360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 1

Christian Gollmann, 01435044

Last update: October 30, 2020

Contents

1 Basic CUDA a)

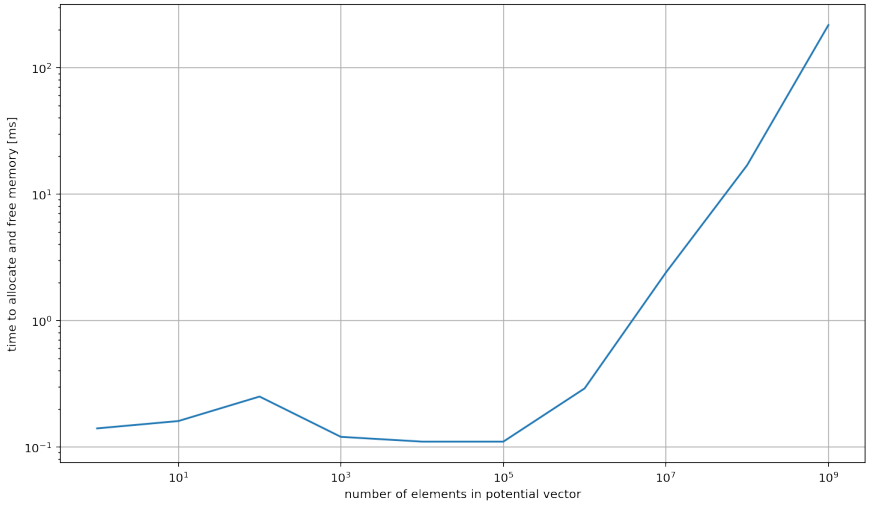

Figure 1: time to allocate and free memory for different sizes N

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)

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

```
1  #include <stdio.h>
2  #include "timer.hpp"
3
4  __global__
5  void init(int N)
6  {
7      double *x, *y;
8
9      x = new double[N];
10     y = new double[N];
11
12     for (int i = 0; i < N; i++)
13     {
14         x[i] = i;
15         y[i] = N-1-i;
16     }
17 }
18
19 int main(void)
20 {
21     int M = 1;
22     int N = 1000000;
23     Timer timer;
24
25     cudaDeviceSynchronize();
26     timer.reset();
27
28     init<<<(M+255)/256, 256>>>(N);
29
30     cudaDeviceSynchronize();
31     double time_elapsed = timer.get();
32
33     printf("Initializing in kernel took %g seconds", time_elapsed);
34
35     return EXIT_SUCCESS;
36 }
```

Listing 3: Copy the data

```
1  #include <stdio.h>
2  #include "timer.hpp"
3
4  int main(void)
5  {
6      int N = 1000000;
7
8      double*x, *y, *d_x, *d_y;
```

```
9       Timer timer;
10
11      x = new double[N];
12      y = new double[N];
13
14      for (int i = 0; i < N; i++)
15      {
16          x[i] = i;
17          y[i] = N-1-i;
18      }
19
20      cudaDeviceSynchronize();
21      timer.reset();
22
23      cudaMalloc(&d_x, N*sizeof(double));
24      cudaMalloc(&d_y, N*sizeof(double));
25      cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
26      cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
27
28      cudaDeviceSynchronize();
29      double time_elapsed = timer.get();
30
31      printf("Initializing outside took %g seconds", time_elapsed);
32
33      cudaFree(d_x);
34      cudaFree(d_y);
35      delete x;
36      delete y;
37
38      return EXIT_SUCCESS;
39  }
```

Listing 4: Copy each individual entry

```
1   #include <stdio.h>
2   #include "timer.hpp"
3
4   int main(void)
5   {
6       int N = 1000000;
7
8       double*x, *y, *d_x, *d_y;
9       Timer timer;
10
11      x = new double[N];
12      y = new double[N];
13
14      for (int i = 0; i < N; i++)
15      {
16          x[i] = i;
17          y[i] = N-1-i;
18      }
19
20      cudaDeviceSynchronize();
21      timer.reset();
22
23      cudaMalloc(&d_x, N*sizeof(double));
```

```
24      cudaMalloc(&d_y, N*sizeof(double));
25      for (int i = 0; i < N; i++)
26      {
27          cudaMemcpy(d_x+i, x+i, sizeof(double), cudaMemcpyHostToDevice);
28          cudaMemcpy(d_y+i, y+i, sizeof(double), cudaMemcpyHostToDevice);
29      }
30
31      cudaDeviceSynchronize();
32      double time_elapsed = timer.get();
33
34      printf("Initialising and copying by piece took %g seconds", time_elapsed);
35
36      cudaFree(d_x);
37      cudaFree(d_y);
38      delete x;
39      delete y;
40
41      return EXIT_SUCCESS;
42  }
```

