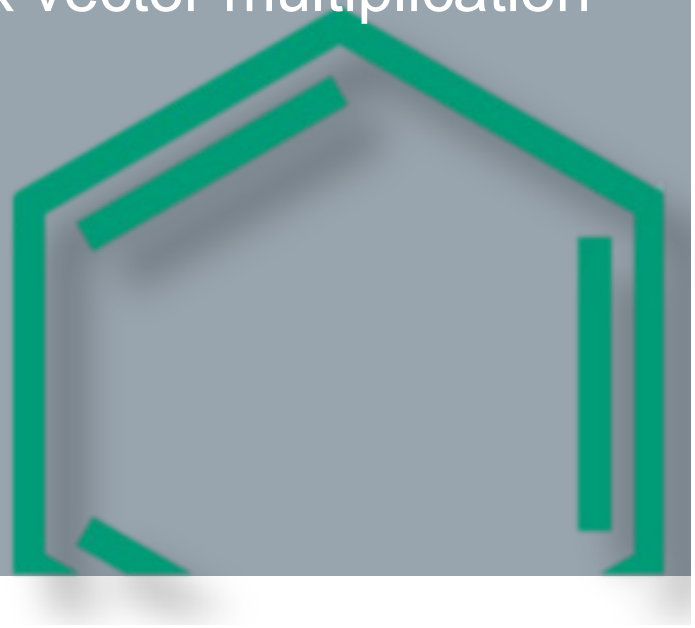
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Hardware Performance Monitoring

Basics and Case-Study for upper triangular matrix vector multiplication

Thomas Gruber, NHR@FAU



Questions?

- If you see a code, can you imagine how the hardware behaves during execution?
- Are there hidden operations that can harm performance?
- Where can we get real performance values from?
- Does my code hit a bottleneck?

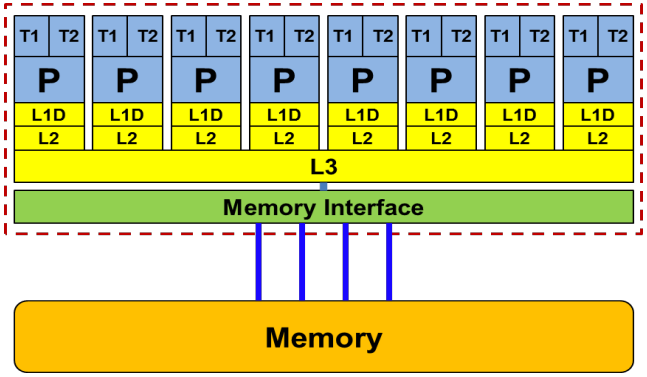
Agenda

- Hardware performance monitoring (HPM)
  - What is it?
  - Why is it used?
  - How does it work?
  - What can be measured?
- HPM Tools
- LIKWID

- Use-case analysis using HPM
  - Dense Triangular Matrix-Vector-Multiplication
  - Dense Quadratic Matrix-Vector-Multiplication

Hardware performance monitoring (HPM) - *What is it?*

- Measure the behavior of hardware running software
- Count filtered hardware operations
- Originally by CPU vendors for validation



Instructions stream:
Load X to Reg1

Event stream:
Issue load operation
L1 lookup
L1 miss
L2 lookup
L2 miss
L3 lookup
L3 miss
MEM lookup
MEM hit
MEM read X
Transfer X to L3
L3 Evict (maybe)
Transfer X to L2
L2 Evict (maybe)
Transfer X to L1
L1 Evict (maybe)
Transfer X to Reg1

Counter0: L1 misses
1

Counter1: L2 hits
0

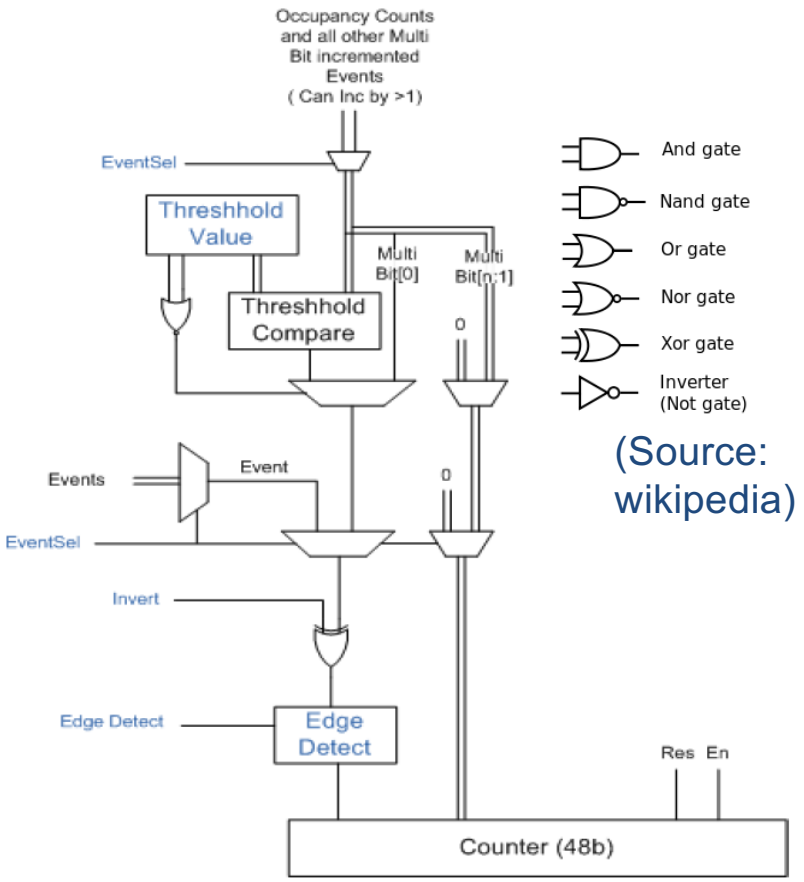
Counter2: Load instr.
1

Counter3: Any miss
3

Hardware performance monitoring (HPM) - *What is it?*

- First introduction in DEC Alpha 21064 (1992)
- In X86 systems with the Pentium (1994)
- Almost all architectures released after 1994

