LSU EE 7700-2 **Homework 6** Solution **Due: 27 March 2013**

Follow the instructions on the class procedures page, `http://www.ece.lsu.edu/gp/proc.html` *for account setup and homework, substituting* `hw6` *for* `hw1` *where appropriate. Also, the file to edit is* `hw4.cu`*, not* `hw6.cc`*. The assignment code is the same as the vertex transformation code used in class.*

Problem 0: Read the background provided in this problem, and compile and run the Homework 6 assignment code; report any problems as soon as possible. The answer to this problem is "It ran fine.", the next problem actually asks questions.

The program runs four GPU versions of our vertex transformation code, `kernel_many_threads`, `kernel_few_threads_d2`, `kernel_few_threads_d4`, and `kernel_few_threads_d8`, set to operate on 2-element vertices.

The code for his assignment is similar to that used in Homework 4. One important change is the forcing of kernels to use a single load instruction to load a vector, rather than one load for each element. (Using four loads to load a 4-element vector can result in re-fetching a line to the L1 cache due to conflict misses.)

Kernel `kernel_many_threads` is similar to the kernel is Homework 4. The other kernels are hand-unrolled versions of `kernel_many_thread`. For example, `kernel_few_threads_d4` contains four copies of the loop body. The unrolling was performed in such a way that all loads in an iteration would be performed before any calculation. The template `kernel_few_threads_d` contains the actual code, `kernel_few_threads_d4` instantiates the template for a degree of 4, etc.

There are two command line arguments, the first is the number of blocks to launch, the second is the size of the array to use, in MiB, (fractional amounts are fine).

The program will launch each kernel in up to 32 configurations, from a 1 warp-block to the maximum block size possible for the kernel. For each launch the kernel execution time and data transfer rate are shown. The data transfer rate is shown in GB/s, as a percentage of the maximum GPU to device memory bandwidth, and as a bar graph.

Note that the number of blocks must be appropriately chosen for some of the problems below.

Problem 1: The inspiration for this assignment is the choice between many-thread systems (like GPUs) and fewer-thread systems (like the so-called manycores). Kernel `kernel_many_threads` is written to rely on a large number of threads to hide latency. The other kernels rely on loop unrolling to hide at least some latency, and so fewer threads will be required, however those threads will use more registers because the compiler is putting greater distance between the instruction that writes a register and the first instruction that uses the register.

(*a*) Perform a set of runs (or just one run if that's all it takes) to determine which is the more efficient approach in terms of the amount of hardware needed. For each kernel indicate the number of threads needed to realize something close to the best performance and also indicate the **total** number of registers needed. (That is, the number of registers the multiprocessor needs in order to run that number of threads for that kernel.) Provide relevant information such as the type of GPU, and number of blocks chosen for each experiment, etc.

Do this for two goals: one to achieve the fastest computation (which should saturate off-chip data bandwidth) and one to achieve fastest performance on at least one multiprocessor. Note that for the first goal memory latency will not be the performance limiter and so it is not necessary to cover all memory latency to realize maximum performance.

This solution is for a Quadro 5000, a CC 2.0 device with a computation to communication ratio of 12 single-precision floating-point instructions per single-precision floating point element, *single*, of bandwidth. The vertex transform code for this assignment performs four multiply/adds per vertex, each vertex consumes two singles of bandwidth to load and

two to store. If every instruction were a MADD the ratio would be 1, but we need to consider loads, stores, and loop index arithmetic. Lets assume an ideal version of the code in which there is no loop index arithmetic, just a vector load and store. Then the number of instructions per vertex is 6, and so the ratio is 1.5. In real code there will typically be one set of loop index arithmetic instructions per iteration, and this will make the ratio higher (making it less likely to be bandwidth bound). If we assume 5 loop index instructions per iteration and an unrolling of degree d then the number of instructions per vertex is $4 + 2 + 5/d$.

Since one goal of the experiments is to determine which is the most efficient method of hiding latency, it's important to run configurations which are not bandwidth bound. The easiest way to do that is to reduce the number of blocks. On a Quadro 5000 there are 11 MP, if only one MP is used then the FP capability will be reduced by $\frac{1}{11}$, but the data bandwidth will not be effected. That changes the computation to communication ratio to the Quadro 5000 to $\frac{12}{11}$ which is lower than 1.5 and so even the most aggressive code, $d \to \infty$, will be compute bound.

