LSU EE 7722 **Homework 1** **Due: 21 March 2023**

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/2023/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 `.../2023/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 `.../2023/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`. It should produce output ending with a line something like this `32 3.7 4 16 13.8 2977 1.0 777 35382 ---`.

The makefile will compile code for a GPU on the system it was run. Re-run make when moving to a different system. 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. 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 `hw01`

The code in `hw01.cu` contains several kernels that compute the output of part of a transformer neural

network layer. The `hw01` program takes one command-line argument, indicating how many blocks to launch. If the argument is zero then the number of blocks will be set to the number of multiprocessors (which is the default). If the argument is negative then the number of blocks will be $-aP$, where a is the argument value and P is the number of MPs on the GPU.

Computation Overview

The code in the assignment package is based on the *transformer network* introduced by Vaswani *et al* [1]. The discussion in this section uses some of the notation from that paper. The code in this assignment computes the following matrix multiplication:

```
for ( int i_ws = 0; i_ws < d_ws; i_ws++ )
  for ( int i_qkv = 0; i_qkv<d_qkv; i_qkv++ )
    {
      acc_t q = 0;
      for ( int i_model = 0; i_model < d_model; i_model++ )
        q += w_qkv[ i_qkv * d_model + i_model ]
          * h_in[ i_ws * d_model + i_model ];
      h_qkv_cpu[ i_ws * d_qkv + i_qkv ] = q;
    }
```

The input matrices are $w_{\text{qkv}} \in \mathbb{R}^{d_{\text{qkv}} \times d_{\text{model}}}$ and $h_{\text{in}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ws}}}$, and the output matrix is $h_{\text{qkv}} \in \mathbb{R}^{d_{\text{qkv}} \times d_{\text{ws}}}$, where d_{model}, d_{qkv}, and d_{ws} are positive integers. The homework package performs the calculation at two different sizes, in the smaller one $d_{\text{model}} = 32$, $d_{\text{qkv}} = 96$, and $d_{\text{ws}} = 29700$. With those sizes matrix w_{qkv} has 108 rows and 36 columns. Matrix h_{in} is much wider, 36×29700.

The data type for the inputs and outputs is `acc_t`, and the data type for weights are `wht_t`. Both of these are defined near the top of the file as `float`. In most DNN systems the weights would be defined as a 16-bit or sometimes even a smaller type.

The code above can be re-written:

```
for ( int i_sample = 0; i_sample < n_samples; i_sample++ )
  for ( int i_word = 0; i_word < n_words; i_word++ )
    for ( int i_qkv = 0; i_qkv<d_qkv; i_qkv++ )
      {
        acc_t q = 0;
        for ( int i_model = 0; i_model < d_model; i_model++ )
          q += w_qkv[ i_qkv * d_model + i_model ]
            * h_in[ ( i_sample * n_words + i_word ) * d_model + i_model ];
        h_qkv_cpu[ ( i_sample * n_words + i_word ) * d_qkv + i_qkv ] = q;
      }
```

by splitting the d_{ws} dimension into two, n_{samples} and n_{words}. The two code fragments are equivalent, the first is obviously simpler, the second reveals more detail about the h_{in} matrix. The two code fragments can read and write the same matrices. The solutions for this assignment can use either method of addressing the arrays.

Kernels and Performance Issues

Kernel `qkv_base` in the assignment package computes h_{qkv} by dividing its elements evenly between threads. It can be run with any block and grid size. This kernel performs inefficiently for about two reasons: the load instructions overwhelm the level-1 (L1) cache ports, and because it is bandwidth limited for the larger input size.

Consider the inner two loops from `qkv_base`:

```
for ( int i = tid; i < n_wq;  i += num_threads ) {
    const int i_qkv = i % d_qkv;
    const int i_word = i / d_qkv;
    acc_t q = 0;
    for ( int i_model = 0; i_model < d_model; i_model++ )
      q += ld.w_qkv_dev[ i_qkv * d_model + i_model ]
```

```
            * ld.h_in_dev[ i_word * d_model + i_model ];
        ld.h_qkv_dev[ i_word * d_qkv + i_qkv ] = q;
    }
```

In the innermost loop there are two loads, one for `ld.w_qkv_dev` and one for `ld.h_in_dev`. Each load is executed by all of the threads in a warp, and based on that a request is made to the cache. In the code above one of the two loads will be to the same address for every thread in a warp when $d_{\text{qkv}} \geq 32$. (You should be able to easily work out which load.) For the other load each thread in a warp will be accessing a different value of the array, and that causes performance problems.

