ETH zürich

High Performance Computing for Science and Engineering II

Spring semester 2020

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

HW 6 - CUDA Reduction, SSA and asynchronous commands

Issued: May 11, 2020
Due Date: June 01, 2020, 08:00am

1-Week Milestone: Solve task 1.

The skeleton codes for this homework are located in the gitlab repository: https://gitlab.ethz.ch/hpcse20/exercise.

Task 1: Reduction

You are given an array $a_0, a_1, \ldots, a_{N-1}$ of `doubles` for some $N \in \mathbb{N}$. Your task is to perform a reduction operation

$$r = a_0 \oplus a_1 \oplus \cdots \oplus a_{N-1},$$

where $\oplus$ is some *commutative* operator[1]. In this task, we consider two operators: $x \oplus y = x + y$ and $x \oplus y = \max(x, y)$.

We start with a warp-level reduction ($N = 32$), continue with block-level reduction ($N = 1024$) and grid-level reduction with $N \leq 1024^2$, and finish with arbitrarily large values of N.

a) [10pts] Implement a device function `double sumWarp(double a)` that returns the sum of all values a within a single warp. The function must return the correct result on at least the 0th thread of each warp (other threads may return arbitrary values). Use the minimum number of required `__shfl_*_sync` operations.

Optional: Think how to do a warp-level reduce-all operation, where every thread returns the total sum, while keeping the same performance.

In total we need $\lceil \log_2(32) \rceil = 5$ shuffle operations. One possible reduction pattern is shown in Figure 1. In code, this pattern looks like the following:

```
__device__ double sumWarp(double a) {
    a += __shfl_down_sync(0xFFFFFFFF, a, 1);
    a += __shfl_down_sync(0xFFFFFFFF, a, 2);
    a += __shfl_down_sync(0xFFFFFFFF, a, 4);
    a += __shfl_down_sync(0xFFFFFFFF, a, 8);
    a += __shfl_down_sync(0xFFFFFFFF, a, 16);
    return a;
}
```

[1] If $\oplus$ were not commutative, we would not be able to utilize parallelism.

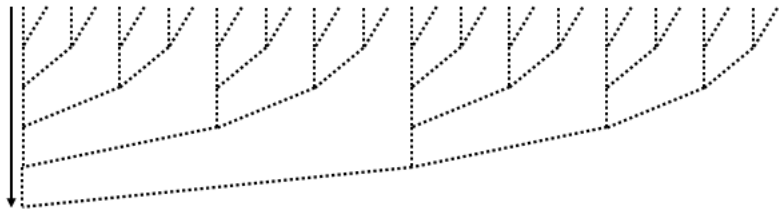

Figure 1: One potential reduction pattern for a warp of size 32.

The first argument is the bitmask of active lanes. Here we say that all 32 lanes are active. The second argument is the value we want to transfer down, and the third argument is the relative upper lane we take the value from. For example, if before the third shuffle operation the variable a is equal to 10.0 on thread 10, the return value of the third shuffle will be 10.0 on thread 6. For larger lane indices where the corresponding upper lane is out of bounds, `__shfl_up_sync` returns the value from the current thread. If we used `__shfl_xor_sync` with exactly the same parameters, the return value would compute the warp sum on all lanes, and not only the first one.

b) [10pts] Implement a device function `Pair argMaxWarp(double a)` that returns the maximum value a within a warp, together with the position of the maximum (a number between 0 and 31, inclusive). Here, `Pair` is a struct containing a `double max` and an `int idx`. The function must return the correct result on (at least) the 0th thread of each warp.

We show two different solutions, a general-purpose one that can be used for any kind of reduction, and an argmax ad-hoc solution.

The general-purpose approach is like following. Each lane prepares a pair (a, i), where a is the argument value and i the lane index. Then, instead of exchanging single floating point numbers, threads exchange whole pairs. The reduction $+$ is replaced with the argmax $\oplus$ operation:

$$(a_1, i_1) \oplus (a_2, i_2) := \begin{cases} (a_1, i_1), & \text{if } a_1 \geq a_2 \\ (a_2, i_2), & \text{otherwise} \end{cases}$$

```
/// Exchange a pair (max, idx).
__device__ Pair shfl_xor_sync(Pair value, unsigned delta){
    return Pair{
        __shfl_xor_sync(0xFFFFFFFF, value.max, delta),
        __shfl_xor_sync(0xFFFFFFFF, value.idx, delta),
    };
}

/// Reduce two pairs.
__device__ Pair argMaxOp(Pair a, Pair b) {
    return a.max >= b.max ? a : b;
}

__device__ Pair argMaxWarp(double a) {
    Pair t{a, (int)threadIdx.x & 31};
    t = argMaxOp(t, shfl_xor_sync(t, 1));
```

```
17      t = argMaxOp(t, shfl_xor_sync(t, 2));
18      t = argMaxOp(t, shfl_xor_sync(t, 4));
19      t = argMaxOp(t, shfl_xor_sync(t, 8));
20      t = argMaxOp(t, shfl_xor_sync(t, 16));
21      return t;
22  }
```

