LSU EE 7722 **Homework 1** **Due: 28 March 2024**

Basic Setup

Follow the instructions for class account setup on `https://www.ece.lsu.edu/gp/proc.html`.
Code for this assignment is in directory `../hw/gpm/2024/hw01`.

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 `.../2024/hw01` 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 `.../2024/hw01` is the current directory). Either method runs a makefile that builds all examples in the directory. It builds three versions of each program, one taking the base name of the main file, such as `hw01`, one with the suffix `-debug`, such as `hw01-debug`, and one with the suffix `-cuda-debug`, such as `hw01-cuda-debug`. The versions with the `-cuda-debug` suffix are compiled with host optimization turned off and CUDA debugging turned on, which facilitates debugging but slows down execution. To debug CUDA or host (CPU) code use the Cuda version of gdb, `cuda-gdb`. Note that the `-cuda-debug` versions will run much more slowly than the regular versions. The executables with the suffix `-debug` are compiled with host optimization turned off but CUDA debugging turned off. Use `gdb` or `cuda-gdb` to debug these.

Running make on a clean directory will produce a large amount of output. The make program and the file it reads, `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.

Quickly check whether the build is successful with the command `./hw01 0 32`. It should produce output ending with a line something like this `32 2566 6.6 27.3 3.8 8.0 4 58 1 2`.

The makefile will compile code for a GPU on the system it was run. Re-run make when moving to a system using a different GPU. 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.

Background and Reference Material

For this assignment one must be able to write, or at least modify, CUDA kernels. A good reference is the CUDA C++ Programming Guide, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Focus on Chapter 5 up to and including 5.3 (Memory Hierarchy), but skip 5.2.1 (Thread Block Clusters). For sample code a good place to start is 2021 Homework 1, and other past assignments given in this course. The CUDA C used in this assignment is very close to C++20. A good reference for C and C++ is `https://en.cppreference.com/w/`.

In the references below some information is provided for specific architectures, either by CC (*e.g.*, 8.0) or by name (*e.g.*, Ampere). Both the CC 8.0 and CC 8.6 GPUs implement the Ampere architecture, 8.9 GPUs implement Ada Lovelace, and 9.0 implements Hopper. For this assignment only consider CC 8.x and 9.0 GPUs. The compute capability (CC) of the lab GPUs is shown on the system status page.

A solution to these problems requires some understanding of the hardware structure, in particular how requests are issued to the L1 cache. Some of that material is reviewed in this assignment. For additional description see Chapter 7 of the Programming Guide for the basics (but not including the L1 cache), and also Chapter 19 (Compute Capabilities) for some more details.

The hardware is covered in greater depth in the Kernel Profiling Guide,
`https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html`. Focus on Section 3.1 (Metrics Guide, Hardware Model) and Chapter 9 (Memory Chart). There is no need to read the material on *how* metrics are collected and there is no need to run the profiler yourself. The assignment code uses the CUPTI API to collect data. In class an SM (or MP) was described as having several–usually four–*warp schedulers*. The Profiling Guide refers to warp schedulers as sub partitions. For this assignment requests to the L1 cache are all global requests. Later in the semester we will make shared and maybe local requests, but probably not texture or surface requests.

Using `hw01`

The code in `hw01.cu` contains several kernels that normalize vectors. The `hw01` program takes up to three

command-line arguments: `./hw01 NBLOCKS BLOCKSIZE INPUTSIZE`. The first indicates how many blocks to launch, the second indicates the number of threads per block, and the last indicates the input size.

To make sure it compiled correctly run it with arguments `./hw01 0 32`, that runs the kernels fewer times (explained below). To run it while working on your solution usually run it as `./hw01`.

