University of California Berkeley

Computer Science Department

CS 267

Assignment 1: Optimizing Matrix Multiplication

Submitted by
Yucong He
Cristobal Pais
Alexander Wu
CS 267
February 13th
Spring 2018

Contents

Assignment 1: Optimizing Matrix Multiplication

Y.He, C.Pais, A.Wu

February 13th 2018

Abstract

An optimized code for a matrix multiplication algorithm in the context of efficient numerical algorithms implementations in C is developed for the NERSC's Cori supercomputer in a single threaded version. The manual optimization process is performed in a cumulative approach starting with the simplest programming enhancements and finishing with the most demanding implementations, such as exploiting vectorization and efficient memory usage. A series of parameters are tuned for each relevant implementation as well as experimenting with the options provided by the compiler when generating the binary files. The best implementation obtained includes loop ordering & unrolling, packing, vectorization and blocking techniques, reaching an average of 60.2 % with respect to the theoretical peak of the machine. Results of the experiments in a different machine are provided. Further improvements are discussed.

1 Introduction

Matrix multiplication is one of the most studied algorithms up to date. Several researchers have tried to obtain better and faster algorithms for performing this common operation, decreasing its theoretical complexity. The algorithm with the lowest complexity is a generalization of the Coppersmith–Winograd algorithm that has an asymptotic complexity of $O(n^{2.3728639})$, by François Le Gall.

Its simple but challenging optimization as well as being part of the kernel of an immense number of algorithms which have numerous applications in applied mathematics, physics, and engineering, creates the necessity of finding better and faster implementations for the current state-of-the-art hardware.

In order to study the complexity of developing and implementing an effective and efficient matrix multiplication algorithm in a High Performance Computer (HPC), both in a single-threaded version (part I, current report) and under a parallel multi-threaded context (part II), a C program containing a series of optimization techniques will be developed by the team, comparing it with the vendor's BLAS tuned library provided with the hardware. Concepts such as cache friendly code, blocking, packing, padding, loop unrolling, loop order optimization, memory alignment, etc., will be studied, described, and applied in order to obtain a well tested and optimized code.

The structure of the report is as follows: In section 2, the main optimization methodology is described as well as the member's contribution and hardware description. Section 3 describes the tested optimization approaches and their potential impact. In section 4, results are discussed for the different optimization techniques, as well as defining and describing the main algorithm developed. The algorithm is tested in a different machine, comparing its performance to the original platform. Finally, section 5 contains the conclusions and future work of the project.

2 Methodology

The main methodology of this project consists of performing an optimization of the original matrix multiplication code (in C) provided with the course material, analyzing the different programming techniques/tricks that can be implemented in order to obtain a better performance. The performance is measured in FLOPS and in a percentage of theoretical peak attained for the specific hardware implementation.

Initially, we analyze the "AS-IS" performance of the three original files provided: (1) simple naive implementation based on a classic three loop approach for performing the multiplications between the elements of both matrices A and B, (2) simple and non-tuned blocked version of the naive multiplication algorithm, and (3) the vendor's optimized BLAS implementation of matrix multiply (MKL). A representative subset of matrices $M \in \mathbf{R}^{n \times n}$ dimensions is used for the experiments where $n \in \{31, 32, 96, 97, 127, 128, 129, 191, 192, 229, 255, 256, 257, 319, 320, 321, 417, 479, 480, 511, 512, 639, 640, 767, 768, 769\}$.

Once the initial results are obtained, a series of incremental modifications will be performed in both the naive and blocked version algorithm (if pertinent), starting from the simplest ones - such as testing the loops order - and finishing with the most programming demanding approaches (e.g. AVX vectorization). Each different programming technique is applied individually to both versions, recording its performance and comparing it with the original implementations.

After finishing the first phase of the optimization, different subsets of the most promising tested techniques are implemented at the same time, analyzing the performance of the new algorithms. Based on the results, best implementations are selected for specific tuning steps. Depending on the techniques applied, a series of parameters are optimized for each approach, performing a series of experiments - mainly following a brute force rule - and keeping the best solutions achieved.

In addition, all reported implementations are tested in a daily (common-user) machine for comparison purposes, being able to understand the importance of developing specific algorithms implementations depending on the hardware characteristics. A series of plots are generated for each experiment in order to visualize the performance of the different implementations.

2.1 Contribution

During the development of the project, each member of the team develops its own experiments, sharing his results in a common Github repository for checking the current state of the implementations while re-using code from other members code. In addition, cross-testing of other members' code is performed. After performing a series of experiments, the best implementation is selected as the main submission file.

Based on the experience and preferences of the members of the group, the members' contributions and tasks can be summarized as follows:

- **Yucong He**
  - General optimization of the matrix multiplication algorithm: Loop unroll, Loop order, Padding, Blocking, Packing, and vectorization.
- **Cristobal Pais**
  - General optimization of the matrix multiplication algorithm: Loop unroll, Loop order, Padding, Blocking, Packing, and vectorization.
  - Tuning.
  - Editor and main writer of the current report.
- **Alexander Wu**
  - Setting up the GitHub repository.
  - Main Tester

2.2 Hardware & Software

The optimization of the matrix multiplication algorithm is developed for a specific hardware and runtime environment from The National Energy Research Scientific Computing Center (NERSC). In addition, for comparison purposes, same code is tested in a common-user machine (e.g. a simple laptop). All experiments, benchmarks, and performance results are implemented using the following hardware and software:

2.2.1 Hardware

1. **NERSC's Cori supercomputer**
   - Intel® Xeon™ Processor E5-2698 v3 ("Haswell") at 2.3 GHz
     - 2 sockets per chip (32 cores per node)
     - Supports SSE, SSE2, SSE3, SSE4, AVX, AVX2 (among others).
   - Memory
     - 64 KB 8-way set Level 1 cache (32KB instructions, 32 KB data)
     - 256 KB 8-way set Level 2 cache
     - 40 MB 20-way set Level 3 cache (shared per socket)
   - Software
     - SUSE Linux version 4.4.74-92.38-default
     - Built with PrgEnv-intel and PrgEnv-gnu/ gcc version 4.8.5
     - Default Compiler flags: -O2 -mavx
2. **Personal computer: Laptop**
   - Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz
     - 4 cores, 8 threads
     - Supports SSE, SSE2, SSE3, SSE4, AVX, AVX2 (among others).
   - Memory
     - 256 KB Level 1 cache
     - 256 KB Level 2 cache
     - 40 MB Level 3 cache (shared per socket)
   - Software
     - Ubuntu 16.04.2 LTS
     - gcc version 4.8.5
     - Default Compiler flags: -O2 -mavx

2.3 Assumptions

Based on the statement of the problem, our implementation of the matrix multiplication algorithm satisfies the following assumptions:

- All matrices in the operation are squared matrices $M \in \mathbf{R}^{n \times n}$.
- All elements/components of the matrices are double-precision floating point values (64 bit).
- The main operation of the algorithm consists of calculating $C = C + AB$ where A, B, and $C \in \mathbf{R}^{n \times n}$.
- The naive implementation of the matrix multiplication with a running complexity of $O(n^3)$ floating-point operations is used as the starting point for our optimizations. No lower runtime complexity algorithms are implemented (e.g. Strassen's algorithm $O(n^{2.8})$).

3 Optimization: tested approaches

In this section, we present a series of different optimization approaches tested for the matrix multiplication algorithm following the methodological scheme presented in section 2.

3.1 Loop order

Different loop orders reach different performance: strided accesses are a critical bottleneck in the naive implementation of the matrix multiplication algorithm where A and B are stored in column-major format. Clearly, large strides will result in an ineffective use of the existing cache lines, reducing the cache line and therefore, the performance of the whole algorithm.

As an initial step, we tested the six possible permutations for the indexes i, j, k in order to compare the performance of the naive and blocked implementations AS-IS by just changing the way the matrices are loaded into memory.

3.2 Loop Unrolling

In order to allow our code to exploit its potential parallelism, loops are unrolled by different factors, exposing the instructions to the compiler for performing multiple operations. This would be particularly useful in the inner loops of our algorithm when dealing with vectorized operations. The test set consists of power of two values up to 8 (larger values resulted in very poor performance in our final implementation).

