FNNMF: A FAST IMPLEMENTATION OF NON-NEGATIVE MATRIX FACTORIZATION

Attila Hirschi, Martina Kessler, Luca Multazzu, Franklyn Sciberras

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Non-Negative Matrix Factorization (NNMF) is a key component of many systems across different fields. Its performance is the key to making multiple applications and systems run efficiently. FNNMF aims at providing a fast implementation of the Multiplicative Update algorithm for Non-Negative Matrix Factorization tailored to the Intel i7-7500U (Skylake) micro-architecture. Employing a series of optimization techniques, including blocking for cache and registers and introducing vector instructions, our implementation manages to obtain speed-up up to 42x compared to the basic implementation.

1. INTRODUCTION

Motivation. Non-Negative Matrix Factorization (NNMF) has several practical applications in a variety of fields.

In astronomy, since astrophysical signals are non-negative, NNMF is used for dimension reduction [1]. In population genetics it can be used to efficiently infer individual ancestry coefficients [2]. Furthermore, NNMF can also be used to predict the scalable internet distance efficiently [3]. Other application include data imputation [4] and bioinformatics [5].

NNFM is therefore a key component in many applications and its performance is of the utmost importance. The algorithm often needs a large number of iterations to converge, especially for large matrices. If not optimized this can translate to a rather long run-time. Therefore, it often ends up being a bottleneck, making the surrounding system less viable for deployment. For example, in population genetics, NNMF was specifically introduced because it outperformed the existing computer programs, an advantage which relies on the performance of the code.

Contribution. In this paper we present an optimization over the basic implementation of the NNMF algorithm. This implementation is particularly targeted at square or close-to-square matrices but still performs well for more skewed sizes. We in-lined the function calls and then proceeded to gather the several existing loops into 4 main ones. We then blocked each of them for cache and registers, using a parameter search to determine the best block sizes. Finally, we employed 256 bit vector instructions to gain more speed-up by parallelising the computation.

Related Work. The original algorithm that we examine in this paper, Multiplicative Update (MU), was proposed in 2001 by Lee, Daniel and Seung, H. Sebastian [6]. The majority of optimized NNMF libraries are written for MatLab: the Mathworks toolbox [7] which provides multiple algorithms and is among the most widely deployed code. Cichocki et al. [8] is another MatLab library specifically targeted at using NNMFs for signal and image processing. Dhillon et al. [9] provide an optimization in C++ which takes advantage of BLAS routines to gain speed-up. LibNMF [10] is one of the most comprehensive existing libraries for NNMF and one of the few written in C. It too makes use of BLAS and LAPACK routines and supports multi-threading.

Our work aims at optimizing the MU algorithm, rather than providing an optimization for multiple existing methods to compute the NNMF. As opposed to the majority of the state of the art, our work is coded in C, which empowers us with increased flexibility and further room for optimization. We aim to eliminate the use of BLAS calls and instead optimize the code manually, focusing on a specific micro-architecture to best exploit both the algorithm's as well as the machine's properties, achieving an overall speed-up. The motivation is that our work should provide a solid basis for a more generalised approach, which would encompass the requirements that a fully-fledged library requires. Specifically, all the arguments and results put forward in this study were done targeting an Intel i7-7500U (Skylake) CPU.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

In this section, we provide an overview of the algorithm and its characteristics. We then proceed to analyze its properties, from which we derive an overall performance cost analysis.

Overview. Given as input a matrix **V**, its dimensions **m** and **n**, and a third dimension **r**, the aim is to output two matrices **W (m x r)** and **H (r x n)** such that W x H is close to input V. We define close as the absolute value of the difference of each element of V with the corresponding value of W x H, being under a threshold value EPS.

Algorithm 1 Multiplicative Update NNMF

```
1: Inputs: V, m, n, r
2: Outputs: W, H
3: while not $are\_close(W \times H, V, EPS)$ do:
4:     for all i, j do:
5:         $H[i][j] = H[i][j] * \frac{(W^T \times V)[i][j]}{(W^T \times W \times H)[i][j]}$
6:     for all i, j do:
7:         $W[i][j] = W[i][j] * \frac{(V \times H^T)[i][j]}{(W \times H \times H^T)[i][j]}$
```

The Algorithm. The algorithm initializes W and H, then updates them for an indefinite amount of iterations until their product is close to V. Each element of H is updated by taking the corresponding element of W^TV, dividing it by the corresponding element of W^TWH and then multiplying by the current element of H. Every value of W is then updated by multiplying it with the division of the corresponding element of VH^T and WHH^T.

Initialization of W and H. We initialized H with random values, which is one of the initialization approaches proposed by Hafshejani et al. For W we decided to use another of their propositions; a variation of the *Random C* method [11]. Provided with the matrix V, m, n, r, and two values q and pool size (q $\leq$ poolsize), the algorithm orders the columns of V by the second norm, randomly selects q out of the first pool size largest columns of V, and sets the first column of W as the mean of the selected vectors. Then it repeats this process for all remaining columns of W.

