INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Exercise 9

Author:
Peter Holzner, 01426733

Submission: January 12, 2021

Contents

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 versions[1] 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.

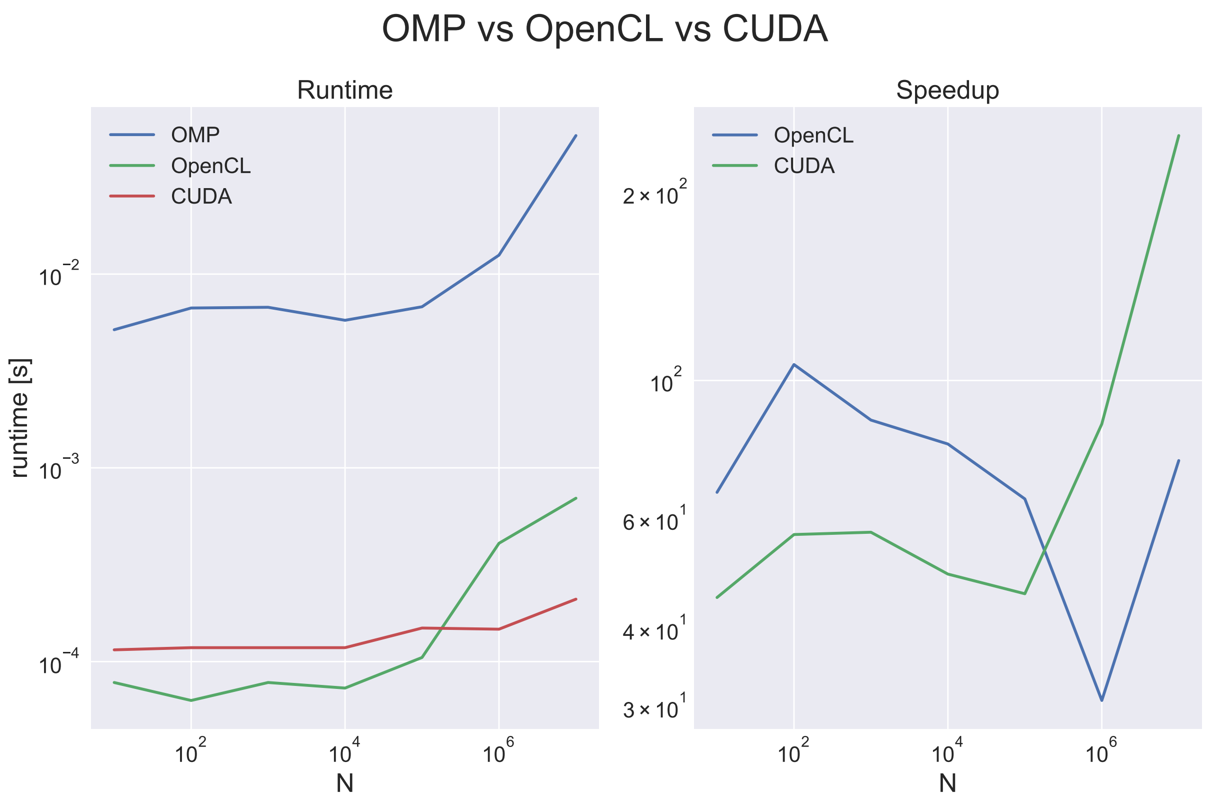

Fig. 1. Runtimes of the OMP dot product compared to my custom OpenCL and CUDA kernels on the left hand side. The right hand side shows the speedup of the OpenCL and CUDA versions compared to the OMP version.

[1]Runtimes 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.

```
// missing deallocations
free(A);
cudaFree(cuda_A);

// wrong grid size
uint grid_size_old = (N+BLOCK_SIZE)/BLOCK_SIZE;
uint grid_size = (N*N+BLOCK_SIZE)/BLOCK_SIZE;

/**My friend's kernel
 */
__global__
void transpose_original(double *A, uint N)
{
  uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  uint row_idx = t_idx / N;
  uint col_idx = t_idx % N;

  if (row_idx < N && col_idx < N)
    A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
}

/** "fixed"
 */
__global__
void transpose(double *A, uint N)
{
  uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  uint row_idx = t_idx / N;
  uint col_idx = t_idx % N;

  if (row_idx < N && col_idx < N
    && col_idx < row_idx
    && t_idx < N*N)
  {
    double tmp = A[row_idx * N + col_idx];
    A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
    A[col_idx * N + row_idx] = tmp;
  }
}
```

