Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Microbenchmarking for architectural exploration

Probing of the memory hierarchy

Saturation effects

OpenMP barrier overhead

Motivation for Microbenchmarking as a tool

- Isolate small kernels to:
  - Separate influences
  - Determine specific machine capabilities (light speed)
  - Gain experience about software/hardware interaction
  - Determine programming model overhead
  - …
- Possibilities:
  - Readymade benchmark collections (epcc OpenMP, IMB)
  - STREAM benchmark for memory bandwidth
  - Implement own benchmarks (difficult and error prone)
  - **`likwid-bench`** tool: Offers collection of benchmarks and framework for rapid development of assembly code kernels

The parallel vector triad benchmark
A "swiss army knife" for microbenchmarking

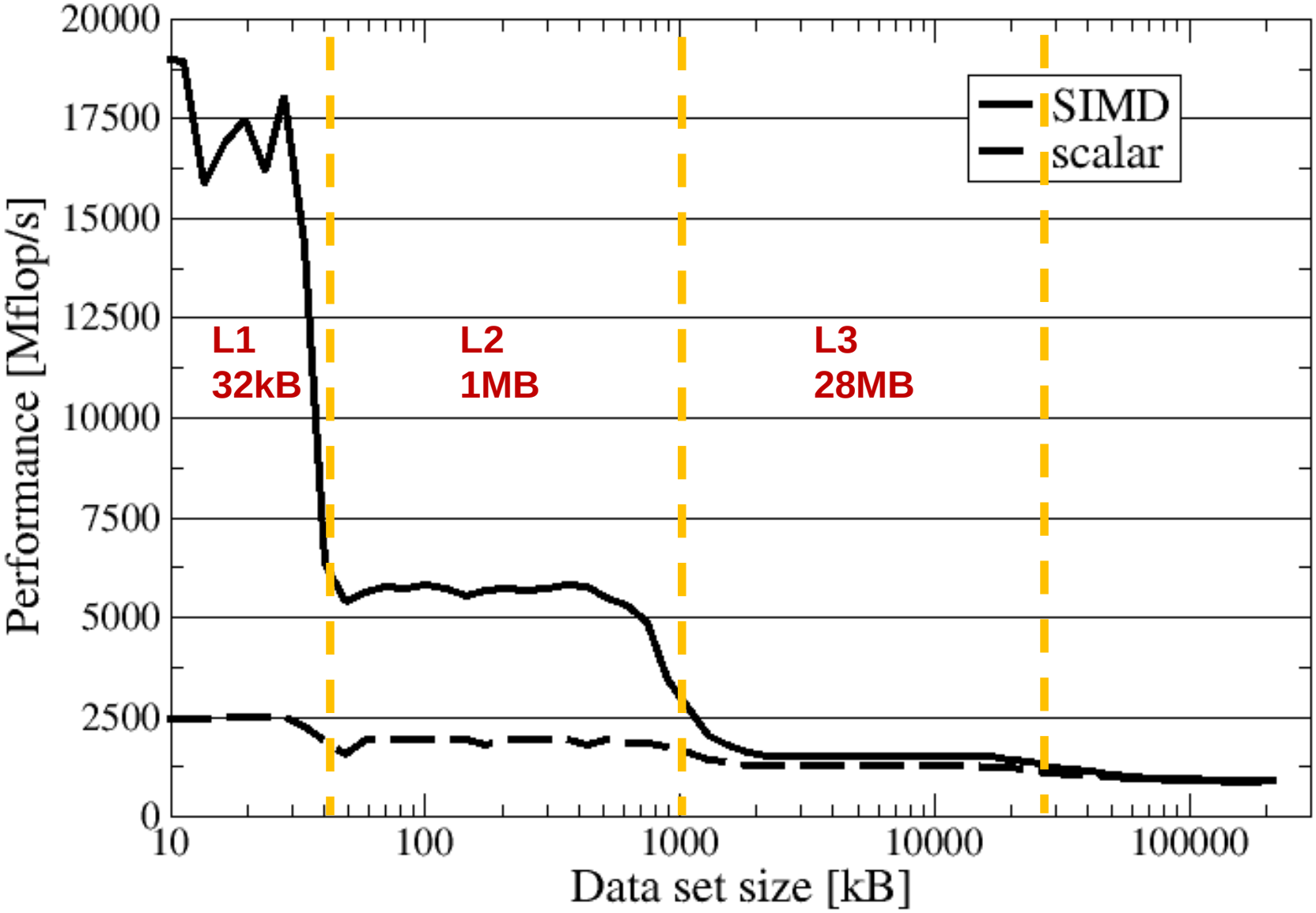
```
double striad_seq(double* restrict a, double* restrict b,
double* restrict c, double* restrict d, int N, int iter) {
    double S, E;
    S = getTimeStamp();
    for(int j = 0; j < iter; j++) {
#pragma vector aligned
        for (int i = 0; i < N; i++) {
            a[i] = b[i] + d[i] * c[i];
        }
        if (a[N-1] > 2000) printf("Ai = %f\n",a[N-1]);
    }
    E = getTimeStamp();
    return E-S; }
```

