1. (19 points) **CUDA Reduce** Figure **??** provides code implementing a reduce problem in CUDA: Store into vector `y` the `m` largest values in the vector `x` (whose length is `n`). For your convenience, a pure CPU version has also been supplied in figure **??**. In answering the questions below, you may assume:

   - The function `reduce_kernel()` is launched with `gridDim = (1, 1, 1)` and `blockDim = (block_size, 1, 1)`.
   - The value of `block_size` is a power of 2 and $32 \leq$ `block_size` $\leq 1024$.
   - The input array `x` contains `n` positive floating point numbers and the output array `y` has been allocated of size `m` but has not been initialized.
   - The values of `n`, `m`, `block_size` and the constant `SIZE` have been chosen so that there are no array indexing or out of memory errors.

   For each of the questions below, briefly justify your answer in a sentence. Answers which do not include a brief justification may not receive full marks.

   (a) (2 points) Consider the line `shared[j*b + i] = 0.0;` in section 1 of `reduce_kernel()`. Will this memory access cause a bank conflict? Briefly explain why or why not.

   No bank conflict. The threads of a warp will have 32 consecutive values for `i` and the same values for `j` and `b`. Thus, they will access 32 consecutive addresses in the shared memory, and each access goes to a different bank.

   (b) (2 points) Consider the line `float curr = x[elem];` in section 2 of `reduce_kernel()`. Will this memory access cause a bank conflict? Briefly explain why or why not.

   No bank conflict because `x` is stored in global memory, not shared memory.

   (c) (2 points) Consider the line `float curr = x[elem];` in section 2 of `reduce_kernel()`. Will this memory access be coalesced? Briefly explain why or why not.

   The references will be coalesced. `elem = k*b + i`; `b = blockDim.x` is a multiple of of 32; and the values of `i` are consecutive for the threads of a warp.

   (d) (2 points) Consider section 2 of `reduce_kernel()`. Give **one** reason why it is a good idea to use shared memory to store the array `shared[]` for this section of the code. Answers which give more than one reason will receive zero.
   Would this code still work if `shared[]` were in global memory?

   Shared memory is the best choice because it can be indexed (unlike registers). The code would work if `shared` were in global memory, but it would be slower.

   (e) (2 points) Consider section 3 of `reduce_kernel()`. Give **one different** reason why it is a good idea to use shared memory to store the array `shared[]` for this section of the code. Answers which give more than one answer or which repeat the answer from the previous part will receive zero.

   Shared memory provides a fast (compared with global memory) mechanism for transferring values between threads in a block. We would need to use atomic operations (slow) if global memory were used instead.

   (f) (1 point) Would the code in section 3 still work if `shared[]` were in global memory?

   No, the code could fail.
   Explanation (not required for a solution): the code is not guaranteed to work because the `__syncthreads()` operation does not provide any guarantees about consistency of global memory.

© 2018 by Mark Greenstreet & Ian M. Mitchell.

(g) (3 points) Consider the line `y[i] = shared[i*b];` in section 4 of `reduce_kernel()`. Will either of these memory accesses cause a bank conflict? Briefly explain why or why not.

> The accesses to `shared[i*b]` will all cause bank conflicts because `b` is a multiple of 32 (an assumption from the problem statement).

(h) (3 points) Consider the line `y[i] = shared[i*b];` in section 4 of `reduce_kernel()`. Will either of these memory accesses be coalesced? Briefly explain why or why not.

> The writes to `y[i]` will be coalesced because threads in the same warp will have consecutive values for `i`.

(i) (2 points) Briefly explain why `__syncthreads()` must be called at the start of section 4 of `reduce_kernel()`.

> If `__syncthreads()` were not called, then the warp that reads `float curr = shared[other_j*b + i + stride];` could perform the read before the other warp writes that location in a previous iteration of the `for(stride...)` loop.

2. (20 points) **Closest Point** Figure **??** provides code implementing a basic version of a closest point calculation, and figure **??** provides a tiled version of the same algorithm. You may assume that `nc` $\gg$ `np` $\gg 1$.

