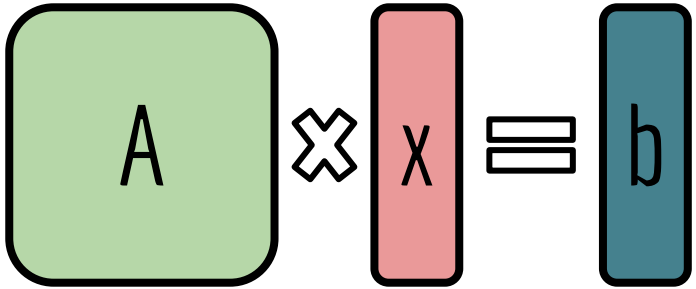
High Performance Computing Project

Abdurahman Kanit Bensaidi, Louis Peyrondet,
Kévin Shao & Alexandre Tabouret

Context

Goal: Solve



GEMM (GEneral Matrix Multiplication)

$$C = \alpha A \times B + \beta C$$

C has size of $M \times N$ A has size of $M \times K$ B has size of $K \times N$

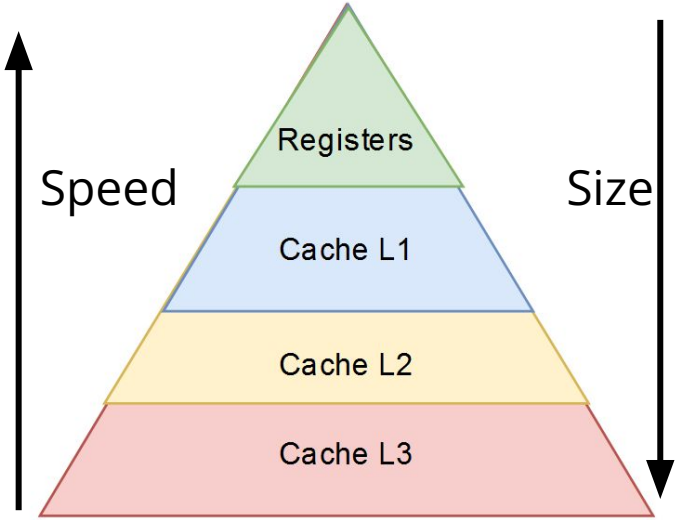
GEMM : Kernels

The goal of a kernel is to be do an operation very efficiently on a small subset of the problem.

We just have to divide our big problem into multiple smaller ones and apply the kernel on them.

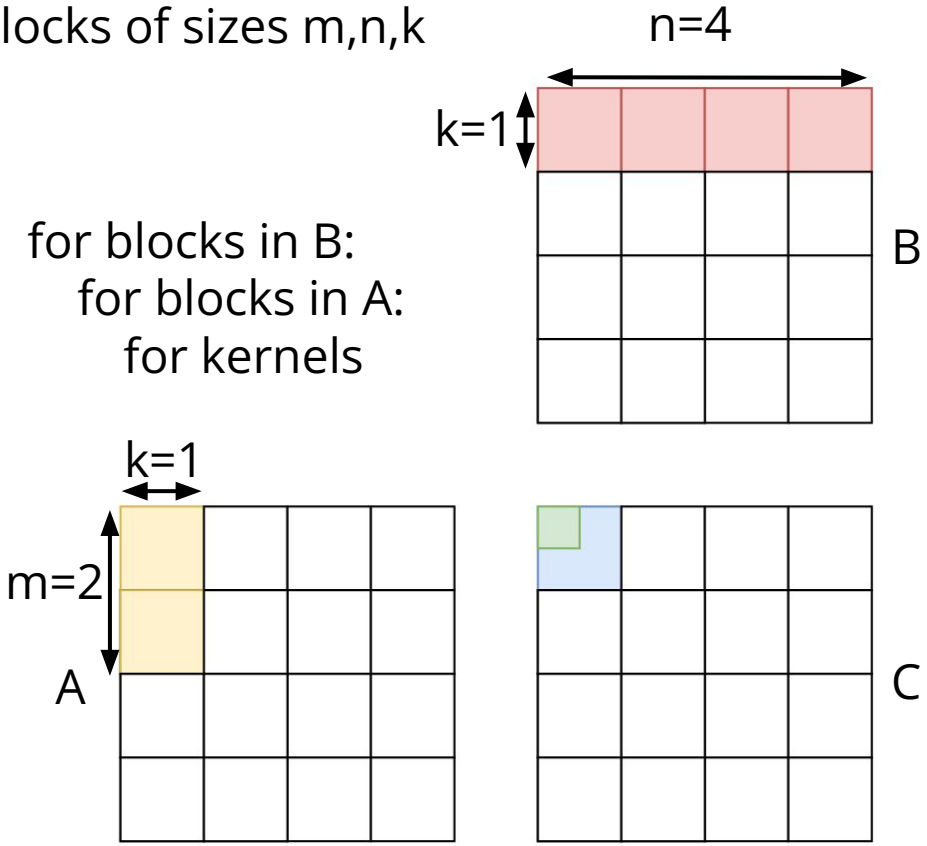
GEMM : Blocks

Goal : Maximize cache utilization



Blocks of sizes m,n,k

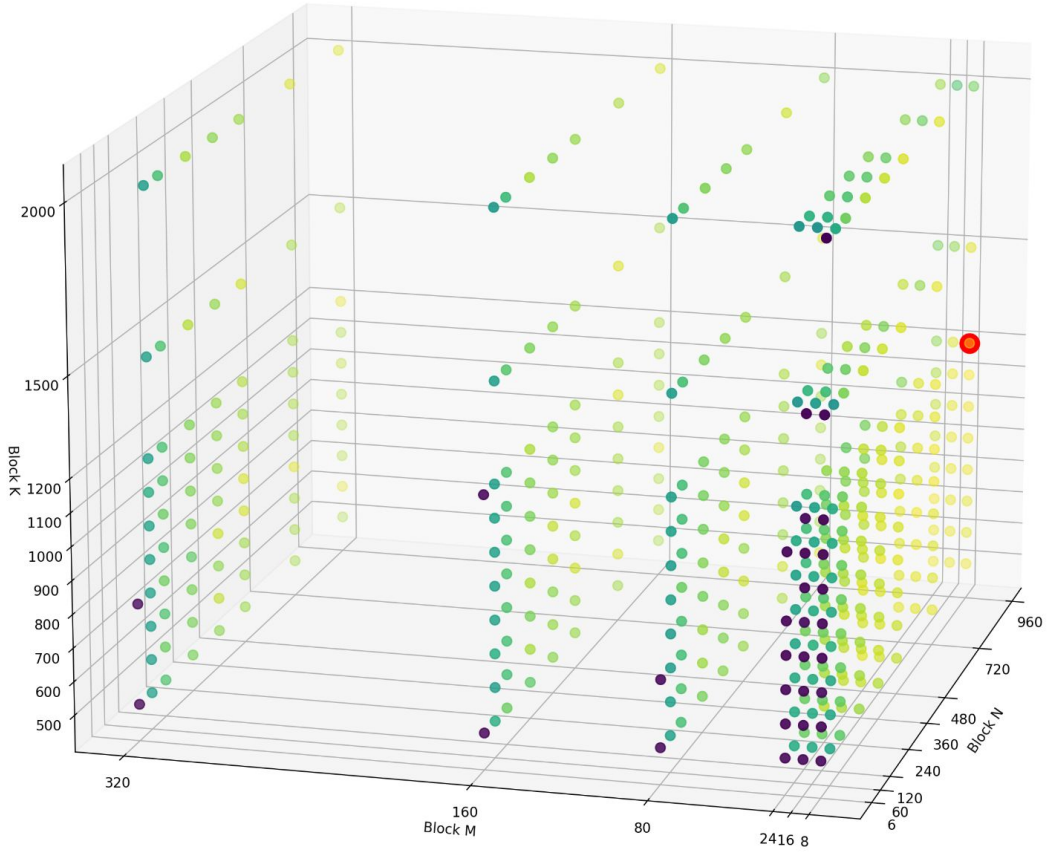
```
for blocks in B:
    for blocks in A:
        for kernels
```



