Vienna University of Technology

360.252 Computational Science on Many Core Architectures

Institute for Microelectronics

Exercise 4

Authors:
Camilo Tello Fachin
12127084

Supervisor:
Dipl.-Ing. Dr.techn. Karl Rupp

November 10, 2022

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Abstract

Here documented the results of exercise 4.

Contents

1 Strided and Offset Memory Access (3 Points)

1.1 Task a

In figure 1 it can be observed that strided memory access has a negative impact on the effective memory bandwidth, the more the data are strided. A recommenddation could be, to try to omit strided mermoy access and more try to access data in blocks which are contiguous by design.

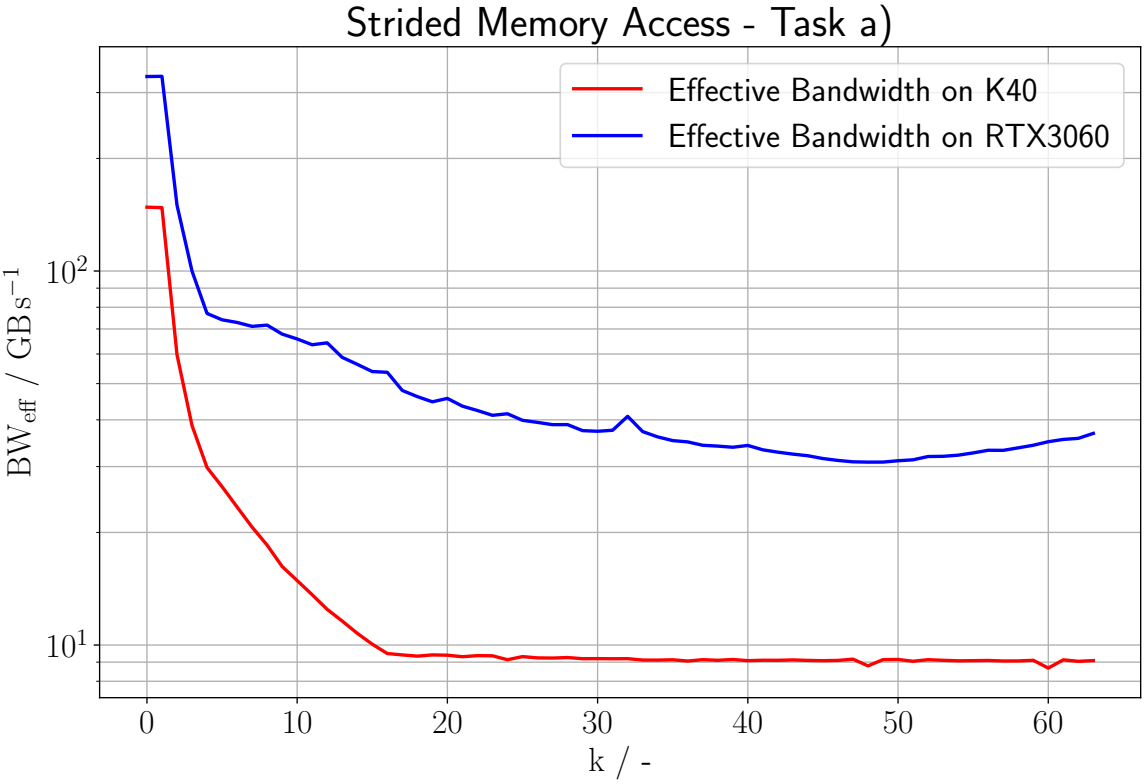

Figure 1: Plot for task 1a - see code in appendix A

1.2 Task b

In figure 2, it can be observed that the effective memory bandwidth is always around the maximal bandwidth and has its peaks for $\mathrm{k} \bmod (4) = 0$ since sizes of this pattern fit better in `<<<256,256>>>`.

