Implementing BLAS Operation using OpenMP

Amit Singh Cem Bassoy

March 26, 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	Apple clang version 12.0.0 (clang-1200.0.32.29)
Compiler Flags	-march=native -ffast-math -Xclang -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

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

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

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

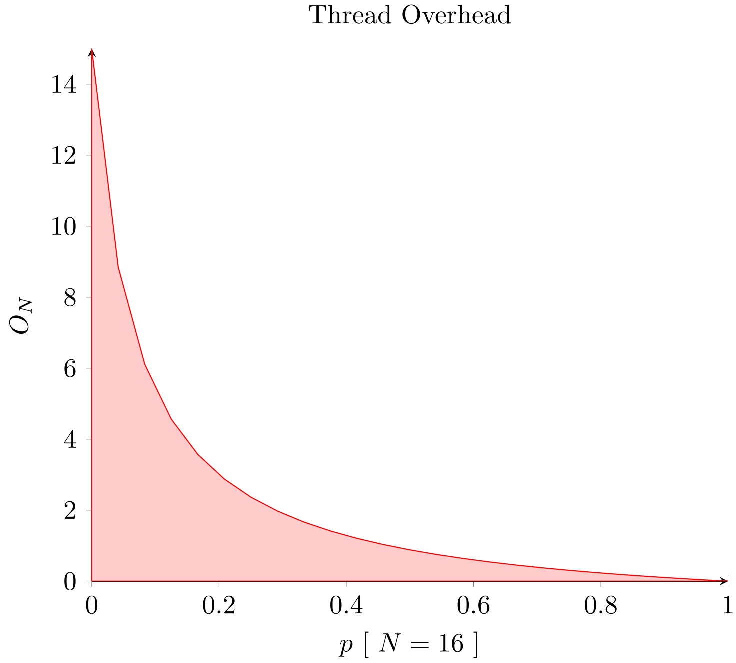
Therefore, $O_N \in [0, N-1)$. We can theorise that p must be a function of Block (B_1).

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

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

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

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



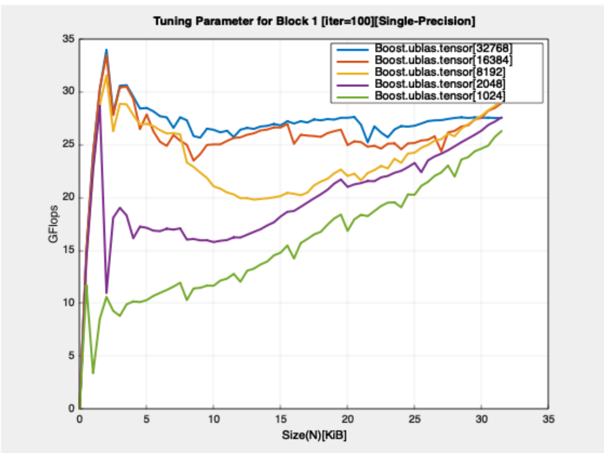
The B_1's lower bound found to be the number of elements that can be fit inside the L_1 cache through the experimental data. This phenomenon happens because the vectors' elements cannot fit inside the cache, and cache misses increase, dominating the thread overhead.

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

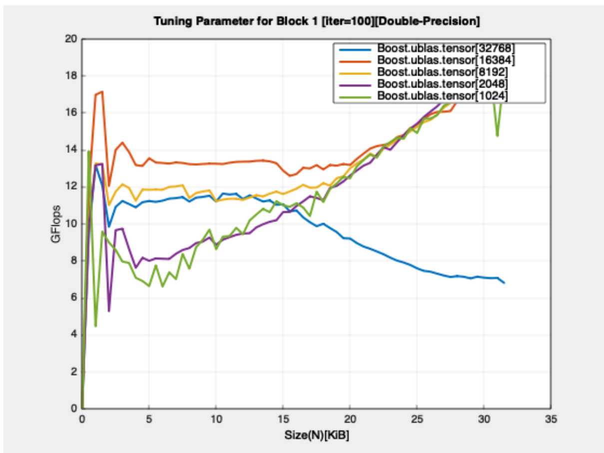
However, the optimal block size for this section found to be twice the vectors' elements that can put inside the cache.

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

Experimental Data for B_1



(a) Single-Precision



(b) Double-Precision

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

2.2.2 The Second Section

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

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

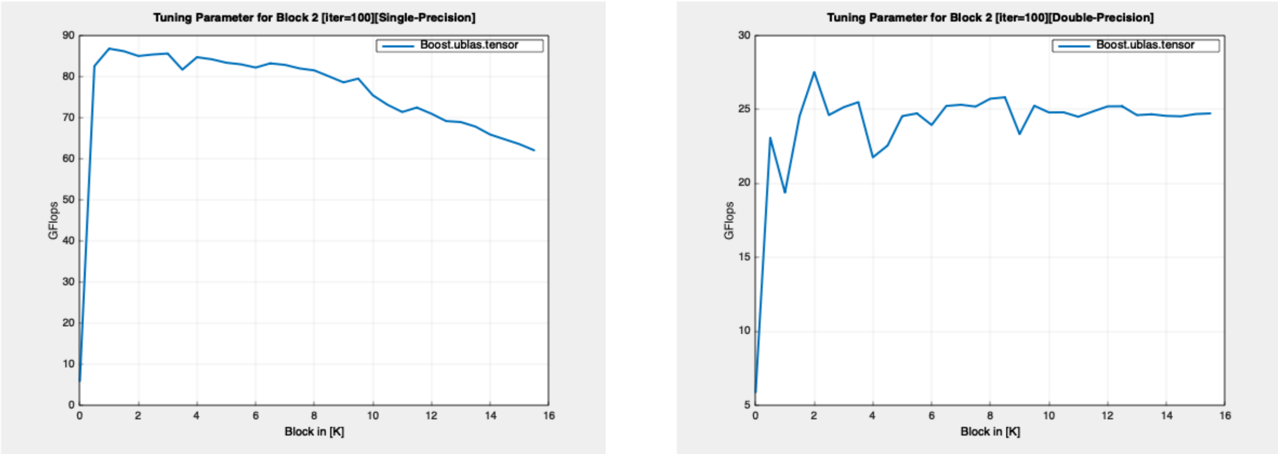
From the equation 2.1, we can infer that the block of both vectors must be equal to give the optimal block size.