Required to get optimal code with Intel compiler!

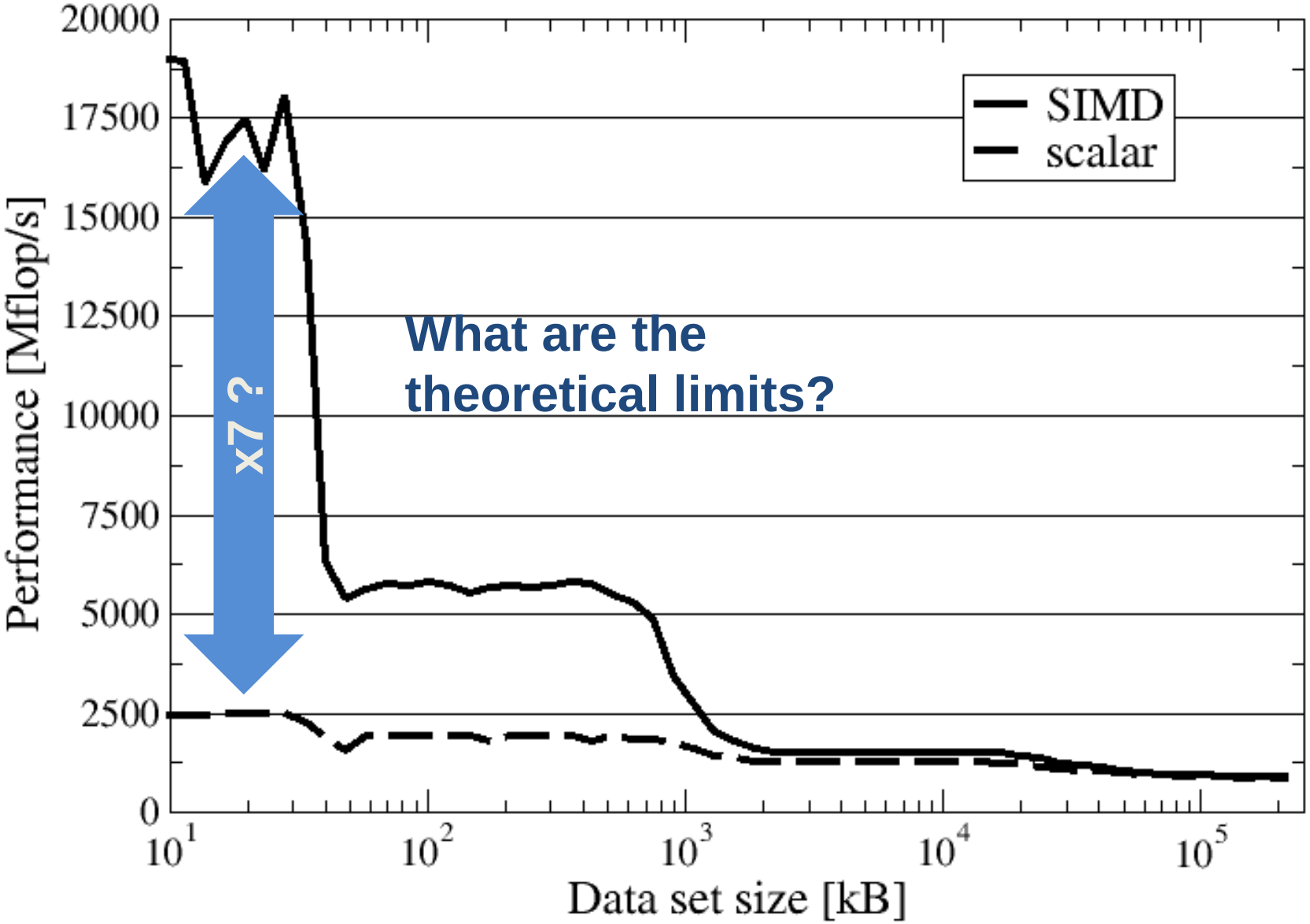
Prevents smarty-pants compilers from doing "clever" stuff

