Name _____

GPU Microarchitecture
EE 7722
Solve-Home Final Examination

Friday, 8 May 2020 to Monday, 11 May 2020 5:00 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 _____ (70 pts)

Problem 2 _____ (30 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck! Help Keep Everyone Safe!

Problem 1: [70 pts] Appearing below are the results of kernels from the Homework 3 solution. (The plots were generated using commit `a05027eea3deaaa55fff3db06bd9ef229cc62caa` and run on an RTX 2080 [TU104]). The output of the baseline kernel, `conv_wbuf`, is shown first for reference. The next two kernels, `conv_prob2_inefficient` and `conv_prob28`, are two possible solutions to Problem 2, the one in which a single loaded input element is to be used to compute `n_per_thd` outputs. For both `n_per_thd` is eight. Kernel `conv_prob2_inefficient` does poorly, worse than `conv_wbuf` at larger block sizes. But `conv_prob28` performs well. Excerpts from code for the two kernels appears on the following pages and is available in the repo. The compute portion of the utilization bars are based on FMA instructions only (they don't account for the load and store instructions).

```
GPU 0: GeForce RTX 2080 SUPER @ 1.81 GHz WITH 7982 MiB GLOBAL MEM
GPU 0: L2: 4096 kiB   MEM<->L2: 496.1 GB/s
GPU 0: CC: 7.5  MP: 48  CC/MP:  64  DP/MP:  2  TH/BL: 1024
GPU 0: SHARED: 49152 B/BL  65536 B/MP  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 5576 SP GFLOPS  174 DP GFLOPS  COMP/COMM:  45.0 SP  2.8 DP

Out array size: 85 * 98304 elements, num weights max 16 * 16
Launching with 48 blocks of up to 32 warps.

Kernel conv_wbuf:
 r  c wp ac   t/μs FP θ GB/s --- Utilization: ++Compute++   **Data**  ----------
16 16  1  1  11981  179    6 *+
16 16  2  2   6217  344   12 *++
16 16  3  3   4515  474   16 *+++
16 16  4  4   3975  538   18 *++++
16 16  8  8   2879  743   25 *******
16 16 12 12   2728  784   27 *******
16 16 16 16   2802  763   26 *******
16 16 20 20   2593  825   28 ********
16 16 24 24   2465  868   30 ********
16 16 28 28   2469  866   30 ********
16 16 32 32   2541  842   29 ********

Kernel conv_prob2_inefficient:  (n_per_thd = 8)
 r  c wp ac   t/μs FP θ GB/s --- Utilization: ++Compute++   **Data**  ----------
16 16  1  1   4088  523   18 *****
16 16  2  2   2973  719   25 *******
16 16  3  3   2819  759   26 *******
16 16  4  4   2798  765   26 *******
16 16  8  8   2739  781   27 *******
16 16 12 12   2736  782   27 *******
16 16 16 16   2722  786   27 *******
16 16 20 20   2722  786   27 *******
16 16 24 24   2741  780   27 *******
16 16 28 28   2756  776   27 *******
16 16 32 32   2765  774   26 *******

Kernel conv_prob28:   (This is conv_prob2t instantiated with n_per_thd = 8)
 r  c wp ac   t/μs FP θ GB/s --- Utilization: ++Compute++   **Data**  ----------
16 16  1  1   2449  873   30 ********
16 16  2  2   1272 1681   57 ***************
16 16  3  3    917 2334   80 ********************
16 16  4  4    713 3001  103 ***************************
16 16  8  8    620 3453  118 ********************************
16 16 12 12    635 3369  115 ********************************
16 16 16 16    565 3785  129 ************************************
16 16 20 20    664 3221  110 ******************************
16 16 24 24    610 3505  120 *********************************
16 16 28 28    713 3002  103 ***************************
16 16 32 32    626 3416  117 ********************************
```

```
template <int w_nr, int w_nc> __global__ void conv_wbuf() {
  const int block_dim = blockDim.x;
  const int n_threads = block_dim * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * block_dim;
  const int n_elt = dapp.out_nr * dapp.out_nc;

  for ( int h=tid; h<n_elt; h += n_threads ) {
      const int ro = h / dapp.out_nc,  co = h % dapp.out_nc;
      float s = 0;

      for ( int rw=0; rw<w_nr; rw++ ) for ( int cw=0; cw<w_nc; cw++ ) {
            const int ri = ro + rw;
            const int ci = co + cw;
            const int iidx = ri * dapp.in_nc + ci;
            const int widx = rw * w_nc + cw;
            const float din = dapp.d_in[iidx];
            s += din * dapp.w[widx];
          }
      dapp.d_out[h] = s;
    }
}
```

```
template <int w_nr, int w_nc> __global__ void conv_prob2_inefficient() {
  constexpr int n_per_thd = 8;
  const int block_dim = blockDim.x;
  const int n_threads = block_dim * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * block_dim;
  const int h_0 = tid * n_per_thd;
  const int n_elt = dapp.out_nr * dapp.out_nc;

  for ( int h=h_0; h<n_elt; h += n_threads*n_per_thd ) {
      const int ro = h / dapp.out_nc,  co = h % dapp.out_nc;
      float s[n_per_thd] = { 0 };

      for ( int rw=0; rw<w_nr; rw++ ) for ( int cw=0; cw<w_nc+n_per_thd-1; cw++ ) {
            const int ri = ro + rw,  ci = co + cw;
            const int iidx = ri * dapp.in_nc + ci;
            const float din = dapp.d_in[iidx];

            // Compute output for each of n_per_thd rows.
            for ( int k=0; k<n_per_thd; k++ ) {
                // Find column number of weight array to use for output k.
                const int cwk = cw - k;
                const int widxk = rw * w_nc + cwk;
                if (  cwk >= 0  &&  cwk < w_nc  ) s[k] += din * dapp.w[ widxk ];
              }}
      // Write output for n_per_thd rows.
      for ( int k=0; k<n_per_thd; k++ ) dapp.d_out[h+k] = s[k];
    }
}
```

```
template <int w_nr, int w_nc, int n_per_thd> __device__ void conv_prob2t()
{
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  const int n_threads = blockDim.x * gridDim.x;
  //  Each thread operates on n_per_thd outputs at a time.
  //  The n_per_thd outputs are in consecutive rows and the same column.
  //  Each thread operates on an entire column.

  for ( int co = tid;  co < dapp.out_nc;  co += n_threads )
    for ( int ro = 0;  ro < dapp.out_nr;  ro += n_per_thd )
      {
        const int oidx = ro * dapp.out_nc + co;

        float s[n_per_thd] = { 0 };

        for ( int rw=0; rw<w_nr+n_per_thd-1; rw++ ) for ( int cw=0; cw<w_nc; cw++ )
            {
              const int ri = ro + rw,  ci = co + cw;
              const int iidx = ri * dapp.in_nc + ci;
              const float din = dapp.d_in[iidx];

              // Compute output for each of n_per_thd rows.
              for ( int k=0; k<n_per_thd; k++ ) {
                  // Find row number of weight array to use for output k.
                  const int rwk = rw - k;
                  const int widxk = rwk * w_nc + cw;
                  if (  rwk >= 0  &&  rwk < w_nr ) s[k] += din * dapp.w[ widxk ];
                }}

        // Write output for n_per_thd rows.
        for ( int k=0; k<n_per_thd; k++ ) dapp.d_out[ oidx + k * dapp.out_nc ] = s[k];
      }
}
```

