



# MuCoSim Introduction (Part II)

Thomas Gruber and Katrin Nusser (HPC @ Uni Erlangen)

[Thomas.Gruber@fau.de](mailto:Thomas.Gruber@fau.de)



Preparation:

Copy `~unrz139/mucosim` to your home  
(WARNING: may overwrite existing files!)

Let's start with some recap:

Go to `mucosim/stream` and compile the code with latest **Intel** suite

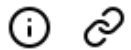
Run code with `OMP_NUM_THREADS=4` a few times and determine min. and max. bandwidth for the `copy` kernel.

# MuCoSim Introduction

COMPILERS



# Compilers @ RRZE



- We provide [GCC](#), [Intel C/C++ Compiler](#) (and PGI and others)
- On warmup also arm-clang (/opt/arm/... license up-on-request)
- Provided through **module** system
- Common compiler names:
  - GCC: **gcc** and **gfortran**
  - Intel: **icc** and **ifort**

Library names  
libtest.so → -ltest

Location(s) of headers

Location(s) of libs

```
gcc -O3 -fopenmp -I<INCDIR> -L<LIBDIR> test.c -o test -ltest  
gfortran -O3 -fopenmp -I<INCDIR> -L<LIBDIR> test.f90 -o test -ltest
```

Compiler options

Input file(s)

Output file

# Compilers @ RRZE

---

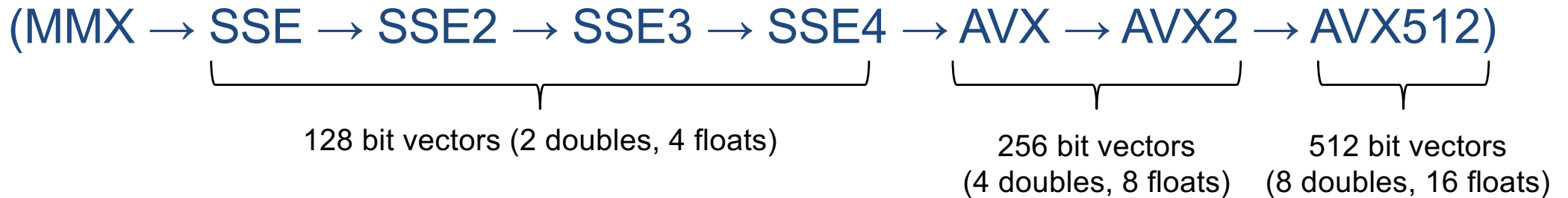
- Common compiler switches:

Meaning	GCC	Intel compilers
General optimization level	-O1, -O2, -O3 (some vendor specific options like <b>-Ofast</b> )	
Hardware feature flags	-m like <b>-mavx2</b>	-x like <b>-xCORE-AVX2</b>
Compiler feature flag	<b>-f</b> like <b>-fopenmp</b>	<b>-q</b> like <b>-qopenmp</b>

- In many cases, the Intel compiler
  - produces „better“ code and often better performing
  - provides fallbacks for GCC flags (**-fopenmp** accepted by ICC)
- CUDA compilers only available at nodes with GPUs

Compile `compile/get_cpuflags.c` with GCC and run it

What is the widest SIMD the system supports?



Compile `compile/triad.c` with recent GCC and ICC and ...

What's the minimum runtime you can achieve on the compute node?

Single hardware thread? Do optimization flags help?

All hardware threads?

```
Remember: Copy folder to your home/workdir/...  
cp -r ~unrz139/mucosim $HOME
```

# MuCoSim Introduction

## PERFORMANCE and TIMING

# How to measure performance?

---

- Performance = **WORK** / **TIME**

- **WORK**

- **1** : Time-to-solution, carefully define problem
- **Flops** : Floating-point operations (specify single-prec. or dbl.-prec.)
- **Particles|LatticeUpdates|Whatever** : Algorithm related work

- **TIME**

- UNIX **time** command can be confusing! Use **real** time  
Sometimes, **time** is a builtin, use **/usr/bin/time**

```
$ time <cmd>
<output>
real    0m0.008s
user    0m0.002s
sys     0m0.002s
```

- Best practice: Use high-resolution timers around region of interest



## How to measure performance? *Inside applications*

---

- Check **snippets** folder for helpful headers
  - **walltime.h: timestamp()** returns the current time in seconds
  - **cycletime.h: cyclestamp()** returns the number of cycles since boot
- For time measurements: **endstamp - startstamp**
- Careful when measuring small intervals:
  - Might be below resolution!
  - **walltime.h: resolution()** to check the current timer
- Check out **test\_times.c** for example usage

