CSE590: Supercomputing, Spring 2016 **Date:** April 8

Homework #3

(Due: April 21)

Task 1. [250 Points] GPU Implementation of Floyd-Warshall's All-Pairs Shortest Path (APSP) Algorithm.

This task is an extension of Task 1 of Homework 1.

Consider a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by a real-valued length $l(v_i, v_j)$. If $(v_i, v_j) \notin E$, $l(v_i, v_j)$ is assumed to have a value $+\infty$. The length of a path is defined as the sum of the lengths of the edges in the path, taken in order. For each pair $v_i, v_j \in V$, $\delta[i, j]$ is defined to be the smallest of the lengths of all paths going from v_i to v_j. We assume that G does not have any negative length cycle.

Floyd-Warshall's algorithm for computing the length of the shortest path between every pair of vertices of G has a standard iterative implementation as shown in Figure 1. The input is an $n \times n$ matrix $D[1 \ldots n, 1 \ldots n]$ in which for every pair $i, j \in [1, n]$, $D[i, j]$ is initialized with $l(v_i, v_j)$. After the algorithm terminates $D[i, j] = \delta[i, j]$ for all $i, j \in [1, n]$.

```
LOOP-FLOYD-WARSHALL-APSP($D, n$)
1. for $k \leftarrow 1$ to $n$ do
2.     for $i \leftarrow 1$ to $n$ do
3.         for $j \leftarrow 1$ to $n$ do
4.             $D[i,j] \leftarrow \min(D[i,j], D[i,k] + D[k,j])$
```

Figure 1: [ITERATIVE IMPLEMENTATION] Looping code implementing Floyd-Warshall's APSP algorithm.

```
$\mathcal{F}_{loop\text{-}FW}(X, U, V)$                                   $\{\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}\}$
1. let $i_1, i_2, j_1, j_2, k_1, k_2 \in [1, n]$ with $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 \geq 0$ be indices such that
   $X \equiv D[i_1 \ldots i_2, j_1 \ldots j_2]$, $U \equiv D[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv D[k_1 \ldots k_2, j_1 \ldots j_2]$.
2. for $k \leftarrow k_1$ to $k_2$ do
3.     for $i \leftarrow i_1$ to $i_2$ do
4.         for $j \leftarrow j_1$ to $j_2$ do
5.             $D[i,j] \leftarrow \min(D[i,j], D[i,k] + D[k,j])$
```

Figure 2: Looping base case for the recursive implementation of Floyd-Warshall's APSP algorithm.

In this task we will consider a modified version of the recursive implementation of Floyd-Warshall's APSP shown in Figure 5 of Homework 1 (though we will only use up to 2 levels of the recursion). We modify each recursive function of that Figure (i.e., Figure 5 of Homework 1) so that one can control the way the input matrices are subdivided at each level of recursion by specifying a vector $tilesize$ of tiling parameters. At recursion depth $d \geq 1$, an $m \times m$ input matrix is subdivided into $r \times r$ submatrices of size $\frac{m}{r} \times \frac{m}{r}$ each, where $r = tilesize[d]$. We assume for simplicity that both m and r are powers of 2 and $r \geq 2$. Pseudocode for each of these functions is given in Figure 4.

Each function accepts a (recursion) depth parameter $d \geq 1$, and three (not necessarily distinct) equal size square submatrices X, U and V of the input matrix D. For each $D[i,j] \in X$, the entries $D[i,k]$ and $D[k,j]$ can be found in U and V, respectively. Each function updates the entries in X using appropriate entries from U and V. The functions differ in the amount of overlap X, U and V have among them. Function $\mathcal{A}_{FW}$ assumes that all three matrices completely overlap, while $\mathcal{D}_{FW}$ expects completely non-overlapping matrices. In the intermediate cases, function $\mathcal{B}_{FW}$ assumes that only X and V overlap, while $\mathcal{C}_{FW}$ assumes overlap only between X and U. Intuitively, the less the overlap among the input matrices the more flexibility the function has in ordering its recursive calls, and thus leading to better parallelism.

