

Exercise 1

Introduction to High-Performance Computing
WS 2019

Tim Cramer, Julian Miller
contact@hpc.rwth-aachen.de

■ 6 exercises

- Exercises will be on Moodle one week before the exercise
- Solve them alone at home, discussion/questions during frontal exercise
- In preparation, **create an HPC account**:
<https://www.rwth-aachen.de/selfservice>
send TIM to contact@hpc.rwth-aachen.de
with “[19WS-HPC] TIM” as subject
- Prerequisites: Basic C/C++ or FORTRAN, Linux/UNIX

■ Cluster and CAVE tour (MF)

- November 19th instead of the lecture

■ Last lecture: time for questions (F)

Lectures & Exercises as of October 25th 2019

Updates will be announced on Moodle

KW	Date	Type	Date	Type
41	08.10.2019	-	11.10.2019	V
42	15.10.2019	V	18.10.2019	V
43	22.10.2019	V	25.10.2019	Ü
44	29.10.2019	V	01.11.2019	-
45	05.11.2019	-	08.11.2019	V
46	12.11.2019	Ü	15.11.2019	V
47	19.11.2019	MF	22.11.2019	V
48	26.11.2019	V	29.11.2019	Ü
49	03.12.2019	V	06.12.2019	V
50	10.12.2019	V	13.12.2019	V
51	17.12.2019	V	20.12.2019	V
02	07.01.2020	Ü	10.01.2020	V
03	14.01.2020	V	17.01.2020	Ü
04	21.01.2020	V	24.01.2020	Ü
05	28.01.2020	V	31.01.2020	F

KW: week, V: lecture, Ü: exercise

MF: Maschinenhallen/AIXCave-führung

F: Fragenstunde

■ Tour through the cluster room and CAVE

- November 19th, instead of lecture
- Please register through the survey on Moodle until November 15th

■ Exams

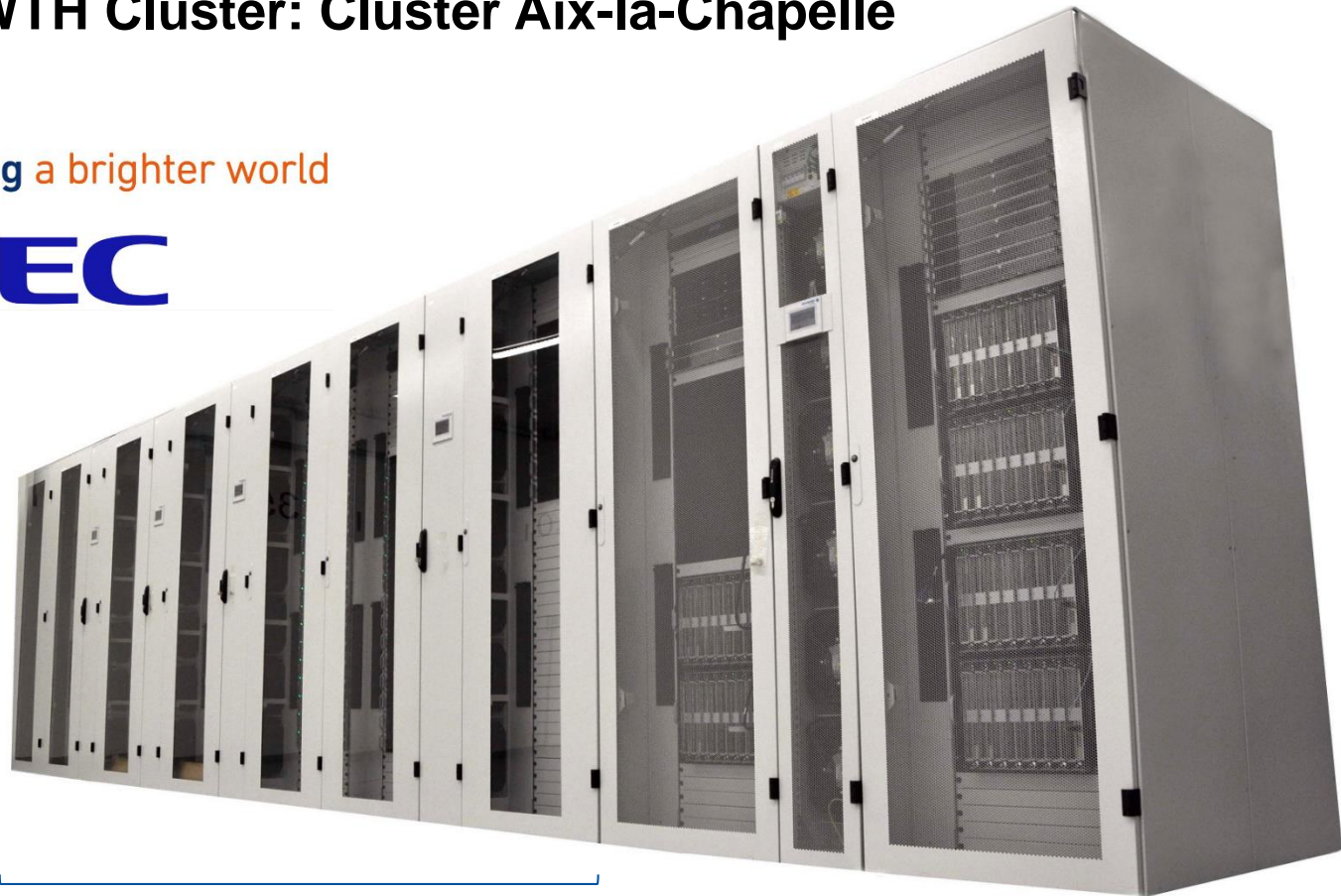
- February 28th, 8:30 - 10:30, Aula 2 (2352|021)
- March 25th, 14:30 – 16:30, Aula 2 (2352|021)
- 120 minutes, no additional materials allowed