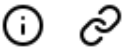
# MuCoSim Introduction

LIKWID

On emmy: Close interactive session and  
open a new one with `:likwid` property

On testfront: Keep your session

# What is LIKWID?



- A toolset for performance-oriented developers/users
- Get **system topology**
- **Place threads** according system topology (affinity domains)
- Run **micro-benchmarks** to check system features
- Measure **hardware events** during application runs
- Determine **energy consumption**
- Manipulate CPU/Uncore **frequencies** and **prefetchers**

# How to use LIKWID on FAU systems

---

- LIKWID is available in the module system

```
$ module avail likwid
```

- Always use newest version (currently 5.2.0)

- Disabled on production systems:

- **likwid-setFrequencies**
- **likwid-features**

Changes settings for all  
following jobs on that system!  
Reset yourself at end of job

- Module sets environment variables (`module show likwid/<version>`):

```
LIKWID_LIBDIR, LIKWID_INCDIR
```

```
gcc -I$LIKWID_INCDIR -LLIKWID_LIBDIR ... -llikwid
```

# System topology with LIKWID

```
$ likwid-topology -g
```

- Thread topology
- Cache topology
- NUMA topology
- Graphical topology

```
Socket 0:                               Intel Xeon CPU E5-2660 v2 @ 2.20GHz)
+-----+-----+-----+--+ +-----+ +-----+ +-----+
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| | 0 20 | | 1 21 | | 2 22 | | | | 7 27 | | 8 28 | | 9 29 | |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| | 32 kB | | 32 kB | | 32 kB | | | | 32 kB | | 32 kB | | 32 kB | |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| | 256 kB | | 256 kB | | 256 kB | | | | 256 kB | | 256 kB | | 256 kB | |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
| |                                     25                               MB | |
| +-----+ +-----+ +-----+ +-+ +-----+ +-----+ +-----+ |
+-----+-----+-----+--+ +-----+ +-----+ +-----+
```

# System topology with LIKWID

```
$ likwid-topology
```

```
CPU name: Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz
```

Product name

```
*****
```

```
Hardware Thread Topology
```

```
*****
```

```
Sockets:      2
```

```
Cores per socket: 18
```

SMT active!

```
Threads per core: 2
```

```
[...]
```

```
*****
```

```
NUMA Topology
```

```
*****
```

```
NUMA domains: 4
```

ClusterOnDie / SNC active (NUMA > Sockets)

# System topology with LIKWID

```
$ likwid-topology
```

```
*****
```

```
Cache Topology
```

```
*****
```

```
Level:      1  
Size:       32 kB  
Cache groups: ( 0 36 ) ( 1 37 ) ( 2 38 ) ...
```

```
Level:      2  
Size:       256 kB  
Cache groups: ( 0 36 ) ( 1 37 ) ( 2 38 ) ...
```

```
Level:      3  
Size:       22 MB  
Cache groups: ( 0 36 1 37 2 38 3 39 4 40 5 41 6 42 7 43 8 44
```

L1 Cache

Cache size

Hardware threads sharing a cache

`likwid-topology -c:`  
more infos about caches  
like cache line size,  
associativity, ...

How many HW threads does your compute node provide?

Does your system has CoD/SNC active?

