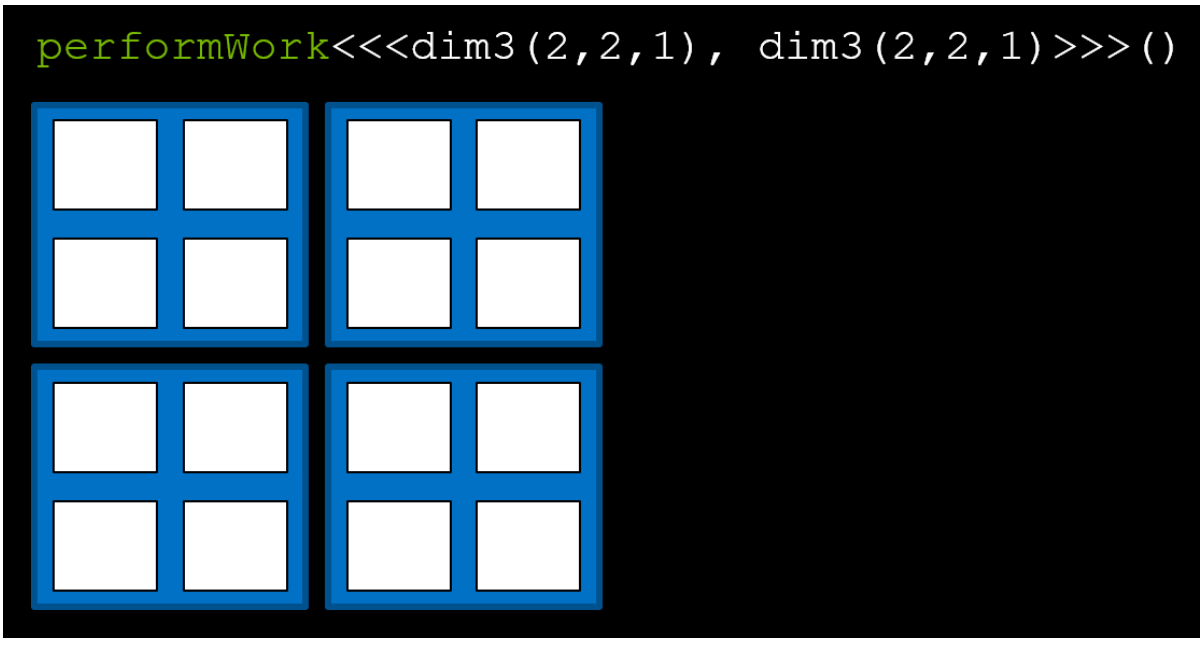
```
[6]  !nvcc ./first.cu -o first -run

     Thread 0 from block 1
     Thread 1 from block 1
     Thread 2 from block 1
     Thread 3 from block 1
     Thread 0 from block 0
     Thread 1 from block 0
     Thread 2 from block 0
     Thread 3 from block 0
```

There are two more variables `gridDim.x` and `blockDim.x`, meaning the number of blocks in a grid and the number of threads in a block respectively and **corresponding** to the two parameters we passed when launched the kernel function.

You might wonder why there is a `.x` after each variable. The truth is that both the blocks and threads can be place in 3 dimensions. With the help of class `dim3`, we can pass a multi-dimensional configuration into a kernel. This trick is only for convenience in some particular applications, for example matrix operations, and would do nothing to the performance.

```
1  performWork<<<dim3(2,2,1), dim3(2,2,1)>>>();
```

The hierarchy of the configuration above will look like this:

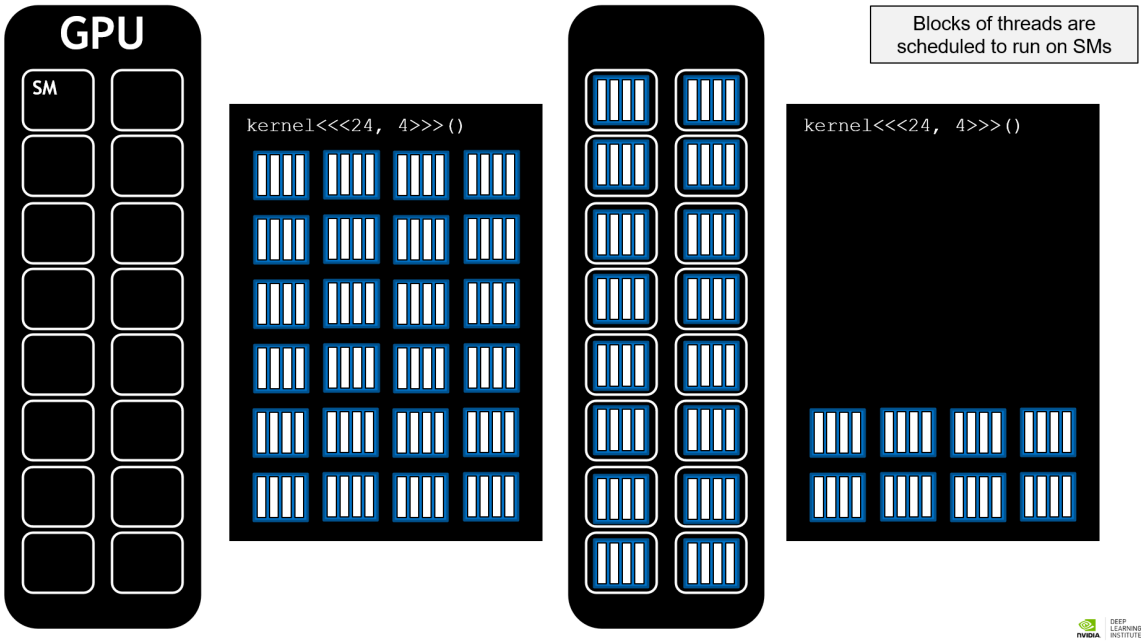


Now we can access these values with `gridDim.x`, `gridDim.y`, `gridDim.z`, `blockDim.x`, `blockDim.y`, `blockDim.z()`, `blockIdx.x`, `blockIdx.y`, `blockIdx.z`, `threadIdx.x`, `threadIdx.y`, `threadIdx.z`.

These values are important for many calculations, as we will discuss later with memory.

Streaming Multiprocessors

In the end this section, we explain **streaming multiprocessors** (SMs) and offer a simple rule for picking proper numbers for `numberOfBlocks` and `threadsPerBlock`.



SMs are basic units that execute tasks. As shown in the figure above, blocks are scheduled to run on SMs. When the number of blocks is multiple of the number of SMs, the execution will be efficient. Therefore, we cannot hardcode `numberOfBlocks` and may use an API to get a value for a particular machine instead. Usually, taking `numberOfBlocks` as `numberOfSMs` times 16, 32, or 64 would be a good choice.

As for `threadsPerBlock`, it can be any integer between 1 and 1024. 512 usually becomes a good choice.

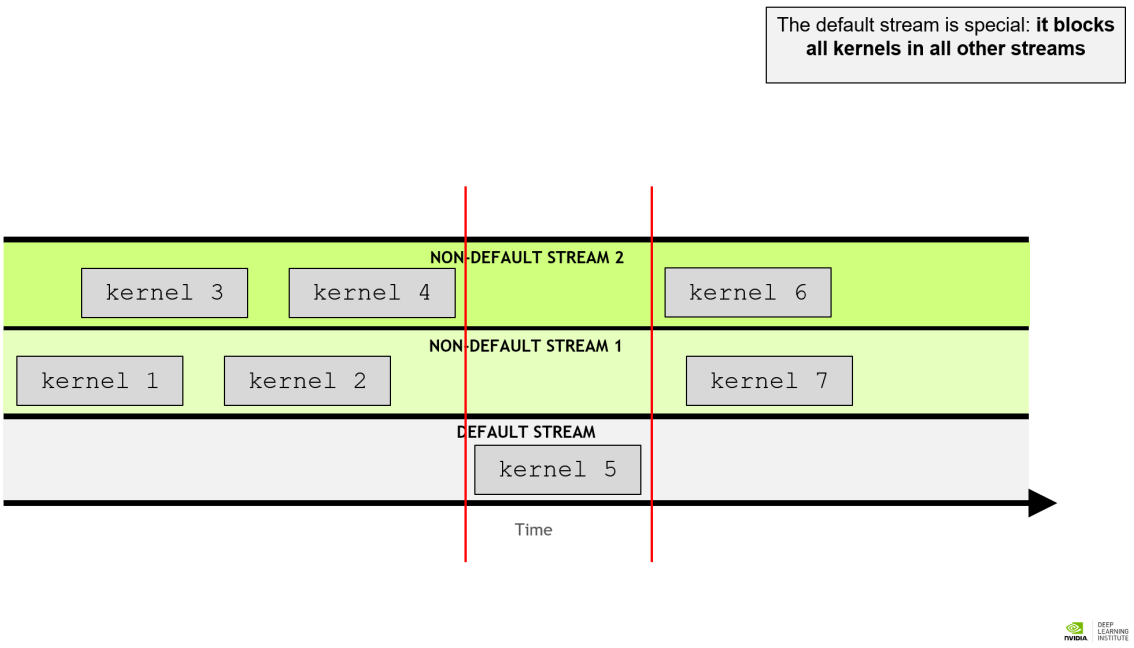
Here is an example that we call a CUDA API to get the number of SMs. The official reference for struct `cudaDeviceProp` is here.

```
int deviceId;
cudaGetDevice(&deviceId);

cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, deviceId);

int numberOfSMs = prop.multiProcessorCount;
```

