**LSU EE 7722** **Homework 1** Solution **Due: 23 March 2015**

**Problem 0:** Read the following information about the assignment package, and follow instructions on course procedures page, `http://www.ece.lsu.edu/gp//proc.html`, for account setup and Programming Homework Workflow. Try compiling and running the code and familiarize yourself with the command line arguments described below.

The homework package is set to compile for an NVIDIA GPU of compute capability 3.5 (the expensive Kepler). It is recommended that you run your code on such devices (including the machines in the lab). An easy way to determine the CC of the GPU in a lab machine is to consult the computer status Web page, `http://www.ece.lsu.edu/koppel/gpup/sys-status.html`. If you must run on a less capable machine edit the makefile, changing `sm_35` to the CC of your machine.

The code in `hw01.cu` launches a series of kernels, each one reads an $S$-element input array of $N$-element vectors $v(0), v(1), \ldots v(S-1)$ and writes an $S$-element output array of $M$-element vectors $u(0), u(1), \ldots u(S-1)$ with $u(h) = Av(h)$ for $0 \leq h < S$, where $A$ is an $M \times N$ matrix. The vector and matrix elements are of type `float`.

Of course, everyone reading this knows that $u_r = \sum_{c=0}^{N-1} A_{r,c} v_c$, where $u_r$, $0 \leq r < M$ are the components of vector $U$. The total computation for each vector is $MN$ multiply-adds, and the total computation for each kernel is $SMN$. The number of operations needed to complete this computation is larger since instructions are needed to bring operands to the multiply-add instructions and send results back to memory. For half-decent code we can expect the number of instructions to be twice as much (meaning one "overhead" instruction for each multiply-add).

Assuming that nothing is read from global memory twice, the total communication is $S(M+N)$ elements, for the homework code that would be $4S(M+N)$ bytes. The computation to communication ratio is $MN/(M+N)$ floating-point operations per floating-point element transfer.

For this assignment assume $S$ is on the order of a million (the default in the code is $2^{20}$) and that $M$ and $N$ are in the range 4 to 100. For smaller values the computation will be communication limited, and for larger values the computation will be compute limited. An NVIDIA K20c can perform 33.9 single-precision multiply-adds for each float read or written. So for this device the computation will be communication-bound for, say, $M = N = 8$ because $64/16 < 33.9$. Letting $M = N$ and solving $N^2/2N = 33.9$ sets the border at $N = 68$.

For performance reasons the values of $M$ and $N$ are given as compile-time constants. In particular, using `#define` statements. This makes it easier for the compiler to unroll loops and reduce the amount of overhead.

The assignment file has several different versions of the kernel. In kernels `mxv_g_only`, `mvx_i_lbuf`, and `mvx_o_lbuf` each matrix-vector multiply is computed by one thread. In `mxv_o_per_thd` $M$ threads cooperate computing a matrix-vector multiplication. In all cases each thread computes many matrix-vector products.

As we discussed in class, memory is accessed inefficiently by kernels `mxv_g_only`, `mvx_i_lbuf`, `mvx_o_lbuf`, and `mxv_o_per_thd`. The first three kernels typically waste 7/8 of each global memory read and write request. Kernel `mxv_o_per_thd` is efficient with writes, but is just as wasteful as the other with loads.

Kernel `mvx_sh` uses shared memory to help improve global memory read and write efficiency. Kernel `mvx_sh_ochunk` is initially identical to `mvx_sh`, but is to be modified as part of this assignment.

