Name _____

GPU Microarchitecture
EE 7722
Solve-Home Final Examination

Thursday, 8 May 2025 to Sunday, 11 May 2025 23:59 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	_____	(20 pts)
Problem 2	_____	(50 pts)
Problem 3	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] Appearing below is an excerpt from the solution to Homework 4. Many student submissions called `syncthreads` once, at position `B` (see the statement labels on the left-hand side). (Few student solutions used `group_sum` to compute the group sums.) The excerpt below calls `syncthreads` three times to avoid problems that are sneaky enough to hide when you are testing your code.

```
__shared__ elt_t ssum[32];
for ( int h = h_start;  h < n_l;  h += n_threads / grp_size ) {
    I0: const size_t idx_vec_start = h * d_l;
    const size_t idx_thd_start = idx_vec_start + sub_lane;
    elt_t thd_sum = 0;
    for ( int i = 0;  i < d_l;  i += grp_size ) thd_sum += l_in[ idx_thd_start + i ];

    I1: const elt_t wp_sum = group_sum(thd_sum,32);

    A: __syncthreads();         // Wait until everyone has read ssum.
    I2: if ( !lane ) ssum[wp_idx] = wp_sum;

    B: __syncthreads();         // Wait until all warps have written ssum.
    I3: if ( !wp_idx ) ssum[threadIdx.x] = group_sum( ssum[threadIdx.x], grp_wps );

    C: __syncthreads();         // Wait until warp 0 has written ssum.
    I4: elt_t vec_sum = ssum[wp_idx];

    I5: const elt_t avg = vec_sum / d_l;
    for ( int i=0; i<d_l; i+=grp_size ) l_out[idx_thd_start+i]=l_in[idx_thd_start+i]-avg;
  }
```

Appearing below is a timing diagram showing when each of two warps, `wp0` and `wp1`, execute the labeled lines above. Similar diagrams used in class were intended to show instruction-level effects such as thread dispatch time and operation latency. But here the diagram shows lines of code (rather than machine instructions) for the purpose of showing the order of execution, thread dispatch time and operation latency are not part of this problem. The position of a line, such as `I1`, indicates when it has completed. Though operation latency is not part of the problem the impact of `syncthreads` is. The symbol for a line including `syncthreads`, such as `A` must be shown occurring at the time for all warps. (The reality is a bit more complicated: no warp can exit a `syncthreads` until all warps have entered it.)

The two warps run on different schedulers so they can run lines at the same time, such as `I0`. In the example below warp `wp0` finishes `I1` earlier than `wp1`, but is forced to wait by the call to `syncthreads` labeled `A`.

In the diagrams below `wp0` and `wp1` are part of the same group.

```
wp0:  I0  I1          A       I2  B   I3       C I4   I5  I0   I1   A  I2 B  I3  C  I4 I5
wp1:  I0        I1   A  I2       B   I3       C I4   I5  I0   I1   A  I2 B  I3  C  I4 I5
```

Continued on the next page.

Problem 1, continued: As requested below, show timing examples which can result in incorrect values of `vec_sum`, or if that's not possible explain why the `syncthreads` is unnecessary. *Note: In the original exam the there was no option to explain why execution would always be correct.*

☐ The `syncthreads` at `A` is removed (shown by an `X`). Either explain why execution will always be correct despite the removal, or complete the timing diagram below to show an execution order that would result in an incorrect value in `vec_sum` for at least one thread, as a result of the removal.

```
wp0:   I0   I1   X  I2  B   I3   C   I4   I5  I0   I1   X  I2  B   I3   C   I4   I5

wp1:
```

☐ The `syncthreads` at `B` is removed (shown by an `X`). Either explain why execution will always be correct despite the removal, or complete the timing diagram below to show an execution order that would result in an incorrect value in `vec_sum` for at least one thread, as a result of the removal.

```
wp0:   I0   I1   A  I2  X   I3   C   I4   I5  I0   I1   A  I2  X   I3   C   I4   I5

wp1:
```

☐ The `syncthreads` at `C` is removed (shown by an `X`). Either explain why execution will always be correct despite the removal, or complete the timing diagram below to show an execution order that would result in an incorrect value in `vec_sum` for at least one thread, as a result of the removal.

```
wp0:   I0   I1   A  I2  B   I3   X   I4   I5  I0   I1   A  I2  B   I3   X   I4   I5

wp1:
```