(a) (3 points) Assume that the data is stored column major. The code currently stores each input data point in `d_points` as a column, so `d_points` is an array with `DIM` rows and `np` columns. A similar layout is used for `d_clusters`. Is this a good choice for arranging the data in `closest_basic_kernel()`, or should we instead have store each point as a row in these arrays? Briefly explain your reasoning.

> It would be better to store each point as a row. The majority of memory references are in the inner loop (the `d` loop) for
>
> ```
> // Then consider the remaining cluster points.
> ```
>
> The reference to
>
> ```
> d_clusters[IDX2F(d,j,nc,DIM)]
> ```
>
> is the same for all threads of a warp, so the row vs. column indexing isn't a big deal. The reference to
>
> ```
> d_points[IDX2F(d,i,np,DIM)]
> ```
>
> has consecutive values of `i` for the threads of a warp. We would like these to be consecutive memory locations to allow the memory accesses to be coalesced. By the assumption that the data is stored in column major order, we want `i` to be the row address. This means that each point is should be a row.

(b) (3 points) Write down the line numbers of `closest_basic_kernel()` which contain a floating point operation. Comparisons between floating point values count as a floating point operations, but integer operations and integer comparisons do not count. Multiply-adds which can be fused count as one operation. How many total floating point operations does one thread perform?

> line 20 (subtract, performed `DIM` times)
> line 21 (fused mult-add, performed `DIM` times)
> line 28 (subtract, performed `(nc-1)*DIM` times)
> line 29 (fused mult-add, performed `(nc-1)*DIM` times)
> line 31 (compare, performed `nc-1` times)
> total number of global memory accesses = `2*DIM + 2*(nc-1)*DIM + (nc-1)`
> = `2*nc*DIM + nc - 1`

(c) (3 points) Write down the line numbers of `closest_basic_kernel()` which contain a global memory access. How many total global memory accesses does one thread perform?

line 20 (two reads, performed `DIM` times each)
line 28 (two reads, performed `(nc-1)*DIM` times each)
line 38 (one write, performed once)
total number of global memory accesses = `2*nc*DIM + 1`.

(d) (1 point) What is the CGMA of `closest_basic_kernel()`? Your answer may depend on `nc`, `np` and/or `DIM`.

$$\begin{aligned}\text{CGMA} &= \frac{\texttt{2*nc*(DIM+1) - 1}}{\texttt{2*nc*DIM + nc - 1}} \\ &\approx \frac{\texttt{DIM+1}}{\texttt{DIM+}\frac{1}{2}}, && \text{for large } \texttt{nc} \\ &\approx 1 + \frac{1}{2\texttt{DIM}}, && \text{for "large enough" } \texttt{DIM}\end{aligned}$$

(e) (3 points) Write down the line numbers of `closest_tiled_kernel()` which contain a floating point operation. Comparisons between floating point values count as a floating point operations, but integer operations and integer comparisons do not count. Multiply-adds which can be fused count as one operation. How many total floating point operations does one thread perform?

line 19 (subtract, performed `DIM` times)
line 20 (fused mult-add, performed `DIM` times)
line 36 (subtract, performed `(nc-1)*DIM` times)
line 37 (fused mult-add, performed `(nc-1)*DIM` times)
line 39 (compare, performed `nc-1` times)
total number of global memory accesses = `2*nc*DIM + nc - 1`
(same as for `closest_basic_kernel`).

(f) (3 points) Write down the line numbers of `closest_tiled_kernel()` which contain a global memory access. How many total global memory accesses does one thread perform?

line 13 (one read, performed `DIM` times)
line 19 (one read, performed `DIM` times)
line 30 (one read, performed `ceil(nc/BLOCK_SIZE)*DIM` times)
line 47 (one write, performed once)
total number of global memory accesses = `(ceil(nc/BLOCK_SIZE)+2)*DIM + 1`.