The initial call is to function $\mathcal{A}_{FW}$ with $d = 1$ and $X = U = V = D$. If X is an $m \times m$ submatrix of D and $m < tilesize[d]$, then X is updated directly by calling the iterative base case function (shown in Figure 2). Otherwise each matrix is subdivided into $r \times r$ submatrices of size $\frac{m}{r} \times \frac{m}{r}$ each, where $r = tilesize[d]$. The submatrix of X at the i-th position from the top and j-th position from the left is denoted by $X_{i,j}$. Then the function executes r supersteps. Superstep $k \in [1,r]$ of function $\mathcal{A}_{FW}$ consists of 3 steps (see Figure 3(a)). In step 1, submatrix $X_{k,k}$ is updated recursively by function $\mathcal{A}_{FW}$. In step 2, the remaining submatrices $X_{i,k}$ and $X_{k,j}$ ($i \neq k, j \neq k$) are updated in parallel by calling functions $\mathcal{B}_{FW}$ and $\mathcal{C}_{FW}$ (see Figure 3(a) for details). In step 3, the remaining submatrices of X are updated in parallel using entries computed in step 2 by calling function $\mathcal{D}_{FW}$ (see Figure 3(a)). Superstep $k \in [1,r]$ of functions $\mathcal{B}_{FW}$ and $\mathcal{C}_{FW}$ consists of 2 steps. In case of function $\mathcal{B}_{FW}$ ($\mathcal{C}_{FW}$), step 1 updates all $X_{i,k}$ ($X_{k,j}$, resp.) by parallel calls to $\mathcal{B}_{FW}$ ($\mathcal{C}_{FW}$, resp.). In step 2, the remaining submatrices of X are updated by parallel calls to function $\mathcal{D}_{FW}$ (e.g., see Figures 3(b) and 3(c)). Each superstep of function $\mathcal{D}_{FW}$ has only one step which updates all submatrices of X in parallel by calling $\mathcal{D}_{FW}$ recursively.

In this task we will assume that each entry of the input matrix D is a single precision floating point number.

(*a*) [**60 Points**] Implement each of $\mathcal{A}_{loop\text{-}FW}$, $\mathcal{B}_{loop\text{-}FW}$, $\mathcal{C}_{loop\text{-}FW}$ and $\mathcal{D}_{loop\text{-}FW}$ for GPU execution GPU. Note that no two of the four functions will have exactly the same implementation beacuse as we have seen in Homework 1, functions $\mathcal{B}_{loop\text{-}FW}$ and $\mathcal{C}_{loop\text{-}FW}$ have more parallelism than $\mathcal{A}_{loop\text{-}FW}$, and $\mathcal{D}_{loop\text{-}FW}$ has more parallelism than functions $\mathcal{B}_{loop\text{-}FW}$ and $\mathcal{C}_{loop\text{-}FW}$. Compare the CPU and GPU running times of each of these functions for $16 \leq n \leq n_g$, where n_g is the largest power of 2 such that three $n_g \times n_g$ matrices of single precision floats fit into the global memory of your GPU. Please consider only powers of 2 as values of n.

(*b*) [**75 Points**] Suppose $n = 2^q \times n_g$ for some positive integer q. Now implement the pseudocode shown in Figure 4 for GPU execution with $tilesize[1] = n/n_g = 2^q$ and $tilesize[2] = \infty$. Use your implementations of the looping codes from part (a). Compare the running time of this algorithm with a parallel CPU implementation of the iterative code in Figure 1 for $1 \leq q \leq 2$.