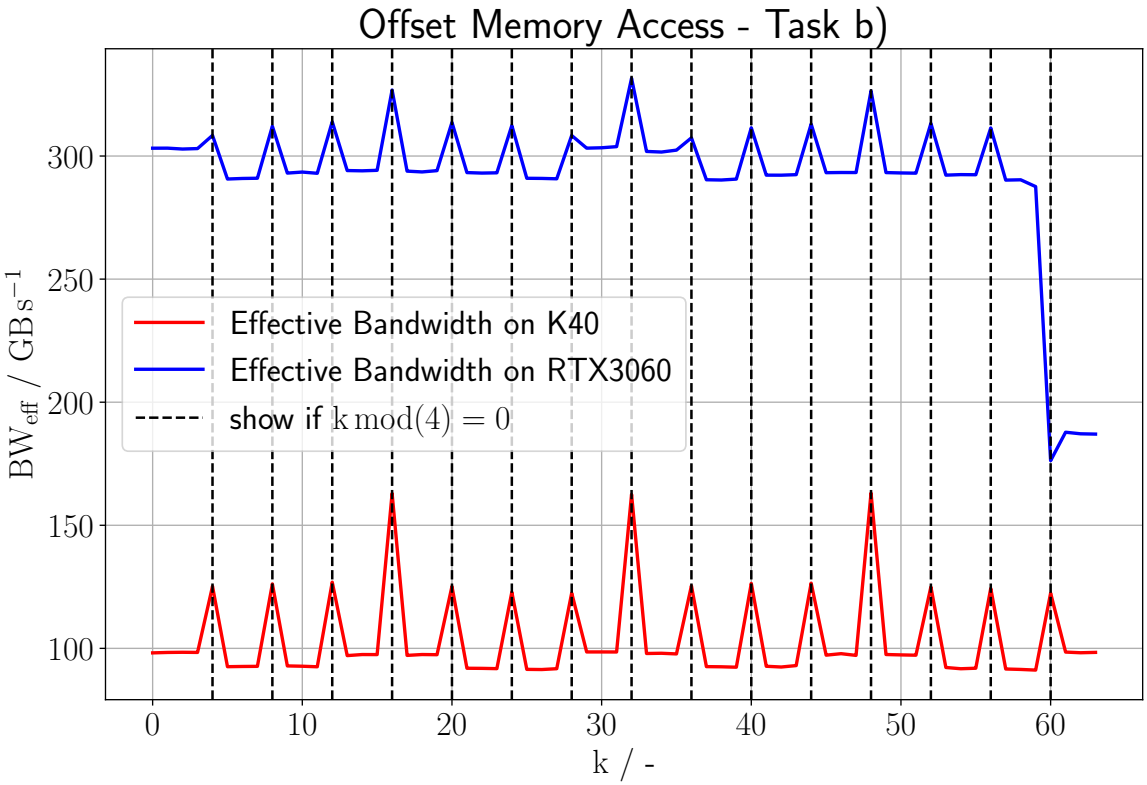

Figure 2: Plot for task 1b - see code in appendix B

2 Dense Matrix Transpose (6 Points)

2.1 Task a

The implementation of my friend yields a memory leakage of 800 bytes in one allocation because the allocated memory is not free'd after the program has finished. Additionally, the implementation does not yield correct results for aribtrary matrix dimensions.

Console Output on RTX3060 Environment for transpose.cu

```
$> timeout 30s cuda-memcheck --tool memcheck --leak-check full ./1a404f09.out 2>&1

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
.
.
.
90, 91, 92, 93, 94, 95, 96, 97, 98, 99,

Time for transpose: 0.00088
 Effective  bandwidth: 0.00181818 GB/sec

0, 10, 20, 30, 40, 50, 60, 70, 80, 90,
.
.
.
9, 19, 29, 39, 49, 59, 69, 79, 89, 99,
========= CUDA-MEMCHECK
========= Leaked 800 bytes at 0x7efd1c600000
========= Saved host backtrace up to driver entry point at cudaMalloc time
========= Host Frame:/lib/x86_64-linux-gnu/libcuda.so.1 [0x292c27]
========= Host Frame:./1a404f09.out [0x30e03]
========= Host Frame:./1a404f09.out [0xc3db]
========= Host Frame:./1a404f09.out [0x41a7f]
========= Host Frame:./1a404f09.out [0x81b2]
========= Host Frame:./1a404f09.out (main + 0xae) [0x7c9f]
========= Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf3) [0x24083]
========= Host Frame:./1a404f09.out (_start + 0x2e) [0x7a6e]
=========
========= LEAK SUMMARY: 800 bytes leaked in 1 allocations
========= ERROR SUMMARY: 1 error
```

2.2 Task b

In order to be able to transpose arbitrarily sized matrices the kernel arguments need to be adapted. See Appendix C line 63, where the the argument $<<<(N+255)/256, 256>>>$ is changed to $<<<(N^2+255)/256, 256>>>$. The effective Bandwidth of the fixed Kernel is 0.00182025 GB/sec.

