**LSU EE 7722** **Homework 2** Solution **Due: 15 April 2015**

**Problem -1:** Start reading the paper "Scaling the Power Wall: A Path to Exascale," by Villa *et al* in Supercomputing 14. It can be accessed from within `lsu.edu` using `http://www.ece.lsu.edu/gp/srefs/p830-villa.pdf`. There are no questions about the paper in this assignment.

**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 Makefile features and output files described below.

This package is similar to the previous assignment, including command-line arguments. For this assignment the machine language code for the various kernels will be analyzed.

Compile the code and look at a directory listing. File `hw02.ptx` contains the compiler intermediate form of the code, file `hw02.cubin` contains the object (binary) form of the GPU code, and file `hw02.sass` contains the diassembled GPU code. The files are roughly created in this order.

Locate the documentation for PTX and SASS. The ptx documentation can be found at `/usr/local/cuda/doc/pdf/ptx_isa_4.2.pdf` or `/usr/local/cuda/doc/html/parallel-thread-execution/index.html` or just Web search for *Parallel Thread Execution ISA Version 4.2*. The documentation for SASS can be found in `/usr/local/cuda/doc/html/cuda-binary-utilities/index.html` or search for *CUDA Binary Utilities*. Create a Web browser link or otherwise keep the documentation handy.

Load the `hw02.ptx` and `hw02.sass` into a text editor (Emacs with the class setup is recommended) and search for `mxv_g_only` or some other kernel in each file. Note that the code in these two files is similar but not identical. You should be able to figure out what most instructions do. The ptx language is thoroughly documented, but for SASS all that's given is a rough description of what each instruction does.

For this assignment you might need to modify the makefile to change the compiler target and disassembly options. First, load `Makefile` into an editor. To change the compiler target locate the text `--gpu-architecture=sm_35`. This tells the compiler which GPU architecture to emit code for, the `35` refers to CC 3.5. Change the `35` for the desired CC. To find the list of supported architectures issue the command `nvcc --help` and search for `gpu-architecture`, or look for the documentation. Try changing to 20 and re-compiling (pressing `F9` in Emacs with the class setup).

To adjust the amount of information in the SASS files add or remove flags stored in the `CUDUMP` variable in `Makefile`. With the flag `--print-line-info` source file name and line numbers are added to the `SASS` files. With the flag `--print-instruction-encoding` the encoded form of the GPU instruction is shown to the right of the disassembled instruction.

**Problem 1:** Determine the following NVIDIA CC 3.5 instruction set features by modifying kernel `tryout` in file `hw02.cu`, building, and examining the disassembled code and the encoded form of the instruction. Note that `tryout` is not run by `hw02`, and if it were run it might encounter a run-time error.

Offsets in load instructions improve performance by reducing the amount of arithmetic instructions needed to compute addresses. For example, in the code fragment below

```
/*0070*/                   LD.E R4, [R8];
/*0068*/                   LD.E R2, [R8+0x80];
```

both load instructions use the same base register, `R8`. They can share the same base register because the compiler was able to determine that the second load was from an address `0x80` bytes

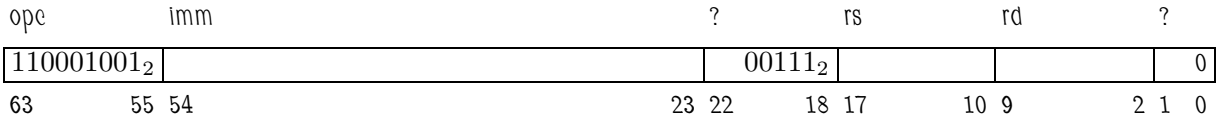
after the address of the first load. If load instructions didn't have offsets, or if the compiler could not determine the offset at compile time additional instructions would be needed to compute the address of the second load. The NVIDIA compiler targeting CC 3.5 uses three additional instructions, though a good CPU compiler might just need one extra instruction. (The difference is due to the lack of 64-bit integer instructions.)

The maximum size of an offset is determined by the ISA. Determine the maximum offset size for CC 3.5. Show which bits in the instruction are used for the offset.

