LSU EE 7722 Homework 2 Solution Due: 16 March 2018

Basic Setup

Follow the instructions for class account setup found on `http://www.ece.lsu.edu/gp/proc.html`. This assignment does not rely upon code in a `hw` directory, instead use the code in file `.../cuda/intro-vtx-transform/vtx-xform-size.cu`.

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 `.../cuda/intro-vtx-transform` 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 `.../cuda/intro-vtx-transform` 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 `vtx-xform-size`, and one with the suffix `-debug`, such as `vtx-xform-size-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, but that should not be a problem for this assignment.

Running make on a clean directory will produce a large amount of output, including warnings. The make program and input `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 in the `intro-vtx-xform` directory prints a diagnostic message about the presence of a Phi. When run a second time that's the only output.)

Quickly check whether the build is successful with the command `./vtx-xform-size`. It should produce output ending with a line starting `mxv_sh_ochunk`.

The makefile will compile code for a GPU on the system it was run, favoring the GPU that's not connected to a display. For this assignment multiple types of GPU need to be used. 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 `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 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}. This is the premier usage of this term. 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 $4(N + M)\,\mathrm{B}$ 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 $2NM$ 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 SNM, 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 $\frac{MN+M+N}{MN} = 1 + \frac{1}{M} + \frac{1}{N}$ or 1.125 for $M = N = 8$.

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_o_per_thd:
wp ac   t/µs FP θ  GB/s Bandwidth Util-------------------------------------------
 4  4  24269   11    6 ***
 8  8  13178   20   10 ******
12 12   8574   31   16 **********
16 16   6741   40   20 ************
20 20   6059   44   22 **************
24 24   5198   52   26 ****************
28 28   4935   54   27 *****************
32 32   4439   60   30 *******************
```

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 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_o_per_thd:
wp ac   t/µs FP θ Ld eff  L2rθ  L2wθ  DRrθ  DRwθ GB/s Bandwidth Util------------
 4  4  24520   11  12.5%  23.7   2.7   4.6   2.7    5 *
 8  8  12458   22  12.5%  45.0   5.4   7.3   5.4   11 ***
12 12   8820   30  12.5%  62.9   7.6   9.6   7.6   15 ****
16 16   8167   33  12.5%  67.4   8.2   9.8   8.2   16 ****
20 20   6189   43  12.5%  88.7  10.8  12.8  10.8   22 ******
24 24   6336   42  12.5%  86.4  10.6  12.3  10.6   21 ******
28 28   5065   53  12.5% 107.8  13.2  15.1  13.2   26 *******
32 32   5364   50  12.5% 101.8  12.5  14.1  12.5   25 *******
```

The `Ld eff` column indicates the average percentage of each load request that is used. The 12.5% value shown above is what one would expect for scattered accesses to 4-byte values. 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. The `DRr`θ and `DRw`θ columns 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: In class we noted that kernels `mxv_i_lbuf` and `mxv_o_lbuf` suffer from request under-utilization but noted that their performance was actually better than what one would expect if only $\frac{1}{8}$ of off-chip bandwidth carried useful data. We attributed this better-than-expected performance to the the level 2 (L2) cache.

Indicate whether data from runs using the commands described above provides evidence that the L2 cache is helping. Do so for a CC 3.5 device (there are two in the lab) and a CC 6.X device. Use the event counter data (the data provided when the `p` option is present) to justify your answer. In your solution indicate the type of GPU, show the performance data, and explain what the L2 cache is doing based on the data.

Based on the data indicate what might be limiting performance.

