



INSTITUTE OF MICROELECTRONICS

360.252 COMPUTATIONAL SCIENCE ON MANY-CORE ARCHITECTURES

Exercise 9

Author:

Peter HOLZNER, 01426733

Submission: January 12, 2021

Contents

1	OpenMP	1
2	Transpose	2
2.1	Fixing my friend's kernel	2
2.2	Blockwise in place transposition	3
2.3	Naive vs Blockwise transposition - Results	5
3	Code and Kernels	6

1 OpenMP

Code listings for this task:

- OpenMP benchmark: [Listing 1](#)

My results weren't too promising. In [Fig. 1](#), the runtime of each version is plotted on the left hand side. On the right side, the speedup of the OpenCL and CUDA versions¹ is shown with the OMP version acting as the reference. As you can clearly see, the OMP version is slower by roughly 1 to 2 orders of magnitude.

OMP was quite easy and comfortable to use - except for one detail - though compared to OpenCL or even basic CUDA (custom kernels and not libraries), at least for this simple program. Copying data to the device was a bit cumbersome when working with `std::vector` as a container. I had to explicitly declare and initialize a pointer to the underlying array to be able to use it in the OMP-for-loop (the `map` directive would not accept anything else). OMP relies on precompiler directives so there are some limitations.

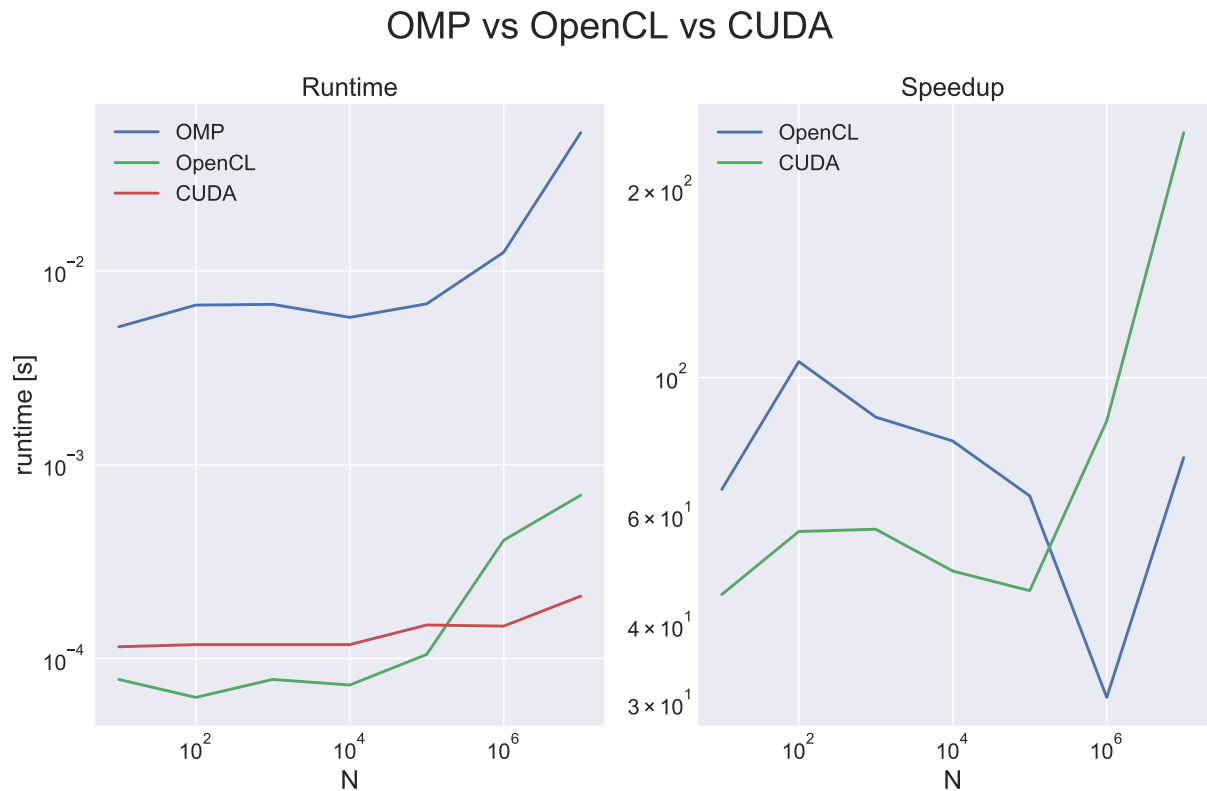


Fig. 1. Runtimes of the OMP dot product compared to my custom OpenCL and CUDA kernels on the left hand side. The right hand side shows the speedup of the OpenCL and CUDA versions compared to the OMP version.