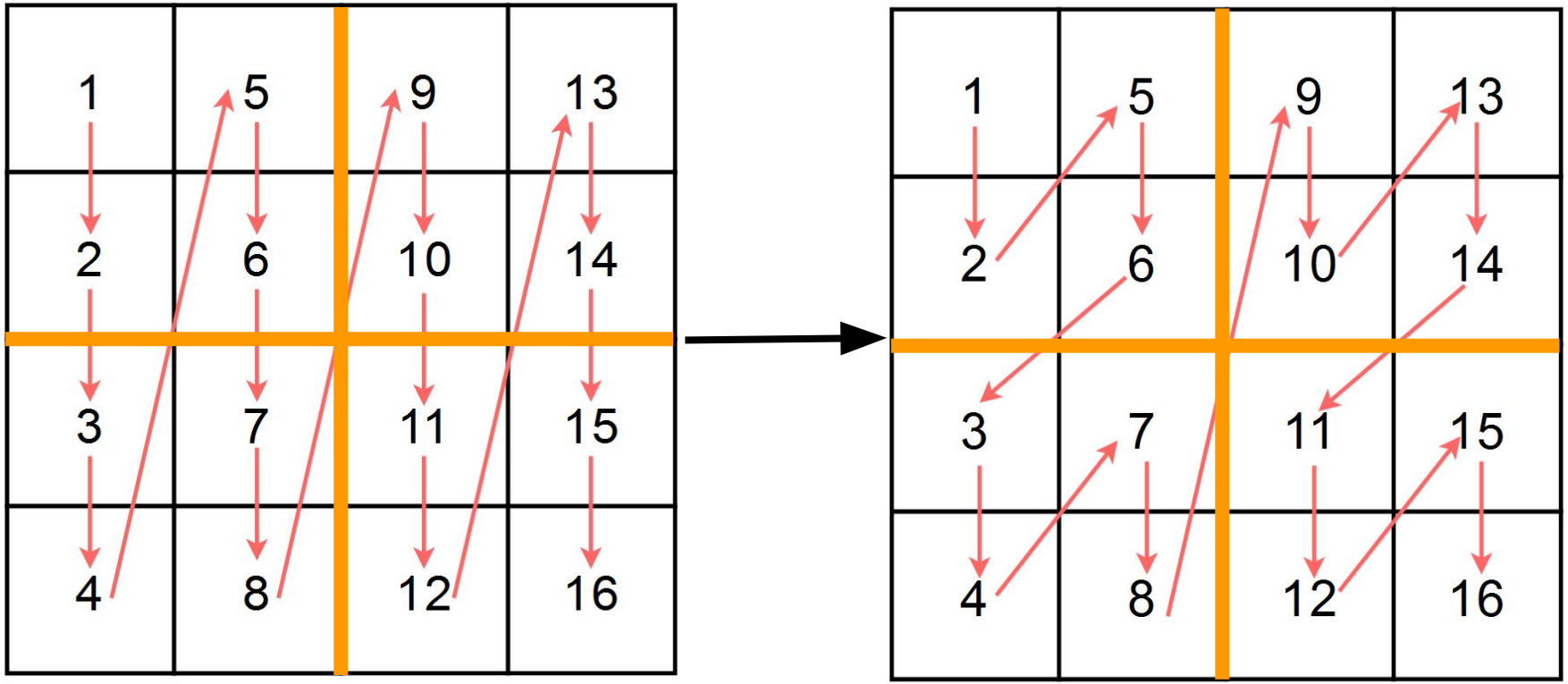
GEMM : Blocks

In practice slightly tweaked block sizes gives better performances

M=8, K=1200, N=960

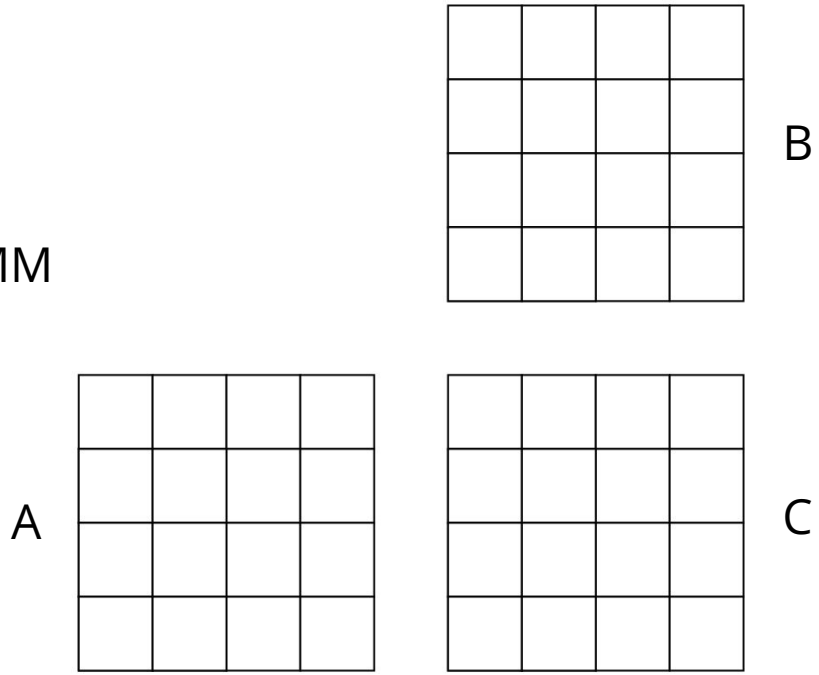


GEMM : Packing to optimize the memory layout



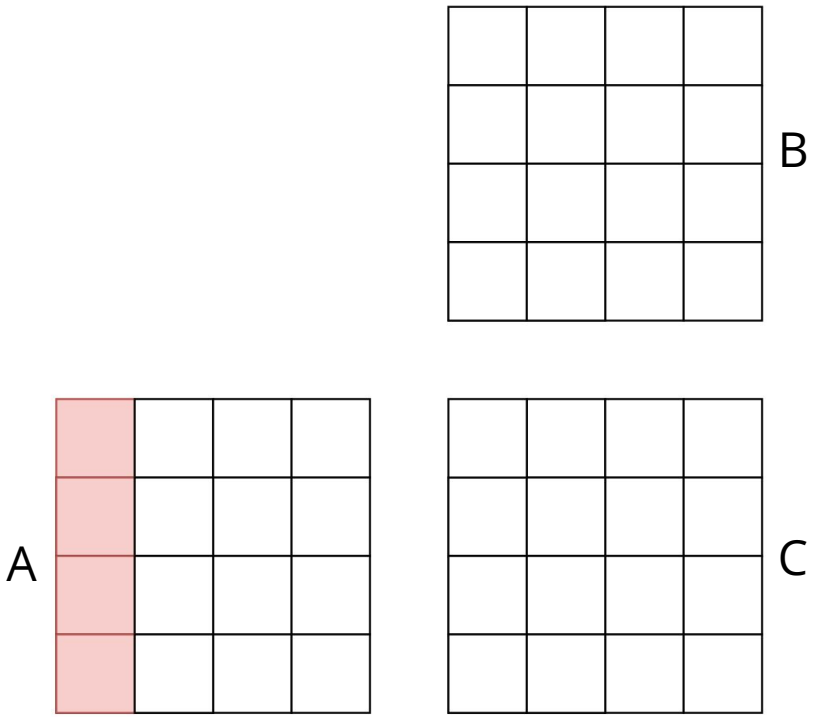
GEMM : Kernels

4*4 kernel for GEMM

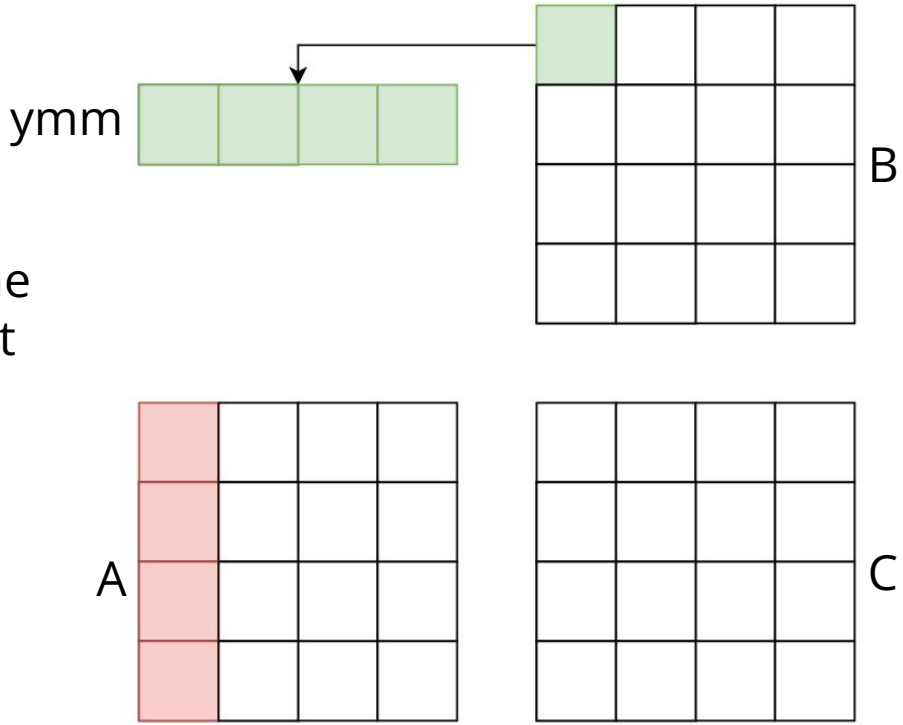


GEMM : Kernels

Load first column of A



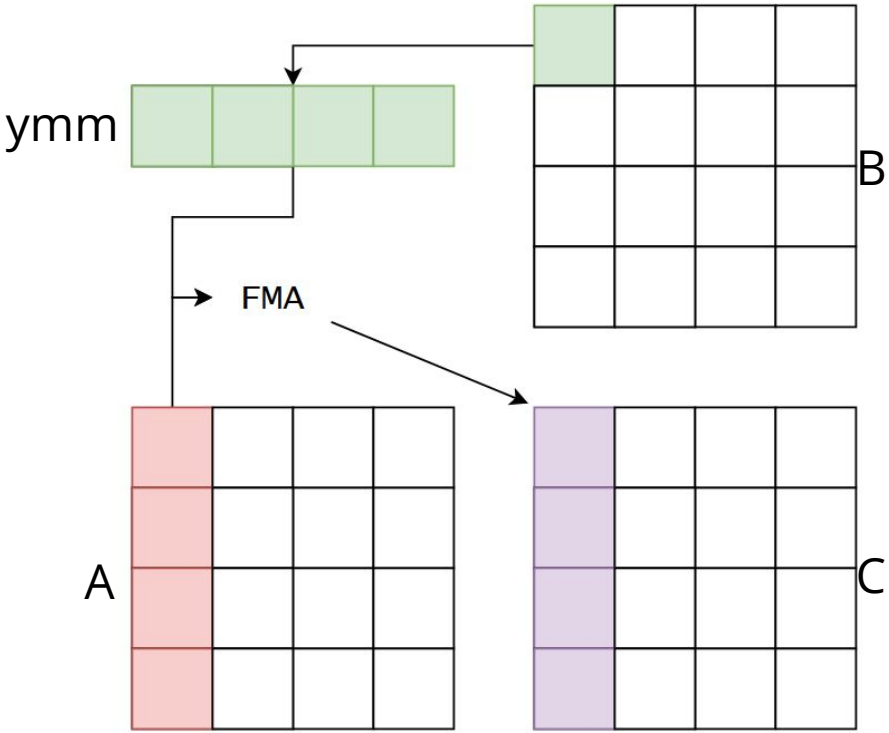
GEMM : Kernels



Fill a register with the first value of the first row of B

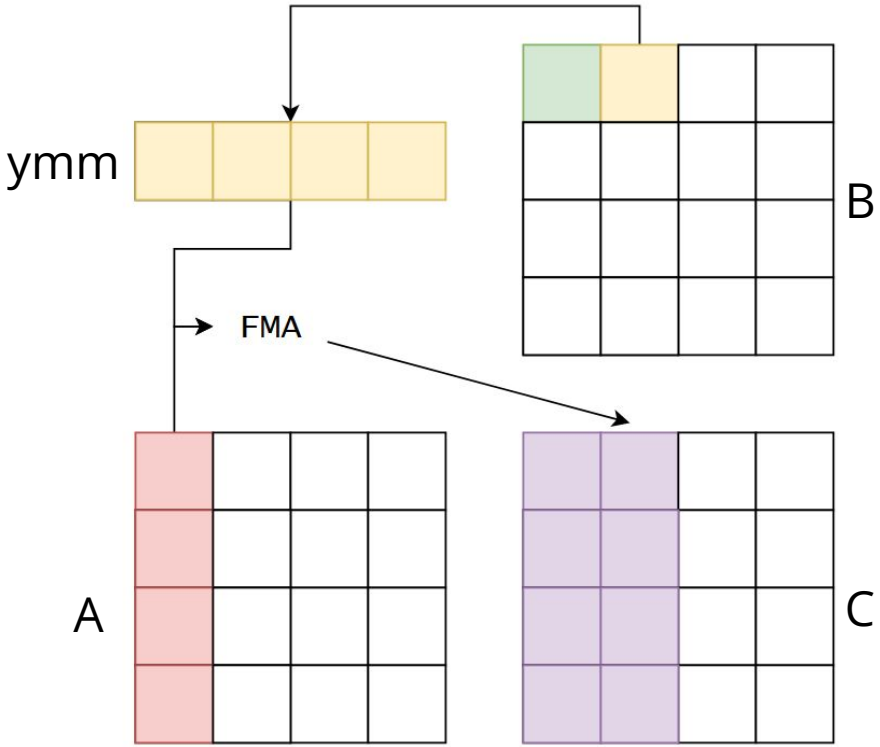
GEMM : Kernels

FMA the 2 registers into C



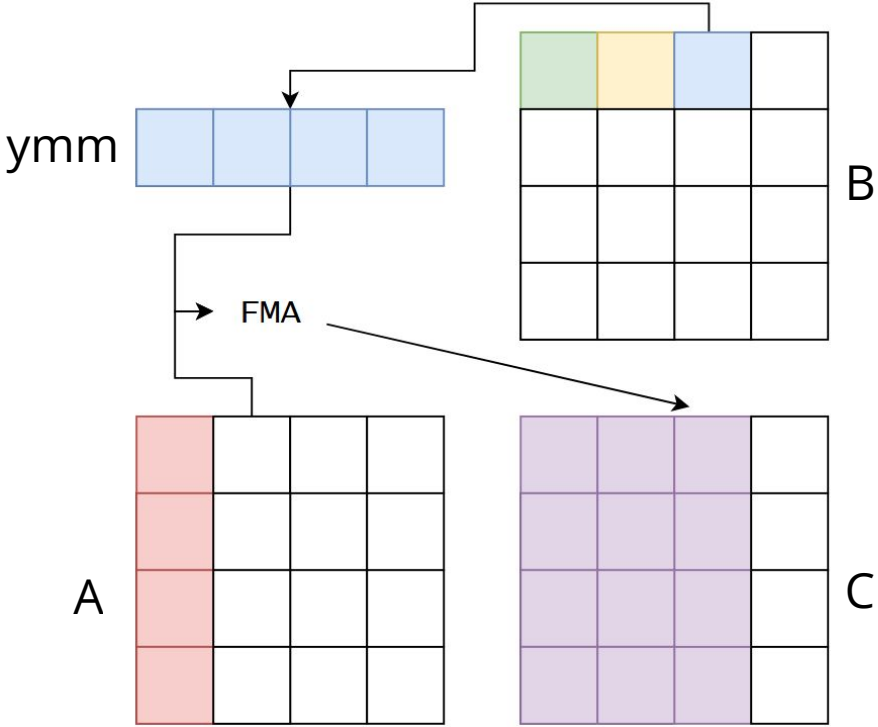
GEMM : Kernels

Get the next value in the row of B



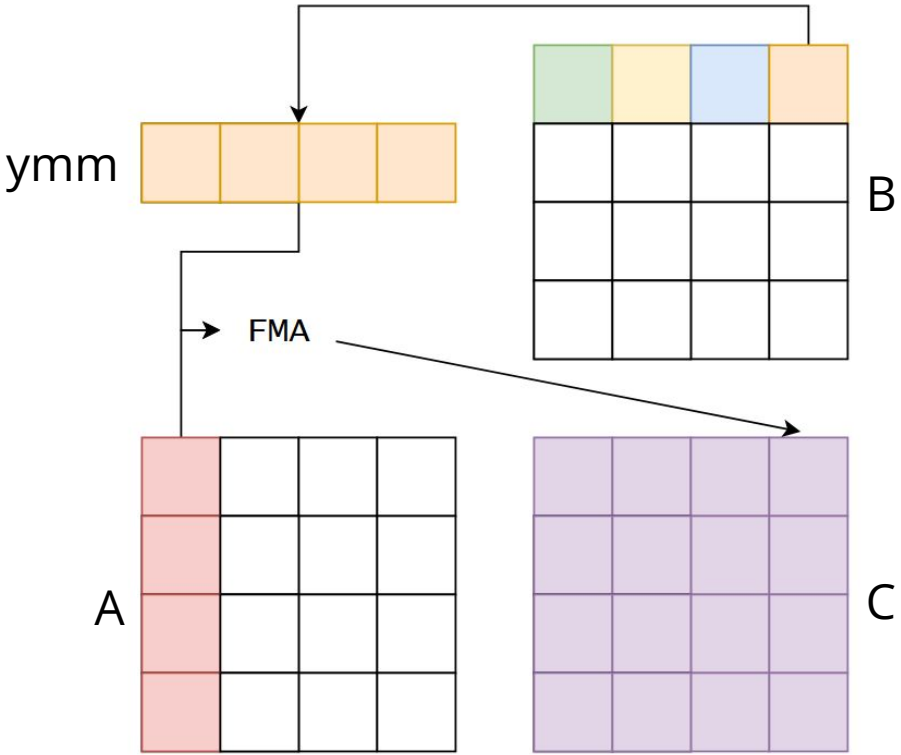
GEMM : Kernels

Again



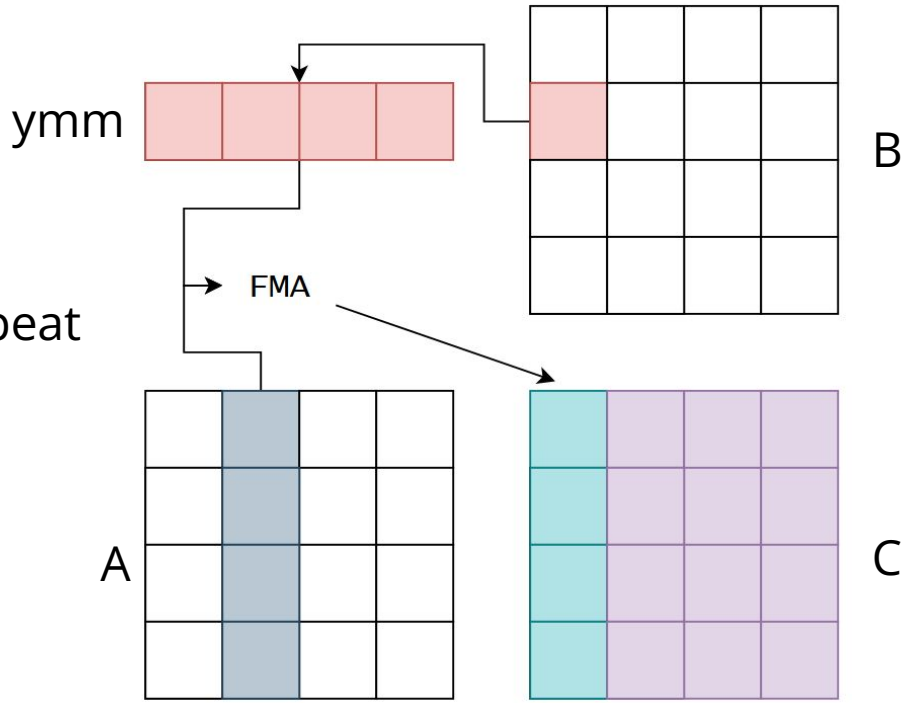
GEMM : Kernels

And again



GEMM : Kernels

Then load the next column of A and repeat



GEMM : Kernels

The kernel should maximise the amount of registers used.
Haswell processors have 16 ymm registers.
We chose a kernel m*n of size 8*6 :