3.3 Blocking & Cache misses

In order to improve the algorithm performance, a classic blocking approach is implemented. Thanks to this technique, it is possible to fit a series of chunks/blocks of the original matrices into the different levels of cache, restricting the computations to those values.

3.3.1 Blocking for L1 Cache

Based on the hardware specifications, the level 1 cache for the Cori supercomputer is 32 KB for instructions and 32 KB for data. In the original algorithm, the A matrix is read by row from left to right while B is read by column from top-to-bottom. However, both matrices are stored in a column-major approach, leading to poor performance due to the high cache miss rate obtained by the naive implementation. We will try to fit the block inside this fast memory level.

In order to deal with the "leftovers"(fringes) from matrices with odd dimensions - not evenly divided into sub-blocks - a specific inner routine will be developed, trying to take advantage of the vectorized operations as much as possible before applying a naive dot product implementation for the last elements to be computed. We expect that this implementation will reach better performance on larger matrices since the proportion of extra fringes to the number of evenly divided sub-blocks increases very slowly with the size of the matrix.

3.3.2 Blocking for L2 Cache

The size of this cache level is 256 KB. As a second level of fast memory, it would be useful to fit our data in this cache when large matrices are multiplied: they would not fit into the level 1 cache. Therefore, different blocking levels are tested in order to obtain the maximum performance.

3.4 Vectorization

Vectorization is one of the classic techniques used for improving the performance of numerical code. Thanks to this approach, we are able to process larger chunks of values at the same time ("in parallel") instead of processing them individually. These chunks are treated as vectors. In 2008, Intel introduced a new set of high-performance instructions called Advanced Vector Extensions (AVX), improving the existing SSE instructions performance.

AVX and AVX2 (Fussed multiply and add - FMA - is supported by AVX2) instructions can be accessed via special C functions called intrinsic functions, an assembly command that performs indivisible operations that (usually) reaches higher performance compared to other C functions/implementations. Each AVX vector contains up to 256 bits of data (e.g. 4 doubles or 8 floats). Therefore, we implement AVX intrinsics into our C code for loading, storing, adding, and multiplying vectors when dealing with matrices blocks.

The instructions included in our experiments are (see [4],[5]):

1. **AVX**
    - _mm_loadu_pd(): Loads two double-precision, floating-point values.
    - _mm256_loadu_pd(): Loads four double-precision, floating-point values.
    - _mm_load1_pd(): Loads a single double-precision floating-point value, copying to both elements.
    - _mm_storeu_pd(): Stores two double-precision, floating-point values.
    - _mm256_storeu_pd(): Stores four double-precision, floating-point values.
    - _mm256_broadcast_sd(): Loads and broadcasts scalar double-precision floating point values to a 256-bit destination operand.
2. **AVX2**
    - _mm_fmadd_pd(): Multiply-adds packed double-precision floating-point values using three float64 vectors.

3.5 Inner kernel blocks dimension

Different blocks operations are tested in order to find the best block dimension in terms of performance, seeking the optimal usage of the faster memory levels (registers, L1 and L2 cache). Thus, kernels performing the following operations are tested: (1) 4x4, (2) 8x4, (3) 8x3, (4) 8x2, (5) 4x2, (6) 2x2. Depending on the dimension of the matrices that are being multiplied, different kernels are called in order to obtain the best performance while dealing with the potential fringes ("leftovers") of matrices where $n \mod 2 \neq 0$, $n \mod 4 \neq 0$, or $n \mod 8 \neq 0$.

3.6 Padding

As a different approach for dealing with odd-dimension matrices ($n \mod 2 \neq 0$) additional rows/columns filled with zeros are added to the original matrices avoiding the extra "dealing with fringes" step inside the algorithm. Therefore, the trade-off between the computations needed for the generation of a new copy of each matrix (or sub-block) and the potential improvement in a vectorized inner kernel performance is analyzed via a series of experiments.

3.7 Packing, copying & memory alignment

Transposition of the relevant matrices/sub-blocks as well as creating copies of the relevant pieces of data for each step are tested in order to exploit the temporal and spatial locality of the different memory levels. As an example, without these transformations we have that the main dot product operation in the Naive

implementation is performed in such a way that the A matrix is accessed in intervals of n, triggering a series of cache misses (loading a new line into the level 1 cache) every subsequent access when dealing with sufficiently large matrices. Depending on the final implementation of the algorithm, same analysis can be performed for B and C, giving us a variety of possibilities to explore.

In terms of copy optimization, we test the performance of our implementation when the whole (or specific blocks of) matrices are duplicated in the memory, trying to exploit cache locality within the inner operations.

3.8 Parameter tuning

Finally, in order to tune the best implementation obtained, a brute force approach is developed for finding the optimal set of parameters such that the performance is maximized for each particular code.

The main parameters optimized in this project are:

- Block Size: Depending on the implementation, different Blocking sizes parameters should be selected in order to minimize the number of cache misses per iteration.
- Unrolling factor: when performing the loop unrolling, different levels are tested in order to attain the maximum performance of the code.
- Compiler flags: Different flags lead to different performance of the tested implementations, arising as one important element to take into account when generating the final binary files. Several options are tested based on the experience of the members of the team, without searching the entire (too big) space of combinations. We focused our analysis in the classic -Ox flags alongside with specific optimizations such as loop unrolling, use of AVX technology, and general code optimization.

3.9 Other techniques

Extra tricks such as function inlining, decoration commands like register, align, etc. are tested. These small changes in the code are kept only when there exists a significant difference in the performance of the code (at least 1%), otherwise, they are discarded for simplicity, keeping the code as clean as possible.

4 Results & Discussion

In this section, we present the main results obtained from a series of experiments including the optimization techniques introduced in section 3.

4.1 Optimization results

Following the methodology introduced in section 2, we develop a series of experiments based on the mentioned optimization techniques. All preliminary tests are performed using the compiler flags included in the original makefile provided by the instructors.

4.1.1 Loop Order

In Figure 1 we can see the different performance achieved by the simple AS-IS naive algorithm when the order of the three loops is exchanged. Based on the results, it is clear that the jki order is the one with the best performance. This is consistent with our expectations since all three matrices A, B, and C are stored in a column major order, the j index becomes critical when accessing the values of the A and C matrices in the inner loop of the algorithm since we are performing a series of "jumps" in the cache instead of exploiting the spatial locality when loading a certain value from the cache. This can easily seen in the results, where both implementations changing the j index in the most inner loop obtains the worse performances across the experiments (indicated in the legend of the plot), around 2.2%.

On the other hand, loops orderings with i as the main index of the most inner loop reach the best performance among these preliminary experiments, reaching performances around 11%-13.3% - a significant boost without actually coding anything. Therefore, we are able to notice the importance of taking into account the way we are addressing and manipulating the data in order to optimize its access, increasing the number of cache hits while decreasing the cache miss.

Hence, we keep in mind this simple but powerful strategy for our final implementation.

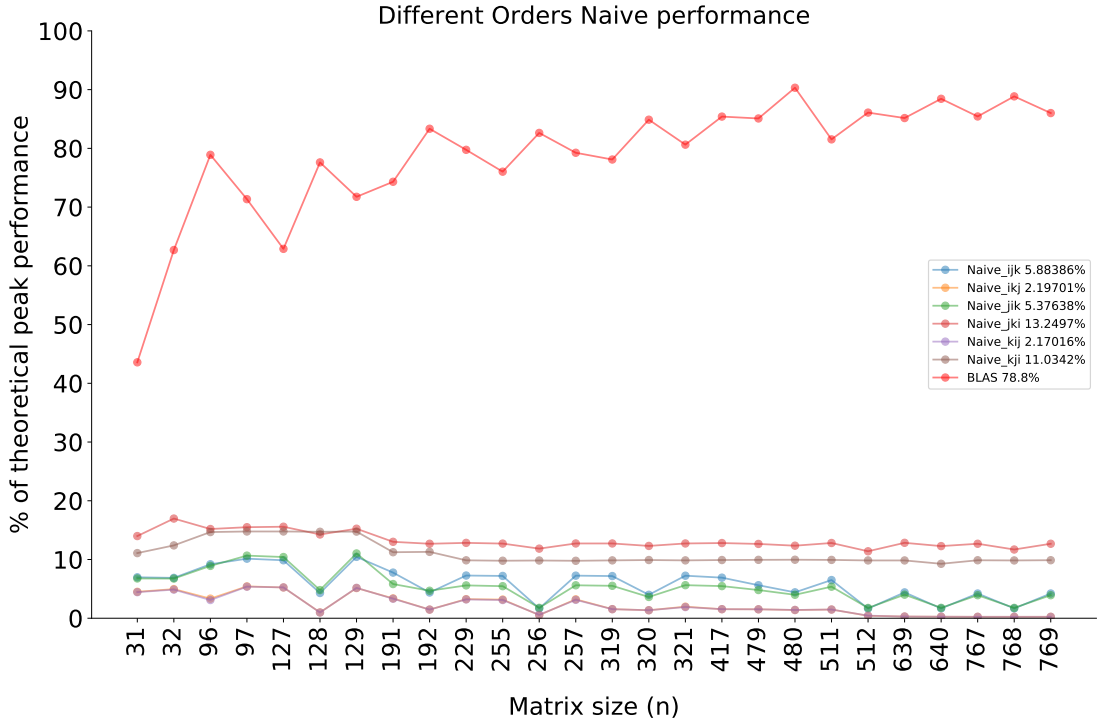

Figure 1: Different loop orders performance using the AS-IS Naive version.