- Hardware logic with filtering
- No extra work done by processing unit



(Source: wikipedia)

(Source: Intel.com)

Differentiation: hardware and application performance monitoring

Application Monitoring:

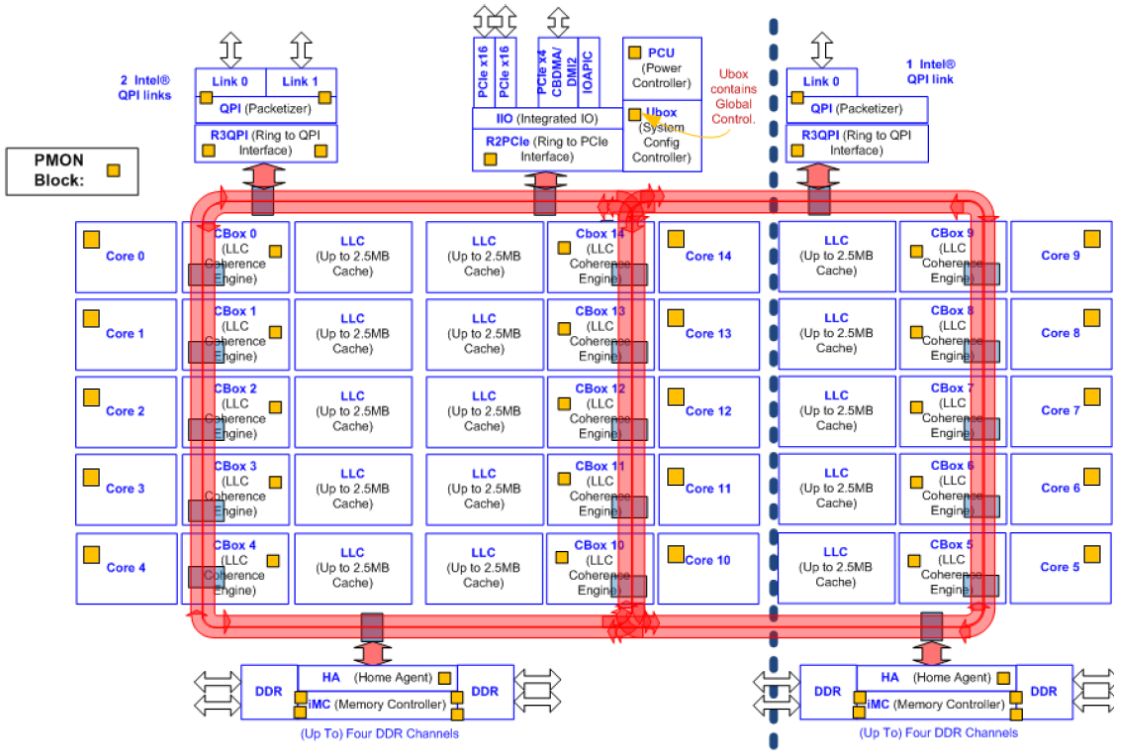
- Application insight
- Fine grained
- In-application measurements defer work → Overhead
- May create huge amount of data

Hardware monitoring:

- Only hardware insight, relation to executed code almost impossible
- Coarse measurements (fine grained only with instrumentation)
- (Almost) no overhead
- Limited amount of measurement facilities limit data generation

Hardware performance monitoring (HPM) - *What can be measured?*

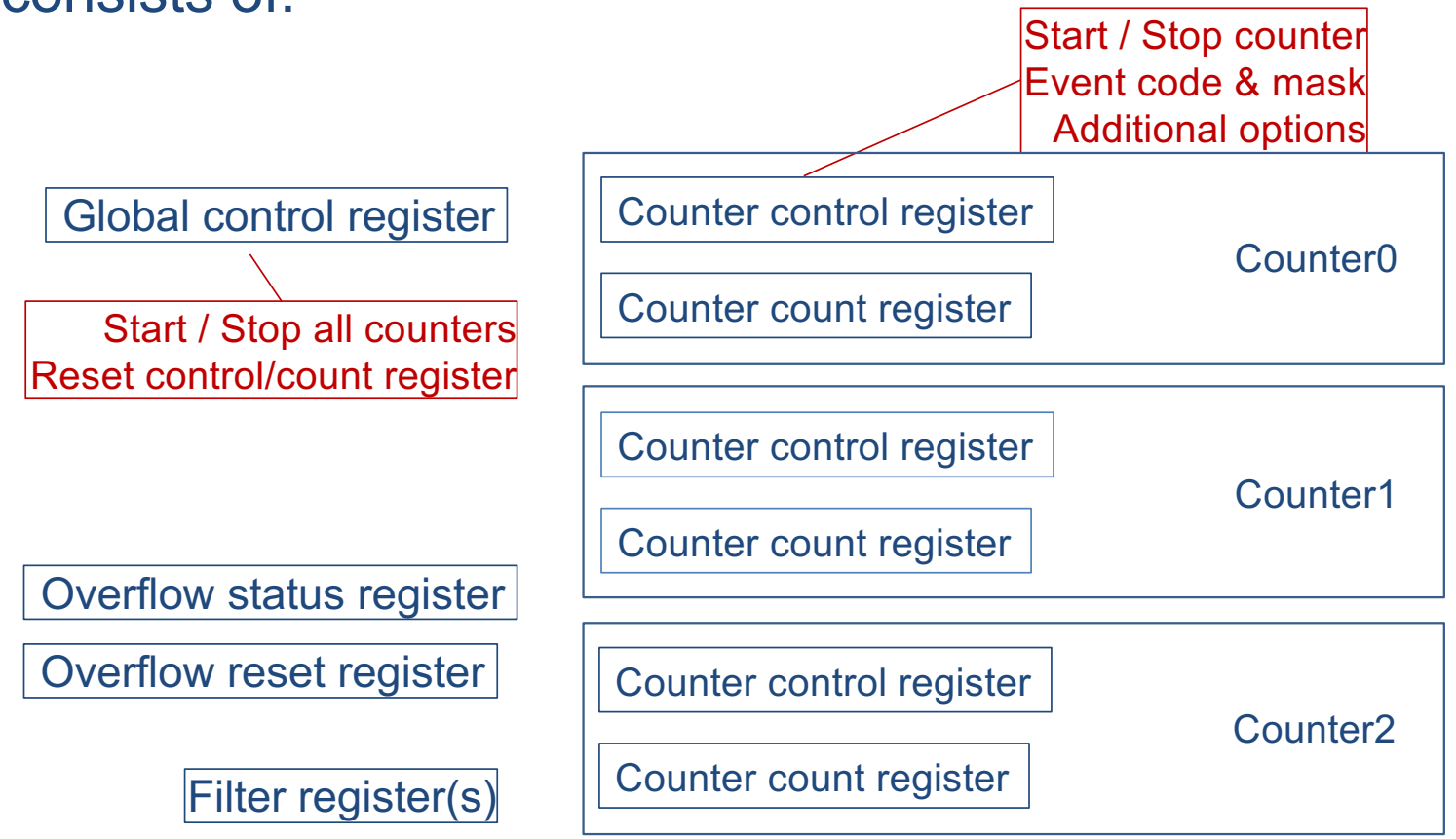
HPM units are scattered over the complete chip



