LSU EE 7722 Homework 3 Due: 28 March 2014

Problem 0: If necessary, follow the instructions for class account setup and homework workflow on the course procedures page `http://www.ece.lsu.edu/gp/proc.html`. Be sure to update your copy of the repo with the command "git pull".

For background and help with CUDA programming see the CUDA C Programming Guide, a version is linked to the course Web pages at `http://www.ece.lsu.edu/gp/ref.html` it can also be found under directory `/usr/local/cuda/doc` on a system with CUDA installed.

The assignment file, `hw03.cu`, contains three versions of a kernel for computing the sum of sections of a scattered array. Data is in array `data_array`, the element numbers to select are in array `idx_array`. The kernel adds up sections of length `clength`, which is a compile-time constant, and writes the sum to output array `sum_array`.

Compile and run the program. The makefile will generate ptx and CUDA assembly language versions of the CUDA code, in files `hw03.ptx` and `hw03.sass`.

The program starts by writing information about the GPUs that are available, it will always use GPU 0.

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 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 0
```

Next it shows the resources used by each of the three kernels.

```
CUDA Kernel Resource Usage:
For sums_0:
       0 shared, 88 const, 0 loc, 24 regs; 1024 max threads per block.
```

After printing this information the program runs each kernel multiple times, in a launch configuration varying from 1 to the maximum number of warps. (The sample below does not show the entire output.)

```
Running kernel sums_0 which uses 24 regs on 13 blocks.
 Num      Time    Data
 Wps        s     GB/s    Pct
  2   7914.784    0.53   0.3%  0
  4   3989.920    1.05   0.5%  0
```

The timing for each configuration is printed, along with a possibly incorrect (see Problem 1) percentage of peak data bandwidth realized by the code. The peak bandwidth is shown as a number and as a line of digits (zeros above). If the percentage, under `Pct`, is 100% that means the kernel is transferring data at the maximum rate (based upon certain assumptions).

If there are any difficulties running the code ask for help. There is nothing to submit for this first problem.

Problem 1: As in the previous assignments, the program shows the execution time of each kernel launched over a range of warp sizes, along with a bar graph showing what percentage of the off-chip memory bandwidth was used. This percentage is based upon an assumption about the amount of data that needs to move on and off the GPU chip, that assumption is coded in the assignment to the variable `data_size` in the routine `main`:

```
  const double data_size = idx_array_bytes * data_array_bytes + sum_array_bytes;
```

This assumption is wrong.

(*a*) Correct the assumed data size for `sums_0`. Explain your reasoning, either in comments next to the assignment or in something written submitted with the solution. (Before solving this look over the remainder of this assignment.)

Problem 2: The loop below, from `sums_0`, accesses two arrays noncontiguously, `data_array` and `idx_array`. Since `clength` is a compile-time constant we can expect the compiler to unroll the loop and perhaps avoid some noncontiguous access to `idx_array` by using vector load instructions.

```
for ( int i=0; i<clength; i++ )
  sum += dapp.d_data_array[dapp.d_idx_array[idx_piece_start+i]];
```

We saw vector load instructions used in the previous homework assignment for loading both the `x` and `y` components of `a`. For example, the load instruction

```
LD.E.64 R14, [R16+0x100];
```

loads 64 bits (denoted, of course, by the 64) and since registers are 32 bits, the values are placed in two registers, R14 and R15.

(*a*) Examine the assembly code for `sums_0` and determine whether the compiler is doing this also for accesses to `idx_array`. Show the assembly language instructions that you think are being used to load from `idx_array`.

(*b*) Describe the potential benefit of using the vector load for two cases: when the L1 cache is being used, and when the L1 cache is not being used.

(*c*) Modify routine `sums_1` so that it does use a vector load. Do this by using an `int4*` for the `idx_array`, but use the same storage currently allocated and trick the compiler into thinking you are accessing `int4*` values (a technique called *type punning*). Please verify that the code is working as intended by inspecting the assembly code and looking for the vector loads.

(*d*) A run on a Fermi system shows that the performance of `sums_0` is best at about 14 warps, and gets steadily worse as more warps are added. In contrast, in a correct solution to `sums_1` the performance improves or stays the essentially same as more warps are added. Explain this phenomenon. On a Kepler the shape of `sums_0` would look more like `sums_1` (though `sums_1` would be faster).

Problem 3: Kernel `sums_0` also suffers from noncontiguous access to `data_array`. Fix the issue in kernel `sums_2` by using shared memory to cache parts of `data_array`.

(*a*) Write the code so that it works correctly for any size of `data_array` and `idx_array`. Don't assume that all of `data_array` will fit in the cache at one time. In addition to using shared memory for `data_array` you might also consider using local memory for `idx_array`, depending on your solution approach.

(*b*) Estimate the amount of off-MP data accessed in your solution in terms of the size of `data_array` and `idx_array`, and perhaps parameters in your program. Indicate if you believe there is a better solution, in particular if the array sizes were different.