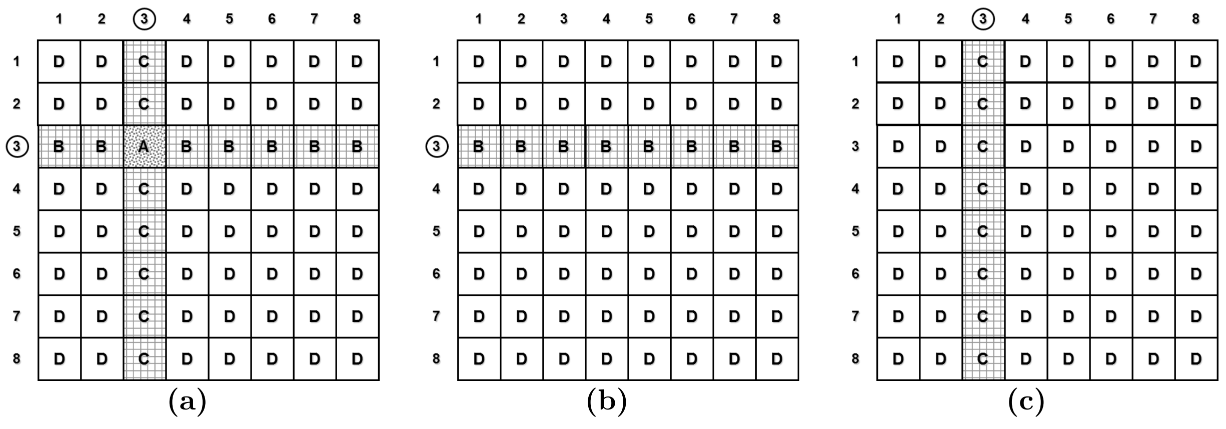

Figure 3: Execution of superstep $k = 3$ of functions $\mathcal{A}_{FW}$ (Figure 3(a)), $\mathcal{B}_{FW}$ (Figure 3(b)) and $\mathcal{C}_{FW}$ (Figure 3(c)). Each cell contains the name of the function used to update the entries in the corresponding submatrix. Submatrices corresponding to dotted cells (if any) are updated first. In the next step all submatrices corresponding to cells with grids are updated in parallel. In the last step submatrices corresponding to white cells are updated simultaneously. Pseudocode for these functions are given in Figure 4.

(*c*) [**75 Points**] Let n_s is the largest power of 2 such that three $n_s \times n_s$ matrices of single precision floats fit into the shared memory of a multiprocessor of your GPU. As in part (*b*), suppose $n = 2^q \times n_g$ for some positive integer q. Now implement the pseudocode shown in Figure 4 for GPU execution with $tilesize[1] = n/n_g = 2^q$, $tilesize[2] = n_g/n_s$ and $tilesize[3] = \infty$. When you reach a subproblem of size $n_s \times n_s$ copy the submatrices required for the computation to the shared memory of the multiprocessor where the looping base case will be executed. Add the running times of this algorithm to the table of running times you produced in part (*b*).

