Implementing BLAS Operation using OpenMP

Amit Singh Cem Bassoy

June 8, 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 ← 0
    #pragma omp simd reduction(+:sum)
     for i ← 0 to n by 1 do
     |   sum ← sum + a[i] × 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_L1 ← ⌊S_L1 / S_data⌋
    number_el_L2 ← ⌊S_L2 / S_data⌋
    block1 ← 2 × number_el_L1
    block2 ← ⌊number_el_L1 / 2⌋
    block3 ← ⌊number_el_L2 / 2⌋
    sum ← 0
    omp_set_num_threads(max_threads)
    if n < block1 then
        sum ← simd_loop(a, b, n)
    end
    else if n < block3 then
        min_threads ← ⌊n / block2⌋
        num_threads ← max(1, max(min_threads, max_threads))
        omp_set_num_threads(num_threads)
        #pragma omp parallel for schedule(static) \
         reduction(+:sum)
        for i ← 0 to n by block2 do
            ib ← min(block2, n − i) sum ← sum + simd_loop(a + i, b + i, ib)
        end
    end
    else
        #pragma omp parallel reduction(+:sum)
         begin
            for i ← 0 to n by block3 do
                ib ← min(block3, n − i)
                ai ← a + i
                bi ← b + i
                #pragma omp for schedule(dynamic)
                 for j ← 0 to ib by block2 do
                    jb ← min(block2, ib − j)
                    sum ← sum + simd_loop(ai + j, bi + j, jb)
                end
            end
        end
    end
    c ← 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

$$\begin{aligned}
0 \leq \frac{1-p}{C_N + p} &\leq \frac{1}{C_N} \\
0 \leq O_N &< \frac{1}{C_N} \\
0 \leq O_N &< N - 1
\end{aligned}$$

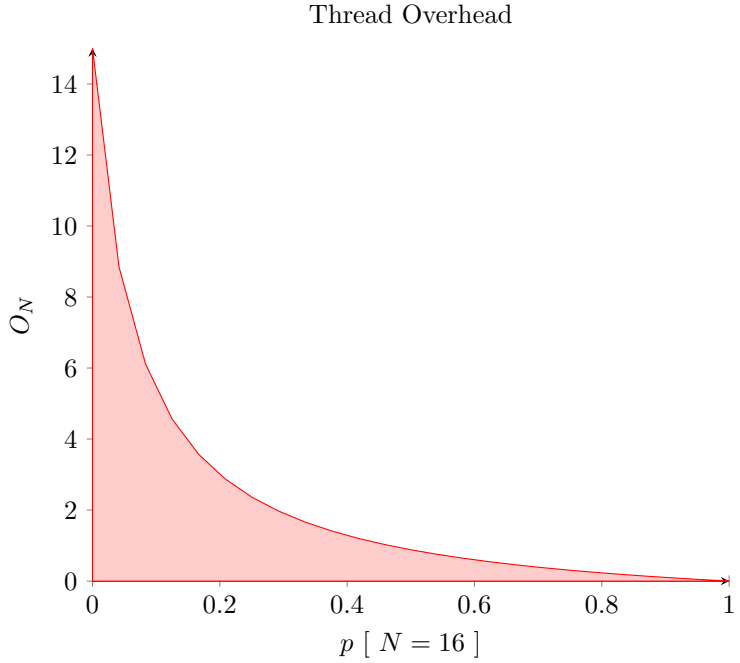
Therefore, $O_N \in [0, N-1)$. We can theorise that p must be a function of Block (B_1).

$$\begin{aligned}
p &\propto B_1 \\
p &= kB_1
\end{aligned}$$

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)



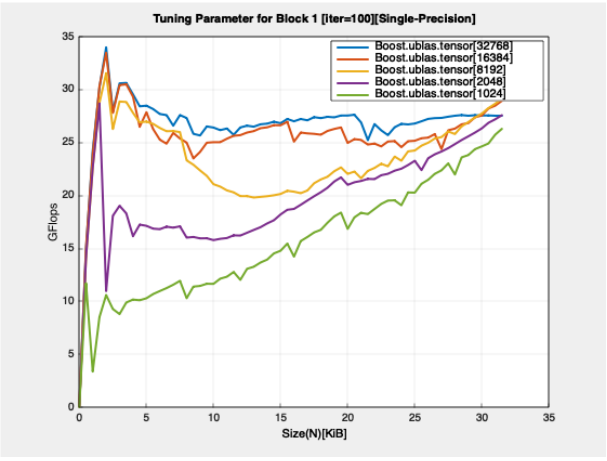
The B_1's lower bound found to be the number of elements that can be fit inside the L_1 cache through the experimental data. This phenomenon happens because the vectors' elements cannot fit inside the cache, and cache misses increase, dominating the thread overhead.

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

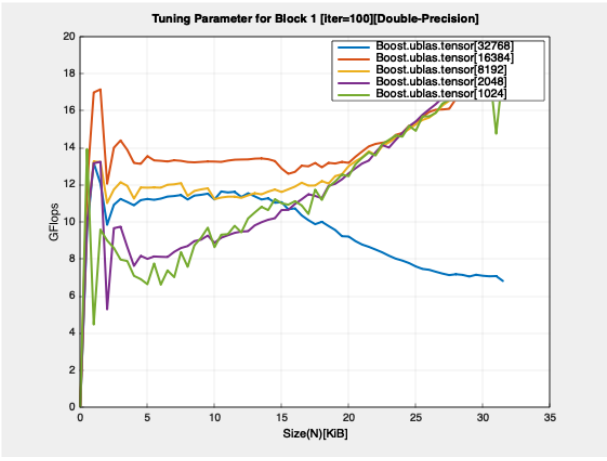
However, the optimal block size for this section found to be twice the vectors' elements that can put inside the cache.

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

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.

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

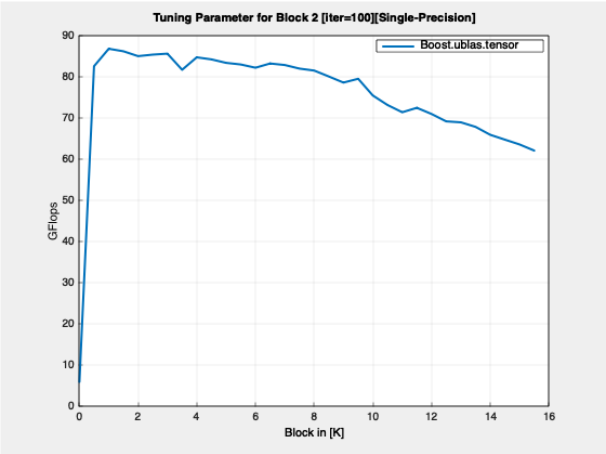
From the equation 2.1, we can infer that the block of both vectors must be equal to give the optimal block size.

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

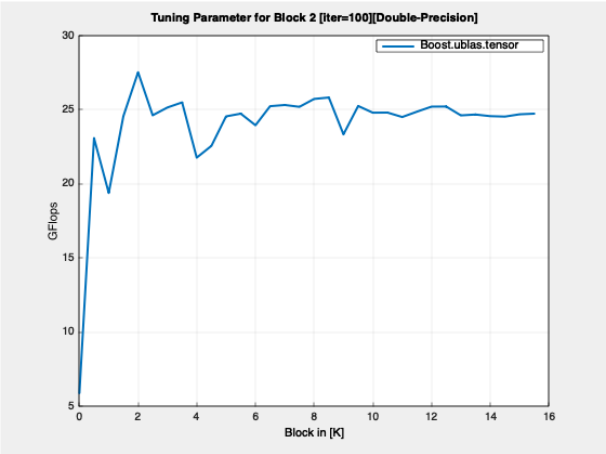
$$\begin{aligned} S_{L_1} &= S_{data}(B_2 + B_2) \\ S_{L_1} &= 2S_{data}B_2 \end{aligned}$$

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

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.

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

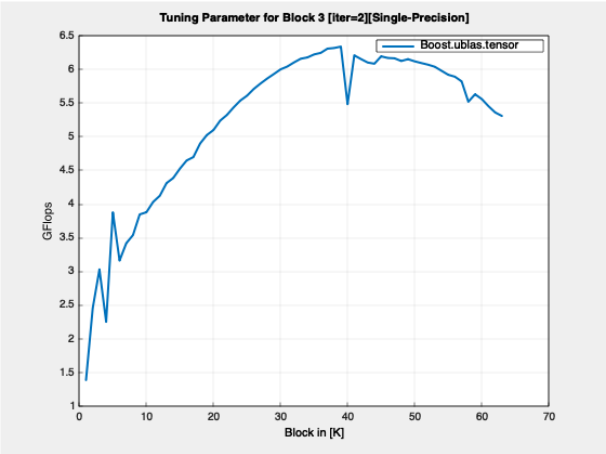
From the equation 2.1, we can infer that the block of both vectors must be equal to give the optimal block size.

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

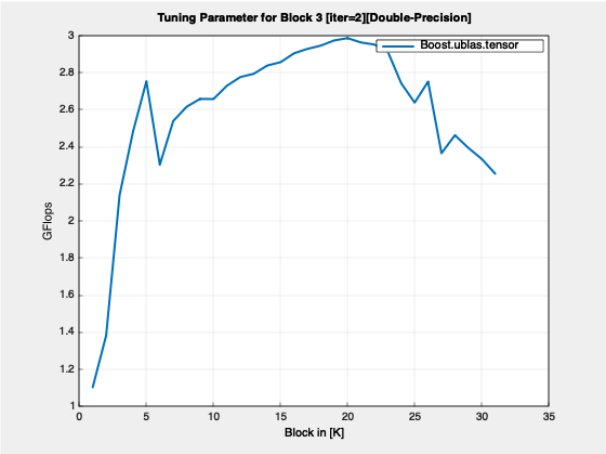
$$S_{L_2} = S_{data}(B_3 + B_3)$$
$$S_{L_2} = 2S_{data}B_3$$

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

Experimental Data for B_2



(a) Single-Precision



(b) Double-Precision

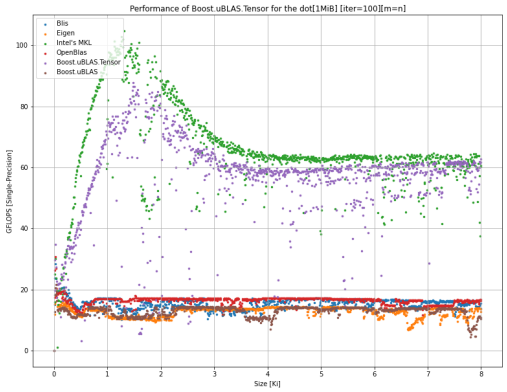
	Block[Single]	**Block[Double]**
Theoretical Optimal	32768 (32K)	16384 (16K)
Experimental Optimal	$\geq$ 30K and $\leq$ 40K	$\geq$ 16K and $\leq$ 20K

2.3 Performance Plots and Speedup Summary

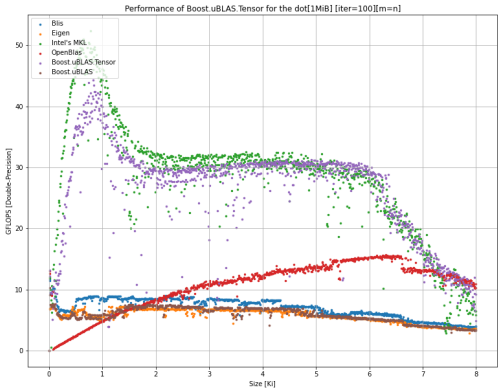
2.3.1 Range[Start: 2, End: 2^{20}, Step: 1024]