Algorithm 2 Random C

```
1: Inputs: V, r, q, poolsize
2: Outputs: W
3: Set: k = 0
4: $V_{sorted} = sort\_columns\_descending(V)[: poolsize]$
5: while $k <= r$ do:
6:     largest[][] = $select\_q\_random\_columns(V_{sorted})$
7:     s[] = $find\_mean\_column(largest)$
8:     W[:][k] = s
9:     k+=1
```

Cost Analysis. The algorithm for NNMF requires a variety of operations involving floating-point numbers, including additions, multiplications and divisions. Therefore, we chose the following cost function:

$$C(m,n,r) = (\textbf{adds}(m,n,r), \textbf{mults}(m,n,r), \textbf{divs}(m,n,r))$$

Where adds, mults and divs stand for floating point additions, multiplication and divisions respectively, as components of the determined cost function. We then manually computed the cost, based on algorithm analysis, with the following result:

adds	NI * (5*m*n*r + m*n + r*m*r + r*r*n) + m*n*r
mults	NI * (5*m*n*r + r*m*r + r*r*n + r*n) + m*n*r
divs	NI * (m*r + r*n)

Where NI is the number of iterations the algorithm takes to converge.

3. YOUR PROPOSED METHOD

In this section, we explain the different steps we took to optimize the code. We used an incremental approach. It consisted of first writing the basic implementation as described above and then setting a performance optimization baseline using BLAS calls. Then we analyzed the state of the algorithm and incrementally employed different optimization techniques on the current state of the algorithm. With each optimization step, we measured the performance achieved and analyzed the effects of the employed optimizations. We then determined whether to discard, modify or commit to such techniques based on the effectiveness in improving performance.

Basic Implementation. As mentioned we started by creating a basic implementation of the algorithm using **Algorithm 1** and **Algorithm 2** as our skeleton. We proceeded with creating two helper functions, which we use throughout the implementation, upon the multiple temporary matrices employed, to aid with satisfying the described algorithms. The first helper function consisted in transposing a matrix and storing the result explicitly. The second function performs matrix-matrix multiplication using the classical triple loop approach. Upon analysis, our basic implementation proved to be memory inefficient, where each matrix-matrix multiplication result was stored in a separate temporary matrix. Listing 1 depicts an overview of such an approach, which we will be using as a running example to demonstrate how our code evolved over the different optimization steps. The full implementation along with the rest of the optimizations described in this section can be found on our repository [12].

```
transpose(H,r,n,Htranspose);
multiply(V,Htranspose,m,n,r,VH);
multiply(W,H,m,r,n,WH);
//where Htranspose, VH, and WH are all temporary matrices
```

Listing 1. Basic implementation example

BLAS. BLAS is an interface that provides functions for simple matrix and vector operations. Given that is used by most of the state-of-the-art libraries being used in the industry, we decided to use it to set an optimization baseline against which we could compare our work. To achieve this we used the BLAS library [13] to remove explicit transposes and triple loop matrix-matrix multiplications and replace them with calls to the function SGEMM (single precision matrix-matrix multiplications). The shortcoming of this implementation lies within the fact that even though it optimized matrix-matrix multiplications, it does not take into account that we are using the same three matrices V, W and H over and over again. Furthermore, the installed BLAS version (3.10.0) proved to be not as efficient as other versions on the market, so the full benefits of such an approach were not observed. As a second baseline and for future work, using OpenBLAS might prove to yield better results [14].

```
 1 for (int i = 0; i < m; i++) {
 2   for (int j = 0; j < r; j++) {
 3     sum_VH = 0;
 4     w_reuse = W[i * r + j];
 5 
 6     for (int k = 0; k < n; k++) {
 7       h_reuse = H[j * n + k];
 8       sum_VH += V[i * n + k] * h_reuse;
 9 
10       if (j == 0) {
11         WH[i * n + k] = 0;
12       }
13 
14       WH[i * n + k] += w_reuse * h_reuse;
15     }
16 
17     VH[i * r + j] = sum_VH;
18 }}
```

Listing 2. Loop gathering example

Loop Gathering. In this optimization step, we attempt to start addressing the most evident shortcomings which emerged within the basic implementation, mainly the excessive memory allocation and all the side effects this brings about (limited cache locality, heavy cache evictions etc.). As a first step, we inlined the matrix-matrix multiplication code to enable further optimizations down the line. We proceeded with removing procedure calls to library functions as well as local functions. The approach here was to replicate the same functionality of these procedure calls, within the main algorithm. More concretely, an example of this is the transpose function, which could be substituted by a change in the access pattern for transposed matrices.
In the next step, we unified loops that accessed the same matrix to improve temporal locality. In total, we end up having four loops, with loop 3 being portrayed in Listing 2. In this code snippet it can be seen that the matrix-matrix multiplication of $V * H$ and the matrix-matrix multiplication of $W * H$ access the same matrix H. Therefore, we did scalar replacement to load the value of H only once. Here, one can also observe the aforementioned transpose procedure call replacement. In this optimization step, we were also able to completely remove one matrix-matrix multiplication of $W * H$ because through inlining it became evident that such values were already being calculated in the verification part of the algorithm when $W * H$ is assessed for its closeness relative to V. As such, we can reuse this precomputed matrix to update H accordingly. We were also able to remove extra memory allocations for temporary matrices W^TWH and WH because they were only being used locally to update H. Instead of storing those values in a matrix and then updating H, it was obvious that we could store the values in a local variable and directly update H, reducing potential access bottlenecks.