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 x 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:

```
// TILE_DIM = 16
// BLOCK_SIZE = 16
// --> each thread will work on 1 row
 __global__
void transpose_blockwise(double *A, uint N)
{
  // Operating on a per Thread-Block basis
  __shared__ double tile_ur[TILE_DIM][TILE_DIM+1];
  __shared__ double tile_ll[TILE_DIM][TILE_DIM+1];

  uint x = blockIdx.x * TILE_DIM;
  uint y = blockIdx.y * TILE_DIM;
  uint tid = threadIdx.x;
  uint stride = TILE_DIM / blockDim.x;

  bool is_ondiag = (x == y) ? true : false;

  if (blockIdx.y >= blockIdx.x) {
    if (is_ondiag)
    {
      // copy to shared
      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        tile_ur[j][tid] = A[(y + tid)*N + j + x];
      }

      // sync
      __syncthreads();

      // copy from shared

      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        A[(y + tid)*N + j + x] = tile_ur[threadIdx.x][j];
      }
    }
    else if (!is_ondiag)
    {
      // copy to shared
      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        tile_ur[j][tid] = A[(y + tid)*N + j + x];
        tile_ll[j][tid] = A[(x + tid)*N + j + y];
      }

      // sync
      __syncthreads();

      // copy from shared

      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        A[(y + tid)*N + j + x] = tile_ll[threadIdx.x][j];
        A[(x + tid)*N + j + y] = tile_ur[threadIdx.x][j];
      }
    }
  }
}
```

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.

```
/**coalesced block transpose
 *
 * I slightly reworked their kernel to better compare it to my own version
 * above.
 * I did it mainly to understand their kernel and I thought it would
 * also interesting for the documentation to illustrate why
 * their approach is better/faster/stronger (Daft Punk would be proud).
 *
 * Author: Robert Crovella (Nvidia Developer Forum Moderator)
 *
 * Reference:
 * - https://forums.developer.nvidia.com/t/efficient-in-place-transpose-of-multiple-square-
    float-matrices/34327/4
 */
__global__ void iptransposeCoalesced(double *data, uint N)
{
  // TILE_DIM = 16
  // BLOCK_ROWS_NV = 8
  // --> each thread will work on 2 rows
  __shared__ double tile_s[TILE_DIM][TILE_DIM+1];
  // padding to avoid bank conflicts... Why not pad by (TILE_DIM+1) instead of just (+1)?
  // Bank size for our device should be 32 x 32bit.
  // We use double precision, so 64bit, floats --> 16 x 64 = 32 x 32
  // We good.
  __shared__ double tile_d[TILE_DIM][TILE_DIM+1];

  uint x = blockIdx.x * TILE_DIM + threadIdx.x;
  uint y = blockIdx.y * TILE_DIM + threadIdx.y;
  //uint width = gridDim.x * TILE_DIM; // you can calculate N this...its just N

  if (blockIdx.y == blockIdx.x){ // handle on-diagonal case
    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
    {
      // tile_ur[j][tid] = A[(y + tid)*N + j + x]; // <-- mine
      tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
    }

    __syncthreads();

    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
    {
      data[(y+j)*N + x] = tile_s[threadIdx.x][threadIdx.y + j];
    }
  }
  else if (blockIdx.y > blockIdx.x) { // handle off-diagonal case
    uint dx = blockIdx.y * TILE_DIM + threadIdx.x;
    uint dy = blockIdx.x * TILE_DIM + threadIdx.y;
    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
    {
      tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
    // }
    // for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV) // Idk why they did it with 2
        loops instead of one?
    // {
      tile_d[threadIdx.y+j][threadIdx.x] = data[(dy+j)*N + dx];
    }
    __syncthreads();

    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      data[(dy+j)*N + dx] = tile_s[threadIdx.x][threadIdx.y + j];
    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      data[(y+j)*N + x] = tile_d[threadIdx.x][threadIdx.y + j];
  }
}
```

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.

