LSU EE 7722 **Homework 2** **Due: 11 April 2025**

Formatted 19:17, 2 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/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 `.../2025/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 `.../2025/hw02` is the current directory). Either method runs a makefile that builds all examples in the directory. It builds two versions of each program, one taking the base name of the main file, such as `hw02` and one with the suffix `-debug`, such as `hw02-debug`. For CUDA assignments, but not this one, a third version is built, with the suffix `-cuda-debug`, such as `hw02-cuda-debug`.

The executables with the suffix `-debug` are compiled with host optimization turned off, host debugging on, but CUDA debugging turned off. Use `gdb` to debug these. (It is possible to use `gdb` to debug executable `hw02`, but due to optimization the values of certain variables can't be printed, breakpoints can't be set at certain lines, and the order of execution may not match the order of statements in the source code.)

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 extensive output starting something like

```
CPU model Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz
Node has 2 CPUs, CPU has 20 cores, total 40 cores. Measured clock 3.40 GHz.
(This run spawns 18 threads.)
L1 Data Per Core     48 kiB, 12-way, line size 64 bytes.
L2 Unif Per Core   1280 kiB, 20-way, line size 64 bytes.
L3 Unif Per Chip  30720 kiB, 12-way, line size 64 bytes.
Vector width 512 bits, 32 vector registers, and
2 vector unit(s) per core. (Assumed.)
Data bw 33.019 GB/s (measured). For 18 threads:  comp/comm = 237.2

Matrix 2048  1024  1024  4096.  Duration simple: 455.049 ms
Stride 2048  1040  1024  4112  = 2048 x 4112. 18 thds.  HW01
-------Tile------- --L2 Occ---         -FP Throughput- Insn/ Util
 t1  t3    m1   m3 m1*d2 d2*m3  Dur/ms  GFLOPs  % Peak Vec I  Mes
 16  16    16 4096  0.05 12.80   23.11  371.67   18.98  2.03   69
  8  32     8 4096  0.03 12.80   19.95  430.62   21.99  2.22   64
```

And finally after lots of output finishing

```
  4  64   456  256  0.05  3.20  289.40  118.73    6.06  2.22   96
  4  64  1824   64  0.05  0.80  343.42  100.05    5.11  2.22   83
  2 128   228  256  0.03  3.20  525.60   65.37    3.34  2.36   96
  2 128   456  128  0.03  1.60  530.45   64.77    3.31  2.36   91
  1 256   114  256  0.01  3.20  856.22   40.13    2.05  2.18   96
```

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

Background and Reference Material
This assignment is written in C++20, with GNU extensions used for vector types. A good reference for C and C++ is `https://en.cppreference.com/w/`.

Assignment Overview
The code in `hw02` is based on the solution to Homework 1. It multiplies `app.a`, a $d_1 \times d_2$ matrix (a matrix having d_1 rows and d_2 columns), by `app.b`, a $d_2 \times d_3$ matrix, where `app` is a structure holding the matrices and other items relevant to this assignments. The matrices are initialized with random numbers uniformly selected in $[0, 1]$. In members `app.a` and `app.b` there is no padding, members `app.ap` and `app.bp` hold those same matrices but with padding. The layout for `app.ap` has `mmd.d1` rows `mmd.s2a` columns, though only the first `mmd.d2` columns hold useful data, the other columns (the padded columns) hold zeros. Similarly the layout for `app.bp` has `mmd.d1` rows and `mmd.s3` columns, but only the first `mmd.d3` columns hold data, the rest hold zeros. Finally, `mmd.gp` is a padded version of the output array with `mmd.d1` rows and `mmd.s3` columns. The strides, `mmd.s2a` and `mmd.s3` are taken from the solution to Homework 1.

Routine `mm_simple` multiplies the arrays using a simple three-level loop nest and writes the product to `app.g_simple`. Its purpose is to compute an answer that will be considered correct, which is why it is kept simple. Unlike in Homework 1, `mm_simple` reads and writes the padded arrays.

Routine `mm_tiled_hw01` is a simplified version of the solution to Homework 1, and should not be modified. It multiplies padded matrix `mmd.ap` by `mmd.bp` and writes the result to padded matrix `mmd.gp`. In `mm_tiled_hw01` each thread operates on its own set of rows, computing a $t_1 \times t_3$ tile d_3/t_3 times to compute one set of t_1 rows. This potentially reuses a $t_1 \times d_2$ section of matrix a d_3 times. The routine is described in more detail further below. In the unmodified assignment `mm_tiled_hw02p1` and `mm_tiled_hw02p2` are identical to `mm_tiled_hw01`. In a correct solution to Problem 1, each thread spawned for `mm_tiled_hw02p1` should operate on its own set of columns. For each group of t_3 columns it should compute a $t_1 \times t_3$ tile d_1/t_1 times, potentially reusing a $d_2 \times t_3$ section of matrix `b`.