As said, this approach (custom shuffle + custom $\oplus$ operator) can be used for any element types and any reduction operator $\oplus$. However, the general-purpose approach may not be optimal in this case, since we shuffle the index, an information that is implicitly available.

The next snippet shows another solution, specific to the argmax operation:

```
1   __device__ Pair argMaxWarp(double a) {
2       double t = a;  // max
3       t = max(t, __shfl_xor_sync(0xFFFFFFFF, t, 1));
4       t = max(t, __shfl_xor_sync(0xFFFFFFFF, t, 2));
5       t = max(t, __shfl_xor_sync(0xFFFFFFFF, t, 4));
6       t = max(t, __shfl_xor_sync(0xFFFFFFFF, t, 8));
7       t = max(t, __shfl_xor_sync(0xFFFFFFFF, t, 16));
8       unsigned ballot = __ballot_sync(0xFFFFFFFF, a == t);
9       int idx = __ffs(ballot) - 1;
10      return {t, idx};
11  }
```

Here, the threads only exchange the value a, while the index is determined by ballot and ffs instructions. After the threads computed the maximum value within the warp, each checks if they have the maximum (`a == t`) and exchange this information with other threads using a ballot instruction. Then each thread computes the location of the first 1 in the ballot bitmask, which indicates the location of the (first) maximum.

c) [15pts] Pick one of the following:

- Implement a device function `double sumBlock(double a)` that returns the sum of all values a within a single block of size exactly $N = 1024 = 32^2$.
- Implement a device function `Pair argMaxBlock(double a)` that returns the maximum and the location of the maximum of all values a within a single block of size exactly $N = 1024 = 32^2$. The location is now a number between 0 and 1023.

It is sufficient for the 0th thread of a block to return the correct result. *Hint:* Utilize the functions from previous subquestions. Think what mechanism you can use to exchange information between threads (or warps) of a *single* block.

Please see the code for details.

Regarding the argmax problem, note that the reduction from the general-purpose solution from the previous subquestion can be used to do a block-level reduction. Namely, the value i in the pair (a, i) does not necessarily have to be a number between 0 and 31, inclusive. Nevertheless, the solution code uses a different approach, where an argmax reduction of pairs (a, i) is not done by shuffling both a and i, but only a, and then the final value i is broadcasted (see `argMaxWarp(Pair a)`).

Note: Without a proper benchmark, it cannot be claimed which approach is the fastest. Due to a potential instruction-level parallelism (ILP), the question of performance is not only about the number of (shuffle) operations, but also of the length of the dependency chain of different shuffle instructions. In other words, it could be the case that in the general-purpose approach, the `max` and `idx` values are shuffled in parallel due to ILP. This could potentially make it faster than the ballot + ffs implementation.

d) [20pts] For more than 1024 elements we would like to parallelize the reduction using multiple blocks[2]. Utilize the single-block reduction from the previous subquestion to implement multi-block reduction for $N \leq 1024^2$.

The reduction can be done in two steps. The first is a kernel that performs block-level reduction and stores the result in a temporary array, one item per block. After that, the second kernel launch reduces the temporary array into one single final result.

See the code for details.

e) [5pts] Explain at least one way to perform sum reduction for a hypothetical case of $N = 2 \times 10^9$, without copying any data back to the host. Note that the maximum number of blocks in one kernel launch is 65536.

The procedure from the previous subquestion can be easily extended to support up to $N \leq 64 \times 1024^2$, by launching one more kernel. This method is, however, limited by the total number of blocks in the first kernel. Instead, we can modify the kernel to operate on more than one element of the array using so-called *grid-strided loops*[3]. The following snippet shows an (untested) possible implementation of the first reduction kernel:

```
__global__ double argMaxReduceKernel1(
        const double *aDev, Pair *bDev, int N) {
    // Should be -infinity, a neutral value for argmax.
    Pair myValue{-1e100, 0};
    for (int idx = blockIdx.x * blockDim.x + threadIdx.x;
         idx < n;
         idx += blockDim.x * gridDim.x) {
        myValue = argMaxOp(myValue, Pair{aDev[idx], idx});
    }
    Pair blockResult = argMaxBlock(myValue);
    if (threadIdx.x == 0)
        bDev[blockIdx.x] = blockResult;
}
```

Note that the second kernel is still needed to perform the final reduction (it could also be done at the end of the first kernel, for `blockIdx.x == 0` only).

