This page is an extra in case you run out of space when answering the exam questions. If you use this page, make sure you specify which question(s) you are answering here.

CPSC 418: Parallel Computing Winter 2016–2017 Term 2

Final Exam

Do all of the questions below. You have 150 minutes (2.5 hours) to complete the exam and there are 120 points available, so use the point allocation for each question as a guide when managing your time.

Although questions 1–4 are all about CUDA, they are exploring independent aspects of this programming paradigm. Even if you get stuck on one, you should be able to make progress on the others.

Where relevant in the CUDA questions, you may assume that kernels are launched with `blockDim.x = warp_size` and `blockDim.y = warp_size` where `warp_size = 32`. Except where otherwise specified, you may also assume that kernels are executing on a GTX 550 Ti (the same as is available in the `linXX` boxes in the lab). The specifications for this GPU are given in figure 1.

The figures are all on the final pages of the exam. You may remove these pages from the exam to make it easier to refer to them while answering the questions.

1. **CUDA and Global Memory (17 points)**

   Figure 2 shows a brute-force implementation of matrix multiplication (it is the same as the code from the March 24 lecture).

   (a) **Coalesced global memory references, definition (3 points):** What does it mean for memory references to be coalesced?

   (b) **Coalesced global memory references, pro or con (4 points):** In general, is it desirable to have coalesced memory references? Briefly explain why or why not.

(c) **Coalesced global memory references, example (5 points):** Give one example of a memory reference in the code from Figure 2 which **can** be coalesced. Briefly explain why the reference can be coalesced.

(d) **Coalesced global memory references, non-example (5 points):** Give one example of a memory reference in the code from Figure 2 which **cannot** be coalesced. Briefly explain why the reference cannot be coalesced.

2. **CUDA and Shared Memory (25 points)**

Figure 3 shows an implementation of matrix multiplication using shared-memory and tiles (it is a corrected version of the code from the March 24 lecture).

(a) **Bank conflicts (7 points):** What is a shared-memory bank conflict? In general, is it desirable to have bank conflicts? Briefly explain why or why not.

(b) **References to `a_sh` (5 points):** Consider the references to `a_sh` in the statement:
`sum += a_sh[i][k] * b_sh[k][j];`
Do these references incur bank conflict(s)? Why or why not?

(c) **References to `b_sh` (5 points):** Consider the references to `b_sh` in the statement:
`sum += a_sh[i][k] * b_sh[k][j];`
Do these references incur bank conflict(s)? Why or why not?

(d) **Effect of `__syncthreads()` (3 points):** Briefly explain what a call to `__syncthreads()` does.

(e) **Effect of not using `__syncthreads()` (5 points):** The code for the `mmult2` kernel includes two calls to `__syncthreads()`. What is a possible error scenario if the **second** call to `__syncthreads()` were deleted: How would the threads (mis)behave and what variables' and/or arrays' values might be affected?

3. **Memory Access vs Compute (25 points)**

   To simplify your formulas in the questions below, you may assume that the kernel is used to compute the product of two N-by-N matrices where N is a multiple of 32 and $N \gg 32$. Count only floating-point operations (ignore the cost of integer operations), and count each floating-point multiplication and each floating-point addition as a separate operation even if they can be fused.

   (a) **Define CGMA (3 points):** What is CGMA? Provide both the meaning of the acronym and a mathematical formula for its value.

   (b) **CGMA on GPUs (4 points):** Briefly explain why CGMA matters on a GPU. Do we want a low value or a high value for the CGMA?

   (c) **CGMA for CPUs? (3 points):** Is CGMA important when designing serial algorithms for traditional CPUs? Briefly explain your answer.

   (d) **Computing CGMA for brute force implementation (6 points):** What is the CGMA for the code from Figure 2?

(e) **Computing CGMA for tiled implementation (9 points):** What is the CGMA for the code from Figure 3?

4. **CUDA and Performance (15 points)**

(a) **GFlops (5 points):** The time to compute the product of two 1024-by-1024 matrices on `lin19` using the `mmult2` kernel from Figure 3 is 0.02 seconds (the average for 100 trials). How many GFlops (giga-floating-point operations per second) does this kernel achieve? Count each floating-point multiplication and each floating-point addition as a separate operation, even if they can be fused, and ignore integer operations.

(b) **Speed-up (5 points):** As noted above, `mmult2` takes 0.02 seconds to compute the product of two 1024-by-1024 matrics on `lin19`. I wrote a reference implementation in C and it took 1.9 seconds running on the CPU. What is the speed-up for the CUDA version from Figure 3 relative to this CPU version of matrix multiplication?

(c) **Faster GPUs (5 points):** The GTX 550 Ti GPUs in the `linXX` machines are fairly old. The recently released GTX 1080 has 2560 CUDA cores (i.e. SPs), can perform 8,228 GFlops, and has a global memory bandwidth of 320 GBytes/second. What is the minimum CGMA needed for the GTX 1080 to achieve its maximum GFlops?

5. **Short Answer (23 points)**

(a) **Map Reduce (6 points):** How does the implementation of map-reduce described in *MapReduce: Simplified Data Processing on Large Clusters* handle node failures (i.e. machine crashes)?

(b) **Shared Memory (3 points):** A vendor of shared memory multi-processors is trying to convince you to buy one by arguing that all cores on a shared memory machine can access all memory locations; consequently, parallel software implemented on these machines will not suffer from communication overhead. Briefly explain why this argument is false on modern hardware.

(c) **PReach (6 points):** What is load balancing?

How does PReach implement load balancing?

