Computational Science on Many-Core Architectures
Exercise 2

Leon Schwarzäugl

October 2022

1 Basic CUDA

1.1 a)

The time needed to allocate and free CUDA arrays was measured for different N as seen in the table below. Unless otherwise mentioned, all measurements were always taken 10 times, with the mean shown. Interestingly, for small N we do not see a strictly monotone rise - this means that for small N the fluctualtions have an impact big enough to make the measurement quite unreliable.

Table 1: Allocation and Free time for CUDA arrays in seconds

N	cudaMalloc	cudaFree
100	0.000117	0.000087
300	0.000113	0.000083
1000	0.000116	0.000085
10000	0.000118	0.000084
100000	0.000121	0.000084
1000000	0.000710	0.000093
3000000	0.002130	0.000118

1.2 b)

The time needed to allocate an CUDA array directly within the kernel (using cudaMemset) and by copying from a host array was measured for different N as seen in the table below.

Table 2: Initialization time for different ways of initialization in seconds

N	cudaMemset	host array
100	0.000122	0.000001
300	0.000120	0.000001
1000	0.000130	0.000003
10000	0.000123	0.000029
100000	0.000127	0.000289
1000000	0.000718	0.003147
3000000	0.002140	0.009247

1.3 c)

The following kernel was implemented:

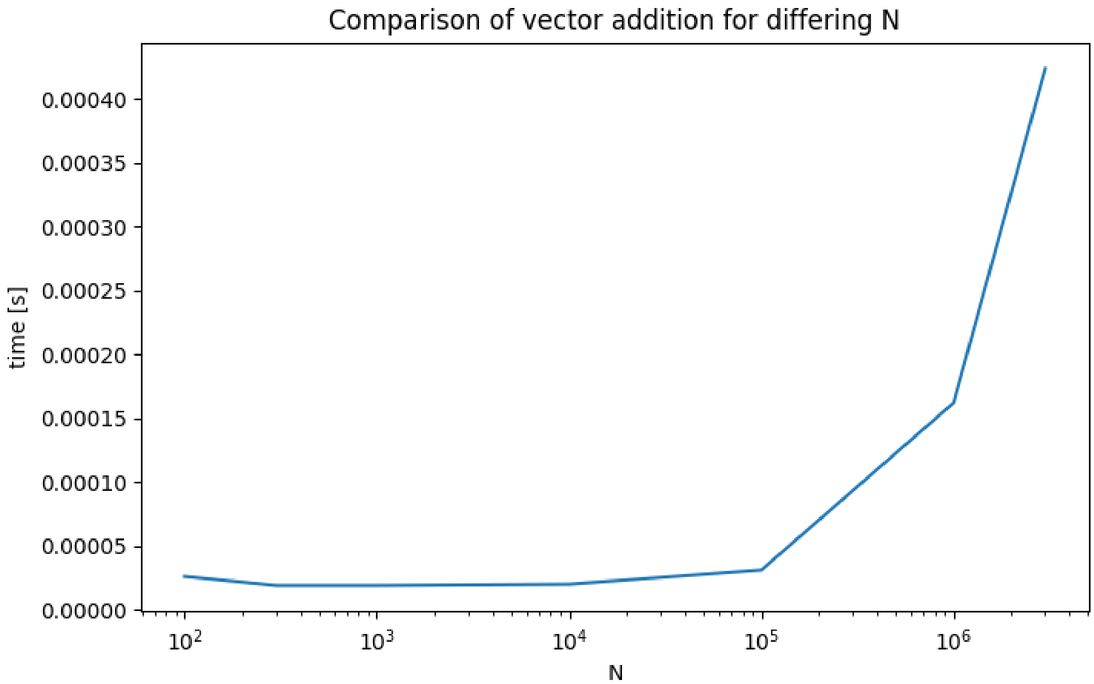
```
    __global__ void add(int n, double *x, double *y, double *z)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) z[i] = x[i] + y[i];
}
```

This implementation delivers correct results given that correct kernel parameters are chosen (more on that below).