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

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

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

Experimental Data for B_2



(a) Single-Precision

(b) Double-Precision

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

2.2.3 The Third Section

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

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

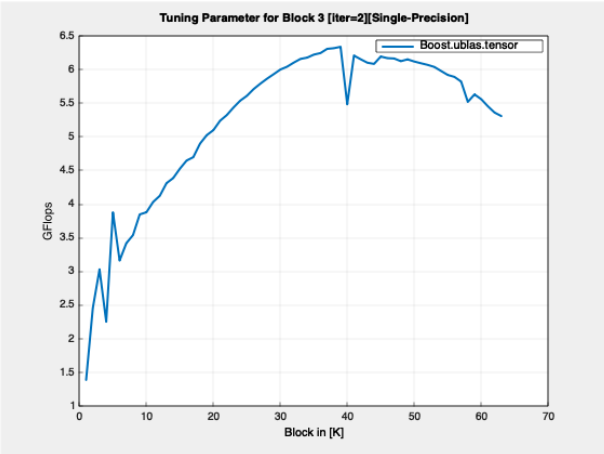
From the equation 2.1, we can infer that the block of both vectors must be equal to give the optimal block size.

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

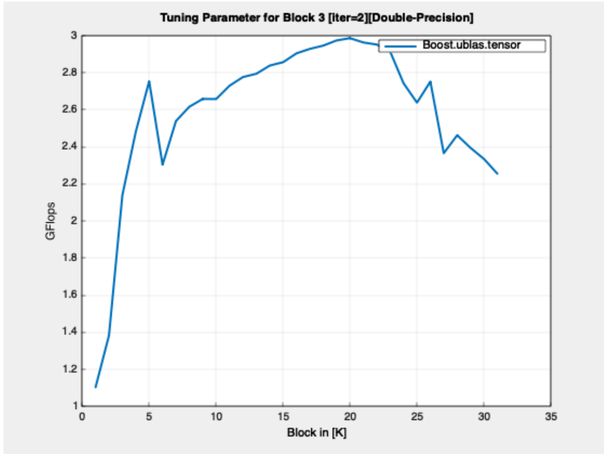
$$S_{L_2} = S_{data}(B_3 + B_3)$$
$$S_{L_2} = 2S_{data}B_3$$

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

Experimental Data for B_2



(a) Single-Precision



(b) Double-Precision

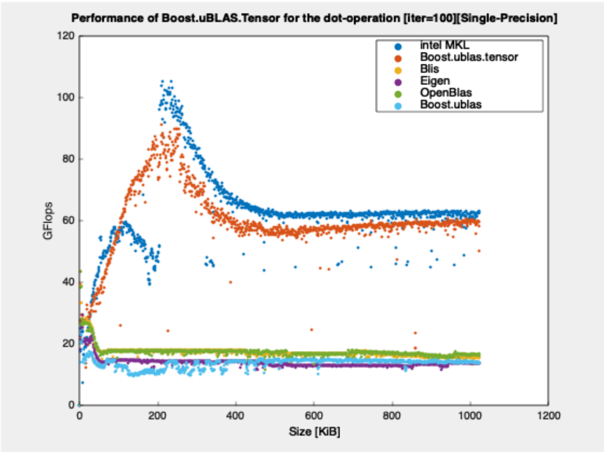
	Block[Single]	**Block[Double]**
Theoretical Optimal	32768 (32K)	16384 (16K)
Experimental Optimal	$\geq$ 30K and $\leq$ 40K	$\geq$ 16K and $\leq$ 20K

2.3 Performance Plots and Speedup Summary

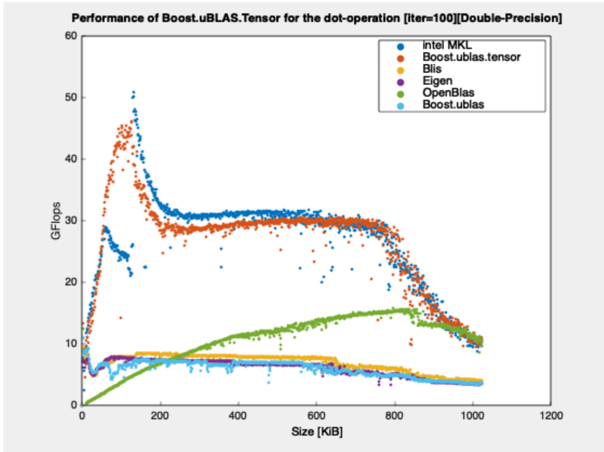
2.3.1 Range[Start: 2, End: 2^{20}, Step: 1024]

