LSU EE 7722 **Homework 2** **Due: 19 April 2021**

Basic Setup

Follow the instructions for class account setup on `https://www.ece.lsu.edu/gp/proc.html`.
Code for this assignment is in directory `../hw/gpm/2021/hw02`.

If the class account has been set up properly, the code can be built from within Emacs by pressing F9 when visiting any file in the `.../2021/hw02` directory or when in an Emacs shell buffer (which can be entered using Alt-x shell Enter). The code can be built from the command line using the command `make -j 4` (assuming `.../2021/hw02` is the current directory). Either method runs a makefile that builds all examples in the directory. It builds two versions of each program, one taking the base name of the main file, such as `hw02`, and one with the suffix `-cuda-debug`, such as `hw02-cuda-debug`. The versions with the `-cuda-debug` suffix are compiled with host optimization turned off and CUDA debugging turned on, which facilitates debugging but slows down execution. To debug CUDA or host (CPU) code use `cuda-gdb`. Note that the `-cuda-debug` versions will run much more slowly than the regular versions.

Running make on a clean directory will produce a large amount of output. The make program and the file it reads, `Makefile`, are designed to build executables in a lazy fashion, meaning that they only create a file if it is not present or if its prerequisites have changed. Therefore a second run of make will take much less time.

Quickly check whether the build is successful with the command `./hw02`. It should produce output ending with a line something like `800 52 48 32  3.6 15.0 401.8  12534  398 +++----------`.

The makefile will compile code for a GPU on the system it was run, favoring the GPU that's not connected to a display. Re-run make when moving to a different system. The Makefile should automatically detect whether the GPU for which the executable was built matches the GPU on the current system, and re-build if needed.

Using `hw02`

The code in `hw02.cu` contains several kernels that compute the output of a fully-connected neural net layer. See the problems for a description of the kernels.

The `hw02` program takes three command-line arguments. The first indicates how many blocks to launch. If the argument is zero then the number of blocks will be set to the number of multiprocessors (which is the default). If the argument is negative then the number of blocks will be $-aP$, where a is the argument value and P is the number of MPs on the GPU.

The second argument is the number of threads per block to try to use to launch each kernel. If the argument is omitted 1024 threads are tried. If the argument is omitted or positive, the actual number of threads used in a launch is the minimum of this argument and the kernel's maximum. (For example, when the second argument is 512, but kernel `foo` has a limit of 256 threads, `foo` will be launched with 256 threads.) When the second argument is `0` (zero) then each kernel will be launched multiple times starting with 4 warps, incrementing by 4 warps until the kernel maximum is reached.

Here are some examples: Run with 256 threads per block: `./hw02 0 256`. Run with 512 threads per block and twice as many blocks as MPs: `./hw02 -2 512`. Run with 256 threads per block and 10 blocks: `./hw02 10 256`.

DNN Computation

The code in `hw02.cu` computes the output of a fully connected neural network layer consisting of n_i input neurons per channel, n_c input channels, n_o output neurons per output channel, n_m output channels, and n_n batches. The simplified loop nest below performs this computation:

```
for ( int in = 0; in < nn; in++ )
  for ( int im = 0; im < nm; im++ )
    for ( int io = 0; io < no; io++ )
      for ( int ic = 0; ic < nc; ic++ )
```

```
            for ( int ii = 0; ii < ni; ii++ )
              ao[io][im][in] += ai[ii][ic][in] * w[ic][im][ii][io];
```

In the homework file the code above is near the end of routine `layer_init`. The output neurons are in `ao`, the input neurons are in `ai`, and *weights* are in array `w`. Notice that the loops can be done in any order. The code in the file shows the re-ordering of the loop used in the solution to Homework 1.

The code sample above assumes that the arrays are multi-dimensional. The arrays used in `hw02.cu` are one-dimensional, so the multiple indices shown above must be converted to a single index. This is how the code is actually written:

```
# pragma omp parallel for
  for ( int in = 0; in < nn; in++ )
    for ( int im = 0; im < nm; im++ )
      for ( int io = 0; io < no; io++ )
        {
          acc_t ac = 0;
          for ( int ic = 0; ic < nc; ic++ )
            for ( int ii = 0; ii < ni; ii++ )
              {
                size_t idx_ai = ii + ni * ( ic + nc * in );
                size_t idx_w  = ic + nc * ( im + nm * ( ii + ni * io ) );
                ac += ai[ idx_ai ] * w[ idx_w ];
              }
          ao[ io + no * ( im + nm * in ) ] = ac;
        }
```

In addition to computing indices, the code above uses a local variable, `ac`, to hold intermediate values of an `ao` value being computed. This avoids unnecessary loads and stores.