Values of t_1 and t_3 are chosen so that when computing a $t_1 \times t_3$ section of matrix `g` the section (declared as `ee`) can be kept in vector registers. Keeping the $t_1 \times t_3$ section of g in registers while it is being computed helps the rate of computation approach saturation (which on the lab computers means two vector instructions per cycle per core). This would not be achieved if each multiply/add instruction were preceded by a vector load and followed by a vector store.

When routine `mm_tiled_hw02p2` is correctly solved each thread will compute a $m_1 \times m_3$ macro tile consisting of $\frac{m_1}{t_1}\frac{m_3}{t_3}$ tiles of size $t_1 \times t_3$. The goal is to see if there is benefit to choosing m_3 such that a $d_2 \times m_3$ section of `b` can fit in the level 2 cache. Further details on `mm_tiled_hw01` and the requirements for `mm_tiled_hw02p1` and `mm_tiled_hw02p2` are described further below.

Routine `mm_do(app,d1,d2,d3,pad)` will construct the $d_1 \times d_2$ and $d_2 \times d_3$ arrays and multiply them in several different ways, each multiplication will use `app.nt` threads. After preparing the `app` structure and constructing the arrays `mm_do` calls `mm_simple` to compute the correct answer.

Routine `mm_do` will then call `mm_tiled_do<t1,t2,t3>(app)` multiple times, each time with a different tile shape, specified by `t1`, `t2`, and `t3`, and for a different kernel using enumeration constants `KV_HW01`, `KV_HW02p1`, `KV_HW02p2`. The values of `t1` and `t3` are chosen so that there are enough vector registers for a $t_1 \times t_3$ tile.

What routine `mm_tiled_do` does depends upon which kernel it is supposed to launch. In all cases it sets `mmd.m3` to the desired macro tile width, m_3. For `mm_tiled_hw01` $m_3 = d_3$ and for `mm_tiled_hw02p1` $m_3 = t_3$. (Those following things closely will see why m_3 was set to those values.) It then calls `mm_tiled_do_1`. For Problem 2, `mm_tiled_hw02p2`, `mm_tiled_do_1` is called several times, each with a different value of m_3.

Routine `mm_tiled_do_1` calls `mm_tiled_do_sample` multiple times (determined by the value of `mmd.app.n_samples`, set to `5` in the unmodified assignment). Each call of `mm_tiled_do_sample` performs a matrix multiplication. A line of output is only shown for the multiplication with the median execution time. This should reduce the variation in execution times, though it still can be high, especially for the smaller matrices.

Routine `mm_tiled_do_sample` spawns `app.nt` threads, each thread starts with routine `mm_tiled<t1,t2,t3>(tid,&mmd)`, where `tid` is a thread ID, which can range from `0` to `app.nt-1`. After the threads complete `mm_tiled_do_sample`

checks the product for correctness, and then constructs a row of the table. Only the row for the execution with the median run time is printed.

Routine `mm_tiled<t1,t2,t3>` then calls `mm_tiled_hw01`, `mm_tiled_hw02p1`, or `mm_tiled_hw02p2`. The solution for this assignment should be put in `mm_tiled_hw02p1` and `mm_tiled_hw02p2`, however feel free to modify other code, for example, to run different configurations or to print additional data in the table.

Using `hw02`

The assignment program, `hw02`, optionally takes a single argument, the number of threads to spawn. If it is run without an argument the number of threads will be set to the number of cores (counting all CPUs) minus two. Unlike Homework 1, in Homework 2 the core count isn't doubled if SMT (*symmetric multiprocessing* which Intel calls *hyperthreading*) is turned on.

The program will first print information about the CPU, see the sample below.

```
CPU model Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz
Node has 2 CPUs, CPU has 20 cores, total 40 cores. Measured clock 3.40 GHz.
(This run spawns 40 threads.)
L1 Data Per Core     48 kiB, 12-way, line size 64 bytes.
L2 Unif Per Core   1280 kiB, 20-way, line size 64 bytes.
L3 Unif Per Chip  30720 kiB, 12-way, line size 64 bytes.
Vector width 512 bits, 32 vector registers, and
2 vector unit(s) per core. (Assumed.)
Data bw 33.870 GB/s (measured). For 40 threads:  comp/comm = 514.0
```

The CPU model is printed on the first line. The second shows the number of cores, the reported clock frequency, and the number of threads requested on the command line. If no argument is given the number of threads is set equal to the number of cores minus two.

