LSU EE 7722 **Homework 3** **Due: 1 April 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/hw03`.

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/hw03` 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/hw03` 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 `hw03`, and one with the suffix `-debug`, such as `hw03-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.

The makefile will compile code for a GPU on the system it was run. If there are multiple GPUs the makefile will compile for a 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 and CUDA version for which the executable was built matches the GPU on the current system, and re-build if needed.

Overview of `vtx_xform_sparse` and `hw03`

The code in `hw03.cu` is based on the `vtx_xform_sparse` code. The description below applies to this assignment, `hw03`, and `vtx_xform_sparse`.

The code in `vtx-xform-sparse.cu` contains several kernels that compute a matrix/vector product calculation 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 components and matrix elements are of type `Elt_Type`, which is hardcoded to `float`.

Constant variable `d_app.d_in` points to an SN-element array of `Elt_Type`. That array holds S vectors, each vector has N components.

Unlike the `vtx-xform-size` code, in `vtx-xform-sparse` there is an S-element array `d_op` which is used to determine whether or not to perform a calculation. Each element of `d_op` is a uniformly distributed random number. A calculation is to be performed for element `h` if `d_op[h] <= d_app.norm_threshold`, otherwise element `h` is *skipped*. Boolean variables with names like `work` are true if a calculation is to be performed.

The value in `d_app.norm_threshold` is called the *threshold* (of course) and is set based on a *work density*. The work density, w, is in the range $[0, 1]$. If $w = 1$ then `d_app.norm_threshold` is set to the maximum possible random number, and so a calculation will be performed for every array element. If $w = 0$ `d_app.norm_threshold` is set below the minimum random number, so no work is done, etc.

All of the kernels in `vtx-xform-sparse` are based on the `ochunk` kernel from `vtx-xform-size`. In the `ochunk` kernel each matrix/vector multiply (calculation) is performed by one thread.

(In contrast, kernel `mxv_o_per_thd` each calculation is performed by M threads, one thread per output component.) To avoid request underutilization threads in `ochunk` cooperate when loading and storing input and output vectors.

The Five Kernels

The `hw03.cu` file runs five kernels, named `mxv`, `mxv_atomic`, `mxv_pfx_shared`, `mxv_pfx_ballot`, and `mxv_pfx_prob1`.

Kernel `mxv` handles sparsity in a simple way and is only efficient when density is close to 1. In `mxv` an individual thread will skip work when it needs to, but this does not reduce execution time unless all threads in a warp skip work. When the density is 0.5 on average half the threads are idle. A warp ballot instruction (intrinsic `__all_sync`) is used to detect when none of the threads have work, in which case no work will be done.

In part using templates, the code constructs three kernels: `mxv_atomic`, `mxv_pfx_shared`, and `mxv_pfx_ballot`. Each of these is an instantiation of `mxv_compress` with a different compress function. At each iteration the compress function writes a work assignment array, `worka`. It is set so that thread `threadIdx.x` operates on the same element that thread `worka[threadIdx.x]` would have.

```
  __shared__ short worka[MAX_BLOCK_SIZE]; // Work assignment.
  for ( int hb = bl_start; hb<stop; hb += num_threads ) {
      const bool work = d_app.d_op[hb + threadIdx.x] <= d_app.norm_threshold;
      CData work_info = compress(work,worka);
      const int work_pos = work_info.amt_work;
      const bool skip = threadIdx.x >= work_pos;
```

The three compress functions are `compress_atomic`, `compress_prefix_shared`, and `compress_prefix_ballot`. The `compress_atomic` function is simplest. A shared variable, `n_w_work`, which is initialized to zero, indicates how many thread wrote `worka` so far. Threads with work use an atomic add to increment `n_w_work`, the return value (the value before the increment) indicates the position in which to write `worka`:

```
__device__ CData compress_atomic(bool have_work, short* const &worka) {
  __shared__ int n_w_work; // Number of threads with work.
  __syncthreads();
  if ( threadIdx.x == 0 ) n_w_work = 0;
  __syncthreads();
  if ( have_work ) worka[atomicAdd(&n_w_work,1)] = threadIdx.x;
}
```

This code works well if the `atomicAdd` library function is efficiently implemented. Rather than relying on `atomicAdd` the code in `compress_prefix_shared` computes its own prefix sum. It uses a straightforward, but not well tuned parallel prefix sum algorithm. A more efficient prefix sum algorithm is used in `compress_prefix_ballot`. Efficiency is improved over `compress_prefix_shared` by using warp ballot and population count instructions to find the prefix within a warp, taking advantage of the fact that the prefix sum is of one-bit values.

See the code for additional details.

Running `vtx_xform_sparse` and `hw03`

The `hw03` and `vtx_xform_sparse` code takes up to three arguments. The first, `arg1` indicates the desired number of blocks per SM. A value of 2 on a device with 20 SMs will launch a kernel with 40 blocks. The second argument, `arg2`, indicates the block size, in warps, and whether

performance data should be collected. If `arg2` starts with a `p` then detailed device data, such as request utilization, will be collected. The remainder of `arg2` is the number of warps per block to launch. If both `arg1` and `arg2` (after the `p`) are 0, the launch configuration will be chosen to maximize the number of warps per SM. If `arg1` is 0 then the number of blocks will be chosen to maximize the number of active blocks per SM. If `arg2` is 0 then the block size will be set to the largest value for which there will be `arg1` active blocks per SM.

