SessionMatrix

May 27, 2023

1 Matrix multiplication

Conceptually, matrix multiplication is a fairly trivial calculation. It will be used here as a vehicle to illustrate important aspects of high performance computing. We are not implying that HPC is only about multiplying matrices. But implementing a matrix multiplication algorithm with decent performance in C++ forces us to discover ways to structure the code for optimal use of the computing hardware, without any domain specific complications obfuscating the relevant transferable insights on high performance computing.

The goal of the following examples is not to write a production ready high performance library for linear algebra. Those libraries already exist for use in C++ applications: e.g., Eigen and Blaze. If the scientific or engineering problem you are trying to solve is expressible in terms of simple linear algebra operations, use one of these libraries for the linear algebra needs and focus on the scientific and engineering aspects. The goal of this course is to gain some insights about *how* these libraries perform so well, not simply how to use them. We are focussing on one very tiny part of linear algebra calculations: matrix multiplication to learn about the life and habitats of performance sucking demons, and how to recognize and work around them. This in turn, should help us improve performance even in code which has nothing to do with linear algebra.

For matrices,

$$A^{(m\times n)} = \begin{pmatrix} A_{11} & A_{12} & ... & A_{1n} \\ A_{21} & A_{22} & ... & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m1} & A_{m2} & ... & A_{mn} \end{pmatrix} \quad B^{(n\times p)} = \begin{pmatrix} B_{11} & B_{12} & ... & B_{1p} \\ B_{21} & B_{22} & ... & B_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ B_{n1} & B_{n2} & ... & B_{np} \end{pmatrix} \tag{1}$$

the product

$$C^{(m\times p)} = A^{(m\times n)} \times B^{(n\times p)} = \begin{pmatrix} C_{11} & C_{12} & ... & C_{1p} \\ C_{21} & C_{22} & ... & C_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ C_{m1} & C_{m2} & ... & C_{mp} \end{pmatrix} \tag{2}$$

is defined as

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

A straight forward implementation would do the calculation something like this (for some definition of a matrix type and variables A, B and C of the matrix type:

```
for (size_t i=0ul; i<C.n_rows(); ++i) {
    for (size_t j=0ul; j<C.n_cols(); ++j) {
        for (size_t k=0ul; k<A.n_cols(); ++k) {
            C(i,j) += A(i,k) * B(k,j);
        }
    }
}
```

1.1 Exercise 1

To test different ideas and there influence on performance, you will find a folder MatMul in your exercises directory. The main program is called "matmul.cc", which parses some command line options and times the matrix multiplication calculation by repeating it a given number of times, optionally verifies the result, and prints a GFlop count. The actual matrix implementation is in a series of Matrix_xyz.hh files, one of which must be included. For this exercise, include the header "Matrix_naive.hh" and no other Matrix_xyz.hh files. There is a CMakeLists.txt file.

- Build and run it with a few different matrix sizes and get familiar with the various options.
- Use the Linux "perf" command to get some info about it. Use "perf list" to get a list of performance related events. Then use "perf stat -B -e comma-separated-list-of-events-to-moniter matmul -size 2048 -reps 2"

This implementation has really poor performance. On my laptop (Thinkpad X1 Carbon, i7-8550U), it takes about 62 seconds for all 3 matrices of size 2048 x 2048, when compiled with GCC 8.2, with usual optimisations (-O3 -march=native -DNDEBUG) turned on. That corresponds to about 0.27 GFlops. Of course, this case is meant to be as bad as it gets (although I am positive with some effort one can do worse). What are some of the reasons why this simple implementation is so slow ?

