Performance Engineering for Legacy Codes on a Cray XC40 with Intel Xeon Phi (KNL)

Matthias Noack (noack@zib.de), Florian Wende,
Thomas Steinke, Alexander Reinefeld

Zuse Institute Berlin

North-German Supercomputing Alliance (HLRN)

TDS (Berlin, ZIB)
- 16 KNC nodes (until July 2016)
- **80 KNL nodes** (since July 2016)
- data warp nodes

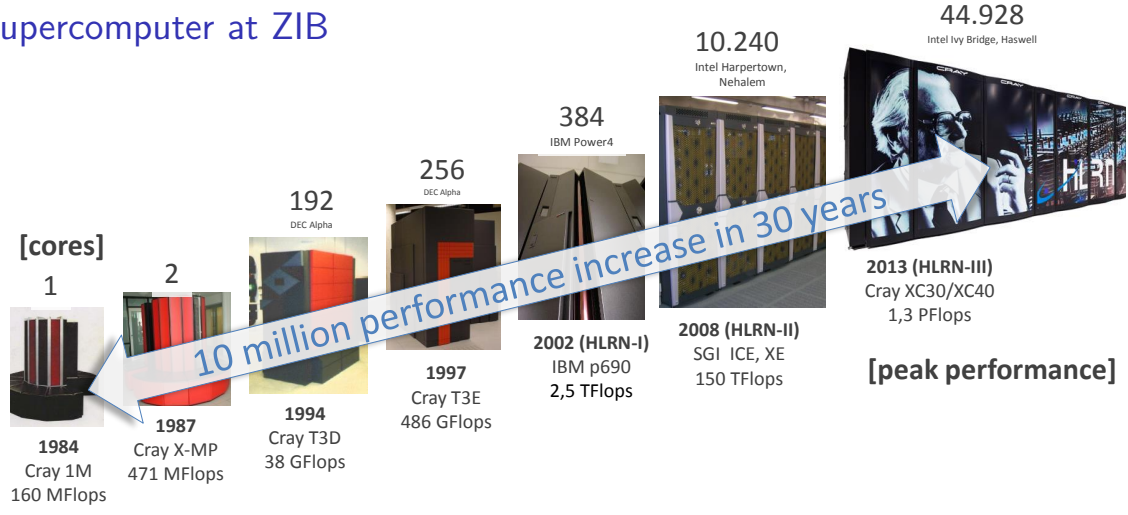
"Konrad" (Berlin, ZIB)
- 1872 Xeon nodes
- 44928 cores

10 Gbps (243 km linear distance)

"Gottfried" (Hanover, LUIS)
- 1680 Xeon nodes
- 40320 cores + 64 SMP servers, 256/512 GB

Applications on the HLRN-III
- many pure MPI codes
- some MPI+OpenMP
- some vectorized

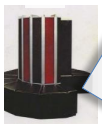
Supercomputer at ZIB



Supercomputer at ZIB

[cores]

1



1984
Cray 1M
160 MFlops

2

1987
Cray X-MP
471 MFlops

192
DEC Alpha

1994
Cray T3D
38 GFlops

256
DEC Alpha

1997
Cray T3E
486 GFlops

384
IBM Power4

2002 (HLRN-I)
IBM p690
2,5 TFlops
200 kWatt
10 M€

10.240
Intel Harpertown, Nehalem

2008 (HLRN-II)
SGI ICE, XE
150 TFlops

44.928
Intel Ivy Bridge, Haswell

2013 (HLRN-III)
Cray XC30/XC40
1,3 PFlops

[peak performance]

10 million performance increase in 30 years

Xeon Phi KNL, 2016

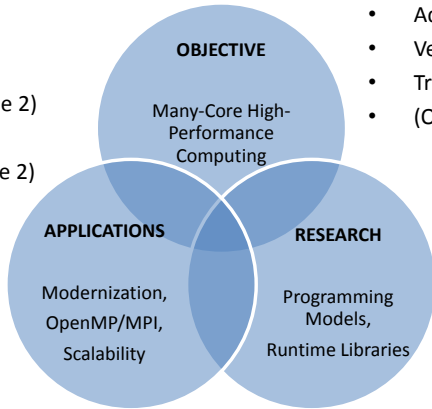
Intel Xeon Phi 7290
72 cores (288 threads)
3 TFLOPS
245 Watt
6662 € (6/2017)

Research Center for Many-Core HPC at ZIB

Intel Parallel Compute Center (IPCC)

Applications

- ***GLAT*** (atomistic thermodynamics)
- ***VASP*** (electronic structure)
- ***BQCD*** (high-energy physics)
- ***HEOM*** (photo-active processes)
- ***BOSS*** (time series analysis, phase 2)
- ***PALM*** (fluid dynamics, phase 2)
- ***PHOENIX3D*** (astrophysics, phase 2)



Challenges

- Adapting data structures for enabling SIMD
- Vectorising complex code structures
- Transition to hybrid MPI + OpenMP
- (Offload with Intel LEO vs. OpenMP 4.x)

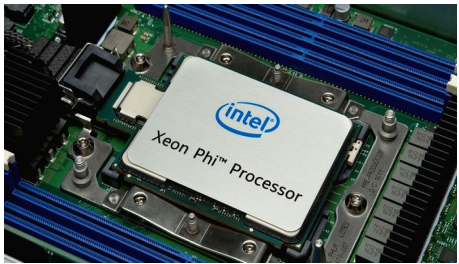
intel

CRAY
THE SUPERCOMPUTER COMPANY

Intel Xeon Phi (Knights Landing) – Architecture

Knights Landing (KNL): Intel's 2. Many Integrated Core (MIC) Architecture[1)]

