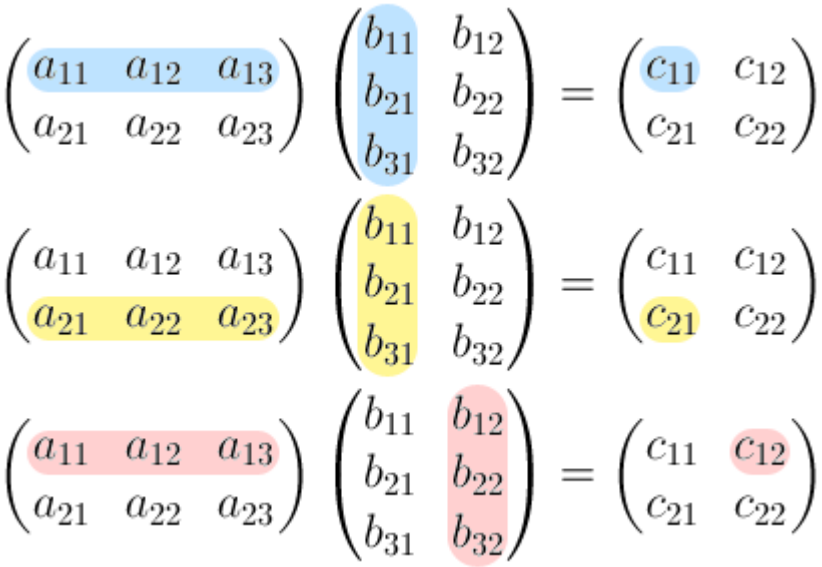
NUMA-aware Matrix-Matrix-Multiplication

Max Reimann, Philipp Otto

About this talk

- Objective:
  Show how to improve performance of algorithms in a NUMA-system with MMM as an example
- Code was written in C with numa.h, pthread.h
- Tested on FSOC
  - ubuntu-0101
    - 2 Nodes, 24 Cores
  - dl980
    - 8 Nodes, 128 Cores
- Compiled with gcc
  - –O3

Naïve Matrix-Matrix-Multiplication



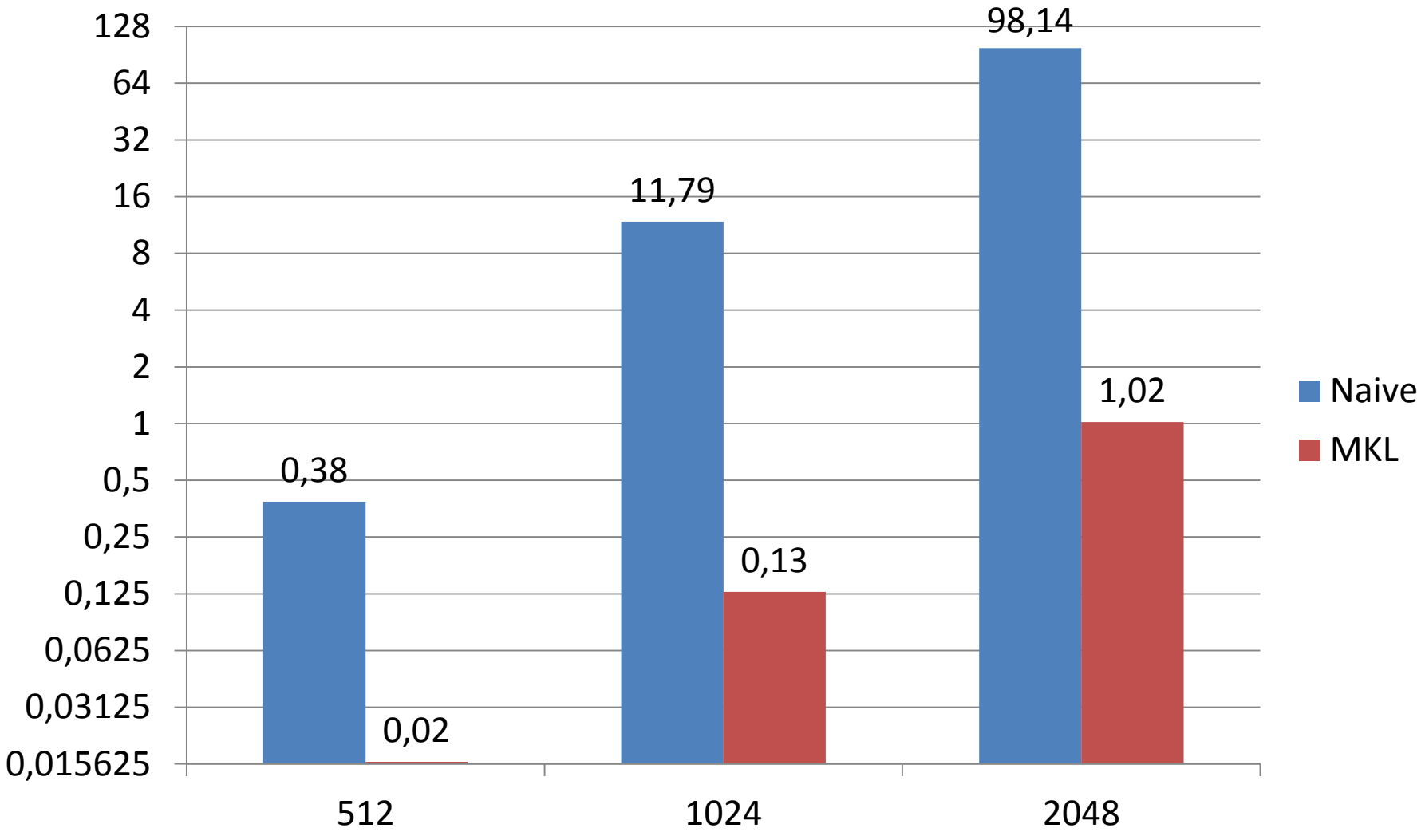
- We will examine MMM for large n x n matrices
- $\mathcal{O}(n^3)$

http://www.mathematrix.de/wp-content/uploads/matrixmul2.png

Naïve MMM implementation

```
1  #define IDX(Y, X) (ndim * Y + X)
2
3  int i, j, k;
4
5  for (i = 0; i < ndim; i++) {
6    for (k = 0; k < ndim; k++) {
7      for (j = 0; j < ndim; j++) {
8        matrixC[IDX(i, k)] += matrixA[IDX(i, j)] * matrixB[IDX(j, k)];
9      }
10   }
11 }
```

Performance of Naive vs. MKL



dl980 on one core

Intel Math Kernel Library (MKL)

- **BLAS**: Basic Linear Algebra Subprograms
  - Standard for Linear Algebra

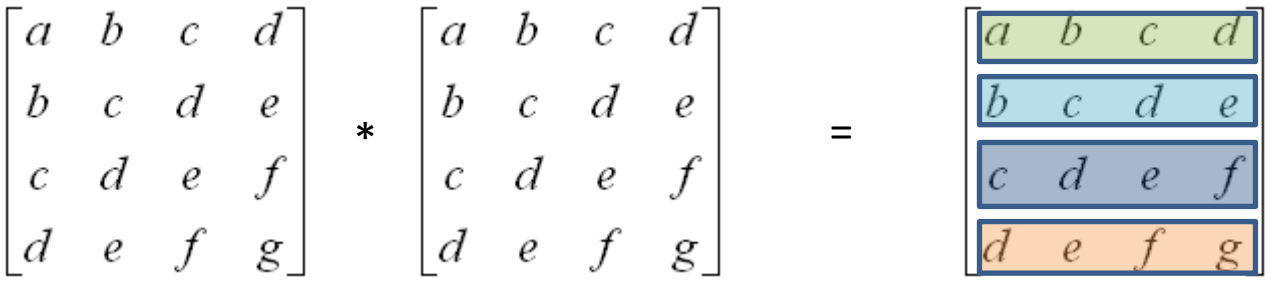
- **MKL**:
  - Implements BLAS for Intel hardware
  - Vectorized and threaded for highest performance

Analysis of Naïve MMM

- Testsetup
  - Use ubuntu-numa machine
  - No thread or memory pinning
- Use numatop/pcm
- Performance tools show:
  - Unused cores (obvious)
  - QPI cannot be fully loaded with one thread