- 0. Get familiar with the RWTH Compute Cluster**
- 1. Pipelining**
- 2. Caches**
- 3. Stream2 Benchmark**

Current RWTH Cluster: Cluster Aix-la-Chapelle

\Orchestrating a brighter world

NEC



water-cooled

air-cooled



■ Phase I (2016): 600+ standard MPI nodes

- 2x Intel Xeon Broadwell-EP (E5-2650v4): 12 cores each, 2.2 GHz
 - Turbo: up to 2.9 GHz (2.5 GHz for AVX instructions)
- 128 GB DDR4-2400 main memory
- Intel Omni-Path x16 HPC network

■ Phase II (2018): 1000+ standard MPI nodes

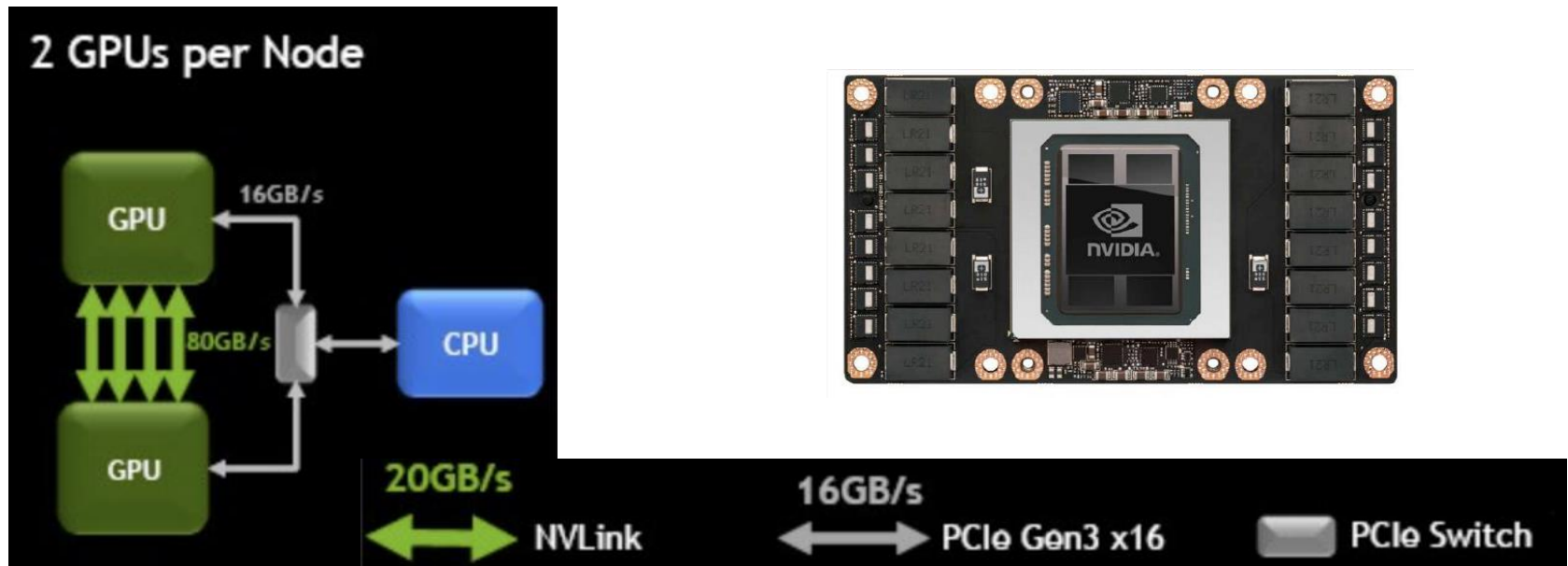
- 2x Intel Xeon Platinum (8160): 24 cores each, 2.1 GHz
 - Turbo: up to 3.7 GHz
- 192 GB DDR main memory
- Intel Omni-Path x16 HPC network