The first argument is used to specify the number of blocks. When there are zero arguments, `./hw01`, or when the first argument is zero, `./hw01 0`, the number of blocks is set equal to the number of SMs. When the first argument is a positive integer, such as `./hw01 5`, the kernels will be launched with that many blocks, five blocks in the example. When the first argument is a negative integer, such as `./hw01 -5`, then each kernel will be launched with that many blocks *per SM*. For a GPU with 40 SMs and running with `./hw01 -5`, a total of $5 \times 40 = 200$ blocks will be launched per kernel. Note that there is no guarantee that five blocks will *simultaneously* run (be resident on) any SM, for example, if the kernels use lots of shared memory or registers fewer than five will run (and the others will have to wait).

The second argument specifies the number of warps per block. A positive value indicates the exact number of warps, for example, `./hw01 -3 4`, will run each kernel with a block size of 4 warps ($4 \times 32 = 128$ threads), and also launch 3 blocks per SM.

In many cases one wants to quickly compare the performance with different block sizes. For that omit the second argument or set it to zero, for example, `./hw01`. The program will launch each kernel multiple times, starting with 1 warp per block, up to 32 warps per block. Also, because the first argument was also omitted, the number of blocks is set equal to the number of SMs. Run time and other information will be shown for each launch.

The third argument specifies the input size. If the argument is positive, is specifies the input size in MiB (2^{20} bytes). For example, `./hw01 0 0 3.6`, specifies that the input size should be 3.6 MiB. If the argument is negative then it specifies the input size in multiples of the L2 cache size. For example, `./hw01 0 0 -0.5` indicates that the input size should be half the size of the L2 cache (and so the input itself will easily fit in the L2 cache). For this assignment (Homework 1 2024) the default is $\frac{1}{4}$ the L2 cache size, so that the input and output can both comfortably fit.

Program Output

Detailed output is obtained by running with 0 as the two command-line arguments:

```
[koppel@grace hw01]$ ./hw01 0 0
```

The first thing printed is information about each GPU connected to the system, followed by a line showing the chosen GPU:

```
GPU 0: NVIDIA GeForce RTX 4090 @ 2.52 GHz WITH 24207 MiB GLOBAL MEM
GPU 0: L2: 73728 kiB   MEM<->L2: 1008.1 GB/s
GPU 0: CC: 8.9  SM: 128  SP-FP32/SM: 128  DP-FP64/SM:  2  TH/BL: 1024
GPU 0: SHARED: 49152 B/BL  102400 B/SM  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 41288 SP GFLOPS  645 DP GFLOPS  COMP/COMM:  163.8 SP  5.1 DP
Using GPU 0
```

This assignment will only work on GPUs of CC 8 or greater.

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 8.9 (Ada Lovelace). The `MEM<->L2` field shows the off-chip bandwidth. `SM` indicates the number of streaming multiprocessors, also just called multiprocessors (MP's). `CC/SM` indicates the number of CUDA cores (single-precision functional units) per SM, `DP/SM` indicates the number of double-precision functional units per SM, 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 SM, this does not indicate whether any particular kernel is using that much shared memory or could use that much. The L1 cache size is usually the same size or a bit larger than the shared memory size. 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. (Most of the rest of the world counts a multiply-add as two operations, but in this class it's

one.) 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.)

Performance Data

Each kernel is run multiple times, starting with one warp per block, in successive runs increasing the number of warps per block. A line of performance data is printed for each run. Appearing below is a portion of the output for an RTX 4090, showing unmodified kernel qkv_base (and that kernel should not be modified) and kernel qkv_base_w2 with the assignment correctly solved.

```
Kernel (norm_base<4>).  Uses 26 registers.  n_l 1179648  d_l 4
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 1   32  5.4  3.0  1.0  1.0  23 1169   77  292 --*****
 2   18  5.4  3.1  1.0  1.0  40 2067  134  517 ---**********
 4   13  5.5  3.2  1.0  1.0  59 3020  188  755 +---***************
 8   12  5.6  3.3  1.0  1.0  63 3246  197  810 +----****************
12   12  5.8  3.3  1.0  1.0  64 3302  199  816 +----****************
16   14  5.9  3.3  1.0  1.2  60 3082  149  695 +---**************
24   28  6.2  3.2  1.0  2.5  45 2312   85  334 --*******
32   36  6.4  3.2  1.0  3.1  42 2193   65  264 -******
 The lines below are fictional and are there to explain the bar graph.
32   36  6.4  3.2  1.0  3.1  42 2193   65  264 +++++++++++++++++++++++++++++++++++++
32   36  6.4  3.2  1.0  3.1  42 2193   65  264 +++++++++++++++++++++++++++++++++++++++
```

