Weekly assignment 3

October 2, 2020

Contents

1 TODO Task 1: Dependency-analysis transformations

I consider the following pseudocode

```
float A[2*M];

for (int i = 0; i < N; i++) {
    A[0] = N;
```

```
    for (int k = 1; k < 2*M; k++) {
A[k] = sqrt(A[k-1] * i * k);
    }

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

And answer the following questions:

1.1 Why neither the outer loop nor the inner loops are parallel

First off, we look at the inner loops: Section 5.2 of the lecture notes start off:

`A loop is said to be parallel if its execution does not cause any (true, anti or output`

We see easily the first loop `k in 0,...,2*M` that there is a RAW dependency In that $A[k_b]$ is read in the next iteration when $k_a = k_b + 1$. For the next loop, we also see dependencies in the next inner loop in B and C.

For the outer loop i present the case of accesses to B, with the following iterations (i_1, j_1), (i_2, j_2) reads from and writes to the same element. We get the equation system: $i_1 + 1 = i_2, j_1 + 1 = j_2 1$, which neccesarily means: $i_1 < i_2, j_1 < j_2$ giving us a well-formed direction vector `[<,<]` and a RAW dependency, and thus the outside isn't parallel by theorem 6

1.2 Why it is safe to privatize `A`

It is safe to privatize `A` Since we use a unique `A` In every iteration, e.g. every iteration `A` is initialized to the following: `A[k] = sqrt(A[k-1] * i * k)`. We Set up the following code:

```
float A[2*M];
float A_[2*M];
for (int i = 0; i < N; i++) {
    float A_[2*M];
/*S1:*/A_[0] = N;

    for (int k = 1; k < 2*M; k++) {
```

```
/*S2:*/A_[k] = sqrt(A_[k-1] * i * k);
    }

    for (int j = 0; j < M; j++) {
/*S3:*/B[i+1, j+1] = B[i, j] * A_[2*j  ];
/*S4:*/C[i,   j+1] = C[i, j] * A_[2*j+1];
    }
}
for (int i = 0; i < 2*M; i++){
    A[i] = A_[i];
}
```

1.3 Why it is safe to distribute the outermost loop across the `A[0] = N;` statement, along with the two inner loops

I look at the dependencies in the graph:

- S1 has no dependencies, since A_ is a new array
- S2 has a RAW dependency with S2 [=, <]
- S3 has a dependency on S3 [<, <]
  - Iter (i_1, j_1) writes to B[i+1, j+1]. Iter (i_2, j_2) reads from B[i, j].

  For these to write to the same memory adress, the following holds:

```
i_1 + 1 = i_2
j_1 + 1 = i_2
=>
i_1 < i_2
j_1 < j_2
```

- S4 has a dependency on S4 [=, <]
  - Iter (i_1, j_1) writes to C[i, j+1], Iter (i_2, j_2) Reads from C[i, j]. The following holds

```
i_1 = i_2
j_1 + 1 = j_2
=>
i_1 = i_2
j_1 < j_2
```

- S3 has a dependency on S2:

1.4 Using direction vectors, Determine which loops are parallel

1.5 Can loop interchange be applied?

2 Task 2: Parallel operator intuition

We look at the pseudocode:

```
float A[N,64];
float B[N,64];
float accum, tmpA;
for (int i = 0; i < N; i++) { // outer loop
/*S0*/accum = 0;
    for (int j = 0; j < 64; j++) { // inner loop
/*S1*/  tmpA = A[i, j];
/*S2*/  accum = sqrt(accum) + tmpA*tmpA; // (**)
/*S3*/  B[i,j] = accum;
    }
}
```

2.1 Is the outer loop parallel?

The outer loop has a dependency. With Accum being written to multiple times in each iteration, we see it form a WAW dependency, and therefore by thm. 6 it cannot be parallel.

2.2 What technique can be used to make it parallel?

Array Expansion(Privatization)

```
float A[N,64];
float B[N,64];
float Accum[N];
float accum, tmpA;
for (int i = 0; i < N; i++) { // outer loop
    Accum[i] = 0;
    for (int j = 0; j < 64; j++) { // inner loop
/*S1*/  tmpA = A[i, j];
/*S2*/  Accum[i] = sqrt(Accum[i]) + tmpA*tmpA; // (**)
/*S3*/  B[i,j] = Accum[i];
    }
```

```
}
accum = Accum[N-1];
```

2.3 Is the inner loop parallel

Looking at S2, we get the following dependency vector:

```
S2 accum
i_1, j_1 reads accum
i_2, j_2 writes accum
i_1 = i_2
j_1 < j_2
deps = [=, <]
```

By Thm. 6 this loop isn't parallel

2.4 Write the loop nest using parallel operators

We realize that each row in B is equivalent to scanning a squared row in map. Intuitively we also realize that we want at least two maps, one for the outer loop, and one for the inner loop. I produce the following output:

```
map (\A_i -> scan (+) 0 (map (\x -> x*x) A_i ) A
```

We verify this using the futhark repl, and a compiled c version beside it. It is tested using