¹Runtimes for these two taken from exercise 8.

2 Transpose

- OpenMP benchmark: [Listing 2](#)

blabla

2.1 Fixing my friend's kernel

My friend made a couple mistakes. First, he forgot to deallocate (`free()`/`cudaFree()`) his memory - that is how you make kitties cry! -, but that happens and is what `cuda-memcheck` (and `valgrind`) is for. Second, he set the `grid_size` incorrectly. Third, I thought that his kernel shouldn't work at all due to how he programmed the switching of the values, but it does and I don't understand why. Either way, my "fixed" version performs similarly in terms of runtime and works correctly, too.

```
1 // missing deallocations
2 free(A);
3 cudaFree(cuda_A);
4
5 // wrong grid size
6 uint grid_size_old = (N+BLOCK_SIZE)/BLOCK_SIZE;
7 uint grid_size = (N*N+BLOCK_SIZE)/BLOCK_SIZE;
8
9 /**My friend's kernel
10 */
11 __global__
12 void transpose_original(double *A, uint N)
13 {
14     uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
15     uint row_idx = t_idx / N;
16     uint col_idx = t_idx % N;
17
18     if (row_idx < N && col_idx < N)
19         A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
20 }
21
22 /** "fixed"
23 */
24 __global__
25 void transpose(double *A, uint N)
26 {
27     uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
28     uint row_idx = t_idx / N;
29     uint col_idx = t_idx % N;
30
31     if (row_idx < N && col_idx < N
32         && col_idx < row_idx
33         && t_idx < N*N)
34     {
35         double tmp = A[row_idx * N + col_idx];
36         A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
37         A[col_idx * N + row_idx] = tmp;
38     }
39 }
```

2.2 Blockwise in place transposition

References used for this part of the exercise:

- <https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/>
- <https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/>
- <https://forums.developer.nvidia.com/t/efficient-in-place-transpose-of-multiple-square-fl/34327/4>

I will use the term used by NVIDIA to describe the 16x16 blocks and call them tiles. Assumptions:

- Tile size `TILE_DIM = 16`
- Matrices are square of dimensions $N \times N$
- N is a multiple of `TILE_DIM = 16` (benchmarked with $N = 512, 1024, 2048, 4096, 8192$)

My first idea was to spawn thread blocks with 16 threads and have them each work on one row of the tile:

```
1 // TILE_DIM = 16
2 // BLOCK_SIZE = 16
3 // --> each thread will work on 1 row
4 __global__
5 void transpose_blockwise(double *A, uint N)
6 {
7     // Operating on a per Thread-Block basis
8     __shared__ double tile_ur[TILE_DIM][TILE_DIM+1];
9     __shared__ double tile_ll[TILE_DIM][TILE_DIM+1];
10
11     uint x = blockIdx.x * TILE_DIM;
12     uint y = blockIdx.y * TILE_DIM;
13     uint tid = threadIdx.x;
14     uint stride = TILE_DIM / blockDim.x;
15
16     bool is_ondiag = (x == y) ? true : false;
17
18     if (blockIdx.y >= blockIdx.x) {
19         if (is_ondiag)
20         {
21             // copy to shared
22             for (uint j = 0; j < TILE_DIM; j += stride)
23             {
24                 tile_ur[j][tid] = A[(y + tid)*N + j + x];
25             }
26
27             // sync
28             __syncthreads();
29
30             // copy from shared
31
32             for (uint j = 0; j < TILE_DIM; j += stride)
33             {
34                 A[(y + tid)*N + j + x] = tile_ur[threadIdx.x][j];
35             }
36         }
37         else if (!is_ondiag)
38         {
39             // copy to shared
40             for (uint j = 0; j < TILE_DIM; j += stride)
41             {
42                 tile_ur[j][tid] = A[(y + tid)*N + j + x];
43                 tile_ll[j][tid] = A[(x + tid)*N + j + y];
44             }
45
46             // sync
```

```

47     __syncthreads();
48
49     // copy from shared
50
51     for (uint j = 0; j < TILE_DIM; j += stride)
52     {
53         A[(y + tid)*N + j + x] = tile_ll[threadIdx.x][j];
54         A[(x + tid)*N + j + y] = tile_ur[threadIdx.x][j];
55     }
56 }
57 }
58 }

```

Unfortunately, that doesn't work very well (=slow) due to the small number of threads in each block. In fact, this version is even slower than the naive approach. My next step would've been to spawn more threads per block (say 32) and have 2 working on one row of the tile - first steps in that direction are the index stride in the for loops above.