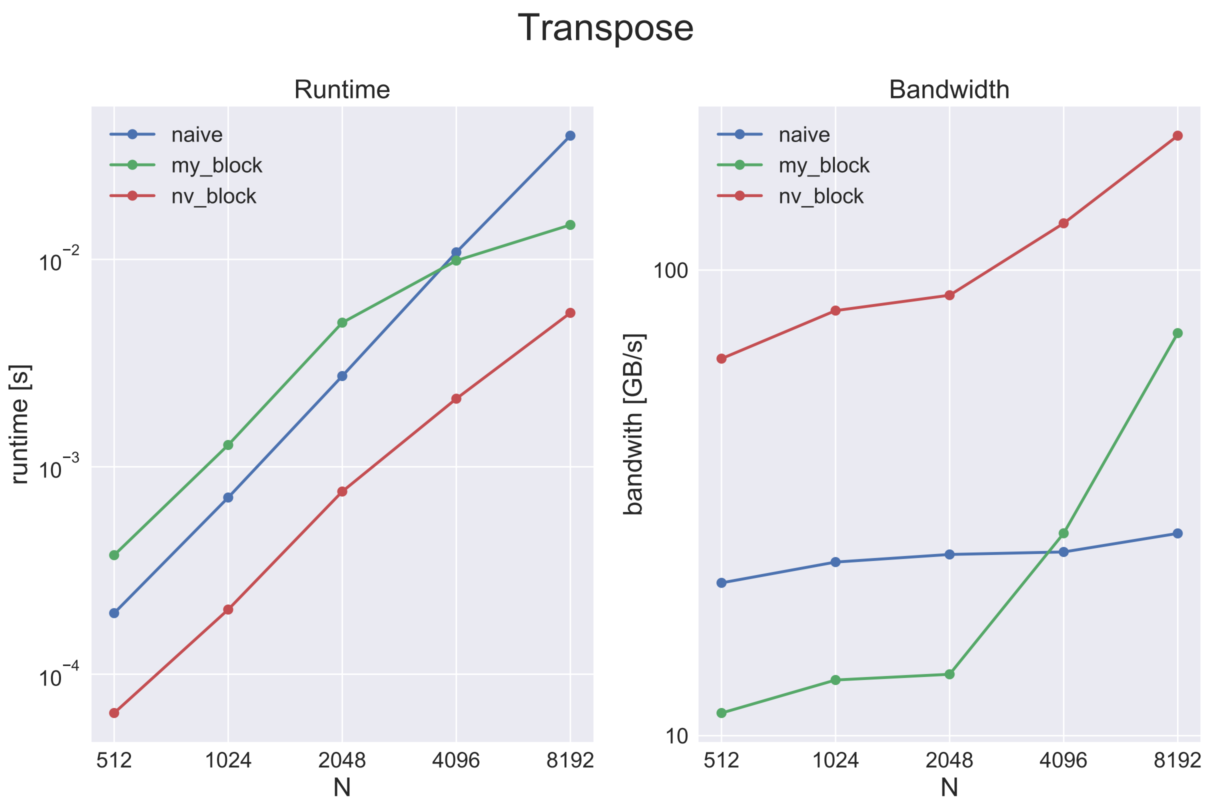

Fig. 2. Benchmark.

