LSU EE 7722 **Homework 4** **Due: 24 April 2025**

Formatted 17:51, 18 April 2025

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/2025/hw04`.

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 `.../2025/hw04` 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 `.../2025/hw04` 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 `hw04`, one with the suffix `-debug`, such as `hw04-debug`, and one with the suffix `-cuda-debug`, such as `hw04-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 and so CUDA code cannot easily debugged with these executable files. Use `gdb` or `cuda-gdb` to debug `hw04-debug`.

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 `./hw04`. It should produce output ending with a line something like this `32 32 0.28  20 9.5 0.2 2.0 54 2791 464 1 ---*********`.

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.*, 8.9) or by name (*e.g.*, Ada Lovelace). CC 8.9 GPUs implement Ada Lovelace architecture, and 9.0 implements Hopper. For this assignment only consider CC 8.9 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 `hw04`

The code in `hw04.cu` contains several kernels that normalize vectors. The `hw04` program takes up to three command-line arguments: `./hw04 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 `./hw04 0 32`, that runs the kernels fewer times (explained below). To run it while working on your solution usually run it as `./hw04`.

The first argument is used to specify the number of blocks. When there are zero arguments, `./hw04`, or when the first argument is zero, `./hw04 0`, the number of blocks is set equal to the number of SMs. When the first argument is a positive integer, such as `./hw04 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 `./hw04 -5`, then each kernel will be launched with that many blocks *per SM*. For a GPU with 40 SMs and running with `./hw04 -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 active 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 until enough active blocks finish).

The second argument specifies the number of threads per block. A positive value indicates the exact number of threads, for example, `./hw04 -3 64`, will run each kernel with a block size of 64 threads (4 warps) 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, `./hw04`. The program will launch each kernel multiple times, starting with 4 warps 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, `./hw04 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, `./hw04 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 4 2025) 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 without command-line arguments:

```
[koppel@grace hw04]$ ./hw04
```

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 24078 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: 102400 B/SM   CONST:  65536 B      NUM REGS:  65536
GPU 0: SHARED:  49152 B/BL  SH RES:   1024 B/BL  SH OPT-IN: 101376 B/BL
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). `SP-F32/SM` indicates the number of 32-bit floating point functional units (once called *CUDA cores*) per SM, `DP-FP64/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 in several places. The amount of shared memory on an SM is indicated by `B/SM`. Whether a block can access that much memory depends on several factors.

The storage indicated by `B/SM` can be used for both shared memory and the L1 cache. The L1 cache size is usually the same size or a bit larger than the shared memory size.

The following entries show the amount of constant address space, `65536 B`, and the number of registers on an SM, `65536`. These numbers do not indicate whether any particular kernel or block is using that much storage.

The second `SHARED` line shows shared memory per block. The first entry, `49152 B/BL`, is the maximum amount of shared memory accessible with the `__shared__` qualifier, called statically allocated shared memory. The last entry, `SH OPT-IN` is the amount that could be accessed using the `extern` qualifier, in which the amount of shared memory is specified by a kernel launch parameter. This assignment (2025 Homework 4) does not need to use much shared memory, and so statically allocated shared memory is fine.

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 bandwidth is shown for both the single-precision (FP32) and double-precision (FP64) functional units. The floating-point bandwidth of the tensor cores is higher, though precision is lower. 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 kernel `norm_group`.

```
(norm_group<0,32>):  Uses 40 registers.  L1 Avail 101376 B (gp=32)
n_l 2304  d_l 2048  L2 Occ 25% + 25%
                          --L2-Cache---
wp gp  Imm t/s I/el  BXW  N*R %pk GB/s FP  BG === Util: FP++  Insn--  L2** ===
 1  1 0.94   35  8.9  0.1  1.0  21 1085  267 == -******
 2  1 0.93   20  8.9  0.1  1.1  37 1897  462 == ---********
 4  1 0.84   16  9.0  0.1  1.4  55 2827  595 == ---**********
 8  1 0.82   16  9.2  0.1  1.5  59 3061  601  2 ---**********
16  1 0.76   18  9.6  0.1  1.9  60 3115  537  4 ---**********
32  1 0.45   18 10.2  0.1  2.0  60 3077  511  8 ---**********

 The lines below are fictional and are there to explain the bar graph.
32  1 0.45   18 10.2  0.1  2.0  60 3077  511  8 +++++++++++++++++++++++++++++++
32  1 0.45   18 10.2  0.1  2.0  60 3077  511  8 +++++++++++++++++++++++++++++++++
```

The output above shows the result a kernel, `norm_group<0,32>`. (The name shown is how the function was named, including the template parameters.)

