

# Implementing BLAS Operation using OpenMP

Amit Singh      Cem Basoy

March 24, 2021

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Hand-tuned Assembly . . . . .	2
1.2	Compiler Intrinsics . . . . .	3
1.3	Compiler Dependent Optimization . . . . .	3
1.4	BLAS Routines . . . . .	4
1.5	Machine Model . . . . .	4
1.6	Performance Metrics . . . . .	4
1.6.1	FLOPS . . . . .	4
1.6.2	Speedup . . . . .	5
1.6.3	Speed-down . . . . .	5
1.6.4	Peak Utilization . . . . .	5
1.7	System Information . . . . .	5
1.8	Compiler Information . . . . .	6
1.9	library Version . . . . .	6
<b>2</b>	<b>Vector-Vector Inner Product</b>	<b>7</b>
2.1	Calculating Number of Operations . . . . .	7
2.2	Algorithm . . . . .	8
<b>3</b>	<b>Outer Product</b>	<b>10</b>
<b>4</b>	<b>Matrix-Vector Product</b>	<b>11</b>
<b>5</b>	<b>Matrix-Matrix Product</b>	<b>12</b>

# 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 be 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 are 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 Ininsics

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 below:

- $C_{L_i}$ : 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

$$S_{L_i} = C_{L_i} W_{L_i} N_{L_i} \quad (1.1)$$

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.

$$FLOPS = \frac{Number\ of\ Operation}{Time\ taken} \quad (1.2)$$

### 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.

$$Speedup = \frac{Flops_{reference}}{Flops_{existing}} \quad (1.3)$$

### 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.

$$Speeddown = \frac{Flops_{existing}}{Flops_{reference}} \quad (1.4)$$

### 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.

$$Peak\ Utilization = \frac{Flops}{Peak\ Performance} \times 100 \quad (1.5)$$

## 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 Utilization found using the formula defined on the [FLOPS, 2021]

$$Peak\ Utilization = Frequency \times Cores \times \frac{FLOPS}{Cycle} \quad (1.6)$$

*Peak Utilization<sub>FP32</sub> = 588.8 GFLOPS*

*Peak Utilization<sub>FP64</sub> = 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

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

$$y \cdot x = x \cdot y \quad (2.2)$$

Equation 2.2 can be easily proven using equation 2.1.

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

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

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

From the equation 2.1

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

## 2.1 Calculating Number of Operations

Using equation 2.1, we fill the below table

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



Total Number of Operations = Number of Multiplication + Number of Addition

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

Total Number of Operations =  $2n - 1$

## 2.2 Algorithm

---

**Algorithm 1:** Inner Product SIMD Function

---

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

---

---

**Algorithm 2:** Dot Product

---

**Input:**  $c, a, b, \text{max\_threads}, n$   
//  $a$  and  $b$  are pointer to the vectors  
//  $c$  is the pointer to the output  
//  $n$  is the size of the vectors  
//  $\text{max\_threads}$  is the user provided thread count  
**begin**  
     $\text{number\_el\_}L_1 \leftarrow \lfloor \frac{S_{L_1}}{S_{data}} \rfloor$   
     $\text{number\_el\_}L_2 \leftarrow \lfloor \frac{S_{L_2}}{S_{data}} \rfloor$   
     $\text{block}_1 \leftarrow 2 \times \text{number\_el\_}L_1$   
     $\text{block}_2 \leftarrow \lfloor \frac{\text{number\_el\_}L_1}{2} \rfloor$   
     $\text{block}_3 \leftarrow \lfloor \frac{\text{number\_el\_}L_2}{2} \rfloor$   
     $\text{sum} \leftarrow 0$   
     $\text{omp\_set\_num\_threads}(\text{max\_threads})$   
    **if**  $n < \text{block}_1$  **then**  
         $\text{sum} \leftarrow \text{simd\_loop}(a, b, n)$   
    **end**  
    **else if**  $n < \text{block}_3$  **then**  
         $\text{min\_threads} \leftarrow \lfloor \frac{n}{\text{block}_2} \rfloor$   
         $\text{num\_threads} \leftarrow \max(1, \max(\text{min\_threads}, \text{max\_threads}))$   
         $\text{omp\_set\_num\_threads}(\text{num\_threads})$   
        **#pragma omp parallel for schedule(static) \**  
            **reduction(+:sum)**  
        **for**  $i \leftarrow 0$  **to**  $n$  **by**  $\text{block}_2$  **do**  
             $\text{ib} \leftarrow \min(\text{block}_2, n - i)$   
             $\text{sum} \leftarrow \text{sum} + \text{simd\_loop}(a + i, b + i, \text{ib})$   
        **end**  
    **end**  
    **else**  
        **#pragma omp parallel reduction(+:sum)**  
        **begin**  
            **for**  $i \leftarrow 0$  **to**  $n$  **by**  $\text{block}_3$  **do**  
                 $\text{ib} \leftarrow \min(\text{block}_3, n - i)$   
                 $\text{ai} \leftarrow a + i$   
                 $\text{bi} \leftarrow b + i$   
                **#pragma omp for schedule(dynamic)**  
                **for**  $j \leftarrow 0$  **to**  $\text{ib}$  **by**  $\text{block}_2$  **do**  
                     $\text{jb} \leftarrow \min(\text{block}_2, \text{ib} - j)$   
                     $\text{sum} \leftarrow \text{sum} + \text{simd\_loop}(\text{ai} + j, \text{bi} + j, \text{jb})$   
                **end**  
            **end**  
        **end**  
    **end**  
     $c \leftarrow \text{sum}$   
**end**

---

## Chapter 3

# Outer Product

## Chapter 4

# Matrix-Vector Product

## 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>.