LSU EE 7722 **Homework 3** **Due: 14 April 2025**

Solution *Formatted 10:54, 2 May 2025*

Basic Setup

Follow the instructions for class account setup on `https://www.ece.lsu.edu/gp/proc.html`.
This assignment does not require writing or running code, but doing so can be helpful. Code for this assignment is in directories `../hw/gpm/2024/hw01` and `../hw/gpm/2024/hw02`. Note that those are 2024 directories and that they have the solution to the 2024 assignments (not to the questions asked here). For further instructions on how to run the 2024 Homework 1 and Homework 2 code see the Homework 1 assignment and Homework 2 assignment.

Background

This assignment is based on the solutions to 2024 Homework 1 and 2024 Homework 2. The subject of those assignments was a kernel that normalized vectors so that the average element value in a normalized vector is zero. (That's not the same a normalizing a vector so that its length is one.) The solution to those assignments is available (as of this writing in the repo only). To complete this 2025 assignment one needs to understand those solutions. That said, it might be helpful to attempt to solve those past assignments yourself.

In 2024 Homework 1 a modified kernel was to be written that lessens the impact of four issues that hurt performance: premature writebacks, bank conflicts, workload imbalance, and cache pressure. These are eliminated or reduced writing a kernel in which several threads normalize a vector, rather than just one. The number of threads normalizing a vector is called the *group size* and template parameter `grp_size` is set to its value. A disadvantage of increasing the group size is a drop in instruction efficiency (a fifth issue that affects performance). See 2024 Homework 1 for a detailed discussion of these five issues.

In 2024 Homework 2 the problem of reduced instruction efficiency and access latency is to be addressed by applying loop unrolling. That's not as simple as telling the compiler to unroll the loop by placing an `unroll` pragma before the loop. As explained in the problem, the compiler won't do a good job. The solution is to unroll a loop in such a way that instruction efficiency is improved and latency can be hidden.

The solutions to these two problems were the subject of 2024 Final Exam Problem 2. This 2025 assignment is a recycled version of that problem.

Problem 1: The following question originally appeared as 2024 Final Exam Problem 2. In the original exam the multiple choice parts of (b) through (e) were part of the question. Here the correct choice is given, and all one needs to do is explain the answer. Appearing below is the solution to 2024 Homework 2.

```
template<int D_L = 0, int grp_size = 1, int unroll_degree = 1>
__global__ void norm_group_u(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, n_l = c_app.n_l;
  const int sub_lane = threadIdx.x % grp_size;
  constexpr int wp_lg = 5, wp_sz = 1 << wp_lg;
  const int lane  = threadIdx.x % wp_sz, wp_id = tid >> wp_lg;

  constexpr int vec_p_wp = wp_sz / grp_size;
  const int vec_p_wp_iter = vec_p_wp * unroll_degree;
  const int h_wp_start = wp_id * vec_p_wp_iter,   inc = unroll_degree * n_threads / grp_size;

  for ( int h_wp = h_wp_start;  h_wp < n_l;  h_wp += inc ) {
      const int hi = h_wp + lane / grp_size;
      elt_t thd_sum[unroll_degree]{};

      for ( int j = 0;  j < unroll_degree;  j++ )
        {
          const int h = hi + j * vec_p_wp;
          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 sum[unroll_degree]{};
      for ( int j = 0;  j < unroll_degree;  j++ ) sum[j] = group_sum(thd_sum[j],grp_size);

      for ( int j = 0;  j < unroll_degree;  j++ )
        {
          const int h = hi + j * vec_p_wp;
          const size_t idx_vec_start = h * d_l;
          const size_t idx_vec_thd_start = idx_vec_start + sub_lane;
          const elt_t avg = sum[j] / 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;
        }
    }}
```

(*a*) What is the minimum L1 cache size for which the `l_in` access in the Normalization Loop hits the L1 cache? Answer in terms of δ (unroll degree), d_l (vector length), g (group size), and B (number of threads per block). Assume that the number of blocks equals the number of SMs.

