University of California Berkeley

Computer Science Department

CS 267

Assignment 1.2: Optimizing Matrix Multiplication using OpenMP

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

Contents

Assignment 1.2: Optimizing Matrix Multiplication using OpenMP

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 multi-threaded version using OpenMP with 32 threads. The manual optimization process is performed in a cumulative approach based on a previously optimized single threaded version, 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, padding, vectorization, and blocking techniques, reaching an average of 18.62 % 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) and under a parallel multi-threaded context (part II, current report), 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

As in the first part of this assignment, 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 as well as our single-threaded optimized version from part 1: (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, (3) the vendor's optimized BLAS implementation of matrix multiply (MKL), and (4) the vectorized multi-kernel single threaded optimized version. A representative subset of matrices $M \in \mathbf{R}^{n \times n}$ dimensions is used for the experiments where $n \in \{31, 32, 96, 97, 127, 128, 129, 191, 192, 229, 255, 256, 257, 319, 320, 321, 417, 479, 480, 511, 512, 639, 640, 767, 768, 769\}$. In addition, since the variance of the results is higher than in the single-threaded version with respect to the size of the matrices tested, same experiments are repeated with the full evaluation set included inside the benchmark.c file.

Once the initial results are obtained, a series of incremental modifications will be performed in both the blocked and optimized single threaded version (if pertinent), starting from the simplest ones - such as modifying the scheduler's rule for the loops parallelization - and finishing with the most programming demanding approaches (e.g. avoid false sharing, memory alignments, etc.). 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 the 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**
  - Algorithm pseudocode.
- **Cristobal Pais**
  - General optimization of the Single thread matrix multiplication algorithm: Loop unroll, Loop order, Padding, Blocking, Packing, and vectorization.
  - Tuning and experiments for the original parallel implementations.
  - Editor and main writer of the current report.
- **Alexander Wu**
  - Main programmer of the the OpenMP (multithread) implementation approach.
  - Setting up the GitHub repository.

- Tuning
- Description of the algorithm.

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 parallel matrix multiply. We started out with our optimized single-thread routine in part 1, and thus, all previous optimization techniques presented in the first part of this project have been tested and already analyzed. However, for completeness we have included the analysis of a series of optimization techniques applied to the vanilla implementations (AS-IS) of the blocking algorithm, provided as part of the assignment.

The new techniques that we apply in this multi-thread implementation are the following:

3.1 Temporary write buffer for C

Since each thread will be concurrently doing writes, performing this directly on C will be both racey and potentially cause false sharing between adjacent sections. As a result, we tried a wide variety of methods to avoid this.

At first, we tried holding a stack buffer within each thread of our parallel section. However, as the array size grew with 32 threads, the smaller stack of each thread caused segmentation faults from stack overflows. In this approach, we would also be forced to write out each buffer in a critical section, serializing a large portion of "reducing" the separate buffers back into C.

We also tried substituting the stack memory for heap memory, and although this would avoid stack overflows per thread, we found there to be a slight performance degradation.

Rather than split the individual buffers across each stack, we tried creating a singular large buffer to be split between threads. This worked better because on top of not overflowing the stack, and not requesting heap memory, we would have references to each buffer after exiting the parallel region so we could perform further optimizations when reducing them back to C.

3.2 "Reducing" the write buffers

A naive implementation would use a single thread to sum together the write buffers created through the parallelized area. However, this is slow, and caused huge performance hits, so we looked for alternatives.

Our first approach was to throw more threads and divide up the computation. Our first attempt used "pragma omp for" as a load sharing construct, with our algorithm looking like: FOR each threads buffer, FOR each entry in C. However, this just ran across the entirety of C and caused a lot of cache misses. In addition, it didn't parallelize well because for a given "entry" in C, only one thread could write directly into C at a time (even though there are many write buffers). We looked for a better approach.

We found that inverting the loop order, FOR each entry in C, FOR each thread buffer, worked better because instead of dividing up the write buffers between threads (resulting in mostly serial, non-cache friendly code),

we would instead divide up the area of C between threads, and sum across the buffers. We further improved this approach by using AVX2 to sum the buffers. This gave us a modest performance improvement.

3.3 Block size tuning

Our parallelized section is the single-threaded optimized do block subroutine from part 1. However, we could now vary the initial block size in the subroutine because this defined a unit of work for the threads, and we wanted to load balance as well as we can.

We discovered that, in general, smaller matrices required smaller block sizes to accommodate for load balancing. As a result, we scale our subroutine block sizes with respect to the size of the input matrices in attempt to get peak performance from our implementation. We generated these block sizes empirically from tests on Cori.

3.4 Thread Scaling and Scheduling

We never have more threads than we do blocks to operate on. We also realized that the overhead of having dynamic scheduling is less than the performance improvements we get by doing so (this allows us an easy way to load balance our threads because of the "fringe" blocks when we have less threads than blocks).

3.5 Copy Elision

Despite all of the optimizations above the amount of synchronization was still pretty high. One final optimization we made was not copying empty spaces. When allocating a buffer per thread, a lot of the times many of the entries in the buffer are not written to. On $O(n^2)$ blocks, we had $O(num\_threads * n^2)$ buffers. However, there are in total only $O(n^3)$ blocks. As noted in the Block size tuning, we scale the block size with the size of the array, so by keeping n relatively small, we find that since $n < num\_threads$, we choose to sum n rather than the $num\_threads$ we were doing before.

By having fewer buffers than threads, we also had more compute at our disposal to increase the bandwidth of the memset we use to clear our buffers at initialization. Using less buffers nearly doubled the bandwidth of our algorithm.

