LSU EE 7722 **Homework 2** **Due: 26 April 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/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 `.../2023/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 `.../2023/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`. It should produce output ending with a line something like this `32  4.6 15 100  3  0.0 1014  1.0 4739  0t   1978 +----`.

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 The Microbenchmarks

When tuning code it is important that one does not waste time trying to make the code go faster than what

the hardware can provide. *Microbenchmarks* can provide data on capabilities that will help one estimate hardware limits. In the course repo a set of microbenchmarks can be found in the `cuda/microbenchmarks` directory. They are compiled in the same way as other code samples. (Say, enter the command `gmake`.) A single executable, `mb`, is used for all of the microbenchmarks.

The `mb` program can run four microbenchmarks, atomic operations (`a`), MADD operation latency (`o`), memory latency (`m`), and memory throughput (`s`). The letters in parentheses indicate the command-line option to use to run the respective benchmark. For example, to run the memory latency benchmark invoke `mb` using the shell command `./mb m`. Here is some output:

```
[cyc2.ece.lsu.edu] % ./mb m
GPU 0: NVIDIA GeForce RTX 4090 @ 2.52 GHz WITH 24214 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

Global Load Latency Microbenchmark (Option 'm')

CUDA Kernel Resource Usage:
For mb_g:
       0 shared, 120 const, 0 loc, 26 regs; 1024 max threads per block.
For mb_ro:
       0 shared, 120 const, 0 loc, 26 regs; 1024 max threads per block.
Array size: 67108864 elts. Block size 32 thds.
Kernel mb_g:
             ---Data Touched--- --------------------Latency----------------------
 nbl    iter    Block     Total   ns  cyc !<--------------500 ns-------------->!
1024   20480  512 kiB   512 MiB  301  758 **********************
1024   10240  256 kiB   256 MiB  303  763 **********************
 512   10240  256 kiB   128 MiB  252  635 ******************
 256   10240  256 kiB    64 MiB  117  295 ********
 128   10240  256 kiB    32 MiB  109  275 ********
  64   10240  256 kiB    16 MiB  111  279 ********
  32   10240  256 kiB     8 MiB  111  279 ********
  16   10240  256 kiB     4 MiB  114  287 ********
   8   10240  256 kiB     2 MiB  118  298 ********
   4   10240  256 kiB     1 MiB  118  297 ********
   2   10240  256 kiB   512 kiB  129  324 *********
   1   10240  256 kiB   256 kiB  129  324 *********
   1   10000  128 kiB   128 kiB  125  315 *********
   1   10000   64 kiB    64 kiB   20   51 *
   1   10000   32 kiB    32 kiB   17   44 *
   1   10000   16 kiB    16 kiB   16   40 *
   1   10000    8 kiB     8 kiB   15   38 *
   1   10000    4 kiB     4 kiB   15   37 *
   1   10000    2 kiB     2 kiB   15   37 *
   1   10000    1 kiB     1 kiB   14   36 *
   1   10000  512   B   512   B   14   36 *
```

In this case, an exact latency is not provided. Instead memory latency is shown when accessing different amounts of memory. The values under the `Block` and `Total` columns indicate the amount of data accessed, and the values under the `ns` and `cyc` columns indicate the respective latency. Based on this data one can determine the sizes of the different cache levels, and their latencies. The L1 cache has a latency of about 38 cycles and appears to be between 64 and 128 kibibytes based on the microbenchmarks. (The API reports a

size of 102 kilobytes.) Similarly the L2 cache latency appears to be about 280 cycles and the size is between 64 and 128 mibibytes. Memory latency appears to be about 760 cycles.