Instruction-Level Parallelism (ILP). The next bottleneck that we wanted to tackle was the lack of Instruction Level Parallelism in certain sections of the code. This phenomenon mainly occurs when you have one operation within an iteration which is dependent on the result obtained in the previous iteration, preventing the CPU from executing these operations from separate iterations in parallel. An example of this can be seen in Listing 2 on line 8, where we are dealing with a matrix-matrix multiplication which is utilizing the += operator as part of the operation, introducing this sequential dependency. To avoid this, we introduced 4 accumulators, where after conducting a parameter search it was deemed to be the optimal accumulator size for the microarchitecture in use. Therefore, we unrolled the loops by a factor of four, where in conjunction with the four accumulators we were able to create enough independent instructions to fill the pipeline. This step proved to be very helpful when implemented on top of Blocking for Registers, which will be discussed further on.

Blocking for Cache. Our code is dealing with potentially very large matrices that do not fit entirely into cache. The loaded elements will most certainly be evicted by the time we will need to reuse them. To alleviate this problem, we blocked our code for the cache. We did not add this optimization on top of the ILP version described in the previous paragraph because we can get ILP for free when we do clever blocking for registers. Instead, we added this optimization on top of the loop gathering version. To simplify the code, we only used floats and we assumed that the matrix size is a multiple of the block size, which was deemed to suffice for the current scope. Naturally, irregular matrix sizes would need to be taken into account when generalizing the optimizations for a wider scope of use. To find a good block size, we ran a parameter search using a large enough matrix of size m=n=r=1024. We used only powers of two for the block size due to the aforementioned assumption. To simplify things, we only tested the block sizes [8, 16, 32, ..., 512]. We restricted our search to this range because for every matrix-matrix multiplication, at least two matrices are involved, which results in 2 * block_size * block_size elements that need to fit into the cache. The L1 cache size of the Intel i7-7500U (Skylake) CPU is 128 KB. Thus, with floats using any block size which is larger than 127 would theoretically result in the elements overflowing our cache. However, we used 512 as the largest size just to make sure that we do not exclude a possible block size. Based on this parameter search it turned out that the optimal block size for our machine among the tested sizes is 64.

Blocking for Registers. As our next optimization, we expand on the rationale explained in the previous section. This time we also keep in mind locality in relation to the CPU registers. Having this micro-architectural knowledge, we try to optimize our access patterns to make use of the data that the CPU has loaded in registers. After running a parameter search on our evaluation machine, it was determined that 16 floats would be the ideal register size to use when blocking for this optimization, whilst still adhering to the previous assumptions. During this step it was also noted that if we introduce a clever access pattern on the innermost loops, we could also obtain instruction-level parallelism for free. This would imply that any dependency between one loop iteration and the other is removed, allowing for more computations to be done in parallel and thus better performance. A example of this can be seen in Listing 3. On line 13 we see how we are accessing matrix WH over index kkk, meaning that with each iteration we are writing to an independent address of the matrix ensuring ILP. This is done whilst also getting spatial locality on each read access to the same matrix. Similarly, we are also getting spatial locality with reading accesses to matrix H using the same pattern. On the other hand, one can observe how read accesses to matrix W are being made more efficient by exploiting temporal locality in between iterations on the inner loop, and spatial locality on each iteration over jjj. The innovative part of this optimization can be summed up in the portrayal of lines 15-17. Here one can observe how it is possible to achieve better locality by splitting the two computations over two separate access patterns at the same level of nesting. We can see how the computation over matrix VH obtains a better speed-up if it is accessed over index jjj rather than kkk. With this slight alteration, we manage to introduce ILP on the write accesses to VH and spatial locality on the read accesses, whilst not hindering access to WH. When it comes to read access to matrices V, we manage to get temporal locality in between iterations on the inner loop, and spatial locality on each iteration over kkk. For the matrix H we have a column-wise access pattern. When choosing the block size small enough, we manage to get temporal locality thanks to the fact that always a block of 64 bytes is loaded into cache.

```
1 for (i = 0; i < m * n; i++){ WH[i] = 0.0; }
2 for (i = 0; i < m * r; i++) { VH[i] = 0.0; }
3 for (i = 0; i < m; i += BLOCKSIZE) {
4   for (k = 0; k < n; k += BLOCKSIZE) {
5     for (j = 0; j < r; j += BLOCKSIZE) {
6       for (jj = j; jj < j + BLOCKSIZE; jj+=NU) {
7         for (ii = i; ii < i + BLOCKSIZE; ii+=MU) {
8           for (kk = k; kk < k + BLOCKSIZE; kk+=KU) {
9             for (iii = ii; iii < ii+MU; iii++){
10              for (jjj = jj; jjj < jj+NU; jjj++){
11                for (kkk = kk; kkk < kk+KU; kkk++){
12                  WH[iii*n+kkk] += W[iii*r+jjj] * H[jjj*n+kkk];
13                }
14              }
15              for (kkk = kk; kkk < kk+KU; kkk++){
16                for (jjj = jj; jjj < jj+NU; jjj++){
17                  VH[iii*r+jjj] += V[iii*n+kkk] * H[jjj*n+kkk];}}}}}}}}}
```