Intel IvyBridge EP
(Source: Intel.com)

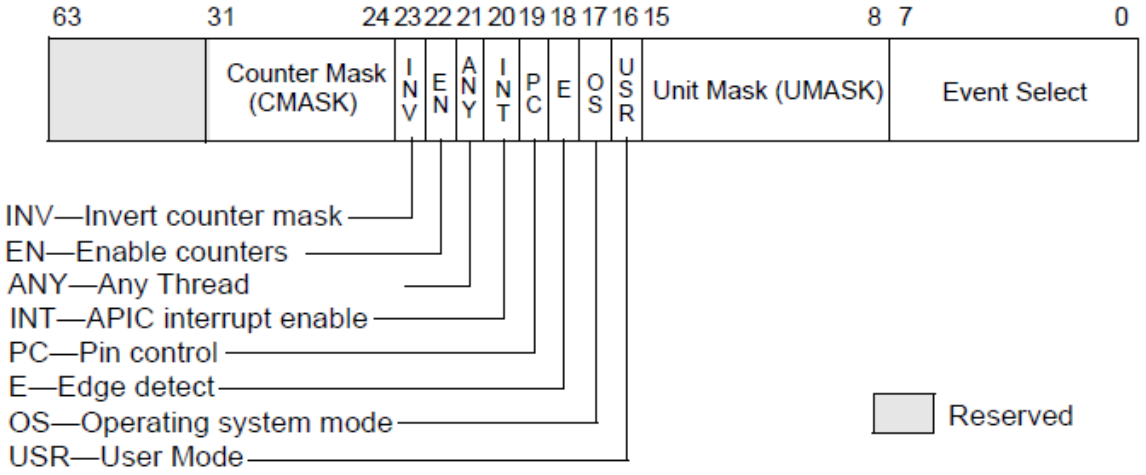
Hardware performance monitoring (HPM) - *How does it work?*

PMU consists of:



Hardware performance monitoring (HPM) - *How does it work?*

Bitmap of a control register (Arch version 3)

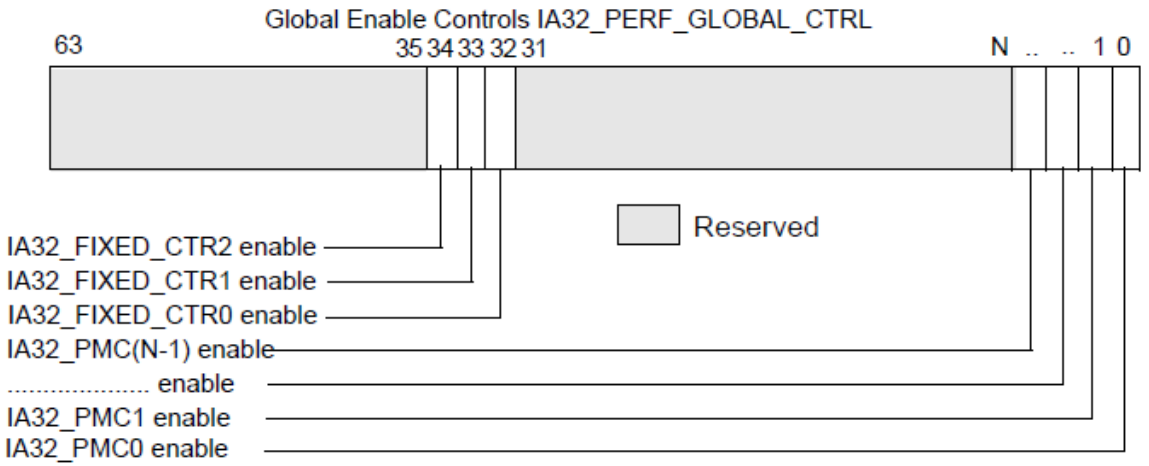


- APIC interrupts when counter overflows (unusable in user-space)
- Pin control for overflow handling: global register or via APIC interrupt

(Source: Intel.com)

Hardware performance monitoring (HPM) - *How does it work?*

Bitmap of a unit control register:



- Global overflow registers (signal & reset) use same bit offsets

Hardware performance monitoring (HPM) - *How does it work?*

- Counter setup:
  - Stop counting globally
  - Stop whole PMU
  - (Stop counter register)
  - Write event+opts to config register
  - Reset counter register
  - Start whole PMU
  - Start counting globally
- Read counter value:
  - (Stop globally, whole PMU or only specific counter)
  - Read counter value
  - (Start globally, whole PMU or only specific counter)

Hardware performance monitoring (HPM) - *What can be measured?*

- Intel Pentium provided 42 events, current Icelake SP offers 3032 events
- Events for:
  - Cache/memory traffic (loads, stores, RFO, …)
  - Special instructions (divide, floating-point ops, …)
  - CISC -> RISC -> CISC translation
  - Branch prediction, resource stalls, …
  - Energy consumption, Temperature (not really HPM)

