LSU EE 7722 Homework 5 Due: 1 May 2019 16:00 CDT

In class we covered NVIDIA GPUs which are reasonably well suited to DNN (deep neural network) computations, especially the CC 7.X devices. The NVIDIA GPUs are successful commercial products and are the result of several generations of refinement and evolution. For that reason we can assume that the design is effective and that features work as intended for graphics and typical scientific workloads. For example, we can feel safe concluding that having a large number (2048!) of thread contexts per SM is a good way of hiding memory latency despite the huge area taken by the registers.

The three papers assigned towards the end of the semester (see the end of this assignment for a complete reference) describe designs for DNN accelerators of various maturity. The Google TPU described by Jouppi *et al* [3] is used in production so we can assume it is effective. It describes a straightforward way of implementing DNNs and one robust enough to handle production work and certainly devoid of complex ideas that would add to development time and have a risk of limited effectiveness.

Chen, Elmer, and Sze [1], introduce a CNN dataflow, row stationary, and compare it to some previously described dataflows. In their comparison methodology they configure implementations of each dataflow by optimizing modeled energy consumption. The idea is to obtain for comparison high-quality configurations of each dataflow under similar hardware constraints. The row-stationary dataflow is not revolutionary, but the equal-footing comparison methodology makes it interesting. Also, the paper is from a project to fabricate a working DNN accelerator chip, Eyeriss, and so one can expect that the design is workable.

The third paper, by Hegde *et al* [2], is the most interesting but also describes results which are less likely to be practical in the end. Their idea is to exploit repeated weights in a filter by computing $w_1(a_1 + a_2)$ instead of $w_1a_1 + w_2a_2$ when $w_1 = w_2$. The ostensible benefit is fewer multiplies and handling fewer weights. The big question is whether the effort needed to deliver the inputs in the right order to the now unique weights is greater than the benefits.

The loop below computes one layer of a CNN.

```
for ( int n=0; n<N; n++ )
  for ( int m=0; m<M; m++ )
    for ( int c=0; c<C; c++ )
      for ( int x=0; x<X; x++ )
        for ( int y=0; y<Y; y++ )
          for ( int i=0; i<I; i++ )
            for ( int j=0; j<J; j++ )
              o[n][m][x][y] += w[m][c][i][j] * i[n][c][x+i-ih][y+j-jh];
```

In class we used the shorthand $F_nF_mF_cF_xF_yF_iF_j$ to indicate this ordering of loops. We'll call the shorthand a *schedule*. It should be clear from the schedule above that the loop body executes $NMCXYIJ$ times. Since the weights, w, are indexed using m, c, i, j there are $MCIJ$ distinct elements (here elements are counted as distinct even if their values are identical) an each element is used $\frac{NMCXYIJ}{MCIJ} = NXY$ times. So far we have not taken into account the places that the element is stored and how it is moved.

To show assignment of calculations to processing elements (which we called threads for convenience) an F in the schedule above would be replaced by a τ, say τ_x. A τ_x indicates that the value of x is based on the position of a PE. The PE assignments (the τ's) would always be in the left-most position. For example, schedule $\tau_x\tau_yF_nF_mF_cF_iF_j$ indicates that the output activation x and y indices match the column and row number of the PE. See the code below:

```
int x = PE_Index.x; // We are assuming such a variable exists.
int y = PE_Index.y; // We are assuming such a variable exists.
for ( int n=0; n<N; n++ )
  for ( int m=0; m<M; m++ )
    for ( int c=0; c<C; c++ )
      for ( int i=0; i<I; i++ )
        for ( int j=0; j<J; j++ )
          o[n][m][x][y] += w[m][c][i][j] * i[n][c][x+i-ih][y+j-jh];
```

Using Chen's notation [1] the schedule $\tau_x \tau_y F_n F_m F_c F_i F_j$ is output stationary and the assignment $\tau_i \tau_j F_n F_m F_c F_x F_y$ is weight stationary. An input-stationary schedule can be expressed by replacing iterator x and y (output activation indices) with x' and y' (use these for input activation indices). Then in the loop body compute $x = x' - i + I/2$.

For convenience we usually assumed a 2D array of PEs of just the size we needed. But it is easy enough to *tile* the assignment to fit any size array. For a $w \times h$ PE array (w columns, h rows) a variable assigned to a PE row or column could be split. For example, set $x_0 = x \bmod w$ and $x_1 = \lfloor x/w \rfloor$. In this case x_1 is the tile number and x_0 is the position within the tile. Here a tile is sized to fit the PE array but iterations can be tiled to fit storage. The following schedule would work when the activation and PE array dimensions don't match: $\tau_{x_0} \tau_{y_0} F_n F_m F_c F_{x_1} F_i F_{y_1} F_j$. This corresponds to loop nest:

```
int x0 = PE_Index.x; // We are assuming such a variable exists.
int y0 = PE_Index.y; // We are assuming such a variable exists.
int X1 = (X+PE.w-1)/PE.w, Y1 = (Y+PE.h-1)/PE.h;

for ( int n=0; n<N; n++ )
  for ( int m=0; m<M; m++ )
    for ( int c=0; c<C; c++ )
      for ( int x1=0; x1<X1; x1++ )
        for ( int i=0; i<I; i++ )
          for ( int y1=0; y1<Y1; y1++ )
            for ( int j=0; j<J; j++ )
              {
                int x = x0 + PE.w * x1,  y = y0 + PE.h * y1;
                uint xi = x + i - Il, yj = y + j - Jl;
                o[n][m][x][y] += w[m][c][i][j] * i[n][c][xi][yj];
              }
```