However, why reinvent the wheel? Especially, when someone else already has made a better version anyway. I came across a NVIDIA Developer Blog entry (see references above) , that talked about just this problem, and an entry on the forums that talked about an in-place version. The optimized (coalesced) block transpose algorithm is shown below. I adapted it slightly, so it is more easily comparable to my own (first) version above - as you can see below it's essentially almost the same. Here, memory accesses are properly coalesced for each warp so they can be optimized by the device.

```

1  /**coalesced block transpose
2  *
3  * I slightly reworked their kernel to better compare it to my own version
4  * above.
5  * I did it mainly to understand their kernel and I thought it would
6  * also interesting for the documentation to illustrate why
7  * their approach is better/faster/stronger (Daft Punk would be proud).
8  *
9  * Author: Robert Crovella (Nvidia Developer Forum Moderator)
10 *
11 * Reference:
12 * - https://forums.developer.nvidia.com/t/efficient-in-place-transpose-of-multiple-square-float-matrices/34327/4
13 */
14 __global__ void iptransposeCoalesced(double *data, uint N)
15 {
16     // TILE_DIM = 16
17     // BLOCK_ROWS_NV = 8
18     // --> each thread will work on 2 rows
19     __shared__ double tile_s[TILE_DIM][TILE_DIM+1];
20     // padding to avoid bank conflicts... Why not pad by (TILE_DIM+1) instead of just (+1)?
21     // Bank size for our device should be 32 x 32bit.
22     // We use double precision, so 64bit, floats --> 16 x 64 = 32 x 32
23     // We good.
24     __shared__ double tile_d[TILE_DIM][TILE_DIM+1];
25
26     uint x = blockIdx.x * TILE_DIM + threadIdx.x;
27     uint y = blockIdx.y * TILE_DIM + threadIdx.y;
28     //uint width = gridDim.x * TILE_DIM; // you can calculate N this...its just N
29
30     if (blockIdx.y == blockIdx.x){ // handle on-diagonal case
31         for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
32         {
33             // tile_ur[j][tid] = A[(y + tid)*N + j + x]; // <-- mine
34             tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
35         }
36
37         __syncthreads();
38
39         for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
40         {
41             data[(y+j)*N + x] = tile_s[threadIdx.x][threadIdx.y + j];
42         }

```

```

43 }
44 else if (blockIdx.y > blockIdx.x) { // handle off-diagonal case
45     uint dx = blockIdx.y * TILE_DIM + threadIdx.x;
46     uint dy = blockIdx.x * TILE_DIM + threadIdx.y;
47     for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
48     {
49         tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
50     }
51     // for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV) // Idk why they did it with 2
52     // loops instead of one?
53     // {
54     tile_d[threadIdx.y+j][threadIdx.x] = data[(dy+j)*N + dx];
55     }
56     __syncthreads();
57     for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
58         data[(dy+j)*N + dx] = tile_s[threadIdx.x][threadIdx.y + j];
59     for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
60         data[(y+j)*N + x] = tile_d[threadIdx.x][threadIdx.y + j];
61 }
62 }

```

2.3 Naive vs Blockwise transposition - Results

In Fig. 2 the runtime (left) and effective bandwidth of each of the 3 kernel versions is shown. The first version ("my_block") of the blockwise transposition is worse than the naive one for all but the largest two matrices - the approach isn't so bad, I guess. I added the case $N = 8192$, since I noticed that "my_block" was actually performing better than the naive version for $N = 4096$ and wanted to see if the trend continued.

The improved version ("nv_block") is the fastest one across the board.

