LSU EE 7722 — Homework 5 — Due: 28 April 2025

Formatted 14:00, 25 April 2025

Basic Setup

Follow the instructions for class account setup on `https://www.ece.lsu.edu/gp/proc.html`.
Code for this assignment is in directory `../hw/gpm/2025/hw04`.

If the class account has been set up properly, the code can be built from within Emacs by pressing `F9` when visiting any file in the `.../2025/hw05` directory or when in an Emacs shell buffer (which can be entered using `Alt`-x shell`Enter`). The code can be built from the command line using the command `make -j 4` (assuming `.../2025/hw05` 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 `hw05`, one with the suffix `-debug`, such as `hw05-debug`, and one with the suffix `-cuda-debug`, such as `hw05-cuda-debug`. The versions with the `-cuda-debug` suffix are compiled with host optimization turned off and CUDA debugging turned on, which facilitates debugging but slows down execution. To debug CUDA or host (CPU) code use the Cuda version of gdb, `cuda-gdb`. Note that the `-cuda-debug` versions will run much more slowly than the regular versions. The executables with the suffix `-debug` are compiled with host optimization turned off but CUDA debugging turned off and so CUDA code cannot easily debugged with these executable files. Use `gdb` or `cuda-gdb` to debug `hw05-debug`.

Running make on a clean directory will produce a large amount of output. The make program and the file it reads, `Makefile`, are designed to build executables in a lazy fashion, meaning that they only create a file if it is not present or if its prerequisites have changed. Therefore a second run of make will take much less time.

Quickly check whether the build is successful with the command `./hw05`. It should produce output ending with a line something like this `32 32 0.28   20  9.5  0.2  2.0  54 2791  464  1 ---********`.

The makefile will compile code for a GPU on the system it was run. Re-run make when moving to a system using a different GPU. The Makefile should automatically detect whether the GPU for which the executable was built matches the GPU on the current system, and re-build if needed.

Background and Reference Material

For this assignment one must be able to write, or at least modify, CUDA kernels. A good reference is the CUDA C++ Programming Guide, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Focus on Chapter 5 up to and including 5.3 (Memory Hierarchy), but skip 5.2.1 (Thread Block Clusters). For sample code a good place to start is 2021 Homework 1, and other past assignments given in this course. The CUDA C used in this assignment is very close to C++20. A good reference for C and C++ is `https://en.cppreference.com/w/`.

In the references below some information is provided for specific architectures, either by CC (*e.g.*, 8.9) or by name (*e.g.*, Ada Lovelace). CC 8.9 GPUs implement Ada Lovelace architecture, and 9.0 implements Hopper. For this assignment only consider CC 8.9 and 9.0 GPUs. The compute capability (CC) of the lab GPUs is shown on the system status page.

A solution to these problems requires some understanding of the hardware structure, in particular how requests are issued to the L1 cache. Some of that material is reviewed in this assignment. For additional description see Chapter 7 of the Programming Guide for the basics (but not including the L1 cache), and also Chapter 19 (Compute Capabilities) for some more details.

The hardware is covered in greater depth in the Kernel Profiling Guide,
`https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html`. Focus on Section 3.1 (Metrics Guide, Hardware Model) and Chapter 9 (Memory Chart). There is no need to read the material on *how* metrics are collected and there is no need to run the profiler yourself. The assignment code uses the CUPTI API to collect data. In class an SM (or MP) was described as having several–usually four–*warp schedulers*. The Profiling Guide refers to warp schedulers as sub partitions. For this assignment requests to the L1 cache are all global requests. Later in the semester we will make shared and maybe local requests, but probably not texture or surface requests.

Using `hw05`

The code in `hw05.cu` contains several kernels that multiply matrices. To make sure it compiled correctly run it with arguments `./hw05`.

The first argument is used to specify the number of blocks. When there are zero arguments, `./hw05`, or when the first argument is zero, `./hw05 0`, the number of blocks is set equal to the number of SMs. When the first argument is a positive integer, such as `./hw05 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 `./hw05 -5`, then each kernel will be launched with that many blocks *per SM*. For a GPU with 40 SMs and running with `./hw05 -5`, a total of $5 \times 40 = 200$ blocks will be launched per kernel. Note that there is no guarantee that five blocks will *simultaneously* run (be active on) any SM, for example. If the kernels use lots of shared memory or registers fewer than five will run (and the others will have to wait until enough active blocks finish).