2.3 Task c

In Task c the in-place requirement is dropped, this is done by adapting the Kernel in the following way:

C++ Listing for EX2 c Kernel Part

```
1  __global__
2  void transpose(double *A, double *B, int N)
3  {
4    int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
5    int row_idx = t_idx / N;
6    int col_idx = t_idx % N;
7
8    if (row_idx < N && col_idx < N) B[row_idx * N + col_idx] = A[col_idx * N + row_idx];
9  }
```

Also one needs to initialize device and host variables of the same size for the B matrix and one mustn't forget to free the additional memory afterwards. See code in appendix C.

A CPP CUDA Code - Strided and Offset Memory Access - Task 1a

C++ Listing for EX1 a)

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

__global__ void addVec_kth(double *x, double *y, double *z, int N, int k) {
  unsigned int total_threads = blockDim.x * gridDim.x;
  unsigned int global_tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (k==0) {
    k = 1;
  }
  for (unsigned int i = global_tid; i<N/k; i += total_threads) {
    z[i*k] = x[i*k] + y[i*k];
  }
}

// findMedian function for any vector lenghts, source geeksforgeeks.com
double findMedian(std::vector<double> a,
                  int n)
{
    if (n % 2 == 0) {
        std::nth_element(a.begin(),
                         a.begin() + n / 2,
                         a.end());
        std::nth_element(a.begin(),
                         a.begin() + (n - 1) / 2,
                         a.end());
        return (double)(a[(n - 1) / 2]
                        + a[n / 2])
               / 2.0;
    }
    else {
        std::nth_element(a.begin(),
                         a.begin() + n / 2,
                         a.end());
        return (double)a[n / 2];
    }
}

int main(void)
{
  // Task 1a//
  double *x, *y, *z, *gpu_x, *gpu_y, *gpu_z;
  double eff_BW;
  Timer timer;
  int N = pow(10.0, 8.0);
  std::vector<int> k_values(64, 0);
```

```
48  for(int i = 0; i<64; i++){
49    k_values[i] = i;
50  }
51  std :: vector<double> exec_timings = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
52  x = (double*)malloc(N*sizeof(double));
53  y = (double*)malloc(N*sizeof(double));
54  z = (double*)malloc(N*sizeof(double));
55  for (int i = 0; i < N; i++) {
56    x[i] = (double)(i);
57    y[i] = (double)(N-i-1);
58  }
59  cudaMalloc(&gpu_x, N*sizeof(double));
60  cudaMalloc(&gpu_y, N*sizeof(double));
61  cudaMalloc(&gpu_z, N*sizeof(double));
62  cudaMemcpy(gpu_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
63  cudaMemcpy(gpu_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
64  cudaMemcpy(z, gpu_z, N*sizeof(double), cudaMemcpyDeviceToHost);
65
66  for (int i = 0; i < 64; i++) {
67    for (int m = 0; m < 11; m++) {
68      timer.reset();
69      addVec_kth<<<256, 256>>>(gpu_x, gpu_y, gpu_z, N, k_values[i]);
70      cudaDeviceSynchronize();
71      exec_timings[m] = timer.get();
72    }
73    if (k_values[i]==0) {
74      eff_BW = 3 * N * sizeof(double) * pow(10,-9) / findMedian(exec_timings, 10);
75    }
76    else{
77      eff_BW = 3 * floor((N/k_values[i])) * sizeof(double) * pow(10, -9) / findMedian(exec_timings, 10);
78    }
79    printf ("%d,%g\n", k_values[i], eff_BW);
80  }
81
82  cudaFree(gpu_x);
83  cudaFree(gpu_y);
84  cudaFree(gpu_z);
85  free(x);
86  free(y);
87  free(z);
```

B CPP CUDA Code - Offset Memory Access - Task 1b

The code for the Offset Memory Access partial exercise is the same as for the Strided Memory Access partial exercise, except the `__global__` part where the offset is defined and the calculation of the effective bandwidth, where one can now also omit the case distinction for k=0.

C++ Listing for EX1 b)