☑ In terms of δ, d_l, g, and B, the minimum cache size is:

The minimum cache space needed is $\delta \frac{d_l}{g} B$ elements. If the element size is four bytes, $4\delta \frac{d_l}{g} B$ B where B is the number of threads per block and B is the symbol for byte.

To help answering the parts below it might be helpful to look at the sample output on the last pages of 2024 Homework 2.

To answer the questions below one must understand how memory and L2 cache access latency can be hidden, and the difference between throughput and latency.

Let $L_2 = 120\,\mu$s denote level 2 (L2) cache access latency (suffered when there is an L1 miss and L2 cache hit). Suppose an individual thread performs a bunch of x accesses to `l_in`. A naive execution time estimate would be xL_2. Naive because of the assumption that the loads in the bunch are performed one at a time, meaning that load $i+1$ does not start until the data from load i arrives. As we've seen in class the compiler will emit the load instructions next to each other and the hardware will send requests for multiple loads without waiting for any of those requests to return data. For the first set of accesses to `l_in` (the first `j-i` loop nest) we would expect the compiler to emit $\delta\frac{d_l}{g}$ loads (per thread). Counting all B threads $x = \delta\frac{d_l}{g}B$ loads will be emitted at the beginning of an `h_wp` iteration. Let Θ_2 denote SM-to-L2 cache bandwidth per SM. When $x/\Theta_2 < L_2$ threads are waiting for data to arrive, and so decreasing x, either by decreasing B or δ will not have a large impact on execution time in our code because once the data arrives there is not much computation to do. But when $x\Theta_2 > L_2$ increasing x will substantially increase execution time. So to answer the questions below one must take into account the number of bunched loads.

(*b*) Consider two configurations of `norm_group_u`. Configuration X is launched with $B \geq 256$ threads per block, and an unroll degree of δ. Configuration Y is launched with $B/2$ threads per block and an unroll degree of 2δ. For both configurations n_l **is large**, $d_l = g = 32$, and the number blocks is equal to the number of SMs.

☑ Might Y run ◯ *at least* 1.5× *faster than X*, ◯ *at least* 1.5× *slower than X*, ⊗ *or at about the same speed as X.* ☑ Explain your answer based on your understanding of how GPUs work. ☑ Indicate whether the data included in the 2024 Homework 2 handout is consistent with the answer or does something different.

About the same speed. That's because the number of roughly simultaneous loads, sometimes called memory-level parallelism (MLP) is the same in X and Y. In Y there are half the number of threads, but because δ is doubled each thread will be issuing twice the number of loads. So overall, the MLP is about the same. In X the B threads in a block will each issue δ loads, for a total of δB, and then wait L_2 units of time for the data. In Y the $B/2$ threads will each issue 2δ loads, for (the same) total of δB. In recent Nvidia devices a warp scheduler has $\frac{1}{4}\frac{B}{32}$ warps to choose from. In X there are twice the number of warps but in Y each warp has twice the work (and there are enough for all four warp schedulers).

Configuration Y has a slight advantage because its unroll degree is twice as high. That means the overhead of processing the beginning of an `h_wp` loop is amortized over twice the number of `j` loop iterations (2δ in Y, δ in X).

The data tabulated in the 2024 Homework 2 solution are consistent with the overall conclusion that execution time is about the same. However by the reasoning above a larger unroll degree should give Y a slight advantage but that's not seen in the data. Yes, instruction overhead is lower (the values under `I/el`), but the 32-warp, $\delta = 1$ configuration has a FP throughput of 745 GFLOP/s while the 16-warp $\delta = 2$ configuration executes a bit more slowly, at 726 GFLOP/s.

(*c*) Consider a system with 114 SMs. In configuration X $B = 256$, $d_l = g = 32$, $\delta = 1$, and $n_l = 912$. Configuration Y is the same except $B = 512$. In both cases 114 blocks are launched. (The unroll degree is **not** changed.).

