Computational Science on Many-Core Architectures Exercise 4

Leon Schwarzäugl

13. November 2023

The code for all tasks can be found at: `https://github.com/Swarsel/CSE_TUWIEN/tree/main/WS2023/Many-Core%20Architectures/e4`

1 Dot product with warp shuffles

1.1 a)

The code for this subtask is basically the same as the example code, just with 4 sums :)

```
__global__ void task1_a(int N, double *x, double *result)
{
    __shared__ double shared_mem_sum[256];
    __shared__ double shared_mem_absum[256];
    __shared__ double shared_mem_sqsum[256];
    __shared__ double shared_mem_nn0[256];

    double sum = 0;
    double absum = 0;
    double sqsum = 0;
    double nn0 = 0;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
        sum += x[i];
        absum += abs(x[i]);
        sqsum += pow(x[i],2);
        if (x[i] == 0) nn0 += 1;
    }

    shared_mem_sum[threadIdx.x] = sum;
    shared_mem_absum[threadIdx.x] = absum;
    shared_mem_sqsum[threadIdx.x] = sqsum;
    shared_mem_nn0[threadIdx.x] = nn0;
    for (int k = blockDim.x / 2; k > 0; k /= 2) {
        __syncthreads();
        if (threadIdx.x < k) {
            shared_mem_sum[threadIdx.x] += shared_mem_sum[threadIdx.x + k];
            shared_mem_absum[threadIdx.x] += shared_mem_absum[threadIdx.x + k];
            shared_mem_sqsum[threadIdx.x] += shared_mem_sqsum[threadIdx.x + k];
            shared_mem_nn0[threadIdx.x] += shared_mem_nn0[threadIdx.x + k];
        }
    }

    if (threadIdx.x == 0) {
        atomicAdd(&result[0], shared_mem_sum[0]);
        atomicAdd(&result[1], shared_mem_absum[0]);
        atomicAdd(&result[2], shared_mem_sqsum[0]);
        atomicAdd(&result[3], shared_mem_nn0[0]);
    }
}
```

1.2 b)

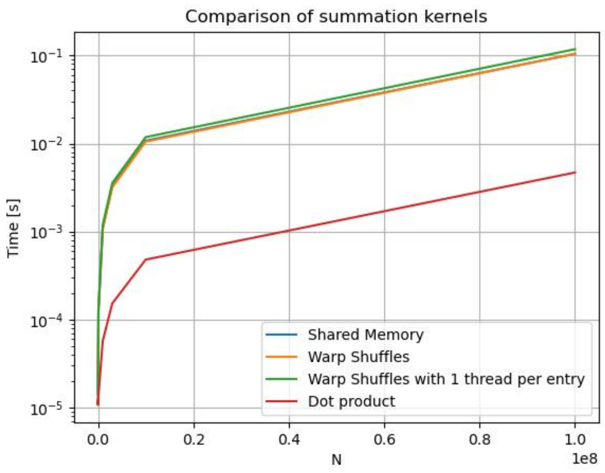
Here we are now using warp shuffles. Not too much changes in the code though. The relevant changes are (apart from no longer needing shared memory), for one, in the reduction part, which now calls the __shfl_xor_sync function, as well as the write in the end, which now needs to be performed by one thread for each warp group.

```
     [...]
     for (int i=warpSize/2; i>0; i=i/2) {
        sum += __shfl_xor_sync(0xffffffff, sum, i);
        absum += __shfl_xor_sync(0xffffffff, absum, i);
        sqsum += __shfl_xor_sync(0xffffffff, sqsum, i);
        nn0 += __shfl_xor_sync(0xffffffff, nn0, i);
    }

    if (threadIdx.x % warpSize == 0) {
    [...]
```

1.3 c)

