CS 5220 Project 1

Matrix Multiplication

Bob(Kunhe) Chen [kc853]
Cornell University

Abstract

We report our progress in the second stage of implementing a dense matrix multiplication function on the totient cluster. We attempt to build a faster kernel for the function which can process small submatrices loaded on the register memory fast using AVX2 intrinsics that is newly available on the Intel Xeon Processor E5-2620. In addition to the kernel function, we add multi-level blocking to speed up the program by storing submatrices in the cache memory to reduce the memory transfer between physical memory and CPU. Due to the time limit on this project, we have some unresolved issues and untested ideas for further improvement, and we are reporting them for possible learning experience.

1 Theory

1.1 Brief Summary of Matrix Multiplication

In this project, we implement a function that computes dense matrix multiplication for

$$\mathbf{C} = \mathbf{AB}, \quad \text{or } \mathbf{C}_{ij} = \sum_{k=1}^{N} \mathbf{A}_{ik}\mathbf{B}_{kj}$$

where the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{\mathbf{N}\times\mathbf{N}}$ and their entries are stored in double precision arrays. A naive implementation is easy:

```
for (i = 0; i < N; ++i) {
        for (j = 0; j < N ++j) {
        double cij = C[j*N+i];
                for (k = 0; k < N; ++k)
                        cij += A[k*N+i] * B[j*N+k];
                C[j*N+i] = cij;
        }
}
```

This implementation, however, fails to consider the CPU cache size, thus creating an increasingly large amount of data traffic between the memory and the cache as the problem size N increases. In other words, the arithmetic intensity (AI) of this algorithm is low. Theoretically, the AI for entry-wise matrix multiplication is o(1). To implement algorithms with this AI, according to the Roofline Performance Model, we are limited by the machine's memory capacity instead of its compute capacity.

1.2 Memory Bottlenecks

Because the computing cost for matrix multiplication remains o(N^3) for most algorithms, the best way to increase AI is to reduce the memory traffic by reusing the data we have already stored in the cache. Ideally, we want all three matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ in the cache. In this case, the arithmetic intensity is not o(1) but o(N) because each entries are only loaded once. For large matrices, we cannot fit them into the cache and therefore have to load each entries multiple times, so as a compromise, we can use block matrix multiplication to carry out the computation in such a way that the submatrices $\mathbf{A}_{IK}$, $\mathbf{B}_{KJ}$ and $\mathbf{C}_{IJ}$ can fit into the cache memory. Then each block matrix multiplication looks like

$$\mathbf{C}_{IJ} = \sum_{K=1}^{p} \mathbf{A}_{IK}\mathbf{B}_{KJ}$$

where I, J, K are the sets for indices that corresponds to the submatrices and $p = N/n$ and n is the size of block matrix. This strategy will reduce the arithmetic intensity by a factor of $1/p$, because we have to reload each entry p times.

1.3 Register and FMA

In addition to using better algorithms, we can potentially take advantage of our knowledge about the hardware, since we are developing and tuning for a specific cluster. The cluster compute node has Intel Xeon Processor E5-2620 that supports AVX 2.0 [1]. AVX2 expands most integer commands to 256 bits and has FMA. So now the idea is to build a fast kernel function that operates on matrices that are sufficiently small to fit into the register all at once and uses the available instructions to process the data faster. Since the width of the SIMD register file is 256 bits, we can operate on 4 double precision variables each time.

2 What we did

The project was mainly done in two stages: the initial stage and experimentation stage. In the initial stage, we used `dgemm_blocked.c` as a template to improve its performance by rearranging looping order and changing data structure of the matrices. After that, in the second stage, we try to implement a faster kernel function using AVX intrinsics.

2.1 Initial Stage

We have a seperate report that summarizes our work in this stage, hence here we only briefly mention some of the important testing we did in this stage. As discussed in section 1.2, the basic goal is to use block matrix multiplication to reduce the memory traffic. A naive implementation of this idea, as in `dgemm_blocked.c` , does not outperform the basic three-loop version. So we try a few things to get a speedup.

2.1.1 Better Looping order

The easiest thing was to change the loop order for dividing the blocks. The original code used looping order (i,j,k) and we tried all possible permutations to test the speed difference. (i, k, j) and (j, k, i) stands out to be much better than others. One possible explanation is that instead of fixing a submatrix $\mathbf{C}$ and sum over submatrices of $\mathbf{A}$ and $\mathbf{B}$, it is faster to fix either one of $\mathbf{A}$ and $\mathbf{B}$ and loop over I and J.

