DV2606 Assigment 2

Emil Karlström and Samuel Jonsson

December 8, 2022

1 Makefile

For `Makefile` information, see the `README.md` file included with the code.

2 Odd-even transposition sorting

2.1 Implementation

The implementations split into three versions, two single-kernel implementations and one multi-kernel:

Stride: For each iteration of the algorithm, we determine what pairs the thread shall compute using the formula `j = i % 2 + threadIdx.x * 2`, then take a stride of size `block_size * 2` to the next pair. There we compare the pair and perform a switch if necessary.

Chunks: We split up the array into a number of contiguous chunks equal to the amount of threads the program uses. Each thread is then mapped onto different chunks using the thread's id.

Multi-Kernel: Works similarly to that of stride in the sense that the way that threads are mapped to which pairs to sort is by using a stride that is equivalent to `BLOCKS * THREADS_PER_BLOCK * 2`. The initial pair each thread works on is calculated as

```
i%2 + (blockIdx.x*blockDim.x+threadIdx.x) * 2
```

The main difference in this implementation is that the for-loop is outside of the scope of each thread and we synchronise all threads using a loop on the host device instead. We start up a number of blocks equal to that of `BLOCKS` and each block contains `THREADS_PER_BLOCK` number of threads.

In all three implementations, to avoid any out-of-bounds errors we make sure that any indices computed stay within the bounds of the array, that is `j < vsize - 1` before comparing and computing eventual swapping of elements.

2.2 Measurements

Figure 1:
Odd-Even Sort Implementation Comparison.
SK = Single Kernel Launch, MK = Multi Kernel Launch.
Measurements in parallel implementations are run with 1024 threads.

Size		Sequential	SK with Stride	SK with Chunks	MK
2^{19}					
	Time (s)	>1200	118.75	1133.73	13.87
	Speedup	-	>10.11x	>1.06x	>85.51x
$100,000$					
	Time (s)	6.02	3.22	9	3.2
	Speedup	-	1.87x	0.669x	1.88x

One can see that the multi-kernel implementation has a speedup of greater than 85 compared to the sequential version. We could in fact not get the sequential version to complete its sorting on the list of 2^{19} items within 20 minutes, at which point we killed the program and came to the conclusion that it will most likely not complete within a reasonable amount of time.

We can also see that our chunk implementation for the single kernel launch has serious degradation in terms of performance, even compared to the sequential version where it performs worse on smaller lists. However it did complete sorting the 2^{19} sized list, which means that it performs slightly better on larger lists. When we changed the algorithm to use a stride rather than chunks, the performance immediately increased by up to 9.5 times.

Our reasoning for this is that when you use strides the device only needs to fetch large contiguous chunks of memory at a time because the threads all sort pairs that are close to each other in memory. If you however use chunks instead of strides, all threads work on different sections spread out all across the list at the same time, which means that the GPU needs to consistently grab data that is very spread out in memory.

3 Gaussian elimination

3.1 Implementation

The program begins with deciding the size of each data block, as well as the number of blocks we need when calculating the matrix:

```
int block_size = BLOCK_DIM_SZ * BLOCK_DIM_SZ;
int BLOCKS = max(1, N / block_size);
```

where `N` is the side of the matrix (a matrix is of size `N×N`) and `BLOCK_DIM_SZ` is the side of a block. We then initialise two `dim3` structures:

```
dim3 blockDims(
    BLOCK_DIM_SZ,
    BLOCK_DIM_SZ
);
dim3 gridDims(
    (int)ceil((float)N/(float)blockDims.x),
    (int)ceil((float)N/(float)blockDims.y)
);
```

which describe how many blocks and threads to start the GPU device with. The reason we use the `dim3` data structure is because it makes it easy to index on the x and y-axes in the matrix when we perform the elimination step.

For each row in the matrix `k`, we normalise the row using two CUDA kernel calls, one being `kernel_normalise_row` which normalises the entire row except the pivot value, and another call `kernel_norm_pivot` afterwards which normalises the pivot value. The reason for having two separate calls is because the pivot value cannot be normalised before the rest of the values. By having two sequential calls we guarantee that they happen in the right order.

After normalising the row, we make a CUDA call to `kernel_elimination` which performs the gauss-jordan elimination steps

```
cuda_A[y*N+x] -= cuda_A[y*N+k] * cuda_A[k*N+x];
```

for every row except the current pivot row `k`.

After gauss-jordan elimination, we calculate the values of vector `b` and `Y` through a CUDA call to `kernel_eval` which also sets all the values in row `k` and column `k` to 0.0 except the pivot value at `<k, k>`. It updates `y` and `b` according to the logic:

```
if(index < k)
  cuda_Y[index] -= cuda_A[index*N+k] * cuda_Y[k];
else if(k < index && index < N)
  cuda_B[index] -= cuda_A[index*+k] * cuda_Y[k];
```

for every index in `Y` and `b` where `k` is the current iteration of the outer most loop.

All steps from the normalisation step is repeated in order $N - 1$ times, as by the original Gauss-Jordan code.

3.2 Measurements

Table 1: Gauss-Jordan Elimination Implementation Comparison. The code is run on an array of size 2048×2048 and 4096×4096 and with `BLOCK_SIZE` 32×32 for the parallel version.

Matrix dim		Sequential	Parallel
2048			
	Time (s)	3.85	0.575
	Speedup	-	6.695x
4096			
	Time (s)	30.20	2.02
	Speedup	-	14.95x

We can see that we get a large speed up of a factor of 6.695 and when we increase the matrix dimensions to $4096x4096$ we get a speedup of nearly 15. We suspect that this is because the amount of time the GPU spends computing the Gauss-Jordan algorithm heavily start outweighing the time it takes to allocate and copy memory to and from the GPU device, meaning the total time spent more closely matches the actual computing time of the algorithm.