Problem 2: [50 pts] Appearing below is an excerpt from the solution to Homework 5. The solution is in the repository and available in HTML form via `https://www.ece.lsu.edu/gp/2025/hw05-sol.cu.html`.

```
constexpr uint warp_tile_nrows = tm * m_ht_n;
constexpr uint warp_tile_ncols = tn * m_wd_n;
const int blk_wp_idx = threadIdx.x / wp_sz;
const int C_row_0_offset = ( blk_wp_idx / n_wps_col ) * warp_tile_nrows;  // Relative to block tile.
const int C_col_0_offset = ( blk_wp_idx % n_wps_col ) * warp_tile_ncols;  // Relative to block tile.
const int n_block_tiles_col = div_ceil( C_ncols, block_tile_ncols );

for ( int i = blockIdx.x; true; i += gridDim.x ) {
    // Compute upper-left row and column of section computed by entire block.
    const int C_col_0_block = i % n_block_tiles_col * block_tile_ncols;
    const int C_row_0_block = i / n_block_tiles_col * block_tile_nrows;

    // Compute upper-left row and column of section computed by entire warp.
    const int C_col_0 = C_col_0_block + C_col_0_offset;
    const int C_row_0 = C_row_0_block + C_row_0_offset;

    fragment<accumulator, tm, tn, tk, float> tile_C_acc[m_ht_n][m_wd_n];
    for ( auto& tca: tile_C_acc ) for ( auto& tcb: tca ) fill_fragment(tcb,0);

    for ( ssize_t i_k = 0; i_k < A_ncols; i_k += tk ) {

        fragment<matrix_a, tm, tn, tk, ab_elt_t, aorg> tile_a[m_ht_n];
        for ( size_t i_ht = 0; i_ht < m_ht_n; i_ht++ )
          load_matrix_sync( tile_a[i_ht],
              a_ptr + ( C_row_0 + i_ht * tm ) * a_row_stride + i_k * a_col_stride,  a_stride );

        for ( size_t i_wd = 0; i_wd < m_wd_n; i_wd++ ) {
            fragment<matrix_b, tm, tn, tk, ab_elt_t, borg> tile_b;
            load_matrix_sync ( tile_b,
                b_ptr + i_k * b_row_stride + ( C_col_0 + i_wd * tn ) * b_col_stride,  b_stride );

            for ( int i_ht = 0; i_ht < m_ht_n; i_ht++ )
              mma_sync( tile_C_acc[i_ht][i_wd], tile_a[i_ht], tile_b, tile_C_acc[i_ht][i_wd] );
          }}

    // Write completed tiles of c to memory.
    for ( int i_wd = 0; i_wd < m_wd_n; i_wd++ ) for ( ssize_t i_ht = 0; i_ht < m_ht_n; i_ht++ )
        store_matrix_sync
          ( &ld.CT_dev[ C_row_0 + i_ht * tm + ( C_col_0 + i_wd * tn ) * A_nrows ],
            tile_C_acc[i_ht][i_wd], A_nrows, mem_col_major );
}
```

For this problem executions of the code above on an RTX 4090 are to be analyzed. The characteristics of the GPU as reported by the code are:

```
GPU 0: NVIDIA GeForce RTX 4090 @ 2.52 GHz WITH 24078 MiB GLOBAL MEM
GPU 0: L2: 73728 kiB   MEM<->L2: 1008.1 GB/s
GPU 0: CC: 8.9  SM: 128  SP-FP32/SM: 128  DP-FP64/SM:  2  TH/BL: 1024
GPU 0: SHARED: 102400 B/SM   CONST:  65536 B      NUM REGS:  65536
GPU 0: SHARED:  49152 B/BL  SH RES:   1024 B/BL  SH OPT-IN: 101376 B/BL
GPU 0: PEAK: 41288 SP GFLOPS  645 DP GFLOPS  COMP/COMM:  163.8 SP  5.1 DP
```

For the analysis of the execution of a run of the `mm_hw05` kernel, let B denote the block size, and G denote the number of blocks launched (the grid size), and assume that G is set to the number of SMs in the device.

Let w_t denote the number of columns (width) of C and h_t denote the number of rows (height) of C that can be computed by the tensor core by one call of `mma_sync` for some input size. Let w_w denote the number of columns (width) of C assigned to a warp, and h_w denote the number of rows (height) of C assigned to a warp. Similarly w_b and h_b denote the number of columns and rows assigned to a block.