Performance measurements of ?dot implementations

(a) Single-Precision

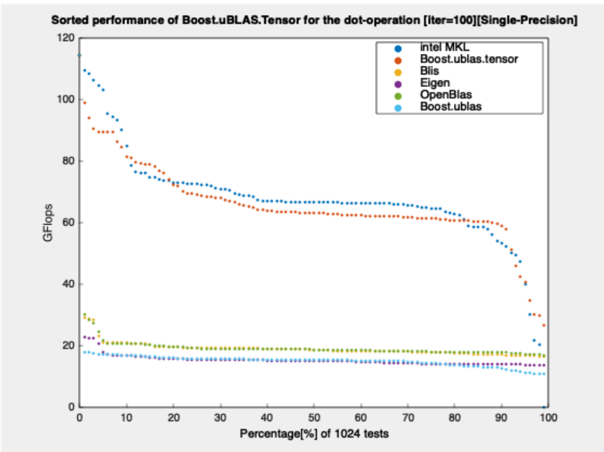


(b) Double-Precision

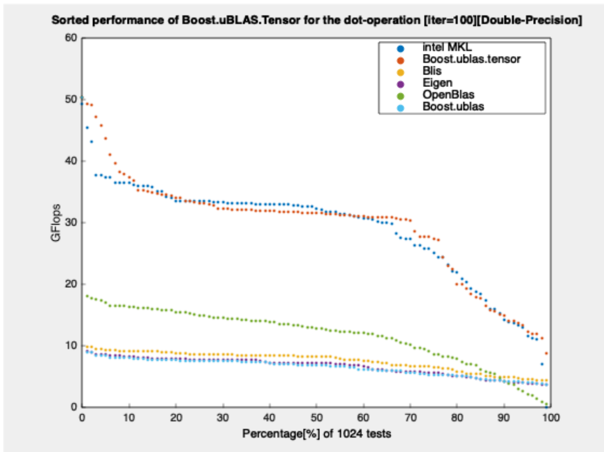


Sorted performance measurements of ?dot implementations

(a) Single-Precision

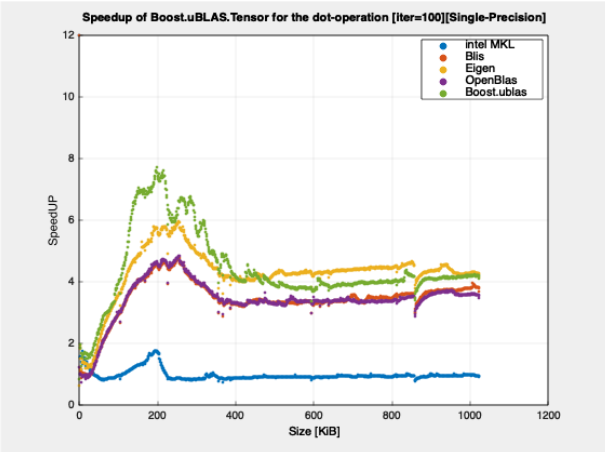


(b) Double-Precision

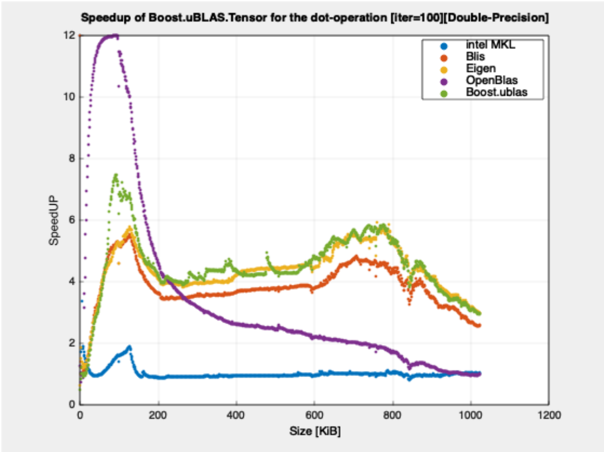


Comparison of the Boost.uBLAS.Tensor ?dot implementation

(a) Single-Precision

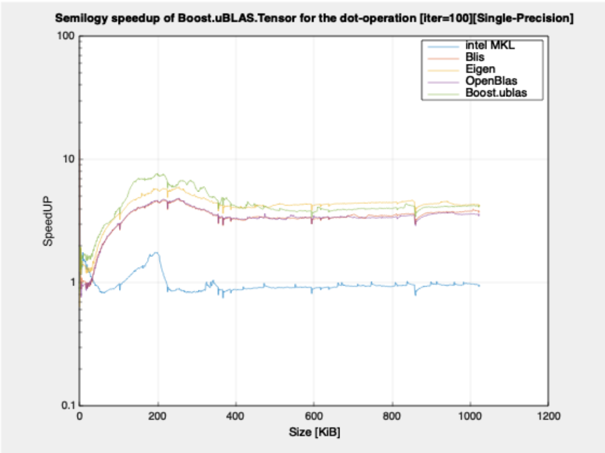


(b) Double-Precision

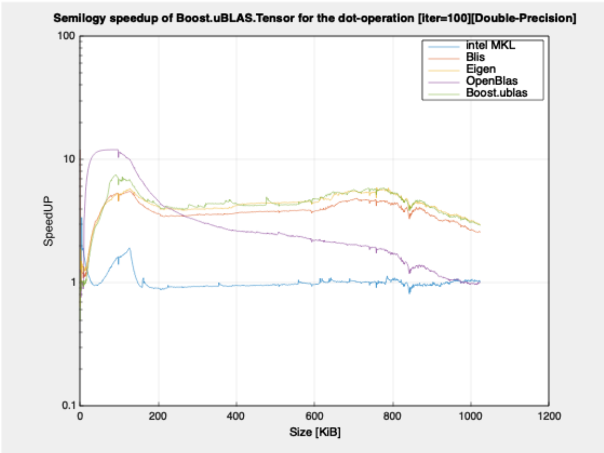


Comparison of the Boost.uBLAS.Tensor ?dot implementation [semilogy]

(a) Single-Precision

