Computational Science on Many-Core Architectures
Exercise 2

Leon Schwarzäugl

20. October 2023

Disclaimer: As I did this lecture up to this exercise last year as well, some of this document is here presented with the same layout, as the conclusions stayed mostly the same. However, I tried to optimize some of the code and wrote a new implementation for the dot product for practice reasons :)

The code for all tasks can be found at: `https://github.com/Swarsel/CSE_TUWIEN/tree/main/WS2023/Many-Core%20Architectures/e2`

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 a bit unreliable as otherwise we are only left with an $O(1)$ overhead.

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

N	cudaMalloc	cudaFree
100	0.000053	0.000031
300	0.000052	0.000028
1000	0.000052	0.000029
10000	0.000052	0.000029
100000	0.000052	0.000029
1000000	0.000181	0.000033
3000000	0.000540	0.000046

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 block sizes (256,256) (going forward I am only going to write down the numbers). Again I iterated over the suggested values from the exercise description. For the individual assignment the values $N = 1000000$ and $N = 3000000$ failed to complete within the allotted 30 seconds; these values are thus omitted.

The effective bandwidth in GB/s was calculated using

$$B = \frac{2 * 8N}{10^9 t},$$

with execution time t, since we are calculating two vector of size $8 * N$ bytes each. The following table presents this information in the form *time[bandwidth]*.

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

N	within kernel	host array whole	host array single
100	0.000012[0.133334]	0.000016[0.1]	0.000437[0.003662]
300	0.000006[0.8]	0.000009[0.533334]	0.001198[0.004006]
1000	0.000005[3.2]	0.000012[1.333334]	0.003862[0.004142]
10000	0.000005[32]	0.000038[4.210526]	0.038074[0.004202]
100000	0.000029[55.172414]	0.000398[4.0201]	0.421316[0.003798]
1000000	0.000239[66.945606]	0.004505[3.55161]	-
3000000	0.000626[76.677316]	0.015030[3.193612]	-

1.3 c)

The following kernel was implemented:

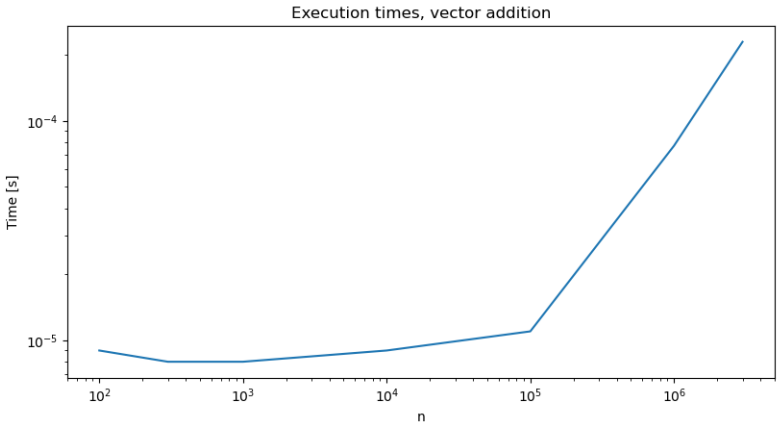
```
__global__ void add(int n, double *x, double *y, double *z)
{
  unsigned int total_threads =  blockDim.x * gridDim.x;
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i = thread_id; i<n; i += total_threads) z[i] = x[i] + y[i];
}
```

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	0.000009
300	0.000008
1000	0.000008
10000	0.000009
100000	0.000011
1000000	0.000077
3000000	0.000229



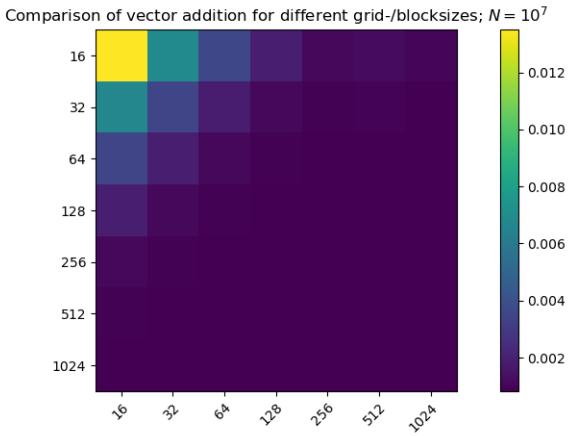
As we can see, for small N the execution times are mostly constant due to an $O(1)$ overhead as well as fluctuations. For bigger N we see the increase that is to be expected as the execution time scales roughly linearly to N.