Then, we can create a kernel configuration like this

```
int numberOfBlocks = numberOfSMs * 32;
int threadsPerBlock = 512;
some_kernel<<<numberOfBlocks, threadsPerBlock>>>();
```

CUDA Streams



GPU tasks are scheduled in **streams** and the kernels in one stream is **serial**. Before, we didn't specify the stream and the kernels are executed in the **default stream**. Default streaming is **blocking**. That is to mean, when there is a task scheduled in the default stream, all the tasks on the **non-default streams** will be blocked, while all the other non-default streams are **non-blocking**, allowing concurrent kernel execution. CUDA offers a stream class `cudaStream_t` and they are created with `cudaStreamCreate()` and destroyed with `cudaStreamDestroy()`. As we didn't mention before, a kernel configuration actually accepts 4 parameters, `numberOfBlocks`, `threadsPerBlock`, `sharedMemoryBytes`, and `stream`. We just left the latter 2 parameter in defaults before. Here is an example with concurrent streams.

```
cudaStream_t stream1;
cudaStream_t stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

some_kernel<<<numberOfBlocks, threadsPerblock, 0, stream1>>>();
some_kernel<<<numberOfBlocks, threadsPerblock, 0, stream2>>>();

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

In this part of program, the kernels in `stream1` and `stream2` will be concurrent. Since we don't need shared memory here, we just left them in 0. Note that although `cudaStreamDestroy()` returns immediately after calling, the streams are not actually destroyed until the kernels in the stream finish, so we don't need to worry about the side effect of destroying a stream.

CUDA Memory Management

So far, we have discussed how to launch a kernel running on device. However, such kernels could only access memory automatically allocated. That is, local variables in kernel function definition. We wish to allocate memory that can be accessed by both the host and the device. CUDA offers a unified memory model so that we don't need to worry about memory access.

Unified Memory

Allocation and Freeing

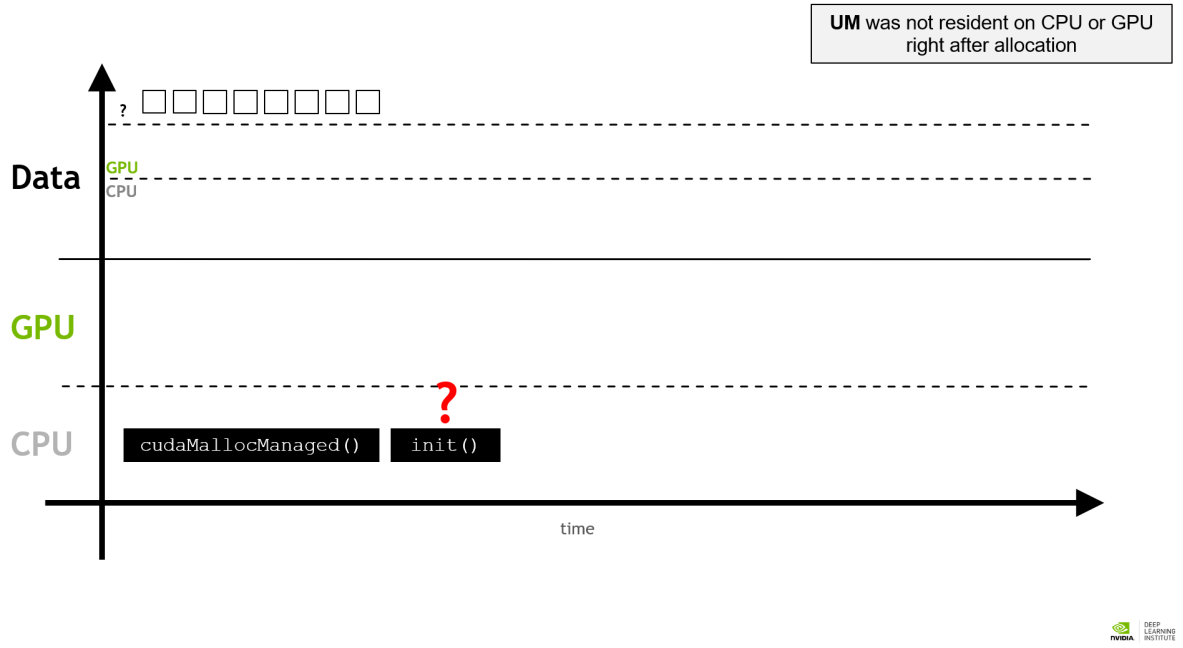
To allocate unified memory that can be accessed both on the device and the host, we call `cudaMallocManaged()` function.

