Computational Science on Many-Core Architectures: Exercise 3:

Viktor Beck, 11713110

8. November 2022

Task 1: Strided and Offset Memory Access

a)

Here we only sum every k-th element in the vector addition. One can see in Fig. 1 that we have an exponential decrease in the effective bandwidth. Probably this is due to spacial locality: the localities of the elements with index `i*k` are not neighbouring for subsequent ks and therefore the kernel can not be as fast as with elements located closer to each other. As expected the newer RTX3060 GPU is faster than the K40. The noise for the RTX GPU is possibly caused by itself attempting to optimize the computations.

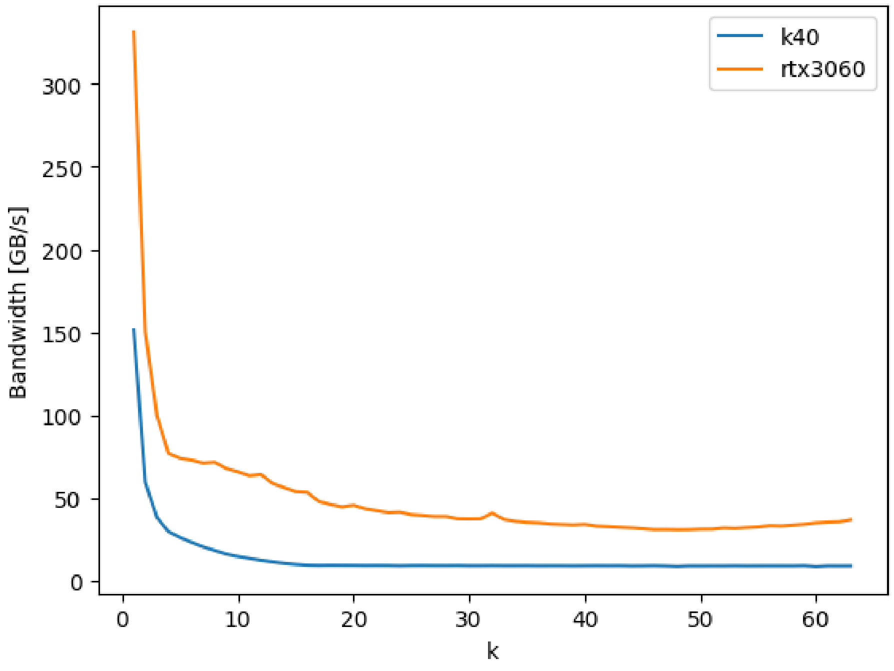

Figure 1: Strided memory access: Bandwidth for vector addition for different k. The time was measured by taking the median over 6 repetitions. Note: for computing the bandwidth the formula `3*(N/k)*sizeof(double)*t` was used, 3 because 2 reads and 1 write.

b)

Here we skipped the first k elements of the vector. In Fig. 2 one can see that the effective bandwidth is jumping significantly for every k which is dividable by 4 and even more by every k which is dividable by 16. These minima are obviously not a coincidence. This is probably because of the blocks and threads per block being dividable by these numbers. However, it seems that taking k = 16, 32, 48, 64, ... (I even tested it for k $>$ 64) the effective bandwidth is the highest and therefore, one could recommend these values of k for future optimizations. Again here we can see that the RTX3060 GPU is much faster - even twice as fast as the K40 GPU.

So in general, one could recommend this approach over the one from a) where we take N/k since here we have better spacial localities when it comes to accessing the elements of the vectors.