■ GPU nodes

- Tech specs same as MPI nodes, plus:
- Phase I (2016): 2x NVIDIA P100 (Pascal architecture)
- Phase II (2018): 2x NVIDIA V100 (Volta architecture)

■ Connected via NVLink



■ Phase I (2016): 8 SMP nodes

- 8x Intel Xeon Broadwell-EX (E7-8860v4): 18 cores each, 2.2 GHz
- 1 TB main memory
- 2x 2 TB NVMe SSD
(see NVMe nodes later)
- Intel Omni-Path x16 HPC network

■ 2 (out of the 8) systems with 1x NVIDIA P100

- PCIe card

■ Air-cooled nodes



■ Main dialog systems (CLAIX18)

- `login18-{1,2,3,4}.hpc.itc.rwth-aachen.de`
- Use frontends to develop program, compile applications, prepare job scripts or debug programs
- cgroups activated for fair-share

■ Different frontends for different purposes

- Only for short (!) tuning purposes: `login18-t.hpc.itc.rwth-aachen.de`
- File transfers: `copy18-{1,2}.hpc.itc.rwth-aachen.de`
- GPU system: `login18-g-{1,2}.hpc.itc.rwth-aachen.de`
- X-Sessions: `login18-x-{1,2}.hpc.itc.rwth-aachen.de`

- **Many compilers, MPIs and ISV software**
- **The module system helps to manage all the packages**
 - List loaded modules
 - `$ module list`
 - List available modules
 - `$ module avail`
 - Load / unload a software
 - `$ module load <modulename>`
 - `$ module unload <modulename>`
 - Exchange a module (Some modules depend on each other)
 - `$ module switch <oldmodule> <newmodule>`
 - Reload all modules (May fix your environment)
 - `$ module reload`
 - Find out in which category a module is:
 - `$ module apropos <modulename>`

\$ module avail

----- /usr/local_rwth/modules/modulefiles/linux/x86-64/DEVELOP -----

archer/3.8-20151105(default)

clang/3.8

clang/3.9(default)

cmake/2.8.12

cmake/3.4.1(default)

cmake/3.5.2

cmake/3.6.0

cuda/75(default)

...

intelvtune/XE2016-u03

intelvtune/XE2017-u00(default)

likwid/system-default(default)

nagfor/6.0

nagfor/6.1(default)

openmpi/1.10.1

openmpi/1.10.1mt

openmpi/1.10.2(default)

...

