LSU EE 7722 **Homework 2** **Due: 12 April 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/hw02`.

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/hw02` 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/hw02` 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 `hw02`, one with the suffix `-debug`, such as `hw02-debug`, and one with the suffix `-cuda-debug`, such as `hw02-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 `./hw02 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 MP (or SM) 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 `hw02`

The code in `hw02.cu` contains several kernels that normalize vectors. The `hw02` program takes up to three

command-line arguments: `./hw02 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 `./hw02 0 32`, that runs the kernels fewer times (explained below). To run it while working on your solution usually run it as `./hw02`.

The first argument is used to specify the number of blocks. When there are zero arguments, `./hw02`, or when the first argument is zero, `./hw02 0`, the number of blocks is set equal to the number of SMs. When the first argument is a positive integer, such as `./hw02 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 `./hw02 -5`, then each kernel will be launched with that many blocks *per SM*. For a GPU with 40 SMs and running with `./hw02 -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, `./hw02 -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, `./hw02`. The program will launch each kernel multiple times, starting with 1 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, `./hw02 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, `./hw02 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 hw02]$ ./hw02 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 MP, in successive runs increasing the number of warps per MP. 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 norm_base.

```
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 SM/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.

Problem 1: One disappointment in the solution to Homework 1 is that when the group size is equal to the vector length many warps are required to attain good performance. For example, consider:

```
Kernel norm_group.  18 regs. D_L 8, Group sz 8,  Unroll 0,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/μs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   55 21.1  0.0  1.0  1.0  13  692   54  173 -***
 8 1.00   29 21.1  0.1  1.0  1.0  25 1308  103  327 --******
12 1.00   21 21.2  0.1  1.0  1.0  35 1823  140  456 ---*********
16 1.00   16 21.2  0.2  1.0  1.0  46 2351  183  588 ---************
24 1.00   13 21.4  0.2  1.0  1.0  56 2904  225  726 +---***************
32 1.00   12 21.5  0.3  1.0  1.0  60 3099  243  775 +---****************
```

The problem is that when the group size is the same as the vector size (8 in the example above) the `i` loops have just one iteration:

```
  for ( int h = h_start;  h < n_l;  h += n_threads / grp_size ) {
      const size_t idx_vec_start = h * d_l;
      const size_t idx_vec_thd_start = idx_vec_start + sub_lane;

      elt_t thd_sum = 0;
      for ( int i = 0;  i < d_l;  i += grp_size )
        thd_sum += l_in[ idx_vec_thd_start + i ];

      const elt_t sum = group_sum(thd_sum,grp_size);
      const elt_t avg = sum / d_l;

      for ( int i = 0;  i < d_l;  i += grp_size )
        l_out[ idx_vec_thd_start + i ] = l_in[ idx_vec_thd_start + i ] - avg;
    }
```

When the number of iterations of the `i` loops is large there are two advantages. Let I denote the number of iterations of the `i` loops. When I is large the time needed for the code outside the `i` loops, such as computing `sum/d_l`, becomes small in comparison to the time for the `i` loops. (Put another way, the work needed to execute the code outside the `i` loop is done d_l/I times, so a larger value of I reduces the amount of work.) So, for work efficiency we want the group size to be 1, yielding the maximum $I = d_l$.

Recall (from class material) that for code like the `i` loops the loads will be issued first, then the computation will follow. So for the first loop if $I = 4$ the compiler will emit four loads, then perform the addition afterward:

```
// hw01-sol.cu:136         thd_sum += l_in[ idx_vec_thd_start + i ];
        /*02d0*/                   LDG.E.CONSTANT R4, [R2.64+-0x8] ;
        /*02e0*/                   LDG.E.CONSTANT R8, [R2.64+-0x4] ;
        /*02f0*/                   LDG.E.CONSTANT R10, [R2.64] ;
        /*0300*/                   LDG.E.CONSTANT R12, [R2.64+0x4] ;
// hw01-sol.cu:136         thd_sum += l_in[ idx_vec_thd_start + i ];
        /*0370*/                   FADD R5, RZ, R4 ;
        /*0380*/                   FADD R5, R5, R8 ;
        /*0390*/                   FADD R5, R5, R10 ;
        /*03a0*/                   FADD R5, R5, R12 ;
```

Nvidia GPUs execute `LDG` instructions (load global memory) by performing a lookup in the L1 cache, and if the data is not there issuing a request. Execution will proceed to the next instruction without waiting

for the data to arrive. Execution only stalls (waits) for the loaded value when an attempt is made to use it. In the code above the `LDG` instructions will execute quickly (it will take four cycles per load in recent devices) whether or not the data is found in the cache. The first instruction to use a loaded value is the first `FADD` instruction, using `R4` as a source, the same `R4` as written by the first `LDG`. If the data has not yet arrived in `R4` then execution will stall until the data arrives. When the data arrives the first `FADD` executes and execution moves to the second `FADD`, which references the data loaded by the second `LDG` carried by `R8`. Since the second load started four cycles after the first, the second `FADD` only need wait a few cycles, not the entire L1 cache miss latency.

Let n denote the total number of loads that need to be executed by a thread, and let I denote how many of those are overlapped (four in the example above), and let L_M denote load latency (say, L1 miss /L2 hit latency). Then the time needed for the loads (before data throughput approaches saturation) is $\frac{n}{I}L_M$. Clearly, a larger I is better.

In Homework 1 the `norm_group` kernel was written to *reduce* I, thus *reducing* the benefits described above. Recall that in doing so other problems were reduced, including premature writebacks, bank conflicts, and cache pressure.

In this assignment the goal is to recover some of the benefits of larger I by properly unrolling the `h` loop. The standard way of unrolling a loop by *degree* d is to make d copies of the loop body and then simplify the d copies. The compiler itself won't do a good job unrolling the `h` loop for two reasons. First, it will assume that something done after the `group_sum` in the original code cannot be moved before `group_sum`. That will make it impossible to overlap loads any more than they already are. This first problem is partially solved by unrolling the `h` loop body in three part: where `thd_sum` is computed, where `group_sum` is called, and finally where the output elements are written, each unrolled part is in a `j` loop:

```
  for ( int hi = h_start;  hi < n_l;  hi += h_inc ) {
      elt_t thd_sum[unroll_degree]{};

      for ( int j = 0;  j < unroll_degree;  j++ )
        {
          int h = hi;  // Homework 2: Compute h based on unroll_degree.
          const size_t idx_vec_start = h * d_l;
          const size_t idx_vec_thd_start = idx_vec_start + sub_lane;

          for ( int i = 0;  i < d_l;  i += grp_size )
            thd_sum[j] += l_in[ idx_vec_thd_start + i ];
        }

      elt_t avg[unroll_degree];
      for ( int j = 0;  j < unroll_degree;  j++ ) {
          elt_t sum = group_sum(thd_sum[j],grp_size);
          avg[j] = sum / d_l;
        }

      for ( int j = 0;  j < unroll_degree;  j++ ) {
          int h = hi;  // Homework 2: Compute h based on unroll_degree.
          const size_t idx_vec_start = h * d_l, idx_vec_thd_start = idx_vec_start + sub_lane;

          for ( int i = 0;  i < d_l;  i += grp_size )
            l_out[ idx_vec_thd_start + i ] = l_in[ idx_vec_thd_start + i ] - avg[j];
        }
    }
}
```

In the code above the first `j` loop will load `unroll_degree` sets of `l_in` values, and the compiler should be able to put all of those loads before the `FADD` instructions (because there is no intervening `group_sum`).

Note that in the code above `h` is not computed properly, so though the code has the form of an unrolled loop it will actually just compute the same output vector `unroll_loop` times.

If the compiler were to unroll the `h` loop it would the `h` values from `unroll_degree` consecutive iterations of the original loop in the unrolled loop. For example, suppose `n_threads / grp_size` is 1024, `unroll_degree` is 4, and `h_start` is 20000. Then the values of `h` chosen by the compiler will be 20000, 21024, 22048, and 23072. Those values are not wrong but they are not efficient for several reasons.

First, because `n_threads` is not a compile-time constant the difference between these `h` values is not a compile time constant and the compiler will have to emit arithmetic instructions to compute the address of each access to `l_in` and `l_out`. If `n_threads/grp_size` were a compile-time constant the compiler would emit instructions to compute just one address and use constant offsets.

A bigger problem with the unrolling described above is that it would bring back the problem of bank conflicts.

(*a*) Kernel `norm_group_u` has starter code for loop unrolling to solve the problem described above. However, as currently written the code computes the correct results but takes about `unroll_degree` times as many instructions to do so. That's because the `hi` loop iterates the same way as the `h` loop in the Homework 2 solution (appearing as `norm_group` in this assignment), despite the fact that the `j` loops do `unroll_degree` times more work.

So for this assignment modify the `hi` loop in `norm_group_u` so that it iterates `1/unroll_degree` fewer times and also compute `h` correctly in the `j` loops.

- Avoid increasing the number of bank conflicts. That is, the number of bank conflicts for `norm_group_u` should not be higher than that of `norm_group` for the same `d_l` and group size.

- Compute `h` so that the difference between the values of `h` in the `j` loop is a compile-time constant. (This can be verified by examining the code in the SASS files.)

Here is the output when correctly solved.