- Report performance for different **N**, choose **iter** so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all architectures, ever!

Schönauer triad on **one** CascadeLake **core** 2.5GHz



Schönauer triad on **one** CascadeLake **core** 2.5GHz



The throughput-parallel vector triad benchmark

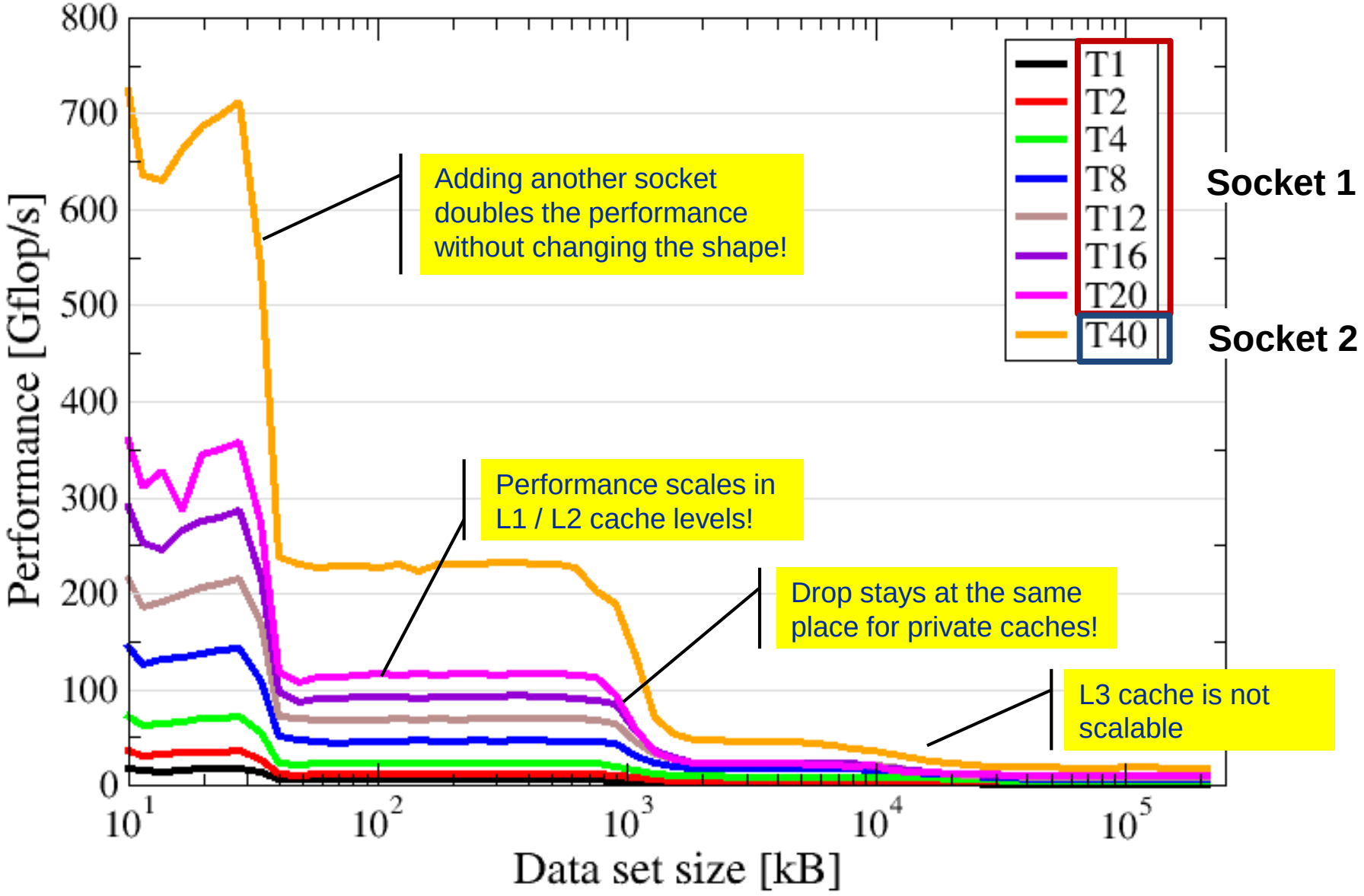
Every core runs its own, independent triad benchmark