The streaming microbenchmark shows the maximum throughput between various layers under different circumstances. It is selected with the `s` option:

```
[cyc2.ece.lsu.edu] % ./mb s
GPU 0: NVIDIA GeForce RTX 4090 @ 2.52 GHz WITH 24214 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

Streaming Microbenchmark (Option 's')
 me_n_iter  49  Per Block     6272 kiB,  Per Kernel  802816 kiB
 l2_n_iter 384  Per Block    49152 kiB,  Per Kernel   55248 kiB
 l1_n_iter 800  Per Block       32 kiB,  Per Kernel    4096 kiB
                    --Insn-- ------L1 Per SM------- -L2 GB/s-- DRAM
Lv Blk V SPL     s /add  %  SW TW  BXW B/cyc  GB/s   /SM /GPU GB/s
ME 128 1  s0    851  2.5  2   4  1  0.0   3.4   7.5   7.5  966  963
L2 128 1  s0   1317  2.4  7   4  1  0.0  17.3  38.2  38.2 4892   45
L1 128 2  s0    321  1.7 41   8  2  0.0 138.1 326.8   0.1   13   15
L1 128 1  s0    639  2.2 33   4  1  0.0  67.7 164.2   0.1    7    9
L2 128 1  s0   1317  2.4  8   4  1  0.0  17.3  38.2  38.2 4892   43
L2  64 1  s0    677  2.4  8   4  1  0.0  31.9  74.3  74.3 4757   82
L2   2 1  s0    621  2.4  0   4  1  0.0  32.6  81.0  81.0  162   81
L2   1 1  s0    623  2.4  0   4  1  0.0  32.6  80.8  80.8   81   82
L2   1 1  p1    622  2.4  0   4  2  0.0  32.6  80.9  80.9   81   81
L2   1 1  p2    629  2.4  0   4  4  0.0  32.5  80.0  80.0   80   80
L2   1 1  p3    746  2.4  0   8  8  0.0  27.4  67.5  70.1   70   68
L2 128 1  s0   1317  2.4  8   4  1  0.0  17.6  38.2  38.2 4893   45
L1 128 4  s0    320  1.4 34  16  4  0.0 147.3 327.4   0.1   13   13
L1 128 2  s0    320  1.7 41   8  2  0.0 139.0 328.1   0.1   13   15
L1 128 1  s0    639  2.2 33   4  1  0.0  67.7 164.2   0.1    7    7
L1 128 1  p1    644  2.2 33   4  2  0.0  66.7 162.9   0.1    7    7
L1 128 1  p2    644  2.2 33   4  4  0.0  66.6 162.9   0.1    7    7
L1 128 1  p3    640  2.2 33   8  8  0.0  67.1 163.9   0.1    7    8
L1 128 1  s1    641  2.2 33   8  2  1.0  67.1 163.7   0.1    7    7
L1 128 1  s2   1271  2.2 17  16  4  3.0  34.7  82.5   0.0    3    3
L1 128 1  s3   2552  2.2  8  32  8  7.0  17.1  41.1   0.0    2    2
L1 128 1  s4   5136  2.2  4  32 16 15.0   8.4  20.4   0.0    1    3
L1 128 1  s5  10372  2.2  2  32 32 31.0   4.1  10.1   0.0    0    1
Key SPL: Stride or Permute Pattern Lg
         s0, sequential pattern (stride 1) within warp.
         si, stride 2^i within warp.
         pi, Low five bits sequential, 2^i distinct values for others.
```

Here is a brief description of the column headings. `Lv` indicates the memory hierarchy level being targeted: memory, L2 cache, or L1 cache.

`Blk` indicates how many blocks are participating. In most cases one would want one block per SM to participate. But for cases where, say, only one block at a time would be busy the rows with `Blk` equal 1 might be useful.

`V` indicates the vector length of the item accessed. A 1 is just an ordinary float, 2 is a two-element vector, and 4 is a four-element vector.

`SPL` describes the access pattern. A value of `s0` is a simple sequential pattern where adjacent threads access adjacent elements. A value of si indicates that thread number τ accesses element number $\tau 2^i$. So, `s0` is sequential, `s1` is a stride of 2, etc. A stride of 2^i for $i > 0$ will result in bank conflicts. A value starting with `p` is for a permutation pattern. Permutation patterns do not have bank conflicts (assuming element number x uses bank $x \bmod 32$), but within a warp there are 2^i distinct values for bits `63:5` of the address for pattern pi. These patterns show how many tag lookups can be performed per warp without slowing things down. Based on the results above, up to four.

The `/add` column shows how many instructions were executed per arithmetic operation. The benchmark computes a sum using the loaded value. An ideal value is $1 + 1/v$ where v is the vector length.

The `%` column shows the instruction issue rate reported by Nvidia's CUPTI system. A value of 100% indicates that at each cycle an instruction could be issued by a warp scheduler. Based on the numbers above it looks like instructions requiring multiple cycles to dispatch, such as loads, make it impossible to reach 100%.

`SW` shows the number of sectors requested per warp for a memory instruction. Anything above 4 will likely result in reduced performance. `TW` shows the number of tag lookups per warp. (Caches manage data in units of *lines*. For each cached line a part of the data's address, called the *tag*, is stored. When a cache is checked for data the lookup address is compared to the tags of the cached data. A cache line in the Nvidia devices of at least CC 7 to 9 is 128 bytes, divided into four 32-byte sectors.) 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 `B/cyc` column shows the data throughput in bytes per SM per cycle. In recent Nvidia devices an SM has four warp schedulers with a warp size of 32, so that at best 128 threads per cycle can be issued. (In most cases one instruction.) So, a value of 128 in the `B/cyc` column means that each thread is getting one byte per cycle. In the microbenchmark threads are usually loading floats, which are 4 bytes. Also, in recent devices a load instruction takes four cycles to dispatch. So a value of 128 would indicate that loads are issuing without delay (beyond their four-cycle dispatch time). An H100, CC 9.0, actually attains about 128 bytes per cycle using scalar loads. This indicates that the scheduler can issue both a load instruction and an FP32 instruction in the same cycle. (Or it indicates this conclusion is wrong.) For the consumer-grade CC 8.9 device about half of this peak is attained for scalar loads. Note that for vector loads even a "cheap" RTX 4090 can execute at about 128 bytes per warp per cycle. That is likely due to the vector load requiring just four cycle to dispatch, but providing 2 or 4 floats. So, for a 2-element vector, dispatch time would be $4 + 1 = 5$ cyc, while the time to read the data eight cycles based on 128 bytes per cycle. So for 2-element vectors instruction issue stalls waiting for the data.

