Name _____

GPU Micorarchitecture
EE 7722
Take-Home Pre-Final Examination

Thursday, 30 April 2015 to Monday, 4 May 2015

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	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(25 pts)
Problem 5	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing below is the assembly code for the `mxv_o_per_thd` kernel from Homework 2, along with the latency analysis from the solution. Based on the analysis provided in the solution, 883 threads would be needed to saturate instruction issue. (Rounding up to a multiple of warp size, the number of warps would be 28, which is 896 threads.)

In this problem, the impact of hypothetical prefetch instructions on the design of an NVIDIA-like GPU will be analyzed. The hypothetical design has a 16 kiB L1 data cache that can back the global address space.

The hypothetical instruction `PREFETCH [R6 + 0x20]` will compute an address by adding the contents of register pair `R6/R7` to the immediate, `0x20`, and then load the contents of global memory at that address into the cache. Similar to load instructions, it will take 200 cycles for the data to arrive in the cache. If a load instruction hits the cache dependent instructions can issue 12 cycles later, otherwise use a 200-cycle latency.

A goal of these prefetch instructions will be to reduce the number of registers needed.

```
Disassembled code for the mxv_o_per_thd kernel from Homework 2.
Column headings:
    AA: Time that instruction will issue.
   BBB: Registers modified by instruction
 CCCCC: Earliest time that dependent instruction can issue.

.L_2:                                          AA BBB    CCCCC
     SHF.L R24, RZ, 0x3, R0;                    0 R24    at 12
     BFE R3, R0, 0x11c;                         1 R3     at 13
     IADD R0, R0, R22;                          2 R0     at 14
     IMAD.U32.U32.HI R2, R3, 0x4, R24;         13 R2     at 25
     SHR R4, R2, 0x2;                          25 R4     at 37
     IMAD R2.CC, R4, R19, c[0x3][0x130];       37 R2     at 49
     IMAD.HI.X R3, R4, R19, c[0x3][0x134];     49 R3     at 61
     LD.E.128 R8, [R2];                        61 R8-R11 at 261
     LD.E.128 R4, [R2+0x10];                   62 R4-R7  at 262
     FMUL R9, R9, R16;                        261 R9     at 273
     FMUL R25, R5, R14;                       262 R25    at 274
     FFMA R5, R13, R8, R9;                    273 R5     at 285
     FFMA R8, R23, R4, R25;                   274 R8     at 286
     ISETP.LT.AND P0, PT, R0, c[0x3][0x0]     275 P0     at 287
     FFMA R4, R10, R17, R5;                   285 R4     at 297
     IADD R3, R24, R12;                       286 R3     at 298
     FFMA R6, R6, R15, R8;                    287 R6     at 299
     FFMA R4, R11, R21, R4;                   297 R4     at 309
     IMAD R2.CC, R3, R20, c[0x3][0x128];      298 R2     at 310
     FFMA R5, R7, R18, R6;                    299 R5     at 311
     F2F.F32.F32 R4, R4;                      309 R4     at 321
     IMAD.HI.X R3, R3, R20, c[0x3][0x12c];    310 R3     at 322
     FADD R4, R4, R5;                         321 R4     at 333
     ST.E [R2], R4;                           333 No output
 @P0 BRA `(.L_2);                             334 No output
```

(*a*) Compute the prefetch distance that should be used, measured in iterations, bytes, or elements. For this part assume that just one prefetch instruction is added, and that no other instructions need to be added. (In other words, assume that the prefetch distance is a compile-time constant to which the immediate will be set. This assumption will be discarded in another subproblem.) *Hint: First work out the distance in iterations, then compute distance in bytes or* `d_app.d_in_f4` *elements.*

(*b*) Compute the reduction in the number of registers needed due to prefetch.

(*c*) Prefetch does require a cache, of course. Compute the amount of cache needed based on your solution to the previous problems. Note that the amount of cache depends on the prefetch distance as well as the number of threads. Re-compute the amount of cache needed for a 400-cycle memory latency.