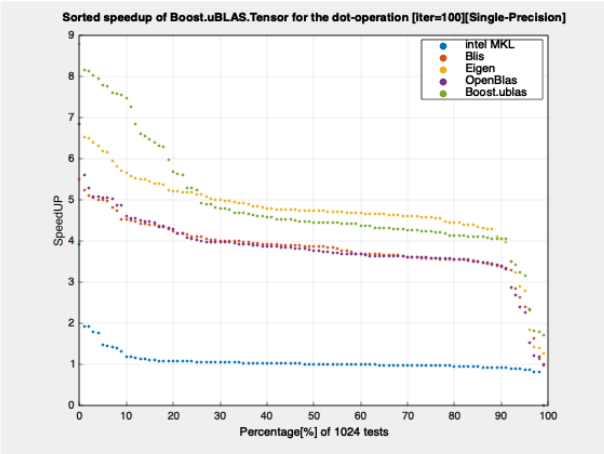


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

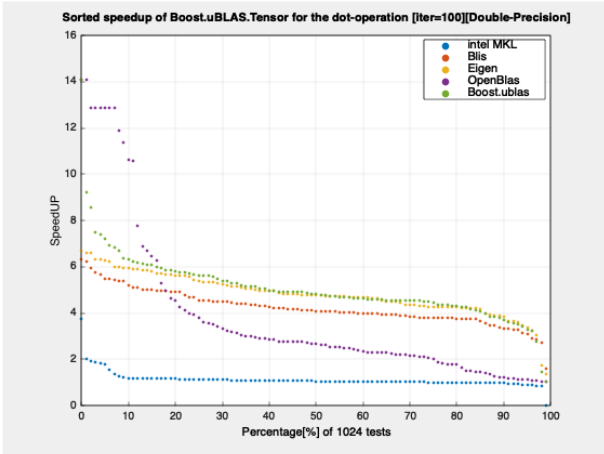


Table 2.1: Speedup Summary For Single-Precision

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

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

Table 2.2: Speedup Summary For Double-Precision

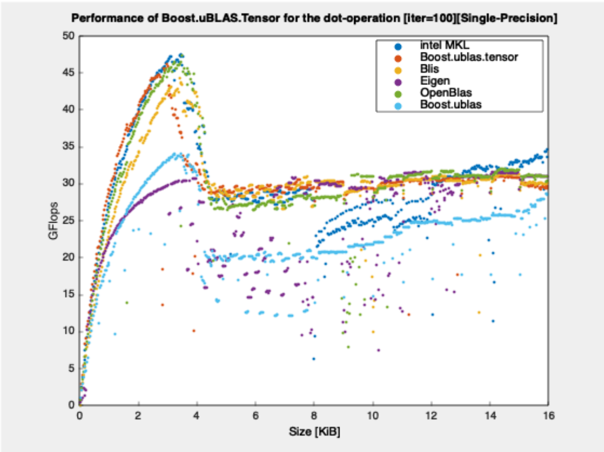
Implementation	**Speedup $\geq$ 1 [%]**	**Speedup $\geq$ 2 [%]**
Boost.uBLAS	99	97
OpenBLAS	99	75
Eigen	99	97
Blis	99	98
Intel's MKL	88	1

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

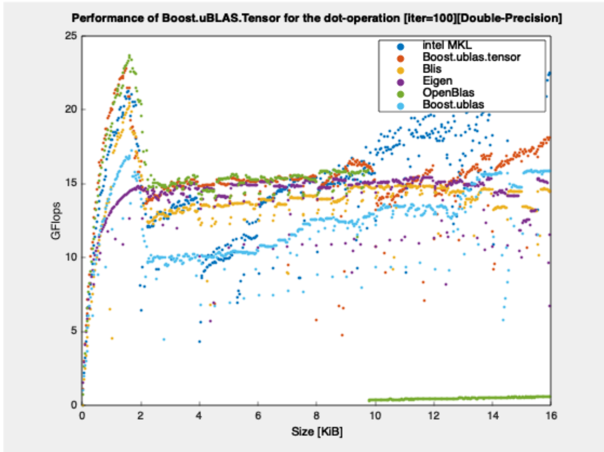
2.3.2 Range[Start: 32, End: 16382, Step: 32]