The output above shows the result a kernel, norm_base<4>. (The name shown is how the function was named, including the template parameter.)

Column wp shows the number of warps per block in the run. If the number of blocks in a launch is not set to the number of SMs then there would also be a column headed ac, which would show the number of resident warps per SM. (The number of resident warps per SM is a multiple of the number of warps per block. By default the number of blocks in a launch is set equal to the number of SMs, and in such a case the value in the ac column would match the wp column.)

For a description of the I/el, BXW, N*R, and N*W columns see the Base Code Performance section further below.

The columns in the L2-Cache group show how much data is moving between the L1 and L2 caches. The N*R column (normalized amount of data read) shows how much data is read, scaled to the ideal amount. Its value is determined using a measured amount of data and a computed amount of ideal data. (Data is measured using the NVIDIA CUPTI profiling API.) A value of 1 is ideal, a value of 2 indicates that on average each element was read twice. The value under the N*W column (normalized amount of data written) shows how much data moved from L1 to L2, normalized to the ideal amount.

The value under the GB/s in the L2 group shows the measured data throughput between the L1 and L2 caches (in either direction, but L2 to L1 dominates). The number includes all SMs. The %pk column shows this L1/L2 data movement as a percentage of peak. The DRAM GB/s column shows the measured data throughput between the L2 cache and off-chip memory.

The t/μs column shows the measured execution time in microseconds. To the right of t/μs is a bar graph showing how busy three resources are (based on certain assumptions). Three resources are tracked, FMA (fused multiply/add) instructions, shown with a +, FMA along with load instructions, shown with a -, and data transfer, shown with a *. The data transfer shown is either L1/L2, indicated with an L2** in the column heading, or L2/Mem, indicated with a Mem** in the column heading. The right-most position of a resource's character indicates what fraction of the time that resource is busy. A resource is being used 100% of the time if its character reaches the rightmost position (the last = in the column heading over the bar graph).

That is true in the last line for the FMA resource, and in the penultimate line for the off-chip data transfer. In the last line we would say that the FP capability is being saturated (a good thing) and in the

penultimate line we would say that data transfer is being saturated (also a good thing given the assumptions made). Those last two lines are fictional. Consider the line for the 12 warp per SM run. The * is a bit more than halfway to the end. That indicates that L1/L2 data throughput is more than half of the peak possible. The instruction utilization, -, includes the FMAs, two loads, and one store per element.

Assignment Introduction

The code for this assignment normalizes vectors, which for this assignment means subtracting the average component value from each component. Consider $x \in \mathbb{R}^d$, a d-component vector of real numbers. Let $x_0, x_1, \ldots x_{d-1}$ denote the d components of vector x. Then $a = \frac{1}{d}\sum_{i=0}^{d-1} x_i$ is the mean (average) component value. Vector $y \in \mathbb{R}^d$ is a normalized version of x if $y_i = x_i - a$ for $i \in [0, d)$.