```
int N = 2<<20;
int* array;
int size = N * sizeof(int);
cudaMallocManaged(&array, size);
```

Thus, the integer array can be accessed both on the device and the host. To free allocated unified memory, we call `cudaFree()` utility.

```
cudaFree(array);
```

Reduce Page Faults



When UM was initially allocated, it may not be resident on CPU or GPU. If the memory was first initialized by CPU then GPU, a page fault occurs. Memory will be transferred from host to device and slow down the tasks. Similarly, if UM was initialized on GPU and then accessed by CPU, a page fault occurs and memory transfer begins. The place memory is resident on depends on the last access. When a page fault is present, memory is transferred is small batch size. If we can predict a page fault, we can transfer the corresponding memory in advance with bigger batch size to increase the efficiency. CUDA offers an API called `cudaPrefetch` to perform such behaviors. Here is an example.

```
int deviceId;
cudaGetDevice(&deviceId);

// Allocate unified memory
int numberOfSMs = prop.multiProcessorCount;
int N = 2<<20;
int* array;
```

```
int size = N * sizeof(int);
cudaMallocManaged(&array, size);

// Initialize the UM on CPU, then access it on GPU
init_on_cpu(array, size);
cudaPrefetchAsync(array, size, deviceId); // Trasfer the memory from host to device
access_memory<<<numberOfBlocks, threadsPerBlock>>>();
cudaDeviceSynchronize();

// Free memory
cudaFree(array);
```

The third parameter of `cudaPrefetchAsync` specifies the direction of memory transfer. When filled with `deviceId`, the memory is transferred from host to device (HtoD), while `cudaCpuDeviceId` specifies a transfer from device to host (DtoH). Note that the variable `cudaCpuDeviceId` can directly be accessed globally and we need no APIs to get this variable.

Index Calculation

With unified memory, we can finally schedule some tasks on GPU. Let's look at an example. Suppose we have two vectors containing integers, each with length 2^22, and we want to accelerate vector addition with CUDA. Our number of threads might no be bigger enough to establish a bijection between thread indices and vector indices, so a thread must execute more than one addition. The convention is to define a variable `stride` that equals to the total number of threads in a (also the only) grid, and increase the loop index by `stride` each time. We assume the kernel configuration is 1-d and here is how we calculate index. Note that we are checking index boundary each time to avoid unexpected memory access.