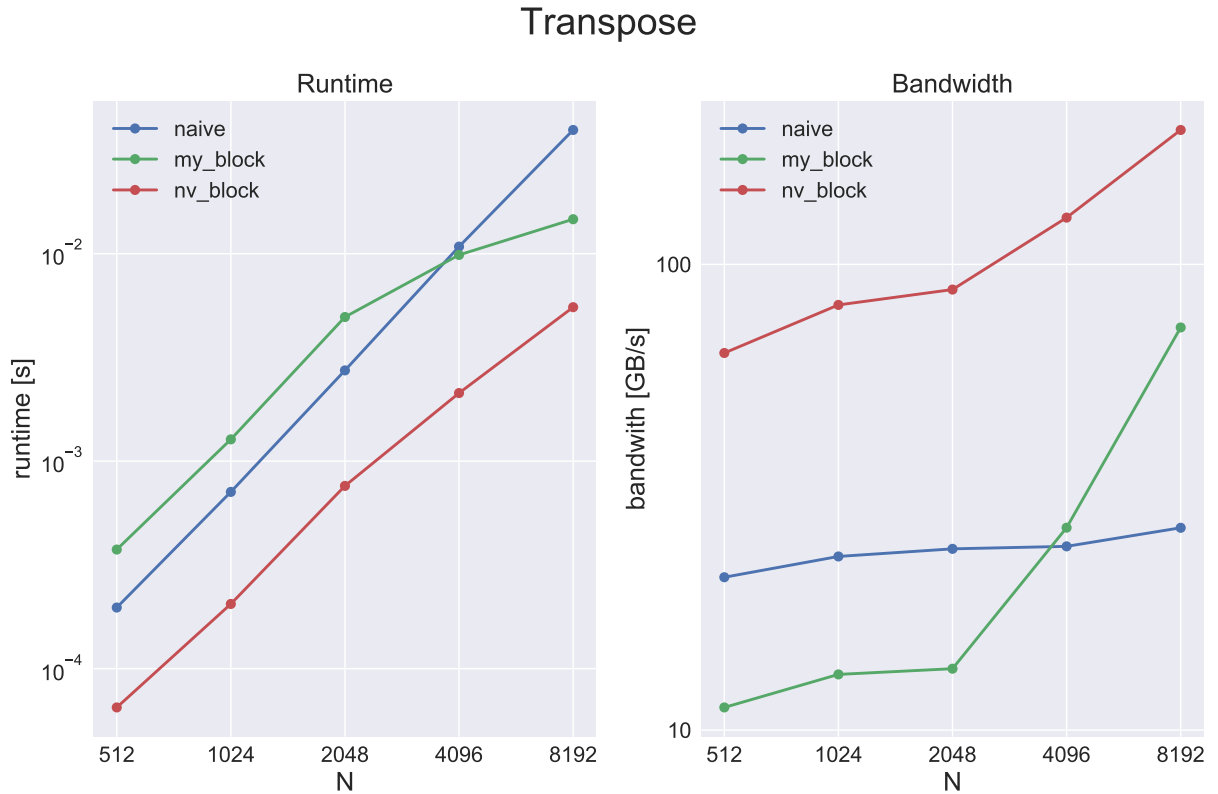


Fig. 2. Benchmark.

3 Code and Kernels

Listings

1	Ex9: OpenMP benchmark	7
2	Ex9: CUDA transpose	8

Listing 1: Ex9: OpenMP benchmark

```

1  #include <omp.h>
2  #include <cmath>
3  #include <algorithm>
4  #include <iostream>
5  #include <fstream>
6  #include <vector>
7  #include <string>
8  #include "timer.hpp"
9
10 // DEFINES
11 #define EX "ex9"
12 #define CSV_NAME "ph_data_omp.csv"
13 #define COUNT
14 #define NUM_TEST 5
15
16 #define X_VAL 1
17 #define Y_VAL 2
18
19 using size_t = std::size_t;
20
21 template <template <typename, typename> class Container,
22          typename ValueType,
23          typename Allocator = std::allocator<ValueType>>
24 double median(Container<ValueType, Allocator> data)
25 {
26     size_t size = data.size();
27     if (size == 0)
28         return 0.;
29     std::sort(data.begin(), data.end());
30     size_t mid = size / 2;
31
32     return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
33 };
34
35 void init_csv(std::ofstream &csv){
36     std::string header = "N;k;runtime";
37     csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
38     csv << header << std::endl;
39 }
40
41 int main()
42 {
43     Timer timer;
44     std::vector<size_t> Ks{1, 2, 3, 4, 5, 6, 7}; // pythonic way to do it... but this "
45         program" is really "scripty" anyway...
46     std::vector<size_t> Ns(Ks.size());
47     size_t N_max = std::pow(10, Ks.back());
48
49     std::vector<double> runtimes(Ks.size(), 0);
50
51     // Init vectors once --> okay here and saves a bit of (unnecessary) time
52     std::vector<double> x(N_max, X_VAL);
53     std::vector<double> y(N_max, Y_VAL);
54     // since map doesn't really like std::vector, we have to get the
55     // underlying data pointers. Although, there seems to be a more C++-style
56     // way via iterators (did not investigate that any further).
57     double *xp = x.data();
58     double *yp = y.data();
59
60     size_t cnt = 0;
61     for (auto &k : Ks)
62     {
63         size_t N = std::pow(10, k); // I know it's slow, but eh...
64         Ns[cnt] = N;
65         std::cout << "k: " << k << " --> N = " << N << std::endl;
66
67         double result = 0;
68         double expected = (double)N * double(X_VAL + Y_VAL) * double(X_VAL - Y_VAL);
69         std::vector<double> tmp(NUM_TEST, 0.0);
70         for (int iter = 0; iter < NUM_TEST; iter++)

