Implementing BLAS Operation using OpenMP

Amit Singh   Cem Bassoy

May 2, 2021

Contents

Chapter 1

Introduction

Linear algebra is a vital tool in the toolbox for various applications, from solving a simple equation to the art of Deep Learning algorithm or Genomics. The impact can felt across modern-day inventions or day to day life. Many tried to optimize these routines by hand-coding them in the assembly or the compiler intrinsics to squeeze every bit of performance out of the CPU; some chip manufacturers provide library specific to their chip. A few high-quality libraries, such as **Intel's MKL**, **OpenBLAS**, **Flame's Blis**, **Eigen**, and more, each one has one common problem, they are architecture-specific. They need to be hand-tuned for each different architecture.

There three ways to implement the routines and each one has its shortcomings:

1. Hand code them in assembly and try to optimize them for each architecture.
2. Use the compiler intrinsics.
3. Simply code them and let the compiler optimize.

1.1 Hand-tuned Assembly

Hand-coded assembly gives high performance due to more control over registers, caches, and instructions used, but that comes with its issues, which we exchange for performance:

- Need a deep understanding of the architecture
- Maintenance of the code
- It violates the DRY principle because we need to implement the same algorithm for a different architecture
- Development time is high
- Debugging is hard
- Unreadable code
- Sometimes lead to micro-optimization or worst performance than the compiler

Intel's MKL, **OpenBLAS** and **Flame's Blis** comes under this category

1.2 Compiler Intrinsics

The compiler intrinsics is one layer above the assembly, and we lose control over which register to use. If an intrinsic has multiple representations, then we do not know which instruction will emit. There are the following issues with this approach:

- Need the knowledge of intrinsic

- Maintenance of the code much better than assembly but not great
- DRY principle achieved with little abstraction
- Development time is better than assembly but not great
- Debugging is still hard
- Unreadable code if not careful
- May lead to micro-optimization or worst performance than the compiler

Eigen uses the compiler intrinsics, and they fixed and avoided some of the above problems with the right abstraction.

1.3 Compiler Dependent Optimization

The compiler has many tools for optimizing code: loop-unrolling,
auto-vectorization, inlining functions, etc. We will rely on code vectorization heavily, but auto-vectorization may or may not be applied if the compiler can infer enough information from the code. To avoid unreliable auto-vectorization, we will use **OpenMP** for explicit vectorization, an open standard and supported by powerful compilers. The main issues are:

- Performance depends on the compiler
- No control over vector instructions or registers
- Auto-vectorization may fail

Boost.uBLAS depends on the auto-vectorization, which does not guarantee the code will vectorize.

1.4 BLAS Routines

There are four BLAS routines that we will implement using **OpenMP**, which uses explicit vectorization and parallelization using threads. Each routine has its chapter, and there we go much deeper with performance metrics.

1. Vector-Vector Inner Product (**?dot**)
2. Vector-Vector Outer Product (**?ger**)
3. Matrix-Vector Product or Vector-Matrix Product (**?gemv**)
4. Matrix-Matrix Product (**?gemm**)

1.5 Machine Model

The machine model that we will follow is similar to the model defined in the [Low et al., 2016], which takes modern hardware into mind. Such as vector registers and a memory hierarchy with multiple levels of set-associative data caches. However, we will ignore vector registers because we let the **OpenMP** handle the registers, and we do not have any control over them. However, we will add multiple cores where each core has at least one cache level that not shared among the other cores. The only parameter we need to put all our energy in is the cache hierarchy and how we can optimize the cache misses.

All the data caches are set-associative and we can characterize them based on the four parameter defined bellow:
C_{L_i}: cache line of the i^{th} level
W_{L_i}: associative degree of the i^{th} level
N_{L_i}: Number of sets in the i^{th} level
S_{L_i}: size of the i^{th} level in Bytes

$$S_{L_i} = C_{L_i} W_{L_i} N_{L_i} \tag{1.1}$$

Let the S_{data} be the width of the type in Bytes.

We are assuming that the cache replacement policy for all cache levels is **LRU**, which also assumed in the [Low et al., 2016] and the cache line is same for all the cache levels. For most of the case, we will try to avoid the associative so that we could derive a simple equation containing the cache size only from the equation 1.1.

1.6 Performance Metrics

1.6.1 FLOPS

It represents the number of floating-point operations that a processor can perform per second. The higher the Flops are, the faster it achieves the floating-point specific operations, but we should not depend on the flops all the time because it might be deceiving. Moreover, it does not paint the whole picture.

$$FLOPS = \frac{Number\ of\ Operation}{Time\ taken} \tag{1.2}$$

1.6.2 Speedup

The speedup tells us how much performance we were able to get when compared to the existing implementation. If it is more significant than one, then reference implementation performs better than the existing implementation; otherwise, if it is less than one, reference implementation performs worse than the existing implementation.

$$Speedup = \frac{Flops_{reference}}{Flops_{existing}} \tag{1.3}$$

1.6.3 Speed-down

The speed down is the inverse of the speedup, and if it is below one, then we performing better than the existing implementation; otherwise, we are performing worse.

$$Speeddown = \frac{Flops_{existing}}{Flops_{reference}} \tag{1.4}$$

1.6.4 Peak Utilization

This tells us how much CPU we are utilizing for floating-point operations when the CPU can compute X amount of floating-point operations.

$$Peak\ Utilization = \frac{Flops}{Peak\ Performance} \times 100 \tag{1.5}$$

Peak Performance can be calculated using the formula defined on the [FLOPS, 2021]

$$Peak\ Performance = Frequency \times Cores \times \frac{FLOPS}{Cycle} \tag{1.6}$$

1.7 System Information

Processor	2.3 GHz 8-Core Intel Core i9
Architecture	x86
L1 Cache	8-way, 32KiB
L2 Cache	4-way, 256KiB
L3 Cache	16-way, 16MiB
Cache Line	64B
Single-Precision(FP32)	32 FLOPs per cycle per core
Double-Precision(FP64)	16 FLOPs per cycle per core
Peak Performance(FP32)	588.8 GFLOPS
Peak Performance(FP64)	294.4 GFLOPS

1.8 Compiler Information

Compiler	Clang version 12.0.0
Compiler Flags	-march=native -ffast-math -fopenmp -O3
C++ Standard	20

1.9 library Version

Intel MKL	2020.0.1
Eigen	3.3.9
OpenBLAS	0.3.13
BLIS	0.8.0

Chapter 2

Vector-Vector Inner Product

This operation takes the equal length of the sequence and gives the algebraic sum of the product of the corresponding entries.

Let x and y be the vectors of length n, then

$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$, $y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$

$$x \cdot y = \sum_{i=0}^{n} (x_i \times y_i) \tag{2.1}$$

$$y \cdot x = x \cdot y \tag{2.2}$$

Equation 2.2 can be easily proven using equation 2.1.

$$y \cdot x = \sum_{i=0}^{n} (y_i \times x_i)$$

We know that multiplication on a scalar is commutative. Using this fact, we can say

$$y \cdot x = \sum_{i=0}^{n} (x_i \times y_i)$$

From the equation 2.1

$$y \cdot x = x \cdot y$$

2.1 Calculating Number of Operations

Using equation 2.1, we fill the below table

Name	**Number**
Multiplication	n
Addition	$n-1$

Total Number of Operations = Number of Multiplication + Number of Addition
Total Number of Operations = $n + (n-1)$
Total Number of Operations = $2n - 1$

2.2 Algorithm

Algorithm 1: Inner Product SIMD Function

```
// $a$ is the pointer to the first vector
// $b$ is the pointer to the second vector
// $n$ is the length of the vectors
Function simd_loop (a, b, n):
    $sum \leftarrow 0$
    #pragma omp simd reduction(+:sum)
    for $i \leftarrow 0$ to $n$ by 1 do
        $sum \leftarrow sum + a[i] \times b[i]$
    end
    return $sum$
end
```

Algorithm 2: Dot Product

```
Input: $c$, $a$, $b$, $max\_threads$, $n$
// $a$ and $b$ are pointer to the vectors
// $c$ is the pointer to the output
// $n$ is the size of the vectors
// $max\_threads$ is the user provided thread count
begin
    $number\_el\_L_1 \leftarrow \lfloor \frac{S_{L_1}}{S_{data}} \rfloor$
    $number\_el\_L_2 \leftarrow \lfloor \frac{S_{L_2}}{S_{data}} \rfloor$
    $block_1 \leftarrow 2 \times number\_el\_L_1$
    $block_2 \leftarrow \lfloor \frac{number\_el\_L_1}{2} \rfloor$
    $block_3 \leftarrow \lfloor \frac{number\_el\_L_2}{2} \rfloor$
    $sum \leftarrow 0$
    $omp\_set\_num\_threads(max\_threads)$
    if $n < block_1$ then
        $sum \leftarrow simd\_loop(a, b, n)$
    end
    else if $n < block_3$ then
        $min\_threads \leftarrow \lfloor \frac{n}{block_2} \rfloor$
        $num\_threads \leftarrow max(1, max(min\_threads, max\_threads))$
        $omp\_set\_num\_threads(num\_threads)$
        #pragma omp parallel for schedule(static) \
         reduction(+:$sum$)
        for $i \leftarrow 0$ to $n$ by $block_2$ do
            $ib \leftarrow min(block_2, n - i)$ $sum \leftarrow sum + simd\_loop(a + i, b + i, ib)$
        end
    end
    else
        #pragma omp parallel reduction(+:$sum$)
         begin
            for $i \leftarrow 0$ to $n$ by $block_3$ do
                $ib \leftarrow min(block_3,\ n - i)$
                $ai \leftarrow a + i$
                $bi \leftarrow b + i$
                #pragma omp for schedule(dynamic)
                 for $j \leftarrow 0$ to $ib$ by $block_2$ do
                    $jb \leftarrow min(block_2, ib - j)$
                    $sum \leftarrow sum + simd\_loop(ai + j, bi + j, jb)$
                end
            end
        end
    end
    $c \leftarrow sum$
end
```

The algorithm fragmented into three sections, and each part represents cache hierarchy.

2.2.1 The First Section

This section handles the data when the vector's length is less than the S_{L_1}. Here, we do not touch the threads because the overhead incurred is significant to paralysis the performance, and we get the worst performance compared with no thread or other libraries. When the whole data can fit in the L_1 cache, it is best not to use threads. We move to the next section when the cache misses large enough to compensate for the thread overhead.

According to the **Amdahl's Law**, we know

$$Speedup_N = \frac{1}{p + \frac{1-p}{N}} - O_N \tag{2.3}$$

p is the proportion of execution time for code that is not parallelizable, where $p \in [0, 1]$
$1 - p$ is the proportion of execution time for code that is parallelizable
N is the number of processor cores or number of threads, where $N \in Z_{\neq 0}$
O_N is the overhead incurred due to the N thread

Using the equation 2.3 for $N = 1$, we know the $Speedup_1 = 1$ since the program is already running on the thread, and we do not need to spawn another thread. Therefore the thread overhead incurred is none ($O_1 = 0$).

In order to perform better than the single-threaded program, we need the $Speedup_N$ to exceed the $Speedup_1$ and $N > 1$. We can now get the lower bound using the fact and the relationship between the vectors' length and the thread overhead.

$$Speedup_N > Speedup_1$$

From the equation 2.3, we get

$$\frac{1}{p + \frac{1-p}{N}} - O_N > 1$$

$$\begin{aligned}
O_N &< \frac{1}{p + \frac{1-p}{N}} - 1 \\
O_N &< \frac{N}{Np + 1 - p} - 1 \\
O_N &< \frac{N}{1 + p(N-1)} - 1 \\
O_N &< \frac{(N-1) - p(N-1)}{1 + p(N-1)} \\
O_N &< \frac{(N-1)(1-p)}{1 + p(N-1)} \\
O_N &< \frac{1-p}{\frac{1}{(N-1)} + p}
\end{aligned}$$

Let $\frac{1}{N-1}$ be C_N and $C_N \leq 1$, where $N > 1$

$$O_N < \frac{1-p}{C_N + p} \tag{2.4}$$

We know $p \in [0, 1]$. Now, we get

$$0 \leq \frac{1-p}{C_N + p} \leq \frac{1}{C_N}$$

$$0 \leq O_N < \frac{1}{C_N}$$

$$0 \leq O_N < N - 1$$

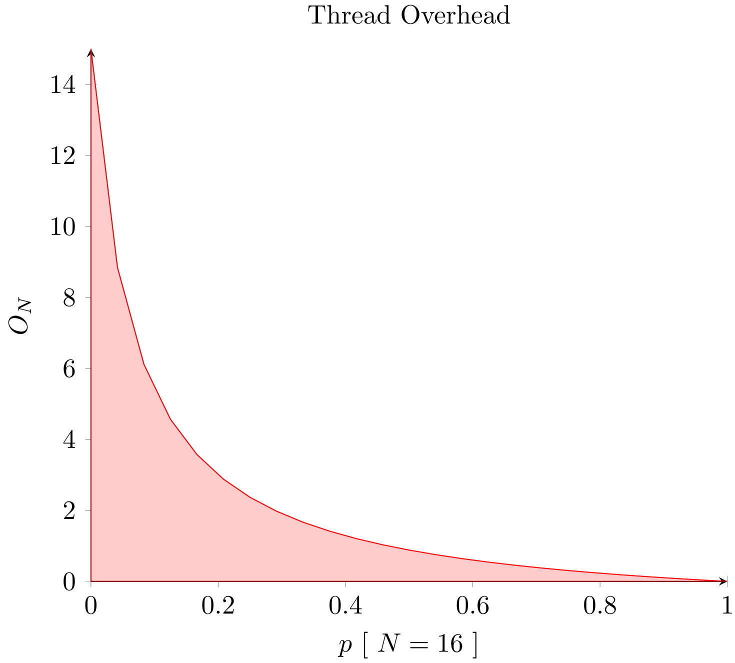
Therefore, $O_N \in [0, N-1)$. We can theorise that p must be a function of Block (B_1).

$$p \propto B_1$$
$$p = kB_1$$

where k is the proportionality constant. Now, replace p in the equation 2.4

$$O_N < \frac{1}{\frac{C_N + kB_1}{1 - kB_1}}$$

As we B_1 increase then $1 - kB_1$ decrease and $C_N + kB_1$ also increase, which over all decrease the overhead (O_N)



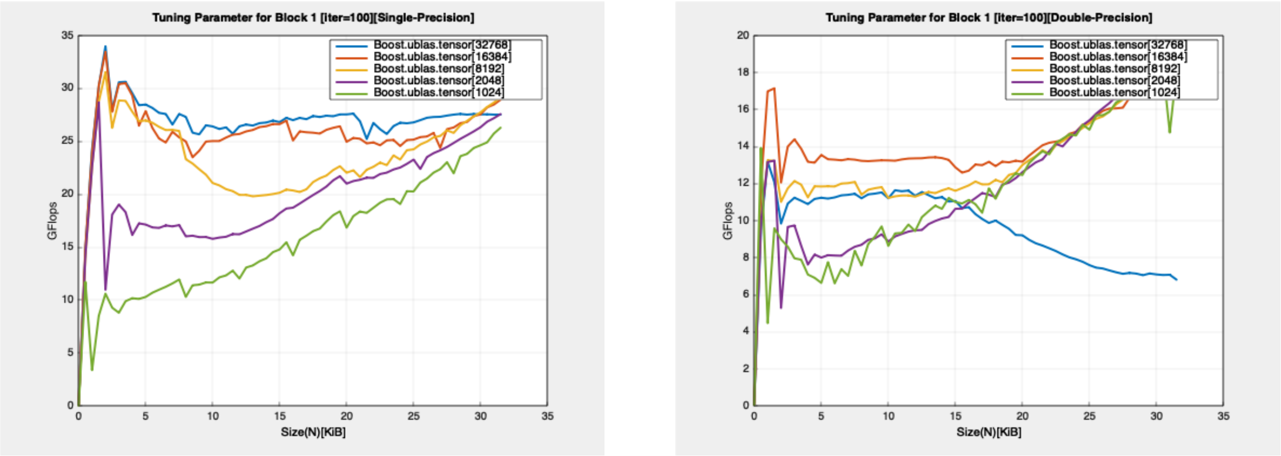
The B_1's lower bound found to be the number of elements that can be fit inside the L_1 cache through the experimental data. This phenomenon happens because the vectors' elements cannot fit inside the cache, and cache misses increase, dominating the thread overhead.

$$B_1 \geq \frac{S_{L_1}}{S_{data}}$$

However, the optimal block size for this section found to be twice the vectors' elements that can put inside the cache.

$$B_1 = \frac{2S_{L_1}}{S_{data}} \tag{2.5}$$

Experimental Data for B_1



(a) Single-Precision

(b) Double-Precision

	Block[Single]	**Block[Double]**
Lower Bound	8192 (8K)	4096 (4K)
Optimal	16384 (16K)	8192 (8K)

2.2.2 The Second Section

Here, the cache misses is large enough to dominate the thread overhead, and the thread can provide speedup more significant than a single thread. To quench the data demand for each thread, we fit both vectors' elements inside the L_1 cache, and the accumulator always is inside the register—the whole L_1 cache used for vectors. The advantage of the threads come in handy because most of the CPU architecture provides a private L_1 cache, which owned by each core, and as the inner product is an independent operation, no two elements are dependent on each other for the result. Each private L_1 cache can fetch a different part of the sequence and run in parallel without waiting for the result to come from another thread, because of which there is no need to invalidate the cache. Once the data comes inside the L_1 cache, it will compute a fragment of the sequence and keeps the accumulated result in the register and then fetch another part. In the end, it will combine all the partial results into one final result.

$$S_{L_1} = S_{data}(Block\ of\ the\ first\ vector + Block\ of\ the\ second\ vector)$$

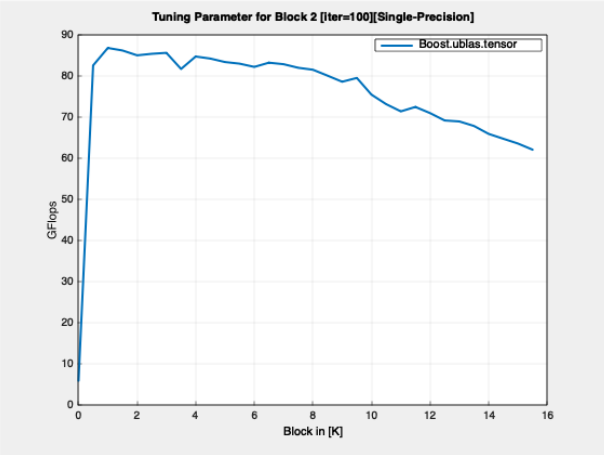
From the equation 2.1, we can infer that the block of both vectors must be equal to give the optimal block size.

$$B_2 = Block\ of\ the\ first\ vector = Block\ of\ the\ second\ vector$$

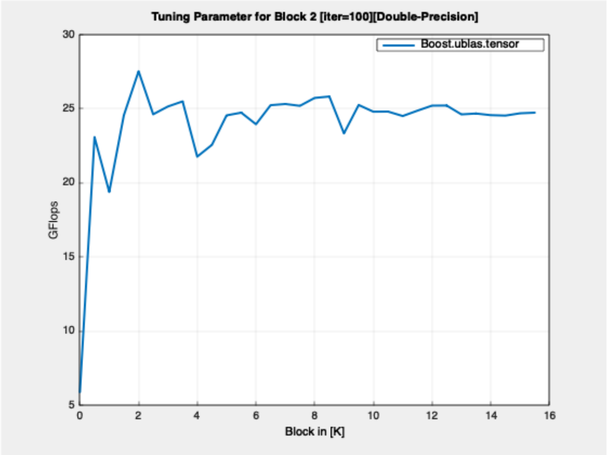
$$S_{L_1} = S_{data}(B_2 + B_2)$$
$$S_{L_1} = 2S_{data}B_2$$

$$B_2 = \frac{S_{L_1}}{2S_{data}} \tag{2.6}$$

Experimental Data for B_2



(a) Single-Precision



(b) Double-Precision

	Block[Single]	**Block[Double]**
Theoretical Optimal	4096 (4K)	2048 (2K)
Experimental Optimal	4096 (4K)	2048 (2K)

2.2.3 The Third Section

Here, once the data starts to exceed the L_2 cache or even L_3 cache, we use the CPU architecture's prefetch feature. This section may or may not affect some architecture; if the L_2 cache is also private for different cores, it will prefetch the data and fill it, and when it goes inside the loop, which utilizes the threads will fetch the data into the L1 cache from the L_2 cache. If the L_2 shared among all the cores, the gain might be minimal or see no gain in speedup. With the same logic we used to derive equation 2.6, we can use it here also.