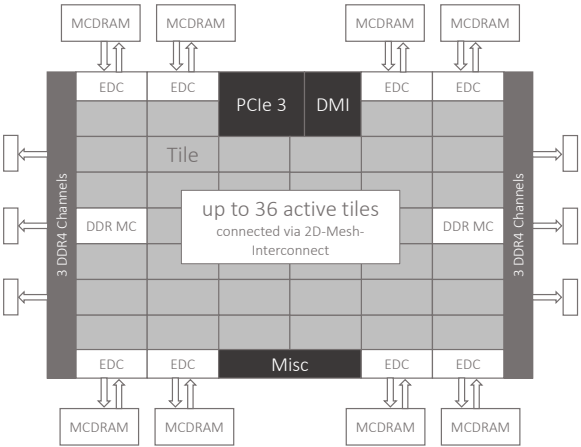
- self-booting CPU, optionall with integrated OmniPath fabric
- 64+ core (based on Intel Atom Silvermont architecture, x86-64)
  - 4-way hardware-threading
  - 512-bit SIMD vector processing (AVX-512)
- On-Chip Multi-Channel (MC) DRAM: 16 GiB
- DDR4 main memory: up to 384 GiB



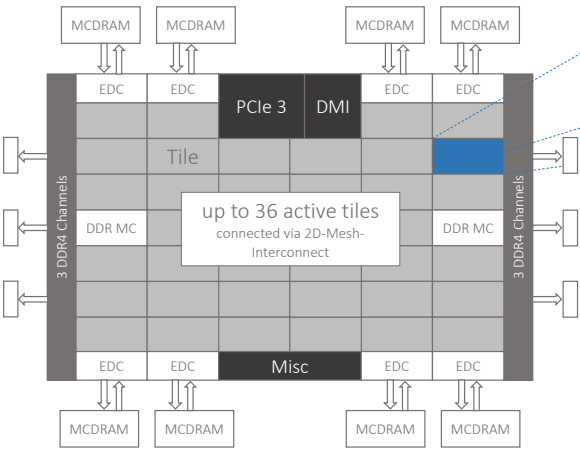
[1)] A. Sodani et al., *Knights Landing: Second-Generation Intel Xeon Phi Product*, IEEE Micro vol. 6, April 2016

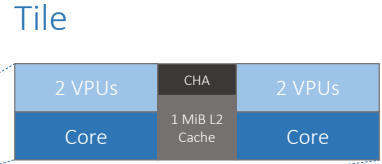
Intel Xeon Phi (Knights Landing) – Architecture

KNL CPU



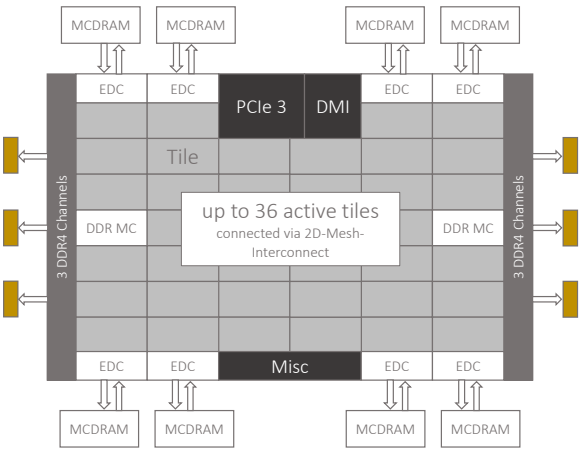
Intel Xeon Phi (Knights Landing) – Architecture





- 2 out-of-order cores
- 2 VPUs per core (AVX-512)
- 1 MiB shared L2-Cache
- Caching/Home-Agent (CHA)
  - interface to 2D-Mesh-Interconnect
  - distributed tag directory
    (MESIF cache coherency protocol)

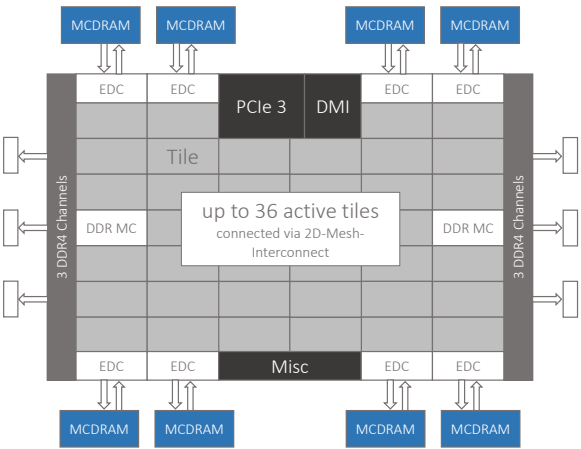
Intel Xeon Phi (Knights Landing) – Architecture



DDR4 Memory (up to 384 GiB)

- 2 memory controller
- 6 DDR4 Channels

Intel Xeon Phi (Knights Landing) – Architecture



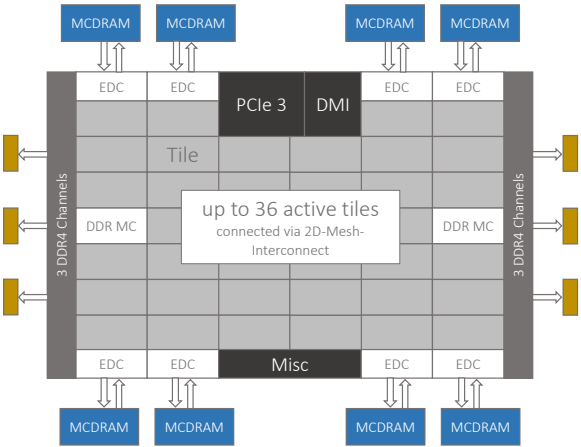
DDR4 Memory (up to 384 GiB)

