Weekly Assignment 2 - Report

Francesco Done'
qwg586@alumni.ku.dk
Erasmus student

October 1, 2020

Contents

1 Task 1: Pen and Paper Exercise Aimed at Applying Dependency-Analysis Transformations

Consider the C-like pseudocode below:

```
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); //S1
   }

   for (int j = 0; j < M; j++) {
      B[i+1, j+1] = B[i, j] * A[2*j ]; //S2
      C[i,  j+1] = C[i, j] * A[2*j+1]; //S3
   }
}
```

- Explain why in the (original) code above neither the outer loop (of index i) nor the inner loops (of indices k and j) are parallel:
  - *Outer loop*: It isn't parallel because of `A[0]` assignment: if it was parallel, each thread would have written this location at same time.
  - *Inner loops*:
    $S1 \rightarrow S1 :\quad (k1) = (k2 - 1) \quad k1 < k2 \quad [\,<\,]$
    $S2 \rightarrow S2 :\quad (i1 + 1,\ j1 + 1) = (i2,\ j2) \quad i1 < i2 \;\&\; j1 < j2 \quad [\,<, <\,]$
    $S3 \rightarrow S3 :\quad (i1,\ j1 + 1) = (i2,\ j2) \quad i1 = i2 \;\&\; j1 < j2 \quad [\,=, <\,]$

    The direction matrix of loop with index k is $[\,<\,]$
    and the one of the loop with index j is $\begin{cases} [<, <] \\ [=, <] \end{cases}$
    so the inner loops cannot be proven parallels by *Parallel Loop Theorem*: in the first loop (index k), this is because theorem says that a loop in a loop nest is parallel iff all its directions are either = or there exists an outer loop whose corresponding direction is <. In the other loop (index j), in the former case this is because B[0, 0] is equal to < and there is no other outer loop to carry dependencies. In the latter case this is because C[1, 1] is equal to < and the outer loop for that row has direction = (instead of <, which would have been necessary to carry the dependencies of the inner loop).

- Explain why is it safe to privatize array A:

  It is safe since privatization can be implemented by performing either array expansion or moving the declaration of the target variable from outside to inside the loop. In our case, the declaration `float A[2*M]` inside the OMP parallel region, **will create an array A for every thread** (and visible/accessible inside the thread context).

- Once privatized, explain why is it safe to distribute the outermost loop across the A[0] = N; statement and across the other two inner loops. Perform (safely!) the loop distribution, while remembering to perform array expansion for A:

Right now it is safe to distribute the outermost loop across the A[0] = N; statement because each thread has its own array, so there are no conflicts in accessing and writing the variable location. Regarding to the inner loops, it is also safe to distribute them according to *Theorem 9.3 (Loop distribution)*.

```
#pragma omp parallel {
   float A[2*M];
   A[0] = N;
   #pragma omp for
   for (int i = 0; i < N; i++) {
      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];
      }
   }
}
```

2 Task 2: Pen and Paper Exercise Aimed at Recognizing Parallel Operators

Assume that both A and B are matrices with N rows and 64 columns. Consider the pseudocode below:

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

- Why is the outer loop not parallel?

  Because `accum` is a shared variable, thus it cannot be accessed and written by each thread at the same time.

- What technique can be used to make it parallel and why is it safe to apply it? Re-write the code such that the outer loop is parallel, i.e., the outer loop does not carry any dependencies.

  Privatization can be used, in order to have a number or `accum` equals to the number of number-of-processor size.

  ```
  float A[N,64];
  float B[N,64];
  float tmpA[N,64];
  float accum[N];

  #pragma omp parallel for
  for(i=0; i<N; i++) {
  //init accum[] with zeros
  accum[i] = 0;

     for(j=0; j<64; j++) {
        //copy of A[]
        tmpA[i,j] = A[i,j];
     }
  }

  # pragma omp parallel for
  for (int i = 0; i < N; i++) { // outer loop
     for (int j = 0; j < 64; j++) { // inner loop
        accum[i] = accum[i] + tmpA[i,j]*tmpA[i,j]; // (**)
     }
  }
  ```

```
# pragma omp parallel for
for (int i = 0; i < N; i++) { // outer loop
   for (int j = 0; j < 64; j++) { // inner loop
      B[i,j] = accum[i];
   }
}
```

- Explain why the inner loop is not parallel.

  It is not parallel because it has some RAW dependencies.

- Assume the line marked with (**) is re-written as accum = accum + tmpA*tmpA. Now it is possible to rewrite both the inner and the outer loop as a nested composition of parallel operators! Please write in your report the semantically-equivalent Futhark program.

```
--let A ...

--init accum[N] with zeros
let accum = replicate N 0 --useless


--init tempA[N,64] with zeros
let tmpA = map2 (\ n v -> replicate n v) (replicate 64 64) (replicate N 0)
-- = [ replicate 64 0, replicate 64 0, replicate 64 0, ... ]
-- = [ [0..0] , [0..0], [0..0], ... ]
-- = N rows with 64 cols initialized with zeros

--copy of A: A[N,64] + zeros[N,64] = copyOfA[N,64]
tmpA = map2 (+) tmpA A

let B = map (\ t -> scan (+) 0 t^2) tmpA
```

3 Task 3: Optimizing Spatial Locality by Transposition in CUDA

The solution implemented in `origProg`, as shown in *Lecture Notes* at page 89, uses:

- Index `i` for the current warp k as `(blockIdx.x * blockDim.x + threadIdx.x)`
- Flattened locations of A as `i*64+j`

```
__global__ void
origProg(float* A, float* B, unsigned int N) {
   unsigned int gid = (blockIdx.x * blockDim.x + threadIdx.x); // == i
   if(gid >= N) return;
   float accum = 0.0;
   unsigned int thd_offs = gid * 64;// i*64

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

Instead, the solution implemented in `transfProg`, as shown in *Lecture Notes* at page 90, uses:

- Index `i` for the current warp k as `(blockIdx.x * blockDim.x + threadIdx.x)`
- Flattened locations of A' as `j*N+i`

```
__global__ void
transfProg(float* Atr, float* Btr, unsigned int N) {
   unsigned int gid = (blockIdx.x * blockDim.x + threadIdx.x); // == i
   if(gid >= N) return;
   float accum = 0.0;

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

The CPU orchestration in file `transpose-main.cu` is shown here below:

```
// ...
 for (int kkk = 0; kkk < REPEAT; kkk++) {
   // 3.a.1
   transposeTiled<float, TILE>(d_A, d_Atr, HEIGHT_A, WIDTH_A);
   // 3.a.2
   transfProg<<< num_blocks, block >>>(d_Atr, d_Btr, num_thds);
   // 3.a.3
   transposeTiled<float, TILE>(d_Btr, d_B, WIDTH_A, HEIGHT_A);
```

```
  }
  // ...
```

Unfortunately the GPU 03 didn't validate `origPorg`, neither `transfProg`, and GPUs 02-04 didn't validate only the second one.