(*d*) In the analysis above we assumed that the only thing we would add to the code would be the prefetch instruction, no other instructions would be added. This won't work for this code example. In other words, we can't just prefetch constant offset. Explain why.

Problem 2: [25 pts] Appearing below is a simplified version of the solution to Homework 1. The simplifications were based on constraining `M` and `N` to multiples of `CS`.

```
extern "C" __global__ void mxv_sh_ochunk() {
  const int CS = 8;  // Chunk Size: Number of input vector elts to read.
  const int num_threads = blockDim.x * gridDim.x;
  const int bl_start = blockIdx.x * blockDim.x / CS;
  const int stop = d_app.num_vecs;
  const int inc = num_threads / CS;
  const int thd_c_offset = threadIdx.x % CS;
  const int thd_r_offset = threadIdx.x % CS;
  const int thd_v_offset = threadIdx.x / CS;

  const int MAX_BLOCK_SIZE = 1024;
  __shared__ Elt_Type vxfer[MAX_BLOCK_SIZE];

  // Number of output vector components assigned to each thread.
  const int ML = ( M + CS - 1 ) / CS;

  for ( int hb = bl_start; hb<stop; hb += inc ) {
      // Vector number assigned to this thread and its CS-1 partners.
      const int vec_num = hb + thd_v_offset;

      // Initialize storage for output vector components.
      Elt_Type vout[ML];
      for ( int rl=0; rl<ML; rl++ ) vout[rl] = 0;

      /// Compute Output Vector
      // Each iteration of the c loop uses CS input vector components
      // to update ML output vector components (per thread).
      //
      for ( int c=0; c<N; c += CS ) {
          // Read in 1 component of input vector and place in shared memory.
          vxfer[threadIdx.x] = d_app.d_in[ vec_num * N + c + thd_c_offset ];

          // Transfer the CS (8) components just read to local memory.
          Elt_Type vin[CS];
          for ( int cc=0; cc<CS; cc++ )
            vin[cc] = vxfer[ thd_v_offset * CS + cc ];

          // Using the 8 components just read, update our ML output
          // vector components.
          for ( int rr=0; rr<ML; rr++ )
            for ( int cc=0; cc<CS; cc++ )
              vout[rr] += d_app.matrix[ rr*CS + thd_r_offset ][c+cc] * vin[cc]; }

      /// Write Output Vector
      for ( int rr=0; rr<ML; rr++ )
        d_app.d_out[ vec_num * M + rr * CS + thd_r_offset ] = vout[rr];
    }}
```

Problem 2, continued:

(*a*) In the homework `CS` was fixed at 8, in this problem we will consider a range of values. For which values of `CS` will the code below make efficient access to global memory. Explain.

```
for ( int c=0; c<N; c += CS ) {
    vxfer[threadIdx.x] =
      d_app.d_in[ vec_num * N + c + thd_c_offset ];
```

(*b*) Estimate the latency of one iteration of the `h` loop. Show the latency in terms of N, M, and C (in mathematical expressions use C instead of `CS`). Assume that global loads have a latency of 200 cycles and that all other instructions have a latency of 12 cycles. Base the latency estimate on the high level code, **NOT** on the assembly language. Include global- and shared-memory loads and stores, as well as floating point arithmetic. Ignore things like address arithmetic and loop index handling. This should NOT be a tedious question. Assume that there are an unlimited number of registers and so decreasing or increasing `CS` will not cause a radical change in code due to register exhaustion.

(*c*) The answer to the question above should show that as `CS` is increased, latency is decreased. That's usually a good thing. Explain why throughput (for example, matrix/vector multiplies per second) *decreases* as `CS` increases. As before, assume that there are an unlimited number of registers. The answer has nothing to do with having enough warps to hide latency.