```
1  __global__ void addVec_kth(double *x, double *y, double *z, int N, int k) {
2    unsigned int total_threads = blockDim.x * gridDim.x;
3    unsigned int global_tid = blockIdx.x * blockDim.x + threadIdx.x;
4    if (k==0) {
5      k = 1;
6    }
7    for (unsigned int i = global_tid ; i<N-k; i += total_threads) {
8      z[i+k] = x[i+k] + y[i+k];
9    }
10 }
11
12 .
13 .
14 .
15
16 eff_BW = 3 * floor((N - k_values[i])) * sizeof(double) * pow(10, -9) / findMedian(exec_timings, 10);
17
18 .
19 .
20 .
```

C CPP CUDA Code - Dense Matrix Transpose - Task 2b

C++ Listing for EX2 b and c

```
1  #include <stdio.h>
2  #include <iostream>
3  #include "timer.hpp"
4  #include "cuda_errchk.hpp"   // for error checking of CUDA calls
5
6  __global__
7  void transpose(double *A, double *B, int N)
8  {
9    int t_idx = blockIdx.x*blockDim.x + threadIdx.x;
10   int row_idx = t_idx / N;
11   int col_idx = t_idx % N;
12
13   if (row_idx < N && col_idx < N) B[row_idx * N + col_idx] = A[col_idx * N + row_idx];
14 }
15
16
17 void print_A(double *A, int N)
18 {
19   for (int i = 0; i < N; i++) {
20     for (int j = 0; j < N; ++j) {
21       std::cout << A[i * N + j] << ", ";
22     }
23     std::cout << std::endl;
24   }
25 }
26
27 int main(void)
28 {
29   int N = 10;
30
31   double *A, *cuda_A, *B, *cuda_B;
32   Timer timer;
33
34   // Allocate host memory and initialize
35   A = (double*)malloc(N*N*sizeof(double));
36   B = (double*)malloc(N*N*sizeof(double));
37
38   for (int i = 0; i < N*N; i++) {
39     A[i] = i;
40   }
41
42   print_A(A, N);
43
44
45   // Allocate device memory and copy host data over
46   CUDA_ERRCHK(cudaMalloc(&cuda_A, N*N*sizeof(double)));
47   CUDA_ERRCHK(cudaMalloc(&cuda_B, N*N*sizeof(double)));
```

```
48
49    // copy data over
50    CUDA_ERRCHK(cudaMemcpy(cuda_A, A, N*N*sizeof(double), cudaMemcpyHostToDevice));
51
52    // wait for previous operations to finish, then start timings
53    CUDA_ERRCHK(cudaDeviceSynchronize());
54    timer.reset();
55
56    // Perform the transpose operation
57    transpose<<<(N*N+255)/256, 256>>>(cuda_A, cuda_B, N);
58
59    // wait for kernel to finish, then print elapsed time
60    CUDA_ERRCHK(cudaDeviceSynchronize());
61    double elapsed = timer.get();
62    std::cout << std::endl << "Time for transpose: " << elapsed << std::endl;
63    std::cout << "Effective bandwidth: " << (2*N*N*sizeof(double)) / elapsed * 1e-9 << " GB/sec" << std::
          endl;
64    std::cout << std::endl;
65
66    // copy data back (implicit synchronization point)
67    CUDA_ERRCHK(cudaMemcpy(B, cuda_B, N*N*sizeof(double), cudaMemcpyDeviceToHost));
68
69    print_A(B, N);
70
71    cudaFree(cuda_A);
72    cudaFree(cuda_B);
73    free(A);
74    free(B);
75
76    CUDA_ERRCHK(cudaDeviceReset()); // for CUDA leak checker to work
77
78    return EXIT_SUCCESS;
79 }
```