```

```

70     {
71         result = 0;
72         timer.reset();
73         // >distribute work on "GPU"< >as parallel for<
74         #pragma omp target teams distribute parallel for \
75             map(to: xp [0:N], yp [0:N]) \
76             map(tofrom: result) \
77             reduction(+: result)
78             for (size_t i = 0; i < N; ++i)
79             {
80                 result += (xp[i] + yp[i]) * (xp[i] - yp[i]);
81             }
82             tmp[iter] = timer.get();
83     }
84     double runtime = median(tmp);
85     runtimes[cnt] = runtime;
86
87     std::string check = (result == expected) ? "Y" : "N";
88     std::cout << "Reduction result: " << result << std::endl;
89     std::cout << "Expected result: " << expected << " [" << check << "]" << std::endl;
90     std::cout << "Runtime: " << runtime << std::endl;
91     cnt++;
92 }
93
94 std::cout << "----- SUMMARY -----" << std::endl;
95
96 std::cout << "Runtimes: " << std::endl;
97 for (auto &rt: runtimes)
98     std::cout << rt << std::endl;
99
100 std::ofstream csv;
101 init_csv(csv);
102 std::string sep = ";";
103 for (size_t idx = 0; idx < Ks.size(); ++idx)
104 {
105     csv << Ns[idx] << sep
106         << Ks[idx] << sep
107         << runtimes[idx] << std::endl;
108 }
109 csv.close();
110
111 std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
112
113 return EXIT_SUCCESS;
114 }

```

Listing 2: Ex9: CUDA transpose

```

1  #include <stdio.h>
2  #include <algorithm>
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  #include <vector>
7  #include <numeric>
8  #include <cmath>
9  #include "timer.hpp"
10
11 // DEFINES
12 #define EX "ex9"
13 #define CSV_NAME "ph_data_transpose.csv"
14 #define N_MAX_PRINT 32
15 #define PRINT_ONLY 10
16 #define NUM_TESTS 9 // should be uneven so we dont have to copy after each iteration
17
18
19 #define TILE_DIM 16
20 #define BLOCK_ROWS 4
21 #define BLOCK_ROWS_NV 8
22 #define BLOCK_SIZE 256
23
24

```

```

25 // ----- HELPERS ----- //
26
27 using size_t = std::size_t;
28
29 template <template <typename, typename> class Container,
30         typename ValueType,
31         typename Allocator = std::allocator<ValueType>>
32 double median(Container<ValueType, Allocator> data)
33 {
34     size_t size = data.size();
35     if (size == 0)
36         return 0.;
37     std::sort(data.begin(), data.end());
38     size_t mid = size / 2;
39
40     return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
41 };
42
43 void print_analysis(double elapsed, uint N)
44 {
45     const uint mem_size = N*N*sizeof(double); // of A
46     std::cout << std::endl << "Time for transpose: " << elapsed << std::endl;
47     std::cout << "Effective bandwidth: " << (2*mem_size) / elapsed * 1e-9 << " GB/sec" << std
48         ::endl;
49     std::cout << std::endl;
50 }
51
52 void print_A(double *A, uint N)
53 {
54     const uint w = 1 + (int)std::log10(N*N);
55     // std::cout << "width: " << w << std::endl;
56     // << std::setfill(' ')
57     for (uint i = 0; i < N; i++) {
58         for (uint j = 0; j < N; ++j) {
59             std::cout << std::setw(w) << A[i * N + j] << ", ";
60         }
61         std::cout << std::endl;
62     }
63 }
64 // ----- KERNELS ----- //
65
66 __global__
67 void transpose(double *A, uint N)
68 {
69     uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
70     uint row_idx = t_idx / N;
71     uint col_idx = t_idx % N;
72
73     if (row_idx < N && col_idx < N
74         && col_idx < row_idx
75         && t_idx < N*N)
76     {
77         double tmp = A[row_idx * N + col_idx];
78         A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
79         A[col_idx * N + row_idx] = tmp;
80     }
81 }
82
83 __global__
84 void transpose_original(double *A, uint N)
85 {
86     uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
87     uint row_idx = t_idx / N;
88     uint col_idx = t_idx % N;
89
90     if (row_idx < N && col_idx < N)
91         A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
92 }
93
94 /** blocked transpose
95  *