(d) **The 0-1 Principle and Sorting Networks (8 points):** Consider the following algorithm for sorting 0-1 sequences: Count the number of zeros in the input sequence and then create an output sequence of the same length as the input sequence with the appropriate number of zeros at the front and the remaining entries set to one. This algorithm cannot sort sequences of arbitrary numbers.

Can this algorithm be implemented by a sorting network? If so, sketch the sorting network for sequences of size 4. If not, explain why not.

6. **Map, Reduce and Scan (15 points)** A sequence of numbers $\{x_i\}_{i=1}^N$ of length N is already distributed across P processors with each processor p_i having K_i elements where $N \gg \overline{K} \geq K_i \geq \underline{K}$ for some maximum $\overline{K}$ and minimum $\underline{K} > 0$ number of elements. When generating an output, it can be distributed across the processors in any convenient manner (including not distributed in the case of scalar outputs). In answering the questions below assume that local operations take 1 unit of time and sending a message of any length between processors takes λ units of time for some $\lambda > 1$.

   (a) **Mean (2 points):** We would like to compute the mean of all of the entries in the data sequence; in other words, the mean of all entries $\{x_i\}_{i=1}^N$. Is this a map, reduce or scan operation (circle one):

   MAP REDUCE SCAN

   (b) **Complexity of Mean (5 points):** What is the "big-O" complexity (in terms of problem parameters N, P, $\overline{K}$, $\underline{K}$ and/or λ) of computing the mean of the entire sequence? Note that your "big-O" analysis may have more than one "leading" term depending on how the problem parameters combine.

   (c) **Running Mean (2 points):** We would like to generate a new sequence containing the running mean of all of the entries in the data sequence; in other words, entry j of the output sequence will contain the mean of all entries $\{x_i\}_{i=1}^j$. Is this a map, reduce or scan operation (circle one):

   MAP REDUCE SCAN

   (d) **Square (2 points):** We would like to generate a new sequence containing the squares of all of the entries in the data sequence; in other words, entry j of the output sequence will contain x_j^2. Is this a map, reduce or scan operation (circle one):

   MAP REDUCE SCAN

   (e) **Relative communication cost (4 points):** Rank the three problems "mean", "square" and "running mean" from least to most communication. Note that constant factors may determine the ranking. Briefly justify your ranking.

Do not write your answers on this page—it will not be graded. You may find it convenient to tear this page off when answering exam questions. If you tear it off, do not submit it with the rest of your exam.

- Number of SMs: 4
- Warp size: 32
- Maximum number of threads per block: 1024
- Maximum number of resident blocks per SM: 8
- Maximum number of resident warps per SM: 48
- Maximum number of resident threads per SM: 1536
- Register file capacity per SM: 128Kbytes
- Shared memory capacity per SM: 48Kbytes
- Number of banks of shared memory for each SM: 32
- Global memory capacity: 1 GByte

Figure 1: GTX 550 Ti specifications

```
__global__ void mmult1(float *a, float *b, float *c, uint32_t n) {
   uint32_t i = blockDim.y*blockIdx.y + threadIdx.y; // my row index
   uint32_t j = blockDim.x*blockIdx.x + threadIdx.x; // my column index
   if((i < n) && (j < n)) {
      float *a_row = a + n*i;
      float *b_col = b + j;
      float sum = 0.0f;
      for(uint32_t k = 0; k < n; k++)
         sum += a_row[k] * b_col[n*k];
      c[n*i + j] = sum;
   }
}
```

Figure 2: Brute-force CUDA implementation of matrix multiplication. Arguments `a` and b are the input matrices, `c` is the output matrix, and `n` is the problem size (all matrices are `n` $\times$ `n`).

Do not write your answers on this page—it will not be graded. You may find it convenient to tear this page off when answering exam questions. If you tear it off, do not submit it with the rest of your exam.

```
#define TILE_WIDTH 32

__global__ void mmult2(float *a, float *b, float *c, uint32_t n) {
   __shared__ float a_sh[TILE_WIDTH][TILE_WIDTH];
   __shared__ float b_sh[TILE_WIDTH][TILE_WIDTH];
   uint32_t i_blk = blockIdx.y; // my block-row index
   uint32_t j_blk = blockIdx.x; // my block-column index
   uint32_t i = threadIdx.y;    // row-index within this block
   uint32_t j = threadIdx.x;    // column-index within this block
   if((i_blk*blockDim.y + i < n) && (j_blk*blockDim.x + j) < n) {
      float *a_row_blk = a + n*i_blk*TILE_WIDTH; // start of our row of blocks
      float *b_col_blk = b + j_blk*TILE_WIDTH;   // start of our column of blocks
      float sum = 0.0f;
      // outer iteration: over the blocks of a_row_blk and b_col_blk
      for(uint32_t kk = 0; kk < n; kk += TILE_WIDTH) {
         // load blocks from a and b
         a_sh[i][j] = a_row_blk[n*i + kk + j];
         b_sh[i][j] = b_col_blk[n*(kk+i) + j];
         __syncthreads(); // first sync
         // inner iteration: over the elements of a_sh and b_sh
         uint32_t k_top = min(TILE_WIDTH, n-kk);
         for(uint32_t k = 0; k < k_top; k++)
            sum += a_sh[i][k] * b_sh[k][j];
         __syncthreads(); // second sync
      }
      // copy our result to the c matrix in global memory
      c[n * (i_blk*TILE_WIDTH + i) + (j_blk*TILE_WIDTH + j)] = sum;
   }
}
```

Figure 3: Matrix multiplication using CUDA shared-memory. Arguments a and b are the input matrices, c is the output matrix, and n is the problem size (all matrices are $n \times n$).