Listing 3. Blocking for cache and registers example

Evaluation of Parameter Search. In the previous 2 sections, it was mentioned that a parameter search proved to be a very important step when identifying ideal blocking sizes both for cache as well as for register. One has to emphasize the fact that such value could vary based on 2 factors: mainly the micro-architecture on which the library is running, as well as the matrix sizes assumed during such parameter search. As such, in a full-fledged release environment, one should have some setup code to determine these values before running the computation. Yet, whilst developing this setup code, the introduced overhead should be kept in mind. The setup code should be run frequently enough to provide overall good parameters which give a good speed-up, but not to the point where finding the best parameters becomes the computation's bottleneck. Due to time restrictions, in our implementation, we did not have the resources to explore this avenue in further detail. As such we only conducted the parameter search for a single matrix size where m=n=r=1024, on a single micro-architecture. From the achieved results it was evident that the chosen parameters have a big effect on the overall speed-up. This also implicates that a variance in the input matrix size could further benefit from a different set of parameters. Accounting for such variances could be considered future work.

Vectorization. The next natural step of our optimization process was to attempt to vectorize manually using intel intrinsics, rather than leaving it up to the compiler. Although we expected our algorithm to be memory bound, we were still confident that by optimizing as best as we could at the computation level, exploiting the flexibility that such intrinsics provide, we could achieve further speed-up. This increased flexibility came at the cost of a bigger challenge when it comes to adapting the code to fit within the vectorization model, as well as decreased code legibility.
To alleviate such challenges, we introduced an intermediary step before vectorizing, where we explicitly disassembled our blocked code by first unrolling and then reassembling into summation counters. Now that our code base was disassembled, it became much easier to identify locations and patterns where multiple computations could be done in parallel. This gave us the required visibility and scope to be able to introduce vectorization, albeit as expected, on its own, it did not help with performance. As such vectorization using 128 bits intrinsics was introduced. This would allow us to run 4 computations in parallel if we were to use floats for our computations, and 2 if we had to use doubles. Overall this reduced the number of cycles required, however, we could still do better. By introducing vectorization using 256 bits intrinsics, we could perform 8 computations in parallel using floats, and 4 using doubles. In Listing 4, one can see how in our final optimization we ended up with a mix of both versions of intrinsics, due to the nature of the application.

