Name _____

EE 7722

Take-Home Pre-Final Examination

Friday, 2 May 2014 to Tuesday, 6 May 2014

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(22 pts)
Problem 2	_____	(12 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(12 pts)
Problem 6	_____	(22 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [22 pts] Consider the code below, based on Homework 3. One difference is that the arrangement of elements in `idx_array` has been changed (but that's not part of the problem). Another difference is that all of `data_array` can be cached in shared memory.

Two versions are shown, `at_start` is similar to the solution to problem 2. In the `on_demand` version shared memory is not loaded until an element is needed.

```
__global__ void at_start(float* sum_array, float* data_array, int* index_array) {
  const int clength = 16;
  const int thread_count = blockDim.x * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int dcache_elts = ( 1 << 15 ) >> 2;

  __shared__ float dcache[dcache_elts];
  for ( int sdidx = threadIdx.x; sdidx < dcache_elts; sdidx += blockDim.x )
    dcache[sdidx] = data_array[sidx];
  __syncthreads();

  for ( int piece = tid; piece < num_pieces; piece += thread_count )
    {
      float sum = 0;
      for ( int i=0; i<clength; i++ )
        {
          const int idx = idx_array[ piece + i * num_pieces ];
          sum += dcache[ idx ];
        }
      sum_array[piece] = sum;
    }
}

__global__ void on_demand(float* sum_array, float* data_array, int* index_array) {
  const int clength = 16;
  const int thread_count = blockDim.x * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int dcache_elts = ( 1 << 15 ) >> 2;

  __shared__ float dcache[dcache_elts];
  for ( int sdidx = threadIdx.x; sdidx < dcache_elts; sdidx += blockDim.x )
    dcache[sdidx] = 0;
  __syncthreads();

  for ( int piece = tid; piece < num_pieces; piece += thread_count )
    {
      float sum = 0;
      for ( int i=0; i<clength; i++ )
        {
          const int idx = idx_array[ piece + i * num_pieces ];
          if ( dcache[ idx ] == 0 ) dcache[ idx ] = data_array[ idx ];
          sum += dcache[ idx ];
        }
      sum_array[piece] = sum;
    }
}
```

Problem 1, continued:

(*a*) Assume that elements in `idx_array` are randomly distributed over the size of `data_array` with a uniform distribution. Also assume that `num_pieces` is very large. Suppose the code is run on a device of CC 3.0, in which global memory is not cached in L1 or in the read-only (texture) cache. How much more data does `on_demand` load from off the multiprocessor compared to `at_start`?

☐ Additional data loaded from off the multiprocessor?

(*b*) Notice that `on_demand` does not use a `syncthreads` after it loads shared memory. Why isn't it needed in this case?

☐ `syncthreads` not needed in `i` loop because:

(*c*) We would like the `i` loop to be unrolled and to get the full benefit of unrolling. In particular, we'd like global accesses to be done as early as possible. That will probably occur with access to `idx_array` but not with accesses to `data_array` in the `on_demand` kernel.

Explain why and indicate the approximate time for one iteration of the `piece` loop in terms of memory access latency, L. (For example, one could say that there are 16 accesses to `idx_array` and 16 to `data_array`, so the iteration time is $32L$. That's wrong, provide the correct answer.) For this problem only consider the `on_demand` kernel.

☐ Why won't accesses occur as early as they could.

☐ Iteration time for one `piece` iteration in terms of L. State assumptions that are made.

Problem 2: [12 pts] Consider an NVIDIA GPU of CC 2.0, and assume that instruction latency is 24 cycles. For details on the timing of these devices see the C Programming Guide v5.5, section 5.4 (Maximize Instruction Throughput) and Appendix G (Compute Capabilities).

(*a*) Suppose the code below is launched in a kernel of four warps per multiprocessor. Show an execution diagram of the code, use the format in the NVIDIA CC GPU lecture notes. Note that `I3` depends on `I1` but `I2` does not.

```
I1: IADD r1, r2, r3
I2: DMADD r4, r6, r8, r12
I3: IMAD r10, r1, r11, r14
```

(*b*) What is the minimum number of warps needed to hide the latency of the `IADD` instruction in the code above. Consider just these instructions.

Problem 3: [20 pts] Appearing below is CUDA C code and the resulting assembler. Note that the code uses both predication and an undiverged branch to implement the if/else construct. The code is to run on a device of CC 2.0. For this problem assume that there are 32 functional units for branch instructions.

```
// C Code:
//   r1 = r2 + r3;
//   if ( cond ) { r4 = r6 * r8 + r12; } else { r4 = __sin(r6); }
//   r10 = r1 * r11 + r14
//
// NVIDIA Assembler
      I1:      IADD r1, r2, r3
      I2: @!p1 BRA.U  ELSE
      I3: @p1  DMADD r4, r6, r8, r12
      I4: @p1  BRA.U  NEXT
ELSE: I5: @!p1 MUFU.sine r4, r6
NEXT: I6:      IMAD r10, r1, r11, r14
```

(*a*) Suppose the code was launched with two warps per multiprocessor in a configuration in which the y and z dimension size is 1 (the way we always have done it in class). Show the execution when `cond = threadIdx.x & 1` (cond is true when `threadIdx.x` is odd. How many warps are needed to hide the latency of `I1` to `I6` (note the dependence)?

☐ Number of warps needed to hide latency.

☐ Show execution for `cond = threadIdx.x & 1`.