3 Code and Kernels

Listings

Listing 1: Ex9: OpenMP benchmark

```cpp
#include <omp.h>
#include <cmath>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include "timer.hpp"

// DEFINES
#define EX "ex9"
#define CSV_NAME "ph_data_omp.csv"
#define COUT
#define NUM_TEST 5

#define X_VAL 1
#define Y_VAL 2

using size_t = std::size_t;

template <template <typename, typename> class Container,
          typename ValueType,
          typename Allocator = std::allocator<ValueType>>
double median(Container<ValueType, Allocator> data)
{
    size_t size = data.size();
    if (size == 0)
        return 0.;
    std::sort(data.begin(), data.end());
    size_t mid = size / 2;

    return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
};

void init_csv(std::ofstream &csv){
    std::string header = "N;k;runtime";
    csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
    csv << header << std::endl;
}

int main()
{
    Timer timer;
    std::vector<size_t> Ks{1, 2, 3, 4, 5, 6, 7}; // pythonic way to do it... but this "
        program" is really "scripty" anyway...
    std::vector<size_t> Ns(Ks.size());
    size_t N_max = std::pow(10, Ks.back());

    std::vector<double> runtimes(Ks.size(), 0);

    // Init vectors once --> okay here and saves a bit of (unnecessary) time
    std::vector<double> x(N_max, X_VAL);
    std::vector<double> y(N_max, Y_VAL);
    // since map doesn't really like std::vector, we have to get the
    // underlying data pointers. Although, there seems to be a more C++-style
    // way via iterators (did not investigate that any further).
    double *xp = x.data();
    double *yp = y.data();

    size_t cnt = 0;
    for (auto &k : Ks)
    {
        size_t N = std::pow(10, k); // I know it's slow, but eh...
        Ns[cnt] = N;
        std::cout << "k: " << k << " --> N = " << N << std::endl;

        double result = 0;
        double expected = (double)N * double(X_VAL + Y_VAL) * double(X_VAL - Y_VAL);
        std::vector<double> tmp(NUM_TEST, 0.0);
        for (int iter = 0; iter < NUM_TEST; iter++)
        {
            result = 0;
            timer.reset();
    //          >distribute work on "GPU"< >as parallel for<
    #pragma omp target teams distribute parallel for \
        map(to: xp [0:N], yp [0:N]) \
        map(tofrom: result) \
        reduction(+: result)
            for (size_t i = 0; i < N; ++i)
            {
                result += (xp[i] + yp[i]) * (xp[i] - yp[i]);
            }
            tmp[iter] = timer.get();
        }
        double runtime = median(tmp);
        runtimes[cnt] = runtime;

        std::string check = (result == expected) ? "Y" : "N";
        std::cout << "Reduction result: " << result << std::endl;
        std::cout << "Expected  result: " << expected << " [" << check << "]" << std::endl;
        std::cout << "Runtime: " << runtime << std::endl;
        cnt++;
    }

    std::cout << "----------- SUMMARY -----------" << std::endl;

    std::cout << "Runtimes: " << std::endl;
    for (auto &rt: runtimes)
        std::cout << rt << std::endl;

    std::ofstream csv;
    init_csv(csv);
    std::string sep = ";";
    for (size_t idx = 0; idx < Ks.size(); ++idx)
    {
        csv << Ns[idx] << sep
            << Ks[idx] << sep
            << runtimes[idx] << std::endl;
    }
    csv.close();

    std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;

    return EXIT_SUCCESS;
}
```

Listing 2: Ex9: CUDA transpose

```
#include <stdio.h>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <cmath>
#include "timer.hpp"

// DEFINES
#define EX "ex9"
#define CSV_NAME "ph_data_transpose.csv"
#define N_MAX_PRINT 32
#define PRINT_ONLY 10
#define NUM_TESTS 9 // should be uneven so we dont have to copy after each iteration


#define TILE_DIM 16
#define BLOCK_ROWS 4
#define BLOCK_ROWS_NV 8
#define BLOCK_SIZE 256