----- /usr/local_rwth/modules/modulefiles/GLOBAL -----

BETA

DEPRECATED

GRAPHICS

MATH

TECHNICS

CHEMISTRY

DEVELOP

LIBRARIES

MISC

UNITE



modules



categories

■ For convenience we provide several environment variables

→ Set by the module system

Variable	Function
\$FC, \$CC, \$CXX	Compiler
\$FLAGS_DEBUG	Compiler option to enable debug information.
\$FLAGS_FAST	Enables several compiler optimization flags.
\$MPIFC, \$MPICC, \$MPI	MPI compiler wrapper.
\$MPIEXEC	The MPI command used to start MPI applications.
\$FLAGS_MPI_BATCH	MPI options necessary for executing in batch mode.
\$FLAGS_OPENMP	Compiler option to enable OpenMP support.
\$OMP_NUM_THREADS	Sets the number of threads for OpenMP applications.
\$FLAGS_MATH_INCLUDE	Include flags for mathematical libraries (e.g. Intel MKL)
\$FLAGS_MATH_LINKER	Linker flags for mathematical libraries (e.g. Intel MKL)

■ How to submit a job

→ `$ sbatch [options] command [arguments]`

■ General parameters

Parameter	Description
<code>--job-name=<name> (-J)</code>	Job name
<code>--output=<path> (-o)</code>	Output file
<code>--mail-type=BEGIN</code>	Send mail when job starts running
<code>--mail-type=END</code>	Send mail when job is done
<code>--mail-user=<mailaddress></code>	Receipient of mails
<code>--account=<project> (-A)</code>	Assign the job to the specified project

■ How to submit a job

→ `$ sbatch [options] command [arguments]`

■ Parameters for job limits / resources

Parameter	Description
<code>--time=<runlimit> (-t)</code>	Sets the hard runtime limit in the format hour:minutes:sec [default: 15]
<code>--mem-per-cpu=<memlimit></code>	Sets a per-process memory limit in MB [default: 512]
<code>--exclusive</code>	Request node(s) exclusive
<code>--gres=gpu:<type>:<n></code>	Request <i>n</i> GPUs of type <i>type</i> per node, where <i>type</i> ={volta,pascal}

■ How to submit a job

→ `$ sbatch [options] command [arguments]`

■ Parameters parallel jobs

Parameter	Description
<code>--ntask=<num_procs></code>	Submits a parallel job with <i>num_procs</i> processes (e.g. MPI) [default: 1]
<code>--cpus-per-task=<num_thr></code>	Use this to submit a shared memory job (e.g. OpenMP) with <i>num_thr</i> threads
<code>--tasks-per-node=<num_ta></code>	Specify the number of processes per node (e.g. hybrid OpenMP/MPI)

→ `$MPIEXEC $FLAGS_MPI_BATCH a.out`

■ Use the magic cookie **#SBATCH** for a batch script **job.sh**

```
#!/usr/local_rwth/bin/zsh
#SBATCH --job-name=lect0038      #Job name
#SBATCH --output=output_%J.txt  #Output, the %J is the job id
#SBATCH -A lect0038              #Request lecture project
#SBATCH --time=00:05:00          #Request 5 minutes max runtime
#SBATCH --mem-per-cpu=512M       #Request 512 MB virtual mem
#SBATCH --mail-user=a@b.de       #Specify your mail
#SBATCH --mail-type=END          #Send a mail when job is done
cd /home/user/workdirectory     #Change to the work directory
helloWorld.out                  #Execute your application
```

■ Submit this job

→ `$ sbatch job.sh`

■ The only supported shell is the default shell „zsh“

→ `#!/usr/local_rwth/bin/zsh`

■ Use `squeue` to display information about LSF jobs

→ `$ squeue -u `whoami``

JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST (REASON)
59190128	c18m	Job_2	ab123456	PD	0:00:00	2	(Priority)
59190123	c18m	Job_1	ab123456	R	3:32:16	1	ncm0152
59189781	c18m	Job_3	ab123456	R	4:02:44	2	ncm[0650-0651]

→ You can modify the format by setting the environment variable
`SQUEUE_FORMAT`

■ Kill a job

→ `$ scancel <jobid>`

■ Documentation

→ <https://doc.itc.rwth-aachen.de/display/CC/Using+the+SLURM+Batch+System>

→ <https://hpc-wiki.info/hpc/SLURM>

0. Get familiar with the RWTH Compute Cluster
- 1. Pipelining**
2. Caches
3. Stream2 Benchmark

- **Given: An architecture with a latency of 3 cycles for a single add operation**
- **Array reduction code**

```
const int N=20;  
double s = 0.0;  
double A[N]; // init A  
for (i=0; i<N; i++)  
    s = s + A[i];
```

- Are there (real) dependencies in the code with respect to the execution of the add operations?
- Or can the execution of the add operation be perfectly pipelined? Why?
- Can you imagine which optimization is important for this reduction example?