Each run of the code launches all of the kernels. A kernel may be launched once, or if the second argument is `0` (see below) launched for different block sizes. The program output starts with data about the GPUs that it will use:

```
Using GPU 0
```

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 MiB GLOBAL MEM
GPU 0: L2: 1310720 kiB   MEM<->L2: 208.0 GB/s
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  DP/MP: 64  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 1761 SP GFLOPS  587 DP GFLOPS  COMP/COMM:  33.9 SP  22.6 DP
Using GPU 0
```

The execution rates shown above (GFLOPS) count a multiply-add as one operation. The `COMP/COMM` line gives the computation to communication ratio in floating-point operations per floating-point element transfers. The assignment code uses SP by default. Please don't try using DP in this assignment. 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, 16448 const, 0 loc, 40 regs; 1024 max threads per block.
For mxv_i_lbuf:
       0 shared, 16448 const, 0 loc, 81 regs; 640 max threads per block.
For mxv_o_lbuf:
       0 shared, 16448 const, 0 loc, 97 regs; 512 max threads per block.
For mxv_o_per_thd:
       0 shared, 16448 const, 0 loc, 33 regs; 1024 max threads per block.
For mxv_sh:
   36864 shared, 16448 const, 256 loc, 52 regs; 1024 max threads per block.
For mxv_sh_ochunk:
    4096 shared, 16448 const, 1176 loc, 255 regs; 256 max threads per block.
```

Next the program prints the vector sizes and launch configuration:

```
Matrix size: 64 x 64.  Vectors: 1048576.   13 blocks of 1024 thds.
Launching with 13 blocks of up to 1024 threads.
```

If the second argument was non-zero then each kernel is run once. The number of warps used to launch it is shown, along with execution time, and computation and communication rate. The computation and communication rates are based on the assumed number of floating-point operations and an ideal amount of off-chip data transfer.

```
K mxv_g_only      32 wp  1483544.312 s     2.895 GFLOPS     0.362 GB/s
K mxv_i_lbuf      20 wp    39803.745 s   107.904 GFLOPS    13.488 GB/s
K mxv_o_lbuf      16 wp   136011.078 s    31.578 GFLOPS     3.947 GB/s
K mxv_o_per_thd   32 wp   543146.545 s     7.908 GFLOPS     0.988 GB/s
K mxv_sh          32 wp    77260.353 s    55.591 GFLOPS     6.949 GB/s
K mxv_sh_ochunk    8 wp   227258.881 s    18.899 GFLOPS     2.362 GB/s
```

If the second argument is zero then each kernel is run multiple times and an ASCII art bar graph is printed. The output below just shows two kernels:

```
Kernel mxv_i_lbuf:
 4 wp   75914 s     57 GF      7 GB/s ******
 8 wp   50625 s     85 GF     11 GB/s **********
12 wp   41198 s    104 GF     13 GB/s ************
16 wp   38542 s    111 GF     14 GB/s **************
20 wp   39875 s    108 GF     13 GB/s ************
```

```
Kernel mxv_o_lbuf:
 4 wp  287669 s     15 GF      2 GB/s *
 8 wp  171621 s     25 GF      3 GB/s **
12 wp  146592 s     29 GF      4 GB/s ***
16 wp  136087 s     32 GF      4 GB/s ***
```

The code 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). 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, if the second argument is 512, but kernel `foo` has a limit of 256 threads, `foo` will be launched with 256 threads.) If the second argument is zero then each kernel will be launched multiple times starting with 4 warps, incrementing by 4 warps until the kernel maximum is reached. The third argument indicates the number of input and output vectors in mibi-elements. If $a_3$ is the value of the third argument, the number of vectors will be $a_3 2^{20}$. The third argument is read as a floating point number, so "0.5" will result in a $2^{19}$ vectors.

The size of the input and output vectors ($N$ and $M$) is hard-coded and cannot be set using a command-line argument. To change $N$ and $M$ edit `hw01.cu` and re-compile. (With dynamic compilation one *could* set $N$ and $M$ on the command line (or in an input file) and still have the benefit of high-quality code.)

Here are some examples.

Running without arguments: `hw01`. This will use $P$ blocks, where $P$ is the number of multiprocessors, with up to 1024 threads per block. One could get the same result by running using `hw01 0 1024` or `hw01 P 1024` where $P$ is replaced by whatever the number of multiprocessors is.

Run with 256 threads per block: `hw01 0 256`. Run with 256 threads per block and 10 blocks: `hw01 10 256`. Run each kernel multiple times: `hw01 0 0`.