Program Output

Detailed output is obtained by running without command-line arguments:

```
[hw04]$ ./hw05
```

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

```
GPU 0: NVIDIA GeForce RTX 4090 @ 2.52 GHz WITH 24078 MiB GLOBAL MEM
GPU 0: L2: 73728 kiB   MEM<->L2: 1008.1 GB/s
GPU 0: CC: 8.9  SM: 128  SP-FP32/SM: 128  DP-FP64/SM:  2  TH/BL: 1024
GPU 0: SHARED: 102400 B/SM   CONST:  65536 B      NUM REGS:  65536
GPU 0: SHARED:  49152 B/BL  SH RES:   1024 B/BL  SH OPT-IN: 101376 B/BL
GPU 0: PEAK: 41288 SP GFLOPS  645 DP GFLOPS  COMP/COMM:  163.8 SP  5.1 DP
Using GPU 0
```

This assignment will only work on GPUs of CC 8 or greater.

Most fields are self-explanatory. For example, `L2` is the size of the level-2 cache and CC indicates that the device is of compute capability 8.9 (Ada Lovelace). The `MEM<->L2` field shows the off-chip bandwidth. `SM` indicates the number of streaming multiprocessors, also just called multiprocessors (MP's). `SP-F32/SM` indicates the number of 32-bit floating point functional units (once called *CUDA cores*) per SM, `DP-FP64/SM` indicates the number of double-precision functional units per SM, and `TH/BL` is the maximum number of threads per block.

The amount of shared memory available is shown in several places. The amount of shared memory on an SM is indicated by `B/SM`. Whether a block can access that much memory depends on several factors. The storage indicated by `B/SM` can be used for both shared memory and the L1 cache. The L1 cache size is usually the same size or a bit larger than the shared memory size.

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

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

The next line, `PEAK`, shows FP operation bandwidth in which a fused multiply-add is counted as one operation. (Most of the rest of the world counts a multiply-add as two operations, but in this class it's one.) The bandwidth is shown for both the single-precision (FP32) and double-precision (FP64) functional units. The floating-point bandwidth of the tensor cores is higher, though precision is lower. The `COMP/COMM` line gives the computation to communication ratio in floating-point operations per floating-point element transfers. (The information above was collected in part using the runtime library's `cudaGetDeviceProperties` function.)

Performance Data

The assignment consists of kernels that multiply matrices using either FP32 functional units or tensor cores. Each kernel is run multiple times, starting with four warps per block, in successive runs increasing the number of warps per block. A line of performance data is printed for each run. Appearing below is a portion of the output for an RTX 4090, showing kernel `mm_tile_wd_ht_dp`.

```
Kernel mm_tile_wd_ht_dp, 117 regs. Shape 5120 x 2064 x 4096. wd=8, ht=8, dp=4  FP32
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: FP++  Insn-- ===========
 4  1.44 15  2  0.7  293  1.0 2127  0t  10533 +++----
 8  1.44 25  2  0.7  151  1.0 1933  0t   6002 ++++++------
16  1.44 32  2  0.7   81  1.0 1264  1t   4918 +++++++-------
 The lines below are fictional and are there to explain the bar graph.
16  1.44 38  2  0.7   81  1.0 1587  0t   3916 ++********************************
16  1.44 38  2  0.7   81  1.0 1587  0t   3916 ++++++++++++++++++++++++++++++++++
```

The output above shows the result a kernel, `mm_tile_wd_ht_dp`.

Column `wp` shows the number of warps per block in the run. If the number of blocks in a launch is not set to the number of SMs then there would also be a column headed `ac`, which would show the number of resident warps per SM. (The number of resident warps per SM is a multiple of the number of warps per block. By default the number of blocks in a launch is set equal to the number of SMs, and in such a case the value in the `ac` column would match the `wp` column.)