(g) (1 point) What is the CGMA of `closest_tiled_kernel()`? Your answer may depend on `nc`, `np`, `DIM`, and/or BLOCK_SIZE.

$$\begin{aligned}\text{CGMA} &= \frac{\texttt{2*nc*DIM + nc - 1}}{\texttt{(ceil(nc/BLOCK\_SIZE)+2)*DIM + 1}} \\ &\approx \texttt{2*BLOCK\_SIZE*}\left(1 + \frac{1}{\texttt{DIM}}\right), && \text{for large } \texttt{nc} \\ &\approx \texttt{2*BLOCK\_SIZE*}, && \text{for "large enough" } \texttt{DIM}\end{aligned}$$

(h) (3 points) Consider the innermost loop (the loop over `j`) in `closest_tiled_kernel()`. Will this code run faster if `DIM` is 2 or if `DIM` is 3? Briefly explain.

The inner-most loop is indexed by `d`, not `j`.
Either way, the version with the larger value for `DIM` will take longer because the

d loop is performed more times. Note: the original intent was to have the DIM=3 loop be faster because of fewer bank-conflicts when accessing shared memory. BUT, we're using Fortran-inspired column major ordering, and DIM isn't part of the index calculation.

3. (18 points) **Short answer questions.**

   (a) (4 points) **Data Parallelism.** We discussed the fact that GPUs are well suited to data parallel problems, such as convolution or matrix-matrix multiplication. Give **one** architectural reason why GPUs are better suited than traditional CPUs to matrix-matrix multiplication. Are there any conditions under which a distributed message passing architecture (such as we explored using Erlang) would be well-suited to matrix-matrix multiplication? If not, briefly explain why not. If so, specify the condition(s).

   - GPUs are better that CPUs because they amortize the overhead of instruction fetch, decode, and pipeline control across many execution units. Operations such as matrix-multiplication and convolution work well in this context.
     Other acceptable answers:
     - GPUs have higher main-memory bandwidth than CPUs because GDDR (and now HBM2) is faster that DDR.
     - GPUs have more floating point units than CPUs.
   - Message-passing architectures work well for problems where the data size is too big to fit on a single GPU or single CPU. GPUs have very small amounts of memory available (e.g. 12-16GB for a GPU).

   (b) (3 points) **Lots of Threads.** One key to achieving high performance on GPUs is to ensure that kernels have many independent threads: Dozens of threads per core / SP. Give **one** reason why it is beneficial to have so many threads per core in a GPU. Give **one** reason why it is also beneficial to have several threads / processes available for each core of a modern CPU, and **one** reason why it is not necessary to have nearly as many threads / processes for the CPU cores as for the GPU cores to achieve reasonable efficiency. Answers which give more than one of each will receive zero.

   - SP pipelines are very deep without bypasses. Having a large pool of threads allows the thread scheduler to find threads that can issue instructions while instructions from other threads are working their way through the pipeline.
   - CPUs also benefit from multiple threads so that one thread can execute while another is blocked due to a cache miss.
     Other valid answers include:
     - Multithreading reduces performance loss from branch mispredication.
     - Multithreading can utilize the multiple functional units of a superscalar processor more effectively than single-threaded execution because instructions from different threads are independent of each other.
   - CPUs don't need as many threads per execution unit as GPUs because the CPU pipelines are not as deep.

   (c) (4 points) **Shared Memory.** We ran in to the term "shared memory" twice in the course: once as a type of parallel architecture (as opposed to message passing) and once as a specialized form of storage on the GPUs. Consider the kernel min_kernel_tree() in figure **??**, which uses the

GPU's shared memory. Does it use a shared memory architecture? Briefly explain your answer. Now consider the kernel `min_kernel_atomic()` in figure **??**, which does not use the GPU's shared memory. Does it use a shared memory architecture? Briefly explain.

- `min_kernel_tree` uses a shared memory architecture, the SPs within an SM access shared memory banks, very much like the "ancient shared memory" architectures from the lecture slides that introduced shared memory machines.
- `min_kernel_atomic` uses a shared memory architecture: the global memory is shared – in particular for the atomic read-modify-write operations performed on `*min`.