Code notes:

The exact amount of CUDA global memory needed for the output array is $4SM$ bytes, but $4(S+B)M$ bytes is allocated. The extra $4BM$ bytes is called the *overrun* area, and it is okay if the kernel writes it. There is also an $4BN$ byte overrun area on the input array, it is okay if the kernel reads it. In some cases the presence of an overrun area enables simpler and faster code by eliminating the need for symmetry-busting end-of-data checks.

**Problem 1:** GPUs rely on lots of threads to hide latency. But how many threads do we need?

(*a*) Why might `mxv_i_lbuf` require fewer threads to hide latency than `mxv_o_per_thd`? *Note: The original assignment said "more threads" rather than "fewer threads."*

Short answer: Threads in `mxv_i_lbuf` do more work per loaded value and so fewer additional threads are needed to fill in the gaps.

Long answer: For both kernels each `h` loop iteration reads $N$ input vector components. Lets assume that the compiler issues all $N$ loads before using the loaded values. Global memory latency is long, assume about 200 cycles, and denote the latency $L_G$. Once the loaded values arrive each thread in `mxv_i_lbuf` will perform $N \times M$ multiply/adds. But each thread in `mxv_o_per_thread` will perform just $N$ multiply/adds. Latency is hidden when the processor has something else to do during the wait. While one warp is waiting for its data to arrive, other warps can be operating on data that has already arrived. Let $t_C$ denote the total time needed to issue the instructions for a warp that operate on the loaded data. A rough estimate of the number of warps needed to hide latency is then $L_G/t_C$. In `mxv_i_lbuf` $t_c$ will be about $M$ times higher and so $M$ times fewer threads are needed. See the diagram below. *Note: A more detailed analysis of this type will be performed in the next homework assignment. Also see Spring 2013 Homework 6.*

```
Kernel mxv_i_lbuf
   !<-------- L_G ------------>!<------ t_C ---------->!
W0 [LD]                       [MA][MA][MA][MA][MA][MA]
W1     [LD]                                             [MA][MA][MA][MA][MA][MA]
...

Kernel mxv_o_per_thread
   !<-------- L_G ------------>!<-t_C->!
W0 [LD]                       [MA][MA]
W1     [LD]                           [MA][MA]
...
```

(*b*) Run the code and see how the two kernels perform with different block sizes. (Of course, do this by setting the second argument to `0`.) Try this for smaller and larger vector sizes.

**Problem 2:** Kernel `mxv_sh` uses shared memory so that input vector elements can be read efficiently and then distributed to the thread that needs them, the same is done for the output elements.

(*a*) Kernel `mxv_sh_ochunk` is initially identical to `mxv_sh`. Modify it so that `CS` (use a value of 8) threads compute a single matrix-vector multiply. The `CS` threads handling a vector should read input vector elements and redistribute them to other threads computing the same vector. The threads would use these values to partially compute the output elements, and then repeat the process, until the entire input vector is read. Then each thread should write its elements of the output vector.

Try to achieve the following:

- The code should work correctly for values of $M$ and $N$ that are multiples of 8.
- Try to get the code working for other values of $M$ and $N$.
- When threads write the output vector, memory requests should be completely used.
- Try to minimize the number of `synchreads` needed.

Solution checked into repo. The solution is in file `hw01sol.cu`. The solution works correctly for all values of $M$ and $N$, but only works well for smaller multiples of eight.

Recall that the `hb` loop iterates over vectors, the value of `hb` is the vector operated on by thread 0 of the block. Since we are now assigning CS threads per vector the starting value of `hb` and the increment will change, both will be divided by CS.

```
const int bl_start = blockIdx.x * blockDim.x / CS;
const int stop = d_app.num_vecs;
const int inc = num_threads / CS;
```

Variable `hb` is the vector number operated on by thread 0. Since CS threads work on a vector, threads 1 to 7 work on the same vector. The vector number operated on by thread `threadIdx` is `hb + threadIdx/CS`. The solution precomputes:

```
const int thd_v_offset = threadIdx.x / CS;
```

and in the `hb` loop:

```
for ( int hb = bl_start; hb<stop; hb += inc )
```

```
const int vec_num = hb + thd_v_offset;
```