First, there is a limit to the number of different *sectors* that can be handled without delay for a warp. A sector is a portion of a cache line, what's important here is that the size of a sector in NVIDIA devices is 32 bytes. Recent devices appear to be able to provide four sectors per warp, which is enough data for 32 4-byte elements.

The average number of sectors requested per warp per load instruction is shown in the program output under the `SW` column. (This is discussed further in the Program Output section.) For `qkv_base` that number is 16. The 16 is an average for the two loads, one requesting just 1 sector, the other requesting 32.

There is a second factor which slows down L1 cache access further. Caches are often split into banks. Each bank can only handle a specific set of memory addresses, so a request to load from (or write to) an address must be sent to the bank that can handle that address. The bank to which an address is mapped is usually specified by a range of bits, say bits at positions 2 through 6, of the address' binary representation. If there are four banks then one can look at two bits. For the NVIDIA devices it is possible that bits 6 and 5 determine the bank (because a sector is 32 bytes), where bit 0 is the least-significant bit. For example, consider addresses $A_1 = 0110\,1100_2$ and $A_2 = 1000\,1100_2$. Because the value of bits 6:5 of A_1 are $11_2 = 3_{10}$, it would be mapped to bank 3, and A_2 would be mapped bank 0.

Suppose a warp requests data that can be found in four sectors. That's only good if each sector is mapped to a different bank. If each sector is mapped to the same bank, say all to bank 0, then it would take four cycles to issue the requests. That's in contrast to one cycle when each request is assigned to a different bank. A *bank conflict* occurs when two or more requests for a sector made in a cycle are mapped to the same bank. The number of bank conflicts per warp is shown under the `BXW` column. Any value greater than zero will affect performance. For the code above the susceptibility to bank conflicts is determined by the value of `d_model`. Values that are a power of 2 are the worst. Kernel `qkv_base` suffers badly from bank conflicts.

At the top of this section `qkv_base` was accused of overwhelming level-1 cache ports. That is due to the number of sectors per warp being greater than four and the number of bank conflicts being much greater than 0.

A second problem, affecting only the larger input, is the bandwidth limit between the L2 and L1 caches. Each MP has its own L1 cache, typically about 64 kiB. All MPs share a single L2 cache, 73728 kiB on an RTX 4090. A crossbar connects the SMs to the L2 cache. For the larger input the crossbar cannot provide data fast enough and so performance suffers. This is discussed further in the Program Output section.

Program Output

Starting a run of `hw01` ...

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

... produces the following output:

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

```
GPU 0: NVIDIA H100 PCIe @ 1.75 GHz WITH 81089 MiB GLOBAL MEM
GPU 0: L2: 51200 kiB   MEM<->L2: 2039.0 GB/s
GPU 0: CC: 9.0  MP: 114  CC/MP: 128  DP/MP: 64  TH/BL: 1024
GPU 0: SHARED: 49152 B/BL  233472 B/MP  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 25609 SP GFLOPS  12804 DP GFLOPS  COMP/COMM:  50.2 SP  50.2 DP
GPU 1: NVIDIA H100 PCIe @ 1.75 GHz WITH 81089 MiB GLOBAL MEM
GPU 1: L2: 51200 kiB   MEM<->L2: 2039.0 GB/s
```

```
GPU 1: CC: 9.0  MP: 114  CC/MP: 128  DP/MP: 64  TH/BL: 1024
GPU 1: SHARED: 49152 B/BL  233472 B/MP  CONST: 65536 B  # REGS: 65536
GPU 1: PEAK: 25609 SP GFLOPS  12804 DP GFLOPS  COMP/COMM:  50.2 SP  50.2 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 9.0 (Hopper). The `MEM<->L2` field shows the off-chip bandwidth. `MP` indicates the number of multiprocessors, also called streaming multiprocessors (SM's). `CC/MP` indicates the number of CUDA cores (single-precision functional units) per MP, `DP/MP` indicates the number of double-precision functional units per MP, 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 MP, this does not indicate whether any particular kernel is using that much shared memory or could use that much. 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.)

Next, the program provides information on the network layers to be tested:

```
Layer shape 0: smpls=300.  wds=99.  heads=4.  d_q=d_k=d_v=8.
Layer shape 0: d_qkv=96.   d_model=32.  d_ws=29700
Layer shape 0: w_qkv: 12 kiB   h_in: 3712 kiB
Layer shape 1: smpls=30.  wds=99.  heads=8.  d_q=d_k=d_v=64.
Layer shape 1: d_qkv=1536.   d_model=512.  d_ws=2970
Layer shape 1: w_qkv: 3072 kiB   h_in: 5940 kiB
```

The layers used are specified in the constant array `ls` near the top of `hw01.cu`.

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 MPs. 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 MP*. For a GPU with 40 MPs 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 MP, 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 MP.

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 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 MPs. Run time and other information will be shown for each launch.

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 `qkv_base` (and that kernel should not be modified) and kernel `qkv_base_w2` with the assignment correctly solved.

```
Kernel (qkv_base<ls_d_model(layer_specs[0])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.5  2  16 15.2    3  1.0   50    487 *-
 2  3.5  3  16 15.3    2  1.0   63    329 **
 3  3.5  4  16 15.3    1  1.0   66    256 **-
 4  3.5  4  16 15.3    2  1.0   76    245 **-
 8  3.5  4  16 15.3    2  1.0   75    236 **-
16  3.5  4  16 15.3    2  1.0   73    237 **-
32  3.5  4  16 15.3    1  1.0   72    237 **-