The data type for the inputs and outputs are `acc_t`, and the data type for weights are `wht_t`. Both of these are defined near the top of the file as `float`. In most DNN systems the weights would be defined as a 16-bit or smaller type. There is commented out code to use one of two 16-bit types for weights. As the code is written these will reduce data traffic, but will result in additional instructions since the 16-bit types will be converted to 32-bit types before use, which is not ideal.

Program Output

Starting a run of `hw02` . . .

```
[koppel@dmk-laptop hw02]$ ./hw02
```

. . . produces the following output:

The first thing printed is information about each GPU connected to the system, followed by a line showing the chosen GPU:

```
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
Using GPU 0
```

Most lab computers have two GPUs, please pay attention to the GPU that is actually being used, GPU 0 above.

Most fields are self-explanatory. For example, `L2` is the size of the level-2 cache and CC indicates that the device is of compute capability 7.5 (Turing). The `MEM<->L2` field shows the off-chip bandwidth. `MP` indicates the number of multiprocessors, also called streaming multiprocessors (SM's). `CC/MP` indicates the number of CUDA cores (single-precision functional units) per MP, `DP/MP` indicates the number of double-precision functional units per MP, and `TH/BL` is the maximum number of threads per block.

The amount of shared memory available is shown per block (B/BL) and per MP, this does not indicate whether any particular kernel is using that much shared memory or could use that much. The same line shows the amount of constant memory, and the number of registers available.

The next line, PEAK, shows FP operation bandwidth in which a fused multiply-add is counted as one operation. (Most of the rest of the world counts a multiply-add as two operations, but in this class it's one.) The COMP/COMM line gives the computation to communication ratio in floating-point operations per floating-point element transfers. (The information above was collected in part using the runtime library's cudaGetDeviceProperties function.)

Next, the program provides information on the network layers to be tested:

```
Layer shape 0: ni=no=28.  nc=nm=20. nn=800.
  Number elts: activations 896000, weights 313600
  Weights size: 1225 kiB   L2 cache units: 0.299
  Act size one batch  : 4480 B   L2 cache units: 0.001
  Act size all batches: 3584000 B   L2 cache units: 0.854
Layer shape 1: ni=no=44.  nc=nm=52. nn=800.
  Number elts: activations 3660800, weights 5234944
  Weights size: 20449 kiB   L2 cache units: 4.992
  Act size one batch  : 18304 B   L2 cache units: 0.004
  Act size all batches: 14643200 B   L2 cache units: 3.491
```

The layers used are specified in the constant array ls near the top of hw02.cu.

The program can either launch each kernel once, with a particular configuration (number of blocks and number of threads per block), or it can launch each kernel multiple times, each with a different block size.

When run without arguments or with a 0 as the second argument, such as ./hw02 0 0, the program, launches each kernel multiple times, starting with 4 warps per block, up to 32 warps per block. Run time and other information will be shown for each launch.

Performance Data

By default each kernel is multiple times, starting with one warp per MP, and in successive runs increasing the number of warps per MP. A line of performance data is printed for each run. The output for an RTX 2080 Super appears below. The last two lines of the output were added as an illustration, they are not true output.

```
Kernel (dnn_base<ls[0].nn,ls[0].nc,ls[0].ni>):
 nn nc ni wp I/op DUse  2Use   t/us FP θ === Util: FP++  Insn-- Data**  ========
800 20 32  1  3.3  2.8  74.9   2664  123 +-------
800 20 32  2  3.3  2.8  63.2   1550  211 +------------
800 20 32  3  3.3  2.6  55.9   1261  260 ++--------------
800 20 32  4  3.3  2.7  47.8   1089  301 ++-----------------
800 20 32  8  3.3  2.5  32.1    903  363 +++----------------------
800 20 32 12  3.3  2.8  30.0    878  373 *++----------------------
800 20 32 12  3.3  2.8  30.0    878  373 ++---**********************************
800 20 32 12  3.3  2.8  30.0    878  373 ***************************************
```

The nn, nc, and ni columns show the shape of the network. The values of no and nm aren't shown, but no is set to the same value as ni and nm is set to the same value as nc. Column wp shows the number of warps per block in the run. If the number of blocks in a launch is not set to the number of MPs then there would be a column headed ac, which would show the number of resident warps per MP. (The number of resident warps per MP is a multiple of the number of warps per block. By default the number of blocks in a launch is set equal to the number of MPs, and in such a case the value in the ac column would match the wp column.)