- 1 register for the B value
- 2 registers for the column of A (8 elements)
- 12 registers to load C (48 elements)
    - 15 out of 16 register used

GEMM : Kernels

"We don't care about implementation details"

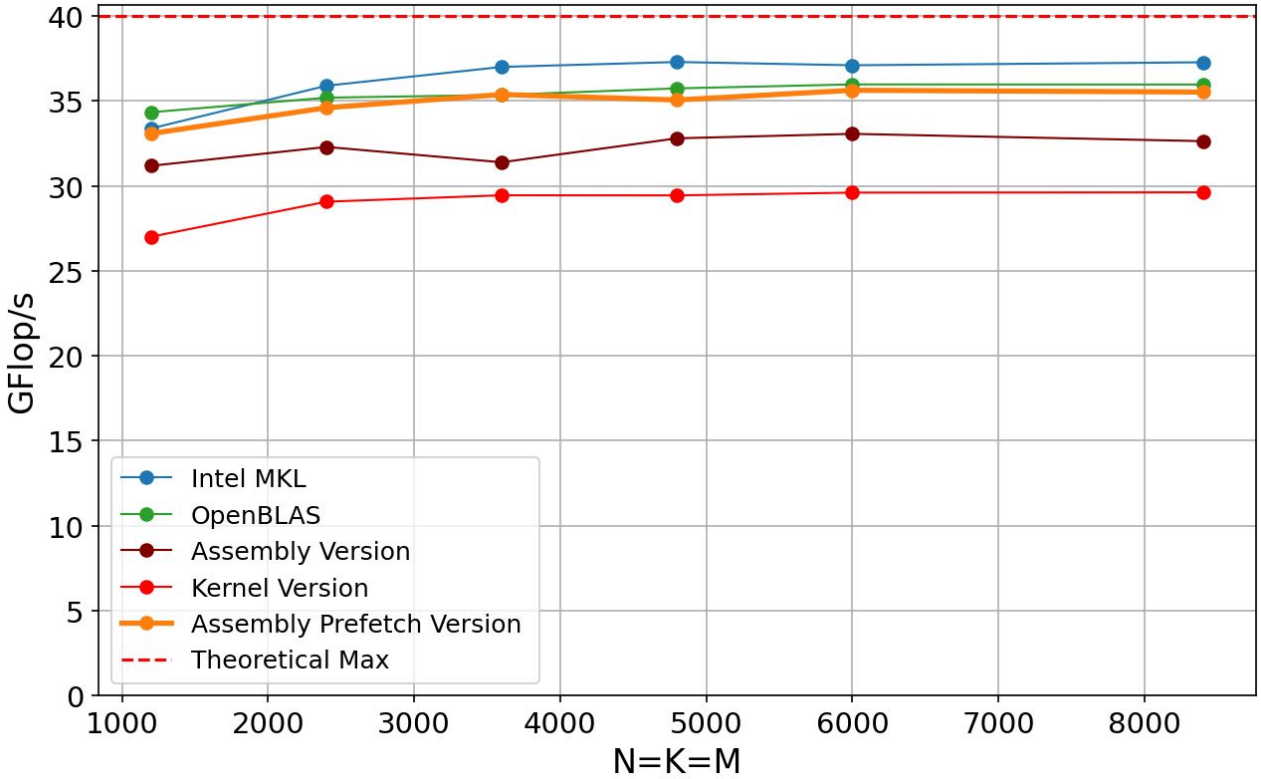
In this case, we **DO** care : that's what makes the difference

Not shown :
Unroll x4

```
kernel8x6_asm:
.LOOP_K:
    # Load A matrix
    vmovapd (%rdi), %ymm12  # Load A[0:3]
    vmovapd 32(%rdi), %ymm13 # Load A[4:7]
    prefetcht0 512(%rdi) # Prefetch 2 4-unroll iterations ahead for A
    prefetcht0 576(%rdi)
    prefetcht0 640(%rdi)
    prefetcht0 704(%rdi)

    prefetcht0 384(%rsi) # Prefetch 2 4-unroll iterations ahead for B
    prefetcht0 432(%rsi)
    prefetcht0 480(%rsi)
    prefetcht0 528(%rsi)

    # Broadcast B matrix elements and multiply
    vbroadcastsd (%rsi), %ymm14              # Broadcast B[0]
    vbroadcastsd 8(%rsi), %ymm15             # Broadcast B[1]
    vfmadd231pd %ymm12, %ymm14, %ymm0        # C += A * B[0]
    vfmadd231pd %ymm13, %ymm14, %ymm1
    vfmadd231pd %ymm12, %ymm15, %ymm2
    vfmadd231pd %ymm13, %ymm15, %ymm3

    vbroadcastsd 16(%rsi), %ymm14            # Broadcast B[2]
    vbroadcastsd 24(%rsi), %ymm15            # Broadcast B[3]
    vfmadd231pd %ymm12, %ymm14, %ymm4
    vfmadd231pd %ymm13, %ymm14, %ymm5
    vfmadd231pd %ymm12, %ymm15, %ymm6
    vfmadd231pd %ymm13, %ymm15, %ymm7
    # Repeat for the other columns ...
```

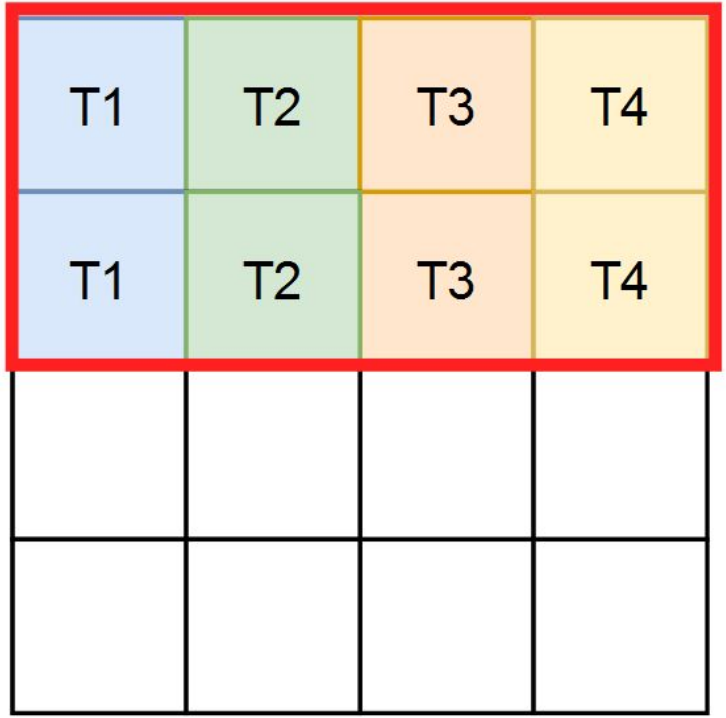
GEMM : Performances

- Reaches **performance equivalent to OpenBLAS** and 95% of IntelMKL
- Using assembly allow explicit control over register spilling
- Works for every kind of matrices
- A and/or B transposed doesn't affect performances



GEMM : Multithreading

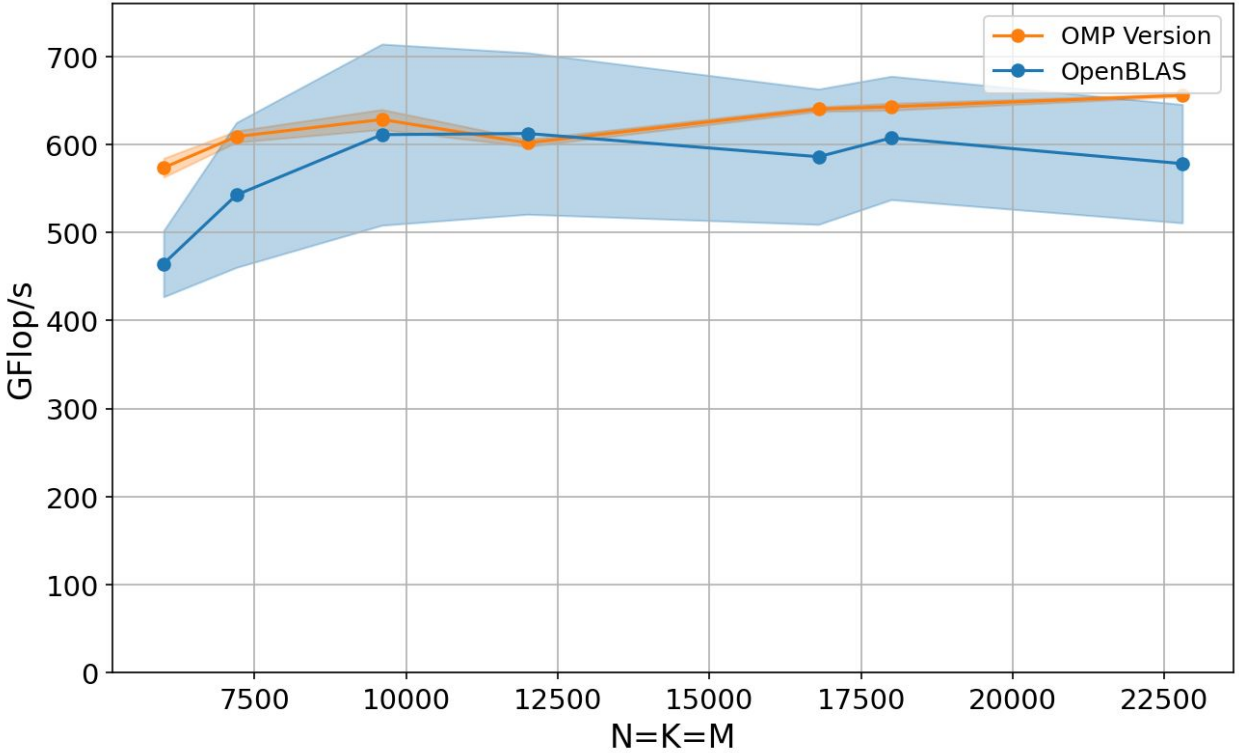
- Multithreading using OpenMP
- Dispatch threads to each compute a group of kernels inside a block
- Each thread have the same amount of work



GEMM : Multithreading

Our implementation :

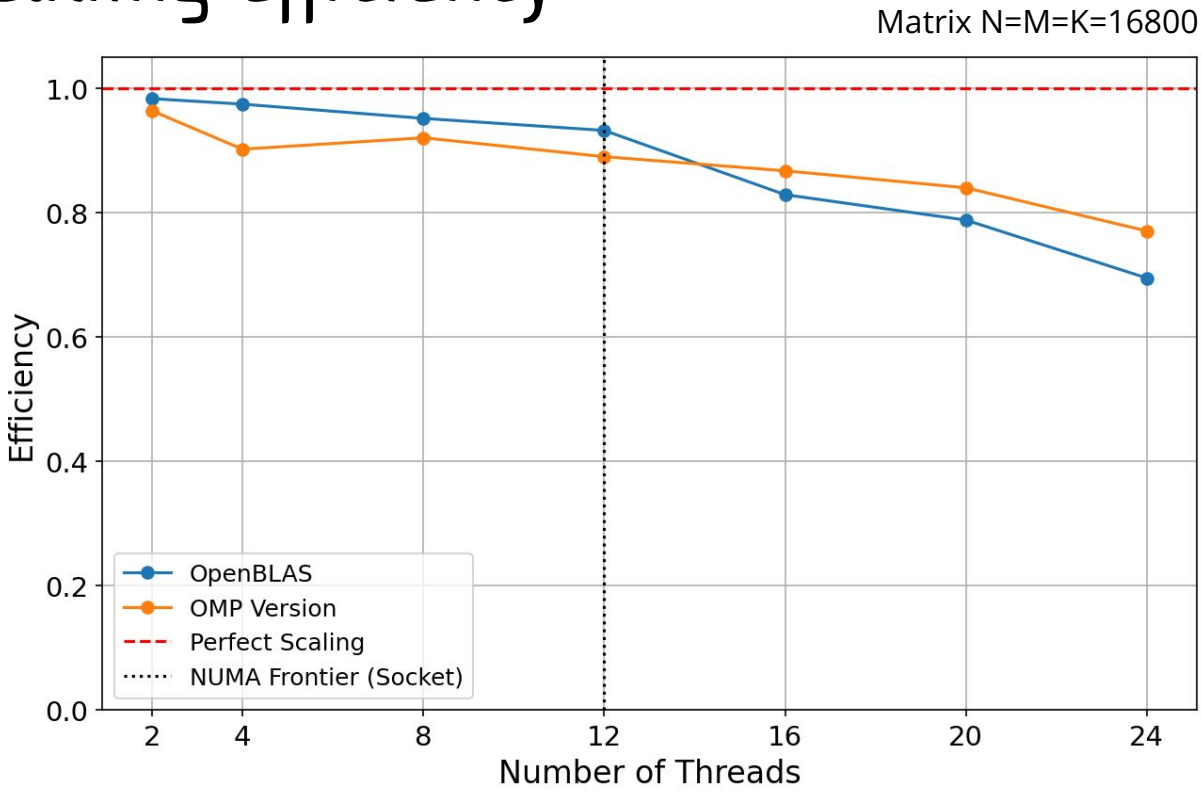
- Beats OpenBLAS with 650 GFlop/s
- Achieves a small standard deviation (8 GFlop/s vs 75 GFlop/s)