Two sets of experiments were run. For the first, the number of blocks is set to one, for the second the number of blocks was chosen to match the number of multiprocessors, eleven.

The original code shows the performance versus the number of warps. For each kernel in each run the minimum number of warps yielding something close to maximum performance will be found. For example, in the d8 kernel 11 warps yielded close to maximum performance (see the table below), 20.7% of maximum bandwidth. (Ten warps yielded 19.9%, a little less, and 12 yielded 20.9%, only a tiny bit more, but small enough to ignore.)

Increasing the number of warps beyond this minimum does not significantly increase performance because some resource is saturated. Since our goal is to hide all latency we would like the saturated resource to be CUDA cores. That is, we want there to be at least two warps ready to execute at any time.

If our configuration is bandwidth limited (for example, running with 11 blocks) then load latency will increase beyond its nominal value (on the order of 400 cycles), the latency will include waiting time as well as the time needed to actually access the memory and for the trip to memory and back. No matter how many warps are present the system will eventually reach a point at which there are no warps ready for execution (because they are all waiting for memory).

The table below shows the minimum number of warps needed to saturate the system for one-block (execute-limited) and eleven-block (bandwidth-limited) runs of the unmodified assignment code.

	---Single-Block---				---Eleven-Blocks--		
Kernel	Regs	Min Warps	Pct BW	Tot Regs	Min Warps	Pct BW	
Many Thd	11	32	18.5	11264	22	81.6	7744
Degree 2	14	24	20.5	10752	17	82.1	7616
Degree 4	21	12	20.9	8064	16	80.3	10752
Degree 8	27	11	20.7	9504	13	79.6	11232

For the single block experiments it was seemingly not possible to fully cover latency with 32 warps using the many threads kernel. Successively fewer warps were needed with greater unrolling. Notice, however, that going from degree 4 to 8 only reduces the number of warps by 1. Looking at the number of registers needed, the degree 4 unrolling is best, using less than 72% of the registers needed by the many-threads code.

(*b*) Comment on whether these experiments allow one to conclude that the many-threads or fewer-threads approach is better. An answer might start "These experiments don't show the full benefit of the fewer-threads approach because the code in `kernel_few_threads_d` ..."

Problem 2: Perform the following hand analysis of kernels `kernel_many_threads` and `kernel_few_threads_d4`. Locate the assembler code (in the file ending with `sass`) and identify the loop body for each routine (for routine `kernel_few_threads_d4` do the unrolled loop, not the loop at the end).

Estimate the latency of the loop body under the assumption that memory instructions have a 400-cycle latency and all others have a 24-cycle latency.

Here are some miscellaneous facts: Instructions with a `.X` completer depend on the most recent instruction with a `.CC` completer. Sixty-four bit load instructions load a pair of registers, the named destination is only the first of two. For example, this load `LD.E.64 R2, [R8];` instruction loads both register `R2` and `R3`. The 128-bit load instructions load four registers.

To find the latency of the loop body (a single iteration) determine the latest time that an instruction will start. For convenience assume a device of CC 2.0, so that instruction $i + 1$ starts two cycles after instruction i when it is not dependent on prior instructions. A dependent instruction starts when its operands have been computed based on the start time and the latencies given above.

In the example below, instruction 0 starts at time 0 and its result is ready at time 24. Instruction 1 starts at 2 since it's not dependent on 0. However, 2 depends on 0 because of `R10` and so it must wait until 24 to start. Instruction 3 depends on instruction 2 through the carry bit (the `.CC` to `.X` dependence). Instruction 4 can start at cycle 50 since by then its operands will be available.

```
0: MOV32I R10, 0x8;           // Start at 0, result ready at 24
1: IMUL.HI R4, R0, 0x8;       // Start at 2, result ready at 26
2: IMAD R8.CC, R0, R10, c [0x2] [0x30]; // Start at 24 (dep on 1st insn via R10) rdy at 48
3: IADD.X R9, R4, c [0x2] [0x34];       // Start at 48 (dep on prev via CC) rdy at 72
4: IMAD R6.CC, R0, R10, c [0x2] [0x40]; // Start at 50, result ready at 74
...
```