Kernel (qkv_base_w2<ls_d_model(layer_specs[0])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.5  5   3  0.0    3  1.0  134    182 ***-
 2  3.5 10   3  0.0    2  1.0  225     92 +******--
 3  3.5 14   3  0.0    1  1.0  251     67 +*********--
 4  3.5 18   3  0.0    2  1.0  347     54 ++**********---
 8  3.5 30   3  0.0    2  1.0  561     32 ++++++++++++++++++------
16  3.5 31   3  0.0    2  1.0  558     31 ++++++++++++++++++------
32  3.5 31   3  0.0    1  1.0  557     31 ++++++++++++++++++------
 The lines below are fictional and are there to explain the bar graph.
32  3.5 31   3  0.0    1  1.0  557     31 ++++++++++++++++++++++++++++++++++++
32  3.5 31   3  0.0    1  1.0  557     31 ++++++++++++++++++++++++++++++++++++++
```

The output above shows the result for two kernels, `qkv_base` and `qkv_base_w2` each run using layer 0. (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 MPs then there would be a column headed `ac`, which would show the number of resident warps per MP. (The number of resident warps per MP 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 MPs, and in such a case the value in the `ac` column would match the `wp` column.)

The group of columns under the heading `Insn` show information about machine (SASS) instructions. The `/itr` column shows the measured number of machine instructions divided by the number the expected number of iterations of the `d_model` loop: $n_{\text{sample}} n_{\text{words}} d_{\text{qkv}} d_{\text{model}}$. For the kernels in this assignment the number should be greater than 3 (two loads and a multiply/add), the closer to 3 the better. A higher number might indicate that the compiler is using more instructions than it needs to, perhaps because of the way the kernel was written. This might slow execution.

The warp scheduler is responsible for choosing a warp with an instruction ready to execute. Sometimes no such warp can be found. The two major reasons in this assignment are that each warp is waiting for an operand to arrive and that the memory system is busy and so memory instructions must be stalled. The number in the `%` column shows the percentage of time that a warp can be chosen. Let n_{body} denote the value in the `/itr` column and let $\theta_{\text{I}}^{\%}$ denote the value in column `%`. In terms of these execution time is roughly proportional to $100 n_{\text{body}} / \theta_{\text{I}}^{\%}$.

For the kernels in this assignment the value of `/itr` is low enough. If in your solution `/itr` is much larger, say 8.1, then you might try looking at your code. For the kernels initially provided with this assignment `%` will be less than 10. An ideal value is 100, but the best in the sample solution is 31.

The columns in the `L1` group show how efficiently load instructions are issued to the L1 cache. These were discussed in a prior section. Briefly, `SW` shows the number of sectors requested per warp for a memory instruction. Anything above 4 will likely result in reduced performance. The value under `BXW` shows the number of bank conflicts per warp for a load instruction. Anything above 0 will result in reduced performance.

The columns in the `L1<->L2` group show how much data is moving between the L1 and L2 caches. The `N-Rd` column (normalized amount of data read) shows how much data is read, scaled to the ideal amount. A value of 1 is ideal, and will be achieved if each element of the `w_qkv` and `h_in` arrays is read exactly once. A value of 2 indicates that on average each element was read twice. For `qkv_base` the `w_qkv` array is read by every block, and so it is read by every MP. Since for layer 0 `w_qkv`, at 12 kiB, is smaller than the cache it is read just once per MP. Accounting *only* for `w_qkv` the value of `N-Rd` would be the number of MPs. However, each element of `h_in` is read by one or two MPs (you should be able to work out the reason for this on your own) and `h_in` is much larger than `w_qkv` in layer 0, so overall `N-Rd` is small, it would be just 1.367 with 128 MPs and assuming each element of `h_in` were read once. In layer 1 `w_qkv` is much larger, 3072 kiB, and could not fit in the L1 cache and so it must be re-read, inflating `N-Rd`.

The value under the `N-Wr` column (normalized amount of data written) shows how much data moved from L1 to L2, normalized to the ideal amount. It is much easier to achieve the ideal here.

The value under the `GB/s` shows the measured rate of data moving between the L1 and L2 caches (in either direction, but L2 to L1 dominates). The number includes all MPs. (A per MP number is a reasonable alternative but I can't keep changing my mind.) This will become a limiting factor for layer 1, but will not be a problem for layer 0.

The `t/`μ`s` column shows the measured execution time in microseconds. To the right of `t/`μ`s` 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 off-chip data transfer, shown with a `*`. 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 16 warp per MP run. The `-` is a bit more than halfway to the end. That indicates that instruction throughput is more than half of the peak possible.

The FMA utilization is computed by assuming one multiply/add per loop iteration, or $n_{\text{sample}} n_{\text{words}} d_{\text{qkv}} d_{\text{model}}$ FMAs. Then the amount of time it would take to issue that many FMAs is computed. That time is divided by the measured execution time to get the utilization. The amount of time to issue the FMAs is based on the GPU being used and should be accurate (up to CC 9.0).

The instruction utilization, `-`, includes the FMA plus two load instructions per FMA.

Problem 1: Initially kernels `qkv_base` and `qkv_base_w2` are identical. Their execution suffers because they are attempting to load on average 16 sectors per warp. One way of fixing this is by rearranging the weight array. Call the rearranged weight array `w2`, its address is `ld.w_qkv2_dev`. In the unmodified assignment kernel `qkv_w_rearrange` writes `w2` with an exact copy of the weight array. Modify `qkv_w_rarrange` so that it writes `w2` with the weight array rearranged so that the loads in `qkv_base_w2` use fewer than 16 warps per load. Modify `qkv_base_w2` to correctly use this `w2`. *Note: 2021 Homework 1 was similar.*

Make sure that errors are not shown for your code.

Here is sample output when this problem is correctly solved and run on an RTX 4090:

```
Kernel (qkv_base<ls_d_model(layer_specs[0])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.5  2  16 15.2    3  1.0   50    487 *-
 2  3.5  3  16 15.3    2  1.0   63    329 **
 3  3.5  4  16 15.3    1  1.0   66    256 **-
 4  3.5  4  16 15.3    2  1.0   76    245 **-
 8  3.5  4  16 15.3    2  1.0   75    236 **-
16  3.5  4  16 15.3    2  1.0   73    237 **-
32  3.5  4  16 15.3    1  1.0   72    237 **-