3.6 Kahan and Floating Point Precision

One final problem we ran into was that often we would get non-deterministic correctness issues on our matrix sums. We discovered this to be because of floating point error. We chose to implement Kahan's algorithm, even though it is stated that "The BLAS standard for linear algebra subroutines explicitly avoids mandating any particular computational order of operations for performance reasons, and BLAS implementations typically do not use Kahan summation." However, without better knowledge, for a large error threshold of 3ϵ, we would be best off minimizing the difference between BLAS output and our own by minimizing the true error ourselves. This caused a minor performance degradation, about 0.5% of the peak.

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 Vanilla implementation

In Figure 1 we can see the different performance achieved by the simple AS-IS OpenMP algorithms provided with the assignment as well as our single-threaded optimized version, with a naive implementation of the OpenMP pragmas for the outer loops of the algorithm (before blocking). From the results, we obtain the following insights: (1) the vendor's optimized BLAS library is not as powerful as in the single-threaded case in terms of the theoretical average peak performance achieved, obtaining a performance around 26% when solving the reduced set of matrices, in comparison to the 80% reached in the first part of the project, (2) both AS-IS implementations of the Naive and Naive blocked multi-threaded algorithms obtain a very poor performance (between $1\% - 2\%$), and (3) our optimized single-threaded algorithm is not even close to its previous performance: now it is around a 2.5% of the theoretical peak.

Therefore, a series of modifications should be performed in order to improve the AS-IS performance of our code, modifications that we will test on the simple blocked algorithm version in order to obtain insights about the incidence of some optimization techniques in the context of the parallel performance of the code.

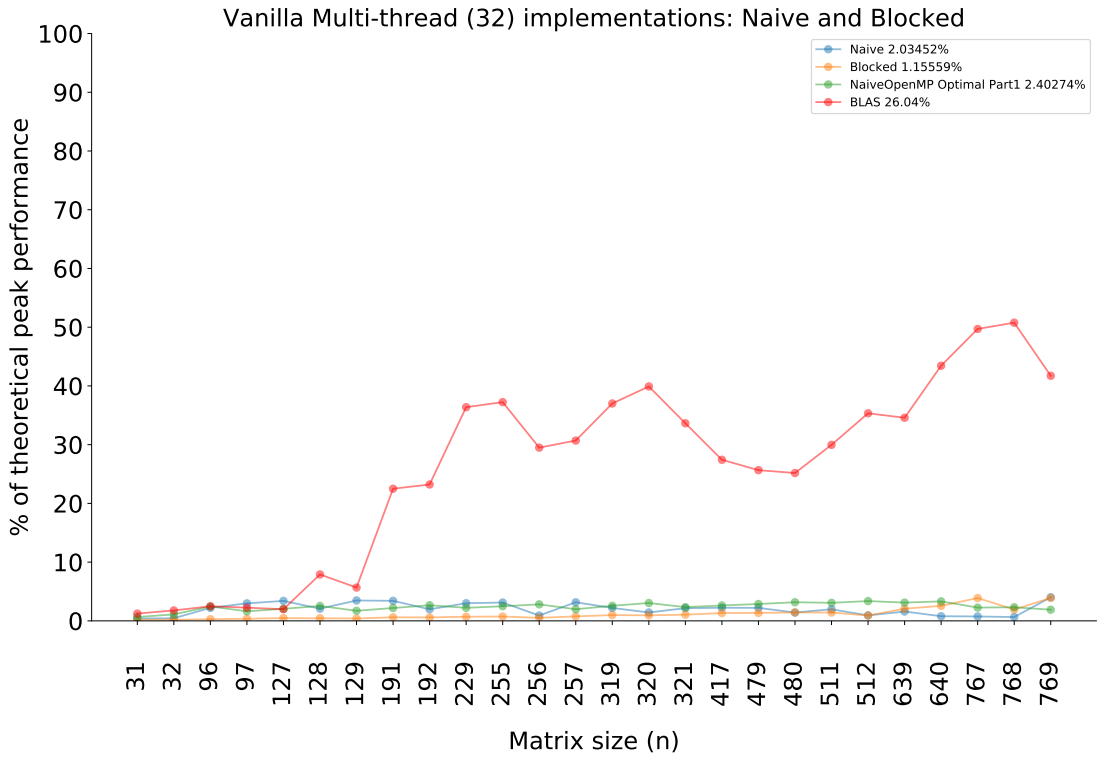

Figure 1: Performance of the vanilla OpenMP implementations of the: (1) Naive, (2) Blocked, and (3) Optimal part 1 algorithms compared with the vendor's BLAS in Cori.

4.1.2 Packing, scheduling, and blocking size

Like in our single-threaded version, we test the impact of memory alignment techniques as well as the scheduling rules that are applied whenever a parallel loop is triggered for assigning and balancing the workload among the different threads. As a first step, the A matrix is packed into a row-major format allowing us to significantly reduce the number of cache misses, optimizing the memory access of the implementation. This operation can be easily performed in parallel via an embarrassingly parallel approach: each iteration of the loop is completely independent of the previous one.

After this operation, different scheduling rules are tested. By default, OpenMP schedules the parallel regions in such a way that When the parallel for block is entered, each thread is processing a set of loop iterations, evenly divided. This default approach - static - is good when the workload is similar among the threads. However, in our case, we expect an uneven workload among the threads due to the presence of fringes and not evenly divided matrices dimensions, therefore, we test two main rules: dynamic and automatic.

