LSU EE 7722 **Homework 2** **Due: 27 March 2019**

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/2019/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 `.../2019/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 `.../2019/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 `-debug`, such as `hw02-debug`. The versions with the `-debug` suffix are compiled with host optimization turned off, which facilitates debugging. At the moment GPU debugging can only be enabled by editing the makefile: Put a `-G` on the line of `CUCC_ONLY` flags.

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 show the run time of each of several kernels:

```
Launching with 20 blocks of up to 1024 threads.
mxv_g_only      32 wp     3366 s   179.418 GF    59.806 GB/s   1.13 I/F   12.5%
mxv_vec_ld      32 wp     1326 s   455.452 GF   151.817 GB/s   1.11 I/F   50.0%
mxv_o_per_thd   32 wp     2375 s   254.313 GF    84.771 GB/s   3.09 I/F   12.5%
mxv_o_per_thd_sol 32 wp     2383 s   253.422 GF    84.474 GB/s   3.09 I/F   12.5%
mxv_sh_ochunk   20 wp     1033 s   584.672 GF   194.891 GB/s   1.50 I/F  100.0%
mxv_sh_ochunk_sol 20 wp     1032 s   585.161 GF   195.054 GB/s   1.50 I/F  100.0%
```

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` is based on the `vtx_xform_size` code. The description below applies to this assignment, `hw02`, and `vtx_xform_size`.

The code in `vtx-xform-size.cu` contains several kernels that compute a matrix/vector product using a constant $M \times N$ matrix A applied to S N-component input column vectors, producing S M-component output column vectors. The values of N and M are hard-coded in the file, whereas S is a command-line argument. Vector and matrix elements are of type `Elt_Type`, which is hardcoded to `float`.

The `vtx-xform-size` 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.

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 for different block sizes (measured in warps). The block sizes will be 1, 2, 3, 4, 8, 12, ... warps. The maximum block size is based on the kernel. (A kernel that uses few registers can launch with a block size of 32, if more registers are used the maximum number of warps is lower.) When `p` is used additional performance data is shown, which is interesting but it can slow things down.

The third argument specifies the number of *mibions* input vectors to use. One mibion is 2^{20}. The default is 1 mibion (1,048,576) vectors. If a_3 is the value of the third argument, the input size will be $a_3 2^{20}$ vectors. The third argument is read as a floating-point number, so "0.5" will result in a 2^{19} vectors input.

Here are some examples: Run with 256 threads per block: `./vtx-xform-size 0 256`. Run with 512 threads per block and twice as many blocks as MPs: `./vtx-xform-size -2 512`. Run with 256 threads per block and 10 blocks: `./vtx-xform-size 10 256`. Run each kernel multiple times using an input size of 1 bibion (2^{30} vectors): `./vtx-xform-size 0 0 1024`.

Program Output

A run of `vtx-xform-size` produces the following output:

A stray message about a CUDA API call.

```
[koppel@dmk-laptop intro-vtx-transform]$ ./vtx-xform-size
Call of cuDeviceGetCount, cbid 4, serial 1
```

Ignore it (the `Call` message).

Information about each GPU connected to the system, followed by a line showing the chosen GPU.

```
GPU 0: Quadro M2200 @ 1.04 GHz WITH 4010 MiB GLOBAL MEM
GPU 0: L2: 1024 kiB   MEM<->L2: 88.1 GB/s
GPU 0: CC: 5.2  MP:  8  CC/MP: 128  DP/MP:  4  TH/BL: 1024
GPU 0: SHARED: 49152 B/BL  98304 B/MP  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 1061 SP GFLOPS  33 DP GFLOPS  COMP/COMM:  48.2 SP  3.0 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 5.2 (a cheap Maxwell). 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. 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 mxv_g_only:
       0 shared, 1088 const, 0 loc, 32 regs; 1024 max threads per block.
For mxv_i_lbuf:
       0 shared, 1088 const, 0 loc, 32 regs; 1024 max threads per block.
For mxv_o_lbuf:
       0 shared, 1088 const, 0 loc, 32 regs; 1024 max threads per block.
For mxv_o_per_thd:
       0 shared, 1088 const, 0 loc, 31 regs; 1024 max threads per block.
For mxv_vec_ld:
       0 shared, 1088 const, 64 loc, 40 regs; 1024 max threads per block.
For mxv_vls:
   16384 shared, 1088 const, 0 loc, 48 regs; 1024 max threads per block.
For mxv_sh:
   36864 shared, 1088 const, 0 loc, 32 regs; 1024 max threads per block.
For mxv_sh_ochunk:
    4096 shared, 1088 const, 0 loc, 54 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.

```
Matrix size: 16 x 16.  Vectors: 1048576.   8 blocks of 1024 thds.
```

The input size can be changed using command-line arguments, that is explained further below.

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 8 blocks of up to 1024 threads.
mxv_g_only      32 wp     6118 μs    43.873 GF    21.937 GB/s   1.21 I/F   12.5%
mxv_i_lbuf      32 wp     6836 μs    39.266 GF    19.633 GB/s   1.22 I/F   12.5%
mxv_o_lbuf      32 wp     6905 μs    38.873 GF    19.437 GB/s   1.22 I/F   12.5%
mxv_o_per_thd   32 wp     4602 μs    58.331 GF    29.166 GB/s   3.44 I/F   12.5%
mxv_vec_ld      32 wp     4144 μs    64.776 GF    32.388 GB/s   1.22 I/F   50.0%
mxv_vls         32 wp     2606 μs   103.017 GF    51.509 GB/s   1.50 I/F  100.0%
mxv_sh          32 wp     4084 μs    65.733 GF    32.867 GB/s   4.14 I/F  100.0%
mxv_sh_ochunk   32 wp     2598 μs   103.337 GF    51.669 GB/s   2.00 I/F  100.0%
```

The μs values are the execution time, the `GF` values show the floating point throughput (assuming NM FP operations per input vector), and the `GB/s` value is the off-chip data throughput, assuming that $4(N + M)$ B crosses the chip boundary for each element.