The clock frequency given above after the phrase "cores at", is the reported clock frequency after performing a bandwidth test, 3.40 GHz above. Note that this frequency differs from the nominal frequency reported with the model name `CPU @ 2.30GHz` in the top line. The reported clock frequency can be higher or lower than the nominal frequency. In this case it was higher because before `hw02` was run the CPU was relatively cool, so when it performed the bandwidth test it was able to run at its high ("turbo") frequency for the duration of the bandwidth test without getting hot enough to end the sprint.

Normally this reported frequency will be used throughout the run to assess how close the multiplications were to peak performance. If `re_check_cpu_clocks` is set to `true` then the CPU clock frequency will be checked after each matrix multiplication.

The line starting `Vector width` shows the number of bits in each vector register, the number of vector registers, and the number of vector units per core. The sample values above are the same as the lab computers, which implement AVX512 instructions. Recent consumer-level processors use AVX2 instructions, which operate on 256-bit registers and have just 16 registers and one vector unit per core.

The line starting `Data bw` shows the measured GPU-to-memory data bandwidth. The line also shows the computation to communication ratio, taking into account data bandwidth, clock frequency, and the vector capabilities. A value of 514 indicates that when both the vector units and off-chip bandwidth are at saturation (operating at their maximum rate) 514 FP operations will be performed for each operand moved from or to memory when 40 cores are used. The `comp/comm` ratio is computed based on the requested number of threads, not the number of cores.

Next the program will multiply matrix pairs of varying sizes. Each pair will first be multiplied by a simple routine, and its product will be treated as correct. Then it will be multiplied by a tiled routine called for a variety of tile sizes.

For each matrix pair the program starts by printing the following information:

```
Matrix 2048  2048  2048  4096.  Duration simple: 899.435 ms
Stride 2048  2064  2048  4112  = 2048 x 4112. 18 thds.  HW01
```

The line starting `Matrix` shows the sizes of the `a` and `b` matrices, 2048×2048 for `a` and 2048×4096

for `b`. `Duration simple` indicates the time needed by `mm_simple` to compute the product (using all cores, it ignores the requested number of threads).

The line starting `Stride` shows the strides chosen for the matrices. That is followed by the number of threads requested, 18 in the example, and the routine (abbreviated) for which results are to be shown, `HW01`, in the example above. Next, the matrix pairs will be multiplied using various tile shapes and for `HW02p2` various macro tile shapes. A table will be printed with one row per tile shape. The table is shown twice, first in the order in which they were run, and the second sorted showing the fastest tile shape first:

```
-------Tile------- --L2 Occ---         -FP Throughput- Insn/ Util
 t1  t3    m1   m3 m1*d2 d2*m3  Dur/ms  GFLOPs  % Peak Vec I  Mes
 16  16    16 4096  0.10 25.60   79.08  217.25   11.09  2.62   82
  8  32     8 4096  0.05 25.60   83.41  205.96   10.52  2.36   89
  4  64     4 4096  0.03 25.60  139.17  123.44    6.30  2.23   98
  2 128     2 4096  0.01 25.60  240.83   71.34    3.64  2.36   93
  1 256     1 4096  0.01 25.60  407.02   42.21    2.16  2.18   97
** Sorted by Execution Time **
-------Tile------- --L2 Occ---         -FP Throughput- Insn/ Util
 t1  t3    m1   m3 m1*d2 d2*m3  Dur/ms  GFLOPs  % Peak Vec I  Mes
 16  16    16 4096  0.10 25.60   79.08  217.25   11.09  2.62   82
  8  32     8 4096  0.05 25.60   83.41  205.96   10.52  2.36   89
  4  64     4 4096  0.03 25.60  139.17  123.44    6.30  2.23   98
  2 128     2 4096  0.01 25.60  240.83   71.34    3.64  2.36   93
  1 256     1 4096  0.01 25.60  407.02   42.21    2.16  2.18   97
```

The first group of columns show the tile and macro tile shape. Note that t_2 is not shown, its value is always 16. The values under `t1`, `t3`, and `m3` are chosen by the code calling the kernels. The value under `m1` is chosen by the kernel.