3 Basic CUDA c)

Listing 5: CUDA kernel that sums two vectors

```
1  #include <stdio.h>
2  #include "timer.hpp"
3  #include <iostream>
4
5  __global__ void sumVectors(double *x, double *y, double *z, int N)
6  {
7      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
8
9      for(size_t i = thread_id; i < N; i += blockDim.x * gridDim.x)
10         z[i] = x[i] + y[i];
11
12 }
13
14 int main(void)
15 {
16     int N = 100;
17
18     double*x, *y, *z, *d_x, *d_y, *d_z;
19     Timer timer;
20
21     x = new double[N];
22     y = new double[N];
23     z = new double[N];
24
25
26     for (int i = 0; i < N; i++)
27     {
28         x[i] = i;
29         y[i] = N-1-i;
30         z[i] = 0;
31     }
32
33
34     cudaMalloc(&d_x, N*sizeof(double));
35     cudaMalloc(&d_y, N*sizeof(double));
36     cudaMalloc(&d_z, N*sizeof(double));
37     cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
38     cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
39     cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
40
41     cudaDeviceSynchronize();
42     timer.reset();
43
44     sumVectors<<<(N+255)/256, 256>>>(d_x, d_y, d_z, N);
45
46     cudaDeviceSynchronize();
47     double time_elapsed = timer.get();
48
49     cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
50
51     printf("Addition took %g seconds", time_elapsed);
52
53     std::cout << std::endl << "z[0] = " << z[0] << std::endl;
```

```
54
55      cudaFree(d_x);
56      cudaFree(d_y);
57      cudaFree(d_z);
58      delete x;
59      delete y;
60      delete z;
61
62      return EXIT_SUCCESS;
63  }
```

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.

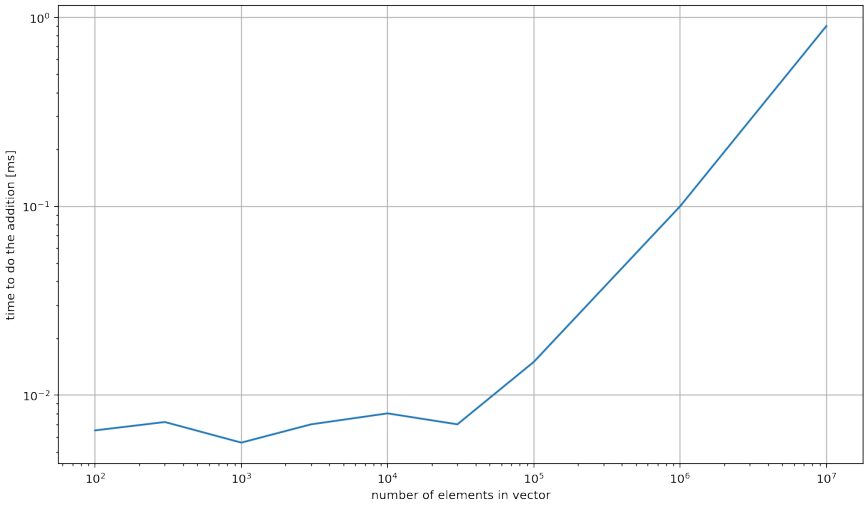

Figure 2: addition time for different values of N

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;
```

```
29          y[i] = N-1-i;
30          z[i] = 0;
31      }
32
33
34      cudaMalloc(&d_x, N*sizeof(double));
35      cudaMalloc(&d_y, N*sizeof(double));
36      cudaMalloc(&d_z, N*sizeof(double));
37      cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
38      cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
39      cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);
40
41      cudaDeviceSynchronize();
42      timer.reset();
43      for( int i = 0; i < 100; i++) {
44          sumVectors<<<(N+255)/256, 256>>>(d_x, d_y, d_z, N);
45          cudaDeviceSynchronize();
46      }
47
48      cudaDeviceSynchronize();
49      double time_elapsed = timer.get();
50
51      cudaMemcpy(z, d_z, N*sizeof(double), cudaMemcpyDeviceToHost);
52
53      printf("Addition took %g seconds", time_elapsed/100);
54
55      std::cout << std::endl << "z[0] = " << z[0] << std::endl;
56
57      cudaFree(d_x);
58      cudaFree(d_y);
59      cudaFree(d_z);
60      delete x;
61      delete y;
62      delete z;
63
64      return EXIT_SUCCESS;
65  }
```

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.

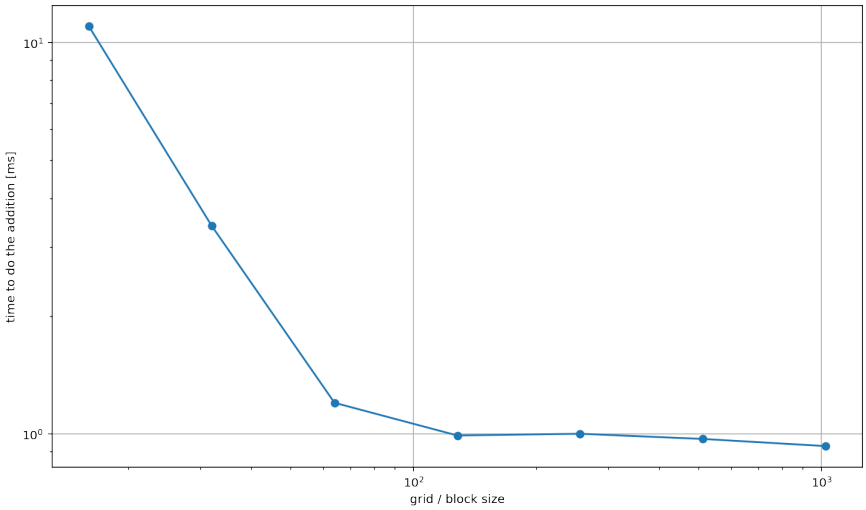

Figure 3: addition time for vector with e7 elements for different grid / block values

Listing 7: kernel call with different values x

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