CPSC 418: Parallel Computing Winter 2016–2017 Term 2

Final Exam Solution Set

Students had 150 minutes (2.5 hours) to complete the exam and there were 120 points available

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.

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?

   If the threads of a warp access *nearby* locations of the global memory, the memory reference is said to be *coalesced*. Coalesced memory references only make one global memory access.

   (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.

   Generally, coalesced reference are prefered. The GPU can take advantage of accessing a large, contigous block of memory and achieve high bandwidth with the data transfer. Conversely, if locations are not coalesced, then several bank accesses may be needed, and several transfers of data from the DRAM to the GPU. Thus, coalesced references are handled faster than non-coalesced ones.

   (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.

   Reading from `b_col` in `b_col[n*k]` is coalesced. That is because all threads in the same warp share the same values for `n` and `k`. The values for `b_col` are consecutive for consecutive threads in a warp because

   ```
   b_col = b + j;
   ```

   where `j = threadIdx.x`, and threads are consecutively indexed first in the `x` dimension and then in `y` and `z`.
   Note (for grading): writing to `c[n*i + j]` **is coalesced**. All threads in a warp have the same value for `n` and `i` by the assumption that `n` is a multiple of 32. Thus, they only differ by `j` which, as explained above, is consecutive for consecutive threads of a warp.

   (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.

   Reading from `a_row` in `a_row[k]` **is not coalesced**. All threads in the same warp access the same location of `a_row` rather than consecutive locations. This means that the global memory delivers a large block of data (128 bytes = 32 floats) and only one is used. A mitigating factor is that each SM has a small cache, and it is likely that this data will be available in the cache on subsequent iterations of the for-loop.

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.

The shared memory for each SM is partitioned into *banks*. For the GPUs we have used, there are 32 banks, selected by bits 3 through 7 of the address. This means that consecutive `float`s are stored in different banks. More generally, if two floats have indices that are different modulo 32, they will be in different banks.
A *conflict* occurs if two threads in a warp access different locations in the same bank of shared-memory for the same load or store. In general, bank conflicts are undesirable because they cause the code to run slower. References to different banks can be handled in the same clock cycle, but references to different locations in the same bank must be handled on consecutive clock cycles. Thus, conflicts increase the number of clock cycles needed to complete a load or store operation.

(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?

These references **do not incur** bank conflicts. For the read of `a_sh[i][k]` all threads in the same warp have the same values of `i` and `k`. Thus, they all access the same location in the same bank. As described above, this is not a conflict.

(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?

These references **do not incur** bank conflicts. For the read of `b_sh[k][j]` the threads in a warp access 32 consecutive locations of the shared memory. Thus, they access each of the 32 banks, and there are no conflicts.

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

`__syncthreads()` implements a barrier: all threads in the block must reach the barrier before any continue beyond it.

(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?

If the second `__syncthreads()` were omitted, then a warp could complete its execution of the `for k` loop and continue to loading values into `a_sh` and `b_sh` for the next iteration of the `for kk` loop **before** some other warp in the block had completed its `for k` loop. This means that the "slow" warp could read values from `a_sh` or `b_sh` that had been overwritten by the "fast" warp and compute an incorrect result.

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.

CGMA stands for "Compute to Global Memory Access ratio". The formula is:

$$\text{CGMA} = \frac{\#FloatingPointOperations}{\#GlobalMemoryAccesses}$$

where *#FloatingPointOperations* is the number of floating point operations performed by the computation (e.g. a CUDA kernel), and *#GlobalMemoryAccesses* is the number of accesses (loads + stores) to the global memory.

(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?

Global memory bandwidth can easily be a performance bottleneck for GPU computations. To fully utilize the computing capabilities of a GPU, we need algorithms that perform a fairly large number of operations on each data value read from or written to the global memory. The CGMA describes this ratio of the amount of computation to the number of memory references. In general, higher values for CGMA are better as code with a higher value will usually have less performance loss from memory bandwidth constraints.

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

Yes. CPUs can execute a hundred instructions or more in the time that it takes to perform on DRAM access. For CPUs, the usual solution is caches. To get good performance, the average line loaded into a cache must be accessed a large number of times before it is replaced by another line from the DRAM.

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

Most of the computation takes place in the `for k` loop. Each iteration of this loop has one floating point multiply to compute the product of `a_row[k]` and `b_col[n*k]` and one floating point add to compute the sum of this product with the previous value of `sum`. Furthermore, each iteration of the loop body has two accesses of global memory: one to read `a_row[k]` and the other to read `b_col[n*k]`. Thus, there are two floating point operations and two global memory accesses per loop iteration. This yields a CGMA $= 1$.
Notes: by convention, CGMA only counts floating point operations. Thus, we don't count the multiplication for `n*k`. Furthermore, an optimizing compiler will replace this with a new variable (i.e. register) – let's call it `nk`. The compile will generate code to set `nk` to zero at the beginning of the loop and add `n` to `nk` at the end of each iteration.

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

We need to consider the `for kk` loop. The loads of `a_sh` and `b_sh` contribute a total of `2*TILE_WIDTH*TILE_WIDTH` = 2*32*32 = 2048 global memory accesses. The `for k` loop performs `TILE_WIDTH` floating point multiplies and `TILE_WIDTH` floating point adds for each thread. That is 64 floating point operations per threads. There are a total of `TILE_WIDTH*TILE_WIDTH` $= 1024$ threads in the block. Thus, the block performs $2^{16} = 65536$ floating point operations for each iteration of the `for kk` loop. The CGMA is

$$\text{CGMA} \quad = \quad \frac{65536}{2048} \quad = \quad 32$$

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.

