

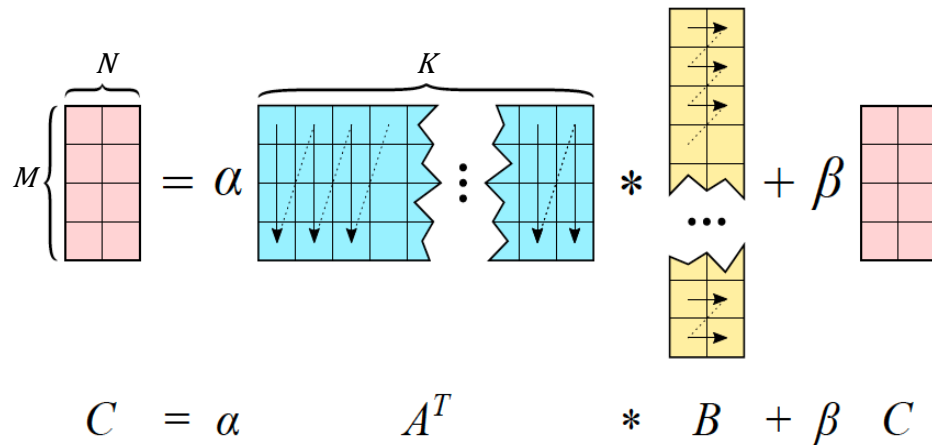
Case study:

**Tall & Skinny Matrix-Transpose Times  
Tall & Skinny Matrix (TSM TTS M)  
Multiplication**



- Block of vectors  $\rightarrow$  Tall & Skinny Matrix (e.g.  $10^7 \times 10^1$  dense matrix)
- Row-major storage format (see SpMVM)
- Block vector subspace orthogonalization procedure requires, e.g. computation of scalar product between vectors of two blocks
- TSMTTSM Mutlification

Assume:  $\alpha = 1$ ;  $\beta = 0$



# TSMITTSM Multiplication



General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, e.g. from Intel MKL<sup>1</sup>:

$$C_{mn} = \sum_{k=1}^K A_{mk} B_{kn}, \quad m = 1..M, n = 1..N$$

double

System	P <sub>peak</sub> [GF/s]	b <sub>s</sub> [GB/s]	Size	Perf.	Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s	52 GB/s	SQ	160 GF/s	91%
			TS	16.6 GF/s	6%
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s	65 GB/s	SQ	550 GF/s	95%
			TS	22.8 GF/s	4%

Matrix sizes:

Square (SQ): M=N=K=15,000

Tall&Skinny (TS): M=N=16 ; K=10,000,000

complex double

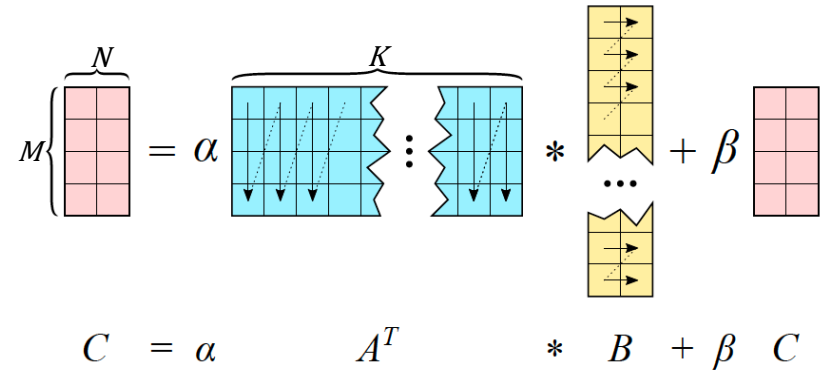
TS@MKL:  
Good or bad?

<sup>1</sup>Intel Math Kernel Library (MKL) 11.3



Computational intensity

$$I = \frac{\text{\#flops}}{\text{\#bytes (slowest data path)}}$$



Optimistic model (minimum data transfer) assuming  $M = N \ll K$  and double precision:

$$I_d \approx \frac{2KMN}{8(KM + KN)} \frac{F}{B} = \frac{MF}{8B}$$

complex double:

$$I_z \approx \frac{8KMN}{16(KM + KN)} \frac{F}{B} = \frac{MF}{4B}$$



Now choose  $M = N = 16 \rightarrow I_d \approx \frac{16 F}{8 B}$  and  $I_z \approx \frac{16 F}{4 B}$ , i.e.  $B_d \approx 0.5 \frac{B}{F}$ ,  $B_z \approx 0.25 \frac{B}{F}$

Intel Xeon E5 2660 v2 ( $b_S = 52 \frac{GB}{s}$ )  $\rightarrow P = 104 \frac{GF}{s}$  (double)

Measured (MKL):  $16.6 \frac{GF}{s}$

Intel Xeon E5 2697 v3 ( $b_S = 65 \frac{GB}{s}$ )  $\rightarrow P = 240 \frac{GF}{s}$  (double complex)

Measured (MKL):  $22.8 \frac{GF}{s}$

**$\rightarrow$  Potential speedup: 6–10x vs. MKL**

# Can we implement a better TSMTTSM kernel than Intel?



```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5 #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8 #pragma simd
9       for (int acol = 0; acol < m; acol++) {
10        c_tmp[bcol*m+acol] +=
11          a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12          a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13      }
14    }
15  }
16
17 #pragma omp critical
18   for (int bcol = 0; bcol < n; bcol++) {
19 #pragma simd
20     for (int acol = 0; acol < m; acol++) {
21       c[bcol*m+acol] += c_tmp[bcol*m+acol];
22     }
23   }
24 }
```

Thread local copy of small (results) matrix

Long Loop (k): Parallel

Outer Loop Unrolling

Compiler directives

Most operations  
in cache

Reduction on  
small result matrix

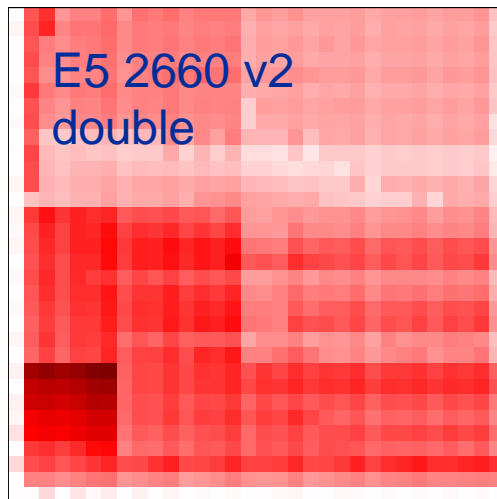
Not shown: Inner Loop boundaries (n,m) known at compile time (kernel generation)  
k assumed to be even

# TSMITTSM MKL vs. “hand crafted” (OPT)



TS: M=N=16 ; K=10,000,000

System	$P_{\text{peak}} / b_s$	Version	Performance	RLM Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s 52 GB/s	TS OPT	98 GF/s	94 %
		TS MKL	16.6 GF/s	16 %
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s 65 GB/s	TS OPT	159 GF/s	66 %
		TS MKL	22.8 GF/s	9.5 %



Speedup  
vs. MKL:  
5x – 25x