$$S_{L_2} = S_{data}(Block\ of\ the\ first\ vector + Block\ of\ the\ second\ vector)$$

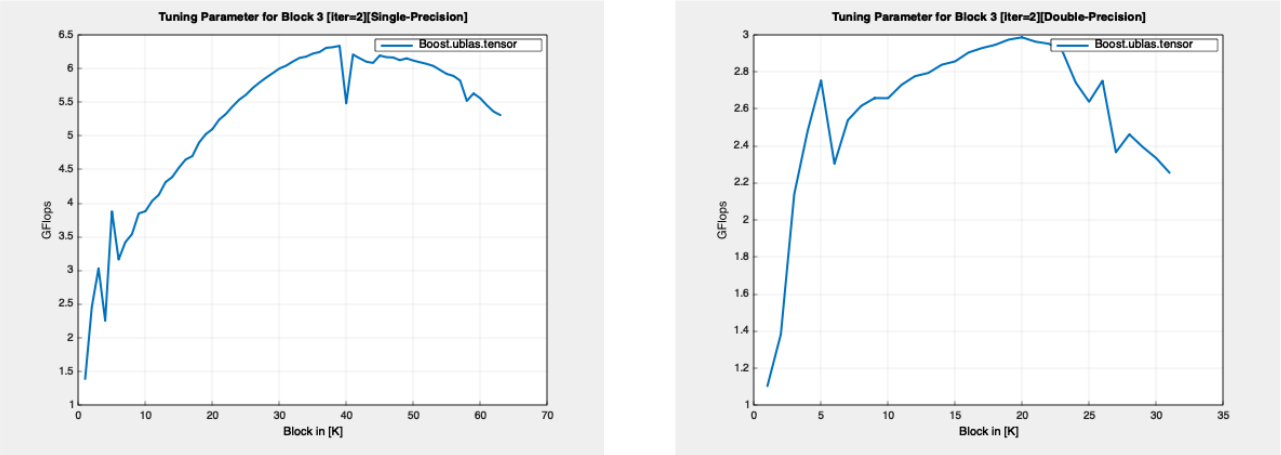
From the equation 2.1, we can infer that the block of both vectors must be equal to give the optimal block size.

$$B_3 = Block\ of\ the\ first\ vector = Block\ of\ the\ second\ vector$$

$$S_{L_2} = S_{data}(B_3 + B_3)$$
$$S_{L_2} = 2S_{data}B_3$$

$$B_3 = \frac{S_{L_2}}{2S_{data}} \tag{2.7}$$

Experimental Data for B_2



(a) Single-Precision

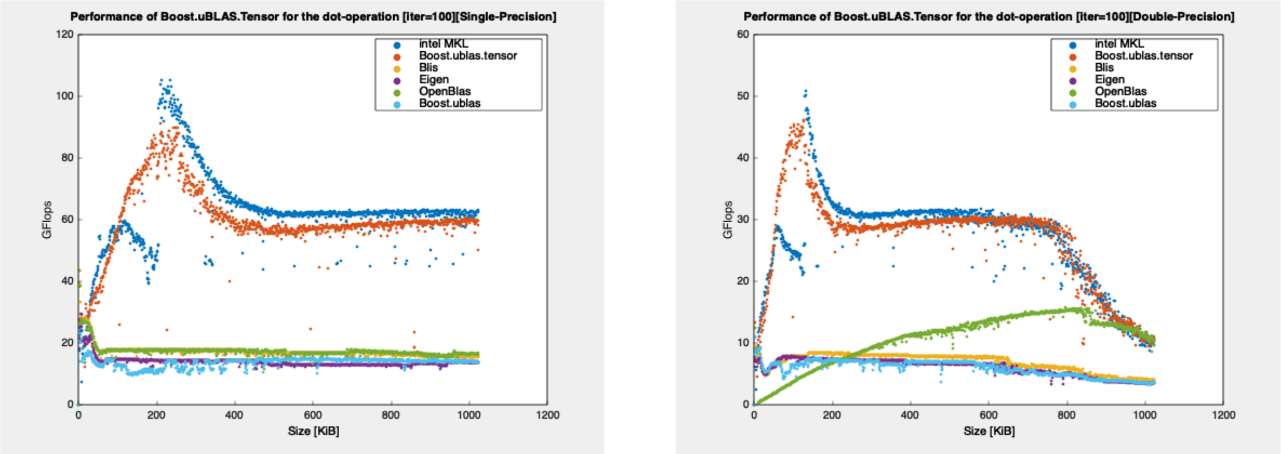
(b) Double-Precision

	Block[Single]	**Block[Double]**
Theoretical Optimal	32768 (32K)	16384 (16K)
Experimental Optimal	$\geq$ 30K and $\leq$ 40K	$\geq$ 16K and $\leq$ 20K

2.3 Performance Plots and Speedup Summary

2.3.1 Range[Start: 2, End: 2^{20}, Step: 1024]

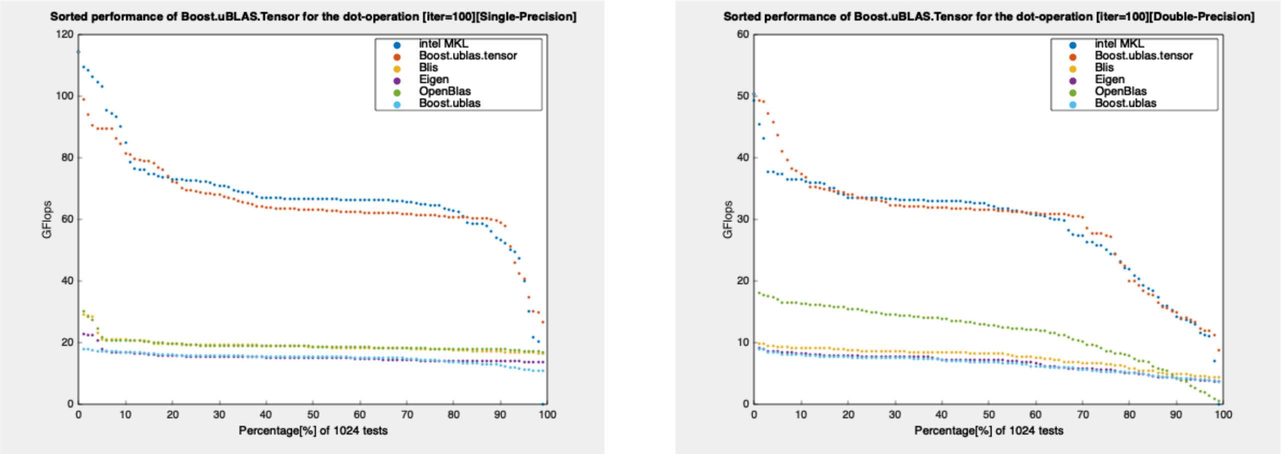
Performance measurements of ?dot implementations



(a) Single-Precision

(b) Double-Precision

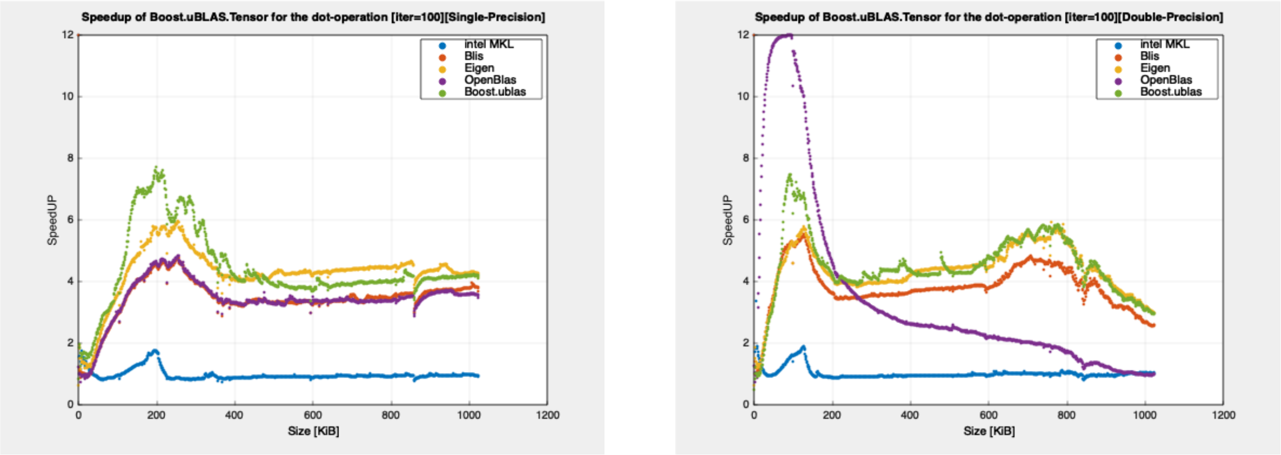
Sorted performance measurements of ?dot implementations



(a) Single-Precision

(b) Double-Precision

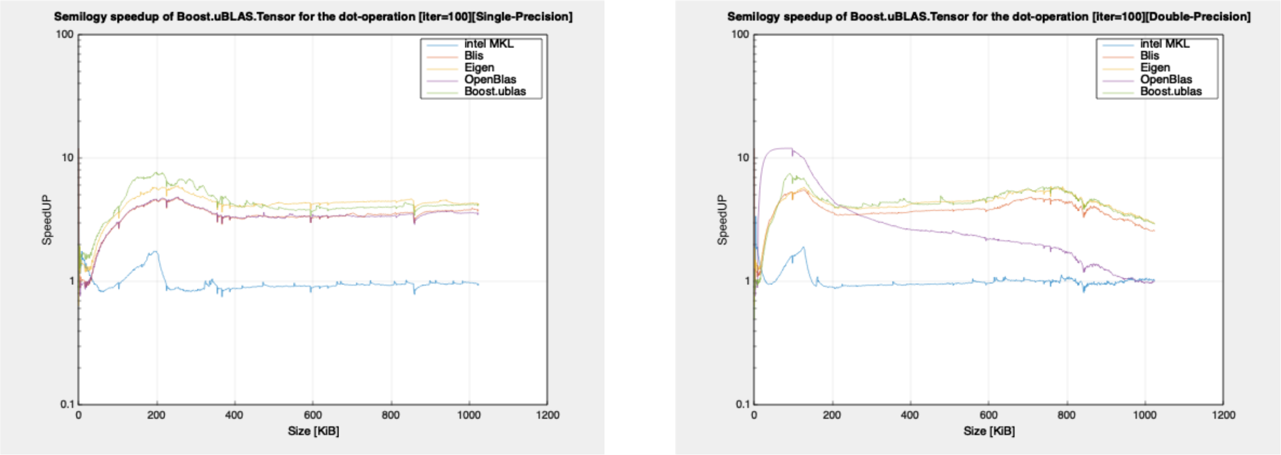
Comparison of the Boost.uBLAS.Tensor ?dot implementation



(a) Single-Precision

(b) Double-Precision

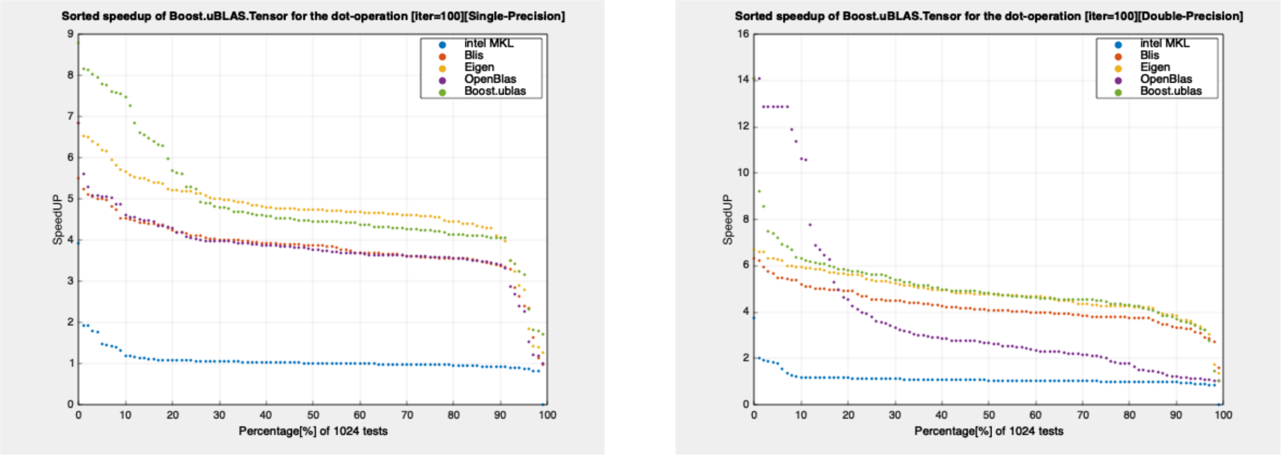
Comparison of the Boost.uBLAS.Tensor ?dot implementation [semilogy]



(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?dot implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 2.1: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	96
OpenBLAS	98	95
Eigen	99	96
Blis	98	96
Intel's MKL	62	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	0	0
Intel's MKL	35	0

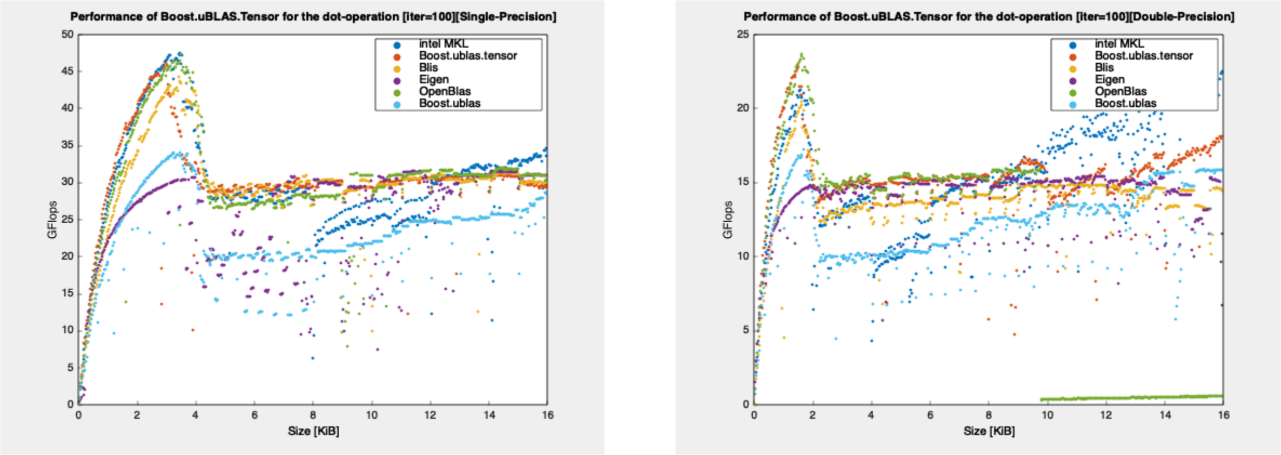
Table 2.2: Speedup Summary For Double-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	97
OpenBLAS	99	75
Eigen	99	97
Blis	99	98
Intel's MKL	88	1

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	0	0
Intel's MKL	9	0

2.3.2 Range[Start: 32, End: 16382, Step: 32]

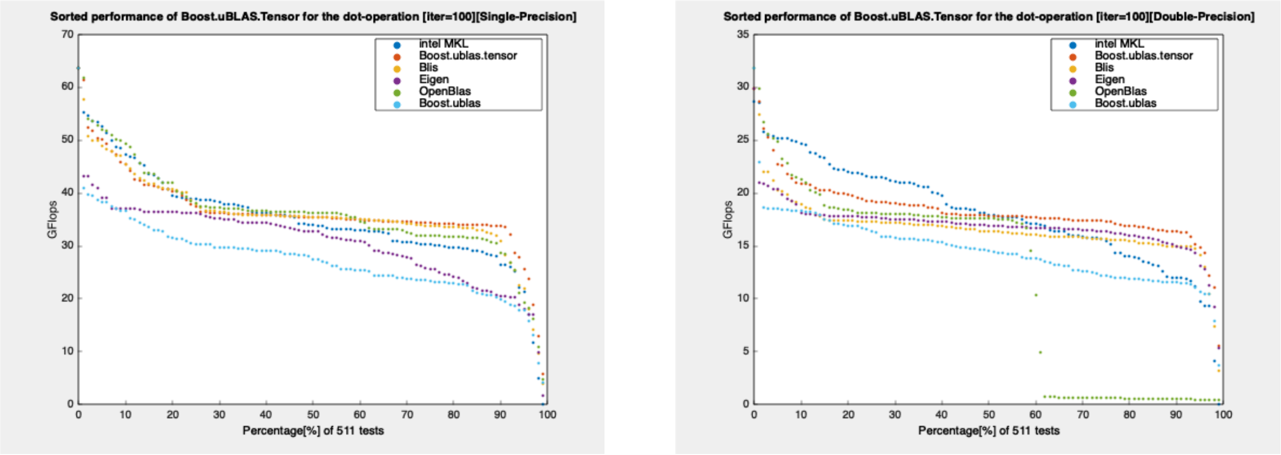
Performance measurements of ?dot implementations



(a) Single-Precision

(b) Double-Precision

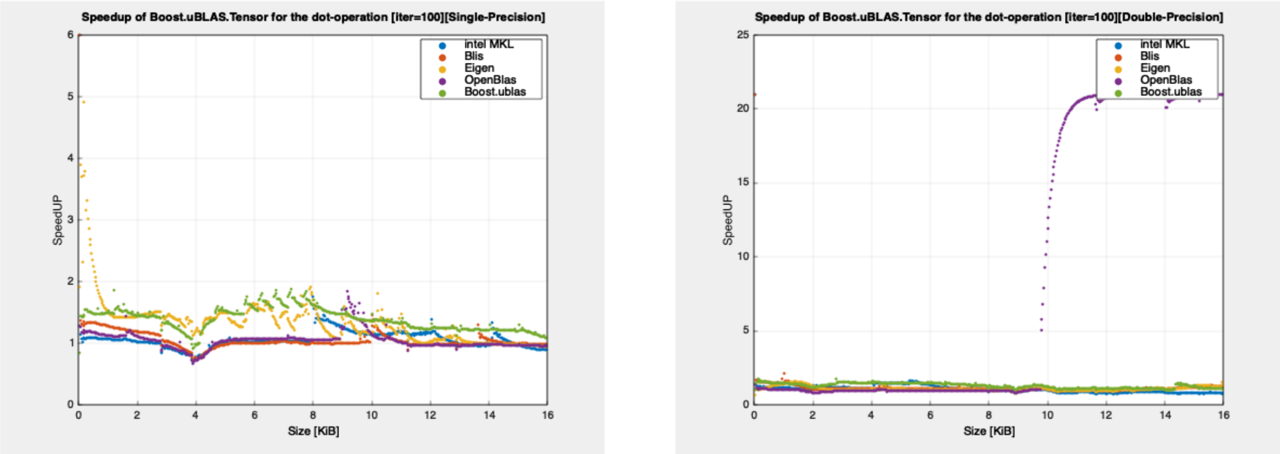
Sorted performance measurements of ?dot implementations



(a) Single-Precision

(b) Double-Precision

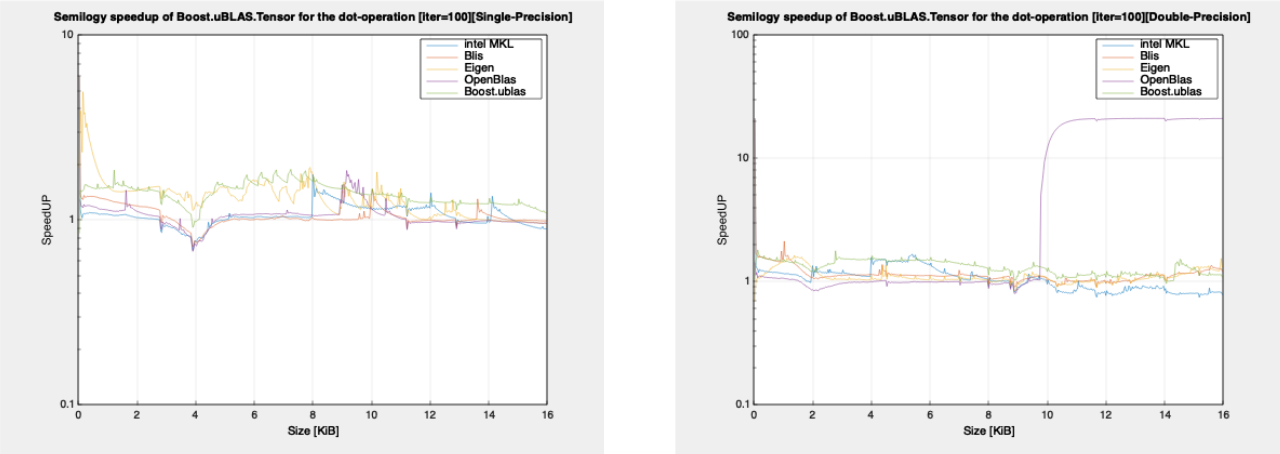
Comparison of the Boost.uBLAS.Tensor ?dot implementation