- Understand an algorithm‘s performance
- Find bottlenecks, real data volume (Loads + Stores + RFOs)
- Hints for tuning the algorithm

HPM Tools

- HPM on X86 accessible through MSRs and PCI devices
  - Different on each architecture
- Low level tools:
  - msr-tools
  - perf_event (>= Linux 2.6.35)
- Medium level tools (on top of perf_event):
  - PAPI
  - Intel VTune
- High level tools (on top of PAPI):
  - Score-P (Collaboration of german universities)
  - TAU (University Oregon), HPCToolkit (Rice University), OpenSpeedshop, …

LIKWID

LIKWID - *Overview*

- Project started in 2009
- Developed by HPC group at the FAU (me )
- Tool suite for performance analysis:
  - System topology information
  - Process/Thread pinning
  - Hardware performance monitoring & energy measurements
  - Microbenchmarking in assembly
- LIKWID tries to measure anything running on a CPU in user-space!
- LIKWID does not require any special kernel module, only default stuff

LIKWID - Hardware performance monitoring

- **Problem:** Access to counters often requires root privileges
  - Some features only in kernel-space
- **Problem:** Own set of counters and events per architecture
- **Problem:** Some events have different scope
  - AMD counts floating-point operations, Intel instructions
  - Uncore and RAPL counter at socket-scope
    Exception memory controller for AMD Zen1 at NUMA node level
  - Energy measurements (RAPL) per socket
    - PLATFORM domain for full node but mainboard vendors have to wire it
    - CORE domain for AMD Zen* at CPU core granularity
- Many more problems

LIKWID – How to overcome the problems

- Different access modes:
  - Direct access for root users (or trustworthy users with suid-root)
  - Daemon access for users
    - Proxy application with suid-root
    - Intercepts all access calls (read, write, exist)
    - Check to forbid access to non-HPM registers
  - Perf_event mode uses kernel interface (partly per processes)
- Architecture-specific events and counters
  - Select events and counter map based on CPUID (vendor, model ID, variant, …)
  - Select setup, start, stop and read functions based on CPUID

LIKWID – How to overcome the problems

- Topology gathering creates affinity domains
  - Tile domain (All hardware threads attached to LLC segment)
  - Socket domain (All hardware threads of a socket)
  - NUMA domain (All hardware threads of a NUMA domain)
  - Node domain (All hardware threads)
- At CPU selection choose one of each domain to be leader

- Why affinity domains? Used for pinning threads!
  - Physical without domain: 0,1,2,3 or 4-8
  - Logical in domain: S0:0-3 or M0:0,4,6,2
  - Expression inside domain: E:N:4:1:2 -> 0,1,2,3,4,5,6,7 = 0,2,4,6

  Check **`likwid-pin`** for more information about the syntax

likwid-perfctr

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz […]
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+----------------------+---------+------------+------------+------------+------------+
|         Event        | Counter |   Core 14  |   Core 15  |   Core 16  |   Core 17  |
+----------------------+---------+------------+------------+------------+------------+
|   INSTR_RETIRED_ANY  |  FIXC0  | 1298031144 | 1965945005 | 1854182290 | 1862521357 |
| CPU_CLK_UNHALTED_CORE|  FIXC1  | 2353698512 | 2894134935 | 2894645261 | 2895023739 |
|  CPU_CLK_UNHALTED_REF|  FIXC2  | 2057044629 | 2534405765 | 2535218217 | 2535560434 |
|    L1D_REPLACEMENT   |   PMC0  |  212900444 |  200544877 |  200389272 |  200387671 |
|    L2_TRANS_L1D_WB   |   PMC1  |  112464863 |  99931184  |  99982371  |  99976697  |
|     ICACHE_MISSES    |   PMC2  |    21265   |    26233   |    12646   |    12363   |
+----------------------+---------+------------+------------+------------+------------+
[… statistics output omitted …]
```

Always measured for Intel CPUs

Configured metrics (this group)

likwid-perfctr (cont.)

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhalted [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241

Preconfigured and extensible metric groups, list with **likwid-perfctr -a**

- **FLOPS_DP: Double Precision MFlops/s**
- **FLOPS_SP: Single Precision MFlops/s**
- **MEM: Main memory bandwidth in MBytes/s**
- **CLOCK: Clock frequency of cores**

MarkerAPI

- The marker API can restrict measurements to code regions
- The API only reads counters, the setup is done by **likwid-perfctr**
- Multiple named regions support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```
#include <likwid-marker.h>
. . .
LIKWID_MARKER_INIT;                 // must be called from serial region

. . .
LIKWID_MARKER_START("Compute");     // call markers for each thread
. . .
LIKWID_MARKER_STOP("Compute");
. . .
LIKWID_MARKER_START("Postprocess");
. . .
LIKWID_MARKER_STOP("Postprocess");
. . .


LIKWID_MARKER_CLOSE;                // must be called from serial region
```

Before LIKWID 5 use **likwid.h**

- Activate macros with **-DLIKWID_PERFMON**
- Run **likwid-perfctr** with **-m** switch to enable marking
- See **https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90** for Fortran example

NHR FAU

Friedrich-Alexander-Universität Erlangen-Nürnberg

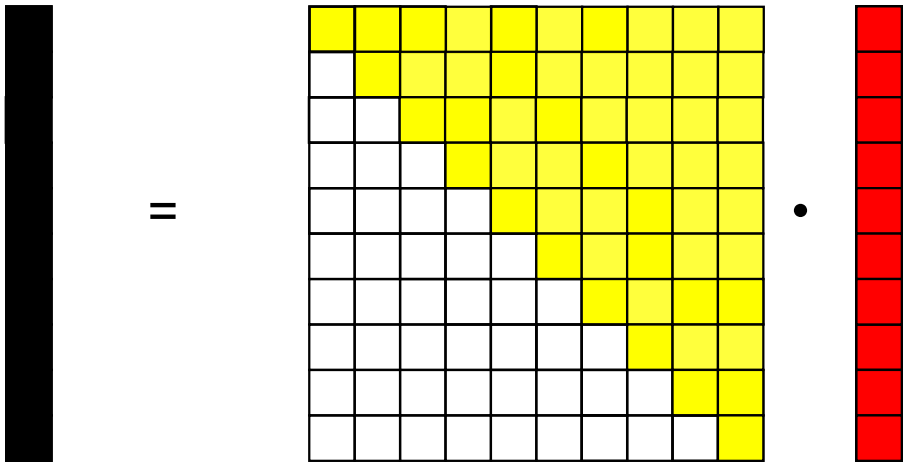
Case-Study for upper triangular matrix vector multiplication

