Programming Massively Parallel Hardware
Assignment 3

Marc Raffy KPF905

October 11, 2021

Contents

1 Task 1

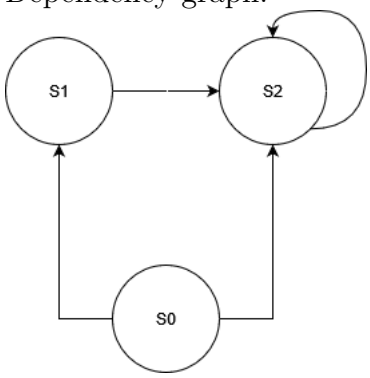
i loop is not parallel because there is a cross iteration WAW situation with the A[0]= N line. Indeed, we write to the same location during multiple iterations of the loop.

k loop is not parallel because of a cross iteration RAW situation. Indeed, we write to A but we also read from A and this would cause conflicts if parallelized.

k loop is not parallel because we have 2 RAW on B and C. The explanation is similar to the previous loop.

It is safe to privatize A because it is not used after the target (in our case i) loop and because all the reads to A are covered by a write.

Dependency graph:



On the dependency graph we can see that there are no cycles between the loops (S0 is the i loop, S1 the k loop and S2 the j loop) and thus we are allowed to safely perform loop distribution. It gives us the following code:

```
float A[2*M][N];
for (int i = 0; i < N; i++) {//parallel
    A[0][i] = N; //L0
}
for (int i = 0; i < N; i++) { // parallel
    for (int k = 1; k < 2*M; k++) { // sequential
        A[k][i] = sqrt(A[k-1][i] * i * k);//L1
    }

}
for (int i = 0; i < N; i++) {//sequential
    for (int j = 0; j < M; j++) {//sequential
        B[i+1, j+1] = B[i, j] * A[2*j  ][i];//L2
        C[i,   j+1] = C[i, j] * A[2*j+1][i];//L3
    }
}
```

Note that in the above code we mention which loops are parallel or not. It is because we computed the following direction vectors:

L0->L0: (=) -> i loop is parallel

L1->L1: (=, <) -> i loop is parallel because the parallelism is supported by i which is sequential

L2->L2: (<, <)
L3->L3: (=, <) i and j loop are sequential.

For the last 2 nested loops, according to the loop interchange theorem we are allowed to perform loop interchange and we would get the following direction vectors:
L2->L2: (<, <)
L3->L3: (<, =) This means that the outermost loop (which is now j) can now carry the dependency and make he innermost loop i parallel.

We can also interchange loops in L1 so that all the nested loops of the program contain a parallel loop and it gives us the following code.

```
float A[2*M][N];
for (int i = 0; i < N; i++) {//parallel
    A[0][i] = N;
}
for (int k = 1; k < 2*M; k++) { // sequential
    for (int i = 0; i < N; i++) { // parallel
        A[k][i] = sqrt(A[k-1][i] * i * k);//L1
    }

}
for (int j = 0; j < M; j++) {//sequential
    for (int i = 0; i < N; i++) {//parallel
        B[i+1, j+1] = B[i, j] * A[2*j  ][i];//L2
        C[i,   j+1] = C[i, j] * A[2*j+1][i];//L3
    }
}
```

2 Task 2

The outermost loop is not parallel because there is a WAW on accum=0. Indeed, across iterations we are writing at the same memory location. To solve it we can apply privatization and it would give us the following code:

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

The inner loop is not parallel because there is a RAW on accum on the line:

```
accum = sqrt(accum) + tmpA*tmpA; // (**)
```

Futhark psuedocode:

```
let AA = map(\row -> map(\cell -> cell**2) row) A
let B  = map(\row -> scan(+) 0 row) AA
```

3 Task 3

Kernel code:

```
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;

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

Main code:

```
// 3.a.1  you probably need to transpose d_A here by
            //        using function "transposeTiled<float, TILE>"
            //        i.e., source array is d_A, result array is d_Atr
            transposeTiled<float, TILE>( d_A, d_Atr, height, width );
            // 3.a.2  you probably need to implement the "transfProg"
            //        kernel in file transpose-kernel.cu.h which takes
            //        input from d_Atr, and writes the result in d_Btr,
            transfProg<<< num_blocks, block >>>(d_Atr, d_Btr, num_thds);
            // 3.a.3  you probably need to transpose-back the result here
            //        i.e., source array is d_Btr, and transposed result
            //        is in d_B.
            transposeTiled<float, TILE>( d_Btr, d_B, width, height );
```

For the main code there is not much to say we simply transpose the array compute the kernel and transpose the result array back to its original shape.
For the kernel code we changed the way we access the arrays. Since that we have coalesced access it means that we can see thng the following way: We have a 2D array with each column being a thread with gid index. There are N columns with length 64. Thus gid +j*N ensures that we access the first element of the column for j=0 and then we offset of N at each iteration meaning that we access the entire column at we should.
Our implementation validates.

memcopy bandwith: 540 GB/s
original implementation: 16 GB/s
coalesced implementation: 171 GB/s
We can see that there is a nice speedup (x11) with the coalesced access. However it still only uses 30% of the bandwidth.

4 Task 4

Kernel code:

```
template <class ElTp, int T>
__global__ void matMultRegTiledKer(ElTp* A, ElTp* B, ElTp* C, int heightA
    , int widthB, int widthA) {
    // ToDo: fill in the kernel implementation of register+block tiled
    //       matrix-matrix multiplication here
    __shared__ ElTp Ash[T][T];

    int ii   = blockIdx.y*T;
    int jjj  = blockIdx.x*T*T;
    int jj   = jjj + threadIdx.y*T;
    int j    = jj + threadIdx.x;

    if(ii < heightA && jjj < widthB){
      float cs[T];
      if(jj < widthB && j < widthB){
        #pragma unroll
        for(int i=0; i < T; i++){
          cs[i] = 0.0;
        }
      }
      for (int kk = 0; kk < widthA; kk+=T){
        Ash[threadIdx.y][threadIdx.x] = ((ii + threadIdx.y < heightA) &&
   (kk+threadIdx.x < widthA)) ?
          A[(ii + threadIdx.y)*widthA + kk + threadIdx.x] : 0.0;
        __syncthreads();
        for(int k = 0; k< T; k++){
          if(jj < widthB && j < widthB){
            float b = B[widthA*k + j];
            #pragma unroll
            for(int i = 0; i< T; i++){
              cs[i] += Ash[i][k] * b;
            __syncthreads();
            }
          }
        }
      }
      if(jj < widthB && j < widthB){
        #pragma unroll
        for(int i=0;i<T;i++){
          C[j*widthB+i] = cs[i];
        }
      }
    }
}
```

Main code:

```
int  dimy = ceil( ((float)HEIGHT_A)/TILE );
      int  dimx = ceil( ((float) WIDTH_B)/(TILE*TILE) );
      dim3 block(TILE, TILE, 1);
      dim3 grid (dimx, dimy, 1);

      unsigned long int elapsed;
      struct timeval t_start, t_end, t_diff;
      gettimeofday(&t_start, NULL);

      for(int k=0; k<GPU_RUNS; k++) {
          // 2. you would probably want to call here the kernel:
          matMultRegTiledKer<float,TILE> <<< grid, block >>>(d_A, d_B,
   d_C, HEIGHT_A, WIDTH_B, WIDTH_A);
      }
```

The code validates. However there have been issues on gpu04 where at first it validated but when I tried again after a few hours it didn't. I assume it is due to the GPU and not my code so I moved to GPU03 and it solved the issue.

Running times:

```
Sequential Naive version runs in: 2192390 microsecs
GPU Naive MMM version ... VALID RESULT!
GPU Naive MMM version runs in: 56079 microsecs
GPU Naive MMM Performance= 153.18 GFlop/s, Time= 56079.000 microsec 256 64
GPU Block-Tiled MMM version ... VALID RESULT!
GPU Block-Tiled MMM version runs in: 19915 microsecs
GPU Block-Tiled MMM Performance= 431.33 GFlop/s, Time= 19915.000 microsec 256 64
GPU Block+Register Tiled MMM version ... VALID RESULT!
GPU Block+Register Tiled MMM version runs in: 108 microsecs
GPU Block+Register Tiled MMM Performance= 79536.43 GFlop/s, Time= 108.000 microsec 16 64
```

We can see that our implementation is MUCH faster that the previous one. The reason for this is that we reduce the number of accesses to A and B. Indeed, now we have a shared A array for each block and only one access to B per thread.