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	0.013499	0.006912	0.003591	0.001871	0.001121	0.001217	0.001019
32	0.006695	0.003466	0.001862	0.001101	0.000878	0.000956	0.000864
64	0.003476	0.001871	0.001103	0.000878	0.000832	0.000860	0.000835
128	0.001874	0.001110	0.000880	0.000833	0.000835	0.000833	0.000830
256	0.001108	0.000879	0.000833	0.000836	0.000832	0.000831	0.000828
512	0.000878	0.000833	0.000834	0.000832	0.000830	0.000827	0.000827
1024	0.000834	0.000863	0.000835	0.000830	0.000827	0.000825	0.000825



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:

```
__global__ void sum(double * result) {
for (int stride = blockDim.x/2; stride>0; stride/=2) {
__syncthreads();
if (threadIdx.x < stride)
result[threadIdx.x] += result[threadIdx.x + stride];
}}
```

```
__global__ void prod(int n, double *x, double *y, double * z) {
  __shared__ double shared_m[256];
  double thread_prod = 0;
  unsigned int total_threads = blockDim.x * gridDim.x;
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (unsigned int i = thread_id; i<n; i += total_threads) {
    thread_prod += x[i] * y[i];
  }
  shared_m[threadIdx.x] = thread_prod;
  for (unsigned 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) {
  z[blockIdx.x] = shared_m[0];
  }
}
```

2.2 b)

This was realized using the same first stage kernel as before:

```
__global__ void prod(int n, double *x, double *y, double *z)
{
  __shared__ double shared_m[256];
  double thread_prod = 0;
  unsigned int total_threads = blockDim.x * gridDim.x;
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  for (unsigned int i = thread_id; i<n; i += total_threads) {
    thread_prod += x[i] * y[i];
  }
  shared_m[threadIdx.x] = thread_prod;
  for (unsigned 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) {
  z[blockIdx.x] = shared_m[0];
  }
}
```

and following summation with

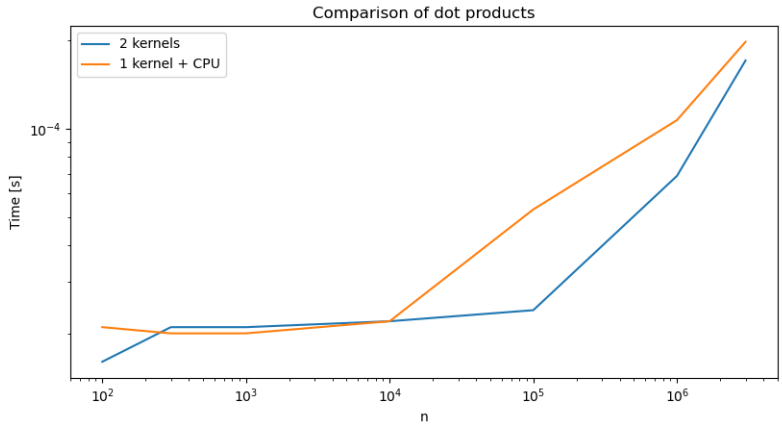
```
cudaMemcpy(z, d_z, 256*sizeof(double), cudaMemcpyDeviceToHost);
for(int i = 0; i < N; i++) sum += z[i];
```

2.3 Comparison

For the following comparison, the first kernel stage was launched using (256,256) while the second stage for the 2-kernel version was launched using (1,256).

Table 5: Comparison of different dot products in seconds

N	10	100	1000	10000	100000	1000000	3000000
2 kernels	0.000016	0.000021	0.000021	0.000021	0.000022	0.000024	0.000171
1 kernel + CPU	0.000021	0.000020	0.000020	0.000022	0.000056	0.000107	0.000198



As we can see, the difference in time is marginal for small N, while the 2 kernel version is 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 iterative approach falls behind. We could even still increase the threads per block which would enhance this effect further.