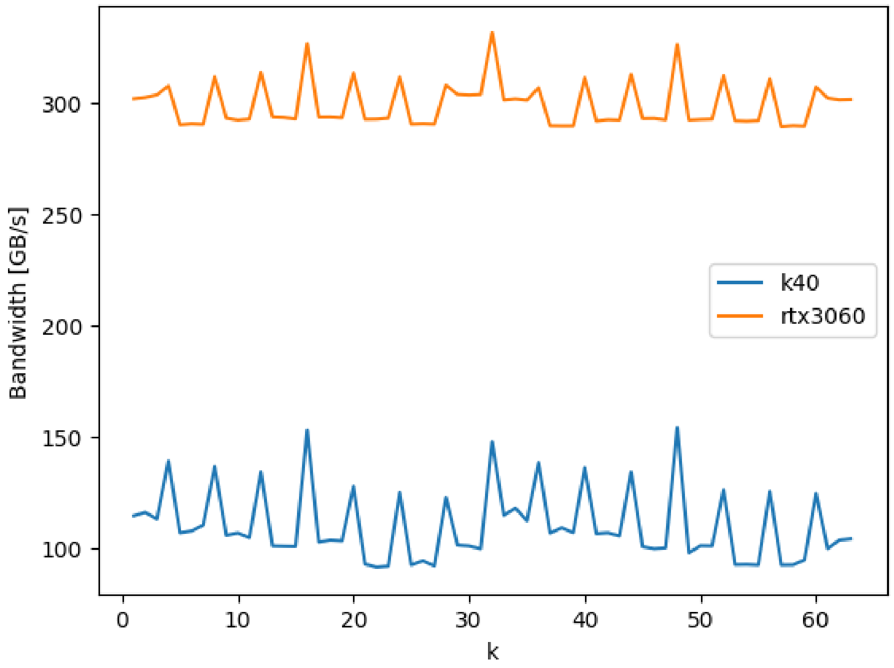

Figure 2: Offset memory access: Bandwidth for vector addition for different k

Task 2: Dense Matrix Transpose

a)

The problem here was a leakage of 800 Bytes (for N=10) which was solved by simply adding `cudafree(cuda_A)` and `free(A)` at the bottom of the main function. Consequently, the leakage was due to the 'unfreed' space of the GPU kernel. Note: I guess in the description of the task you meant that the problem here is the leakage and not the wrong results of the transposition itself. I corrected this in b).

b)

The transposed matrix of my friend's code does not produce correct results for the case that the total number of threads is smaller than N*N. If we use a for loop in the transpose function such that each thread can work again on the task we get the problem that a thread does not know which elements are already in the correct place in the matrix. E.g.: thread X writes to A[42]; after that, thread Y reads from A[42] to write to A[...] but this means that it is already reading the updated value which leads to an incorrect transposition. In an naive approach we simply make sure that the number of threads is always bigger than N*N:

```
            transpose<<<(N*N+255)/256, 256>>>(cuda_A, N);
```

c)

Here a new matrix B was introduced. The problem here is that we have double the amount of data that has to be passed to the GPU kernel than before with just one matrix. The naive approach was the following:

```
__global__ void transpose(double *A, double *B, int N)
{
    int t_idx = blockIdx.x * blockDim.x + threadIdx.x;
    // loop for the case N*N > number of threads
    for (size_t i = t_idx; i < (N*N); i += blockDim.x*gridDim.x){
        int row_idx = i / N;
        int col_idx = i % N;
        B[row_idx * N + col_idx] = A[col_idx * N + row_idx];
    }
}
```

d)

A more sophisticated version of the kernel was then implemented (with inspiration from the link you provided) as follows:

```
const int TILE_DIM = 16;
const int BLOCK_ROWS = 16;

__global__ void transposeCoalesced(double *A, double *B)
{
    //create blocks (tiles) in shared memory
    __shared__ double tile[TILE_DIM][TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    //write from A into shared tile
    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS){
        tile[threadIdx.y+i][threadIdx.x] = A[(y+i)*width + x];
    }

    __syncthreads(); // wait until all blocks are finished
    //transpose block
    x = blockIdx.y * TILE_DIM + threadIdx.x;
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    // assign blocks to right position in B
    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS){
        B[(y+i)*width + x] = tile[threadIdx.x][threadIdx.y + i];
    }
}
```

Here we cut the matrix into tiles/blocks of 16x16 elements. Each tile is transposed separately and then assigned to the right position in matrix B.

e)

The effective bandwidth of the naive implementations of b) and c) and the coalesced one (with loading blocks of 16x16) can be seen below. We can observe that the difference between the different approaches start out similarly but for larger N (up to 2^{14}) there is a significant difference. It seems that the coalescent implementation of d) stabilizes for N between 10^2 and 10^3 while the implementation of b) drops down. Surprisingly with the GTX GPU the implementation of c) works out quite well compared to the others. As expected, there is a difference between the GTX and the K40 GPU.