Performance measurements of ?dot implementations

(a) Single-Precision

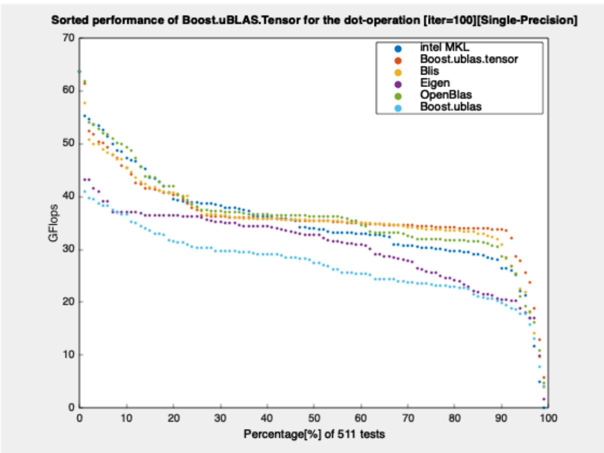


(b) Double-Precision

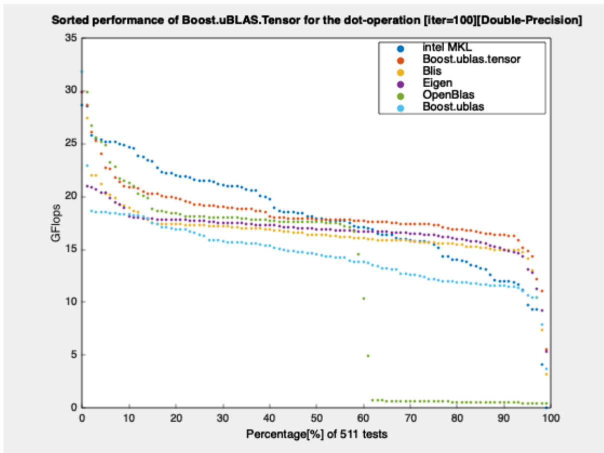


Sorted performance measurements of ?dot implementations

(a) Single-Precision

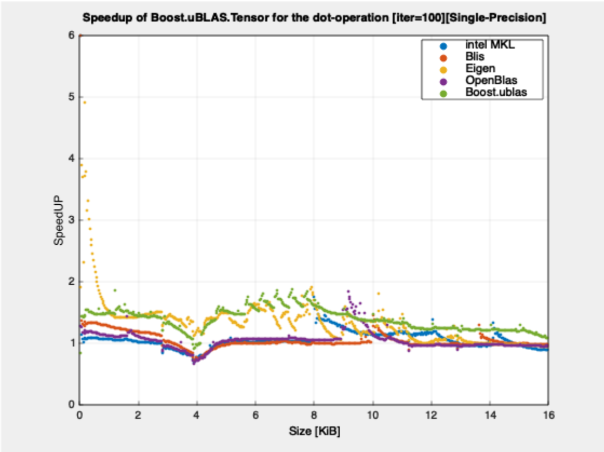


(b) Double-Precision

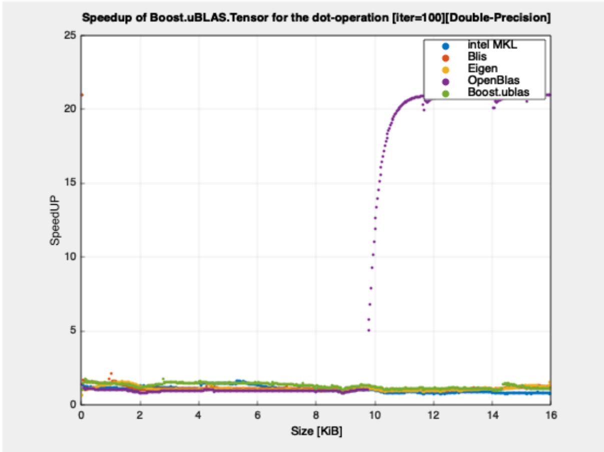


Comparison of the Boost.uBLAS.Tensor ?dot implementation

(a) Single-Precision

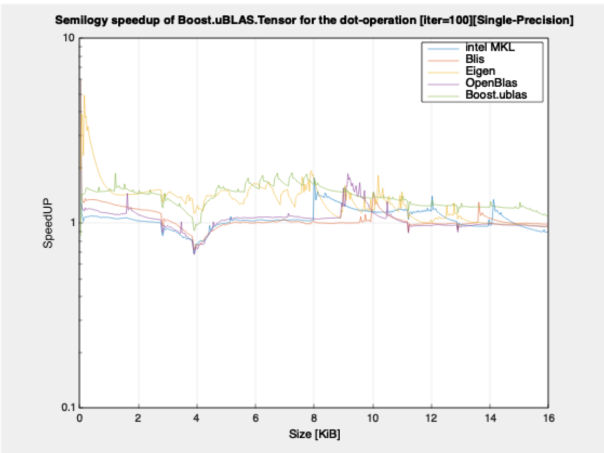


(b) Double-Precision

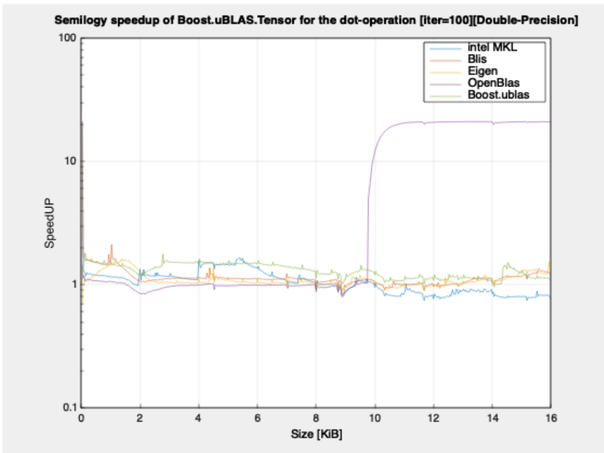


Comparison of the Boost.uBLAS.Tensor ?dot implementation [semilogy]