(d) (3 points) **Map-Reduce and Moments.** Assume that we store a very large collection of data values $\{x_i\}_{i=0}^{n-1}$ distributed across a large number of workers in a data center. The values are stored as *(Key1, Value1)* pairs such that *Key1* is $i$ and *Value1* is $x_i$. Further assume that all of the workers know the expected value of the data set $E[x]$. We would like to compute the central moments $\{\mu_k\}_{k=2}^{k\max}$ where

$$\mu_k \approx \frac{\sum_{i=0}^{n-1}(x_i - E[X])^k}{n}.$$

This calculation is the same problem explored in Homeworks 2 (Erlang) and 5 (CUDA). We would like to store the results as *(Key3, Value3)* pairs where *Key3* is $k$ and *Value3* is $\mu_k$. Is it possible to perform this computation using the Map-Reduce programming pattern (ignore efficiency considerations)?

[✓ yes] ☐ no (1 point)

If yes, explain what *(Key2, Value2)* representation you would use. If not, briefly explain why not (2 points).

*Key2*=`k` inspired by *Key3*. This means we could easily extend the code to handle multiple values for `k` in a single Map-Reduce run.

*Value2*={`N, Sum`} where `N` is the number of values for this map worker, and `Sum` is the sum of these values, minus `E[X]` raised to the `k`th power.

This representation allows the reduce worker to compute final sum and the total for $n$.

(e) (3 points) **BLAS and Complexity.** You have a computational problem and have managed to write it in two different forms (assume all matrices are $n \times n$ and vectors are of length $n$):

- A series of $n$ matrix-vector multiplications (Level-2 BLAS).
- A single matrix-matrix multiplication (Level-3 BLAS) plus $n$ vector-vector saxpy operations (Level-1 BLAS).

Assuming that $n$ is small enough that all the matrices and vectors can be stored in the GPU's global memory, which version would run faster? Briefly explain your answer.

The second approach will be faster.
In the first approach, a vector and matrix will be loaded from global memory with each of the $n$ matrix-vector multiplications. Each matrix-element is used once. This yields a CGMA between 1 and 2, with $n^3$ total memory accesses.
In the second approach, the matrix-matrix multiplication will be done with blocks. As with the first method, $O(n^3)$ floating point operations will be performed, but the CGMA will be large enough to fully utilize the GPU. The saxpy operations are lousy for CGMA, but there are only $n^2$ memory accesses instead of $n^3$.

(f) (1 point) ☺ Take three, slow, deep breaths. Relax. Write down one sentence that is positive about anything.

I've enjoyed the class this term – a big thank-you to everyone taking it, to our TAs, and to Ian.

4. **Performance Modeling** (10 points) Give one example for how each kind of performance loss listed below can occur in a CUDA program, or give a short (one or two sentence) explanation of why it cannot occur.

(a) (**2 points**): Communication.

GPUs communicate using memory, and memory accesses are slow. Threads in the same block can communicate using shared memory, which is fairly fast. Communication between different blocks requires using global memory – either by using atomics or by launching multiple kernels. These global memory accesses introduce communication overhead that can be very large. Likewise, communication between the CPU and GPU involves slow, global-memory accesses.

(b) (**2 points**): Synchronization.

Calls to `__syncthreads()` incur synchronization overhead.
Other valid answers:

- Atomics.
- Using multiple kernels to provide communication between blocks. All threads in one kernel must complete execution before the next kernel is launched.

(c) (**2 points**): Extra computation.

Anytime a value is recomputed by multiple threads. For example, in the `reduce_kernel`, all threads compute `num_elem` at line 7.
Another valid answer:

- An algorithm that runs well on a GPU may perform more operations than a sequential algorithm (or algorithm for a different parallel architecture). The implementation of scan using $\mathcal{O}(N \log N)$ operations instead of $\mathcal{O}(N)$ for sequential or many parallel implementations is an example. Using bitonic sort with an extra $\mathcal{O}(\frac{N}{P} \log^2 P)$ operations is another example.

(d) (**2 points**): Resource contention.