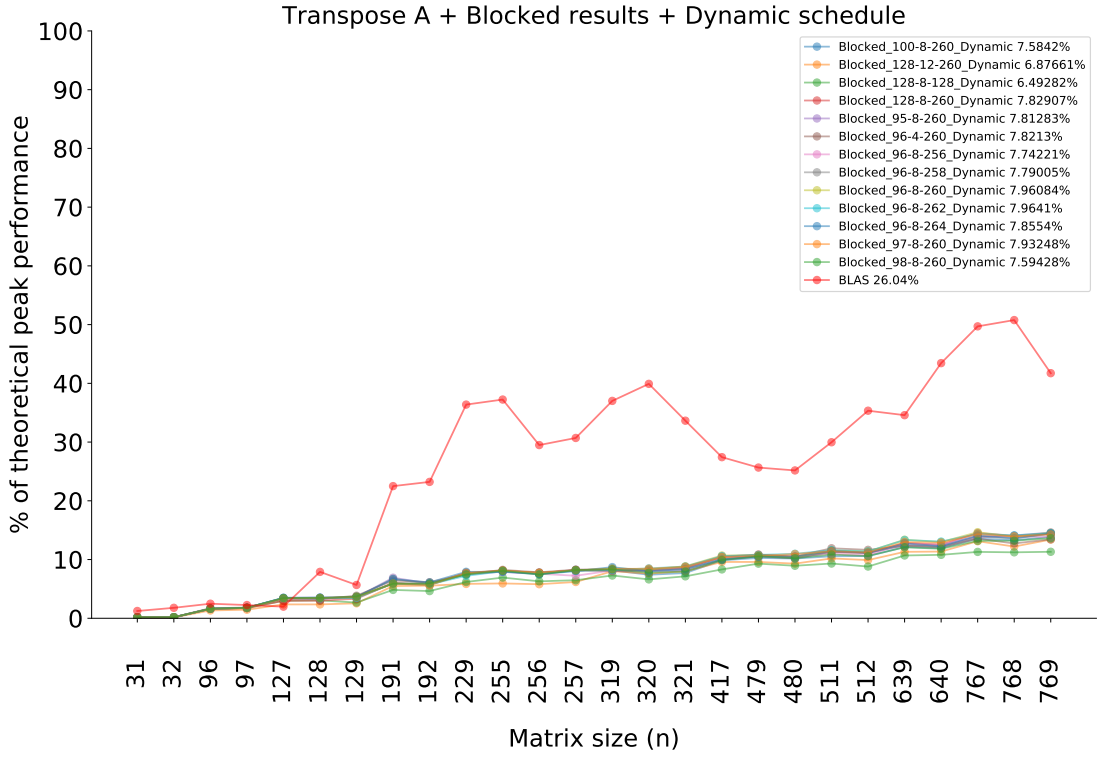

Figure 2: Blocked and transposed version using a dynamic schedule approach for different block sizes.

The dynamic schedule rule can lead to significant performance boost when different iterations take different amounts of time, being able to compensate the inherent overhead of the technique (each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration). Based on the results from Figure 2, we can see how a simple extra flag for the OpenMP pragma is giving us a "free" performance boost when combined with optimal block sizes. In contrast to the AS-IS implementation, we split the block size into three main parameters (associated with i, j, and k), obtaining more flexibility when tuning the size of the blocks. Different combinations with improved performance thanks to an efficient use of the memory hierarchy of Cori can be seen in Figure 2. With these simple techniques, we are able to obtain an average performance around 8%, without having to implement any efficient kernel (just the naive dot product), giving us a lot of space for improving the current implementation [1].

In a similar way, we test different blocking sizes with the automatic (auto) scheduling rule. In this case, the scheduling rule decision is delegated to the compiler and/or runtime system. The idea is to take advantage of the natural compiler behavior in order to obtain a free performance boost. Looking at the results in Figure 3 we can see a similar - slightly worse - performance in comparison to the dynamic approach. Hence,

[1] Full instances results can be seen in the Appendix.

the dynamic scheduling seems to be suitable for the optimization of our code and will be tested in our final implementation.

Again, one of the most significant parameters to tune in the final version of our code will be the block size when performing the blocking algorithm. As before, we perform a brute force approach for finding the optimal values for our implementation.

