TÉCNICO
LISBOA

Assignment 2

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2022/2023

Overview

In this assignment you will write a couple of parallel programs in CUDA. While this assignment is not a very challenging one, efficiently parallelizing the scan may require thinking in a data parallel manner and some code performance tuning.

Deadlines and Submission

You will submit the work developed for Lab2 via Fenix (until 21st of October at 23h59, i.e., Friday).

The submission should be made in a single `zip` file with the following content:

- Your report (writeup) in a file called `report.pdf` (it must be pdf)
- Your implementation (codes) for <u>all</u> programs (to keep submission small, please do a `make clean` in program directories prior to creating the archive, and remove any residual output, etc.)

When handed in, all codes must be compilable and runnable out of the box on the `cuda` machines!

Your report should have <u>at most</u> 10 pages and should be well structured. For each part (problem) you should: elaborate your solution, explain your rationale (how did you arrive there, why it makes sense, describe all (if any) specific measurements/experiments that were conducted by you to prove your hypothesis etc). You should also discuss and analyze the obtained results!

<u>NOTE</u>: For grading purposes, we expect you to report on the performance of code run on the `cuda` machine. However, for development, you may also want to run the programs on your own machine.

Environment Setup

For this assignment, you will need to run (the final version of) your code on `cuda1` or `cuda2` machine (hostname: `{cuda1,cuda2}.scdeec.tecnico.ulisboa.pt`). These machines contain NVIDIA GeForce GTX 1070 GPU with the following characteristics:

```
Device 0: "NVIDIA GeForce GTX 1070"
  CUDA Driver Version / Runtime Version          11.4 / 10.1
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8120 MBytes (8513978368 bytes)
  (15) Multiprocessors, (128) CUDA Cores/MP:     1920 CUDA Cores
  GPU Max Clock rate:                            1721 MHz (1.72 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

The CUDA C programmer's guide PDF version[1] or web version[2] is an excellent reference for learning how to program in CUDA (you can also find some CUDA SDK samples on `cuda` machines in `usr/local/cuda/samples` folder, if you think that you need more codes to practice). In addition, there are a wealth of CUDA tutorials/examples on the web (just Google!) and on the NVIDIA developer site[3].

The access to `cuda` machines is performed through `ssh`, by using the command:

```
ssh csph<group_number>@{cuda1,cuda2}.scdeec.tecnico.ulisboa.pt
```

where you should substitute *<group_number>* with your group number (e.g., the username for the group number 3 is **`csph3`**).

To access the account, you should use the password that you had chosen and set during our last lab session. You should maintain this password until the end of this course, i.e., the responsible stuff from the CSPH course (read: Prof. Aleksandar) should always have access to your account. The accounts that we can't access, will not be graded!

IMPORTANT: Since `cuda` machines are shared resources, please verify if nobody else is performing experiments before running your program, in order not to affect the results of other groups. To check the users logged in the machine, you can use the **`who`** command. It is highly recommended to schedule the utilization of `cuda` machines among yourselves to avoid the simultaneous use.

The assignment starter code is available on the course webpage (section: Labs).

Part 1: CUDA SAXPY

To gain a bit of practice writing CUDA programs your warm-up task is to re-implement the SAXPY function from Assignment 1 in CUDA. Starter code for this part of the assignment is located in the `/saxpy` directory of the assignment tarball.

Please finish off the implementation of SAXPY in the function `saxpyCuda` in `saxpy.cu`. You will need to allocate device global memory arrays and copy the contents of the host input arrays `X`, `Y`, and `result` into CUDA device memory prior to performing the computation. After the CUDA computation is complete, the result must be copied back into host memory. Please see the definition of `cudaMemcpy` function in Programmer's Guide or take a look at the helpful tutorial pointed to in the assignment starter code.

As part of your implementation, add timers around the CUDA kernel invocation in `saxpyCuda`. After your additions, your program should time two executions:

- The provided starter code contains timers that measure **the entire process** of copying data to the GPU, running the kernel, and copying data back to the CPU.

[1] https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[3] https://docs.nvidia.com/cuda/

- You should also insert timers the measure *only the time taken to run the kernel*. (They should not include the time of CPU-to-GPU data transfer or transfer of results from the GPU to the CPU.)

When adding your timing code in the latter case, you'll need to be careful: By default, a CUDA kernel's execution on the GPU is *asynchronous* with the main application thread running on the CPU. For example, if you write code that looks like this:

```
double startTime = CycleTimer::currentSeconds();
saxpy_kernel<<<blocks, threadsPerBlock>>>(N, alpha, device_x, device_y, device_result);
double endTime = CycleTimer::currentSeconds();
```

You'll measure a kernel execution time that seems amazingly fast! (Because you are only timing the cost of the API call itself, not the cost of actually executing the resulting computation on the GPU).

Therefore, you will want to place a call to `cudaDeviceSynchronize()` following the kernel call to wait for completion of all CUDA work on the GPU. This call to `cudaDeviceSynchronize()` returns when all prior CUDA work on the GPU has completed. Note that `cudaDeviceSynchronize()` is not necessary after the `cudaMemcpy()` to ensure the memory transfer to the GPU is complete, since `cudaMempy()` is synchronous under the conditions we are using it.

```
double startTime = CycleTimer::currentSeconds();
saxpy_kernel<<<blocks, threadsPerBlock>>>(N, alpha, device_x, device_y, device_result);
cudaDeviceSynchronize();
double endTime = CycleTimer::currentSeconds();
```

Note that in your measurements that include the time to transfer to and from the CPU, a call to `cudaDeviceSynchronize()` **is not** necessary before the final timer (after your call to `cudaMemcopy()` that returns data to the CPU) because `cudaMemcpy()` will not return to the calling thread until after the copy is complete.

Questions:

1. What performance do you observe compared to the sequential CPU-based implementation of SAXPY (recall your results from `saxpy` on Program 5 from Assignment 1)?
2. Compare and explain the difference between the results provided by two sets of timers (timing only the kernel execution vs. timing the entire process of moving data to the GPU and back in addition to the kernel execution). Are the bandwidth values observed *roughly* consistent with the reported bandwidths available to the different components of the machine? (Hint: You should use the web to track down the memory bandwidth of an NVIDIA GTX 1070 GPU, and the maximum transfer speed of the computer's PCIe-x16 bus. It's PCIe 3.0[4], and a 16-lane bus connecting the CPU with the GPU.)

[4] https://en.wikipedia.org/wiki/PCI_Express

Part 2: CUDA Parallel Prefix-Sum

Now that you're familiar with the basic structure and layout of CUDA programs, as a second exercise you are asked to come up with parallel implementation of the function `find_repeats` which, given a list of integers `A`, returns a list of all indices `i` for which `A[i] == A[i+1]`.

For example, given the array `{1,2,2,1,1,1,3,5,3,3}`, your program should output the array `{1,3,4,8}`.

Exclusive Prefix Sum

We want you to implement `find_repeats` by first implementing parallel exclusive prefix-sum operation. Exclusive prefix sum takes an array `A` and produces a new array output that has, at each index `i`, the sum of all elements up to but not including `A[i]`. For example, given the array `A={1,4,6,8,2}`, the output of exclusive prefix sum `output={0,1,5,11,19}`.

The following code is a recursive C-code implementation of a work-efficient, parallel implementation of scan. **Note: Some of you may wish to skip the following recursive implementation and jump to the iterative version below.**