```
main (replicate 3 (0...63) :> [3][64]i32)
```

Recognizing that there will never occur a value not representable by i32 in the given range. The code output matches that of an optimised C program

3 Task 3: Spatial locality optimizing in CUDA

For this task, i had to implement the code from the previous example in handed-out CUDA code. Following the previously implemented code closely, along with the code example on page 90 of the lecture notes, i implemented the following kernel:

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

```
    float accum = 0.0;
    unsigned int thd_offs = gid * N;

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

With the CPU orchestration looking like this:

```
// Transpose A
transposeTiled<float, TILE>(d_A, d_Atr, HEIGHT_A, WIDTH_A);
// Run the kernel
transfProg<<< num_blocks, block >>>(d_Atr, d_Btr, num_thds);
// Transpose B'
transposeTiled<float, TILE>(d_Btr, d_B, WIDTH_A, HEIGHT_A);
```

For benchmarking i ran the code 100 times, recording each timestamp. I got the following output:

```
Transpose on GPU runs in: 1381 microsecs, GB/sec: 388.755205
GPU TRANSPOSITION   VALID!
Original Program runs on GPU in: 33613 microsecs, GB/sec: 15.972122
GPU PROGRAM   VALID!
Coalesced Program with manifested transposition runs on GPU in: 1648 microsecs, GB/sec
GPU PROGRAM   VALID!
```

I ran the above benchmark/validation test on the `GPU04` machine, with a NVIDIA GTX 2080 graphics card. It is possible to see that, even with the added memory operations of transposing twice that we get an almost 10x speedup by using the coalesced memory reads.

4 Task 4: Matrix-Matrix Multiplication in CUDA

After some help from my TA, i managed to convert the psudocode in section 6.4 into the following kernel:

```
template <class ElTp, int T>
__global__ void matMultRegTiledKer(ElTp* A, ElTp* B, ElTp* C, int heightA, int widthB,
    // ToDo: fill in the kernel implementation of register+block tiled
```

```
    //       matrix-matrix multiplication here
    int ii = blockIdx.y * T;
    int j__= blockIdx.x * T * T;
    int i; // Always sequential
    int j_ = (threadIdx.y*T) + j__;
    int j  = j_ + threadIdx.x;
    int gidy = threadIdx.y + blockIdx.y*blockDim.y;
    int tidy = threadIdx.y, tidx = threadIdx.x;
    __shared__ float Ash[T][T];
    ElTp cs[T];
    // printf("%d: %d \n", j, gidy);
    #pragma unroll
    for (int i = 0; i < T; i++){
cs[i] = 0;
    }
    __syncthreads();
    // So now we add the sequential K loop that actually does "something"
    for(int kk = 0; kk < widthA; kk +=T ){
// Copy the array slice A[ii:ii+T, j] into shared memory
if ((kk+tidx) < widthB && gidy < heightA)
    Ash[tidy][tidx] = A[(kk+tidx) + (gidy*widthA)];
else
    Ash[tidy][tidx] = 0.0f;

__syncthreads();
// Then synchronize
for (int k = 0; k < T; k++) {
    float b;
    if (kk+k < widthA && j < widthB)
b = B[(kk+k)*widthB+(j)];
    else
b = 0.0f;
    #pragma unroll
    for(int i = 0; i < T; i++){
cs[i] += Ash[i][k] * b;
    }
}
__syncthreads();
    }
    #pragma unroll
```

```
    for(int i = 0; i < T; i++){
if ((ii + i < heightA) && (j < widthB)){
    C[((ii + i) * widthB) + j] = cs[i];
}
    }

}
```

We parralelize jjj and jj, ii and iii, with their indices being calculated at the top of the kernel. Running the code on GPU04, with the following parameters:

```
#define WIDTH_A  2048//1024//1024 //1024//2048
#define HEIGHT_A 2048//1024//2048//2048//2048
#define WIDTH_B  2048//4096//2048
#define TILE     16

#define GPU_RUNS 100
```

Results in the following benchmark:

```
./assignm3-task4
Sequential Naive version runs in: 4663266 microsecs
GPU Naive MMM version ... VALID RESULT!
GPU Naive MMM version runs in: 14640 microsecs
GPU Naive MMM Performance= 1173.49 GFlop/s, Time= 14640.000 microsec 128 128
GPU Block-Tiled MMM version ... VALID RESULT!
GPU Block-Tiled MMM version runs in: 11718 microsecs
GPU Block-Tiled MMM Performance= 1466.11 GFlop/s, Time= 11718.000 microsec 128 128
GPU Block+Register Tiled MMM version ... VALID RESULT!
GPU Block+Register Tiled MMM version runs in: 2860 microsecs
GPU Block+Register Tiled MMM Performance= 6006.95 GFlop/s, Time= 2860.000 microsec 8 1
```

As can be seen, we're approaching 6 Teraflops using our register-optimised version, getting almost a 6x speedup than the naive GPU multiplication.