Computational Science on Many-Core Architectures: Exercise 5:

Viktor Beck, 11713110

November 2022

Task 1: Performance Modeling: Parameter Identification

To ensure the correctness of the estimated values of this task at least 6 repetitions were made for each measurement and then the median of those was taken as the time (for that particular N).

a)

The PCI Express latency for `cudaMemcpy()` was estimated by simply measuring the time in a program for vector addition for different vector sizes N.

```
        // repetitions
        for (int j = 0; j < reps; j++){

            // MEASURING TIME FROM HERE
            timer.reset();

            cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);

            elapsed = timer.get();
            // TO HERE
            time_vec.push_back(elapsed);

        }
```

Since the latency is independent from N an estimate can be made from looking at the constant part of the measurements (for small N) which can be seen in the Figure. The estimates for the two GPUs are:

- RTX: $t = 4\mu s$
- K40: $t = 7\mu s$

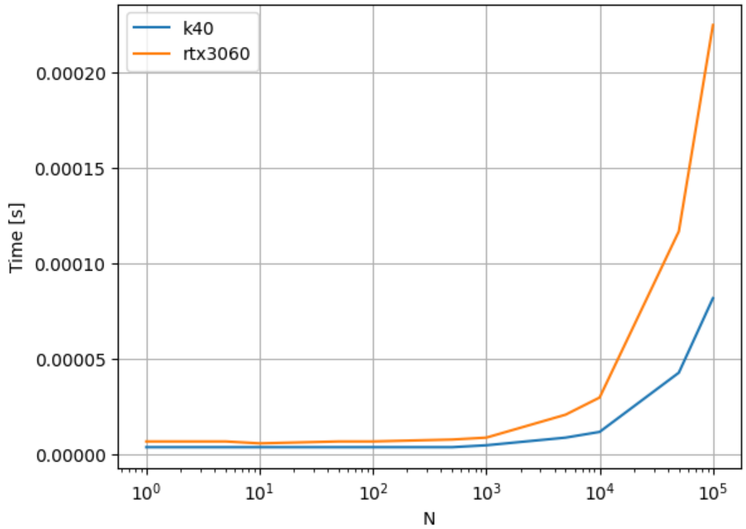

Figure 1: Times for `cudaMemcopy()` for different N

b)

The measurements were taken similar to a) but here we are simply measuring the call of an empty kernel. It is empty such that we only have the call and nothing else in the measurement.

```
__global__ void empty(int n, double *x, double *y, double *z){}
```

By looking at the constant part of the plot the following estimates were made:

- RTX: $t = 5\mu s$
- K40: $t = 11\mu s$

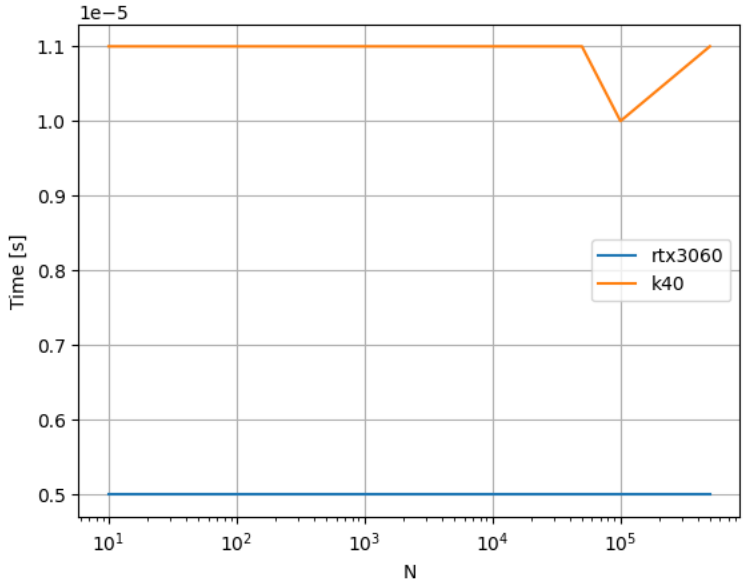

Figure 2: Times for kernel launch (256x256) for different N

c)