Parallelization I

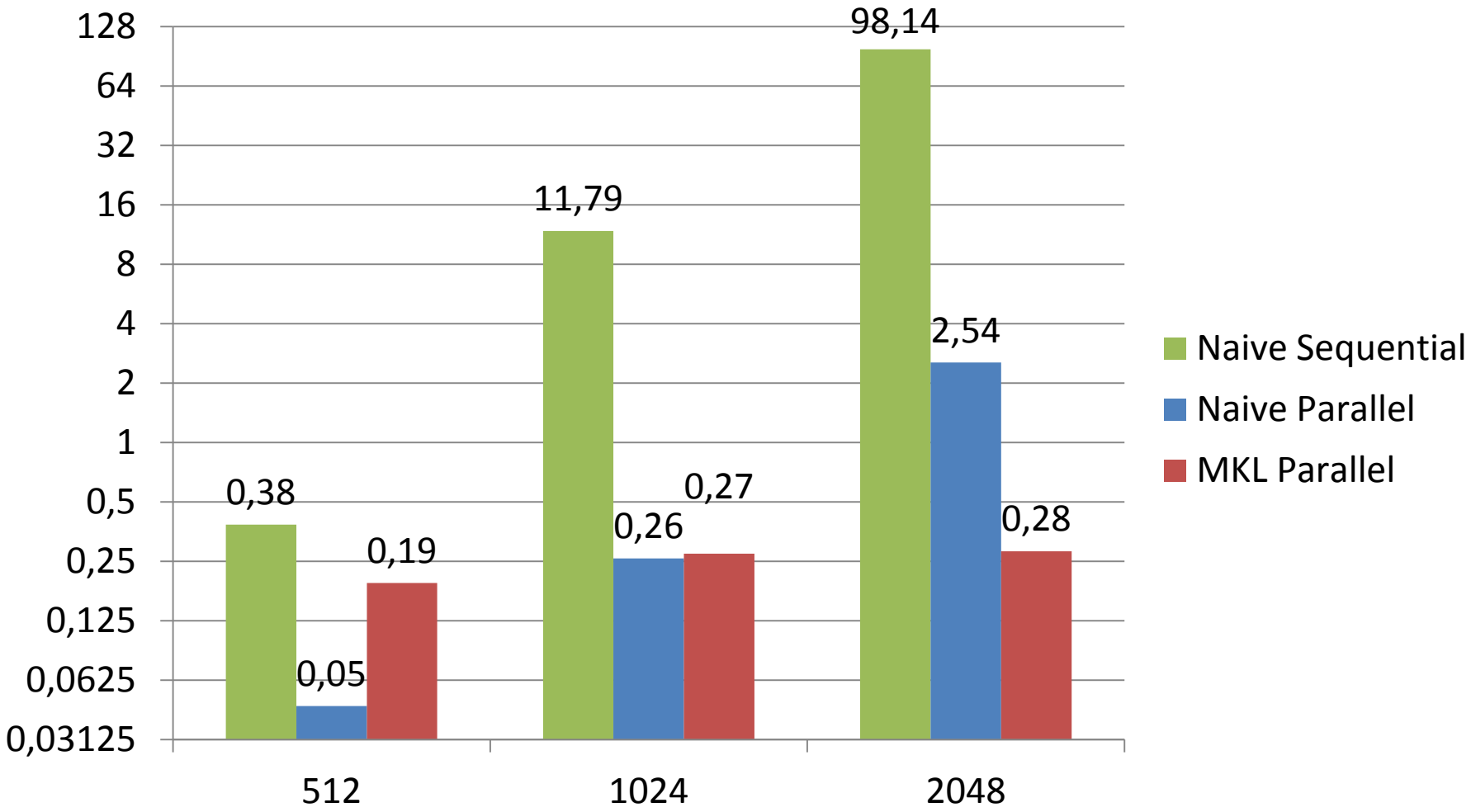
- How can the work be divided?
  - 1. Partition computation of matrixC by rows or columns
    - Problem: All threads need matrixA and matrixB
    - Solution:
      - Accept overhead for remote memory access or
      - Copy input/output matrices to the other nodes (preprocessing)



Parallelization – Partition by rows

```
1  int i, j, k, sum;
2
3  for (i = startRow; i < endRow; i++) {
4    for (k = 0; k < ndim; k++) {
5      sum = 0;
6      for (j = 0; j < ndim; j++) {
7        sum += matrixA[IDX(i, j)] * matrixB[IDX(j, k)];
8      }
9      matrixC[IDX(i, k)] += sum;
10   }
11 }
```

Parallelization – Partition by rows



dl980 on 128 cores

Parallelization II

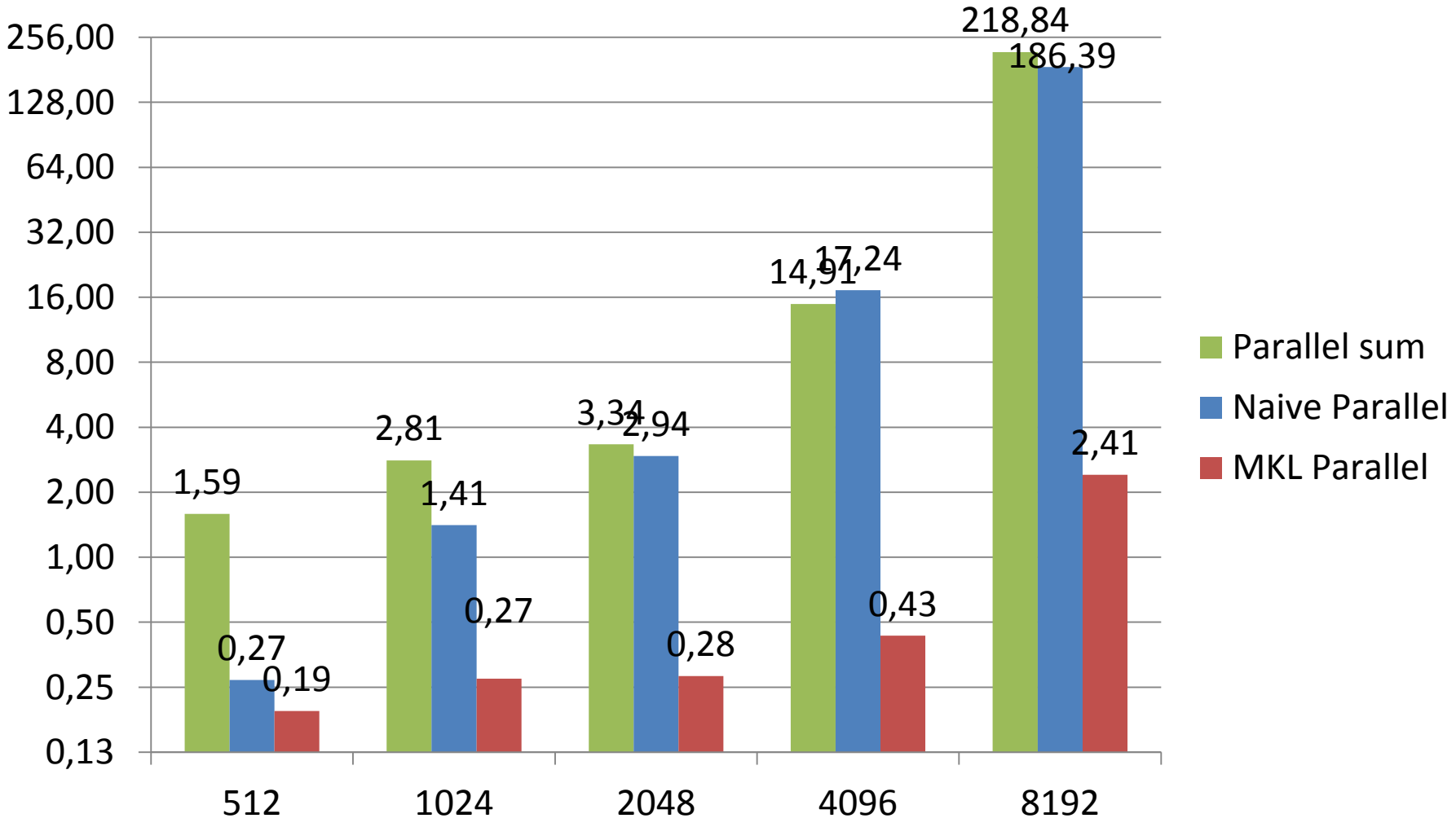
- How can the work be divided?
  - 2. Partition computation of matrixC by summands
    - Benefit:
      - for computing the i-th summand, only the i-th row of matrixA / column of matrixB is needed
      - This allows to only copy the needed parts to the other nodes
    - Disadvantage:
      - matrixB has to be transposed to be able to partition the memory (preprocessing)
      - locking or merging of matrixC is needed

Parallelization II

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} =$$

$$\begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix}$$

Performance of „Parallel Sum“ Method



dl980 on 128 cores

Strassen

- Runtime Complexity:
  - Naive algorithm $\mathcal{O}(n^3)$
- Can we get better?
  - Strassens algorithm, published 1969, was the first to improve asymptotic complexity
  - Runtime $\mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}(n^{2.8})$
    - Algorithms today can get O($n^{2.35}$), but are not pratical
  - Uses only 7 multiplications instead of 8 per recursion step

Matrix definition

For matrices A,B,C with dimension $n = 4k, k \in \mathbb{N}$
A,B,C can be viewed as 2x2 block matrices:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix},\ B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix},\ C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Conventional Algorithm uses 8 (expensive) multiplications:

$$\begin{aligned} C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\ C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\ C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2} \end{aligned}$$

Strassen's algorithm

Define temporary matrices:

$M_1 := (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2})$
$M_2 := (A_{2,1} + A_{2,2}) \cdot B_{1,1}$
$M_5 := (A_{1,1} + A_{1,2}) \cdot B_{2,2}$
$M_6 := (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2})$

$M_3 := A_{1,1} \cdot (B_{1,2} - B_{2,2})$
$M_4 := A_{2,2} \cdot (B_{2,1} - B_{1,1})$
$M_7 := (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})$

Only 7 multiplications!

