Computational Science on Many-Core Architectures
Exercise 2

Leon Schwarzäugl

17. 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 fluctuations 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 and by copying from a host array was measured. The kernel was started using grid- and blocksizes (256,256) (going forward I am only going to write down the numbers).

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

N	within kernel	host array
100	0.000022	0.000001
300	0.000021	0.000001
1000	0.000022	0.000003
10000	0.000032	0.000029
100000	0.000105	0.000289
1000000	0.001351	0.003147
3000000	0.004109	0.009247

1.3 c)

The following kernel was implemented:

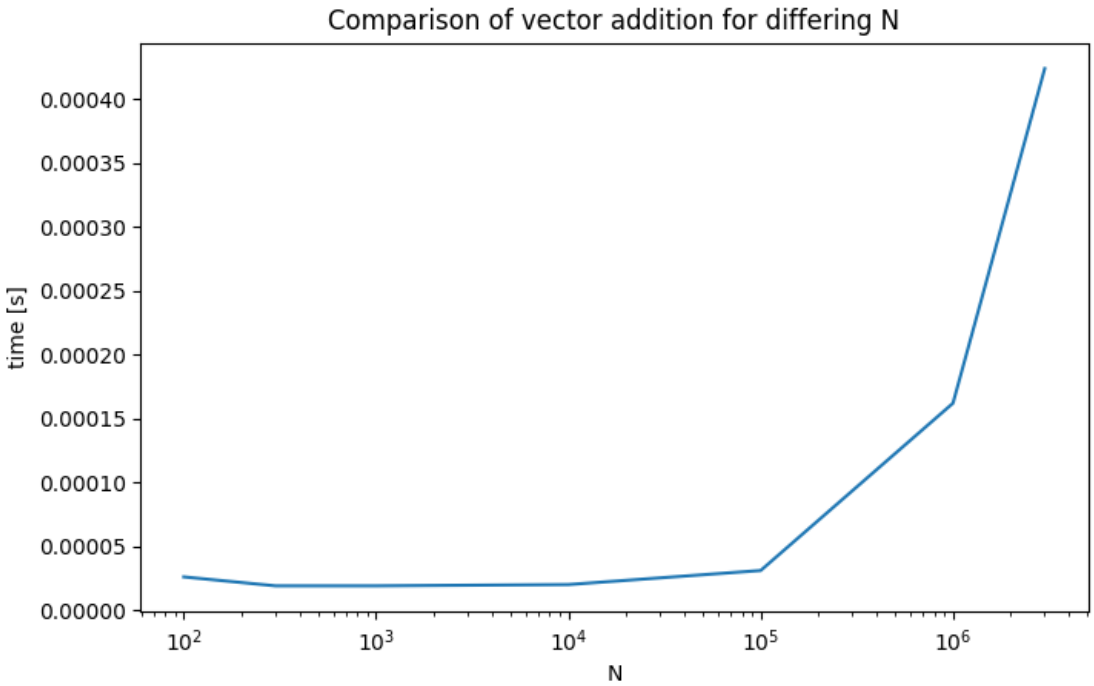
```
__global__ void add(int n, double *x, double *y, double *z)
{
  int total_threads =  blockDim.x * gridDim.x;
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i = thread_id; i<n; i += total_threads) z[i] = x[i] + y[i];
}
```

1.4 d)

The kernel was started on (256,256).

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	0.000021
100000	0.000031
1000000	0.000189
3000000	0.000491



