LSU EE 7722 **Homework 3** **Due: 9 April 2018**

Basic Setup

Follow the instructions for class account setup found on
`http://www.ece.lsu.edu/gp/proc.html`. This assignment uses code in the hw03 directory, file hw03.cu is to be modified.

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 `.../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 `.../hw03` is the current directory). The makefile builds three versions of each program, named `hw03`, `hw03-debug`, and `hw03-cuda-debug`. The versions with the `-debug` suffix are compiled with host optimization turned off, which facilitates debugging. The `hw03-cuda-debug` version is compiled for CUDA kernel debugging, using cuda-gdb. It runs without performance data (such as the data plotted under `I/op`) and is not optimized. Use `hw03-cuda-debug` when using cuda-gdb to debug kernel code. Host code can be debugged that way too, but execution is faster when gdb is used on `hw03-debug`.

Quickly check whether the build is successful with the command `./hw03`. It should produce a table showing data on runs of several kernels. The last few lines should look something like:

```
mxv_sh_ochunk_sol_mn 32 wp   598991 s     7.170 GF     0.896 GB/s   2.59 I/F
mxv_sh_ochunk_mn     32 wp    23329 s    11.507 GF     5.753 GB/s   6.84 I/F
mxv_sh_ochunk_mn     32 wp   100838 s    10.648 GF     2.662 GB/s   3.28 I/F
mxv_sh_ochunk_mn     32 wp   597686 s     7.186 GF     0.898 GB/s   2.59 I/F
```

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 `hw03`

The code in `hw03.cu` is based on `vtx-xform-size.cu` and accepts the same command-line arguments. It runs multiple kernels based on `mxv_sh_ochunk` from `vtx-xform-size.cu`. Recall that `vtx-xform-size.cu` contained many kernels that multiplied vectors by a fixed matrix $M \times N$ matrix. The `hw03` code operates only on square ($N \times N$) matrices, but uses multiple matrix sizes in a single run.

Each kernel reads S n-component input vectors and writes S n-component output vectors. The value of S is specified by a command-line option (see below), and is the same for all kernels in a run. The vector size, n, varies from kernel to kernel. It is shown in parenthesis after the kernel name, such as `mxv_sh_ochunk_mn(32,32)`. Vector components and matrix elements are of type `Elt_Type`, which is hardcoded to `float`.

Routine `mxv_sh_ochunk` is a templated routine based on the routine with the same name from `vtx-xform-size.cu`. The template argument indicates the matrix size, the code runs three specializations named `mxv_sh_ochunk_16`, `mxv_sh_ochunk_32`, and `mxv_sh_ochunk_64`. The compiler does a good job with `mxv_sh_ochunk_16` and `mxv_sh_ochunk_32` but does poorly with kernel `mxv_sh_ochunk_64`.

Routine `mxv_sh_ochunk_mn` is similar to `mxv_sh_ochunk` except that the matrix size is read from `d\_app.n`. The advantage is that the code can run for any matrix size (actually, any n that's a multiple of 8 and not greater than 64). The disadvantage is that the compiled code will not be nearly as efficient because the compiler does not know the matrix size. See Problem 1 for more

details. Routine `mxv_sh_ochunk_sol_mn` is initially identical to `mxv_sh_ochunk_mn`. Modify the former for your solution and compare against the later.

The `hw03` 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 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: `./hw03 0 256`. Run with 512 threads per block and twice as many blocks as MPs: `./hw03 -2 512`. Run with 256 threads per block and 10 blocks: `./hw03 10 256`. Run each kernel multiple times using an input size of 1 bibion (2^{30} vectors): `./hw03 0 0 1024`.

