LSU EE 7722 **Homework 1** **Due: 23:59 CDT 10 March 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/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 `.../2025/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 `.../2025/hw01` 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 `hw01` and one with the suffix `-debug`, such as `hw01-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 `hw01`, 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 `./hw01`. It should produce extensive output starting something like

```
CPU model Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz
CPU has 40 cores at 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 65.787 GB/s (measured). For 40 threads:  comp/comm = 264.6

Matrix 256  256  256  256.  Duration simple: 3.923 ms
Stride 256  256  256  256  = 256 x 256. Padding Not Requested
---Tile---         -FP Bandwidth-- Utiliz- ---E--- ---V---- ---C--- ---P---
 t1 t2  t3  Dur/ms  GFLOPs  % Peak Exp Mes %L2 %BW %L2  %BW %L2 %BW %L2 %BW
  1 16   1    7.15    2.35    0.05   0   0   0  29   0    0   0   0   0   0
  4 16   4    5.78    2.90    0.07   0   0   0   9   0    0   0   0   0   0
  8 16   8    5.54    3.03    0.07   0   0   0   5   0    0   0   0   0   0
  8 16  16    2.82    5.94    0.14   0   0   0   7   0    0   0   0   0   0
```

And finally after lots of output finishing

```
256 16   8  336.62  102.07    2.34   0   0   1  91   0    0   0   0   0   0
256 16  16  146.72  234.18    5.37   0   0   1 108   0    0   0   0   0   0
256 16  32   82.92  414.39    9.50   0   0   3 101   0    0   0   0   0   0
256 16  64   78.25  439.11   10.07   0   0   5  60   0    0   0   0   0   0
256 16 128  376.42   91.28    2.09   0   0  10   8   0    0   0   0   0   0
```

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/`.

A solution to these problems requires some understanding of the CPU hardware structure, in particular how array references such as `a[1000]` are converted into memory addresses, how memory addresses are mapped to cache sets, and about the multi-level caches provided on the Intel CPUs (and other CPUs). Some of that material is reviewed in this assignment.

Assignment Overview

The code in `hw01` 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 $[0, 1]$.

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.

Despite using an OpenMP pragma for parallelization, `mm_simple` is slow. Also provided in the assignment is a tiled routine, in `mm_tiledr`. The assignment code will run it at various tile sizes and report results. Those results will show great variation in run times. Part of this assignment will be to explain the run times, and to fix a problem that plagues matrices with power-of-two dimensions.

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`. Routine `mm_tiled_do` spawns `app.nt` threads, each thread starts with routine `mm_tiled<t1,t2,t3>(tid,&app)`, where `tid` is a thread ID, which can range from `0` to `app.nt-1`. After the threads complete `mm_tiled_do` checks the product for correctness, and then prints data about the run (a row of the table shown in the output examples).

Routine `mm_tiled<t1,t2,t3>` then calls `mm_tiledr<t1,t2,t3,vec\_bits,n\_bytes>`, using the correct vector register size and count. Routine `mm_tiledr` performs the tiled matrix multiplication.

Using `hw01`

The assignment program, `hw01`, 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 maximum concurrency, which is the number of cores if SMT (hyperthreading) is turned off.

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

```
CPU model Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz
CPU has 40 cores at 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 70.306 GB/s (measured). For 40 threads:  comp/comm = 247.6
```

The CPU model is printed on the first line. The second shows the number of cores *as of this writing it actually shows the reported concurrency*, 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.

The clock frequency given above after the phrase "cores at", is the reported clock frequency after performing a bandwidth test, 3.4 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 `hw01` 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 too hot.