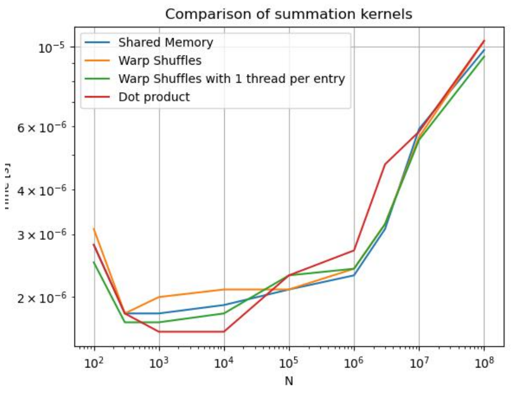
The kernel code did not change for this subtask. The only difference lies in the kernel call which should now yield for each entry one thread, which leads to the kernel call task1_a<<<(N[j]+255)/256, 256>>>(N[j], cuda_x, cuda_results);

1.4 Data interpretation



We compared these versions with a reference dot product implementation. The reference is a lot faster, which does not come as a surprise seeing as we are computing less values. The other three kernels offer similar execution times, with only the one thread per entry version performing a bit worse than the others for bigger N, while notably performing faster than the others for small $N < 100000$, and even barely beating the dot product reference in the case $N = 100$.

Small side question: I wondered if it also would be an acceptable benchmark here to omit the final call to cudaDeviceSynchronize (even though my gut feeling says no); even without it, the output data seems to be correct (although it might just manage to catch up during checking, which would skew the results), in which case it seems fine to me to stop the timer here. For the "official benchmark", I still used the synchronization however. The data looks a bit different without it:



2 Multiple dot products

2.1 a) & b)

The requested kernel was implemented and can be found in the file "2a.cu". It pretty much just does what was asked, computing 8 dot products concurrently and using warp shuffles as the reduction operation. The kernel is called $K/8$ times, each time computing 8 dot products. I also made sure that the result is correct (using the CPU reference to check).

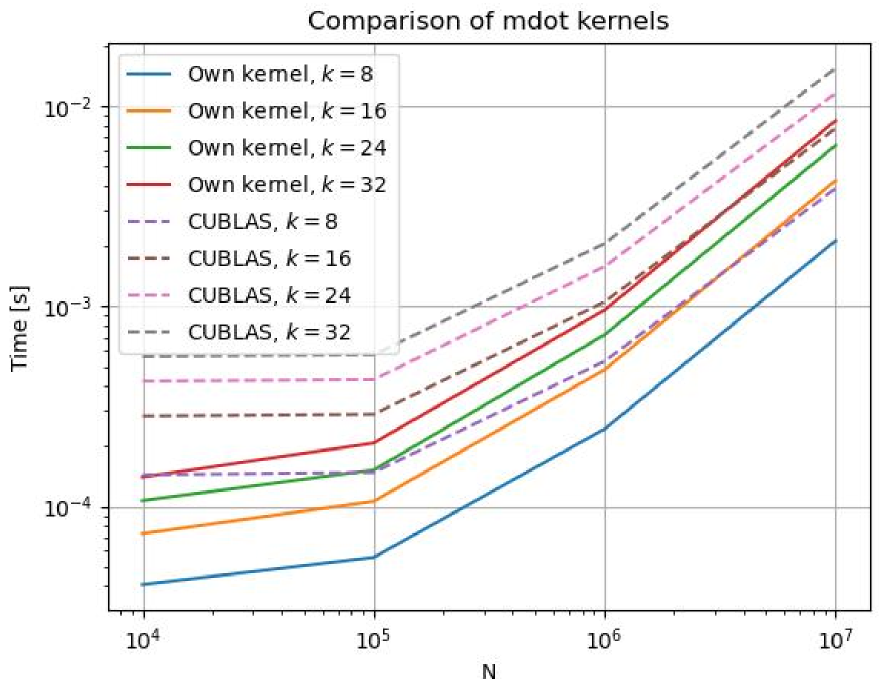
```
__global__ void mdot(double *x, double **y, double *results, int i, int N) {
    double alpha1{0}, alpha2{0}, alpha3{0}, alpha4{0}, alpha5{0}, alpha6{0}, alpha7{0}, alpha8{0};

    for(int j = blockIdx.x * blockDim.x + threadIdx.x; j < N; j += blockDim.x*gridDim.x) {
        double val_w = x[j];
        alpha1 += val_w * y[i][j];
        alpha2 += val_w * y[i+1][j];
        alpha3 += val_w * y[i+2][j];
        alpha4 += val_w * y[i+3][j];
        alpha5 += val_w * y[i+4][j];
        alpha6 += val_w * y[i+5][j];
        alpha7 += val_w * y[i+6][j];
        alpha8 += val_w * y[i+7][j];
    }

    for (int j=warpSize/2; j>0; j=j/2) {
        alpha1 += __shfl_xor_sync(0xffffffff, alpha1, j);
        alpha2 += __shfl_xor_sync(0xffffffff, alpha2, j);
        alpha3 += __shfl_xor_sync(0xffffffff, alpha3, j);
        alpha4 += __shfl_xor_sync(0xffffffff, alpha4, j);
        alpha5 += __shfl_xor_sync(0xffffffff, alpha5, j);
        alpha6 += __shfl_xor_sync(0xffffffff, alpha6, j);
        alpha7 += __shfl_xor_sync(0xffffffff, alpha7, j);
        alpha8 += __shfl_xor_sync(0xffffffff, alpha8, j);
    }

    if (threadIdx.x % warpSize == 0) {
        atomicAdd(&results[i], alpha1);
        atomicAdd(&results[i+1], alpha2);
        atomicAdd(&results[i+2], alpha3);
        atomicAdd(&results[i+3], alpha4);
        atomicAdd(&results[i+4], alpha5);
        atomicAdd(&results[i+5], alpha6);
        atomicAdd(&results[i+6], alpha7);
        atomicAdd(&results[i+7], alpha8);
    }
}
```

