Name ______________________________

GPU Microarchitecture
EE 7722
Take-Home Final Examination

Monday, 1 May 2017 to Friday, 5 May 2017

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	________	(35 pts)
Problem 2	________	(30 pts)
Problem 3	________	(10 pts)
Problem 4	________	(10 pts)
Problem 5	________	(15 pts)
Exam Total	________	(100 pts)

Alias ______________________________

Good Luck!

Problem 1: [35 pts] In class a GPU radix sort program was presented that operated in two passes per digit. The first pass computed histograms and the second pass moved the elements to be sorted. See the slides presented in class `http://www.ece.lsu.edu/gp/notes/set-radix.pdf`. The radix sort code itself is in the repo in files cuda/sort/radix-sort-kernel.cu and cuda/sort/radix-sort.cc .

(*a*) In the analysis presented in the slides and in class the keys are written in both pass 1 and pass 2. Suppose instead that in pass 1 the kernel only writes the block histogram (it no longer writes the keys or tile histograms). To compute the block histogram it will still sort tiles and compute tile histograms, but it won't write the sorted tiles. The pass 2 kernel will load a tile of keys, sort them, write them out and update the prefix sum for the next tile.

This variation performs less data transfer put more computation.

☐ Re-do the data and computation analysis presented in the slides for this variation.

☐ Assuming a computation to communication ratio of 55 instructions per 32-bit (integer) transfer, determine a value of c_s for which this variation will work better. ☐ State assumptions such as block size.

(*b*) The analysis presented in the slides ignored request utilization. As a result it favored small block sizes (to reduce tile sort time) and a large radix (to reduce the number of rounds). Modify the analysis to take into account the request size. Make reasonable assumptions such as assuming keys are random but uniformly distributed over the range 0 to $2^{32} - 1$.

☐ Modify sort analysis taking into account request size.

Problem 2: [30 pts] The code fragments below are based on the `vtx-xform-sum` used in class. Remember that in this code each thread computes a value (in routine `reduce_thread`) and different methods are used to compute a global sum (a sum over all threads).

```
extern "C" __global__ void reduce_thd_atomic_blk() {
  const Elt_Type thd_sum = reduce_thread();
  __shared__ Elt_Type our_sum;
  if ( threadIdx.x == 0 ) our_sum = 0;
  __syncthreads();   // The first syncthreads.
  atomicAdd( &our_sum, thd_sum );
  __syncthreads();   // The second syncthreads.
  if ( threadIdx.x == 0 ) d_app.d_thd_sum[blockIdx.x] = our_sum;
}
```

(*a*) In the version above an atomic add operating on shared memory is used to compute a block-wide sum. Explain what would happen if each `syncthreads` in the code were removed.

☐ Describe how result might be wrong if first `syncthreads` removed.

☐ Describe how result might be wrong if second `syncthreads` removed.

(*b*) Appearing below is the assembly code corresponding to the `atomicAdd` from the code above. The add itself is performed by an ordinary add instruction, `FADD`.

```
        /*0190*/                    BAR.SYNC 0x0;
        /*0198*/                    PSETP.AND.AND P2, PT, !PT, PT, PT;
        /*01a0*/                    SSY `(.L_76);
.L_77:  /*01c8*/                    LDSLK P1, R2, [RZ];
        /*01d0*/                @P1 FADD R2, R2, R9;
        /*01d8*/                @P1 STSCUL P2, [RZ], R2;
        /*01e0*/               @!P2 BRA `(.L_77);
        /*01e8*/                    NOP.S   (*"TARGET= .L_76 "*);
.L_76:  /*01f0*/                    BAR.SYNC 0x0;
```

☐ Explain how the other instructions ensure that the add is atomic.

☐ Explain why this method of performing an atomic add has poor performance in the CUDA code above.

Problem 2, continued: The code below is based on `reduce_atomic_sum_grid`, but there are three options. Option 0 corresponds to the code used in class. Note that two of the three option lines need to be commented out for the code to compile.

```
extern "C" __global__ void reduce_atomic_sum_grid() {
  const Elt_Type thd_sum = reduce_thread();
  const int idx = 0;                          // Option 0.
  const int idx = blockIdx.x;                 // Option 1.
  const int idx = threadIdx.x % gridDim.x;    // Option 2.

  atomicAdd( &d_app.d_thd_sum[idx], thd_sum );
}
```

Appearing below is the assembly code for the atomic add, with Option 0:

```
        /*0170*/                    MOV.S R2, c[0x3][0x78]
        /*0178*/                    MOV R3, c[0x3][0x7c];
        /*0188*/                    RED.E.ADD.F32.FTZ.RN [R2], R0;
        /*0190*/                    EXIT;
```

The table below shows the execution time of the three options, and for comparison the sum block code from the previous problem and the tree reduction code (named `reduce_thd_tree_blk`). The data was collected for a launch of 52 blocks on a GPU with 13 MPs, with an expected occupancy of 4 blocks per MP.