For the RTX 4090 and A and B matrix elements of type FP16 CUDA allows $w_t = 16$ and $h_t = 16$. (Two other shapes are possible but won't be considered for this problem.)

In an execution of `mm_hw05` the number of tensor core tiles computed per warp is specified with a width and height, shown as `wd=`X `ht=`Y in the output. This indicates that a warp should compute a portion of C that is X tensor core tiles wide and Y tensor core tiles high, or $h_w = Yh_t$ rows by $w_w = Xw_t$ columns of C. The values X and Y are parameters to the kernel (in template parameters `m_wd_n` and `m_ht_n`). The block size is also chosen for a kernel, but it is up to the kernel to choose w_b and h_b. Let B denote the number of threads in a (CUDA) block and $W = B/32$ the number of warps. Kernel `mm_hw05` will choose w_{bw} and h_{bw} such that $W = w_{bw}h_{bw}$. With that choice $w_b = w_{bw}w_w$ and $h_b = h_{bw}h_w$. The kernel further restricts both w_{bw} and h_{bw} to be powers of 2. (This was a problem requirement imposed to simplify assignment of rows and columns to blocks.) Finally, the kernel chooses w_{bw} and h_{bw} so that w_b/h_b is the smallest value ≥ 1. That is, $w_b = h_b$ is the preferred value.

In one `i` iteration a block computes a $h_b \times w_b$ portion of C, with each warp computing a $h_w \times w_w$ portion consisting of $\frac{w_w h_w}{w_t h_t}$ tensor core tiles of C (in array `tile_C_acc`). Let m, n, and k denote the dimensions of the matrices where A is $m \times k$ (rows $\times$ columns), B is $k \times n$, and C is $m \times n$. To compute a $h_w \times w_w$ portion of C a warp executes k/k_t iterations of the `i_k` loop, where k_t is the number of columns of the A matrix (same as rows of B matrix) that can be computed by one call to `mma_sync`.

A $h_t \times k_t$ portion of the A matrix or a $k_t \times w_t$ portion of the B matrix is loaded from device memory to registers (we hope) by each call to `load_matrix_sync`. The code is written so that within an `i_k` iteration a warp never loads the same portion twice (unlike some homework solutions).

(*a*) In terms of the variables, indicate the total amount of memory read from the L2 cache in one `i_k` iteration. Assume the L1 cache is of unlimited size but is empty at the beginning of the iteration. Though it doesn't mater for this problem, assume L2 is unlimited and that the A and B matrices are in the L2 cache before execution starts and will remain there.

☐ Amount of memory read in one `i_k` iteration ☐ per warp, ☐ per block, and ☐ total (per grid). ☐ Don't forget to indicate the unit (elements, bytes).

(*b*) Recall that the numbers under the `N-Rd` column in the Homework 5 code output (and similarly labeled columns in the output of other course code examples) show the amount of memory moving from the L2 cache to SMs normalized to a (not necessarily achievable) minimum, in this case the combined size of the A and B matrices. A value of 1 is ideal (and impossible with more than one SM), a value of 2 means twice the minimum amount of data was moved, etc.

For each of the executions below compute what values under `N-Rd` are expected. If the problem is solved correctly the estimated values of `N-Rd` for four-warp configurations should be close to the measured values. Some of the others will not be close.

Continued on the next page.

```
Kernel mm_hw05, 218 regs. Shape 5120 x 2064 x 4096. wd=4  ht=4  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/μs === Util: TC++  Insn-- ===========
 4  2.29 29  8  0.9   36  1.0 1622  0t    886 +++--
 8  2.29 45  8  1.0   31  1.0 2142  0t    581 ++++----
```

☐ Compute estimated `N-Rd` values for the two executions above. ☐ Show the expression used to calculate the estimate -at least once.

```
Kernel mm_hw05, 128 regs. Shape 5120 x 2064 x 4096. wd=4  ht=2  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/μs === Util: TC++  Insn-- ===========
 4  3.18 24  8  0.9   53  1.0 1947  0t   1085 ++---
 8  3.18 32  8  1.0   45  1.0 2216  0t    802 +++----
16  3.19 39  8  1.0   37  1.0 2105  1t    706 +++-----
```

☐ Compute estimated `N-Rd` values for the two executions above. ☐ Show the expression used to calculate the estimate at least once.