The 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 either `FMMA` (scalar multiply/add) or `HMMA` (tensor core warp multiply/add) instructions. The closer the value is to one the better. If it's less than 1 then something is computed incorrectly. 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 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 columns in the `L1` group show how efficiently load instructions are issued to the L1 cache. 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 `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 `t/µs` *column* shows the measured execution time in microseconds. *The* `FP θ` *column* shows floating point throughput in GFLOPs. *To the right of* `FP θ` is a bar graph showing how busy three resources are (based on certain assumptions). Three resources are tracked, FMA (fused multiply/add) or tensor core instructions, shown with a `+`, FMA along with load instructions, shown with a `-`, and data transfer, shown with a `*`. The data transfer shown is either L1/L2, indicated with an `L2**` in the column heading, or L2/Mem, indicated with a `Mem**` in the column heading. The right-most position of a resource's character indicates what fraction of the time that resource is busy. A resource is being used 100% of the time if its character reaches the rightmost position (the last `=` in the column heading over the bar graph).

That is true in the last line for the FMA resource, and in the penultimate line for the off-chip data transfer. In the last line we would say that the FP capability is being saturated (a good thing) and in the penultimate line we would say that data transfer is being saturated (also a good thing in some situations including the 2025 Homework 4 assignment). Those last two lines are fictional. Consider the line for the 8 warp per SM run. The `*` is a about halfway to the end. That indicates that L1/L2 data throughput is more than half of the peak possible. The instruction utilization, `-`, includes the FMAs, two loads, and one store per element.

Assignment Introduction

The assignment code includes several kernels for multiplying matrices, they are taken from the examples in the `cuda/matrix_mult` directory in the course repo. A run of the assignment code will launch only three of those kernels, `mm_tile_wd_ht_dp`, `mm_tc`, and `mm_hw05`.

Kernel `mm_tile_wd_ht_dp` does not use tensor cores and computes with 32-bit precision. Kernel parameters specify the width and height of a tile of C computed by each thread. This is the best of the non-tensor core kernels presented in class.

Initially `mm_tc` and `mm_hw05` are identical, the solution to this assignment should be put in `mm_hw05`. These kernels use tensor cores and read input matrices with 16-bit floating point values. The kernel arguments (shown ad `wd` and `ht` in the output) indicate the width and height of the macro tile computed by each warp. See Problem 1.

The file contains other kernels for multiplying matrices, but these are not run. Comments explain how they work and are there for reference.

Here is sample output of the unmodified code:

```
Kernel mm_tile_wd_ht_dp, 117 regs. Shape 5120 x 2064 x 4096. wd=8, ht=8, dp=4  FP32
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: FP++  Insn-- ===========
 4  1.44 15  2  0.7  293  1.0 2127  0t  10533 +++----
 8  1.44 25  2  0.7  151  1.0 1933  0t   6002 +++++------
16  1.44 32  2  0.7   81  1.0 1264  1t   4918 ++++++-------

Kernel mm_tc, 38 regs. Shape 5120 x 2064 x 4096. wd=1  ht=1  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  7.09  5  8  0.9   89  1.0 1323  0t   5178 --
 8  7.09  9  8  0.9   80  1.0 2236  0t   2761 +---
16  7.10 10  8  0.9   81  1.0 2344  1t   2666 +---
32  7.10 11  8  0.9   77  1.0 2413  0t   2464 +----

Kernel mm_tc, 96 regs. Shape 5120 x 2064 x 4096. wd=8  ht=1  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  3.90  9  8  0.9   27  1.0  766  0t   2757 +--
 8  3.90 20  8  0.9   18  1.0 1072  0t   1340 ++---
16  3.90 29  8  0.9   13  1.0 1269  0t    871 +++-----
```

In the unmodified code the tensor core kernel outperforms the scalar kernel, taking $\frac{871}{4918} = \frac{1}{5.65}$ the amount of time based on the best runs above. That sounds good but actually they should take $\frac{1}{16}$ based on NVIDIA specifications and the fact that the tensor core is using operands of only 16 bits. Notice that though the best `mm_tc` run is faster than the `mm_tile_wd_hp_dp` runs, the bar graphs are shorter. That's because the bar graphs show the fraction of full potential and the `mm_tc` run is not getting as much out of the tensor cores as the `mm_tile_wd_hp_dp` is getting out of FP32 functional units.