[2] We could always implement reduction for an arbitrary N even with a single thread, but then we would not have any parallelism.

[3] For more info, see https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

Task 2: SSA - Trajectory Binning

In this task we are revisiting the Stochastic Simulation Algorithm (SSA) of Homework 4 Task 1. Please review the exercise description and the solution code from HW4 Task 1.

Here, you are asked to implement the GPU kernels for the **Trajectory binning**. A trajectory from SSA shows one possible realization of the time evolution of the system. Trajectory binning is a simple way to generate average trajectories over a number of sample trajectories. Since we do not have the same time steps for all samples (i.e. array of triples (Sa, Sb, t)), we need to average over time and over the samples. We assume that the simulation is run for times $t \in [0, t_{end}]$. We can then compute averaged quantities $\overline{S_{ik}}$ for realizations of S_i in bin k associated to the time interval $[k\delta t, (k+1)\delta t)$, whereas δt is the width of the bins and $k = 0 \dots \lceil t_{end}/\delta t \rceil - 1$.

a) [10pts] In the CUDA version of the SSA algorithm (see `dimerizationKernel`) each thread simulates one trajectory for `numIters` time steps. Some trajectories may reach t_{end} and finish, some may not. Therefore we need to check if all trajectories finished, and if not, we call the `dimerizationKernel` again, until all trajectories reached t_{end}.

Implement the CUDA kernel `reduceIsDoneKernel`. This function stores the amount of samples/trajectories which are finished within a block and stores the count in the array `blocksDoneCount`.

The `reduceIsDoneKernel` amounts to implementing a block-level reduction of booleans to integers. A valid solution is to copy-paste the solution from the Task 1 of this homework and change `double` to `int`. We can, however, make a small optimization. Instead of doing manual shuffle-based reduction of booleans, for the warp-level reduction of booleans to integers (the first half of block-level reduction), we can use a `__ballot_sync` to exchange all 32 booleans to all 32 lanes of a warp, and then `__popc` to compute their sum.

b) [20pts] Implement the binning functionality described above. For this, sum the quantities of Sa and Sb associated to a bin, e.g. $\sum_{i:t(i)\in[k\delta t,(k+1)\delta t]} Sa(i)$, and store the accumulated value in a buffer. Try to parallelize the binning mechanism as much as possible using multiple threads and blocks. Also store the number of summation terms in a buffer such that you can average the calculated values later. You may want to make use of the reduction kernels implemented in Task 1 of this Homework.

First we need to explain the storage of the variables `x` and `t`. These variables, although being represented in code as 1D arrays are in fact 3D and 2D matrices, respectively. The array `x` is a row-major representation of a matrix $x_{\xi,i,s}$, where ξ is the species (0 or 1), i the trajectory index (iteration) and s the sample index. Similarly, the array `t` is a row-major representation of a matrix $t_{i,s}$.

The goal of this subtask is to compute, for each bin k and for each species ξ, the sum of $x_{\xi,i,s}$ for all i and s where $t_{i,s}$ is within the bounds of the bin $[k\delta t, (k+1)\delta t)$. In mathematical terms, we are interested in a quantity $b_{\xi,k}$ defined as

$$b_{\xi,k} \coloneqq \sum_{i,s} [k\delta t \le t_{i,s} < (k+1)\delta t] x_{\xi,i,s}, \tag{1}$$

where $[condition]$ is 1 if the condition is met, 0 otherwise. The solution we have goes as follows. We use 2D indexing on blocks. The first dimension (`blockIdx.x`), as usually, is used to determine the sample index s (together with `threadIdx.x`). The second dimension

denotes the bin index k. Thus, each block handles up to 1024 samples, and multiple blocks together construct the result of a single bin k.

To efficiently compute the sum 1, we need to quickly determine for a given sample s which iterations i are within the bin bounds. Since $t_{i,s}$ is sorted with respect to i ($t_{i,s}$ is the current time at step i of a single trajectory s), we utilize binary search. One binary search is needed to find i_{start}, one to find i_{end}, the smallest and the largest i within the bin bounds, respectively. After that, we simply sum up all the values $x_{0,i,s}$ and $x_{1,i,s}$ for $i \in [i_{\text{start}}, i_{\text{end}}]$.