- We are not using all available processor cores
- We are not making any effort to use the 256bit wide vector registers or fused multiply add instructions many contemporary CPUs are capable of
- Another far more influencial aspect is staring at us in the output of the "perf" program. In my case ...

```
sandipan@bifrost:~/C++/MatMul/build> perf stat -B -e \
L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,\
L1-icache-load-misses,branch-misses,LLC-load-misses,\
LLC-loads,LLC-store-misses matmul -size 2048 -reps 2 -threads 8

Time taken to fill the matrices = 0.109758 seconds
Z=X*Y: size=        2048 average time = 62.2075 seconds 0.276103 Gflops

 Performance counter stats for 'matmul -size 2048 -reps 2 -threads 8':

    34,495,308,357      L1-dcache-load-misses:u   #   66.86% of all L1-dcache hits     (37.49%)
    51,593,604,680      L1-dcache-loads:u                                              (50.00%)
    17,199,419,793      L1-dcache-stores:u                                             (50.00%)
           265,178      L1-icache-load-misses:u                                        (50.00%)
         8,537,072      branch-misses:u                                                (50.00%)
```

```
     1,104,075,177      LLC-load-misses:u         #    7.78% of all LL-cache hits      (50.00%)
    14,194,046,923      LLC-loads:u                                                    (50.00%)
           732,977      LLC-store-misses:u                                             (25.00%)

     124.534512063 seconds time elapsed
```

- Because of the size of the matrix, and the fact that C++ convention for multi-dimensional arrays stores the matrices in the "row major" order, almost every access of the matrix element A(i,k) in the inner loop over k, incurs an L1-cache miss, and sometimes even a trip to the main memory

Observe that the equation defining the matrix product can also be read in terms of p component vectors $\mathbf{C_i}$ being a linear combination of the p component vectors $\mathbf{B_k}$:

$$\mathbf{C_i} = \sum_{k=1}^{n} A_{ik}\mathbf{B_k}$$

This equation is identical to the original matrix multiplication definition given above, but this new way of writing it suggests a different implementation:

```
for (size_t i=0ul; i<C.n_rows(); ++i) {
    for (size_t k=0ul; k<A.n_cols(); ++k) {
        for (size_t j=0ul; j<C.n_cols(); ++j) {
            C(i,j) += A(i,k) * B(k,j);
        }
    }
}
```

i.e., a simple interchange of the two inner loops. The value A(i,k) is a constant for the inner loop and the matrix B(k,j) is accessed along row k in the innermost loop (over j). What difference does such a simple interchange make ?

1.2 Exercise 2

Interchange the j and k loops in the multiplication operator in the Matrix_naive.hh file, and examine the performance using the perf program. Here is a typical output:

```
sandipan@bifrost:~/C++/MatMul/build> perf stat -B -e \
L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,\
L1-icache-load-misses,branch-misses,LLC-load-misses,\
LLC-loads,LLC-store-misses matmul -size 2048 -reps 2 -threads 8

Time taken to fill the matrices = 0.113768 seconds
Z=X*Y: size=        2048 average time = 3.88088 seconds 4.42571 Gflops

 Performance counter stats for 'matmul -size 2048 -reps 2 -threads 8':

     2,448,285,680      L1-dcache-load-misses:u   #   18.85% of all L1-dcache hits     (37.47%)
    12,985,176,496      L1-dcache-loads:u                                              (50.00%)
     4,300,751,264      L1-dcache-stores:u                                             (50.05%)
           186,472      L1-icache-load-misses:u                                        (50.07%)
```

```
       8,518,363      branch-misses:u                                                       (50.05%)
     533,823,586      LLC-load-misses:u          #   41.72% of all LL-cache hits            (50.00%)
   1,279,405,484      LLC-loads:u                                                           (49.95%)
          47,895      LLC-store-misses:u                                                    (24.97%)

     7.883486662 seconds time elapsed
```

Now we have far fewer L1 cache misses, fewer LLC loads in general and fewer LLC load misses. Overall impact is quite significant, and we have about 4.5 Gflops. This is such a trivial change that many compilers will automatically interchange the j and k loops of the Exercise 1 version of the code, so that you don't actually see any differences in practice. It is fortunate (for educational purposes) that gcc 8.2 refuses to do this optimisation for the version of the code given to you for the exercise.

1.3 Exercise 3 : expression templates and other cleanup

The next step does not have a great impact on performance in the small test program here, but reduces the number of dynamic allocations, and still allows us to write "C = A * B" without returning a big matrix by value. We use a single `vector<double>` instead of `vector<vector<double>>`. The function operator*() does not do any calculations, but just returns an object of the type "matprod", containing references to the original operands A and B, and the operation to be done on them, i.e., multiplication. matrix class has an overloaded assignment operator taking a matprod object as an argument, and that calls the actual multiplication function. This is implemented in `Matrix_xtmp.hh`.