```

```
// ------------- HELPERS ------------- //

using size_t = std::size_t;

template <template <typename, typename> class Container,
          typename ValueType,
          typename Allocator = std::allocator<ValueType>>
double median(Container<ValueType, Allocator> data)
{
    size_t size = data.size();
    if (size == 0)
        return 0.;
    std::sort(data.begin(), data.end());
    size_t mid = size / 2;

    return size % 2 == 0 ? (data[mid] + data[mid - 1]) / 2 : data[mid];
};

void print_analysis(double elapsed, uint N)
{
  const uint mem_size = N*N*sizeof(double); // of A
  std::cout << std::endl << "Time for transpose: " << elapsed << std::endl;
  std::cout << "Effective bandwidth: " << (2*mem_size) / elapsed * 1e-9 << " GB/sec" << std
      ::endl;
  std::cout << std::endl;
}

void print_A(double *A, uint N)
{
  const uint w = 1 + (int)std::log10(N*N);
  // std::cout << "width: " << w << std::endl;
  // << std::setfill(' ')
  for (uint i = 0; i < N; i++) {
    for (uint j = 0; j < N; ++j) {
      std::cout << std::setw(w) << A[i * N + j] << ", ";
    }
    std::cout << std::endl;
  }
}

// ------------- KERNELS ------------- //

__global__
void transpose(double *A, uint N)
{
  uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  uint row_idx = t_idx / N;
  uint col_idx = t_idx % N;

  if (row_idx < N && col_idx < N
    && col_idx < row_idx
    && t_idx < N*N)
  {
    double tmp = A[row_idx * N + col_idx];
    A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
    A[col_idx * N + row_idx] = tmp;
  }
}

__global__
void transpose_original(double *A, uint N)
{
  uint t_idx = blockIdx.x*blockDim.x + threadIdx.x;
  uint row_idx = t_idx / N;
  uint col_idx = t_idx % N;

  if (row_idx < N && col_idx < N)
    A[row_idx * N + col_idx] = A[col_idx * N + row_idx];
}

/** blocked transpose
 *
 * I used the given reference as a base and modified it to be in place
 * by adding a second shared memory block.
 *
 * Reference:
 * - https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/
 */
__global__
void transpose_blockwise(double *A, uint N)
{
  // Operating on a per Thread-Block basis
  __shared__ double tile_ur[TILE_DIM][TILE_DIM+1];
  __shared__ double tile_ll[TILE_DIM][TILE_DIM+1];

  uint x = blockIdx.x * TILE_DIM;
  uint y = blockIdx.y * TILE_DIM;
  uint tid = threadIdx.x;
  uint stride = TILE_DIM / blockDim.x;

  bool is_ondiag = (x == y) ? true : false;

  if (blockIdx.y >= blockIdx.x) {
    if (is_ondiag)
    {
      // copy to shared
      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        tile_ur[j][tid] = A[(y + tid)*N + j + x];
      }

      // sync
      __syncthreads();

      // copy from shared

      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        A[(y + tid)*N + j + x] = tile_ur[threadIdx.x][j];
      }
    }
    else if (!is_ondiag)
    {
      // copy to shared
      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        tile_ur[j][tid] = A[(y + tid)*N + j + x];
        tile_ll[j][tid] = A[(x + tid)*N + j + y];
      }

      // sync
      __syncthreads();

      // copy from shared

      for (uint j = 0; j < TILE_DIM; j += stride)
      {
        A[(y + tid)*N + j + x] = tile_ll[threadIdx.x][j];
        A[(x + tid)*N + j + y] = tile_ur[threadIdx.x][j];
      }
    }
  }
}

/**coalesced block transpose
 *
 * I slightly reworked their kernel to better compare it to my own version
 * above.
 * I did it mainly to understand their kernel and I though it would
 * also interesting for the documentation to illustrate why
 * their approach is better/faster/stronger (Daft Punk would be proud).
 *
 * Author: Robert Crovella (Nvidia Developer Forum Moderator)
 *
 * Reference:
 * - https://forums.developer.nvidia.com/t/efficient-in-place-transpose-of-multiple-square-
    float-matrices/34327/4
 */
__global__ void iptransposeCoalesced(double *data, uint N)
{
  // TILE_DIM = 16
  // BLOCK_ROWS_NV = 8
  // --> each thread will work on 2 rows
  __shared__ double tile_s[TILE_DIM][TILE_DIM+1]; // padding to avoid bank conflicts... did
     not really understand it!