It is important to specify and analyze the buffering and movement of weights, inputs, and partial sums. This will be shown in by adding buffering indicators to the schedule. A buffering indicator consists of an array symbol (w for weights and i for input activations) either in parenthesis if it is being written into a register file or braces if it is written into the global buffer. The position in the schedule indicates when the elements are buffered and which elements are buffered. For example, consider $\tau_{x_0} \tau_{y_0} F_n F_m F_c (w) F_{x_1} F_i F_{y_1} F_j$. In this case w is buffered when a particular value of m and c are available. So only IJ elements need to be loaded:

```
int x0 = PE_Index.x; // We are assuming such a variable exists.
int y0 = PE_Index.y; // We are assuming such a variable exists.
int X1 = (X+PE.w-1)/PE.w, Y1 = (Y+PE.h-1)/PE.h;

for ( int n=0; n<N; n++ )
  for ( int m=0; m<M; m++ )
    for ( int c=0; c<C; c++ )
      {
        for ( int i=0; i<I; i++ ) for ( int j=0; j<J; j++ ) RF.w[i][j] = w[m][c][i][j];
        for ( int x1=0; x1<X1; x1++ )
          for ( int i=0; i<I; i++ )
            for ( int y1=0; y1<Y1; y1++ )
              for ( int j=0; j<J; j++ )
                {
                  int x = x0 + PE.w * x1;
                  int y = y0 + PE.h * y1;
                  const int xi = x + i - Il, yj = y + j - Jl;
                  o[n][m][x][y] += RF.w[i][j] * i[n][c][xi][yj];
                }
    }
```

From the schedule shorthand, $\tau_{x_0}\tau_{y_0}F_nF_mF_c(w)F_{x_1}F_iF_{y_1}F_j$, it is easy to determine things like how much storage is needed for w, and how much throughput is needed. The amount of storage is determined by dividing the total storage, $MCIJ$ by the indices that appear in enclosing loops (or to the left in the shorthand). In the example, that's m and c, so divide by MC. Elements `w[m][c][I][J]` are loaded (a lower case index means one index, say `m=1`, upper case means all valid indices, say `I=0,1,2,..`). None of the indices are computed from the PE index, so all PEs load the same value so not much bandwidth is needed from the global buffer (or DRAM). (At this point we haven't specified where the data is coming from.) The buffering of w is enclosed by the n, m, and c loops so it is done NMC times. Including redundancy the number of times each PE loads a value is $NMCIJ$. If we are accounting for energy use and those values are in the global buffer, then there are $NMCIJ$ global buffer reads (each value read is used by all wh PEs) and $whNMCIJ$ register file writes. The number of register file reads is $whNMCX_1IY_1J = XYNMCIJ$.

Now consider $\tau_{x_0}\tau_{y_0}F_nF_mF_cF_{x_1}F_i(w)F_{y_1}F_j$. In this case each RF stores just J weights. However each RF file is loaded $NMCX_1I$ times. This works out to a factor of X_1I times more than for the $\tau_{x_0}\tau_{y_0}F_nF_mF_c(w)F_{x_1}F_iF_{y_1}F_j$ schedule but the amount of data loaded is only X_1 times higher.

If we wanted to minimize both storage and redundancy for weights, while ignoring everything else, we could use schedule $\tau_{x_0}\tau_{y_0}F_mF_cF_iF_j(w)F_nF_{x_1}F_{y_1}$. Each weight is loaded just once and the amount of RF storage needed is 1. However the amount of storage needed for partial sums and inputs is higher.

Problem 2: *Note: There is no Problem 1. Problem 2 is the first problem.* Use the shorthand described above to show scheduling and buffering for each of the following dataflows (using Chen's terminology [1]). Show register file buffering only. If the scheduling (loop order) can't be determined from the paper, make a good choice. Tile iterations to fit storage, as suggested in the paper.

WS

SOC-MOP OS

MOC-MOP OS

NLR

RS

Problem 3: Answer the following question about the row-stationary dataflow adapted to NVIDIA GPUs. Should each PE correspond to: a single thread, a single warp, or a single block. Describe the best choice and explain why it is best. Consider the amount of storage needed and the ease of communication.

References:

[1] Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 367–379. `https://doi.org/10.1109/ISCA.2016.40.`

[2] Hegde, K., Yu, J., Agrawal, R., Yan, M., Pellauer, M., and Fletcher, C. W. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2018), ISCA '18, IEEE Press, pp. 674–687. `https://doi.org/10.1109/ISCA.2018.00062.`

[3] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani,

E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 1–12. `http://doi.acm.org/10.1145/3079856.3080246`.