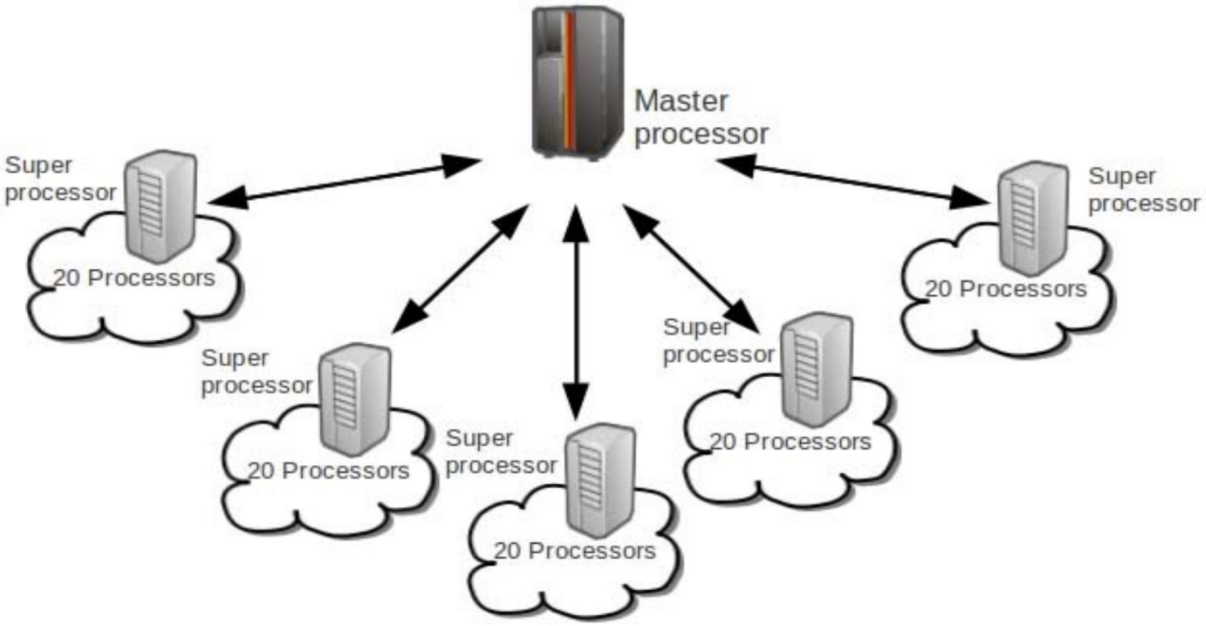
Block sizes : M=768,K=900,N=11520

GEMM : Multithreading efficiency

- Better efficiency than OpenBLAS for threads >= 16
- OpenBLAS impacted by NUMA
- Could achieve better efficiency



GEMM : Distributed version



GEMM : Giving data to each node

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

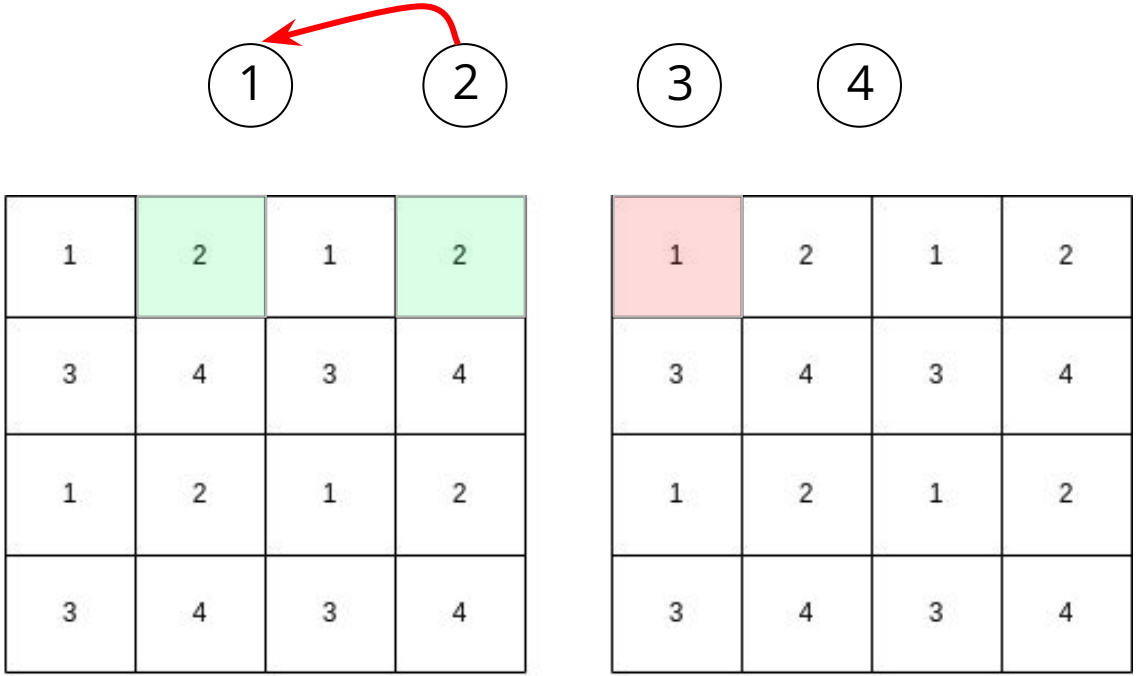
GEMM : Giving data to each node

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

GEMM : Giving data to each node



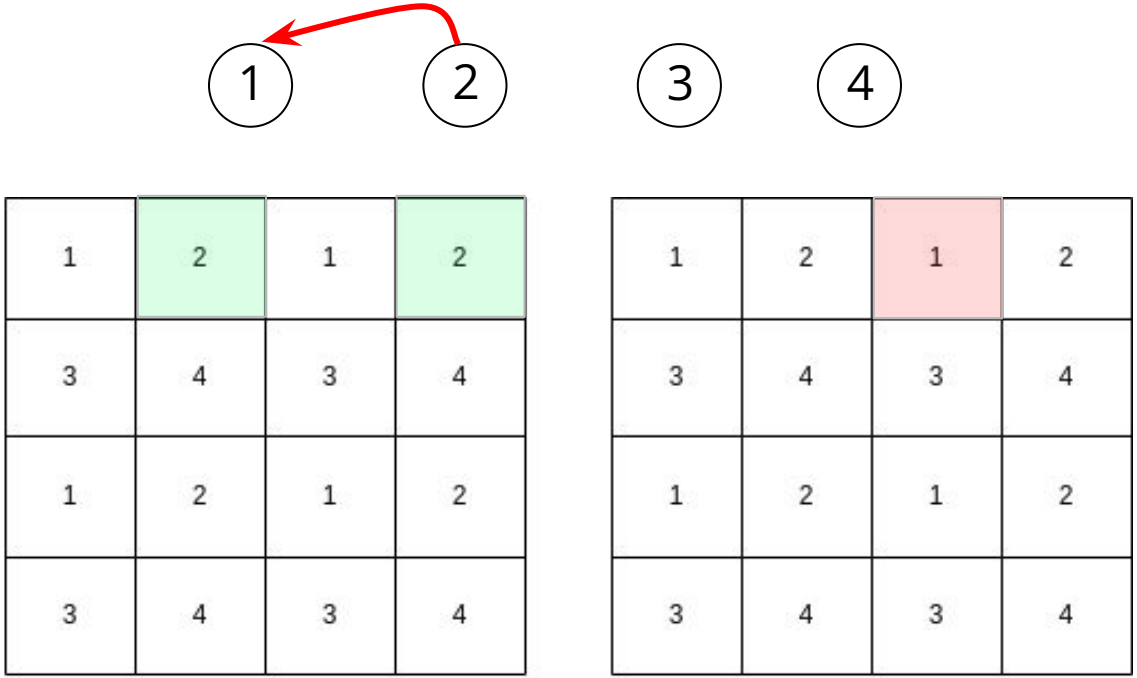
GEMM : Giving data to each node

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

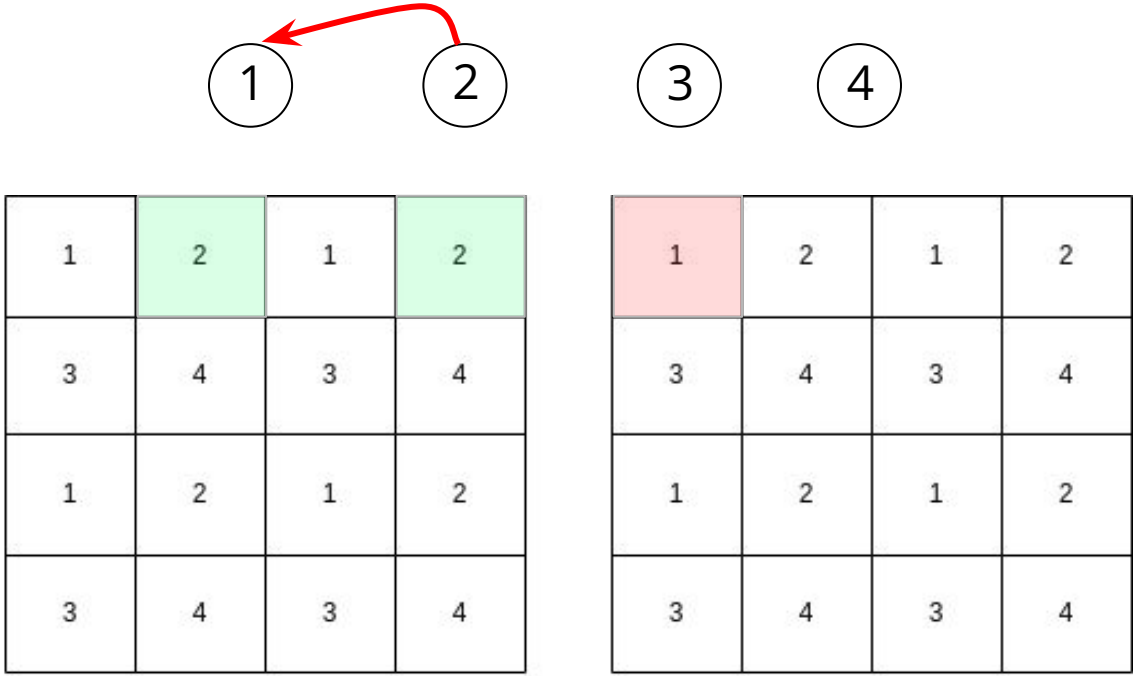
1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

