LSU EE 7722 **Homework 2** **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.

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.

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_{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.
```

(*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.

(*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.

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.