TOP25 - lab 3

General Matrix Multiply (GEMM) memory layout and cache blocking optimizations

Gautier MIGUET
Github repository

Contents

1 Introduction

The goal of this lab is to implement a series of optimizations regarding a modification of the General Matrix Multiply (GEMM) algorithm. The initial state of the code is the following:

- A basic implementation of the algorithm
- A high degree of parallelization with Kokkos and OpenMP
- Matrix storage using `Kokkos::LayoutRight`

After multiple optimizations, the code should behave the same while having a much improved execution time.

2 Methodology

2.1 System information

All code shown here has been developed and tested on 2 separate machines, each with different hardware and installations of linux. For the sake of clarity, most of the discussion will be about **System A**, with some comparison with System B when it seems necessary.

	System A	**System B**
Hardware		
CPU	Intel i5-1135G7	Intel i5-13600KF
Usable* CPU Cores	4	6
CPU Frequency	4.2 GHz	5.1 GHz
Memory	16 GB	32 GB
L1d Cache	48 KiB	48 KB
L2 Cache	1280 KiB	2 MiB
L3 Cache	8 MiB	24 MB
Software		
Linux Distribution	Linux Mint 21	Debian 13 Testing
Kernel Version	6.8.0	6.12.21
C++ Compiler	gcc 11.4.0	gcc 14.2.0
Python Version	3.10.12	3.13.2
Perf Version	6.8.12	6.12.21

Note: For reasons explained in the following section, System B's E-cores are excluded from all performance measurements. CPU frequency and cache values are for P-cores on System B.

2.2 Preparation

In order to have consistent measurements between optimizations, a couple of bash and python scripts are set up prior to any modification to the C++ code.

A bash script to get accurate number of cores on System B. Since Intel's 12th generation of processors, cores are mixed between performance and efficiency. This is problematic for our use case, as splitting an equal workload between cores results in P-cores finishing the task much faster than E-cores, due to their higher clock speed and bigger caches. The easiest way to resolve this issue is simply to avoid any execution on E-cores. This script returns the count of P-cores on the machine, allowing OpenMP to pin threads to those only.

2.3 Performance measurements

Performance measurement is done in 3 ways:

- A strong scaling study
- A weak scaling study
- L1 and LL cache load and misses using the `perf` tool

The main metrics are:

- Execution time
- Speedup and efficiency (from strong and weak scaling)
- Cache loads and miss ratio

These 3 metrics give us enough information about general performace of the algorithm, its scalability across multiple cores and the efficiency of our memory accesses.

Measurements is done on square matrices. For strong scaling, the size is constant ($N = 1000$). For weak scaling, matrix size is determined by a constant number of operations per core: $N = \sqrt[3]{1'000'000 * \text{cores}}$. This gives us speedup and efficiency. Last, `perf` is used on $N = 2500$ matrices. This is a big enough problem, taking about 20 seconds to solve (on the base code), while giving accurate cache loads/misses ratios.

2.4 Result verification

In order to garantee a correct result, the checksum of the C matrix is computed. A function simply loops over the entire matrix and adds all the values in a single double. This allows a quick and easy way to verify the GEMM results, whithout having to recompute the whole matrix with a slow version of the algorithm.

However, we need to remove the random initialization of the A and B matrices, even if a random seed is set, the out of order nature of the multi-threaded initialization changes the output every time the code is run.

We change the initialization from

```
    M(i, j) = drand48();
```

to

```
    M(i, j) = std::max(i, j) % 6 - 2;
```

We mod over the indices to prevent the final value from exploding.

The initial value of the checksum, for 2500-sized matrices is `18437536359`. We can simply compare every new version of the code with this value to validate the algorithm.

2.5 Project execution

A final bash script is made for global workflow, compiling and running the project with both `perf` and the python strong/weak scaling studies.

After each optimization, a commit is tagged with `git tag` in order to facilitate switching between versions of the code.