(a) Single-Precision

(b) Double-Precision

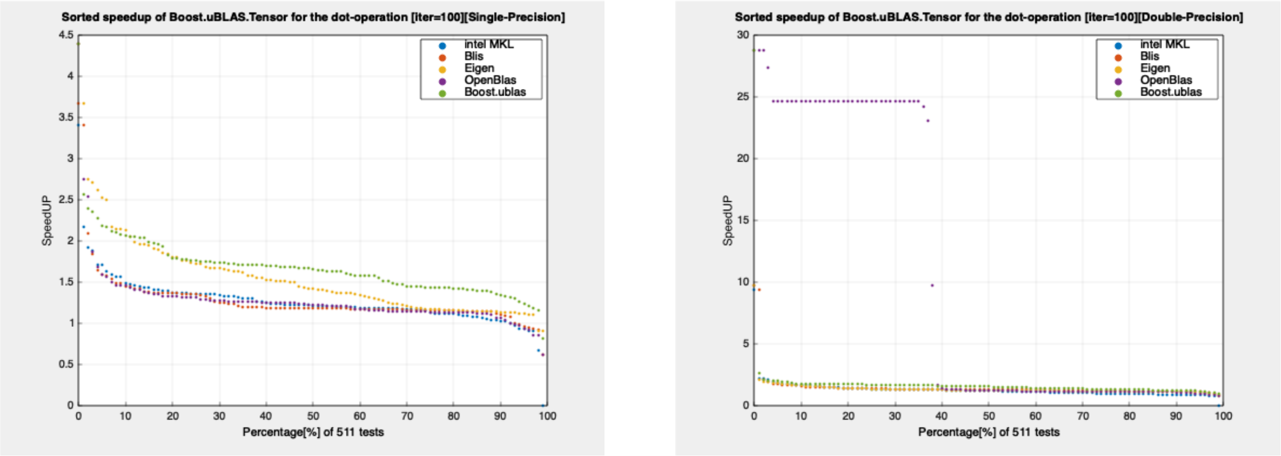
Comparison of the Boost.uBLAS.Tensor ?dot implementation [semilogy]



(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?dot implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 2.3: Speedup Summary For Single-Precision

Implementation	**Speedup ≥ 1** [%]	**Speedup ≥ 2** [%]
Boost.uBLAS	99	7
OpenBLAS	97	6
Eigen	99	13
Blis	98	1
Intel's MKL	94	5

Implementation	**Speed-down ≥ 1** [%]	**Speed-down ≥ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	1	0
Eigen	0	0
Blis	0	0
Intel's MKL	3	0

Table 2.4: Speedup Summary For Double-Precision

Implementation	**Speedup ≥ 1** [%]	**Speedup ≥ 2** [%]
Boost.uBLAS	99	97
OpenBLAS	99	38
Eigen	99	6
Blis	99	3
Intel's MKL	71	2

Implementation	**Speed-down ≥ 1** [%]	**Speed-down ≥ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	3	0
Eigen	0	0
Blis	0	0
Intel's MKL	26	0

2.4 Performance Metrics

Range[Start: 2, End: 2^{20}, Step: 1024]

Table 2.5: GFLOPS For Single-Precision

Implementation	Max	Average
Boost.uBLAS.Tensor	85.9582	54.7368
Boost.uBLAS	20.4992	13.2452
Intel's MKL	99.87	58.5317
OpenBLAS	36.2821	15.7624
Blis	40.9594	15.7895
Eigen	29.0721	13.0817

Table 2.6: GFLOPS For Double-Precision

Implementation	Max	Average
Boost.uBLAS.Tensor	48.9481	23.7548
Boost.uBLAS	11.586	5.58147
Intel's MKL	54.1943	25.1467
OpenBLAS	15.5243	10.1211
Blis	9.18747	6.43665
Eigen	9.7007	5.45126

Table 2.7: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	14.5989	9.29634
Boost.uBLAS	3.48153	2.24953
Intel's MKL	16.9616	9.94085
OpenBLAS	6.16204	2.67703
Blis	6.95641	2.68164
Eigen	4.93751	2.22176

Table 2.8: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	16.6264	8.06888
Boost.uBLAS	3.93547	3.93547
Intel's MKL	18.4084	8.54167
OpenBLAS	5.27318	3.43787
Blis	3.12074	2.18636
Eigen	3.29507	1.85165

Table 2.9: Speedup(Boost.uBLAS.Tensor) For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	3.08878	8.6454
Intel's MKL	0.931885	0.971519
OpenBLAS	1.76191	2.88331
Blis	2.21467	2.83157
Eigen	2.8202	7.68346

Table 2.10: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	5.58372	4.4268
Intel's MKL	0.95572	1.00273
OpenBLAS	2.57905	2.4175
Blis	5.12564	3.82981
Eigen	6.42996	4.38639

Chapter 3

Outer Product

The outer product is an expansion operation. Two vectors give a matrix containing the vectors' dimensions; the dimensions where the two vectors were having one dimension and transform it into two-dimension.

Let x and y be the vectors of length n and m respectively.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix},\ y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$A = x \otimes y^T \tag{3.1}$$

Or

$$A = x \otimes y^T + A \tag{3.2}$$

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \otimes \begin{bmatrix} y_0 & y_1 & \dots & y_n \end{bmatrix} = \begin{bmatrix} x_0 \times y_0 & x_0 \times y_1 & \dots & x_0 \times y_n \\ x_1 \times y_0 & x_1 \times y_1 & \dots & x_1 \times y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_n \times y_0 & x_n \times y_1 & \dots & x_n \times y_n \end{bmatrix}$$

Where A is the resultant matrix of dimensions n and m.

The routine **?ger** implements the equation 3.2, so we are doing the same. Once we implement the operation for one layout, another layout found using the exact implementation by rearranging the inputs. For example, if the implementation uses the column-major layout, then the row-major can be obtained by taking the matrix's transpose or exchanging the vectors

Taking the transpose on the both side in equation 3.2.

$$A^T = (x \otimes y^T + A)^T$$

Transpose on the matrix addition is distributive, so we get

$$A^T = (x \otimes y^T)^T + A^T$$

We can transpose inside the outer product, but the vectors exchange their position and transpose self-cancellation operation.

$$A^T = y \otimes x^T + A^T \tag{3.3}$$

If A is a column-major then A^T is must be row-major and vice-versa.

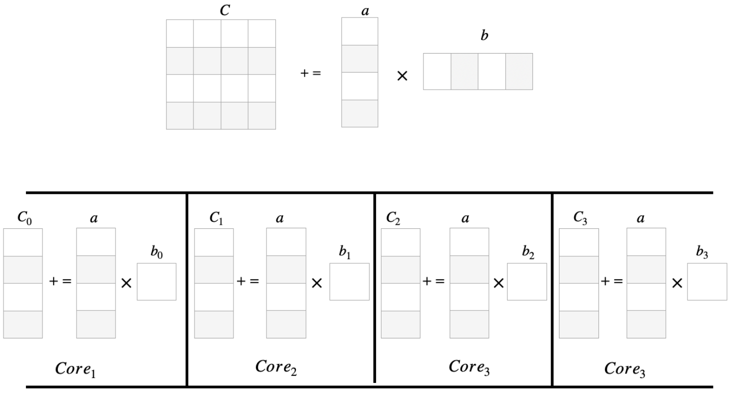
3.1 Calculating Number of Operations

Using equation 3.1, we fill the below table

Name	**Number**
Multiplication	$n \times m$
Addition	0

Total Number of Operations = Number of Multiplication + Number of Addition
Total Number of Operations = $n \times m + 0$
Total Number of Operations = $n \times m$

3.2 Algorithm



(a) Block Diagram

Algorithm 3: Outer Product SIMD Function

```
// $c$ is the pointer to the output matrix
// $a$ is the pointer to the first vector
// $b$ is the pointer to the second vector
// $n$ is the length of the vectors
Function outer_simd_loop (c, a, b, n):
    $cst \leftarrow b[0]$
    #pragma omp simd
    for $i \leftarrow 0$ to $n$ by 1 do
        $c[i] \leftarrow c[i] + a[i] \times cst$
    end
    return $sum$
end
```

Algorithm 4: Vector-Vector Outer Product

```
Input: $c, wc, a, na, b, nb, max\_threads$
// $a$ and $b$ are vectors
// $c$ is the pointer to the output matrix
// $na$ is the size of the vector $a$
// $nb$ is the size of the vector $b$
// $wc$ is the leading dimension of the matrix $c$
// $max\_threads$ is the user provided thread count
begin
    $MinSize \leftarrow 256$
    $number\_el\_L_2 \leftarrow \lfloor \frac{S_{L_2}}{S_{data}} \rfloor$
    $upper\_limit \leftarrow \lfloor \frac{number\_el\_L_2 - na}{na} \rfloor$
    $num\_threads \leftarrow max(1, min(upper\_limit, max\_threads))$
    $omp\_set\_num\_threads(num\_threads)$
    #pragma omp parallel for if($nb > MinSize$)
     for $i \leftarrow 0$ to $nb$ by $1$ do
        $aj \leftarrow a$
        $bj \leftarrow b + i$
        $cj \leftarrow b + i \times wc$
        $outer\_simd\_loop(cj,\ aj,\ bj,\ na)$
    end
end
```

Here, we start threads if the second vector's size exceeds 256 because to avoid thread overhead for a small vector. The number is not concrete, so it can be any value until it small enough, or we can remove it. There is a problem we have to give a thought about and solve. This problem arises when multiple threads try to fetch the data, and if the data not found, it will evict the cached data using LRU policy. If the evicted data is still in use and it does not find the data will again evict and may or may not propagate to the other cores.

To avoid the cache eviction problem, we decrease the threads spawned if the data cannot fit inside the cache.

Let the length of the first vector be n and a small chunk of the second vector be m_c.

S_{L_2} = a block of matrix + length of the first vector

We are not trying to fit the second vector because each core can put the element from the second vector inside the register and access each column of the resultant matrix and the whole first vector. This algorithm is performing a rank-1 update in each core.

$$S_{L_2} = S_{data}(n \times m_c + n)$$
$$n \times m_c = \frac{S_{L_2}}{S_{data}} - n$$

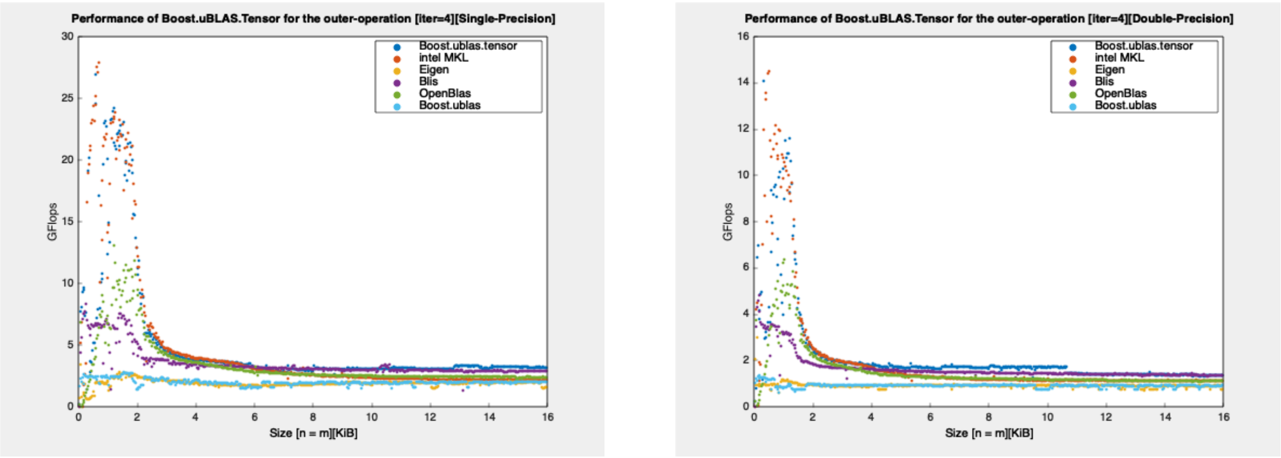
$$m_c = \frac{S_{L_2}}{S_{data} \times n} - 1 \tag{3.4}$$

As the n increase, the block m_c decreases. We control the second vector's blocks through the outer loop, where we divide the second vector through threads, and each thread contains a single element. Therefore, the m_c block drops below the maximum allowed threads, we start to spawn m_c amount of threads.

$$num\_threads = max(1, min(m_c, max\_threads)) \tag{3.5}$$

3.3 Performance Plots and Speedup Summary

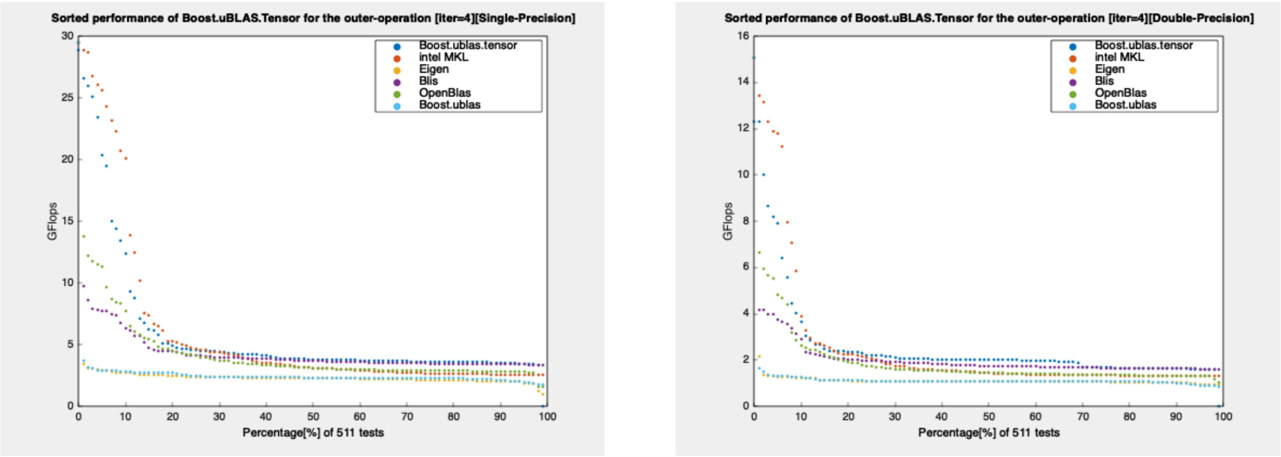
Performance measurements of ?ger implementations



(a) Single-Precision

(b) Double-Precision

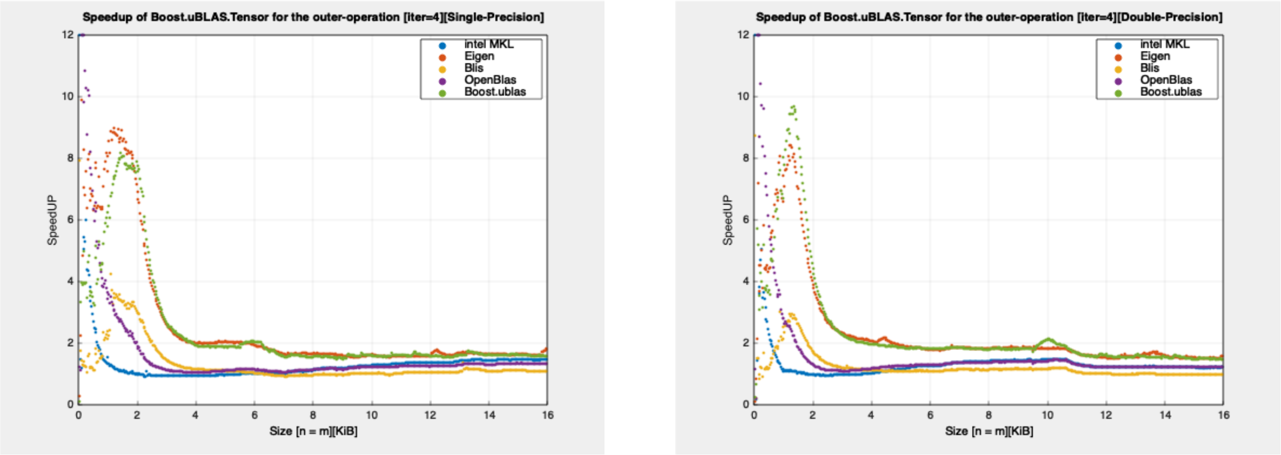
Sorted performance measurements of ?ger implementations



(a) Single-Precision

(b) Double-Precision

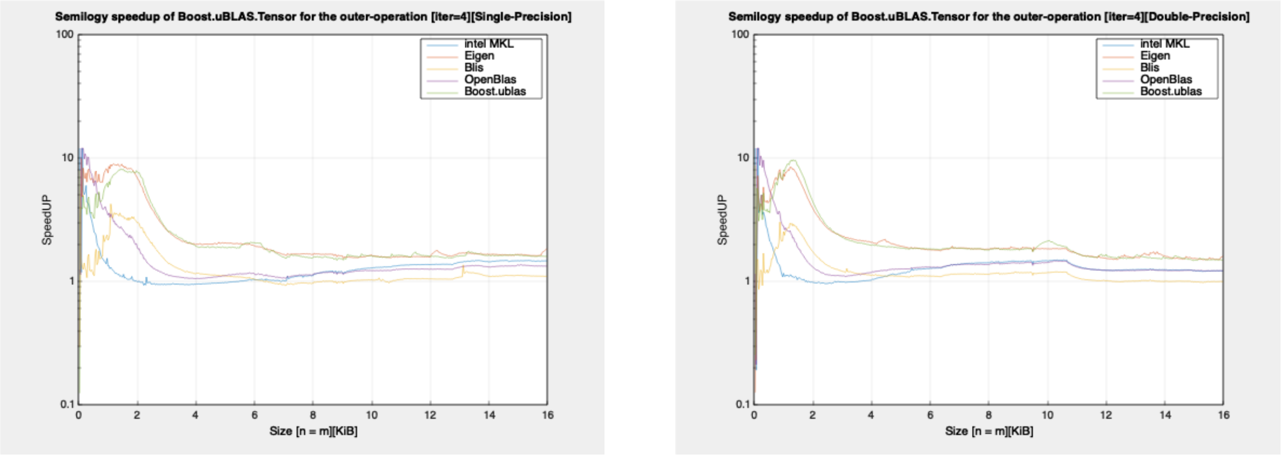
Comparison of the Boost.uBLAS.Tensor ?ger implementation



(a) Single-Precision

(b) Double-Precision

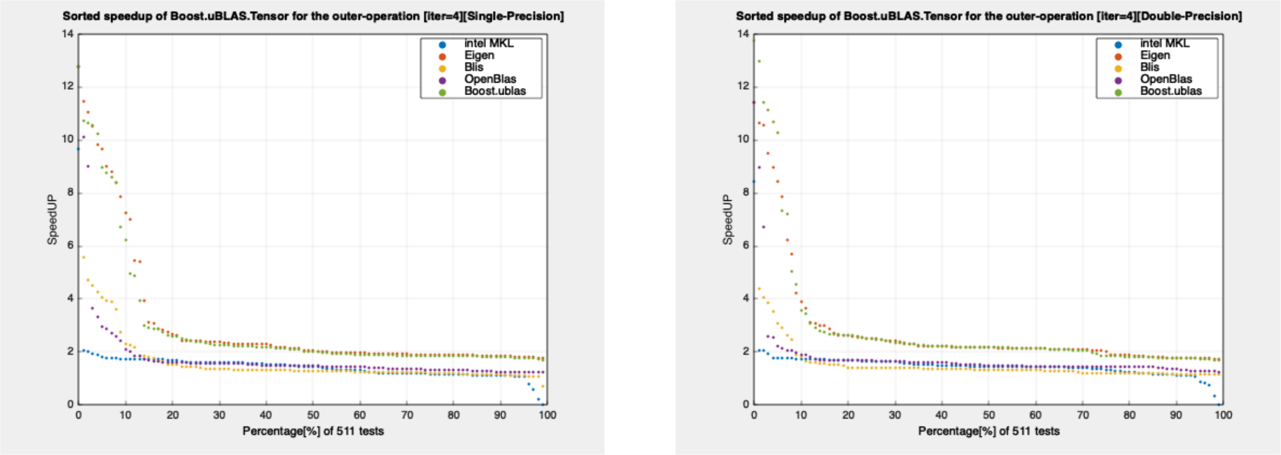
Comparison of the Boost.uBLAS.Tensor ?ger implementation [semilogy]



(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?ger implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 3.1: Speedup Summary For Single-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
Boost.uBLAS	99	50
OpenBLAS	99	11
Eigen	99	52
Blis	98	12
Intel's MKL	95	2

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	0	0
Intel's MKL	2	0

Table 3.2: Speedup Summary For Double-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
Boost.uBLAS	99	71
OpenBLAS	99	8
Eigen	99	74
Blis	99	8
Intel's MKL	94	2

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	0	0
Intel's MKL	3	0

3.4 Performance Metrics

Range[Start: 32**, End:** 16382**, Step:** 32]

Table 3.3: GFLOPS For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	26.9675	5.00698
Boost.uBLAS	2.82578	2.01385
Intel's MKL	27.9	4.89873
OpenBLAS	13.1052	3.4242
Blis	8.35253	3.52469
Eigen	5.24754	1.84399

Table 3.4: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	14.1047	2.17063
Boost.uBLAS	1.34503	0.925927
Intel's MKL	14.4913	2.0621
OpenBLAS	6.35947	1.54396
Blis	4.81067	1.66087
Eigen	3.01793	0.926035

Table 3.5: Utilization[%] For Single-Precision

Implementation	Max	Average
Boost.uBLAS.Tensor	4.58008	0.850371
Boost.uBLAS	0.479922	0.342027
Intel's MKL	4.73845	0.831985
OpenBLAS	2.22575	0.581555
Blis	1.41857	0.598622
Eigen	0.580557	0.328019

Table 3.6: Utilization[%] For Double-Precision

Implementation	Max	Average
Boost.uBLAS.Tensor	4.79099	0.737305
Boost.uBLAS	0.456872	0.314513
Intel's MKL	4.92232	0.700442
OpenBLAS	2.16015	0.524444
Blis	1.63406	0.564155
Eigen	1.02511	0.31455

Table 3.7: Speedup(Boost.uBLAS.Tensor) For Single-Precision

Implementation	Max	Average
Boost.uBLAS	9.54339	2.48627
Intel's MKL	0.966577	1.0221
OpenBLAS	2.05777	1.46224
Blis	3.22867	1.42055
Eigen	7.88912	2.59245

Table 3.8: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	Max	Average
Boost.uBLAS	10.4865	2.34428
Intel's MKL	0.973321	1.05263
OpenBLAS	2.2179	1.40588
Blis	2.93196	1.30692
Eigen	4.67363	2.344

Chapter 4

Matrix-Vector Product

When we take the product of the matrix and the vector, it will result in a vector. The product is a contraction operation, which means that the matrix's dimension reduced from two to one. To apply the contraction, then the length of one of the matrix dimensions must be the same as the vector's length.