→ pure hardware probing, no impact from OpenMP overhead

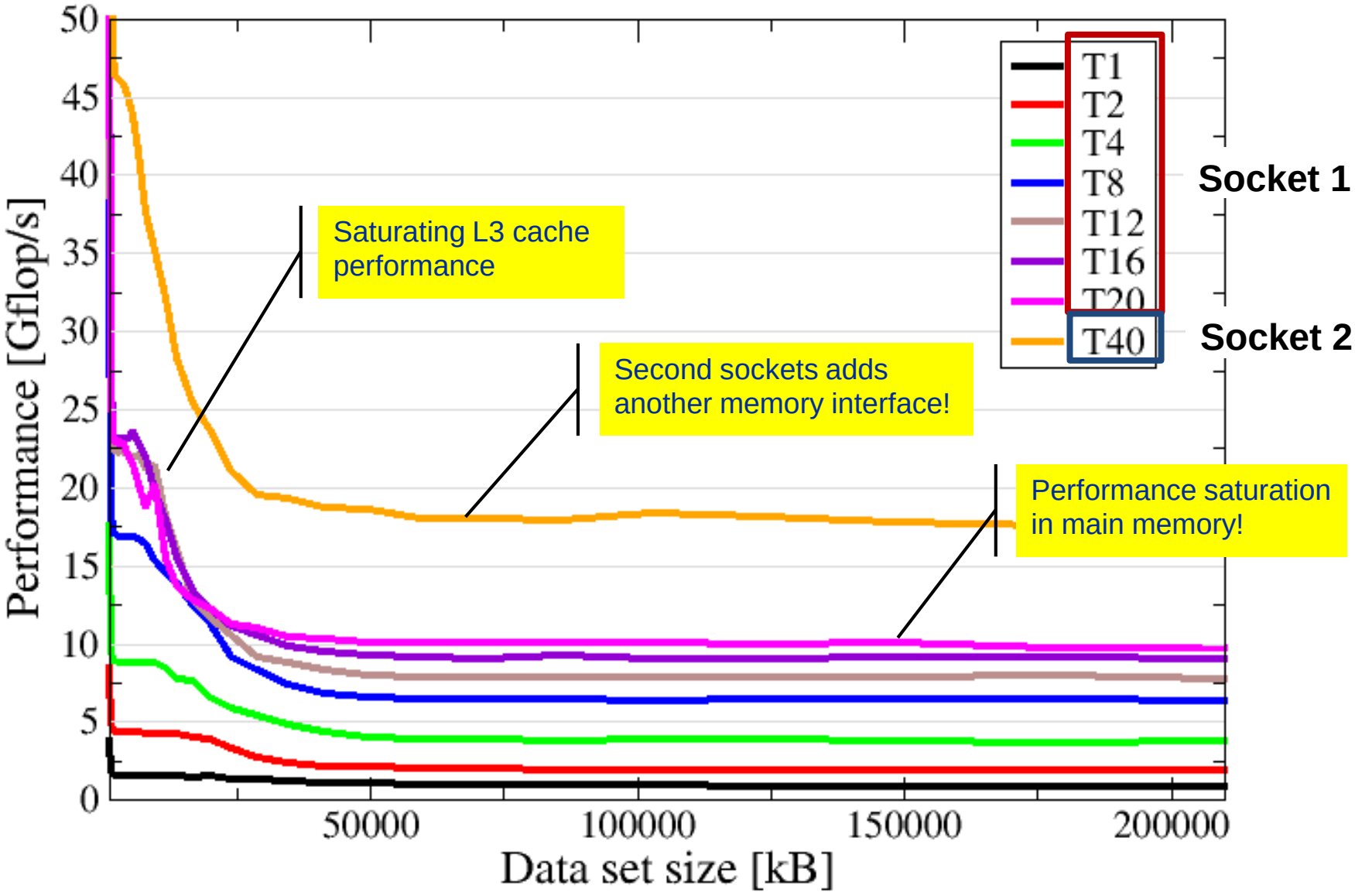
```
#pragma omp parallel
  {
      double* al;
      posix_memalign((void**) &al, ARRAY_ALIGNMENT, N * sizeof(double));
#pragma omp single
      S = getTimeStamp();
      for(int j=0; j<iter; j++) {
#pragma vector aligned
          for (int i=0; i<N; i++) {
              al[i] = b[i] + d[i] * c[i];
          }
          if (al[N-1] > 2000) printf("Ai = %f\n",al[N-1]);
      }
#pragma omp single
      E = getTimeStamp();
  }
```