High Performance Computing

Case-Study dTMVM

- Triangular matrices often used: LU factorization -> Gauss elemination, compute inverse matrix or dominant of a matrix
- Dimension N:
  - $\boldsymbol{N \cdot {}^{N+1}/_{2}}$ **elements in matrix**
  - $\boldsymbol{N \cdot {}^{N+1}/_{2}}$ **multiplications + $\boldsymbol{N \cdot {}^{N+1}/_{2}}$ additions**

  → **Computational intensity = 2 FLOPs / 8 Bytes (counting only loads)**



Case-Study dTMVM (Code + Instrumentation)

```
#include <likwid-marker.h>
[…] // defines, fillMatrix and init matrix & vectors
LIKWID_MARKER_INIT;
#pragma omp parallel { LIKWID_MARKER_REGISTER(„Compute“); }
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        LIKWID_MARKER_START(“Compute”);
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            double current = 0.0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        LIKWID_MARKER_STOP(“Compute”);
        while (cvec[N>>1] < 0) {dummy();break;} // to avoid removal of loop
    }
}
LIKWID_MARKER_CLOSE;
```

Include header file, initialize library and register each thread

Data often accessed, likely to be kept in cache

Read counters

Data more likely located in memory.
With N=10000 and ROUNDS=1000
$(10000+1)/2 \cdot 10000 \cdot 8\ \text{Bytes} \cdot 1000 \approx 400\ \text{GBytes}$

Write results to file for evaluation by likwid-perfctr

Case-Study dTMVM

Run in Socket 0 domain
using CPUs indexed 0 and 1
in ‚phyical cores first' sorted list

Marker API

Loads of matrix (Memory) and vector (L2)

Measure different metrics using LIKWID:

```
likwid-perfctr -C S0:0-1 -g <Group> -m ./dtmvm
```

Group	Metric	CPU 0	CP
All groups	Cycles-per-instruction	2.5	1.2
L2	L2 data volume [GBytes]	602	184
L3	L3 data volume [GBytes]	285	98
MEM	Memory data volume [GBytes]	412	-
DATA	Load to store ratio	3320.5 : 1	8.5 : 1
FLOPS_DP	MFLOP/s	4897	1704
L2CACHE	L2 request rate	0.75	0.13
L2CACHE	L2 miss ratio	0.30	0.30

Only loads of the matrix.
Streamed through L3 cache.

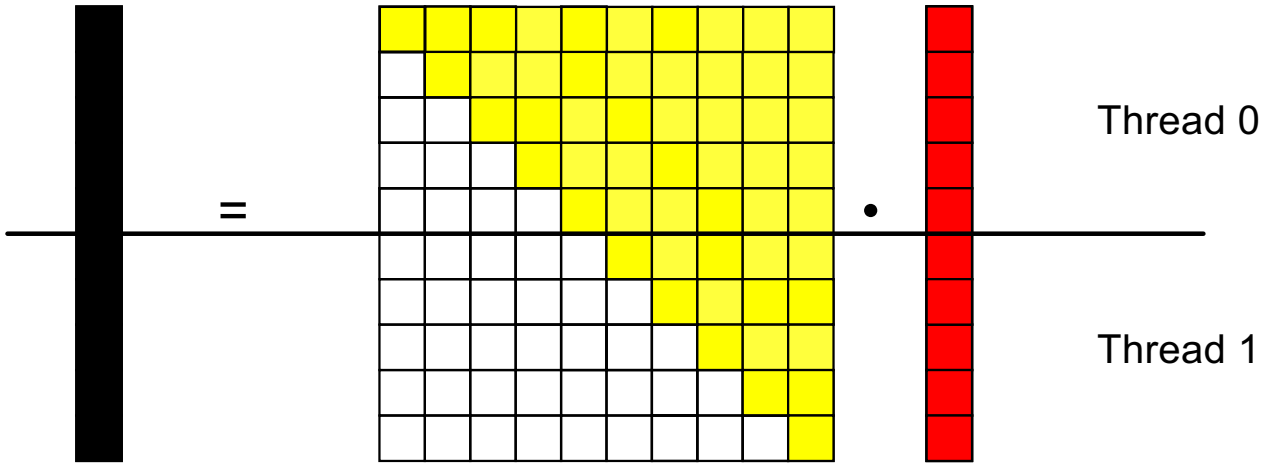
N=10000, ROUNDS=1000, Intel E5-2687W

Case-Study dTMVM

- Why is CPI lower for HWThread 1? Is it executing better on HWThread 1?
  OpenMP for pragma performs implicit barrier at end of loop
  → Busy waiting executes many ‚short' instructions
  → CPI decreases
- Why is the load to store ratio so different?
  HWThread 0 loads long lines ($\mathbf{N} \rightarrow {}^{N}/_{\mathbf{2}}$) for single store to rhs
  HWThread 1 loads only short lines (${}^{N}/_{\mathbf{2}} \rightarrow \mathbf{1}$) for single store to rhs
- L2 request rate of 75% for HWThread 0 and only 13% for HWThread 1?
  Careful about interpretations of metric names!
  Request rate = Requests to L2 / all finished instructions
  Value for HWThread 1 skewed by implicit barrier like CPI