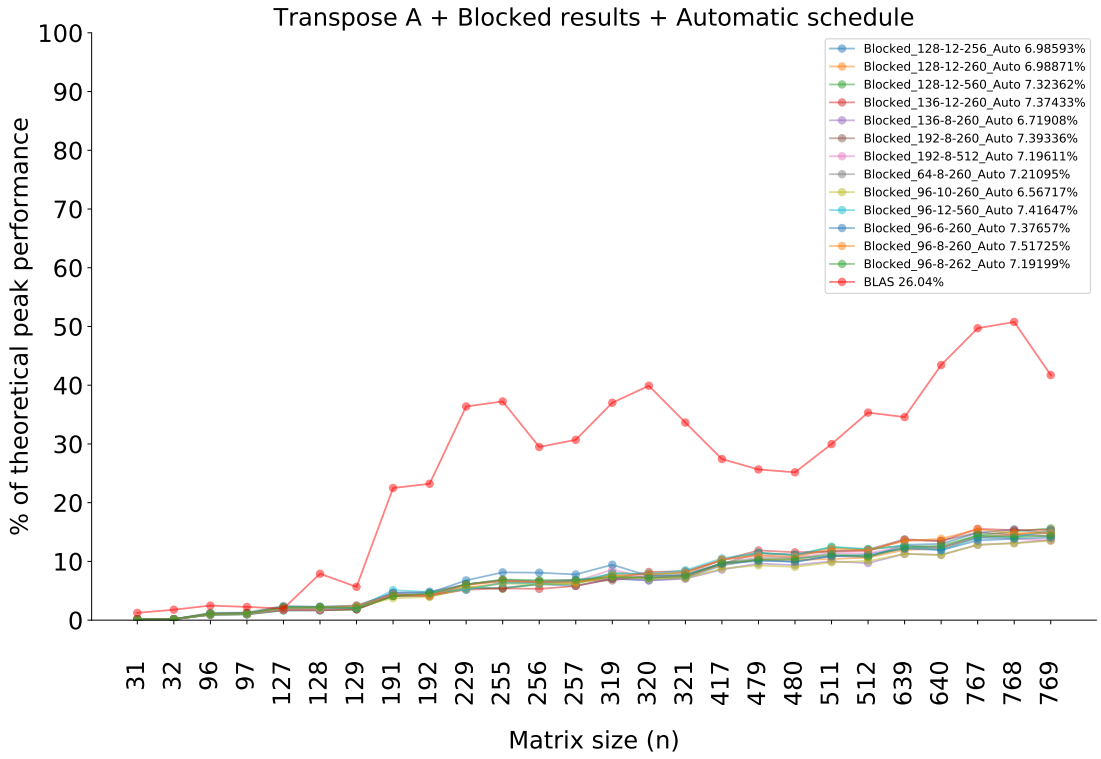

Figure 3: Blocked and transposed version using an automatic schedule approach for different block sizes.

Applying the previous techniques to our single-threaded optimized version, we are able to reach performances around 10%, still far away from its great performance when solving the instances with only one thread. Therefore, the next steps are focused on obtaining a better and more natural parallel implementation, taking into account the temporary buffers of the C matrix as well as applying the "reducing" the write buffers technique indicated in section 3.2.

4.1.3 Loops: Order and Unroll

Based on the analysis of the first part of the project, we are implementing the same optimal unrolling factors as well as loop orders obtained for the optimized single-threaded program (see part 1, section 4.1.1, 4.1.4, and 4.2.2 for the Pseudo-code of the algorithm).

4.1.4 Vectorization

Similarly, the main inner kernels for performing operations on 8x4, 8x3, 4x4, and 2x2 matrices are kept exactly as they were implemented in the optimized single-threaded algorithm. All of them consisting of a series of vectorized operations exploiting the AVX/AVX2 intrinsics commands for obtaining a significant performance boost in terms of calculations.

4.1.5 Padding

Based on the implementation in part 1, this technique is not included in our final program due to the performance hits experienced among all the tested instances. Besides the necessity of generating and allocating

the modified/padded matrices in memory, our tested padding function may not as efficient as it can be. Further research can be performed in this particular topic for checking potential performance improvements in future versions of our code.

4.2 Optimized approach

Using these results and our understanding of Cori's hardware architecture, we are ready to merge the most relevant techniques into an efficient and optimized multi-threaded 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

As stated in the previous discussion/analysis, the final optimized implementation is mainly based on our optimized single-threaded program, including the relevant insights obtained from the blocked algorithm analysis. The full description of this implementation can be seen in the part 1 report of this project, section 4.2.1., however, we include the most relevant points below.

In our best performance single-threaded 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. These operations are complemented and optimized using the loop unrolling technique in two levels. Different kernels and loop unrolling schemes were tested in order to maximize the total number of components from A and B processed at the same time, based on a brute search algorithm.

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 2, we present the detailed revised pseudo-code for our parallel version of the matrix multiplication. We first calculate the size and shape of the fringes, then assign each thread an appropriate part of the copy of A and B matrix to be calculated. We then dynamically schedules threads to run at appropriate blocks, calling the "do_block" procedure which is described in our part 1. Since there's some floating point error when we are collecting back the calculation results, we apply Kahan's algorithm to efficiently reduce the error.

Algorithm 1 Version 2 Pseudo-code Main

```
1: procedure SQUARE_DGEMM(INT N, DOUBLE* A, DOUBLE* B, DOUBLE* C)
2:     step 1: BLOCKSIZE setup according to lda.    // e.g If lda = 63, SIZE = 48; See attached code.
3:     int OPMAX = 32;
4:     int blockEdge = (lda - 1) / BLOCKSIZE + 1;
5:     int M, N;
6:     int intermed = max(blockEdge*blockEdge);
7:     int numThreads = min(OPMAX, intermed);
8:     int buf = 1024;                                                    // buffer to store intermediate results.
9:                                                  //hacking so that we get byte-aligned contiguous memory.
10:    int cLen = lda * lda / buf * buf + 2 * buf;
11:    int numCopies = blockEdge - 1;
12:    char hack[(numCopies) * cLen * sizeof(double)];
13:                                                                                    // 32-bit aligned
14:    double* copies = hack + (1024 - (long)hack % 1024);
15:    #pragma omp parallel num_threads(numThreads)
16:    int tid = omp_get_thread_num();
17:    if (numCopies > 0) then
18:        int threadsPerBlock = numThreads / (numCopies);
19:        if (tid < threadsPerBlock * (numCopies)) then
20:            int blockIndex = tid / threadsPerBlock;
21:            int bufIndex = tid % threadsPerBlock;
22:            int numElemsNormal = lda * lda / threadsPerBlock / 128 * 128;
23:            int numElems = numElemsNormal;
24:            if (bufIndex == threadsPerBlock - 1) then
25:                numElems = lda * lda - (threadsPerBlock - 1) * numElems;
26:            memset(copies + blockIndex * cLen + bufIndex * numElemsNormal, 0, numElems *
    sizeof(double));
27:    #pragma omp barrier                                                    // Columns
28:    #pragma omp for collapse(2) schedule(dynamic)
29:    for (int j = 0; j < lda; j+ = BLOCKSIZE) do
30:        // Blocks the columns of the matrix A
31:        // Correct block dimensions if block "goes off edge of" the matrix
32:                                                                                    // Rows
33:        for (int i = 0; i < lda; i+ = BLOCKSIZE) do
34:            // Blocks the rows of the matrix C
35:            // Correct block dimensions if block "goes off edge of" the matrix
36:            double * copy = j == 0 ? C : copies + (j - 1) / BLOCKSIZE * cLen;
37:            N = min(BLOCKSIZE, lda - j);
38:            M = min(BLOCKSIZE, lda - i);
39:            // Call the do_block function: Pack B if i = 0 (improving performance)
40:            do_block(1, lda, M, lda, N, A + i + j*lda, B + j, copy + i);
41:    // Now do Kahan's algorithm...
```

