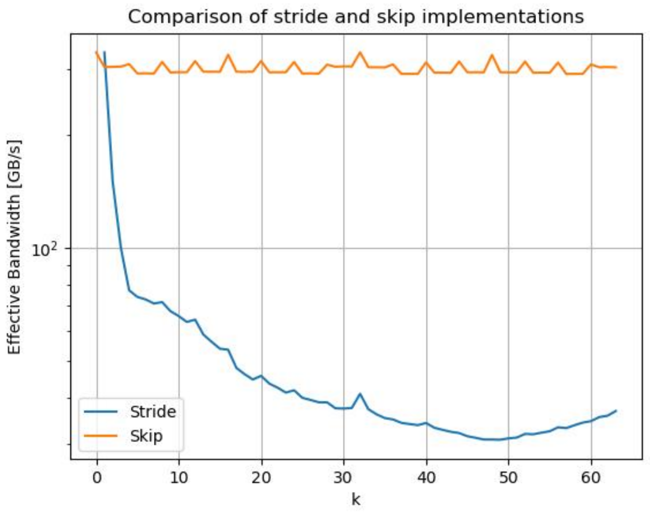
Computational Science on Many-Core Architectures
Exercise 3

Leon Schwarzäugl

3. November 2023

The code for all tasks can be found at: `https://github.com/Swarsel/CSE_TUWIEN/tree/main/WS2023/Many-Core%20Architectures/e3`

1 Strided and Offset Memory Access



1.1 a)

As we can see, the effective bandwith drops very quickly with rising k. This can be explained due to data locality: data is accessed in a contiguous manner by hardware, which we are not making use of when performing large strides, as then our threads are accessing addresses that are far apart. Hence the effective bandwidth is higher for small k. It should be recommended to keep this effect in mind and implement code in a manner that makes use of data locality - in this case, avoiding larger strides.

1.2 b)

Here we see a more constant effective bandwidth; this is again to be expected, since we are simply leaving a contiguous block of addresses out at the start; the fluctuations can be explained by the same effect, as the effective bandwidth seems to be especially high for multiples of 4. It seems that leaving away a number of entries that is not divisible by it causes the need to load another chunk of data which in turn lessens the effective bandwidth a little. As for a recommendation, it can be advised to keep this effect in mind and only skip blocks of 4 entries at once, for expample.

2 Dense Matrix Transpose

2.1 a)

I identified two issues:

1. There is a memory leak, since the free and cudaFree functions are not called at the end.
2. The kernel only works up to the entry assigned to the n-th thread, as there is nothing implemented towards looping over the rest of the data.

2.2 b)

A simple kernel was implemented that stores the upper triagonal value in a temporary variable and then updates both values corresponding to the row-column pair:

```
__global__ void transpose(double *A, int N)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int total_threads = blockDim.x * gridDim.x;

  for (unsigned int i = thread_id; i < N*N; i += total_threads) {
      int r = i / N;
      int c = i % N;

      if (r < c) {
         double temp = A[N * r + c];
         A[N * r + c] = A[N * c + r];
         A[N * c + r] = temp;
      }
   }
}
```

Also, the mentioned free-operations were added.
For the given $N = 10$ an effective bandwidth of $0.106667 \frac{\text{GB}}{\text{sec}}$ was measured.

2.3 c)

Dropping the in-place requirement simplifies the value-updating and we can drop the need for the temporary variable:

```
__global__ void transpose(double *A, double *B, int N)
{
  int thread_id = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int total_threads = blockDim.x * gridDim.x;
  for (unsigned int i = thread_id; i < N*N; i += total_threads) {
      int r = i / N;
      int c = i % N;
      if (r < N && c < N) {
         B[N * r + c] = A[N * c + r];
      }
   }
}
```

2.4 d)

The implemented kernel is similar to the one presented in the NVIDIA-blog; the core change emerges due to the possibility of a data race because we are transposing in-place. As such we need to make sure that we have all our working data ready before we begin to write. I did this by introducing two tiles which hold the two tiles that will be swapped. Once we have these saved, the writes can be done using these and we are no longer threatened by other blocks.

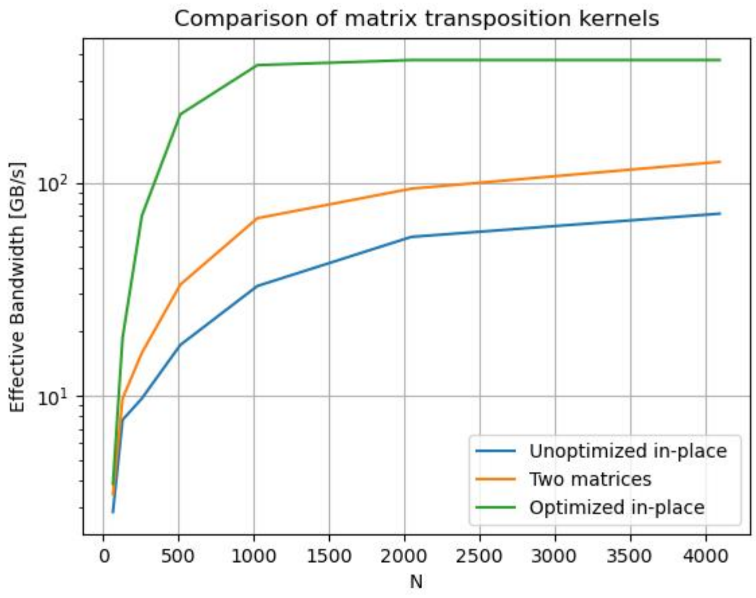
```
__global__ void transpose(double *A)
{
  // the +1's are for avoiding shared memory bank conflicts
  __shared__ float tile_1[16+1][16+1];
  __shared__ float tile_2[16+1][16+1];
  int x = blockIdx.x * 16 + threadIdx.x;
  int y = blockIdx.y * 16 + threadIdx.y;
  int width = gridDim.x * 16;
  int t_x = blockIdx.y * 16 + threadIdx.x;
  int t_y = blockIdx.x * 16 + threadIdx.y;

  tile_1[threadIdx.y][threadIdx.x] = A[y * width + x];
  tile_2[threadIdx.y][threadIdx.x] = A[t_y * width + t_x];
  __syncthreads();

  // for diagonal elements just perform the transpose,
  // for off-diagonal elements also swap the respective tiles
  if (blockIdx.y < blockIdx.x) A[t_y * width + t_x] = tile_1[threadIdx.x][threadIdx.y];
  A[y * width + x] = tile_2[threadIdx.x][threadIdx.y];
}
```

Note: Another way to achieve the same result would be by - I think! - using co-operative groups with grid-level sync() as shown in `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#grid-synchronization`. I tried to do this first before trying this approach, but I think it can only be done when a specific compiler flag is set, which was not possible in the exercise environment.

2.5 e)



The comparison of the different kernels shows that our optimizations have had a positive impact on the effective bandwidth. We can see that the dropping of the in-place requirement led to a first increase in performance. The implementation using shared memory where the reads and writes are coalesced leads to the biggest increase in effective bandwidth.

3 Trick AND Treat

At the start, I just moved the lines timer.reset() and timer.get() to convenient spots, which already showed quite well that the GPU version was by far superior.

Then I thought I should make extra sure and added a little sanity test to the CPU version, because it performed so poorly when compared to the GPU version. Even for low $N \approx 10000$, the CPU version already took close to the maximum alotted 30 seconds.

To make doubly sure, I ran some more tests on the same N for the GPU version, where surprisingly I noticed that the runtime t is about $t = -0.000002$s - so it literally saves time to run that code. :-)