The `I/F` value shows the number of instructions executed divided by a *reference* number of instructions. The reference number is based on the number of instructions we expect a good compiler to emit, perhaps simplifying things a bit. The reference is currently set to $MN + N + M + 4 + 3$. Search for `num_ops` in the `main` routine.

A value of 2.00 means that about twice as many instructions were actually executed than we expected. The `I/F` value is computed by actually measuring the number of instructions executed using event counters built in to the GPU, and dividing that count by the number by SNM, the number of multiply-add instructions expected for S inputs. A value of exactly 1 for `I/F` would be

great, but there is no guarantee that the reference number is attainable.

The last column, showing percents, shows read request utilization.

When run with a 0 as the second argument, such as `./vtx-xform-size -1 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 mxv_g_only:
wp ac   t/μs  Lw/μs I/op FP  GB/s Data BW Util---------------------------------
 1  1   4439    2.7  1.1  136   45 ******
 2  2   2413    2.9  1.1  250   83 ***********
 3  3   1964    3.6  1.1  308  103 **************
 4  4   1765    4.3  1.1  342  114 ****************
 8  8   1749    8.6  1.1  345  115 ****************
```

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 `Lw/μs` column is the latency of computing one matrix/vector multiplication. To be precise, it is the run time (for example, 4439 μs) divided by the number of multiplications performed by each thread. For the `mxv_g_only` kernel that would be $\frac{S}{BG}$. For the `mxv_o_per_thd` kernel, in which M threads perform the same computation, the total time is divided by $\frac{SM}{BG}$.

The FPθ column shows the FP throughput based on the measured execution time and the assumption that SMP floating point operations were performed. The number under `GB/s` is the minimum off-chip bandwidth, computed by dividing $4S(M+N)$ 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. Note that this number is computed using measured time and an ideal amount of data crossing the chip boundary.

When run using a `p` instead of a `0`, `vtx-xform-size` collects hardware utilization data related to load and store instructions. A sample is shown below.

```
Kernel mxv_sh_ochunk:
                          R-Eff-% -L2-Cache-- ---DRAM----
wp ac   t/μs  Lw/μs I/op  Ld  St  Rd   Wr   Rd   Wr  FP  GB/s Data BW Util-
 1  1   6008    0.5  1.5 100  50  18.8  33.5  18.8  16.8  101   34 *
 2  2   3294    0.5  1.5 100  50  32.3  61.1  32.6  30.6  183   61 **
 3  3   2365    0.5  1.5 100  50  44.6  85.1  44.6  42.6  255   85 ***
 4  4   1926    0.6  1.5 100  50  54.4 104.5  54.3  52.3  314  105 ****
 8  8   1145    0.7  1.5 100  50  90.0 175.9  89.9  88.0  528  176 *******
```

The `R-Eff-%` columns show the average percentage of each load and store request that is used. The 100% for loads is ideal. The columns under `-L2-Cache--` show the throughput from the L2 caches to the MPs due to load and store instructions, in GB/s. (Load instructions result in read requests, store instructions result in write requests.) The columns under `---DRAM----` show the throughput from DRAM (off chip) to the L2 cache and the L2 cache to DRAM, respectively, in

GB/s. These four throughput values are based on request sizes, not on how much of those requests are actually needed. This data is collected using event counters.

Problem 1: Kernel `mxv_o_per_thd` performs poorly due to load request underutilization. Initially kernel `mxv_o_per_thd_sol` is identical to `mxv_o_per_thd`. Modify `mxv_o_per_thd_sol` so that it loads the input vector components as an `Elt_Type4` rather an `Elt_Type`. See kernel `mxv_vec_ld` for an example of how to load and use this data type. Like `mxv_o_per_thd`, kernel `mxv_o_per_thd_sol` should still assign `M` threads per matrix/vector multiplication. The only difference should be is in how it loads its input vector components.

A correct solution should result in higher performance.

Problem 2: Kernel `mxv_sh_ochunk` uses shared memory to distribute input vector components to threads, enabling it to avoid read request underutilization. It also assigns 8 (the value of `mxv_sh_ochunk_CS`) threads to each matrix/vector multiplication.

Modify `mxv_sh_ochunk_sol` (which is initially identical to `mxv_sh_ochunk`) so that it uses warp shuffle instructions to distribute input vector components instead of using shared memory.

Spoiler Alert: A correct solution should result in about equal performance.

Problem 3: One might expect `mxv_sh_ochunk_sol` to run faster because the store to shared memory:

```
vxfer[threadIdx.x] =
  d_app.d_in[ ( hb + thd_v_offset ) * N + c + thd_c_offset ];
```

is no longer needed. We still need to use the warp shuffle instructions to read the value, and maybe we can expect that there will be one warp shuffle in the solution kernel for each shared load in the original kernel. Based on this we might expect there to be fewer instructions and for the interval latency to be lower.

Determine how the following factors effect these expectations:

- The number of shuffle functional units.
- The actual instructions generated by the compiler.
- Differences in critical path.