Next, read in input vector components eight at a time and operate on them. The starting position of the vector is `vec_num * N`. In each `c` loop iteration eight components are read, those eight components start at position `vec_num * N + c`. The first thread in a group of 8 reads that element, the second thread reads `vec_num * N + c + 1`, an so on. Variable `thd_c_offset` gives the position of a thread in the group of 8, using that variable a thread loads element `vec_num * N + c + the_c_offset`. The value read is placed into shared memory, since each thread reads one value we can use `threadIdx.x` as the shared memory index. Finally, we need to make sure that we don't read beyond the `N`th component:

```
for ( int c=0; c<N; c += CS )
    vxfer[threadIdx.x] =
      c + thd_c_offset < N
      ? d_app.d_in[ vec_num * N + c + thd_c_offset ]
      : 0;
```

After the transfer from global to shared memory above, each thread reads in eight components:

```
Elt_Type vin[CS];
for ( int cc=0; cc<CS; cc++ )
  vin[cc] = vxfer[ thd_v_offset * CS + cc ];
```

Because each group of 8 is within a warp, there is no need for syncthreads.

The rest of the solution is straightforward.

(*b*) Describe how your kernel works at different input and output vector sizes. Indicate whether you think it should go faster. Indicate whether the results agree with your expectations, and if not provide a possible reason.

When characterizing the performance pay attention to the amount of local memory used by your thread (the number to the left of "loc" in the output showing kernel resource usage). Local memory usage will often result in bad performance.

The solution works well for $M$ and $N$ which are small multiples of 8. On a Kepler K20c it outperforms `mxv_sh` and the other kernels.

For non-multiples of 8 performance is horrible, primarily because the compiler doesn't unroll the `c` loop. (That can be fixed using an unroll pragma.)

Performance is poor for large values of $M$ and $N$, even multiples of 8. The performance drops off even where `mxv_sh` does well. The reason has to do with registers and constant memory.

At first, one might think that `mxv_sh` would be at a disadvantage when it comes to registers. Each thread in `mxv_sh` has an $M$-element local array to buffer output components, we expect the compiler to assign registers for these. Kernel `mxv_o_chunk` has just an $M/8$-element local array and so uses fewer registers for these variables.

The problem is with access to `d_app.matrix`. Recall that an NVIDIA Kepler arithmetic instruction operand can be a constant memory value. For example,

```
/*05c8*/                   FFMA R33, R13, c[0x3][0xf8], R33;
```

in the instruction above the second source operand, `c[0x3][0xf8]` is from constant memory. However, a constant memory value can only appear as an operand if the address is known at compile time. Otherwise constant memory must be loaded using a `ldc` instruction. For example,

```
/*0090*/                   LDC R17, c[0x3][R19+0x4];
/*0158*/                   FFMA R28, R17, R21, RZ;
```

In `mxv_sh` each thread accesses the same value of matrix, when the loop below is unrolled each access to `matrix` is at a location known to the compiler and the same for all threads. (In other words, `threadIdx.x` nor anything else

that can vary from thread to thread is not used to compute `r`, `c`, or `cc` below, they are computed from constants `M` and `CS`.

```
for ( int r=0; r<M; r++ )
  for ( int cc=0; cc<CS; cc++ )
    if ( c+cc < N ) vout[r] += d_app.matrix[r][c+cc] * vin[cc];
```

The situation for `mxv_o_chunk` is different:

```
for ( int rr=0; rr<ML; rr++ ) {
    const int r = rr * CS + thd_r_offset;
    for ( int cc=0; cc<CS; cc++ )
     vout[rr] += d_app.matrix[r][c+cc] * vin[cc]; }
```

Here `r` does depend on `threadIdx.x` and so will be different for each thread. Since the address is not constant the compiler can't use a constant space operand and instead must use a `ldc` to load a register. When `M` and `N` are small the needed values of `matrix` can be loaded into registers before the `hb` loop, and so execution will still be efficient. But if there are not enough registers then `ldc` instructions must be placed inside the `hb` loop slowing things down.