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 247 indicates that when both the vector units and off-chip bandwidth are at saturation (operating at their maximum rate) 247 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: 4958.823 ms
Stride 2048  2048  2048  4096  = 2048 x 4096. Padding Not Requested
```

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 in a solution to a problem below if `Padding Requested` is shown. If `Padding Not Requested` is shown then the strides should match the matrix dimensions. In the unmodified assignment the strides will be equal to the matrix dimensions in either case. Strides are shown for `a`, `b`, and `g`. They are shown for `g` because they don't need to be consistent with `a` and `b`.

Next, the matrix pairs will be multiplied using various tile sizes. A table will be printed with one row per tile size:

```
---Tile---          -FP Throughput- Clk- Utiliz- ---E--- ---V---- ---C--- ---P---
 t1 t2  t3  Dur/ms  GFLOPs  % Peak  MHz Exp Mes %L2 %BW %L2  %BW %L2 %BW %L2 %BW
  8 16   8   94.12  182.53    4.41 3230   0   0   0 520   0    0   0   0   0   0
  8 16  16   61.51  279.30    7.77 2808   0   0   0 597   0    0   0   0   0   0
  8 16  32   54.36  316.05    8.80 2805   0   0   0 563   0    0   0   0   0   0
  8 16  64   61.16  280.89    7.82 2805   0   0   0 451   0    0   0   0   0   0
 16 16   8   96.61  177.83    4.88 2845   0   0   0 380   0    0   0   0   0   0
 16 16  16   63.61  270.09    7.52 2807   0   0   0 385   0    0   0   0   0   0
```

The first group of columns show the tile shape. Note that t_2 is always 16. 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 columns under `Utiliz`, utilization, are to show expected and measured utilization. In the unmodified assignment they are zero, when Problem X is correctly solved they will show the expected and the measured percentage of time threads are busy.

Their are four column pairs headed with a single letter, `E`, `V`, `C`, and `P`. The letter refers to a place in `hw01.cc`, for example, `Point E`. The value under `L2` shows the percentage of level 2 cache needed for perfect reuse by the code at Point x in the program. The value under `BW` shows the percentage of off-chip (to memory) bandwidth consumed if there was perfect reuse at Point x and nowhere else. See Problem X.

Problem 1: One reason that performance of our matrix multiplication suffers is due to load imbalance: some threads sit idle while others are busy. Assume that the number of threads, n_τ, is equal to the number of cores. There are two causes for load imbalance in our code. One is due to the way work is assigned to threads. Each thread is assigned at most $\lceil \frac{d_1}{t_1 n_\tau} \rceil t_1$ rows of the matrix. Work is evenly divided among threads when $\frac{d_1}{t_1 n_\tau}$ is an integer. So, for $d_1 = 1024$, $t_1 = 16$, and $n_\tau = 32$ we have $\frac{1024}{16 \times 32} = 2$, meaning each thread is assigned $2 \times 16 = 32$ rows. But suppose $d_1 = 1024$, $t_1 = 64$, and $n_\tau = 32$. Then $\frac{1024}{64 \times 32} = \frac{1}{2}$. In this case half the threads will be assigned $t_1 = 64$ rows and the rest will be assigned zero rows. Half of our performance potential is lost (assuming our code is compute bound, which our matrix multiply is if we get the tiling right).

Define *expected utilization* as follows. Let w_i denote the amount of work assigned to thread i. Then $U_E = \frac{\sum_{0 \le i < n_\tau} w_i}{n_\tau \max_{0 \le i < n_\tau} w_i}$. Define a second metric, *measured utilization* as follows. Let b_i denote the measured amount of time thread i is busy (presumably doing useful work). Then $U_M = \frac{\sum_{0 \le i < n_\tau} b_i}{n_\tau \max_{0 \le i < n_\tau} b_i}$.

Modify `mm_tiled_do` so that the expected utilization is assigned to `util_expected` and the measured utilization is assigned to `util_measured`. Note that an ideal value is 1, not 100. In the unmodified assignment both are assigned zero.

The values of these variables are printed, as a percentage, in the table under the `Utiliz` group of columns. There is no need to modify the code printing their values in the table.

To compute the measured utilization it will be necessary to measure the execution time of each thread. The best place to do this is inside `mm_tiledr`. There is a per-thread array, `misc\_data`, that can be used to carry duration and other information. That array is already used to carry a value of `t1v` from `mm_tiledr` to `mm_tiled_do`, use it as an example.

- [ ] Assign the expected utilization to `util_expected`.
- [ ] Its value should be computed using matrix dimensions, tile shape, and the number of threads.
- [ ] Assign the measured utilization to `util_measured`.
- [ ] Compute this by measuring the execution time of each thread using the `Thd_Misc_Data` structure to hold execution times, and any other data needed.

The utilization values should reveal that the tiling used in this assignment does not work well for smaller arrays, especially considering measured utilization.

Problem 2: The point of tiling is to organize the access of data to make good use of the cache.

For a particular tiling there are two quantities of interest: the amount of cache needed, and the amount of data that will need to be transferred if there is sufficient cache. In this problem those quantities will be computed and shown in the table.

If this were a classic $t \times t$ tiling then the amount of cache needed would be between $3t^2$ and $t^2 + t + 1$ elements, depending on how it was done. With this much cache each element of `a` and `b` would be reused $t - 1$ times in each tile. To compute each tile $2t^2$ data elements would be read and t^2 elements of cache would be used for the output array tile. The total number of tiles computed would be $(d/t)^3$ and so $2d^3/t$ data elements would be read and d^2 written. With such a tiling t is chosen to be at least the computation to communication ratio of the device and hope there is enough cache.

The assignment code is different than classic tiling for several reasons. For one thinig there are three tile dimensions, t_1, t_2, and t_3.

The code in `mm_tiledr` includes comments `Point E`, `Point P`, `Point C`, and `Point V`. In `mm_tiled_do` are variables starting with `point_` that are to be assigned values as described below, except the variables for Point E, which are assigned correctly and are described below.

Variable `point_e_cache_size_min_elts` is assigned to the minimum cache size needed so that elements of `a` and `b` are perfectly reused in the code after Point E. That is, there will be at most one miss to any particular element of `a` and `b` if there is this much cache. For example, if `a[7]` is accessed multiple times, only the first access to `a[7]` will miss the cache.

Consider the code at Point E:

```
for ( int kk = 0;  kk < d2;  kk += t2 ) {
    /// Point E
    for ( int k = kk;  k < kk + t2;  k++ )
      for ( int r = rr;  r < rr + t1;  r++ )
        for ( int c = cc;  c < cc + t3;  c++ )
          ee[r-rr][c-cc] += a[ r*d2 + k ] * b[ k*d3 + c ];
  }
```

The cache space is needed for `ee`, `a`, and `b`. First, observe that a tile t_1 rows high and t_2 columns wide is read from `a`, for a total of $t_1 t_2$ elements. However the amount of cache needed is less than that. In fact because of the order of the loops, each element of `a` is read, used t_3 consecutive times, and not used again. So, only one element of space is needed for `a`. For `b` only t_3 elements of space are needed. In the `c` loop t_3 different elements of `b` are read. Those same elements are accessed in each iteration of the `r` loop. But each `k` loop iteration uses different elements of `b` so only t_3 storage locations are needed. Finally, $t_1 t_3$ elements are needed for `ee`. These values have been written before Point E, and are needed after Point E, so space is needed for all of them.

Based on this:

```
const size_t point_e_cache_size_min_elts =
  t1 * t3  // For ee (though registers might be used for ee)
  + t3     // For b[ k*d3 + c ].
  + 1;     // For a[ r*d2 + k ].
```

Variable `point_e_xfer_elts` should be set to the maximum number of elements of `a`, `b`, and `g` moved between memory and caches if perfect reuse is achieved in Point E.

First, consider `a` and `b`. As stated earlier, the code at Point E uses $t_1 t_2$ elements of `a` and $t_2 t_3$ elements of `b`. Because there is perfect reuse those values will be transferred once, for a total size of $t_1 t_2 + t_2 t_3$. The code at Point E executes $\frac{d_1}{t_1}\frac{d_2}{t_2}\frac{d_3}{t_3}$ times. (The $\frac{d_1}{t_1}$ factor accounts for all threads. It is $\frac{d_1}{n_\tau t_1}$ for each thread.) So the total xfer including `a` and `b` is $\frac{d_1}{t_1}\frac{d_2}{t_2}\frac{d_3}{t_3}(t_1 t_2 + t_2 t_3)$.

Since we are assuming sufficient cache there are no transfers from and to memory for `ee`. But, `ee` is written to `g`. Since each element of `g` is written once the total transfer is $d_1 d_3$. Based on this:

```
const size_t point_e_n_executions = ( d1 / t1 ) * ( d2 / t2 ) * ( d3 / t3 );
const size_t point_e_xfer_elts =
  t1 * t3 * ( d1 / t1 ) * ( d3 / t3 )  // Yes, it's just d1 * d3.
  + point_e_n_executions * ( t1 * t2 + t2 * t3 );
```

The value of `point_e_cache_size_min_elts` is used to compute the value of `L2` under the `E` group in the table. The value is the percentage of the L2 cache that is needed for perfect reuse, computed by dividing `point_e_cache_size_min_elts` by the level 2 cache size (in units of elements).

The value of `point_e_xfer_elts` is used to compute the value under `%BW`. The number of elements that could have been transferred to or from memory during the matrix multiplication, `data_limit_elts`, is computed. Then `point_e_xfer_elts` is divided by `data_limit_elts`. If the value is, say 0.5 (or 50%) then half of the bandwidth would be needed to transfer `point_e_xfer_elts`. If the value is 2 (or 200%) then twice the bandwidth would be needed, which means that fewer than `point_e_xfer_elts` were actually transferred during execution.

```
const double data_limit_elts = data_bw_eps * duration_s;
mmd.table.header_span_start("E");
mmd.table.entry( "%L2", "%3.0f",
```