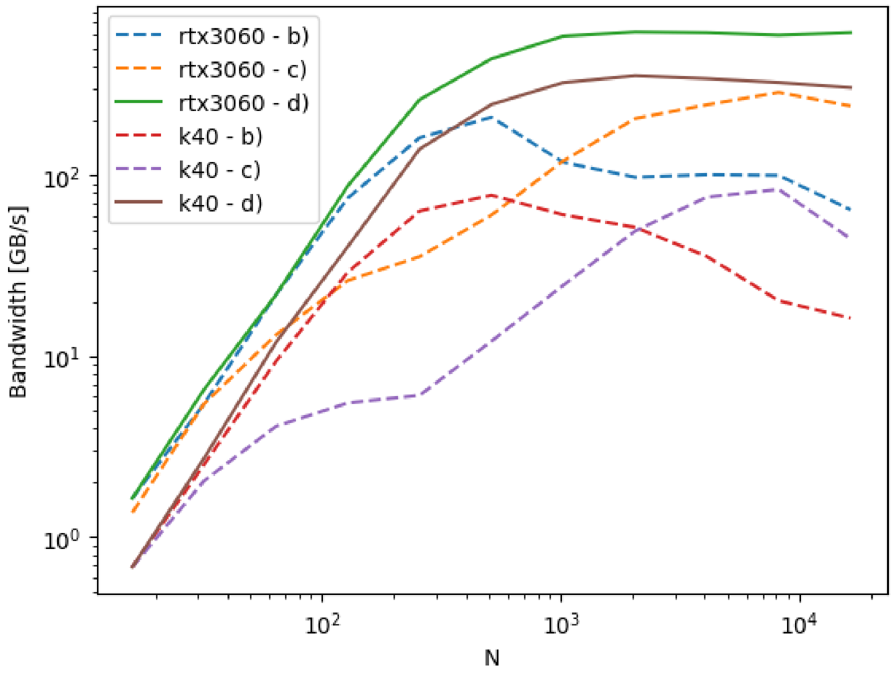

Figure 3: Effective bandwidth for different N (16 to 16384) for the K40 and the GTX 3060 GPU

Appendix

Task 1.ab)

```

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

using Vector = std::vector<double>;


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

int main(void)
{

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

    Vector bandwidth;
    Timer timer;

    //int N_min = 100000000;
    //int N_max = 100000000;
    int reps = 6; // has to be an even number

    int N = 100000000;

    // 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);

    for (int k = 1; k < 64; k++){
        //std::cout << "k = " << k << std::endl;

        Vector time_vec;

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

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

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

            cudaDeviceSynchronize();
            time_vec.push_back(timer.get());
            //std::cout << timer.get() << std::endl;

            // TO HERE
        }
        std::sort(time_vec.begin(), time_vec.end());
        bandwidth.push_back(3*(N-k)*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("Gb/s:\n");
    for (const auto& value : bandwidth){
        std::cout << value << "," << std::endl;
    }


    return EXIT_SUCCESS;
}
```

Task 2.ab).

```
#include <stdio.h>
#include <iostream>
#include "timer.hpp"
#include "cuda_errchk.hpp" // for error checking of CUDA calls
#include <vector>
#include <algorithm>

__global__
void transpose(double *A, int N)
{
    int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
    int row_idx = t_idx / N;
    int col_idx = t_idx % N;
    if (row_idx < N && col_idx < N){
        int tmp;
        tmp = A[col_idx * N + row_idx];
        A[col_idx * N + row_idx] = A[row_idx * N + col_idx];
        A[row_idx * N + col_idx] = tmp;
    }