Compose final matrix

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,1} = M_2 + M_4$$
$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

Strassen - Example

Substituting the $M_i s$ by their term gives back the original formula:

$$
\begin{aligned}
C_{1,2} &= M_3 + M_5 \\
&= A_{1,1} \cdot (B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
&= A_{1,1}B_{1,2} - A_{1,1}B_{2,2} + A_{1,1}B_{2,2} + A_{1,2}B_{2,2} \\
&= A_{1,1}B_{1,2} + A_{1,2}B_{2,2}
\end{aligned}
$$

$$
\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \cdot \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}
$$

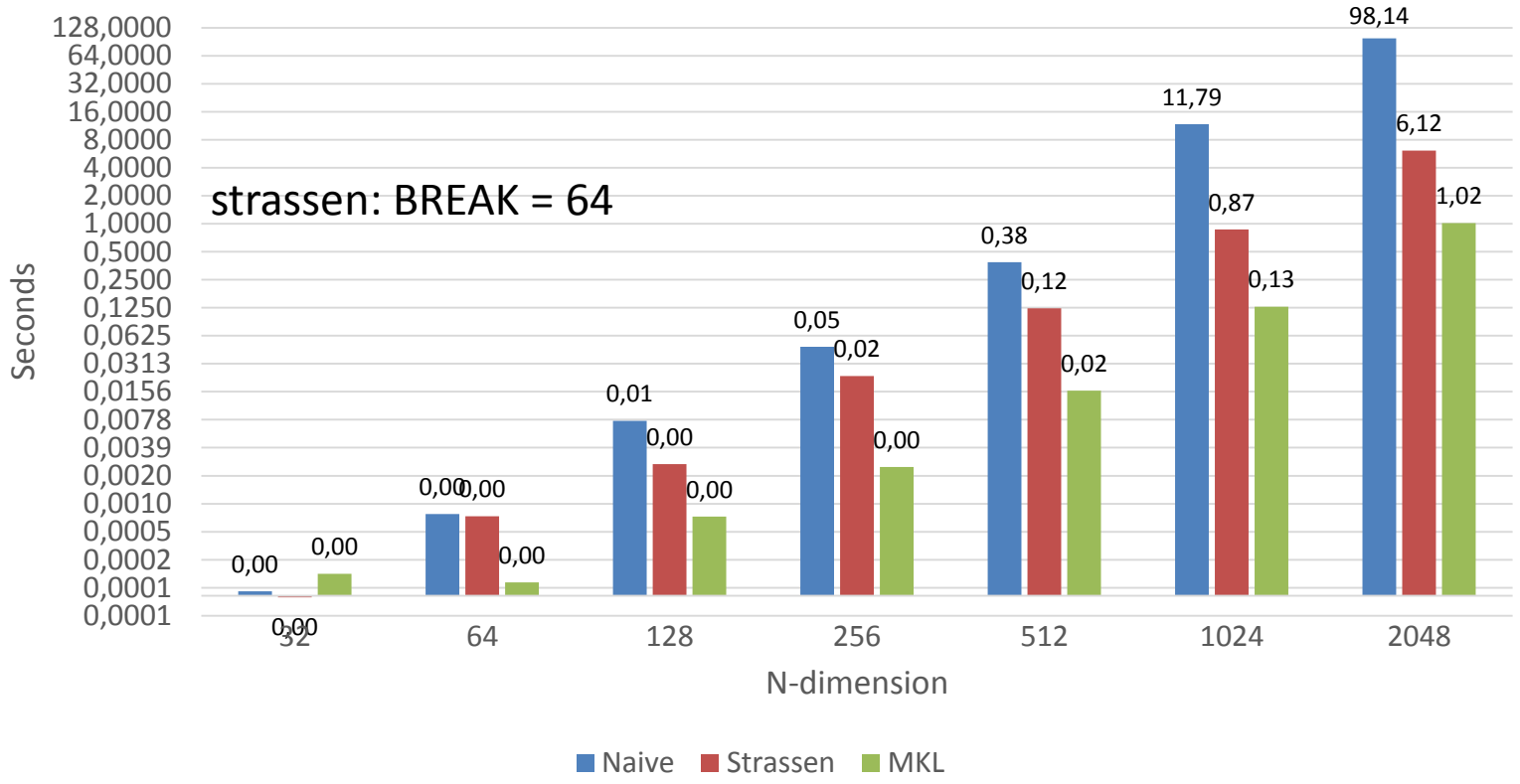
Strassen - Analysis

- Cost: 7 multiplications and 18 additions
  - 8 multiplications and 4 additions for naïve
- Only practical for large matrices n > 1000
  - Although our results indicate otherwise (later)
- Define cutoff point for recursion
  - If n is sufficiently small, do naïve multiplication

Strassen - Implementation

```
1  void strassen_multiply(int n, matrix a, matrix b, matrix c, matrix d) {
2    if (n <= BREAK) {
3      naive_multiply(n, a, b, c, d);
4    } else {
5      n /= 2;
6
7      sub(n, a12, a22, d11);
8      add(n, b21, b22, d12);
9      strassen_multiply(n, d11, d12, c11, d21); // p7
10     sub(n, a21, a11, d11);
11     add(n, b11, b12, d12);
12     strassen_multiply(n, d11, d12, c22, d21); // p6
13     // ...
14   }
15 }
```

Execution Time: Single-threaded



dl980 on 1 core

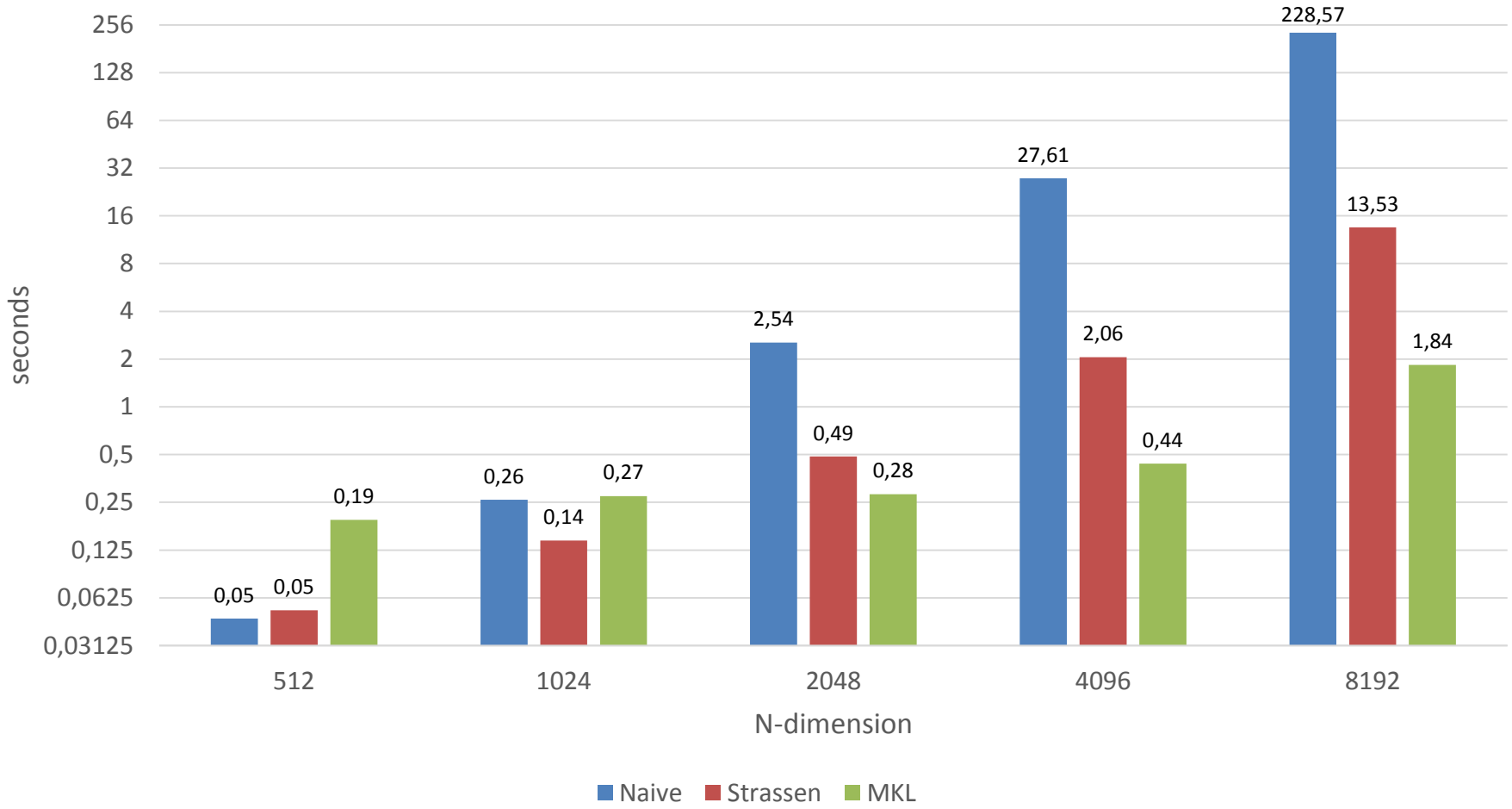
Parallelization of Strassen I

- Data dependencies:
  - Have to do additions in M_i before multiplication
    - e.g. $\mathrm{M}_1 = \left(A_{1,1} + A_{2,2}\right) \cdot \left(B_{1,1} + B_{2,2}\right)$
  - Have to calculate M_i before calculating C
    - $C_{1,2} = M_3 + M_5$
- Easiest solution:
  - Calculate in M_is in parallel
  - Then calculate $C_{i,j}$ in parallel