```

```

96  * I used the given reference as a base and modified it to be in place
97  * by adding a second shared memory block.
98  *
99  * Reference:
100 * - https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/
101 */
102 __global__
103 void transpose_blockwise(double *A, uint N)
104 {
105     // Operating on a per Thread-Block basis
106     __shared__ double tile_ur[TILE_DIM][TILE_DIM+1];
107     __shared__ double tile_ll[TILE_DIM][TILE_DIM+1];
108
109     uint x = blockIdx.x * TILE_DIM;
110     uint y = blockIdx.y * TILE_DIM;
111     uint tid = threadIdx.x;
112     uint stride = TILE_DIM / blockDim.x;
113
114     bool is_ondiag = (x == y) ? true : false;
115
116     if (blockIdx.y >= blockIdx.x) {
117         if (is_ondiag)
118         {
119             // copy to shared
120             for (uint j = 0; j < TILE_DIM; j += stride)
121             {
122                 tile_ur[j][tid] = A[(y + tid)*N + j + x];
123             }
124
125             // sync
126             __syncthreads();
127
128             // copy from shared
129
130             for (uint j = 0; j < TILE_DIM; j += stride)
131             {
132                 A[(y + tid)*N + j + x] = tile_ur[threadIdx.x][j];
133             }
134         }
135         else if (!is_ondiag)
136         {
137             // copy to shared
138             for (uint j = 0; j < TILE_DIM; j += stride)
139             {
140                 tile_ur[j][tid] = A[(y + tid)*N + j + x];
141                 tile_ll[j][tid] = A[(x + tid)*N + j + y];
142             }
143
144             // sync
145             __syncthreads();
146
147             // copy from shared
148
149             for (uint j = 0; j < TILE_DIM; j += stride)
150             {
151                 A[(y + tid)*N + j + x] = tile_ll[threadIdx.x][j];
152                 A[(x + tid)*N + j + y] = tile_ur[threadIdx.x][j];
153             }
154         }
155     }
156 }
157
158 /**coalesced block transpose
159 *
160 * I slightly reworked their kernel to better compare it to my own version
161 * above.
162 * I did it mainly to understand their kernel and I thought it would
163 * also be interesting for the documentation to illustrate why
164 * their approach is better/faster/stronger (Daft Punk would be proud).
165 *
166 * Author: Robert Crovella (Nvidia Developer Forum Moderator)
167 */

```

```

168 * Reference:
169 * - https://forums.developer.nvidia.com/t/efficient-in-place-transpose-of-multiple-square-float-matrices/34327/4
170 */
171 __global__ void iptransposeCoalesced(double *data, uint N)
172 {
173     // TILE_DIM = 16
174     // BLOCK_ROWS_NV = 8
175     // --> each thread will work on 2 rows
176     __shared__ double tile_s[TILE_DIM][TILE_DIM+1]; // padding to avoid bank conflicts... did
177     __shared__ double tile_d[TILE_DIM][TILE_DIM+1];
178
179     uint x = blockIdx.x * TILE_DIM + threadIdx.x;
180     uint y = blockIdx.y * TILE_DIM + threadIdx.y;
181     //uint width = gridDim.x * TILE_DIM; // you can calculate N this...its just N
182
183     if (blockIdx.y == blockIdx.x){ // handle on-diagonal case
184         for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
185         {
186             // tile_ur[j][tid] = A[(y + tid)*N + j + x]; // <-- mine
187             tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
188         }
189
190         __syncthreads();
191
192         for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
193         {
194             data[(y+j)*N + x] = tile_s[threadIdx.x][threadIdx.y + j];
195         }
196     }
197     else if (blockIdx.y > blockIdx.x) { // handle off-diagonal case
198         uint dx = blockIdx.y * TILE_DIM + threadIdx.x;
199         uint dy = blockIdx.x * TILE_DIM + threadIdx.y;
200         for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
201         {
202             tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
203             // }
204             // for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV) // Idk why they did it with 2
205             // loops instead of one?
206             // {
207             tile_d[threadIdx.y+j][threadIdx.x] = data[(dy+j)*N + dx];
208             // }
209             __syncthreads();
210
211             for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
212                 data[(dy+j)*N + dx] = tile_s[threadIdx.x][threadIdx.y + j];
213             for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
214                 data[(y+j)*N + x] = tile_d[threadIdx.x][threadIdx.y + j];
215         }
216     }
217     /** original kernel by nvidia (for reference)
218     */
219     __global__ void iptransposeCoalesced_nvidia(double *data)
220     {
221         __shared__ double tile_s[TILE_DIM][TILE_DIM+1];
222         __shared__ double tile_d[TILE_DIM][TILE_DIM+1];
223
224         int x = blockIdx.x * TILE_DIM + threadIdx.x;
225         int y = blockIdx.y * TILE_DIM + threadIdx.y;
226         int width = gridDim.x * TILE_DIM; // you can calculate N this...its just N
227
228         if (blockIdx.y > blockIdx.x) { // handle off-diagonal case
229             int dx = blockIdx.y * TILE_DIM + threadIdx.x;
230             int dy = blockIdx.x * TILE_DIM + threadIdx.y;
231             for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
232                 tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*width + x];
233             for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
234                 tile_d[threadIdx.y+j][threadIdx.x] = data[(dy+j)*width + dx];
235             __syncthreads();
236             for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)