Performance measurements of ?dot implementations

(a) Single-Precision

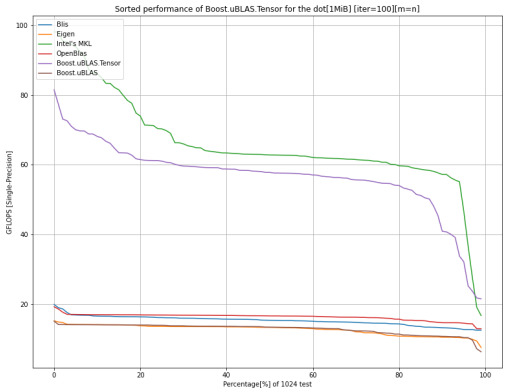


(b) Double-Precision

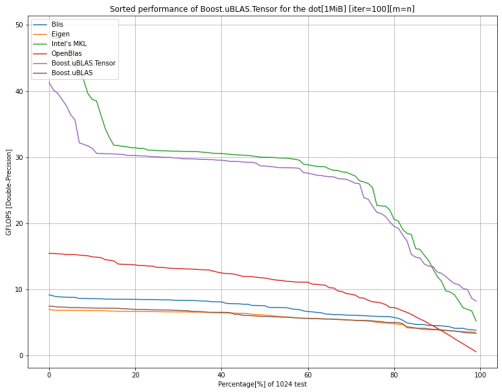


Sorted performance measurements of ?dot implementations

(a) Single-Precision

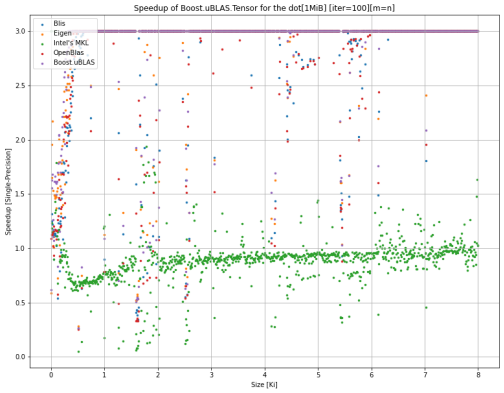


(b) Double-Precision

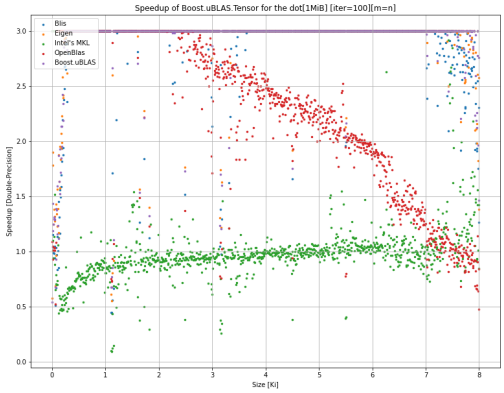


Comparison of the Boost.uBLAS.Tensor ?dot implementation

(a) Single-Precision

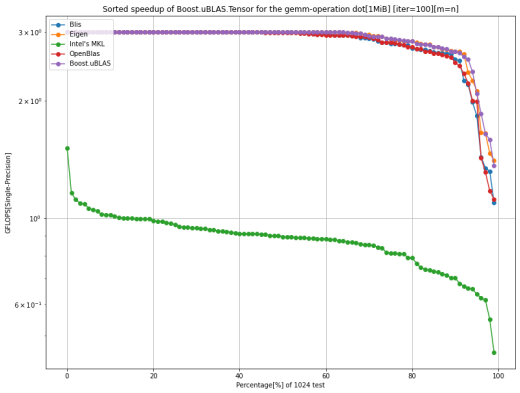


(b) Double-Precision

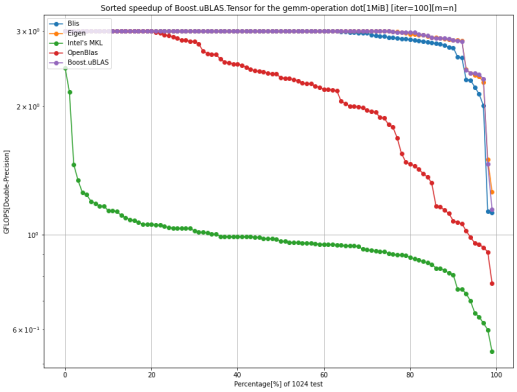


Comparison of the Boost.uBLAS.Tensor ?dot implementation [semilogy]

(a) Single-Precision

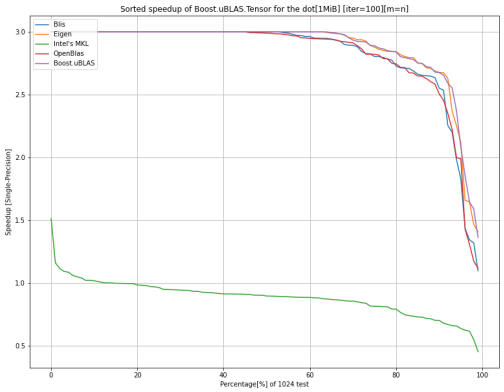


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

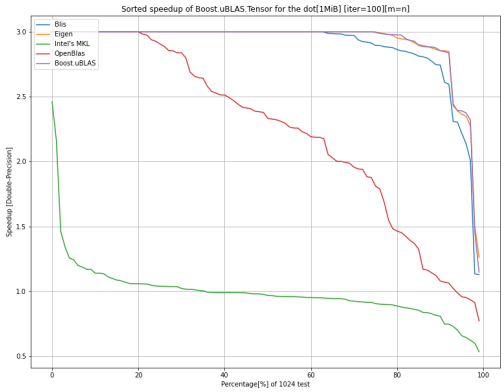


Table 2.1: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	95
OpenBLAS	99	93
Eigen	99	95
Blis	99	93
Intel's MKL	14	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	89	0

Table 2.2: Speedup Summary For Double-Precision

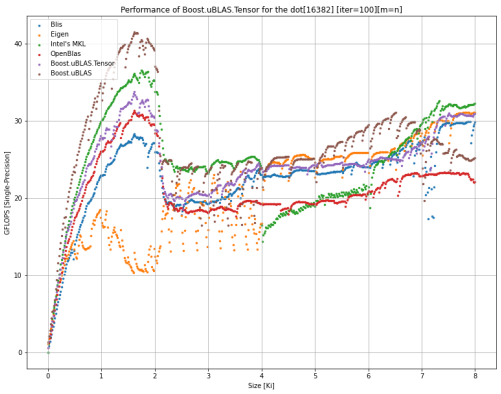
Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	97
OpenBLAS	93	67
Eigen	99	97
Blis	99	97
Intel's MKL	35	1

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	5	0
Eigen	0	0
Blis	0	0
Intel's MKL	64	1

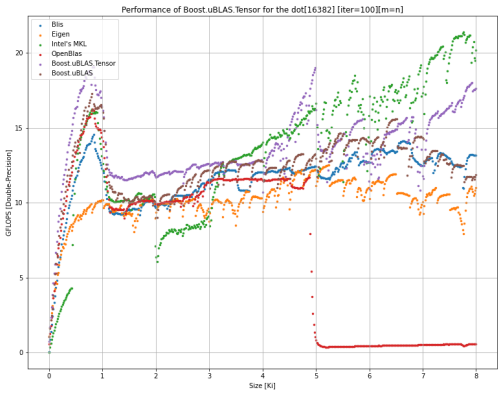
2.3.2 Range[Start: 32, End: 16382, Step: 32]

Performance measurements of ?dot implementations

(a) Single-Precision

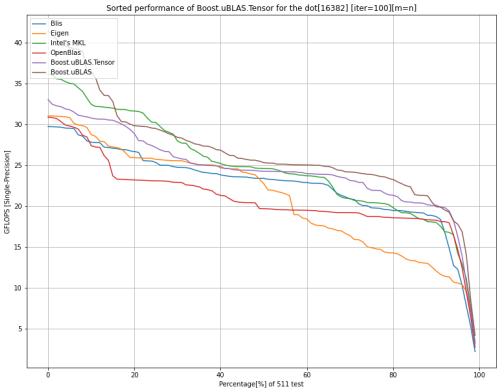


(b) Double-Precision

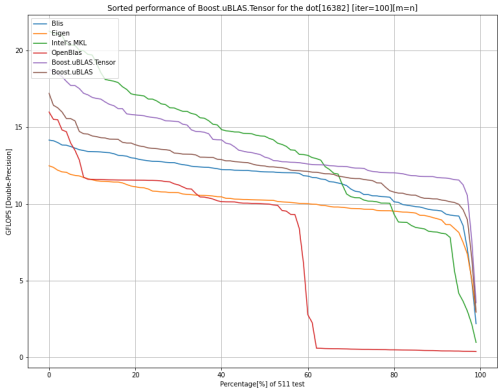


Sorted performance measurements of ?dot implementations

(a) Single-Precision

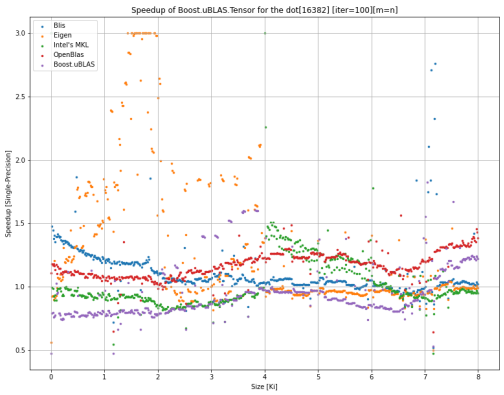


(b) Double-Precision

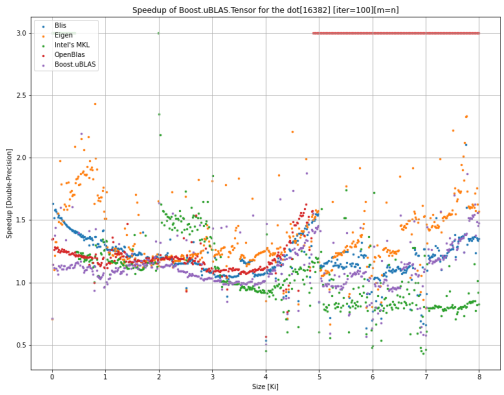


Comparison of the Boost.uBLAS.Tensor ?dot implementation

(a) Single-Precision

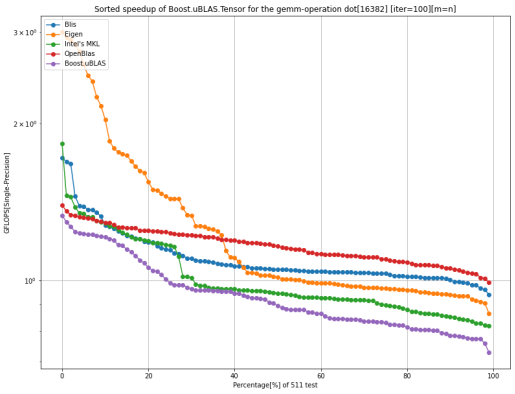


(b) Double-Precision

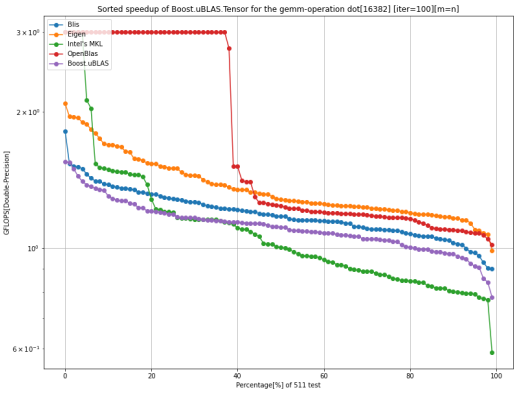


Comparison of the Boost.uBLAS.Tensor ?dot implementation [semilogy]

(a) Single-Precision

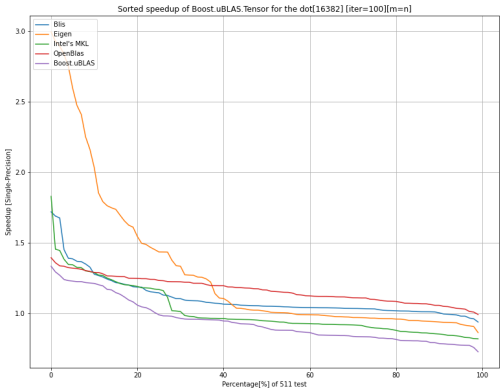


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

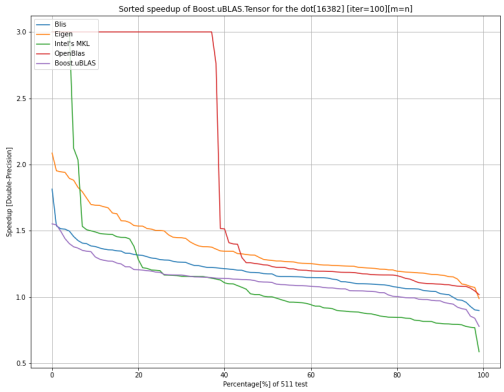


Table 2.3: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	24	0
OpenBLAS	98	0
Eigen	55	10
Blis	90	0
Intel's MKL	30	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	75	0
OpenBLAS	0	0
Eigen	47	0
Blis	8	0
Intel's MKL	68	0

