LSU EE 7722 **Homework 1** **Due: 12 February 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/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 `.../2019/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 `.../2019/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 `-debug`, such as `hw01-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.

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 like `Run completed, total errors: 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 `hw01`

The code in `hw01.cu` contains several kernels that operate on an array of structures. See Problem 1 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.

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.

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 "0.5" will result in a 2^{19} 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

A run of `hw01` produces the following output:

A stray message about a CUDA API call.

```
[koppel@dmk-laptop hw01]$ ./hw01
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 ss_g_only:
       0 shared, 48 const, 0 loc, 23 regs; 1024 max threads per block.
For ss_l1ro:
       0 shared, 48 const, 0 loc, 25 regs; 1024 max threads per block.
For ss_sh:
       0 shared, 48 const, 0 loc, 25 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: 1048576. Grid: 13 blocks of 1024 thds. Structure Size: 36 B, slen = 28
```

```
Launching with 13 blocks of up to 1024 threads.
```

The array size can be changed using command-line arguments, that is explained further below. Variable `slen` controls the size of the structure.

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 20 blocks of up to 1024 threads.
ss_g_only       32 wp      712 µs    45.650 GF   106.026 GB/s   4.33 I/F   12.5%
ss_l1ro         32 wp      847 µs    38.366 GF    89.107 GB/s   4.26 I/F   12.5%
ss_sh           32 wp      389 µs    83.537 GF   194.021 GB/s   5.27 I/F  100.0%
```

The μs values are the execution time, the `GB/s` value is the off-chip data throughput, and `I/F` is a rough measure of how many instructions are executed.

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 a hand computed ideal number of instructions. A value of exactly 1 for `I/F` would be suspiciously low because that would mean there could be no load instructions to read the input nor stores to write the output.

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

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 ss_g_only:
wp ac t/µs I/op GB/s Data BW Util-------------------------------------------------
 4  4  786  4.3   96 *****************
 8  8  766  4.3   99 ******************
12 12  758  4.3  100 ******************
16 16  742  4.3  102 ******************
20 20  730  4.3  103 *******************
24 24  731  4.3  103 *******************
28 28  717  4.3  105 *******************
32 32  727  4.3  104 *******************
```

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 $4S(36+36)$ 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`, `hw01` collects hardware utilization data related to load and store instructions. A sample is shown below.

```
Kernel ss_g_only:
                Req U %      -L2 Cache-- ---DRAM----
wp ac t/μs I/op  Ld  St  Shx  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util----------
 4  4  672  4.3  12  13  0.0  56.2 449.7  56.2  56.1  112 *******
 8  8  644  4.3  12  13  0.0  58.6 468.6  58.6  58.8  117 ********
12 12  646  4.3  12  13  0.0  58.4 467.2  58.4  58.6  117 ********
16 16  636  4.3  12  13  0.0  59.4 475.2  59.4  59.7  119 ********
20 20  623  4.3  12  13  0.0  60.6 484.5  60.6  61.6  121 ********
24 24  632  4.3  12  13  0.0  59.7 477.7  59.7  63.2  119 ********
28 28  613  4.3  12  13  0.0  61.6 492.9  61.6  62.6  123 ********
32 32  613  4.3  12  13  0.0  61.6 492.5  61.6  62.4  123 ********
```

The two columns under `Req U %` indicate the average percentage of each load and store request that is used. The 12% value shown above is what one would expect for scattered accesses to 4-byte values. The `Shx` column shows the number of times shared memory is read for each shared load. Zero indicates that shared memory isn't being used. Otherwise, 1 is ideal and higher numbers indicate serialization due to bank conflicts. For example, a 2 would indicate that on average shared loads have to be done twice.

The columns under `-L2 Cache--` show the throughput from the L2 caches to the MPs due to load and store instructions, in GB/s. 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 also collected using event counters.

Problem 1: The kernels `ss_g_only`, `ss_l1r0`, and `ss_sh` each have an input `ss_in` which is the address of an array of structures, and `ss_out` which is the address of another array of structures. Both arrays have `app.n_elts` elements (the element is a structure). Kernel `ss_g_only` is written so that the read-only cache will be used for the input array, whereas with `ss_l1r0` there is no MP caching. Those two kernels are to be compared for this problem. Kernel `ss_sh` is to be used in the next problem.