1.4 d)

Table 3: Execution times for vector addition, differing N; in seconds

N	kernel ex. time
100	0.000026
300	0.000019
1000	0.000019
10000	0.000020
100000	0.000031
1000000	0.000162
3000000	0.000424



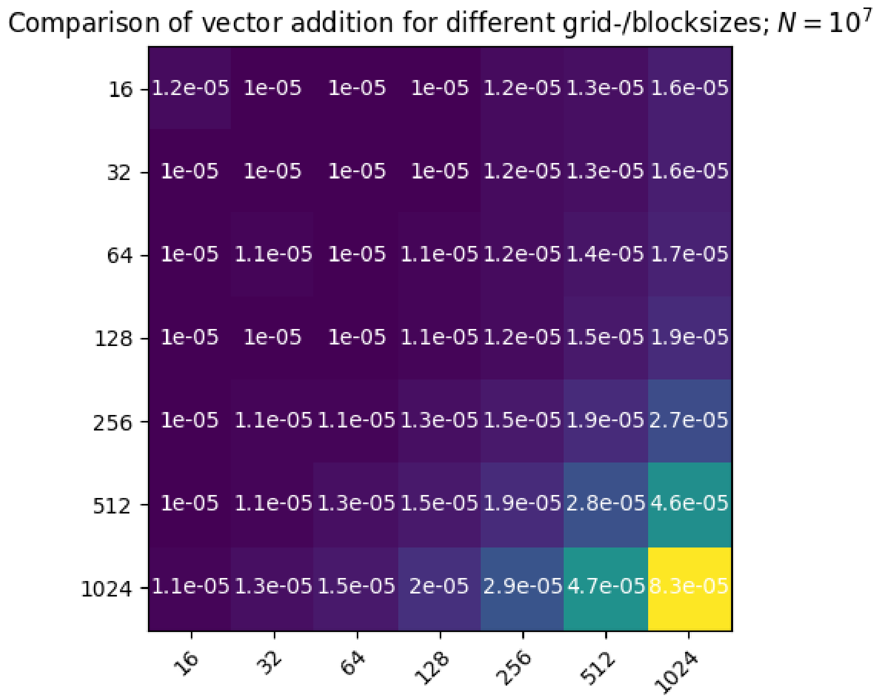
As we can see, for small N fluctuations seem to dominate the measurement, as it makes little (although not none after thinking of the last exercise) sense why a smaller N would take more time to compute in this case. For bigger N we see the increase that is to be expected.

1.5 e)

This time, the execution time was measured over 5 iterations only (because the 30 seconds limit was reached otherwise).

Table 4: Execution time for vector addition, differing grid/block sizes; in seconds

Grid v, block >	16	32	64	128	256	512	1024
16	0.000012	0.000010	0.000010	0.000010	0.000012	0.000013	0.000016
32	0.000010	0.000010	0.000010	0.000010	0.000012	0.000013	0.000016
64	0.000010	0.000011	0.000010	0.000011	0.000012	0.000014	0.000017
128	0.000010	0.000010	0.000010	0.000011	0.000012	0.000015	0.000019
256	0.000010	0.000011	0.000011	0.000013	0.000015	0.000019	0.000027
512	0.000010	0.000011	0.000013	0.000015	0.000019	0.000028	0.000046
1024	0.000011	0.000013	0.000015	0.000020	0.000029	0.000047	0.000083



From this, it seems most configurations fare quite well, except (1024,1024),(512,1024), and (1024,512). In general, time seems to increase the higher the sums of block and grid size get. Of course it is very important to note that with the aforementioned vector addition kernel none of the results are correct, as only the first gridSize $*$ blockSize elements will be computed, and $1024^2 = 1048576 < 10^7$.

2 Dot Product

2.1 a)

The following kernels were implemented:

```
__device__ void sum(double * shared_m, double * result) {

  for (int stride = blockDim.x/2; stride>0; stride/=2) {
    __syncthreads();
    if (threadIdx.x < stride) shared_m[threadIdx.x] += shared_m[threadIdx.x + stride];
  }
    if (threadIdx.x == 0) result[blockIdx.x] = shared_m[0];
  }

__global__ void dot_product(int N, double *x, double *y, double * result) {

  __shared__ double shared_m[1024];
  double thread_sum = 0;
  int total_threads =  blockDim.x * gridDim.x;;
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i = thread_id; i<N; i += total_threads) thread_sum += x[i] * y[i];
  shared_m[threadIdx.x] = thread_sum;
  sum(shared_m, result);
}
```