Table 2.4: Speedup Summary For Double-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	80	0
OpenBLAS	99	38
Eigen	98	0
Blis	92	0
Intel's MKL	51	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	22	0
OpenBLAS	1	0
Eigen	0	0
Blis	7	0
Intel's MKL	50	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	87.7198	56.7261
Boost.uBLAS	21.7268	12.9477
Intel's MKL	104.652	65.8419
OpenBLAS	30.7292	16.4290
Blis	28.4656	15.4050
Eigen	20.4854	12.8286

Table 2.6: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	44.2618	26.1435
Boost.uBLAS	10.2785	5.8748
Intel's MKL	52.3413	28.0202
OpenBLAS	15.6984	10.6529
Blis	11.7113	7.0479
Eigen	7.5165	5.7548

Table 2.7: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	14.8980	9.6342
Boost.uBLAS	3.6900	2.1990
Intel's MKL	17.7737	11.1824
OpenBLAS	5.2189	2.7902
Blis	4.8345	2.6163
Eigen	3.4791	2.1787

Table 2.8: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	15.0345	8.880
Boost.uBLAS	3.4913	1.995
Intel's MKL	17.7789	9.5177
OpenBLAS	5.3323	3.6185
Blis	3.9780	2.3939
Eigen	2.5531	1.9547

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

Implementation	**Max**	**Average**
Boost.uBLAS	4.0374	4.3811
Intel's MKL	0.838	0.861
OpenBLAS	2.8546	3.4528
Blis	3.0816	3.6823
Eigen	4.2820	4.4218

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

Implementation	**Max**	**Average**
Boost.uBLAS	4.3062	4.450
Intel's MKL	0.8456	0.9330
OpenBLAS	2.8195	2.4541
Blis	3.7794	3.7093
Eigen	5.8885	4.5428

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.

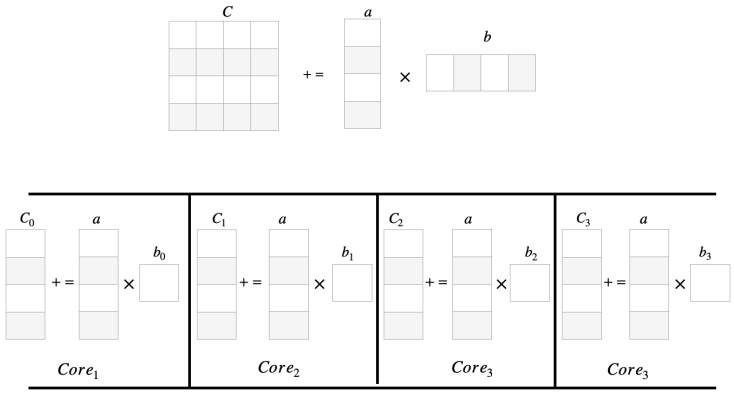
3.1 Calculating Number of Operations

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 ← b[0]
    #pragma omp simd
     for i ← 0 to n by 1 do
     |   c[i] ← c[i] + a[i] × 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.

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

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

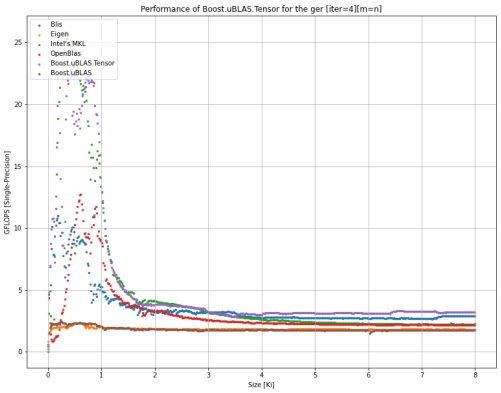
As the n increase, the block m_c decreases. We control the second vector's blocks through the outer loop, where we divide the second vector through threads, and each thread contains a single element. Therefore, the m_c block drops below the maximum allowed threads, we start to spawn m_c amount of threads.

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

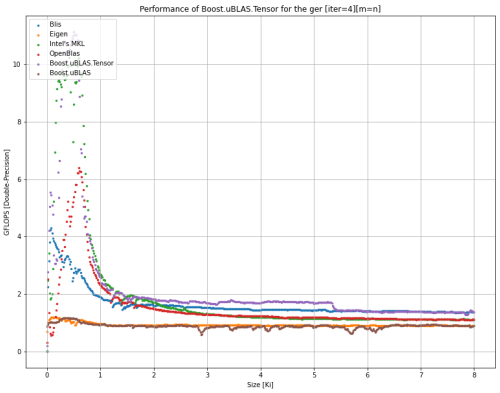
3.3 Performance Plots and Speedup Summary

Performance measurements of ?ger implementations

(a) Single-Precision

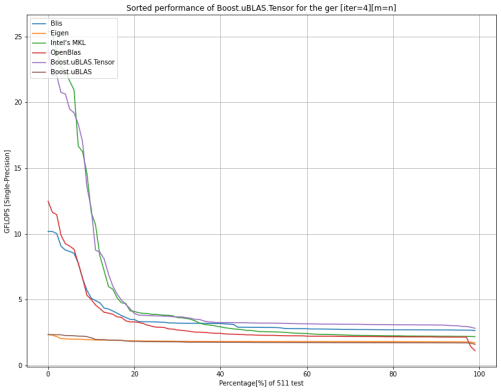


(b) Double-Precision

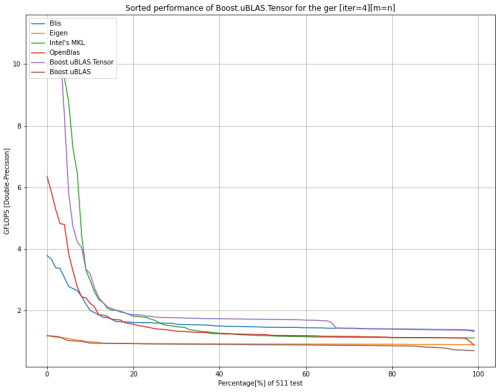


Sorted performance measurements of ?ger implementations

(a) Single-Precision

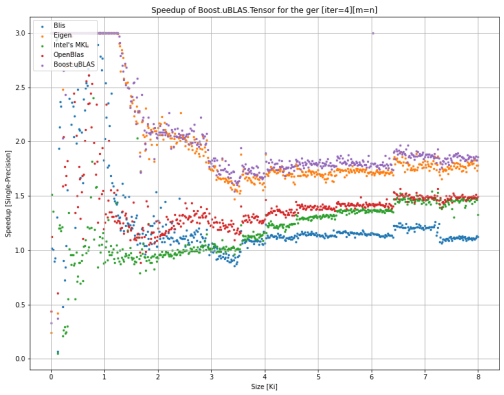


(b) Double-Precision

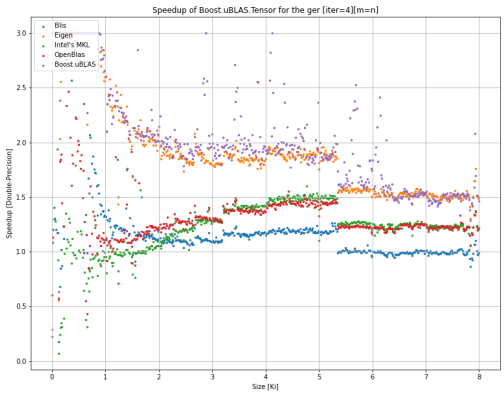


Comparison of the Boost.uBLAS.Tensor ?ger implementation

(a) Single-Precision

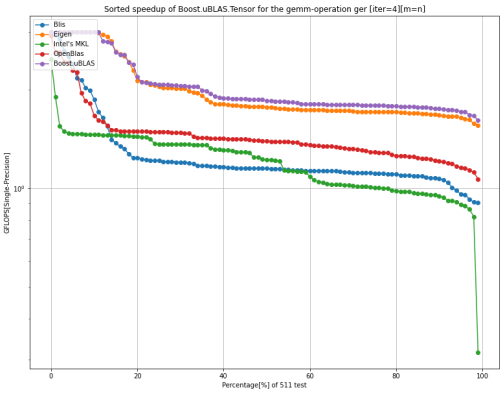


(b) Double-Precision

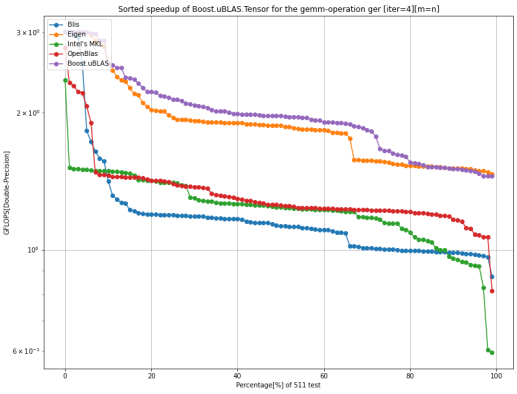


Comparison of the Boost.uBLAS.Tensor ?ger implementation [semilogy]

(a) Single-Precision

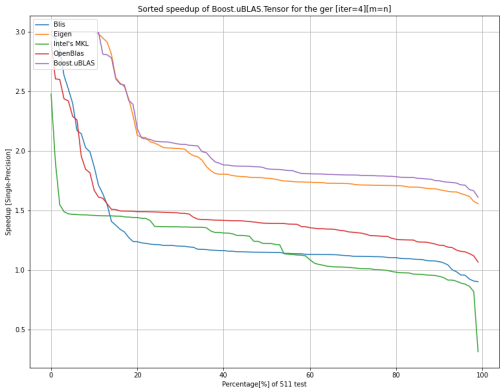


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

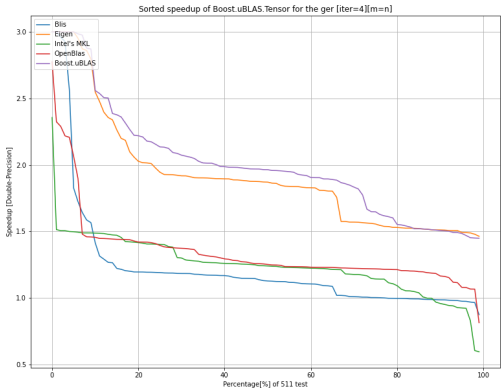


Table 3.1: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	34
OpenBLAS	99	6
Eigen	99	31
Blis	93	8
Intel's MKL	77	0

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

Table 3.2: Speedup Summary For Double-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	38
OpenBLAS	98	5
Eigen	99	23
Blis	76	4
Intel's MKL	86	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	22	0
Intel's MKL	13	1

3.4 Performance Metrics

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

Table 3.3: GFLOPS For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	35.316	5.2823
Boost.uBLAS	2.48448	1.8241
Intel's MKL	26.8859	4.9064
OpenBLAS	13.4809	3.2026
Blis	13.1177	3.5963
Eigen	2.36464	1.8591

Table 3.4: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	12.3296	2.2376
Boost.uBLAS	1.21345	0.9012
Intel's MKL	11.7715	2.0042
OpenBLAS	7.00484	1.5713
Blis	4.9330	1.65328
Eigen	1.2312	0.9391

Table 3.5: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	5.9979	0.8971
Boost.uBLAS	0.4219	0.3098
Intel's MKL	4.5662	0.8333
OpenBLAS	2.2895	0.5439
Blis	2.2278	0.6107
Eigen	0.40160	0.3157

Table 3.6: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	4.1880	0.7600
Boost.uBLAS	0.4121	0.306
Intel's MKL	3.998	0.6807
OpenBLAS	2.379	0.5337
Blis	1.6756	0.5615
Eigen	0.41820	0.31899