After understanding and running the program a few times, analyse it with perf as follows

```
perf record matmul -size 2048 -reps 2 -threads 8
perf annotate -M intel
```

The CMake setup for the exercises uses the compiler option "-ggdb3" even for the release builds, so that perf can map the statistics back to the source code. The keyboard shortcut "h" shows you a list of keyboard shortcuts. Find out how to navigate through the output of "perf annotate". Go to the hottest portion of the code. Here is my output:

```
  0.00          vbroad ymm1,xmm3
              ↓ jbe    648
  0.04          nop
                              crowi[j] += aik * browk[j];
 42.13  3c0:    vmovup ymm2,YMMWORD PTR [rax+r9*1]
 33.99          vfmadd ymm2,ymm1,YMMWORD PTR [rsi+r9*1]
 15.45          vmovup YMMWORD PTR [rsi+r9*1],ymm2
                  size_t ncols_() const { return nc; }
  0.01          add    r9,0x20
                cmp    r9,QWORD PTR [rbp-0x160]
  8.21        ↑ jne    3c0
```

We see that the two move operations, moving information from a memory location to the ymm2 register and the other way round consume a lot of time! This is not to be interpreted as "vmovup" is a slow instruction. We have to pay attention to the memory locations involved in the operation. Size of one row of this matrix is 2048 x 8 = 16kb. The 32kb level-1 cache of the processor can not hold more than 2 rows of this matrix. As we go through the rows of the matrix B and store the result in the i'th row of C, we will be evicting the row of C previously in level-1 cache. And every subsequent access along the row of B must also be serviced from level-2 or a lower level cache, or sometimes even the main memory. Here are the latencies of different levels of cache of my processor:

Cache	Latency	Associativity	Total size
L1D	4-5 cyc	8 way	32 KiB
L2	12 cyc	4 way	256 KiB
L3/LLC	42 cyc	16 way	2MiB/core
DRAM	42 cyc + 51ns		
Kaby lake specs			

You can find a lot of this information directly using a variety of commands. The following are installed on JUSUF. Take a minute to get to know the CPUs on the supercomputer where you are running your programs!

```
lshw -C CPU
lscpu
getconf -a | grep -i cache
```

Let's return to the matrix multiplication problem. As we see, in a tight loop like the portion of code shown above, not being able to service the memory request from L1 is undesirable, since the latency of higher levels of cache is considerably larger. That's why the load and store instructions in the line around the vfmadd take that much time: we exhaust the L1d cache and are forced to constantly fetch from the slower cache levels. We have a simple pattern of memory access, and the pre-fetcher knows what to fetch, but, it can not put it anywhere in L1 without evicting something else we are using. How can we fix this ?

1.4 Exercise 4: Better cache use by tiling

Matrix multiplication can be written in terms of sub-matrices of the original matrices as follows:

$$A^{(m\times n)} = \begin{pmatrix} (A_{11}) & (A_{12}) & \ldots & (A_{1n}) \\ (A_{21}) & (A_{22}) & \ldots & (A_{2n}) \\ \vdots & \vdots & \vdots & \vdots \\ (A_{m1}) & (A_{m2}) & \ldots & (A_{mn}) \end{pmatrix} \quad B^{(n\times p)} = \begin{pmatrix} (B_{11}) & (B_{12}) & \ldots & (B_{1p}) \\ (B_{21}) & (B_{22}) & \ldots & (B_{2p}) \\ \vdots & \vdots & \vdots & \vdots \\ (B_{n1}) & (B_{n2}) & \ldots & (B_{np}) \end{pmatrix} \tag{3}$$

$$\left(C_{ij}\right) = (A) \times (B) = \sum_{k=1}^{n} \left(A_{ik}\right)\left(B_{kj}\right) \tag{4}$$