[1] `http://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz`

2.1.2 Blocking

The block matrix size is defaulted to 16×16 and we change that to observe the speed difference. After varying the block matrix size from 2 to 256, we observe that the speed goes up and down with a peak at 64. So we chose this block matrix size for our code.

We also noticed that there was room for improvement by introducing another level of blocking. The original blocking strategy only divides the matrices once before carrying out the computation. We can add one more blocking level to fit submatrices into L3 cache and then subdivide them to further fit them into the L1 cache before finally computing them in the registers. This way, we reduce the data transfer between cache and memory as well as between different levels of memory, which may further increase our performance.

2.1.3 Flags

We have tried different compiler flags in addition to `-O3` but they don't seem to have a significant effect on the performance of our code.

2.1.4 Transpose Copy

Another improvement is made by changing the data structure of our matrices. The given matrices are stored in its column form whereas the matrix multiplication is carried out assuming that A is in its row form. There are two disadvantages to using the original data structure. First, we will have to stride N values each time we try to access the next data. Second, the processor will load four consecutive entries in one column to take advantage of spatial locality but it won't be able to do so as long as our data structure stays the same.

So, to make sure we can benefit from the spatial locality, we preallocate a memory of size $n \times n$ for submatrix $\mathbf{A}$ and transpose it before passing it to the kernel function. To copy and transpose $\mathbf{A}$ will cost us $o(N^2)$ memory operation but can help us reduce the cost when we carry out $o(N^3)$ computation. Ultimately, we want the reduction in computing time to compensate for the relatively cheap overhead cost.

2.2 Experimentation Stage

The objective in stage 2 is to implement a fast kernel function to further increase the speedup from last stage. Our approach is to take advantage of the AVX2 intrinsics provided on the totient compute node.

2.2.1 AVX

We started from the old code from spring 2014[2]. This code uses AVX and assume register width of 128 bits that will fit two double precision variables. Because of this, the kernal function is built to compute matrices size $2 \times P$, where P is taken to maximize the use of all available registers.

The code carries out matrix multiplication by viewing $\mathbf{C}$ as a sum of outer products from matrix $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{C} = \sum_{j}^{p} \sum_{i}^{p} \mathbf{A}_{:,i} \mathbf{B}_{j,:}$$

[2]urlhttps://bitbucket.org/dbindel/cs5220-s14/wiki/sse

where : refers to the MATLAB notation of taking the entire row/column. It makes sense in this case because in the older architecture, shuffle and multiplication intrinsics use the same port and the shuffle function will make it easy for the compiler to schedule multiplies along with adds. For example, the code loads $\mathbf{A}_{:,1}$ and $\mathbf{B}_{1,:}$ into `_m128d` variables a, b, and the operation is carried out in the following sequence:

1. Call vector multiplication, `_mm_mul_pd(a,b)` and add the result to the diagonal elements of submatrix **C**.
2. Shuffle the elements in `b` using `swap_sse_doubles(b)`.
3. Call vector multiplication again and add the result to the off-diagonal elements of submatrix **C**.

Repeat the process P times to go through the entire submatrices.

Only a few changes need to be done for this kernel function to work with our existing program layout from the last stage. First, we need the submatrices to be stored on aligned memories (`_mm_malloc`) for the intrinsics to use them. Second we want to transpose **B** instead of **A**. Lastly, the matrix **C** has to be stored in diagonal form.

To make the AVX intrinsics work, we need to keep the variables on aligned memories. We will address this issue later in the report.

2.2.2 AVX2

After successfully running the AVX kernel function in our program, we move on to implementing the AVX2 intrinsics for better usage of the architecture. For AVX2, it mainly operates with `_m256d` intrinsics which handles 4 doubles at the same time. Therefore, our kernal function is able to deal with matrix of size $4 \times P$. For simplicity, we start with 4*4 case, which can be easily extended to more general case if needed. We have developed two ways of doing it

Matrix inner product We store submatrix A in the register using `_m256d` varibles a_i. Then we read each entry of matrix $\mathbf{B}_{ij}$ and copy it to fill a register vector, b. Then we sum up the the product $a_i\,.*b$. Then add this to a column of C until we finish updating the entire column of B. For example, we load $\mathbf{A}_{1,:}$ and $\mathbf{B}_{1,1}$ into a_i's and b then

(a) Mutiply the two vectors a_0 and b and add the result to c. We can use the FMA intrinsics that is new in AVX 2. So the code looks like `c = _mm256_fmadd_pd(a0, bij, c)`.

(b) We keep doing this for the entire column of B

(c) We loop through the columns to finish the matrix multiplication.