Each kernel is to normalize `n_l` vectors of `d_l` components, the x vectors are read from `l_in` and the y vectors are written to `l_out`. (The letter l is for layer.) The starting point is the base kernel, `norm_base`, in which each thread normalizes one vector:

```
template< int D_L = 0 > __global__ void
norm_base(elt_t* __restrict__ l_out, const elt_t* __restrict__ l_in)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;
  const int d_l = D_L ?: c_app.d_l;
  const int n_l = c_app.n_l;

  for ( int h = tid;  h < n_l;  h += n_threads )
    {
      elt_t sum = 0;

      // The Sum Loop
      for ( int i = 0;  i < d_l;  i++ ) sum += l_in[ h * d_l + i ];
      const elt_t avg = sum / d_l;

      // The Norm Loop
      for ( int i = 0;  i < d_l;  i++ ) l_out[ h * d_l + i ] = l_in[ h * d_l + i ] - avg;
    }
}
```

The code will be run for values of $d_l \in 4, 8, 32, 128, 1024$ and for n_l chosen so that the `l_in` and `l_out` arrays can both fit in the L2 cache (the default, but input size can be changed from the command line).

Base Code Performance

The base code runs acceptably for $d_l = 4$ and $d_l = 8$, but does horribly for $d_l \geq 32$:

```
Kernel (norm_base<4>).  Uses 26 registers.  n_h 1179648  d_h 4
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 1   33  5.4  3.0  1.0  1.0  22 1149   64  287 --*****
 2   19  5.4  3.1  1.0  1.0  39 2029  126  507 ---**********
 4   12  5.5  3.3  1.0  1.0  62 3192  107  798 +---*****************
 8   11  5.6  3.3  1.0  1.0  64 3288  237  821 +----*****************
16   14  5.9  3.3  1.0  1.2  56 2913  162  655 +---**************
24   29  6.2  3.2  1.0  2.5  44 2272   44  327 --******
32   35  6.4  3.2  1.0  3.1  43 2207   66  266 -******

Kernel (norm_group<8,1>).  Uses 32 registers.  n_h 589824  d_h 8
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
```

```
 1   68  5.3  6.5  1.0  1.0  11  559  462  140 -***
 2   20  5.3  7.0  1.0  1.0  36 1854   83  464 ---*********
 4   13  5.3  7.1  1.0  1.0  57 2943  185  735 +---***************
 8   15  5.4  7.1  1.0  1.1  49 2547  168  614 ---*************
16   71  5.5  6.3  1.0  6.3  37 1929   31  133 -**
24   78  5.6  6.3  1.0  6.8  36 1876   28  121 -**
32   80  5.8  6.3  1.0  6.9  36 1868   28  118 -**

Kernel (norm_base<32>).  Uses 48 registers.  n_h 147456  d_h 32
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 1  149  4.4 24.1  1.0  8.0  22 1136   49   63 **
 2  156  4.4 24.0  1.0  8.0  21 1088    9   60 **
 4  141  4.4 24.3  1.0  8.0  23 1202   17   67 **
 8  142  4.5 24.9  1.0  8.0  23 1199   17   67 **
16  144  4.6 26.2  1.0  8.0  23 1180   15   66 **
24  147  4.7 26.6  1.0  8.0  23 1161   10   64 **
32  174  4.8 23.4  3.4  8.0  24 1238   13   54 *

Kernel (norm_base<0>).  Uses 40 registers.  n_h 4608  d_h 1024
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 1  235  6.5 24.3  2.0  8.0  16  805   24   40 *
 2  228  6.5 24.2  2.0  8.0  16  829  153   41 *
 4  209  6.5 25.2  2.0  8.0  18  905   12   45 *
 8  393  6.5 25.9  2.0  8.0   9  480    7   24 *
16  771  6.6 27.1  2.1  8.0   5  247    4   12
24 1166  6.6 28.0  2.4  8.0   3  168    4    8
32 1577  6.7 27.2  3.7  8.0   3  140    3    6
```

There appears to be two possible culprits: *premature writebacks*, seen in the `N*W` column, and *bank conflicts*, seen in the `BXW` column. At $d_l = 1024$ there are additional problems: not every thread has something to do (*workload imbalance*), and *cache pressure*.