Parallelization of Strassen II

- Level 1 can be scheduled to 7 threads
- Level n can be scheduled to 7^n threads
  - Most systems have number of processors on base 2
- We used manual parallelization
  - 49 distinct functions for Ms and 16 for Cs
  - Code bloating and not scalable, BUT:
- Automatic parallelization is hard
  - Thread load becomes very unbalanced
  - Every level needs 7 temporary matrices
    - Exponential rising memory requirements

Execution Time – 49 Threads

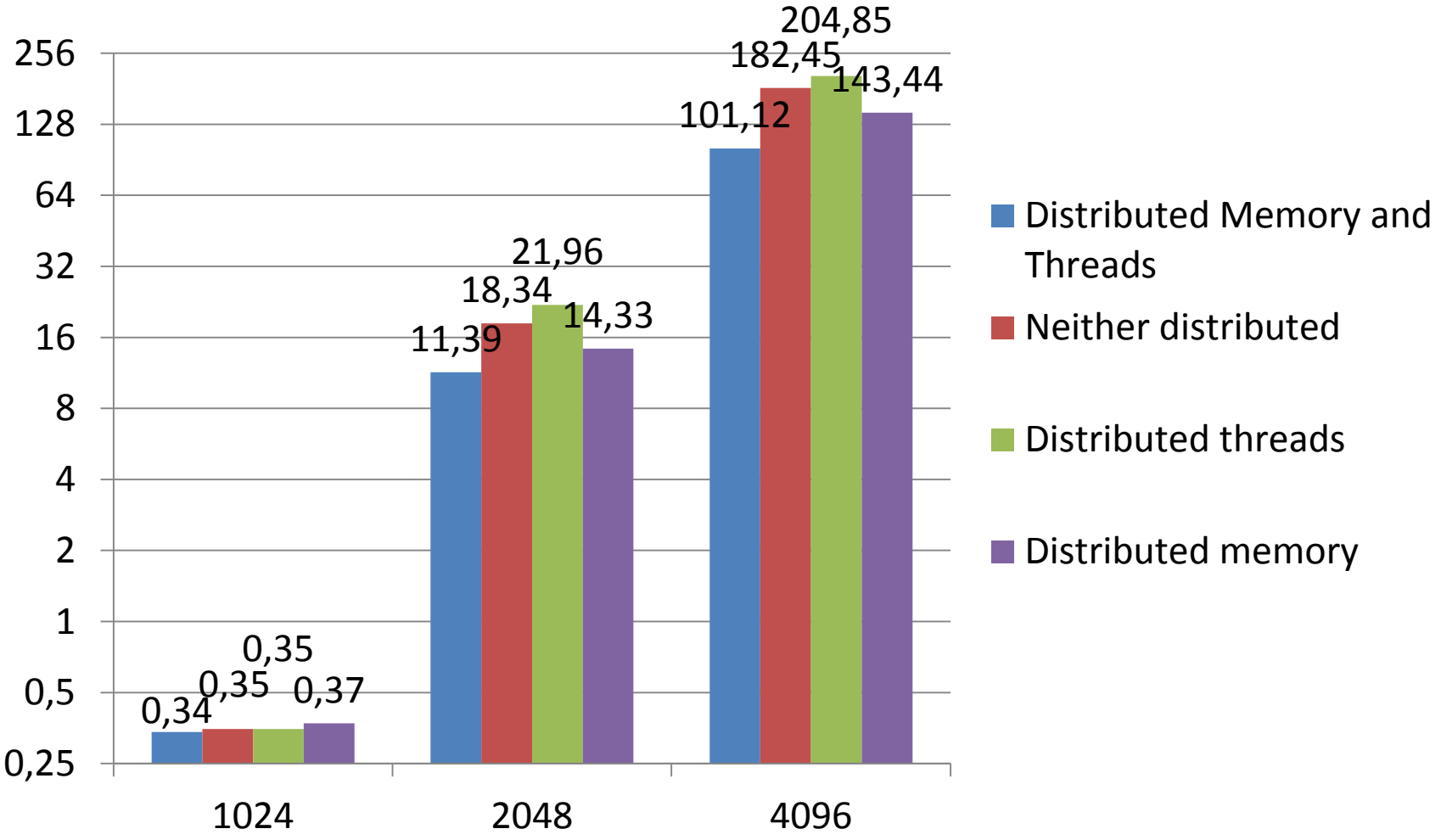


dl980 on 49 cores

NUMA-Optimizations

- Try to have as much memory local as possible to avoid remote memory access
  - Because it is slower by a factor of ~ 1.4
- Partition data and work depending on #nodes and #cores
- Pin threads to nodes with the memory they need
- (Topology for other algorithms)

Distributing memory and threads



Parallel naive on ubuntu-numa0101 on 24 cores

DEMO

Application of NUMA-Optimizations

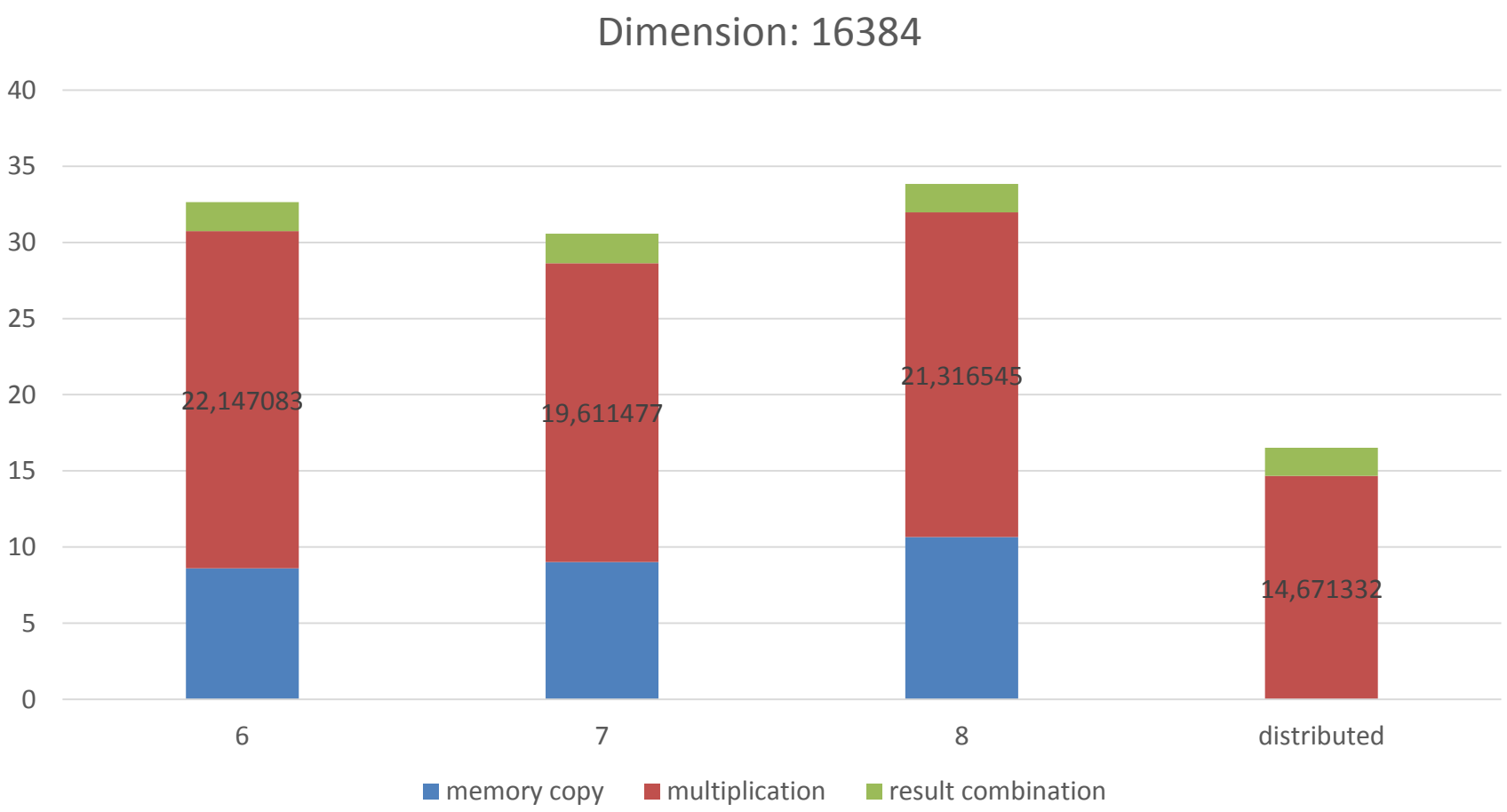
- **Copy** all data to every node:
  - Duration of preprocessing:
    - 11.11s for a 8192x8192 matrix to 8 nodes
- **Partition** data and **move** to corresponding nodes
  - Duration of preprocessing:
    - 1.03s for a 8192x8192 matrix to 8 nodes
- **Pin** threads to nodes
  - `int numa_run_on_node(int node);`