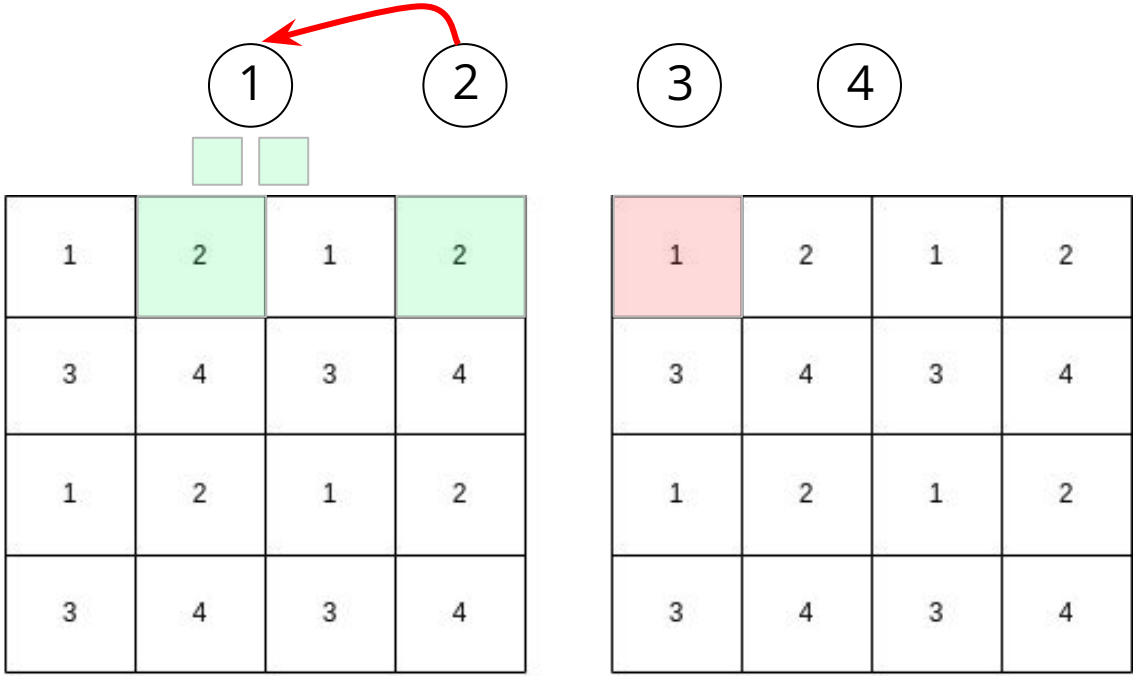
GEMM : Giving data to each node



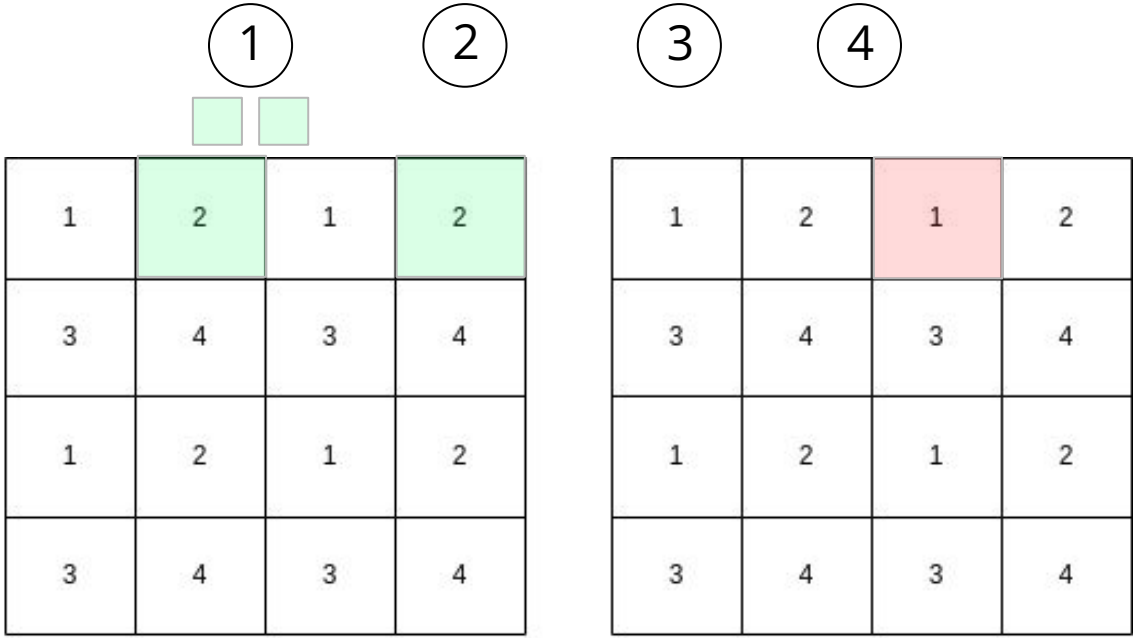
GEMM : Giving data to each node



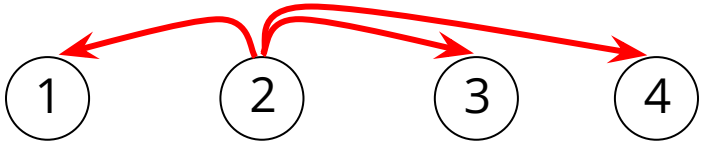
GEMM : Giving data to each node



GEMM : Giving data to each node



GEMM : Giving data to each node



1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

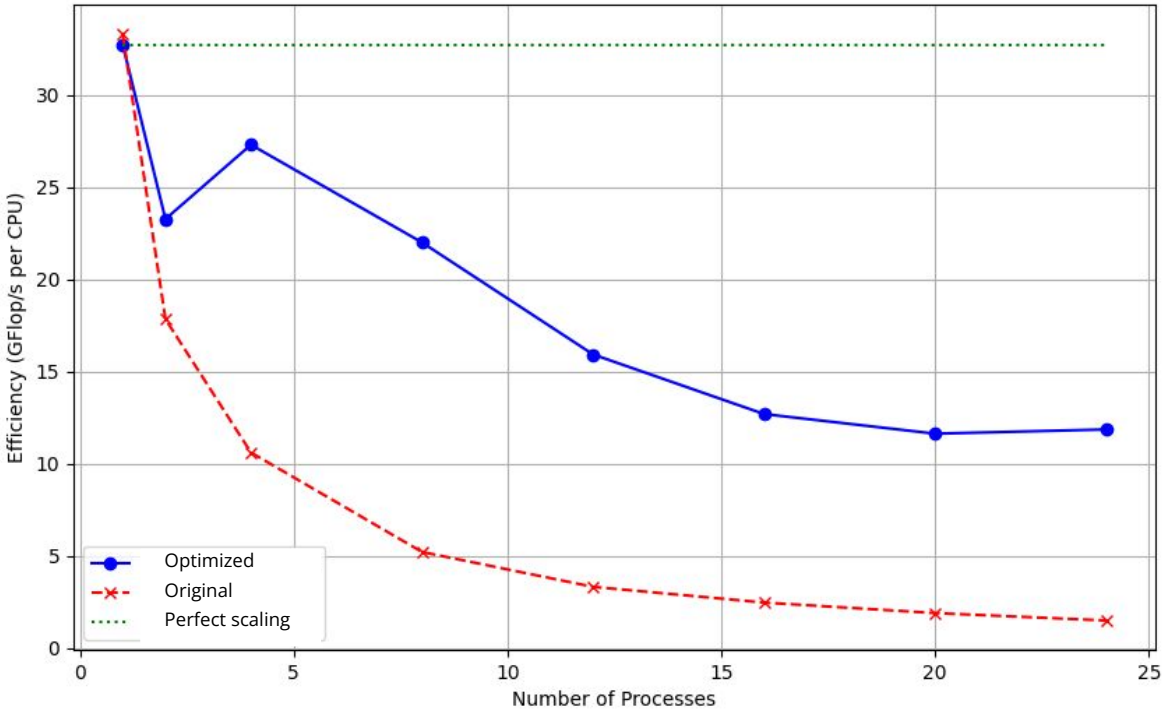
GEMM : Giving data to each node

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

1	2	1	2	(1,2)
3	4	3	4	(3,4)
1	2	1	2	(1,2)
3	4	3	4	(3,4)

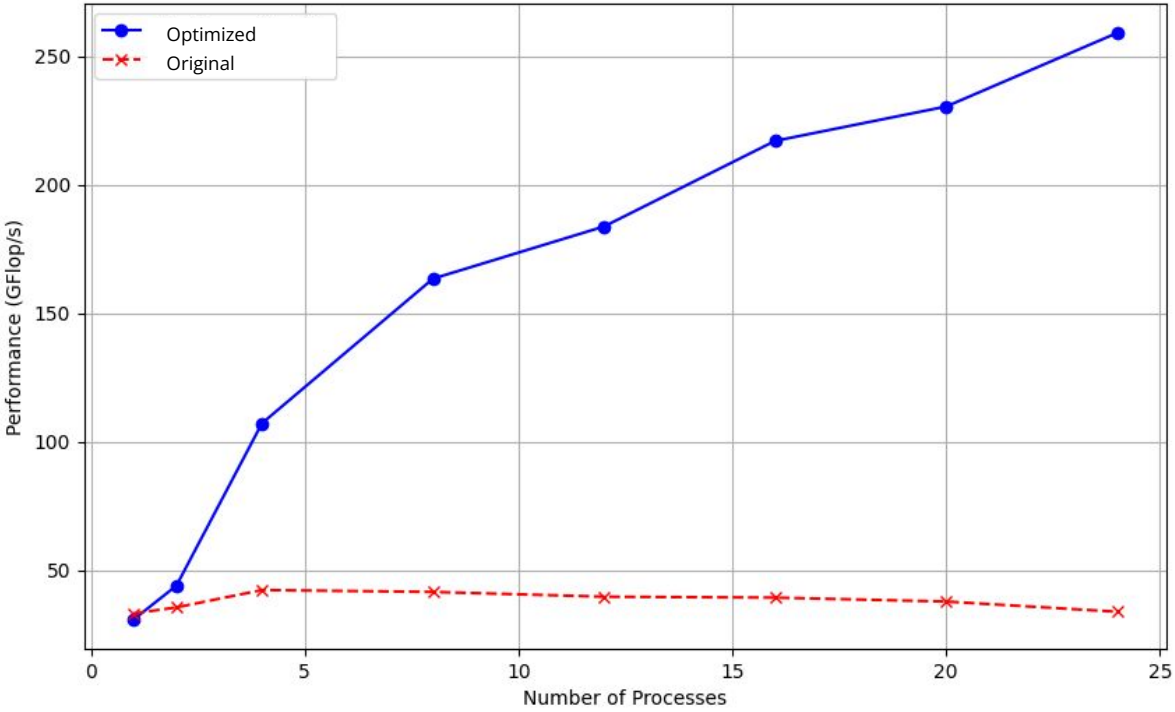
GEMM : MPI Results

Weak scalability, with each process computing 5000^3 flops



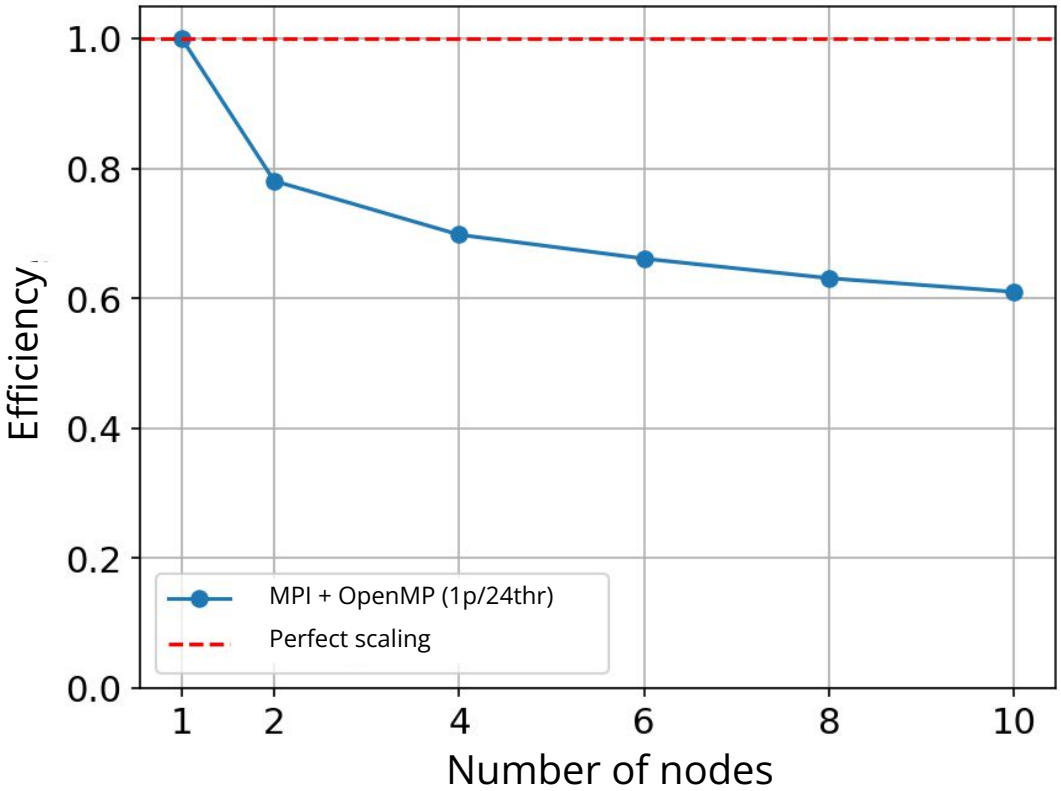
GEMM : MPI Results

Strong scalability for N = M = K = 13440 and b = 13440/n



GEMM : MPI + X weak scaling

MPI + OpenMP with 1 process per node with 24 threads each



GEMM : MPI + X

Peak performance on 10 nodes :
4 TFlop/s !!!

(Do not take this seriously, this is a joke)

Home »Lists »TOP500 »June 2000

JUNE 2000

TOP 10 Sites for June 2000

For more information about the sites and systems in the list, click on the links or view the complete list.

TOP500 Release

THE LIST

Statistics

PERFORMANCE DEVELOPMENT

SUBLIST GENERATOR

LIST STATISTICS

TREE MAPS

HISTORICAL CHARTS

Downloads

TOP500 LIST (XML)

TOP500 LIST (EXCEL)

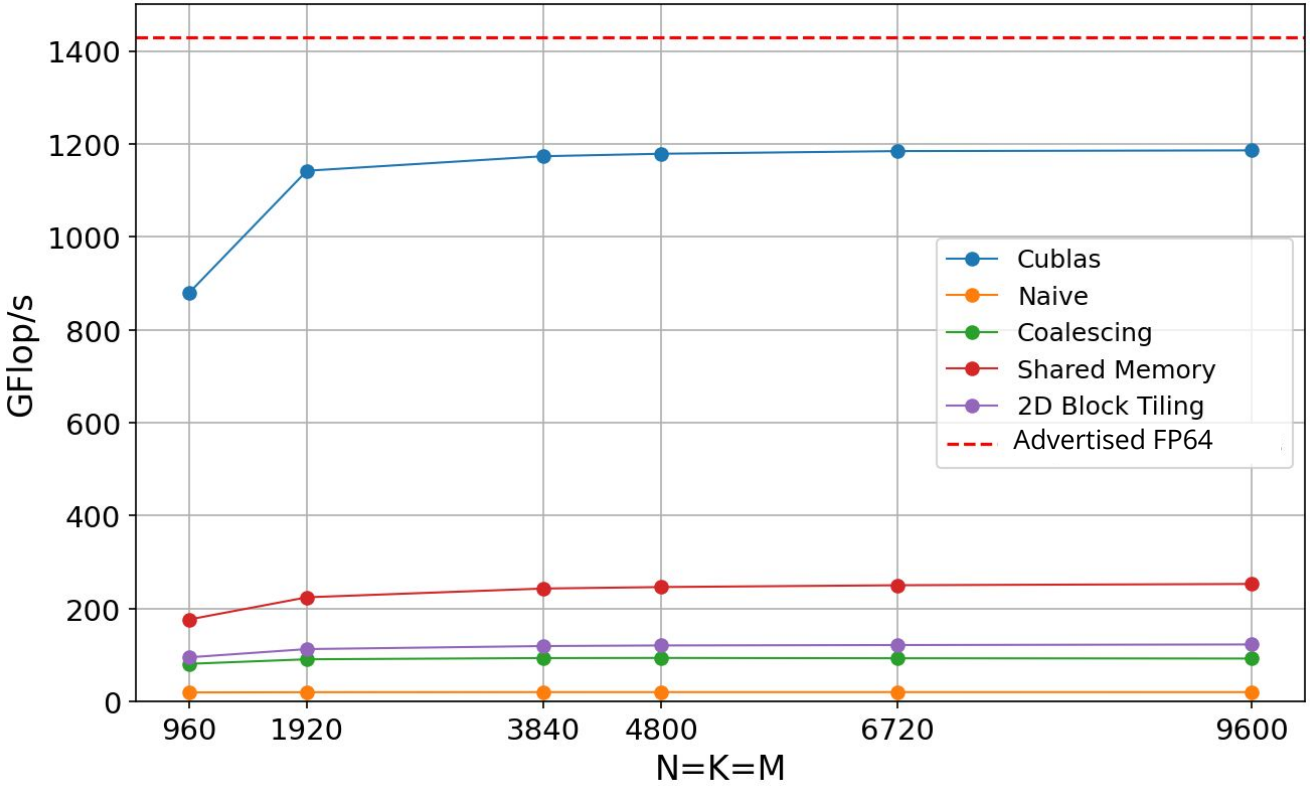
TOP500 POSTER

POSTER IN PDF

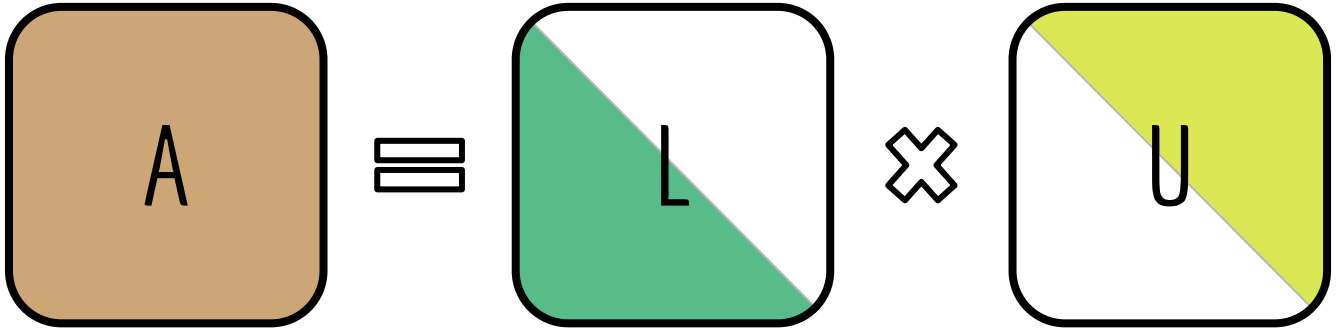
Rank	System	Cores	Rmax (GFlop/s)	Rpeak (GFlop/s)	Power (kW)
1	Plafrim, LAKA ENSEIRB-MATMECA France	240	4000.00	4000.00	
2	ASCI Blue-Pacific SST, IBM SP 604e, IBM Lawrence Livermore National Laboratory United States	5,808	2,144.00	3,856.50	
3	ASCI Blue Mountain, HPE Los Alamos National Laboratory United States	6,144	1,608.00	3,072.00	
4	SP Power3 375 MHz, IBM	1,336	1,417.00	2,004.00	