■ Dependencies within loops may prevent efficient software pipelining

→ Software pipelining: interleaving of loop iterations to meet latency requirements (done by the compiler)

No Dependency

```
//FORTRAN
DO I=1,N
  A(I)=A(I)*c
END DO
```

```
//C
for(i=0;i<N;++i)
  A[i]=A[i]*c;
```

Real Dependency

```
//FORTRAN
DO I=2,N
  A(I)=A(I-1)*c
END DO
```

```
//C
for(i=1;i<N;++i)
  A[i]=A[i-1]*c;
```

Pseudo Dependency

```
//FORTRAN
DO I=1,N-1
  A(I)=A(I+1)*c
END DO
```

```
//C
for(i=0;i<N-1;++i)
  A[i]=A[i+1]*c;
```

1.1 Pipelining Issues – Solution

■ Array reduction code

```
const int N=20;  
double s = 0.0;  
double A[N]; // init A  
for (i=0; i<N; i++)  
    s = s + A[i];
```

```
LOAD r1.0 ← 0  
i ← 0  
loop:  
    LOAD r2.0 ← a(i)  
    ADD r1.0 ← r1.0+r2.0  
    ++i →? loop  
result ← r1.0
```

cycles	1	2	3	4	5
ADD 1	s←A[0]+s			s←A[1]+s	
ADD 2		s←A[0]+s			...
ADD 3			s←A[0]+s		

■ There is a real dependency for reduction variable s.

→ Add operation cannot be pipelined without optimization.

1.1 Pipelining Issues – Solution

■ With loop unrolling, the code can be optimized for pipelining.

```
const int N=20;
double s = 0.0;
double A[N]; // init A
for (i=0; i<N; i++)
    s = s + A[i];
```

```
const int N=20;
double s1 = 0.0;
double s2 = 0.0;
double s3 = 0.0;
double A[N]; // init A
//remainder iterations
//must also be handled
for (i=0; i<N; i+=3) {
    s1 = s1 + A[i];
    s2 = s2 + A[i+1];
    s3 = s3 + A[i+2];
}
s1 = s1 + s2 + s3;
```

```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 0
loop:
    LOAD r4.0 ← a(i)
    LOAD r5.0 ← a(i+1)
    LOAD r6.0 ← a(i+2)
    ADD r1.0 ← r1.0+r4.0
    ADD r2.0 ← r2.0+r5.0
    ADD r3.0 ← r3.0+r6.0
    i+=3 →? loop
    result ← r1.0+r2.0+r3.0
```

Note: Compiler should automatically do unrolling if optimization is enabled. Usually no need to write code as on the left bottom.

cycles	1	2	3	4	5
ADD 1	s1←A[0]+s1	s2←A[1]+s2	s3←A[2]+s3	s1←A[3]+s1	
ADD 2		s1←A[0]+s1	s2←A[1]+s2	s3←A[2]+s3	...
ADD 3			s1←A[0]+s1	s2←A[1]+s2	s3←A[2]+s3

- **N = 20, 3 cycles add latency**
- **Assume perfect pipeline utilization of the execution of the add operation.**

→ What can be the maximum speedup with the given N?

→ What is the theoretical speedup for an infinite big N?

■ **Solution**

→ Speedup:
$$T_p = \frac{T_{seq}}{T_{pipe}} = \frac{mN}{(N+m-1)} = \frac{m}{1+\frac{m-1}{N}}$$

→ With N=20, m=3:
$$T_p = \frac{3 \cdot 20}{(20+3-1)} = \frac{60}{22} \approx 2.72$$

→ Theoretical max speedup:
$$T_p = \frac{3}{1+\frac{3-1}{N}} \xrightarrow{N \rightarrow \infty} 3$$

→ With an array length of 20, we are already close to the maximum speedup.