Parallelization – Partition by rows

Copying memory to different nodes

```
1  double** matrixACopies = malloc(NUM_NODES * sizeof(double*));
2  double** matrixBCopies = malloc(NUM_NODES * sizeof(double*));
3  double** matrixCCopies = malloc(NUM_NODES * sizeof(double*));
4
5  for (int i = 0; i < NUM_NODES; ++i) {
6    matrixACopies[i] = (double *) numa_alloc_onnode(matrixSize, i);
7    matrixBCopies[i] = (double *) numa_alloc_onnode(matrixSize, i);
8    matrixCCopies[i] = (double *) numa_alloc_onnode(matrixSize, i);
9
10   memcpy(matrixACopies[i], matrixA, matrixSize);
11   memcpy(matrixBCopies[i], matrixB, matrixSize);
12 }
```

Strassen Memory Distribution Effects



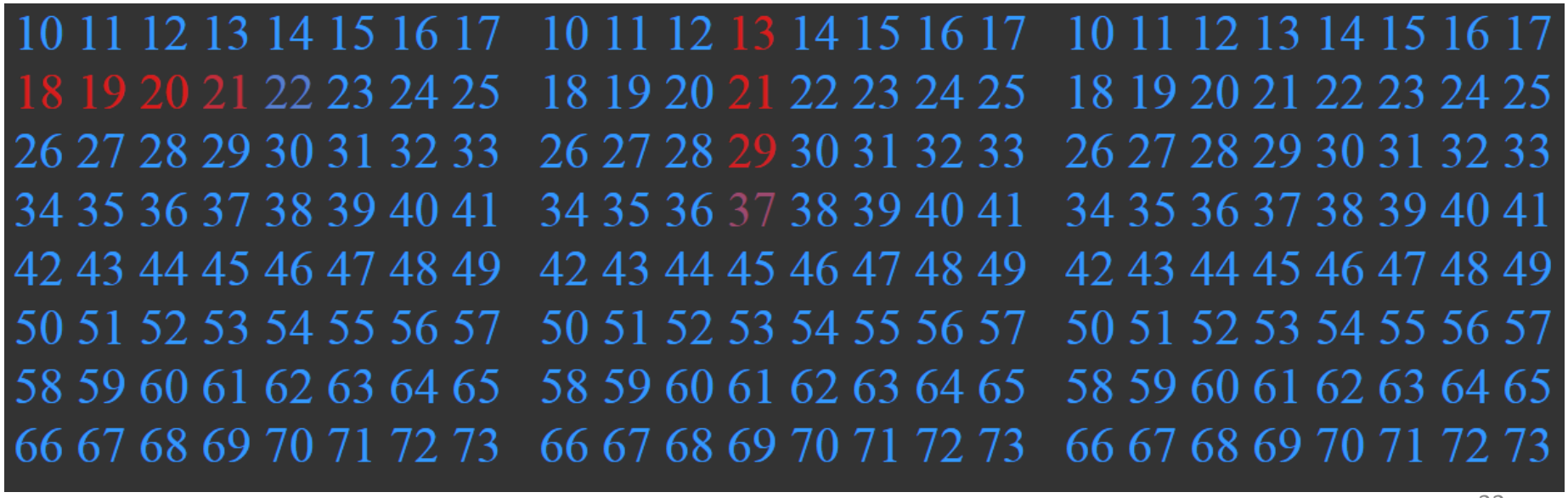
dl980 on 128 core

Other optimization techniques

- **Tiling**
- **Vectorization**
- Scalar replacement
- Precomputation of constants
- (unrolling)