Use the following method to help find dependencies. In Emacs put the cursor over the start of a destination register, perhaps the first `R10` in the example above. Then press C-s C-w (control-s followed by a control w). This should highlight all occurrences of R10. (The C-s starts a search, and C-w tells Emacs to search for other occurrences of the word under the cursor. A second C-s would move to the next occurrence.) Press C-g to exit the search.

(*a*) Show the latency of the loop body of each kernel, as described above.

The latency of **many_threads** is 484 cycles and the latency of **d4** is 578 cycles. The starting cycle for each instruction is shown below. If an instruction had to wait for a source operand the dependent instruction address appears after the start cycle.

```
many_threads:
/*0038*/  IMUL.HI R4, R0, 0x8;                      //   0
/*0040*/  IMAD R8.CC, R0, R10, c [0x2] [0x30];     //   2
/*0048*/  IADD.X R9, R4, c [0x2] [0x34];           //  26  dep 0x40
/*0050*/  IMAD R6.CC, R0, R10, c [0x2] [0x40];     //  28
/*0058*/  IADD R0, R0, c [0x2] [0x0];              //  30
/*0060*/  LD.E.64 R2, [R8];                        //  32
/*0068*/  IADD.X R7, R4, c [0x2] [0x44];           //  34
/*0070*/  ISETP.LT.AND P0, pt, R0, c [0x2] [0x14], pt; // 54 dep 0x58
/*0078*/  FFMA R4, R2, c [0x2] [0x4], RZ;          // 432 dep 0x60
/*0080*/  FFMA R2, R2, c [0x2] [0xc], RZ;          // 434
/*0088*/  FFMA R4, R3, c [0x2] [0x8], R4;          // 456 dep 0x78
/*0090*/  FFMA R5, R3, c [0x2] [0x10], R2;         // 458
/*0098*/  ST.E.64 [R6], R4;                        // 482 dep 0x90 via R5
/*00a0*/  @P0 BRA 0x38;                            // 484


Function : kernel_few_threads_d4
/*0088*/  IMAD R10, R16, R15, R14;                 //   0
/*0090*/  IADD R0, R0, R16;                        //   2
/*0098*/  IADD R15, R15, 0x1;                      //   4
/*00a0*/  IMUL.HI R11, R10, 0x8;                   //   8
```

```
/*00a8*/  IMAD R12.CC, R10, R18, c [0x2] [0x30];  //  24 dep 0x88
/*00b0*/  ISETP.LT.AND P0, pt, R0, R17, pt;       //  26 dep 0x90
/*00b8*/  IADD.X R13, R11, c [0x2] [0x34];        //  48 dep 0xa8
/*00c0*/  IMAD R10.CC, R10, R18, c [0x2] [0x40];  //  50
/*00c8*/  LD.E.64 R8, [R12];                      //  72 dep 0xb8 via R13
/*00d0*/  LD.E.64 R4, [R12+0x100];                //  74
/*00d8*/  LD.E.64 R2, [R12+0x200];                //  76
/*00e0*/  LD.E.64 R6, [R12+0x300];                //  78
/*00e8*/  IADD.X R11, R11, c [0x2] [0x44];        //  80
/*00f0*/  FFMA R12, R8, c [0x2] [0x4], RZ;        // 472 dep 0xc8
/*00f8*/  FFMA R13, R8, c [0x2] [0xc], RZ;        // 474
/*0100*/  FFMA R19, R4, c [0x2] [0x4], RZ;        // 476
/*0108*/  FFMA R8, R9, c [0x2] [0x8], R12;        // 496 dep 0xf0
/*0110*/  FFMA R9, R9, c [0x2] [0x10], R13;       // 498
/*0118*/  FFMA R20, R4, c [0x2] [0xc], RZ;        // 500
/*0120*/  FFMA R12, R2, c [0x2] [0x4], RZ;        // 502
/*0128*/  FFMA R4, R5, c [0x2] [0x8], R19;        // 504
/*0130*/  ST.E.64 [R10], R8;                      // 522 dep 0x110
/*0138*/  FFMA R13, R2, c [0x2] [0xc], RZ;        // 524
/*0140*/  FFMA R19, R6, c [0x2] [0x4], RZ;        // 526
/*0148*/  FFMA R6, R6, c [0x2] [0xc], RZ;         // 528
/*0150*/  FFMA R5, R5, c [0x2] [0x10], R20;       // 530
/*0158*/  FFMA R2, R3, c [0x2] [0x8], R12;        // 532
/*0160*/  FFMA R3, R3, c [0x2] [0x10], R13;       // 548 dep 0x138
/*0168*/  FFMA R8, R7, c [0x2] [0x8], R19;        // 550
/*0170*/  FFMA R9, R7, c [0x2] [0x10], R6;        // 552
/*0178*/  ST.E.64 [R10+0x100], R4;                // 554
/*0180*/  ST.E.64 [R10+0x200], R2;                // 572 dep 0x160
/*0188*/  ST.E.64 [R10+0x300], R8;                // 576 dep 0x170
/*0190*/  @P0 BRA 0x88;                           // 578
```