Problem 3, continued:

(*b*) Suppose the code was launched with two warps per multiprocessor. Show the execution when `cond = threadIdx.x & 32` (cond is true when the warp number [`threadIdx.x/32`] is odd. How many warps are needed to hide latency? Describe a potential load imbalance problem and how loss of performance due to the load imbalance might be avoided.

☐ Number of warps needed to hide latency.

☐ Load imbalance problem due to:

☐ Loss of performance avoided if:

☐ Show execution for `cond = threadIdx.x & 32`.

(*c*) Suppose half of the threads must execute the `if` part and half the `else` part. How should `cond` be set to achieve this in the most efficient way?

☐ Setting for `cond` for best performance.

Problem 4: [12 pts] We know that NVIDIA GPUs from 1.3 to at least 3.5 will try to coalesce the addresses in a half warp into contiguous load requests, hopefully just one request, but as many as 16 requests if the addresses are far apart. This coalescing of addresses is programmer friendly, but it probably consumes energy.

Based on the instruction descriptions, Phi must have such coalescing hardware too, but it's optional.

(*a*) Show a Phi instruction which would use something like the coalescing network. Briefly explain why.

☐ Phi instruction that would need a coalescing network.

☐ Explain.

(*b*) Show a Phi instruction which would likely not use something like the organizing network. Briefly explain why.

☐ Phi instruction that would not need a coalescing network.

☐ Explain.

Problem 5: [12 pts] One difference between the organization of NVIDIA GPUs and the Phi highlighted in class is many threads v. prefetch.

Consider a loop containing a prefetch instruction that, in iteration i, prefetches data which will be needed in iteration $i + d$, where d is a constant called the *prefetch distance*. The choice of the prefetch distance is based on the estimated memory latency and on the estimated time needed for an iteration of the loop. If the distance is too small the prefetched data will arrive after it is needed, forcing the *demand fetch* instruction to stall. If the distance is too large cache space is wasted.

(a) Suppose the compiler chooses to prefetch 30 iterations ahead. Suppose further that memory latency is 200 cycles and an iteration takes 10 cycles. The prefetch is for an 8-byte item.

Imagine prefetched values were sent to a special cache. What would be the minimum size of this prefetch cache for the code above? (Meaning, if the cache were smaller than this size a prefetched value would be evicted before it was used.)

(b) This prefetching also occurs in NVIDIA organizations, though not with a prefetch instruction. Explain how multithreading realizes a similar benefit. For this question assume that NVIDIA loads are not cached. The use of cache space by prefetched values is a storage overhead for prefetch in Phi (and conventional systems). What is the storage overhead in NVIDIA systems for achieving the prefetch benefit? How does the storage overhead compare to prefetch systems?

☐ Multithreading realizes a similar benefit to prefetch because:

☐ Compare storage overhead in the two cases.

(c) Loop unrolling too can realize some benefits of prefetching. Explain why loop unrolling won't help for Phi in this way.

Problem 6: [22 pts] Consider a CUDA-like programming system for Echelon in which a kernel could be launched in either an MIMD, SIMT, and SIMD (vector) execution mode. Regardless of the execution mode threads will have a variable named `threadIdx` with two components, `lane` and `pos`. As one might guess, `threadIdx.lane` can vary from 0 to 7 and `threadIdx.pos` from 0 to 63.

In MIMD mode no attempt is made to keep threads converged. In SIMT mode 8-thread groups within a lane remain converged, and in SIMD mode 8-thread groups, with each thread in a different lane, remain converged.

We would like it to be possible to generate 64-byte memory requests. In this problem consider the best way of doing so in each mode.

(*a*) Describe the best way to generate a 64-byte memory request for code operating in SIMD mode. Assign `start`, etc. in the code below so that access to `a` will generate 64-byte requests.

```
__global__ void prob_exec_config(double *a, double *b, double *c) {
  // Available: thread_count, num_lanes, thd_per_lane, num_element;
  const int start = ;
  const int stop = ;
  const int inc = ;

  for ( int idx = start; idx < stop;  idx += inc  ) c[idx] = s * a[idx];
}
```

☐ Assign variables above.

☐ Show the path taken by the requested data using the diagram from page 14 (8) of the Micro paper.

Problem 6, continued:

(*b*) Describe the best way to generate a 64-byte memory request for code operating in SIMT mode. Assign `start`, etc. in the code below so that access to `a` will generate 64-byte requests.

```
__global__ void prob_exec_config(double *a, double *b, double *c) {
  // Available: thread_count, num_lanes, thd_per_lane, num_element;
  const int start = ;
  const int stop = ;
  const int inc = ;

  for ( int idx = start; idx < stop;  idx += inc  ) c[idx] = s * a[idx];
}
```

☐ Assign variables above.

☐ Show the path taken by the requested data using the diagram from page 14 (8) of the Micro paper.

(*c*) Describe the best way to generate a 64-byte memory request for code operating in MIMD mode. Assign `start`, etc. in the code below so that access to `a` will generate 64-byte requests. *Hint: For MIMD mode think about something done for Homework 1.*

```
__global__ void prob_exec_config(double *a, double *b, double *c) {
  // Available: thread_count, num_lanes, thd_per_lane, num_element;
  const int start = ;
  const int stop = ;
  const int inc = ;

  for ( int idx = start; idx < stop;  idx += inc  ) c[idx] = s * a[idx];
}
```

☐ Assign variables above and make other modifications as necessary.

☐ Show the path taken using the diagram from page 14 (8) of the Micro paper.

(*d*) Why would the hardware necessary for executing such code be simplest for SIMD mode? Think about the width (in bits) of connections.