4.1.2 Blocking

As a second step, we apply the simple AS-IS blocked algorithm - without optimizing the block size parameter - included with the problem statement, testing all the six possible loops orderings as before. From the results in Figure 2 we can see that again we obtain a "free" performance boost when the best loop orderings detected in the naive implementation are mixed with the simplest blocking technique, reaching theoretical performance levels around 17%-18% of the possible peak.

Based on these results, it is clear that our final implementation will exploit the blocking algorithm as well as being aware of the cache misses and hits depending on how the data is loaded and accessed: spatial and temporal locality must be exploited to obtain good performance in our code.

The explanation behind these new results follows the logic indicated in section 3: working with sub-block matrices allows our code to exploit the temporal and spatial locality from the cache lines, allowing us to significantly impact the performance of the algorithm. This technique also keeps the performance levels more balanced among the different experiment sizes n (without extreme peaks).

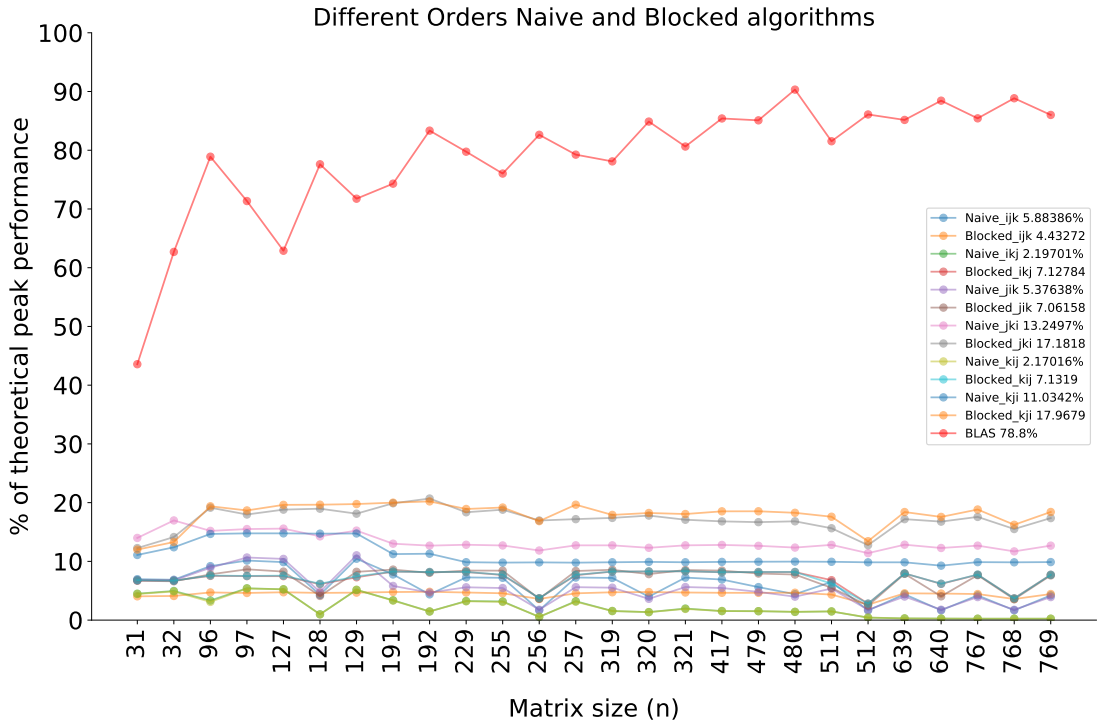

Figure 2: Different loop orders performance using the Naive Blocked version.

4.1.3 Vectorization & Inner kernel blocks dimension

As a first approach for vectorizing the additions and multiplications needed for updating the value of the C matrix, we implement AVX intrinsics following the naive three loops approach for only even matrices that can be divided by 4 ($n \mod 4 = 0$). This is for simplicity of the first experiments, including a 4x4 (loading vectors with 4 double precision floating point numbers each) inner kernel for updating the C matrix.

From the results obtained, we can see that the performance is slightly increased across all loop orders. Initially, we expected a greater performance boost since we are performing 4 operations at one instead of a serial approach with one operation, however, we notice that this first naive implementation is not really exploiting the vectorized operations since we still have a very high overhead in the most inner loop (where computations are performed). After analyzing the performance of the code, we proceed to test different loop unroll schemes in order to obtain the maximum performance of the current AVX implementation, giving us a starting point for merging it with the blocking strategy.

Different kernel dimensions are tested in order to find the optimal one in terms of performance for our optimized final version.

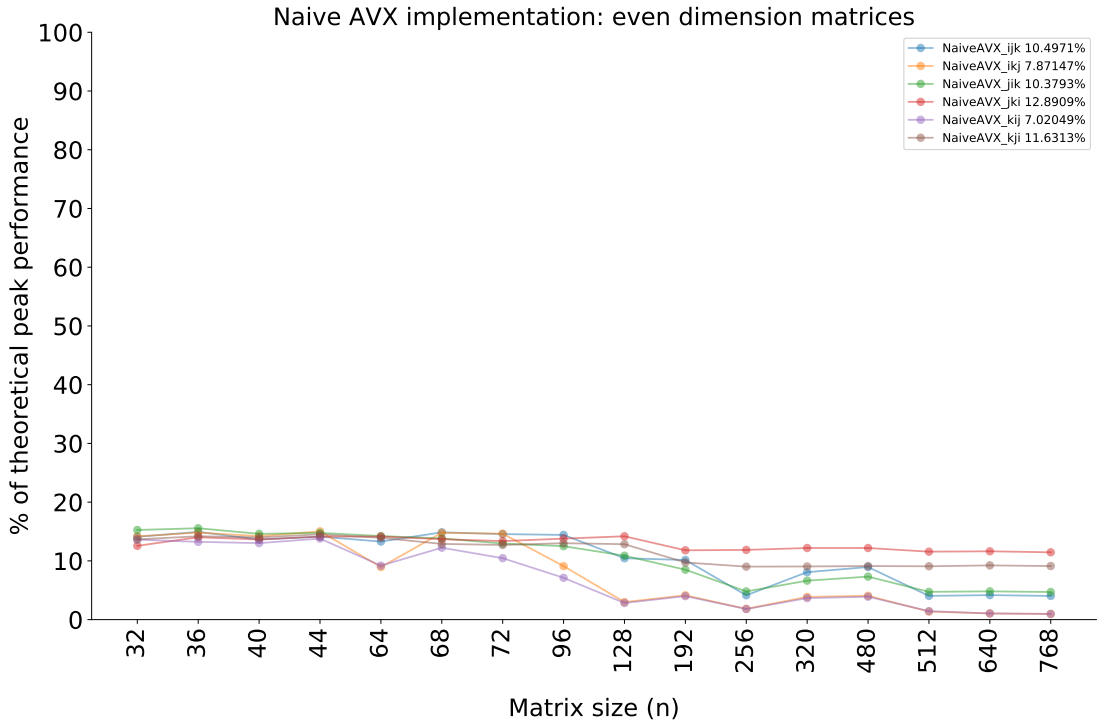

Figure 3: Different loop orders performance using a Naive AVX version.