Bank conflicts when accessing shared memory. Other acceptable answers:

- Other valid answer include that some GPUs have a small number of *double precision* floating point units per SM. If a kernel uses lots of double precision operations, there will be resource contention. Another example is global memory bandwidth – with blocks executing on all SM's, they can generate more global memory traffic than the memory can support and warps will block waiting for memory accesses.
- Atomic operations such as `atomic_add` can cause contention for locks. Note: the textbook and CUDA documentation have descriptions that make it reasonable to assume that memory is accessed by each thread, one-at-a-time, for atomics; therefore, we are giving full credit for this answer. On the other hand, nVidia's GPU architects know what they are doing, and implementing reduce networks in hardware has been done for several decades. Kristen just coded up an example using `atomic_add` while we're grading the exams (thanks, Kristian!) and showed that atomics introduce a small amount of overhead, but not very much.

(e) (**2 points**): Idle processors.

Branch divergence. For example, when only some SPs are active during a reduce. Other acceptable answers:

- If a SM doesn't have enough warps, then the SM will be idle on cycles that the warp-schedule cannot issue an instruction.
- If a grid doesn't have enough blocks, then some SMs will be idle. For example, if a grid only has one block, then all but one of the SMs will be idle when that kernel executes.
- If there is random variation in the execution time of blocks, then some SMs will be idle at the end of a kernel's execution

Hint: feel free to use example from code for other questions on this exam—but if you would rather give your own example that is fine. Your examples should be **short**.

5. **A Different Kind of Tiling** (13 points) Many physical simulation problems are solved using *finite element methods* (FEM). These methods model a physical region using a grid of points. A sequential implementation of FEM involves a sequence of updates of the grid. Each update requires an update of each grid point. We will assume that a single update of a grid-point takes unit time. Thus a sequential implementation uses $N$ time units to update a grid with $N$-points.

We will assume that we have $P$ processors, a two-dimensional grid, and that the grid can be divided into $P$ "tiles" of size $K \times K$. Note that this implies that $N = PK^2$. To update a tile, a processor sends four messages, one to each of the neighbouring tiles. Each message has $K$ words. It takes time $\lambda + W$ to exchange messages of $W$ words with a neighbour. Each processor can exchange messages with only one neighbour at a time. You may also assume that computation and communication **cannot** be overlapped. After receiving messages from its 4 neighbours, the processor updates the grid-points for its tile using the sequential approach.

(a) (**1 point**) What is the time for the sequential algorithm to update a grid of $N = 10,240,000$ points? Hint: this is not a trick question, this is an *easy* question.

As stated in the problem description, $N$ time units for a sequential algorithm with $N$ elements, so $10,240,000$ time units.

(b) (**1 point**) What is $K$, if $P = 256$ and $N = 10,240,000$?

From the problem description $N = PK^2$ so $K = \sqrt{N/P} = \sqrt{10,240,000/256} = \sqrt{40,000} = 200$.

(c) (**2 points**) What is time for the parallel algorithm to update a grid of $N = 10,240,000$ points using $P = 256$ processors? Include both communication and computation time. Assume that $\lambda = 10,000$.

Computation time will be $N/P = K^2 = 40,000$ since the updates can proceed in parallel. Each processor also needs to send four messages. Each message takes time $\lambda + K = 10,000 + 200 = 10,200$, and the total communication time is 40,800. The total (computation + communication) time is $80,800$.

(d) (**1 point**) What is the speed-up if $N = 10,240,000$, $P = 256$, and $\lambda = 10,000$?

$10,240,000/80,800 \approx 126.73$

We can amortize the communication cost if we use *overlapping* tiles. Let $M$ be an integer – it's the number of rows or columns that each tile overlaps with each of its neighbouring tiles. To update a tile, each processor sends four messages, one to each of the neigbouring tiles. Each message consists of $M(K + M - 1)$ words. After receiving messages from its 4 neighbours, the processor performs $M$ updates to its tile; thus, one step of the parallel algorithm counts as $M$ steps of the sequential version. For the first of the $M$ updates, the processor works on a tile with $(K+M-1) \times (K+M-1)$ points, for the second iteration, the processor works on a tile with $(K + M - 2) \times (K + M - 2)$ points. For the $i^{th}$ of the $M$ updates, the processor works on a tile with $(K + M - i) \times (K + M - i)$ points.

(e) (**4 points**) What is time for the parallel algorithm to perform $M = 4$ updates to a grid of $N = 10,240,000$ points using $P = 256$ processors? Include both communication and computation time. Assume that $\lambda = 10,000$.