In this assignment two causes of reduced performance are to be addressed. The first is load dispatch bandwidth. It takes four cycles to dispatch the threads in each load, and so it is better that the loaded values be used as many times as possible. See Problem 1.

The other issue is the time needed to move data. See Problem 2.

One culprit for lower performance mentioned in class was bank conflicts. That is **not** part of this assignment. That problem does not occur here because the number of columns of the A matrix is set to a convenient value (so that no padding is needed). There is still about one bank conflict per access but that can't be avoided with tensor cores as far as I know.

Appearing below are some sample kernels from a correctly solved solution.

```
Kernel mm_tc, 38 regs. Shape 5120 x 2064 x 4096. wd=1  ht=1  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  7.09  5  8  0.9   89  1.0 1315  0t   5207 --
 8  7.09  9  8  0.9   80  1.0 2226  0t   2773 +---
16  7.10 10  8  0.9   83  1.0 2394  0t   2665 +---
32  7.10 11  8  0.9   77  1.0 2437  0t   2446 +----

Kernel mm_hw05, 42 regs. Shape 5120 x 2064 x 4096. wd=1  ht=1  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  7.13  3  8  0.9   71  1.0  648  0t   8478 -
 8  7.13  6  8  0.9   53  1.0  925  0t   4479 --
16  7.13 10  8  0.9   36  1.0 1100  0t   2540 +---
32  7.14 14  8  0.9   27  1.0 1704  0t   1247 ++-------


Kernel mm_hw05, 255 regs, 632 LOCAL. Shape 5120 x 2064 x 4096. wd=8  ht=4  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  5.91 10  5  0.3   46 39.2 2459  1t   2754 +
 8  5.91 11  5  0.3   60 39.3 3309  2i   2382 +-


Kernel mm_hw05, 234 regs. Shape 5120 x 2064 x 4096. wd=8  ht=2  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  2.54 30  8  1.0   18  1.0 1681  0t    855 +++---
 8  2.54 47  8  1.0   14  1.0 1967  0t    566 ++++-----

Kernel mm_hw05, 212 regs. Shape 5120 x 2064 x 4096. wd=4  ht=4  tiles.  FP16
   --Insn-- --L1--- -- L1<->L2 ---
wp  /itr %  SW  BXW N-Rd N-Wr GB/s Imb   t/µs === Util: TC++  Insn-- ===========
 4  2.36 30  8  1.0   18  1.0 1687  0t    853 +++--
 8  2.36 44  8  1.0   16  1.0 2273  0t    570 ++++----
```

The first two kernels, for 1×1 tiles per warp, show the benefit of assigning tiles to improve reuse within a block. The values under `N-Rd` are clearly better for `mm_hw05` for larger blocks. The performance for 4- and 8-warp blocks is not as good due to some other reasons which at the moment are unknown. The third kernel computes 8×4 tiles per warp. There are not enough registers (the maximum is 255 per thread) and so some values, perhaps of C, have to be spilled and filled to local memory. That can be seen on the part of the heading line reading `632 LOCAL` which indicates that 632 memory locations are used for values that would have been in registers if there were enough registers. As a result instruction usage is high, 5.91, but not terrible.

The last two kernels are the best performing of the preliminary solution. Notice that the 8×2 tile version is slightly better than 4×4 despite make less efficient uses of loads.

Problem 1: Kernel `mm_hw05` (and `mm_tc`) have template parameters `m_wd_n` and `m_ht_n` which indicate that a warp should compute a macro tile which is `m_wd_n` tensor core tiles wide (along a row) and `m_ht_n` tensor core tiles high (along a column), for a total of `m_wd_n * m_ht_n` tensor core tiles. A tensor core tile is the tile computed by an execution of `mma_sync`. In the unmodified assignment both kernels correctly honor `m_wd_n` but ignore `m_ht_n` (in affect assuming `m_ht_n=1`).