(*a*) Based on the measured performance of the kernels, is there evidence that cache size (either L1 or L2) is a factor in the performance of any of the kernels? Explain. Note that the level 1 (per SM) cache size is 64 kiB and the level 2 cache size is given in the output above. (This part is based on measured performance [the bar charts appearing above], the next part [on the next page] is analytic.)

☐ Based on measured performance of `conv_wbuff` cache size is/isn't a factor because:

☐ Based on measured performance of `conv_prob2_inefficient` cache size is/isn't a factor because:

☐ Based on measured performance of `conv_prob28` cache size is/isn't a factor because:

(*b*) One possible reason that the inefficient kernel runs slowly is that it is making inefficient use of the level 1 cache. For each kernel compute the amount of cache storage needed by one block to compute one `h` iteration in `conv_prob2_inefficient` and one `ro` iteration in `conv_prob2t`. Do so in terms of B, the block size, w_r and w_c, the number rows and columns in the weight array, and n, the value of `n_per_thd`.

If the amount of needed cache storage were actually available and the cache were perfect, then the second time element `din.d_in[x]` were accessed it would be found in the cache and so latency of the second (and subsequent accesses) to `x` would be low.

☐ Cache size needed for an `h` iteration in `conv_prob2_inefficient` in terms of B, w_r, w_c, and n.

☐ Cache size needed for an `ro` iteration `conv_prob2t` in terms of B, w_r, w_c, and n.

(*c*) Which kernel, `conv_prob2_inefficient` or `conv_prob2t`, makes better load and store request utilization? Compute the request utilization of important memory accesses for each. Comment on whether the impact of this might be significant. Compute these in terms of n (value of `n_per_thd`).

☐ Request utilization (a percent or fraction) by `conv_prob2_inefficient` is:

☐ Request utilization (a percent or fraction) by `conv_prob2t` is:

(*d*) Is there evidence that poor request utilization affected the performance of any of the kernels above? Explain.

☐ Poor request utilization seemed to (did not seem to) affect the kernels because:

(*e*) In class we used interval analysis to determine the number of threads needed to saturate a resource. What would be an appropriate code to base an interval on in the `conv_prob2t` kernel. Make the interval no larger than is needed.