Algorithm 2 Version 2 Pseudo-code Main

```
1: procedure SQUARE_DGEMM(int n, double* A, double* B, double* C)
2:     // Kahan's algorithm
3:     if (BLOCKSIZE < lda) then
4:         int unrollFactor = 32;
5:         int down = lda * lda / unrollFactor * unrollFactor;
6:         volatile register __m256d c_f1, c_f2, c_buf1, c_vec, y_vec, t_vec;
7:         double zeroes[4] = 0;
8:         #pragma omp for
9:         for (inti = 0; i < down; i+ = unrollFactor) do
10:            for (intk = 0; k < unrollFactor; k+ = 4) do
11:                // Load the values from C into the AVX data variables
12:                c_f1 = _mm256_load_pd(C + i + k);
13:                c_vec = _mm256_load_pd(zeroes);
14:                for (intj = 0; j < num_copies; j + +) do
15:                    c_buf1 = _mm256_load_pd(copies + j * c_len + i + k);
16:                    y_vec = _mm256_add_pd(c_vec, c_buf1);
17:                    t_vec = _mm256_add_pd(c_f1, y_vec);
18:                    c_vec = _mm256_sub_pd(y_vec, _mm256_sub_pd(t_vec, c_f1));
19:                    c_f1 = t_vec;
                   _mm256_storeu_pd(C + i + k, c_f1);
       #pragma omp for
20:        for (int i = down; i < lda * lda; i + +) do
21:            volatile double sum = C[i];
22:            volatile double c = 0.0;
23:            for (intj = 0; j < num_copies; j + +) do
24:                volatile double y = copies[j * c_len + i] + c;
25:                volatile double t = sum + y;
26:                c = y - (t - sum);
27:                sum = t;
               C[i] = sum;
```

4.2.3 Results

Based on the described pseudo-code, our final implementation is tested in Cori. We observed a good improvement pattern similar to the BLAS implementation. We have talked about the comparisons in detail in section 3 above. Since we only implements the most relevant features, there's some gap between BLAS results and our results, which shall be anticipated: our multi-threaded optimized version is able to reach a performance around 12.6% for the representative matrices subset and around 18.3% when it is tested with all the instances provided. This difference is easily explained by the fact that increasing the size of the matrices implies a better usage of the parallel implementation, significantly boosting the performance of the average theoretical peak reached in contrast to the small subset of testing instances.

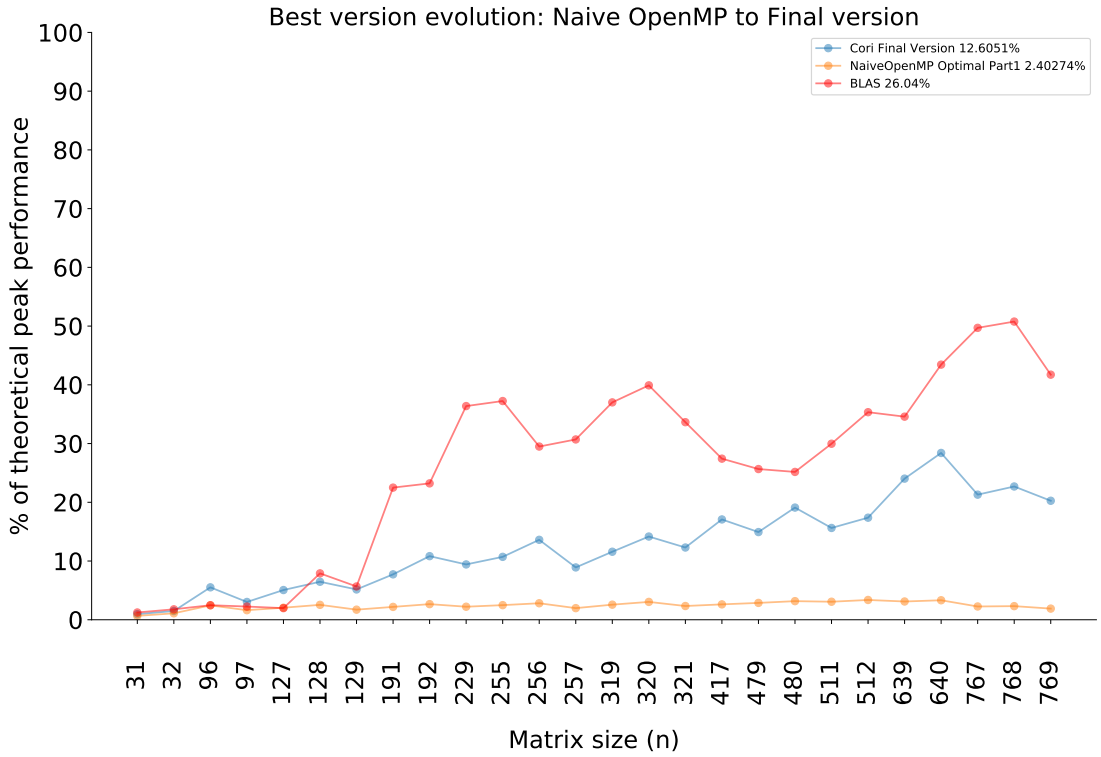

Figure 4: Best version evolution from its naive OpenMP implementation to the final version using the insights obtained from the experiments and optimization analysis.