(*d*) [**40 Points**] Repear part (*c*) but do not copy the required submatrices to the shared memory even when you reach a subproblem of size $n_s \times n_s$. Read everything directly from the global memory during execution of the looping base cases. Add the running times of this algorithm to the table of running times you extended in part (*c*).

```
$\mathcal{A}_{FW}( X, U, V, d )$        {Each of $X$, $U$ and $V$ points to the same $m \times m$ square submatrix of the $n \times n$ input
                                        matrix $D$. We assume for simplicity that $r$ divides $m$ (if $m > 1$), where $r = tilesize[d]$.}

1.  $r \leftarrow tilesize[d]$
2.  if $r > m$ then $\mathcal{A}_{loop\text{-}FW}(X, U, V)$
    else
3.      Split each of $X$, $U$ and $V$ into $r \times r$ square submatrices of size $\frac{m}{r} \times \frac{m}{r}$ each. The submatrix of $X$ (or $U$ or $V$)
        at the $i$-th position from the top and $j$-th position from the left is denoted by $X_{ij}$ (or $U_{ij}$ or $V_{ij}$, resp.).
4.      for $k \leftarrow 1$ to $r$ do
5.          $\mathcal{A}_{FW}( X_{kk}, U_{kk}, V_{kk}, d+1 )$
6.          parallel : $\mathcal{B}_{FW}( X_{kj}, U_{kk}, V_{kj}, d+1 ),\ j \in [1, r],\ j \neq k$
                       $\mathcal{C}_{FW}( X_{ik}, U_{ik}, V_{kk}, d+1 ),\ i \in [1, r],\ i \neq k$
7.          parallel : $\mathcal{D}_{FW}( X_{ij}, U_{ik}, V_{kj}, d+1 ),\ i, j \in [1, r],\ i \neq k,\ j \neq k$
```

```
$\mathcal{B}_{FW}( X, U, V, d )$        {$X \equiv V \equiv D[i_1..i_2, j_1..j_2]$ and $U \equiv D[i_1..i_2, k_1..k_2]$, where
                                        $i_2 - i_1 = j_2 - j_1 = k_2 - k_1$, $[i_1, i_2] = [k_1, k_2]$ and $[j_1, j_2] \cap [k_1, k_2] = \emptyset$.}

1–3. Same as steps 1–3 of $\mathcal{A}_{FW}$ except that $\mathcal{B}_{loop\text{-}FW}(X, U, V)$ is called instead of $\mathcal{A}_{loop\text{-}FW}(X, U, V)$
4.      for $k \leftarrow 1$ to $r$ do
5.          parallel : $\mathcal{B}_{FW}( X_{kj}, U_{kk}, V_{kj}, d+1 ),\ j \in [1, r]$
6.          parallel : $\mathcal{D}_{FW}( X_{ij}, U_{ik}, V_{kj}, d+1 ),\ i, j \in [1, r],\ i \neq k$
```

```
$\mathcal{C}_{FW}( X, U, V, d )$        {$X \equiv U \equiv D[i_1..i_2, j_1..j_2]$ and $V \equiv D[k_1..k_2, j_1..j_2]$, where
                                        $i_2 - i_1 = j_2 - j_1 = k_2 - k_1$, $[j_1, j_2] = [k_1, k_2]$ and $[i_1, i_2] \cap [k_1, k_2] = \emptyset$.}

1–3. Same as steps 1–3 of $\mathcal{A}_{FW}$ except that $\mathcal{C}_{loop\text{-}FW}(X, U, V)$ is called instead of $\mathcal{A}_{loop\text{-}FW}(X, U, V)$
4.      for $k \leftarrow 1$ to $r$ do
5.          parallel : $\mathcal{C}_{FW}( X_{ik}, U_{ik}, V_{kk}, d+1 ),\ i \in [1, r]$
6.          parallel : $\mathcal{D}_{FW}( X_{ij}, U_{ik}, V_{kj}, d+1 ),\ i, j \in [1, r],\ j \neq k$
```

```
$\mathcal{D}_{FW}( X, U, V, d )$        {$X \equiv D[i_1..i_2, j_1..j_2]$, $U \equiv D[i_1..i_2, k_1..k_2]$ and $V \equiv D[k_1..k_2, j_1..j_2]$, where
                                        $i_2 - i_1 = j_2 - j_1 = k_2 - k_1$, $[i_1, i_2] \cap [k_1, k_2] = \emptyset$, and $[j_1, j_2] \cap [k_1, k_2] = \emptyset$.}

1–3. Same as steps 1–3 of $\mathcal{A}_{FW}$ except that $\mathcal{D}_{loop\text{-}FW}(X, U, V)$ is called instead of $\mathcal{A}_{loop\text{-}FW}(X, U, V)$
4.      for $k \leftarrow 1$ to $r$ do
5.          parallel : $\mathcal{D}_{FW}( X_{ij}, U_{ik}, V_{kj}, d+1 ),\ i, j \in [1, r]$
```

Figure 4: [Recursive Implementation] The initial call is $\mathcal{A}_{FW}(X, X, X, 1)$, where X points to $D[1 \ldots n, 1 \ldots n]$ and n is assumed to be a power of 2. Each $tilesize[i]$ is assumed to be power of 2 and at least as large as 2.

APPENDIX 1: What to Turn in

One compressed archive file (e.g., zip, tar.gz) containing the following items.

- Source code, makefiles and job scripts.
- A PDF document containing all answers and plots.

APPENDIX 2: Things to Remember

- **If you are using TACC resources, please never run anything that takes more than a minute or uses multiple cores on TACC login nodes.** TACC policy strictly prohibits such usage. They reserve the right to suspend your account if you do so. All runs must be submitted as jobs to compute nodes (even when you use Cilkview or PAPI).
- Please store all data in your work folder ($WORK), and not in your home folder ($HOME).
- When measuring running times please exclude the time needed for reading the input and writing the output. Measure only the time needed by the algorithm.