```
                    1e2 * point_e_cache_size_min_elts / l2_size_elts );
  mmd.table.entry( "%BW", "%3.0f", 1e2 * point_e_xfer_elts / data_limit_elts );
  mmd.table.header_span_end();
```

For Point E, the values under `L2` will be very low, meaning not much cache is needed. The values under `BW` will be greater than 100%, meaning that fewer elements were transferred and so there must be some other kind of reuse.

Point V is similar to Point E, except it is in the vector part of the code. Also, Point V operates along all of d_2, not just a length t_2 tile. The code at Point V operates on a $t_{1v} \times t_3$ tile. The value of t_{1v} is chosen so that there will be enough vector registers for the `ee` array. Each element of the `ee` array is a vector register of `vec_wid_elts` elements. The number of columns in `ee` is `t3/vec_wid_elts` vectors or `t3` elements. Array `a` is accessed in the same way as at Point E, as scalars. Array `b` is accessed as vectors. This access is done through pointer `brow`. For purposes of analysis consider `brow[ ccc / vec_wid_elts ]` equivalent to `bv[ ( k*d3 + ccc ) / vec_wid_elts ]`. This accesses elements in row `k`, from column `ccc` to column `ccc + vec_wid_elts-1`.

☐ Assign variable `point_v_cache_size_min_elts` to the minimum cache size in units of elements needed so that elements of `a` and `b` are perfectly reused in the code after Point V until the end of the block.

Point C is just above Point V. Point C handles the full $t_1 \times t_3$ tile. The amount of cache needed for Point C is **not** just `t1/t1v` times the amount needed for Point V.

☐ Assign variable `point_c_cache_size_min_elts` to the minimum cache size in units of elements needed so that elements of `a` and `b` are perfectly reused in the code after Point V until the end of the block.

☐ Assign variable `point_c_xfer_elts` to the the maximum number of elements of `a`, `b`, and `g` moved between memory and caches if perfect reuse is achieved in Point C.

Points E, C, and V are at places in the code that operate on tiles. Point P is different because the code iterates a $t_1 \times t_3$ tile along an entire row. The questions below ask about the amount of cache needed to achieve perfect reuse for Point P. In effect, that makes it a $t_1 \times d_3$ tile, which potentially is very wide. In such a tile elements of `a` are reused d_3 times, which sounds great, but elements of `b` are reused only t_1 times. The cache needed to achieve this could have been used more effectively in a tile closer to square in shape. That said:

☐ Assign variable `point_p_cache_size_min_elts` to the minimum cache size in units of elements needed so that elements of `a` and `b` are perfectly reused in the code after Point P until the end of the block.

☐ Assign variable `point_P_xfer_elts` to the the maximum number of elements of `a`, `b`, and `g` moved between memory and caches if perfect reuse is achieved in Point P.

Problem 3: In the unmodified code the performance on larger matrices suffers due to conflict misses caused by power-of-two strides. Consider the multiplication of a 1024×1024 with a 1024×1024 matrix. On a lab computer with Xeon Silver 4316 the best tiling reaches about 21% of peak performance. But the multiplication of a 2048×2048 with a 2048×4096 matrix reaches only 11.4% of peak. One might assume that there is not enough cache for the larger matrix, but as other problems in this assignment will show there's plenty. The problem is that cache space is wasted due to conflict misses.

The caches used in most processors, including the Intel processors in the lab, are *set-associative*. The cache manages memory addresses in units called *lines*. A typical line size is 64 bytes, meaning that 64 memory addresses map to the same line. For such a line size memory addresses $1200_{16}, 1201_{16}, 1202_{16}, \ldots 123f_{16}$ are all on the same line, and address 1240_{16} is on a different line. Note that $1200_{16} + 64_{10} = 1200_{16} + 40_{16} = 1240_{16}$.

Each memory address is mapped to a *set*. The *associativity* of a cache (number of *ways*) is the number of different lines a set can hold. Let $L = 2^l$ denote the line size, $S = 2^s$ denote the number of sets, and a denote the associativity.

For address A define the *tag* of A to be $C_{\text{tag}}(A) - \lfloor \frac{A}{SL} \rfloor$, where S is the number of sets and L is the line size. Define the *index* of A to be $C_{\text{index}}(A) = \lfloor \frac{A}{L} \rfloor \bmod S$. The value of the index is in the range 0 to $S-1$.

Two addresses are in the same set if they have the same index. Two addresses are part of the same line if they have the same tag and index. A set in an a-way cache can hold at most a lines. When a processor issues a load or store instruction for address A is will perform a lookup for a line in set $C_{\text{index}}(A)$ with a tag equal to $C_{\text{tag}}(A)$. If such a line is found the lookup will be said to *hit*, otherwise it is a *miss*. On a miss the cache controller will issue a *read request* for the data, either in the next level of the cache or memory. When the requested data arrives it will be placed in set $C_{\text{index}}(A)$. If that set had already held a lines then one of those lines would be *replaced* or *evicted* (the two terms are similar) by the arriving line. The choice of line to replace is determined by the *replacement* policy. A common replacement policy is *least-recently used (LRU)*, in which the line accessed longest ago is replaced.

Suppose the line size is $L = 2^4 = 16$ (one hexadecimal digit), the number of sets is $S = 2^8 = 256$ (two hexadecimal digits), and the cache is $a = 3$ or three-way set associative. Because the line size and number of sets are both multiples of 16 one can determine the set and tag by looking at an address in hexadecimal. Address 1234_{16} maps to set 23_{16} (the second and third hex digit), or put another way $C_{\text{index}}(1234_{16}) = 23_{16}$. The tag of this address is 1, it is the value of the digits to the left of the index. Addresses 1234_{16} and 1238_{16} are both in set 23_{16} and because they differ only in the least significant bits are the same line.

Consider a thread that executes loads to the following addresses

$$1000_{16}, 1030_{16}, 2000_{16}, 2004_{16}, 3000_{16}, 4000_{16}, 5000_{16}, 103c_{16}$$

in the cache described above. Address 1030_{16} is in set 3, and all the others are in set 0. Starting from a cold (empty) cache and after executing loads for $1000_{16}, 1030_{16}, 2000_{16}, 2004_{16}$ set 0 of the cache will hold two lines, one for 1000_{16} (spanning addresses 1000_{16} to $100f_{16}$) and one for 2000_{16}, and set 3 will hold one line, for 1030_{16}. No problem, because the cache is 3-way. After 3000_{16} set 0 will hold 3 lines. Next 4000_{16} is accessed. This line will be put into set 0, but some other line will have to be evicted. Assuming an LRU replacement policy line 1000_{16} will be evicted. Next, 5000_{16} is accessed, evicting 2000_{16}. Finally for access, $103c_{16}$ the lookup to set 3 will hit since a line with tag 1 is there, the line brought in by the earlier access of 1030_{16}.

The cache above has 256 sets. A really bad access pattern is $1000_{16}, 2000_{16}, 3000_{16}, \ldots$, because all of the addresses are in set 0 but have different tags. Though the capacity of the cache is 3×256 lines for this sequence the cache only holds 3 of them. Is this some freak unlucky occurrence? Not for us. Consider accesses to a $d_1 \times d_2$ matrix of 4-byte elements. Let $a_{r,c}$ denote the element in row r and column c. Our code actually stores the data in a one-dimensional array, and computes the index as `a[ r*d1 + c ]`. Suppose the address of $a_{0,0}$ is 1000_{16}. Then the address of $a_{r,c}$ would be $1000_{16} + 4(d_1 r + c)$. Let $d_1 = 1024$. Then $4d_1 = 4096 = 1000_{16}$. So the address of $a_{1,0}$ is 2000_{16}, the address of $a_{2,0}$ is 3000_{16}, and so on. All of these addresses are in the same set. Can this cause problems *for us?*

Consider the level 2 cache provided for each core in the Xeons in the lab. They are 20-way set associative, have a line size of $L = 2^6 = 64$ bytes and $S = 2^{10} = 1024$ sets. Their total capacity is $aSL = 20 \times 2^{10} 2^6\,\text{B} = 20 \times 2^{16}\,\text{B} = 1280\,\text{kiB}$. In this cache there are 6 line bits and 10 set bits, meaning that the tag starts at the 16th bit, or conveniently for us, the fith hexadecimal digit. That is, the tag of address 7654321_{16} is 765_{16}. Consider the execution of our tiled code from routine `mm_simple` on this cache:

```
for ( int r=0; r<d1; r++ )
  for ( int c=0; c<d3; c++ )
    {
      elt_t e = 0;
      for ( int k=0; k<d2; k++ ) e += a[ r*d2 + k ] * b[ k*d3 + c ];
      g_simple[ r*d3 + c ] = e;
    }
```

The `k` loop will access `d2` elements of `b`, each element on its own line. We would hope that in the second iteration of the `c` loop accesses to `b` should hit the cache because its likely that element `b[ k*d3 + 0 ]` and `b[ k*d3 + 1 ]` are on the same line. *Note to non-native English speakers: The phrase "we would hope" is*

preparing us for disappointment. Consider again square arrays with $d_1 = d_2 = d_3 = 2^{10} = 1024$. The cache has 1024 sets, and each of these can hold 20 lines, so that's more than enough for the 1024 iterations of the `k` loop. Suppose `&b[0] = 0x100000`. This address is in set 0, tag `0x10`. Then `&b[ 1*d3 ] = &b[ 1024 ] = 0x100000 + 1024*4 = 0x100000 + 0x1000 = 0x101000`. That's set `0x40` (64) and tag `0x10`. Next, `&b[ 2*d3 ] = &b[ 2048 ] = 0x102000`. That's set `0x80` (128) and tag `0x10`. Notice that the set number will *always* be a multiple of 64. That mean, out of 1024 sets available only $\frac{1024}{64} = 16$ will be used. That's enough space for $16 \times 20 = 320$ lines. But the `k` loop will access 1024 lines of `b`, and we would like to have those lines present for the second iteration of the `c` loop. But the level 2 cache in a Xeon core won't accommodate us. As a result accesses to `b` in the second `c` iteration will miss the level 2 cache. Such misses are called *conflict misses* because they would not have happened in a cache with a higher associativity. These conflict misses slow the simple code due to the latency of accessing the level 3 cache (which is large enough). True, the level 3 cache is shared by all cores on the chip, but it is large enough for multiplying two 1024×1024 square matrices so no problem.

What about accesses to `a`? They are not a problem in the loop above.

Tiled execution can forestall conflict misses, but the underlying problem remains. Consider this loop from `mm_tilder`:

```
for ( int k = kk;  k < kk + t2;  k++ )
  for ( int r = rr;  r < rr + t1;  r++ )
    for ( int c = cc;  c < cc + t3;  c++ )
      ee[r-rr][c-cc] += a[ r*d2 + k ] * b[ k*d3 + c ];
```

If $t_2 \leq a$ ($a = 20$ for the Xeon L2 caches) then certainly their can't be a conflict miss in accesses to `b`. For $d_3 = 1024$ based on the analysis for the simple code we know the cache can hold 320 lines to that allows a large tile, $t_2 \leq 320$. Though that's true for this inner tile, there is potential reuse of elements of `b` from one `cc` iteration to the next, so even with tiling conflict misses should be avoided.

Conflict misses in matrices can be avoided by using a *padded* layout. Consider again `b[ k*d3 + c]`. This was a problem because `d3` was a power of 2. Suppose $d_1 = 1040$. The sequence of memory addresses for `b[ 0*d3 ]`, `b[ 1*d3 ]`, `b[ 2*d3 ]`, `b[ 3*d3 ]`, will be `0x100000`, `0x101040`, `0x102080`, `0x1030c0` accessing sets 0, 65, 130, 195, ... It can be shown that the sets for addresses in the sequence from $k = 0$ to $k = 1023$ will be different, so each access will bring a line into its own set. Dimension d_1 was chosen so that it is the smallest number $\geq d_1$ for which the greatest common divisor of $\lfloor \frac{ed_1}{L} \rfloor$ and S is 1, where e is the number of bytes per element. In the examples here, $e = 4$. The easy way to compute this is to find a value of d_1 for which the number of lines of storage, $\frac{ed_1}{L}$, is odd.

So using the method above we could replace a 1024×1024 matrix with a 1024×1040 column matrix and avoid some conflict misses. But, what if the matrices we need to multiply are 1024×1024? No problem, allocate a 1024×1040 matrix but copy the data into just the first 1024 columns. The matrix has dimensions 1024×1024 but the rows are stored *at a stride of* 1040. The remaining $1040 - 1024 = 16$ columns are called *padding* and the matrix is said to be *padded*. The code computing the matrix product will ignore the padding columns.

(*a*) Modify the code is `hw01.cc` so that it uses padded matrices as described below. Accomplishing this requires several changes:

- ☐ In `mm_do` set the correct strides to variables `s1a`, etc.
- ☐ In `mm_do` copy data from `a` and `b` into their padded counterparts, `ap` and `bp`.
- ☐ In `mm_tiledr` modify the code so that it correctly reads and writes the padded matrices.
- ☐ In `mm_tiled_do` modify the code so that it accesses the correct matrix product.

It is probably a good idea to do this for one matrix at a time, for example, start with `a`, get it working, then move on to `b` and `g`. Details on what needs to be done are described below.

Matrices are allocated by the `MM_Data` object, this is done in routine `mm_do` where an `MM_Data` object is constructed. Routine `mm_do` is called with matrix dimensions `d1`, `d2`, and `d3`. When this problem is completed the routine will compute strides `s1a`, etc. The constructor for `MM_Data` is invoked with an `app` data structure, the array dimensions and strides:

```
void mm_do( App_Data& app, int d1, int d2, int d3, bool pad = false ) {

  // Initially set strides equal to dimensions. This results in a matrix without padding.
  int s1a = d1, s2a = d2, s3b = d3;
  int s1g = d1, s2b = d2, s3g = d3;

  if ( pad )
    {
      // Set s1a, etc to avoid conflict misses.
    }

  // Construct a Matrix Multiply Data Object
  MM_Data mmd(app,d1,d2,d3,s1a,s1g,s2a,s2b,s3b,s3g);
```

The constructor will allocate arrays `a`, `b`, and `g` in both padded and unpadded forms. The unpadded matrix dimension are of course `a`: $d_1 \times d_2$, `b`: $d_2 \times d_3$, and `g`: $d_1 \times d3$. The padded matrices are of size: `ap`: $s_{1a} \times s_{2a}$, `bp`: $s_{2b} \times s_{3b}$, and `gp`: $s_{1g} \times s_{3g}$.

The values of `d1`, `d2`, and `d3` are call arguments and can't be changed. The values of `s1a` through `s3g` are set to default values but they should be changed to the smallest values needed to avoid conflict misses. Set these values in the block guarded by the `if ( pad )` statement.

After `MM_Data` is constructed `mm_do` initializes matrices `a` and `b` with random numbers and then copies `a` and `b` into their padded counterparts. This copy code however is not correct. Modify the code so that it correctly copies the arrays.

```
  // Fill a and b matrices with random numbers.
  ranges::generate(mmd.a,rand_pm1);
  ranges::generate(mmd.b,rand_pm1);

  //  Copy data from the a and b matrices to their padded counterparts.
  //
  //  [ ] Modify the code for strided storage in ap and bp.
  //
  for ( int r=0; r<d1; r++ )
    for ( int c=0; c<d2; c++ )
      mmd.ap[ r * d2 + c ] =  mmd.a[ r * d2 + c ];
  for ( int r=0; r<d2; r++ )
    for ( int c=0; c<d3; c++ )
      mmd.bp[ r * d3 + c ] =  mmd.b[ r * d3 + c ];
```

Next, `mm_do` calls `mm_simple` to compute a product and put it in array `g_simple`. It prints matrix sizes and strides, and then calls the matrix multiply routines at various tile sizes:

```
  mm_simple( mmd );

  printf("\nMatrix %d  %d  %d  %d.  Duration simple: %.3f ms\n",
         d1, d2, d2, d3, dur_simple_s * 1000 );
  printf("Stride %d  %d  %d  %d  = %d x %d.\n",
         s1a, s2a, s2b, s3b, s1g, s3g );

  mm_tiled_do<1,16,1>( mmd );
  mm_tiled_do<4,16,4>( mmd );
```

```
mm_tiled_do<8,16,8>( mmd );
mm_tiled_do<8,16,16>( mmd );
mm_tiled_do<8,16,32>( mmd );
```

Note that excerpts are shown above, the actual code includes statements for timing and lots more comments.

The actual matrix multiplication is done in routine `mm_tiledr`. Template parameters specify the tile shape, `t1`, `t2`, and `t3`. As described elsewhere, two different methods are used depending on the value of `t3`. When `t3` is 8 or larger matrix elements in `b` and `g` are accessed as vectors. Otherwise, as plain old scalars.

The unmodified code accesses matrices `ap` and `bp`, but treats them as unpadded. Modify the code so it makes padded access. This can be done by changing certain variables. There is no need to add new statements, loops, etc.

It might be easier to first get the unpadded code working (the code guarded by `if (!use_vec)`). When doing this put a return in `mm_do` before `mm_tiled_do` is called for a tile size of 8 or larger.

Finally, modify `mm_tiled_do` so that the code checking the product correctly accesses data from `gp`. Look for the code headed by comment `Check Matrix Product`. Modify the code assigning `hd` and any other code that needs to be changed.

If this problem is solved correctly there should be big improvement in matrices of size 2048×2048 and larger. There will be little gain in the smaller matrices.