Name _____

GPU Microarchitecture
EE 7722
Take-Home Final Examination

Thursday, 3 May 2018 to Sunday, 6 May 2018

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	_____	(30 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(50 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] The code appearing on the next page is based on the solution to Homework 4 in which the 1-bit split routine is improved. When `use_pop` is set to true a warp ballot operation will be used to find the prefix sum within a warp. Otherwise warp shuffle instructions will be used.

Analyze the execution of each. Assume instruction latency of these instructions is 6 cycles, but for throughput of the instructions use NVIDIA resources.

(*a*) Show assumed assembly language instructions for the code. Reasonable simplifications are okay. It is important to identify special machine instructions for operations such as the `ballot_sync`, `popc`, and `shfl_up_sync`. If necessary inspect SASS code and look for similar PTX instructions.

(*b*) Compute the latency of each variation from the start of the code fragment up to (but not including) the syncthreads. Use e for the value of `elt_per_thread`.

(*c*) Determine the number of threads needed to saturate instruction issue for each case.

Problem 1, continued: Code excerpt from 1-bit split.

```
// Initialize data for prefix sum of bit bit_pos, and make copy of key.
//
int my_ones_write = 0;

const bool use_pop = true;

const int wp_lg = 5;
const int wp_sz = 1 << wp_lg;
const int wp_mk = wp_sz - 1;
const int lane = threadIdx.x & wp_mk;
const int wp_idx = threadIdx.x >> wp_lg;
const uint32_t msk = 0xffffffff;
int my_pf_wp = 0;

for ( int i = 0; i < elt_per_thread; i++ )
  {
    const int sidx = threadIdx.x * elt_per_thread + i;

    // Make a copy of key.
    //
    const Sort_Elt key = p1s.keys[ sidx ];
    const bool one = key & bit_mask;
    const uint32_t have_work_wp_v = __ballot_sync(msk,one);
    // Shift off bits corresponding to higher-numbered lanes.
    const uint32_t have_work_pf_v = have_work_wp_v << ( 31 - lane );
    const uint32_t my_pf_wp_i = __popc(have_work_pf_v);
    my_pf_wp += my_pf_wp_i;
  }

if ( lane == wp_mk ) p1s.prefix[wp_idx+1] = my_pf_wp;
__syncthreads();
```

Problem 2: [20 pts] Answer the following questions about TSIMT as described in Lucas TACO 15 a32.

(*a*) The instruction-to-instruction latency of dependent operations (what we call latency in class) of NVIDIA GPUs have dropped, from 22 in the Fermi generation to about 11 in Kepler to 6 in Pascal, and down to 4 in Volta, at least the scientific-computing priced GV100. Reducing latency might increase costs by requiring bypass networks. Is reducing latency as important in TSIMT? Explain.

(*b*) The TSIMT core presented in the paper is based on CC 1.x (Tesla) generation GPUs. It matches the instruction bandwidth of 8 FP instructions per cycle per SM. Design a TSIMT core for a Pascal 6.0 device (say, a GP100), lets call it the TiP core. Assume that the warp size is something specified in a launch. Note that the GP100 has 64 FP functional units per SM, not 128 as in other Pascal and Maxwell devices. The TiP core should match the 6.0 SM in FP bandwidth, the number of thread contexts, the number of registers, etc. Other reasonable changes can be made, such as whether a lane is scalar or can issue multiple threads per cycle (spatiotemporal).

Explain why in such a design it is important to have a user-configurable warp size, especially for when the number of threads per SM is low.

Problem 3: [50 pts] An advantage of TSIMT not mentioned in the paper (and perhaps anywhere else) is that it is possible to execute reduction- and prefix-like calculations across the threads in a warp. Consider the prefix calculation needed for the 1-bit split sort. The amount of work for B elements in a serial implementation is $B-1$ additions. But the amount of work in our parallel prefix computation is $O(B \lg B)$ operations. Since the threads in a warp execute sequentially in a TSIMT organization it is possible to design it so that an instruction in warp position i uses a value computed by the same instruction for warp position $i-1$. (The first thread in a warp is at position 0, the second thread is at position 1, etc. In SIMT GPUs a warp position is called a lane, but lane has a different meaning for TSIMT designs.) Call a TSIMT design with that functionality *TiY* (the Y is for bypass, since results would be bypassed from one thread to another [executing the same instruction] in a warp). TiY instructions can use a special source register, `R<`. The value of `R<` for the thread in warp position i is the result computed by the thread in warp position j where $j < i$ is the highest numbered active thread. If i is the highest-numbered active thread in the warp the value of `R<` is 0. (This initial value might vary by instructions, such as 1 for a multiply instruction, -1 for an AND, etc.)