(*b*) Count the number of instructions in the loop bodies.

There are 14 and 34 instructions.

(*c*) Based on the answer to the last two parts, determine the minimum number of warps per multiprocessor needed to make full use of the CUDA cores on a device of compute capability 2.0. Assume unlimited data bandwidth (but global load latency is still 400 cycles). Note that this is a lower bound on the number of warps needed to completely hide latency.

The many-threads kernel has a latency of 484 cycles and consists of 14 instructions. A CC 2.0 SM has 32 CUDA cores, so over 484 cycles $484 \times 32 = 15488$ instructions or 484 warps could be executed (issued or started would be the more precise word). Since a loop body has 14 instructions it would take at least $\lceil 484/14 \rceil = 35$ warps to have enough instructions to keep all the CUDA cores busy. Note that this is a lower bound, meaning due to scheduling issues more warps would be needed. Also note that a CC 2.0 device can have up to 48 warps per MP, but the limit for a block is 32, meaning that to realize the 35-warp minimum one would need to have, say, two 18-warp blocks per multiprocessor.

For the d4 kernel $578/34 = 17$ warps would be needed. Fewer warps are needed because more loads are allowed to overlap.

(*d*) Using data from the device used in the previous problem, determine the minimum number of warps per multiprocessor needed to make full use of the memory bandwidth.

The many-threads kernel reads two singles (8 bytes) and writes two singles in an iteration, for a total of 16 bytes.

The device used for these experiments is a Quadro 5000 with a clock frequency of $1.03\,\mathrm{GHz}$ and bandwidth of

$120\,\mathrm{GB/s}$. A single thread of the many-threads will use $16\,\mathrm{B}\frac{1.03\,\mathrm{GHz}}{484} = 0.034\,\mathrm{GB/s}$ of bandwidth. To saturate the bandwidth would require $120/0.034 = 3524$ threads or 110 warps, or 10 warps per MP.

For the `d4` kernel a single iteration operates on four vertices, and so moves 64 bytes. The bandwidth consumed is $64\,\mathrm{B}\frac{1.03\,\mathrm{GHz}}{578} = 114\,\mathrm{GB/s}$. To saturate bandwidth one would need 1052 threads or 33 warps or just 3 warps per MP.

(*e*) Compare these answers to the experiments performed in the previous problem. Comment on how closely they agree.

The hand analysis determined that it would take at least 35 warps for many-threads to hide all latency, and that is consistent with experimental results.

The hand analysis of `d4` predicted that 17 warps would be needed, but experiments showed that performance topped out at 12 warps. One possible reason for the discrepancy is the assumption that all instructions use the 32 CUDA cores. In fact, the integer multiply and madd instructions, and the loads and stores only have 16 functional units available. There are 12 such instructions in `d4`. Another reason for the discrepancy is that some other resource is saturated, perhaps due to the order of accessed elements.