The `GB/s` columns show the data throughput. The column under `L1 Per SM` shows the data throughput between the L1 cache and the SM registers. The values under the `L2` columns show the throughput between the L1 and L2 caches. Those throughputs are shown per SM and for the entire GPU. Finally, the value under the `DRAM` column shows the throughput between the L2 cache and DRAM. In the row targeting memory, `Lv=ME`, the throughputs under each `GB/s` column should all be the same (after accounting for whether they are per SM or for the whole GPU). For the rows targeting the L2 cache, the throughput under the `DRAM` column should be much less than the others (ideally zero). For the rows targeting the L1 cache, the throughput under `L2` should be much smaller than those under `L1`.

Using `hw02`

The code in `hw02.cu` contains several kernels that compute the output of part of a transformer neural network layer. The `hw02` program takes one command-line argument, indicating how many blocks to launch. If the argument is zero or missing 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.

Program Output

Starting a run of `hw02` ...

```
[koppel@grace hw02]$ ./hw02
```

... 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 GeForce RTX 4090 @ 2.52 GHz WITH 24214 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). The `MEM<->L2` field shows the off-chip bandwidth. `SM` indicates the number of streaming multiprocessors. `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 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 hw02.cu.

When run without arguments or with a 0 as the first argument, such as ./hw02 0, the program launches each kernel multiple times, starting with 4 warps per block, up to 32 warps per block. For this assignment (hw 2) the number of warps per block will always be a power of 2. The number of blocks is set equal to the number of MPs. Run time and other information will be shown for each launch. If the argument $-n$ then it will launch with n blocks per MP.

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. The 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]),1,1,1>), using 39 regs
   ---Insn---- --L1--- -- L1<->L2 ---
wp /itr %  TAc SW  BXW N-Rd N-Wr GB/s Imb   t/s === Util: FP++  Insn-- ========
 4  3.6 17 100  3  0.0    2  1.0  173  0t    108 +-----
 8  3.6 30 100  3  0.0    2  1.0  302  0t     59 +----------
16  3.6 31 100  3  0.0    2  1.0  437  0t     40 ++--------------
32  3.7 31 100  3  0.0    1  1.0  460  0t     38 ++----------------
```

The output above shows the result one kernel, qkv_base each run using layer 0. (The name shown is how the function was named, including the template parameter.) The number of registers used, 39, is also shown.

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 /iter 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 TAc column shows workload balance within a block. A value of 100 is ideal. Lower numbers, such as 99p, are followed by a letter. The letter indicates the principal cause of the imbalance. A p indicates predicate-off instructions, and an d indicates warp divergence.

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 `/iter` column and let $\theta_{\text{I}}^{\%}$ denote the value in column `%`. In terms of these execution time is roughly proportional to $100n_{\text{body}}/\theta_{\text{I}}^{\%}$.

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 `Imb` shows workload imbalance. A `0` is ideal. A value of `10t` indicates that execution is taking twice as long, 100% longer, based on time measurements. A value of `5i` indicates that on block is using 50% more instructions than the average block.

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: Modify `qkv_hw2_a` so that it improves instruction issue utilization by having one thread compute $m_w \times m_h$ elements, m_h rows and m_w columns, of the output matrix, h_{qkv}. Doing so requires reading $m_w d_{\text{model}}$ elements of h_{in} (m_w columns, each column having d_{model} rows) and $m_h d_{\text{model}}$ elements of w_{qkv}, and then performing $m_w m_h d_{\text{model}}$ multiply/add operations, thus reducing the dispatch time to $[4/m_w + 4/m_h + 1]$ cyc for recent NVidia devices counting only load and multiply/add instructions. (Dispatch time would be $[4 + 4 + 1]$ cyc if each thread computed one element of h_{qkv} per iteration. Routine `qkv_base` computes m_w elements per iteration and has a dispatch time of $[4/m_w + 4 + 1]$ cyc cycles.

Template parameter `m_wd` is already used for the number of columns of h_{qkv} to compute. Use template parameter `m_ht` for the number of rows of matrix h_{qkv} to compute. Template parameter `m_dp` will be used in the next problem. Assume that these template parameters will only be set to powers of two.

There are two main challenges in solving this problem correctly: distributing work evenly and avoiding bank conflicts. The impact of bank conflicts can be reduced in part by using vector load instructions (but those are optional).

Problem 2: Modify `qkv_hw2_a` or `qkv_hw2_b` so that it improves workload balance by using m_d threads to compute each element of h_{qkv}, where m_d is the value of template parameter `m_dp`. The default is `m_dp=1`, meaning that each element of h_{qkv} is computed by one thread. In fact, each thread is computing `m_ht * m_wd` elements, but that doesn't change the fact that each element is computed by one thread. Modify one (or both) of the routines so that each element is computed by m_d threads. To do so each thread executes a portion of the `i_model` loop and then the resulting partial sum is added to the partial sum computed by the other threads computing the same element of `h_qkv`.

There are several ways to add together partial sums. Choose a method that is fastest. The easiest is to use an atomic add. Try using an atomic add initially, but then compare it to a technique using shared memory or warp shuffles.

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`.