Problem 3: [15 pts] Once again consider the solution to Homework 1.

The input vector is read from global memory, written to shared array `vxfer` and then read into local array `vin`. Show how warp shuffle instructions (sometimes called swizzle instructions in class) can be used to avoid the use of shared memory. For a description of warp shuffle instructions, and the intrinsics needed to use them see Appendix B of the CUDA C Programming Guide.

Problem 4: [25 pts] Consider the following attempt at optimization of the code in the Homework 2 solution. The idea is to examine the value of components of the input matrix and to avoid multiplication if the value is 1, and to avoid even updating the output for values of 0. Suppose that input components have a value of 1 with probability $p_1 = .1$ and they have a value of 0 with probability $p_0 = .2$. The probabilities are independent, meaning that if knowing the first component of input vector 27 has a value of 0 does not help in determining the value of any other components of that or any other vector.

For the questions below use `CS = 8`.

```
// Unoptimized Loop Nest
for ( int rr=0; rr<ML; rr++ )
  for ( int cc=0; cc<CS; cc++ )
    vout[rr] += d_app.matrix[ rr*CS + thd_r_offset ][c+cc] * vin[cc];

// Optimized: Check for special cases.
for ( int cc=0; cc<CS; cc++ )
  if ( vin[cc] == 0 )
    {
      // Do nothing.
    }
  else if ( vin[cc] == 1 )
    {
      // Avoid multiplication.
      for ( int rr=0; rr<ML; rr++ )
        vout[rr] += d_app.matrix[ rr*CS + thd_r_offset ][c+cc];
    }
  else
    {
      // Perform the original calculation.
      for ( int rr=0; rr<ML; rr++ )
        vout[rr] +=
          d_app.matrix[ rr*CS + thd_r_offset ][c+cc] * vin[cc];
    }
```

(*a*) Estimate the impact of this optimization on iteration latency. This can be based in part on the latency analysis performed earlier. Do this by showing the latency of the original code, the latency of the "optimized" code under certain scenarios (based on encountered values of `vin`), and the expected latency of the "optimized" code.

(*b*) Removing one of the two special cases should improve performance. Which one should be removed, and why would performance be better?

(*c*) Suppose that if two consecutive components of an input vector were zero then the remainder of the components would be zero. In other words let $v = [v_0, v_1, \ldots, v_{N-1}]$ be some input vector. If, say, $v_2 = 0$ and $v_3 = 0$ then elements $v_4 = 0$, $v_5 = 0$, etc. The code below has an "optimization" intended to take advantage of this property. Explain why it would not work and find a better way to exploit the two-consecutive element property.

```
// Optimized: Check for special cases.
for ( int cc=0; cc<CS; cc++ )
  if ( vin[cc] == 0 )
    {
      // Do nothing.
      if ( cc < CS - 1 && vin[cc+1] == 0 ) break;
    }
  else if ( vin[cc] == 1 )
    {
      // Avoid multiplication.
      for ( int rr=0; rr<ML; rr++ )
        vout[rr] += d_app.matrix[ rr*CS + thd_r_offset ][c+cc];
    }
  else
    {
      // Perform the original calculation.
      for ( int rr=0; rr<ML; rr++ )
        vout[rr] +=
          d_app.matrix[ rr*CS + thd_r_offset ][c+cc] * vin[cc];
    }
```

Problem 5: [10 pts] Answer each question below.

(*a*) In current NVIDIA processors the number of functional units per MP varies by type. For example, there are 192 single-precision floating point units but only 32 integer multiply units. Explain how it would be more difficult to vary the number of functional units by type in a system using temporal multithreading (as described for Echelon). Describe how it might be possible to vary the number of functional units with additional hardware.

(*b*) In NVIDIA designs the large number of threads make up for the lack of prefetch instructions. The Phi has prefetch instructions, and yet it still has 4 thread contexts per core. Why does it need more than one thread per core?