A TiY can compute a prefix sum for values in a warp using just one add instruction. Call this the *bypass method*. For example,

```
// High level serial code.
p[0] = 0;
for ( int i=1; i<32; i++ ) p[i] = p[i-1] + a[i];


// Thread i operates on element a[i].
// R2 contains a[i], R< is p[i-1], R1 will be p[i]
//  R< is value written to R1 by thread i-1, which is p[i-1]
IADD R1, R<, R2
```

The amount of work for a sum in a 32-thread warp is just 32, rather than $32 \times 5c$, where c is the number of instructions per iteration. (See the first problem.) For TiY not only is the amount of work less but each step is just one instruction, rather than the several instructions needed to implement the reduction tree.

Wow, this sounds great! In this problem take a closer look at the idea and analyze it for conventional CC 6.0 GPUs and TiY GPUs which roughly match Pascal. In 6.0 there are only 64 FP32 units per SM, so each warp has its own lane.

The example above used an integer add instruction, not just because that's what was needed in the radix sort but because an integer sum can be computed in one cycle and so the thread in warp position i could execute in the cycle after warp position $i-1$ when using `R<` registers.

(*a*) Before designing exotic TiY hardware, lets see if we can get the same work efficiency of the bypass method using existing GPU designs or proposed TSIMT designs.

If the goal is to compute the prefix of 32-element arrays efficiently, why not assign one thread to each 32-element array? That is, write the entire array (the values of `my_ones_write` in the radix sort examples) to shared memory and then assign one thread to each 32-element chunk. For example, thread 0 handles elements 0-31, thread 1 handles 32-63, etc.

Those threads will be executing a loop unrolled into 31 add instructions plus whatever else is needed.

Compare the execution time of this conventional method to the bypass method. Be sure to show assumed instructions (it won't just be adds) and state any other assumptions made.

☐ Show assumed code and analyze execution time.

☐ Discuss how significant the differences are: What features give TiY a big advantage? Are we assuming capabilities for TiY hardware that could easily be incorporated in conventional designs (or should not be made for TiY)?

(*b*) The bypass method described above efficiently finds the prefix sum within a warp. How should the prefix sum between warps be computed in a TiY? Assume 32-warp blocks of 32-thread warps and a two resident blocks per SM. Based on an assumed number of instructions or syncthreads latency, try to find a break-even point where the bypass method and the tree method take the same amount of time for the inter-warp prefix. (Don't forget that the amount of work to do for the inter-warp computation is much less since each block computes a prefix of $B/32$ items. In the intra-warp prefix computation, done using the bypass method, each block computes $B/32$ prefix sums, each of 32 items.)

TSIMT can execute a parallel prefix computation more efficiently than conventional GPUs. This won't work for the method used in the class code. Either base your answer on the classroom code, or find a method that uses $O(N)$ work to find the parallel prefix of N elements.

- [ ] Outline method used.
- [ ] Estimate execution time of inter-warp prefix.
- [ ] State assumptions, such as the time for syncthreads.

(*c*) A TSIMT design makes it easy to change the warp size, of course the same applies to TiY. Find the optimal warp size, W, for a block of size B threads when the bypass method is used for the inter- and intra-warp prefix sums.

☐ Optimal warp size in terms of B.

(*d*) Suppose we want to use bypass registers to perform reductions with operations that have a latency greater than one cycle, such as FP operations. Describe how an instruction like `FADD.RED R1, R2` would operate assuming that `FADD` has an L-cycle latency. The instruction should write `R1` with the sum of all warps' thread values of `R2`. Show how what operations would be issues, and how long before the result is written. Use W for the warp size. It should go without saying that $(W-1)L$ is way too long. It's okay to assume that W/L is an integer.

☐ Describe how reduction would be implemented.

☐ How many cycles are needed to issue all operations to the adder?