Interesting is to notice the fact that our program is able to obtain better performance in small matrices in comparison to the vendor's BLAS implementation. A possible explanation lies in the fact that our code includes a series of different kernels for dealing with matrices of different sizes, as well as the thread scaling technique described in section 3. Once we pass the instance $n = 129$, we can see that BLAS is beating our implementation consistently, with some similar performance peaks inside the interval $[400, 640]$. Same pattern is repeated with larger instances, as seen in Figure 5. In addition, interesting is to see the high variability of the BLAS results reaching some very high peaks followed by poor performance valleys depending on the matrices dimensions, clearly explained by all the already discussed optimization techniques in part I and part II of the project.

Although the existing gap with the BLAS code results, we consider the final version a successful implementation considering the original starting point of our single-threaded optimized code under a naive OpenMP approach, as seen in Figure 4.

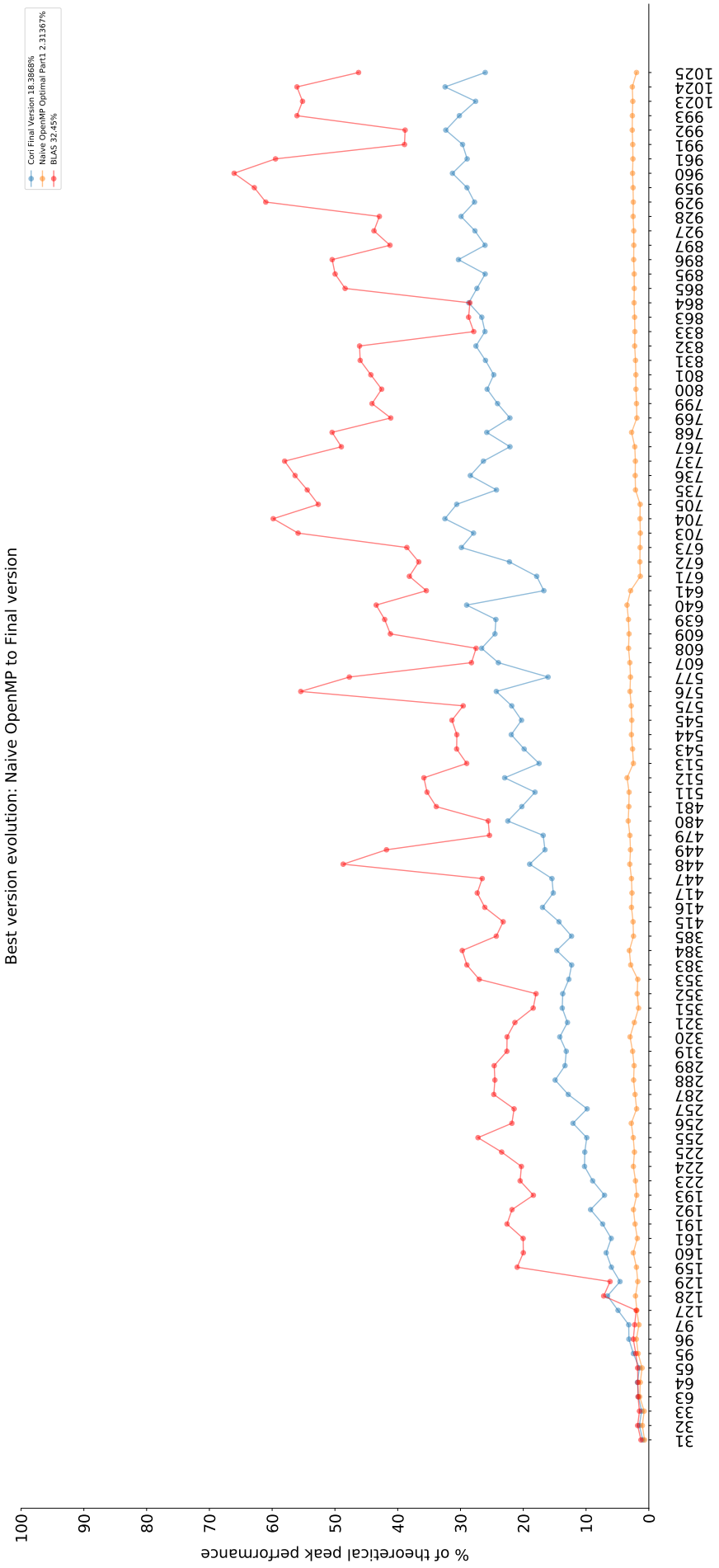

Figure 5: Best version evolution from its naive OpenMP implementation to the final version using the insights obtained from the experiments and optimization analysis (all results).

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 6.

As seen in Figure 6, the performance of the code in the personal computer, as expected, is significantly less than the one obtained in Cori even when we take into account the laptop's turbo boost bias: 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. However, we show the biased results in order to explicitly notice how the optimized code for Cori is not suitable for our laptop since a lot of the optimizations performed are hardware dependent.

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.