Kernel	Time / μs
reduce_thd_atomic_blk	377.152
grid Option 0	337.312
grid Option 1	295.520
grid Option 2	213.888
reduce_thd_tree_blk	208.576

(*c*) Provide possible reasons for the performance differences.

☐ Possible reason that Option 1 is faster than Option 0:

☐ Possible reason that Option 2 is faster than Option 1:

Problem 2, continued:

(*d*) Determine or estimate the amount of data transferred to and from the MPs for each of the five kernels above for the global `atomicAdd` operation or the write to `d_thd_sum`. Don't consider the data read or written by routine `reduce_thread`. (The first and last should be simple.) In your answer use B for the block size, G for the number of blocks, and M for the number of multiprocessors.

For the grid kernels, describe how you assume the hardware operates, then provide an estimate based on that assumption.

☐ Amount of data transferred for `reduce_thd_atomic_blk`.

☐ Amount of data transferred for `reduce_thd_tree_blk`.

☐ Describe assumed hardware implementing reduction.

☐ Amount of data transferred for Option 0 (based on assumption).

☐ Amount of data transferred for Option 1 (based on assumption).

☐ Amount of data transferred for Option 2 (based on assumption).

Problem 3: [10 pts] Suppose that most of the area and energy on a Phi chip were used by the vector units and vector registers. Consider an alternative Phi that uses a 1024 b vectors instead of 512 b vectors, but which has half the number of cores, 30 instead of 60. Assume that the alternative and original designs have the same clock frequency and so the same peak FP rate.

(*a*) Suppose that the cores in the alternative design are the same as in the original design except for the vector unit and registers. Doubling the vector size and halving the core count this way can hurt the performance of some programs.

☐ How might doubling the vector size reduce the performance of some programs?

☐ How might halving the number of cores this way reduce the performance of some other programs? (The performance loss should have a different cause than in the previous answer.)

(*b*) If the cores in the alternative design are the same except for the vector unit and registers, than the overall cost of the alternative design will be less. Suggest several ways for the cores in the alternative design to be changed to bring cost in line with the original design.

☐ In the alternative design the best thing to do is increase …

☐ In the alternative design one could also increase … but it would not be as good as the previous answer.

Problem 4: [10 pts] Answer each question below.

(*a*) Why does the Phi need branch prediction, but not NVIDIA GPUs? Illustrate with a diagram.

☐ Difference between NVIDIA GPUs and Phi that enables GPUs to omit branch prediction:

(*b*) In an NVIDIA CC 3.5 device each warp scheduler schedules at most 16 warps. But the block scheduler schedules up to $2^{31} - 1$ blocks.

☐ What would be the approximate cost and performance impact if the warp scheduler had to schedule from a larger number, say 32 warps? (The number of schedulers doesn't change.)

☐ Why can the block scheduler schedule a much larger number of blocks? ☐ What makes the block scheduler's job easier?

Problem 5: [15 pts] The code below uses indirection to load from an array named `TPairfunc`. Further below, is Phi code for the loop body (actually several unrolled iterations).

```
for ( int i=0; i<npairs; i++ )
{
  int othsite = plist[i];
  ans += TPairfunc[othsite] * PsiInv[i];
}
```

```
..LN1679: vpaddd    652736(%rdx,%r15){uint16}, %zmm6, %zmm0       #52.11 c1
..L193:   vgatherdpd (%rdi,%zmm0,8), %zmm2{%k1}                   #52.11
..LN1689: jkzd      ..L192, %k1   # Prob 50%                      #52.11
..LN1690: vgatherdpd (%rdi,%zmm0,8), %zmm2{%k1}                   #52.11
..LN1691: jknzd     ..L193, %k1   # Prob 50%                      #52.11
..L192:   vfmadd231pd 2368(%rsi,%rcx,8), %zmm2, %zmm8             #52.4 c33
```

(*a*) The `vpaddd` instruction is doing two things that an NVIDIA ADD instruction can't do. What are they.

☐ Operations done by `vpaddd` that would require additional NVIDIA instructions.

(*b*) Explain how the set of four instructions starting at L193 works.

☐ The four instructions ...

(*c*) The equivalent NVIDIA assembler code for the instructions starting at L193 would use just use address calculation instructions and one load instruction. Does that mean NVIDIA GPUs perform gathers faster than Phi? Ignore address computation in your answer.

☐ NVIDIA GPUs are ☐ faster or ☐ about the same speed. (Check one.)

☐ Explain.