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

Implementation	**Max**	**Average**
Boost.uBLAS	14.2146	2.895
Intel's MKL	1.3135	1.0766
OpenBLAS	2.6197	1.6493
Blis	2.6922	1.4688
Eigen	14.9350	2.8412

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

Implementation	**Max**	**Average**
Boost.uBLAS	10.1607	2.4827
Intel's MKL	1.0474	1.1164
OpenBLAS	1.7601	1.42407
Blis	2.4993	1.3534
Eigen	10.014	2.3827

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
// k is the contracting dimension
// m is the non-contracting dimension
// w is the leading dimension of the matrix
Function simd_loop0 (c, a, b, k, w):
    #pragma omp simd reduction(+ : c[0 : m_r])
     for i ← 0 to k by 1 do
        for j ← 0 to m_r by 1 do
            c[i] ← c[i] + a[j + i * w] * b[i]
        end
    end
end
Function simd_loop1 (c, a, b, m, k, w):
    #pragma omp simd reduction(+ : c[0 : m_r])
     for j ← 0 to m by 1 do
        for i ← 0 to k by 1 do
            c[i] ← c[i] + a[j + i * w] * b[i]
        end
    end
end
```

Algorithm 6: 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 pointer to the extents of the matrix a
// w_a is the pointer to the strides of the matrix a
// n_b is the size of the vector b
// $max\_threads$ is the user provided thread count
begin
  $omp\_set\_num\_threads(max\_threads)$
  // m_r is the micro-tile
  $M \leftarrow n_a[0]$
  $N \leftarrow n_a[1]$
  $lda \leftarrow max(w_a[0], w_a[1])$
  // N_b macro-tile for contracting dimension
  // M_b macro-tile for non-contracting dimension and should be the multiple of m_r
  #pragma omp parallel for
  for $i \leftarrow 0$ **to** M **by** M_b **do**
    $i_b \leftarrow min(M - i,\ m_b)$
    $a_i \leftarrow a + i \times w_a[0]$
    $b_i \leftarrow b$
    $c_i \leftarrow c + i$
    for $k \leftarrow 0$ **to** N **by** N_b **do**
      $k_b \leftarrow min(N - k,\ k_b)$
      $a_k \leftarrow a_i + k \times w_a[1]$
      $b_k \leftarrow b_i + k$
      $c_k \leftarrow c_i$
      $M_{iter} \leftarrow \lfloor \frac{i_b}{m_r} \rfloor$
      $M_{rem} \leftarrow i_b - (M_{iter} \times m_r)$
      $simd\_loop_1(c_k, a_k, b_k, M_{rem}, k_b, lda)$
      $a_k \leftarrow a_k + M_{rem} \times w_a[0]$
      $c_k \leftarrow c_k + M_{rem}$
      for $ii \leftarrow 0$ **to** M_{iter} **by** 1 **do**
        $a_p \leftarrow a_i + ii \times m_r \times w_a[1]$
        $b_p \leftarrow b_i + ii \times m_r$
        $c_p \leftarrow c_i$
        $simd\_loop_0(c_p, a_p, b_p, k_b, lda)$
      end
    end
  end
end

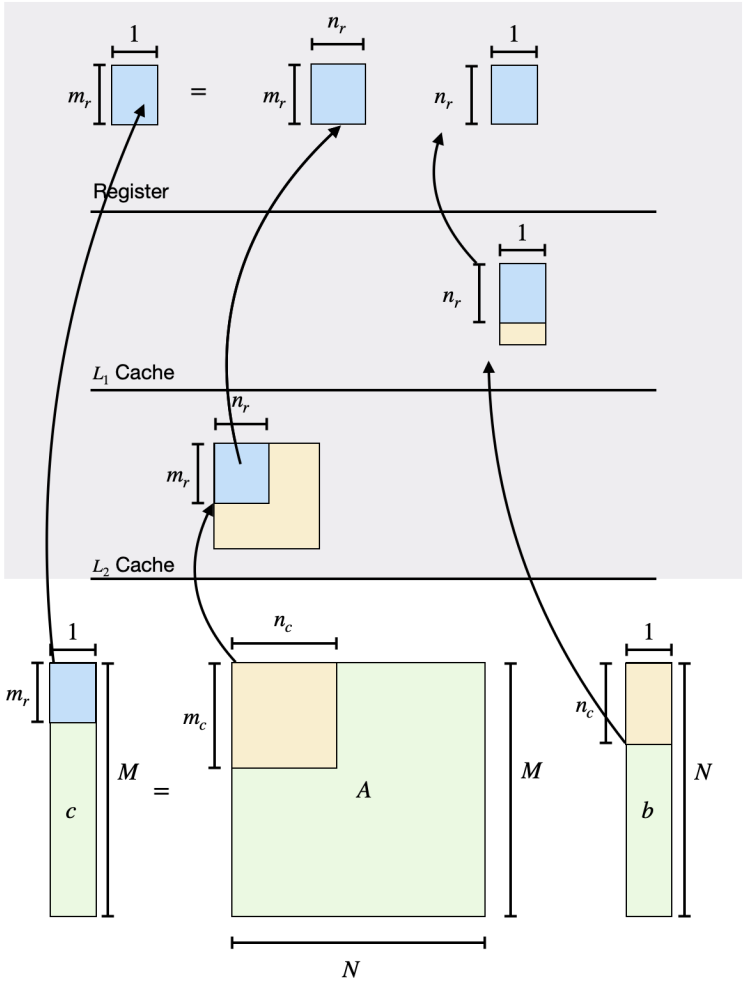
The Column-Major product is quite tricky and complicated if we cannot control the instruction set. The row elements are x strides away from each other and paralyse the cache predictor or cache locality. The cache predictor brings the cache line size elements from memory, which are place contiguously.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

The algorithm for the matrix times vector is similar to Blis' algorithm, which describes the inner working of the matrix times matrix in the paper Low et al. [2016]. However, there are a few differences which are as following:

- We do not pack the matrix.
- We do not use the micro-tile of the contracting dimension.
- We use two levels of cache.

We do not pack because of the time complexity of the packing and the matrix times vector complexity. The copying and operation time complexity is $O(N^2)$, but the copying overwhelms the operation and degrades the performance.

Figure 4.1: Column-Major Matrix-Vector Block Diagram



Let macro-tile of the matrix A be A_c, macro-tile of the vector b be b_c, micro-tile of the input matrix be A_r, micro-tile of the input vector be b_r, and micro-tile of the output vector be c_r.

To maximize the performance, we try to maximize the reuse factors—the innermost micro-tiles, c_r, calculated using the equation defined in the paper Low et al. [2016]. Afterwards, we go from cache L_1 to cache L_2, figuring out the size of the macro-tiles. Then, the macro-tile, A_c, is put in the cache L_2 to maximize the reuse factor and macro-tile, b_c kept in the cache L_1. Finally, we keep the macro-tile b_c in the cache L_1 because it avoids bringing the micro-tile c_r multiple times and increasing the reuse of c_r, which avoids polluting the cache or evicting the following tile and increases the probability of the next tile to present inside the L_1 cache.

For finding the m_r and n_r refer to the section 5.2.1, where we went into much more detail.

4.2.2 Macro-Tiles n_c and m_c

To find n_c, we have to focus on the cache L_1. For the b_c not to be evicted from the L_1, we want $A_{r_{prev}}$ to replace $A_{r_{next}}$ rather than replacing b_c because we assume the cache replacing algorithm to be LRU, and the old cache line

is replaced by the new cache line. Therefore, we have to use A_r to find the size n_c and consider c_r, which will go through the caches to load inside the register. When c_r loads, then it will replace the tiles that we put inside the caches in order to maximize the reuse factor and needs a whole one cache line reserved for it.

$$A_r = S_{data} m_r n_c = C_{A_r} N_{L_1} C_{L_1} \tag{4.3}$$

$$b_c = S_{data} n_c = C_{b_c} N_{L_1} C_{L_1} \tag{4.4}$$

Where C_{A_r} is a integral factor and denotes the number of cache-line taken by the micro-tile, A_r.

$$C_{A_r} + C_{b_c} \leq W_{L_1} - 1 \tag{4.5}$$

Dividing the equation 4.3 and 4.4, we get

$$m_r = \frac{C_{A_r}}{C_{b_c}}$$
$$C_{b_c} = \frac{C_{A_r}}{m_r}$$

Substituting in 4.5, we get

$$C_{A_r}(1 + \frac{1}{m_r}) \leq W_{L_1} - 1$$

$$C_{A_r} = \lceil \frac{W_{L_1} - 1}{1 + \frac{1}{m_r}} \rceil$$

But $1 + \frac{1}{m_r} \approx 1$ because $\frac{1}{m_r}$ tends toward zero for large value of m_r and we can ignore the denominator.

$$C_{A_r} = W - 1 \tag{4.6}$$

$$n_c = \frac{(W_{L_1} - 1) N_{L_1} C_{L_1}}{S_{data} m_r}$$

Similarly, we can calculate the m_c, and we get

$$m_c = \frac{(W_{L_2} - 1) N_{L_2} C_{L_2}}{S_{data} n_c}$$

But, there is a one difference here we need to worry about the A_c and for others we can reserve one cache-line for them pass through.

4.2.3 Row-Major

Algorithm 7: Matrix-Vector Product SIMD Function

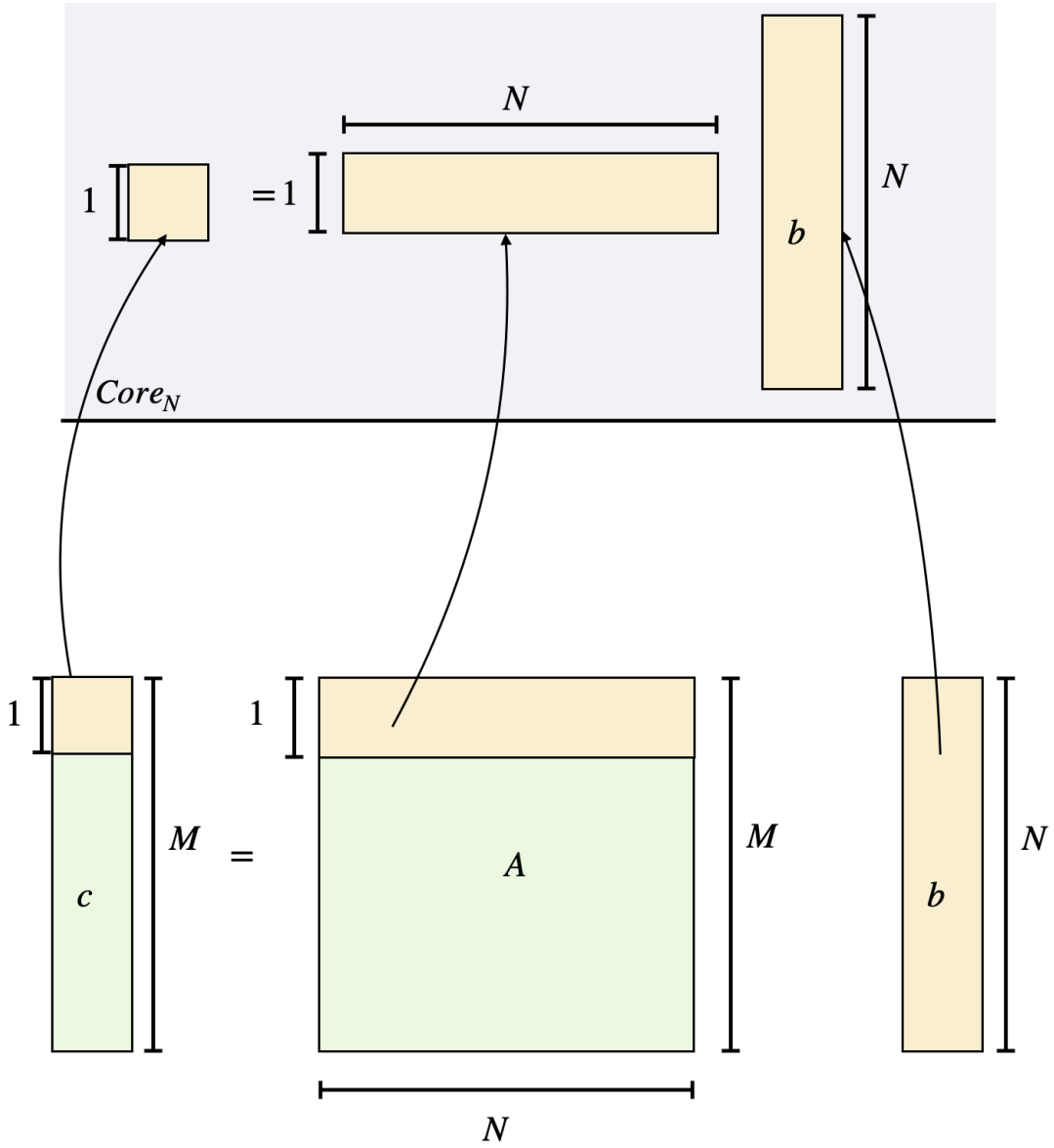
```
// a is the pointer to the input matrix
// b is the pointer to the input vector
// n_b is the block of the output vector
Function simd_loop (a, b, n_b):