(a) Single-Precision

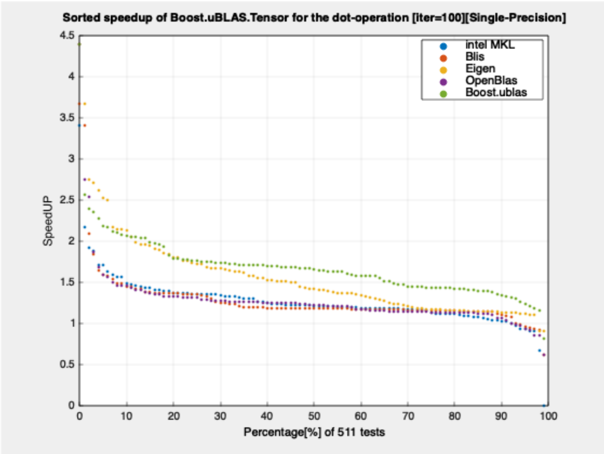


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

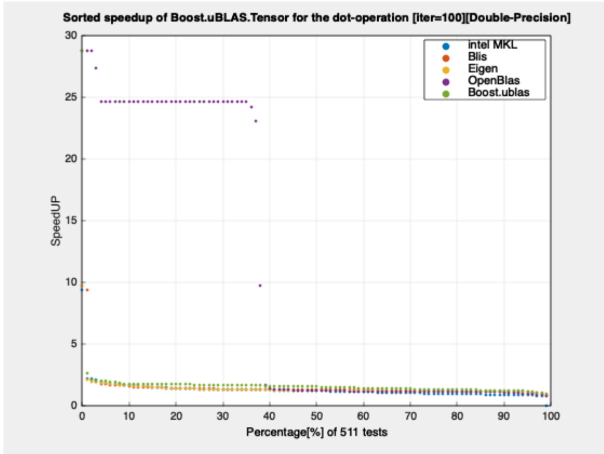


Table 2.3: Speedup Summary For Single-Precision

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

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

Table 2.4: Speedup Summary For Double-Precision

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

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

2.4 Performance Metrics

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

Table 2.5: GFLOPS For Single-Precision

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

Table 2.6: GFLOPS For Double-Precision

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

Table 2.7: Utilization[%] For Single-Precision

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

Table 2.8: Utilization[%] For Double-Precision

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

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

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

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

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

Chapter 3

Outer Product

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

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

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

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

Or

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

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

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

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

Taking the transpose on the both side in equation 3.2.

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

Transpose on the matrix addition is distributive, so we get

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

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

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

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

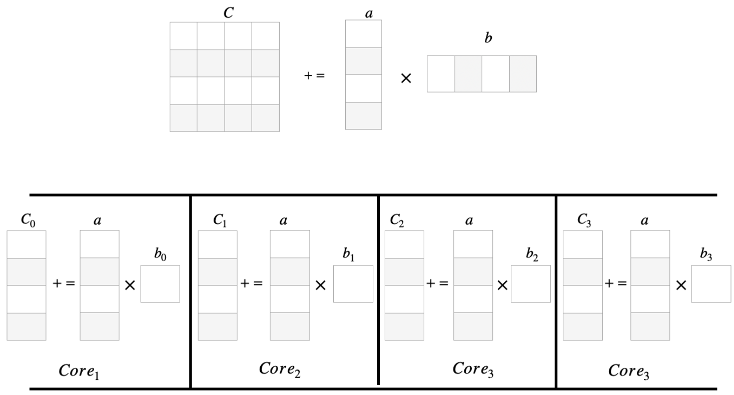
3.1 Calculating Number of Operations

Using equation 3.1, we fill the below table

Name	Number
Multiplication	$n \times m$
Addition	0

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

3.2 Algorithm



(a) Block Diagram

Algorithm 3: Outer Product SIMD Function

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

Algorithm 4: Vector-Vector Outer Product

```
Input: c, wc, a, na, b, nb, max_threads
// a and b are vectors
// c is the pointer to the output matrix
// na is the size of the vector a
// nb is the size of the vector b
// wc is the leading dimension of the matrix c
// max_threads is the user provided thread count
begin
    MinSize ← 256
    number_el_L2 ← ⌊S_L2 / S_data⌋
    upper_limit ← ⌊(number_el_L2 − na) / na⌋
    num_threads ← max(1, min(upper_limit, max_threads))
    omp_set_num_threads(num_threads)
    #pragma omp parallel for if(nb > MinSize)
     for i ← 0 to nb by 1 do
        aj ← a
        bj ← b + i
        cj ← b + i × wc
        outer_simd_loop(cj, aj, bj, na)
    end
end
```