The values under `m1*d2` shows the size of a section m_1 rows by d_2 columns of the `a` matrix divided by the size of the level 2 cache. A value of `0.10`, for example, indicates that it is one tenth the L2 cache size. Similarly, the values under `d2*m3` show how large a d_2 row by m_3 column section of matrix `b` is compared to the L2 cache. In the output above it is 25.6 times as large. The code in `mm_tiled_hw01` iterates across columns, and so the value under `m1*d2` is relevant for interpreting performance. When Problem 1 is correctly solved the value under `d2*m3` will be relevant for interpreting `mm_tiled_hw02p1`. For Problem 2 the values under both columns will be relevant.

The `Dur/ms` column shows execution time in milliseconds. The pair of columns under `FP Throughput` show the floating point throughput in billions of floating-point operations per second, `GFLOPs`, and as a percent of peak performance of the vector units. The peak performance is based on the clock frequency reported at the beginning of the run. If for some reason the clock goes up or down, those peak numbers will be wrong, sort of. A value of 100 for peak performance is ideal.

The column headed `Insn/ Vec I` (that's over two lines) show the number of executed instructions scaled to the number of vector multiply/add instructions. Lower values are better, but a value of one or lower is impossible. Let w denote the number of elements per vector register. For the lab machines $w = 16$ and for lower-line processors $w = 8$. (The assignment code has not been tested on many machines and will not not work if the vector registers are less than 256 bits.) Note that it takes $d_1 d_2 d_3$ multiply/add operations to multiply the matrices. If vector instructions are used, and each vector holds w elements then $d_1 d_2 d_3 / w$ vector multiply/add instructions are needed, plus of course instructions to load and store values, keep track of loop iterations, etc. Let n_I denote the number of instructions executed when computing a matrix multiplication. Dividing these two scales the number of instructions to the lower bound on the number needed: $n_\mathrm{I} \frac{w}{d_1 d_2 d_3}$. A value of 1 would mean that there are only multiply/add instructions, which is impossible. Values less than 1 are of course also impossible. A value of 2.6 indicates that for each multiply/add instruction there are 1.6 other instructions. The value is shown in the column is $n_\mathrm{I} \frac{w}{d_1 d_2 d_3}$.

Note that the scaled instruction count, `Insn/ Vec I`, varies based on the tile size. This is because the code was written so that the compiler can unroll the `c`, `r`, and `k` loops in the kernels and for some values of

t_1 and t_3 the compiler can compute memory addresses with fewer instructions (or more reuse) than others.

The `Util Mes` column is measured utilization shown as a percentage. A value of 100% means all threads are busy all the time. A value of 50% means on average half the threads are busy.

Description of `mm_tiled_hw01`

Routine `mm_tiled_hw01` is a simplified version of the solution to Homework 1, see the excerpt below. The outer loop, `rr`, iterates over rows. The loop start, `tid*t1`, and increment, `mmd.app.nt*t1`, are chosen so that no two threads have the same value of `rr`. In each `rr` iteration a thread computes a series of $t_1 \times t_3$ tiles, the first at column 0 (`cc=0`), the second at column t_3 (`cc=t3`), the third at $2t_3$, and so on. The `cc` loop iterates d_3/t_3 times.

```
for ( int rr = tid * t1;  rr < d1;  rr += mmd.app.nt * t1 )
  for ( int cc = 0;  cc < d3;  cc += t3 ) {

      // One Complete t1 * t3 Output Tile (of g) Computed Below
      vf ee[t1][t3/vec_wid_elts]{};   // Output tile.
      for ( int k = 0;  k < d2; k++ )
          {
            vf* brow = (vf*)assume_aligned<64>( &bv[ k*s3/vec_wid_elts ] );
            for ( int c = cc;  c < cc + t3;  c += vec_wid_elts )
              for ( int r = rr; r < rr + t1; r++ )
                ee[r-rr][(c-cc)/vec_wid_elts] +=
                  a[ r*s2a + k ] * brow[ c / vec_wid_elts ];
          }
      // Write completed t1 * t3 tile to g.
      for ( int r = rr;  r < rr + t1;  r++ )
        for ( int c = cc;  c < cc + t3;  c += vec_wid_elts )
          gv[ r * s3 / vec_wid_elts + c / vec_wid_elts ] = ee[r-rr][(c-cc)/vec_wid_elts];
}
```

The value of t_3 will always be a factor of d_3. Furthermore, the value of t_3 will always be a multiple of the number of array elements that can fit in a vector register on an AVX512 system (the lab computers) and AVX2 (midrange and lower Intel chips). So for the lab t_3 will be a multiple of 16 and if run on AVX2 chips t_3 will be a multiple of 8. The values of t_1 and t_3 are chosen so that there are enough vector registers for a $t_1 \times t_3$ tile. This is important for matrix multiplication because a good implementation will be compute-bound and so a high proportion of instructions should be multiply/adds. This would not be achieved if each multiply/add instruction were preceded by a vector load and followed by a vector store.