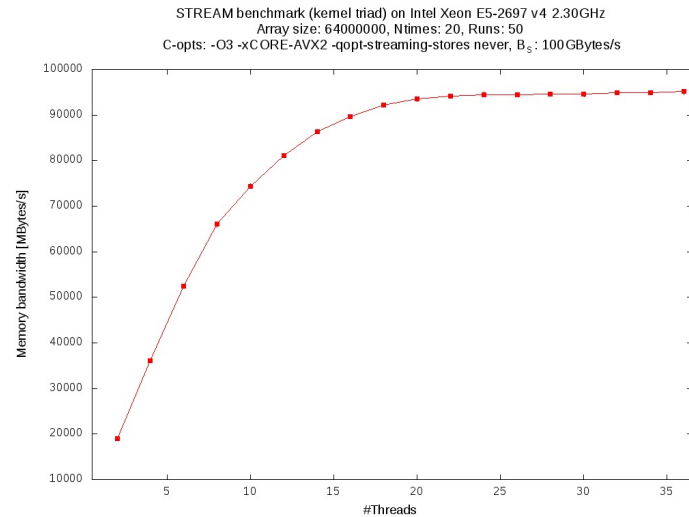
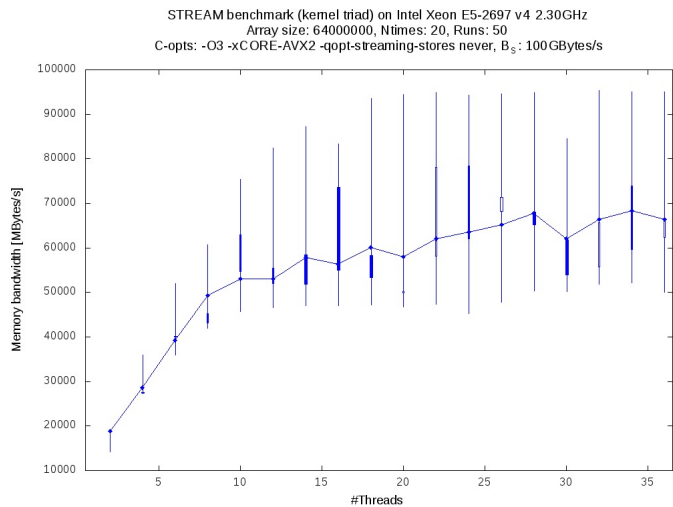
Is SMT active?

What's the L3 cache size?



# Task Affinity

- OS task scheduler places tasks (=processes/threads) on HW threads
- OS scheduler moves tasks to different cores from time to time
- STREAM benchmark:



This is how it looks when you saturate a shared resource

# Task Affinity

---

- Limiting set of possible HW threads per process/thread
- There are several reasons for caring about affinity:
  - Eliminating performance variation
  - Making use of architectural features
  - Avoiding resource contention
- Many tools/methods for affinity:
  - **taskset**: Limit set of HW threads (threads can be moved around)
  - **sched.h**: Application threads pin themselves
  - OpenMP/MPI-specific: Vendor-specific, often not portable
  - **numactl**: Limit application threads to NUMA domain (can be moved around)
  - **likwid-pin**
- Choose what fits best! Remember to set thread count!

# Task Affinity - numactl



- `numactl -C <cpulist> <executable>`
  - Same like `taskset` -> no real pinning!
- `numactl` provides more features regarding memory allocation
  - Bind memory to specific NUMA domains (`-m <nodelist>`)
  - Interleave memory in specific NUMA domains (`-i <nodelist>`)
- Some output functionality  
(`-s` for current settings and `-H` for NUMA hardware inventory)

Commonly not the way to do CPU pinning! But the right way for memory pinning!

# Task Affinity - `likwid-pin`



- **Pin processes/threads** without touching application code
- Supports **most threading solutions**
- **Requirement:** Application must be dynamically linked
- Support for multiple CPU selection syntaxes:
  - Physical: 0,4,5 or 0,4-5 → CPUs with ID 0,4 and 5
  - Logical: **S0**:0-3 → First four phy. cores on **Socket 0**
  - Expression based: E:N:20:1:2 → 20 threads, one out of two
- `likwid-pin -c <cpusel> ./a.out`

N: node  
Sx: socket  
My: NUMA domain  
Cz: LLC

Combine CPU selections with @

Skipping SMT threads

How many affinity domains does your system provide? (`--help`)

Compile `pin/hello_pthread.c` (`-pthread`)

Run it a few times, how often do threads share a CPU?

Pin `hello_pthread` (5 threads)

Pin `hello_pthread` to the first two physical HW threads of all sockets

What happens? Who wins?

`OMP_NUM_THREADS=10 likwid-pin -c 0-4 ./hello_pthread`

Run `stream` with 4 threads pinned differently (`N:0-3, S0:0-1@S1:0-1`).

What's the fastest CPU selection for `triad`?

