Name _____

EE 7722

Solve-at-Home Final Examination

Wednesday 7 May 2014 (late) to Saturday, 10 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	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] The code below is a simplified form of the solution to Homework 2.

(*a*) Modify the code so that array `keep` is in global memory. *Hint: The size of the array, among other things, will have to be changed.*

```
__global__ void dots_iterate2(float *a, float *b) {
  const int wp_lg = 5;
  const int warp_size = 1 << wp_lg;
  const int thread_count = blockDim.x * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int lane = threadIdx.x & ( warp_size - 1 );
  const int wp_num = tid >> wp_lg;
  const int idx_start = wp_num * warp_size * unroll_degree + lane;

  for ( int idx = idx_start; idx < dapp.array_size;
        idx += unroll_degree * thread_count ) {
      float keep[unroll_degree];
      for ( int i=0; i<unroll_degree; i++ )
          // Note: i * warp_size is a compile-time constant.
          keep[ i ] = dapp.v0 + dapp.v1 * a[idx + i * warp_size];


      for ( int i=0; i<unroll_degree; i++ )
        b[idx + i * warp_size] = keep[ i ];


    }}
```

(*b*) Suppose the code is executed on a device of CC 2.0 (Fermi) in which global memory is backed by the L1 cache. Since local memory also uses the L1 cache, shouldn't the original and modified codes have the same or very similar performance?

Problem 1, continued:

(*c*) Modify the code below that `keep` is in shared memory. *Hint: The size of the array, among other things, will have to be changed. The size should not match the original size or the size in the solution to the first part.*

```
__global__ void dots_iterate2(float *a, float *b) {
  const int wp_lg = 5;
  const int warp_size = 1 << wp_lg;
  const int thread_count = blockDim.x * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int lane = threadIdx.x & ( warp_size - 1 );
  const int wp_num = tid >> wp_lg;
  const int idx_start = wp_num * warp_size * unroll_degree + lane;

  for ( int idx = idx_start; idx < dapp.array_size;
        idx += unroll_degree * thread_count ) {
      float keep[unroll_degree];
      for ( int i=0; i<unroll_degree; i++ )
          // Note: i * warp_size is a compile-time constant.
          keep[ i ] = dapp.v0 + dapp.v1 * a[idx + i * warp_size];


      for ( int i=0; i<unroll_degree; i++ )
        b[idx + i * warp_size] = keep[ i ];


    }}
```

Problem 2: [20 pts] The CUDA library `atomicAdd` instruction operating on shared memory compiles into these instructions:

```
// CUDA
atomicAdd(&shared_array[sidx],5);

// Fermi machine instructions.

       /*01b0*/        LDSLK P1, R15, [R3];
       /*01b8*/    @P1 FADD R15, R15, 5;
       /*01c0*/    @P1 STSUL [R3], R15;
```

(*a*) Explain what the machine instructions are doing.

(*b*) Explain why using the atomicAdd to find the sum of all values in a block is not a good idea.

(*c*) The code generated for an atomicAdd operating on something in global memory is different, it emits a specialized reduction instruction, which sends the operation to be performed outside the multiprocessor, in a memory controller. Why might the approach used for shared memory (above) have much worse performance if used for global memory?

```
// CUDA
atomicAdd(&global_array[gidx],5);

// Fermi machine instructions.
/*0200*/        RED.E.ADD [R4], R7;
```

Problem 3: [20 pts] Appearing below is the solution to Homework 5 Problem 2.

```
void* sums_1(Thread_Data *td)
{
  const int tid = td->tid;
  App_Common* const dapp = td->app;
  const int num_pieces = dapp->num_pieces;
  const int num_threads = dapp->num_threads;
  const int piece_per_thread = ( num_pieces + num_threads - 1 ) / num_threads;
  const int start = tid * piece_per_thread;
  const int stop = min( num_pieces, start + piece_per_thread );
  ptri512 idx_array = dapp->idx_array;
  float* const data_array = dapp->data_array;
  float* const sum_array = dapp->sum_array;

  for ( int piece = start; piece < stop; piece++ )
    {
      const int idx_piece_start = piece * clength;
      float sum = 0;
      for ( int i=0; i<clength; i++ )
        sum += data_array[ idx_array[idx_piece_start+i] ];

      sum_array[piece] = sum;
    }
  return NULL;
}
```

(*a*) What would be the impact of doubling the vector width to 1024 bits on the code above? The value of `clength` would remain at 16.

In particular, describe how vectors would be used, and whether any changes would need to be made to the code above to nudge the compiler to generate code correctly.

Problem 4: [20 pts] In NVIDIA GPUs each thread has its own set of registers. Consider a modified version in which there were also warp registers. There would be one set of registers per warp. The threads in lane 0 could write a warp register, and the value could be read by any thread in the warp.

(*a*) Which variables below could easily be put into warp registers. DO NOT include registers that are compile-time constants.

```
__global__ void sums_2() {
  const int thread_count = blockDim.x * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int dcache_elts = ( 1 << 15 ) >> 2;
  const int nrounds = ( dapp.data_array_elts + dcache_elts -1 ) / dcache_elts;

  for ( int round = 0; round < nrounds; round++ ) {
      __shared__ float dcache[dcache_elts];
      const int chunk_start = round * dcache_elts;
      if ( round != 0 ) __syncthreads();
      for ( int sdidx = threadIdx.x; sdidx < dcache_elts; sdidx += blockDim.x )
        dcache[sdidx] = dapp.data_array[chunk_start + sdidx];
      __syncthreads();

      for ( int piece = tid; piece < dapp.num_pieces; piece += thread_count )
        {
          const int idx_piece_start = piece * clength;
          float sum = 0;
          for ( int i=0; i<clength; i++ )
            {
              const int didx = dapp.idx_array[idx_piece_start+i];
              const unsigned int sidx = didx - chunk_start;
              if ( sidx < dcache_elts ) sum += dcache[sidx];
            }
          dapp.sum_array[piece] += sum;
        }}}
```

(*b*) Consider the load instruction that would be used for loading from `data_array`:

```
 LD.E R11, [r1];
```

The value of `r1` is computed from `data_array`, `chunk_start` and `sdidx`. That would seem to make it impossible to replace `r1` with a warp register (say `w1`). Explain how it might be possible to use a warp register in place of `r1`, and explain how the `LD` instruction would operate.

Problem 5: [20 pts] Appearing below is the `at_start` routine from the pre-final.

```
__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;
    }
}
```

(*a*) Show values for `idx_array` that would result in worst-case performance due to shared memory issues.

(*b*) Suppose such patterns happened frequently. Modify the code to fix the problem (but don't change the results computed by the code). *Hint: That would mean changing the order in which things are placed.*