Communication: Each message is $\lambda + M(K+M-1) = 10,000+(4)(200+4-1) = 10,812$, so total communication time is $43,248$.

Computation: $M = 4$ iterations, first one is $(K + M - 1) = 200 + 4 - 1 = 203^2$, second is $202^2$, third is $201^2$ and fourth is $200^2$ for a total of $162,414$.

Total: the total time for four updates is therefore $205,662$.

(f) (**2 points**) What is the speed-up if $N = 10,240,000$, $P = 256$, $\lambda = 10,000$, and $M = 4$?

$(4)(10,240,000)/205,662 \approx 199.16$.

(g) (**2 point**) Does this method of overlapping tiles improve or degrade performance? Give a short explanation for this improvement or degradation in terms of the kinds of overhead described in Question **??**, i.e. communication, synchronization, extra computation, resource contention, and/or idle processors.

Using overlapping tiles improves performance. It introduces some extra computation – neighbouring tiles compute the same values for points in their overlap regions. In return for this slight increase in computation, the communication overhead is greatly reduced. The net effect is a large improvement in efficiency.
Note: It is possible to get even higher efficiencies by usinng larger values of $M$, but I wanted to keep the calculations simple for the exam.

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

```
__global__ void reduce_kernel(float *x, float *y, uint n, uint m) {

  __shared__ float shared[SIZE];

  uint i = threadIdx.x;
  uint b = blockDim.x;
  uint num_elem = ceil((double)n / (double)b);

  // Section 1.
  for(uint j = 0; j < m; j++)
    shared[j*b + i] = 0.0f;

  // Section 2.
  for(uint k = 0; k < num_elem; k++) {
    uint elem = k * b + i;
    if(elem < n) {
      float curr = x[elem];
      for(uint j = 0; j < m; j++) {
        float temp = shared[j*b + i];
        if(curr > temp) {
          shared[j*b + i] = curr;
          curr = temp;
        }
      }
    }
  }

  // Section 3.
  for(uint stride = b / 2; stride >= 1; stride = stride>>1) {
    __syncthreads();
    if(i < stride) {
      for(uint other_j = 0; other_j < m; other_j++) {
        float curr = shared[other_j*b + i + stride];
        for(uint j = 0; j < m; j++) {
          float temp = shared[j*b + i];
          if(curr > temp) {
            shared[j*b + i] = curr;
            curr = temp;
          }
        }
      }
    }
  }

  // Section 4.
  __syncthreads();
  if(i < m)
    y[i] = shared[i*b];
}
```

Figure 1: CUDA code for the reduction problem for question **??**.

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

```
void reduce_cpu(float *x, float *y, uint n, uint m) {

  for(uint j = 0; j < m; j++)
    y[j] = 0.0f;

  for(uint i = 0; i < n; i++) {
    float curr = x[i];
    for(uint j = 0; j < m; j++) {
      float temp = y[j];
      if(curr > temp) {
        y[j] = curr;
        curr = temp;
      }
    }
  }

}
```

Figure 2: CPU code which performs the same reduction as the CUDA code in figure **??**.

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

```
// Fortran column major indexing.  We don't actually need the number
// of columns (n), but we'll include it as an argument anyway for when
// we do C-style (row major) indexing.
#define IDX2F(i,j,m,n) (((j)*(m))+(i))

// For each element in d_points, identify the point in d_clusters
// which is closest.  In case of a tie, identify the first.
__global__ void closest_basic_kernel(const float *d_clusters, const uint nc,
                                     const float *d_points, const uint np,
                                     uint *d_closest) {

  // Assign a thread to each element of d_points.
  const uint i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < np) {
    // Initialize with first cluster point.
    float closest_dist2 = 0.0f;
    uint closest_index = 0;
    for(uint d = 0; d < DIM; d++) {
      float diff = (d_points[IDX2F(d,i,np,DIM)] - d_clusters[IDX2F(d,0,nc,DIM)]);
      closest_dist2 += diff * diff;
    }

    // Then consider the remaining cluster points.
    for(uint j = 1; j < nc; j++) {
      float dist2 = 0.0f;
      for(uint d = 0; d < DIM; d++) {
        float diff = (d_points[IDX2F(d,i,np,DIM)]-d_clusters[IDX2F(d,j,nc,DIM)]);
        dist2 += diff * diff;
      }
      if(dist2 < closest_dist2) {
        closest_dist2 = dist2;
        closest_index = j;
      }
    }

    // Write back the result.
    d_closest[i] = closest_index;
  }
}
```

Figure 3: Basic CUDA kernel for the closest point problem for question **??**.