3 Optimizations

3.1 Base program

In order to base our improvements on something, we run our whole test suite on the base algorithm, without any modification. The state of the GEMM kernel is the following:

```
KOKKOS_LAMBDA(int i) {
      for (int j = 0; j < int(B.extent(1)); ++j) {
        double acc = 0.0;
        for (int k = 0; k < int(A.extent(1)); ++k) {
          acc += A(i, k) * B(k, j);
        }
        C(i, j) *= beta + alpha * acc;
      }
    });
```

It is safe to imagine the `KOKKOS_LAMBDA` as the first loop of our GEMM algorithm; we therefore have 3 `for` loops, iterating over `i`, `j` and `k`.

As is, the execution is about 17 seconds. `perf` reports the following cache information:

```
 Performance counter stats for './build/src/top.matrix_product 2500 2500 2500':

    31 864 308 783      L1-dcache-loads
    25 138 474 700      L1-dcache-load-misses       #   78,89% of all L1-dcache accesses
     1 062 472 394      LLC-loads
       495 945 599      LLC-misses                  #   46,68% of all LL-cache accesses
```

As we can see, there are a huge amount of cache accesses (31 billions), with most of them missing, increasing execution time.

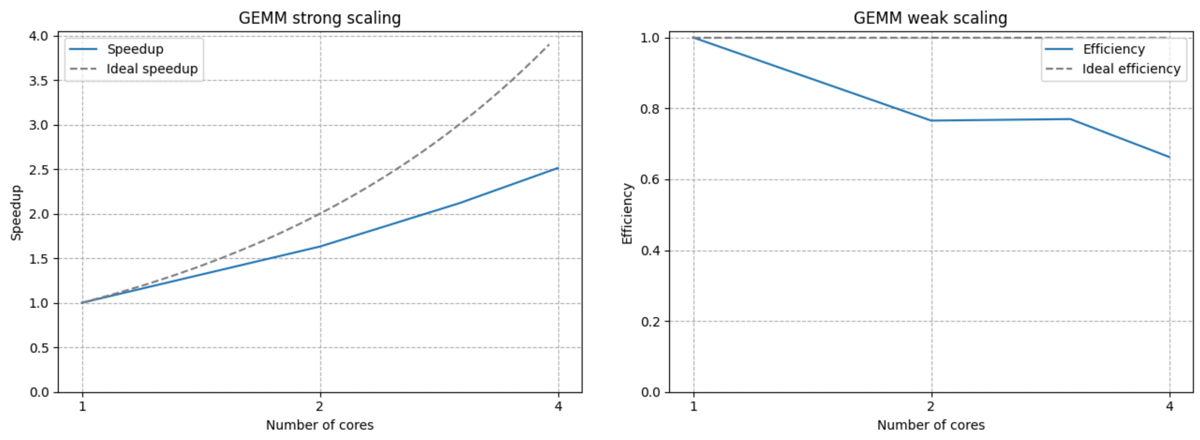

Figure 1: Base algorithm strong and weak scaling

The scaling here is quite bad, as opposed to what we would expect from the very parallel nature of the GEMM algorithm.

3.2 Kokkos LayoutLeft

The first optimization is the transposition of the B matrix in memory. In the base version of the code, The B matrix is stored using `Kokkos::LayoutRight` (row-major). However, as we can see in the code, the GEMM function accesses the B matrix at indices `(k, j)` making an extremely inneficient use of cache lines, slowing down the execution considerably.

Transposing the matrix (using `Kokkos::LayoutLeft` instead of `Kokkos::LayoutRight`) changes the access pattern of its elements, taking full advantage of spacial locality.

The two other matrices don't require any transformation, as they are accessed in the same order as their elements are layed out.