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

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

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

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

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

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

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

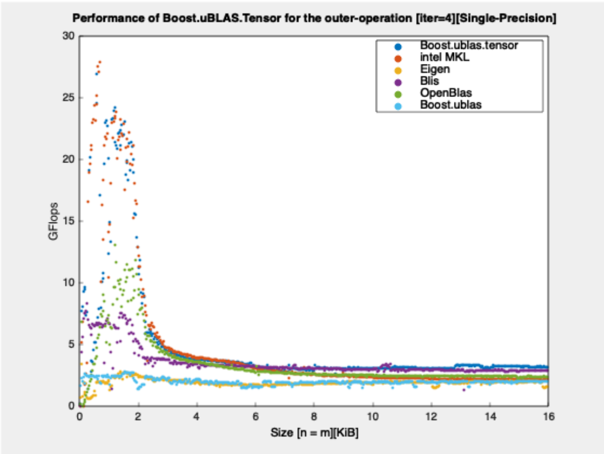
As the n increase, the block m_c decreases. We control the second vector's blocks through the outer loop, where we divide the second vector through threads, and each thread contains a single element. Therefore, the m_c block drops below the maximum allowed threads, we start to spawn m_c amount of threads.

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

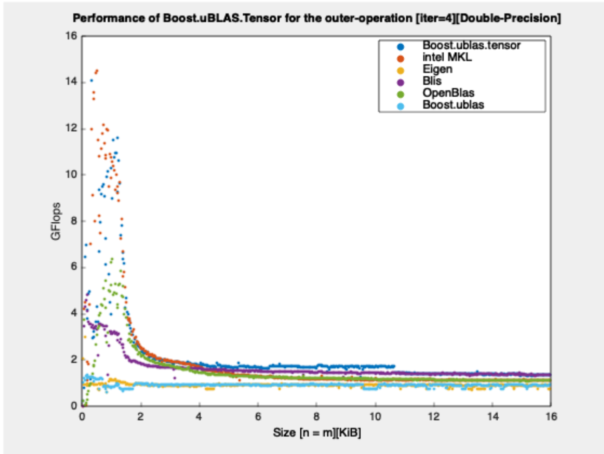
3.3 Performance Plots and Speedup Summary

