TÉCNICO LISBOA

Assignment 3A

- Template -

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2024/2025

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

NO SUBMISSIONS! This work serves as a tutorial and preparation – **NO EVALUATION**. It is intended for your practice and learning via examples and tutorials. This assignment provides a set of exercises for you to practice! You can resolve them at our lab session or at home. Surely, you can discuss your solutions and doubts with the teacher of your labs in our next lab session. Again, there is no submission for this part of the work and your solutions will not be graded!

You can consider these examples as a representative set of problems to get you prepped for the work to be developed in the class (which will be evaluated). Those assignments are not yet provided to you!

Environment Setup

For this assignment, you will need to run (the final version of) your codes on `LSD2` machines, which contain NVIDIA RTX 4070 Ti GPUs with the following characteristics:

```
Device 0: "NVIDIA GeForce RTX 4070 Ti"
  CUDA Driver Version / Runtime Version          12.3 / 12.3
  CUDA Capability Major/Minor version number:    8.9
  Total amount of global memory:                 12010 MBytes (12593004544 bytes)
  (060) Multiprocessors, (128) CUDA Cores/MP:    7680 CUDA Cores
  GPU Max Clock rate:                            2610 MHz (2.61 GHz)
  Memory Clock rate:                             10501 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 50331648 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        102400 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  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 make use of some CUDA SDK samples[3], if you think that you need more codes to practice). In addition, there is a wealth of CUDA tutorials/examples on the web (just Google!) and on the NVIDIA developer site[4].

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

Part 1: Warm up! CUDA SAXPY

To gain a bit of practice writing CUDA programs your warm-up task is to re-implement the SAXPY function from Assignment 2 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 the 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.
- You should also insert timers to 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 the completion of all CUDA work on the GPU. This call to `cudaDeviceSynchronize()` returns when all prior CUDA work on the GPU has been 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 `cudaMemcpy()` that returns data to the CPU) because `cudaMemcpy()` will not return to the calling thread until after the copy is complete.

1. What performance do you observe compared to the sequential CPU-based implementation of SAXPY (recall your results from `saxpy` from Assignment 2)?
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).

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)) );
```

You can't wrap kernel launches! 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 x.cu:743"
```

Therefore, while debugging, it's recommended that you wrap **all** CUDA API calls (at least in the code that you wrote). Credit: adapted from [this Stack Overflow post](https://stackoverflow.com/questions/14038589/what-is-the-canonical-way-to-check-for-errors-using-the-cuda-runtime-api)[5]

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

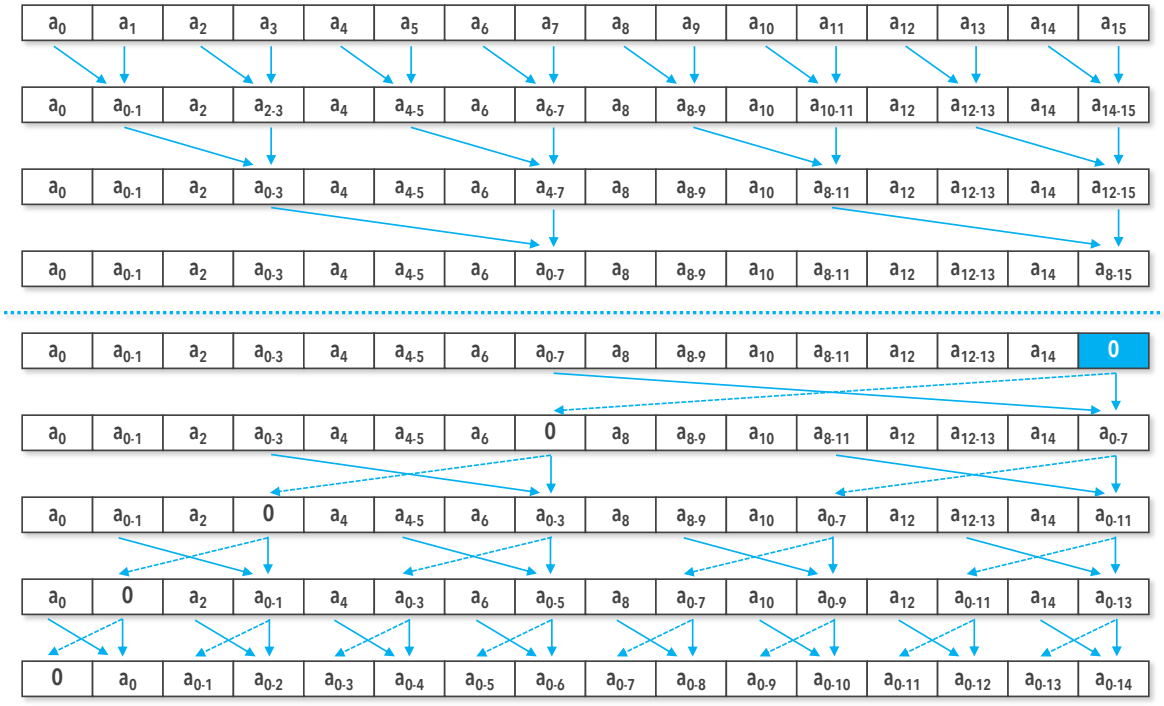
Part 2: CUDA Parallel Prefix-Sum

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 a 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 analyzing 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 figure below depicts the work-efficient exclusive scan implementation on an array of 16 elements.



The processing above the dashed horizontal line is referred to as the `upsweep` phase, while below this line we get the `downsweep` phase. As you can notice, in the upsweep phase, different pairs of elements are added in each step (with the power-of-two strides). After this phase, the last element is set to zero and the downsweep phase begins. In the downsweep phase, again, different pairs of elements are added (see full arrows), while some elements are also copied to different locations (see dashed arrows). The algorithm is a pure magic – we recommend staring at it to convince yourself that it really works!

The following "C-like" code expresses the above-referred iterative version of the 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;
        }
    }
}
```

The code in the lab kit (see `exclusive_scan` function in `scan/scan.cu`) provides an implementation of this algorithm, a version of parallel prefix sum in CUDA. The `exclusive_scan` function runs on the host and includes multiple launches of CUDA kernels. Note that for both upsweep and downsweep phases, the outer for-loop (from the code above) is present in the host code, while the `parallel_for` loop is substituted with the respective upsweep_kernel or downsweep_kernel launches.

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
```

Analyze the given code. What do you observe regarding the configuration of the kernels at each invocation? Will they run with the same number of blocks and the number of threads per block? Why? How would you describe the functionality of the upsweep_kernel and downsweep_kernel?

Imagine how a naïve implementation of scan might look like: 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. What do you think about that solution? Will it provide a good performance? (Hint: Consider the last outmost loop iteration of the upsweep phase, where only two threads would do work!).

Run the code for different input sizes, and on random input arrays, e.g., array with 1M, 10M, 20M and 40M elements! Then run the code with the argument `--thrust` that uses the Thrust Library's[6] implementation of exclusive scan[7]. What do you notice regarding the execution time?

Implementing "Find Repeats" Using Prefix Sum

Once you have concluded the analysis of the `exclusive_scan` function, implement the function `find_repeats` in `scan/scan.cu`. This will involve writing some 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 the provided `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.

To run your find repeats implementation, you shoud invoke the `./cudaScan` program with the `--test find_repeats` argument.

You can pass the argument `-i random` to run the program on a random. 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. Once it has passed the correctness test, run the code on random input arrays with 1M, 10M, 20M, and 40M elements!

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