Raw data used for Problem 1

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 4799 MiB GLOBAL MEM
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: L2: 1280 kiB   MEM to L2: 208.0 GB/s  SP 1760.9 GFLOPS  OP/ELT 33.86
GPU 1: Quadro 5000 @ 1.03 GHz WITH 2559 MiB GLOBAL MEM
GPU 1: CC: 2.0  MP: 11  CC/MP:  32  TH/BL: 1024
GPU 1: SHARED: 49152 B  CONST: 65536 B  # REGS: 32768
GPU 1: L2: 640 kiB   MEM to L2: 120.0 GB/s  SP 361.2 GFLOPS  OP/ELT 12.04
Using GPU 1

CUDA Routine Resource Usage:
For kernel_many_threads:
       0 shared, 72 const, 0 loc, 11 regs; 1024 max threads per block.
For kernel_few_threads_d2:
       0 shared, 72 const, 0 loc, 14 regs; 1024 max threads per block.
For kernel_few_threads_d4:
       0 shared, 72 const, 0 loc, 21 regs; 1024 max threads per block.
For kernel_few_threads_d8:
       0 shared, 72 const, 0 loc, 27 regs; 1024 max threads per block.

Preparing for 1 blocks operating on 1048576 vectors of 2 elements.
Running kernel kernel_many_threads which uses 11 regs.
 Num      Time    Data
 Wps       s    GB/s    Pct
  1  21227.200    0.79   0.7%  *
  2  10782.112    1.56   1.3%  *
  3   7108.608    2.36   2.0%  *
  4   5282.432    3.18   2.6%  **
  5   4267.552    3.93   3.3%  **
  6   3650.496    4.60   3.8%  **
  7   3115.680    5.38   4.5%  ***
  8   2750.016    6.10   5.1%  ***
  9   2454.112    6.84   5.7%  ***
 10   2222.112    7.55   6.3%  ****
 11   2011.360    8.34   7.0%  ****
```

```
 12   1854.432    9.05   7.5%  ****
 13   1711.712    9.80   8.2%  *****
 14   1601.280   10.48   8.7%  *****
 15   1497.248   11.21   9.3%  *****
 16   1397.088   12.01  10.0%  *****
 17   1330.080   12.61  10.5%  ******
 18   1244.352   13.48  11.2%  ******
 19   1200.896   13.97  11.6%  ******
 20   1141.920   14.69  12.2%  ******
 21   1088.928   15.41  12.8%  *******
 22   1051.840   15.95  13.3%  *******
 23   1003.136   16.72  13.9%  *******
 24    963.168   17.42  14.5%  ********
 25    928.128   18.08  15.1%  ********
 26    906.432   18.51  15.4%  ********
 27    874.656   19.18  16.0%  ********
 28    825.120   20.33  16.9%  *********
 29    820.416   20.45  17.0%  *********
 30    787.008   21.32  17.8%  *********
 31    774.304   21.67  18.1%  *********
 32    764.640   21.94  18.3%  *********
Running kernel kernel_few_threads_d2 which uses 14 regs.
 Num      Time    Data
 Wps         s    GB/s    Pct
  1  11891.712    1.41   1.2%  *
  2   5725.792    2.93   2.4%  **
  3   3856.128    4.35   3.6%  **
  4   2995.104    5.60   4.7%  ***
  5   2448.256    6.85   5.7%  ***
  6   2022.112    8.30   6.9%  ****
  7   1749.920    9.59   8.0%  ****
  8   1507.680   11.13   9.3%  *****
  9   1353.792   12.39  10.3%  ******
 10   1235.872   13.58  11.3%  ******
 11   1144.224   14.66  12.2%  ******
 12   1056.416   15.88  13.2%  *******
 13    984.960   17.03  14.2%  *******
 14    890.752   18.83  15.7%  ********
 15    844.896   19.86  16.5%  *********
 16    820.544   20.45  17.0%  *********
 17    774.464   21.66  18.1%  *********
 18    746.304   22.48  18.7%  **********
 19    684.896   24.50  20.4%  **********
 20    665.312   25.22  21.0%  **********
 21    661.856   25.35  21.1%  **********
 22    642.304   26.12  21.8%  **********
 23    638.304   26.28  21.9%  **********
 24    619.520   27.08  22.6%  ***********
 25    619.936   27.06  22.6%  ***********
 26    625.376   26.83  22.4%  **********
 27    620.992   27.02  22.5%  ***********
```