```

```

237     data[(dy+j)*width + dx] = tile_s[threadIdx.x][threadIdx.y + j];
238     for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
239         data[(y+j)*width + x] = tile_d[threadIdx.x][threadIdx.y + j];
240 }
241
242 else if (blockIdx.y == blockIdx.x){ // handle on-diagonal case
243     for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
244         tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*width + x];
245     __syncthreads();
246     for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
247         data[(y+j)*width + x] = tile_s[threadIdx.x][threadIdx.y + j];
248 }
249 }
250
251 int main(void)
252 {
253     //
254     // ----- SETUP ----- //
255     //
256     // 512, 1024, 2048, 4096
257     uint N = 32;
258     // (x, y, z)
259     dim3 dimGrid(N/TILE_DIM, N/TILE_DIM, 1);
260     dim3 dimBlock(TILE_DIM, 1, 1);
261
262     dim3 dimGrid_nv(N/TILE_DIM, N/TILE_DIM, 1);
263     dim3 dimBlock_nv(TILE_DIM, BLOCK_ROWS_NV, 1);
264
265     double *A, *A2, *A3;
266     double *cuda_A;
267     Timer timer;
268
269     // Allocate host memory and initialize
270     A = (double*)malloc(N*N*sizeof(double));
271     A2 = (double*)malloc(N*N*sizeof(double));
272     A3 = (double*)malloc(N*N*sizeof(double));
273
274     for (uint i = 0; i < N*N; i++) {
275         A[i] = i;
276         A2[i] = i;
277         A3[i] = i;
278     }
279
280     if (N <= N_MAX_PRINT) {
281         print_A(A, N);
282         // print_A(A2, N); // A2 looks the same...
283     }
284
285     // Allocate device memory and copy host data over
286     cudaMalloc(&cuda_A, N*N*sizeof(double));
287
288     //
289     // ----- BENCHMARKS ----- //
290     //
291
292     // ----- Naive Transpose ----- //
293     // copy data over
294     cudaMemcpy(cuda_A, A, N*N*sizeof(double), cudaMemcpyHostToDevice);
295     // wait for previous operations to finish, then start timings
296     cudaDeviceSynchronize();
297
298     uint grid_size = (N*N+BLOCK_SIZE)/BLOCK_SIZE;
299
300     std::vector<double> tmp(NUM_TESTS, 0);
301     for (uint iter = 0; iter < NUM_TESTS; ++iter){
302         timer.reset();
303         // Perform the transpose operation
304         // # blocks, threads per block
305         transpose<<<grid_size, BLOCK_SIZE>>>(cuda_A, N);
306         // transpose_original<<<grid_size, BLOCK_SIZE>>>(cuda_A, N);
307
308         // wait for kernel to finish, then print elapsed time