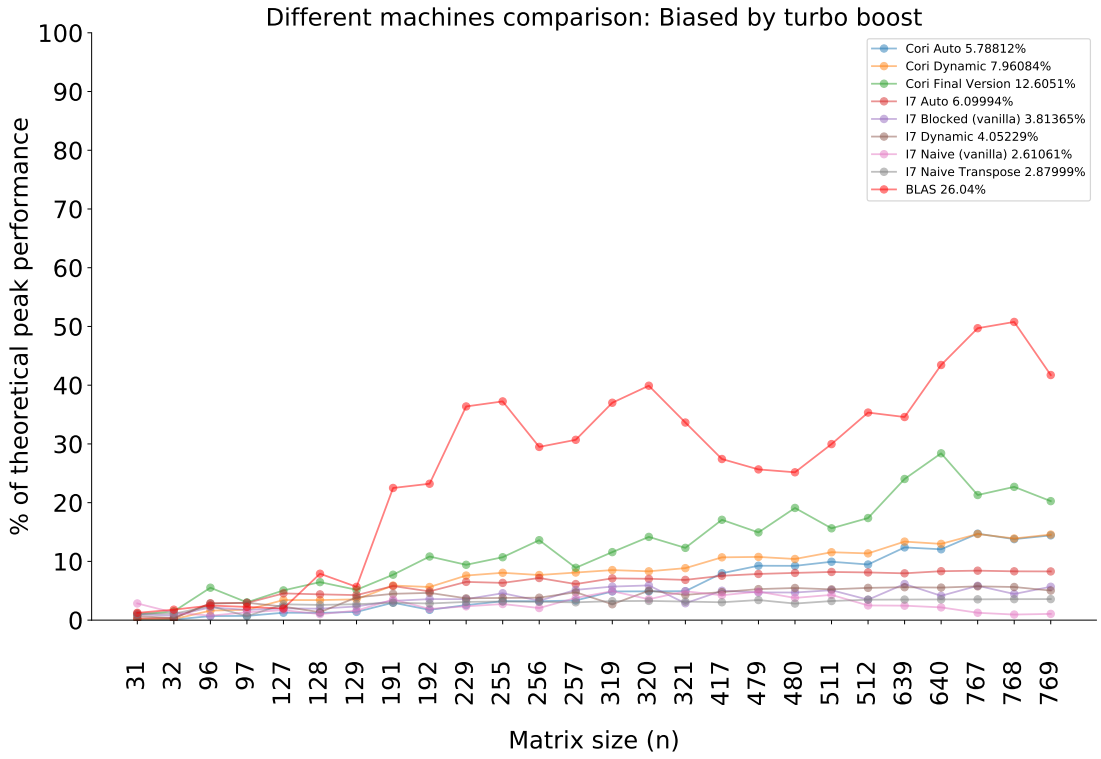

Figure 6: Different machines performance comparison: Turbo bias is allowed for explicitly showing the poor performance on the personal laptop.

From the results in Figures 6 and 7, we can clearly see that while the best implementation in Cori reaches performance levels around 12.6% and 18.4% for the sample and full instances sets, we are not able to obtain performance values above 6% for our laptop, even allowing it to use the turbo boost feature indicated above. As an interesting detail, we can notice how the best implementation for the laptop consists of the blocked approach using the automatic schedule rule, hence, the compiler *gcc* is performing a good job when selecting the most suitable scheduling approach for the program.

The average performance peak obtained when running the Cori's optimized final version in our laptop is around $2-3\%$ - turbo boost allowed - even worse than the naive transpose and blocked algorithms as seen in Figures 6 and 7. This is mainly due to memory's hierarchy problems as well as the number of threads: our laptop can only handle 8 (4 cores with hyper-threading) with a sustainable performance.

