Name _____

GPU Microarchitecture
EE 7722
Solve-Home Final Examination

Monday, 6 May 2024 to Friday, 10 May 2024 16:00 CDT

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	_____	(30 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] Appearing below is `norm_base`, the original version of the normalization kernel from Homework 1. Recall that the kernel accesses each element of `l_in` twice. If the cache were large enough the second access would hit the L1 cache, otherwise it can miss. Further below is a kernel making a misguided attempt to avoid the miss on the second access.

```
template< int d_l > __global__ void
norm_base(elt_t* __restrict__ l_out, const elt_t* __restrict__ l_in){
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;
  const int n_l = c_app.n_l;
  for ( int h = tid;  h < n_l;  h += n_threads ) {
      elt_t sum = 0;
      for ( int i = 0;  i < d_l;  i++ ) sum += l_in[ h * d_l + i ];
      const elt_t avg = sum / d_l;
      for ( int i = 0;  i < d_l;  i++ ) l_out[ h * d_l + i ] = l_in[ h * d_l + i ] - avg;
    }}
```

(*a*) The kernel below places the `l_in` elements into shared memory. Template parameter `d_l` is the vector length (it is never zero, as it was in the assignment), and parameter `BLOCK_SIZE` is the number of threads per block. Put in the array size and initialize variables `so` and `sc` so that the code runs correctly and efficiently. (See next problem about how efficiently it can run.)

```
template< int d_l, int BLOCK_SIZE > __global__ void
norm_fe_sh(elt_t* __restrict__ l_out, const elt_t* __restrict__ l_in)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  assert( blockDim.x == BLOCK_SIZE );
  const int n_threads = blockDim.x * gridDim.x;
  const int n_l = c_app.n_l;

  __shared__ elt_t l_sh[           ];   // ☐ Put in expression for size of lsh.

  const int so = 1;                     // ☐  Code for correct and efficient l_sh access.


  const int sc = 1;                     // ☐  Code for correct and efficient l_sh access.


  for ( int h = tid;  h < n_l;  h += n_threads )
    {
      elt_t sum = 0;
      for ( int i = 0;  i < d_l;  i++ )
        {
          elt_t elt = l_in[ h * d_l + i ];
          l_sh[ so + i * sc ] = elt;
          sum += elt;
        }
      const elt_t avg = sum / d_l;
      for ( int i = 0;  i < d_l;  i++ )
        l_out[ h * d_l + i ] = l_sh[ so + i * sc ] - avg;
    }
}
```

Problem 1, continued: Suppose `norm_fe_sh` from the previous problem is completed correctly. It probably still won't be faster than `norm_base` and can be slower.

(*b*) Why might `norm_fe_sh` be slower than the original kernel even when all cache accesses are hits. (The answer is obvious, though overlooked by beginners.)

☐ `norm_fe_sh` is slower because:

(*c*) What is it about shared memory in current GPUs (CC 8.0, 8.5, 8.9, 9.0, and some earlier generations) that makes it unlikely that `norm_fe_sh` can successfully avoid misses that would occur in the second access to `l_in` in the original kernel? *If necessary review notes or lookup how shared memory is implemented in hardware. One good reference is the Compute Capabilities chapter of the Nvidia C Programmers Guide. Think a little bit outside the box. The answer is easy to understand.*

☐ Don't expect to avoid misses in your `norm_base` by switching to a completed `norm_fe_sh` because:

Problem 2: [30 pts] Appearing below is the solution to Homework 2.

```
template<int D_L = 0, int grp_size = 1, int unroll_degree = 1>
__global__ void norm_fe_group_u(elt_t* __restrict__ l_out, const elt_t* __restrict__ l_in)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;
  const int d_l = D_L ?: c_app.d_l, n_l = c_app.n_l;
  const int sub_lane = threadIdx.x % grp_size;
  constexpr int wp_lg = 5, wp_sz = 1 << wp_lg;
  const int lane  = threadIdx.x % wp_sz, wp_id = tid >> wp_lg;

  constexpr int vec_p_wp = wp_sz / grp_size;
  const int vec_p_wp_iter = vec_p_wp * unroll_degree;
  const int h_wp_start = wp_id * vec_p_wp_iter,   inc = unroll_degree * n_threads / grp_size;

  for ( int h_wp = h_wp_start;  h_wp < n_l;  h_wp += inc ) {
      const int hi = h_wp + lane / grp_size;
      elt_t thd_sum[unroll_degree]{};

      for ( int j = 0;  j < unroll_degree;  j++ )
        {
          const int h = hi + j * vec_p_wp;
          const size_t idx_vec_start = h * d_l;
          const size_t idx_vec_thd_start = idx_vec_start + sub_lane;

          // Reduction Loop
          for (int i=0;  i<d_l;  i+=grp_size ) thd_sum[j] += l_in[ idx_vec_thd_start + i ];
        }

      elt_t sum[unroll_degree]{};
      for ( int j = 0;  j < unroll_degree;  j++ ) sum[j] = group_sum(thd_sum[j],grp_size);

      for ( int j = 0;  j < unroll_degree;  j++ )
        {
          const int h = hi + j * vec_p_wp;
          const size_t idx_vec_start = h * d_l;
          const size_t idx_vec_thd_start = idx_vec_start + sub_lane;
          const elt_t avg = sum[j] / d_l;

          // Normalization Loop
          for ( int i = 0;  i < d_l;  i += grp_size )
            l_out[ idx_vec_thd_start+i ] = l_in[ idx_vec_thd_start + i ] - avg;
        }
    }
}
```

(*a*) What is the minimum L1 cache size for which the `l_in` access in the Normalization Loop hits the L1 cache? Answer in terms of δ (unroll degree), d_l (vector length), g (group size), and B (number of threads per block). Assume that the number of blocks equals the number of SMs.