- 2 memory controller
- 6 DDR4 Channels

MCDRAM (16 GiB)

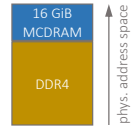
- 8 on-chip units, each 2 GiB

Intel Xeon Phi (Knights Landing) – Architecture

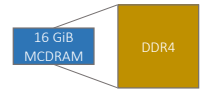


Memory Modes:

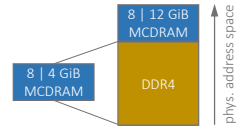
- Flat-Mode
  DDR4 and MCDRAM in same address space



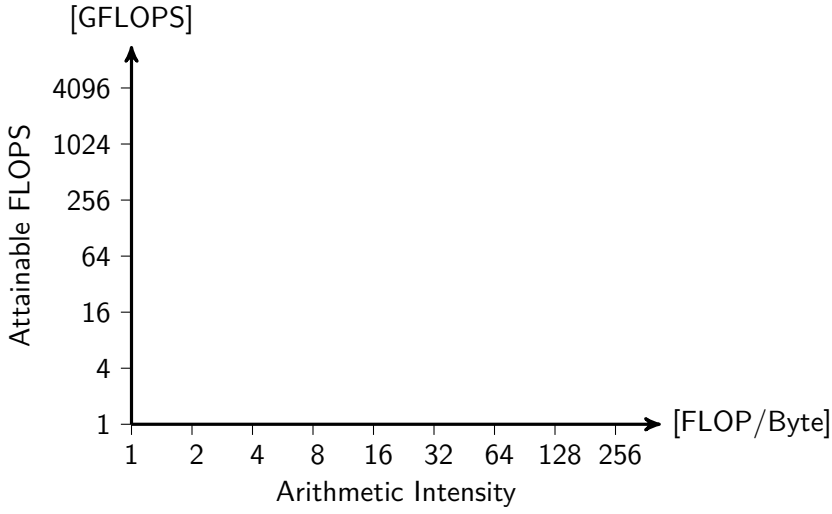
- Cache-Mode
  MCDRAM = direct-mapped Cache for DDR4



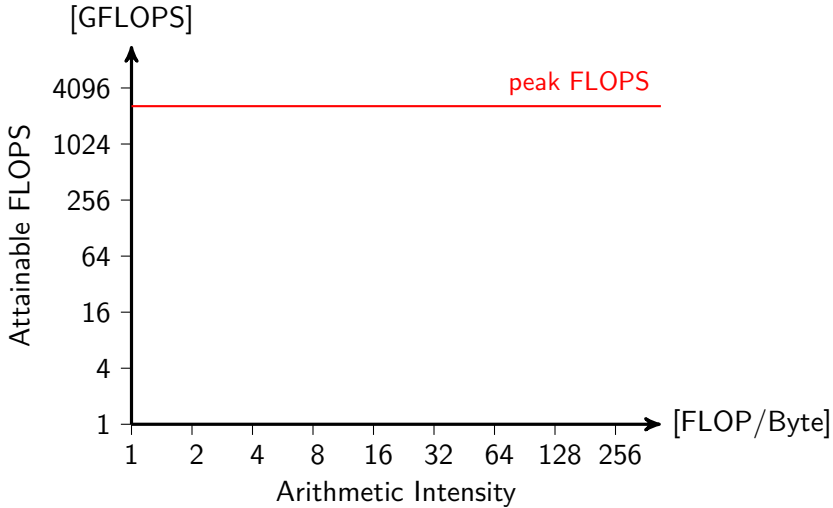
- Hybrid-Mode
  8 | 4 GiB MCDRAM in Cache-Mode, remainder in Flat-Mode



KNL in the Roofline Model (Samuel Williams, 2008)