Every thread works on private copy of a!

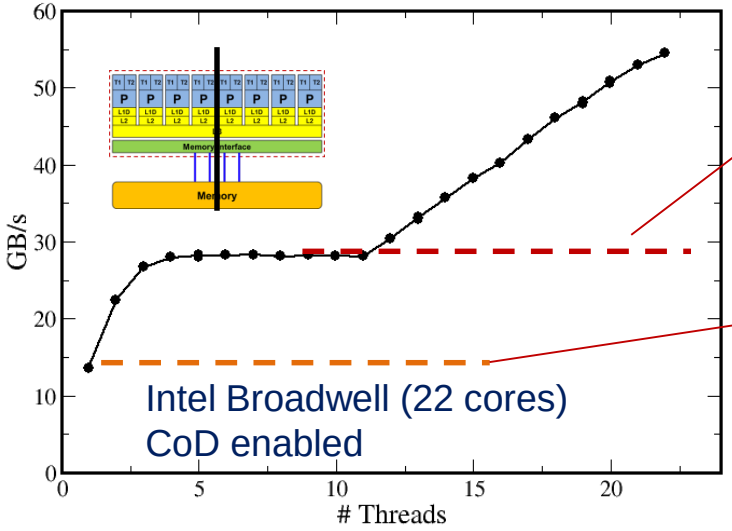
Throughput vector triad on one CascadeLake node (2.5 GHz)



Throughput vector triad on CascadeLake (memory close-up)