4.1.4 Loop Unroll

Initially, a simple loop unrolling scheme where the inner loop is unrolled by a factor of four is tested for all the six different loops orders. From the results in Figure 4 we can see a significant performance boost for small even matrices. The main reason behind this pattern consists of the fact that we save some branches and arithmetic operations due to the increased step size of the inner loop, obtaining an efficient performance of the 4x4 vectorized kernel. On the other hand, since no blocking is performed, the performance is clearly degraded as the matrix size becomes larger due to number cache misses (as we already notice in previous results).

Different combinations of loop unroll schemes for all indexes using power of two values (2, 4, 8, 16, and 32) are tested. No improvements are achieved with the current naive implementation, however, same schemes will be tested when mixing this version with other optimization techniques, leading us to obtain the optimal parameters for our final version implementation.

4.1.5 Padding

In order to be able to extend the current AVX vectorized implementation to handle the odd dimension matrices, additional columns/rows containing zeros are added to original matrices A and B in order to be able to use the same 4x4 kernel like in the even case. In addition, a simple blocking with a pre-defined block size of 32 (one of the best values found for the AS-IS blocked implementation) is used.

Based on the results, we can see in Figure 5 how the performance is significantly increased for even matrices reaching an average around 40% of the theoretical performance peak in Cori for matrices with $n \in \{32, 96, 192, 320, 480\}$. Therefore, we have a series of peaks in the performance of the algorithm associated with the even matrices and low performance with the odd matrices.

The explanation behind this poor performance is clear: we are creating new padded copies of A and B at the beginning of the algorithm (the full matrices) and then extracting the original solution for populating

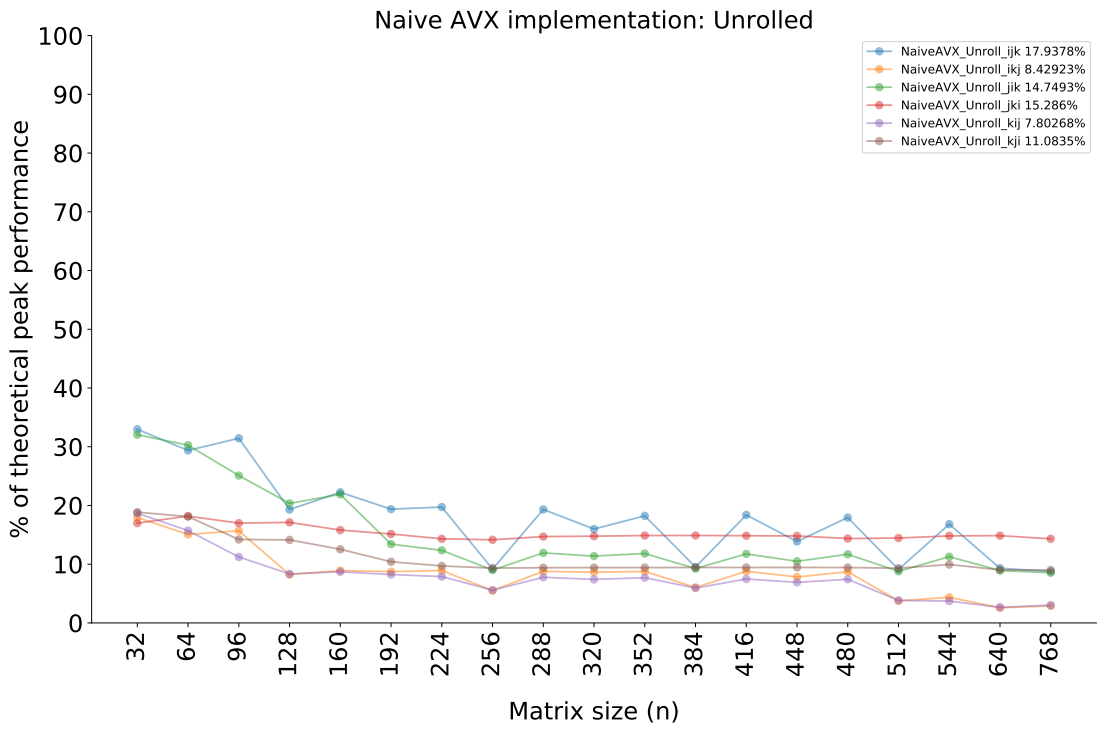

Figure 4: Different loop orders performance using a Naive AVX version and loop unrolling.

the resulting C matrix and thus, the code requires and spent a significant amount of time just creating these auxiliary matrices and extracting the solution, leading to poor performance in these instances besides the fact that we are exploiting the vectorized 4x4 efficient unrolled kernel.

In addition, different performance is achieved depending on the unrolling factors implemented reaching an average of 24% of performance with respect to the theoretical peak in its best version.

At this point of the optimization process, it is clear that a vectorized kernel with an appropriate unroll scheme, as well as a blocking approach are mandatory for obtaining significant performance boosts of the code. On the other hand, although the padding technique simplifies the treatment of odd matrices allowing us to avoid a more complex implementation of a sub-routine for dealing with the remainder fringes of these matrices, it is clear that in order to boost the performance of these instances, a specific and efficient way to deal with these cases is needed for our final implementation.

4.1.6 Packing

Following the ideas from section 3.7, matrix transposition operations as well as copying the sub-blocks/entire matrix before performing the computations in order to exploit the memory hierarchy as well as temporal/spatial locality are tested.

As a first approach, a simple matrix transposition scheme is applied to the simplest Naive implementation in order to check the preliminary impact of this technique in the performance of the algorithm when the access to the A matrix is performed in a row-major approach. As expected, we obtain a significant performance boost by this simple implementation, reaching an average peak of 15% with respect to Cori's maximum theoretical capacity.

This performance is achieved by the original i, j, k loop order, corroborating our initial thought: transposing the A matrix leads the code to access it in a row-major fashion ($A[k + i * n] * B[k + j * n]$ instead of $A[i + k * n] * B[k + j * n]$) in the most inner loop (k), leading to a significant decrease in the number of cache misses. Similar pattern can be seen with the j, i, k order. On the other hand, we can see that the performance of the other loop orders having the i index in the most inner loop (say j, k, i and k, j, i) is very poor. The explanation is exactly the same: since the A matrix has been transposed, changing the i index

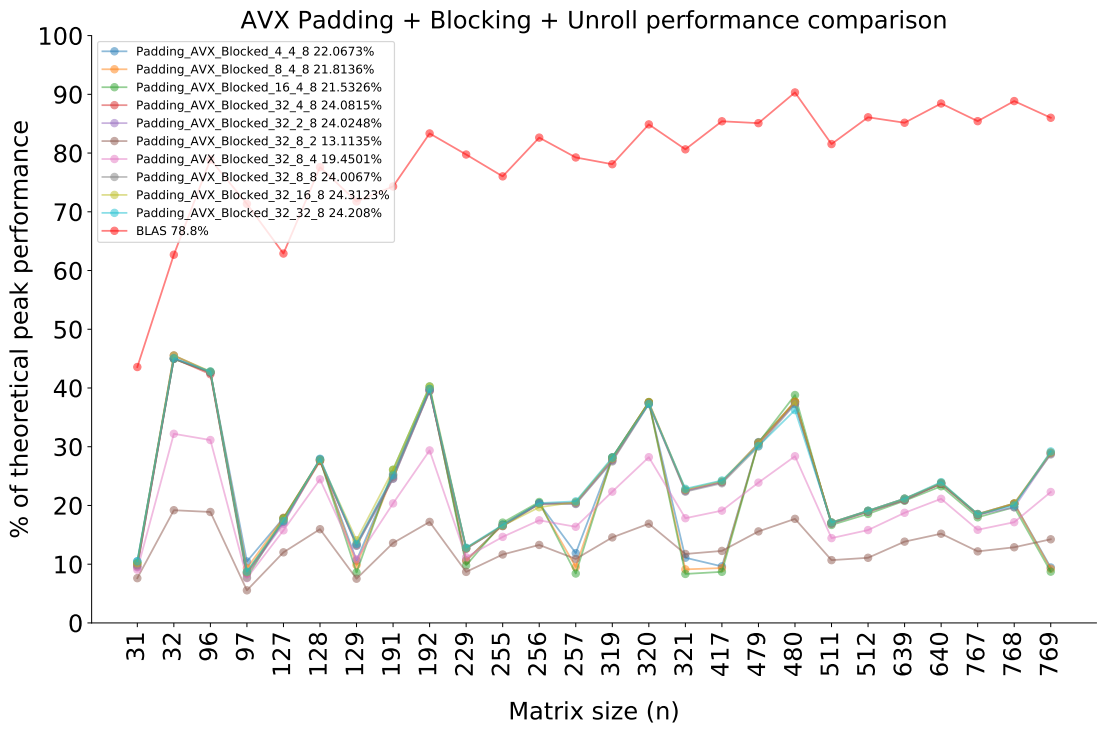

Figure 5: Different loop orders performance using an AVX version, loop unrolling, and padding.

will produce the cache misses the we originally experience in the AS-IS naive algorithm.