  __shared__ double tile_d[TILE_DIM][TILE_DIM+1];

  uint x = blockIdx.x * TILE_DIM + threadIdx.x;
  uint y = blockIdx.y * TILE_DIM + threadIdx.y;
  //uint width = gridDim.x * TILE_DIM; // you can calculate N this...its just N

  if (blockIdx.y == blockIdx.x){ // handle on-diagonal case
    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
    {
      // tile_ur[j][tid] = A[(y + tid)*N + j + x]; // <-- mine
      tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
    }

    __syncthreads();

    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
    {
      data[(y+j)*N + x] = tile_s[threadIdx.x][threadIdx.y + j];
    }
  }
  else if (blockIdx.y > blockIdx.x) { // handle off-diagonal case
    uint dx = blockIdx.y * TILE_DIM + threadIdx.x;
    uint dy = blockIdx.x * TILE_DIM + threadIdx.y;
    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
    {
      tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*N + x];
    // }
    // for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV) // Idk why they did it with 2
      loops instead of one?
    // {
      tile_d[threadIdx.y+j][threadIdx.x] = data[(dy+j)*N + dx];
    }
    __syncthreads();

    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      data[(dy+j)*N + dx] = tile_s[threadIdx.x][threadIdx.y + j];
    for (uint j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      data[(y+j)*N + x] = tile_d[threadIdx.x][threadIdx.y + j];
  }
}

/** original kernel by nvidia (for reference)
 */
__global__ void iptransposeCoalesced_nvidia(double *data)
{
  __shared__ double tile_s[TILE_DIM][TILE_DIM+1];
  __shared__ double tile_d[TILE_DIM][TILE_DIM+1];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM; // you can calculate N this...its just N

  if (blockIdx.y > blockIdx.x) { // handle off-diagonal case
    int dx = blockIdx.y * TILE_DIM + threadIdx.x;
    int dy = blockIdx.x * TILE_DIM + threadIdx.y;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*width + x];
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      tile_d[threadIdx.y+j][threadIdx.x] = data[(dy+j)*width + dx];
    __syncthreads();
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
```

```
      data[(dy+j)*width + dx] = tile_s[threadIdx.x][threadIdx.y + j];
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      data[(y+j)*width + x] = tile_d[threadIdx.x][threadIdx.y + j];
  }

  else if (blockIdx.y == blockIdx.x){ // handle on-diagonal case
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      tile_s[threadIdx.y+j][threadIdx.x] = data[(y+j)*width + x];
    __syncthreads();
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS_NV)
      data[(y+j)*width + x] = tile_s[threadIdx.x][threadIdx.y + j];
  }
}

int main(void)
{
  //
  // -------------- SETUP -------------- //
  //
  // 512, 1024, 2048, 4096
  uint N = 32;
  //           (x         , y          , z)
  dim3 dimGrid(N/TILE_DIM, N/TILE_DIM, 1);
  dim3 dimBlock(TILE_DIM, 1, 1);

  dim3 dimGrid_nv(N/TILE_DIM, N/TILE_DIM, 1);
  dim3 dimBlock_nv(TILE_DIM, BLOCK_ROWS_NV, 1);

  double *A, *A2, *A3;
  double *cuda_A;
  Timer timer;

  // Allocate host memory and initialize
  A = (double*)malloc(N*N*sizeof(double));
  A2 = (double*)malloc(N*N*sizeof(double));
  A3 = (double*)malloc(N*N*sizeof(double));

  for (uint i = 0; i < N*N; i++) {
    A[i] = i;
    A2[i] = i;
    A3[i] = i;
  }

  if (N <= N_MAX_PRINT) {
    print_A(A, N);
    // print_A(A2, N); // A2 looks the same...
  }

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

  //
  // -------------- BENCHMARKS -------------- //
  //

  // -------------- Naive Transpose -------------- //
  // copy data over
  cudaMemcpy(cuda_A, A, N*N*sizeof(double), cudaMemcpyHostToDevice);
  // wait for previous operations to finish, then start timings
  cudaDeviceSynchronize();

  uint grid_size = (N*N+BLOCK_SIZE)/BLOCK_SIZE;

  std::vector<double> tmp(NUM_TESTS, 0);
  for (uint iter = 0; iter < NUM_TESTS; ++iter){
    timer.reset();
    // Perform the transpose operation
    //            # blocks   , threads per block
    transpose<<<grid_size, BLOCK_SIZE>>>(cuda_A, N);
    // transpose_original<<<grid_size, BLOCK_SIZE>>>(cuda_A, N);