T_{seq} :	#cycles w/o pipeline
T_{pipe} :	#cycles w/ pipeline
T_p :	speedup
m :	#stages
N :	#operations

0. Get familiar with the RWTH Compute Cluster
1. Pipelining
- 2. Caches**
3. Stream2 Benchmark

2. Caches



High
Performance
Computing



IT Center



- **2.1 Associativity**
- **2.2 Locality**
- **2.3 Calculate Speedup**
- **2.4 Capacity**
- **2.5 Coherence**

■ Task of a Cache-controller: Mapping of Data to Memory

- lookup (comparison) to find an address within the cache (cache hit)
- otherwise: cache-miss (load data from main memory)
- function necessary to map addresses to cache entries

■ direct mapped

- part of the address selects the entry (used as an index)

■ fully associative

- address is compared to all entries

■ n-way (set) associative

- part of the address selects the block (block has n entries)

See lecture
02 – Modern Processors 2
slide 92 f.

2.1 Associativity

Memory blocks

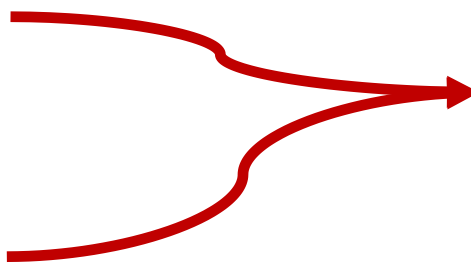
0: ...

1: ...

2: ...

3: ...

4: ...



	TAG	DATA
0	0 4	
1		
2		
3		

Disadvantage: In case of frequent accesses to addresses which map to the same entry we always have a cache miss.

■ direct mapped

→ part of the address selects the entry (used as an index)

■ fully associative

→ address is compared to all entries

■ n-way (set) associative

→ part of the address selects the block (block has n entries)

See lecture
02 – Modern Processors 2
slide 92 f.

2.1 Associativity

Memory blocks

0: ...

1: ...

2: ...

3: ...

4: ...

	TAG	DATA	TAG	DATA	TAG	DATA	TAG	DATA
	0	4						

The entry can be everywhere.
Disadvantages: higher latency +
more HW required

■ direct mapped

→ part of the address selects the entry (used as an index)

■ fully associative

→ address is compared to all entries

■ n-way (set) associative

→ part of the address selects the block (block has n entries)

See lecture
02 – Modern Processors 2
slide 92 f.

2.1 Associativity

Memory blocks

0: ...

1: ...

2: ...

3: ...

4: ...

	TAG	DATA	TAG	DATA
0	0		4	
1				

Two-Way-Set-Associative

■ direct mapped

→ part of the address selects the entry (used as an index)

■ fully associative

→ address is compared to all entries

■ n-way (set) associative

→ part of the address selects the block (block has n entries)

See lecture
02 – Modern Processors 2
slide 92 f.

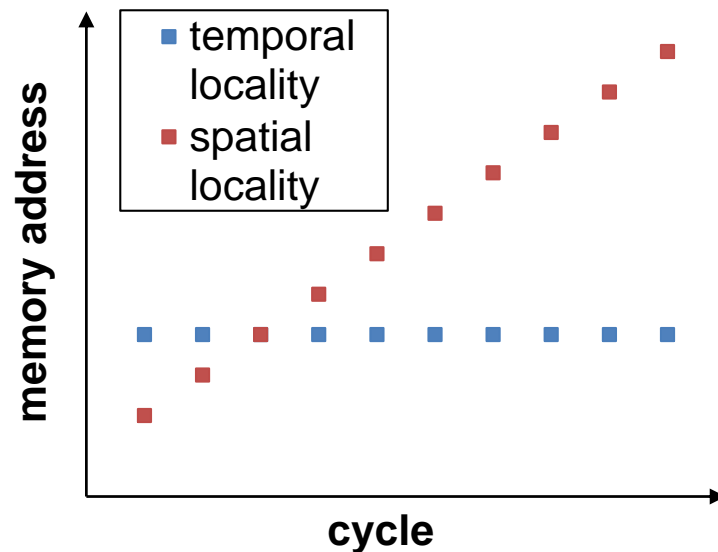
■ Caches use locality in space and time to speedup loads and stores

→ spatial locality

→ continuous data access (nearby address)

→ temporal locality

→ repeating data access (same address) in the near future



See lecture
02 – Modern Processors 2
slide 76

2.3 Speedup



High
Performance
Computing



IT Center



■ Task: Calculate the average access time for an arbitrary memory location and the performance gain!