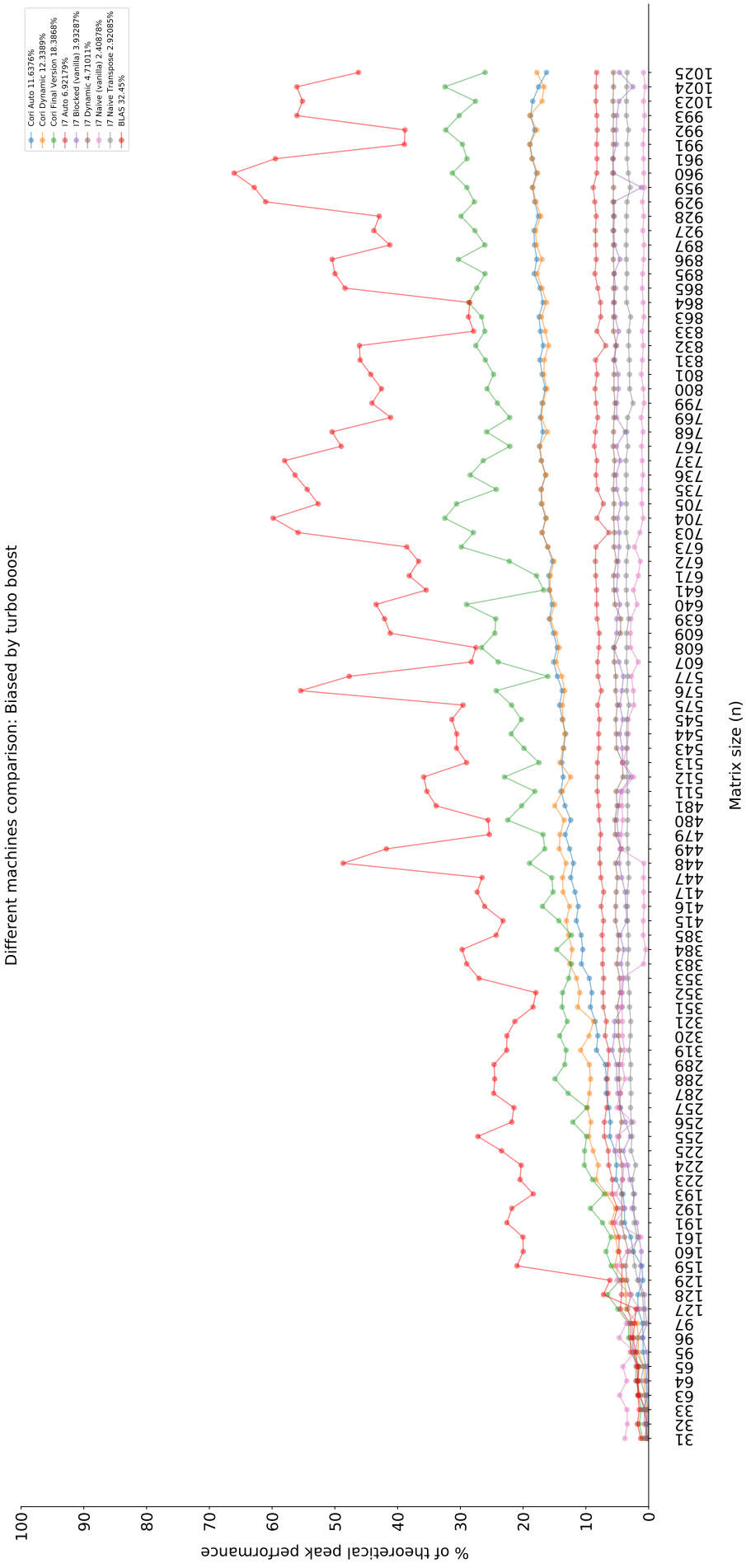

Figure 7: Different machines performance comparison: Turbo bias is allowed for explicitly showing the poor performance on the personal laptop (full results).

5 Conclusions & Future Work

1. A successful implementation of the multi-threaded version of dgemm squared double precision floating point matrix multiplication algorithm has been implemented using OpenMP instructions. The usefulness of this library is proven as well as its simplicity of usage: instructions are simple and easy to add to any program, allowing the programmer to easily decide which portions of the code should be executed by a team of threads or by just the main one, without adding too much complexity to the original serial-threaded program.

   However, the optimization process from the optimized single-threaded version from part I has been extremely challenging: a series of experiments needed to be performed in order to exploit he parallel implementation of the algorithm and boost the performance of the code. Hence, the team has been able to understand the inherent complexity and challenges that a professional programmer faces when trying to optimize algorithms for a particular hardware architecture while dealing with shared-memory implementations.

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. In contrast to the optimized single-threaded implementation, a series of numerical problems regarding the floating point precision arose during the implementation of the multi-threaded version. Kahan's algorithm allowed us to deal with these complications without significantly impacting the performance of our best version implementation.

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. In addition, an MPI implementation can be developed in order to compare the performance of a distributed-memory paradigm versus a shared-memory approach. However, programming complexity is significantly increased.

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

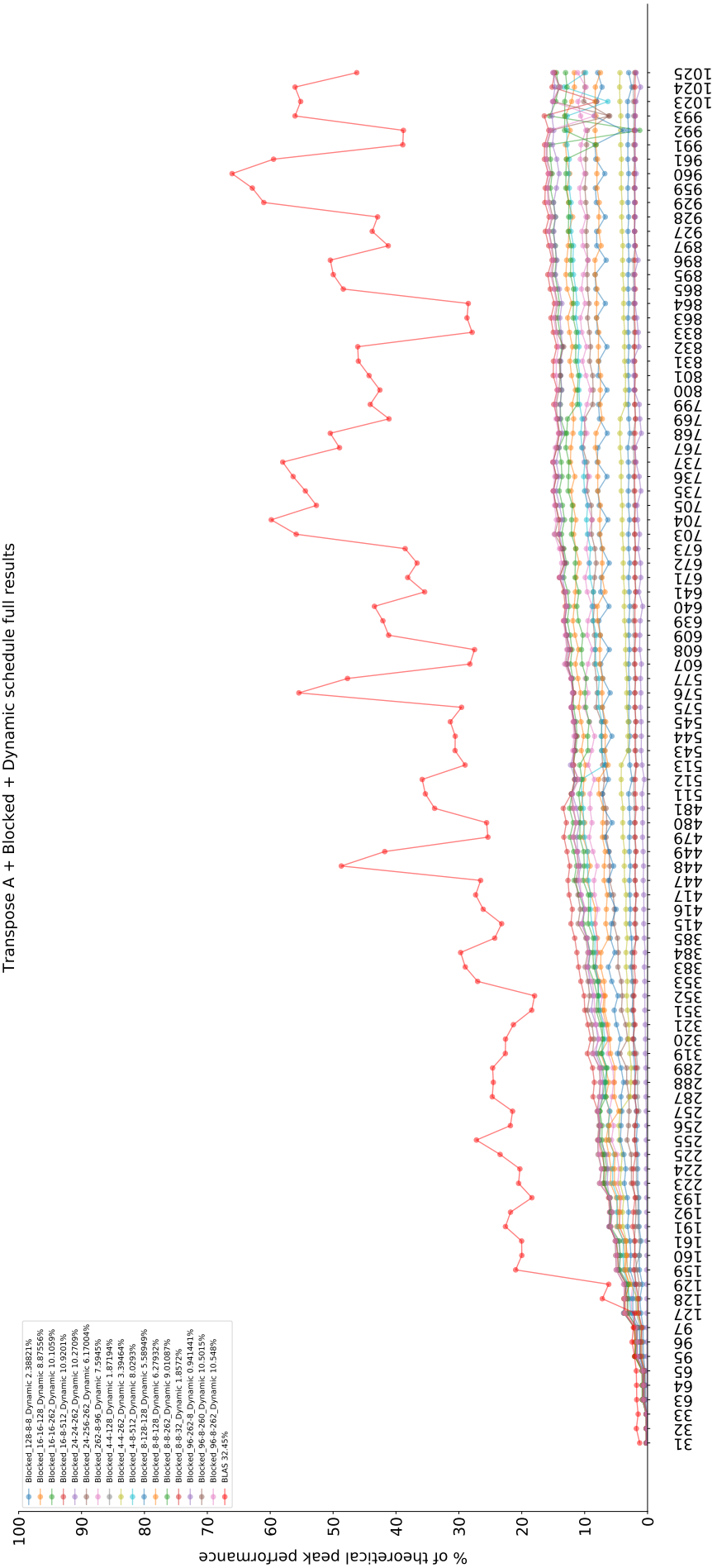

Figure 8: Blocked and transposed version using an automatic schedule approach for different block sizes (all results).