One key global aspect that came into play in this step (concerning both 128 bit intrinsics as well as 256 bit ones), was memory alignment. To reduce latency, we tried to utilize intrinsics which work with aligned memory, where generally they offer less latency compared to their unaligned memory counterparts [15]. However, this also comes with the implication that the developer needs to ensure that memory is always correctly aligned. To achieve this, memory was allocated using *aligned_malloc* and its counterpart *aligned_free*, to free the concerned memory [16]. However, a cross-compatibility issue related to VTune running on a Windows environment, prevented us from using such a utility on Windows. As such memory-alignment-aware allocation and freeing functions were built to be able to keep utilizing the lower latency intrinsics.

```
 1 for (i = 0; i < m * n; i++) { WH[i] = 0.0; }
 2 for (i = 0; i < m * r; i++) { VH[i] = 0.0; }
 3
 4 for (i = 0; i < m; i += BLOCKSIZE) {
 5   for (j = 0; j < n; j += BLOCKSIZE) {
 6     for (k = 0; k < r; k += BLOCKSIZE) {
 7       for (ii = i; ii < i + BLOCKSIZE; ii += MU) {
 8         for (jj = j; jj < j + BLOCKSIZE; jj += NU) {
 9           for (kk = k; kk < k + BLOCKSIZE; kk += KU) {
10             for (iii = ii; iii < ii + MU; iii++) {
11               wh_ptr = &WH[iii * n + jj];
12               wh1 = _mm256_load_ps(wh_ptr);
13               wh3 = _mm256_load_ps(wh_ptr + 8);
14               int iiir = iii * r;
15               for (kkk = kk; kkk < kk + KU; kkk++) {
16                 h_ptr = &H[kkk * n + jj];
17                 w = _mm256_set1_ps(W[iiir + kkk]);
18                 h1 = _mm256_load_ps(h_ptr);
19                 h3 = _mm256_load_ps(h_ptr + 8);
20                 wh1 = _mm256_fmadd_ps(w, h1, wh1);
21                 wh3 = _mm256_fmadd_ps(w, h3, wh3);
22               }
23               _mm256_store_ps(wh_ptr, wh1);
24               _mm256_store_ps(wh_ptr + 8, wh3);
25 }}}}}}}
26
27 for (i = 0; i < m; i += 4) {
28   for (j = 0; j < r; j += 4) {
29     //vh_vars represents vars[vh11, vh44]
30     vh_vars = _mm256_setzero_ps();
31     for (k = 0; k < n; k += 8) {
32       // Load from v=[v1, v4] and h=[h1, h4]
33       v = _mm256_load_ps(&V[(i+[0,3]) * n + k]);
34       h = _mm256_load_ps(&H[(j+[0,3]) * n + k]);
35
36       // Add all i=[1,4] and j=[1,4] combinations along k and store in
          respective vh_vars variable
37       vhij = _mm256_fmadd_ps(vi, hj, vhij);
38     }
39     // Add horizontally up to vh434
40     vh112 = _mm256_hadd_ps(vh11, vh12);
41     vh134 = _mm256_hadd_ps(vh13, vh14);
42     ...
43
44     // Split 256 bit line into 2 lines of 128 bits and add them together - this
           needs to be done until vh434_128
45     vh112_128 = _mm_add_ps(_mm256_castps256_ps128(vh112), _mm256_extractf128_ps
         (vh112, 1));
46     vh134_128 = _mm_add_ps(_mm256_castps256_ps128(vh134), _mm256_extractf128_ps
         (vh134, 1));
47     ...
48
49     // We collapse using hadd
50     vh1 = _mm_hadd_ps(vh112_128, vh134_128);
51     ...
52     vh4 = _mm_hadd_ps(vh412_128, vh434_128);
53
54     // Store result: vh=[vh1, vh4]
55     _mm_store_ps(&VH[(i + [0,3]) * r + j], vh);
56   }
57 }
```

Listing 4. Snippet from final optimization

Choosing the Optimization's Supported Data Types. As briefly mentioned in the previous section, during our optimization process, we were always facing the dilemma of whether to choose floats or doubles as the data type of our optimization, or whether to support both. The dilemma was always a question of an increased number of computations running in parallel when vectorizing at the cost of lower accuracy. After analysing the nature of our algorithm it was concluded that matrices W and H only provide an approximate value when multiplied and compared to the input matrix V, with a degree of accuracy up to a certain threshold which determines the point of convergence, implying that the algorithm is of an approximative nature. As such it was decided that we could afford to lose a few bits of precision, whilst gaining a higher speed-up, by utilizing float as our supported data type. We also concluded that supporting both data types would have not provided any added benefit for the extra effort.

Handling Aliasing. A prevalent aspect which can be noted throughout our optimised code, more specifically in the last optimization is the handling of memory aliasing. Due to the nature of our algorithm, we tried to minimize the total amount of memory allocated and reuse as much as possible. This was bound to create some aliasing, where we would be accessing the same memory locations from different parts of the computation. To ensure that we would not be creating unnecessary bottlenecks for our compiler, we tried to adopt the technique of first loading the necessary data from memory into the local scope. Then performing our computations within that scope and only when we are done, we would store back the results in the respective memory locations. Lines 12 to 24 in Listing 4 outline the idea, even though this comes for free when using intrinsics, due to the framework enforcing this coding style.

4. EXPERIMENTAL RESULTS

In this section we show the setup of the benchmark[12] and interpret the results of the measured performances.

Experimental setup. We run our implementations on an *Intel i7-7500U* (Skylake/Kaby Lake architecture), with a base frequency of 2.9GHz and L1/ L2/ L3 cache sizes of 128 KB/ 512 KB/ 4 MB. For compilation, we used *GCC* version 9.4.0 with the flags:
`-O3 -ffast-math -march=native`.

We measured the run-time (in CPU cycles) as follows: For each implementation and input sizes M, N and R, we generate two different matrices V with a set seed. Each matrix V is then run three times against a given implementation with 10 iterations, and the median is taken.

As for the input sizes M, N and R, we have four experimental classes where all values start with the size 128 and then increase each one (or all) with a step-size of 128 until 1280. The only exception is when we increase along the parameter R, where we keep M and N at 1280 as it is not reasonable to have a higher value of R than M and N.