```
 28    624.832   26.85  22.4%  ***********
 29    623.456   26.91  22.4%  ***********
 30    619.104   27.10  22.6%  ************
 31    619.872   27.07  22.6%  ************
 32    620.320   27.05  22.5%  ************
Running kernel kernel_few_threads_d4 which uses 21 regs.
 Num      Time    Data
 Wps       s    GB/s    Pct
  1   6800.608    2.47   2.1%  **
  2   3439.584    4.88   4.1%  **
  3   2324.416    7.22   6.0%  ***
  4   1759.552    9.53   7.9%  ****
  5   1436.416   11.68   9.7%  *****
  6   1203.040   13.95  11.6%  ******
  7   1047.872   16.01  13.3%  *******
  8    927.232   18.09  15.1%  ********
  9    831.552   20.18  16.8%  *********
 10    761.504   22.03  18.4%  *********
 11    715.200   23.46  19.5%  **********
 12    668.576   25.09  20.9%  ***********
 13    657.568   25.51  21.3%  ***********
 14    647.840   25.90  21.6%  ***********
 15    649.184   25.84  21.5%  ***********
 16    643.904   26.06  21.7%  ***********
 17    646.400   25.95  21.6%  ***********
 18    644.480   26.03  21.7%  ***********
 19    646.016   25.97  21.6%  ***********
 20    649.280   25.84  21.5%  ***********
 21    650.368   25.80  21.5%  ***********
 22    653.312   25.68  21.4%  ***********
 23    649.664   25.82  21.5%  ***********
 24    652.608   25.71  21.4%  ***********
 25    655.616   25.59  21.3%  ***********
 26    652.192   25.72  21.4%  ***********
 27    662.464   25.33  21.1%  ***********
 28    658.304   25.49  21.2%  ***********
 29    651.072   25.77  21.5%  ***********
 30    655.200   25.61  21.3%  ***********
 31    654.976   25.62  21.3%  ***********
 32    650.528   25.79  21.5%  ***********
Running kernel kernel_few_threads_d8 which uses 27 regs.
 Num      Time    Data
 Wps       s    GB/s    Pct
  1   5651.040    2.97   2.5%  **
  2   2851.872    5.88   4.9%  ***
  3   1922.464    8.73   7.3%  ****
  4   1457.280   11.51   9.6%  *****
  5   1196.672   14.02  11.7%  ******
  6   1065.312   15.75  13.1%  *******
  7    917.120   18.29  15.2%  ********
  8    827.648   20.27  16.9%  *********
```

```
  9    743.616   22.56  18.8%  **********
 10    697.696   24.05  20.0%  **********
 11    675.680   24.83  20.7%  ***********
 12    793.856   21.13  17.6%  *********
 13    680.672   24.65  20.5%  ***********
 14    688.576   24.37  20.3%  **********
 15    704.288   23.82  19.9%  **********
 16    727.872   23.05  19.2%  **********
 17    713.856   23.50  19.6%  **********
 18    710.144   23.63  19.7%  **********
 19    716.992   23.40  19.5%  **********
 20    721.312   23.26  19.4%  **********
 21    727.968   23.05  19.2%  **********
 22    735.104   22.82  19.0%  **********
 23    734.464   22.84  19.0%  **********
 24    735.328   22.82  19.0%  **********
 25    743.776   22.56  18.8%  **********
 26    750.464   22.36  18.6%  **********
 27    746.400   22.48  18.7%  **********
 28    737.536   22.75  19.0%  **********
 29    737.440   22.75  19.0%  **********
 30    746.144   22.49  18.7%  **********
 31    742.816   22.59  18.8%  **********
 32    746.144   22.49  18.7%  **********