In the figure below we can see the peak memory bandwidth for both GPUs measured with the vector addition kernel we have already used in a previous exercise. The highest values measured for the bandwidths are:

- RTX: $B_{peak} = 331.446$ GB/s
- K40: $B_{peak} = 167.598$ GB/s

The GTX 3060 is allowing a total bandwidth of 360 GB/s, the K40 a bandwidth of 288 GB/s. This is quite close to our own measurement. Also, one can see that we have the typical S-shape of the plot:

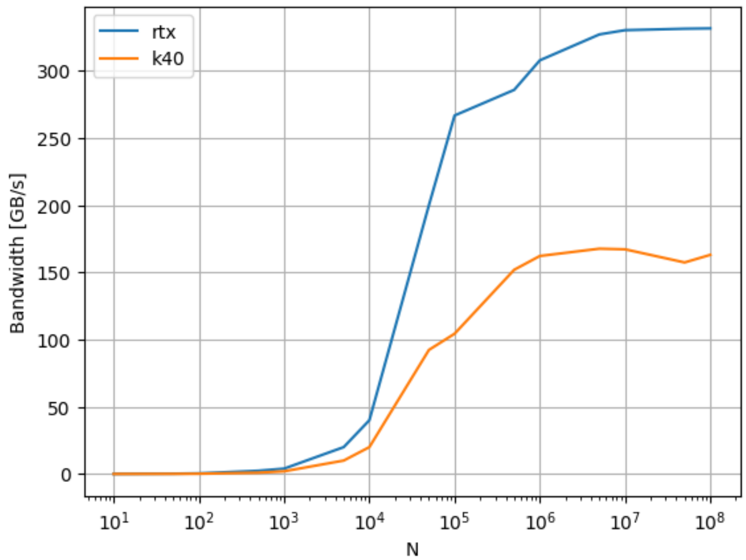

Figure 3: Bandwidth of the vector addition (256x256) for different N

d)

Here only a single thread (blocks x threads = 1x1) is doing the `atomicAdd()` (with an arbitrary input) such that we only have one computation at a time in a for loop which iterates N times.

```
__global__ void singleAdd(int n, double *x, double *y)
{
    for (unsigned int i=0; i < n; i++)
    {
        atomicAdd(&x[i],1);
    }
}
```

One can see in the figure below the number of `atomicAdd()` per second. The maximum is $1.81362 * 10^8$ for N = 10^8.

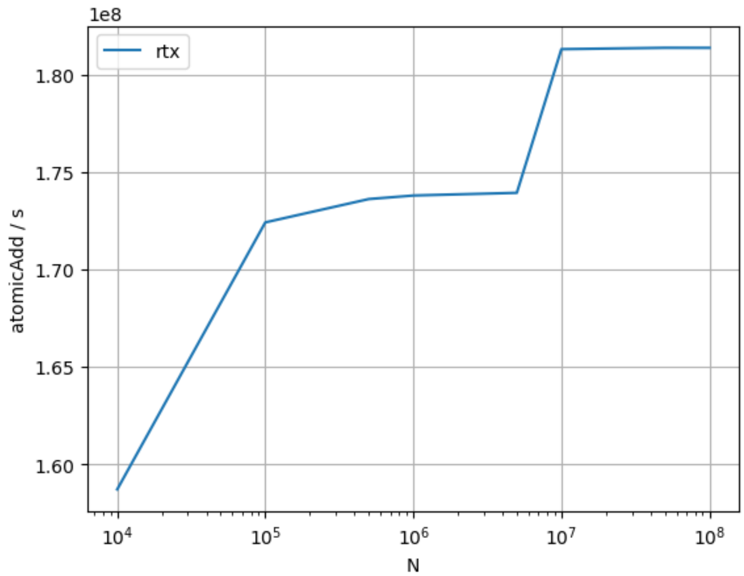

Figure 4: atomicAdd for different N (here N is the number of iterations the single thread has to do)

e)

Here we get the following peak floating point rates per second for the multiply-add (I simply reused the vector addition kernel - see code):

- RTX: 28.2187 GFlops/s
- K40: 12.221 GFlops/s

They were calculated as $2 * 8 * N/t * 10^{-9}$ GFlops/s.