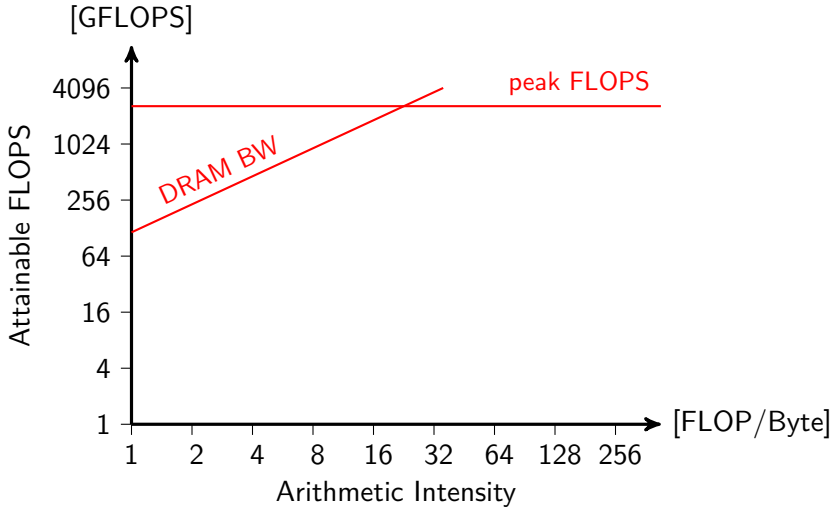


KNL in the Roofline Model - adding peak FLOPS



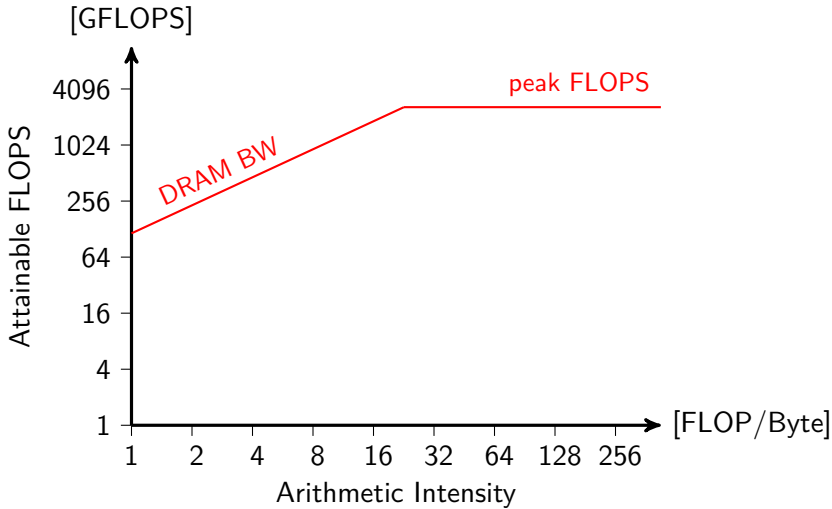
Numbers for Intel Xeon Phi 7250: 2611.2 GFLOPS, 115.2 GiB/s

KNL in the Roofline Model - adding peak DRAM bandwidth



Numbers for Intel Xeon Phi 7250: 2611.2 GFLOPS, 115.2 GiB/s

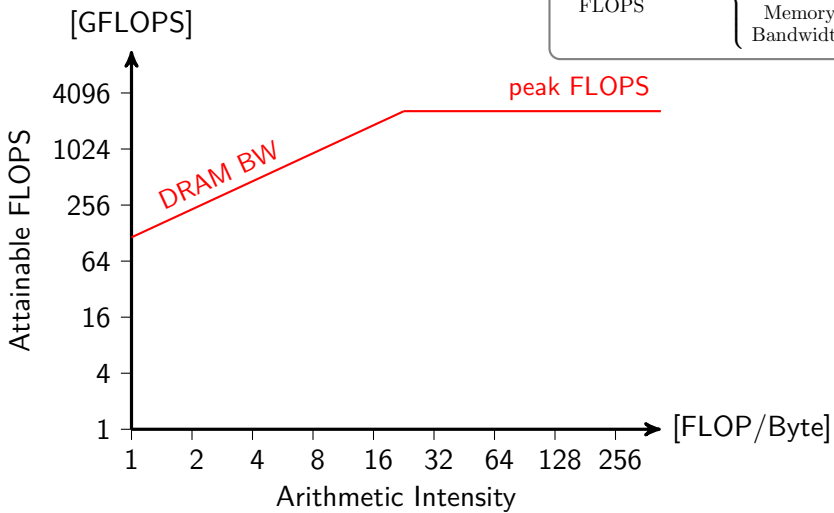
KNL in the Roofline Model



Numbers for Intel Xeon Phi 7250: 2611.2 GFLOPS, 115.2 GiB/s

KNL in the Roofline Model

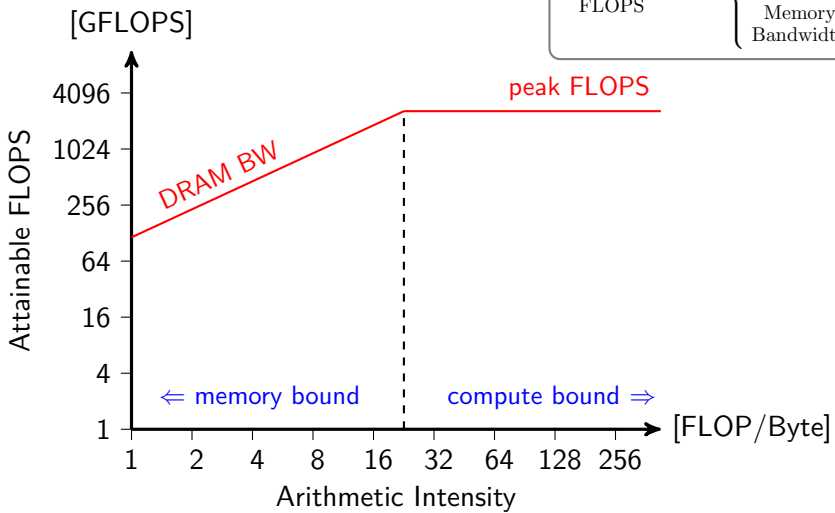
$$\text{Attainable FLOPS} = \min \begin{cases} \text{Peak FLOPS} \\ \text{Memory Bandwidth} \times \text{Arithmetic Intensity} \end{cases}$$