Let the A be the matrix with dimensions m and n, and the length of the vector x be n.

The resultant vector y will have the dimension m.

$$A = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mn} \end{bmatrix}, x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$y = Ax + y \tag{4.1}$$

$$Av = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} a_{0i}x_i \\ \sum_{i=0}^{n} a_{1i}x_i \\ \vdots \\ \sum_{i=0}^{n} a_{mi}x_i \end{bmatrix}$$

The vector times matrix can calculated by taking the transpose on the both sides.

$$\begin{aligned} y^T &= (Ax + y)^T \\ y^T &= (Ax)^T + y^T \\ y^T &= x^T A^T + y^T \end{aligned}$$

The column-major and row-major layout for the vector in the memory is non-distinguishable. Hence, we can use this fact and we get $x = x^T$. If A is the column-major layout then A^T is the row-major layout and vice-versa.

$$y = xA^T + y \tag{4.2}$$

4.1 Calculating Number of Operations

Using equation 4.1 or 4.2, we fill the below table

Name	Number
Multiplication	m or n
Addition	$m-1$ or $n-1$

$$\text{Total Number of Operations} = (\text{Number of Multiplication} + \text{Number of Addition}) \times \begin{cases} n \\ m \end{cases}$$

$$\text{Total Number of Operations} = \begin{cases} n \times (m + m - 1) \\ m \times (n + n - 1) \end{cases}$$

$$\text{Total Number of Operations} = \begin{cases} n \times (2m - 1) \\ m \times (2n - 1) \end{cases}$$

4.2 Algorithm

The matrix-vector product has two different algorithms: Column-Major and Row-Major. We need to handle two different layouts differently. The Row-Major has the most straightforward algorithm because the row elements lay contiguously in the memory, which improves the cache locality compared against Column-Major, where the row elements placed with a specific stride and hinder the cache locality.

4.2.1 Column-Major

Algorithm 5: Matrix-Vector Product SIMD Function

```
// $c$ is the pointer to the output vector
// $a$ is the pointer to the input matrix
// $b$ is the pointer to the input vector
// $block$ is the block of the output vector
// $w$ is the leading dimension of the matrix
// $simd\_loop_N$ is a function where N signifies the amount of the unrolling done
Function $simd\_loop_N$ (c, a, b, block, w):
    #pragma omp simd
    for $i \leftarrow 0$ to $block$ by 1 do
        $c[i] \leftarrow c[i] + A[i + w * 0] * b[0]$;
        $c[i] \leftarrow c[i] + A[i + w * 1] * b[1]$;
                    ⋮
        $c[i] \leftarrow c[i] + A[i + w * N] * b[N]$;
    end
end
```

Algorithm 6: Matrix-Vector Product Loop

```
// $c$ is the pointer to the output vector
// $a$ is the pointer to the input matrix
// $b$ is the pointer to the input vector
// $n_a$ is the size of the column of the matrix $a$
// $w_a$ is the leading dimension of the matrix $a$
// $n_b$ is the size of the vector $b$
// $block$ is the block of the output vector
// $main\_loop_N$ is a function where N signifies the amount of the unrolling done
Function $main\_loop_N$ ($c$, $a$, $n_a$, $w_a$, $b$, $n_b$, $block$):
    if $n_b == 0$ then
        return;
    end
    $ai \leftarrow a$
    $bi \leftarrow b$
    $ci \leftarrow c$
    #pragma omp for schedule(dynamic)
    for $i \leftarrow 0$ to $n_a$ by $block$ do
        $aj \leftarrow ai + i$
        $bj \leftarrow bi$
        $cj \leftarrow ci + i$
        $ib \leftarrow min(block, na - i)$
        for $j \leftarrow 0$ to $n_b$ by $1$ do
            $ak \leftarrow a + j \times N \times wa$
            $bk \leftarrow bk + j \times N$
            $ck \leftarrow cj$
            $simd\_loop_N(ck, ak, bk, ib, w_a)$
        end
    end
end
```

Algorithm 7: Col-Major Matrix-Vector Product

```
Input: $c$, $a$, $n_a$, $w_a$, $b$, $n_b$, $max\_threads$
// $c$ is the pointer to the output vector
// $a$ and $b$ are pointer to the matrix
// $b$ are pointer to the vector
// $n_a$ is the size of the column of the matrix $a$
// $w_a$ is the leading dimension of the matrix $a$
// $n_b$ is the size of the vector $b$
// $max\_threads$ is the user provided thread count
begin
    $omp\_set\_num\_threads(max\_threads)$
    $number\_el\_L_1 \leftarrow \lfloor \frac{S_{L_1}}{S_{data}} \rfloor$
    $half\_block \leftarrow \lfloor \frac{number\_el\_L_1}{2} \rfloor$
    $max\_size \leftarrow max(na, nb)$
    $small\_block \leftarrow 2^{\lfloor \frac{k}{2} \rfloor}$ // Where $2^k$ is a power of two representation of $number\_el\_L_1$
    $k\_factor \leftarrow nearest\_power\_of\_2(\frac{number\_el\_L_1}{\log_2 small\_block})$
    $block_2 \leftarrow max(1, \frac{small\_size}{2^{\lfloor \frac{max\_size}{k\_factor} \rfloor}})$
    if $n_a > number\_el\_L_1$ then
        $block_1 \leftarrow half\_block$
    end
    else
        $block_1 \leftarrow small\_block$
    end
    $n_{itr} \leftarrow \lfloor \frac{n_b}{block_2} \rfloor$
    $n_{rem} \leftarrow n_b - (block_2 \times \lfloor \frac{n_b}{block_2} \rfloor)$ // This equation gives the remainder
    #pragma omp parallel
     begin
        $main\_loop_1(c, a, w_a, n_a, b, n_{rem}, block_1)$
        $at \leftarrow a + w_a \times n_{rem}$
        $bt \leftarrow b + n_{rem}$
        $main\_loop_{block_2}(c, at, w_a, n_a, bt, n_{iter}, block_1)$
    end
end
```

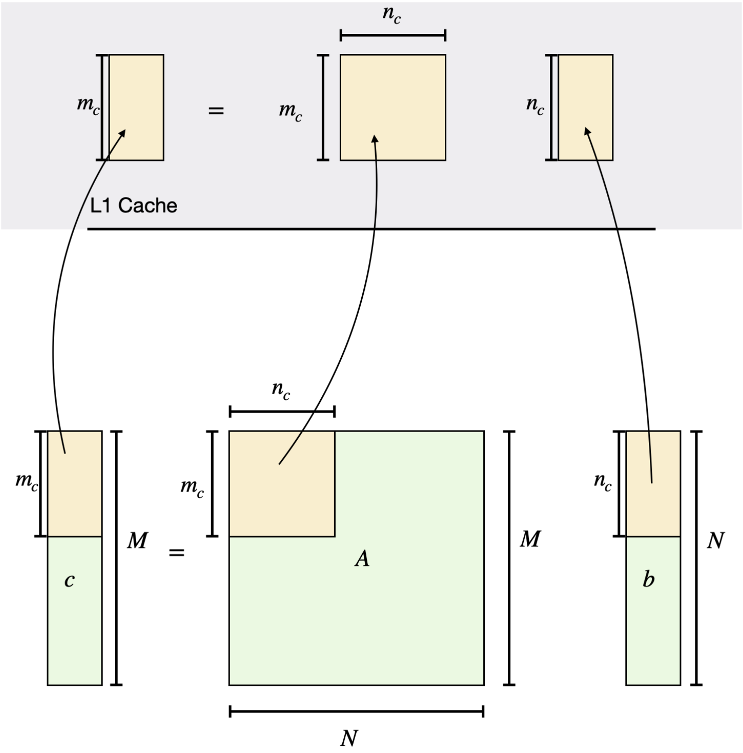
The Column-Major product is quite tricky and complicated if we cannot control the instruction set. The row elements are x strides away from each other and paralyse the cache predictor or cache locality. The cache predictor brings the cache line size elements from memory, which are place contiguously. Each element in the result vector is the product of the matrix's row vector and the whole vector.

For OpenMP to do its magic, we need to fill the L_1 cache with the result vector block, block of the input vector's matrix and block. We divide the load through the threads, where each thread has its private cache, and they calculate a small block of the result vector without hindering the other vectors. The result vector calculation is done by dividing it into smaller chunk or block and handing over the block to each thread, which removes the dependency within the different blocks and eradicates the data races and synchronization. The loop which manages the block for the result vectors must be multithreaded, and each thread needs to fill the L_1 cache with the output vector and a column vector of the matrix as large as it can so that the registers can fetch the blocks as fast as possible from the cache rather than fetching from memory, which adds an overhead to the performance. To fit the output vector, the matrix and the vector, we previously discussed fact and the 4.1 for the variable convention; we derive the following equation.

$$S_{L_1} = S_{data}(m_c n_c + m_c + n_c) \tag{4.3}$$

For the small length of the result vector, which is less than the L1 cache size, to maximize the block in the L1 cache, the row block and column block must be identical, $m_c = n_c$, which makes the matrix a square matrix.

Figure 4.1: Column-Major Matrix-Vector Block Diagram



Let $m_c = n_c = B_{small}$

$$\begin{aligned} S_{L_1} &= S_{data}(m_c n_c + m_c + n_c) \\ S_{L_1} &= S_{data}(B_{small} \times B_{small} + B_{small} + B_{small}) \\ S_{L_1} &= S_{data}(B_{small}^2 + 2B_{small}) \end{aligned}$$

$$B_{small}^2 + 2B_{small} = \frac{S_{L_1}}{S_{data}} \tag{4.4}$$

B_{small}^2 is much larger than the $2B_{small}$ in 4.4, so we can ignore the smaller term.

$$B_{small}^2 < \frac{S_{L_1}}{S_{data}}$$

We can represent $\frac{S_{L_1}}{S_{data}}$ in form of power of two because we know both S_{L_1} and S_{data} are power of two. Therefore, the division must produce a power of two. Let the exponent of two be k, which gives us

$$\begin{aligned} B_{small}^2 &< 2^k \\ B_{small} &< 2^{\frac{k}{2}} \end{aligned}$$

Here, k can be both even or odd. We will have a fractional value if k is odd, and we cannot have a fractional value because we have no control over the registers, and registers are always discrete, which is a multiple of a power of two. We try to maximize the register use, and if the value is fractional, it will not fill the registers fully, and it is a wastage of processing power. There might be a case when the register has only one element, and subsequent iteration will have to fill the register with more elements; we could have filled the register with the following elements and had processed more elements rather than waiting for subsequent iteration, which is a wastage of cycles.

$$\begin{aligned} 2^{\lfloor \frac{k}{2} \rfloor} \leq 2^{\frac{k}{2}} \leq 2^{\lceil \frac{k}{2} \rceil} \\ \lfloor \frac{k}{2} \rfloor \leq \frac{k}{2} \leq \lceil \frac{k}{2} \rceil \end{aligned}$$

using this relation and taking the lower bound, we have

$$B_{small} \leq 2^{\lfloor \frac{k}{2} \rfloor} \tag{4.5}$$

The eqauation 4.5 is only valid if the length of the output vector is less than the L_1 cache.

B_{small} is the lower bound, so now, we need to find the upper bound. To find the upper bound, we will use the fact that each thread has its private cache, an L_1 cache. Using the equation 4.3, we can maximize the output vector's block by putting an input vector element into the register, making the block of the input vector one, m_c by putting an input vector element into the register, which makes the block of the input vector one, $n_c = 1$. We do not need n_c in the L_1 cache, so we remove it from the equation 4.3.

$$\begin{aligned} S_{L_1} &= S_{data}(m_c n_c + m_c) \\ S_{L_1} &= S_{data}(m_c + m_c) \\ S_{L_1} &= S_{data}(2m_c) \\ m_c &= \frac{S_{L_1}}{2S_{data}} \end{aligned}$$

When $M < \frac{S_{L_1}}{S_{data}}$ then $m_c = B_{small}$, otherwise, $m_c = \frac{S_{L_1}}{2S_{data}}$.

Now, we need to find block n_c, where n_c is an unrolling factor. The unrolling factor helps to bring the block of column vectors in the cache, and the block is the size of the cache line. If the cache line size and m_c are of the same size, then the cache misses minimized. We found that n_c is a decreasing value through experiments and depends on the maximum dimension, where it reaches the minimum value of one when the maximum dimension is equal to or exceeds the L_1 cache size.

We see this behaviour due to the cache misses as the dimension increase in size, the block n_c starts to replace the block of output vector m_c using the LRU policy, so on the subsequent iteration, the replaced block of the output m_c again needs to bring back to the L_1 cache.

Let cm_X be the cache miss where X is the size of the block. For the large block of n_c and the large dimension.

$Total\ Cache\ Misses$ = cache miss for bringing the block m_c of the output into the cache + cache miss for bringing the block $m_c \times n_c$ of the matrix A into the cache + cache miss for bringing the block $m_c - Size(Register)$ of the output into the cache back which was replaced due to eviction + cache miss for bringing the block n_c of the input into the cache.

$$Total\ Cache\ Misses = cm_{m_c} + cm_{m_c n_c} + cm_{m_c - Size(Register)} + cm_{n_c}$$

If the $n_c = 1$ for the large length then $cm_{n_c} = 0$ because it will reside inside the register and there is no cache eviction. Therefore, $cm_{m_c - Size(Register)} = 0$, so we have

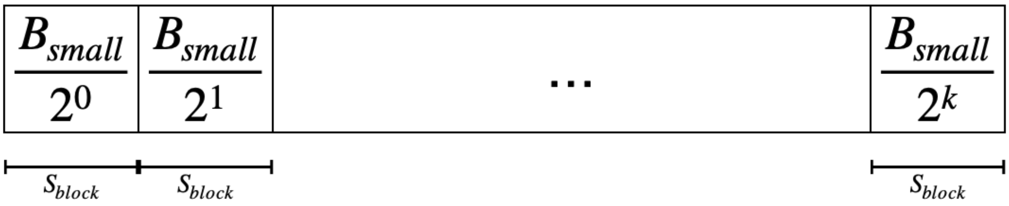
$$\begin{aligned} Total\ Cache\ Misses &= cm_{m_c} + cm_{m_c} \\ Total\ Cache\ Misses &= 2cm_{m_c} \end{aligned}$$

Using this fact, we can deduce

$$1 \leq n_c \leq B_{small}$$

Since n_c is a decreasing value and we decrease n_c by the power of two because we need to use the register efficiently. Let S_{max} be the maximum dimension of the matrix.

Figure 4.2: Input Vector Block Divison Diagram



Using the previously derived facts, we know $\frac{B_{small}}{2^k} = 1$ when $\frac{S_{L_1}}{S_{data}} \leq S_{max}$

$$\frac{B_{small}}{2^k} = 1$$

$$B_{small} = 2^k$$

$$k = \log_2 B_{small}$$

$$kS_{block} = \frac{S_{L_1}}{S_{data}}$$

$$S_{block} = \frac{S_{L_1}}{kS_{data}}$$

$$S_{block} = \frac{S_{L_1}}{S_{data} \log_2 B_{small}} \tag{4.6}$$

If the S_{block} is not the power of two then get the nearest power of two.

$$2^q \leq S_{block} \leq 2^{q+1}$$

Therefore,

$$Block_2 = max(1, \frac{B_{small}}{2^{\lfloor \frac{S_{max}}{S_{block}} \rfloor}}) \tag{4.7}$$

4.2.2 Row-Major

Algorithm 8: Matrix-Vector Product SIMD Function

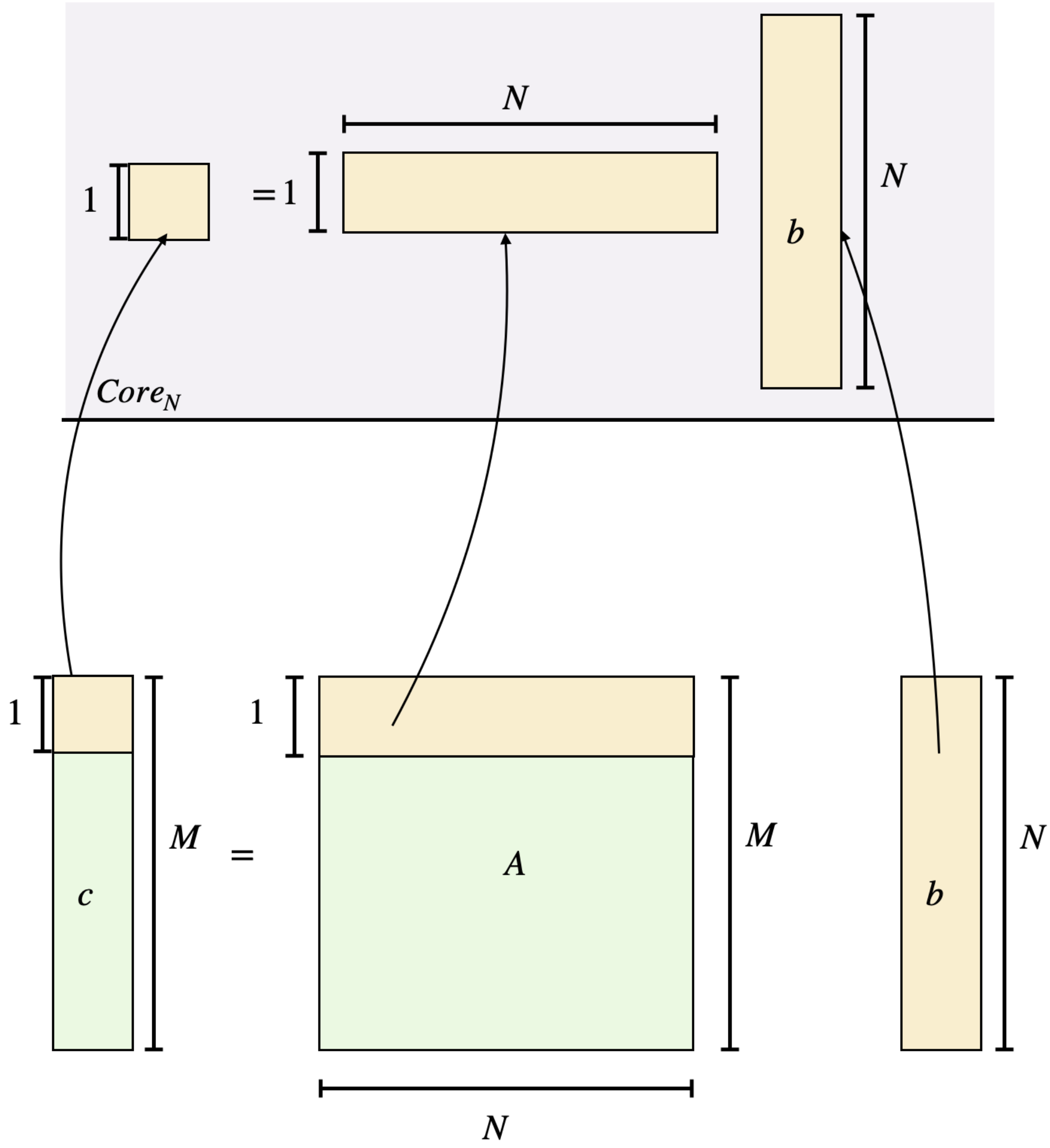
```
// $a$ is the pointer to the input matrix
// $b$ is the pointer to the input vector
// $n_b$ is the block of the output vector
Function simd_loop (a, b, $n_b$):
    $sum \leftarrow 0$
    #pragma omp simd
     for $i \leftarrow 0$ to $n_b$ by $1$ do
        $sum \leftarrow sum + a[i] \times b[i]$
    end
    return $sum$
end
```

