360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 2

Christian Gollmann, 01435044

Last update: November 1, 2020

Contents

1 Basic CUDA a)

Here I have to apologize for the small figure annotations, I only realized how small they are after already having included the figure and I didn't have the figure code anymore.

It can be seen that up to a certain point, the size of allocated memory has no impact on performance, after that the impact scales with the needed memory. This probably is due to the GPU's memory system. There might be an equivalence to the CPU's caches.

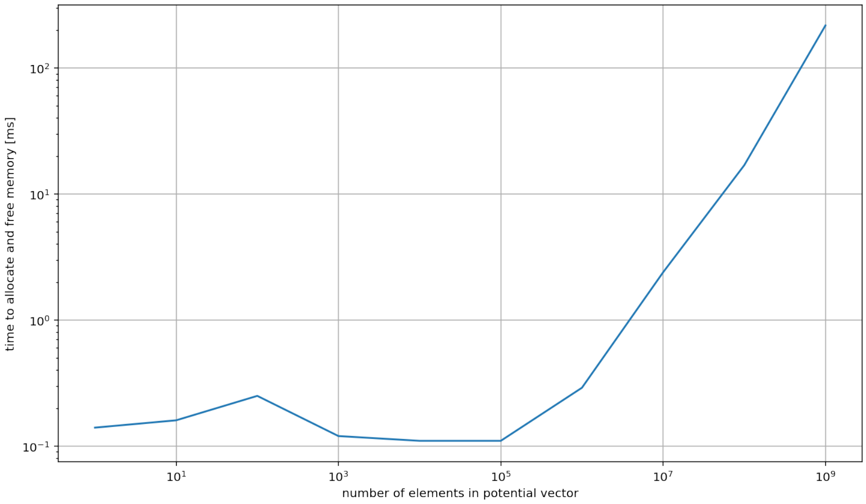

Figure 1: time to allocate and free memory for different sizes N

Listing 1: Code used to generate the output above

```
#include <stdio.h>
#include "timer.hpp"

int main(void)
{
    int N = 1000;
    double *d_x;
    Timer timer;

    cudaDeviceSynchronize();
    timer.reset();

    for(int i = 0; i < 100; i++) {
        cudaMalloc(&d_x, N*sizeof(double));
        cudaFree(d_x);
        cudaDeviceSynchronize();
    }

    cudaDeviceSynchronize();
    double time_elapsed = timer.get();

    printf("Allocation and Freeing took %g seconds", time_elapsed/100);

    return EXIT_SUCCESS;
}
```

2 Basic CUDA b)

Here, clearly copying each entry individually is the slowest as I expected it to be.

Initializing vectors for 1M elements		
Method	time to initialize [ms]	bandwidth [MB/s]
Initialize in kernel	13.8	579.7
Copy the data	3	2666.7
Copy each individual entry	7742	1

Listing 2: Initialize directly within a dedicated CUDA kernel

```
#include <stdio.h>
#include "timer.hpp"

__global__
void init(int N)
{
    double *x, *y;

    x = new double[N];
    y = new double[N];

    for (int i = 0; i < N; i++)
    {
        x[i] = i;
        y[i] = N-1-i;
    }
}

int main(void)
{
    int M = 1;
    int N = 1000000;
    Timer timer;

    cudaDeviceSynchronize();
    timer.reset();

    init<<<(M+255)/256, 256>>>(N);

    cudaDeviceSynchronize();
    double time_elapsed = timer.get();

    printf("Initializing in kernel took %g seconds", time_elapsed);

    return EXIT_SUCCESS;
}
```

Listing 3: Copy the data

```
#include <stdio.h>
#include "timer.hpp"

int main(void)
{
    int N = 1000000;
```

```

    double*x, *y, *d_x, *d_y;
    Timer timer;

    x = new double[N];
    y = new double[N];

    for (int i = 0; i < N; i++)
    {
        x[i] = i;
        y[i] = N-1-i;
    }

    cudaDeviceSynchronize();
    timer.reset();

    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);

    cudaDeviceSynchronize();
    double time_elapsed = timer.get();

    printf("Initializing outside took %g seconds", time_elapsed);

    cudaFree(d_x);
    cudaFree(d_y);
    delete x;
    delete y;

    return EXIT_SUCCESS;
}
```