Solve this problem by modifying the code in `tryout`, compiling, and inspecting the SASS output. Remember that `tryout` is not actually run, so don't worry about the code generating runtime errors, such as exceeding array bounds.

Based on experimentation, the largest offset obtainable is `0x7ffffffc`, which spans 31 bits. But negative offsets are also possible, so the full size is 32 bits. Comparison of the encoded forms of several `LD.E` instructions reveals the position of the immediate field, and of the register fields (which must be 8 bits each since CC 3.5 devices can access 255 registers). The instruction format is:

```
LD.E rd, [ rs + imm ]
```




```
Sample instructions used to infer format:

LD.E R7, [R14];                         /* 0xc4800000001c381c */
LD.E R5, [R8];                          /* 0xc4800000001c2014 */
LD.E R4, [R8+0x80];                     /* 0xc4800000401c2010 */
LD.E R4, [R8+-0x80];                    /* 0xc4ffffffc01c2010 */
LD.E R2, [R8+0x6aaaaaa8];               /* 0xc4b55555541c2008 */
LD.E R2, [R8+0x7ffffffc];               /* 0xc4bfffffe1c2008 */

c4800000001c381c  LD.E R7, [R14]
c4 1 000 0000 000 0 001 c381c   <- Break up into parts.
hh b bbb hhhh hhh b bbb hhhhh   <- h, hex;  b, binary.
```

**Problem 2:** The code in `mxv_o_per_thd` uses type punning to access the input vector elements using 4-element vector load instructions. Two seemingly identical versions of the vector load is performed. Both are correct and with both the compiler has emitted a 4-element vector load instruction. However the one marked Plan A is slower than the one marked Plan B. Explain why.

The index expression in Plan B consists of two terms, $hN/4$ and $c/4$. Term $hN/4$ does not change in the `c` loop and so the compiler can compute it once per `h` loop iteration. The compiler will completely unroll the `c` loop and so the expression `c/4` is a constant that can be used as an offset in a load instruction. For example, for $N = 8$ there will be two loads, one corresponding to $c = 0$ and the other corresponding to $c = 4$. The offsets will be 0 and $4/4 \times 16 = 16$, respectively. The same base register will be used for both loads, which will be set to the value of $hN/4$ computed before the `c` loop.

So how is Plan A different? Because the compiler does not realize that both $c$ and $hN$ will be multiples of 4. Therefore it can't be sure that $(hN + c)/4$ will be the same as $hN/4 + c/4$, and so it must re-calculate the expression for each value of $c$. If the expression were changed to $(hN + c) \times 4$ the compiler would use offsets for the loads. Note that since `N` is a compile-time constant set to 8 the compiler should have known that $hN$ would be a multiple of 4.

**Problem 3:** Hand-estimate the performance of the `mxv_o_per_thd` kernel in the following way. *See 2013 Homework 6 Problem 2 for a similar problem.*

(*a*) Using the instruction latencies provided below, compute the latency of a single `h` loop iteration. Call the latency $t_{\mathrm{L}}$ (for time of loop body). Note that to compute the latency one must keep track of instruction dependencies.

```
 Latency     Instruction
 / Cycles    Category
 ----------------------
 200         Global loads and stores
  24         Shared memory loads and stores and instructions use shared
             memory operands.
  12         All other instructions.
```

The loop latency is $\boxed{t_{\mathrm{L}} = 334\,\mathrm{cyc}}$. The work to compute this latency is shown below. The column headed **AA** shows the cycle number the instruction will start, **BBB** shows the registers modified by the instruction, and **CCCCC** shows when the registers will be available.