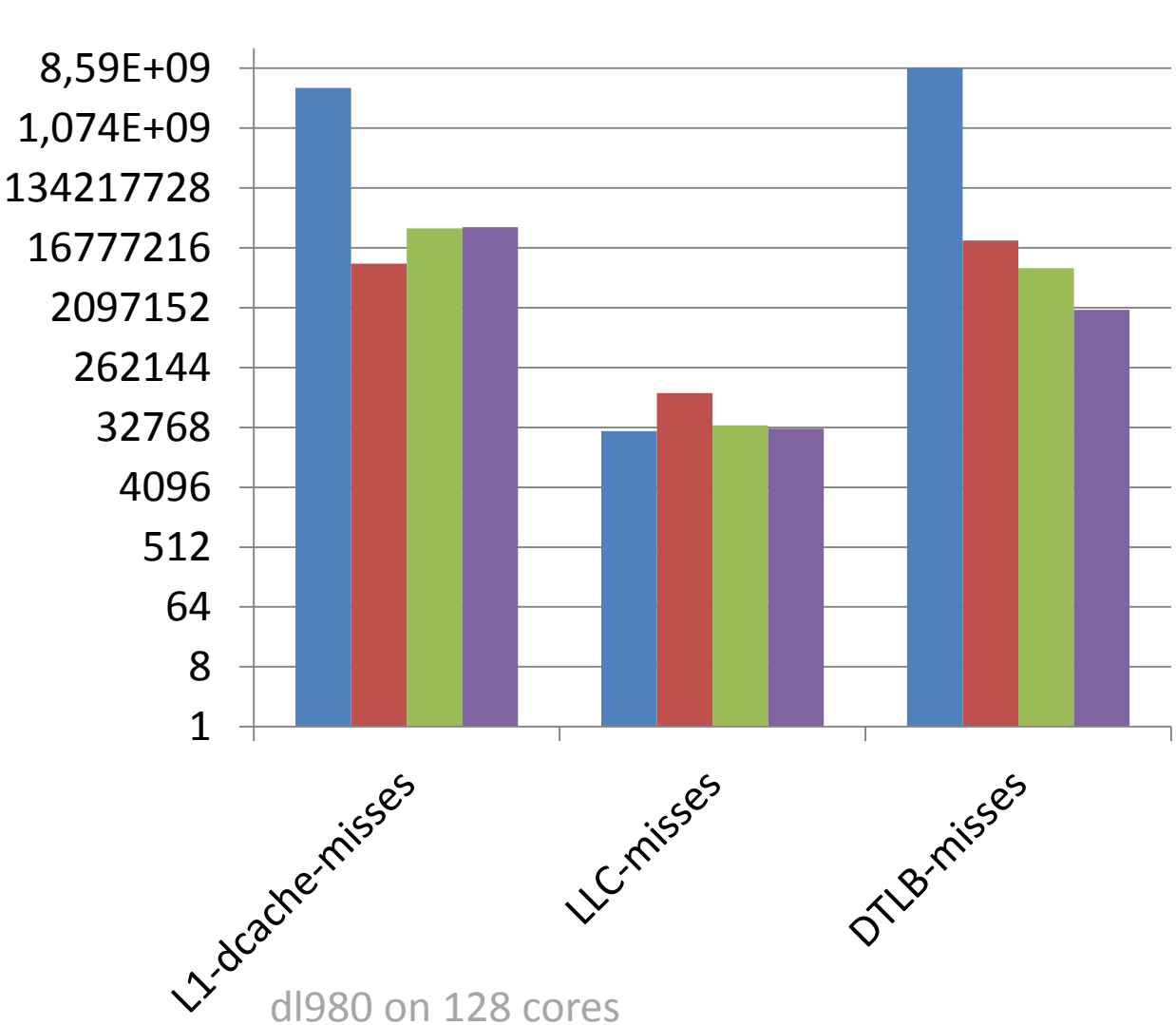
Tiling

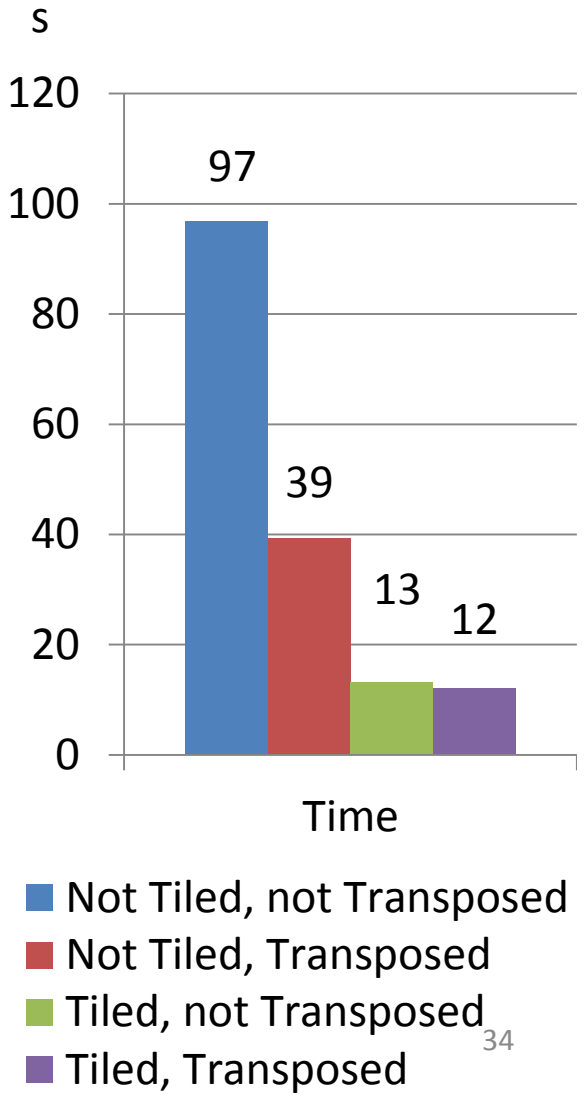
- Divide computational work into tiles to leverage cache
- Tile size depends on cache size
- `gcc -DCLS=$(getconf LEVEL1_DCACHE_LINESIZE)`



Performance of Tiling

```
perf stat -e L1-dcache-misses,LLC-misses,DTLB-misses bin/matrixmult -n 2048
```





dl980 on 128 cores

Vectorization

- SIMD : Single Instruction Multiple Data
- All recent Intel and AMD Processors have Streaming Instructions Extensions (SSE)
- An instruction is simultaneously applied to multiple floats
- Can only operate efficiently on aligned data (16 bit aligned)
- SSE operate on 128bit registers
  - Newer Intel processors have Advanced Vector Instructions (AVX) with 256 bit
  - Dl980 machine only support 128bit operations

Auto-Vectorization

- Can this be done automatically?
  - Gcc –O3 tries to auto-vectorize
    - only possible for simple statements

```
1  void test7(double * restrict a, double * restrict b)
2  {
3      size_t i;
4
5      double *x = __builtin_assume_aligned(a, 16);
6      double *y = __builtin_assume_aligned(b, 16);
7
8      for (i = 0; i < SIZE * SIZE; i++)
9      {
10         x[i] += y[i];
11     }
12 }
```

Assembler

```
1  <+0>:    xor    %eax,%eax
2  <+2>:    movabs $0x800000000,%rdx
3  <+12>:   nopl   0x0(%rax)
4  <+16>:   movapd (%rdi,%rax,1),%xmm0
5  <+21>:   addpd  (%rsi,%rax,1),%xmm0
6  <+26>:   movapd %xmm0,(%rdi,%rax,1)
7  <+31>:   add    $0x10,%rax
8  <+35>:   cmp    %rdx,%rax
9  <+38>:   jne    0x4c0 <test7+16>
10 <+40>:   repz retq
```

Aligned Malloc

```
1  // 1. variant - Intel specific
2  #include <mm_malloc.h>
3  void *memory = _mm_malloc(size, 16);
4
5  // 2. variant
6  void *memory = numa_alloc_onnode(size + 15, node);
7  assert(memory); // some kind of error handling
8  // round up to multiple of 16, add 15 and round down by masking
9  void *alignedMem = (void *)(((unsigned long)memory + 15) & ~0x0f);
```

Example:

- Numa_alloc returns adrr: 0x1232, not 16bit aligned
- We add 15, so addr = 0x1241 or 0b1001001000001
- Now we clear last 4 bits by ANDing ~0x0f (=0xfff0)
- => result 0x1240 is now 16bit aligned

Intrinsics

Example

```
__m128 _mm_mul_ps (__m128 a, __m128 b)
```

Synopsis

```
__m128 _mm_mul_ps (__m128 a, __m128 b)
#include "xmmintrin.h"
Instruction: mulps xmm, xmm
```

Description

Multiply packed single-precision (32-bit) floating-point elements in a and b, and store the results in dst.

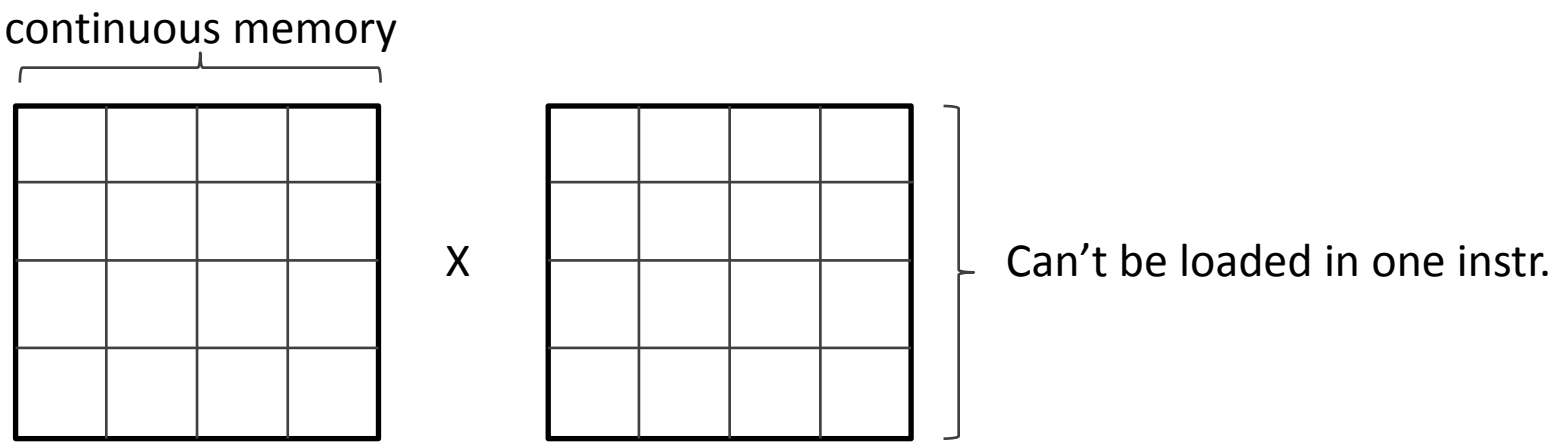
Performance

Architecture	Latency	Throughput
Sandy Bridge	5	1
Westmere	4	1
Nehalem	4	1

Source: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

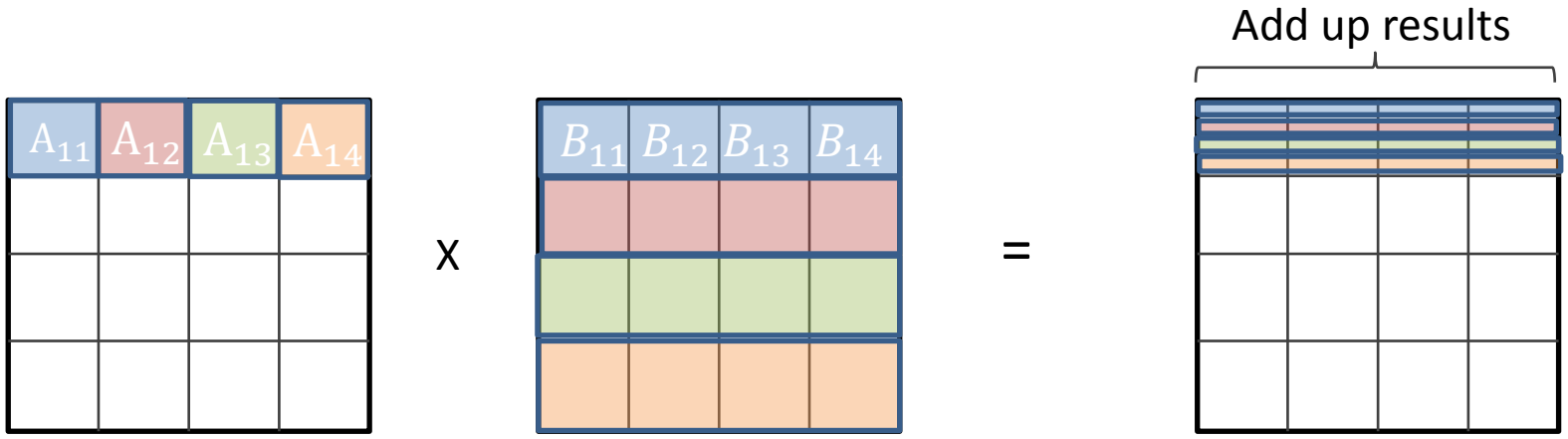
Use Parallelism for MMM

- We try to construct a 4x4 matrix multiplication
- How to process rows ?