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.)

The `Imm` *column* shows workload balance. A value of 1 is ideal and indicates that all blocks finished at the same time. A value of 0.3 indicates that the average block run time is 0.3 times the maximum run time (which is bad).

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 `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. If visible, *the* `DRAM GB/s` *column* shows the measured data throughput between the L2 cache and off-chip memory. For the 2025 Homework 4 assignment the `DRAM` column is only visible if the input does not comfortably fit in the L2 cache.

The `t/`μ`s` *column* shows the measured execution time in microseconds. *The* `FP` θ *column* shows floating point throughput in GFLOPs. *To the right of* `FP` θ 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 in some situations including the 2025 Homework 4 assignment). Those last two lines are fictional. Consider the line for the 8 warp per SM run. The `*` is a about 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 $d_l \in 1024, 2048, 4096$ 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).

2024 Base Code Performance

The description below is for the 2024 assignment. In the 2025 assignment `norm_group` is complete. 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 `hw04.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.

2025 Assignment Background

In the 2024 assignment group size was limited to no more than 32 threads (one warp). In this assignment (2025) group size will be set to multiples of 32. A run of the assignment code runs three kernels, `norm_base` and `norm_group` from the solution to the 2024 assignment and `norm_hw04` which is new for this assignment. Kernel `norm_group` is run only with a group size of 32. Kernel `norm_hw04` is run on group sizes from 32 (1 warp) up to 1024 (32 warps). These kernels will be run with input vectors of size 1024, 2048, and 4096 elements. These vectors are long enough to risk level-1 cache misses when making a second access to the vector. Something that can be fixed with group sizes larger than 32.

(The number of vectors is chosen so that the entire input fits in the level 2 cache when the code is run with the default parameters. Because of this any miss experienced by a kernel can be satisfied by data in the level 2 cache. If the inputs did not fit in the level 2 cache, for example if the code were run using command `hw04 0 0 -2`, then the performance benefits of properly choosing the group size would be much smaller.)

The `norm_hw04` kernel is run with each possible group size and block size. Consider output from a correctly solved assignment:

```
(norm_hw04<0,32>):  Uses 40 registers.  L1 Avail 101248 B (gp=64)
n_l 2304  d_l 2048  L2 Occ 25% + 25%
                          --L2-Cache---
wp gp  Imm t/s I/el  BXW  N*R %pk GB/s FP  BG === Util: FP++  Insn--  L2** ===
 1  1 0.94   41  9.3  0.0  1.1  18  950  231 == -*****
 2  1 0.94   26  9.4  0.0  1.2  31 1591  366 == --*******
 2  2 0.96   26 11.0  0.0  1.1  29 1486  361  1 --*******
 4  1 0.84   19  9.5  0.1  1.4  47 2433  505 == ---**********
 4  2 0.94   17 11.1  0.1  1.2  46 2384  552  1 ---**********
 4  4 0.97   19 13.9  0.1  1.1  40 2071  503  1 ---*********

 8  1 0.79   17  9.7  0.0  1.6  54 2770  539  2 ---**********
 8  2 0.89   15 11.3  0.1  1.1  52 2699  627 == ---**************
 8  4 0.95   15 14.0  0.1  1.1  50 2558  622  2 ---*************
 8  8 0.93   15 19.7  0.1  1.0  48 2474  616  2 ---*************

16  1 0.76   19 10.1  0.1  1.9  57 2922  497  4 ---*********
16  2 0.84   15 11.8  0.1  1.5  61 3151  623  4 ---*************
16  4 0.91   14 14.5  0.1  1.1  54 2784  673 == +---**************
16  8 0.94   13 20.0  0.1  1.0  56 2893  718  4 +---***************
16 16 0.94   16 30.2  0.0  1.0  45 2337  583  4 ---***********

32  1 0.46   19 10.7  0.1  2.0  59 3060  508  8 ---**********
32  2 0.76   20 12.7  0.1  1.9  55 2854  484  8 ---*********
32  4 0.83   15 15.4  0.1  1.4  60 3090  630  8 ---**************
32  8 0.90   14 21.0  0.1  1.1  53 2712  654 == +---*************
32 16 0.94   15 31.0  0.1  1.0  49 2528  629  8 ---**************
32 32 0.97   18 51.1  0.1  1.0  40 2069  514  8 ---**********
```