Because the entire submatrix **B** is small enough to fit into the register, the entry-wise operation will not cost us extra memory movement. Moreover, the fused multiplication add intrinsics should give us better performance than combining individual instruction.

Also note that in this case we do not need to make any changes to data structure.

Matrix dot product Another approach is similar to the AVX implementation. The idea comes from online discussion[3] It employs permutes of vector elements and carries out 4 vector dot product in one loop. Because our kernel is assumed to handle 4×4 submatrices, this implementation will work perfectly. The code first loads the entire submatrix **A** into the register using a_0, a_1, a_2, a_4 (row-form), and then it loads one column of **B** into b to perform a dot product:

[3] `http://stackoverflow.com/questions/10454150/intel-avx-256-bits-version-of-dot-product-for-double-precision-flo`

(a) It first perform all four vector products (entry-wise, not dot product):

```
__m256d xy0 = _mm256_mul_pd( a0, b );
__m256d xy1 = _mm256_mul_pd( a1, b );
__m256d xy2 = _mm256_mul_pd( a2, b );
__m256d xy3 = _mm256_mul_pd( a3, b );
```

(b) Then it adds up the vectors in pairs:

```
// low to high: xy00+xy01 xy10+xy11 xy02+xy03 xy12+xy13
__m256d temp01 = _mm256_hadd_pd( xy0, xy1 );
// low to high: xy20+xy21 xy30+xy31 xy22+xy23 xy32+xy33
__m256d temp23 = _mm256_hadd_pd( xy2, xy3 );
```

The comments show what matrix entries are in each data slot.

(c) To rearrange the entries, the code uses `_mm256_permute2f128_pd` and `_mm256_blend_pd`, which essentially swap the entries according to an extra control argument passed onto the function:

```
// low to high: xy02+xy03 xy12+xy13 xy20+xy21 xy30+xy31
__m256d swapped = _mm256_permute2f128_pd( temp01, temp23, 0x21 );
// low to high: xy00+xy01 xy10+xy11 xy22+xy23 xy32+xy33
__m256d blended = _mm256_blend_pd(temp01, temp23, 0b1100);
```

(d) Now that the entries are organized, we should add the two vectors using

```
__m256d dotproduct = _mm256_add_pd( swapped, blended );
```

and then add it to the column of **C**.

Repeat this process until we go through all the columns in matrix **B**.

This implementation requires us take transpose of A before passing it to the kernel function.

2.2.3 Copy & Transpose

Because AVX and AVX2 work only on variables on aligned memory space, we make explicit copies of submatrices before passing them to the kernel function. For each block, we preallocate aligned memory in the beginning of program using `_mm_malloc()`. A copy function is also made to load the submatrices given starting index (I_1, J_1). As discussed before, a transpose function can easily be modified from a copy function with little extra cost. Another extra cost comes from moving the data from submatrix copies back to the original data array.

An idea we try in copying a matrix is to enforce a regular computing pattern for the kernel function. More specifically, on the edges of matrices, instead of computing multiplication of rectangular submatrices, we fill the edges with 0's to make the block matrix square. Given the block size n, the extra compute cost is on the order of $o(nN)$ for the 0's. However, we are saving the if statements to check whether the block matrices are rectangular as well as the trouble to deal with the general case. A rought estimate says it takes order of $o(N^2)$ operations to check the block matrix shapes. Therefore, we can cut down some potential cost there.

Moreover, a regular computing pattern means we can choose the block matrix size such that it is conveniently divisible for the kernel function to process each subblocks. For example, if our kernel function handles matrices of size 4×6, we can choose an outer block matrix of size 12×12 or its multiples. This way, we can keep the fast kernel while tuning the block size to maximize the cache memory usage.

3 Results

Unfortunately, we don't see an improvement in speed after experimenting with the different ideas in the second stage. In the initial stage, we have achieved peak performance at around 6.7Mflops/s. As we try to improve the kernel function, the program will not behave well for optimization level higher than -O2. Because of this, the result from using AVX2 kernel does not exceeed 2Mflops/s, which is slower than the result from using the f2c implementation but faster than basic and blocked, assuming the same optimization level. We report the problem here for possible learning opportunity.

The program will compute **C** and return correct value when compiled with -O1 level, but it will set all entries of **C** to 0's when compiled with -O2 and -O3 level. It is natural to assume an indexing bug in the program but we can't find any. To make matter worse, if we add a `printf` function right after the kernel function, the program will return correct value even when compiled with -O3 optimization. Certainly, the flop rate is low due to the `printf` function. Once the `printf` function is moved outside the kernel loop to print the higher level block matrix, the code breaks down and set all values to 0's again.

