Supercomputing

Hands-On 6: Portable Parallel Programming with CUDA

2022

Contents

Introduction

This Hands-on comprises 2 sessions. Next table shows the documents and files needed to develop each one of the exercises.

Session 1	Portable Sequential Code	`saxpy.c`, and `saxpy.cu`
Session 2	Unified Memory (cudaMallocManaged)	`saxpy-cudaMallocManaged.cu`

Please remind that in order to compile CUDA programs, we should include the proper compilation option, such as:

```
$ nvcc saxpy.cu -o saxpy
```

To execute an 1GPU in CUDA program with several threads, you can use the following example (changing the of the problem accordingly):

```
$ ./saxpy 10
```

1 Add Vectors Benchmark

This subprograms perform the following computation, using the scalar α and vectors x and y:

$$z = \alpha x + y,$$

where x, y, and z are vectors and α is scalar. SAXPY stands for Single-Precision it is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. SAXPY is a combination of scalar multiplication and vector addition, and it is very simple: it takes as input two vectors of 32-bit floats x and y with n elements each, and a scalar value α. It multiplies each element $x[i]$ by α and adds the result to $y[i]$. A simple C implementation looks like this.

```
...
void saxpy(int n,  float *x, float *y){
  for (int i = 0; i < n; i++)
   y[i] = x[i] + y[i];

}
...
```

Figure 1: Sequential SAXPY code.

Given this basic example code, I can now show you steps ways to SAXPY on GPUs. Note that I chose SAXPY because it is a really short and simple code, but it shows enough of the syntax of each CUDA programming approach to compare them. Because it is so simple, and does very little computation, SAXPY is not really a great computation to use for comparing the performance of the different approaches. The principal goal is to demonstrate ways to program on the CUDA platform today, not to suggest that any one is better than any other.

1.1 Include the CUDA library

CUDA provides extensions for many common programming languages, in the case of this lab, C/C++. These language extensions easily allow developers to run functions in their source code on a GPU. Then the next step will be to modify the code in the file `saxpy.c` to include the call to CUDA library and the rename the code with the extension *.cu (`saxpy.cu`).

```
...
#include <cuda.h>
...
```

Figure 2: Include the CUDA library in the code `saxpy.cu`.

1.2 Allocating memory to be accessed on the GPU and the CPU

The next step will be to modify the code in the file `saxpy.cu` to insert allocation memory on GPU. A first approach could be to use the command `cudaMalloc` considering the variables xd, and yd. After this change, you can insert the command to transfer the data to GPU with `cudaMemcpy` orientate from host to device.

```
...
float *xd, *yd;

  cudaMalloc( (void**)&xd, sizeof(float) * n );
  cudaMalloc( (void**)&yd, sizeof(float) * n );

  cudaMemcpy(xd, x, sizeof(float) * n, cudaMemcpyHostToDevice);
  cudaMemcpy(yd, y, sizeof(float) * n, cudaMemcpyHostToDevice);
...
```

Figure 3: Allocation memory and copy data on GPU in the code `saxpy.cu`.

1.3 Create the grid for computation on GPU

The execution configuration allows programmers to specify details about launching the kernel to run in parallel on multiple GPU threads. More precisely, the execution configuration allows programmers to specify how many groups of threads - called thread blocks, or just blocks - and how many threads they would like each thread block to contain. The syntax for this is: `<<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK>>>`.

```
...
 int NUMBER_OF_BLOCKS = 1;
 int NUMBER_OF_THREADS_PER_BLOCK = n;

 saxpy<<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK >>>(n, xd, yd);

 cudaDeviceSynchronize();
...
```

Figure 4: Launch Parallel Kernels in the code `saxpy.cu`.

However, unlike a normal sequential program on your host (The CPU) will continue to execute the next lines of code in your program. The command `cudaDeviceSynchronize` makes the host (The CPU) wait until the device (The GPU) have finished executing ALL the threads you have started, and thus your program will continue as if it was a normal sequential program.

1.4 Compose the Kernel function

The `__global__` keyword indicates that the following function will run on the GPU, and can be invoked globally, which in this context means either by the CPU, or, by the GPU. Often, code executed on the CPU is referred to as host code, and code running on the GPU is referred to as device code. Notice the return type void. It is required that functions defined with the `__global__` keyword return type void.

Each thread is given an index within its thread block, starting at 0. Additionally, each block is given an index, starting at 0. Just as threads are grouped into thread blocks, blocks are grouped into a grid, which is the highest entity in the CUDA thread hierarchy. In summary, CUDA kernels are executed in a grid of 1 or more blocks, with each block containing the same number of 1 or more threads. CUDA kernels have access to special variables identifying both the index of the thread (within the block) that is executing the kernel, and, the index of the block (within the grid) that the thread is within. These variables are `threadIdx.x` and/or `blockIdx.x`. In this example the choice was a unidimensional grid, only `threadIdx.x`:

```
...
 __global__ void saxpy(int n,  float *x, float *y){
 int i = threadIdx.x;
 if(i < n)
   y[i] = x[i] + y[i];
}
...
```

Figure 5: The kernel CUDA for SAXPY algorithm in the code `saxpy.cu`.

1.5 Copy the data from GPU to Host

To allocate, and obtain a pointer that can be referenced in both host and device code, replace calls to cudaMallocManaged, orientate from device to host as in the following example:

```
...
cudaMemcpy(y, yd, sizeof(float) * (n), cudaMemcpyDeviceToHost);
...
```

Figure 6: Copy the data from GPU to Host in the code `saxpy.cu`.

1.6 Free the GPU memory

To free memory, and obtain a pointer that can be referenced in both host and device code, free with cudaFree as in the following example:

```
...
  cudaFree(xd);
  cudaFree(yd);
...
```

Figure 7: Free the GPU memory in the code `saxpy.cu`.

2 Unified Memory (`cudaMallocManaged`)

The program in `saxpy-cudaMallocManaged.cu` allocates memory, using `cudaMallocManaged` for a n elements array of integers, and then seeks to initialize all the values of the array in parallel using a CUDA kernel.

From the previous action answer the following questions:

- How Unified Memory works?
- Is `cudaMallocManaged` slower than `cudaMalloc`?

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

__global__ void saxpy(int n,  float *x, float *y){
 int i = threadIdx.x;
 if(i < n)
   y[i] = x[i] + y[i];
}

void printVector(float *vector, int n){
for (int i=0; i < n ; ++i)
 printf("%1.0f\t", vector[i]);
printf("\n\n");
}

void generateVector(float *vector, int n){
for (int i=0; i < n ; ++i)
 vector[i] = i + 1;
}

int main(int argc, char *argv[]){
  int n = atoi(argv[1]);
  float *x,*y;

  cudaMallocManaged(&x, sizeof(float) * n);
  cudaMallocManaged(&y, sizeof(float) * n);

  generateVector(x, n);
  printVector(x, n);
  generateVector(y, n);
  printVector(y, n);

  int NUMBER_OF_BLOCKS = 1;
  int NUMBER_OF_THREADS_PER_BLOCK = n;

  saxpy <<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK >>> (n, x, y);

  cudaDeviceSynchronize();

  printVector(y, n);

  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

Figure 8: Allocates memory using `cudaMallocManaged` in the code `saxpy-cudaMallocManaged.cu`.

References

[1] P. Vingelmann, H.P. Fitzek, NVIDIA CUDA, release: 10.2.89, `https://developer.nvidia.com/cuda-toolkit`, 2020.