KØBENHAVNS UNIVERSITET

PMPH - Assignment 3

Mikkel Willén

1. november 2022

Indhold

Task 1 **2**

Task 2 **3**

Task 3 **3**

Task 4 **4**

Task 1

The first inner loop isn't parallel, since it needs to access the previously calculated element, to calculate the new element.
The second inner loop isn't parallel, since it also needs to access previously calculated elements, to calculate new elements.
The outer loop isn't parallel, since array A is overwriten in each iteration of the loop.

Since each element of array A is changed with each iteration, we can privatize array A.

```
float A[N, 2*M];

// parallel
for (int i = 0; i < N; i++) {
    A[i, 0] = N;
}

// parallel
for (int i = 0; i < N; i++) {
    // sequential
    for (int k = 1; k < 2 * M; k++) {
        A[i, k] = sqrt(A[i, k - 1] * i * k);
    }
}

// sequential
for (int i = 0; i < N; i++) {
    // sequential
    for (int j = 0; j < M; j++) {
        B[i + 1, j + 1] = B[i, j] * A[i, 2 * j    ];
        C[i,     j + 1] = C[i, j] * A[i, 2 * j + 1];
    }
}
```

Computing the direction vector for the two last loops, we get:

$$\begin{aligned}
&S_1 \rightarrow S_1 : (i+1, j+1) && = (i, j) \\
&i_1 < i_2 \quad \& \quad j_1 < j_2 \\
&S_2 \rightarrow S_2 : (i, j+1) && = (i, j) \\
&i_1 = i_2 \quad \& \quad j_1 < j_2 \\
&S_1 \rightarrow S_1 : [<, <] \\
&S_2 \rightarrow S_2 : [=, <]
\end{aligned}$$

With this we see, that we safely can make a loop interchange, in the last loop nest, since we get an $<$ in the outer most loop and we get the following code, with maximum parallelism:

```
float A[N, 2*M];

// parallel
for (int i = 0; i < N; i++) {
    A[i, 0] = N;
}

```

```
// parallel
for (int i = 0; i < N; i++) {
    // sequential
    for (int k = 1; k < 2 * M; k++) {
        A[i, k] = sqrt(A[i, k - 1] * i * k);
    }
}

// parallel
for (int j = 0; j < N; j++) {
    // sequential
    for (int i = 0; i < M; i++) {
        B[i + 1, j + 1] = B[i, j] * A[i, 2 * j    ];
        C[i,     j + 1] = C[i, j] * A[i, 2 * j + 1];
    }
}
```

Task 2

The outer loop is not parallel, since the accumulator is changed with each iterations of the inner loop, and reset with each iterations of the outer loop.

This could be fixed by privatizing the accum value and defining tmpA in the inner loop.

```
float A[N,64];
float B[N,64];
float accum[N]
for (int i = 0; i < N; i++) { // outer loop
    accum[i] = 0;
    for (int j = 0; j < 64; j++) { // inner loop
        float tmpA = A[i, j];
        accum[i] = sqrt(accum[i]) + tmpA * tmpA; // (**)
        B[i,j] = accum[i];
    }
}
```

The inner loop isn't parallel, since each iteration depends on the previous calculation of the accumulator.

The following is semantically-equivalent futhark code to if line (**) is rewriten as accum = accum + tmpA * tmpA

```
scan (+) 0 (map(/a -> a*a) A)
```

Task 3

```
transposeTiled<float, TILE>(d_A, d_Atr, num_thds, width);
transfProg<<< num_blocks, block >>>(d_Atr, d_Btr, num_thds);
transposeTiled<float, TILE>(d_Btr, d_B, width, num_thds);
```

```
__global__ void
transfProg(float* Atr, float* Btr, unsigned int N) {
    const unsigned int lid = threadIdx.x;
    const unsigned int gid = blockIdx.x * blockDim.x + lid;
```

```
    if (gid < N) {
        float accum = 0;
        for (int j = 0; j < 64; j++) {
            float tmpA = Atr[gid + j * N];
            accum = sqrt(accum) + tmpA * tmpA;
            Btr[gid + j * N] = accum;
        }
    }
}
```

Since the number of rows and the number of columns are switched, when we compose A and B, we have to iterate through elements of the array indexed by the threadId plus the blockId times the size of the block plus the number of columns of the transposed arrays times the number of the current iteration of the a sequential loop, from 0 to the number of rows in the tranposed arrays. This is done in the above program.
Testing the program, we get following speeds:

```
Naive Memcpy GPU Kernel runs in: 994 microsecs, GB/sec: 540.11

Original Program runs on GPU in: 33463 microsecs, GB/sec: 16.043718
GPU PROGRAM   VALID!
Coalesced Program with manifested transposition runs on GPU in: 3135 microsecs,
GB/sec: 171.250702
GPU PROGRAM   VALID!
```

We get a speedup of around 10.68 times, though it is still around 3.15 times slower than naive memcpy.

Task 4