The code computing a macro tile in the unmodified assignment is:

```
    fragment<matrix_a,    tm, tn, tk, ab_elt_t, aorg> tile_a;
    fragment<matrix_b,    tm, tn, tk, ab_elt_t, borg> tile_b;
    fragment<accumulator, tm, tn, tk, float> tile_C_acc[m_wd_n];

    for ( auto& tca: tile_C_acc ) fill_fragment(tca, 0);

    for ( ssize_t i_k = 0; i_k < A_ncols; i_k += tk )
      {
        load_matrix_sync
          ( tile_a,  a_ptr + C_row_0 * a_row_stride + i_k * a_col_stride,  a_stride );

        for ( ssize_t i_wd = 0; i_wd < m_wd_n; i_wd++ )
          {
            load_matrix_sync
              ( tile_b,  b_ptr + i_k * b_row_stride
                + ( C_col_0 + i_wd * tn ) * b_col_stride,
                b_stride );

            mma_sync( tile_C_acc[i_wd], tile_a, tile_b, tile_C_acc[i_wd] );
          }
      }
    for ( int i_wd = 0; i_wd < m_wd_n; i_wd++ )
      store_matrix_sync
        ( &ld.CT_dev[ C_row_0 + ( C_col_0 + i_wd * tn ) * A_nrows ],
          tile_C_acc[i_wd], A_nrows, mem_col_major );
  }
```

In the code above the data loaded into `tile_a` is used for `m_wd_n` computations of an `mma_sync`, which is a good thing because load instructions take four cycles to dispatch (on CC 8.9 and other recent generations). The larger `m_wd_n` is, the lower the impact of the dispatch time of those loads.

Regardless of the value of `m_wd_n` `tile_b` will be used for just one `mma_sync` in the code above.

(*a*) Modify `mm_hw05` so that `tile_b` will be used for `m_ht_n` calls to `mma_sync`, and (as is currently the case) so that `tile_a` is used for `m_wd_n` calls to `mma_sync`.

To correctly solve this one must change the code shown above and also the code computing `C_row_0` and `C_col_0`.

(*b*) The unmodified code will run `mm_hw05` with a variety of macro tile shapes (values of `m_wd_n` and `m_ht_n`). Those shapes are specified in the `SPECIALIZE_KERNEL` macro, such as `1,1`, and `8,2`. Both `m_wd_n` and `m_ht_n` must be a power of 2. Try finding shapes that will yield better performance by adding lines to the macro. (One might want to do this after solving the next problem.) Search for `SPECIALIZE_KERNEL` and look for the comment explaining how to modify it.

Problem 2: As we should know effective tiling can reduce the amount of data movement in matrix multiplication. Tiling is effective when the compute devices sharing storage and communication operate on a square tile of the output matrix. Modify `mm_hw05` so that the warps in a block operate on a roughly square part of C.

In the unmodified code a simple method is used to assign macro tiles to warps:

```
constexpr int wp_sz = 32;
const int tid = blockIdx.x * blockDim.x + threadIdx.x;
const int wp = tid / wp_sz;
const int n_warps = blockDim.x * gridDim.x / wp_sz;

for ( int i = wp; true; i += n_warps )
  {
    const ssize_t C_row_0 = i % C_nrow_tiles * tm;
    const int C_col_0_raw = i / C_nrow_tiles * m_wd;
    if ( C_col_0_raw >= B_ncols ) break;
    const bool partial_cols = C_col_0_raw + m_wd > B_ncols;
    const ssize_t C_col_0 = partial_cols ? B_ncols - m_wd : C_col_0_raw;
```

With the code above warps in a block are assigned macro tiles along a column. This is ideal when `m_wd_n` is equal to the number of warps, but a value of `m_wd_n` that large may cause problems with larger blocks. Also, the code above does not work well when `m_ht_n` is larger.

(*a*) Modify the kernel so that macro tiles operated on by all the warps in a block form a square or close to it for each iteration of the `i` loop above. Assume that the number of rows of A and number of columns of B are a multiple of 128. Also assume that the number of threads in a block is always a power of 2.

It is possible to take this a step further and assign rows and columns so that all of the macro tiles from all of the warps in all of the blocks collectively form something close to a square during an `i` iteration. What makes this more tedious is that the number of blocks need not be a power of 2, and in fact it can be prime. For that reason, in this assignment only try to arrange the macrotiles from a block into a square, not all the blocks collectively. See `mm_tile_wd_ht_dp` for an example of how to assign rows and columns to threads so that rows and columns operated on by all threads in a kernel approximately form a square.