Performance measurements of ?ger implementations

(a) Single-Precision

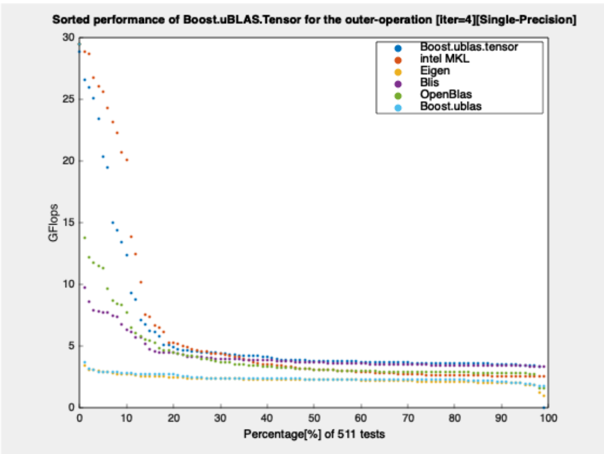


(b) Double-Precision

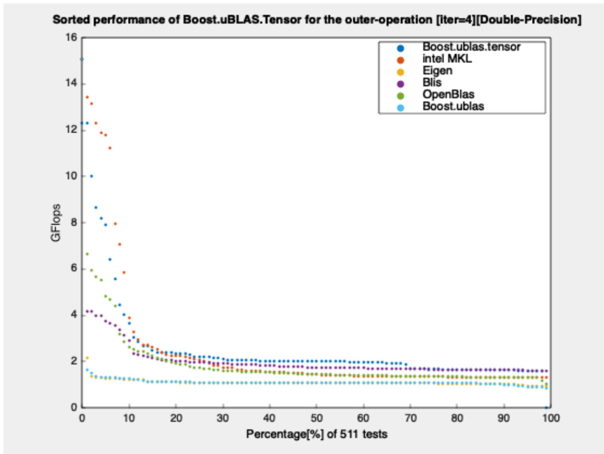


Sorted performance measurements of ?ger implementations

(a) Single-Precision

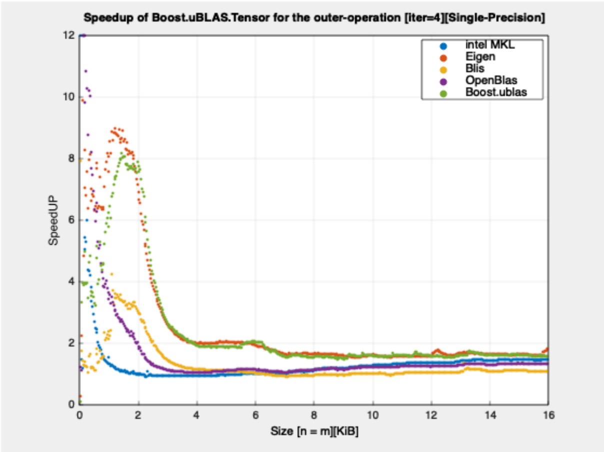


(b) Double-Precision

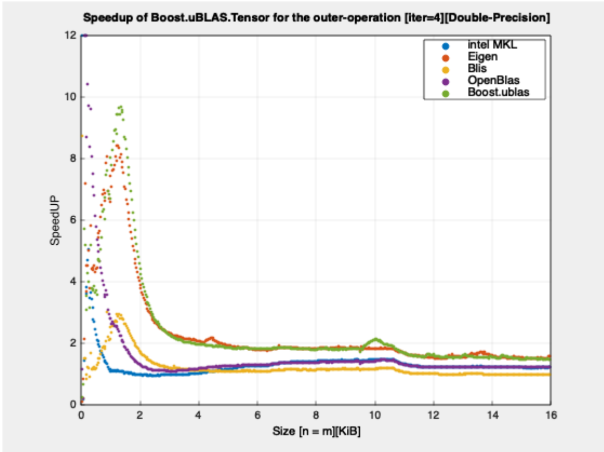


Comparison of the Boost.uBLAS.Tensor ?ger implementation

(a) Single-Precision

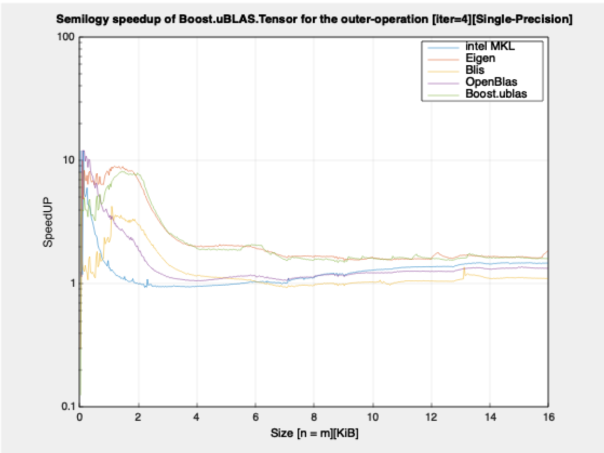


(b) Double-Precision

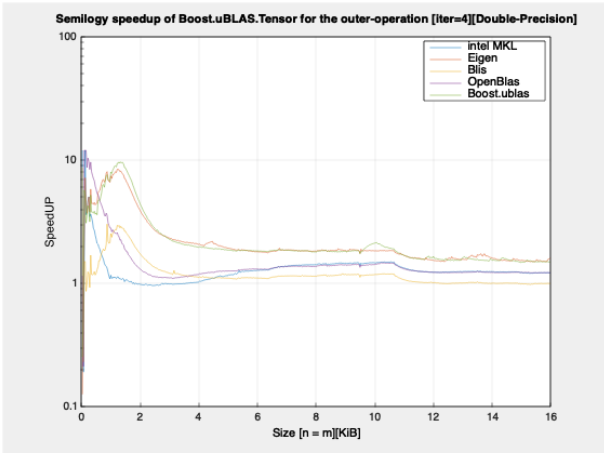


Comparison of the Boost.uBLAS.Tensor ?ger implementation [semilogy]

(a) Single-Precision

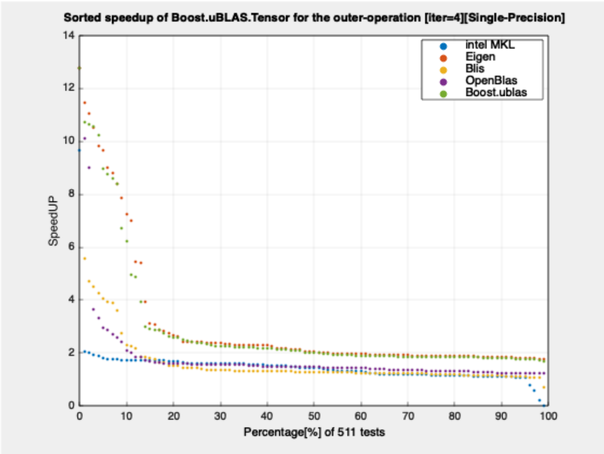


(b) Double-Precision



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

(a) Single-Precision



(b) Double-Precision

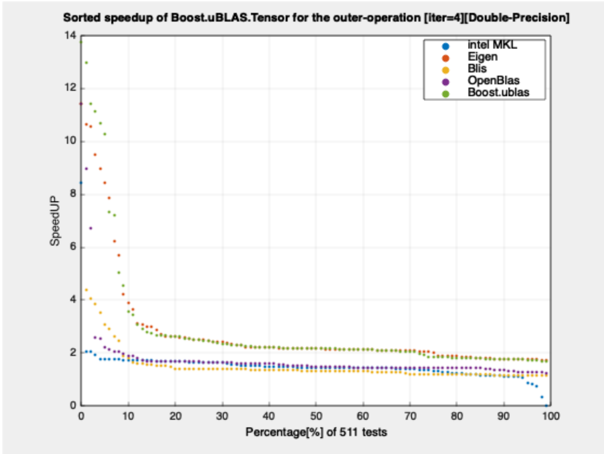


Table 3.1: Speedup Summary For Single-Precision

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

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

Table 3.2: Speedup Summary For Double-Precision

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

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

3.4 Performance Metrics

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

Table 3.3: GFLOPS For Single-Precision

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

Table 3.4: GFLOPS For Double-Precision

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

Table 3.5: Utilization[%] For Single-Precision

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

Table 3.6: Utilization[%] For Double-Precision

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

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

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

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

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

Chapter 4

Matrix-Vector Product

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

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

The resultant vector y will have the dimension m.

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

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

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

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

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

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

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

4.1 Calculating Number of Operations

Using equation 4.1 or 4.2, we fill the below table

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

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

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

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

Chapter 5

Matrix-Matrix Product

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`.