To measure the run-time in CPU cycles we used the *tsc_x86* C library [17] because, among the different options,

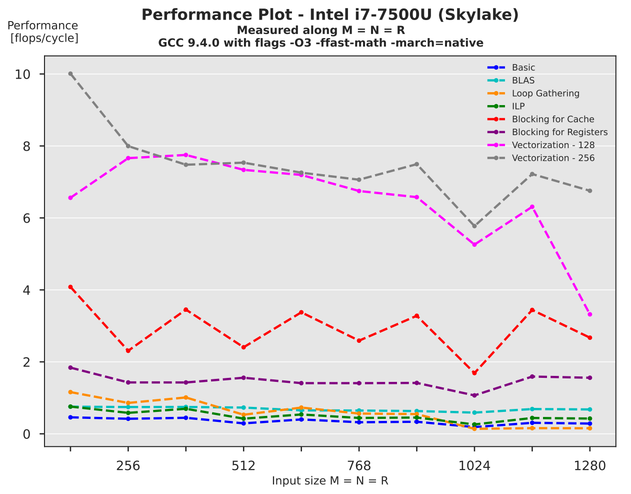

Fig. 1. Performance along M, N and R.

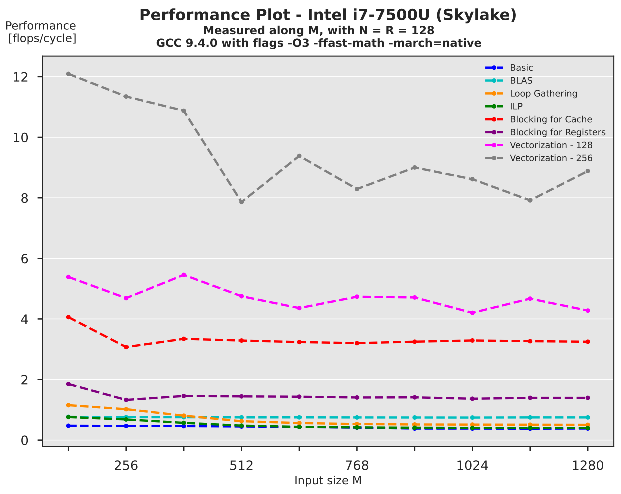

Fig. 2. Performance along M.

it gave the most consistent results.

To gather the data, specifically the data movement information, we used the VTUNE [18] profiler. We ran the memory access analysis, collecting, among other information, the number of loads and stores.

For the roofline analysis and the calculation of the speed-up vs. basic we used squared matrices with size $M = N = R = 1280$.

Results. Case 1 (along $M = N = R$): In Fig. 1, we can see that after blocking, we gain a significant performance increase, and a huge performance increase once the code is vectorized. Blocking for registers falls short compared to blocking for cache, this is only the case when the flag `-ffast-math` is used, therefore we hypothesize that the compiler has more freedom to reorder operations; inspecting the binaries, both were vectorized to a large part, but blocking for registers had a lot more nested code as well as vector shuffling. Blocking for cache (and also to some degree the vectorized versions) exhibits a zig-zag like pattern, which is likely due to cache misses with a power stride pattern. The *BLAS* version only gains some traction after a large enough input size. The vectorized versions perform very similarly for smaller matrix sizes, and the 256 bit vectorized code shows a significant improvent over 128 bit only when the matrix sizes are large. The memory bandwidth may be the reason behind it, limiting the ability of the CPU to fully benefit from the larger vectors. The maximum speed-up over the basic implementation is at $M = N = R = 1024$ with a factor of 30.67. We reached 31.3% of the peak performance in this setup.

Case 2 (along M, with $N = R = 128$): In Fig. 2, we increase the size of the number of rows (M) of the input matrix, keeping the other two sizes constant. Compared to Case 1, we see the vectorized code for 128 bit instrinsics is slower. Furthermore, the zig-zag pattern induced by cache misses is less evident for the blocking for cache and registers version. This plot displays that the final implementation performs similarly even when the input matrix has a much larger number of rows than columns. The maximum speed-up over the basic implementation is at $M = 128$ with a factor of 25.55. We reached 37.8% of the peak performance in this setup.

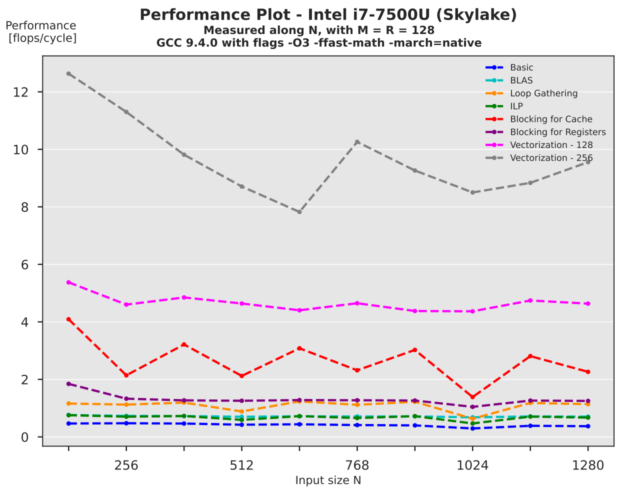

Fig. 3. Performance along N.