    // wait for kernel to finish, then print elapsed time
    cudaDeviceSynchronize();
    tmp[iter] = timer.get();
  }
  double elapsed = median(tmp);
  // copy data back (implicit synchronization point)
  cudaMemcpy(A, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  // -------------- Block Transpose -------------- //
  // copy data over
  cudaMemcpy(cuda_A, A2, N*N*sizeof(double), cudaMemcpyHostToDevice);
  // wait for previous operations to finish, then start timings
  cudaDeviceSynchronize();
  for (uint iter = 0; iter < NUM_TESTS; ++iter){
    timer.reset();
    // Perform the transpose operation
    //                    # blocks  , threads per block
    transpose_blockwise<<<dimGrid, dimBlock>>>(cuda_A, N);

    // wait for kernel to finish, then print elapsed time
    cudaDeviceSynchronize();
    tmp[iter] = timer.get();
  }
  double elapsed_block = median(tmp);
  // copy data back (implicit synchronization point)
  cudaMemcpy(A2, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  // -------------- NVIDIA Block Transpose -------------- //
  // copy data over
  cudaMemcpy(cuda_A, A3, N*N*sizeof(double), cudaMemcpyHostToDevice);
  // wait for previous operations to finish, then start timings
  cudaDeviceSynchronize();
  for (uint iter = 0; iter < NUM_TESTS; ++iter){
    timer.reset();
    // Perform the transpose operation
    //                    # blocks  , threads per block
    iptransposeCoalesced<<<dimGrid_nv, dimBlock_nv>>>(cuda_A, N);

    // wait for kernel to finish, then print elapsed time
    cudaDeviceSynchronize();
    tmp[iter] = timer.get();
  }
  double elapsed_block_nv = median(tmp);
  // copy data back (implicit synchronization point)
  cudaMemcpy(A3, cuda_A, N*N*sizeof(double), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  //
  // -------------- OUTPUT -------------- //
  //

  std::cout << "---- Naive Transpose ----" << std::endl;
  print_analysis(elapsed, N);
  std::cout << "---- Block Transpose ----" << std::endl;
  print_analysis(elapsed_block, N);
  std::cout << "---- NVIDIA Block Transpose ----" << std::endl;
  print_analysis(elapsed_block_nv, N);

  if (N <= N_MAX_PRINT) {
    std::cout << std::endl;
    std::cout << "---- Naive Transpose ----" << std::endl;
    print_A(A, N);
    std::cout << "---- Block Transpose ----" << std::endl;
    print_A(A2, N);
    std::cout << "---- NVIDIA Block Transpose ----" << std::endl;
    print_A(A3, N);
  }

  std::cout << "---- Parameters: ----" << std::endl;
  std::cout << "-- Naive";
  std::cout << "grid_size: " << grid_size << std::endl;
  std::cout << "BLOCK_SIZE: " << BLOCK_SIZE << std::endl;
  std::cout << "-- My Block";
  std::cout << "TILE_DIM: " << TILE_DIM << std::endl;
  std::cout << "BLOCK_ROWS: " << BLOCK_ROWS << std::endl;
  std::cout << "dimGrid: (" << dimGrid.x << ", "<< dimGrid.y << ", "<< dimGrid.z << ")" <<
      std::endl;
  std::cout << "dimBlock: (" << dimBlock.x << ", "<< dimBlock.y << ", "<< dimBlock.z << ")"
      << std::endl;
  std::cout << "-- NVIDIA Block";
  std::cout << "BLOCK_ROWS_NV: " << BLOCK_ROWS_NV << std::endl;
  std::cout << "dimGrid_nv: (" << dimGrid_nv.x << ", "<< dimGrid_nv.y << ", "<< dimGrid_nv.z
       << ")" << std::endl;
  std::cout << "dimBlock_nv: (" << dimBlock_nv.x << ", "<< dimBlock_nv.y << ", "<<
      dimBlock_nv.z << ")" << std::endl;

  std::ofstream csv;
  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";
  if (N == 512)
  {
    csv.open(CSV_NAME, std::fstream::out | std::fstream::trunc);
    csv << header << std::endl;
  }
  else
  {
    csv.open(CSV_NAME, std::fstream::out | std::fstream::app);
  }
  csv << N << ";"
      << elapsed << ";"
      << elapsed_block << ";"
      << elapsed_block_nv  << ";"
      << 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 << "\n";
  csv.close();

  std::cout << "Data: https://gtx1080.360252.org/2020/" << EX << "/" << CSV_NAME;

  // My friend was a bit sloppy and forgot these two lines...
  free(A);
  free(A2);
  free(A3);
  cudaFree(cuda_A);
  // Well, happens to the best!

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

  return EXIT_SUCCESS;
}
```