Therefore, two main different packing schemes are tested in order to develop our final version:

1. Transform the A (or B depending on the implementation) matrix into row-major order, pre-allocating the new matrix memory. These allows us to perform a matrix multiplication using unidirectional loads of both matrices for performing the computations. Thus, we take advantage of spatial locality at all cache levels, optimizing the usage of cache space.

   However, performing the transposition of the entire matrix doubles the total number of memory accesses, loading and storing an entire new matrix as well as being unable to performs operations to hide latency. In addition, an efficient packing sub-routine must be programmed since the same problem of strided memory accesses can happen if we use a simple naive implementation.

2. Instead of packing the entire matrix A (or B, C depending on the approach), a sub-block re-packing approach can be applied in order to take even more advantage of the space locality and temporality when performing the computations inside the blocking loop. After a series of tests, this implementation arises as the optimal one for our final version, as can be seen in the next section.

4.2 Optimized approach

Using these results and our understanding of the Cori's hardware architecture, we are ready to merge the most relevant techniques into an efficient and optimized matrix multiplication algorithm, taking advantage of the hardware's memory hierarchy as well as the vectorized intrinsics functions. A series of intermediate tests are performed in order to optimize the relevant parameters of the final version. The most relevant results are included.

4.2.1 Description

The final optimized implementation is based on the simple AS-IS blocking algorithm, using it as the basic structure of our code. As can be seen in Algorithm 1 the main difference is that we eliminate the last loop for k in the main function, since we are dealing with it inside the *do_block*() function.

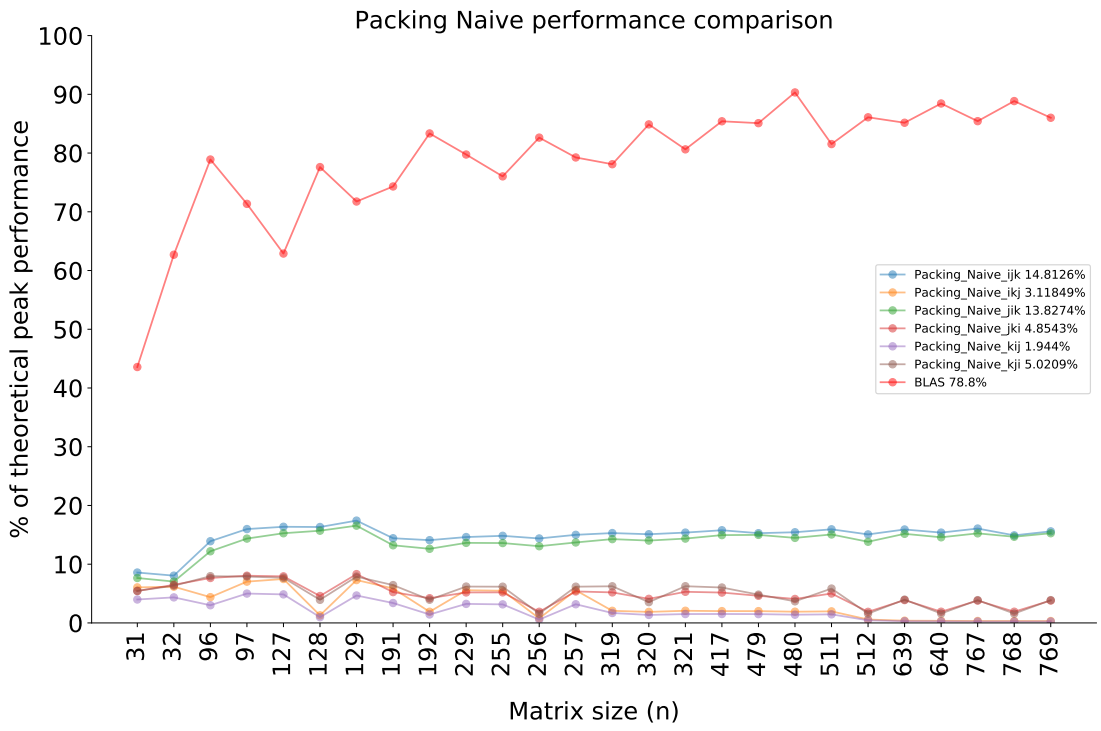

Figure 6: Naive algorithm performance for different loop orders using packing (A in these experiments).

As an extra parameter, a boolean flag indicating if this is the first call to the *do_block*() function from inside the inner loop i is passed as an argument: this will be used inside the function for performing the packing procedure of the B matrix.

In our best performance implementation, the B matrix is duplicated and transposed (re-packed) at the beginning of the inner blocking algorithm, just before starting any computation of the actual algorithm. Alongside this, the relevant components of the A matrix are duplicated and packed to the relevant block size depending on the vectorized inner kernel to be called, hence, when blocking for the L1 cache the A matrix is copied by sub-blocks instead of the full matrix, buffering the parts of the A matrix that are part of the working sub-block.

AVX/AVX2 multiplications and additions operate on three vectors containing at most 4 double precision float numbers at the same time. Thus, we can obtain speed-ups up to four times - in theory - due to the amount of data that can be processed in parallel. Using the AVX intrinsics introduced in section 3.7, we have that the processor can perform a multiplication and addition of vectors at the same time thanks to the FMA unit (AVX2 intrinsic).

Different kernels dimensions where tested as indicated in section 3.5. After a series of experiments, we obtained the following efficient scheme:

1. If possible, operations are performed in blocks of 8x4 by calling the corresponding kernel. This kernel obtained the best performance across all the experiments when the optimal unrolling scheme is implemented. In this case, we have $8 \times 4 = 32$ double precision floating point numbers, each one with a size of 8 bytes, reaching a total of $8 \times 32 = 256$ bytes in memory.

2. If there are remainder fringes after using this kernel that can be processed by the 4x4 kernel, the A matrix is re-packed and the kernel is called for performing the computations. This allows us to improve the performance when dealing with small matrices, e.g. when $n = 31$ since there are 7 rows left after using the 8x4 kernel. Hence, we can optimize these extra sub-blocks using this smaller kernel implementation instead of calling the $SolveFringes()$ sub-routine based on a 2x2 kernel (details below).

3. In the case that the sub-block dimensions cannot be processed by the 8x4 kernel, an 8x3 kernel implementation is called if the dimensions are pertinent. The logic behind this implementation consists

of handling those cases where the remainder columns do not match the 8x4 core by one column, processing the maximum number of elements at once via the smaller kernel.

4. Finally, if we have fringes that do not match any of the previous kernels, the $SolveFringes()$ function is called. This function deals with the edge cases using a 2x2 vectorized kernel to solve as many sub-blocks as possible in the remaining matrix. After this call, we check for possible remaining rows and columns where a simple unrolled dot product is performed for filling these entries. Thanks to this approach, we are able to obtain a significant performance boost for the odd matrices, in a very simple and easy-to-follow code .

These operations are complemented and optimized using the loop unrolling technique in two levels. Different loop unrolling factors were tested in order to maximize the total number of components from A and B processed at the same time. After a brute search approach, we selected an unrolling scheme of 4 for the outer blocking loop corresponding to the rows and 8 for the inner blocking columns loop obtaining an important performance boost since both loops (unrolled by the compiler) are executed fewer times and thus, there are less branches in the final assembly code generated leading to less overhead.