Numbers for Intel Xeon Phi 7250: 2611.2 GFLOPS, 115.2 GiB/s

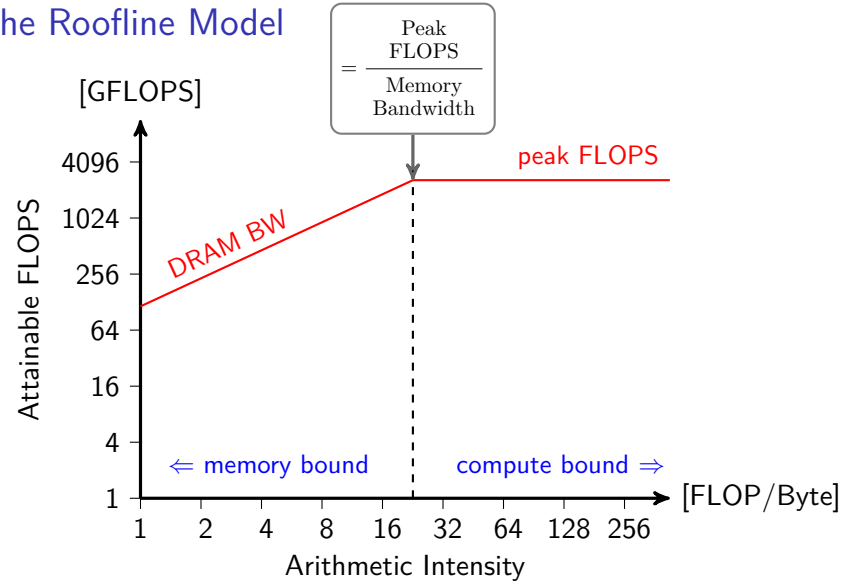
KNL in the Roofline Model

$$\text{Attainable FLOPS} = \min \begin{cases} \text{Peak FLOPS} \\ \text{Memory Bandwidth} \times \text{Arithmetic Intensity} \end{cases}$$

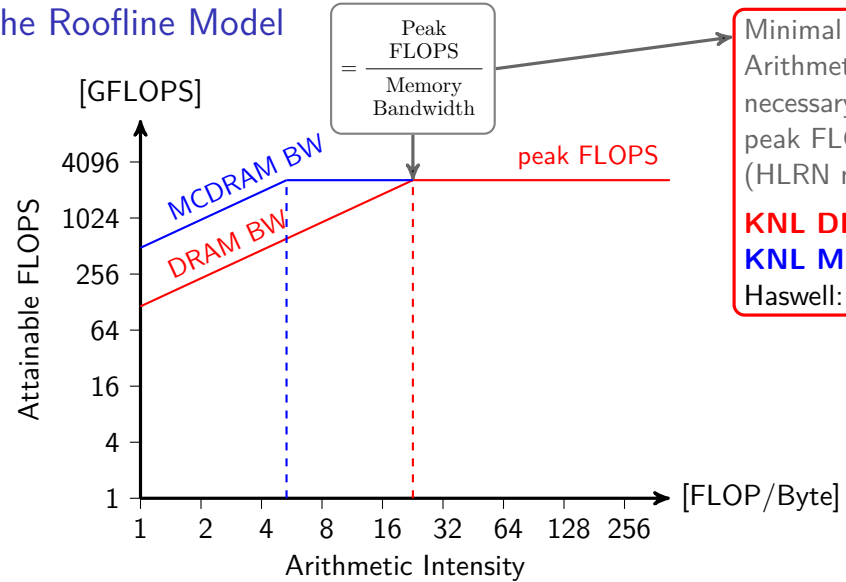


Numbers for Intel Xeon Phi 7250: 2611.2 GFLOPS, 115.2 GiB/s

KNL in the Roofline Model



KNL in the Roofline Model



Refining the Ceilings - FLOPS

Realistic peak FLOPS

- Xeon Phi 7250 (HLRN Cray TDS), advertised with 3.05 TFLOPS peak DP

Refining the Ceilings - FLOPS

Realistic peak FLOPS

- Xeon Phi 7250 (HLRN Cray TDS), advertised with 3.05 TFLOPS peak DP
  - 1.4 GHz $\times$ 68 core $\times$ 8 SIMD $\times$ 2 VPUs $\times$ 2 FMA $=$ 3046.4 GFLOPS

Refining the Ceilings - FLOPS

Realistic peak FLOPS

- Xeon Phi 7250 (HLRN Cray TDS), advertised with 3.05 TFLOPS peak DP
  - 1.4 GHz $\times$ 68 core $\times$ 8 SIMD $\times$ 2 VPUs $\times$ 2 FMA $=$ 3046.4 GFLOPS
  - AVX frequency is only 1.2 GHz and might throttle down under heavy load

Refining the Ceilings - FLOPS

Realistic peak FLOPS

- Xeon Phi 7250 (HLRN Cray TDS), advertised with 3.05 TFLOPS peak DP
  - 1.4 GHz $\times$ 68 core $\times$ 8 SIMD $\times$ 2 VPUs $\times$ 2 FMA $=$ 3046.4 GFLOPS
  - AVX frequency is only 1.2 GHz and might throttle down under heavy load

  $\Rightarrow$ **actual peak: 2611.2 GFLOPS**

Refining the Ceilings - FLOPS

Realistic peak FLOPS

- Xeon Phi 7250 (HLRN Cray TDS), advertised with 3.05 TFLOPS peak DP
  - 1.4 GHz $\times$ 68 core $\times$ 8 SIMD $\times$ 2 VPUs $\times$ 2 FMA = 3046.4 GFLOPS
  - AVX frequency is only 1.2 GHz and might throttle down under heavy load

  $\Rightarrow$ **actual peak: 2611.2 GFLOPS**

Add more FLOPS ceilings

- without instruction level parallelism (ILP), i.e. dual VPUs and FMA
  - 1.2 GHz $\times$ 68 core $\times$ 8 SIMD = **652.8 GFLOPS** (25 %)