    sum ← 0
    #pragma omp simd
     for i ← 0 to n_b by 1 do
        sum ← sum + a[i] × b[i]
    end
    return sum
end
```

Algorithm 8: 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 ← a
    bi ← b
    ci ← c
    #pragma omp parallel for schedule(static)
     for i ← 0 to n_a by 1 do
        aj ← ai + i * w_a
        bj ← bi
        cj ← ci + i
        cj ← 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.

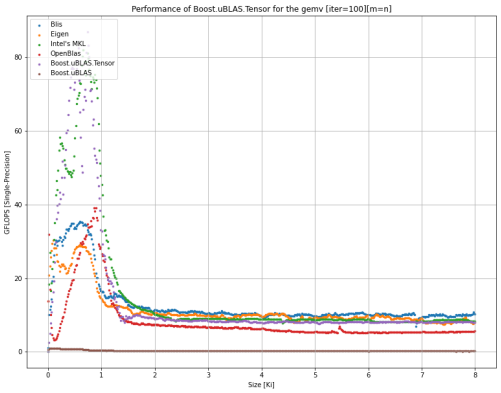
Figure 4.2: Row-Major Matrix-Vector Block Diagram



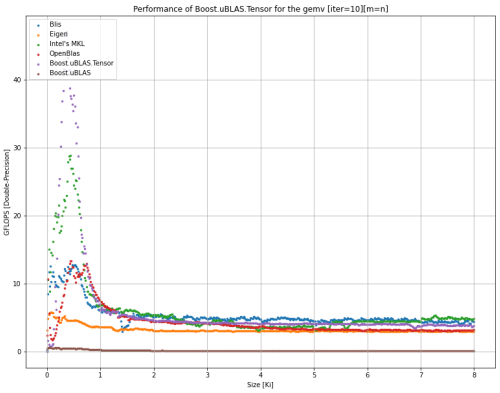
4.3 Performance Plots and Speedup Summary For Column-Major

Performance measurements of ?gemv implementations

(a) Single-Precision

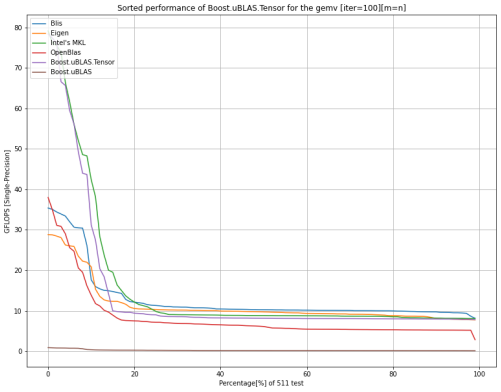


(b) Double-Precision

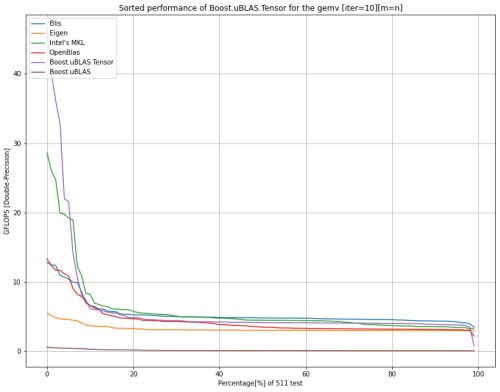


Sorted performance measurements of ?gemv implementations

(a) Single-Precision

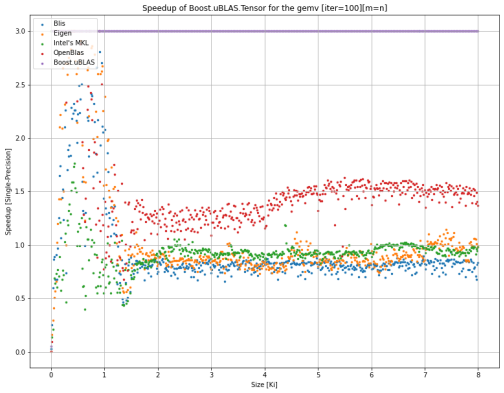


(b) Double-Precision

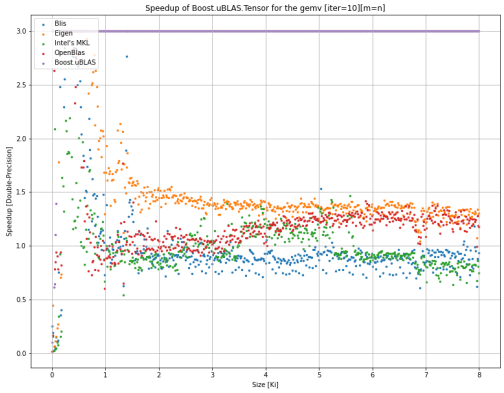


Comparison of the Boost.uBLAS.Tensor ?gemv implementation

(a) Single-Precision

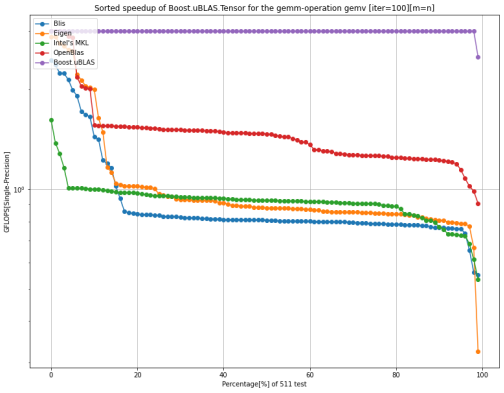


(b) Double-Precision

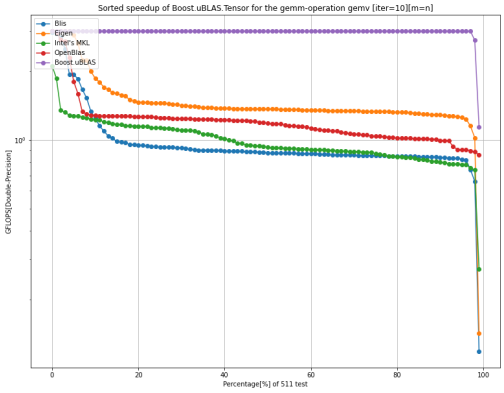


Comparison of the Boost.uBLAS.Tensor ?gemv implementation [semilogy]

(a) Single-Precision

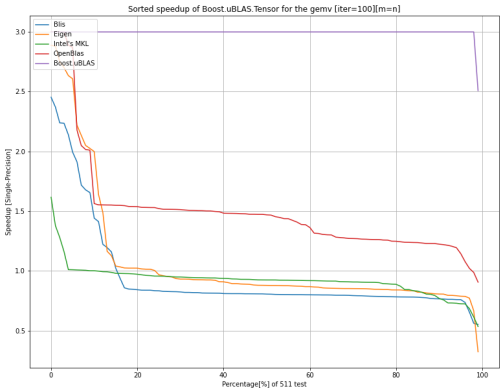


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

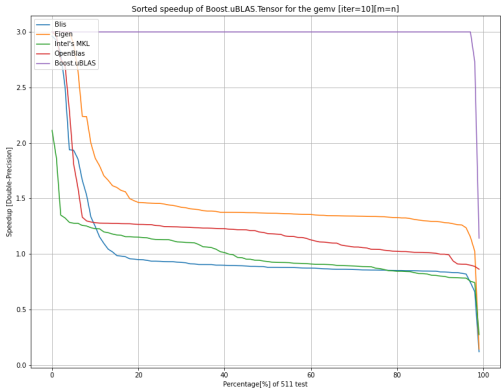


Table 4.1: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	99
OpenBLAS	97	9
Eigen	24	9
Blis	98	98
Intel's MKL	10	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	3	0
Eigen	76	0
Blis	86	0
Intel's MKL	91	0

Table 4.2: Speedup Summary For Double-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	98
OpenBLAS	89	4
Eigen	98	8
Blis	14	3
Intel's MKL	40	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	9	0
Eigen	1	1
Blis	86	1
Intel's MKL	60	2

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	107.22	14.4079
Boost.uBLAS	1.02227	0.271
Intel's MKL	87.6178	15.789
OpenBLAS	44.5053	8.4120
Blis	36.8396	12.9113
Eigen	32.9823	11.4342

Table 4.4: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	52.144	6.2511
Boost.uBLAS	0.6601	0.1828
Intel's MKL	33.6268	6.0042
OpenBLAS	16.1558	4.4325
Blis	17.4944	5.41638
Eigen	6.30737	3.26405

Table 4.5: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	18.2099	2.4469
Boost.uBLAS	0.17361	0.0460
Intel's MKL	14.8807	2.6816
OpenBLAS	7.5586	1.4286
Blis	6.2567	2.1928
Eigen	5.6016	1.9419

Table 4.6: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	17.711	2.1233
Boost.uBLAS	0.2242	0.0620
Intel's MKL	11.4221	2.039
OpenBLAS	5.4877	1.505
Blis	5.9423	1.8398
Eigen	2.14244	1.1087

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

Implementation	**Max**	**Average**
Boost.uBLAS	104.884	53.092
Intel's MKL	1.2237	0.9125
OpenBLAS	2.4091	1.7127
Blis	2.9104	1.11590
Eigen	3.2508	1.26006

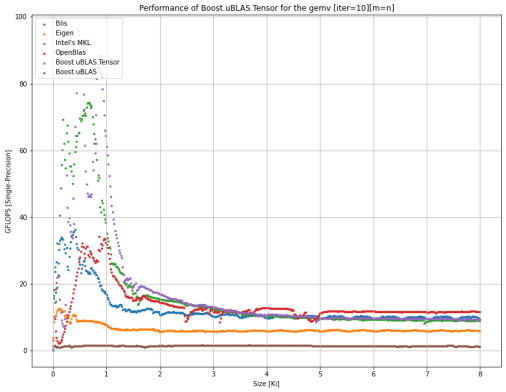
Table 4.8: Speedup(Boost.uBLAS.Tensor) For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS	78.9937	34.195
Intel's MKL	1.5506	1.0411
OpenBLAS	3.2275	1.4102
Blis	2.9806	1.1541
Eigen	8.2671	1.9151

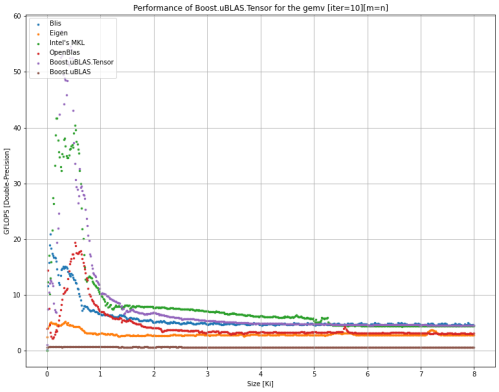
4.5 Performance Plots and Speedup Summary For Row-Major

Performance measurements of ?gemv implementations

(a) Single-Precision

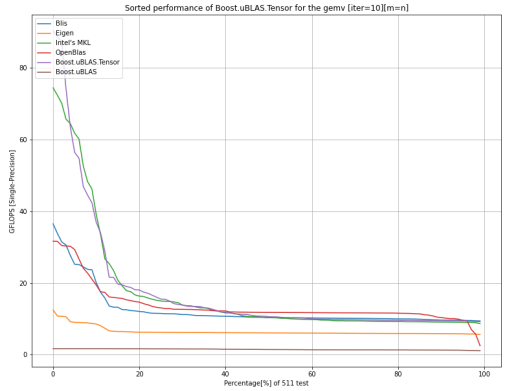


(b) Double-Precision

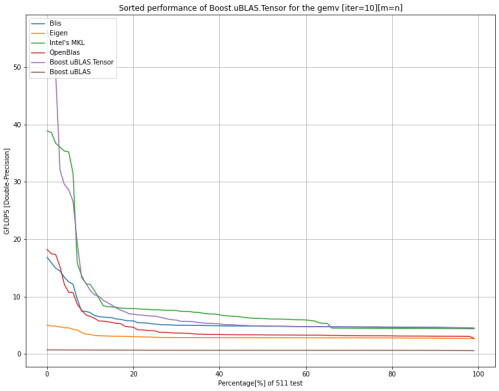


Sorted performance measurements of ?gemv implementations

(a) Single-Precision

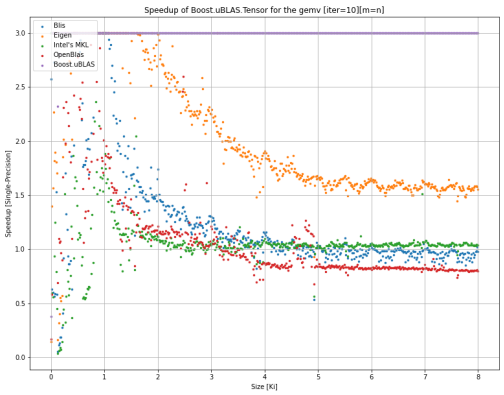


(b) Double-Precision

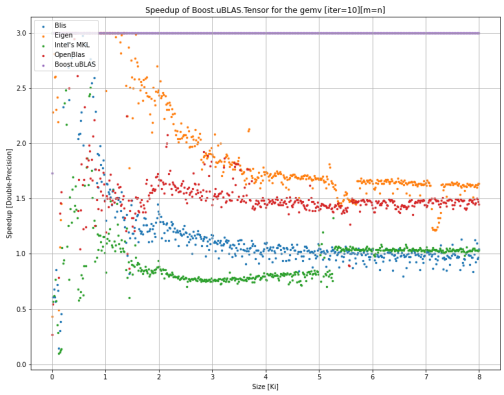


Comparison of the Boost.uBLAS.Tensor ?gemv implementation

(a) Single-Precision

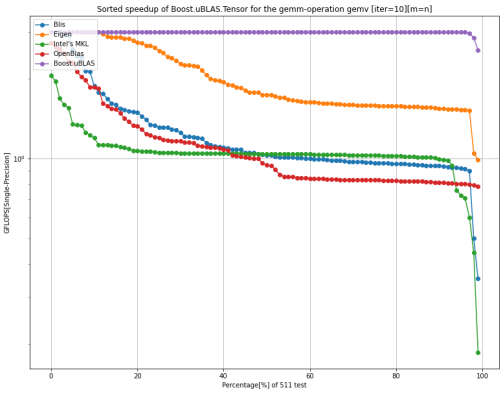


(b) Double-Precision

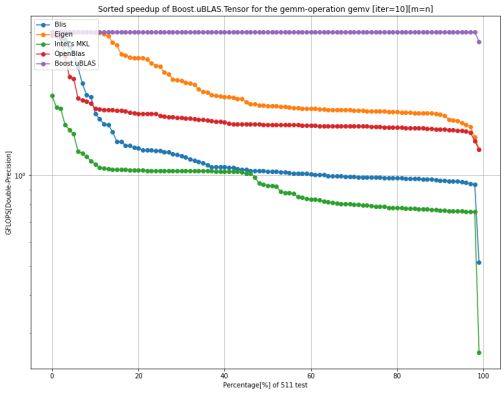


Comparison of the Boost.uBLAS.Tensor ?gemv implementation [semilogy]

(a) Single-Precision

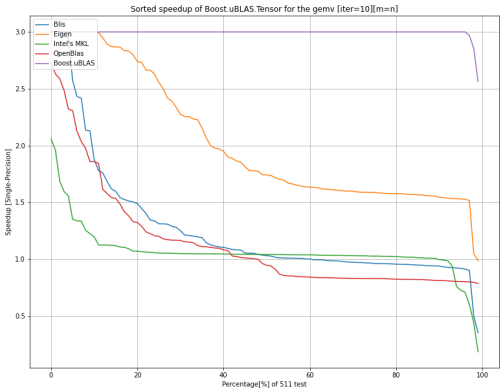


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

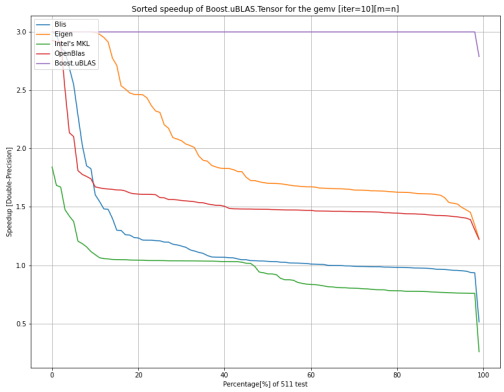


Table 4.9: Speedup Summary For Single-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	99
OpenBLAS	47	7
Eigen	98	36
Blis	60	9
Intel's MKL	89	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	54	0
Eigen	2	0
Blis	40	1
Intel's MKL	11	1

Table 4.10: Speedup Summary For Double-Precision

Implementation	**Speedup $\geq$ 1** [%]	**Speedup $\geq$ 2** [%]
Boost.uBLAS	99	99
OpenBLAS	99	5
Eigen	99	33
Blis	63	7
Intel's MKL	46	0

Implementation	**Speed-down $\geq$ 1** [%]	**Speed-down $\geq$ 2** [%]
Boost.uBLAS	0	0
OpenBLAS	0	0
Eigen	0	0
Blis	36	0
Intel's MKL	53	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	130.099	17.812
Boost.uBLAS	1.7571	1.4884
Intel's MKL	87.2948	17.253
OpenBLAS	41.5645	13.712
Blis	42.0662	12.5435
Eigen	13.0251	6.5156

Table 4.12: GFLOPS For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	77.7866	8.096
Boost.uBLAS	0.7437	0.666
Intel's MKL	49.3442	8.396
OpenBLAS	25.0278	4.4855
Blis	23.9759	5.7617
Eigen	5.57867	3.0228

Table 4.13: Utilization[%] For Single-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	22.095	3.0253
Boost.uBLAS	0.2984	0.2527
Intel's MKL	14.825	2.930
OpenBLAS	7.0591	2.3289
Blis	7.1443	2.1303
Eigen	2.2121	1.1066

Table 4.14: Utilization[%] For Double-Precision

Implementation	**Max**	**Average**
Boost.uBLAS.Tensor	26.4220	2.750
Boost.uBLAS	0.2526	0.226
Intel's MKL	16.7609	2.8520
OpenBLAS	8.5012	1.52362
Blis	8.1439	1.9571
Eigen	1.8949	1.0267

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

Implementation	**Max**	**Average**
Boost.uBLAS	74.041	11.9676
Intel's MKL	1.4903	1.032
OpenBLAS	3.1300	1.2990
Blis	3.0927	1.4200
Eigen	9.9883	2.7338

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

Implementation	**Max**	**Average**
Boost.uBLAS	104.589	12.1426
Intel's MKL	1.5764	0.96423
OpenBLAS	3.108	1.8049
Blis	3.2443	1.40512
Eigen	13.9435	2.6783

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 9: 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 × N_r} ← 0
    half_{N_r} ← N_r / 2
    for k ← 0 to k_c by 1 do
        for j ← 0 to half_{N_r} by 1 do
            #pragma omp simd
            for i ← 0 to M_r by 1 do
                buff[j × M_r + i] ← buff[j × M_r + i] + bk[j] × ak[i]
                buff[(j + half_{N_r}) × M_r + i] ← buff[j × M_r + i] + bk[j + half_{N_r}] × ak[i]
            end
        end
    end
    // Copy the buff into the output Matrix c
end
```