```
__global__ void vectorAdd(int* a, int* b, int* res, int N) {
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;
  for (int i = idx; i < N; i += stride) {
    res[i] = a[i] + b[i];
  }
}
```

Manual Memory Management and Streaming

Basic Commands

Although UM is powerful enough, we may still want to manage the memory ourselves to further optimize the efficiency. Here are some commands for manually managing the memory.

- `cudaMalloc` will allocate the memory on GPU. The memory is **not** accessible on CPU.
- `cudaFree` frees the memory allocated on device.
- `cudaMallocHost` will allocate the memory on CPU just like what a normal `malloc` does. The memory will be page locked on host. Too many page locked memory would reduce CPU performance.
- `cudaFreeHost` frees the memory allocated on host.
- `cudaMemcpy` **copies** the memory DtoH or HtoD, instead of **transferring**.

- `cudaMemcpyAsync` allows **asynchronously** memory copying (explained later).

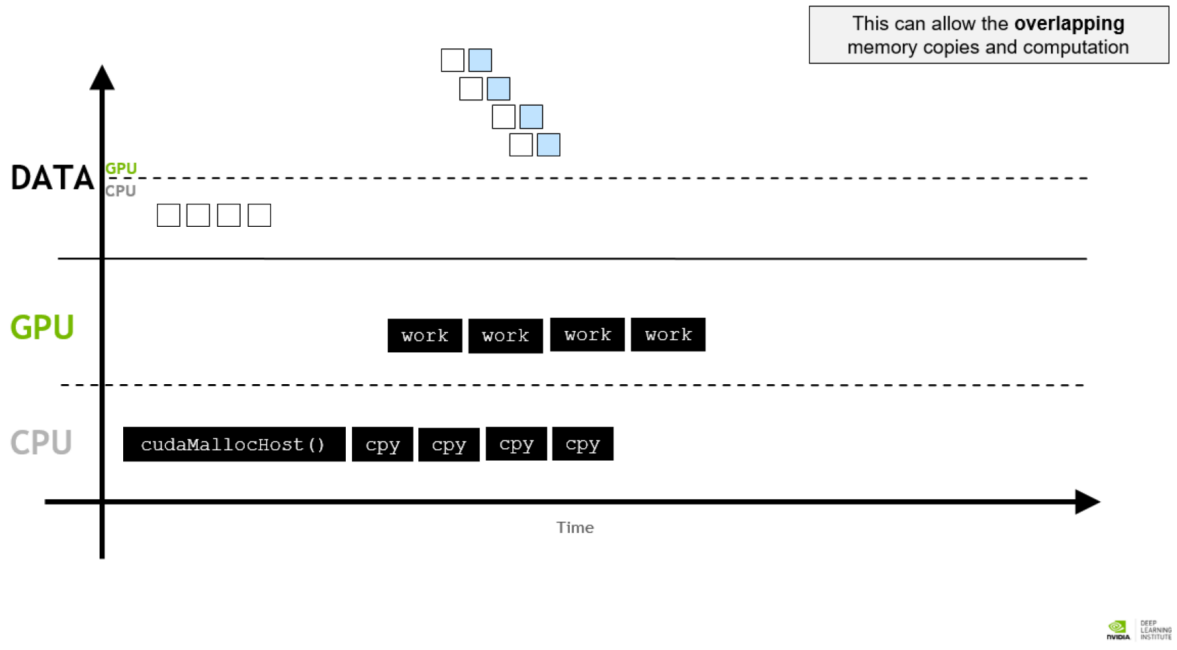
Let's look at an example.

```
int N = 2<<20;
int size = N * sizeof(int);
int* array_device;
int* array_host;
cudaMalloc(&array_device, size);
cudaMallocHost(&array_host, size);

init_on_cpu(array_host, N);
cudaMemcpy(array_device, array_host, size, cudaMemcpyHostToDevice);
some_kernel<<<blocks, threads, 0, stream>>>();
cudaMemcpy(array_host, array_device, size, cudaMemcpyDeviceToHost);

cudaFree(array_device);
cudaFreeHost(array_host);
```

Note that `cudaMemcpy` first accepts the destination pointer then source pointer, unlike Linux `cp` first-source-then-destination convention. `cudaMemcpyDeviceToHost` and `cudaMemcpyHostToDevice` are two variables that can be directly and globally accessed that specify the direction of memory copying.

Asynchronous Memory Copying

In the last example, memory copy starts after kernel finishes. To optimize the process, we can start asynchronous (or concurrent) memory copying right after a part of kernel finishes.



An simple approach is to divide the kernel in several segments. A segment of memory copying starts right after a segment of kernel finishes. We will refactor the former vector addition program and take it as an example.