# MuCoSim Introduction

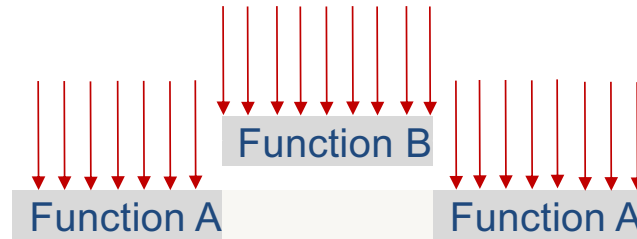
Analysis of applications



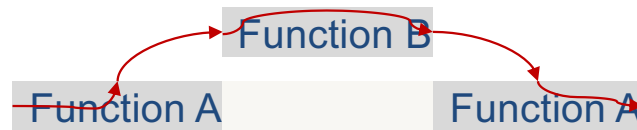
# Measurement techniques

gprof, xray, perf, ...

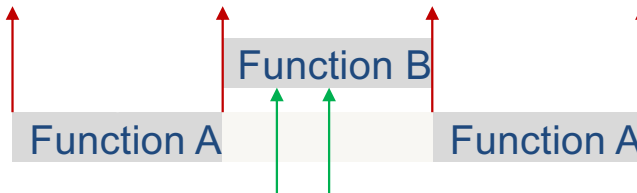
- Sampling



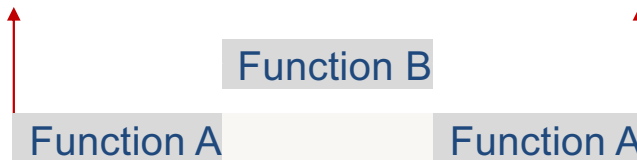
- Tracing



- Instrumentation



- Profiling



- Read state periodically
- Follow the execution path
- Add `read_state()` where desired
- Create an overview what happened

# Runtime profile - Find out **hotspots** in the code

---

- Many tools available: **gprof**, **xray**, **perf**, ...
- For **gprof** build with **-pg**
- Creates XML and tabular output files with fields:
  - Time and time share for function
  - Call and exit count
  - File and line of function
- Run application like normal
- Afterwards: **gprof <exec> gmon.out**



# Runtime profile

Time(%)	Self (sec)	Call count	Function
44.52	2.47	100	<code>triad()</code>
25.96	1.44	100	<code>add()</code>
16.94	0.94	100	<code>copy()</code>
100.00	0.00	1	<code>main()</code>

How often a function was called

Besides function runtime, how can we measure resource usage?

So, how to restrict measurements to the `triad()` function?

Compile `stream` in `runtime_profile` with runtime profiling  
(use `*.c` and include header path `-I.`)

Look at the hotspots in the code.  
Can you name a reason for the runtime difference?

# What is hardware performance monitoring?

## *Overview about HPM*

---

- Performance monitoring units (**PMUs**) at hardware level
- Introduced for x86 with Intel Pentium (1994)
- Originally used by CPU vendors for **hardware validation**
- **No additional CPU work** to handle hardware events in PMUs
- Accessing PMUs requires CPU work → **Overhead**
- Limited number of counters per PMU (x86: 4 per unit)

## Hardware Performance Monitoring with LIKWID - `likwid-perfctr`

---

- `likwid-perfctr` sets up system topology and perfmon
- Setup, start, read and stop PMUs
- Execute application on given CPU set (`-C`)
- Evaluate counter values

```
likwid-perfctr -C 0 -g INST_RETIRED_ANY:FIXCO <app>
```

- LIKWID needs you to specify which **counter** runs which **event**
- Combine multiple (event+counter)s with ‘,’
- For advanced usage, the events can be enriched with options **threshold**, **invert**, **count\_kernel**, **edge\_detect**,

# LIKWID - HPM with `likwid-perfctr`

---

```
$ likwid-perfctr -C 0,1 -g L2_TRANS_L1D_WB:PMC0 ./app
+-----+-----+-----+-----+
|          Event          | Counter |   Core 0   |   Core 1   |
+-----+-----+-----+-----+
| Runtime (RDTSC) [s]    |   TSC   | 2.573182e+00 | 2.573182e+00 |
|   L2_TRANS_L1D_WB     |   PMC0  | 281176518   | 281240170   |
+-----+-----+-----+-----+
```

- Event names (in many cases) **not intuitive**
- Events are **architecture-specific**
- Some sound promising but return bad counts, others are broken
- **More interest in real metrics** like volume of loaded/stored data

# LIKWID - HPM with `likwid-perfctr`

- LIKWID defines performance groups  
≈ eventlist + derived metrics + documentation
- List all groups: `likwid-perfctr -a`

You can also define own performance groups!

```
$ likwid-perfctr -C 0,1 -g L2 ./app
```