GPU 0: Tesla K20c @ 0.71 GHz WITH 4799 MiB GLOBAL MEM
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: L2: 1280 kiB   MEM to L2: 208.0 GB/s  SP 1760.9 GFLOPS  OP/ELT 33.86
GPU 1: Quadro 5000 @ 1.03 GHz WITH 2559 MiB GLOBAL MEM
GPU 1: CC: 2.0  MP: 11  CC/MP:  32  TH/BL: 1024
GPU 1: SHARED: 49152 B  CONST: 65536 B  # REGS: 32768
GPU 1: L2: 640 kiB   MEM to L2: 120.0 GB/s  SP 361.2 GFLOPS  OP/ELT 12.04
Using GPU 1

CUDA Routine Resource Usage:
For kernel_many_threads:
       0 shared, 72 const, 0 loc, 11 regs; 1024 max threads per block.
For kernel_few_threads_d2:
       0 shared, 72 const, 0 loc, 14 regs; 1024 max threads per block.
For kernel_few_threads_d4:
       0 shared, 72 const, 0 loc, 21 regs; 1024 max threads per block.
For kernel_few_threads_d8:
       0 shared, 72 const, 0 loc, 27 regs; 1024 max threads per block.

Preparing for 11 blocks operating on 1048576 vectors of 2 elements.
Running kernel kernel_many_threads which uses 11 regs.
 Num      Time    Data
 Wps       s    GB/s    Pct
  1   1957.728    8.57   7.1%  ****
```

```
  2   1004.448   16.70  13.9%  *******
  3    679.776   24.68  20.6%  ***********
  4    510.528   32.86  27.4%  **************
  5    442.048   37.95  31.6%  ****************
  6    381.024   44.03  36.7%  ******************
  7    332.736   50.42  42.0%  ********************
  8    282.272   59.44  49.5%  ************************
  9    271.648   61.76  51.5%  *************************
 10    274.272   61.17  51.0%  ************************
 11    242.208   69.27  57.7%  ****************************
 12    241.792   69.39  57.8%  ****************************
 13    216.640   77.44  64.5%  *******************************
 14    222.912   75.26  62.7%  ******************************
 15    201.088   83.43  69.5%  *********************************
 16    196.032   85.58  71.3%  *********************************
 17    190.816   87.92  73.3%  **********************************
 18    192.192   87.29  72.7%  **********************************
 19    177.280   94.64  78.9%  ************************************
 20    174.304   96.25  80.2%  ************************************
 21    171.680   97.72  81.4%  *************************************
 22    170.048   98.66  82.2%  *************************************
 23    171.040   98.09  81.7%  *************************************
 24    170.432   98.44  82.0%  *************************************
 25    170.944   98.14  81.8%  *************************************
 26    171.328   97.92  81.6%  *************************************
 27    171.136   98.03  81.7%  *************************************
 28    171.456   97.85  81.5%  *************************************
 29    172.320   97.36  81.1%  ************************************
 30    172.352   97.34  81.1%  ************************************
 31    172.800   97.09  80.9%  ************************************
 32    173.056   96.95  80.8%  ************************************
Running kernel kernel_few_threads_d2 which uses 14 regs.
 Num      Time    Data
 Wps       s    GB/s    Pct
  1   1089.856   15.39  12.8%  *******
  2    559.136   30.01  25.0%  *************
  3    413.568   40.57  33.8%  *****************
  4    302.848   55.40  46.2%  ***********************
  5    317.152   52.90  44.1%  **********************
  6    259.072   64.76  54.0%  ***************************
  7    244.352   68.66  57.2%  *****************************
  8    218.848   76.66  63.9%  ********************************
  9    217.888   77.00  64.2%  ********************************
 10    185.728   90.33  75.3%  ***************************************
 11    181.856   92.26  76.9%  ****************************************
 12    186.208   90.10  75.1%  ***************************************
 13    190.784   87.94  73.3%  **************************************
 14    196.992   85.17  71.0%  ************************************
 15    199.616   84.05  70.0%  ************************************
 16    205.920   81.47  67.9%  ***********************************
 17    170.528   98.38  82.0%  *****************************************
```