Suppose `hw03` is run on a GPU with 20 SMs. Running `./hw03 2 32` will launch 40 blocks, each with 32 warps. The number of active blocks per SM in this case may be 1 or 2, depending on the amount of shared memory and number of registers in the respective kernel. Running `./hw03 2 0` will guarantee 2 blocks per SM, and the block size will be the largest for which 2 blocks fit on an SM. (A CUDA API routine is used to determine the number of active blocks per SM. Nothing actually forces that number of blocks to be resident, nor is the number of active blocks checked. Therefore, if the API routine is wrong then the number of active blocks per SM will be wrong.)

Running `./hw03 2 p32` will do the same kind of launch as a previous example, but will collect detailed data. Running `./hw03 2 0` will will launch 40 blocks and choose the largest block size, say 16 warps, for which 2 blocks can be active. Running `./hw03 0 0` will pick the block size and number of blocks that will maximize the number of warps per SM.

The third argument, `arg3`, specifies the number of vectors per SM. A value of v indicates that there should be $v2^{20}$ vectors per SM. (That means the input array has $Nv2^{20}$ floats and the output array has $Mv2^{20}$ floats.) For example, Running `./hw03 0 0 2.5` will run with $20 \times 2.5 \times 2^{20}$ vectors.

Setting `arg3` too high will overflow the signed 32-bit integers used to index the various arrays.

Program Output

A run of `vtx-xform-sparse` and `hw03` 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: GeForce GTX 1080 @ 1.73 GHz WITH 8119 MiB GLOBAL MEM
GPU 0: L2: 2048 kiB   MEM<->L2: 320.3 GB/s
GPU 0: CC: 6.1  MP: 20  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: 4438 SP GFLOPS  139 DP GFLOPS  COMP/COMM:  55.4 SP  3.5 DP
GPU 1: GeForce GTX 1080 @ 1.73 GHz WITH 8111 MiB GLOBAL MEM
GPU 1: L2: 2048 kiB   MEM<->L2: 320.3 GB/s
GPU 1: CC: 6.1  MP: 20  CC/MP: 128  DP/MP:  4  TH/BL: 1024
GPU 1: SHARED: 49152 B/BL  98304 B/MP  CONST: 65536 B  # REGS: 65536
GPU 1: PEAK: 4438 SP GFLOPS  139 DP GFLOPS  COMP/COMM:  55.4 SP  3.5 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 6.1 (a cheap Pascal). 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 (FP32 or single-precision functional units) per MP, `DP/MP` indicates the number of double-precision (FP64) 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:
   32800 shared, 336 const, 0 loc, 24 regs; 1024 max threads per block.
For mxv_atomic:
   34852 shared, 336 const, 0 loc, 40 regs; 1024 max threads per block.
For mxv_pfx_shared:
   34852 shared, 336 const, 0 loc, 40 regs; 1024 max threads per block.
For mxv_pfx_ballot:
   34852 shared, 336 const, 0 loc, 40 regs; 1024 max threads per block.
For mxv_pfx_prob1:
   34852 shared, 336 const, 0 loc, 46 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 6.1 device).

Next, the program provides information on the input size.

```
Matrix size: 8 x 8.  Vectors: 20971520.
```

The input size shown above is the entire input size. Using the command-line arguments the input size *per SM* is specified. So in the example above the input size per SM is 20971520/20 = 1048576. The input size can be changed using command-line arguments, that is explained further below.

The program will launch each kernel several times, the first launch at a density (work) of 1, with subsequent launches at lower densities, with the last launch at a density of zero. The density is shown under the `work` column. Each line shows the result of one run.

```
Kernel mxv:
wp ac  work  t/s I/op GB/s Data BW Util----------------------------------------
32 32 1.000  7651  1.8  186 ******************************
32 32 0.750  7712  2.3  141 **********************
32 32 0.500  7717  3.3   98 ***************
32 32 0.250  7813  5.8   54 ********
32 32 0.021  4151  9.9   27 ****
32 32 0.004  1412  4.7   63 **********
32 32 0.000   545  2.4  154 ************************
```

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 SM (which is the product of the number

of warps per block and the number of active blocks per SM). 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 SMs and so there is no way to have more than one block per SM.

The $t/\mu s$ column shows the measured execution time, the `GB/s` value is the off-chip data throughput, assuming that $4S + 4Sw(N + M)$ B crosses the chip boundary, where S is the number of vectors, w is the work density and N and M are the number of components in the input and output vectors.

The `I/op` 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 $w(MN + N + M) + 4 + 3$. 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/op` 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 the reference amount. A value of exactly 1 for `I/op` would be great, but there is no guarantee that the reference number is attainable.