☑ Will Y run ◯ *faster than X*, ◯ *slower than X*, ⊗ *or at about the same speed as X.* ☑ Explain your answer based on your understanding of how GPUs work. ☑ Don't forget to account for the value of n_l, which is not large in this part.

In X each thread executes just one `h_wp` loop iteration (because $\frac{n_l}{114B/32} = \frac{912}{114\times256/32} = 1$). In Y B is doubled and so half the threads have no work. Because of the way work is divided threads in the first $\frac{114}{2} = 57$ blocks will execute one iteration, while threads in the other blocks will have nothing to do. An individual SM running Y will either have twice as much work as it would in X or none at all. Since $d_l = g$ and $\delta = 1$ there will be just one load for each `h_wp` iteration per thread. There is not much other work to do, and so the iteration time will be dominated by L2 access latency, L_2. Doubling the number of threads will not change the latency of an individual thread. Though the data transfer needed from an individual SM doubles (or is zero), the data transfer needed off-chip is the same in X and Y. So, execution time is about the same.

(*d*) Consider a system with 114 SMs. In configuration X $B = 256$, $d_l = g = 32$, $\delta = 1$, and $n_l = 912$. Configuration Y is the same except $\delta = 2$. (The block size is **not** changed.).

☑ Given the way the homework code performed, will Y run ◯ *faster than X*, ◯ *slower than X*, ⊗ *or at about the same speed as X.* ☑ Explain your answer based on your understanding of how GPUs work. ☑ Don't forget to account for the value of n_l, which is not large in this part.

About the same for similar reasons to the previous part. Here the number of threads is the same but because δ is doubled there is twice as much work assigned to a thread, (going from 1 to 2 `h_wp` loop iterations). For similar reasons half the SMs will be idle. And for similar reasons, an `h_wp` iteration is still dominated by a single L_2, though in Y two loads are issued.

One important thing to notice is that the compiler can issue both `j`-loop loads before writing them to `thd_sum`. Since `d_l` is a compile-time constant the compiler should figure out that there's no arithmetic to do. And even if there were (the part below) the compiler would know enough to issue all the loads before adding up the loaded values.

(*e*) Consider a system with 114 SMs. In configuration X $B = 256$, $d_l = 1024$, $g = 32$, $\delta = 1$, and $n_l = 912$. Configuration Y is the same except $\delta = 2$. (The block size is **not** changed.).

☑ Given the way the homework code performed, will Y run ◯ *faster than X*, ⊗ *slower than X*, ◯ *or at about the same speed as X.* ☑ Explain your answer based on your understanding of how GPUs work. ☑ Don't forget to account for the value of n_l, which is not large in this part.

In this case there are 32 `i`-loop iterations. The compiler will issue the loads before the additions, so there will be 32 concurrent loads per thread in X and 64 per thread in Y. As before, in X each thread will execute one `h_wp` iteration and in Y a thread executes either 1 or 0, leaving half the SMs in Y idle.

First, L1 cache space. In X $\delta d_l B/g = 8192$ elements per `h_wp`-loop iteration must survive from the first to the second access, requiring about $32\,\mathrm{kiB}$ of cache space. That should comfortably fit. In Y $\delta = 2$ and so the amount of cache space needed doubles, but there should be enough assuming that the writes to `l_out` don't consume cache space. (They would not consume cache space in caches that have a *write around* allocation policy.) Assume that cache is not a factor.

In the previous parts it was reasonable to conclude that L_2 (L2 cache access latency) dominated iteration time. But, because there are 32 concurrent accesses per thread L2-to-SM data bandwidth and compute become a factor. If the data is found in the L2 cache there will be sufficient bandwidth but compute, including load issue time, will become a factor. So doubling the number of vectors on an SM would increase the amount of time, at worst doubling it. If the data were off-chip then memory bandwidth would be a factor and X and Y would be about the same speed. The total array is less than $4\,\mathrm{MiB}$ and so can easily fit in L2.