☐ Show code forming interval below.

```
template <int w_nr, int w_nc, int n_per_thd> __device__ void conv_prob2t()
{
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  const int n_threads = blockDim.x * gridDim.x;

  for ( int co = tid;  co < dapp.out_nc;  co += n_threads )
    for ( int ro = 0;  ro < dapp.out_nr;  ro += n_per_thd )
      {
        const int oidx = ro * dapp.out_nc + co;

        float s[n_per_thd] = { 0 };

        for ( int rw=0; rw<w_nr+n_per_thd-1; rw++ ) for ( int cw=0; cw<w_nc; cw++ )
            {
              const int ri = ro + rw,  ci = co + cw;
              const int iidx = ri * dapp.in_nc + ci;
              const float din = dapp.d_in[iidx];

              for ( int k=0; k<n_per_thd; k++ ) {
                  // Find row number of weight array to use for output k.
                  const int rwk = rw - k;
                  const int widxk = rwk * w_nc + cw;
                  if (  rwk >= 0  &&  rwk < w_nr ) s[k] += din * dapp.w[ widxk ];
                }}

        for ( int k=0; k<n_per_thd; k++ ) dapp.d_out[ oidx + k * dapp.out_nc ] = s[k];
      }
}
```

(*f*) Consider the computation of the interval latency. Part of that computation is the time needed by a warp scheduler to dispatch threads. That would take two cycles per FMA (fused multiply/add) and eight cycles per load on a Turing device. When n is small the latency might depend on something else, depending on whether the compiler chooses to change the order of some computation. (Changing the order is algebraically correct but will change the result with typical floating-point representations.) What is it that the latency depends on, and how can changing the order effect things?

☐ What does the latency depend on?

☐ How can algebraic re-ordering avoid the latency impact?

Problem 2: [30 pts] As has been pointed out in class, the seven-level loop nest used to describe computation in CNN layers is fairly simple and one might expect that a straightforward textbook procedure could be used to design an optimal CNN accelerator. But no. Many papers describing CNN accelerators claim an optimal design, but only optimal within a constrained set of possibilities. Both Parashar [1] and Yang [2] try to look at broader set of possible designs and draw conclusions about which accelerator hardware design or accelerator software design (dataflow, blocking) is best.

(*a*) Consider representing the CUDA GPU model using Yang or Parashar's representation. Important issues to consider are storage levels, computation elements (by which I mean PEs, MACs, and stuff on a GPU), and communication. To understand CUDA communication issues think about ways in which one thread can obtain a value computed by another. How that's done depends upon the two threads' thread and block indices.

First, what CUDA construct should be represented as a PE (using either Yang or Parashar's definition)?

☐ Which would be best represented as a PE: a thread, a warp, a warp scheduler, a functional unit, or an SM? Explain.

(*b*) Yang uses $A|B$ to describe an assignment of work to a 2-D array of PEs such that the horizontal position (or column number or x-axis value) indicates a value of A (say the left most PE uses $a = 0$, the next one $a = 1$, etc) and the vertical position indicates a value of B. This is intended for CNN (or TPU) accelerators in which the PEs are arranged in a 2-D array and in which communication takes advantage of this arrangement. Suppose the thread index of a CUDA thread were used as the horizontal position and the block index were used as the vertical position. So for a $A|B$ dataflow `a=threadIdx.x` and `b=blockIdx.x`.

Is that a good way of representing CUDA threads in Yang and Parashar's models? If not explain what about the performance of the CUDA code would not match the performance as modeled by these schemes (making needed adjustments to model the GPU). Propose a better way of mapping threads, warps, and blocks into these representations.

☐ Is it a good idea to use thread index for the horizontal position and block index for the vertical position?
☐ Explain. ☐ Describe any problems with this representation.

☐ Propose an alternative mapping, one which provides 2-D PE-array-like communication. (Remember that we are describing how the CUDA model of a GPU is represented, we are not changing the GPU itself.)

(*c*) Figure 5 in Parashar and Listing 2 in Yang show how a blocking of the loop relates to storage levels. An important part of their analyses is determining how many time each level is read and written.

CUDA shared memory, as we all know, is private to a block. Can Yang and Parashar's representation handle this? If so, how (other than making a block the equivalent of a PE)?

☐ Can, and if so how can, per-block storage be represented in Yang's and Parashar's representations?

References:

[1] Parashar, A., Raina, P., Shao, Y. S., Chen, Y., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., and Emer, J. Timeloop: A systematic approach to DNN accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2019), pp. 304–315. https://ieeexplore.ieee.org/abstract/document/8695666.

[2] Yang, X., Gao, M., Liu, Q., Setter, J., Pu, J., Nayak, A., Bell, S., Cao, K., Ha, H., Raina, P., Kozyrakis, C., and Horowitz, M. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS 20, Association for Computing Machinery, p. 369383. https://doi.org/10.1145/3373376.3378514.