It is therefore possible to arrange the product of two large matrices in terms of lots of products of smaller matrices. The advantage of the product of the smaller matrices is that if they are small enough, both the RHS as well as the product matrix will simultaneously entirely fit inside the fastest cache. The tight loop in the previous example will not need to communicate with the slower caches or the main memory. An implementation based on this idea is in "Matrix_blocks.hh". Include it in the main, and analyse with perf. Here is my output:

```
sandipan@bifrost:~/C++/MatMul/build> perf stat -B -e \
L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,\
L1-icache-load-misses,branch-misses,LLC-load-misses,LLC-loads,\
LLC-store-misses matmul -size 2048 -reps 2 -threads 8

Time taken to fill the matrices = 0.0873381 seconds
Z=X*Y: size=        2048 average time = 0.879522 seconds 19.5284 Gflops

 Performance counter stats for 'matmul -size 2048 -reps 2 -threads 8':

       535,000,814      L1-dcache-load-misses:u   #    6.00% of all L1-dcache hits    (37.14%)
     8,911,762,098      L1-dcache-loads:u                                             (49.69%)
     4,320,865,002      L1-dcache-stores:u                                            (49.90%)
            47,050      L1-icache-load-misses:u                                       (50.12%)
           112,533      branch-misses:u                                               (50.19%)
        11,766,962      LLC-load-misses:u         #   40.91% of all LL-cache hits     (50.31%)
        28,765,852      LLC-loads:u                                                   (50.10%)
         2,191,116      LLC-store-misses:u                                            (24.66%)

       1.849980881 seconds time elapsed
```

1.5 Exercise 5: Parallel calculation using TBB and parallel STL

The multiplication of the block matrixes in the previous example can be done in parallel in many ways. Accumulation of the block i, j in the result matrix is independent of every other block. The parallelisation is trivial and can be achieved in many many ways. In `Matrix_blocks.hh`, there are portions of code showing how to do that using the parallel algorithms of C++ standard template library. They are commented out. Enable the parallel parts to perform the matrix multiplication in parallel. Try larger matrix sizes, e.g., 16384, and 128 threads on JUSUF, e.g.,

```
matmul -size 16384 -reps 5 -threads 128
```

Remember: For simplicity, the `Matrix_blocks.hh` in the examples directory is limited to square matrices of dimensions which are multiples of 256.

1.6 Exercise 6: Explicit vectorization using xsimd

The central calculation:

```
for (size_t j=0ul; j < n; ++j) C(i, j) = C(i, j) + A(i, k) * B(k, j)
```

is a completely unsequenced loop, i.e., different values of j can be executed in arbitrary order to produce the same results. The loop is therefore trivially vectorisable. The right hand side also makes clear that the calculation consists of a series of “fused multiply-add” operations. Auto-vectorizers built into present day compilers do a **fine job** translating such code using SIMD instructions. Sometimes, they do not have enough information in the code written above to ensure that a SIMD calculation would be correct. The memory locations for C_{ij} and B_{kj} could partially overlap. In our example, surely not. But, the compiler can not infer that from the code we have written. One could give all kinds of hints to the compiler using OpenMP pragmas, or do manual vectorisation with a SIMD library like XSIMD. The need for doing that is diminishing because of the excellent work done by compiler vendors.

For this exercise, we will use the XSIMD library introduced earlier to vectorise by hand. The necessary code is in the file `MatrixView.hh` which is included by `Matrix_blocks.hh`, but the vectorized loop is commented out. Experiment by uncommenting the manual vectorization section in `MatrixView.hh`.

1.7 Exercise 7: Numeric intensity

CPU cache access is much faster than main memory access. But the table above, with the cache information tells us that even for the fastest cache, latency of a new memory fetch is about 4-5 cycles. Accessing information already in the CPU registers is essentially immediate. One could think of the registers-to-cache relation as analogous to the cache-to-memory relation. We gained a lot of performance by ensuring that things don't drop out of the L1 cache before we have used them heavily. Could we do something similar with information in the registers ?