→ $T_m = 180ms$

→ $T_c = 20ms$

→ Hit rate: $\beta = 0.5$

T_m :	access time local memory
T_c :	access time cache
T_{av} :	average access time

■ Caches use locality to speedup loads and stores

→ Hit rate: β

→ Average access time: $T_{av} = \beta T_c + (1 - \beta) T_m$

T_m : access time local memory

T_c : access time cache

T_{av} : average access time

τ : speedup

→ Performance gain: $G(\beta, \tau) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1 - \beta) \tau T_c} = \frac{\tau}{\beta + \tau(1 - \beta)}$

→ Here:

$$T_{av} = 0.5 \cdot T_c + 0.5 \cdot T_m = 100ms,$$

$$\text{with } T_c = 20ms, T_m = 180ms$$

$$G(\beta, \tau) = \frac{T_m}{T_{av}} = 1.8$$

See lecture
02 – Modern Processors 2
slide 78 f.

2.4 Capacity



High
Performance
Computing



IT Center



- Assume a direct mapped cache has a size of $C = 1 \text{ MiB}$ (2^{20} byte) and one cache line has a length of $C_L = 64 \text{ byte} = 2^6 \text{ byte}$. How many bits of a memory address are used as an index to identify the corresponding cache line when this cache line is fetched from main memory?

NOTE: There was a error in version 1 of the exercise.
The cache line length is denoted by C_L , while N_{CL}
denotes the number of cache lines.

- Assume a direct mapped cache has a size of $C = 1 \text{ MiB}$ (2^{20} byte) and one cache line has a length of $C_L = 64 \text{ byte} = 2^6$ byte. How many bits of a memory address are used as an index to identify the corresponding cache line when this cache line is fetched from main memory?

- Solution

→ $1 \text{ MiB} = 1024 \cdot 1024 \text{ byte} = 2^{20} \text{ byte}$

→ Capacity of a cache $C = N_{CL} C_L \rightarrow N_{CL} = \frac{C}{C_L} = \frac{2^{20}}{2^6} = 2^{14}$

→ Required bits to address all cache lines: $\text{ld}(2^{14}) = 14$

→ By reserving 14 bits for the index all cache lines of a direct mapped cache can be addressed!

■ Modern Processors are Multi-core with Multiple levels of Caches

- Low-level caches are typically private per core while high-level caches are shared between cores
- multi-core processors: memory can be accessed and altered by multiple cores
 - Modifications of cached data must be visible to all cores

■ What is cache coherence?

- guarantees same result for access to the same address
- implemented in hardware through directories or bus snooping
- Modern multi-core processors use write-invalidate protocols (MSI, MESI,...)

0. Get familiar with the RWTH Compute Cluster
1. Pipelining
2. Caches
- 3. Stream2 Benchmark**

■ Measures sustainable memory bandwidth (MB/s)

→ Exposes clearly performance difference between reads & writes

■ Following operations are tested (default type: double)

fill: $\vec{a} = q$

copy: $\vec{a} = \vec{b}$

daxpy: $\vec{a} = \vec{b} + q \cdot \vec{c}$

dot: $sum = sum + \vec{a} \cdot \vec{b}$

Kernel	Code	Bytes/iter read	Bytes/iter written	FLOPS/iter
FILL	$a(i) = q$	0	8	0
COPY	$a(i) = b(i)$	8	8	0
DAXPY	$a(i) = a(i) + q \cdot b(i)$	16	8	2
DOT	$sum += a(i) * b(i)$	16	0	2

See lecture
02 – Modern Processors 2
slide 73 f.

- **Compile stream2 benchmark:** `make`
- **Bind the program to one core:** `taskset -c <ID> ./stream2.exe`
 - `$ lstopo-no-graphics` shows the architecture
- **Run the program in an exclusive batch job:**
`$ sbatch batchStream2.sh`
- **Plot the results:** `gnuplot -e "FILE='stream2.txt'" plotData.plt`
 - What can you see? Explain the stairs in the figure.
- **Solution**
 - The Stream2 benchmark shows the different cache levels and their size.
 - Best bandwidth from L1 cache. If data does not fit into L1 cache anymore, it must be carried from L2 cache (next stair) and so on...