Metric	Core 0	Core 1
Runtime (RDTSC) [s]	2.6439	2.6439
L2D load bandwidth [MBytes/s]	6744.8121	6743.6037
<b>L2D load data volume [GBytes]</b>	17.8325	17.8293
L2D evict bandwidth [MBytes/s]	3372.4061	3371.8019
<b>L2D evict data volume [GBytes]</b>	8.9163	8.9147

# LIKWID - Performance groups

---

- **FLOPS\_AVX**: Packed AVX MFlops/s
- **FLOPS\_DP**: Double Precision MFlops/s
- **FLOPS\_SP**: Single Precision MFlops/s
- **DATA**: Load to store ratio
- **L2**: L2 cache bandwidth in MBytes/s
- **L3**: L3 cache bandwidth in MBytes/s
- **MEM**: Main memory bandwidth in MBytes/s
- **ENERGY**: Power and Energy consumption
- **MEM\_DP**: Memory & DP FLOP/s & Energy
- **MEM\_SP**: Memory & SP FLOP/s & Energy

Overcounting on Intel SandyBridge & IvyBridge. No **FLOPS\_\*** groups on Intel Haswell.

## Hardware Performance Monitoring with LIKWID - `likwid-perfctr`

---

- `$ likwid-perfctr -C 0,1 -g FLOPS_DP ./a.out`  
Measure DP FLOP/s of the whole application run of on CPUs 0, 1
- `$ likwid-perfctr -c 0,1 -g DATA ./a.out`  
Measure load/store ratio on CPUs 0,1. Application is **not** pinned!
- `$ likwid-perfctr -g MEM_DP -H`  
Get **help** for performance group MEM\_DP
- `$ likwid-perfctr -e (| less)`  
List all events and counters, search with `-E <searchstr>`



Compile `perfctr/triad.c`

Measure the memory bandwidth (**MEM**)  
from 1 to number of phys. cores per socket  
At which core count does it saturate?

Compile `perfctr/pi.c`

Measure the FLOP rate from 4 to 10 processes on one socket  
Does it have a saturation point?

How well is it vectorized? What is the max. vectorization ratio you can achieve? Are all operations done with „best“ vectorization?

# LIKWID - *HPM of functions*

- LIKWID offers MarkerAPI for code region measurements

```
#include <likwid-marker.h>
LIKWID_MARKER_INIT; // in serial region
LIKWID_MARKER_REGISTER("Compute"); // in parallel region
LIKWID_MARKER_START("Compute");
<code>
LIKWID_MARKER_STOP("Compute");
LIKWID_MARKER_CLOSE; // in serial region
```

Older version use  
<likwid.h>


Reduces startup  
overhead

Multiple regions and  
nesting allowed

- Compile with `-DLIKWID_PERFMON`

# Add marker API to code (restructure loops)

---


```
#pragma omp parallel for 
  <loop>
```

```
#pragma omp parallel
{
  LIKWID_MARKER_START("Compute");
  #pragma omp for
    <loop>

  LIKWID_MARKER_STOP("Compute");
}
```

# Add marker API to code (closed-source library calls)

---

```
some_parallel_f()  #pragma omp parallel  
{  
    LIKWID_MARKER_START("foo")  
}  
#pragma omp parallel  
{  
    LIKWID_MARKER_STOP("foo")  
}
```

# LIKWID - HPM of functions

Compile @ RRZE:

```
$CC -DLIKWID_PERFMON $LIKWID_INC $LIKWID_LIB code.c -o code -llikwid
```

Defined by LIKWID module at RRZE

```
likwid-perfctr -C <cpustr> -g <group> -m ./a.out
```

Use capital C  
MarkerAPI requires pinned threads

Tells likwid-perfctr to  
use MarkerAPI mode

Copy `perfctr/pi.c` to `marker/pi.c`

Add MarkerAPI calls around loop for each OpenMP thread & compile  
Measure the FLOP rate from 4 to number of phys. cores per socket

Compile `marker/stream.c` (use `-I. *.c` )

What are the read and write memory bandwidths of each hotspot for 4 threads?

Compare results to the application output of stream.

Is there a difference and if yes, why?



Thank you for your attention!

Erlangen National High Performance Computing Center (NHR@FAU)

Martensstraße 1, 91058 Erlangen

<http://www.rrze.fau.de>

[Thomas.Gruber@fau.de](mailto:Thomas.Gruber@fau.de)