A 1024-by-1024 matrix multiplication requires 1024^3 multiplies and 1024^3 adds, so $(2)1024^3$ floating point multiplications and additions. That's a total of 2,147,483,648 floating point operations. These are performed in $O.02$ seconds; therefore,

$$\begin{aligned} \text{GFlops} \quad &= \quad \frac{(2)1024^3}{0.02} * \frac{1\,\text{GFlop}}{2^{30}\,\text{Flops}} \\ &= \quad 100\text{GFlops} \end{aligned}$$

Note: some solutions may use the conversion that 1 GFlop $= 10^9$ Flops, in which case the answer is 107.37 GFlops. Either answer is acceptable.

(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?

For full credit, a more quantitative answer is expected.

$$\begin{aligned} \mathit{SpeedUp} &= \frac{T_{\text{CPU}}}{T_{\text{GPU}}} \\ &= \frac{1.9\text{s}}{0.02\text{s}} \\ &= 95 \end{aligned}$$

which is pretty impressive.
We can also express this as

$$\begin{aligned} \mathit{SpeedUp} &= \left(\frac{T_{\text{CPU}}}{T_{\text{GPU}}} - 1\right) \\ &= \left(\frac{1.9\text{s}}{0.02\text{s}} - 1\right) \\ &= 94 = 9400\% \end{aligned}$$

Either answer is acceptable.

(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?

320 GBytes/second is 80 Gfloats/second (1 float is 4 bytes). Thus, the required CGMA is

$$\begin{aligned} \text{CGMA}_{\text{min}} &= \frac{8{,}228\ \text{GFlops}}{80\ \text{GFloats/s}} \\ &= 102.85 \end{aligned}$$

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)?

The system makes use of the functional programming features of referential transparency and no side-effects.
Key features: A master node pings each worker periodically to detect failures and restarts failed workers.
Additional features: Map workers which fail must be restarted whether they completed or not because output is only stored locally. Reduce workers only need to be restarted if they did not complete because they write output to the distributed file system. The system is vulnerable to master node failures. Reduce nodes which already received data from a failed map node will use that data rather than asking for data from the reassigned map node. If map and reduce operators are deterministic, it will not matter which map worker's results are used. If map or reduce are not deterministic, then the results satisfy a relaxed form of consistency.

(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.

Modern shared memory processors have complicated caching protocols and often non-uniform access times even to the global memory. The only other option—a flat memory without any caches—is far too slow compared to the processor speed. Any memory location which is written by more than one core will cause communication of data between the processors (either directly between caches or by forcing a flush and then read to main memory). Parallel algorithms on shared memory machines must also typically implement synchronization or locking, which will require communication.

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

Load balancing refers to any method for moving tasks between processes to make sure that all process complete at roughly the same time. Does not need to be part of the answer, but we want load balancing to avoid performance loss due to idle processors when one process finishes its work early, and other processes still have a large amount of work to do.

How does PReach implement load balancing?

PReach uses a method that the authors called "light weight load balancing". The key idea is that processes send the size of their current work queue along with other data when sending newly encountered states to other workers. If a worker discovers that its work queue is much larger than that of some other task, it sends some of its work to the other process. They used a threshold of a factor of five difference in the work queue size. They found that this coarse-grained method for balancing incurred less communication overhead than finer grained approaches.
Note that PReach does use a "credit" system to manage communication between processors, but that is only to keep message queues from becoming too long (which appears to lead to quadratic message acceptance cost in the Erlang runtime); it is not used for 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.

This algorithm cannot be implemented by a sorting network. There are several possible proofs.

- Sorting networks satisfy the 0-1 principle, so if they sort all 0-1 sequences correctly they will sort all sequences correctly. This algorithm sorts 0-1 sequences correctly but cannot sort other sequences correctly; hence it does not satisfy the 0-1 principle and cannot be implemented with a sorting network.
- The proposed algorithm uses a conditional-increment operation but sorting networks only have the compare-and-swap operation.
- We can inductively show that the outputs of any sorting network are a permutation of the inputs. The proposed algorithm can only produce zeros or ones, so it will not produce a permutation of the input if the input contains any value that is not a zero or a one. Therefore, this computation cannot be performed by a sorting network.

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?

It is a reduce.

(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.

$\mathcal{O}(\overline{K}+\lambda \log P)$, where the first term is an upper bound on the cost of computing the sum on each individual processor and the second term is the cost of the reduce tree.

(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?

It is a 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?

It is a map.

(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.

$$\text{square (map)} \ < \ \text{mean (reduce)} \ \leq \ \text{running mean (scan)}.$$

The precise communication needs depend on if a single master process must initiate the operation or if all processes can self-initiate. In the former case, there is at least one pass down the tree to initiate the processors. The map operation requires no further communication. The reduce operation requires at least one pass back up the tree (if we are willing to get the answer only at the root) or a pass up and a pass down (if we want the answer everywhere). The scan operation always requires a pass up and a pass down the tree.

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`).