# High Performance Computing for Science and Engineering II.

Exercise Set 6

Florian Mahlknecht

2020-05-31

## 1 Q1: Reduction

### 1.1 Warp Level

To propagate the reduction results to all threads, the following implementation has been chosen:

```
double sum = a;
sum += __shfl_xor_sync(0xFFFFFFFF, sum, 1);
sum += __shfl_xor_sync(0xFFFFFFFF, sum, 2);
sum += __shfl_xor_sync(0xFFFFFFFF, sum, 4);
sum += __shfl_xor_sync(0xFFFFFFFF, sum, 8);
sum += __shfl_xor_sync(0xFFFFFFFF, sum, 16);
return sum;
```

Listing 1: Warp level reduction

Similarly the case of argmax has been handeled.

### 1.2 Block level reduction

At block level shared memory is exploited:

```
double result = sumWarp(a);
__shared__ double sdata[32];

if (threadIdx.x % 32 == 0)
    sdata[threadIdx.x / 32] = result;
__syncthreads();

if (threadIdx.x < 32) {
    result = sumWarp(sdata[threadIdx.x]);
}

return result;
```

Listing 2: Block level reduction

### 1.3 1024 blocks

To further reduce across blocks, without overwriting the source array, additional global memory is needed. Therefore a buffer has been allocated and a simple function which saves the previously obtained block level results to this buffer, according to the `blockIdx.x`. A subsequent use of this function completes the task:

```
int numBlocks = (N+1024-1)/1024;

if (numBlocks > 1) {
    double* bufferDev;
    CUDA_CHECK(cudaMalloc(&bufferDev, numBlocks * sizeof(double)));
    CUDA_LAUNCH(sumReduce, numBlocks, 1024, aDev, bufferDev, N);
    CUDA_LAUNCH(sumReduce, 1, 1024, bufferDev, bDev, numBlocks);
    CUDA_CHECK(cudaFree(bufferDev));
} else {
    CUDA_LAUNCH(sumReduce, 1, 1024, aDev, bDev, N);
}
```

Listing 3: Across blocks

```
__global__ void sumReduce(const double *aDev, double *bDev, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    double a = idx < N ? aDev[idx] : 0.0;
    double sum = sumBlock(a);

    if (threadIdx.x == 0)
        bDev[blockIdx.x] = sum;  // atomicAdd(bDev, sum);
}
```

Listing 4: Sum reduce

For compute capability > 6.0, another option, wihtout the need of additional buffer, is given as a comment in listing 4. Given the blocking nature however, the performance of the atomicAdd is excpeted to drop compared to the buffered approach.

### 1.4 Larger arrays

The code given in listing 3 works for arrays up to $1024^2$ elements. At this limit, the results from the block reduction cannot be elaborated anymore within 1024 threads. Therefore the code needs to be extended in a recursive fashion. This means that the results from the first block reduction are fed into the same function again (with the buffer as input and output), until ≤ 1024 elements are left and therefore a single block reduction can be used to produce one single result.

The described approach would work until the maximum number of blocks in a kernel launch is reached for the first kernel execution, i.e. for 64 Mi elements. Afterwords multiple subsequent kernels launches need to be scheduled with shifted input and output arrays. This part is implemented in the benchmarking code of section 3.

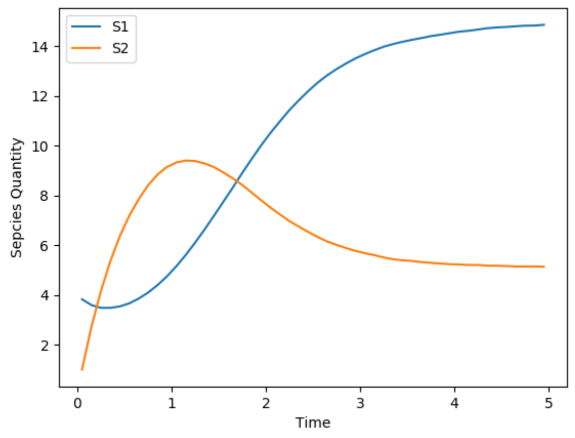
In this way larger arrays up to the GPU memory bound limit, accounting also for the linearly growing buffer demand, could be handled.

## 2 Q2: SSA - Trajectory Binning

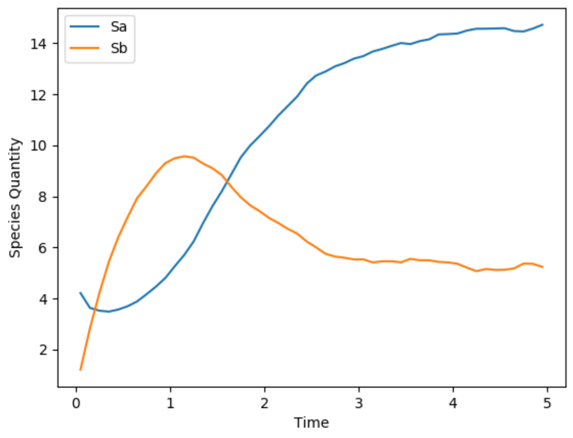
A test run on Piz Daint of the SSA algorithm yields:

```
class34@nid02356:~/hpcse-ii/hw6/p2/build> ./ssa
Testing blocksDoneKernel...
    Passed.
SSA_GPU  numItersPerPass: 1000  numSamples: 200000  required memory: ~1722.8MB
===== DIMERIZATION =====
Execution Loop 0. Remaining samples: 200000/200000
Execution Loop 1. Remaining samples: 200000/200000
Execution Loop 2. Remaining samples: 0/200000
Average number of time steps per sample: 0.617655
```