Hardware specification (Intel Skylake)

→ L1D cache:

32 KB per core →

$$\frac{32 \text{ KB}}{8 \text{ B/double}} = 4K \text{ elements}$$

→ L2 cache:

1024 KB per core →

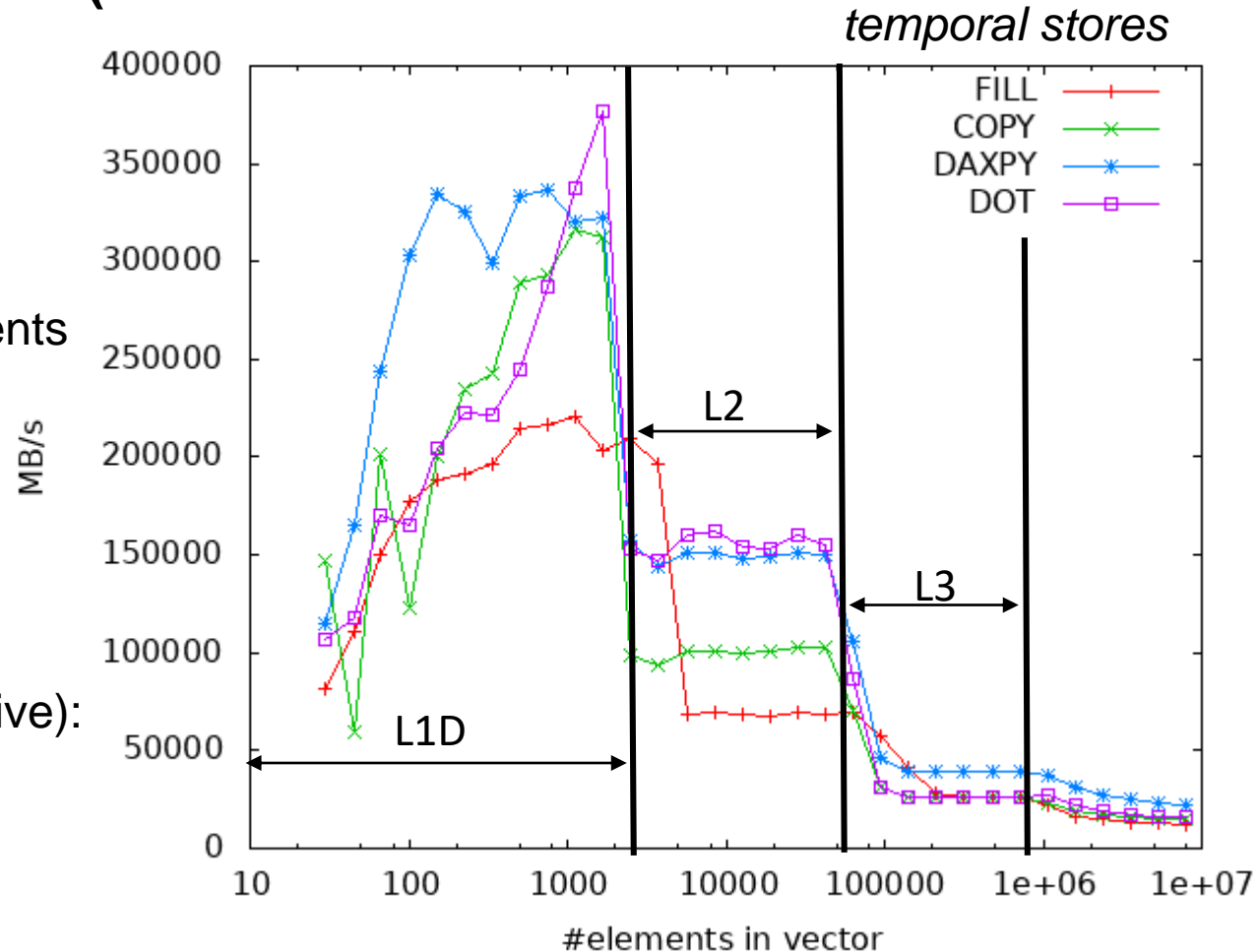
128K elements

→ L3 cache (non-inclusive):

33 MB per socket →

4125K elements

→ Memory: 192 GB per node



Thank you for your attention.