Recent NVIDIA devices' L1 cache have what I'll call a *nearly write through* policy. In a *write through* cache, every write to a cache results in a write to the next layer (L2, say), even if the write hits the cache. The advantage is that one does not need to keep track of which lines are *dirty* (holding data different than the next layer or memory) or *clean* holding the same data as memory. (It would be more correct to use the term *sector* rather than line, but this description is long enough.) In a *write back* cache, when a store instruction hits the cache it writes only the cache line (making it dirty), the data is not propagated to the next level. Dirty cache lines will need to be written to the next layer eventually, usually when the line is evicted (kicked out because the space is needed). The NVIDIA L1 caches use a nearly write through policy. On a write hit the line becomes dirty. It will remain dirty for a short time, only a few clock cycles, then it will be written back. Suppose there is a second write to the same line shortly later. If that second write occurs after the line is written back, the line will need to be written back a second time. This situation is called a *premature writeback* here (it's not a standard term). If that second write occurs a very short time after the first the block is written back just once. Consider the `N*W` column for the $d_l = 4$ case. When there are fewer warps the amount of written data is ideal (the value in the column is 1.0). When there are more warps the SM is busier and so those second writes arrive too late, and so the amount of data leaving the L1 cache is higher than the ideal, reaching 3.1. The NVIDIA caches manage cache lines in units of *sectors*. In current devices a sector is 32 bytes. That's why the high value in the `N*W` column is 8 (a 32-byte sector holds 8 4-byte values).

Recall that bank conflicts occur when more than one thread in a warp attempts to load or store the

same bank. There are 32 banks, and the bank number required by a load or store is equal to bits 6:2 of the address. Because `elt_t` is four bytes, the bank number of access `l_in[ idx ]` is just `idx` modulo 32. For example `l_in[4]` needs bank 4, `l_in[31]` needs bank 31, `l_in[32]`, needs bank 0, and `l_in[36]` needs bank 4. If these were accessed by threads in the same warp (at the same time) there would be a bank conflict on bank 0. As a result the load would have to be issued twice (assuming there were no further bank conflicts). A best case is when the loads made by the 32 threads are `l_in[0]`, `l_in[1]`, `l_in[2]`, ..., `l_in[31]`. The worst case is an access like `l_in[0]`, `l_in[32]`, `l_in[64]`, `l_in[96]`, ..., `l_in[992]`. All of these threads access bank 0, and so the instruction would need to be issued 32 times (rather than once). If the data for all these threads were in the L1 cache the load would take a painful 32 times longer ($4 \times 32 = 128$ cycles on recent devices). But, if the load missed the cache then the issue time would still be 128 cycles, but the code might still need to wait 300 cycles or so for the data to arrive, reducing the impact of bank conflicts. The number in the `BXW` column shows the number of bank conflicts. The ideal value is 0, the worst case is 31.

Workload imbalance is not directly shown, but it can be inferred. For the $d_l = 1024$ case there are $n_l = 4608$ vectors, which works out to 36 vectors per SM on an RTX 4090. Consider the 4-warp case. The code (by default) will set the number of blocks to the number of SMs, 128 on the RTX 4090. Block 0 starts with vectors 0 to 127, block 1 starts with vectors 128-255, ..., block 35 operates on vectors 4480 to 4607, and blocks 36-128 have nothing to do. Launching with just one warp per block keeps all SMs busy on the first iteration of the `h` loop, but on the second iteration only the first few blocks will be busy.

Cache pressure can be seen in the `N*R` column. Note that each element of `l_in` is read twice, once when computing the sum, and a second time to compute `l_out`. We would like that second read to hit the L1 cache. For the $d_l = 4$ and $d_l = 8$ cases it looks like it does, since `N*R` is 1. But for $d_l = 1024$ it looks like each element of `l_in` is read at least twice. That's because the L1 cache isn't big enough to hold all the elements read when computing the sum, so they have to be read a second time. Note that the cache must be large enough to hold $d_l B$ elements, where B is the number of threads per block. Even for the one warp case ($B = 32$) the number of elements is 32768 occupying 128 kiB, larger than the L1 on a 4090.