Case-Study dTMVM

- Parallelization with OpenMP

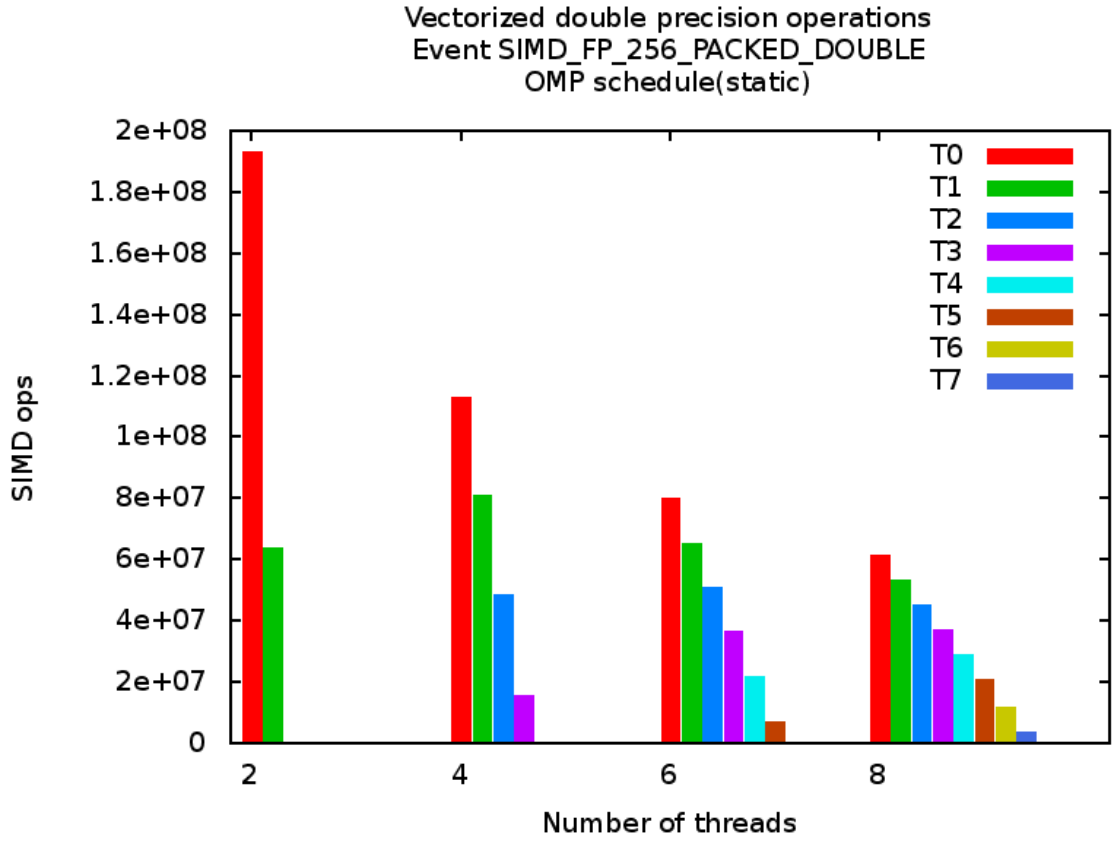


- Calculated work/load ratio Thread 0 : Thread 1 = 3 : 1
- L2 data volume ratio Thread 0 : Thread 1 = 3.3 : 1
- L3 data volume ratio Thread 0 : Thread 1 = 2.9 : 1
- MFLOP/s ratio Thread 0 : Thread 1 = 2.9 : 1

Data volume no good metric for work.

Case-Study dTMVM

Scaling run with SIMD FLOPs.