With respect to the CPU version, implemented in homework 4, this GPU version executes the simulation much faster. Figure 1 shows the ouput of both versions, which remains similar as expected.



(a) CPU



(b) GPU

Figure 1: SSA Output Plot

## 3 Q3: Communication pipelining

The benchmarking run gives the following results:

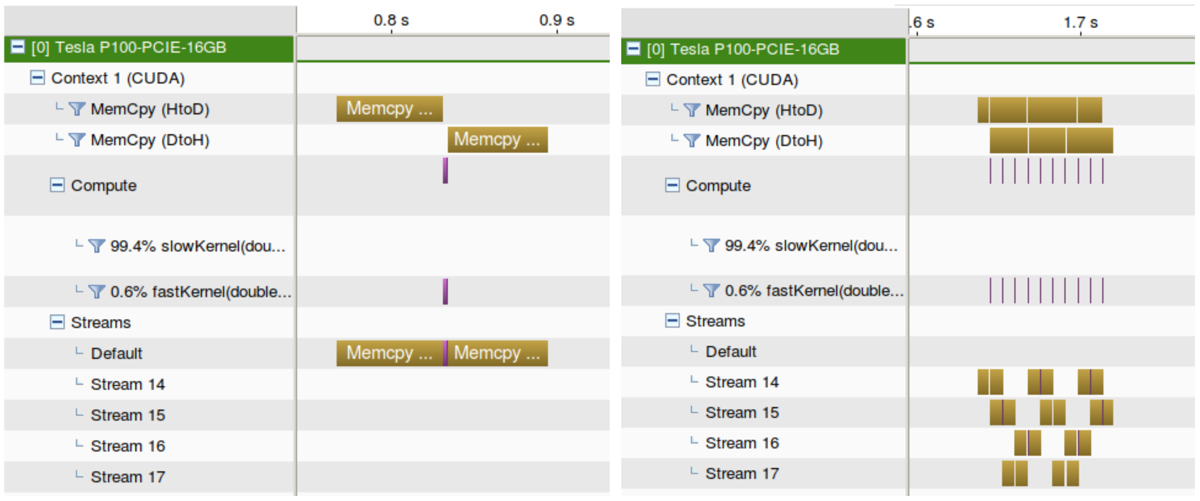
```
class34@nid02358:~/hpcse-ii/hw6/p3> ./overlap
sync  fastK  N=  1000000  up=0.000673s k=0.000017s down=0.000650s tot=0.001339s
sync  fastK  N=100000000  up=0.064111s k=0.000020s down=0.063604s tot=0.127734s
async fastK  N=100000000  chunkSize=100000000    numStreams=1    time=0.127761s
async fastK  N=100000000  chunkSize= 10000000    numStreams=4    time=0.082381s
async fastK  N=100000000  chunkSize= 10000000    numStreams=8    time=0.082281s
async fastK  N=100000000  chunkSize=  1000000    numStreams=4    time=0.083859s
async fastK  N=100000000  chunkSize=  1000000    numStreams=8    time=0.081317s

sync  slowK  N=  1000000  up=0.000672s k=0.000017s down=0.006218s tot=0.006906s
sync  slowK  N=100000000  up=0.064113s k=0.000020s down=0.601602s tot=0.665734s
async slowK  N=100000000  chunkSize=100000000    numStreams=1    time=0.665768s
async slowK  N=100000000  chunkSize= 10000000    numStreams=4    time=0.553544s
async slowK  N=100000000  chunkSize= 10000000    numStreams=8    time=0.553515s
async slowK  N=100000000  chunkSize=  1000000    numStreams=4    time=0.542378s
async slowK  N=100000000  chunkSize=  1000000    numStreams=8    time=0.542342s
```

Generally, the asynchronous approach with performs better for a stream number > 1 with the fast kernel, whereas the speedup with the slow kernel is not as significant. This has to be expected, since the kernels occupy all the available SMs, and therefore cannot be run in parallel.

Furthermore, one can observe that doubling the streams in the asynchronous run from 4 to 8 does not give further benefits. The timings remain essentially stable at around 80 ms for the asynchronous calls of the fast kernel and 550 ms for the slow kernel. Lowering the chunk size in the slow kernel gives an improvement of around 10 ms, whereas with the fast kernel the impact is negligible.