```
 18    171.488   97.83  81.5%  ****************************************
 19    171.616   97.76  81.5%  ****************************************
 20    171.520   97.81  81.5%  ****************************************
 21    171.328   97.92  81.6%  ****************************************
 22    174.976   95.88  79.9%  ****************************************
 23    173.088   96.93  80.8%  ****************************************
 24    174.304   96.25  80.2%  ****************************************
 25    174.592   96.09  80.1%  ****************************************
 26    177.024   94.77  79.0%  ***************************************
 27    174.144   96.34  80.3%  ****************************************
 28    174.016   96.41  80.3%  ****************************************
 29    175.136   95.80  79.8%  ****************************************
 30    174.112   96.36  80.3%  ****************************************
 31    175.680   95.50  79.6%  ***************************************
 32    176.544   95.03  79.2%  ***************************************
Running kernel kernel_few_threads_d4 which uses 21 regs.
 Num      Time    Data
 Wps       s    GB/s    Pct
  1    660.128   25.42  21.2%  ***********
  2    359.584   46.66  38.9%  *******************
  3    275.648   60.86  50.7%  ************************
  4    227.456   73.76  61.5%  ******************************
  5    197.184   85.08  70.9%  ***********************************
  6    186.240   90.08  75.1%  *************************************
  7    185.696   90.35  75.3%  *************************************
  8    191.264   87.72  73.1%  ************************************
  9    194.656   86.19  71.8%  ************************************
 10    198.912   84.34  70.3%  ***********************************
 11    202.368   82.90  69.1%  **********************************
 12    213.472   78.59  65.5%  ********************************
 13    216.064   77.65  64.7%  ********************************
 14    216.992   77.32  64.4%  ********************************
 15    213.760   78.49  65.4%  ********************************
 16    173.920   96.47  80.4%  ****************************************
 17    173.632   96.63  80.5%  ****************************************
 18    173.728   96.57  80.5%  ****************************************
 19    176.512   95.05  79.2%  ***************************************
 20    175.008   95.87  79.9%  ****************************************
 21    175.328   95.69  79.7%  ****************************************
 22    175.296   95.71  79.8%  ****************************************
 23    176.384   95.12  79.3%  ***************************************
 24    175.040   95.85  79.9%  ****************************************
 25    177.440   94.55  78.8%  ***************************************
 26    175.904   95.38  79.5%  ***************************************
 27    177.056   94.76  79.0%  ***************************************
 28    175.744   95.46  79.6%  ***************************************
 29    176.032   95.31  79.4%  ***************************************
 30    176.256   95.19  79.3%  ***************************************
 31    176.768   94.91  79.1%  ***************************************
 32    176.864   94.86  79.0%  ***************************************
Running kernel kernel_few_threads_d8 which uses 27 regs.
```

Num Wps	Time s	Data GB/s	Pct	
1	541.408	30.99	25.8%	**********
2	318.048	52.75	44.0%	**********
3	245.248	68.41	57.0%	**********
4	207.296	80.93	67.4%	**********
5	188.512	89.00	74.2%	**********
6	189.664	88.46	73.7%	**********
7	201.824	83.13	69.3%	**********
8	208.640	80.41	67.0%	**********
9	211.232	79.43	66.2%	**********
10	224.448	74.75	62.3%	**********
11	223.232	75.16	62.6%	**********
12	224.128	74.86	62.4%	**********
13	177.792	94.36	78.6%	**********
14	177.088	94.74	78.9%	**********
15	177.248	94.65	78.9%	**********
16	178.496	93.99	78.3%	**********
17	178.496	93.99	78.3%	**********
18	177.664	94.43	78.7%	**********
19	178.496	93.99	78.3%	**********
20	178.016	94.25	78.5%	**********
21	179.168	93.64	78.0%	**********
22	178.752	93.86	78.2%	**********
23	177.664	94.43	78.7%	**********
24	179.872	93.27	77.7%	**********
25	176.928	94.83	79.0%	**********
26	177.888	94.31	78.6%	**********
27	178.208	94.14	78.5%	**********
28	180.384	93.01	77.5%	**********
29	179.520	93.46	77.9%	**********
30	179.584	93.42	77.9%	**********
31	177.920	94.30	78.6%	**********
32	180.544	92.93	77.4%	**********