```

```
}

void print_A(double *A, int N)
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; ++j) {
            std::cout << A[i * N + j] << ",␣";
    }
    std::cout << std::endl;
    }
}

int main(void)
{
    std::vector<double> N_vec = {16,32,64,128,256,512,1024,2048,4096,8192,16384};
    std::vector<double> bandwidth;

    double *A, *cuda_A;
    Timer timer;
    int reps = 6;

    for (const auto &N : N_vec){
        std::vector<double> time_vec;

        // Allocate host memory and initialize
        A = (double*)malloc(N*N*sizeof(double));

        for (int i = 0; i < N*N; i++) {
        A[i] = i;
        }

        //print_A(A, N);

        // Allocate device memory and copy host data over
        CUDA_ERRCHK(cudaMalloc(&cuda_A, N*N*sizeof(double)));

        // copy data over
        CUDA_ERRCHK(cudaMemcpy(cuda_A, A, N*N*sizeof(double), cudaMemcpyHostToDevice));

        // repetitions
        for (int j = 0; j < reps; j++){
            // wait for previous operations to finish, then start timings
            CUDA_ERRCHK(cudaDeviceSynchronize());
            timer.reset();

            // Perform the transpose operation
            transpose<<<(N*N+255)/256, 256>>>(cuda_A, N);

            // wait for kernel to finish, then print elapsed time
            CUDA_ERRCHK(cudaDeviceSynchronize());
            time_vec.push_back(timer.get());
        }
        std::sort(time_vec.begin(), time_vec.end());
        bandwidth.push_back(4*N*N*sizeof(double)/time_vec[reps/2]*1e-9);

        // copy data back (implicit synchronization point)
        CUDA_ERRCHK(cudaMemcpy(A, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost));

        //print_A(A, N);

        cudaFree(cuda_A);
        free(A);
    }

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

    CUDA_ERRCHK(cudaDeviceReset());  // for CUDA leak checker to work

    return EXIT_SUCCESS;
}
```

Task 2.c).

```
#include <stdio.h>
#include <iostream>
#include "timer.hpp"
#include "cuda_errchk.hpp" // for error checking of CUDA calls
#include <vector>
#include <algorithm>

__global__ void transpose(double *A, double *B, int N)
{
    int t_idx = blockIdx.x * blockDim.x + threadIdx.x;
    // loop for the case N*N > number of threads
    for (size_t i = t_idx; i < (N*N); i += blockDim.x*gridDim.x){
        int row_idx = i / N;
        int col_idx = i % N;
        B[row_idx * N + col_idx] = A[col_idx * N + row_idx];
    }
}

void print_A(double *A, int N)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; ++j)
        {
            std::cout << A[i * N + j] << ",␣";
        }
        std::cout << std::endl;
    }
}

int main(void)
{
    //int N = 33;
    std::vector<double> N_vec = {16,32,64,128,256,512,1024,2048,4096,8192,16384};
    std::vector<double> bandwidth;

    // declare A and B
    double *A, *B, *cuda_A, *cuda_B;
    Timer timer;
    int reps = 6;

    for (const auto &N : N_vec){
        std::vector<double> time_vec;

        // Allocate host memory and initialize
        A = (double *)malloc(N * N * sizeof(double));
        B = (double *)malloc(N * N * sizeof(double));

        for (int i = 0; i < N * N; i++)
        {
            A[i] = i;
            B[i] = 0;
        }

        //print_A(A, N);
        //print_A(B, N);

        // Allocate device memory and copy host data over
        CUDA_ERRCHK(cudaMalloc(&cuda_A, N * N * sizeof(double)));
        CUDA_ERRCHK(cudaMalloc(&cuda_B, N * N * sizeof(double)));

        // copy data over
        CUDA_ERRCHK(cudaMemcpy(cuda_A, A, N * N * sizeof(double), cudaMemcpyHostToDevice));
        CUDA_ERRCHK(cudaMemcpy(cuda_B, B, N * N * sizeof(double), cudaMemcpyHostToDevice));

        // repetitions
        for (int j = 0; j < reps; j++){
            // wait for previous operations to finish, then start timings
            CUDA_ERRCHK(cudaDeviceSynchronize());
            timer.reset();

            // Perform the transpose operation
            transpose<<<(N + 255) / 256, 256>>>(cuda_A, cuda_B, N);

            // wait for kernel to finish, then print elapsed time
            CUDA_ERRCHK(cudaDeviceSynchronize());
            time_vec.push_back(timer.get());
        }
        std::sort(time_vec.begin(), time_vec.end());
        bandwidth.push_back(4*N*N*sizeof(double)/time_vec[reps/2]*1e-9);

        // copy data back (implicit synchronization point)
        CUDA_ERRCHK(cudaMemcpy(A, cuda_A, N * N * sizeof(double), cudaMemcpyDeviceToHost));
        CUDA_ERRCHK(cudaMemcpy(B, cuda_B, N * N * sizeof(double), cudaMemcpyDeviceToHost));

        //print_A(B, N);

        cudaFree(cuda_A);
        free(A);
        cudaFree(cuda_B);
        free(B);
    }

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

    CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work

    return EXIT_SUCCESS;
}
```