A possible cause for this behavior is we are updating the columns of **C** in the kernel function too fast without giving enough attention to the latency in each intrinsics. Also, it could be the compiler try to erase temporary variables that has no temporal locality. Later it was suggested that we use valgrind to debug memory problems. A problem with debugging is that the AVX2 intrinsics will not run on the head node because its lack of instruction set. When we try to debug on the compute node, the program does not return any useful message for us. Due to time limit, the issue is left unresolved because we want to experiment with other aspects of the problem as well.

Blocking One of the reasons for us to move from AVX to AVX2 is because the 2×2 kernel function is extremely slow, even compared to basic implementation. According to the theory, we only get a performance boost if our block matrix is big enough. In other words, the arithmetic intensity is o(N/p) where p is the number of submatrices we need to cover the entire matrix. In the limit of $p \to N$, we go back to the linear case, which is essentially the basic implementation. Therefore, we want the kernel function to be as large as possible. Therefore, a kernel function that efficiently deals with 4×4 matrices is much more desirable than the one optimized for older cluster.

That being said, block matrices of 4×4 will never be able to maximize the use of cache memory. A logical move is to build more block level targeted at filling different cache levels. However, each blocking stage comes at extra cost to copy the original matrix, so we want to balance the overhead cost as well. In addition, because we are enforcing a regular computing pattern by filling incomplete submatrices with 0 entries, the overhead cost also comes from computing those extra entries.

We first use three-level blocking and make copies of all block matrices for every level. We observe speedup from using the kernel function alone, but the speed seems to be limited at 2Mflops/s. We suspect that the copy function is not efficient and takes too much memory cost for the kernel function to compensate. As a result, we changed the strategy to pass the outer matrices by pointers and starting indices. Then we make explicit copies of a moderate sized submatrix. This avoids the cost to make copies of the submatrices and make the cost to compute edge cases smaller.

We also use different optimization flags to compensate the disadvantage of not having -O2 optimization. It doesn't work well for our function but some flags improve the performance of naive blocked version made it faster than the basic version.

4 Analysis

Although it is not clear why the program stops working with higher level compiler optimization, it is still worth mentioning that the best performance does not surpass basic and f2c that are compiled with the same optimization level. Again, according to the roofline performance model, the performance is not limited by the cluster's computing capacity but rather its memory transfer capacity. The advantage of having a fast kernel is not obvious if the memory transfer capacity is the main bottleneck here.

The other issue is if we should make explicit copies of submatrices. We observe a significant increase in the speed when we use blocking, but later we see that the speed is limited to 2Mflops/s no matter what kernel function we use. The major cost comes from copying the matrices and to enforce a regular pattern. When the sizes of original matrix and its submatrix are comparable, the copying operation in fact kills the performance. The blocking strategy will not work unless we spend time toning it.

Lastly, profiling is absent in our attempts to optimize the performance. We followed the suggestions to start from a fast kernel and build multi-level blocking to optimize the program from inside, however, it is later realized that the effect of our work cannot be properly reflected in the final result. We need a testing framework for kernel function, which is missing in the beginning phase of testing. In retrospect, we should have put more work into building a testing frame and profiling system to get feedback for the work we do.

5 Conclusion

There are certainly many other things to try for better performance. First, we can try to fix the kernel function and hopefully it will get better performance. Also, we have lots of ideas to try to manage the memory traffic more efficiently. Instead of just using loops, we can create a function that assign the submatrices and pass them onto the next level function. The benefit of this is we do not have to make changes to the loops every time we make slight changes to the kernel function. And since we need multi-level blocking, using function will make the code much cleaner and easier to debug. Lastly, the idea of recursive division of the matrix will require us have a function and our efforts will add to that.

Using intrinsics is a good learning exercise but it does not yield any fruitful results. By the time we realize the code is hard to debug, we have already got too invested in the work to give it up. A compiler optimized code gets very good performance, especially when we use "icc" instead of "gcc". Moreover, the platform limitation of the AVX intrinsics really makes it hard to generalize and limited. Time invested in improving the memory traffic is probably more rewarding.

Lastly, the absence of profiling in the development really makes the work inefficient. Although we realize the testing frame and profiling are needed from the early stage of the project, little work has been done to carry it out. In the end, unnecessarily much work has been put into implementing a fast kernel and we have little time left to investigate other aspects of the project.

It is overall an exciting learning experience, but the poor organization of workflow makes our work inefficient.