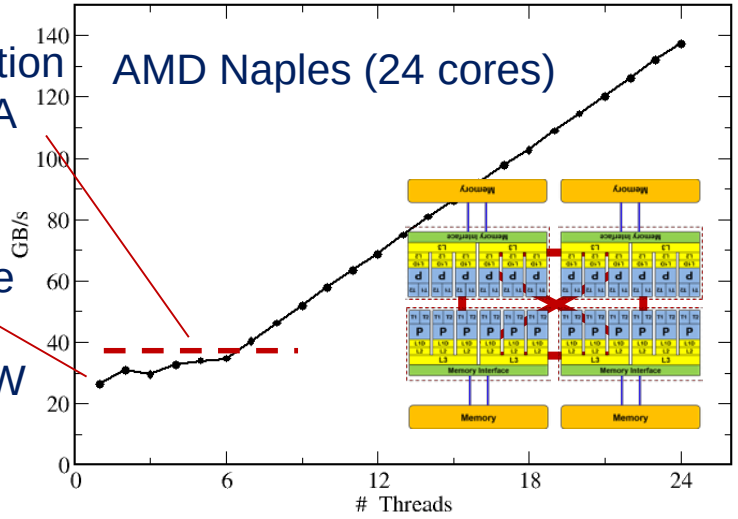


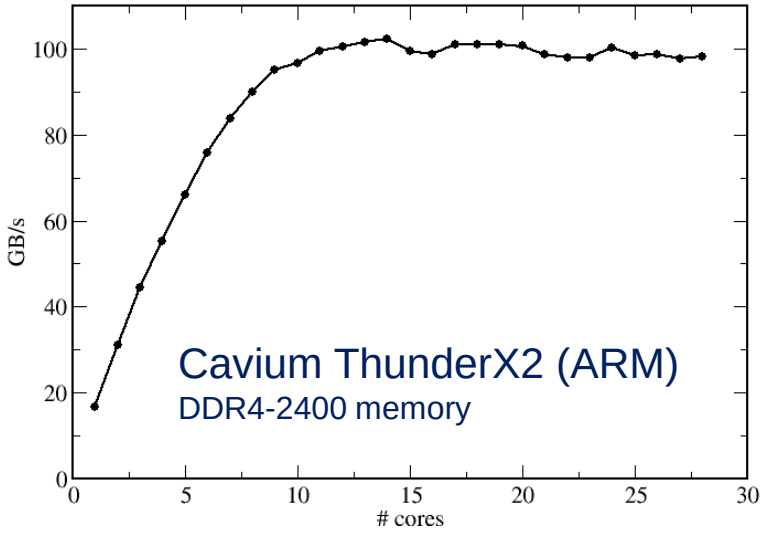
Attainable memory bandwidth (UPDATE!)