GEMM : CUDA

- Best version is shared memory with 250 GFlop/s
- 2D Block Tiling needs more work (each thread computes a sub-block of C)



GETRF (GEneral TRiangular Factorization)



A has size of M × N

Lower triangular

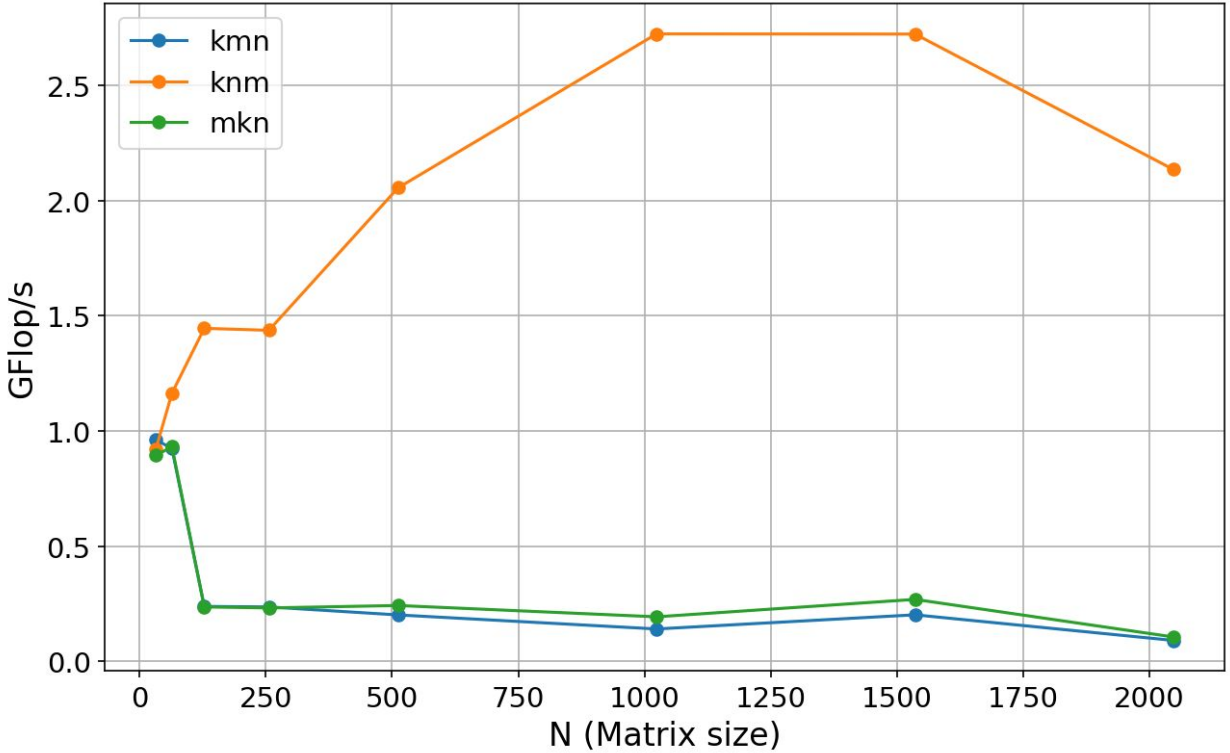
Upper triangular

GETRF : Sequential Version

```
int dgetrf_seq(int M, int N, double *A)
{
    int m, n, k;

    for( k=0; k<K; k++ ) {
        for( m=k+1; m<M; m++ ) {
            A[m, k] = A[m, k]/A[k, k];
            for( n=k+1; n<N; n++ ) {
                A[m, n] = A[m, n] - A[m, k]*A[k, n];
            }
        }
    }
    return ALGONUM_SUCCESS; /* Success */
}
```

$$\begin{pmatrix} 1 & 7 & 3 & 4 & 9 \\ 2 & 3 & 5 & 2 & 2 \\ 4 & 6 & 8 & 6 & 6 \\ 9 & 3 & 4 & 7 & 1 \\ 1 & 8 & 0 & 6 & 3 \end{pmatrix}$$

GETRF : Sequential Version (Loop Orders)

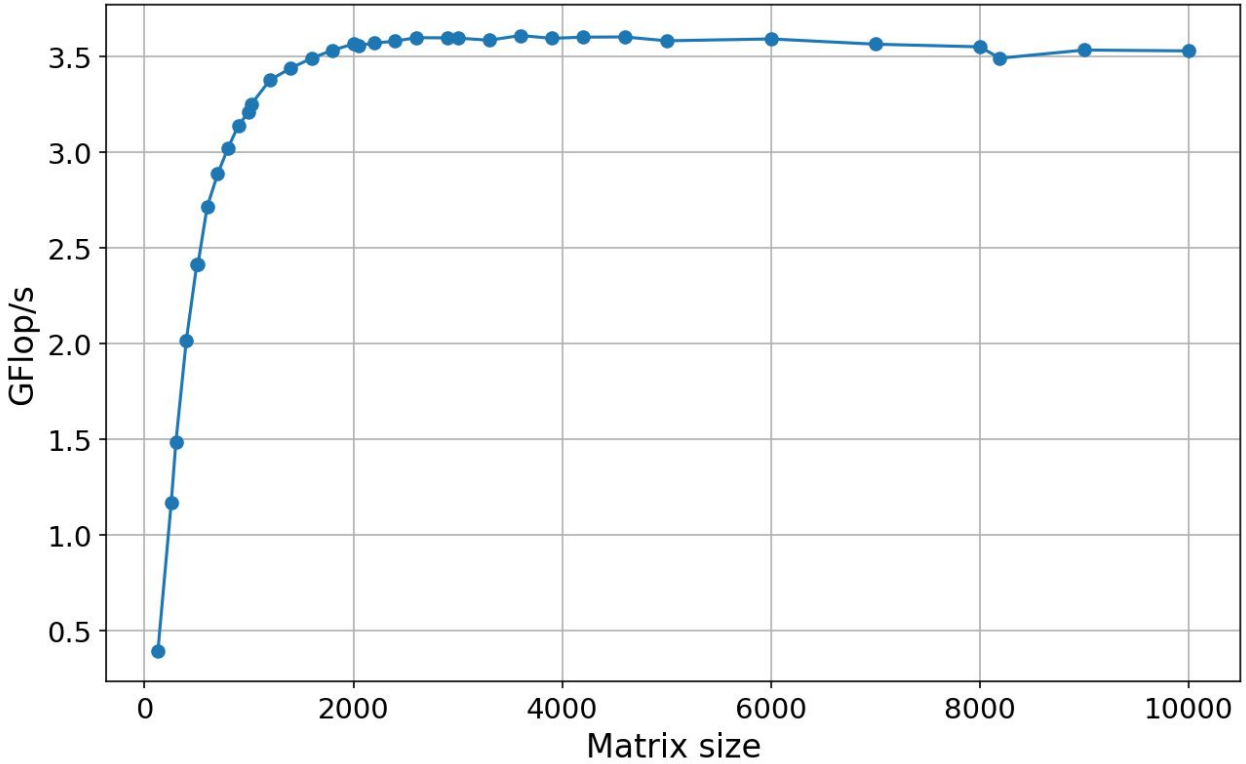


GETRF : Vectorization

$$\begin{pmatrix} 1 & 7 & 3 & 4 & 9 \\ 2 & 3 & 5 & 2 & 2 \\ 4 & 6 & 8 & 6 & 6 \\ 9 & 3 & 4 & 7 & 1 \\ 1 & 8 & 0 & 6 & 3 \end{pmatrix}$$

Speedup of 1.5 by vectorizing

GETRF : Cuda


GFlop/s
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
2000
4000
6000
8000
10000
Matrix size

GETRF : Block-ing

1 - Compute L/U_{00} using GETRF

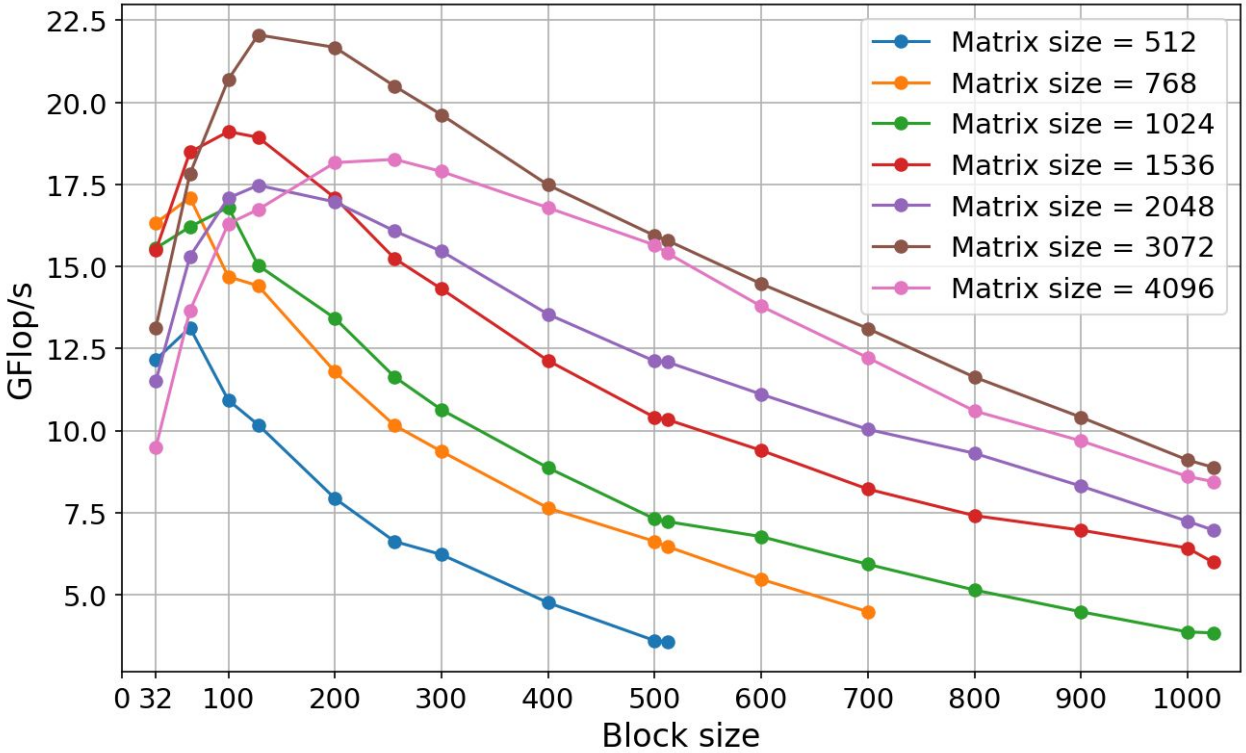
2 - Compute L_{10} and U_{01} using TRSM

3 - Compute A'_{11} using GEMM

4 - Apply the process on A'_{11}

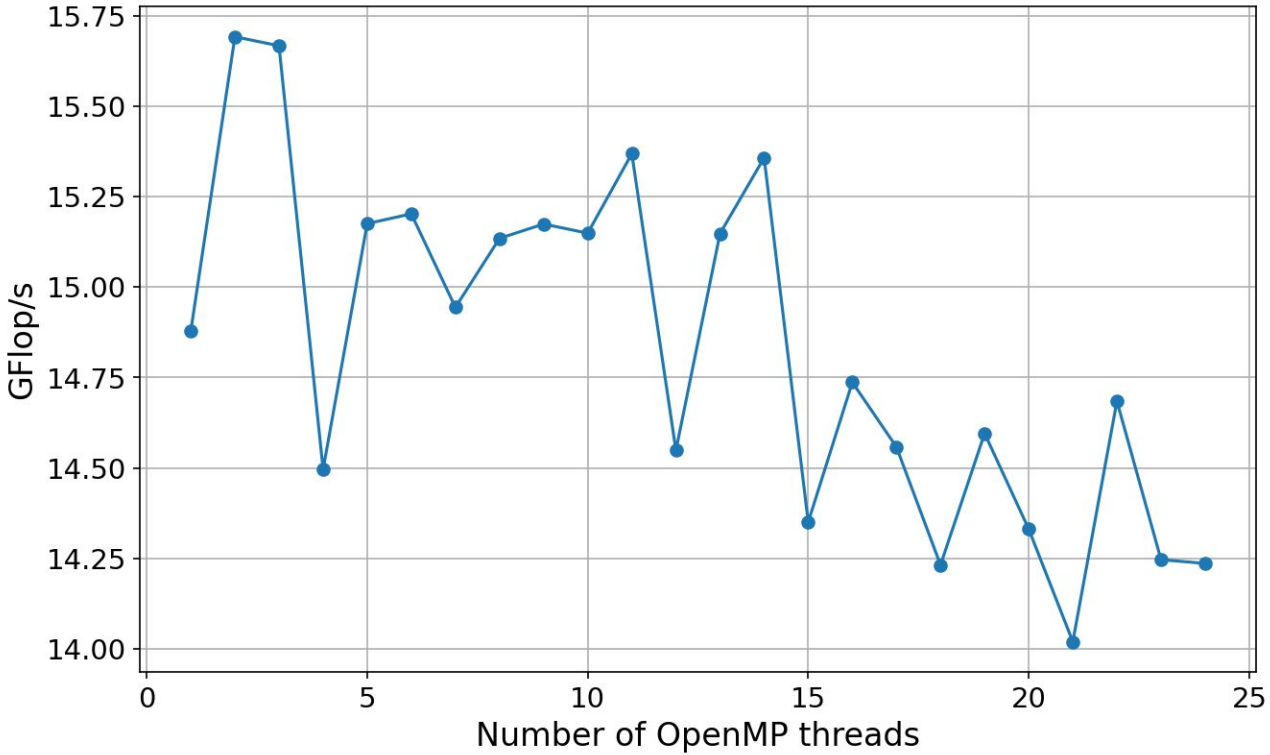
GETRF	TRSM	TRSM	TRSM	TRSM
TRSM	GEMM	GEMM	GEMM	GEMM
TRSM	GEMM	GEMM	GEMM	GEMM
TRSM	GEMM	GEMM	GEMM	GEMM
TRSM	GEMM	GEMM	GEMM	GEMM