Use parallelism for MMM

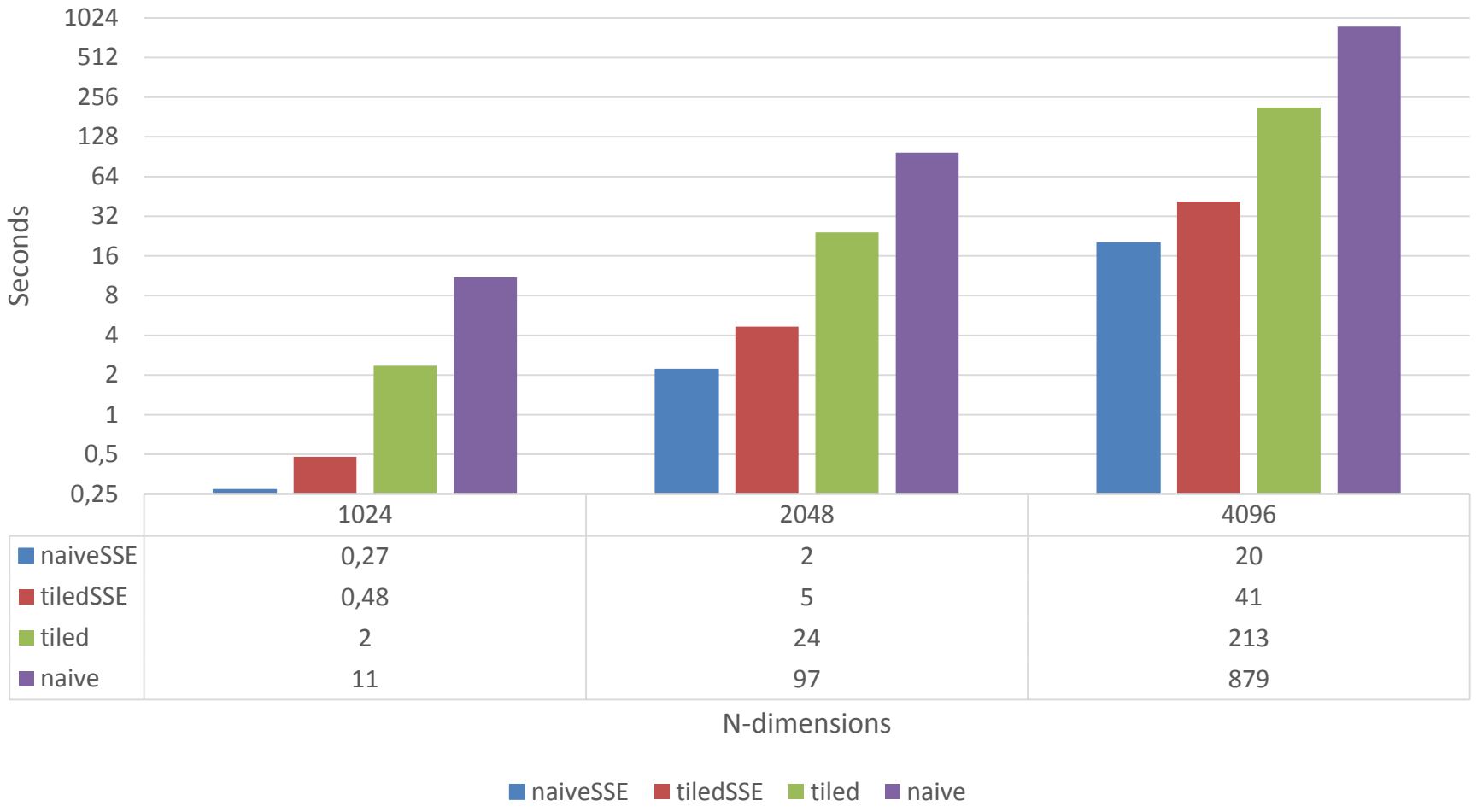
- We try to construct a 4x4 matrix multiplication
- How to process rows ?
- Idea: process all elements of row of B in parallel



4x4 Kernel

```
41 void M4x4_SSE(int n, float *restrict A, float *restrict B, float *restrict C)
42 {
43     __m128 row1 = _mm_load_ps(&B[IDX(0, 0)]);
44     __m128 row2 = _mm_load_ps(&B[IDX(1, 0)]);
45     __m128 row3 = _mm_load_ps(&B[IDX(2, 0)]);
46     __m128 row4 = _mm_load_ps(&B[IDX(3, 0)]);
47     for (int i = 0; i < 4; i++)
48     {
49         //Set all four words with the same value
50         __m128 cell_i1 = _mm_set1_ps(A[IDX(i, 0)]);
51         __m128 cell_i2 = _mm_set1_ps(A[IDX(i, 1)]);
52         __m128 cell_i3 = _mm_set1_ps(A[IDX(i, 2)]);
53         __m128 cell_i4 = _mm_set1_ps(A[IDX(i, 3)]);
54         //multiply i'th row of A with all columns of B
55         __m128 prod1 = _mm_mul_ps(cell_i1, row1);
56         __m128 prod2 = _mm_mul_ps(cell_i2, row2);
57         __m128 prod3 = _mm_mul_ps(cell_i3, row3);
58         __m128 prod4 = _mm_mul_ps(cell_i4, row4);
59         //add up results of i'th row
60         __m128 row = _mm_add_ps(
61                         _mm_add_ps(prod1, prod2),
62                         _mm_add_ps(prod3, prod4));
63         //store result row in C
64         _mm_store_ps(&C[IDX(i, 0)], row);
65     }
66 }
```