Algorithm 10: 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 ← max(w_c[0], w_c[1])
    for j ← 0 to N by N_r do
        a_i ← a
        b_i ← b + j × K
        c_i ← c + j × w_c[1]
        j_b ← max(N_r, N − j)
        for i ← 0 to M by M_r do
            a_k ← ai + i × K
            b_k ← bi
            c_k ← ci + i × w_c[o]
            i_b ← max(M_r, M − i)
            simd_loop(c_k, ldc, a_k, b_k, K, i_b, j_b)
        end
    end
end
```

Algorithm 11: 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

$$M_r N_r \geq N_{VEC} L_{effective} N_{effective} \tag{5.3}$$

To find the effective latency and throughput, we cannot directly take the algebraic sum for the throughput, which will give the wrong result. Instead, taking the algebraic sum of the latency gives the actual effective latency, and taking the harmonic mean gives the effective throughput.

$$L_{effective} = 2L_{L/S} + L_{VFMA}$$
$$N_{effective} = \frac{3}{\frac{1}{N_{VEC}} + \frac{2}{N_{L/S}}}$$

$$M_r = \lceil \frac{\sqrt{N_{VEC} L_{effective} N_{effective}}}{N_{VEC}} \rceil N_{VEC}$$
$$N_r = \lceil \frac{N_{VEC} L_{effective} N_{effective}}{M_r} \rceil$$

Note that our algorithm relies on N_r being an even number. Therefore, if we get odd, we have chosen the following even number, but we cannot choose the last even number because our lower limit ¡¡¡¡¡¡¡ HEAD is the solution to the equation. ======= is the solution of the equation. ¿¿¿¿¿¿¿ 3155ab277c10bf96acca5eadef8fd7c7fe88dbff

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.

$$M_r = \lceil \frac{\sqrt{9N_{VEC}}}{N_{VEC}} \rceil N_{VEC}$$
$$N_r = \lceil \frac{9N_{VEC}}{M_r} \rceil$$

If we solve N_r here, we get 5 which is an odd number. Therefore, we choose 6 because it is the following even number.

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

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

Table 5.4: Tuning Parameter K_B

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

Substituting the C_{A_r} into the equation 5.5.

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}.

$$M_B K_B S_{data} = C_{A_c} N_{L_2} C_{L_2}$$
$$N_r K_B S_{data} = C_{B_r} N_{L_2} C_{L_2}$$

Now, dividing the both equation and we get

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

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

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.

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

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

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

Table 5.5: Tuning Parameter M_B

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

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.

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.

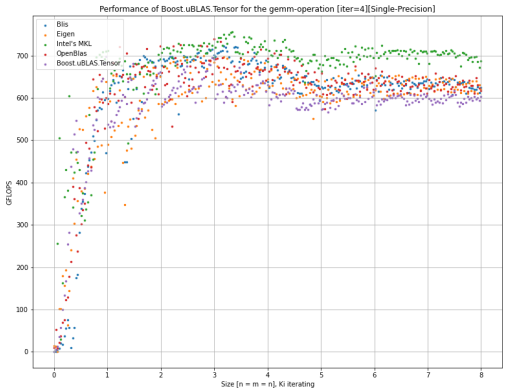
Table 5.6: Tuning Parameter N_B

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

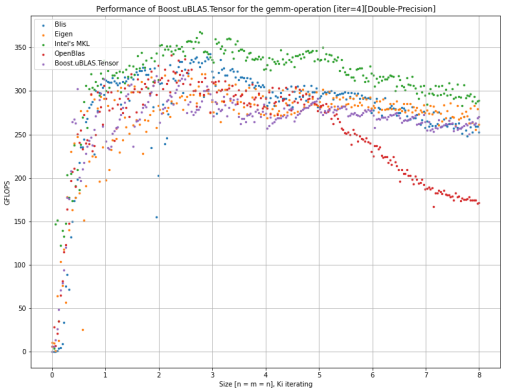
5.3 Performance Plots and Speedup Summary

Performance measurements of ?gemm implementations

(a) Single-Precision

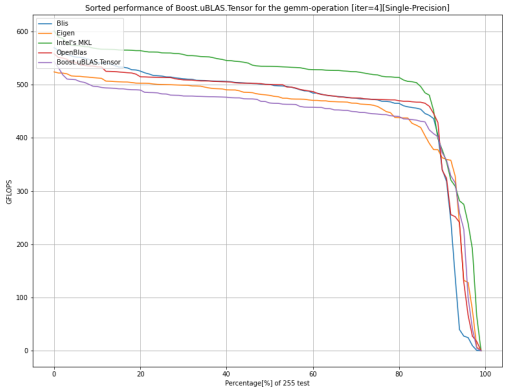


(b) Double-Precision

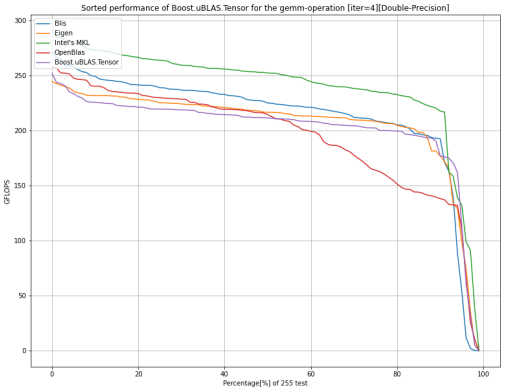


Sorted performance measurements of ?gemm implementations

(a) Single-Precision

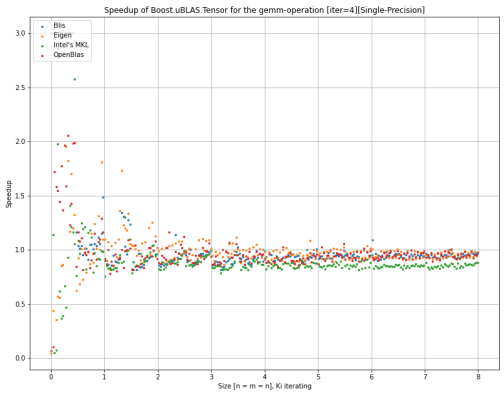


(b) Double-Precision

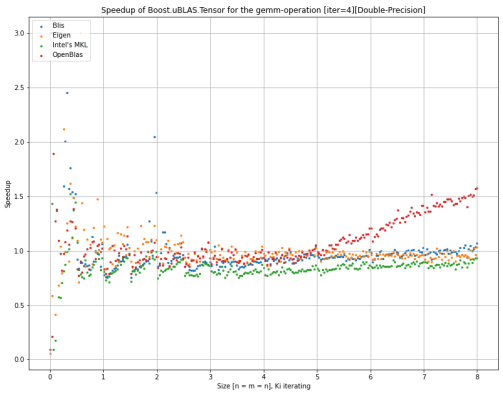


Comparison of the Boost.uBLAS.Tensor ?gemm implementation

(a) Single-Precision

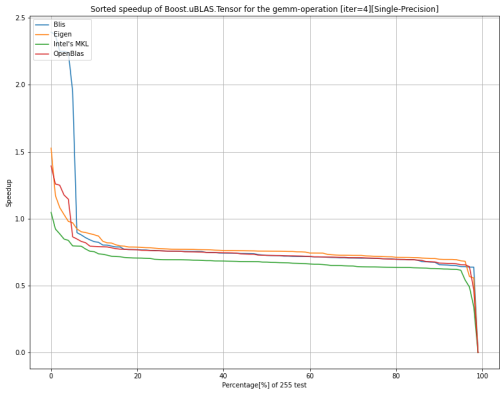


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

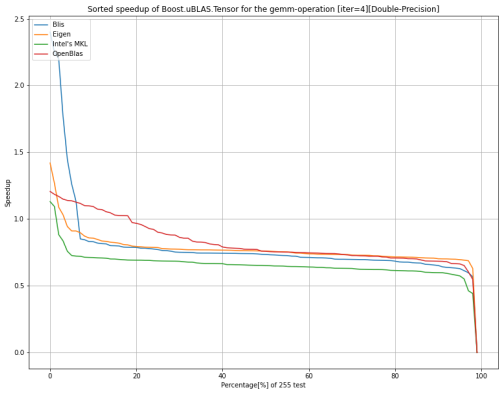


Table 5.7: Speedup Summary For Single-Precision

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.

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

$$C = A^T$$

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

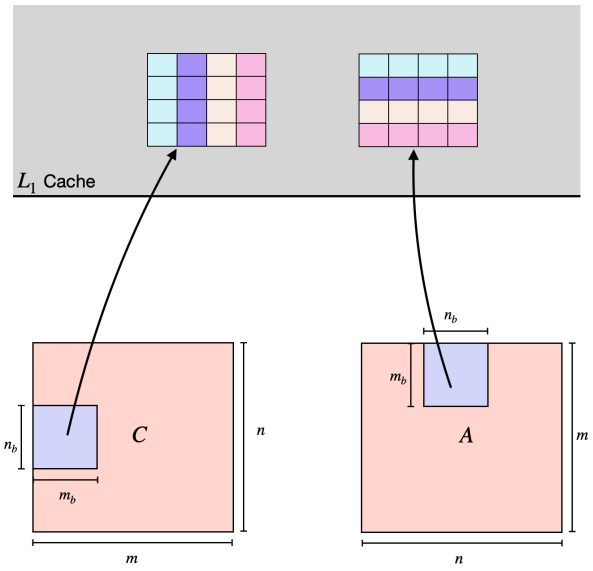
Involution Property: $(A^T)^T = A$

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 12: 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 ← 0 to m by 1 do
        cj ← c + wc[1] × i
        aj ← a + wa[0] × i
        #pragma omp simd
         for j ← 0 to n by 1 do
         |  cj[wc[0] × j] ← aj[wa[1] × j]
        end
    end
end
```