Listing 4: Copy each individual entry

```
#include <stdio.h>
#include "timer.hpp"

int main(void)
{
    int N = 1000000;

    double*x, *y, *d_x, *d_y;
    Timer timer;

    x = new double[N];
    y = new double[N];

    for (int i = 0; i < N; i++)
    {
        x[i] = i;
        y[i] = N-1-i;
    }

    cudaDeviceSynchronize();
    timer.reset();
```

```

    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    for (int i = 0; i < N; i++)
    {
        cudaMemcpy(d_x+i, x+i, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_y+i, y+i, sizeof(double), cudaMemcpyHostToDevice);
    }

    cudaDeviceSynchronize();
    double time_elapsed = timer.get();

    printf("Initialising and copying by piece took %g seconds", time_elapsed);

    cudaFree(d_x);
    cudaFree(d_y);
    delete x;
    delete y;

    return EXIT_SUCCESS;
}
```

3 Basic CUDA c)

Listing 5: CUDA kernel that sums two vectors

```
#include <stdio.h>
#include "timer.hpp"
#include <iostream>

__global__ void sumVectors(double *x, double *y, double *z, int N)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    for(size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
        z[i] = x[i] + y[i];

}

int main(void)
{
    int N = 100;

    double*x, *y, *z, *d_x, *d_y, *d_z;
    Timer timer;

    x = new double[N];
    y = new double[N];
    z = new double[N];


    for (int i = 0; i < N; i++)
    {
        x[i] = i;
        y[i] = N-1-i;
        z[i] = 0;
    }


    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_z, N*sizeof(double));
    cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);

    cudaDeviceSynchronize();
    timer.reset();

    sumVectors<<<(N+255)/256, 256>>>(d_x, d_y, d_z, N);

    cudaDeviceSynchronize();
    double time_elapsed = timer.get();

    cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);

    printf("Addition took %g seconds", time_elapsed);

    std::cout << std::endl << "z[0] = " << z[0] << std::endl;
```

```
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    delete x;
    delete y;
    delete z;

    return EXIT_SUCCESS;
}
```

4 Basic CUDA d)

For small values of N (<e5) it doesn't make much difference but after hitting a certain threshold, execution time seems to increase exponentially. Consider though that I always call the kernel with different values of N but still those are interesting results. There's much resemblance to the plot in Figure 1.