Algorithm 9: Row-Major Matrix-Vector Product

```
Input: $c$, $a$, $n_a$, $w_a$, $b$, $n_b$, $max\_threads$
// $c$ is the pointer to the output vector
// $a$ and $b$ are pointer to the matrix
// $b$ are pointer to the vector
// $n_a$ is the size of the column of the matrix $a$
// $w_a$ is the leading dimension of the matrix $a$
// $n_b$ is the size of the vector $b$
// $max\_threads$ is the user provided thread count
begin
    $omp\_set\_num\_threads(max\_threads)$
    $ai \leftarrow a$
    $bi \leftarrow b$
    $ci \leftarrow c$
    #pragma omp parallel for schedule(static)
     for $i \leftarrow 0$ to $n_a$ by $1$ do
        $aj \leftarrow ai + i * w_a$
        $bj \leftarrow bi$
        $cj \leftarrow ci + i$
        $cj \leftarrow simd\_loop(aj, bj, n_b)$
    end
end
```

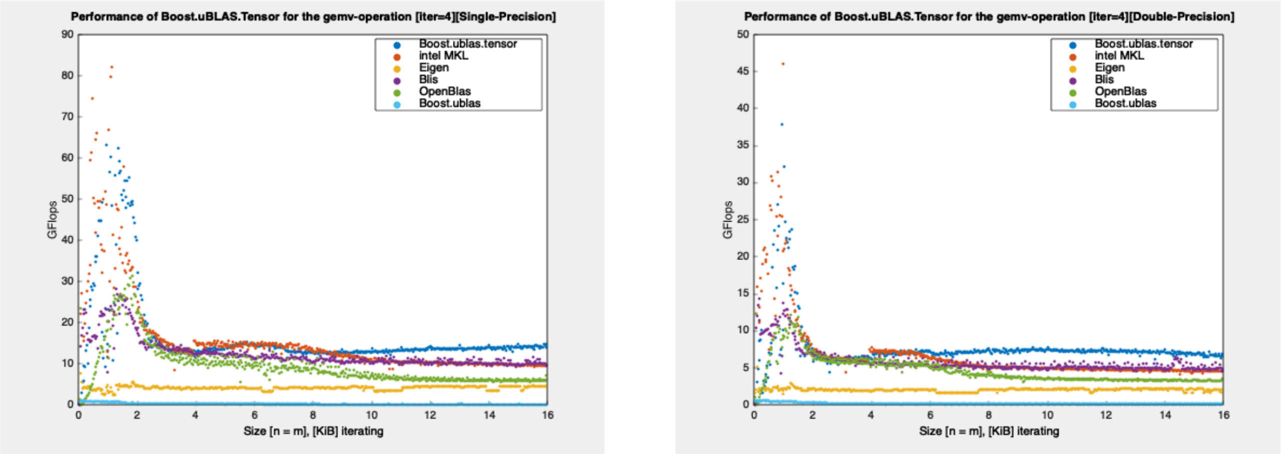
The Row-Major product is the simplest one because the row elements are contiguous in memory; therefore, each thread assigned to each row vector, and they are responsible for each element of the output vector. The distribution of the row-vector to the threads simplifies the problem to the vector-vector inner product. The inner product of the row-vector and the input vector gives an element of the output vector.

Figure 4.3: Row-Major Matrix-Vector Block Diagram



4.3 Performance Plots and Speedup Summary For Column-Major

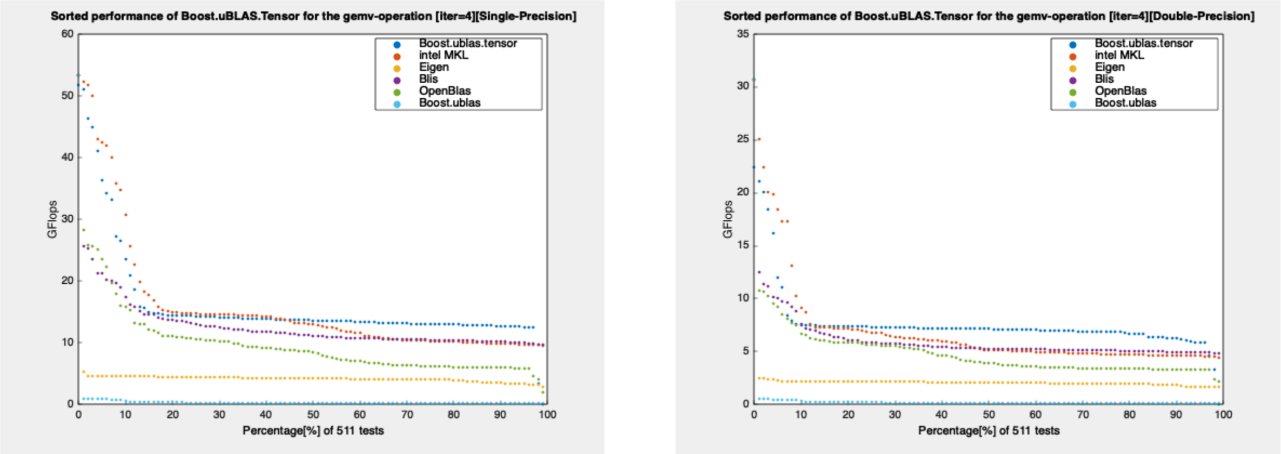
Performance measurements of ?gemv implementations



(a) Single-Precision

(b) Double-Precision

Sorted performance measurements of ?gemv implementations



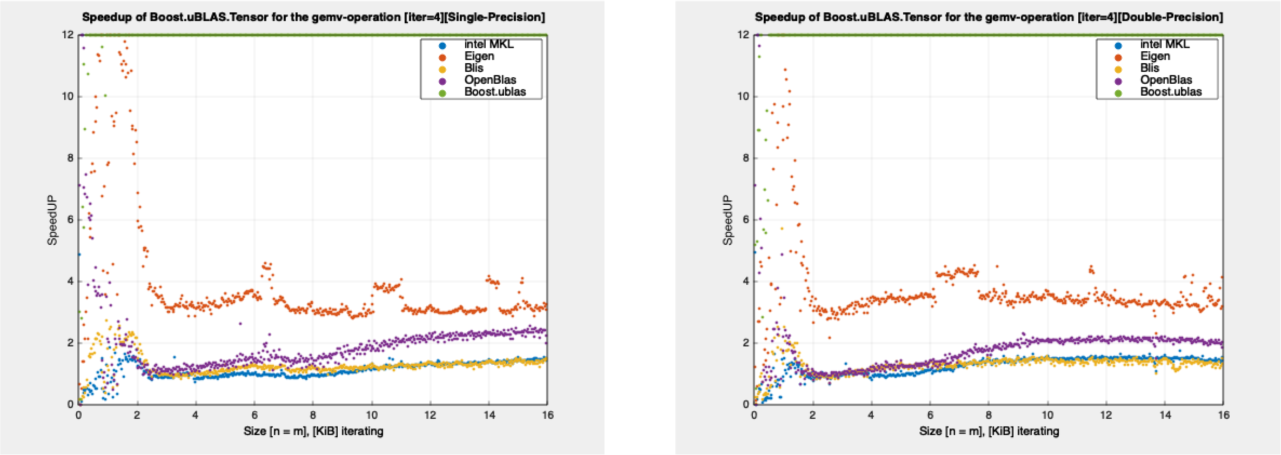
(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?gemv implementation

(a) Single-Precision

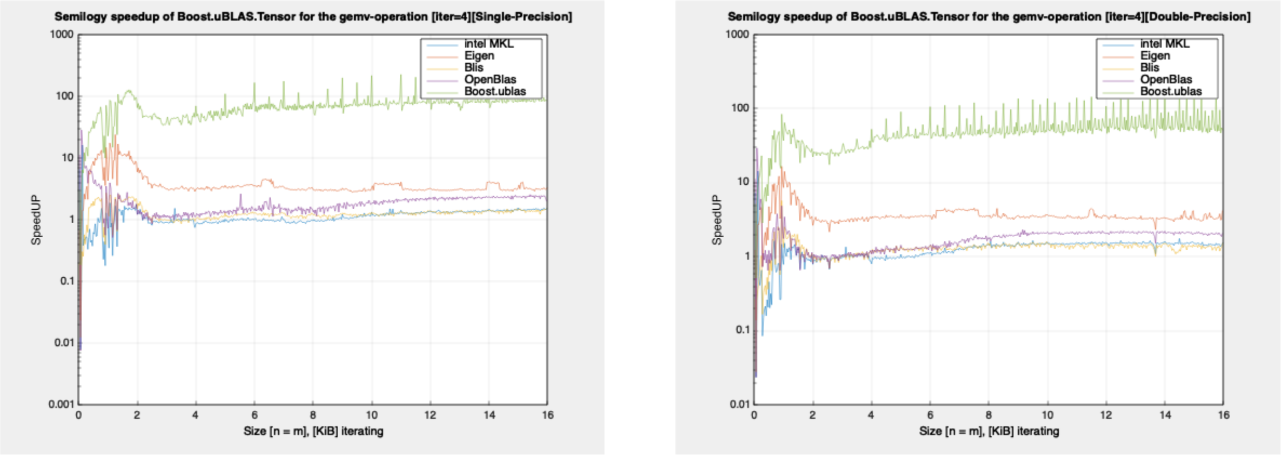
(b) Double-Precision



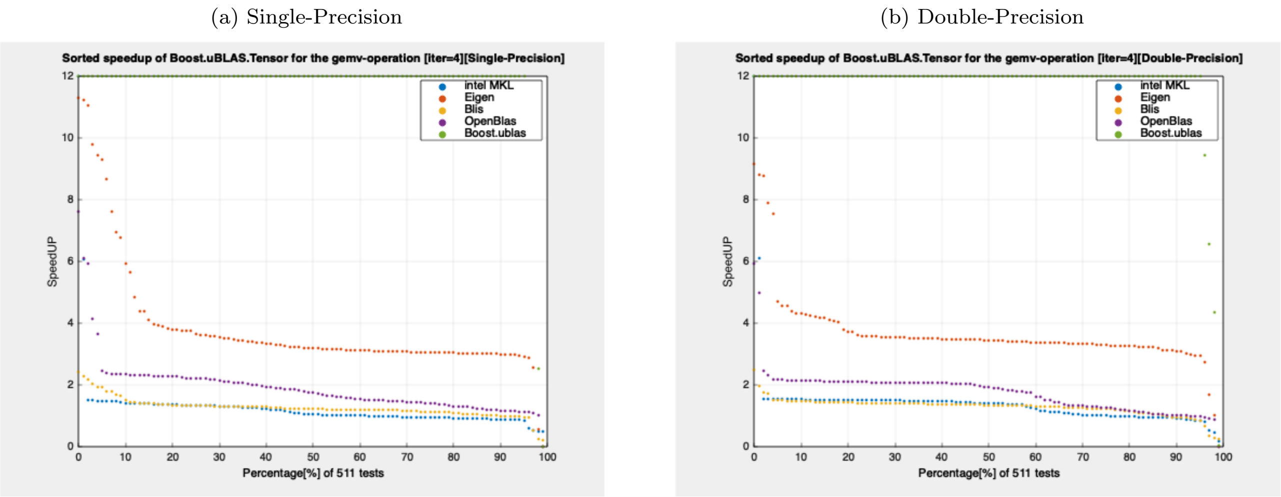
Comparison of the Boost.uBLAS.Tensor ?gemv implementation [semilogy]

(a) Single-Precision

(b) Double-Precision



Comparison of the Boost.uBLAS.Tensor ?gemv implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 4.1: Speedup Summary For Single-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
Boost.uBLAS	97	97
OpenBLAS	98	38
Eigen	97	97
Blis	98	98
Intel's MKL	61	1

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	1	1
Blis	7	1
Intel's MKL	37	0

Table 4.2: Speedup Summary For Double-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
Boost.uBLAS	98	98
OpenBLAS	92	47
Eigen	98	96
Blis	87	0
Intel's MKL	76	1

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
Boost.uBLAS	0	0
OpenBLAS	6	0
Eigen	0	0
Blis	11	2
Intel's MKL	22	1

4.4 Performance Metrics For Column-Major

Range[Start: 32, **End:** 16382, **Step:** 32]

Table 4.3: GFLOPS For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	63.1299	16.3994
Boost.uBLAS	1.04371	0.275396
Intel's MKL	82.2283	15.6702
OpenBLAS	31.4436	9.49543
Blis	28.142	12.4837
Eigen	7.59804	4.15489

Table 4.4: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	37.9093	7.66141
Boost.uBLAS	0.585254	0.184873
Intel's MKL	46.0515	6.78875
OpenBLAS	12.3714	4.69513
Blis	14.4262	5.90106
Eigen	5.1347	2.0504

Table 4.5: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	10.7218	2.78522
Boost.uBLAS	0.17726	0.0467724
Intel's MKL	13.9654	2.66137
OpenBLAS	5.34028	1.61268
Blis	4.77956	2.12019
Eigen	1.29043	0.705654

Table 4.6: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	12.8768	2.60238
Boost.uBLAS	0.198795	0.0627965
Intel's MKL	15.6425	2.30596
OpenBLAS	4.20224	1.59481
Blis	4.90021	2.00443
Eigen	1.74412	0.696469

Table 4.7: Speedup(Boost.uBLAS.Tensor) For Single-Precision

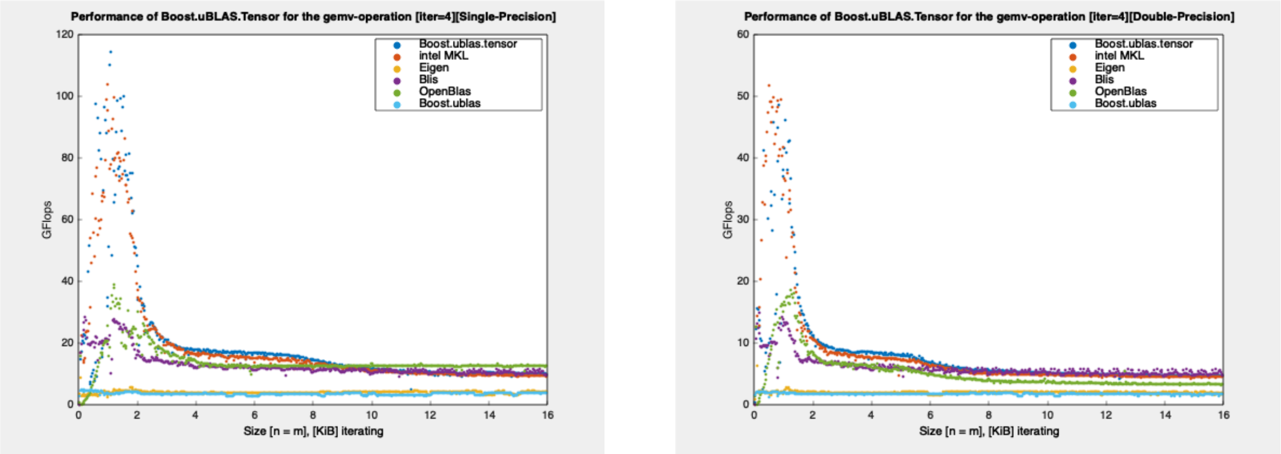
Implementation	**Max**	**Average**
Boost.uBLAS	60.4862	59.5483
Intel's MKL	0.767739	1.04653
OpenBLAS	2.00772	1.72708
Blis	2.24326	1.31366
Eigen	8.30871	3.947

Table 4.8: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	64.7741	41.4415
Intel's MKL	0.823193	1.12854
OpenBLAS	3.06427	1.63178
Blis	2.6278	1.29831
Eigen	7.38297	3.73654

4.5 Performance Plots and Speedup Summary For Row-Major

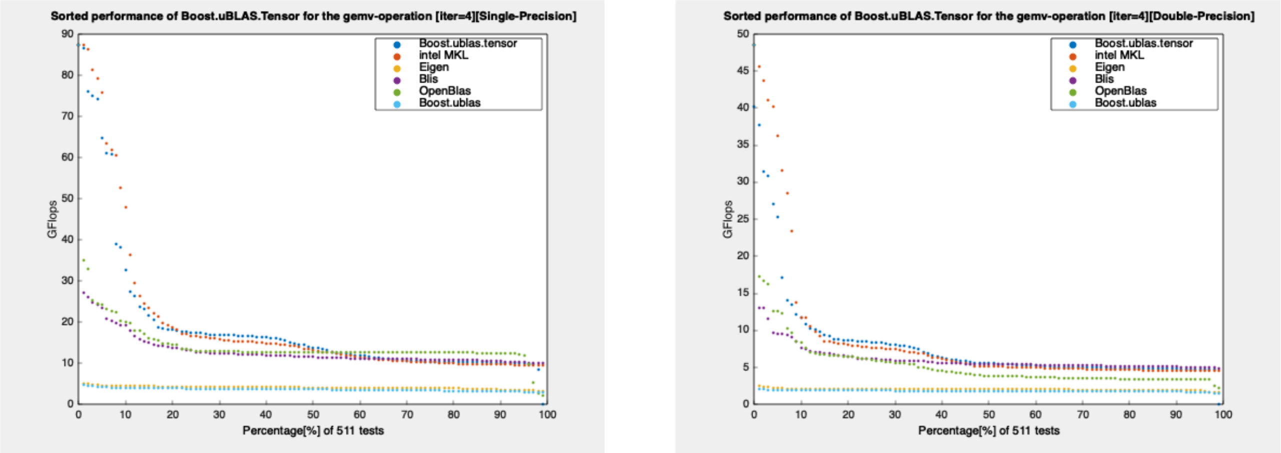
Performance measurements of ?gemv implementations



(a) Single-Precision

(b) Double-Precision

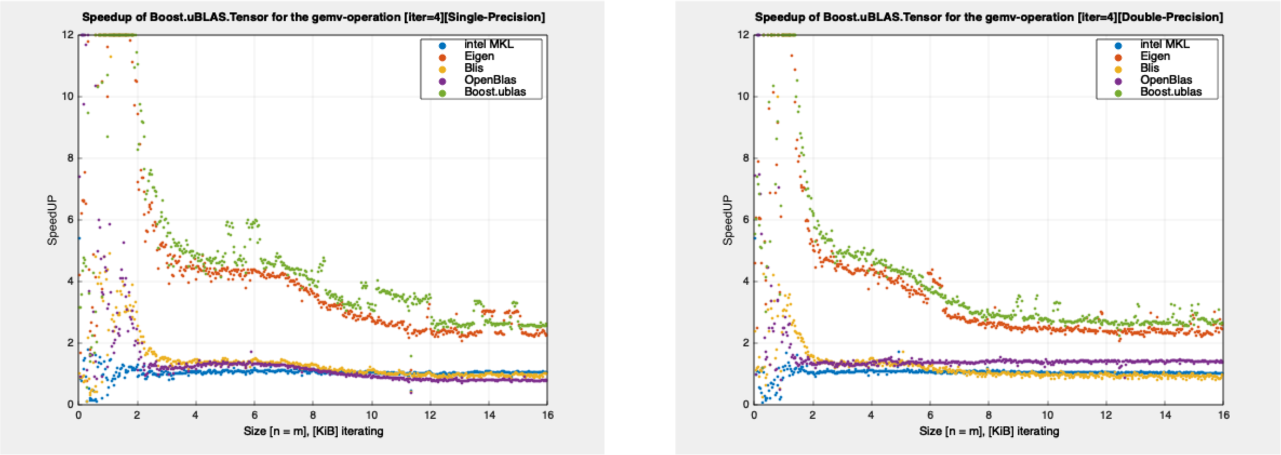
Sorted performance measurements of ?gemv implementations