Refining the Ceilings - FLOPS

Realistic peak FLOPS

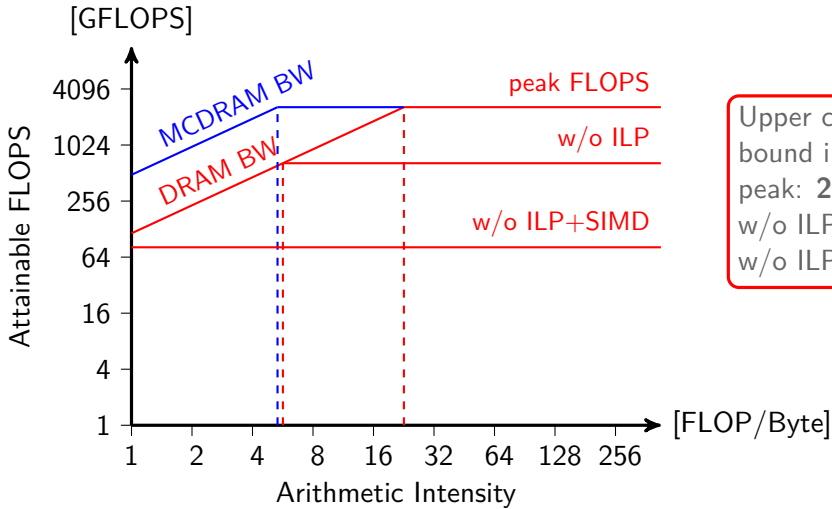
- Xeon Phi 7250 (HLRN Cray TDS), advertised with 3.05 TFLOPS peak DP
  - 1.4 GHz $\times$ 68 core $\times$ 8 SIMD $\times$ 2 VPUs $\times$ 2 FMA $=$ 3046.4 GFLOPS
  - AVX frequency is only 1.2 GHz and might throttle down under heavy load

  $\Rightarrow$ **actual peak: 2611.2 GFLOPS**

Add more FLOPS ceilings

- without instruction level parallelism (ILP), i.e. dual VPUs and FMA
  - 1.2 GHz $\times$ 68 core $\times$ 8 SIMD $=$ **652.8 GFLOPS** (25 %)
- without ILP, and without SIMD
  - 1.2 GHz $\times$ 68 core $=$ **84.6 GFLOPS** (3.2 %)

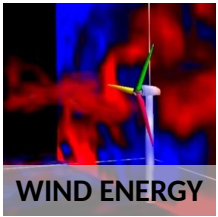
KNL in the Roofline Model

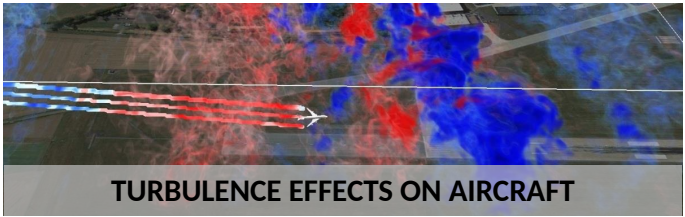


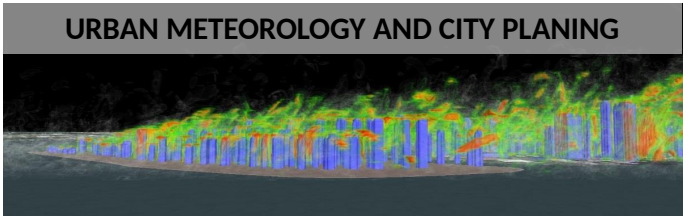
Upper compute bound in GFLOPS:
peak: **2611.2**
w/o ILP: **652.8**
w/o ILP+SIMD: **84.6**

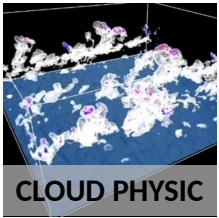
Numbers for Intel Xeon Phi 7250: 2611.2 GFLOPS, 115.2 GiB/s

Case study I: Atmospheric Research with PALM



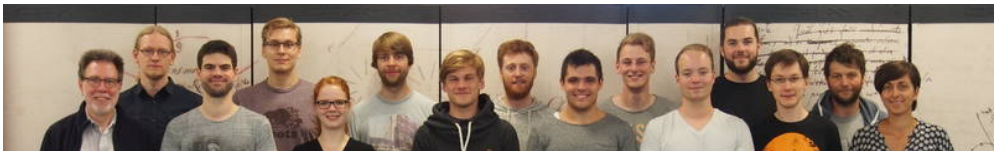






The PALM Code

- continuously developed **since 1997** by the PALM group (Siegfried Raasch et al.)



Leibniz Universität Hannover

The PALM Code

- continuously developed **since 1997** by the PALM group (Siegfried Raasch et al.)

- **Fortran** 95/2003.
- hybrid **MPI + OpenMP** code
- **140 kLOC**, 79 modules and 171 source files
- highly scalable, tested for up to 43,200 cores

Leibniz Universität Hannover

The PALM Code

- continuously developed **since 1997** by the PALM group (Siegfried Raasch et al.)