Case 3 (along N, with $M = R = 128$): In Fig. 3, we scale up N (the number of columns of the input matrix) and notice similar results to Case 2, with the vectorized version with 256 bit instructions showing significant speedup over all other versions, including the 128 bit vectorized code. Moreover, we see a re-emergence of the zig-zag pattern for blocking for cache which appeared in case 1. Overall, this plot shows that even when V is disproportionally wide the

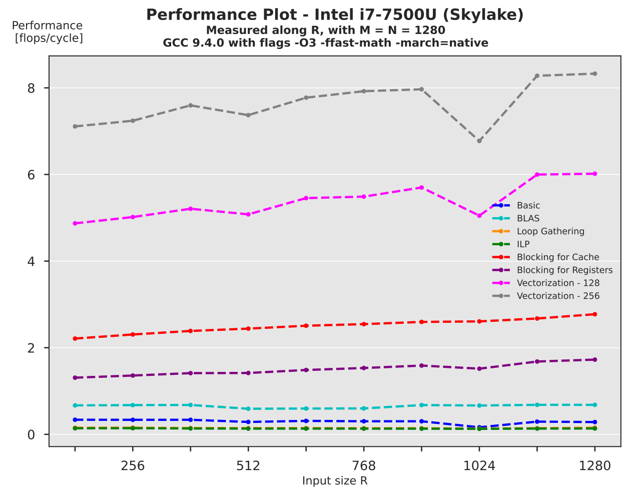

Fig. 4. Performance along R.

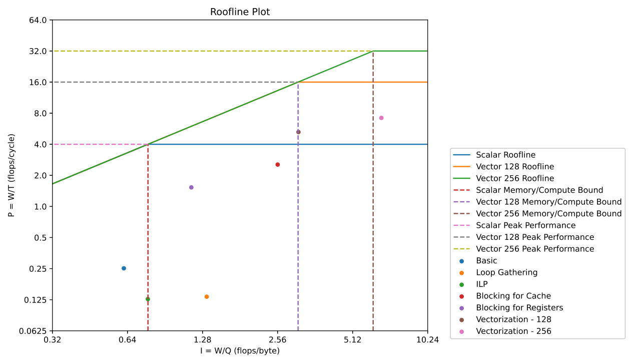

Fig. 5. Roofline plot of all implementations, obtained by *VTune* with $M = N = R = 1280$ and 5 iterations.

optimized code still performs as expected. The maximum speed-up over the basic implementation is at $N = 1024$ with a factor of 28.91. We reached 39.5% of the peak performance in this setup.

Case 4 (along R, with $M = N = 1280$): In Fig. 4 we can see that, in contrast to the other cases, the performance increases when R approaches M and N. There is a significant performance drop in the vectorized versions at size 1024. The loop gathering implementation is overlapped by the ILP implementation. The maximum speed-up over the basic implementation is at $R = 1024$ with a factor of 41.57. We reached 26.0% of the peak performance in this setup.

Roofline Plot: Fig. 5 shows the roofline plot. The microarchitecture we were using has memory bandwidth of $\beta = 5[B/cycle]$. The peak performance with 256 bit vectorization is $\pi = 32\ [flops/cycle]$. We observe that, under the peak roofline, the basic implementation is significantly

Optimization	Speed-up vs. Basic
BLAS	2.5x
Loop Gathering	0.57x
ILP	1.57x
Blocking for Cache	9.84x
Blocking for Register	5.7x
Vectorization 128 bits	12x
Vectorization 256 bits	25x

Table 1. Speed-up of the optimization compared to the basic implementation for matrices of size $M = N = R = 1280$.

memory bound. With loop gathering, we managed to improve the operational intensity, but decreased the performance. However, this step was necessary to enable further optmizations. As it can be seen, blocking for cache and register have both increased operational intensity and performance. Our fastest version, vectorization with 256 bits, is the only version which is slightly compute-bound. In the roofline plot and the four performance plots it can be seen that we manage to have a performance of roughly 8 flops/cycle for the vectorization with 256 bits. This is 25% of the peak performance.

Speedup vs. Basic: In Table 1 we show the speed-up of each optimization with respect to the basic implementation. It can be seen that the loop gathering optimization is slower for squared matrices of size 1280. This is only the case for this large size, for smaller sizes such as 1024 this optimization improved the performance. One possible reason for this slow-down is that on large sizes is that doing the transposes within the multiplication and gathering loops produces a worse access pattern for locality, but in the long-term this is helpful for blocking and vectorization. As already discussed in the case 1 there is a slow-down for blocking for register compared to blocking for cache. Finally, we can report a speed-up of up to 42x for our most optimized version compared to the basic implementation.

5. CONCLUSIONS

Non-negative matrix factorization is a very important building block in a variety of applications. Therefore, its performance is of great importance. In this work, we applied several methods, such as blocking, vectorizing and optimizing for ILP, to optimize the algorithm.

Our results show a significant speedup of up to 42x of FNNMF over the original implementation. Furthermore, the roofline analysis suggests that we were able to move from a significantly memory bound code, to one that is on the border between memory and compute bound.