```
Kernel norm_group.  18 regs. D_L 8, Group sz 8,  Unroll 0,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   54 21.1  0.0  1.0  1.0  13  695   54  174 -***
 8 1.00   29 21.1  0.1  1.0  1.0  25 1310  102  327 --******
12 1.00   21 21.2  0.1  1.0  1.0  35 1792  138  448 ---********
16 1.00   16 21.2  0.2  1.0  1.0  46 2372  184  593 ---************
24 1.00   13 21.4  0.3  1.0  1.0  55 2861  224  715 +---**************
32 1.00   12 21.5  0.3  1.0  1.0  60 3121  243  780 +---****************

Kernel norm_group_u.  18 regs. D_L 8, Group sz 8,  Unroll 1,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   55 23.1  0.0  1.0  1.0  13  681   55  170 -***
 8 1.00   29 23.1  0.1  1.0  1.0  25 1298  101  325 --******
12 1.00   21 23.2  0.1  1.0  1.0  34 1770  136  442 --*********
16 1.00   16 23.2  0.2  1.0  1.0  45 2316  180  579 ---************
24 1.00   13 23.3  0.3  1.0  1.0  56 2915  226  729 +---***************
32 1.00   12 23.4  0.3  1.0  1.0  61 3130  242  783 +---****************

Kernel norm_group_u.  20 regs. D_L 8, Group sz 8,  Unroll 2,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   32 16.6  0.0  1.0  1.0  23 1191   93  298 --******
 8 1.00   17 16.6  0.2  1.0  1.0  42 2175  173  544 ---***********
12 1.00   13 16.7  0.2  1.0  1.0  54 2808  215  702 +---**************
16 1.00   12 16.7  0.3  1.0  1.0  62 3181  255  795 +---*****************
24 1.00   12 16.8  0.3  1.0  1.0  63 3249  251  812 +-----*****************
32 1.00   11 16.9  0.3  1.0  1.0  64 3312  269  828 +-----*****************

Kernel norm_group_u.  26 regs. D_L 8, Group sz 8,  Unroll 4,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   18 13.3  0.1  1.0  1.0  40 2048  159  512 ---**********
 8 1.00   12 13.4  0.2  1.0  1.0  59 3037  246  759 +---***************
12 1.00   12 13.4  0.2  1.0  1.0  63 3243  250  811 +-----*****************
16 1.00   12 13.5  0.3  1.0  1.0  63 3240  263  810 +-----*****************
24 1.00   11 13.6  0.3  1.0  1.0  65 3343  258  836 +-----*****************
32 1.00   11 13.7  0.3  1.0  1.0  66 3405  266  851 +-----******************
```

In the output pay attention to the number of instructions per iteration (`I/el`) and the number of bank conflicts `BXW`. Output from the unsolved version appears on the next page.

In the unsolved version:

```
Kernel norm_group.  18 regs. D_L 8, Group sz 8,  Unroll 0,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   54 21.1  0.0  1.0  1.0  13  693   54  173 -***
 8 1.00   29 21.1  0.1  1.0  1.0  26 1321  103  330 --******
12 1.00   21 21.2  0.1  1.0  1.0  35 1813  139  453 ---*********
16 1.00   16 21.2  0.2  1.0  1.0  46 2354  182  588 ---************
24 1.00   13 21.4  0.2  1.0  1.0  56 2895  223  724 +---***************
32 1.00   12 21.5  0.3  1.0  1.0  61 3131  241  783 +---****************

Kernel norm_group_u.  18 regs. D_L 8, Group sz 8,  Unroll 1,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   54 21.1  0.0  1.0  1.0  13  695   54  174 -***
 8 1.00   28 21.1  0.1  1.0  1.0  26 1325  103  331 --******
12 1.00   21 21.2  0.1  1.0  1.0  35 1814  140  453 ---*********
16 1.00   16 21.2  0.2  1.0  1.0  46 2373  185  593 ---************
24 1.00   13 21.4  0.2  1.0  1.0  55 2858  220  714 +---**************
32 1.00   12 21.5  0.3  1.0  1.0  60 3121  242  780 +---****************

Kernel norm_group_u.  18 regs. D_L 8, Group sz 8,  Unroll 2,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   69 31.1  0.0  1.0  1.0  11  544   44  136 -**
 8 1.00   36 31.1  0.1  1.0  1.0  20 1055   82  264 -******
12 1.00   26 31.2  0.1  1.0  1.0  28 1461  115  365 --*******
16 1.00   20 31.2  0.2  1.0  1.0  37 1927  150  482 ---*********
24 1.00   15 31.4  0.3  1.0  1.0  49 2531  199  633 +---************
32 1.00   13 31.5  0.4  1.0  1.0  58 3015  236  754 +---***************

Kernel norm_group_u.  16 regs. D_L 8, Group sz 8,  Unroll 4,  n_l 589824  d_l 8
                      -----L2-Cache----- DRAM
wp  Utl t/µs I/el  BXW  N*R  N*W %pk GB/s GB/s FP θ === Util: FP++  Insn--  L2** ====
 4 1.00   98 49.1  0.0  1.0  1.0   7  385   30   96 -*
 8 1.00   51 49.1  0.1  1.0  1.0  14  743   58  186 -****
12 1.00   36 49.2  0.1  1.0  1.0  21 1063   82  266 -******
16 1.00   27 49.2  0.2  1.0  1.0  27 1394  109  348 --*******
24 1.00   20 49.3  0.3  1.0  1.0  37 1912  148  478 ---*********
32 1.00   16 49.4  0.5  1.0  1.0  46 2357  190  589 ---************
```