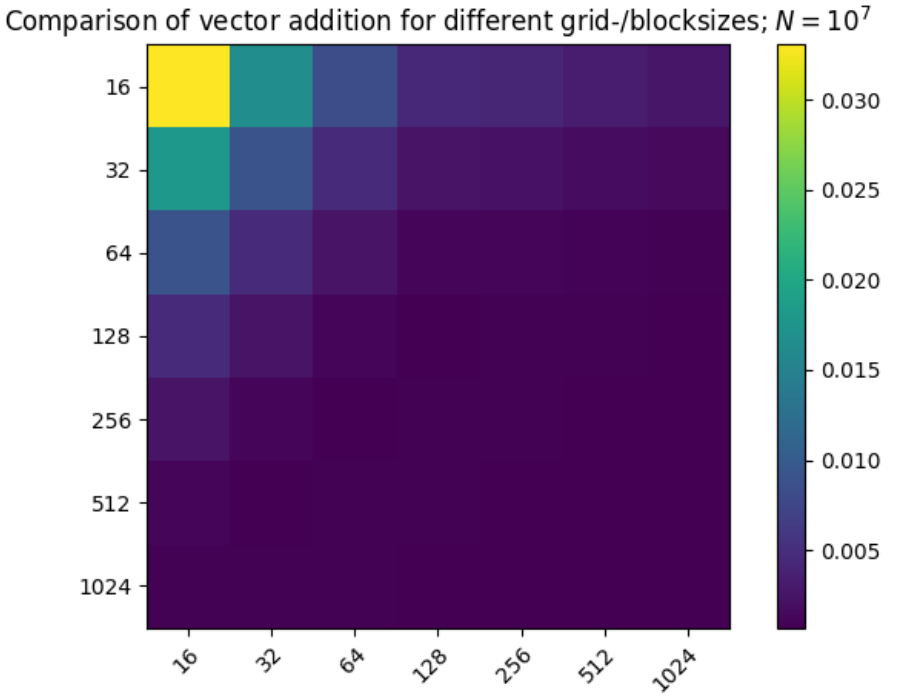
As we can see, for small N fluctuations seem to dominate the measurement, as it makes little (although not none after thinking of the last exercise) sense why a smaller N would take more time to compute in this case. For bigger N we see the increase that is to be expected.

1.5 e)

The kernel was again launched using (256,256)

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

Grid v, block >	16	32	64	128	256	512	1024
16	0.033051	0.016575	0.008385	0.004257	0.004152	0.003140	0.002652
32	0.017971	0.009011	0.004612	0.002345	0.002289	0.001741	0.001468
64	0.009010	0.004610	0.002343	0.001186	0.001326	0.001004	0.000932
128	0.004616	0.002347	0.001186	0.000813	0.000927	0.000854	0.000718
256	0.002351	0.001188	0.000819	0.000935	0.000851	0.000717	0.000746
512	0.001205	0.000821	0.000937	0.000851	0.000718	0.000745	0.000701
1024	0.000828	0.000940	0.000863	0.000721	0.000744	0.000696	0.000724



From this, it seems most configurations fare quite well, except (16,16),(16,32), and (32,16). In general, time increases the lower the sums of block and grid size get. This is because with these low sizes, we are forcing a lot of iterations and we are not using our resources well.

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 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) z[i] = x[i] * y[i];
}
```

and following summation with

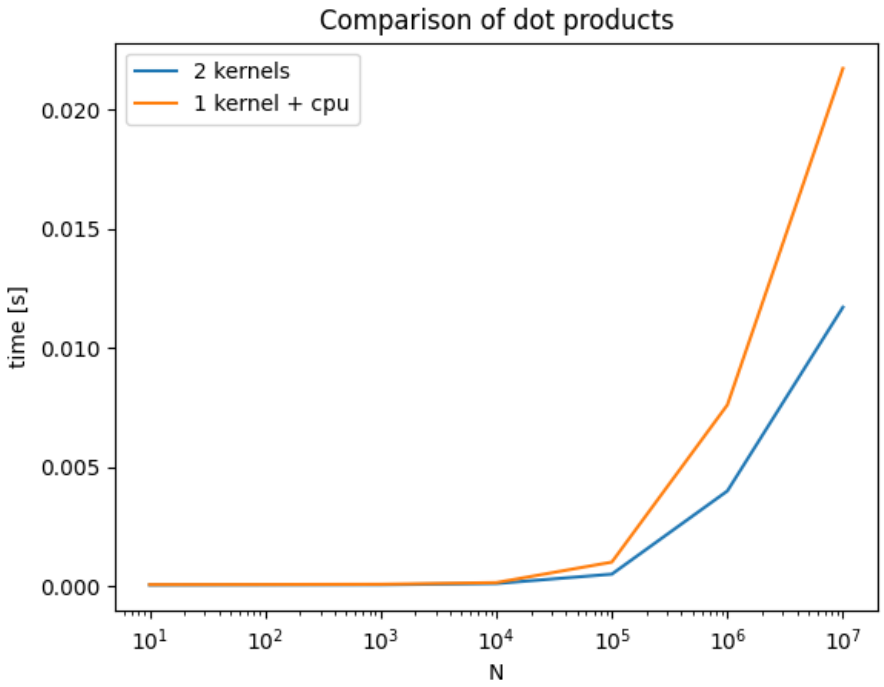
```
cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
for(int i = 0; i < N; i++) sum += z[i];
```

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 (256,256) - 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. This does not seem like a surprise, because even when we need to do multiple iterations on the two kernel version, we can still benefit from parallelization, while the purely iterative approach falls behind.