GETRF : Sequential Version (Block-ed)



GETRF : OpenMP

Matrix size: 2048
Block size: 64



GETRF : MPI (1)

1 - GETRF ⟶ broadcast result

2 - Broadcast TRSM results of the first line
Send TRSM results of the second line to processes computing GEMM

3 - Receive TRSM results if needed and compute GEMM

4 - Apply the process on A'_{11}

Block-cyclic distribution

0	2	0	2	0	2
1	3	1	3	1	3
0	2	0	2	0	2
1	3	1	3	1	3
0	2	0	2	0	2
1	3	1	3	1	3

GETRF : MPI (2)

1 - GETRF ⟶ broadcast result

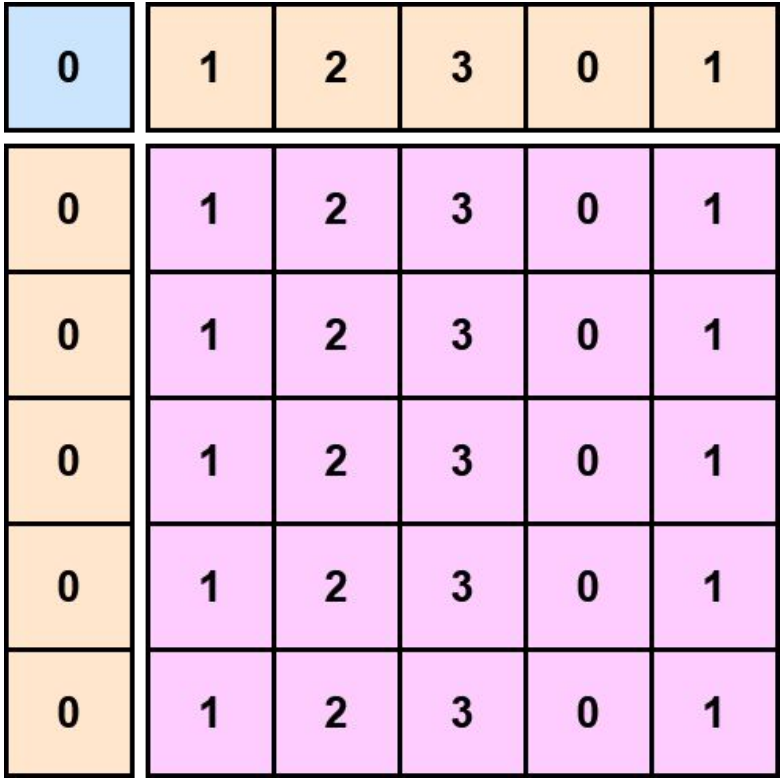
2 - Each process compute its TRSM in the first line / Process 0 computes the TRSM in the second line and broadcast the result.

3 - Compute GEMM of the second line

4 - Repeat 2 and 3 on remaining lines

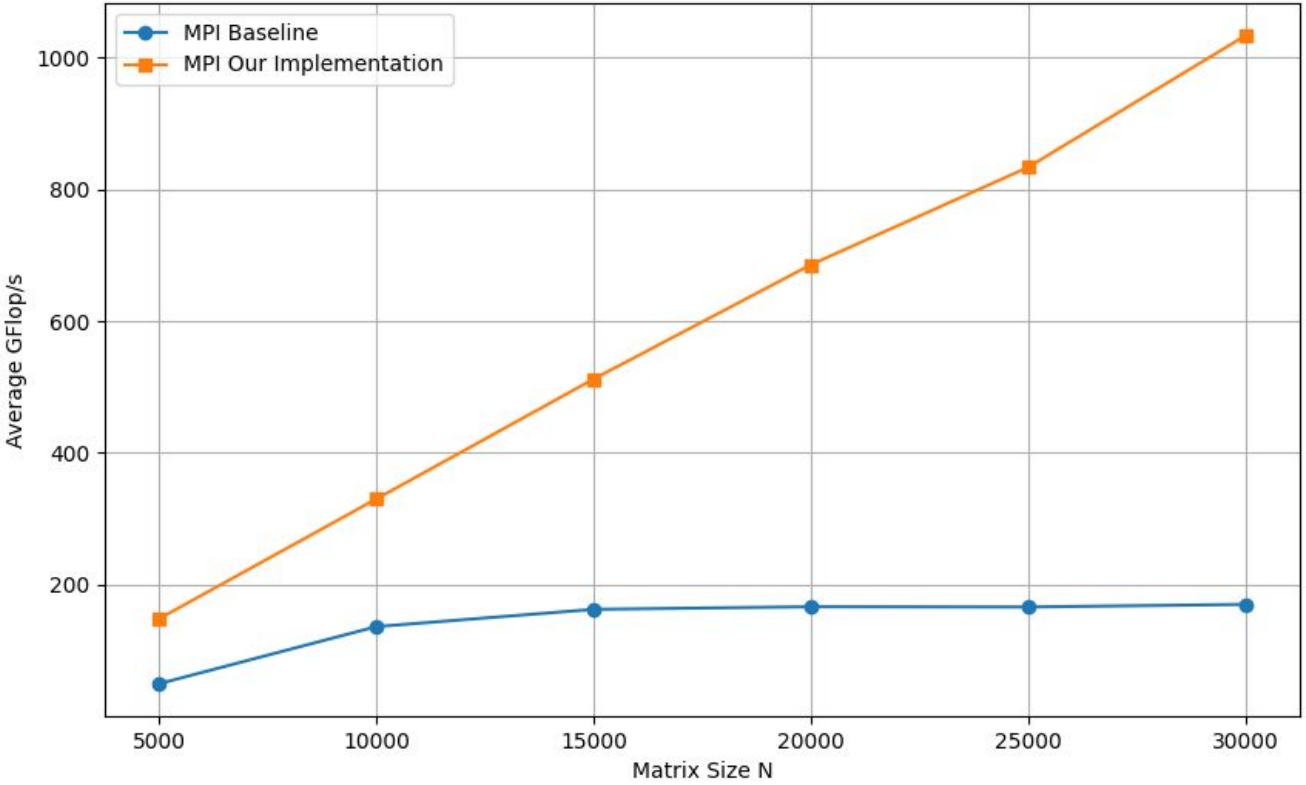
5 - Apply the process on A'_{11}

Column distribution



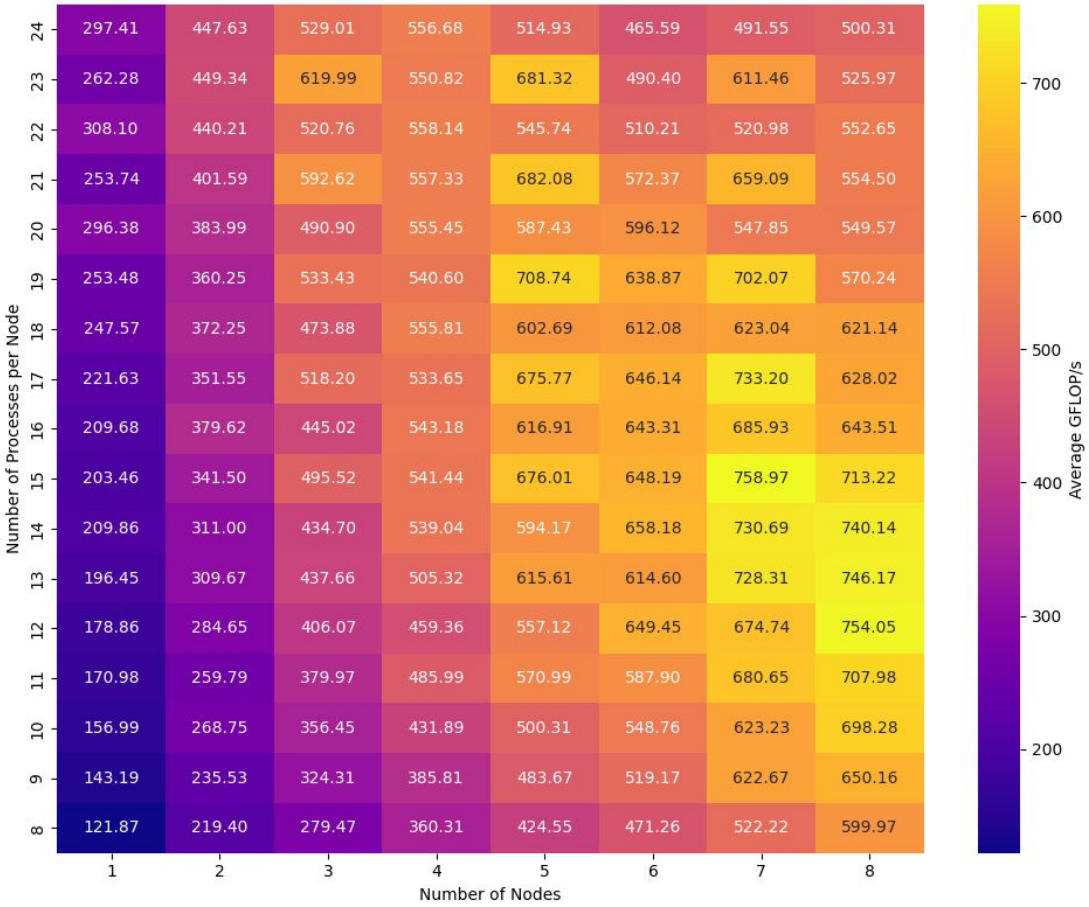
GETRF : MPI (4)

blocks size = 320
8 nodes
12 processes per node



GETRF : MPI (5)

M = N = 30000
blocks = 150

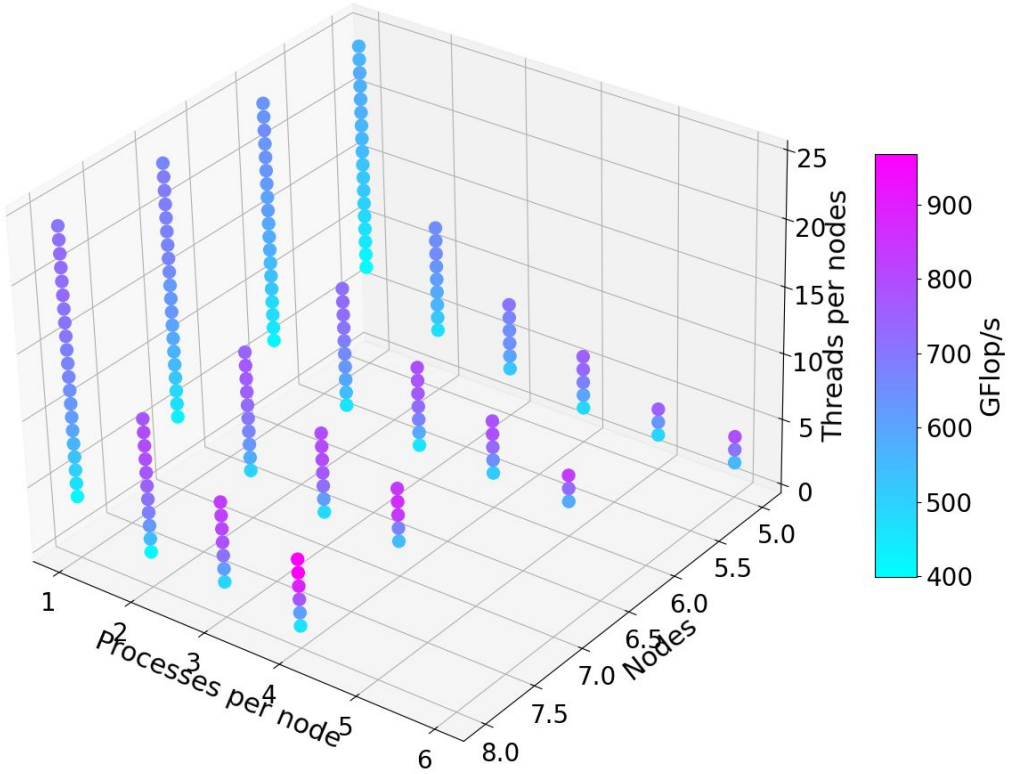


GETRF : MPI + OpenMP

M = N = 35000

MPI blocks size = 1024

OMP blocks size = 100



Experimentation

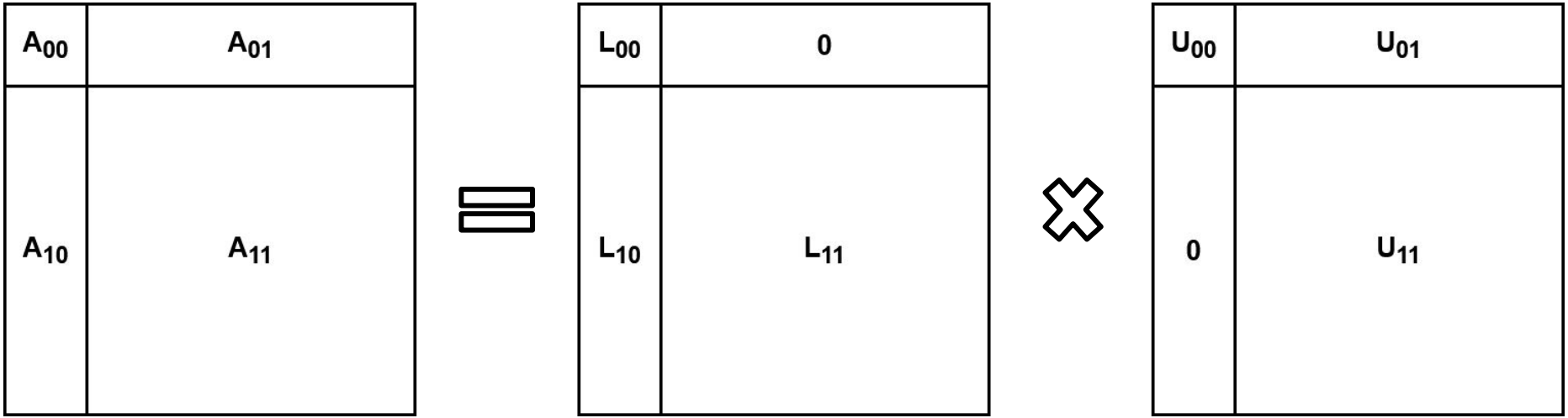
- Turbo boost enabled on miriel039 and miriel040;
- Cluster-On-Die BIOS option is disabled on miriel023;
  - We noticed those misconfigurations during the analysis of our results and benchmarks.