Further work could be done on the auto-tuning scripts which determine the block-size, since they were tuned for only specific values of M, N, R. Moreover, it may be possible to achieve further speed-up by changing loop-ordering whenever the 3 dimensions are skewed (for example M is much larger than N and R).

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Attila Hirschi. Worked with the rest of the team on the *loop gathering* phase. Wrote the blocking code for loop 3 and 4. Vectorized loop 4 and the threshold condition. Worked together with Martina on the parameter search for the block and register size. Wrote the benchmark and performance plot scripts.

Martina Kessler. Implemented the BLAS version. Worked with the rest of the team on the *loop gathering* phase. Worked together with Attila on the parameter search for the block and register size. Introduced vector instructions (128 bits and 256 bits) to loop 3. Worked, along with the rest of the team, on installing VTUNE and using it to analyse the code and to gather data for the Roofline plot.

Luca Multazzu. Worked with the rest of the team on the *loop gathering* phase. Tackled the task of adding instruction-level parallelism (ILP) to the gathered loop implementation. Introduced vector instructions (128 bits and 256 bits) to loop 1 and loop 2 together with Franklyn. Worked, along with the rest of the team, on installing VTUNE and using it to analyse the code and to gather data for the Roofline plot.

Franklyn Sciberras. Wrote the initialization algorithm for matrix V using Random C. Worked with the rest of the team on the *loop gathering* phase. Wrote the blocking code for loop 1 and 2. Wrote the blocking code for registers on all 4 loops. Introduced vector instructions (128 bits and 256 bits) to loop 1 and loop 2 together with Luca. Worked, along with the rest of the team, on installing VTUNE and using it to analyse the code and to gather data for the Roofline plot. Wrote the roofline plot script.

7. REFERENCES

[1] R. Tandon and S. Sra, "Sparse nonnegative matrix approximation: new formulations and algorithms," Tech. Rep. 193, Max Planck Institute for Biological Cybernetics, Tübingen, Germany, Sept. 2010.

[2] Mathieu F Frichot E, Trouillon T, Bouchard G, and Francois O, "Fast and efficient estimation of individual ancestry coefficients," 2014.

[3] Yun Mao, Lawrence K. Saul, and Jonathan M. Smith, "Ides: An internet distance estimation service for large networks," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2273–2284, 2006.

[4] Bin Ren, Laurent Pueyo, Christine Chen, Élodie Choquet, John H. Debes, Gaspard Duchêne, François Ménard, and Marshall D. Perrin, "Using data imputation for signal separation in high-contrast imaging," *The Astrophysical Journal*, vol. 892, no. 2, pp. 74, mar 2020.

[5] Leo Taslaman and Björn Nilsson, "A Framework for Regularized Non-Negative Matrix Factorization, with Application to the Analysis of Gene Expression Data," *PLoS ONE*, vol. 7, no. 11, pp. e46331, Nov. 2012.

[6] Daniel Lee and H. Sebastian Seung, "Algorithms for non-negative matrix factorization," in *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp, Eds. 2000, vol. 13, MIT Press.

[7] "The mathworks. matlab statistics toolbox," Available at `http://www.mathworks.com/products/statistics`.

[8] Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari, "Csiszár's divergences for non-negative matrix factorization: Family of new algorithms," in *Independent Component Analysis and Blind Signal Separation*, Justinian Rosca, Deniz Erdogmus, José C. Príncipe, and Simon Haykin, Eds., Berlin, Heidelberg, 2006, pp. 32–39, Springer Berlin Heidelberg.

[9] Inderjit S. Dhillon and Suvrit Sra, "Generalized nonnegative matrix approximations with bregman divergences," in *Proceedings of the 18th International Conference on Neural Information Processing Systems*, Cambridge, MA, USA, 2005, NIPS'05, p. 283–290, MIT Press.

[10] Andreas Janecek, Stefan Grotthoff, and Wilfried Gansterer, "libnmf - a library for nonnegative matrix factorization.," *Computing and Informatics*, vol. 30, pp. 205–224, 01 2011.

[11] Sajad Fathi Hafshejani and Zahra Moaberfard, "Initialization for nonnegative matrix factorization: a comprehensive review," 2021.

[12] "Repository: Nmf optimizations," Available at `https://gitlab.inf.ethz.ch/COURSE-ASL/asl22/team30`.

[13] "Blas (basic linear algebra subprograms)," Available at `http://www.netlib.org/blas/`.

[14] "Openblas: An optimized blas library," Available at `https://www.openblas.net/`.

[15] "Intel intrinsics guide," Available at https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[16] "Microsoft docs: aligned_malloc and aligned_free," https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/aligned-malloc?view=msvc-170.

[17] "benchmark based on time-stamp counter," Available at https://github.com/mkurnosov/tscbench/blob/master/tsc_x86.c.

[18] "Intel vtune profiler," Available at https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.4bsoes.