Assignment II: CUDA Basics

Amirhossein Namazi, Calin Capitanu

November 13, 2021

Exercise 1

Hello World!

The program is compiled using the compiler for CUDA programs, **nvcc**. Since the GPU that the code is ran on has the Ampere architecture (30 series Nvidia), the **-arch=sm_80** flag is given. Finally, the source code and an output binary name are given, all resulting in this command:

```
nvcc -arch=sm_80 exercise_1.cu -o exercise_1}
```

CUDA programs take advantage of the parallel processing of the GPU using CUDA Threads. These threads are independent execution threads that execute one specific action (one function, or "kernel" in the CUDA Jargon). These CUDA threads are grouped into CUDA thread blocks. Each thread block has the same number of threads, defined at the beginning of launching a kernel. When talking about memory management, blocks of CUDA threads can share one type of memory, while single threads also have their own memory (registers). Obviously, each thread in the thread block executes the same kernel (or function).

Performing SAXPY on the GPU

The problem of the number of blocks when **ARRAY_SIZE** is not a multiple of the number of threads per block is easily fixable with the following formula for the number of blocks:

$BLOCKS = (ARRAY\_SIZE + TPB - 1)/TPB$

This makes sure that all of the computations will have one thread to execute on, even if *ARRAY_SIZE % TPB != 0.*

The first time analysis we did was using the cpu time seconds retrieved from the system function *gettimeofday()*. Results were surprizing for this, by varying from 10,000 items in an array, all the way to 1,000,000,000. All of the runs, the CPU time was better than the one of the GPU. There are two explanations that we found possible from this:

1. A lot of the time is consumed on the transfer of data from the CPU to the GPU through the PCIe lanes, which are way smaller compared to the ones internal to the CPU, or internal in the GPU.
2. The second reason, which might be specific to the machine this has been run on, is the fact that the CPU has 32 execution threads at a really efficient IPC, which leads to really good times, and in the case *nvcc* is able to optimize things on the CPU to be ran in parallel (which I highly doubt), it could yield interesting results on it as well.

Some of the results when running with varying sizes of the *ARRAY_SIZE*:

```
Computing SAXPY on the CPU... Done! Took: 0.000122 seconds
Computing SAXPY on the GPU... Done! Took: 0.000221 seconds
Comparing the output for each implementation... Correct

Computing SAXPY on the CPU... Done! Took: 0.011765 seconds
Computing SAXPY on the GPU... Done! Took: 0.012260 seconds
Comparing the output for each implementation... Correct

Computing SAXPY on the CPU... Done! Took: 0.123760 seconds
Computing SAXPY on the GPU... Done! Took: 0.125628 seconds
Comparing the output for each implementation... Correct

Computing SAXPY on the CPU... Done! Took: 1.213933 seconds
Computing SAXPY on the GPU... Done! Took: 1.233103 seconds
Comparing the output for each implementation... Correct
```

Unfortunately, running *nvprof* did not work as expected, since we received the following error:

```
======== Warning: nvprof is not supported on devices with compute capability 8.0 and higher.
                  Use NVIDIA Nsight Systems for GPU tracing and CPU sampling and NVIDIA
    Nsight Compute for GPU profiling.
                  Refer https://developer.nvidia.com/tools-overview for more details.
```

CUDA simulation and GPU Profiling

Bonus Exercise

Calculating PI with CUDA

Appendix

Exercise 1 Code

```c
#include <stdio.h>
#define TPB 256


__global__ void helloWorldKernel(){
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  printf("Hello World! My threadId: %d\n", idx);
}

int main(){

  helloWorldKernel<<<1, TPB>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Exercise 2 Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define TPB 256

#define CPU true
#define GPU true
#define ARRAY_SIZE 10000

#define EPSILON 0.001

double cpuSecond() {
   struct timeval tp;
   gettimeofday(&tp,NULL);
   return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);
}

__global__ void SAXPY_GPU(float *d_x, float *d_y, const float a){
  const int idx = threadIdx.x + blockDim.x * blockIdx.x;
  d_y[idx] = d_x[idx] * a + d_y[idx];
}

void SAXPY_CPU(float *x, float *y, const float a){
  for(int i = 0; i < ARRAY_SIZE; i++){
    y[i] = a*x[i] + y[i];
  }
}

int main(){

  float *x, *y, *y_gpu;
  x = (float *) malloc(sizeof(float) * ARRAY_SIZE);
  y = (float *) malloc(sizeof(float) * ARRAY_SIZE);
  y_gpu = (float *) malloc(sizeof(float) * ARRAY_SIZE);

  for(int i = 0; i < ARRAY_SIZE; i++){
    x[i] = rand() % 100;
    y[i] = rand() % 100;
  }
  float a = 3.45;

  float *d_x, *d_y;
  cudaMalloc(&d_x, sizeof(float) * ARRAY_SIZE);
  cudaMalloc(&d_y, sizeof(float) * ARRAY_SIZE);

  cudaMemcpy(d_x, x, sizeof(float) * ARRAY_SIZE, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, sizeof(float) * ARRAY_SIZE, cudaMemcpyHostToDevice);

  printf("Computing SAXPY on the CPU...");
  double start_cpu = cpuSecond();
  SAXPY_CPU(x,y,a);
  printf("Done! Took: %f seconds\n", cpuSecond() - start_cpu);

  printf("Computing SAXPY on the GPU...");
  double start_gpu = cpuSecond();
  SAXPY_GPU<<<(ARRAY_SIZE + TPB - 1)/TPB, TPB>>>(d_x, d_y, a);
  cudaDeviceSynchronize();
  printf("Done! Took: %f seconds\n", cpuSecond() - start_cpu);

  cudaMemcpy(y_gpu, d_y, sizeof(float) * ARRAY_SIZE, cudaMemcpyDeviceToHost);

  bool comp = true;
  for(int i = 0; i < ARRAY_SIZE; i++){
    if(abs(y[i] - y_gpu[i]) > EPSILON){
      comp = false;
      printf("%f\n", abs(y[i] - y_gpu[i]));
    }
  }

  printf("Comparing the output for each implementation...");
  if(comp)
    printf("Correct\n");
  else
    printf("Incorrect\n");



  return 0;
}
```