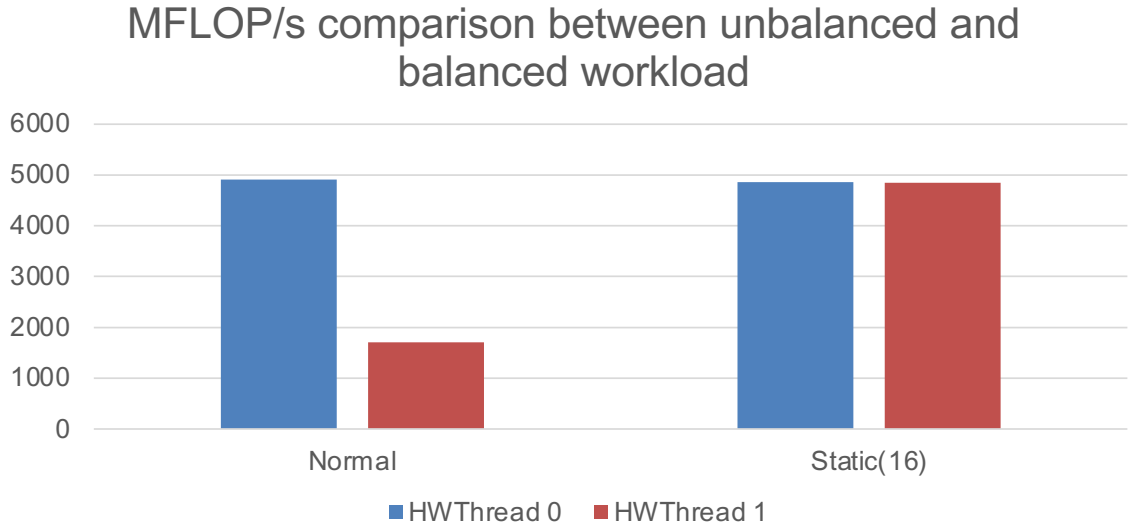


N=10000, ROUNDS=1000, Intel E5-2687W

Case-Study dTMVM

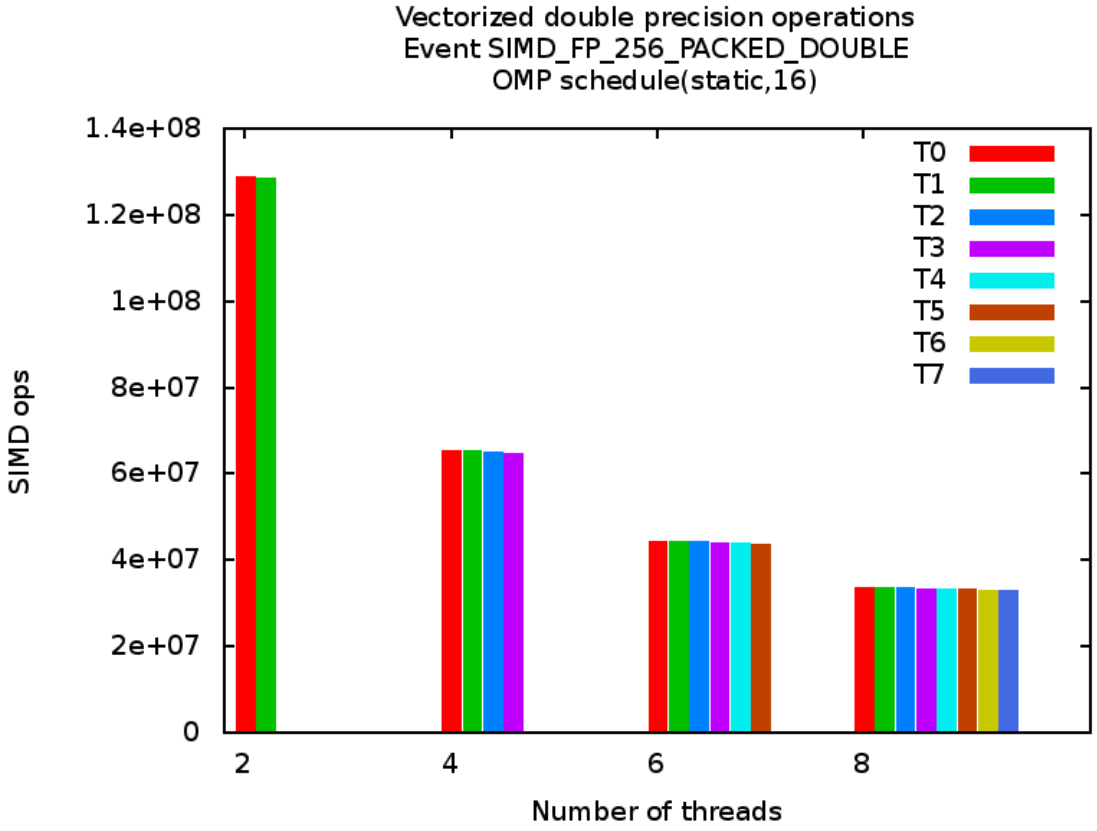
- How can we fix the load imbalance between the threads?
  - Use a manual decomposition of the matrix
  - Change OpenMP scheduler (commonly more overhead than static)
  - Default static scheduler splits iteration space by number of threads.

  Can be manually adjusted → schedule (static, 16) : 16 lines per thread



Case-Study dTMVM

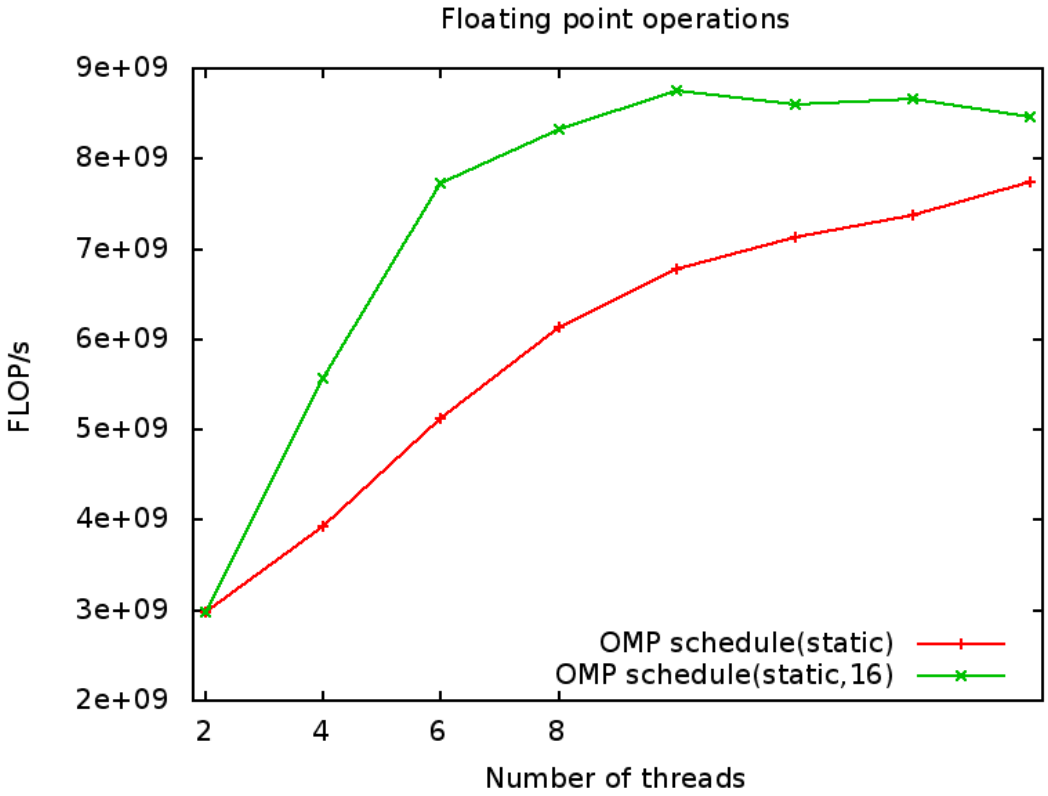
Scaling run with SIMD FLOPs compiled with schedule(static, 16)