```
// Perform vector addition in segments
  int numberOfSegments = 4;
  int segmentN = N / numberOfSegments;
```

```
  int segmentSize = size / numberOfSegments;
  for (int i = 0; i < numberOfSegments; i++) {
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    int offset = i * segmentN;
    vectorAdd<<<blocks, threads, 0, stream1>>>(a_device + offset, b_device + offset,
                                               c_device + offset, segmentN);
    cudaMemcpyAsync(c_host + offset, c_device + offset, segmentSize, cudaMemcpyDeviceToHost, stream);
    cudaStreamDestroy(stream);
  }
  cudaDeviceSynchronize();
```

The piece of program above divides the kernel `vectorAdd` in 4 segments. After one segment of kernel finishes, asynchronous memory copying starts. Every stream first executes the kernel then copying the corresponding memory to the host. Note that we must **carefully** handle all the indices here to avoid illegal memory access. After all of theses are done, the stream is destroyed (recall stream destruction behavior). The whole accelerated program is pasted below for your reference. Don't forget to destroy unused streams and free unused memories.

```
#include <stdio.h>

__global__ void init_on_gpu(int* a, int init_val, int N) {
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;
  for (int i = idx; i < N; i += stride) {
    a[i] = init_val;
  }
}

__global__ void vectorAdd(int* a, int* b, int* res, int N) {
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;
  for (int i = idx; i < N; i += stride) {
    res[i] = a[i] + b[i];
  }
}

int verify_on_cpu(int* a, int res, int N) {
  for (int i = 0; i < N; i++) {
    if (a[i] != res) {
      return 0;
    }
  }
  return 1;
}

int main() {
  // Variable declarations
  int N = 2<<20;
  int size = N * sizeof(int);
  int* a_device;
  int* b_device;
```

```
  int* c_device;
  int* c_host;
  cudaMalloc(&a_device, size);
  cudaMalloc(&b_device, size);
  cudaMalloc(&c_device, size);
  cudaMallocHost(&c_host, size);

  // Kernel configuration
  int deviceId;
  cudaGetDevice(&deviceId);
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, deviceId);
  int SMs = prop.multiProcessorCount;
  int blocks = SMs * 32;
  int threads = 512;

  // Stream creation
  cudaStream_t stream1;
  cudaStream_t stream2;
  cudaStream_t stream3;
  cudaStreamCreate(&stream1);
  cudaStreamCreate(&stream2);
  cudaStreamCreate(&stream3);

  // Initialize arrays on device
  init_on_gpu<<<blocks, threads, 0, stream1>>>(a_device, 2, N);
  init_on_gpu<<<blocks, threads, 0, stream2>>>(b_device, 3, N);
  init_on_gpu<<<blocks, threads, 0, stream3>>>(c_device, 0, N);
  cudaDeviceSynchronize();

  // Perform vector addition in segments
  int numberOfSegments = 4;
  int segmentN = N / numberOfSegments;
  int segmentSize = size / numberOfSegments;
  for (int i = 0; i < numberOfSegments; i++) {
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    int offset = i * segmentN;
    vectorAdd<<<blocks, threads, 0, stream1>>>(a_device + offset, b_device +
offset, c_device + offset, segmentN);
    cudaMemcpyAsync(c_host + offset, c_device + offset, segmentSize,
cudaMemcpyDeviceToHost, stream);
    cudaStreamDestroy(stream);
  }
  cudaDeviceSynchronize();

  // Verify results
  if (verify_on_cpu(c_host, 5, N)) {
    printf("Results are correct!\n");
  } else {
    printf("Results are incorrect!\n");
  }

  // Free memories
  cudaFree(a_device);
  cudaFree(b_device);
```

```
  cudaFree(c_device);
  cudaFreeHost(c_host);
}
```

Error Handling

CUDA usually does not report runtime errors, so we must detect and record them manually. CUDA offers a class `cudaError_t` to handle errors. The return values of CUDA APIs are `cudaError_t`, allowing us to catch error directly.

```
#include <stdio.h>
int main() {
  int* array;
  int size = -1;
  cudaError_t err;
  err = cudaMallocManaged(&array, size);
  if (err != cudaSuccess) {
  fprintf(stderr, "Error: %s\n", cudaGetErrorString(err));
  }
}
```

This program will throw an error because -1 is not a valid size.