Data collected from a K20c (CC 3.5) and a GTX 1080 (CC 6.1) appear below. For the K20c the peak throughput is $41\,\mathrm{GB/s}$ which is $\frac{41}{208} = .197$ peak, which is higher than what would be possible with $\frac{1}{8} = .125$ request utilization. However, the event counters show that the combined read and write throughput between the L2 cache and DRAM is about $122\,\mathrm{GB/s}$ which is feasible and is what we would expect if $\frac{1}{8}$ of a request is begin used. That is, a total of $2^{20} \times 16$ elements are read, each using an entire $32\,\mathrm{B}$ request, for a total of $2^{20} \times 16 \times 32\,\mathrm{B} \approx 536.9\,\mathrm{MB}$. That is transferred over $3259\,\mu\mathrm{s}$ for a throughput of $536.9\,\mathrm{MB}/3259\,\mu\mathrm{s} = 164.7\,\mathrm{GB/s}$ which matches the L2 read throughput. The read throughput between the SMs (MPs) and the L2 cache is $164.8\,\mathrm{GB/s}$, compared to $20.6\,\mathrm{GB/s}$ from L2 to DRAM, a factor of $\frac{164.8}{20.6} = 8$ difference, which is exactly what one would expect if a 32-byte chunk from DRAM were retained in the L2 cache long enough for all of it to be written. In contrast the L2 cache is doing less to reduce write throughput between the L2 cache and DRAM. The results for the 6.1 device are similar.

The performance counter data show that off-chip (L2 to DRAM) bandwidth is not being saturated. The K20c can execute 33.9 SP (single-precision) instructions per single precision value transferred. For a 16×16 matrix there are 16 FMAs per 2 elements (one from vin and one from vout), or 8 FMA per element. The number of instructions per FMA is 1.2. That would work out to $16.2 = 16 + 3.2$ instructions. Rounding 3.2 up to 4, and assuming $\frac{1}{3}$ the throughput for such instructions makes that the equivalent of $16 + 4 \times 3 = 28$ instructions. Even so, that's $28/2 = 14$ instructions per element which is below FP saturation.