In addition to the columns described in the background sections, this assignment includes two assignment-specific columns: `gp` and `BG`. Column `gp` shows the group size, measured in warps. Column `BG` shows a guess of what the group size should be (Problem 2) **for the particular block size**. A value of `==` in the `BG` column means that the kernel is being run with the group size returned by `hw04_group_size_choose`. Notice that the `BG` column in the excerpt above is not always correct, such as for a block size of 16 warps.

One parameter of `hw04_group_size_choose` is the amount of level 1 cache space available. The amount of space varies by kernel. The end of the first line in the excerpt above reads `L1 Avail 101248 B (gp=64)`. That shows the amount of L1 space available to the kernel that has the least L1 space available. The amount of L1 space available to a kernel depends on how much shared memory is declared. The more memory declared, such as `__shared__ int my_array[4096];`, the less L1 cache space available.

In the runs above all possible group sizes are tried for each block size. Notice that as group size increases

instruction overhead, shown in `I/el`, increases (a bad thing). However, as group size increases, the number of times an element is transferred from the L2 cache to the L1 cache, shown in `N*R`, decreases, a good thing. This means one must choose a group size that's large enough so that the vectors fit in the cache, but no larger to avoid inflating instruction overhead. That's the goal of Problem 2.

Problem 1: In the unmodified code kernel `norm_hw04` works correctly when template parameter `GRP_SIZE` is a power of two from 1 to 32. Larger values of `GRP_SIZE` won't cause an error, but the value will be ignored and instead a size of 32 will be used. That is, the kernel will use the smaller of `GRP_SIZE` or 32 threads to normalize each vector. Modify `norm_hw04` so that it works correctly when template parameter `GRP_SIZE` is set to a value ≥ 32. The value will still be a power of 2 and will not be larger than the block size. The kernel *does not have to* work for `GRP_SIZE` values less than 32.

Note that routine `group_sum` won't work when parameter `group_size` is greater than 32. Solve this problem using shared memory. Do not rely on library functions to compute the sum.

In a correct solution the fastest `norm_hw04` runs should be faster than the fastest `norm_group` runs, especially at larger vector sizes.

Problem 2: A larger group size has two benefits. It reduces workload imbalance when the number of vectors is small. It also can reduce the number of level 1 (L1) cache misses. The disadvantage of larger groups sizes is that it increases overhead.

Recall that the normalization code reads a vector twice. Once when finding the sum of its elements, and a second time when those elements are normalized. Cache misses are reduced because as group size is increased the number of vectors being normalized by a block at any particular time drops. For a large enough group size all vectors will fit in the L1 cache.

Modify routine `hw04_group_size_choose` so that it returns a good group size (number of threads) to use for the given L1 cache size, block size, and vector length. The returned group size must be a power of two and can't be smaller than 32, nor larger than the block size, nor larger than the vector length. It is okay if a tuning constant (informally, fudge factor) is used to account for the fact that data other than the input and output vectors might occupy the cache, and other factors.

For your convenience, the unmodified code includes a sample use of C++20 standard library function `bit_ceil`, which rounds an unsigned integer up to a power of two. For example, `bit_ceil(12)` returns 16, `bit_ceil(16)` also returns 16, and `bit_ceil(17)` returns 32. (There is also a `bit_floor` function.)