```
Error: out of memory
```

However, the return values of custom kernels are `void`, meaning we cannot catch error in the same way when launching those kernels. CUDA offers `cudaGetLastError` to catch the last error thrown. Also, we can create a utility function to encapsulate such processes.

```
inline void checkCudaError() {
  cudaError_t err;
  err = cudaGetLastError();
  if (err != cudaSuccess) {
    fprintf(stderr, "Error: %s\n", cudaGetErrorString(err));
    assert(err == cudaSuccess);
  }
}
```

Therefore, we can check the errors when launching kernels.

```
some_kernel<<<-1, -1>>>();
checkCudaError();
```

which yields

```
Error: invalid configuration argument
```

Such error handling may help debug the CUDA program.

Performance Profiling

NVIDIA Nsight Systems command line tool (nsys) is a command line profiler that will gather following information:

- Profile configuration details
- Report file(s) generation details
- **CUDA API Statistics**
- **CUDA Kernel Statistics**
- **CUDA Memory Operation Statistics (time and size)**
- OS Runtime API Statistics

To install `nsys` in Colab, run the following commands:

```
!wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/nsight-systems-2023.2.3_2023.2.3.1001-1_amd64.deb
!apt update
!apt install ./nsight-systems-2023.2.3_2023.2.3.1001-1_amd64.deb
!apt --fix-broken install
```

Then, we can profile the running information of our CUDA program. Here we take vector addition as an example. `profile` means we want to profile the executable and `--stats=true` tells `nsys` to print all the information.

```
!nsys profile --stats=true vector-add
```

A part of printed information is pasted below:

```
[5/8] Executing 'cuda_api_sum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)     Med (ns)    Min (ns)    Max (ns)     StdDev (ns)               Name
 --------  ---------------  ---------  ------------  ----------  ---------  -----------  -------------  ---------------------------------
     96.0      192,376,747          3  64,125,582.3   102,474.0     94,203  192,180,070  110,898,439.5  cudaMalloc
      2.0        4,032,889          1   4,032,889.0  4,032,889.0  4,032,889    4,032,889            0.0  cudaHostAlloc
      0.9        1,727,437          1   1,727,437.0  1,727,437.0  1,727,437    1,727,437            0.0  cudaFreeHost
      0.5          946,023          3     315,341.0    389,557.0    163,210      393,256      131,762.3  cudaFree
      0.4          718,243          2     359,121.5    359,121.5     97,247      620,996      370,346.5  cudaDeviceSynchronize
      0.1          280,743          7      40,106.1      6,209.0      5,024      229,250       83,630.6  cudaLaunchKernel
      0.1          134,190          1     134,190.0    134,190.0    134,190      134,190            0.0  cudaGetDeviceProperties_v2_v12000
      0.0           73,363          7      10,480.4      4,131.0      2,853       47,072       16,168.7  cudaStreamCreate
      0.0           27,767          4       6,941.8      5,098.5      4,612       12,958        4,034.3  cudaMemcpyAsync
      0.0           15,667          4       3,916.8      3,328.5      3,175        5,835        1,282.0  cudaStreamDestroy
      0.0            2,002          1       2,002.0      2,002.0      2,002        2,002            0.0  cuModuleGetLoadingMode

[6/8] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------------
     54.7          141,309          4  35,327.3  35,311.5    35,071    35,615        241.0  vectorAdd(int *, int *, int *, int)
     45.3          116,989          3  38,996.3  40,479.0    34,783    41,727      3,701.8  init_on_gpu(int *, int, int)

[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)      Operation
 --------  ---------------  -----  ---------  ---------  --------  --------  -----------  ------------------
    100.0          660,782      4  165,195.5  163,995.5   160,699   172,092      5,186.2  [CUDA memcpy DtoH]