Another factor which can affect performance is *instruction efficiency*. It doesn't in the base executions shown above, but it should be a factor in the solution. Instruction efficiency is the number of instructions per element (more generally per unit work) shown in column `I/el`. (Each vector has d_l elements, a run of the kernel operates on n_l vectors, for a total of $d_l n_l$ elements.) Smaller numbers are better. First, lets estimate a lower bound. The Sum Loop (see the comments in the code above) requires at least two instructions per element: a load instruction and an add instruction. (For one thing that assumes that the memory address of `l_in[h*d_l]` is computed before the loop starts, and that constant offsets are used in the unrolled loop.) Here is an excerpt for $d_l = 4$ showing only the load and add instructions (found in `hw01.sass`):

```
LDG.E.CONSTANT R6, [R2.64] ;
LDG.E.CONSTANT R8, [R2.64+0x4] ;
LDG.E.CONSTANT R10, [R2.64+0x8] ;
LDG.E.CONSTANT R12, [R2.64+0xc] ;
FADD R5, RZ, R6 ;
FADD R5, R5, R8 ;
FADD R5, R5, R10 ;
FADD R7, R5, R12 ;
```

The Norm Loop might need another load, a subtract, and a store. It turns out that for the $d_l = 4$ case, a second load is not needed. Also note that for the $d_l = 4$ case `sum/d_l` and `l_in[..] - avg` are implemented together using a multiply/add (`FFMA`) with `R7` holding the sum:

```
FFMA R9, R7.reuse, -0.25, R6 ;
FFMA R11, R7.reuse, -0.25, R8 ;
FFMA R13, R7.reuse, -0.25, R10 ;
FFMA R15, R7, -0.25, R12 ;
STG.E [R4.64], R9 ;
STG.E [R4.64+0x4], R11 ;
STG.E [R4.64+0x8], R13 ;
STG.E [R4.64+0xc], R15 ;
```

So for the $d_l = 4$ cases a lower bound is just four instructions per element: the `LDG`, `FADD`, `FFMA`, and `STG`. The best reported number for $d_l = 4$ is 5.4, accounting for other instructions before and after these loops, including the instructions to compute the addresses in `R2` and `R4`. Since the number of these other instructions shouldn't change with d_l, the reported `I/elt` number goes down with higher d_l. For $d_l = 32$ we reach 4.4, close to our lower bound.

The template parameter `D_L` is used for $d_l = 4$, $d_l = 8$, and $d_l = 32$, so the code will use a small number of instructions. (When d_l is specified as a template parameter the compiler will know its value and will perfectly unroll the loop and otherwise optimize the code. When the `D_L` template parameter is zero the code gets the value of d_l from `c_app.d_l`, a constant variable set at runtime, and so the compiler won't know d_l.)

For $d_l > 32$ the template parameter is set to zero. (That was my choice, there is no reason why it could not be set for $d_l = 128$, etc.) Regardless of whether the `D_L` template parameter was used for $d_l = 1024$, there would have to be two loads for each element of `l_in` because there are not enough registers to hold 1024 values. (The limit is 255 registers per thread.) This pushes the lower bound up to 5 instructions per element. The measured number of instructions per element is as low as 6.5 for $d_l = 1024$, which isn't bad.

Problem 1: Recall that the base kernel suffers from *premature writebacks*, *bank conflicts*, *workload imbalance*, and *cache pressure*. In this problem, all of these problems will be fixed! Though not perfectly, in part because instruction efficiency may be reduced (meaning `I/el` will be higher). The problems will be fixed by using a group of `grp_size` threads to normalize a single vector, rather than one thread as is done in `norm_base`. Call the value of `grp_size` the group size. Suppose that `grp_size=2` and `d_l=32`. Then we would like each of two threads to compute the sum of 16 elements of one 32-element vector. Each thread's sum will be added together (using pre-written routine `group_sum`), and finally each thread will normalize and write the output elements.