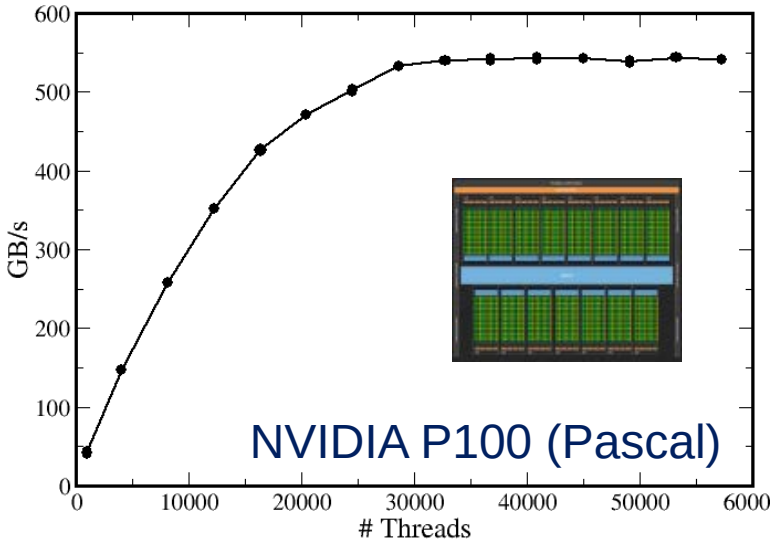


BW saturation in NUMA domain

Single core does not saturate BW





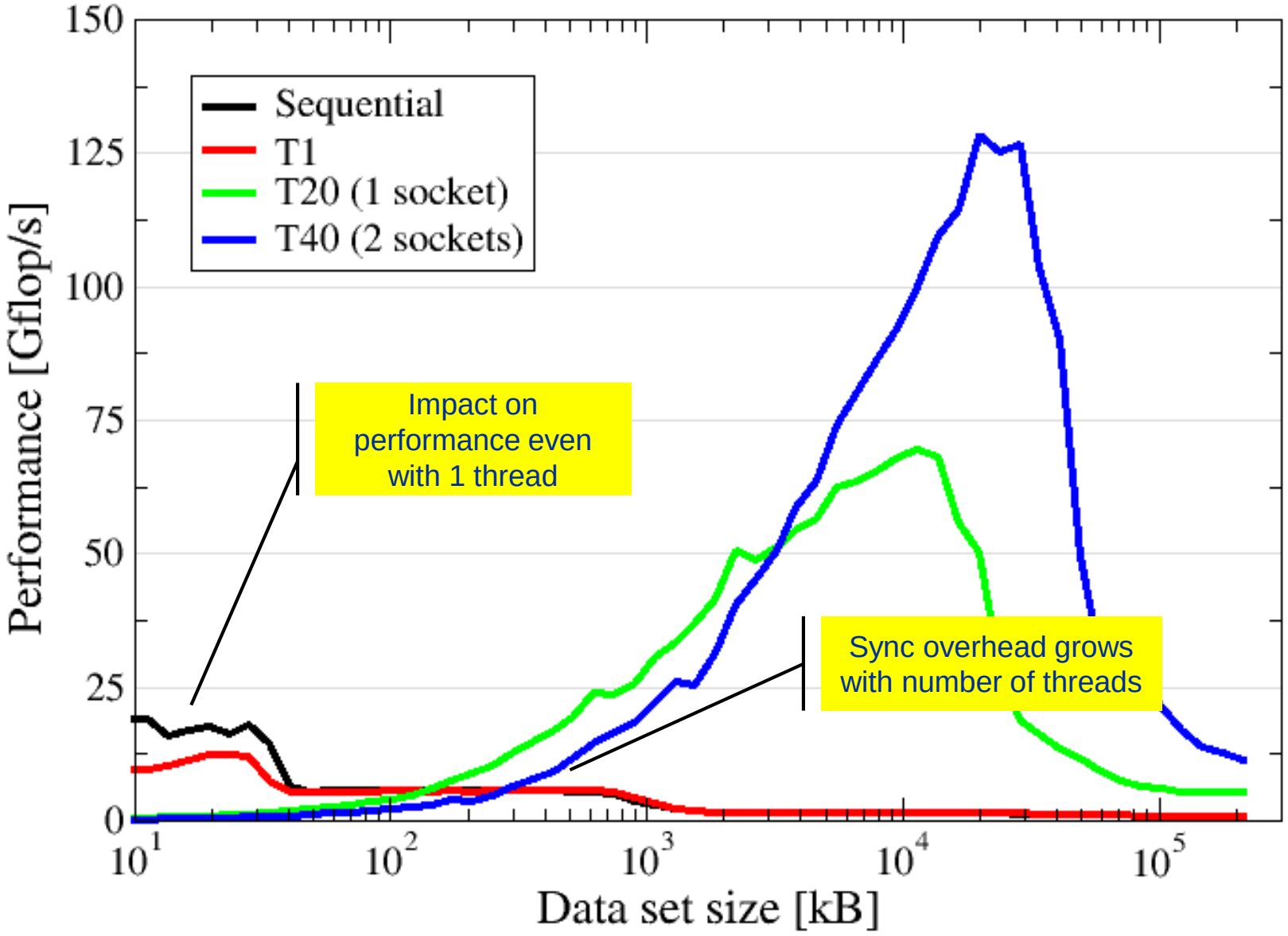


OpenMP **work sharing** in the benchmark loop

```
    S = getTimeStamp();
#pragma omp parallel
    {
        for(int j = 0; j < iter; j++) {
#pragma omp for
#pragma vector aligned
            for (int i=0; i<N; i++) {
                a[i] = b[i] + d[i] * c[i];
            }
            if (a[N-1] > 2000) printf("Ai = %f\n",a[N-1]);
        }
    }
    E = getTimeStamp();
```

Implicit barrier

OpenMP vector triad on CascadeLake node (2.2 GHz)



OpenMP performance issues on multicore

Synchronization (barrier) overhead

Welcome to the multi-/many-core era

Synchronization of threads may be expensive!

```
!$OMP PARALLEL …
…
!$OMP BARRIER
!$OMP DO
…
!$OMP ENDDO
!$OMP END PARALLEL
```

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP programs.

Determine costs via simple benchmark

On x86 systems there is no hardware support for synchronization!

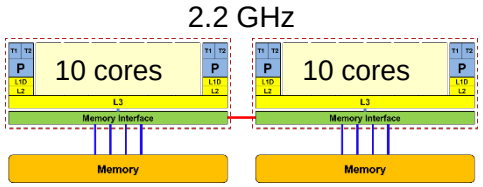
- Next slide: Test OpenMP Barrier performance…
- for different compilers
- and different topologies:
  - shared cache
  - shared socket
  - between sockets
- and different thread counts
  - 2 threads
  - full domain (chip, socket, node)

Thread synchronization overhead on IvyBridge-EP

Barrier overhead in CPU cycles

2 Threads	Intel 16.0	GCC 5.3.0
Shared L3	**599**	**425**
SMT threads	**612**	**423**
Other socket	**1486**	**1067**

Strong topology dependence!