Program Output

A run of `hw03` produces the following output:

A stray message about a CUDA API call.

```
[koppel@dmk-laptop hw03]$ ./hw03
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_sh_ochunk_16:
    4096 shared, 16456 const, 0 loc, 56 regs; 1024 max threads per block.
For mxv_sh_ochunk_32:
    4096 shared, 16456 const, 0 loc, 160 regs; 384 max threads per block.
For mxv_sh_ochunk_64:
    4096 shared, 16456 const, 1200 loc, 255 regs; 256 max threads per block.
For mxv_sh_ochunk_sol_mn:
    4096 shared, 16456 const, 32 loc, 40 regs; 1024 max threads per block.
For mxv_sh_ochunk_mn:
    4096 shared, 16456 const, 32 loc, 40 regs; 1024 max threads per block.
```

The `max threads per block` shown above is based on the kernel and reflects register usage. The number to the left of `loc` shows the amount of local memory used per thread, and the number of the left of `regs` is the number of registers used. For the kernels used in this assignment any use of local memory is bad. The solution should be coded so that the compiler emits code using registers for expressions using the local address space. This is what happens with `mxv_sh_ochunk_16` and `mxv_sh_ochunk_32`.

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

```
Max matrix: 64 x 64.  Num vectors (S): 1048576.   Grid size: 8 blocks.
Elements per thread: 1024.0 (4 wp) - 128.0 (32 wp)
```

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_sh_ochunk_16     32 wp     2614 s   102.680 GF    51.340 GB/s   2.00 I/F
mxv_sh_ochunk_32     12 wp     5179 s   207.310 GF    51.828 GB/s   1.33 I/F
mxv_sh_ochunk_64      8 wp   162291 s    26.465 GF     3.308 GB/s   2.16 I/F
mxv_sh_ochunk_sol_mn 32 wp    24161 s    11.110 GF     5.555 GB/s   6.84 I/F
mxv_sh_ochunk_sol_mn 32 wp   101836 s    10.544 GF     2.636 GB/s   3.28 I/F
mxv_sh_ochunk_sol_mn 32 wp   600334 s     7.154 GF     0.894 GB/s   2.59 I/F
mxv_sh_ochunk_mn     32 wp    25267 s    10.624 GF     5.312 GB/s   6.84 I/F
mxv_sh_ochunk_mn     32 wp   101971 s    10.530 GF     2.632 GB/s   3.28 I/F
mxv_sh_ochunk_mn     32 wp   597203 s     7.192 GF     0.899 GB/s   2.59 I/F
```

The μs values are the execution time, the `GF` values show the floating point throughput (assuming N^2 FP operations per input vector), and the `GB/s` value is the off-chip data throughput, assuming that $8N$ B crosses the chip boundary for each element.

The `I/F` value shows the number of instructions executed per expected multiply-add. A value of 2.00 means that $2N^2$ instructions are executed for each input vector. Presumably one of those

instructions is a multiply-add. 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 SN^2, the number of multiply-add instructions expected for S inputs. 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 vector nor stores to write the output vector. Including these would give a ratio of $\frac{N^2+2N}{N^2} = 1 + \frac{2}{N}$ or 1.125 for $M = N = 16$.

When run with a 0 as the second argument, such as `./hw03 -1 0`, the program, launches each kernel multiple times, starting with 4 warps per block, up to 32 warps per block or the kernel's maximum. Run time and other information will be shown for each launch. An excerpt for one kernel appears below:

```
Kernel mxv_sh_ochunk_32(32,32):
wp ac   t/s I/op GB/s FP  FP Utilization---------------------------------------
 4  4  10084  1.3   27  106 *****
 8  8   6813  1.3   39  158 *******
12 12   5154  1.3   52  208 **********
```

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 FPθ column shows the FP throughput based on the measured execution time and the assumption that SN^2 floating point operations were performed. The number under `GB/s` is the minimum off-chip bandwidth, computed by dividing $4S(2N)$ by the measured execution time. The stars in last column show FP utilization based on the `FP` θ column. If the stars extend to the maximum length (shown by the hyphens to the right of `FP Utilization` in the column heading) then FP computation 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`, `hw03` collects hardware utilization data related to load and store instructions. A sample is shown below.