The kernels read in an element perform an operation, and write out the modified structure. The structure and one of the kernels appears below:

```
const int slen = 28;
struct Some_Struct
{
  float f0, f1;
  char str[slen];
};
extern "C" __global__ void ss_l1r0(Some_Struct* ss_out, const Some_Struct* ss_in)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;
  for ( int h=tid; h<d_app.n_elts; h += num_threads )
    {
      Some_Struct elt = ss_in[h];
      Some_Struct elt_out;
      bool ord = elt.f0 <= elt.f1;
      elt_out.f0 = ord ? elt.f0 : elt.f1;
      elt_out.f1 = ord ? elt.f1 : elt.f0;
      int delta = elt.f0 == elt.f1 ? 0 : ord ? -1 : 1;
      for ( int i=0; i<slen; i++ ) elt_out.str[i] = elt.str[i] + delta;
      ss_out[h] = elt_out;
    }
}
```

The method used to assign array elements to a thread (the way that `h` is computed) is the same method used elsewhere in this course, for example, the code samples in `vtx_xform_size`. If `ss_in` were an array of floats, then requests would be used efficiently. However the default structure size is 36 B, and the compiler will use $36/4 = 9$ load instructions to load `ss_in[h]` from global memory. (The compiler is smart enough to use a 32-bit load to load 4 consecutive `elt.str` elements. The data below show that as expected, the request utilization (`Req U %`) is about 1/8.

```
Kernel ss_l1r0:
                Req U %      -L2 Cache-- ---DRAM----
wp ac t/µs I/op  Ld  St  Shx  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util----------
 4  4  778  4.3  12  13  0.0 388.2 388.1  48.5  48.7   97 ******
 8  8  756  4.3  12  13  0.0 399.6 399.6  49.9  50.1  100 ******
12 12  741  4.3  12  13  0.0 407.8 407.8  51.0  51.1  102 *******
16 16  731  4.3  12  13  0.0 413.3 413.3  51.7  51.9  103 *******
20 20  714  4.3  12  13  0.0 423.0 423.0  52.9  53.1  106 *******
24 24  713  4.3  12  13  0.0 423.8 423.7  53.0  53.5  106 *******
28 28  705  4.3  12  13  0.0 428.4 428.4  53.5  54.0  107 *******
32 32  713  4.3  12  13  0.0 423.5 423.5  52.9  53.8  106 *******
```

Run the code on one of the workstation computers. In the answers below, indicate which machine you are using.

(*a*) What looks like its limiting the performance of each kernel? Use the performance counter data in your answers.

(*b*) What kind of advantage is the L1 read-only cache providing? Use the performance counter data in your answers.

See 2018 Homework 2 problem 1 for a roughly similar problem.

Problem 2: Initially, the code in kernel `ss_sh` is almost identical to `ss_l1r0`. Modify `ss_sh` so that it uses shared memory to avoid request under-utilization. Notice that in the `ss_sh` a shared array and some punned pointers have been declared:

```
extern "C" __global__ void ss_sh(Some_Struct* ss_out, const Some_Struct* ss_in)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;
  const int ss_size_words = sizeof(Some_Struct) / sizeof(uint32_t);
  __shared__ Some_Struct ss_blk[1024];

  // Cast pointers to Some_Struct to pointers to integers so that
  // Some_Struct data can be moved around as a simple array of integers.
  uint32_t* const ss_blk_wds = (uint32_t*) &ss_blk[0];
  uint32_t* const ss_in_wds = (uint32_t*) &ss_in[0];
  uint32_t* const ss_out_wds = (uint32_t*) &ss_out[0];
```

Modify the code so that it copies elements from global memory using pointer `ss_in_wds` into shared memory using pointer `ss_blk_wds`, performs the operation using pointer `ss_blk`, and then writes the result out using pointers `ss_blk_wds` and `ss_out_wds`. If the problem is solved correctly request utilization for loads and stores should be 100% and performance should be higher. For example,

```
                Req U %      -L2 Cache-- ---DRAM----
wp ac t/µs I/op  Ld  St  Shx  Rd θ  Wr θ  Rd θ  Wr θ GB/s Data BW Util----------
 4  4  421  5.3 100 100  1.0  89.8  89.7  89.7  89.8  179 ************
 8  8  410  5.3 100 100  1.0  92.1  92.0  92.0  92.1  184 ************
12 12  400  5.3 100 100  1.0  94.4  94.3  94.3  94.4  189 ************
16 16  399  5.3 100 100  1.0  94.7  94.7  94.7  94.8  189 *************
20 20  398  5.3 100 100  1.0  94.9  94.9  94.9  94.9  190 *************
24 24  394  5.3 100 100  1.0  95.9  95.8  95.8  95.9  192 *************
28 28  396  5.3 100 100  1.0  95.3  95.3  95.3  95.3  191 *************
32 32  395  5.3 100 100  1.0  95.5  95.4  95.5  95.4  191 *************
```

The output will be tested for correctness, make sure that no errors are reported.