(a) Single-Precision

(b) Double-Precision

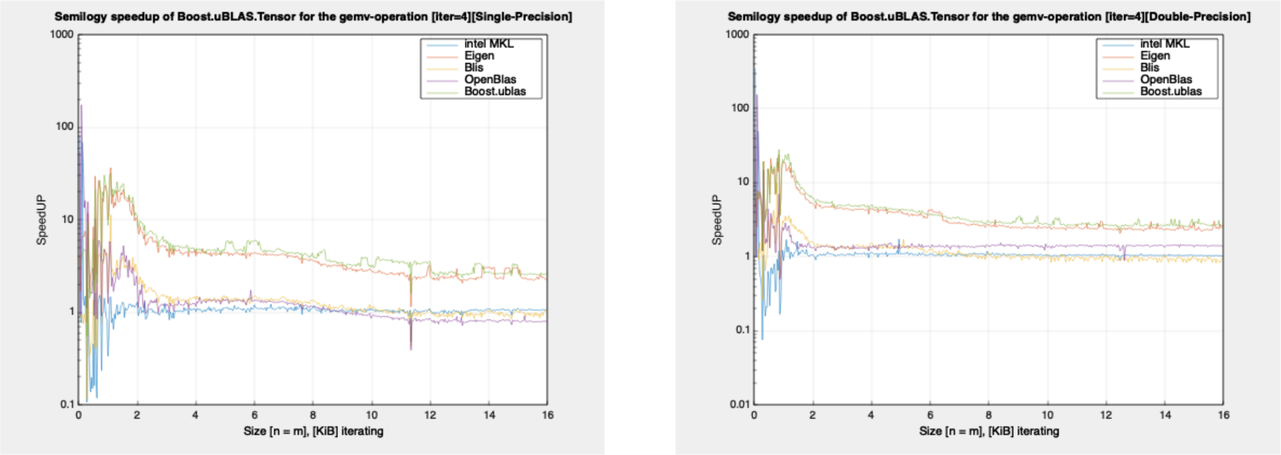
Comparison of the Boost.uBLAS.Tensor ?gemv implementation



(a) Single-Precision

(b) Double-Precision

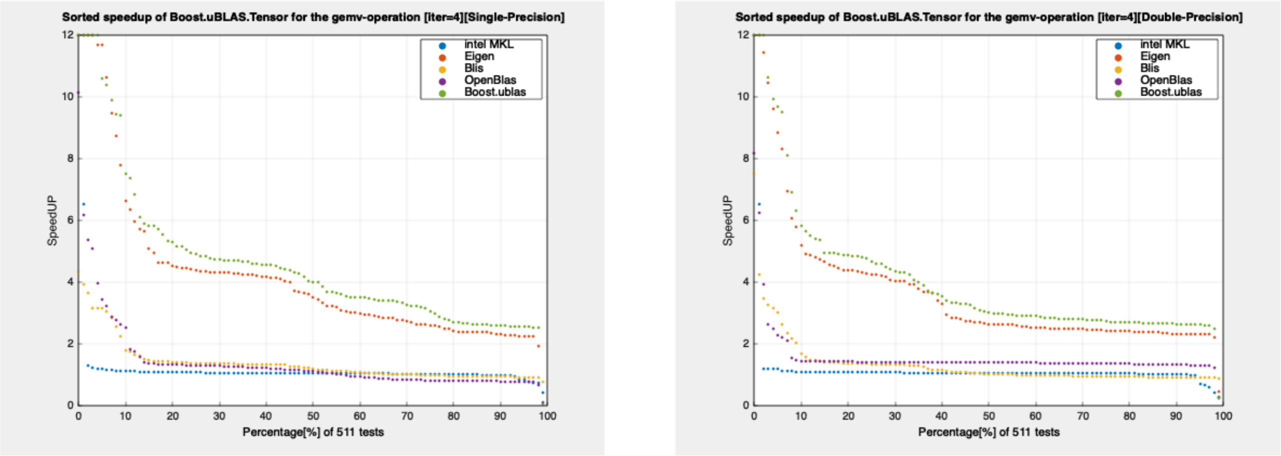
Comparison of the Boost.uBLAS.Tensor ?gemv implementation [semilogy]



(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?gemv implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 4.9: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	98	98
OpenBLAS	56	10
Eigen	98	97
Blis	73	9
Intel's MKL	88	1

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	42	0
Eigen	0	0
Blis	25	0
Intel's MKL	10	0

Table 4.10: Speedup Summary For Double-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	98	98
OpenBLAS	98	7
Eigen	98	98
Blis	59	9
Intel's MKL	92	1

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	39	0
Intel's MKL	6	1

4.6 Performance Metrics For Row-Major

Range[Start: 32, End: 16382, Step: 32]

Table 4.11: GFLOPS For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	114.495	19.4496
Boost.uBLAS	4.86818	3.62716
Intel's MKL	103.86	19.2955
OpenBLAS	39.0813	14.0183
Blis	28.5224	12.8971
Eigen	8.70003	4.11144

Table 4.12: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	48.6174	8.13646
Boost.uBLAS	2.19048	1.82291
Intel's MKL	51.6863	8.51818
OpenBLAS	18.6354	5.19158
Blis	14.9491	6.06728
Eigen	6.28753	2.04718

Table 4.13: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	19.4456	3.30327
Boost.uBLAS	0.826797	0.616026
Intel's MKL	17.6393	3.27708
OpenBLAS	6.63744	2.38082
Blis	4.84416	2.1904
Eigen	1.47759	0.698275

Table 4.14: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	16.5141	2.76374
Boost.uBLAS	0.744048	0.619195
Intel's MKL	17.5565	2.89341
OpenBLAS	6.32996	1.76344
Blis	5.07782	2.0609
Eigen	2.13571	0.695374

Table 4.15: Speedup(Boost.uBLAS.Tensor) For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	23.5192	5.36222
Intel's MKL	1.1024	1.00799
OpenBLAS	2.92968	1.38745
Blis	4.01423	1.50807
Eigen	13.1604	4.73061

Table 4.16: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	22.1949	4.46344
Intel's MKL	0.940624	0.955187
OpenBLAS	2.60887	1.56724
Blis	3.25219	1.34104
Eigen	7.73235	3.97447

Chapter 5

Matrix-Matrix Product

The matrix-matrix product is a binary operation that produces another matrix from two matrices. For the procedure to be valid, the number of columns of the first matrix must be equal to the rows of the second matrix. The resulting matrix's dimension is an amalgam of the dimension of the two matrics.

Let the first matrix be A with the dimensions $m \times k$; the second matrix be B with the dimensions $n \times k$, and the result matrix be C.

Hence, the dimensions of the matrix C will be $m \times n$.

$$A = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0k} \\ a_{10} & a_{11} & \dots & a_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mk} \end{bmatrix}, B = \begin{bmatrix} b_{00} & b_{01} & \dots & b_{0n} \\ b_{10} & b_{11} & \dots & b_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k0} & b_{k1} & \dots & b_{kn} \end{bmatrix}, C = \begin{bmatrix} c_{00} & c_{01} & \dots & c_{0n} \\ c_{10} & c_{11} & \dots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m0} & c_{m1} & \dots & c_{mn} \end{bmatrix}$$

$$C = A \times B \tag{5.1}$$

$$\begin{bmatrix} c_{00} & c_{01} & \dots & c_{0n} \\ c_{10} & c_{11} & \dots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m0} & c_{m1} & \dots & c_{mn} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0k} \\ a_{10} & a_{11} & \dots & a_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mk} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & \dots & b_{0n} \\ b_{10} & b_{11} & \dots & b_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k0} & b_{k1} & \dots & b_{kn} \end{bmatrix}$$

$$c_{i,j} = \sum_{l=0}^{k} (a_{i,l} \times b_{l,j}) \tag{5.2}$$

5.1 Algorithm

The algorithm we implemented using OpenMP came from the Low et al. [2016], and we simply following the algorithm provided in the paper and all the tuning parameter also comes from the same paper using their calculation. However, there are a few difference in the calculation because we cannot control the assembly which emitted by the compiler. The calculation is not far from the initially provided by the paper.

Algorithm 10: Matrix-Matrix Product SIMD Function

```
// $c$ is the pointer to the output Matrix
// $ldc$ is leading dimension of the output Matrix
// $a$ is the pointer to the input Matrix
// $b$ is the pointer to the input Matrix
// $k_c$ is the block of the Matrix in the contracting dimension
// $m$ is the block of the Matrix $A$ in the non-contracting dimension
// $n$ is the block of the Matrix $B$ in the non-contracting dimension
// $M_r$ is the micro-tile of the Matrix $A$ in the non-contracting dimension
// $N_r$ is the micro-tile of the Matrix $B$ in the non-contracting dimension
Function simd_loop (c, ldc, a, b, $k_c$, m, n):
    $buff_{M_r \times N_r} \leftarrow 0$
    $half_{N_r} \leftarrow \frac{N_r}{2}$
    for $k \leftarrow 0$ to $k_c$ by $1$ do
        for $j \leftarrow 0$ to $half_{N_r}$ by $1$ do
            #pragma omp simd
            for $i \leftarrow 0$ to $M_r$ by $1$ do
                $buff[j \times M_r + i] \leftarrow buff[j \times M_r + i] + bk[j] \times ak[i]$
                $buff[(j + half_{N_r}) \times M_r + i] \leftarrow buff[j \times M_r + i] + bk[j + half_{N_r}] \times ak[i]$
            end
        end
    end
    // Copy the $buff$ into the output Matrix $c$
end
```

Algorithm 11: Matrix-Matrix Product Micro-Kernel

```
// $c$ is the pointer to the output Matrix
// $w_c$ is pointer to the strides of the output Matrix
// $a$ is the pointer to the input Matrix
// $w_a$ is pointer to the strides of the output Matrix
// $b$ is the pointer to the input Matrix
// $w_b$ is pointer to the strides of the output Matrix
// $M$ is the block of the Matrix $A$ in the non-contracting dimension
// $N$ is the block of the Matrix $B$ in the non-contracting dimension
// $K$ is the block of the Matrix in the contracting dimension
// $M_r$ is the micro-tile of the Matrix $A$ in the non-contracting dimension
// $N_r$ is the micro-tile of the Matrix $B$ in the non-contracting dimension
Function micro_kernel (c, ldc, a, b, M, N, K):
    $ldc \leftarrow max(w_c[0], w_c[1])$
    for $j \leftarrow 0$ to $N$ by $N_r$ do
        $a_i \leftarrow a$
        $b_i \leftarrow b + j \times K$
        $c_i \leftarrow c + j \times w_c[1]$
        $j_b \leftarrow max(N_r, N - j)$
        for $i \leftarrow 0$ to $M$ by $M_r$ do
            $a_k \leftarrow ai + i \times K$
            $b_k \leftarrow bi$
            $c_k \leftarrow ci + i \times w_c[o]$
            $i_b \leftarrow max(M_r, M - i)$
            $simd\_loop(c_k, ldc, a_k, b_k, K, i_b, j_b)$
        end
    end
end
```

Algorithm 12: Matrix-Matrix Product

```
Input: $c$, $n_c$, $w_c$ $a$, $n_a$, $w_a$, $b$, $n_b$, $w_b$, $max\_threads$
// $c$ is the pointer to the output Matrix
// $n_c$ is pointer to the dimensions of the output Matrix
// $w_c$ is pointer to the strides of the output Matrix
// $a$ is the pointer to the input Matrix
// $n_a$ is pointer to the dimensions of the output Matrix
// $w_a$ is pointer to the strides of the output Matrix
// $b$ is the pointer to the input Matrix
// $n_b$ is pointer to the dimensions of the output Matrix
// $w_b$ is pointer to the strides of the output Matrix
// $max\_threads$ is the user provided thread count
// $M_B$ is the block of the Matrix $A$ in the non-contracting dimension
// $N_B$ is the block of the Matrix $B$ in the non-contracting dimension
// $K_B$ is the block of the Matrix in the contracting dimension
// $M_r$ is the micro-tile of the Matrix $A$ in the non-contracting dimension
// $N_r$ is the micro-tile of the Matrix $B$ in the non-contracting dimension
begin
    $M \leftarrow na[0]$
    $N \leftarrow nb[1]$
    $K \leftarrow nb[0]$
    $a_i \leftarrow a$
    $b_i \leftarrow b$
    $c_i \leftarrow c$
    $buff\_size_A \leftarrow K_B \times (M_B + 1) \times max\_threads$
    $buff\_size_B \leftarrow K_B \times (N_B + 1)$
    $buff_A(buff\_size_A)$
    $buff_B(buff\_size_B)$
    #pragma omp parallel
    begin
        for $j \leftarrow 0$ to $N$ by $N_B$ do
            $a_k \leftarrow a_i$
            $b_k \leftarrow b_i + j \times w_b[1]$
            $c_k \leftarrow c_i + j \times w_c[1]$
            $j_b \leftarrow max(N_B, N - j)$
            for $k \leftarrow 0$ to $K$ by $K_B$ do
                $a_i \leftarrow a_k + k \times w_a[1]$
                $b_i \leftarrow b_k + k \times w_b[0]$
                $c_i \leftarrow c_k$
                $k_b \leftarrow max(K_B, K - j)$
                // Pack $B$ into $buff_B$
                #pragma omp for schedule(dynamic)
                for $i \leftarrow 0$ to $M$ by $M_B$ do
                    $t_{id} \leftarrow omp\_get\_thread\_num()$
                    $a_j \leftarrow a_j + i \times w_a[0]$
                    $b_j \leftarrow b_j$
                    $c_j \leftarrow c_j + i \times w_c[0]$
                    $i_b \leftarrow max(M_B, M - i)$
                    // Pack $A$ into $buff_A$
                    $micro\_kernel(c_j, w_c, (buff_A + t_{id} \times k_b \times M_B), buff_B, i_b, j_b, k_b)$
                end
            end
        end
    end
end
```

5.2 Tuning Parameter

If the reader wants to know the reasoning and rationale behind each of the derivation of the tuning parameter, they may read the original Low et al. [2016] paper.

5.2.1 Micro-Tiles M_r and N_r

Let L_{VFMA} and N_{VFMA} be the latency and throughput of the FMA instruction, N_{VEC} be the width of the vector register, and $L_{L/S}$ and $N_{L/S}$ be the latency and throughput of the Load and Store instruction.

Using equation from the Low et al. [2016], we get

$$M_r N_r \geq N_{VEC}(L_{VFMA} + L_{L/S})(N_{VFMA} + N_{L/S}) \tag{5.3}$$

$$\begin{aligned} M_r &= \lceil \frac{\sqrt{N_{VEC}(L_{VFMA} + L_{L/S})(N_{VFMA} + N_{L/S})}}{N_{VEC}} \rceil N_{VEC} \\ N_r &= \lceil \frac{N_{VEC}(L_{VFMA} + L_{L/S})(N_{VFMA} + N_{L/S})}{M_r} \rceil \end{aligned}$$

Calculating M_r and N_r for Intel SkyLake with the AVX2.

Table 5.1: Latency and Throughput

Instruction	**Latency**	**Throughput**
VFMA	4	0.5
Load/Store	7	0.5

$N_{VEC} = 8$ for Single-Precision and $N_{VEC} = 4$ for Double-Precision.

$$\begin{aligned} M_r &= \lceil \frac{\sqrt{11 N_{VEC}}}{N_{VEC}} \rceil N_{VEC} \\ N_r &= \lceil \frac{11 N_{VEC}}{M_r} \rceil \end{aligned}$$

Table 5.2: Tuning Parameter M_r and N_r for Intel SkyLake with the AVX2

Tuning Parameter	**Single**	**Double**
M_r	16	8
N_r	6	6

5.2.2 Macro-Tiles M_B, N_B and K_B

Let C_{A_r} and C_{B_r} be the number of cache line taken by the micro-tile A_r and B_r.

Similarly, to derive the K_B, we will use the derivation from the Low et al. [2016], which is mapped to the parameter k_c in the paper.

According to the paper, if we do not want the micro-tile of B_r to replaced by the new micro-tile of A_r, the C_{A_r} and C_{B_r} should follow the below constraint.

$$C_{A_r} + C_{B_r} \leq W_{L_1}$$

But here, we also need to consider the cache line of the micro-tile of C, which will L_1 cache as the route to the register.

$$C_{A_r} + C_{B_r} \leq W_{L_1} - 1 \tag{5.4}$$

We know that

$$M_r K_B S_{data} = C_{A_r} N_{L_1} C_{L_1} \tag{5.5}$$

$$N_r K_B S_{data} = C_{B_r} N_{L_1} C_{L_1} \tag{5.6}$$

Dividing the 5.5 and 5.6, we get

$$\frac{C_{A_r}}{C_{B_r}} = \frac{M_r}{N_r}$$

Here we part ways little bit from the paper and we take floor rather than ceil.

$$C_{B_r} = \lfloor \frac{N_r}{M_r} C_{A_r} \rfloor \tag{5.7}$$

Substituting the 5.7 into the 5.4, we get

$$C_{A_r} \lfloor (1 + \frac{N_r}{M_r}) \rfloor = W_{L_1} - 1$$
$$C_{A_r} = \lceil \frac{W_{L_1} - 1}{1 + \frac{N_r}{M_r}} \rceil$$

Table 5.3: Cache Info for the Intel SkyLake

Level	W_{L_i}	N_{L_i}	C_{L_i}
L_1	8	64	64
L_2	4	1024	64
L_3	16	$16Ki$	64

$$C_{A_r} = \lceil \frac{7}{1 + \frac{N_r}{M_r}} \rceil$$

Substituting the C_{A_r} into the equation 5.5.

Table 5.4: Tuning Parameter K_B

Tuning Parameter	**Single**	**Double**
K_B	384	256

To get M_B, we do the same process as we did to find K_B, but we need to fill the macro-block of matrix A into the L_2 cache.

$$C_{A_c} + C_{B_r} \leq W_{L_2} - 1$$
$$C_{A_c} = W_{L_2} - 1 - C_{B_r}$$

We the C_{B_r} from the equation 5.7. Here, there is a catch what if the C_{A_c} comes out to be 0. Therefore, to avoid the this problem can derive the relationship between the C_{A_c} and C_{B_r}.

$$
\begin{aligned}
M_B K_B S_{data} &= C_{A_c} N_{L_2} C_{L_2} \\
N_r K_B S_{data} &= C_{B_r} N_{L_2} C_{L_2}
\end{aligned}
$$

Now, dividing the both equation and we get

$$\frac{C_{A_c}}{C_{B_r}} = \frac{M_B}{N_r}$$

$$
\begin{aligned}
C_{A_c} \lfloor (1 + \frac{N_r}{M_B}) \rfloor &= W_{L_2} - 1 \\
C_{A_c} &= \lceil \frac{W_{L_2} - 1}{1 + \frac{N_r}{M_B}} \rceil
\end{aligned}
$$

Using the observation, we can see M_B will be very large compared to the N_r. Therefore, we reach to the conclusion that N_r cannot be larger than the M_B and can never be zero. But their may be the case where they are same.

$$
\begin{aligned}
0 < \frac{N_r}{M_B} &\leq 1 \\
1 < 1 + \frac{N_r}{M_B} &\leq 2 \\
\frac{1}{2} \leq \frac{1}{1 + \frac{N_r}{M_B}} &< 1 \\
\frac{W_{L_2} - 1}{2} \leq \frac{W_{L_2} - 1}{1 + \frac{N_r}{M_B}} &< W_{L_2} - 1
\end{aligned}
$$

For the best possible solution or performance, we chose the lower bound from the above equation.

$$
\begin{aligned}
C_{A_c} &= \lfloor \frac{W_{L_2} - 1}{2} \rfloor \\
M_B &= \lfloor \frac{(W_{L_2} - 1)}{2} \rfloor \frac{N_{L_2} C_{L_2}}{K_B S_{data}}
\end{aligned}
$$

To the get the best optimal value of M_B, we try to find nearest possible multiple of M_r, which is not greater than the M_B.

