LSU EE 7722 **Homework 1** **Due: 6 March 2020**

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/2020/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 `.../2020/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 `.../2020/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 and 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 `K conv_inbuf_b 32 wp 100.096 s 335.222 GFLOPS 83.808 GB/s`.

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 convolution of an input array, `dapp.d_in`, with a short vector, `dapp.d_w`. 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.*

The third argument specifies the size of the array, in *mibiions*. (One mibion is 2^{20}.) The default is 1 mibion (1,048,576) elements. If a_3 is the value of the third argument, the input size will be $a_3 2^{20}$ elements. The third argument is read as a floating-point number, so "0.476837158203" will result in a 500,000 elements.

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`. Run each kernel multiple times using an input size of 1 billion (10^9 elements): `./hw01 0 0 953.674`.

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.)

The program will next print information about each kernel:

```
CUDA Kernel Resource Usage:
For conv_simple:
       0 shared, 192 const, 0 loc, 58 regs; 1024 max threads per block.
For conv_efficient:
       0 shared, 192 const, 0 loc, 62 regs; 1024 max threads per block.
For conv_wbuf:
       0 shared, 192 const, 0 loc, 62 regs; 1024 max threads per block.
For conv_inbuf_a:
       0 shared, 192 const, 0 loc, 64 regs; 1024 max threads per block.
For conv_inbuf_b:
       0 shared, 192 const, 0 loc, 64 regs; 1024 max threads per block.
```

The `max threads per block` shown above is based on the kernel and reflects register usage.

Though it does not happen above, it is possible that a kernel can be limited to less than 1024 threads per block because it uses more than 64 registers (on a CC 5.2 device).

Next, the program provides information on the input size and launch configuration.

```
Array size: 16777216 elements, num weights 32
Launching with 48 blocks of up to 1024 threads.
```

The array size can be changed using command-line arguments, that is explained further below. Variable `Dj` near the top of the file controls the number of weights (the size of the `w` array).

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. Without arguments is runs each kernel once and prints one line per kernel.

```
Launching with 48 blocks of up to 1024 threads.
K conv_simple     32 wp   140417.664 µs     3.823 GFLOPS     0.956 GB/s
K conv_efficient  32 wp      860.032 µs   624.245 GFLOPS   156.062 GB/s
K conv_wbuf       32 wp      858.720 µs   625.199 GFLOPS   156.300 GB/s
K conv_inbuf_a    32 wp      587.392 µs   913.991 GFLOPS   228.498 GB/s
K conv_inbuf_b    32 wp      579.552 µs   926.355 GFLOPS   231.589 GB/s
```

The μs values are the execution time, the GFLOPS shows the computation rate measured in billions of FP operations per second, and the `GB/s` value is the off-chip data throughput. Recall that a multiply-add is counted as one FP operation.

When run 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:

```
Kernel conv_simple:
wp ac   t/µs  FP  GB/s Data BW Util-----------------------------------------------
 1  1    463   72   18 **
 2  2    314  107   27 ***
 3  3    395   85   21 **
 4  4    342   98   25 **
 8  8    428   78   20 **
12 12    856   39   10 *
16 16    980   34    9
20 20   1192   28    7
24 24   1079   31    8
28 28    981   34    9
32 32    885   38    9 *
```

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 `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 bandwidth utilization based on the `GB/s` number. If the stars extend to the maximum length (shown by the hyphens to the right of `Bandwidth Util` in the column heading) then 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.

Problem 1: In the unmodified code `conv_wbuf` is the same as `conv_efficient`, and so suffers from the same problem: Repeated access to the weights in the expression `dapp.d_w[j]`. Fix this by loading the weights in to an array in the local address space before the `h` loop and accessing that local array in the loop. Do not use `dapp.w[j]` in this problem.

A correct solution to this problem will result in a 5× speedup on a Pascal (CC 6.x) or earlier device but only a modest 1.5× speedup on Turing (CC 7.5) devices. The workstation lab has both types of machines, see `https://www.ece.lsu.edu/koppel/gpup/sys-status.html` to find a machine with they type of GPU you would like to use.

Hint: This is an easy problem.

Problem 2: In the unmodified code `conv_inbuf_a` is similar to `conv_efficient` except that it computes (but does not use) warp and lane IDs for the thread and it uses constant memory for weights, `dapp.w[j]`. (In this problem don't change the code loading weights.) The execution time of an unmodified `conv_inbuf_a` should be about the same as a correctly solved `conv_wbuf`.

Notice that in `conv_inbuf_a` and in all of the other unmodified kernels most elements of `dapp.d_in` are each accessed `Dj` times. That's not a problem on Turing (CC 7.5) devices because only the first access to a particular element of `dapp.d_in` will consume off-chip bandwidth and suffer large latencies. Subsequent accesses will likely hit the L1 cache and so will have low access latency and not use MP-to-L2 cache nor off-chip pathways. But Pascal (CC 6.X) and older devices lack a low-latency L1 cache and so performance on these devices will be poor. (The texture cache latency is about 3× higher than Turing L1 cache latency, and is not automatically used.)

For this problem modify `conv_inbuf_a` so that each element of `dapp.d_in` is loaded just once and placed in shared memory where it will be accessed as many times as needed (which is `Dj` times). Solve this by assigning a contiguous portion of the array to each warp. Let B denote the number of threads per block, G the number of blocks, $W = 32$ the warp size, and n the number of elements in the array. Then there are BG/W warps, and so $c = \lceil \frac{nW}{BG} \rceil$ array elements per warp. Modify `conv_inbuf_a` so that warp 0 (threads for which `wp_idx==0`) accesses elements $0, 1, \ldots, c-1$, warp 1 accesses $c, c+1, \ldots, 2c-1$, and so on. Write the code for `Dj`≤ 32. Each warp should compute 32 elements of `dapp.d_out` per `h` iteration (one per thread) and will need to buffer `32+Dj-1` elements of the input array in shared memory. (For simplicity buffer `64` elements.) (The amount of shared memory needed per block is based on the number of warps in the block.) Each thread should load one element of the input array per output element computed (except for the first output element, in which case it need to load two input elements).

Kernels `conv_inbuf_a` and `conv_inbuf_b` are identical and both can be used for solving this problem. For example, one might try to use `conv_inbuf_a` to access elements in the correct order (assigning each warp a contiguous slice of the array) and use `conv_inbuf_b` to buffer elements in shared memory (but without changing access order). When both are working combine the solutions. in either kernel, preferably `a`.

See prior semesters' homework for examples. The solutions to all programming assignments are in the repo, with file names like `hw01-sol.cu`.

Problems of this type are best solved using a debugger. In fact, if you don't use a debugger you will be wasting time and suffering needless frustration. See the course procedures page for basic steps on using the CUDA debugger.