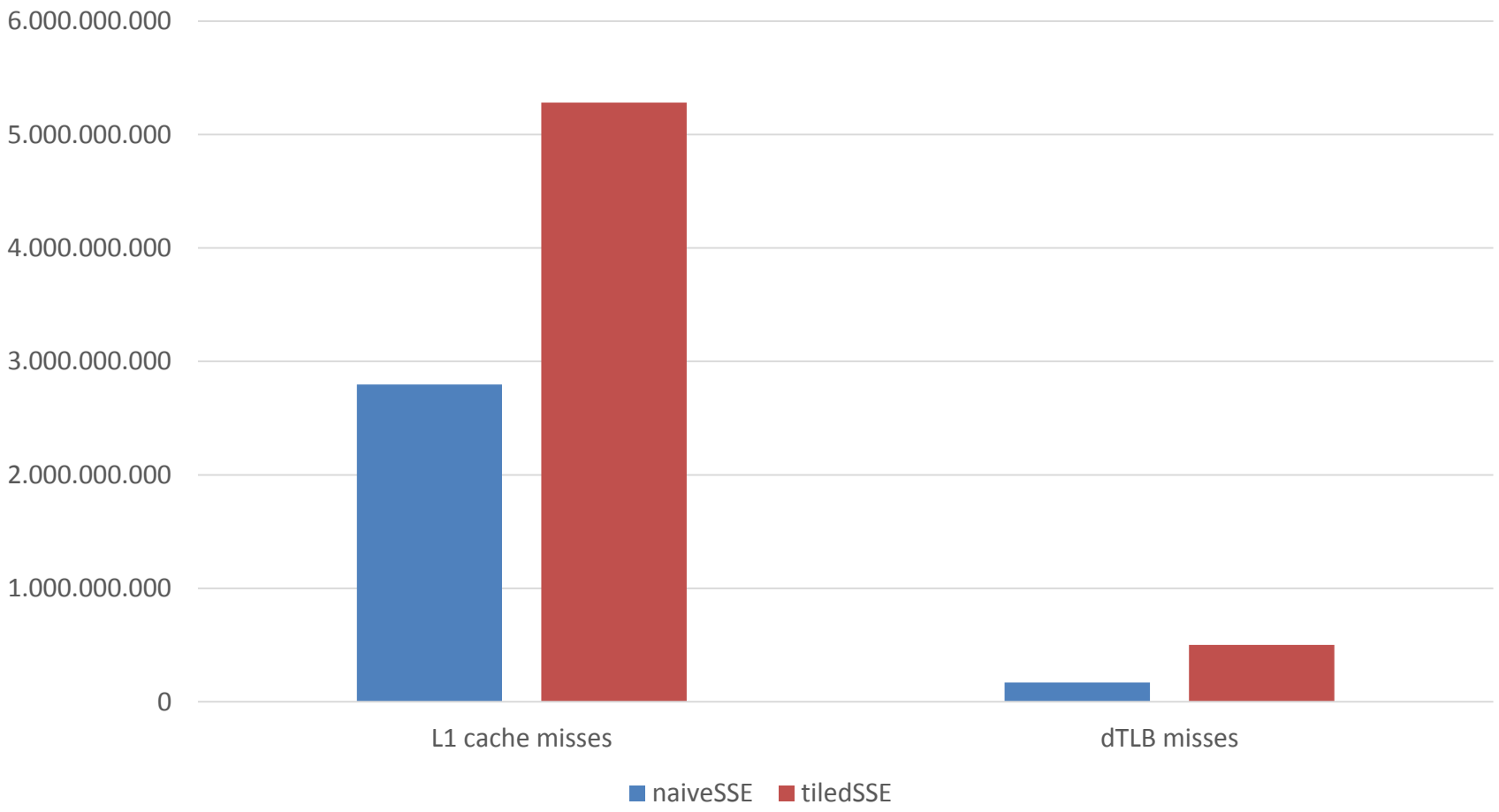
SSE – Single Threaded



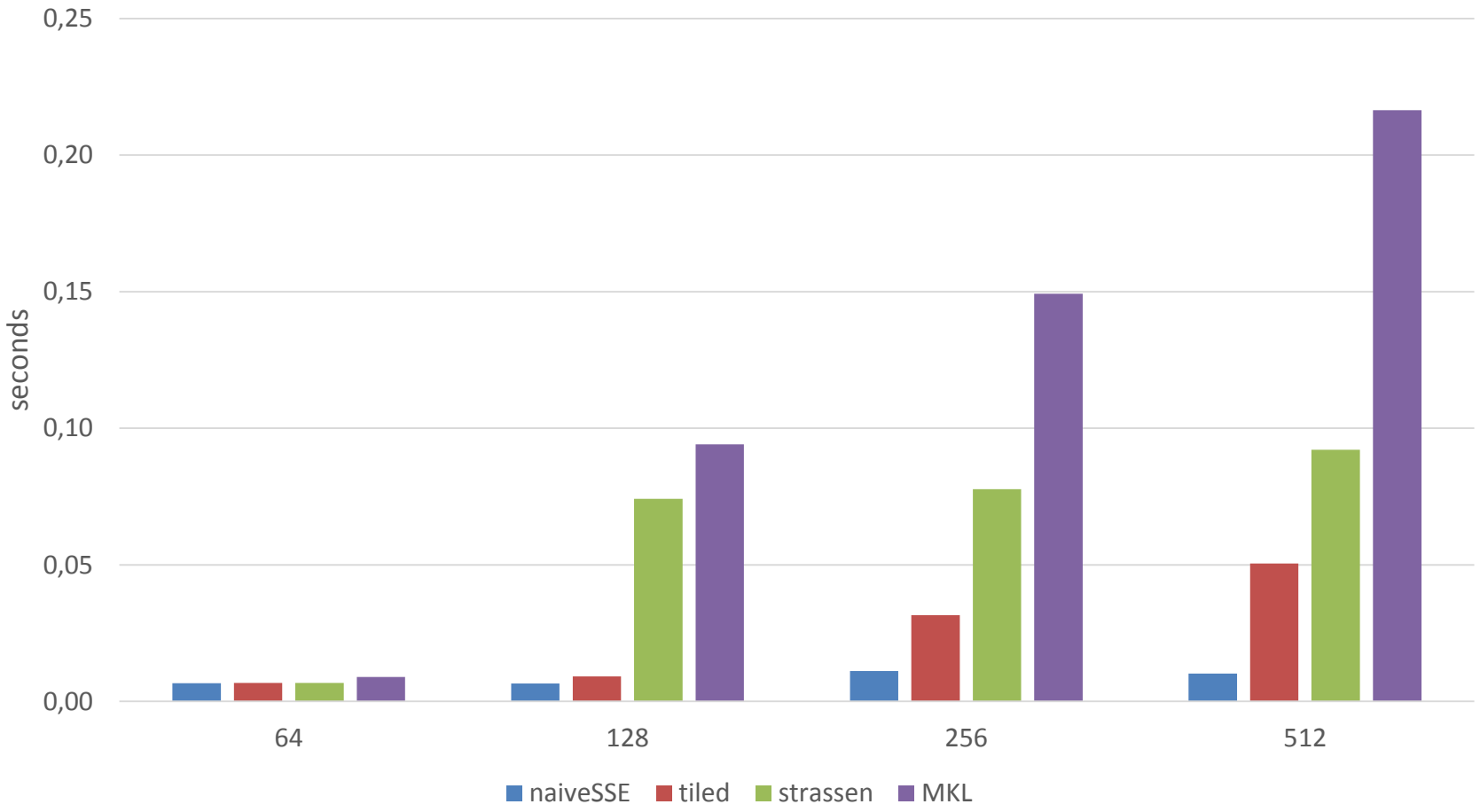
	1024	2048	4096
naiveSSE	0,27	2	20
tiledSSE	0,48	5	41
tiled	2	24	213
naive	11	97	879

dl980 on 1 core

Cache Misses of SSE Variants

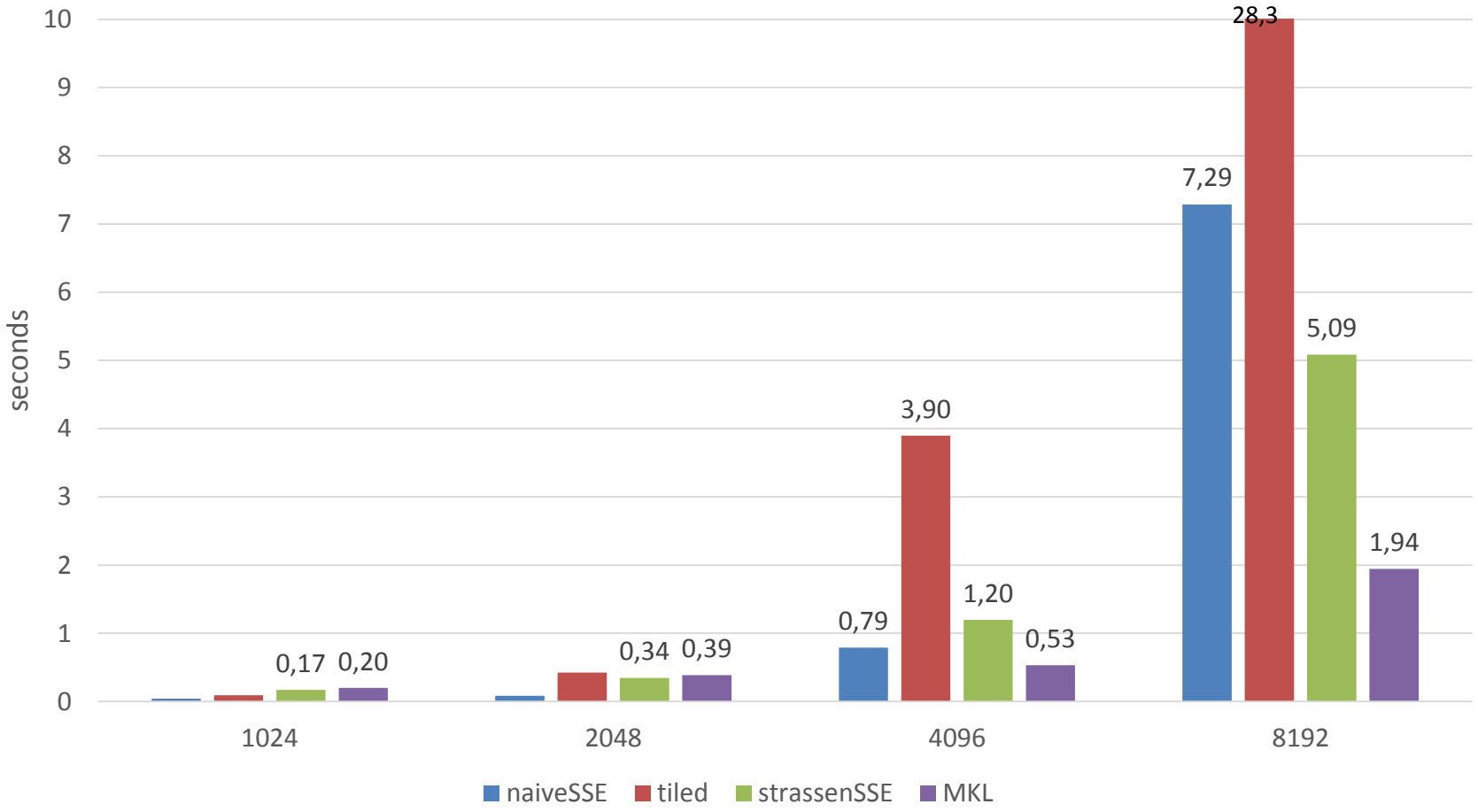


Performance for Small Matrices



dl980 on 128 cores

Performance for Large Matrices



dl980 on 128 cores

Summary

- Analyze algorithm for bottlenecks
  - IO optimization
  - Hardware specific optimization
    - Cache size
    - NUMA architecture
    - Specific instructions (SSE)
- Try to minimize remote memory access
- Visualisations can facilitate understanding