```
Kernel mxv_sh_ochunk_32(32,32):
wp ac   t/s I/op Ld eff SM eff  L2r  L2w    FP% GB/s FP  FP Utilization----------
-----------------
 4  4   1883  1.3 100.0%  40.0%  71.3 142.5  13.0%  143  570 *****
 8  8   1464  1.3 100.0%  40.0%  91.7 183.4  18.4%  183  733 ******
12 12   1459  1.3 100.0%  40.0%  92.0 183.9  19.1%  184  736 ******
```

The `Ld eff` column indicates the average percentage of each load request that is used. The 100% value shown above reflects the efficient use of requests made by the `ochunk` kernels.

The `L2r`θ column shows the throughput from the L2 caches to the MPs due to load instructions in GB/s. The `L2w`θ column shows the throughput from the MPs to the L2 caches due to store instructions, also in units of GB/s. These two throughput values are based on request sizes, not on how much of those requests are actually needed.

The `SM eff` value is the number of shared-memory access instructions divided by the number of executions of these instructions. A value of 100% indicates no bank conflicts. A value of 50% indicates an average of 2 distinct accesses to each bank. The ideal value for access to 32-bit data is 100%, the ideal for 64-bit data is 50%, and for 128-bit data is 25%. The compiler will sometimes

group 2 or 4 accesses to 32-bit items into one access to a 64-bit or 128-bit item. In such cases the lower `SM eff` does not reflect unrealized potential because executing, say, two instructions with 100% efficiency will use as much issue bandwidth as one access with 50% efficiency. That is the case with the `ochunk` code. The efficiency of 40% reflects a mixture of 32-bit and larger accesses.

The value in the `FP\%` column is the FP utilization reported by the NVIDIA CUPTI library. The value in the `FP` θ column is computed using measured time and assuming Sn^2 FP operations.

Problem 1: Summary: make `mxv_sh_ochunk_sol_mn` fast. It should work correctly for values of $8 \leq n \leq 64$ that are powers of 2. Please **do not** use separate code for each value of n. That is, **don't do something like this**:

```
extern "C" __global__ void mxv_sh_ochunk_sol_mn() {
  #define C(n) case n: mxv_sh_ochunk<n>(); break;
  switch( d_app.n ){ // DON'T DO THIS.
    C(8); C(16); C(32); C(64);
    default: return;
  }
  #undef C
}
```

Keep in mind that for $N \geq 32$ the code should be FP bound, so don't waste time on reducing off-chip memory throughput.

Currently, `mxv_sh_ochunk_sol_mn` is similar to `mxv_sh_ochunk` except that the matrix size is read from `d_app.n` rather than a template argument `eN` (or from a `#define` macro, `N`, as in the `vtx-xform-size` examples from class). The significance of this is that to the compiler `d_app.n` is a variable that can take on any value, whereas the compiler has the exact value when n is provided as a template argument or a macro.

As can be seen by running the code, the consequences are extreme. For a 16×16 matrix the execution time is over ten times slower without compile-time constants on a K20, almost 20 times slower on a GP100. This loss of performance is due to the way in which `mxv_sh_ochunk_sol_mn` is written.

There are several factors resulting in the loss of performance. These include:

The use of local memory rather than registers for local address space accesses. This occurs with arrays `vin` and `vout`. Notice the use of `STL` in the code below:

```
for ( auto& v: vout ) v = 0;  // CUDA C code.
 /*00f0*/                   MOV R4, RZ;
 /*00f8*/                   MOV R5, RZ;
 /*0108*/                   MOV R6, RZ;
 /*0110*/                   MOV R7, RZ;
 /*0120*/                   STL.128 [R15], R4;  // Local store.
```

Frequent access to constant memory in which different threads in a warp access different constant memory locations. In the sample below this occurs in accesses to `d_app.matrix[r][c+cc]` because of `r`:

```
const int thd_r_offset = threadIdx.x % CS;
const int r = rr * CS + thd_r_offset;
for ( int cc=0; cc<CS; cc++ )
  if ( c+cc < n ) // If needed when n not a multiple of CS.
    vout[rr] += d_app.matrix[r][c+cc] * vin[cc];
```

```
/*02f0*/              @!P2 LDC R27, c[0x3][R25+0x14];
/*02f8*/              @!P2 LDL R31, [R26];
/*0308*/                   ISETP.GE.AND P1, PT, R28, c[0x3][0x1040], PT;
/*0310*/              @!P2 FFMA R27, R4, R27, R31;
/*0318*/              @!P2 STL [R26], R27;
```

A high instruction overhead. That is, `mxv_sh_ochunk_16` executes 2.0 instructions per FMA (including the FMA), whereas `mxv_sh_ochunk_mn(16,16)` executes 6.8 instructions per FMA (see the values in the `I/op` column). The reason the instruction overhead of `mxv_sh_ochunk_16` is low is because it pre-loads matrix elements into registers and because its loops are completely unrolled and so there is no need to check the number of iterations, compute new addresses for each vector component, etc.

In your solution try the following approaches to fix the problem:

Avoid the use of local memory for local address-space accesses. For example, with `vin`, make sure that all indices to arrays are compile-time constants (after loop unrolling). Also, be sure to write every element of such arrays, even if some may not be written.

Whether any local memory is used can be found under the Kernel Resource Usage report printed at the start of each run. Examination of the `hw03.sass` code may help identify what local memory is used for, look for `LDL` and `STL` instructions.

Copy `d_app.matrix` *into shared memory.* Also avoid bank conflicts when reading shared memory. At times the compiler will group multiple 32-bit accesses into a single 64- or 128-bit access, and such an access will result in bank conflicts. For example,

```
/*0330*/                   LDS.U.128 R8, [R48];
```

Such bank conflicts are not bad because re-executing a 128-bit access four times uses no more instruction bandwidth than avoiding bank conflicts with four 32-bit shared memory accesses.

Read one element of the matrix from shared memory and use it on several input vectors. (This requires some thought.) That is, the way the code works now is a group of 8 (`CS`) threads computes `vout[h*n] = matrix * vin[h*n]`, then `vout[h*n+S] = matrix * vin[h*n+S]`, then `vout[h*n+2S] = matrix * vin[h*n+2S]`, ... Each matrix element might be read from shared memory (or constant memory) once for each of the computations. Instead, one might operate on element `h`, `h+S`, and `h+2S` at the same time, so each matrix element could be used 3 times (3 is just an example, it's not necessarily optimal).

A second benefit is that for access to `vin[h*n]`, `vin[h*n+S]`, and `vin[h*n+2S]`, only the address for `vin[h*n]` will need to be computed. The address for the other two are a constant distance away (`S` and `2S`) which can be offsets in the load instructions, so long as `S` is a compile time constant.

(*a*) Run `hw03-cuda-debug` under cuda-gdb. For instructions on how to use gdb see Programming Homework Work Flow on the course procedures page. Be sure to use `cuda-gdb` rather than `gdb` when debugging kernel code. Set a breakpoint on the line in `mxv_sh_ochunk_mn` that writes `vout[rr]` to global memory. Print out the value of `vout[rr]`. Do not attempt the parts below without first doing this.

(*b*) Make `mxv_sh_ochunk_sol_mn` run quickly using the approaches described above, and others that you can think of. Do not change the layout of `vin` and `vout` and do not write specialized code for each size.

(*c*) An important quality indicator is the number of instructions per FMA, `I/op` in the table.

Show an example of where you reduced instruction count. Explain what you did at the CUDA C source level and show SASS code excerpts from the unmodified `mxv_sh_ochunk_mn` kernel and your solution kernel illustrating the reduction. The SASS code should be relevant to what you did, don't show the entire routine.

(*d*) Show an example of instruction use that is higher than you think it could be. An example might be computation of addresses. Show the SASS code and discuss any untried approaches to reducing it further.