Another possibility is that there is a limit on the data bandwidth between SMs and the L2 cache.

```
-- Data Collected from a K20c, CC 3.5, Data BW 208 GB/s
Matrix size: 16 x 16.  Vectors: 1048576.   13 blocks of 1024 thds.
Launching with 13 blocks of up to 1024 threads.
Kernel mxv_i_lbuf:
                                 -L2 Cache-- ---DRAM----
wp  ac    t/μs I/op Ld eff  Rd θ   Wr θ   Rd θ   Wr θ FP θ GB/s Data BW Util---------
 4   4    6968  1.2  12.5%  77.1   77.1    9.6   61.8   39   19 *
 8   8    4332  1.2  12.5% 123.9  123.9   15.5   82.0   62   31 ***
12  12    3494  1.2  12.5% 153.7  153.7   19.2   98.0   77   38 ***
16  16    3326  1.2  12.5% 161.4  161.4   20.2  104.5   81   40 ****
20  20    3330  1.2  12.5% 161.2  161.2   20.2  103.4   81   40 ****
24  24    3291  1.2  12.5% 163.1  163.1   20.4  107.3   82   41 ****
28  28    3218  1.2  12.5% 166.8  166.8   20.9  111.3   83   42 ****
32  32    3259  1.2  12.5% 164.8  164.7   20.6  111.5   82   41 ****
Kernel mxv_o_lbuf:
                                 -L2 Cache-- ---DRAM----
wp  ac    t/μs I/op Ld eff  Rd θ   Wr θ   Rd θ   Wr θ FP θ GB/s Data BW Util---------
 4   4    6879  1.2  12.5%  78.1   78.1    9.8   56.6   39   20 *
 8  8   4366  1.2  12.5% 123.0 123.0  15.4  81.4   61   31 ***
12 12   3504  1.2  12.5% 153.2 153.2  19.2  96.1   77   38 ***
16 16   3329  1.2  12.5% 161.3 161.3  20.2 106.3   81   40 ****
20 20   3263  1.2  12.5% 164.5 164.5  20.6 115.0   82   41 ****
24 24   3300  1.2  12.5% 162.7 162.7  20.3 113.6   81   41 ****
28 28   3232  1.2  12.5% 166.1 166.1  20.8 116.3   83   42 ****
32 32   3365  1.2  12.5% 159.6 159.6  20.0 111.7   80   40 ****
```

```
-- Data Collected from a GTX 1080, CC 6.1, Data BW 320 GB/s
Kernel mxv_i_lbuf:
                         -L2 Cache-- ---DRAM----
wp ac   t/µs I/op Ld eff  Rd θ  Wr θ  Rd θ  Wr θ FP θ GB/s Data BW Util---------
 4  4   1493  1.2  12.5% 359.6 359.6  45.0  45.1  180   90 *****
 8  8   1446  1.2  12.5% 371.4 371.4  46.4  46.5  186   93 ******
12 12   1419  1.2  12.5% 378.4 378.3  47.3  47.6  189   95 ******
16 16   1404  1.2  12.5% 382.4 382.4  47.8  48.9  191   96 ******
20 20   1386  1.2  12.5% 387.4 387.4  48.4  52.0  194   97 ******
24 24   1367  1.2  12.5% 392.7 392.7  49.1  61.9  196   98 ******
28 28   1387  1.2  12.5% 387.2 387.1  48.6  61.9  194   97 ******
32 32   1395  1.2  12.5% 384.9 384.8  48.4  70.4  192   96 ******
Kernel mxv_o_lbuf:
                         -L2 Cache-- ---DRAM----
wp ac   t/µs I/op Ld eff  Rd θ  Wr θ  Rd θ  Wr θ FP θ GB/s Data BW Util---------
 4  4   1475  1.2  12.5% 363.9 363.9  45.5  45.6  182   91 *****
 8  8   1437  1.2  12.5% 373.5 373.5  46.7  46.8  187   93 ******
12 12   1446  1.2  12.5% 371.3 371.3  46.4  46.7  186   93 ******
16 16   1401  1.2  12.5% 383.2 383.2  47.9  49.2  192   96 ******
20 20   1397  1.2  12.5% 384.4 384.3  48.0  52.8  192   96 ******
24 24   1377  1.2  12.5% 390.0 390.0  48.8  60.9  195   97 ******
28 28   1385  1.2  12.5% 387.6 387.6  48.6  65.0  194   97 ******
32 32   1386  1.2  12.5% 387.5 387.5  48.7  76.1  194   97 ******

Kernel mxv_o_per_thd:
                         -L2 Cache-- ---DRAM----
wp ac   t/µs I/op Ld eff  Rd θ  Wr θ  Rd θ  Wr θ FP θ GB/s Data BW Util---------
 4  4   4671  3.4  12.5% 114.9  14.4  14.4  14.4   57   29 *
 8  8   2690  3.4  12.5% 199.6  24.9  24.9  25.0  100   50 ***
12 12   1970  3.4  12.5% 272.5  34.1  34.1  34.1  136   68 ****
16 16   1597  3.4  12.5% 336.2  42.0  42.0  42.0  168   84 *****
20 20   1409  3.4  12.5% 381.0  47.6  47.6  47.7  190   95 ******
24 24   1304  3.4  12.5% 411.6  51.4  51.4  51.5  206  103 ******
28 28   1221  3.4  12.5% 439.7  55.0  55.0  55.0  220  110 *******
32 32   1166  3.4  12.5% 460.5  57.6  57.6  57.6  230  115 *******
```

Problem 2: The number of threads needed in a kernel launch can be determined using a technique called *GPU interval analysis*. Interval analysis starts with an *interval* which is the time during the execution of one iteration of a well-chosen loop. The loop is well chosen if the execution time of the program is some multiple of the time for an execution of the loop.

For the `mxv_o_lbuf` kernel the interval will be one iteration of the `h` loop.

The first step in GPU interval analysis is to compute the latency for a thread executing one iteration of the loop while assuming no resource contention due to other threads. Compute this latency so that if L_ι is the latency of one iteration, then the time needed for x iterations of the loop by a thread is xL_ι.

For this assignment estimate the interval latency by accounting for *instruction issue rate* and *exposed* operation latency. The instruction issue rate will be used to compute the earliest time that the instruction can be issued based on instruction throughput and assuming one warp.