Comment: In principle, the summation can be avoided. If instead of storing x we store a prefix sum

$$X_{\xi,i,s} = \sum_{i'=0}^{i} x_{\xi,i',s} = X_{\xi,i-1,s} + x_{\xi,i,s},$$

we can compute the desired sum simply as

$$\sum_{i=i_{\text{start}}}^{i_{\text{end}}} x_{\xi,i,s} = X_{\xi,i_{\text{end}},s} - X_{\xi,i_{\text{start}}-1,s}.$$

In any case, once the summation is known, it is atomically added to bin accumulators $b_{\xi,k}$. Finally, apart from storing only $b_{\xi,k}$ for $\xi \in \{0,1\}$, we also store $b_{2,k}$ as the total number of points within the bin ($x_{2,i,s} := 1$), which will be needed to perform the final averaging.

The presented solution is probably one of many possible ones. If you had a different correct solution with similar or better performance, consider it valid.

c) [10pts] Implement a kernel in order to average the bins calculated in the previous subtask. Make sure that the implementation is correct (compare with the results from Homework 4). Finally, deallocate all extra buffers you allocated.

In this step, we need to normalize the bins and simply perform the following operation:

$$b^{\text{final}}_{\xi,k} := \frac{b_{\xi,k}}{b_{2,k}}.$$

Please see the `README` file for compilation instructions. For your implementation, try to follow the `TODO` comments in the code. The files that need to be modified are: `ssa.cu` (allocate and deallocate memory, add function calls of your kernels), `kernels.cu` (for the actual implementation of the kernels) and `kernels.h` (extend with your function prototypes).

Task 3: Communication–computation overlap

Imagine we are operating on so large buffers that they do not even fit in the GPU memory[4]. For example, on Piz Daint, nodes have 64GB of host memory, while GPUs have only 16GB, so if we require 50GB of data, the data has to reside on the host.

If we need to apply a computationally expensive kernel on these buffers, using a GPU may be beneficial. However, for the data to fit on the GPU, it has to transferred and processed in chunks. Your task is to implement the function `runAsync`, which executes the given kernel on an array of size N, in chunks of size C, using S CUDA streams. By using more than one stream, we hope to parallelize communication and computation.

The skeleton code includes two dummy kernels `fastKernel` and `slowKernel`, which emulate a memory-bound and a compute-bound kernel, respectively. The kernels take an input buffer a, an output buffer b and the chunk size.

a) [10pts] Create the streams, and allocate chunk memory for each of the streams. Each stream requires two chunk buffers: one for a, one for b. Do not forget to destroy the streams and free the memory at the end.

See the solution code.

b) [20pts] Split the array `aHost` logically into chunks of size C. For each chunk, asynchronously copy it from the host to the device, (asynchronously) launch the kernel, and asynchronously copy the result back to the host. In the case N is not divisible by C, truncate the last chunk.

The skeleton code will check if the whole array was processed. Do not forget to synchronize the streams before the check.

Report the speedup compared to the synchronous execution `runSync` which uploads and processes everything in one go (as a simplification, we work with buffers that do fit in the GPU's memory).

See the solution code. The tricky part is handling the chunk size when N is not divisible by C. One way to approach this is to cut the last chunk:

```
// The end condition is equivalent to
//    chunk < (N + chunkSize - 1) / chunkSize
for (int chunk = 0; chunk * chunkSize < N; ++chunk) {
  // Where the chunk starts.
  int offset = chunk * chunkSize;

  // Chunk size is min(max size, remaining elements).
  int size = std::min(chunkSize, N - offset);

  // Assign chunks to streams in a cyclic fashion.
  int stream = chunk * numStreams;

  ...
}
```

The benchmarks give following results:

[4]Such scenarios are plausible in computational fluid dynamics.

- Fast kernel (memory bound) with 1 stream or synchornized: 0.128s.
- Fast kernel (memory bound) with 4 and 8 streams: 0.082s.
- Slow kernel (compute bound) with 1 stream or synchornized: 0.665s.
- Slow kernel (compute bound) with 4 and 8 streams: 0.542s.

Numbers depend slightly on the chunk size. Smaller chunks allow for a better overlap. However, too small chunks would lead to too large management overhead. Results for 4 and 8 streams are virtually identical.