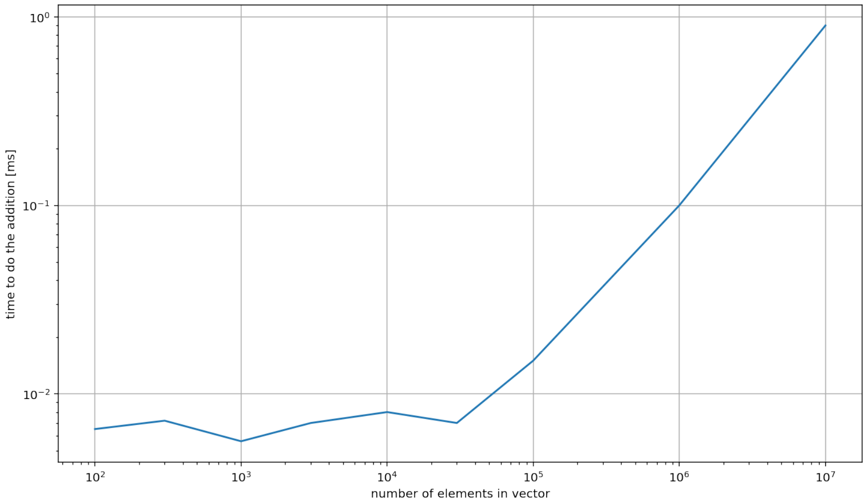

Figure 2: addition time for different values of N

Listing 6: addition benchmark

```
#include <stdio.h>
#include "timer.hpp"
#include <iostream>

__global__ void sumVectors(double *x, double *y, double *z, int N)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    for(size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
        z[i] = x[i] + y[i];

}

int main(void)
{
    int N = 100;

    double*x, *y, *z, *d_x, *d_y, *d_z;
    Timer timer;

    x = new double[N];
    y = new double[N];
    z = new double[N];


    for (int i = 0; i < N; i++)
    {
```

```
        x[i] = i;
        y[i] = N-1-i;
        z[i] = 0;
    }


    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_z, N*sizeof(double));
    cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);

    cudaDeviceSynchronize();
    timer.reset();
    for( int i = 0; i < 100; i++) {
        sumVectors<<<(N+255)/256, 256>>>(d_x, d_y, d_z, N);
        cudaDeviceSynchronize();
    }

    cudaDeviceSynchronize();
    double time_elapsed = timer.get();

    cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);

    printf("Addition took %g seconds", time_elapsed/100);

    std::cout << std::endl << "z[0] = " << z[0] << std::endl;

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    delete x;
    delete y;
    delete z;

    return EXIT_SUCCESS;
}
```

5 Basic CUDA e)

Now I called the vector addition for a vector with e7 elements with varying grid and block sizes. It seems that small values (16, 32, 64) lead to a not so good performance.

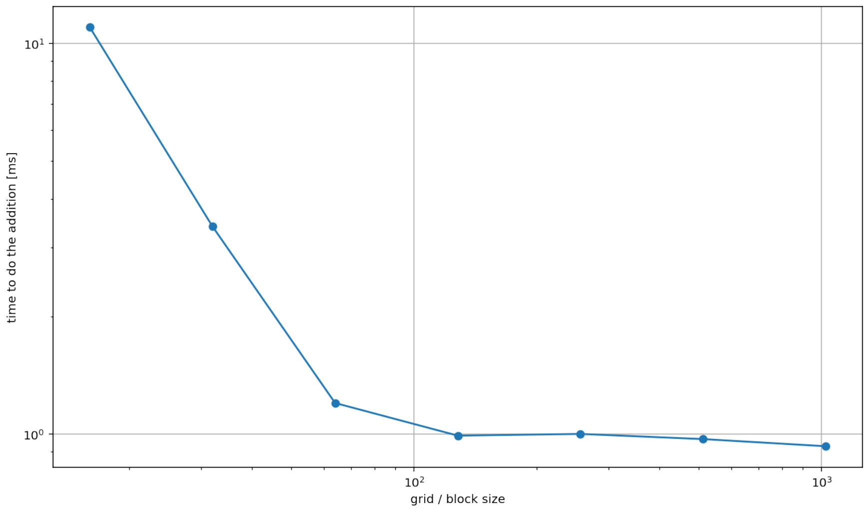

Figure 3: addition time for vector with e7 elements for different grid / block values

Listing 7: kernel call with different values x

```
sumVectors<<<x, x>>>(d_x, d_y, d_z, N);
```

6 Dot Product a)

For the Dot Product Exercises I got some inspiration from https://bitbucket.org/jsandham/ algorithms_in_cuda/src/master/dot_product/ and added / changed code as needed. I am aware that my implementations are not the prettiest and might also be error prone. But they get the job done for those specific examples.

Listing 8: Dot Product with two GPU stages

```
#include <stdio.h>
#include <iostream>
#include "timer.hpp"
#include <random>


__global__ void dot_product_first_part(double *x, double *y,
double *temporary, unsigned int n)
{
    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    __shared__ double cache[256];

    double temp = 0.0;
    while(index < n){
        temp += x[index]*y[index];

        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    for(int i = blockDim.x/2; i>0; i/=2)
    {
        __syncthreads();
        if(threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

    if(threadIdx.x == 0){
        temporary[blockIdx.x] = cache[0];
    }
}

__global__ void dot_product_second_part(double* temporary, double* dot)
{
    for(int i = blockDim.x/2; i>0; i/=2)
    {
        __syncthreads();
        if(threadIdx.x < i)
            temporary[threadIdx.x] += temporary[threadIdx.x + i];
    }

    __syncthreads();
```

```
    if(threadIdx.x == 0){
        *dot = temporary[0];
    }

}

int main()
{
    unsigned int n = 10000;
    unsigned int x = 256;
    double *h_prod;
    double *d_prod;
    double *h_x, *h_y;
    double *d_x, *d_y;
    double *d_temporary;
    Timer timer;

    h_prod = new double[n];
    h_x = new double[n];
    h_y = new double[n];


    // fill host array with data
    for(unsigned int i=0;i<n;i++){
        h_x[i] = 1;
        h_y[i] = 2;
    }

    // start timer
    timer.reset();

    // allocate memory
    cudaMalloc(&d_prod, sizeof(double));
    cudaMalloc(&d_x, n*sizeof(double));
    cudaMalloc(&d_y, n*sizeof(double));
    cudaMalloc(&d_temporary, x*sizeof(double));


    // copy data to device
    cudaMemcpy(d_x, h_x, n*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, n*sizeof(double), cudaMemcpyHostToDevice);


    dot_product_first_part<<<x, x>>>(d_x, d_y, d_temporary, n);
    dot_product_second_part<<<1, x>>>(d_temporary, d_prod);


    // copy data back to host
    cudaMemcpy(h_prod, d_prod, sizeof(double), cudaMemcpyDeviceToHost);

    // get runtime
    double time_elapsed = timer.get();


    // report results
    std::cout<<"dot product computed on GPU is: "<<*h_prod<<" and took "
```

```
    << time_elapsed << " s" <<std::endl;


    // free memory
    free(h_prod);
    free(h_x);
    free(h_y);
    cudaFree(d_prod);
    cudaFree(d_x);
    cudaFree(d_y);

}
```