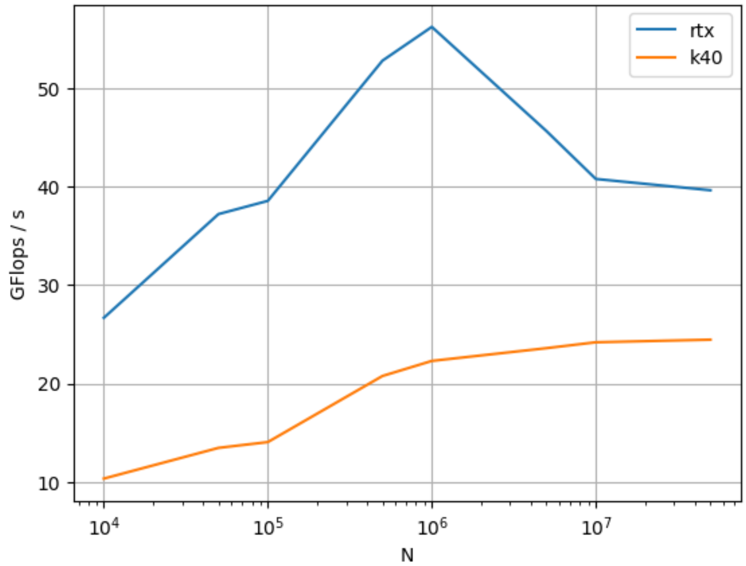

Figure 5: atomicAdd for different N (here N is the number of iterations the single thread has to do)

Appendix

Task 1.a)e)

```

#include <iostream>
#include <stdio.h>
#include <vector>
#include <algorithm>
#include "timer.hpp"

using Vector = std::vector<double>;

__global__ void addition(int n, double *x, double *y, double *z)
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i = id; i < n; i += blockDim.x*gridDim.x){
    //z[i] = x[i] + y[i];
    z[i] += x[i] + y[i];
  }
}

int main(void)
{
    double *x, *y, *z, *d_x, *d_y, *d_z;
    double elapsed;

    Vector time_memcpy;
    Vector flops;

    Timer timer;
    int reps = 6; // has to be an even number

    Vector N_vec = {10000,50000,100000,500000,1000000,5000000,10000000,50000000};
    for (const auto &N : N_vec){
        Vector time_vec;

        // Allocate host memory and initialize
        x = (double*)malloc(N*sizeof(double));
        y = (double*)malloc(N*sizeof(double));
        z = (double*)malloc(N*sizeof(double));

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

        // Allocate device memory and copy host data over
        cudaMalloc(&d_x, N*sizeof(double));
        cudaMalloc(&d_y, N*sizeof(double));
        cudaMalloc(&d_z, N*sizeof(double));

        // repetitions
        for (int j = 0; j < reps; j++){

            // MEASURING TIME FROM HERE
            timer.reset();

            cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);

```

```
            elapsed = timer.get();
            // TO HERE
            time_vec.push_back(elapsed);

        }

        cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_z, z, N*sizeof(double), cudaMemcpyHostToDevice);

        addition<<<256, 256>>>(N, d_x, d_y, d_z);

        std::sort(time_vec.begin(), time_vec.end());
        time_memcpy.push_back(time_vec[reps/2]);
        //bandwidth.push_back(3*(N)*sizeof(double)/time_vec[reps/2]*1e-9);
        flops.push_back(2 * 8 * N / time_vec[reps/2] * 1e-9);

        //printf("Elapsed: %g\n", time);

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

        cudaFree(d_x);
        cudaFree(d_y);
        cudaFree(d_z);
        free(x);
        free(y);
        free(z);
    }

    printf("s:\n");
    for (const auto& value : time_memcpy){
        std::cout << value << "," << std::endl;
    }
    printf("\nGFlops/s:\n");
    for (const auto& value : flops){
        std::cout << value << "," << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Task 1.b)c)