Algorithm 13: 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

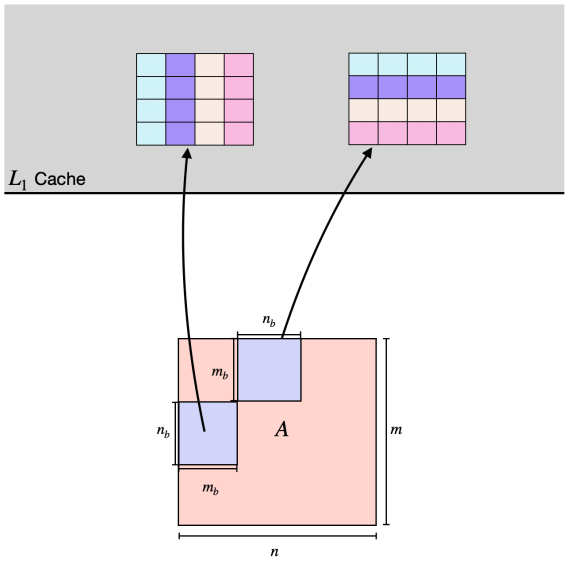
6.1.2 In-Place Transpose

Algorithm 14: In-Place Matrix Transpose SIMD Function

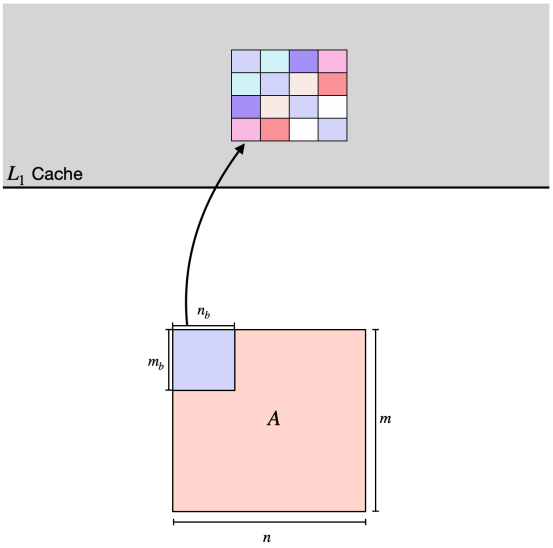
// c is the pointer to the output Matrix
// wc is the pointer to the strides of the output Matrix
// a is the pointer to the input Matrix A
// wa is the pointer to the strides of the input Matrix A
// m is the row-block of the Matrix A
// n is the col-block of the Matrix A
// is_same_block tells us if 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 15: 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, $m_b = n_b = B$.

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

$$B = \sqrt{\frac{S_{L_1}}{2S_{data}}}$$

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

$$B = \sqrt{\frac{2^k}{2}}$$

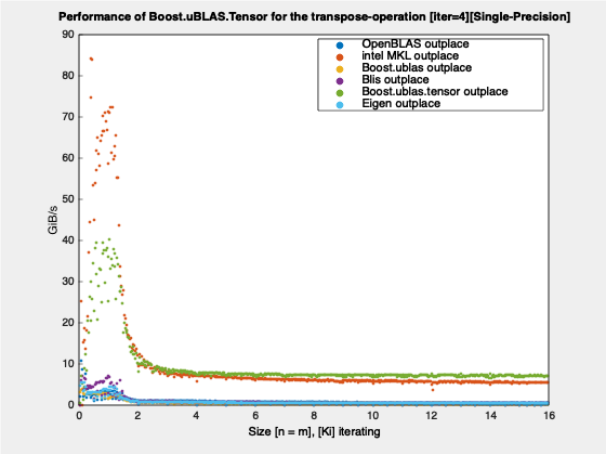
$$B = \sqrt{2^{k-1}}$$

$$B = 2^{\lfloor \frac{k-1}{2} \rfloor}$$

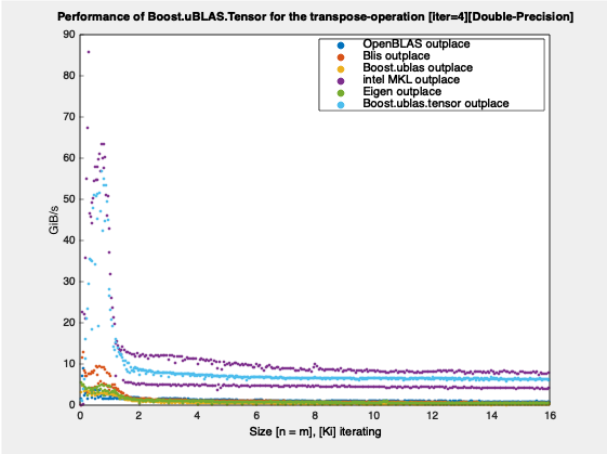
6.3 Performance Plots and Speedup Summary For Out-Of-Place