7 Dot Product b)

Listing 9: Dot Product with GPU and CPU combined

```
#include <stdio.h>
#include <iostream>
#include "timer.hpp"
#include <random>


__global__ void dot_product(double *x, double *y, double *temporary, unsigned int n)
{
    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    __shared__ double cache[256];

    double temp = 0.0;
    while(index < n){
        temp += x[index]*y[index];

        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    for(int i = blockDim.x/2; i>0; i/=2)
    {
        __syncthreads();
        if(threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

    if(threadIdx.x == 0){
        temporary[blockIdx.x] = cache[0];
    }
}



int main()
{
    unsigned int n = 10000;
    unsigned int x = 256;
    double *h_prod;
    double *d_prod;
    double *h_x, *h_y;
    double *d_x, *d_y;
    double *d_temporary;
    double *h_temporary;
    Timer timer;

    h_prod = new double[n];
    h_x = new double[n];
    h_y = new double[n];
```

```
    h_temporary = new double[x];


    // fill host array with data
    for(unsigned int i=0;i<n;i++){
        h_x[i] = 1;
        h_y[i] = 2;
    }

    // start timer
    timer.reset();

    // allocate memory
    cudaMalloc(&d_prod, sizeof(double));
    cudaMalloc(&d_x, n*sizeof(double));
    cudaMalloc(&d_y, n*sizeof(double));
    cudaMalloc(&d_temporary, x*sizeof(double));
    cudaMemset(d_prod, 0.0, sizeof(double));


    // copy data to device
    cudaMemcpy(d_x, h_x, n*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, n*sizeof(double), cudaMemcpyHostToDevice);


    dot_product<<<x, x>>>(d_x, d_y, d_temporary, n);


    // copy data back to host
    cudaMemcpy(h_temporary, d_temporary, x*sizeof(double), cudaMemcpyDeviceToHost);

    // sum up elements
    double dot = 0;
    for(int i = 0; i < x; i++)
    {
        dot += h_temporary[i];
    }

    // get runtime
    double time_elapsed = timer.get();


    // report results
    std::cout<<"dot product computed on GPU and CPU is: "<<dot<<" and took "
    << time_elapsed << " s" <<std::endl;


    // free memory
    free(h_prod);
    free(h_x);
    free(h_y);
    cudaFree(d_prod);
    cudaFree(d_x);
    cudaFree(d_y);

}
```