Modern C++ does not allow us to dictate what variables should be stored in a register (the old keyword `register` does not do anything). But we know that our vectorized operations with `xsimd::batch<T, Arch>` load batches of values to registers. What if we made a tiny matrix made of $S \times S$ values, where S is the size of a SIMD register ? On a machine with AVX512, this becomes an 8×8 matrix, taking 8 zmm registers. Two input and one output matrix require 24 such registers. An AVX512 capable CPU has 32 of them. So, in principle, we can write a matrix multiplication using these registers and unroll all loops completely for a "core" part of our computation. The larger matrices can be built by composing these "Atomic" matrices. The example `Matrix_ni.hh` explores this idea. There is an `AtomicMatrix` class defined in `AtomicMatrix.hh` included in that file.

The key aspect of the `AtomicMatrix` is in its core unrolled multiplication loop:

```
    uloop<0ul, size>{}([&](size_t k){ // k-i-j loop order!
        uloop<0ul, size>{}([&](size_t i){
            crows[i] = simdlib::fma(simdlib::set_simd(A(i,k)), brows[k], crows[i]);
        });
    });
```

This is a compile-time unrolled loop, created with some trivial meta-programming. The interesting part is in the order of the i, j and k loops. The j loop has become a single `fma` instruction. If we did an i-k-j loop like before, we would have a data dependency between successive operations in the inner loop. If we write it like above, i.e., a k-i-j loop, we act on different registers in successive iterations of the i loop, which enables better pipeline use! We also make much better use of information already available in registers with this approach. Test the impact of this implementation by using `Matrix_ni.hh`.

```
sandipan@bifrost:~/C++/MatMul/build> perf stat -B -e \
L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,\
L1-icache-load-misses,branch-misses,LLC-load-misses,LLC-loads,\
LLC-store-misses matmul -size 2048 -reps 2 -threads 8

Time taken to fill the matrices = 0.0847436 seconds
Z=X*Y: size=        2048 average time = 0.148013 seconds 116.042 Gflops
```

```
 Performance counter stats for 'matmul -size 2048 -reps 2 -threads 8':

       507,625,502      L1-dcache-load-misses:u   #    9.14% of all L1-dcache hits    (37.71%)
     5,550,908,156      L1-dcache-loads:u                                             (50.21%)
        77,973,686      L1-dcache-stores:u                                            (49.82%)
            64,340      L1-icache-load-misses:u                                       (49.81%)
         1,221,031      branch-misses:u                                               (49.75%)
           547,163      LLC-load-misses:u         #   11.71% of all LL-cache hits     (49.79%)
         4,674,039      LLC-loads:u                                                   (50.18%)
         1,126,735      LLC-store-misses:u                                            (25.09%)

       0.384639105 seconds time elapsed
```

1.8 Exercise 8

The final idea presented is one where we recursively divide the input matrix into smaller and smaller blocks. At each stage, we split the matrix in 4 parts and multiply the matrices in terms of the quarters. Once the quarters become smaller than a threshold, we do not divide any more but treat the blocks as being directly made of our `AtomMatrix` of the previous example. The recursive division adapts nicely to all levels of cache and the `AtomMatrix` gives a nice little push to have a very decent performance on our supercomputer. The recursive parallelization is done using `tbb::parallel_invoke`. In order to reduce the number of cache misses during the multiplication of the small matrices, we store the elements using a Z-order curve.

This obviously makes our solution too specific to matrix multiplication, and that too, only for square matrices of dimensions which are powers of 2. But our goal here is not really to make a production ready matrix library, but so see different aspects of high performance computing. The Z-order organisation of matrix coefficients simply illustrates that non-trivial organisation of data, when carefully chosen, can be more machine friendly, and hence yield better performance. This final example is in a file called `Matrix_recursive_blocks.hh`.

1.9 Summary

You will have noticed the diminishing reward-to-effort ratio through the different stages of the above exercises. A proper linear algebra library taking advantage of the specific properties of the problem and the hardware makes our lives simpler, but behind such libraries is the efforts of a lot of individuals. The purpose of this chapter was to look a bit closer at what those efforts entail, and show you a few of the hundreds of such techniques in the wild.

[ ]:

```
```