☐ In terms of δ, d_l, g, and B, the minimum cache size is:

Problem 2, continued:

(*b*) Consider two configurations of `norm_group_u`. Configuration X is launched with $B \geq 256$ threads per block, and an unroll degree of δ. Configuration Y is launched with $B/2$ threads per block and an unroll degree of 2δ. For both configurations n_l is large, $d_l = g = 32$, and the number blocks is equal to the number of SMs.

☐ Might Y run ◯ *at least* 1.5× *faster than X*, ◯ *at least* 1.5× *slower than X*, ◯ *or at about the same speed as X.* ☐ Explain.

(*c*) Consider a system with 114 SMs. In configuration X $B = 256$, $d_l = g = 32$, $\delta = 1$, and $n_l = 912$. Configuration Y is the same except $B = 512$. In both cases 114 blocks are launched. (The unroll degree is **not** changed.).

☐ Will Y run ◯ *faster than X*, ◯ *slower than X*, ◯ *or at about the same speed as X.* ☐ Explain.

(*d*) Consider a system with 114 SMs. In configuration X $B = 256$, $d_l = g = 32$, $\delta = 1$, and $n_l = 912$. Configuration Y is the same except $\delta = 2$. (The block size is **not** changed.).

☐ Given the way the homework code performed, will Y run ◯ *faster than X*, ◯ *slower than X*, ◯ *or at about the same speed as X.* ☐ Explain.

(*e*) Consider a system with 114 SMs. In configuration X $B = 256$, $d_l = 1024$, $g = 32$, $\delta = 1$, and $n_l = 912$. Configuration Y is the same except $\delta = 2$. (The block size is **not** changed.).

☐ Given the way the homework code performed, will Y run ◯ *faster than X*, ◯ *slower than X*, ◯ *or at about the same speed as X.* ☐ Explain.

Problem 3: [40 pts] Lie 23 Micro Magazine [1] describes the Cerebras CS-2 system including the WSE-2, a wafer-scale processor designed for machine learning workloads. In the Core Architecture section of the paper on the right column of page 19 a paragraph boldly starts out:

> *Instead of the traditional approach, the Cerebras architecture provides full memory bandwidth to all data paths at full performance, removing the need for data reuse. [1]*

The phrase *full performance* probably means that the floating-point bandwidth of four FP16 operations per cycle per PE can be sustained even when sourcing operands from *memory*, with memory appropriately defined. For this problem memory will refer to any storage holding operands, not just global memory.

(*a*) Can the memory referred to in the quote include MemoryX, the off-wafer system described as a good place to store weights? Explain why or why not.

☐ Can memory from the quote include MemoryX? ☐ Explain.

(*b*) Are bank conflicts the reason that Nvidia GPUs access data at lower than full memory bandwidth, a problem that Cerebras has avoided? *Hint: The answer is no, for an obvious reason.*

☐ Are bank conflicts what slow Nvidia GPUs down, something that the WSE-2 avoids? ☐ Explain.

(*c*) Consider code on an Nvidia GPU (say, CC 8.9) accessing data in shared memory. Suppose that the input data needed is in shared memory before the code starts and that the result data need only be written to shared memory. What fraction of "full performance" as used in the quote can a kernel on an Nvidia GPU reach when operating on data in shared memory. A fraction of $\frac{1}{4}$ would mean one quarter the peak floating point bandwidth of 128 FP32 operations per cycle per SM. There is no single answer to this question. Provide an answer that is reasonably fair. Illustrate your answer with a short code fragment. (The interval analysis notes, `https://www.ece.lsu.edu/gp/2024/lsli05-iv.pdf`, may be helpful, though they are more concerned with choosing the number of threads to saturate performance rather than what that performance will be.)

☐ What fraction of "full performance" can be reached on an Nvidia CC 8.9 GPU sourcing data from shared memory. ☐ Show a short code sample to justify your answer.

(*d*) For the Nvidia GPUs the fraction of "full performance" should be less than 1. What feature of the WSE-2, one that Nvidia GPUs lack, enables code in a PE run at full performance when accessing data only within the PE? Base the answer to this question on what kept the Nvidia code from reaching full performance in the previous problem. This feature is described in the Core Architecture section of [1].

☐ What feature enables WSE-2 PE code to run at full performance when reading data in the PE SRAM?

(*e*) The "full performance" claim is easy to justify when operands are retrieved from the local SRAM. Can code on the WSE-2 sustain full performance when it reads operands that are obtained from neighbors? Such data will have to pass through the network, something the WSE-2 does efficiently.

☐ Describe a computation using neighbor's data that can be sustained at full speed. (Consider the operands needed by arithmetic instructions.)

☐ Describe a computation using neighbor's data that cannot be sustained at full speed.

☐ Consider a computation in which data every ten iterations comes from a PE at a distance of 5 PEs to the left. ☐ Why is full performance attainable despite the latency?

☐ Consider a computation in which data at each iteration comes from a PE at a distance of 5 PEs to the left. ☐ Why would bandwidth prevent full performance?

References:

[1] Lie, S. Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning. *IEEE Micro 43*, 3 (2023), 18–30. http://dx.doi.org/10.1109/MM.2023.3256384.