Consider Maxwell- and Pascal-generation devices. Most have four warp schedulers per MP, 128 single-precision functional units per MP, and 64 load/store (LDST) units per MP, or 32 SP FP units per scheduler and 16 LDST units per scheduler. Assume that at most one instruction per cycle can be issued. Then for a thread SP FP instructions, such as `FFMA`, will be issued at a rate of one per cycle. (Remember that we are assuming only one warp of threads for purposes of issue.) For memory instructions (loads and stores) instructions will be issued every two cycles. See the code excerpt and pipeline diagram below, where `FFMA` instructions advance at one per cycle but `LDG` advance at half that rate.

Let L_F denote the latency of a floating-point operation, and L_M denote the minimum latency of a load instruction that misses the L2 cache. For Maxwell- and Pascal-generation devices $L_\mathrm{F} = 6\,\mathrm{cyc}$ and a typical value is $L_\mathrm{M} = 400\,\mathrm{cyc}$. Note that L_F is a fixed duration, meaning that once an instruction is dispatched to the functional unit a dependent instruction can be dispatched L_F cycles later and expect to find its source value ready. In contrast L_M holds for a load to uncached data on a lightly loaded system. The actual latency would be higher if there were contention for, say, off-chip bandwidth, or lower if the load hit the L2 cache. For interval analysis contention is usually assumed to be zero, but whether an accesses hits the L2 cache is part of the analysis.

In the code fragment below `F0` depends on `L0`, and so stalls for 394 cycles. Instruction `F8` depends on `F0`, but since the issue time is larger than latency there is no stall.

```
L0:    LDG.E R16, [R2];
L1:    LDG.E R12, [R14];
L2:    LDG.E R11, [R14+0x4];
...
F0:    FFMA R13, R16, c[0x3][0x4], RZ; // Issue: t = 0, avail t = 6
F1:    FFMA R17, R16, c[0x3][0x24], RZ;// Issue: t = 1, avail t = 7
F2:    FFMA R18, R16, c[0x3][0x44], RZ;// ...
F3:    FFMA R19, R16, c[0x3][0x64], RZ;
F4:    FFMA R21, R16, c[0x3][0x84], RZ;
F5:    FFMA R22, R16, c[0x3][0xa4], RZ;
F6:    FFMA R14, R16, c[0x3][0xc4], RZ;
F7:    FFMA R15, R16, c[0x3][0xe4], RZ;
F8:    FFMA R13, R12, c[0x3][0x8], R13;// Issue:
...
S0:    STG.E [R4], R2;
S1:    STG.E [R4+0x4], R14;
S2:    STG.E [R4+0x8], R13;
...

Time 0   1   2   3   4   5   6   ... 400 401 402 403 404 405 406 407 408
wp0: [L0   ] [L1   ] [L2   ]         F0  F1  F2  F3  F4  F5  F6  F7  F8
```

Consider a modified version of the `mxv_o_lbuf` kernel in which the entire input vector is read before performing arithmetic:

```
  for ( int h=start; h<stop; h += inc )
    {
      Elt_Type vout[M]; for ( int r=0; r<M; r++ ) vout[r] = 0;
      Elt_Type vin[N];  for ( int c=0; c<N; c++ ) vin[c] = d_app.d_in[ h * N + c ];

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

      for ( int r=0; r<M; r++ ) d_app.d_out[ h * M + r ] = vout[ r ];
    }
```

Accounting only for the load, multiply/add, and store instructions we get $L_\iota(K_0) = L_\mathrm{M} + MN + M$ if $N \geq L_\mathrm{F}$ and $L_\mathrm{M} \geq N/2$, where K_0 is a shorthand name for the modified kernel and its assumed code. For values $M = N = 16$ and the latencies above we get $L_\iota(K_0) = 400 + 256 + 16 = 672\,\mathrm{cyc}$.

Once the latency for an interval is computed, it is possible to bound the number of threads that can be executed in parallel by using resource constraints. For example, let d_ι denote the amount of off-chip data read or written by a thread during one iteration and let Θ_M indicate the off-chip bandwidth. Let P indicate the number of threads in a launch. The amount of data read during the interval is Pd_ι, the data throughput would be $\frac{Pd_\iota}{L_\iota}$, solving $\frac{Pd_\iota}{L_\iota} = \Theta_M$ yields $P = \Theta_M \frac{L_\iota}{d_\iota}$, the number of threads needed to saturate off-chip bandwidth.