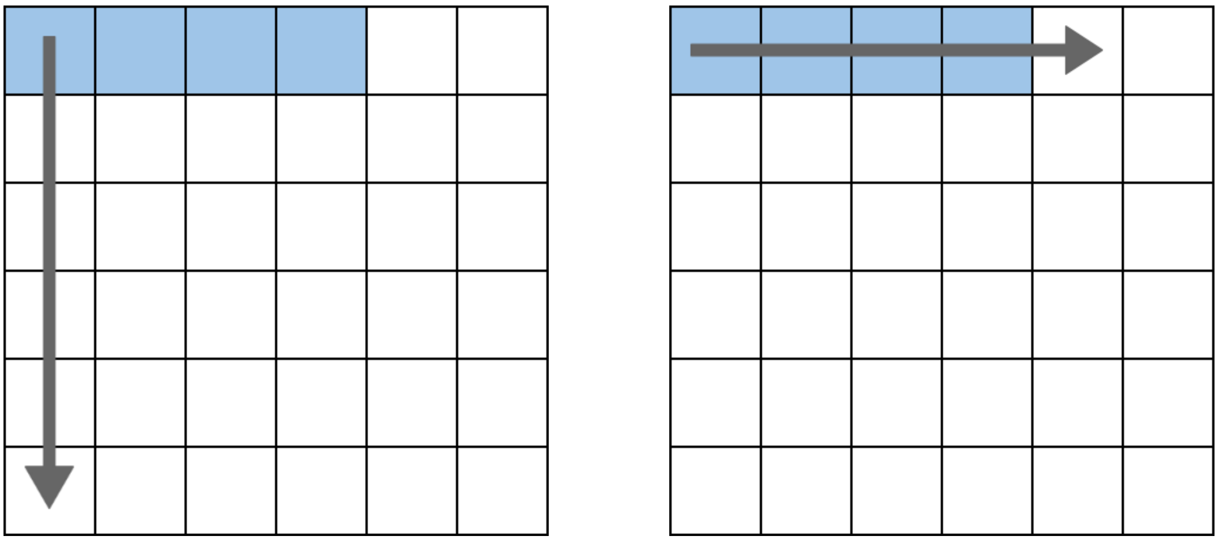

Figure 2: B matrix access pattern with LayoutRight (left) LayoutLeft (right).
Blue cells represent a single cache line.

Here, the blue cells represent cached data of the B matrix after the first (top left) value has been accessed. It's easy to see that the use of `LayoutLeft` reduces memory accesses drastically.

After this implementation, the cache accesse are way more efficient:

```
 Performance counter stats for './build/src/top.matrix_product 2500 2500 2500':

    16 193 097 486      L1-dcache-loads
     2 091 348 450      L1-dcache-load-misses       #   12,92% of all L1-dcache accesses
        20 752 617      LLC-loads
         1 986 857      LLC-misses                  #    9,57% of all LL-cache accesses
```

L1 cache loads are halved, while hits went from 22 to 87 percent. As for L3, almost all loads are prevented, going from more than 1 billion to as little as 20 millions.

Execution time is greatly reduced going from about 17 seconds to 6.5. Scaling however, stays quite similiar to the base code.

3.3 Cache Blocking

The default GEMM kernel loops over i, then j and last k. This access pattern can be represented as followed:

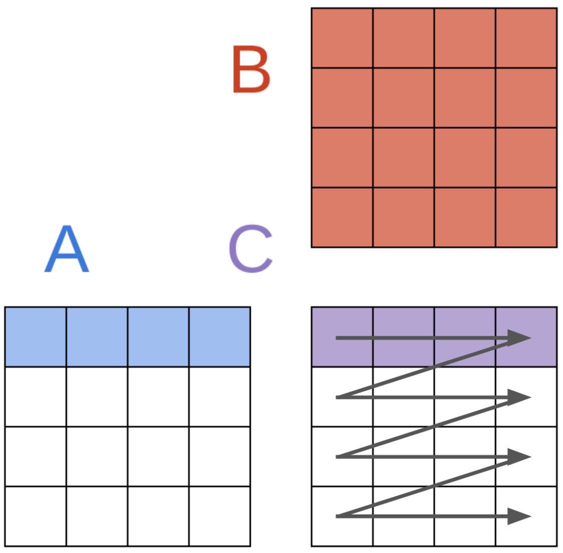

Figure 3: Default access pattern

Here, we compute the first 4 elements of the C matrix. However, (assuming a cache line can contain 4 values), we need to load 5 cache lines for this computation: the first 4 values of the A matrix, and 4 columns of 4 values of the B matrix (here, B is still transposed). In the case where our cache can only contain 4 cache lines, the entirety of the B matrix would have to be fetched again in order to compute the second row of the C matrix.