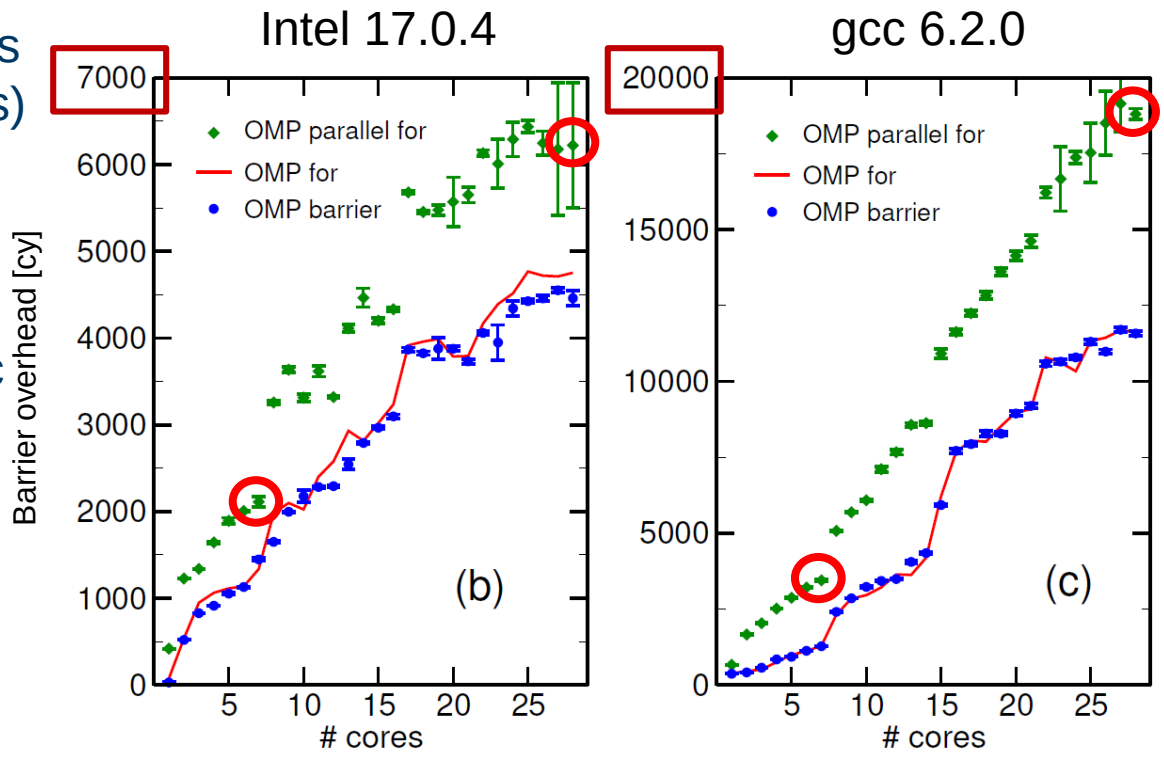
Full domain	Intel 16.0	GCC 5.3.0
Socket (10 cores)	1934	1301
Node (20 cores)	**4999**	**7783**
Node +SMT	5981	9897

Overhead grows with thread count

- Strong dependence on compiler, CPU and system environment!
- **OMP_WAIT_POLICY=ACTIVE** can make a big difference

Comparison of barrier synchronization cost with increasing number of threads

1. 2x Haswell 14-core CoD mode
2. Optimistic measurements (repeated 1000s of times)
3. No impact from previous activity in cache
4. Ideal scaling: logarithmic



Conclusions from the microbenchmarks

- **Microbenchmarks** can yield **surprisingly deep insights**

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - **Know where your threads are running**
    - **Know where your data is** (see later for that)

- **Bandwidth bottlenecks are ubiquitous**
- **Synchronization overhead** may be an issue
  - … and also depends on affinity!
  - Many-core poses new challenges in terms of synchronization