Conclusion

- **Great performance** for **GEMM**, and works on any matrix size and transposition;
- **Relatively good results** for **GETRF**, and works on any matrix size;
- Many **parameters left to be tested** such as:
  - Optimize MPI + OpenMP block sizes
  - Process binding for MPI
- **Improvements** to be done:
  - GEMM MPI algorithm (Cannon's algorithm, 2.5D algorithms)
  - Cuda (GETRF especially)
- **Other solutions/implementations** to test:
  - GETRF OpenMP with for (instead of tasks)

Questions ?

GETRF : Blocking (1)



$$(1) \quad L_{00}U_{00} = A_{00}$$

$$(2) \quad L_{10}U_{00} = A_{10}$$

$$(3) \quad L_{00}U_{01} = A_{01}$$

$$(4) \quad L_{10}U_{01} + L_{11}U_{11} = A_{11}$$

GETRF : Blocking (2)

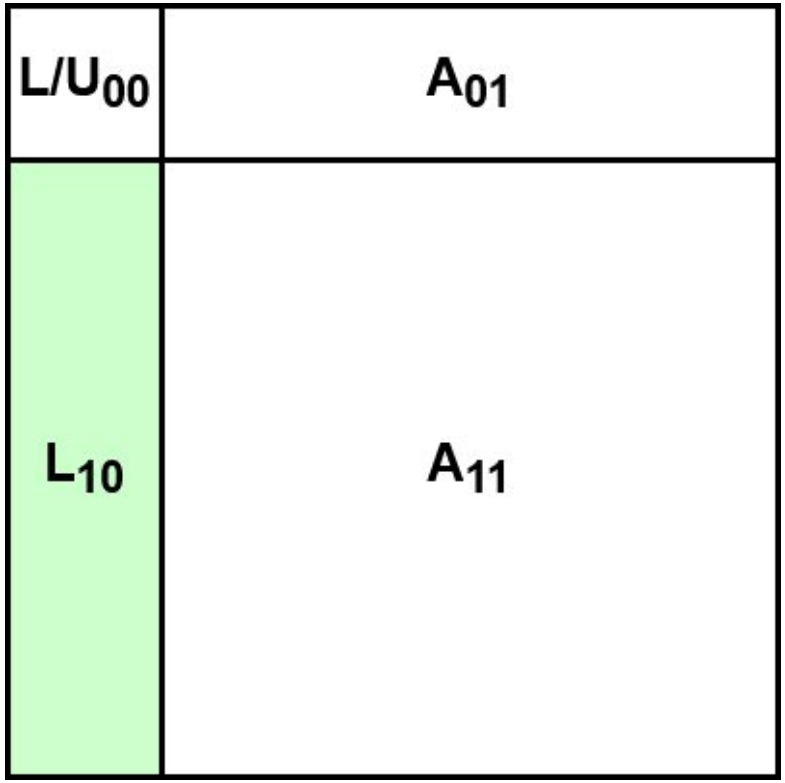
$$(1) \quad L_{00}U_{00} = A_{00}$$

$$GETRF(A_{00}) = (L_{00}, U_{00})$$

GETRF : Blocking (3)

$$(2) \quad L_{10}U_{00} = A_{10}$$

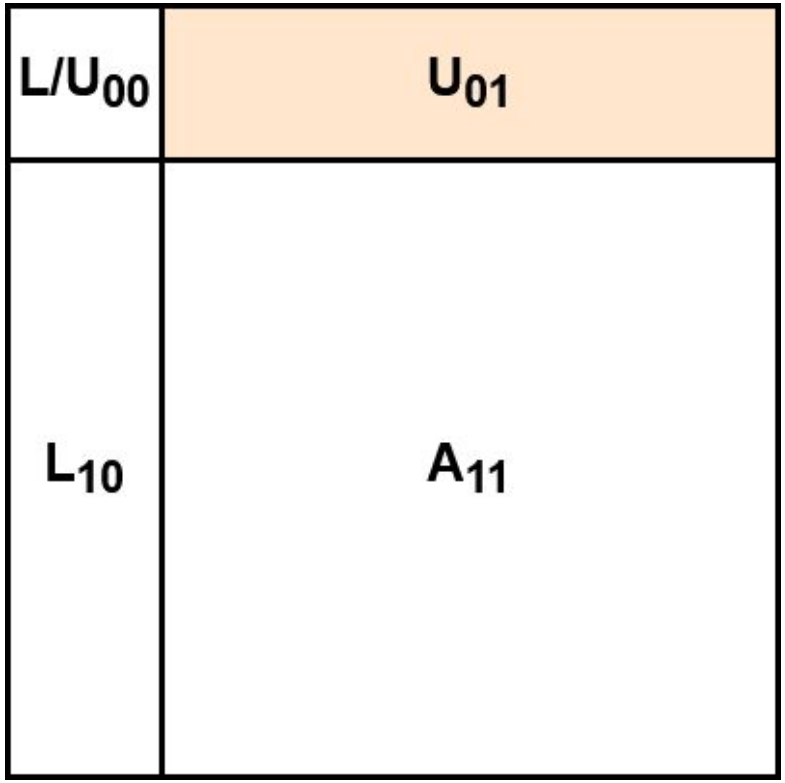
$$TRSM(A_{10}, U_{00}) = L_{10}$$



GETRF : Blocking (4)

$$(3) \quad L_{00}U_{01} = A_{01}$$

$$TRSM(A_{01}, L_{00}) = U_{01}$$

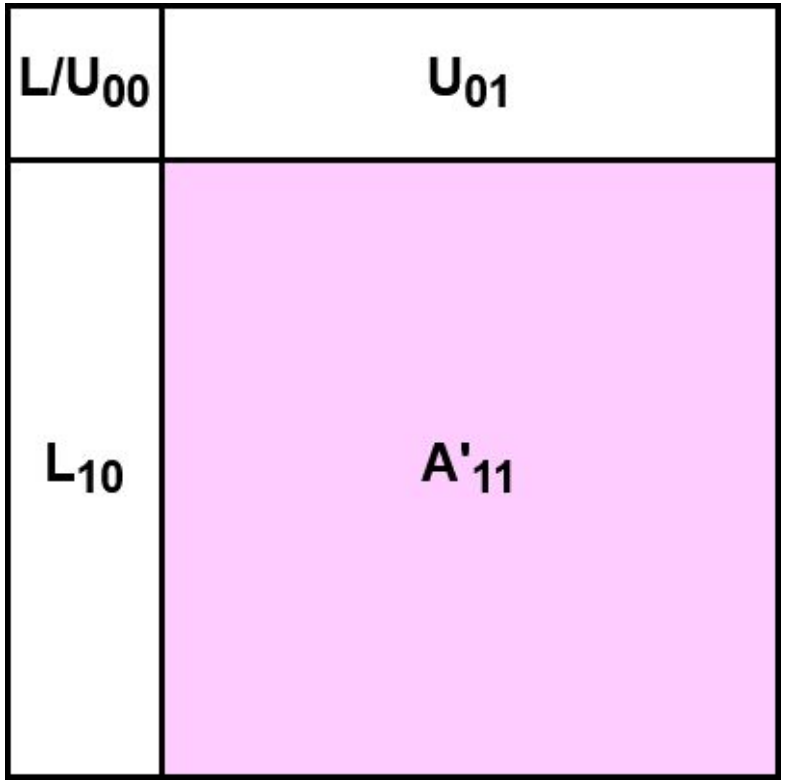


GETRF : Blocking (5)

$$(4) \quad L_{10}U_{01} + L_{11}U_{11} = A_{11}$$

$$(4) \quad A'_{11} = \underbrace{A_{11} - L_{10}U_{01}}_{\text{GEMM}} = L_{11}U_{11}$$

Apply the process on A'_{11} to compute L_{11} and U_{11}



GETRF : Blocking (6)

1 - Compute L/U_{00} using GETRF

2 - Compute L_{10} and U_{01} using TRSM

3 - Compute A'_{11} using GEMM

4 - Apply the process on A'_{11}

GETRF	TRSM	TRSM	TRSM	TRSM
TRSM	GEMM	GEMM	GEMM	GEMM
TRSM	GEMM	GEMM	GEMM	GEMM
TRSM	GEMM	GEMM	GEMM	GEMM
TRSM	GEMM	GEMM	GEMM	GEMM

GEMM : Theoretical performances

$$\text{GFlop/s} = \text{CPU Freq} \times \frac{\text{SIMD Width}}{\text{Type size}} \times \text{FMA} \times \text{FMA Throughput}$$

GEMM : Theoretical performances

model name : Intel(R) Xeon(R)
CPU E5-2680 v3 @ 2.50GHz

AVX2

FMA (a += b*c)

$$\text{GFlop/s} = 2.5\,\text{GHz} \times \frac{256}{64} \times 2 \times 2 = 40\,\text{GFlop/s}$$

sizeof(double)

Latency and Throughput

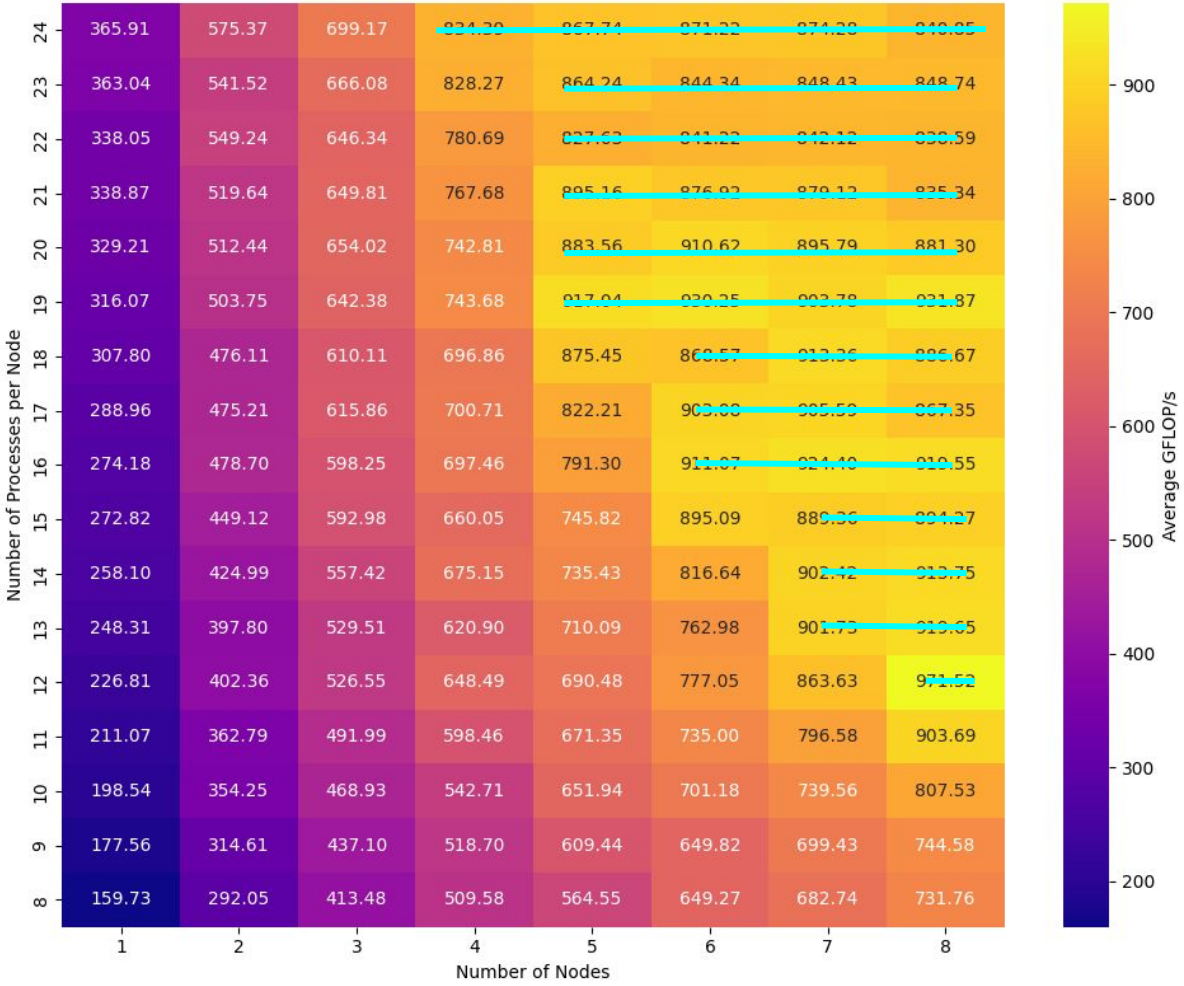
Architecture	Latency	Throughput (CPI)
Alderlake	4	0.5
Icelake Intel Core	4	0.5
Icelake Xeon	4	0.5
Sapphire Rapids	4	0.5
Skylake	4	0.5

$$\text{GFlop/s} = \text{CPU Freq} \times \frac{\text{SIMD Width}}{\text{Type size}} \times \text{FMA} \times \text{FMA Throughput}$$

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_ALL&text=_mm256_fmadd_pd

M = N = 30000

blocks = 320



M = N = 30000
blocks = 320