Note that each iteration of the `cc` loop applies a $t_1 \times t_3$ tile to compute `g` somewhere on row `rr`. Each iteration reads the same $t_1 \times d_2$ elements of `a`. If these elements of `a` could be kept in the L2 cache then execution would be faster since the access to `a` in the second `cc` iteration would hit the L2 cache. Based on plotted data (under column `m1*d2`) this is certainly the case. However because `d2*m3` (where $m_3 = d_3$ for the `mm_tiled_hw01`) is 25 times larger than the L2 cache the values brought to the L2 cache for one `rr` iteration will likely not be there for a subsequent `rr` iteration.

Problem 1: Initially `mm_tiled_hw02p1` is identical to `mm_tiled_hw01`.

Modify `mm_tiled_hw02p1` so that a thread computes a $t_1 \times t_3$ tile along the d_1 rows in a group of t_3 columns. That is, after a thread computes a $t_1 \times t_3$ tile consisting of rows r to $r + t_1 - 1$ and columns c to $c + t_3 - 1$, it should next compute a tile consisting of rows $r + t_1$ to $r + 2t_1 - 1$ and columns c to $c + t_3 - 1$. The initial row for a thread is $r = 0$, the initial column should be chosen based on the thread id. (In the unmodified code a thread computes a $t_1 \times t_3$ tile along the d_3 columns in a group of t_1 rows.)

Because in most cases $t_3 > t_1$ a correctly solved routine should have better cache performance and so run faster.

- [ ] Modify `mm_tiled_hw02p1` so that a thread applies tiles across the columns of a row, as described above.

- [ ] Discuss whether the difference in performance is explained by the program output, in particular the data under `L2 Occ` and `Insn/ Vec I`.

Problem 2: A thread in `mm_tiled_hw01` computes a $t_1 \times t_3$ tile along the columns of a group of t_1 rows, so to achieve reuse the cache would need to hold $t_1 \times d_2$ elements of `a`. If this reuse were achieved the number of times the `a` matrix would be read (to L2) would be once and the number of times the `b` matrix were read would be d_1/t_1 (or $\frac{d_1}{nt_1}$ times per thread, where n is the number of threads). In a correctly solved `mm_tiled_hw02p1` a thread computes a $t_1 \times t_3$ tile along rows in a group of t_3 columns, so to achieve reuse the cache would need to hold $d_2 \times t_3$ elements. Each element of `b` would be read once, and each element of `a` would be read d_3/t_3 times (counting all threads).

Modify `mm_tiled_hw02p2` so that each thread computes `g` in one or more *macro tiles* of size $m_1 \times m_3$. Note that each macro tile consists of $\frac{m_1}{t_1}\frac{m_3}{t_3}$ tiles of size $t_1 \times t_3$. The value of m_3 is given in variable `mmd.m3`, and should not be changed. The value of m_1 should be chosen as part of the problem, and of course the chosen value should be a multiple of t_1. Suppose $m_1 = 2t_1$. First the code would compute m_3/t_3 tiles of size $t_1 \times t_3$ all on the same t_1 rows. If the L2 cache is large enough to hold $d_2 \times m_3$ elements of `b` then those elements will be found in the cache when doing the second set of m_3/t_3 tiles. If m_3 is too large this benefit would be lost. Of course if $m_1 = t_1$ the benefit is lost too. One simple solution is to set $m_1 = d_1$, but that would result in workload imbalance when d_3/m_3 is not a multiple of n (the number of threads).

- [ ] Modify `mm_tiled_hw02p2` so that it computes macro tiles as described above.

- [ ] Try to do this in a way that improves on `mm_tiled_hw02p1` by taking advantage of the fact that more values of `a` are being reused when $m_3 > t_3$.

- [ ] Try to avoid workload imbalance.

- [ ] Discuss whether the difference in performance is explained by the program output, in particular the data under `L2 Occ` and `Insn/ Vec I`?