For our matrix/vector kernels, $d_\iota = 4(M+N)\,\mathrm{B}$. Then $P = \Theta_M \frac{(L_\mathrm{M}+MN+M)\,\mathrm{cyc}}{4(M+N)\,\mathrm{B}}$. For $N = M$ (square matrices) we have $P = \Theta_M \frac{(L_\mathrm{M}+N^2+N)\,\mathrm{cyc}}{8N\,\mathrm{B}} = \frac{\Theta_M}{8}(\frac{L_\mathrm{M}}{N} + N + 1)\frac{\mathrm{cyc}}{\mathrm{B}}$ for $N \geq L_\mathrm{F}$. Notice that this expression has a minimum at $N = \sqrt{L_\mathrm{M}}$, or 20 for our default global memory latency. As N

shrinks below 20 more and more threads are needed because each thread is loading less data but the iteration time, L_ι, can't fall below L_M. When N grows above 20 the computation time, N^2, grows quadratically while data grows linearly, so more threads are needed. Note that Θ_M is often given in GB/s while the latency is given in cycles. Cycles can be converted to seconds by dividing by the clock frequency, often denoted ϕ. In that case $L_\iota(K_0) = (L_\mathrm{M}+MN+M)\,\mathrm{cyc} = \frac{1}{\phi}(L_\mathrm{M}+MN+M)\,\mathrm{s}$.

Another limiter is instruction throughput. For that one must compute the time needed during an interval to issue the instructions. Let Θ_IF indicate the bandwidth of FP instructions and Θ_IM indicate the bandwidth of load and store instructions on a multiprocessor. Let n_IF and n_IM indicate the number of instructions of the respective type executed by a thread in the interval. Then the time for Q threads to issue for one interval is $Q(\frac{n_\mathrm{IF}}{\Theta_\mathrm{IF}} + \frac{n_\mathrm{IM}}{\Theta_\mathrm{IM}})$ and based on this instruction issue limit $Q = L_\iota / \left(\frac{n_\mathrm{IF}}{\Theta_\mathrm{IF}} + \frac{n_\mathrm{IM}}{\Theta_\mathrm{IM}}\right)$. Note that Q is the number of active threads on a multiprocessor, which is some multiple of the block size.

For K_0 we have $n_\mathrm{IF} = MN$ and $n_\mathrm{IM} = M + N$. Substituting these values and L_ι gives

$$\begin{aligned}
Q(K_0) =& L_\iota / \left(\frac{n_\mathrm{IF}}{\Theta_\mathrm{IF}} + \frac{n_\mathrm{IM}}{\Theta_\mathrm{IM}}\right) \\
=& (L_\mathrm{M} + MN + M) \,/ \left(\frac{MN}{\Theta_\mathrm{IF}} + \frac{M+N}{\Theta_\mathrm{IM}}\right) \\
=& \left(L_\mathrm{M} + N^2 + N\right) / \left(\frac{N^2}{\Theta_\mathrm{IF}} + \frac{2N}{\Theta_\mathrm{IM}}\right) \\
=& (L_\mathrm{M}/N + N + 1) \,/ \left(\frac{N}{\Theta_\mathrm{IF}} + \frac{2}{\Theta_\mathrm{IM}}\right) \\
=& \left(L_\mathrm{M}/N^2 + 1 + 1/N\right) / \left(\frac{1}{\Theta_\mathrm{IF}} + \frac{2}{N\Theta_\mathrm{IM}}\right)
\end{aligned}$$

For large N the interval time is dominated by the N^2 instructions so the number of threads is based on FP instruction bandwidth Θ_IF. When N is smaller load latency determines the thread count.

(a) Does the interval analysis above explain the behavior of the `mxv_o_lbuf` kernel obtained on a CC 6.1 device? If not, provide a possible explanation and suggest a way of modifying the analysis.

The calculations below show that for a GTX 1080 the iteration latency would be $388.4\,\mathrm{ns}$ so to saturate the $320.3\,\mathrm{GB/s}$ bandwidth would require 971.9 threads or just 48.6 threads per MP. To saturate instruction issue assuming only N loads, N stores, and N^2 multiply/adds would require 268.8 threads per MP.