Streams improved the performance of the memory-bound kernel by about 35%. We do not reach 50% performance boost because simultaneous upload and download do slow each other down partially. In the compute-bound kernel, the performance benefit is about 19%. This number depends directly on the computational load of the kernel.

c) [10pts] Use `nvprof -o output.%h.%p.nvvp ./overlap` to profile your code, and `nvvp` to visualize the result. You will need to pass the `-Y` flag to `ssh` (both for `ela` and `daint`) to use `nvvp`. Take a screenshot of the visualization for the slow and the fast kernel. How many different operations can the GPU execute in parallel? How does that depend on their type (copying vs kernel)?

Note: To make the visualization cleaner, instead of invoking `runBenchmarks()` from `main()`, invoke `profile()`.

See Figure 2 and Figure 3 for the screenshots of `nvcc` for the fast and the slow kernel, respectively. In the former, the bottleneck is the communication, while in the latter the bottleneck is the execution. The former plot also shows the difference in within-GPU bandwidth and host–device bandwidth. In the latter plot, we can notice how all streams uploaded their first chunks before the first stream finished its kernel execution. Note that your timings and stream IDs might be different, depending on how you allocated the streams and how many different cases you were running.

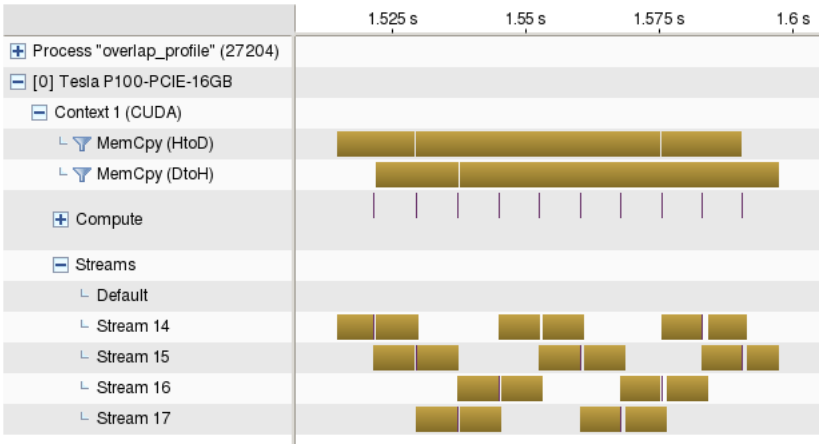

Figure 2: Processing the fast (memory-bound) kernel with 4 streams.

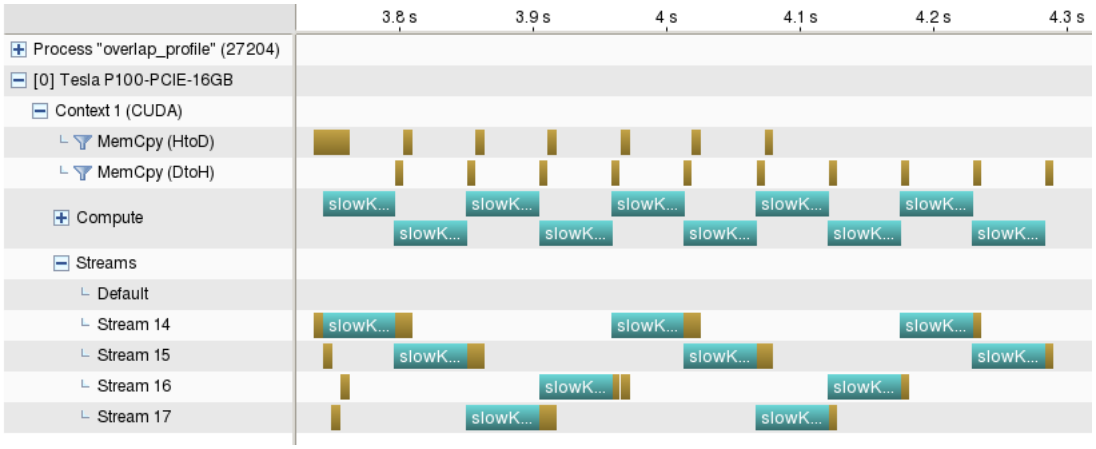

Figure 3: Processing the slow (compute-bound) kernel with 4 streams.

Guidelines for reports submissions:

- Report all your answers in a pdf. Archive your pdf and source code (e.g.: .tar, .zip) and submit it via Moodle until May 25, 2020, 08:00am.