The table was constructed under the assumption that warps can be issued at a rate of one per cycle.

```
.L_2:                                          AA BBB    CCCCC
     SHF.L R24, RZ, 0x3, R0;                    0 R24    at 12
     BFE R3, R0, 0x11c;                         1 R3     at 13
     IADD R0, R0, R22;                          2 R0     at 14
     IMAD.U32.U32.HI R2, R3, 0x4, R24;         13 R2     at 25
     SHR R4, R2, 0x2;                          25 R4     at 37
     IMAD R2.CC, R4, R19, c[0x3][0x130];       37 R2     at 49
     IMAD.HI.X R3, R4, R19, c[0x3][0x134];     49 R3     at 61
     LD.E.128 R8, [R2];                        61 R8-R11 at 261
     LD.E.128 R4, [R2+0x10];                   62 R4-R7  at 262
     FMUL R9, R9, R16;                        261 R9     at 273
     FMUL R25, R5, R14;                       262 R25    at 274
     FFMA R5, R13, R8, R9;                    273 R5     at 285
     FFMA R8, R23, R4, R25;                   274 R8     at 286
     ISETP.LT.AND P0, PT, R0, c[0x3][0x0]     275 P0     at 287
     FFMA R4, R10, R17, R5;                   285 R4     at 297
     IADD R3, R24, R12;                       286 R3     at 298
     FFMA R6, R6, R15, R8;                    287 R6     at 299
     FFMA R4, R11, R21, R4;                   297 R4     at 309
     IMAD R2.CC, R3, R20, c[0x3][0x128];      298 R2     at 310
     FFMA R5, R7, R18, R6;                    299 R5     at 311
     F2F.F32.F32 R4, R4;                      309 R4     at 321
     IMAD.HI.X R3, R3, R20, c[0x3][0x12c];    310 R3     at 322
     FADD R4, R4, R5;                         321 R4     at 333
     ST.E [R2], R4;                           333 No output
 @P0 BRA `(.L_2);                             334 No output
```

(*b*) Compute the minimum number of threads it would take for the kernel to completely utilize the dispatch hardware of a CC 3.5 device. Call this number $n_d$. Use the instruction throughputs given in Section 5.4 of the C Programming Guide version 7. For example, suppose the thread had 3 `FMADD` instructions and 4 integer add instructions. The time to issue $n_d$ such threads would be $3n_d/192 + 4n_d/32$. To saturate the device solve $3n_d/192 + 4n_d/32 = t_{\mathrm{L}}$ for $n_d$.

In the table below the instructions are grouped by type, and the issue rate for each type is shown. The rates for load and store instructions and for branch instructions are assumed. Using these instruction types and rates we get:

$$1n_d/32+9n_d/192+2n_d/160+5n_d/32+1n_d/160+3n_d/64+1n_d/32+2n_d/64+1n_d/64 = \frac{121}{320}n_d = t_{\mathrm{L}}$$

$$n_d = \frac{320}{121}t_{\mathrm{L}} = \frac{320}{121}334 = 883.306$$

.

We would need about 883 threads, or rounding up, 28 warps.

```
Instructions Grouped by Category.

     F2F.F32.F32 R4, R4;                    309 R4     at 321
     1 insn  32 / 32  warps per cycle

     FADD R4, R4, R5;                       321 R4     at 333
     FFMA R4, R10, R17, R5;                 285 R4     at 297
     FFMA R4, R11, R21, R4;                 297 R4     at 309
     FFMA R5, R13, R8, R9;                  273 R5     at 285
     FFMA R5, R7, R18, R6;                  299 R5     at 311
     FFMA R6, R6, R15, R8;                  287 R6     at 299
     FFMA R8, R23, R4, R25;                 274 R8     at 286
     FMUL R25, R5, R14;                     262 R25    at 274
     FMUL R9, R9, R16;                      261 R9     at 273
     9 insn.  192 / 32  warps per cycle

     IADD R0, R0, R22;                        2 R0     at 14
     IADD R3, R24, R12;                     286 R3     at 298
     2 insn.  160 / 32  warps per cycle

     IMAD R2.CC, R3, R20, c[0x3][0x128];    298 R2     at 310
     IMAD R2.CC, R4, R19, c[0x3][0x130];     37 R2     at 49
     IMAD.HI.X R3, R3, R20, c[0x3][0x12c];  310 R3     at 322
     IMAD.HI.X R3, R4, R19, c[0x3][0x134];   49 R3     at 61
     IMAD.U32.U32.HI R2, R3, 0x4, R24;       13 R2     at 25
     5 insn.  32 / 32 warps / cycle

     ISETP.LT.AND P0, PT, R0, c[0x3][0x0]   275 P0     at 287
     1 insn.  160 / 32  wp / cyc

     LD.E.128 R4, [R2+0x10];                 62 R4-R7  at 262
     LD.E.128 R8, [R2];                      61 R8-R11 at 261
     ST.E [R2], R4;                         333 No output
     3 insn.  64 / 32  wp / cyc (Assumed)

     BFE R3, R0, 0x11c;                       1 R3     at 13
     1 insn.  32 / 32  wp / cycle

     SHF.L R24, RZ, 0x3, R0;                  0 R24    at 12
     SHR R4, R2, 0x2;                        25 R4     at 37
     2 insn.  64 / 32  wp / cycle
```

```
@P0 BRA `(.L_2);                              334 No output
    1 insn.  64 / 32  wp / cycle (Assumed)
```