```
void exclusive_scan_recursive(int* start, int* end, int* output, int* scratch) {
    int N = end - start;
    if (N == 0)
        return;
    else if (N == 1) {
        output[0] = 0;
        return;
    }
    // sum pairs in parallel.
    for (int i = 0; i < N/2; i++)
       output[i] = start[2*i] + start[2*i+1];
    // prefix sum on the compacted array.
    exclusive_scan_recursive(output, output + N/2, scratch, scratch + (N/2));
    // finally, update the odd values in parallel.
    for (int i = 0; i < N; i++) {
        output[i] = scratch[i/2];
        if (i % 2)
            output[i] += start[i-1];
    }
}
```

While the above code expresses our intent well and recursion is in fact supported on modern GPUs, its use can lead to fairly low performance. Instead, we can express the algorithm in an iterative manner.

The following "C-like" code is an iterative version of scan. In the pseudocode, we use `parallel_for` to indicate potentially parallel loops.

```
void exclusive_scan_iterative(int* start, int* end, int* output) {
    int N = end - start;
    memmove(output, start, N*sizeof(int));

    // upsweep phase
    for (int two_d = 1; two_d < N/2; two_d*=2) {
        int two_dplus1 = 2*two_d;
        parallel_for (int i = 0; i < N; i += two_dplus1) {
            output[i+two_dplus1-1] += output[i+two_d-1];
        }
    }

    output[N-1] = 0;

    // downsweep phase
    for (int two_d = N/2; two_d >= 1; two_d /= 2) {
        int two_dplus1 = 2*two_d;
        parallel_for (int i = 0; i < N; i += two_dplus1) {
            int t = output[i+two_d-1];
            output[i+two_d-1] = output[i+two_dplus1-1];
            output[i+two_dplus1-1] += t;
        }
    }
}
```

We would like you to use this algorithm to implement a version of parallel prefix sum in CUDA. You must implement `exclusive_scan` function in `scan/scan.cu`. Your implementation will consist of both host and device code. The implementation will require multiple CUDA kernel launches (e.g., one for each `parallel_for` loop in the pseudocode above seems like a reasonable starting point) and **you are not allowed to merge upsweep and downsweep phases**.

Note: In the starter code, the `cudaScan` function does not assume that the input array's length (`N`) is a power of 2. We solve this problem by rounding the input array length to the next power of 2 when allocating the corresponding buffers on the GPU. However, the code only copies back `N` elements from the GPU buffer back to the CPU buffer. This fact should simplify your CUDA implementation.

Compilation produces the binary `cudaScan`. Command line usage is as follows:

```
Usage: ./cudaScan [options]

Program Options:
  -m  --test <TYPE>    Run specified function on input.  Valid tests are: scan, find_repeats
(default: scan)
  -i  --input <NAME>   Run test on given input type. Valid inputs are: ones, random (default:
random)
  -n  --arraysize <INT>  Number of elements in arrays
  -t  --thrust           Use Thrust library implementation
  -?  --help             This message
```

Implementing "Find Repeats" Using Prefix Sum

Once you have written `exclusive_scan`, implement the function `find_repeats` in `scan/scan.cu`. This will involve writing more device code, in addition to one or more calls to `exclusive_scan()`. Your code should write the list of indices of repeated elements into the provided output pointer (in device memory), and then return the size of the output list.

When calling your `exclusive_scan` implementation, remember that the contents of the start array are copied over to the output array. Also, the arrays passed to `exclusive_scan` are assumed to be in device memory.

Grading: We will test your code for correctness and performance on random input arrays. Consider that your implementation must **correctly** process big arrays, e.g., array with 1M, 10M, 20M and 40M elements! Please consult `Ref_Timings.txt` file for reference timings. You should be able to easily beat those times with your baseline implementation, i.e., the implementation with almost no performance tuning (just a direct port of the algorithm pseudocode to CUDA).

Important note: Your job is to come up with the fastest data parallel implementation that you can make and try to surpass the reference timings as much as you can. However, there's one trick: you are **NOT** allowed to merge upsweep and downsweep phases (they must be separated in your implementation). Also, a naïve implementation of scan might launch N CUDA threads for each iteration of the parallel loops in the pseudocode, and using conditional execution in the kernel to determine which threads actually need to do work. Such a solution will not be performant! (Consider the last outmost loop iteration of the upsweep phase, where only two threads would do work!).

Test Harness: You can pass the argument `-i random` to run the program on a random array - we will do this when grading. To aid in debugging, you can pass `-i ones`, which will run the program on an array with all elements set to 1. We encourage you to come up with alternate inputs to your program to help you evaluate it. You can also use the `-n <size>` option to change the length of the input array.

The argument `--thrust` will use the Thrust Library's[5] implementation of exclusive scan[6].

[5] https://thrust.github.io
[6] https://thrust.github.io/doc/group__prefixsums.html

Tips and Hints: Catching CUDA errors

By default, if you access an array out of bounds, allocate too much memory, or otherwise cause an error, CUDA won't normally inform you; instead it will just fail silently and return an error code. You can use the following macro (feel free to modify it) to wrap CUDA calls:

```
#define DEBUG

#ifdef DEBUG
#define cudaCheckError(ans) { cudaAssert((ans), __FILE__, __LINE__); }
inline void cudaAssert(cudaError_t code, const char *file, int line, bool abort=true) {
   if (code != cudaSuccess) {
      fprintf(stderr, "CUDA Error: %s at %s:%d\n",
        cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}
#else
#define cudaCheckError(ans) ans
#endif
```

Note that you can undefine DEBUG to disable error checking once your code is correct for improved performance. You can then wrap CUDA API calls to process their returned errors as such:

```
cudaCheckError( cudaMalloc(&a, size*sizeof(int)) );
```

Note that you can't wrap kernel launches directly. Instead, their errors will be caught on the next CUDA call you wrap:

```
kernel<<<1,1>>>(a); // suppose kernel causes an error!
cudaCheckError( cudaDeviceSynchronize() ); // error is printed on this line
```

All CUDA API functions, `cudaDeviceSynchronize`, `cudaMemcpy`, `cudaMemset`, etc. can be wrapped.

IMPORTANT: if a CUDA function error'd previously, but wasn't caught, that error will show up in the next error check, even if that wraps a different function. For example:

```
...
line 742: cudaMalloc(&a, -1); // executes, then continues
line 743: cudaCheckError(cudaMemcpy(a,b)); // prints "CUDA Error: out of memory at cudaRenderer.cu:743"
...
```

Therefore, while debugging, it's recommended that you wrap **all** CUDA API calls (at least in code that you wrote). Credit: adapted from this Stack Overflow post[7]

[7] https://stackoverflow.com/questions/14038589/what-is-the-canonical-way-to-check-for-errors-using-the-cuda-runtime-api

Part 3: Individual short report

This part of the assignment is individual. Each student should write a short report (up to 2 pages in PDF, separated from the main report, i.e., the report with the previous problems in this assignment).

In the first part of your short report, you should describe your machine, i.e., the computer that you use for development of CSPH labs (your laptop, desktop, etc). What is the OS that you are running? Describe your CPU (number of cores, execution contexts, SIMD capability, frequency, caches, TDP etc). What about your GPU and DRAM? How are your CPU and GPU different from the ones thought at the theoretical classes? Don't forget that Google is your friend when answering those questions.

The second part of this short report is aimed to be a tutorial. This tutorial should document the toolchain that you use to access `cuda` machines, as well as all the necessary steps that you undertook to set it up. Think like this: By following the steps described in your tutorial, any user should be able to recreate your environment for code development and edit/compile/run any of our assignments on `cuda` machines. Basically, what we expect from this part is to provide sufficient information for painless toolchain setup to access and use the cuda machines. You should also consider answering the following set of questions: What is the software that you use to connect to `cuda` machines? How do you transfer files? Do you edit files remotely or locally? Did you experience any problems setting up the environment? Which? How did you overcome them?