Task 2.d).

```
#include <stdio.h>
```

```
#include <iostream>
#include "timer.hpp"
#include "cuda_errchk.hpp" // for error checking of CUDA calls
#include <vector>
#include <algorithm>

const int TILE_DIM = 16;
const int BLOCK_ROWS = 16;

__global__ void transposeCoalesced(double *A, double *B)
{
    //create blocks (tiles) in shared memory
    __shared__ double tile[TILE_DIM][TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    //write from A into shared tile
    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS){
        tile[threadIdx.y+i][threadIdx.x] = A[(y+i)*width + x];
    }

    __syncthreads(); // wait until all blocks are finished
    //transpose block
    x = blockIdx.y * TILE_DIM + threadIdx.x;
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    // assign blocks to right position in B
    for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS){
        B[(y+i)*width + x] = tile[threadIdx.x][threadIdx.y + i];
    }
}

void print_A(double *A, int N)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; ++j)
        {
            std::cout << A[i * N + j] << ",␣";
        }
        std::cout << std::endl;
    }
}

int main(void)
{
    std::vector<double> N_vec = {16,32,64,128,256,512,1024,2048,4096,8192,16384};
    std::vector<double> bandwidth;

    // declare A and B
    double *A, *B, *cuda_A, *cuda_B;
    Timer timer;
    int reps = 6;

    for (const auto &N : N_vec){
        std::vector<double> time_vec;

        dim3 dimGrid(N/TILE_DIM, N/TILE_DIM, 1);
        dim3 dimBlock(TILE_DIM, BLOCK_ROWS, 1);

        // Allocate host memory and initialize
        A = (double *)malloc(N * N * sizeof(double));
        B = (double *)malloc(N * N * sizeof(double));

        for (int i = 0; i < N * N; i++)
        {
            A[i] = i;
            B[i] = 0;
        }

        //print_A(A, N);
        //print_A(B, N);

        // Allocate device memory and copy host data over
        CUDA_ERRCHK(cudaMalloc(&cuda_A, N * N * sizeof(double)));
        CUDA_ERRCHK(cudaMalloc(&cuda_B, N * N * sizeof(double)));

        // copy data over
        CUDA_ERRCHK(cudaMemcpy(cuda_A, A, N * N * sizeof(double), cudaMemcpyHostToDevice));
        CUDA_ERRCHK(cudaMemcpy(cuda_B, B, N * N * sizeof(double), cudaMemcpyHostToDevice));

        // repetitions
        for (int j = 0; j < reps; j++){
            // wait for previous operations to finish, then start timings
            CUDA_ERRCHK(cudaDeviceSynchronize());
            timer.reset();

            // Perform the transpose operation
            transposeCoalesced<<<dimGrid, dimBlock>>>(cuda_A, cuda_B);

            // wait for kernel to finish, then print elapsed time
            CUDA_ERRCHK(cudaDeviceSynchronize());
            time_vec.push_back(timer.get());
        }
        std::sort(time_vec.begin(), time_vec.end());
        bandwidth.push_back(4*N*N*sizeof(double)/time_vec[reps/2]*1e-9);

        // copy data back (implicit synchronization point)
        //CUDA_ERRCHK(cudaMemcpy(A, cuda_A, N * N * sizeof(double), cudaMemcpyDeviceToHost));
        CUDA_ERRCHK(cudaMemcpy(B, cuda_B, N * N * sizeof(double), cudaMemcpyDeviceToHost));

        //print_A(B, N);

        cudaFree(cuda_A);
        free(A);
        cudaFree(cuda_B);
        free(B);
    }

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

    CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work

    return EXIT_SUCCESS;
}
```