8 Dot Product c)

Listing 10: Dot Product with atomicAdd on GPU

```
#include <stdio.h>
#include <iostream>
#include "timer.hpp"
#include <random>


__global__ void dot_product(double *x, double *y, double *dot, unsigned int n)
{
    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    __shared__ double cache[256];

    double temp = 0.0;
    while(index < n){
        temp += x[index]*y[index];

        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    for(int i = blockDim.x/2; i>0; i/=2)
    {
        __syncthreads();
        if(threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

    if(threadIdx.x == 0){
        atomicAdd(dot, cache[0]);
    }
}



int main()
{
    unsigned int n = 10000;
    double *h_prod;
    double *d_prod;
    double *h_x, *h_y;
    double *d_x, *d_y;
    Timer timer;

    h_prod = new double[n];
    h_x = new double[n];
    h_y = new double[n];


    // fill host array with data
```

```
    for(unsigned int i=0;i<n;i++){
        h_x[i] = 1;
        h_y[i] = 2;
    }

    // start timer
    timer.reset();

    // allocate memory
    cudaMalloc(&d_prod, sizeof(double));
    cudaMalloc(&d_x, n*sizeof(double));
    cudaMalloc(&d_y, n*sizeof(double));
    cudaMemset(d_prod, 0.0, sizeof(double));


    // copy data to device
    cudaMemcpy(d_x, h_x, n*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, n*sizeof(double), cudaMemcpyHostToDevice);


    dot_product<<<256, 256>>>(d_x, d_y, d_prod, n);

    // copy data back to host
    cudaMemcpy(h_prod, d_prod, sizeof(double), cudaMemcpyDeviceToHost);

    // get runtime
    double time_elapsed = timer.get();


    // report results
    std::cout<<"dot product computed on GPU is: "<<*h_prod<<" and took "
    << time_elapsed << " s" <<std::endl;


    // free memory
    free(h_prod);
    free(h_x);
    free(h_y);
    cudaFree(d_prod);
    cudaFree(d_x);
    cudaFree(d_y);

}
```

9 Dot Product d)

It can be seen that at least in my case, there is no difference in runtime for the different methods a to c. This surprises me but maybe is due to how I time the executions. Though, I wanted to take all the steps necessary (also the cudaMalloc and cudaMemcpy) into account. Again there are some similarities in respect to the number of vector entries with Figure 1 and Figure 2. So this behaviour could be memory related.

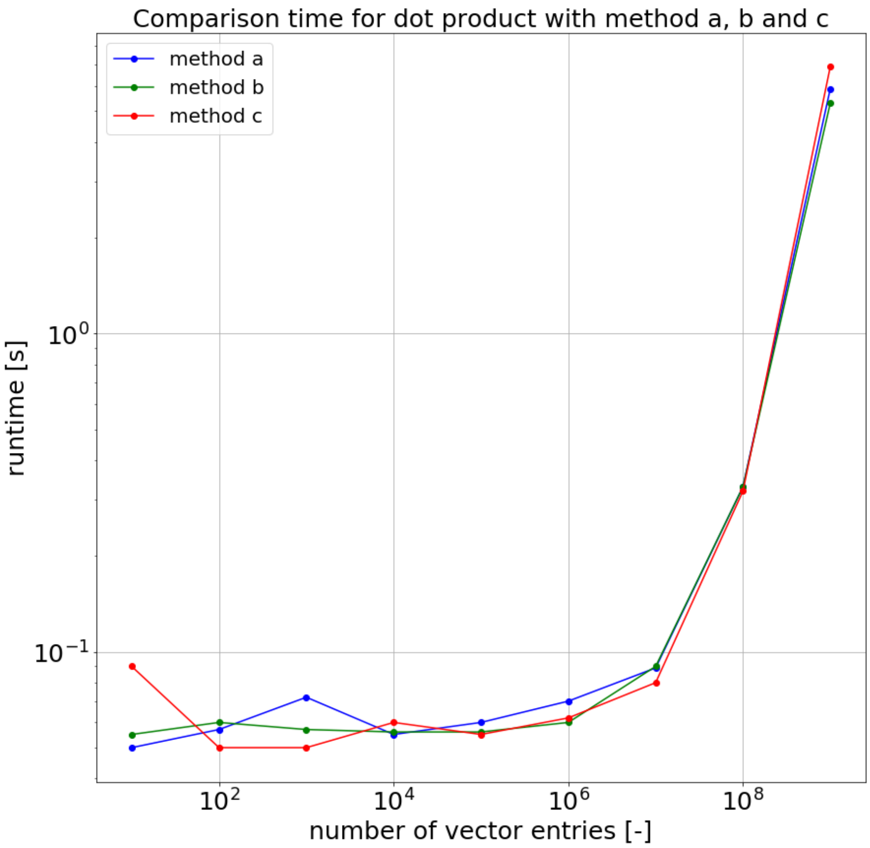

Figure 4: Runtimes for calculating dot product with different methods.