- **Fortran** 95/2003.
- hybrid **MPI + OpenMP** code
- **140 kLOC**, 79 modules and 171 source files
- highly scalable, tested for up to 43,200 cores

- runs on the HLRN supercomputing facilities at Berlin (ZIB) and Hannover (LUIS)
- modernisation target within the **Intel Parallel Computing Center at ZIB**

PALM Hackathon at ZIB

left to right: Matthias Noack, Florian Wende, Helge Knoop, Matthias Sühring, Tobias Gronemeier; behind the camera: Thomas Steinke

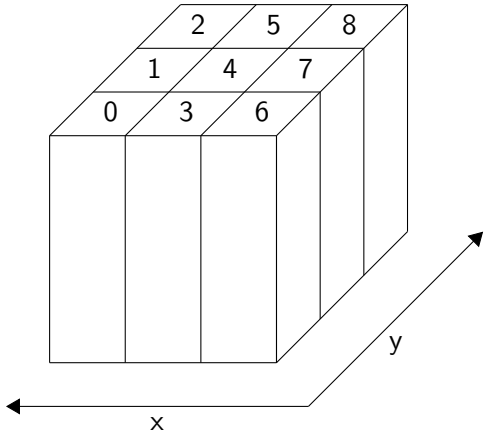
Parallelization Strategy

Parallelization Strategy

MPI

OpenMP

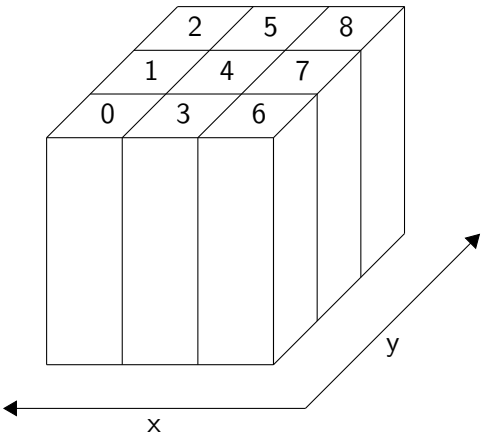
2D domain decomposition



Parallelization Strategy

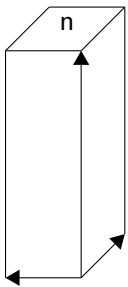
MPI

2D domain decomposition



OpenMP

outer of 3 nested loops threaded



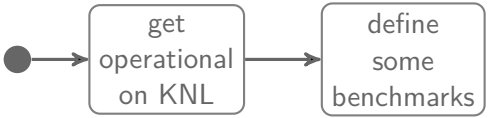
- different variants for different targets
- e.g. cache optimised with decomposed inner loop
- **no vectorisation**
  - ⇒ use of **floating point exceptions** prevents automatic vectorisation
  - ⇒ no explicit SIMD constructs

KNL Optimisation Strategy

KNL Optimisation Strategy

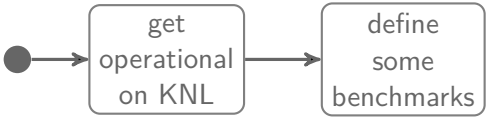
- build scripts
- job scripts
- bug fixing

KNL Optimisation Strategy



- build scripts
- job scripts
- bug fixing

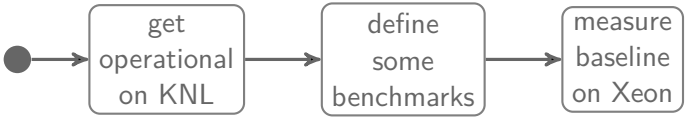
KNL Optimisation Strategy



- build scripts
- job scripts
- bug fixing

- small
- medium
- large

KNL Optimisation Strategy



- build scripts
- job scripts
- bug fixing

- small
- medium
- large

Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes (Konrad)

- 2 $\times$ Intel **Xeon** E5-2680v3 (Haswell)
  - 2 $\times$ 12 cores at 2.5 GHz

  $\Rightarrow$ **960 GFLOPS** per node
  - 64 GiB DDR3, **136 GiB/s**

Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes (Konrad)

- 2 × Intel **Xeon** E5-2680v3 (Haswell)
  - 2 × 12 cores at 2.5 GHz
  - ⇒ **960 GFLOPS** per node
  - 64 GiB DDR3, **136 GiB/s**

Haswell Results

	nodes	runtime
small	4	131.3 s
medium	8	147.6 s
large	16	164.6 s

Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes (Konrad)

- 2 × Intel **Xeon** E5-2680v3 (Haswell)
  - 2 × 12 cores at 2.5 GHz
  - ⇒ **960 GFLOPS** per node
  - 64 GiB DDR3, **136 GiB/s**

HLRN KNL TDS node, 80 nodes

- 1 × Intel **Xeon Phi** 7250 (KNL)
  - 68 cores at 1.4 GHz
  - ⇒ **2611.2 GFLOPS** with AVX clock "3.05 TFLOPS"
  - 96 GiB DDR4, **115.2 GiB/s**
  - 16 GiB **MCDRAM**, **490 GiB/s**

Haswell Results

	nodes	runtime
small	4	131.3 s
medium	8	147.6 s
large	16	164.6 s

Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes (Konrad)

- 2 × Intel **Xeon** E5-2680v3 (Haswell)
  - 2 × 12 cores at 2.5 GHz
  - ⇒ **960 GFLOPS** per node
  - 64 GiB DDR3, **136 GiB/s**

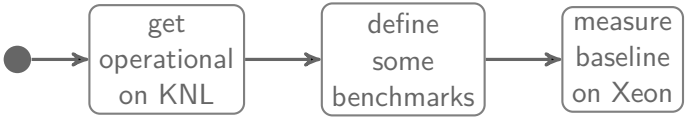
HLRN KNL TDS node, 80 nodes

- 1 × Intel **Xeon Phi** 7250 (KNL)
  - 68 cores at 1.4 GHz
  - ⇒ **2611.2 GFLOPS** with AVX clock "3.05 TFLOPS"
  - 96 GiB DDR4, **115.2 GiB/s**
  - 16 GiB **MCDRAM**, **490 GiB/s**
- Upper bounds for speed-up:
  - ⇒ compute bound: **2.7**×
  - ⇒ memory bound: **3.6**× (MCDRAM)