If this is done correctly the number of bank conflicts and premature writebacks will be reduced. In fact, when `grp_size=8` a correct solution should eliminate all premature writebacks, and when `grp_size=d_l` (for $d_l \leq 32$) there should be 0 bank conflicts. Increasing the group size should also reduce cache pressure and workload imbalance.

(*a*) In the unmodified assignment file `norm_group` is nearly identical to `norm_base`. Modify `norm_group` so that it uses a group size of `grp_size`, where `grp_size` is the second template parameter. (The first template parameter is `D_L` which is set to d_l if $d_l \leq 32$ or to zero if $d_l > 32$.) The value of `grp_size` will be a power of 2 from 1 to 32. In your solution take advantage of the fact that `d_l` will always be a power of 2 and `d_l` will be no larger than `grp_size`.

Use the provided routine `group_sum(VAL,GRP_SIZE)`, which returns the sum of `VAL` for all callers in the group based on the group size. The only difference between `norm_base` and `norm_group` in an unmodified assignment is that `group_sum` is called:

```
elt_t thd_sum = 0;
for ( int i = 0;  i < d_l;  i++ ) thd_sum += l_in[ h * d_l + i ];
const elt_t sum = group_sum(thd_sum,1);
const elt_t avg = sum / d_l;
```

In the code above the group-size argument in to `group_sum` is 1, so it just returns the `thd_sum` argument. In a correct solution the second argument would be changed from `1` to `grp_size`. Obviously other changes need to be made so that the vector elements are divided between the threads in a group.

Continued on next page.

When `hw01` is run `norm_group` is launched for each d_l size multiple times, each with a different group size and block size. The heading shown above the table of results shows the name of the kernel, including template parameters. It also shows the number of registers (which can be ignored for this assignment), the number of vectors (n_l), and the vector size (d_l). In the example below the group sizes are 2 (first table) and 4 (second table). The results are from a correct solution.

```
Kernel (norm_group<4,2>).  Uses 16 registers.  n_l 1179648  d_l 4
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 1   95 10.5  1.0  1.0  1.0   8  399   19  100 -**
 2   48 10.5  1.0  1.0  1.0  15  785   44  196 -****
 4   25 10.6  1.1  1.0  1.0  29 1483   92  371 --*******
 8   15 10.6  1.2  1.0  1.0  50 2603  151  650 +---**********
12   12 10.7  1.3  1.0  1.0  60 3088  218  770 +---**********
16   12 10.7  1.3  1.0  1.0  61 3151  184  785 +---**********
24   11 10.8  1.3  1.0  1.0  64 3310  107  824 +----**********
32   11 10.9  1.3  1.0  1.0  66 3410  181  852 +----**********

Kernel (norm_group<4,4>).  Uses 18 registers.  n_l 1179648  d_l 4
                  -----L2-Cache----- DRAM
wp t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 1  227 19.0  0.0  1.0  1.0   3  167   32   42 *
 2  115 19.0  0.0  1.0  1.0   6  329   63   82 **
 4   50 19.1  0.0  1.0  1.0  15  756   41  189 -****
 8   26 19.1  0.1  1.0  1.0  28 1442   95  360 --*******
12   19 19.2  0.1  1.0  1.0  38 1954  113  488 ---*********
16   15 19.2  0.2  1.0  1.0  49 2540  136  635 +---**********
24   13 19.4  0.2  1.0  1.0  58 2995  165  749 +---**********
32   12 19.5  0.2  1.0  1.0  61 3126  174  782 +---**********
```

In some cases the `norm_group` kernel will be launched with a group size of 1. The results in this case should be the same as `norm_base`, though they don't have to be.

(*b*) Find an expression for the amount of workload imbalance expected on an RTX 4090 based on the number of vectors (n_l), vector sizes (d_l), and group sizes (g or `grp_size`), and block sizes (B or `32 * wp`). Try to write a formula, for example where a value of less than 1 indicates workload imbalance.