# BLIS Report

## Basic background

**BLIS:** BLIS is the BLAS-like Library Instantiation Software (BLIS) framework. Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.

**GEMM**: GEMM stands for General Matrix Multiplication. Level 3 of BLAS contains matrix-matrix operations, including a "general matrix multiplication" (gemm).

**Cache topology**

CPU name: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (64B cache line)

CPU clock: 2.40 GHz L1: 32 kB, L2: 256 kB, L3: 35 MB

The chips on Hamilton supports AVX2 + FMA. These are 256bit wide registers and each chip has 16 physical AVX2 registers. So, we can deduce that Peak performance is 2.4GHz 16 (422) = 38.4flops/s.

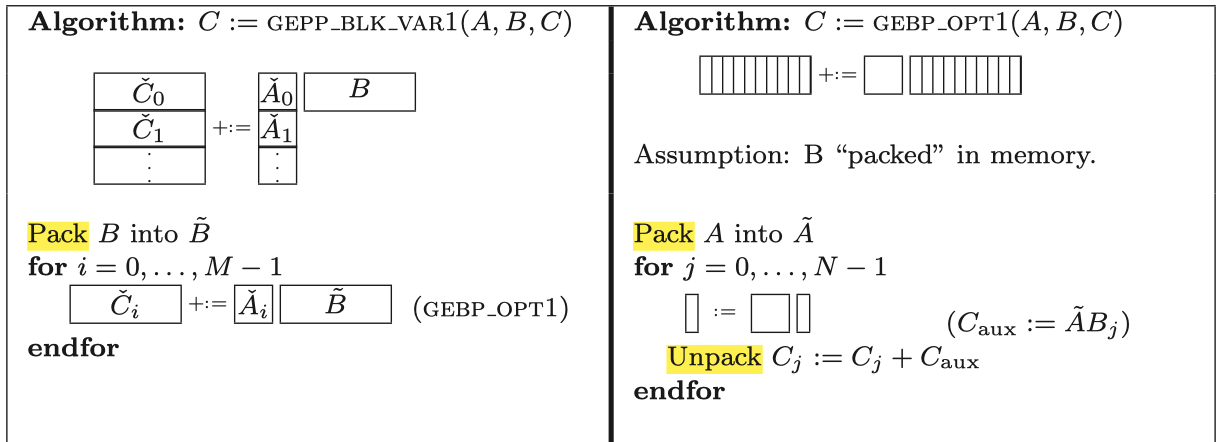
## Choices of parameters

**Basic necessary calculations**

Bandwidth estimation: From previous tests in previous classes we can have an initial estimation on bandiwith. **L1** (less than 32kB) we see sustained bandwidth of around 300GByte/s providing up to 70GFlops/s; **L2** (less than 256kB) provides around 80GByte/s and around 19GFlops/s; **L3** (less than 35MB) provides around 36GByte/s and around 8GFlops/s. **Main memory** provides around 13GByte/s and around 3.25GFlops/s.

Matrix product calculation quantum: flops, where we give m = n = k = 4000. It can be estimated around 1.28 8G flops