```

#include <iostream>
#include <stdio.h>
#include <vector>
#include <algorithm>
#include "timer.hpp"

using Vector = std::vector<double>;


__global__ void addition(int n, double *x, double *y, double *z)
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  for (size_t i = id; i < n; i += blockDim.x*gridDim.x){
    z[i] = x[i] + y[i];
    //printf("z: %g\n", z[i*k]);
  }
}
```

```

__global__ void empty(int n, double *x, double *y, double *z){}

int main(void)
{
    double *x, *y, *z, *d_x, *d_y, *d_z;
    double elapsed;

    Vector bandwidth;
    Vector time_kernel_launch;

    Timer timer;
    int reps = 10; // has to be an even number

    Vector N_vec = {10,100,1000,10000,50000,100000,500000};
    for (const auto &N : N_vec){
        Vector time_vec;

        // Allocate host memory and initialize
        x = (double*)malloc(N*sizeof(double));
        y = (double*)malloc(N*sizeof(double));
        z = (double*)malloc(N*sizeof(double));

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

        // Allocate device memory and copy host data over
        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);

        // repetitions
        for (int j = 0; j < reps; j++){

            // MEASURING TIME FROM HERE
            cudaDeviceSynchronize();
            timer.reset();

            empty<<<256, 256>>>(N, d_x, d_y, d_z);
            //addition<<<256, 256>>>(N, d_x, d_y, d_z);

            cudaDeviceSynchronize();
            elapsed = timer.get();
            time_vec.push_back(elapsed);

            // TO HERE
        }
        std::sort(time_vec.begin(), time_vec.end());
        time_kernel_launch.push_back(time_vec[reps/2]);
        bandwidth.push_back(3*(N)*sizeof(double)/time_vec[reps/2]*1e-9);
        //printf("Elapsed: %g\n", time);

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

        cudaFree(d_x);
```

```
        cudaFree(d_y);
        cudaFree(d_z);
        free(x);
        free(y);
        free(z);
    }

    printf("s:\n");
    for (const auto& value : time_kernel_launch){
        std::cout << value << "," << std::endl;
    }

    printf("Gb/s:\n");
    for (const auto& value : bandwidth){
        std::cout << value << "," << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Task 1.d)

```

#include <iostream>
#include <stdio.h>
#include <vector>
#include <algorithm>
#include "timer.hpp"

using Vector = std::vector<double>;

__global__ void singleAdd(int n, double *x, double *y)
{
    for (unsigned int i=0; i < n; i++)
    {
        atomicAdd(&x[i],1);
    }
}

int main(void)
{
    double *x, *y, *d_x, *d_y;
    double elapsed;

    Vector add_per_second;

    Timer timer;
    int reps = 6; // has to be an even number

    Vector N_vec = {10000,100000,500000,1000000,5000000,10000000,50000000,100000000};
    for (const auto &N : N_vec)
    {
        Vector time_vec;

        x = (double*)malloc(N*sizeof(double));
        cudaMalloc(&d_x, N * sizeof(double));
        cudaMemcpy(d_x, x, N * sizeof(double), cudaMemcpyHostToDevice);

        y = (double*)malloc(N*sizeof(double));
        cudaMalloc(&d_y, N * sizeof(double));
```

```
        cudaMemcpy(d_y, y, N * sizeof(double), cudaMemcpyHostToDevice);

        // repetitions
        for (int j = 0; j < reps; j++)
        {

            // MEASURING TIME FROM HERE
            cudaDeviceSynchronize();
            timer.reset();

            singleAdd<<<1, 1>>>(N, d_x, d_y);

            cudaDeviceSynchronize();
            elapsed = timer.get();
            time_vec.push_back(elapsed);

            // TO HERE
        }
        std::sort(time_vec.begin(), time_vec.end());
        add_per_second.push_back(N/time_vec[reps / 2]);

        cudaFree(d_x);
        free(x);
    }

    printf("s:\n");
    for (const auto &value : add_per_second)
    {
        std::cout << value << "," << std::endl;
    }

    return EXIT_SUCCESS;
}
```