4.2.2 Pseudocode

In order to complete the description of our algorithm, we present the pseudo-code of the two main components of the implementation. In the Algorithm 1 it can be seen that the main function called by the benchmark script consists of simple two-loops blocking for the rows and columns of the matrices to be solved as blocks. Once inside the inner loop, the *do_block*() subroutine is called from the main.

All relevant operations for the matrix multiplication algorithm are performed inside the *do_block*() function. Since the main kernel of the implementation consists of a 8x4 vectorized matrix multiplication, the unroll factors are selected accordingly for both loops (i and j). Matrices B and A are packed if needed, keeping the dimensions consistent for the kernel operations.

Algorithm 1 Version 1 Pseudo-code Main

```
1: procedure SQUARE_DGEMM(INT N, DOUBLE* A, DOUBLE* B, DOUBLE* C)
2:     BLOCKSIZE ← BS, int M, N ← 0;                          //Initialize variables
3:     for j = 0; j < n; j+ = BLOCKSIZE do                    //Blocking
4:         N = min(BLOCKSIZE, n - j);                         //Be aware of edges
5:         for i = 0; i < n; i+ = BLOCKSIZE do
6:             M = min(BLOCKSIZE, n - i);
7:             do_block(i == 0, n, M, n, N, A + i + j * n, B + j, C + i)   //Call blocking procedure
```

Depending on the dimensions of the matrices to be multiplied (A and B), several operations are computed using the AVX/AVX2 intrinsics for simple and efficient vectorization of the code ($kernel8x4()$). Once the operations are finished, the presence of extra fringes is tested. Based on the dimension of the remaining elements, a 4x4 or an 8x3 vectorized kernel is called.

Finally, if there are still fringes that do not fit in one of these categories, an auxiliary function call $SolveFringes()$ performs the multiplication of these elements, based on a 2x2 vectorized kernel as well as a simple naive dot point algorithm for the last elements. Thanks to this implementations, we are able to perform more simultaneous operations (SIMD), obtaining better performance of the overall algorithm. As mentioned in the description, several kernels were tested (and many more can be implemented) and the ones with the best performance across all the experiments are included in the current project.

4.2.3 Results

Based on the described pseudo-code, our final implementation is tested in Cori for different block sizes. After a tuning procedure based on a simple brute force algorithm with $BlockSize \in \{8, 16, 32, 64, 96, 128, 160, 192, 256\}$

Algorithm 2 Version 1 Pseudo-code do_block

```
1: procedure DO_BLOCK(int toPackB, int lda, int M, int n, int K, double *A, double *B, double *C)
2:     int U1 ← 4, U2 ← 8                                          //Unroll factors
3:     int j_lim ← (n/U1) * U1, int i_lim ← (M/U2) * U2            //Unroll limits U1 = 4, U2 = 8 default
4:     for (j = 0; j < j_lim; j+ = U1) do                          //First Unroll (4)
5:         if (toPackB) then
6:             toRow(B)                                            //Pack B at the beginning
7:         for (i = 0; i < i_lim; i+ = U2) do                      //Second unroll (8)
8:             if (j == 0) then
9:                 Pack8xK(A, K)                                   //Pack A to the correct dimension
10:            kernel8x4(A, B, C, K, n, i_lim, j_lim)              //Perform a 8x4 vectorized mmult
11:    if (M mod 8 ≠ 0 && n mod 4 == 0) then                       //Potential fringes
12:        if (M mod 4 = 0) then
13:            Pack4xK(A, K)                                       //Pack A to the correct dimension
14:            kernel4x4(A, B, C, K, n, i_lim, j_lim)              //Perform a 4x4 vectorized mmult
15:        else SolveFringes(A, B, C)
16:    if (M mod 8 ≠ 0 && n mod 4 ≠ 0) then
17:        if (n mod 4 = 3) then
18:            Pack8xK(A, K)                                       //Pack A to the correct dimension
19:            kernel8x3(A, B, C, K, n, i_lim, j_lim)              //Perform a 8x3 vectorized mmult
20:        else SolveFringes(A, B, C, K, M, n)
```

we are able to obtain a significant improvement in the performance of the matrix multiplication algorithm using a $Blocksize = 128$ (see Appendix), mainly due to the smart use of the memory's architecture of Cori and the vectorized and efficient kernels programmed. In Figure 7 we can appreciate that our optimized algorithm is able to get a 60% of the theoretical performance peak in Cori, almost three times better than the Padding implementation obtained before merging all the useful techniques studied in section 4.1 and only an 18.8% below the optimized BLAS library (our ultimate performance goal).

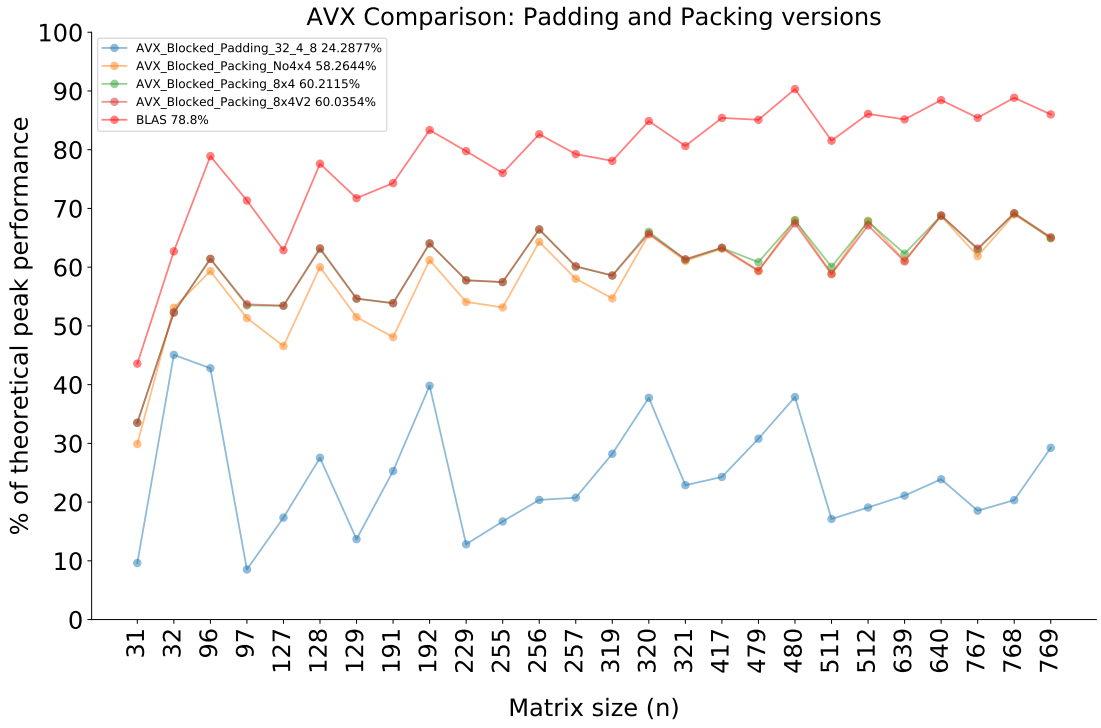

Figure 7: Performance comparison for the best versions obtained.

In addition, we can observe the usefulness of implementing the extra 4x4 kernel computation in the case that the remainder fringes can be processed by this auxiliary function. This improvement is clear in small and odd matrices ($n \leq 319$) where we can clearly see the difference in the performance peaks, gaining an extra 2% of performance by just a simple addition of code.

Interesting is to note that the performance pattern is very similar to the one obtained by the BLAS implementation: several ideas that we are applying in our code are part of these optimized libraries as well as from [1], [3]. As expected, better performance is obtained in evenly divided matrices since no fringes are needed to be processed via the inner 2x2 kernel and the simple linear dot product (non-vectorized operation), allowing our code to exploit the vectorization as its best efficiency.

For completeness, further improvements that can be included (not tested due to time constraints) to the current version of the code are:

1. Adding extra kernels for processing more specific cases depending on the matrix dimensions in order to take advantage of vectorization as much as possible.
2. Second blocking loop for fitting the data into the level 2 cache.
3. Replacement of C code by assembly code, giving us more opportunities to optimize the general structure of the code and memory allocation.
4. Exhaustive testing of different compiler flags: Only a decent (less than 10) combinations were tested based on the experience of the group and the compiler's website [2]. The optimal one found consists of the following options: *-O2 -funroll-loops -msse -msse2 -msse3 -mavx -march=core-avx2.*

For completeness, we include the performance plot in Figure 10 of our implementation for all the different sizes inside the benchmark file, in comparison to the vendor's tuned BLAS. As can be seen, the performance of our algorithm is very stable, obtaining almost the same performance as with the representative matrix set (62% of the theoretical performance peak).