Table 5.5: Tuning Parameter M_B

Tuning Parameter	**Single**	**Double**
M_B	32	32

Now, we need to find the tuning parameter N_B, which depends on the L_3 cache. In order to find the last tuning parameter, we need to introduce the elephant in the room, which is the issue that some machine may share the L_3 cache among all the threads and make our life harder. This introduces the cache thrashing, where multiple cache will load the tiles from L_3 cache and they will evict the block of the matrix B, and we again need to load it from next iteration.

The reason for the cache thrashing is simple, as we load a block of A and a block of C multiple times, they constantly are replacing the cache lines, but the block of B will the oldest to update because we keep the micro-tile of the block B in the L_1 cache for reuse and making it the last one to update. Therefore, according to the LRU cache algorithm, the block of B will be replaced.

Using the same equation to find the right number of cache line, we get

$$C_{A_c} + C_{B_c} \leq W_{L_3} - 1$$

But C_{A_c} is the number occuping cache line for one thread and we need the number of cache line for T threads.

$$TC_{A_c} + C_{B_c} \leq W_{L_3} - 1$$

Now, we again find ourselves in a problem. What if $TC_{A_c} >= W_{L_3} - 1$? The number of cache line for C_{B_c} cannot be zero or negative.

Therefore we again the take the same approach as we did to find the M_B and we get

$$C_{B_c} = \lfloor \frac{W_{L_3} - 1}{2} \rfloor$$

$$N_B = \lfloor \frac{(W_{L_3} - 1)}{2} \rfloor \frac{N_{L_3} C_{L_3}}{K_B S_{data}}$$

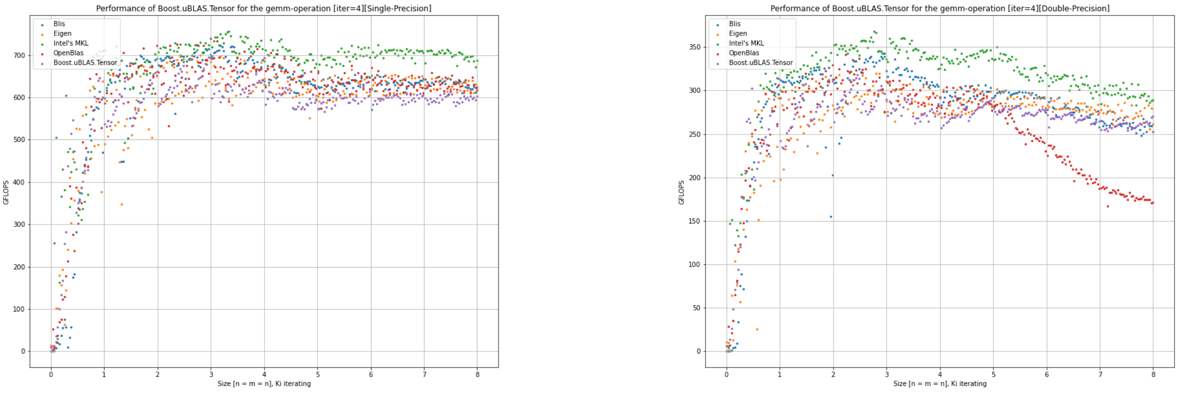
Again, we get the nearest multiple of N_r for the block N_B, but it should not be greater than N_B.

Table 5.6: Tuning Parameter N_B

Tuning Parameter	**Single**	**Double**
N_B	4776	3582

5.3 Performance Plots and Speedup Summary

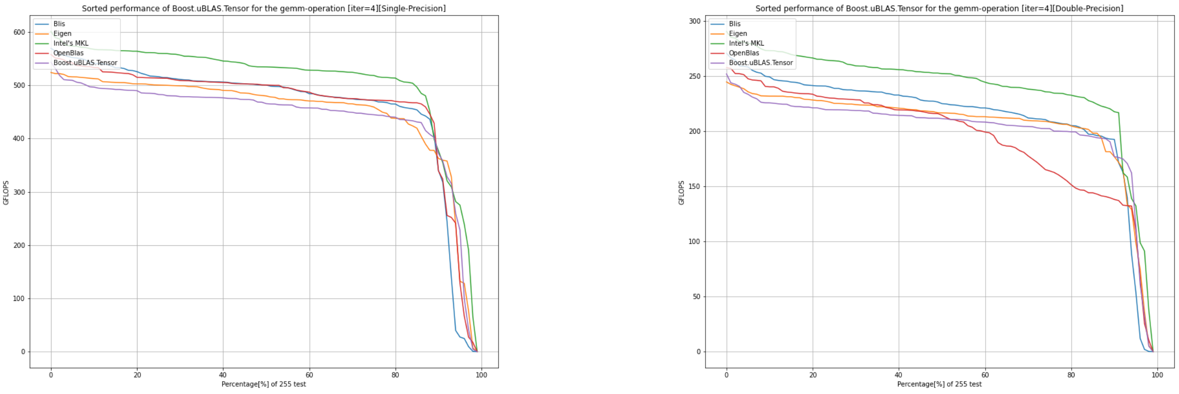
Performance measurements of ?gemm implementations



(a) Single-Precision

(b) Double-Precision

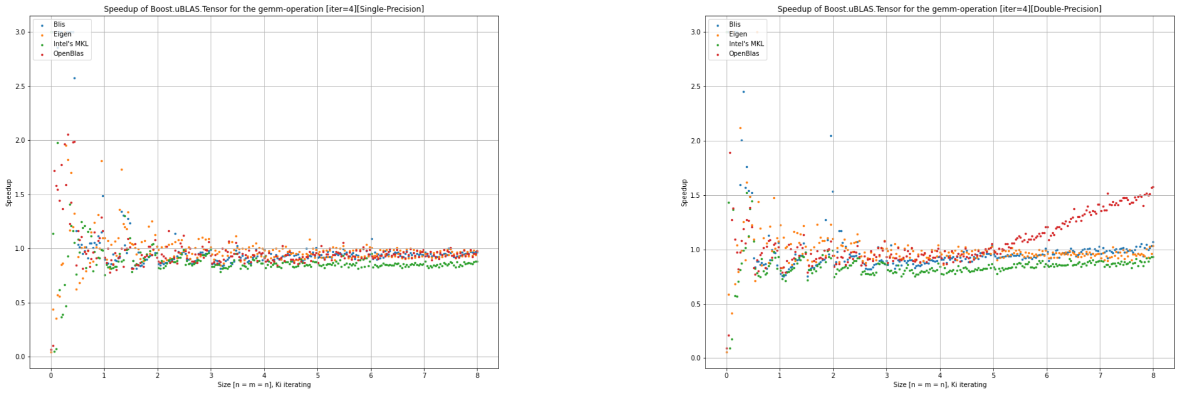
Sorted performance measurements of ?gemm implementations



(a) Single-Precision

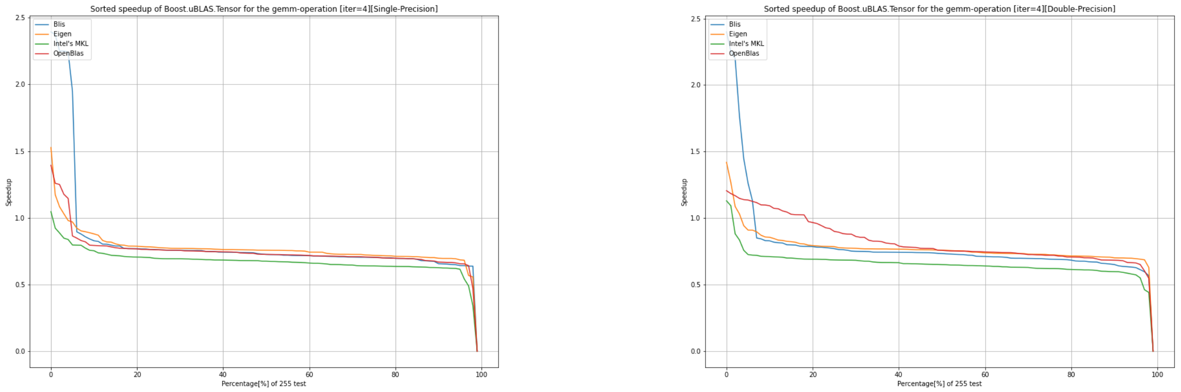
(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?gemm implementation


(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor ?gemm implementation [sorted]


(a) Single-Precision

(b) Double-Precision

Table 5.7: Speedup Summary For Single-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
OpenBLAS	4	0
Eigen	3	0
Blis	5	4
Intel's MKL	0	0

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
OpenBLAS	99	99
Eigen	99	99
Blis	99	99
Intel's MKL	99	99

Table 5.8: Speedup Summary For Double-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
OpenBLAS	18	0
Eigen	3	0
Blis	6	2
Intel's MKL	1	0

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
OpenBLAS	99	99
Eigen	99	99
Blis	99	99
Intel's MKL	99	99

Range[Start: 32, End: $8Ki$, Step: 32]

Table 5.9: GFLOPS For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	698.297	577.354272
Intel's MKL	755.743	664.758359
OpenBLAS	739.551	607.188286
Blis	721.793	602.726757
Eigen	699.169	589.249039

Table 5.10: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	318.398	264.381474
Intel's MKL	368.052	312.992397
OpenBLAS	341.611	253.483622
Blis	342.725	278.499451
Eigen	320.632	268.866619

Table 5.11: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	118.596637	98.056092
Intel's MKL	128.353091	112.900537
OpenBLAS	125.603091	103.123011
Blis	122.587126	102.365278
Eigen	118.744735	100.076263

Table 5.12: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	108.151495	89.803490
Intel's MKL	125.017663	106.315352
OpenBLAS	116.036345	86.101774
Blis	116.414742	94.598998
Eigen	108.910326	91.326976

Table 5.13: Speedup(Boost.uBLAS.Tensor) For Single-Precision

Implementation	**Max**	**Average**
Intel's MKL	0.923987	0.868518
OpenBLAS	0.944218	0.950865
Blis	0.967448	0.957904
Eigen	0.998753	0.979814

Table 5.14: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Intel's MKL	0.865090	0.844690
OpenBLAS	0.932048	1.042992
Blis	0.929019	0.949307
Eigen	0.993033	0.983318

Chapter 6

Matrix Transpose

The Matrix transpose is an operation where the columns of the matrix and the rows of the matrix interchanged. This operation also affects the matrix layout; if the matrix is a Column-Major, then after the transpose, it will be the Row-Major and vice-versa. The transpose operation is an involution (self-inverse) operation for the matrix or the vector.

$$A = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0k} \\ a_{10} & a_{11} & \dots & a_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mk} \end{bmatrix}$$

$$C = A^T$$

$$\begin{bmatrix} c_{00} & c_{01} & \dots & c_{0n} \\ c_{10} & c_{11} & \dots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m0} & c_{m1} & \dots & c_{mn} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{10} & \dots & a_{m0} \\ a_{01} & a_{11} & \dots & a_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0k} & a_{1k} & \dots & a_{mk} \end{bmatrix}$$

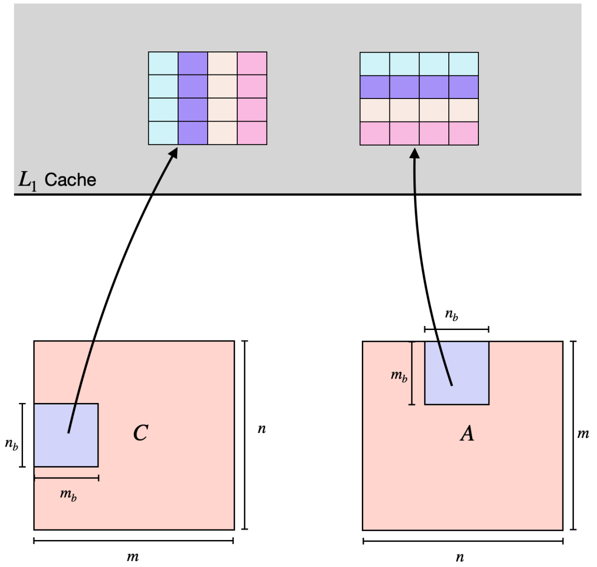
Involution Property: $(A^T)^T = A$

6.1 Algorithm

We divide the matrix into smaller blocks to optimise the transpose operation to keep the block inside the L1 cache and increase the cache hit. For the in-place algorithm, we use the fact that the matrix is symmetrical around the diagonal, which reduces the cost of bringing the other block matrix when we can swap the entries around the diagonal. This fact is valid for the block which is on the diagonal but for the other blocks, and we simply swap every with the corresponding block.

6.1.1 Out-Of-Place Transpose

Block Diagram for Out-Of-Place Transpose



Algorithm 13: Out-Of-Place Matrix Transpose SIMD Function

```
// $c$ is the pointer to the output Matrix
// $wc$ is the pointer to the strides of the output Matrix
// $a$ is the pointer to the input Matrix $A$
// $wa$ is the pointer to the strides of the input Matrix $A$
// $m$ is the row-block of the Matrix $A$
// $n$ is the col-block of the Matrix $A$
Function simd_loop ($c$, $wc$, $a$, $wa$, $m$, $n$):
    for $i \leftarrow 0$ to $m$ by $1$ do
        $cj \leftarrow c + wc[1] \times i$
        $aj \leftarrow a + wa[0] \times i$
        #pragma omp simd
         for $j \leftarrow 0$ to $n$ by $1$ do
            $cj[wc[0] \times j] \leftarrow aj[wa[1] \times j]$
        end
    end
end
```

Algorithm 14: Out-Of-Place Matrix Transpose

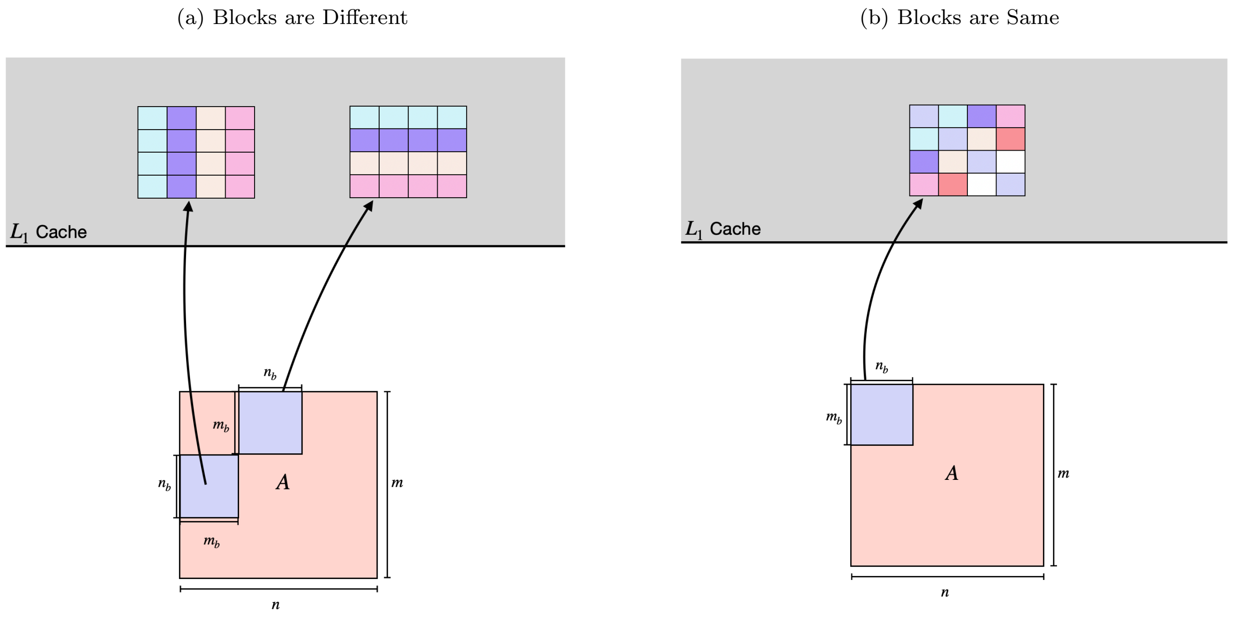
```
Input: $c$, $n_c$, $w_c$ $a$, $n_a$, $w_a$, $num\_threads$
// $c$ is the pointer to the output Matrix
// $n_c$ is pointer to the dimensions of the output Matrix
// $w_c$ is pointer to the strides of the output Matrix
// $a$ is the pointer to the input Matrix
// $n_a$ is pointer to the dimensions of the output Matrix
// $w_a$ is pointer to the strides of the output Matrix
// $num\_threads$ is the user provided thread count
// $block$ is the block size for the simd loop
begin
    $m \leftarrow n_a[0]$
    $n \leftarrow n_a[1]$
    $max\_blocks \leftarrow max(1, \lfloor \frac{m}{block} \rfloor)$
    $max\_threads \leftarrow min(max\_blocks, num\_threads)$
    #pragma omp parallel for num_threads(max_threads) schedule(dynamic)
     for $j \leftarrow 0$ to $m$ by $block$ do
        $ib \leftarrow min(m - i, block)$
        $aj \leftarrow a + i \times wa[0]$
        $cj \leftarrow c + i \times wc[1]$
        for $i \leftarrow 0$ to $n$ by $block$ do
            $jb \leftarrow min(n - i, block)$
            $ak \leftarrow a + j \times wa[1]$
            $ck \leftarrow c + j \times wc[0]$
            $simd\_loop(ck, w_c, ak, w_a, ib, jb)$
        end
    end
end
```

6.1.2 In-Place Transpose

Algorithm 15: In-Place Matrix Transpose SIMD Function

```
// $c$ is the pointer to the output Matrix
// $wc$ is the pointer to the strides of the output Matrix
// $a$ is the pointer to the input Matrix $A$
// $wa$ is the pointer to the strides of the input Matrix $A$
// $m$ is the row-block of the Matrix $A$
// $n$ is the col-block of the Matrix $A$
// $is\_same\_block$ tells us if the $c$ and $b$ are pointing at the same block
Function $simd\_loop$ $(c, wc, a, wa, m, n, is\_same\_block)$:
    for $i \leftarrow 0$ to $m$ by $1$ do
        $cj \leftarrow c + wc[1] \times i$
        $aj \leftarrow a + wa[0] \times i$
        $idx \leftarrow i * is\_same\_block$
        #pragma omp simd
         for $i \leftarrow idx$ to $n$ by $1$ do
            $swap(cj[wc[0] \times j], aj[wa[1] \times j])$
        end
    end
end
```

Block Diagram for In-Place Transpose



(a) Blocks are Different

(b) Blocks are Same

```
Algorithm 16: In-Place Matrix Transpose
Input: $c$, $n_c$, $w_c$ $a$, $n_a$, $w_a$, $num\_threads$
// $c$ is the pointer to the output Matrix
// $n_c$ is pointer to the dimensions of the output Matrix
// $w_c$ is pointer to the strides of the output Matrix
// $a$ is the pointer to the input Matrix
// $n_a$ is pointer to the dimensions of the output Matrix
// $w_a$ is pointer to the strides of the output Matrix
// $num\_threads$ is the user provided thread count
// $block$ is the block size for the simd loop
begin
    $m \leftarrow n_a[0]$
    $n \leftarrow n_a[1]$
    $max\_blocks \leftarrow max(1, \lfloor \frac{m}{block} \rfloor)$
    $max\_threads \leftarrow min(max\_blocks, num\_threads)$
    #pragma omp parallel for num_threads(max_threads) schedule(dynamic)
     for $j \leftarrow 0$ to $m$ by $block$ do
        $ib \leftarrow min(m - i, block)$
        $aj \leftarrow a + i \times wa[0]$
        $cj \leftarrow c + i \times wc[1]$
        for $i \leftarrow 0$ to $n$ by $block$ do
            $jb \leftarrow min(n - i, block)$
            $ak \leftarrow a + j \times wa[1]$
            $ck \leftarrow c + j \times wc[0]$
            $simd\_loop(ck, w_c, ak, w_a, ib, jb, i == j)$
        end
    end
end
```

6.2 Tuning Parameter

Here, we try to keep the block of the output matrix and the input matrix in the L_1 cache, which is the help the CPU to access the block as fast as possible. The transpose operation is a memory-bound algorithm; therefore, we do not control the operation.