($c$) Launching the kernel with a block with $n_d$ threads would only saturate dispatch if no other resource became saturated first. Let $M$ denote the number of multiprocessors. What is the minimum amount of off-chip bandwidth needed to enable full dispatch utilization by the kernel launched with $n_d$ threads per MP? Solve the problem based the answers to the previous problem and based upon the amount of off-chip data transfer performed by the kernel.

Each thread loads eight floats per iteration and stores one float per iteration. However groups of $m$ threads read the same input element, where $m$ is the size of the output vector. (In the program that's called `M`, which is not to be confused with the number-of-multiprocessors $M$ given above.) So the total amount of data read per thread is $8/m$ floats, using the value in the code, $m = 8$, we have a net of 1 float read per cycle. Each thread writes a different value, so the net amount of data per thread is $8/m + 1 = 2$ floats or eight bytes. Counting $M$ blocks (one per MP) of $n_d$ threads each the total data is $8Mn_d\,\mathrm{B}$ per iteration.

To compute the amount of bandwidth needed we need to know the number of iterations per second. The Kepler K20c has a clock frequency of $710\,\mathrm{MHz}$, or a clock period of $1.4\,\mathrm{ns}$, so one iteration is $t_\mathrm{L} = 334\,\mathrm{cyc} = 470\,\mathrm{ns}$. That corresponds to $1/t_\mathrm{L} = 2.13$ million iterations per second. The amount of bandwidth needed is thus $\frac{8Mn_d\,\mathrm{B}}{t_\mathrm{L}} = 15.0M\,\mathrm{GB/s}$. For a K20c with $M = 13$ we'd need $195\,\mathrm{GB/s}$ of bandwidth which is close to the limit of the chip.

**Problem 4:** One point often made in class is that latencies in GPUs are long to keep the hardware simple. One possible way of reducing latencies is by providing bypass paths so that a dependent instruction would only have to wait, say, 6 cycles (a floating point functional unit latency) rather than 12 cycles (perhaps the latency from register file source operand read to register file result write).

Bypass paths aren't free, but neither are registers. Compute how many fewer registers will be needed if the latency between dependent non-global-memory instructions drops from 12 to 6 cycles. Do this by re-computing latency and $n_d$. To determine the number of registers used by a thread in each kernel see the program output.

Repeating the iteration latency analysis with a 6-cycle latency yields a $t_\mathrm{L} = 263\,\mathrm{cyc}$, for a savings of 71 cycles. The number of threads needed to saturate instruction dispatch is now $n_d = \frac{320}{121}t_\mathrm{L} = \frac{320}{121}263 = 695.5$. Based on the code output, kernel `mxv_o_per_thd` uses 26 registers. The number of registers needed per MP has dropped from 22958 (or $89.7\,\mathrm{kiB}$) to 18096 (or $70.7\,\mathrm{kiB}$).

We might be disappointed because halving the latency did not halve the register requirement. Or, we might have thought that the improvement would be tiny because memory latency was over 10 times larger to begin with. In our kernel memory latency was exposed in only one place (the `FMUL R9`), whereas non-memory functional unit latency was exposed in several places, such as the first `IMAD` and `SHR`.

The numbers above are for a GPU in which data bandwidth will not become saturated. The percentage reduction in the number of registers is comparable if we consider data bandwidth as the limiter.