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 `Data BW 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 the first character of the second argument is a `p` additional data will be collected and shown. The output from a run using `./hw03 0 p` appears below:

```
Kernel mxv:
                                R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/s I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
32 64 1.000  7707  1.8 100.0% 100  50  98.0 174.1  98.0  87.1  185 *******
32 64 0.750  7700  2.3  93.7% 100  50  98.0 174.3  98.0  87.2  142 *****
32 64 0.500  7700  3.3  87.5% 100  50  98.1 174.3  98.1  87.2   98 ***
32 64 0.250  7695  5.8  81.3% 100  50  98.1 174.4  98.1  87.2   55 **
32 64 0.021  4122  9.9  76.1% 100  50 101.9 163.0 101.9  81.6   27 *
32 64 0.004  1367  4.7  75.8% 100  50 123.1 123.4 123.1  61.9   65 **
32 64 0.000   475  2.4  78.4% 100  50 176.6   0.1 176.5   2.1  176 *******
Kernel mxv_atomic:
                                R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/s I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
24 48 1.000  7531  3.0  74.1% 100  48 100.3 183.8 100.3  89.1  189 *******
24 48 0.750  6744  3.2  70.8% 100  49  87.5 152.1  87.2  74.7  162 ******
24 48 0.500  4874  3.5  66.4% 100  49  86.7 141.5  86.4  69.0  155 ******
24 48 0.250  3262  4.2  55.9% 100  47  78.1 108.6  77.8  51.6  129 *****
24 48 0.021  1242  6.9  16.5% 100  34  81.6  34.5  81.5  12.3   91 ***
24 48 0.004  1056  7.1   6.6% 100  13  85.8  21.0  85.8   3.6   85 ***
24 48 0.000   670  6.1   3.2% 100  39 125.2   0.0 125.2   1.5  125 *****
Kernel mxv_pfx_shared:
                                R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/s I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
24 48 1.000  7637  5.5  69.1% 100  48  98.9 181.2  98.9  87.9  187 *******
```

```
24 48 0.750  6813  6.5  66.0% 100  49  86.6 150.5  86.2  73.9  160 ******
24 48 0.500  5022  8.1  62.2% 100  49  84.1 137.3  83.7  66.9  150 ******
24 48 0.250  3801 12.2  55.9% 100  47  67.0  93.2  66.5  44.3  110 ****
24 48 0.021  3410 32.6  45.7% 100  34  29.7  12.6  29.1   4.5   33 *
24 48 0.004  3306 38.5  45.5% 100  13  27.4   6.7  26.5   1.1   27 *
24 48 0.000  3054 39.3  47.4% 100  39  27.5   0.0  27.5   0.3   27 *
Kernel mxv_pfx_ballot:
                              R-Eff-% -L2-Cache-- ---DRAM----
wp ac  work  t/s I/op SM eff  Ld  St  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util-
24 48 1.000  7546  2.9  74.1% 100  48 100.1 183.4 100.1  89.0  189 *******
24 48 0.750  6769  3.1  70.8% 100  49  87.2 151.5  86.9  74.4  161 ******
24 48 0.500  4855  3.3  66.4% 100  49  87.0 142.1  86.7  69.2  156 ******
24 48 0.250  3284  3.8  55.9% 100  47  77.6 107.9  77.3  51.3  128 *****
24 48 0.021  1137  6.6  13.3% 100  34  89.1  37.7  89.0  13.4   99 ****
24 48 0.004   958  7.5   3.9% 100  13  94.6  23.2  94.6   3.9   93 ***
24 48 0.000   600  6.9   1.1% 100  39 139.9   0.0 139.9   1.6  140 *****
```

The `SM eff` column shows the efficiency of shared loads. Lower values indicate more bank conflicts. 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: None of the kernels perform well when the density is low. The reason is that the amount of work in an entire block is too low to keep the device busy. For example, at a density of $1 - \sqrt[32]{1/2}$ there is a probability of $(1 - (1 - \sqrt[32]{1/2}))^{32} = .5$ that a warp has no work, and the expected number of threads with work is $B(1 - \sqrt[32]{1/2}) = 0.0214B$, where B is the block size. For a 1024-thread block on average 21.9 threads will be active, not even a single warp. The underlying problem is that compress routine gathers work from B candidates (one per thread), which is too few when the density is low.

For this problem improve on the situation by modifying `mxv_compress_prob1` so that it examines input elements until it finds enough work to keep all threads in a warp busy (or until it reaches the end of the array). The code in `mxv_compress_prob1` is similar to the `mxv_compress` kernel, except that the there is no call to a `compress` function. Instead, the `compress_prefix_ballot` code has been copied in.

One way to solve this is to have each warp gather its own work. At each `hb` iteration threads in a warp will add their work to `worka` (or some other array). When a warp has 32 elements it computes the products. The advantage of having each warp gather its own work is that there will be no need for frequent `__syncthreads` calls.

When analyzing the performance of the sparse kernels, pay attention to the number of iterations of the `hb` loop. The default vector size is 2^{20} elements per SM. Consider a launch with one 1024-thread block per SM and a density of $1 - \sqrt[32]{7/8} = 0.00416$. In that case there will be just 4 elements per `hb` iteration and only about four "full" iterations of work.