[8/8] Executing 'cuda_gpu_mem_size_sum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  --------  -----------  ------------------
      8.389      4     2.097     2.097     2.097     2.097        0.000  [CUDA memcpy DtoH]
```

Here we can trace the running time of kernels and the memory behavior. You can compare the results of different `numberOfBlocks` and see what kind of `memcpy` page faults will bring about.

Final Exercise

An n-body simulator predicts the individual motions of a group of objects interacting with each other gravitationally. Below is a simple, though working, n-body simulator for bodies moving through 3 dimensional space.

In its current CPU-only form, this application takes about 5 seconds to run on 4096 particles, and **20 minutes** to run on 65536 particles. Your task is to GPU accelerate the program, retaining the correctness of the simulation.

Considerations to Guide Your Work

Here are some things to consider before beginning your work:

- Especially for your first refactors, the logic of the application, the `bodyForce` function in particular, can and should remain largely unchanged: focus on accelerating it as easily as possible.
- The code base contains a for-loop inside `main` for integrating the interbody forces calculated by `bodyForce` into the positions of the bodies in the system. This integration both needs to occur after `bodyForce` runs, and, needs to complete before the next call to `bodyForce`. Keep this in mind when choosing how and where to parallelize.
- Use a **profile driven** and iterative approach.
- You are not required to add error handling to your code, but you might find it helpful, as you are responsible for your code working correctly.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#include "files.h"

#define SOFTENING 1e-9f

/*
 * Each body contains x, y, and z coordinate positions,
 * as well as velocities in the x, y, and z directions.
 */

typedef struct { float x, y, z, vx, vy, vz; } Body;

/*
 * Calculate the gravitational impact of all bodies in the system
 * on all others.
 */

void bodyForce(Body *p, float dt, int n) {
  for (int i = 0; i < n; ++i) {
    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

    for (int j = 0; j < n; j++) {
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float dz = p[j].z - p[i].z;
```

```
      float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
      float invDist = rsqrtf(distSqr);
      float invDist3 = invDist * invDist * invDist;

      Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
    }

    p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
  }
}


int main(const int argc, const char** argv) {

  // The assessment will test against both 2<11 and 2<15.
  // Feel free to pass the command line argument 15 when you generate ./nbody report files
  int nBodies = 2<<11;
  if (argc > 1) nBodies = 2<<atoi(argv[1]);

  // The assessment will pass hidden initialized values to check for correctness.
  // You should not make changes to these files, or else the assessment will not work.
  const char * initialized_values;
  const char * solution_values;

  if (nBodies == 2<<11) {
    initialized_values = "files/initialized_4096";
    solution_values = "files/solution_4096";
  } else { // nBodies == 2<<15
    initialized_values = "files/initialized_65536";
    solution_values = "files/solution_65536";
  }

  if (argc > 2) initialized_values = argv[2];
  if (argc > 3) solution_values = argv[3];

  const float dt = 0.01f; // Time step
  const int nIters = 10;  // Simulation iterations

  int bytes = nBodies * sizeof(Body);
  float *buf;

  buf = (float *)malloc(bytes);

  Body *p = (Body*)buf;

  read_values_from_file(initialized_values, buf, bytes);

  double totalTime = 0.0;

  /*
   * This simulation will run for 10 cycles of time, calculating gravitational
   * interaction amongst bodies, and adjusting their positions to reflect.
```

```
   */

  for (int iter = 0; iter < nIters; iter++) {
    StartTimer();

  /*
   * You will likely wish to refactor the work being done in `bodyForce`,
   * and potentially the work to integrate the positions.
   */

    bodyForce(p, dt, nBodies); // compute interbody forces

  /*
   * This position integration cannot occur until this round of `bodyForce` has completed.
   * Also, the next round of `bodyForce` cannot begin until the integration is complete.
   */

    for (int i = 0 ; i < nBodies; i++) { // integrate position
      p[i].x += p[i].vx*dt;
      p[i].y += p[i].vy*dt;
      p[i].z += p[i].vz*dt;
    }

    const double tElapsed = GetTimer() / 1000.0;
    totalTime += tElapsed;
  }

  double avgTime = totalTime / (double)(nIters);
  float billionsOfOpsPerSecond = 1e-9 * nBodies * nBodies / avgTime;
  write_values_to_file(solution_values, buf, bytes);

  // You will likely enjoy watching this value grow as you accelerate the application,
  // but beware that a failure to correctly synchronize the device might result in
  // unrealistically high values.
  printf("%0.3f Billion Interactions / second", billionsOfOpsPerSecond);

  free(buf);
}
```