Rearranging the kernel loops, we can change the access pattern of both A and B increasing the ratio of computed values to accessed cache lines. In the folowing figure, we still compute 4 elements of C, but we manage to save a cache line from being loaded.

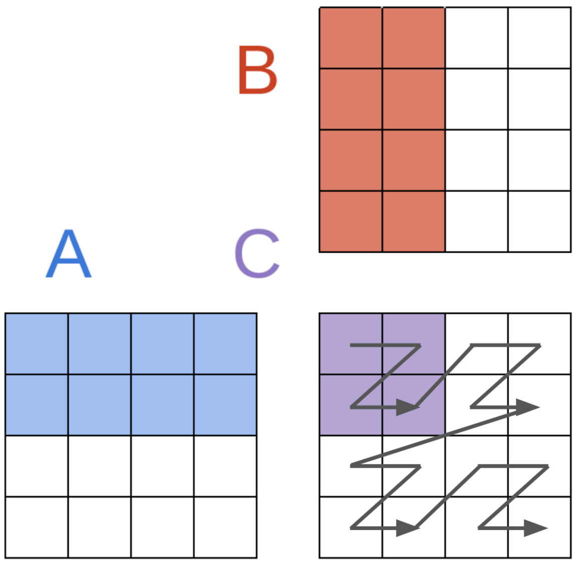

Figure 4: 2D cache blocking stencil

The kernel then becomes:

```
KOKKOS_LAMBDA(int bi) {
      for (int bj = 0; bj < blocks_j; ++bj) {
        int i_lim = std::min((int) A.extent(0), bi * BLOCK_SIZE + BLOCK_SIZE);

        for (int i = bi * BLOCK_SIZE; i < i_lim; ++i) {
          int j_lim = std::min((int) B.extent(1), bj * BLOCK_SIZE + BLOCK_SIZE);

          for (int j = bj * BLOCK_SIZE; j < j_lim; ++j) {

            double acc = 0.0;
            for (int k = 0; k < int(A.extent(1)); ++k) {
              acc += A(i, k) * B(k, j);
            }
            C(i, j) *= beta + alpha * acc;

          }}}
    });
```

Note the `i_lim` and `j_lim` variables used to compute the last range of values without going over the matrix bounds.

A few other things change in the code; We define `constexpr const int BLOCK_SIZE = 10;` to manage the size of our 2D stencil, and we set

```
int blocks_i = A.extent(0) / BLOCK_SIZE + (A.extent(0) % BLOCK_SIZE != 0);
int blocks_j = B.extent(1) / BLOCK_SIZE + (B.extent(1) % BLOCK_SIZE != 0);
```

to divide our problem.

After these changes, `perf` reports the following metrics:

```
Performance counter stats for './build/src/top.matrix_product 2500 2500 2500':

    15 773 157 582      L1-dcache-loads
     2 193 501 867      L1-dcache-load-misses       #   13,91% of all L1-dcache accesses
         1 298 123      LLC-loads
           240 339      LLC-misses                  #   18,51% of all LL-cache accesses
```

L1 accesses don't change much, but LLC loads are divided by 15. Overall, execution time decreases from 6.5 seconds to 4.2.

Note that the value of `10` for our block size is set experimentally. In fact, System B has better results with a block size of `16`, due to the bigger caches.

This optimization also has the benefit of increasing the scalability of the algorithm, going up to a 3.75 times speedup on 4 cores (see the conclusion for graphs).