4.3 Machines comparison

Using the hardware architectures described in section 2.2.1, we compare the performance of our matrix multiplication code in a different computer obtaining the results summarized in Figure 8.

Based on these preliminary results, it is interesting to notice that the Cori optimized implementation obtains better results in the Laptop with an I74700MQ processor (details in section 4.41). In order to interpret this pattern we need to understand that besides the fact that both hardware are "similar enough" - explaining the somewhat similar performance - the laptop's processor is using turbo mode by default, reaching a maximum frequency of 3,40 GHz in contrast to the basic frequency indicated by the manufacturer (2,4 GHz). Hence, we need to recalculate the theoretical performance peak in order to perform a fair comparison for both machines.

As seen in Figure 11, the performance of the code in the personal computer, as expected, is significantly less than the one obtained in Cori when we take into account the turbo boost bias. This is consistent with the theory since the best performance code has been specifically optimized for Cori's hardware & software: different cache levels sizes (memory hierarchy), processor, compilers, OS, etc. Based on the lectures notes and class, this is one of the known drawbacks and limitations from the code's optimization techniques that exploit the hardware architecture for gaining more performance, impacting the portability of the code to other machines.

Hence, we can see a performance drop from around 62% in Cori's supercomputer to a 43.7% in our laptop. At this point, the explanation behind this pattern is clear: the code is optimized for Cori's hardware, preventing our computer to take advantage of the optimized code at its 100%, mainly leading to memory alignment issues when dealing with different cache size lines as well as other memory components, all elements that have a significant impact in the performance of the code.

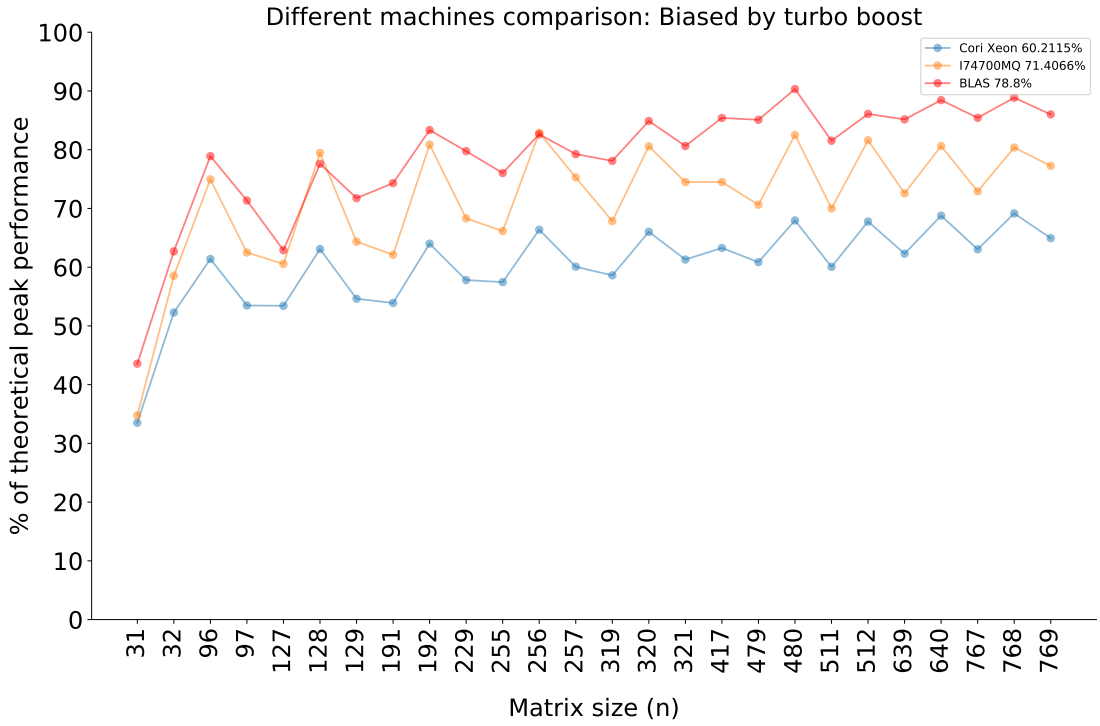

Figure 8: Performance comparison for the best versions obtained in different machines, turbo boost biased.

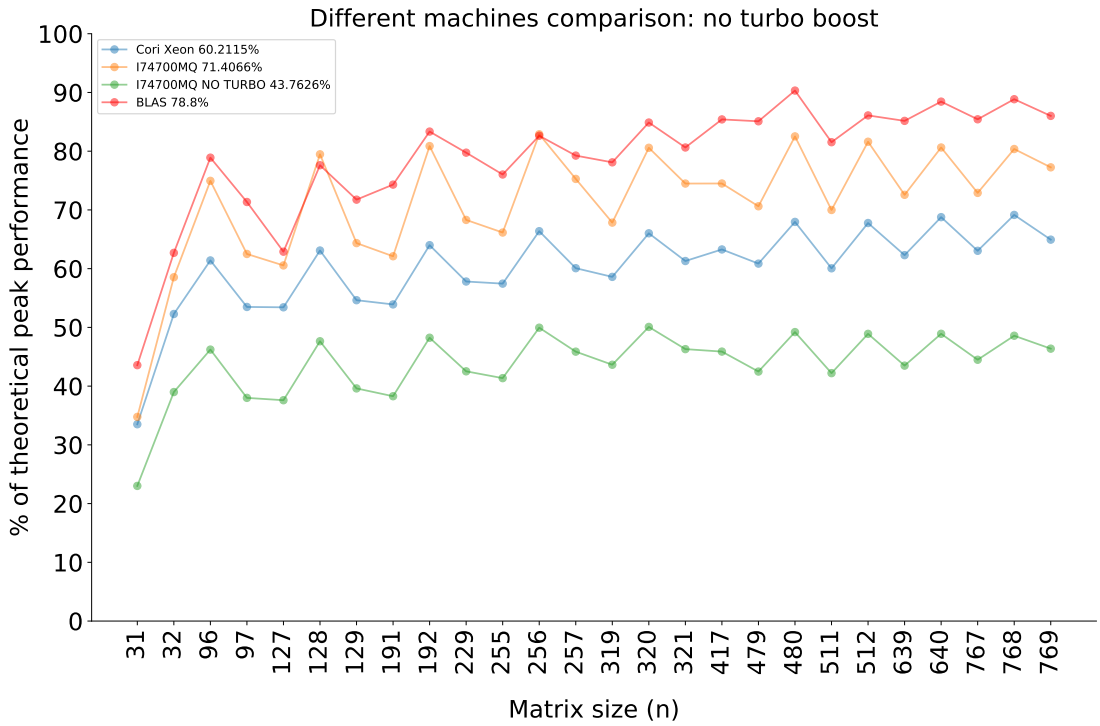

Figure 9: Performance comparison for the best versions obtained in different machines (no turbo bias).