The t/μs column shows the measured execution time in microseconds. The FP θ column shows off-chip data throughput based on measured execution time and an assumed amount of data. The assumed amount of data is actually an absolute minimum: the total size of the input, output, and weight arrays. The value

in the `FP` θ column is correct only if each weight and input, and output crosses the chip boundary exactly once. (See the description of `DUse` below.)

To the right of `FP` θ is a bar graph showing how busy three resources are (based on certain assumptions). Three resources are tracked, FMA (fused multiply/add) instructions, shown with a `+`, FMA along with load instructions, shown with a `-`, and off-chip data transfer, shown with a `*`. The right-most position of a resource's character indicates what fraction of the time that resource is busy. A resource is being used 100% of the time if its character reaches the rightmost position (the last `=` in the column heading over the bar graph). That is true in the last line for the FMA resource, and in the penultimate line for the off-chip data transfer. In the last line we would say that the FP capability is being saturated (a good thing) and in the penultimate line we would say that data transfer is being saturated (also a good thing given the assumptions made). Those last two lines are fictional. Consider the line for the 8 warp per MP run. The `-` is about halfway to the end. That indicates that instruction throughput is about half of the peak possible.

The FMA utilization is computed by assuming one multiply/add per loop iteration, or $n_n n_o n_m n_i n_c$ FMAs. Then the amount of time it would take to issue that many FMAs is computed. That time is divided by the measured execution time to get the utilization. The amount of time to issue the FMAs is based on the GPU being used and should be accurate (up to CC 7.5). For the code in this assignment FMA utilization should be brought closer to 100%.

The instruction utilization, `-`, includes the FMA plus two load instructions per FMA. Including two load instructions per FMA is correct for the `dnn_base` kernel, but is something that can be reduced. Instruction utilization is much higher than FMA utilization. That's because on recent NVIDIA devices there are four FP32 units for each LS unit, so it takes four time as long to dispatch the threads in a warp for a load or store instruction, than it does for a 32-bit FMA. As a result the time that instruction issue is busy considering loads is $(1 + 4 + 4)/1 = 9$ times as long as the time considering must the FMA.

The `I/op`, `DUse`, and `2Use` columns show measured characteristics of the executing code—if your computer allows it. Otherwise, either the columns will not be present or (until this is fixed) there will be a non-helpful error message.

(Please report systems on which the columns do not appear.) The `I/op` column shows the measured number of instructions, divided by the assumed number of FMA operations. In the example above the value is 3.3, which is close to our estimate of 3 instructions (one FMA plus two loads). The value under `DUse` is the number of bytes crossing the chip boundary divided by the minimum amount of data (the sum of the size of the input, output, and weight arrays). In the example above the value under `DUse` is 2.8, meaning on average each item crosses the chip boundary 2.8 times. The ideal value under `DUse` of course is 1. The value under `2Use` is the amount of data that moves between the L2 cache and the MPs divided by the minimum. It should never be lower than `DUse`. In the example above it is much higher, indicating that the L2 cache is doing its job of reducing off-chip data transfer.

Problem 1: The solution to Homework 1 seemed to achieve good performance based on its self-reported resource utilization bar graph. But as discussed in the introduction above, the utilization of FMA hardware is not so good due to those pesky load instructions. (Store instructions also have a lower throughput but our code doesn't execute them as often.)

As discussed in class the two-loads-per-FMA assumption is not a lower bound. Because individual weights and individual inputs are reused it is possible to read them once and use them multiple times.

Load throughput can also be reduced by coaxing the compiler into issuing vector loads. The compiler will do that if it figures out that it needs to load two or four consecutive values and that the address of the first value is a multiple of $2 \times 4 = 8\,\mathrm{B}$ or $4 \times 4 = 16\,\mathrm{B}$ (meaning the address is a multiple of the size of the data being loaded). The compiler can tell if it needs to load 2 or 4 consecutive items by examining the loop nest. It can't normally tell that the address is aligned, but we can tell the compiler to assume that it is a multiple of some number using `__builtin_assume_aligned`. (Look up the use of that on your own.)

(*a*) Modify kernels `dnn_sol_a` and `dnn_sol_b` to reduce the number of loads by re-using loaded values and using vector loads. To do so the loops will need to be re-done. If needed the weights can be re-organized in the `w2` array. (Initially both `w` and `w2` are the organization found for the Homework 2 solution.)

Any solution that exploits re-use correctly will have to contend with load imbalance issues. This will ultimately limit the performance of solutions, especially on the smaller of the two networks.

(*b*) Indicate whether your solution is performing as expected and what is limiting the performance of your solution. Look at factors such as iteration latency and load imbalance. For load imbalance determine how many `h` iterations your code does. Imbalance is bad if 48000 threads perform 2 iterations and on thread performs 3.