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

```
__global__ void closest_tiled_kernel(const float *d_clusters, const uint nc,
                                     const float *d_points, const uint np,
                                     uint *d_closest) {

  // Index within the grid.
  const uint ig = blockIdx.x * blockDim.x + threadIdx.x;
  // Index within the block.
  const uint ib = threadIdx.x;

  __shared__ float sh_points[BLOCK_SIZE * DIM];
  if(ig < np)
    for(uint d = 0; d < DIM; d++)
      sh_points[IDX2F(d,ib,BLOCK_SIZE,DIM)] = d_points[IDX2F(d,ig,np,DIM)];

  // Initialize with first cluster point.
  float closest_dist2 = 0.0f;
  uint closest_index = 0;
  for(uint d = 0; d < DIM; d++) {
    float diff = (sh_points[IDX2F(d,ib,np,DIM)] - d_clusters[IDX2F(d,0,nc,DIM)]);
    closest_dist2 += diff * diff;
  }

  __shared__ float sh_clusters[BLOCK_SIZE * DIM];
  for(uint j_start = 1; j_start < nc; j_start += BLOCK_SIZE) {  // +1BB for missing 'uint'
    const uint j_last = min(BLOCK_SIZE, nc - j_start);

    __syncthreads();
    if(ib < j_last)
      for(uint d = 0; d < DIM; d++)
        sh_clusters[IDX2F(d,ib,BLOCK_SIZE,DIM)] = d_clusters[IDX2F(d,j_start+ib,np,DIM)];

    __syncthreads();
    for(uint j = 0; j < j_last; j++) {
      float dist2 = 0.0f;
      for(uint d = 0; d < DIM; d++) {
        float diff = (sh_points[IDX2F(d,ib,np,DIM)] - sh_clusters[IDX2F(d,j,nc,DIM)]);
        dist2 += diff * diff;
      }
      if(dist2 < closest_dist2) {
        closest_dist2 = dist2;
        closest_index = j_start + j;
      }
    }
  }

  if(ig < np)
    d_closest[ig] = closest_index;
}
```

Figure 4: Tiled CUDA kernel for the closest point problem for question **??**. The macro `IDX2F()` is defined in figure **??**.

**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.

```
__global__ void min_kernel_tree(int *x, const uint n, int *min) {

  const uint i = threadIdx.x;

  // Allocate shared memory to hold the partial results for this thread.
  __shared__ double sh_min[BS];
  // Initialize with first value assigned to this thread.
  sh_min[i] = x[i];

  // Examine the remaining values assigned to this thread.
  for(uint j = i + GS; j < n; j += GS)
    sh_min[i] = min(x[j],sh_min[i]);

  // Perform the reduction tree.
  for(uint stride = BS >> 1; stride >= 1; stride = stride >> 1) {
    __syncthreads();
    if(i < stride)
      sh_min[i] = min(sh_min[i],sh_min[i + stride]);
  }

  // Copy the final result out of shared memory and back to global memory.
  if(i == 0)
    min[0] = sh_min[0];
}


__global__ void min_kernel_atomic(int *x, const uint n, int *min) {

  const uint i = blockIdx.x * blockDim.x + threadIdx.x;

  // Initialize with first value assigned to this thread.
  int my_min = x[i];

  // Examine the remaining values assigned to this thread.
  for(uint j = i + GS; j < n; j += GS)
    my_min = min(x[j], my_min);

  // Combine results from different threads.
  atomicMin(min, my_min);
}
```

Figure 5: Two kernels to determine the minimum of an array of integers for question **??**.