4 Conclusion

To conclude, let's compare all of our optimizations:

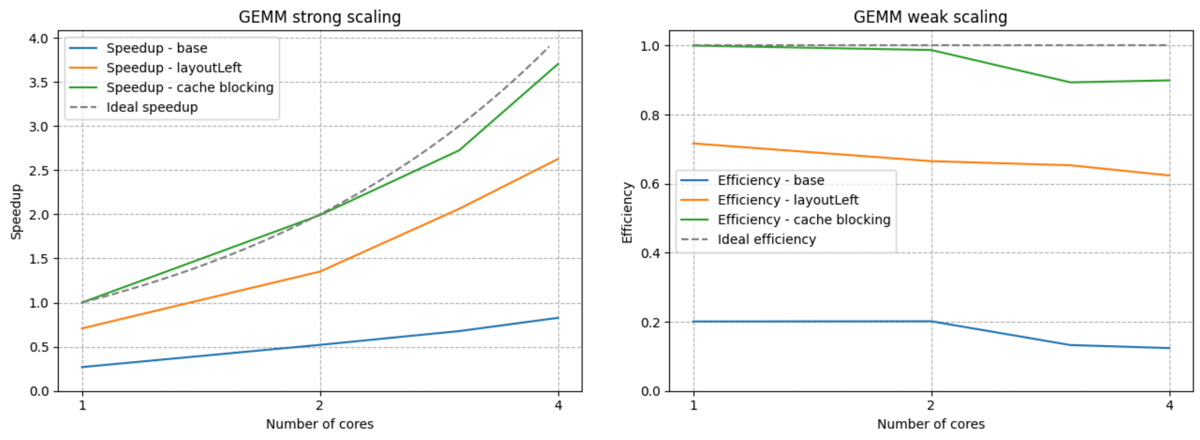

Figure 5: Combined strong and weak scaling of all states of the code.

Overall, the code went from 17 seconds to about 4.2. This represents about a 4x speedup. The number of L1 cache loads is halved, and LLC loads are divided by more than 800. As we can see from the figure above, the scalability of the code also improves, probably due to a more efficient use of all cache levels, especially those private to each core.

4.1 Just for fun

Here's what happens when we use both E-cores and P-cores on System B:

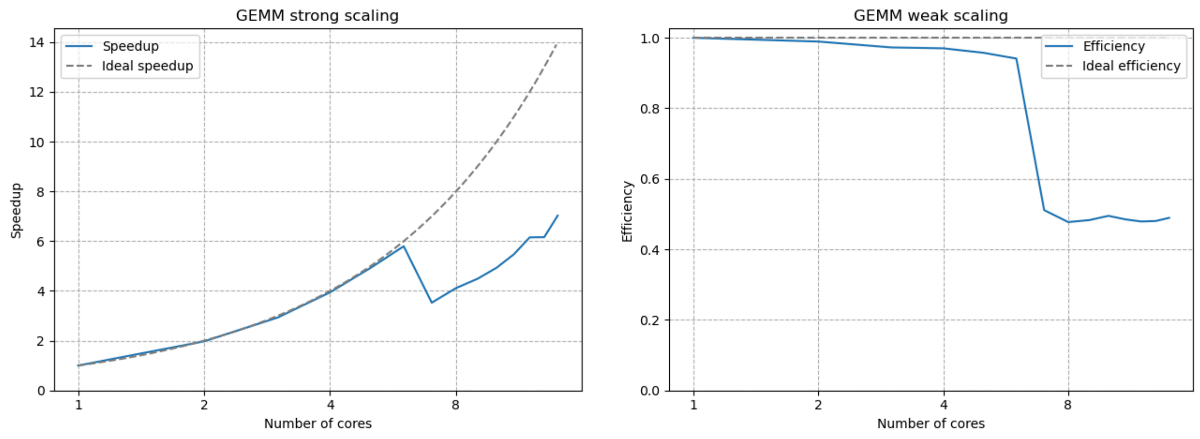

Figure 6: I'm buying AMD next time :/

Here, cores 1-6 are P-cores, and 7-14 are E-cores.

With some tweaking of the algorithm, and a better spreading of tasks on all types of cores, it would be possible to greatly increase the speedup of the algorithm. In fact, other benchmarks run on this system show that E-cores represent about 40% of all GFlops of the CPU. Unfortunalty, enabling this extra performance would take too much time, and woudln't be worth the effort as most compute-focused CPUs don't have such a heterogeneous architecture.