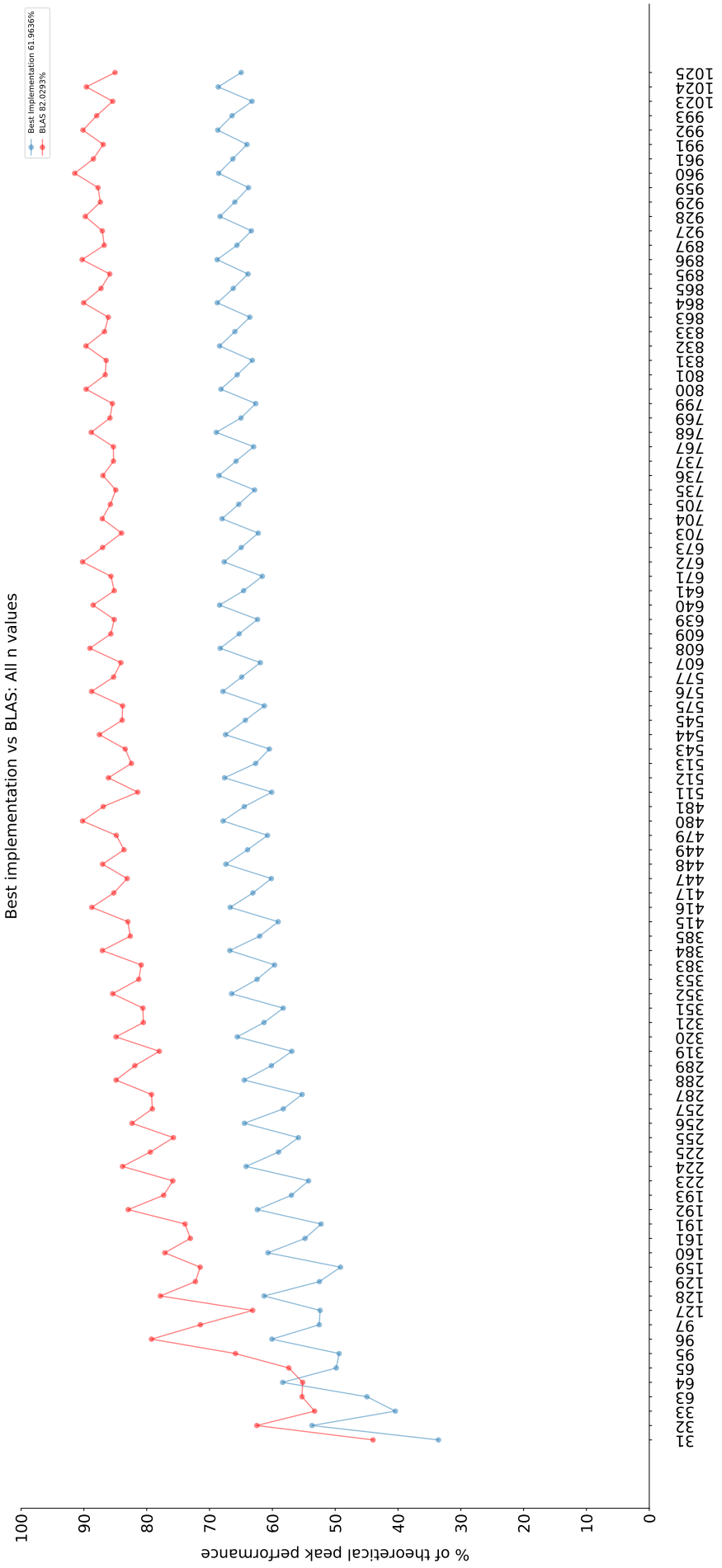

Figure 10: Performance comparison for the best version vs the BLAS library for all n values.

5 Conclusions & Future Work

1. A successful implementation of the dgemm squared double precision floating point matrix multiplication algorithm is implemented, reaching an average performance around 60% of the theoretical peak of the machine's hardware architecture. This is around 20% less than the vendor's tuned BLAS library, indicating that the current implementation has a good potential for improving its performance by a series of extra optimizations steps/tricks inside its code. More experiments and approaches should be tested in the future in order to achieve BLAS's performance.

2. The importance of developing cache-friendly programs has been studied and learned in order to be able to obtain significant improvements in our project. The relevance of knowing and understanding the underlying hardware architecture arises as one of the critical points when optimizing code for a particular machine.

   Alongside this, several new optimization techniques - not previously known by the members of this project - were applied in a successful way, understanding how hard is to actually obtain an optimized code for algorithms as simple as the square matrix multiplication.

3. Additional techniques like "peeling" could be applied to the current version of the code in order to improve the performance of odd-dimension matrices, allowing the algorithm to work with perfect tiles (no fringes in the main operations), exploiting the optimized (vectorized) matrix multiplication kernels.

4. Since one of the personal computer used for comparison purposes was very similar - in terms of hardware - to the one presented by Cori (manufacturer, cache levels, supported technologies), a similar performance pattern/level is achieved. In contrast, experiments in other computers having different hardware configurations reached worse performance than the one obtained in Cori.

5. As an extension of the current project, cache-oblivious algorithms can be implemented including some of the techniques that were discussed and applied in our final version. The main advantage of these type of algorithms lies on the fact that its portability can be far better than the cache-aware approach, since we are optimizing the code for a particular machine's architecture (like Cori), potentially leading to very poor results in different machines.

6. As a second part of the current project, a parallel implementation of the matrix multiplication algorithm using OpenMP is developed, using the current state of the project as a starting point. Main experiments will be performed using the 32 cores available in Cori Phase I.

6 References

1. Chellappa, S., Franchetti, F., & Püschel, M. (2007, July). How to write fast numerical code: A small introduction. In International Summer School on Generative and Transformational Techniques in Software Engineering (pp. 196-259). Springer, Berlin, Heidelberg.
2. GCC GNU compiler flags reference, https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86_002d64-Options.html
3. Goto, K., & Geijn, R. A. (2008). Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS), 34(3), 12.
4. Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
5. Lomont, C. (2011). Introduction to intel advanced vector extensions. Intel White Paper, 1-21.
6. Lecture notes and slides from course CS267, Computer Science Department, University of California Berkeley.

7 Appendix

7.1 Best implementation: Block Size comparison

For completeness, we include a subset of the different Block sizes tested for the best-optimized version of the algorithm. Values above 64 tend to obtain very similar performance while 32 obtains the worst performance, mainly due to the poor use of the memory hierarchy structure in Cori when dealing with larger matrices ($n > 128$).

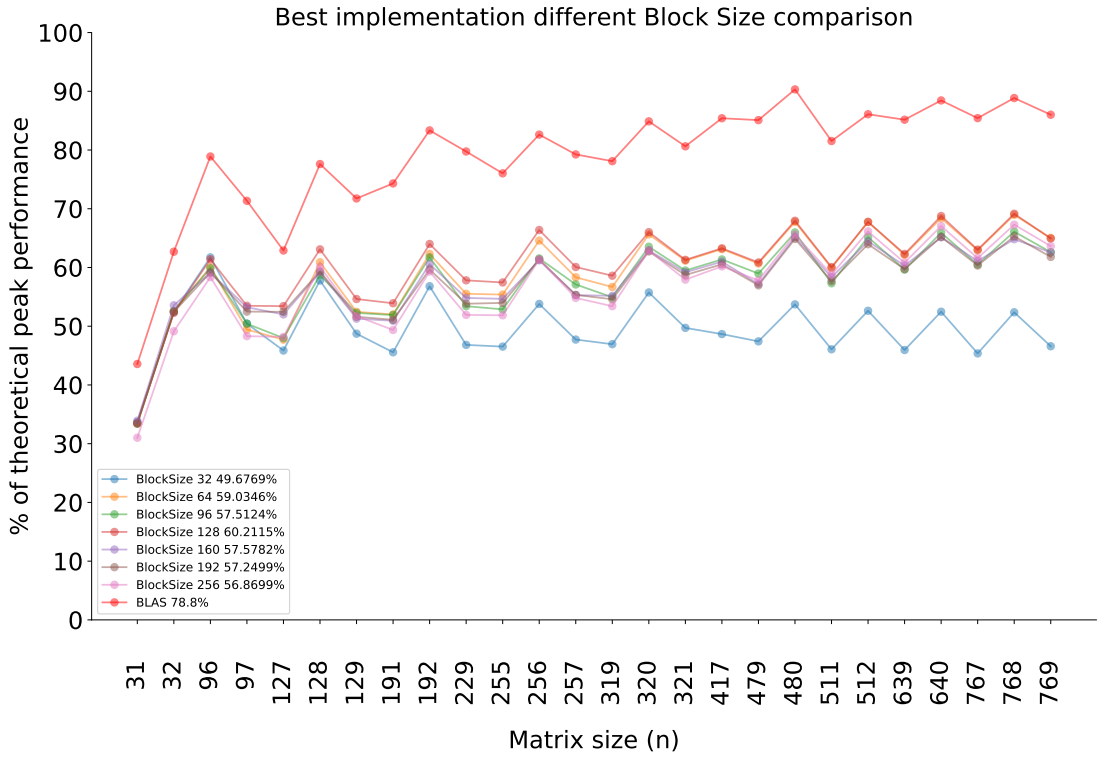

Figure 11: Performance comparison for different block sizes for the best implementation.