2.2 b)

This was realized using the kernel:

```
__global__ void prod(int n, double *x, double *y, double *z){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) z[i] = x[i] * y[i];
}
```

and following summation with

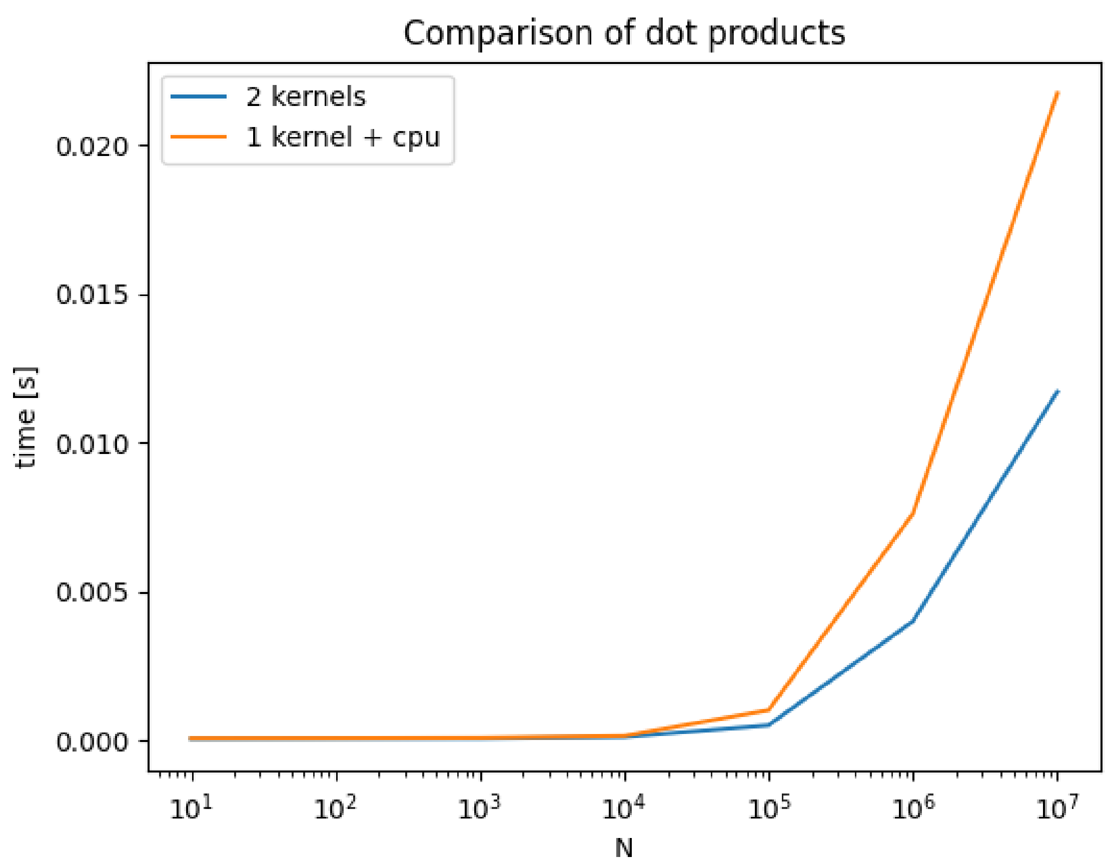
```
cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
for(int i = 0; i < N; i++) sum += z[i];
```

2.3 Comparison

For the following comparison, the 2 kernel version was launched using (1,1024) while the 1 kernel + CPU version was launched using (4096,256) - these were respectively the values that I had found delivered the fastest results while being correct.

Table 5: Comparison of different dot products in seconds

N	10	100	1000	10000	100000	1000000	3000000
2 kernels	0.000070	0.000077	0.000080	0.000131	0.000525	0.004015	0.011726
1 kernel + CPU	0.000086	0.000090	0.000100	0.000170	0.001030	0.007627	0.021745



As we can see, the difference in time is marginal for small N, while the 2 kernel version is quite a lot faster for higher N.