Kernel (qkv_base_w2<ls_d_model(layer_specs[0])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.5  5   3  0.0    3  1.0  134    182 ***-
 2  3.5 10   3  0.0    2  1.0  225     92 +******--
 3  3.5 14   3  0.0    1  1.0  251     67 +*********--
 4  3.5 18   3  0.0    2  1.0  347     54 ++**********---
 8  3.5 30   3  0.0    2  1.0  561     32 ++++++++++++++++++------
16  3.5 31   3  0.0    2  1.0  558     31 ++++++++++++++++++------
32  3.5 31   3  0.0    1  1.0  557     31 ++++++++++++++++++------

Kernel (qkv_base<ls_d_model(layer_specs[1])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.7  2  16 13.3 1044  1.0  741  13035 -
 2  3.7  3  16 13.3 1028  1.0 1263   7528 ---
 3  3.7  4  16 13.3 1023  1.0 1540   6140 ---
 4  3.7  4  16 13.3 1020  1.0 1564   6032 ---
 8  3.7  4  16 14.3 1016  1.0 1558   6032 ---
16  3.7  4  16 14.8 1223  1.0 1812   6238 ---
32  3.7  2  16 11.4 3491  1.0 3032  10634 --

Kernel (qkv_base_w2<ls_d_model(layer_specs[1])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  4.6  1   3  0.0 1044  1.0  561  17206 -
 2  4.6  3   3  0.0 1028  1.0 1098   8657 --
 3  4.6  4   3  0.0 1023  1.0 1431   6611 ---
 4  4.6  5   3  0.0 1020  1.0 1939   4865 ----
 8  4.6 10   3  0.0 1016  1.0 3799   2473 +-------
16  4.6 15   3  0.0 1014  1.0 5846   1604 *-----------
32  4.6 15   3  0.0 1014  1.0 6016   1558 *-----------
```

Problem 2: Even with the previous problem correctly solved, execution will be inefficient for the larger layer. In the sample output for the previous problem notice that instruction throughput (the % column) is only 15%. Also notice that `N-Rd` is 1014 and the L1/L2 traffic reaches a ceiling of about 6000 GB/s. The problem is that each MP must read the same data about 1000 (one thousand!) times. The problem can be fixed by changing the assignment of threads to output elements (elements of `h_qkv_dev`). The idea is that each time an element of `w_qkv` is read by an MP, it should be used by multiple threads, not just one. Ideally, by multiple threads in a warp.

Modify kernels `qkv_strip` and `qkv_strip_w2` so that they achieve this. When this is solved correctly the value under `N-Rd` for the larger layer will be lower, as will the number under `GB/s`. Here is the output of a correct solution:

```
Kernel (qkv_strip<ls_d_model(layer_specs[1])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.5  2   6  4.5  254  1.0  246   9604 --
 2  3.5  4   6  4.5  254  1.0  446   5298 ----
 3  3.5  7   6  4.5  256  1.0  672   3543 +----
 4  3.5  8   6  4.5  260  1.0  788   3063 +-----
 8  3.5  9   6  4.7  269  1.0  992   2515 +-------
16  3.5 11   6  4.9  260  1.0 1084   2228 +--------
32  3.5 10   6  4.9  261  1.0 1044   2322 +-------

Kernel (qkv_strip_w2<ls_d_model(layer_specs[1])>):
   --Insn-- --L1--- -- L1<->L2 ---
wp /itr  %  SW  BXW N-Rd N-Wr GB/s   t/μs === Util: FP++  Insn-- Data**  =======
 1  3.5  2   3  1.4  265  1.0  248   9933 --
 2  3.5  5   3  1.2  267  1.0  479   5178 ----
 3  3.5  7   3  1.1  271  1.0  725   3478 +-----
 4  3.5  9   3  1.3  262  1.0  914   2669 +------
 8  3.5 15   3  1.3  265  1.0 1573   1567 *----------
16  3.5 22   3  1.5  261  1.0 2224   1092 *+----------------
32  3.5 24   3  1.5  257  1.0 2415    991 *+------------------
```

References:

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc. `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.