```

```

309     cudaDeviceSynchronize();
310     tmp[iter] = timer.get();
311 }
312 double elapsed = median(tmp);
313 // copy data back (implicit synchronization point)
314 cudaMemcpy(A, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost);
315 cudaDeviceSynchronize();
316
317 // ----- Block Transpose ----- //
318 // copy data over
319 cudaMemcpy(cuda_A, A2, N*N*sizeof(double), cudaMemcpyHostToDevice);
320 // wait for previous operations to finish, then start timings
321 cudaDeviceSynchronize();
322 for (uint iter = 0; iter < NUM_TESTS; ++iter){
323     timer.reset();
324     // Perform the transpose operation
325     // # blocks , threads per block
326     transpose_blockwise<<<dimGrid, dimBlock>>>(cuda_A, N);
327
328     // wait for kernel to finish, then print elapsed time
329     cudaDeviceSynchronize();
330     tmp[iter] = timer.get();
331 }
332 double elapsed_block = median(tmp);
333 // copy data back (implicit synchronization point)
334 cudaMemcpy(A2, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost);
335 cudaDeviceSynchronize();
336
337 // ----- NVIDIA Block Transpose ----- //
338 // copy data over
339 cudaMemcpy(cuda_A, A3, N*N*sizeof(double), cudaMemcpyHostToDevice);
340 // wait for previous operations to finish, then start timings
341 cudaDeviceSynchronize();
342 for (uint iter = 0; iter < NUM_TESTS; ++iter){
343     timer.reset();
344     // Perform the transpose operation
345     // # blocks , threads per block
346     iptransposeCoalesced<<<dimGrid_nv, dimBlock_nv>>>(cuda_A, N);
347
348     // wait for kernel to finish, then print elapsed time
349     cudaDeviceSynchronize();
350     tmp[iter] = timer.get();
351 }
352 double elapsed_block_nv = median(tmp);
353 // copy data back (implicit synchronization point)
354 cudaMemcpy(A3, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost);
355 cudaDeviceSynchronize();
356
357 //
358 // ----- OUTPUT ----- //
359 //
360
361 std::cout << "---- Naive Transpose ----" << std::endl;
362 print_analysis(elapsed, N);
363 std::cout << "---- Block Transpose ----" << std::endl;
364 print_analysis(elapsed_block, N);
365 std::cout << "---- NVIDIA Block Transpose ----" << std::endl;
366 print_analysis(elapsed_block_nv, N);
367
368 if (N <= N_MAX_PRINT) {
369     std::cout << std::endl;
370     std::cout << "---- Naive Transpose ----" << std::endl;
371     print_A(A, N);
372     std::cout << "---- Block Transpose ----" << std::endl;
373     print_A(A2, N);
374     std::cout << "---- NVIDIA Block Transpose ----" << std::endl;
375     print_A(A3, N);
376 }
377
378 std::cout << "---- Parameters: ----" << std::endl;
379 std::cout << "-- Naive";
380 std::cout << "grid_size: " << grid_size << std::endl;

```

```

381     std::cout << "BLOCK_SIZE: " << BLOCK_SIZE << std::endl;
382     std::cout << "-- My Block";
383     std::cout << "TILE_DIM: " << TILE_DIM << std::endl;
384     std::cout << "BLOCK_ROWS: " << BLOCK_ROWS << std::endl;
385     std::cout << "dimGrid: (" << dimGrid.x << ", " << dimGrid.y << ", " << dimGrid.z << ")" <<
        std::endl;
386     std::cout << "dimBlock: (" << dimBlock.x << ", " << dimBlock.y << ", " << dimBlock.z << ")"
        << std::endl;
387     std::cout << "-- NVIDIA Block";
388     std::cout << "BLOCK_ROWS_NV: " << BLOCK_ROWS_NV << std::endl;
389     std::cout << "dimGrid_nv: (" << dimGrid_nv.x << ", " << dimGrid_nv.y << ", " << dimGrid_nv.z
        << ")" << std::endl;
390     std::cout << "dimBlock_nv: (" << dimBlock_nv.x << ", " << dimBlock_nv.y << ", " <<
        dimBlock_nv.z << ")" << std::endl;
391
392     std::ofstream csv;
393     std::string header = "N;naive;my_block;nv_block; grid_size;BLOCK_SIZE;TILE_DIM;BLOCK_ROWS;
        dimGrid.x;dimGrid.y;dimBlock.x;dimBlock.y;dimGrid_nv.x;dimGrid_nv.y;dimBlock_nv.x;
        dimBlock_nv.y";
394     if (N == 512)
395     {
396         csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
397         csv << header << std::endl;
398     }
399     else
400     {
401         csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
402     }
403     csv << N << ";";
404     << elapsed << ";";
405     << elapsed_block << ";";
406     << elapsed_block_nv << ";";
407     << grid_size << ";"; << BLOCK_SIZE << ";";
408     << TILE_DIM << ";"; << BLOCK_ROWS << ";";
409     << dimGrid.x << ";"; << dimGrid.y << ";";
410     << dimBlock.x << ";"; << dimBlock.y << ";";
411     << dimGrid_nv.x << ";"; << dimGrid_nv.y << ";";
412     << dimBlock_nv.x << ";"; << dimBlock_nv.y << "\n";
413     csv.close();
414
415     std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;
416
417     // My friend was a bit sloppy and forgot these two lines...
418     free(A);
419     free(A2);
420     free(A3);
421     cudaFree(cuda_A);
422     // Well, happens to the best!
423
424     cudaDeviceReset(); // for CUDA leak checker to work
425
426     return EXIT_SUCCESS;
427 }

```