**Matrix splitting principle**

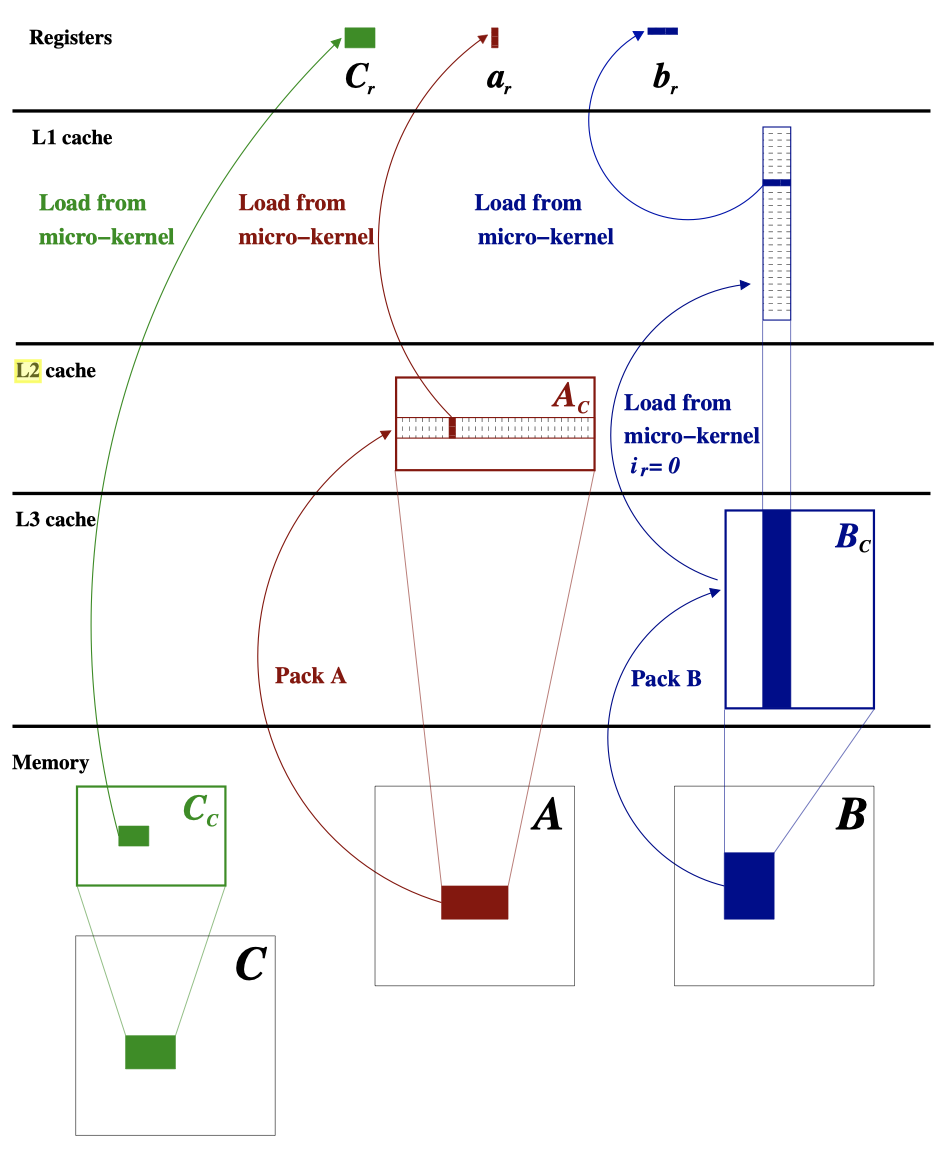
****

*Fig. 1. Optimized implementation of gebp*

According to GEBP tiling and the conclusion given in the paper (Anatomy of High-Performance Matrix Multiplication). Mc Nc, Nr computing power / 2 / , Mc Kc K, Mc Kc < L2.

From the deduction above, we can turn to test the parameters starting from the estimated value.