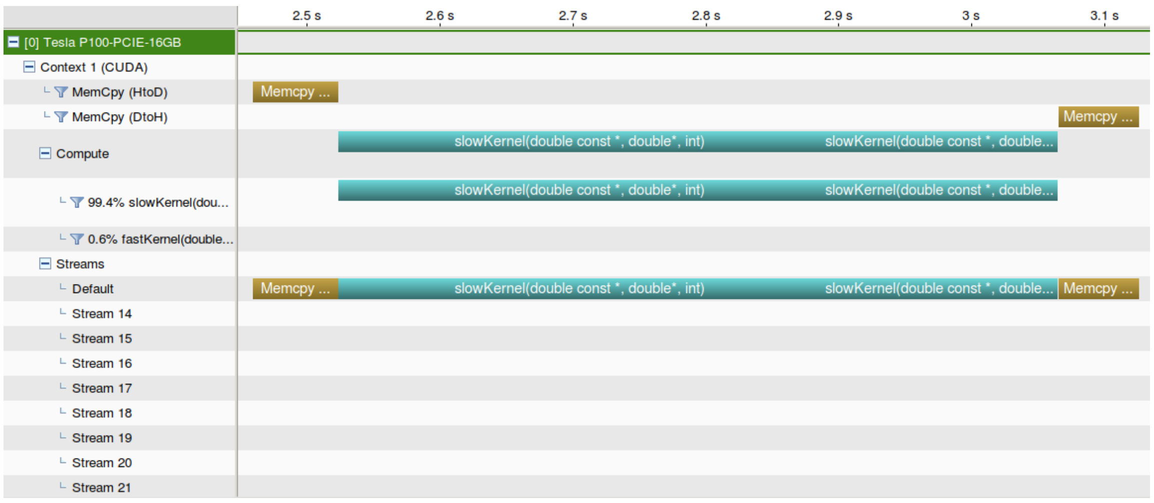
Figures 2 and 3a compare the profiling results of the pipelining approach for the fast and the slow kernel respectively.



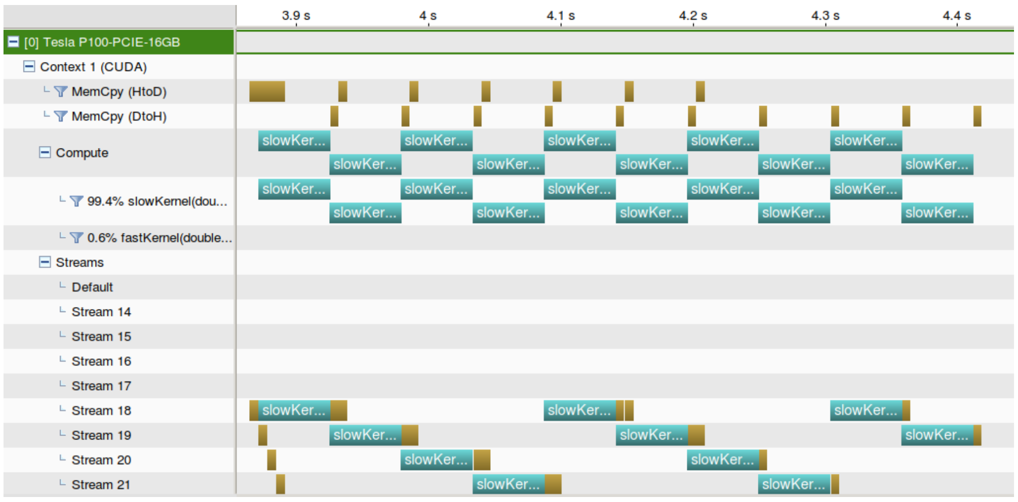
(a) Sequential (b) Pipelining

Figure 2: Fast Kernel Profiling

As discussed before already, the GPU can perform only one kernel execution at a given time step. This can be appreciated especially when looking at figs. 2b and 3b. The memory operations on the other hand, can be executed simultaneously, one H2D, and one D2H, as shown in fig. 2b.



(a) Sequential



(b) Pipelining

Figure 3: Slow Kernel Profiling

This memory operation parallelism can not be exploited in the same fashion anymore when the kernel becomes significantly slower, as fig. 3b illustrates. Instead of two overlapping memory operation, just one compute and one memory operation is executed at a time. The results take too long to compute, and before the write back could be executed, all buffers have already been uploaded to the device.

The suspect is confirmed with the final result presented in table 1. As discussed before, the improvement for the fast kernel is more significant. However, in both cases pipelining provides an advantage.

| Type | Start [s] | End [s] | Time [ms] | Change [%] |
|---|---|---|---|---|
| Slow Kernel Sequential | 2.46 | 3.13 | 666.46 | |
| Slow Kernel Pipelining | 3.86 | 4.42 | 553.84 | -16.90% |
| Fast Kernel Sequential | 0.77 | 0.89 | 127.82 | |
| Fast Kernel Pipelining | 1.64 | 1.72 | 82.62 | -35.36% |

Table 1: Pipelining Comparison