The performance counters show 1.2 instructions per FMA, or a total of $256 \times 0.2 = 51.2$ non-FMA instructions. We've already accounted for 32 loads and stores, so there are just 19 extra instructions. Assume that these add 19 cycles to latency (which means that they are before the first loads or after the first FMA). Including those extra instructions only slightly changes P and Q.

The execution on the GTX 1080 reaches a peak at 4 wp / MP or 128 thds per MP, which is consistent with this.

This problem was for a 6.1 device, but let's consider a 3.5 device anyway. A similar analysis for the K20 finds 118 threads per MP are needed to saturate off-chip bandwidth. That is inconsistent with observed behavior where a configuration with 4 wps (128 threads) per MP yielded just 1/2 the performance of larger blocks. Inspection of the SASS code shows that dependent FMA instructions are closer together than L_F and so the latency of these instructions is exposed.

```
% Solution Calculations
% For a GTX 1080.  MP: 20.  Data 320.3 GB/s
%  Clock: 1.73 GHz
%  Theta_IF = 128
%  Theta_IM =  64

%   n_IF_1 = 16 * 16 = 256
%   n_IM_1 = 16 + 16 = 32
%   Extra: 256 * .2 = 51.  Assume same thpt as IF

%
%  N = 16
% Latency/iter: 400 + N^2 + N = 672 cyc = 388.4 ns
% Latency/iter (correct store II): 400 + N^2 + 2N = 688 cyc = 397.7 ns
% Data/iter: 4(16+16) = 128 B
% Theta_M = 320.3 GB/s
% P = 320.3 GB/s 388.4 ns / 128 B = 971.9
%   => 971.9 / 20 = 48.6 thds / MP
% Issue Time Ideal:
%  256/128 + 32 / 64 = 2.5 cyc
%  Q = 672 / 2.5 = 268.8  thds per SM

% Accounting for 1.2 I/op.
%  Latency/iter: 400 + N^2 + N + 19 = 691 cyc = 399.4 ns
%  P = 320.3 GB/s 399.4 ns / 128 = 999 = 50 thds / MP
% Issue Time based on 1.2 I/op (which is 51 insn total or 51-32= 19 non l/s)
%  (256+19)/128 + 32/64 = 2.648 cyc
%  Q = 691 / 2.648 = 261 thds per SM

% For a GTX K20c.  MP: 13.  Data 208.0 GB/s
%  Clock: .71 GHz
%  Theta_IF = 128
%  Theta_IM =  64

%   n_IF_1 = 16 * 16 = 256
%   n_IM_1 = 16 + 16 = 32
%   Extra: 256 * .2 = 51.  Assume same thpt as IF

%
%  N = 16
% Latency/iter: 400 + N^2 + N = 672 cyc = 946.5 ns
% Data/iter: 4(16+16) = 128 B
% Theta_M = 208.0 GB/s
% P = 208.0 GB/s 946.5 ns / 128 B = 1538
%   => 1538 / 13 = 118
% Issue Time Ideal:
%  256/128 + 32 / 64 = 2.5 cyc
%  Q = 672 / 2.5 = 268.8  thds per SM
% Issue Time based on 1.2 I/op
%  (256+51)/128 + 32/64 = 2.898 cyc
%  Q = 672 / 2.898 = 231.8 thds per SM
```

(*b*) Using interval analysis as described above, determine the minimum number of threads needed to saturate off-chip bandwidth and instruction issue for kernel `mxv_o_lbuf` assuming that the order of machine (SASS) instructions follows the order of instructions in the CUDA code. That is, don't assume the entire input vector will be loaded before starting on the multiply-adds. In your analysis assume that the level 2 cache is effective. Use $L_\mathrm{M} = 400\,\mathrm{cyc}$ for the latency of a load that misses and $L_2 = 200\,\mathrm{cyc}$ for the latency of a load that hits the level 2 cache. Also use $L_\mathrm{F} = 6\,\mathrm{cyc}$ for instruction latency of non-memory instructions.