Performance measurements of transpose implementations

(a) Single-Precision

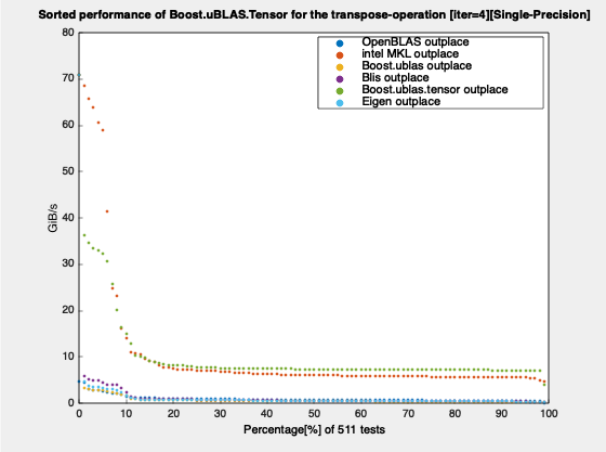


(b) Double-Precision

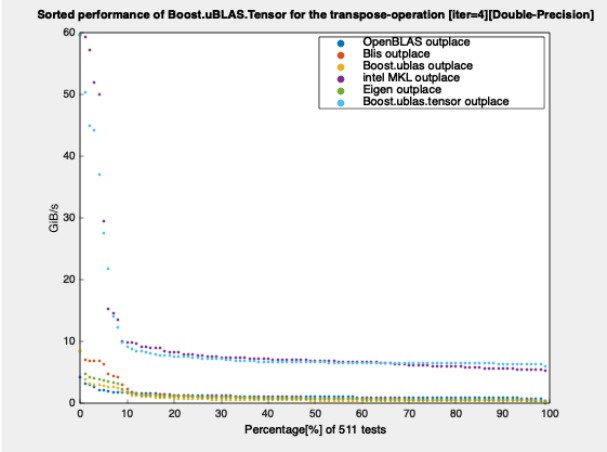


Sorted performance measurements of transpose implementations

(a) Single-Precision

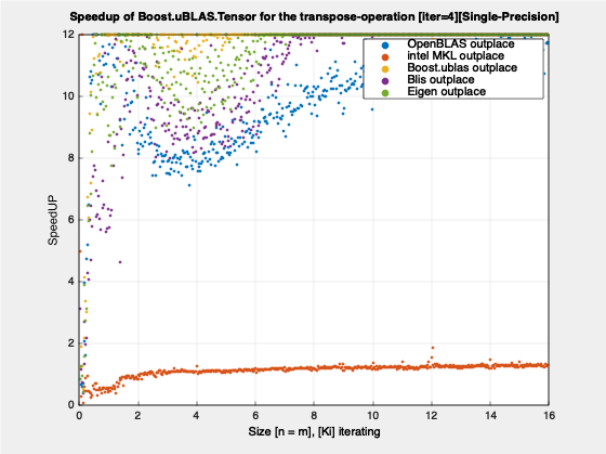


(b) Double-Precision

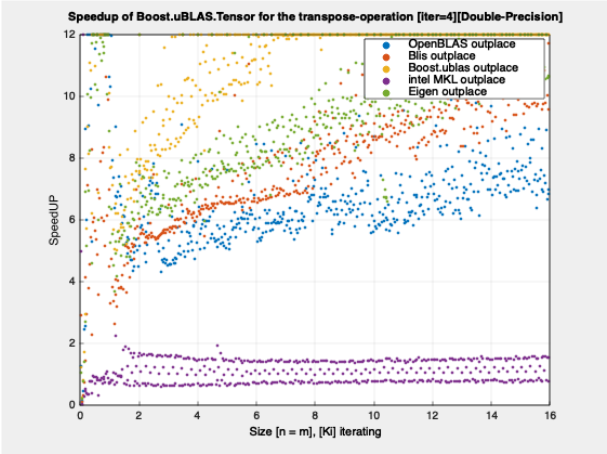


Comparison of the Boost.uBLAS.Tensor transpose implementation

(a) Single-Precision

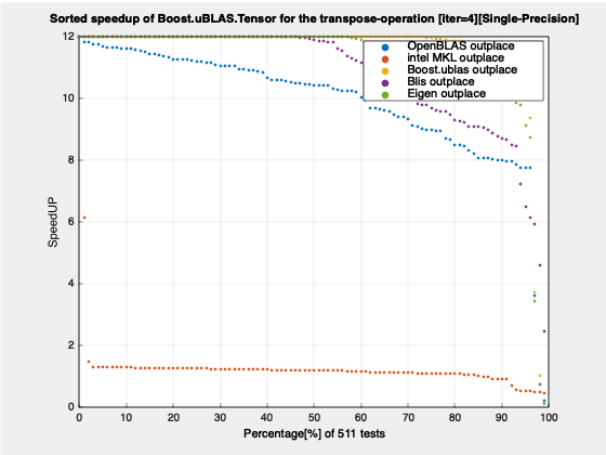


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

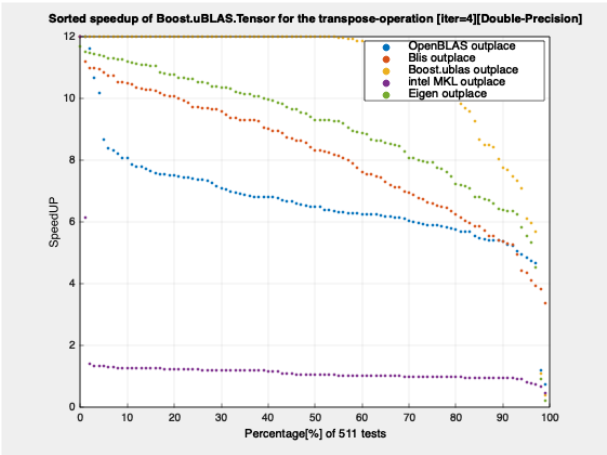


Table 6.1: Speedup Summary For Single-Precision

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

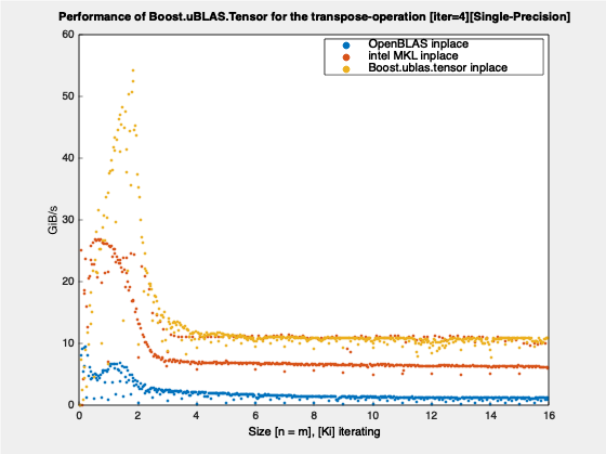
Table 6.6: Speedup(Boost.uBLAS.Tensor) For Double-Precision

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

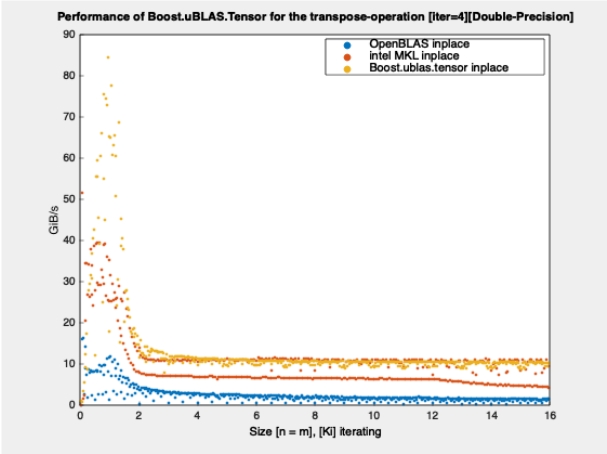
6.4 Performance Plots and Speedup Summary For In-Place

Performance measurements of transpose implementations

(a) Single-Precision

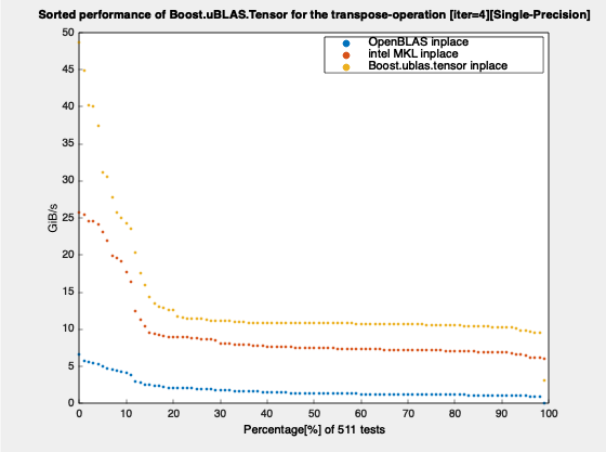


(b) Double-Precision

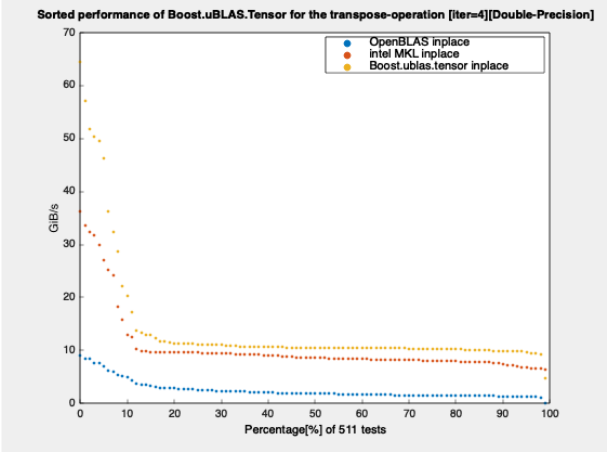


Sorted performance measurements of transpose implementations

(a) Single-Precision

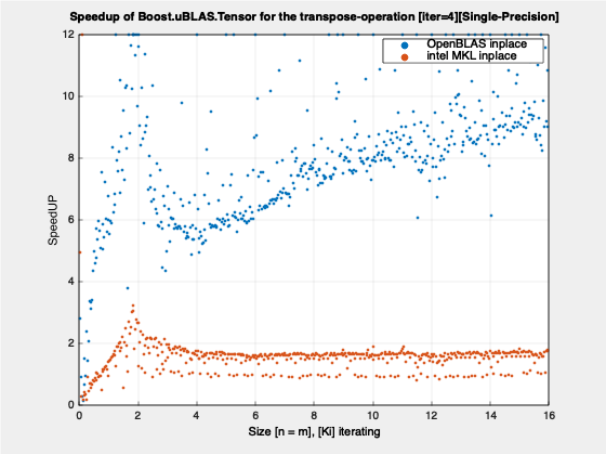


(b) Double-Precision

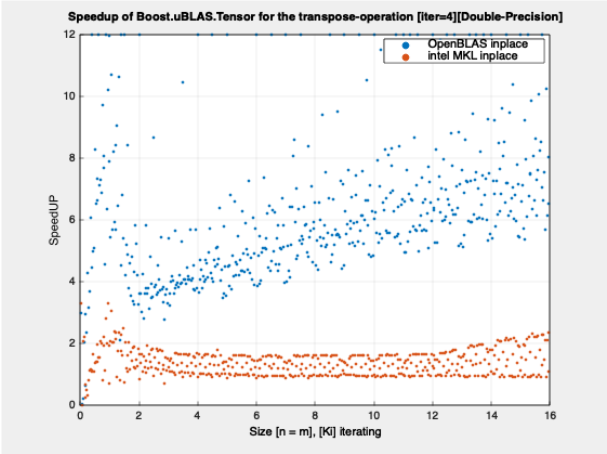


Comparison of the Boost.uBLAS.Tensor transpose implementation

(a) Single-Precision

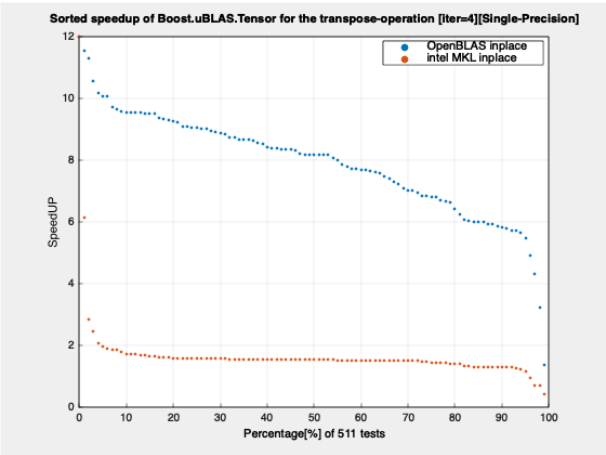


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

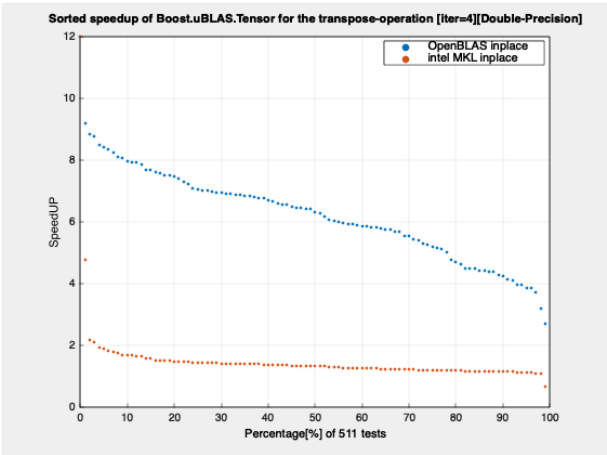


Table 6.7: Speedup Summary For Single-Precision

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`.