(*c*) After insertion of `__syncthreads()` as shown below the values under `N-Rd` are much closer to the estimated values. However the execution times are higher. *Note: In the original exam the execution times were described as lower, not higher.*

```
for ( ssize_t i_k = 0; i_k < A_ncols; i_k += tk )
  {
    __syncthreads();
```

☐ Explain why the estimates for `N-Rd` more closely match measured values in the version with `syncthreads()`.

(*d*) One way to avoid the overhead of `syncthreads()` is not to use it as often. Consider

```
for ( ssize_t i_k = 0; i_k < A_ncols; i_k += tk )
  {
    if ( i_k % interval == 0 ) __syncthreads();
```

☐ Estimate a value for variable `interval` based on the tile shapes and SM characteristics including L1 cache size.

(*e*) The two runs below are similar in that they read the same amount of data and have the same execution efficiency and yet the 2×8 run is much faster. The only difference is the number of registers used by the respective kernels.

```
Kernel mm_hw05, 234 regs. Shape 5120 x 2064 x 4096. wd=2  ht=8  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  2.54 24  8  0.9   36  1.0 1289  0t   1114 ++--
 8  2.54 40  8  1.0   27  1.0 1659  0t    663 +++-----

Kernel mm_hw05, 168 regs. Shape 5120 x 2064 x 4096. wd=8  ht=2  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  2.54 10  8  0.9   36  1.0  573  1t   2508 +-
 8  2.54 19  8  0.9   27  1.0  781  1t   1407 ++--
```

☐ How could the difference in the number of registers explain why the `wd=8 ht=2` runs more slowly than the `wd=2 ht=8` run? ☐ Assume that the `wd=8 ht=2` kernel would have done better with more registers but the compiler did not think it had enough registers available. ☐ *Do not* explain *why* there are fewer registers.

Problem 3: [30 pts] Park 24 ASPLOS [1] shows how a processor in memory (PIM) system can be used to overcome the bandwidth-limited nature of the transformer attention mechanism [2] when used for generation.

One interesting question when looking at possible PIM designs is to find break-even points. To compute `score[i] = dot( query, key[i] )` one needs to send `query` to each GEMV unit (assume column-wise partitioning). Let m denote the number of GEMV units. A break-even point would be the value of m, call it m_e for which the time taken to compute scores on AttAcc would be the same as computing them on a GPU. Of course, one would only consider AttAcc if $m > m_e$.

(*a*) Compute m_e for AttAcc and the DGX large as described by Park. For this analysis don't worry about where the GEMV are placed, our concern is how many there are. For this analysis assume the data bandwidth of the DGX system is $\Theta_M = 26.89\,\text{TB/s}$. (See Section 7.1.) For the score calculation use symbol L for the context length and d for the embedding dimension. The score computation computes an L-element score array, element i is computed by taking the dot product of the d-component query by the d-component element i of the key array. (There is one query and L keys.) For this problem assume that there is just one head.

☐ Indicate the amount of data that would need to move between the DGX and memory when the DGX computes the score.

☐ Indicate the number of FP operations needed to compute the score by the AttAcc system.

☐ Compute a value of m_e based on the computation rate of the GEMV units. (See the Homework 6 solution.) State any assumptions made.

☐ Discuss how the value compares to the values Park uses.

(*b*) Note that the computation `score[i] = dot( query, key[i] )` is done efficiently by AttAcc. Consider a seemingly similar computation `pscores[j,i] = dot( query[j], key[i] )` in which `query` is an array of d-element vectors (as `key` is an array of d-element vectors). As some may have noticed, this is a matrix $\times$ matrix multiplication. If AttAcc is so much better than a GPU at computing `score` why isn't it also better at matrix multiplication?

☐ Explain why AttAcc would not work for matrix $\times$ matrix multiplication. For your answer think about where `query` and `key` might be stored and how that differs from the situation for matrix $\times$ vector multiplication that AttAcc computes for transformer attention scores, ☐ and whether that difference is important for matrix multiplication with the second matrix sized like `keys`.

References:

The papers linked below should be available for free when accessed from within `lsu.edu`*. With enough patience it is possible for students to access them for free off-campus by going through the library. However, those taking the exam can also get a free copy by E-mailing me.*

[1] Park, J., Choi, J., Kyung, K., Kim, M. J., Kwon, Y., Kim, N. S., and Ahn, J. H. AttAcc! unleashing the power of PIM for batched transformer-based generative model inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2024), ASPLOS '24, Association for Computing Machinery, p. 103119. `https://doi.org/10.1145/3620665.3640422`.

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc. `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.