**Strategy**

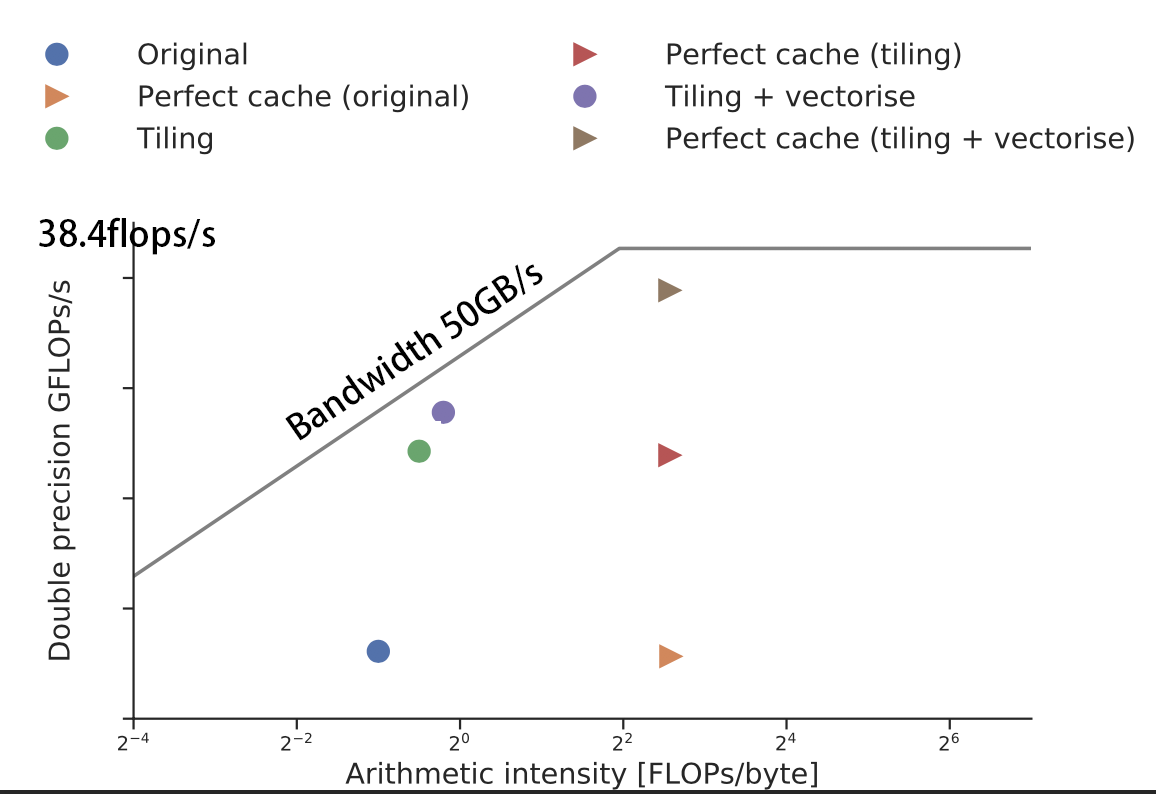
****

*Fig. 2. Packing in BLIS implementation*

We can pre-give a set of parameters then watch the calculating power as the performance factor using *likwid*- *perfctr*. And from the picture above, we can use L2 miss rate to another performance factor since L2 stores A matrix and it is also securable via *likwid-perfctr MEM\_DP*.

**We get the plot of FLOP/s – m = n = k = (500 - 4000):** \*It may vary basing on different conditions.

*Fig. 3 Tuned. performance curve*

****

*Fig. 4 L2 miss ratio & estimated roofline model*

**Final Choices of parameters are below after tests:** MC=128 KC=128 NC=128 MR=32 NR=32

**Results analysis:** Since we use *likwid*- *perfctr* to test the performance, we simultaneously get the corresponding bandwidth data which are around 50GByte/s. That means, L2 cache supports the most of bandwidth of the BILS calculation which confirms our hypothesis. And we can furtherly find that the tuned L2 miss ration is steadily lower than the original one.

In addition, due to the limitation of BLIS implementation, intensity hardly go beyond 2, so the Computing power is limited in the roofline area. Through proper tiling and vectorising, the GFLOPs/s can be estimated to have reached the bottleneck in this area. Thus, we can assert that this set of parameters is decently tuned.

## Choices of compiler flags

**Changing O2/O3:** Use -O2/-O3 to enable complier to turn on SIMD mode and vectorized compilation and SIMD mode. Using O3 to enable NEON processor which is a coprocessor with a Single Instruction Multiple Data (SIMD) architecture. It is a unique SIMD instruction set name on the armv7 arm64 chip. SIMD can complete multiple data parallel calculations with one instruction. For example, perform scalar multiplication on 4 floats at the same time. So, it is important here to enable SIMD mode and O3 here.

**Using likwid flag:** We have used a lot of likwid bench tests to get some deep running information, it would be wise to compile the code accordingly.

## Choices of code annotations/optimisations

Using *#pragma omp simd* to tells the compiler “Please vectorise” which will empower the calculation.

Using *#pragma GCC unroll(4)* to tell GCC compiler unroll the loop which will save the cost of read operands.