Show the number of threads needed as expressions, and compute actual values for one of the Pascal CC 6.1 GPUs.

One advantage of following the order of instructions in the CUDA code is that $N-1$ fewer registers are needed. Is this worth it?

The CUDA C code loads an element and uses it in N FMA instructions, then loads the next element, etc. Using an element size of 4 bytes and a request or line size of 32 bytes, we would expect that $\frac{1}{8}$ of the loads would miss the L2 cache and $\frac{7}{8}$ would hit. The time for an iteration accounting for the loads and FMAs would be $L_\mathrm{M} + N + L_2 + N + \cdots L_\mathrm{M} + \cdots = \frac{1}{8}NL_\mathrm{M} + \frac{7}{8}NL_2 + N^2$. Assuming that the stores issue at one per cycle the total iteration time is $L_\iota(K_1) = \frac{1}{8}NL_\mathrm{M} + \frac{7}{8}NL_2 + N^2 + N$. The expressions for the amount of data and issue time are the same as those used in part a.. So:

$$\begin{aligned}
P &= \Theta_M \frac{\left(\frac{1}{8}NL_\mathrm{M} + \frac{7}{8}NL_2 + N^2 + N\right)\,\mathrm{cyc}}{8N\,\mathrm{B}}\\
&= \Theta_M \frac{\left(\frac{1}{8}L_\mathrm{M} + \frac{7}{8}L_2 + N + 1\right)\,\mathrm{cyc}}{8\,\mathrm{B}}\\
&= 320.8\,\mathrm{GB/s}\frac{\left(\frac{1}{8}400 + \frac{7}{8}200 + 16 + 1\right)\,\mathrm{cyc}}{8\,\mathrm{B}}\\
&= 5609.4
\end{aligned}$$

This works out to 280 threads or 8 warps per MP.

That's substantially more but still easily attainable. That means the compiler can conserve per-thread registers. That doesn't save per-block registers because more threads per block are needed.

(*c*) Perform the interval analysis on kernel `mxv_o_per_thd`. When working out the analysis don't forget that each thread is computing only one component of the output vector and that neighbors are using the same input vector. Take that into account when computing d_ι.

Show the number of threads needed as expressions, and compute actual values for one of the Pascal CC 6.1 GPUs.

Consider an iteration of the h loop:

```
Elt_Type vout = 0;
for ( int c=0; c<N; c++ ) vout += d_app.matrix[r][c] * d_app.d_in[ h * N + c ];
d_app.d_out[ h * M + r ] = vout;
```

Assuming that the `c` loop is unrolled, there will be N loads followed by N FMA instructions. Notice that the N FMA instructions are dependent on each other (since they are adding to `vout`), so the iteration latency includes the load latency plus N FMA latencies, and one issue time for the store: $L_\iota = L_\mathrm{M} + NL_\mathrm{F} + 1$.

Each thread loads $4N\,\mathrm{B}$ of data, but because M threads are accessing the same data item the contribution of one thread to the amount of data accessed is $d_\iota = 4(\frac{N}{M} + 1)\,\mathrm{B}$, or $8\,\mathrm{B}$ when $N = M$and assuming that the L2 cache is effective.

The number of threads to saturate latency is:

$$\begin{aligned} P &= \Theta_M \frac{L_\mathrm{M} + NL_\mathrm{F} + 1\,\mathrm{cyc}}{8\,\mathrm{B}} \\ &= 320.8\,\mathrm{GB/s} \frac{(400 + 16 \times 6 + 1)\ \mathrm{cyc}}{8\,\mathrm{B}} \\ &= 11502 \end{aligned}$$

This works out to 575 threads per MP or 18 warps per MP (remembering that a 1080 has 20 MPs).

The number of threads is consistent with the data from Problem 1 in which performance improvement for `mxv_o_per_thd` starts leveling off between 16 and 20 warps.

```
Iter Latency:
 L_M + N L_f + 1
 400 + 16 6 + 1 = 497 cyc = 287.3 ns
Data: 8 B
 P = 320.3 GB/s 287.3 ns / 8 B = 11502 = 575 / MP = 18 wp / MP
```