Let the column block be m_b and the row block be n_b. Therefore, the block occupies space of $m_b \times n_b \times S_{data}$ in the L_1 cache.

$$A_{block} = m_b \times n_b \times S_{data}$$

$$C_{block} = (A_{block})^T$$

$$C_{block} = n_b \times m_b \times S_{data}$$

Now, we need both blocks to be present in the L_1, which means block of matrix A and the block matrix B should occupy $A_{block} + C_{block}$.

$$\begin{aligned}
S_{L_1} &= A_{block} + C_{block} \\
S_{L_1} &= m_b \times n_b \times S_{data} + n_b \times m_b \times S_{data} \\
S_{L_1} &= (m_b \times n_b + n_b \times m_b) \times S_{data} \\
\frac{S_{L_1}}{S_{data}} &= 2m_b \times n_b \\
m_b \times n_b &= \frac{S_{L_1}}{2S_{data}}
\end{aligned}$$

In order to maximize the bandwidth, we choose a square matrix, and the compiler decides which instructions to emit, or we could say that if the rows and columns are equivalent in terms of size, we get the best possible bandwidth.

Therefore, $m_b = n_b = B$.

$$\begin{aligned}
B^2 &= \frac{S_{L_1}}{2S_{data}} \\
B &= \sqrt{\frac{S_{L_1}}{2S_{data}}}
\end{aligned}$$

An astute reader may observe that cache size and data size are the power of two for most of the part because the machine's native language is binary.

We can say that $\frac{S_{L_1}}{S_{data}}$ must a function of the power of two and let the power be some k.

$$\frac{S_{L_1}}{S_{data}} = 2^k$$

Now, after replacing with the power of two we get

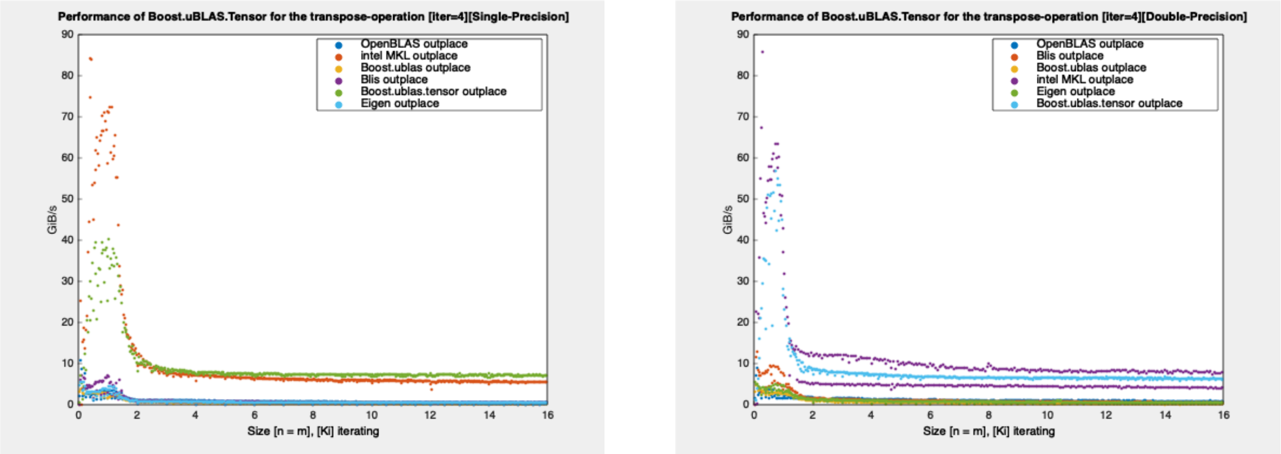
$$\begin{aligned}
B &= \sqrt{\frac{2^k}{2}} \\
B &= \sqrt{2^{k-1}} \\
B &= 2^{\lfloor \frac{k-1}{2} \rfloor}
\end{aligned}$$

6.3 Performance Plots and Speedup Summary For Out-Of-Place

Performance measurements of transpose implementations

(a) Single-Precision

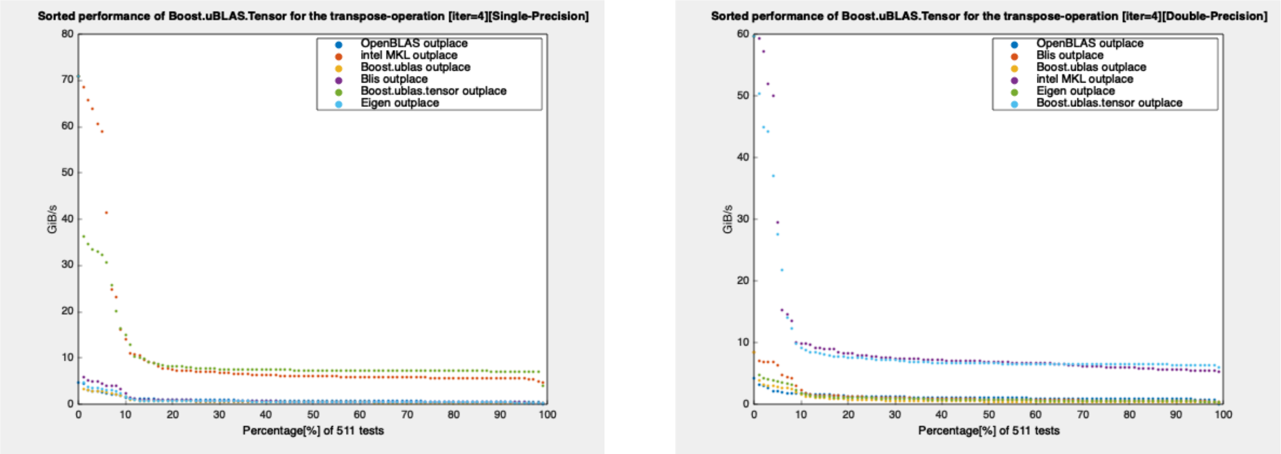
(b) Double-Precision



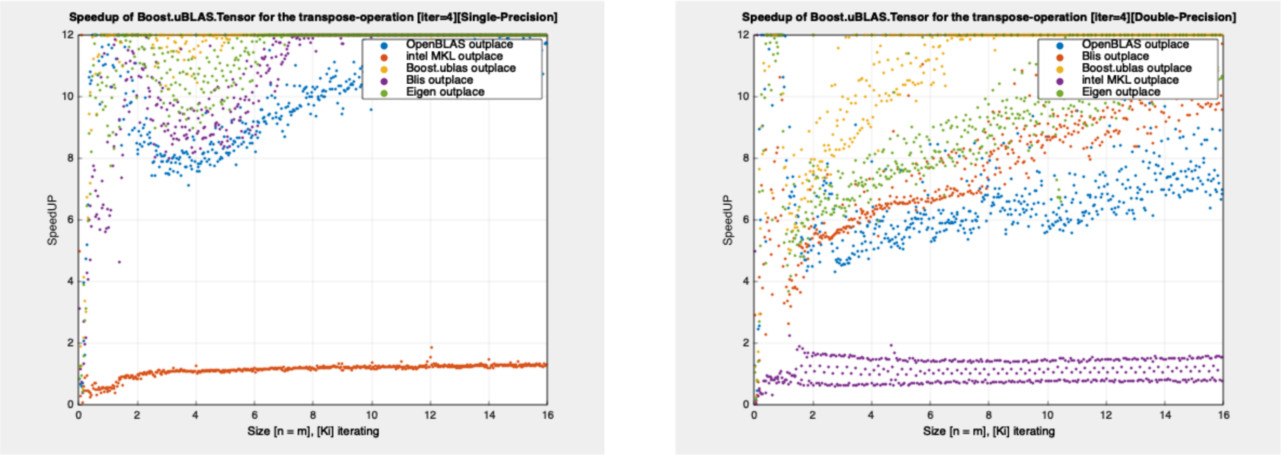
Sorted performance measurements of transpose implementations

(a) Single-Precision

(b) Double-Precision



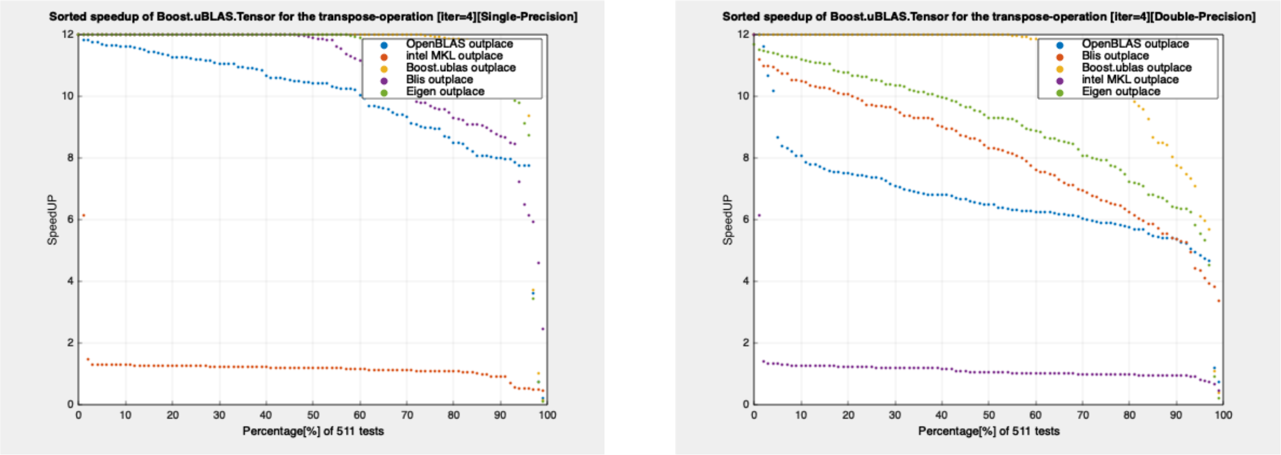
Comparison of the Boost.uBLAS.Tensor transpose implementation



(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor transpose implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 6.1: Speedup Summary For Single-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
Boost.uBLAS	98	97
OpenBLAS	97	97
Eigen	97	97
Blis	99	99
Intel's MKL	86	1

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
Boost.uBLAS	0	0
OpenBLAS	1	0
Eigen	1	0
Blis	0	0
Intel's MKL	12	1

Table 6.2: Speedup Summary For Double-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
Boost.uBLAS	98	97
OpenBLAS	98	97
Eigen	97	97
Blis	99	99
Intel's MKL	71	1

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	1	0
Intel's MKL	27	0

Range[Start: 32**, End:** $16Ki$**, Step:** 32]

Table 6.3: GiBs For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	4.09509	0.680433
Boost.uBLAS.Tensor	40.1745	9.58504
Intel's MKL	84.1344	10.7844
OpenBLAS	10.919	0.99106
Blis	7.17433	1.00462
Eigen	5.65426	0.836292

Table 6.4: GiBs For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	4.98701	0.762708
Boost.uBLAS.Tensor	56.7747	8.8159
Intel's MKL	85.8162	9.80812
OpenBLAS	8.90865	1.24028
Blis	13.0382	1.34253
Eigen	5.52208	1.0204

Table 6.5: Speedup(Boost.uBLAS.Tensor) For Single-Precision

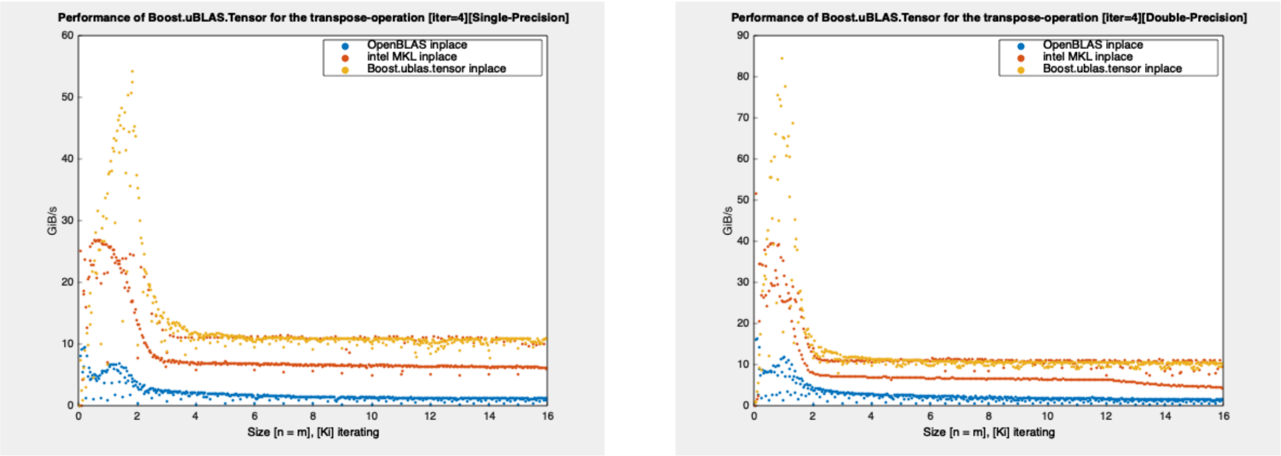
Implementation	**Max**	**Average**
Boost.uBLAS	9.81041	14.0867
Intel's MKL	0.477504	0.888787
OpenBLAS	3.67931	9.6715
Blis	5.59975	9.54096
Eigen	7.10518	11.4614

Table 6.6: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	11.3845	11.5587
Intel's MKL	0.661585	0.898837
OpenBLAS	6.37298	7.10797
Blis	4.35447	6.56664
Eigen	10.2814	8.63965

6.4 Performance Plots and Speedup Summary For In-Place

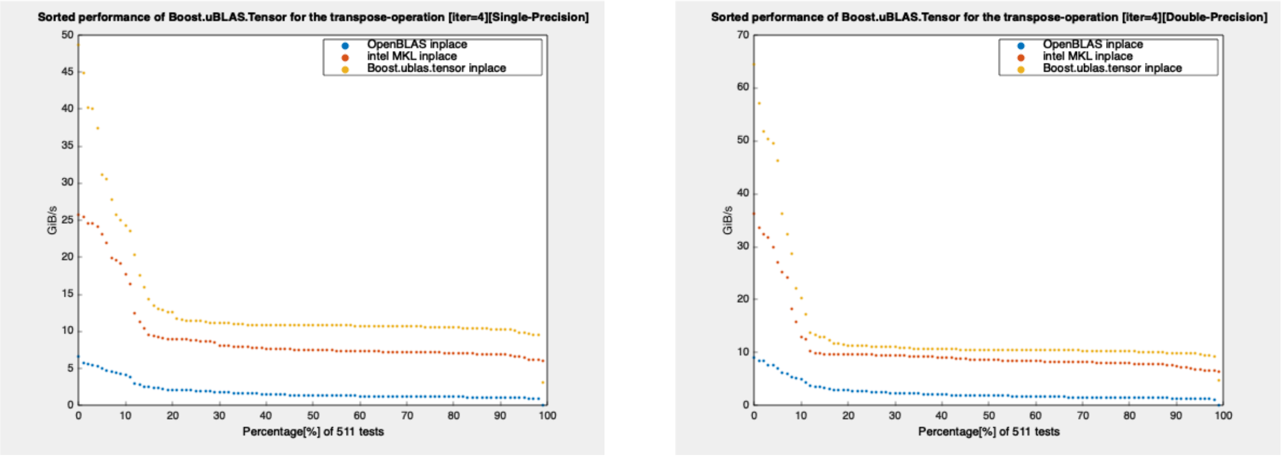
Performance measurements of transpose implementations



(a) Single-Precision

(b) Double-Precision

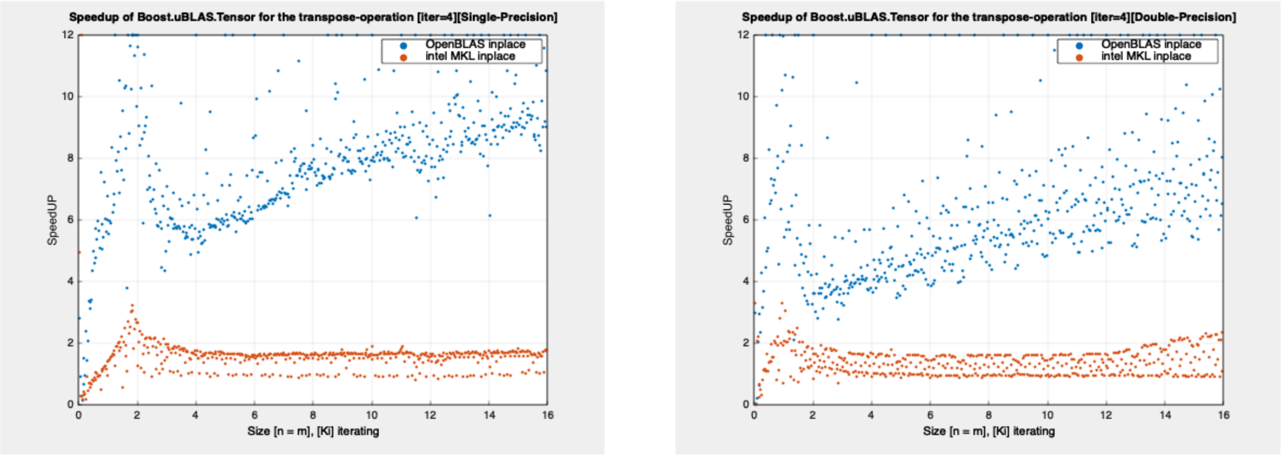
Sorted performance measurements of transpose implementations



(a) Single-Precision

(b) Double-Precision

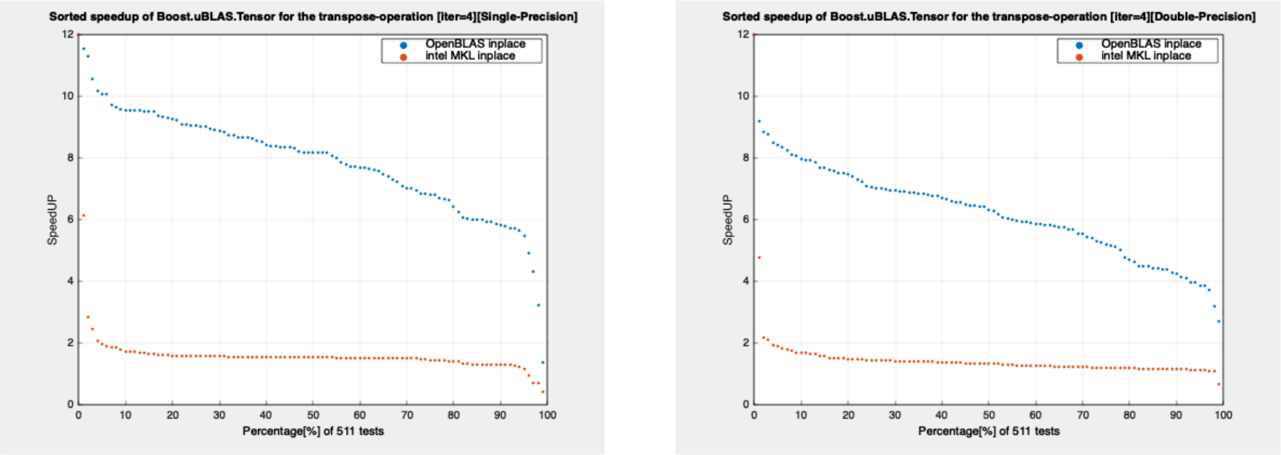
Comparison of the Boost.uBLAS.Tensor transpose implementation



(a) Single-Precision

(b) Double-Precision

Comparison of the Boost.uBLAS.Tensor transpose implementation [sorted]



(a) Single-Precision

(b) Double-Precision

Table 6.7: Speedup Summary For Single-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
OpenBLAS	99	98
Intel's MKL	95	4

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
OpenBLAS	0	0
Intel's MKL	3	0

Table 6.8: Speedup Summary For Double-Precision

Implementation	**Speedup** $\geq$ **1** [%]	**Speedup** $\geq$ **2** [%]
OpenBLAS	99	99
Intel's MKL	98	3

Implementation	**Speed-down** $\geq$ **1** [%]	**Speed-down** $\geq$ **2** [%]
OpenBLAS	0	0
Intel's MKL	0	0

Range[Start: 32, End: $16Ki$, Step: 32]

Table 6.9: GiBs For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	54.1773	13.5745
Intel's MKL	26.9047	9.3994
OpenBLAS	9.53067	1.89466

Table 6.10: GiBs For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	84.4904	13.7677
Intel's MKL	51.6235	10.5006
OpenBLAS	16.47	2.51434

Table 6.11: Speedup(Boost.uBLAS.Tensor) For Single-Precision

Implementation	**Max**	**Average**
Intel's MKL	2.01367	1.44419
OpenBLAS	5.68452	7.16462

Table 6.12: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Intel's MKL	1.63667	1.31113
OpenBLAS	5.12995	5.47565

Bibliography

FLOPS. Flops — Wikipedia, the free encyclopedia. = https://en.wikipedia.org/wiki/FLOPS,, 2021. [Online; accessed 24-March-2021].

Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12:1–12:18, August 2016. URL `http://doi.acm.org/10.1145/2925987`.