LSU EE 7722 **Homework 1** **Due: 23 February 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/hw01`.

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/hw01` 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/hw01` 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 `hw01`, and one with the suffix `-cuda-debug`, such as `hw01-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 `./hw01`. It should produce output ending with a line something like `800 20 28 32 32  5442  46  1 ++++`.

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 `hw01`

The code in `hw01.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 `hw01` 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) or `p` then each kernel will be launched multiple times starting with 4 warps, incrementing by 4 warps until the kernel maximum is reached. When `p` is used additional performance data is shown, which is interesting but it can slow things down. *Note:* `p` *does not work on this assignment.*

Here are some examples: Run with 256 threads per block: `./hw01 0 256`. Run with 512 threads per block and twice as many blocks as MPs: `./hw01 -2 512`. Run with 256 threads per block and 10 blocks: `./hw01 10 256`.

DNN Computation

The code in `hw01.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. For example, the version below computes the same result:

```
for ( int ic = 0; ic < nc; ic++ )
  for ( int ii = 0; ii < ni; ii++ )
    for ( int in = 0; in < nn; in++ )
      for ( int im = 0; im < nm; im++ )
        for ( int io = 0; io < no; io++ )
          ao[io][im][in] += ai[ii][ic][in] * w[ic][im][ii][io];
```

An important thing to notice is that each element of `w` is read `nn` times, each element of `ai` is read `no*nm` times and that each element of `ao` is read and written `ni*nc` times. In a good implementation the re-used values will be found where they are needed (say, in a cache).

The code sample above assumes that the arrays are multi-dimensional. The arrays used in `hw01.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 `hw01` ...

```
[koppel@dmk-laptop hw01]$ ./hw01
```

... 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 `hw01.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 `./hw01 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. An excerpt for one kernel appears below:

```
Launching with 48 blocks of up to 32 warps.
Kernel (dnn_base<ls[0].nn,ls[0].nc,ls[0].ni>):
 nn nc ni wp ac   t/us FP θ GB/s --- Utilization: ++Compute++   **Data**  ------
800 20 28  1  1   5423   46    1 ++++
800 20 28  2  2   5411   46    1 ++++
800 20 28  3  3   5411   46    1 ++++
800 20 28  4  4   5413   46    1 ++++
800 20 28  8  8   5410   46    1 ++++
800 20 28 12 12   5422   46    1 ++++
800 20 28 16 16   5415   46    1 ++++
800 20 28 20 20   5409   46    1 ++++
800 20 28 24 24   5411   46    1 ++++
800 20 28 28 28   5408   46    1 ++++
800 20 28 32 32   5410   46    1 ++++
```

The first three columns show the dnn dimensions, where `nn` is the number of batches, `nc` is the number of input channels, and `ni` is the number of neurons in an input. For all the networks simulated the number of output neurons per channel is the same as the number of input neurons per channel, `no=ni`, and the number of output channels is the same as the number of input channels, `nm=nc`.

The `wp` column shows the number of warps per block that the kernel was launched with. The `ac` column shows the number of warps assigned to an MP (which is the product of the number of warps per block and the number of active blocks per MP). The number in the `ac` column is computed by an NVIDIA API using information about the kernel and the GPU. In the example above the `wp` and `ac` numbers are the same because the number of blocks is the same as the number of MPs and so there is no way to have more than one block per MP.

The $t/\mu s$ column shows the measured execution time. The number under `FP` Θ is the floating point throughput based on measured time and an ideal number of floating-point operations. The number under `GB/s` is the minimum off-chip bandwidth, computed by dividing the size of the input and output arrays by the measured execution time.

The stars in last column show data and compute bandwidth utilization. If the stars extend to the maximum length (shown by the hyphens to the right of `Utilization` in the column heading) then either compute or off-chip bandwidth is being saturated (fully utilized).

Note that this number is computed using measured time and an ideal amount of data crossing the chip boundary.

The compute utilization is based on an assumed number of floating point instructions. The number of floating point instructions, all multiply-adds, is assumed to be $n_n n_i n_c n_o n_m$. (Search for `num_ops_fp` in `hw01.cu`.) The number of load/store instructions is assumed to be $n_n n_i n_c n_o n_m + n_n n_o n_m n_i n_c + n_n n_o n_m$. (Search for `num_ops_ls` in `hw01.cu`.)

The assumption about the number of floating point instructions should reflect the actual number. The number of load/store instructions is computed assuming that there will be one load instruction for each element used and that there is one store for each `ao` written. These reflect the way the initial code is written. The kernels can be written so that the number of load instructions is be lower.

Problem 1: In `hw01.cu` there are three kernels, to compute the DNN output, `dnn_base`, `dnn_sol_a`, and `dnn_sol_b`. Initially each of these are identical and do not run very efficiently. Improve the efficiency of `dnn_sol_a` and `dnn_sol_b` as described below. The reasons for having three identical kernels is so that you can compare ideas side-by-side. In addition to these kernels, you may need to modify the code at the end of routine `layer_init`.

Initially the kernels are inefficient. Consider the main loop nest:

```
for ( int in = blockIdx.x; in < nn; in += gridDim.x )
  for ( int im = threadIdx.x; im < nm; im += blockDim.x )
    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++ )
              ac +=
                ai[ ii + ni * ( ic + nc * in ) ]
                * w[ ic + nc * ( im + nm * ( ii + ni * io ) ) ];
          ao[ io + no * ( im + nm * in ) ] = ac;
        }
```

This code assigns an input batch to a particular block, and an output channel to a particular thread. That means that a particular thread will compute all output values (values of `io`) for particular output channels (values of `im`) and particular batches (values of `in`). This division of work may be effective if the number of batches (`nn`) is a multiple of the number of SMs and if the number of output channels is multiple of the block size. But that's not the case here.

The code also suffers because access patterns to the arrays do not make efficient use of memory requests.

In this problem fix these problems. This can be done by re-arranging or re-doing the loop nests and by changing the layout of the alternative weight array, `w2`, at the end of `layer_init`.