CS121 Problem Set 3

Due: 23:59, December 19, 2021

1. Submit your solutions to Gradescope (www.gradescope.com).
2. In "Account Settings" in Gradescope, set FULL NAME to your Chinese name and enter your STUDENT ID.
3. If you submit handwritten solutions, write neatly and submit a clear scan.

When submitting your homework, be sure to match each of your solutions to the corresponding problem number

4. Submit your solutions to Gradescope (www.gradescope.com).
5. In "Account Settings" in Gradescope, set FULL NAME to your Chinese name and enter your STUDENT ID.
6. If you submit handwritten solutions, write neatly and submit a clear scan.
7. When submitting your homework, be sure to match each of your solutions to the corresponding problem number.

1a) Consider the sequential code shown in the figure below. List all the dependence relationships in the code, indicating the type of dependence in each case. Draw the loop-carried dependence graph (LDG).

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = a[i-1][j] + b[i][j];
    S2: b[i][j] = c[i][j]
  }
```

1b) Using the LDG derived in Q1a, describe how to obtain a parallel algorithm, explaining any modifications required to improve the efficiency. Express your parallel algorithm using OpenMP.

2) Consider the loop shown in the figure below. Indicate the loop-carried dependence relationships. By restructuring the code, is it possible to eliminate this dependence and parallelize the loop? Express your answer using OpenMP.

```
a[0] = 0;
for (i=1; i<=n; i++)
  S1: a[i] = a[i-1] + i;
```

3) One way to get a numerical approximation to π is to sum the following sequence:

$$\pi = 4(1 - 1/3 + 1/5 - 1/7 + \dots)$$

A sequential algorithm to calculate π using this formula is given in the figure below. By analyzing the loop-carried dependences in this code, show how the algorithm may be parallelized, expressing your answer using OpenMP. Indicate clearly any modifications required to the code in order to allow parallelization and improve the efficiency.

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++) {
    S1: sum += factor/(2*i+1);
    S2: factor = -factor;
}
pi = 4.0*sum;
```

4a) Consider the following CUDA kernel for copying a matrix `idata` to another matrix `odata`. One reason for doing this is simply to test the memory bandwidth achievable on a GPU. At a high level, the kernel launches a 2D grid of thread blocks each of size `[TILE_DIM, BLOCK_ROWS]`, and each thread block copies a tile of values of size `TILE_DIM x TILE_DIM` from `idata` to `odata`. Suggested values are `TILE_DIM=32`, `BLOCK_ROWS=8`.

Give a detailed explanation of how the code works. Given an `NX x NY` matrix, how many thread blocks should be launched? Why do we want to set `BLOCK_ROWS` less than `TILE_DIM`? What does `width` represent? Why does the loop iterate over the variable `j`? What do the index calculations like `x*width+(y+j)` do?

```
__global__ void copy(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}
```

4b) We now modify the above kernel to transpose matrix `idata` to another matrix `odata`. Again, explain in detail how the code works. Do you expect the kernel to achieve good performance (compared to copying the matrix) when transposing a large matrix? Explain your reasoning.

```
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```