N=10000, ROUNDS=1000, Intel E5-2687W

Case-Study dTMVM

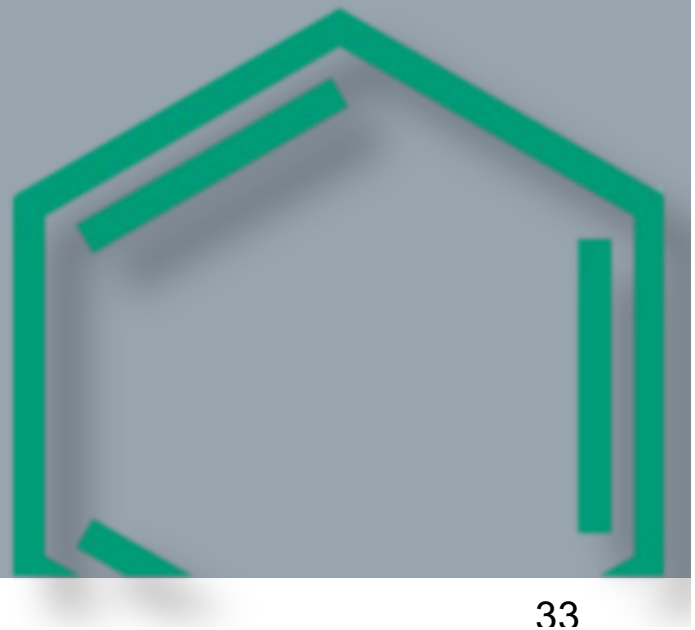
OK, we fixed the load imbalance, what about performance?



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case-Study for quadratic matrix vector multiplication



Case-Study dMVM

- Using a quadratic matrix
- Calculate on a multi-socket system

- Where does the data reside on multi-socket systems?
- Can the operating system distribute the data optimally?
- Can we initialize the data for optimal locality?
- How can we measure the inter-socket traffic?

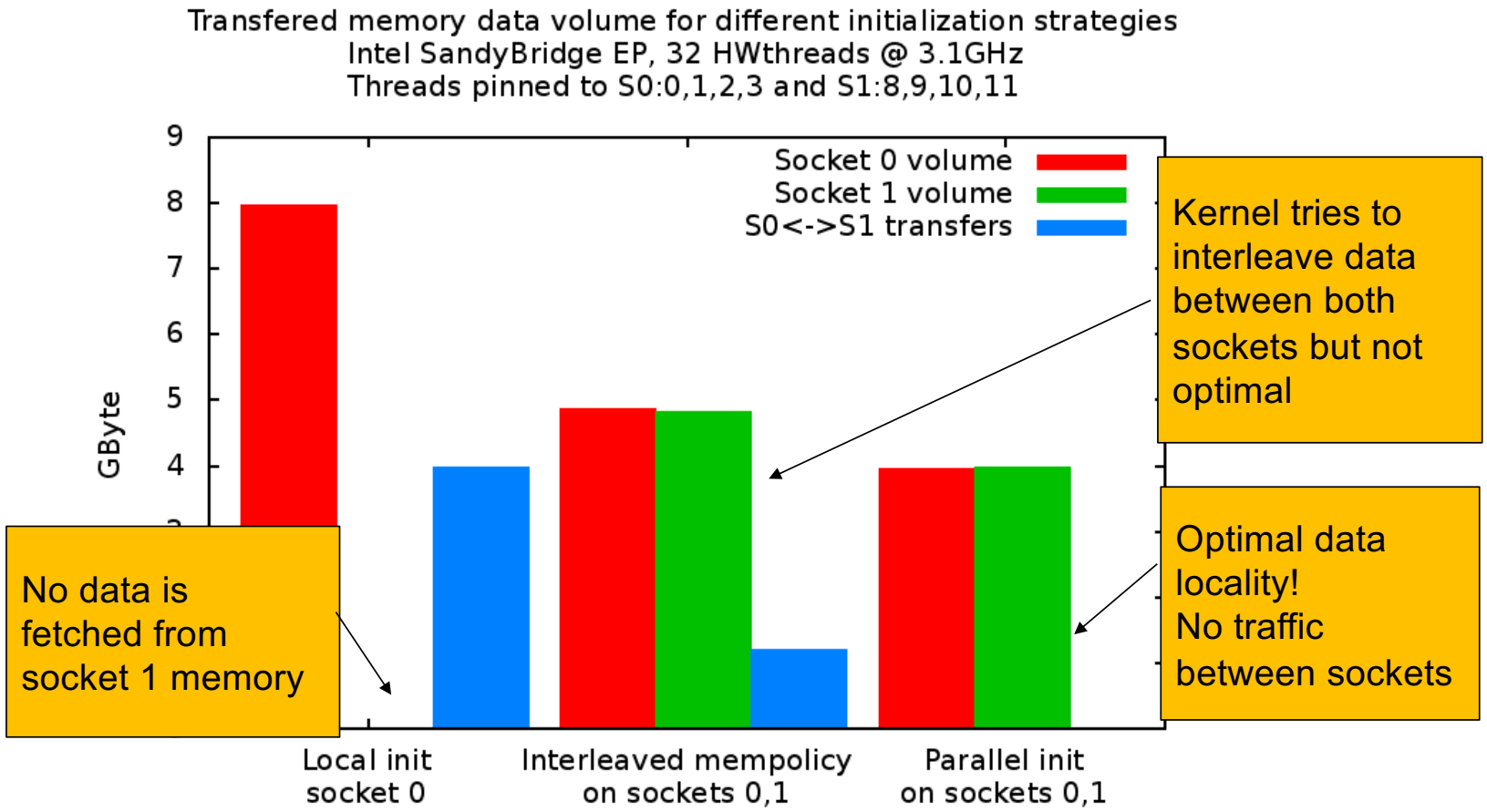
```
double* ptr = malloc(ARRAY_SIZE * sizeof(double))
if (!ptr)
    return;

for (int i=0; i< ARRAY_SIZE; i++)
    ptr[i] = M_PI;
```

No real allocation, just reservation

First touch allocation policy. The data is allocated at the nearest memory

Case-Study dMVM



- Interleaving memory policy
  Use **`numactl --interleave <NUMA node IDs|all>`**
  Round-Robin placement of memory pages
- For optimal placement, the code has to distribute the data

```
double* ptr = malloc(ARRAY_SIZE * sizeof(double))
if (!ptr)
    return;
#pragma omp for
for (int i=0; i< ARRAY_SIZE; i++)
    ptr[i] = M_PI;
```

Each threads triggers first touch policy to place the data local to them

- Commonly in OpenMP, the threads get the same slice of iteration space in equally sized loops

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

LIKWID:
https://github.com/RRZE-HPC/likwid