The biggest problem in this task arose from me trying to use y as an array of pointers on the device. In the start I tried to implement this just normally, accessing cuda_y as I thought should simply work. However, after many hours of studying my code and not understanding why that did not work and failing to find corresponding info in the documentation, I found the answer in one of the NVIDIA sample codes (`https://github.com/NVIDIA/cuda-samples/blob/master/Samples/4_CUDA_Libraries/batchCUBLAS/batchCUBLAS.cpp` from line 342 onwards); in order to access such an array in the way of y[i][j], we need to create an array of pointers on the host which point to device arrays. This part I was still able to understand from the given example code. However, in order to actually use the array in this more intuitive way, we now need to copy this host array to device array memory and access the array of pointers using that pointer. In practice, that meant adding the lines:

```
double **cuda_y_pointer; cudaMalloc((void **)&cuda_y_pointer, sizeof(*cuda_y) * K);
cudaMemcpy(cuda_y_pointer, cuda_y, sizeof(*cuda_y) * K, cudaMemcpyHostToDevice);
```

In hindsight, it would probably have been easier to simply pass a normal array to the device and then taking steps of size N to access the next vector instead.

2.2 c)



The kernels were compared for

$$(K \times N), \quad K \in \{8, 16, 24, 32\}, \quad N \in \{10000, 100000, 1000000, 10000000\}.$$

We can see that the concurrent implementation beats the corresponding CUBLAS version in every case, although it must be noted that the run time increase with increasing N is greater for my implementation, so there should be a $(K \times N)$ at which the CUBLAS implementation becomes faster again.

2.3 d)

The easiest way to achieve that would simply be to call another kernel after the main call that computes the dot products for the 7 or less vectors that will still be left unaccounted for after its call. We can of course write a kernel for each case of left over vectors $k_{\text{left}} \in [1, 7]$, then calling the respective one to keep up our concurrent approach, but especially for larger K the extra cost of computing these last dot products using just a "normal" dot product kernel (as in task 1) should not be all too big.