Haswell Results

	nodes	runtime
small	4	131.3 s
medium	8	147.6 s
large	16	164.6 s

KNL Optimisation Strategy

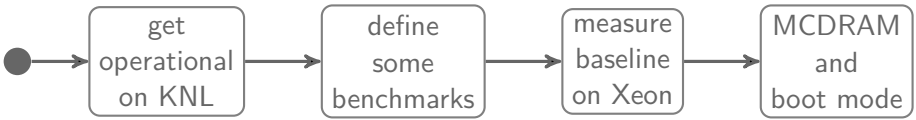


- build scripts
- job scripts
- bug fixing

- small
- medium
- large

✓

KNL Optimisation Strategy

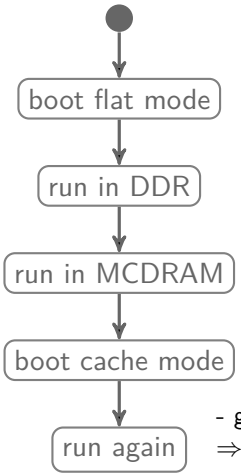


- build scripts
- job scripts
- bug fixing

- small
- medium
- large

✓

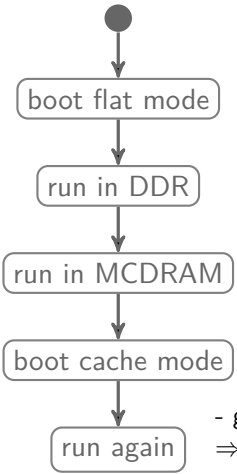
MCDRAM Usage and Boot Mode



- problem $<$ 16 GiB
- upper bound for gain from MCDRAM

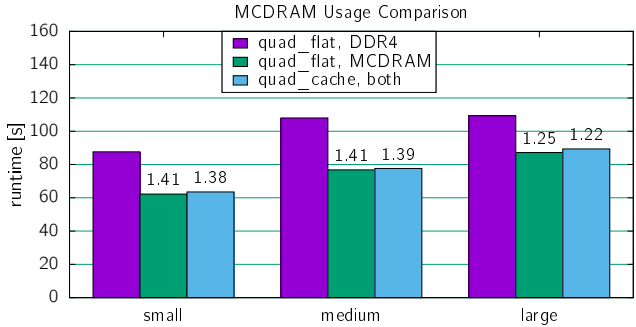
- good enough?
$\Rightarrow$ decide about explicit placement

MCDRAM Usage and Boot Mode



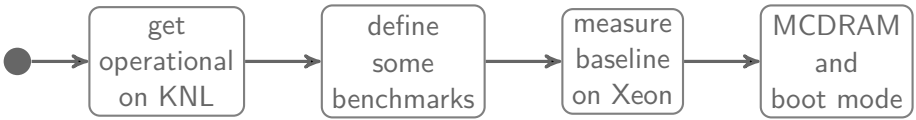
- problem $<$ 16 GiB
- upper bound for gain from MCDRAM

- good enough?
$\Rightarrow$ decide about explicit placement



- 25 - 41 % gain from MCDRAM
- $\leq$ 3 % loss from Cache-Mode
  $\Rightarrow$ no need for explicit placement
- 2 MiB pages worked best

KNL Optimisation Strategy



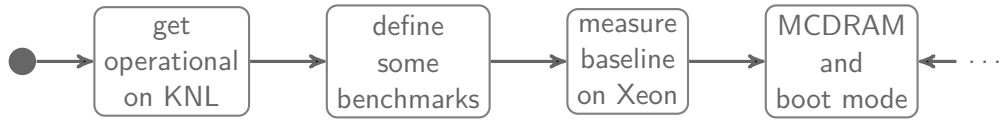
- build scripts
- job scripts
- bug fixing

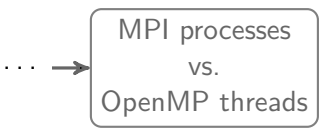
- small
- medium
- large

✓

⇒ quad_cache

KNL Optimisation Strategy





MPI processes vs. OpenMP threads

Tuning run for optimal per-node config

ranks	64	32	16	8	4	2	1
threads	1	2	4	8	16	32	64
small			X				
medium				X			
large		X					

≤ 2 % off from best

≤ 15 % off from best

worse

X fastest

MPI processes vs. OpenMP threads

Tuning run for optimal per-node config

ranks	64	32	16	8	4	2	1
threads	1	2	4	8	16	32	64
small			X				
medium				X			
large		X					

≤ 2 % off from best

≤ 15 % off from best

worse

X fastest

Conclusion

- fastest
  - ⇒ small: 16 ranks × 4 thread
  - ⇒ medium: 8 ranks × 8 threads
  - ⇒ large: 32 × 2 threads
- 16 × 4 performs for all setups
- fastest vs. slowest config: **2.3**×

MPI processes vs. OpenMP threads

Tuning run for optimal per-node config

ranks	64	32	16	8	4	2	1
threads	1	2	4	8	16	32	64
small			X				
medium				X			
large		X					

≤ 2 % off from best

≤ 15 % off from best

worse

X fastest

Conclusion

- fastest
  - ⇒ small: 16 ranks × 4 thread
  - ⇒ medium: 8 ranks × 8 threads
  - ⇒ large: 32 × 2 threads
- 16 × 4 performs for all setups
- fastest vs. slowest config: **2.3**×
- very low impact on speedups between MCDRAM usage models
  - absolute numbers vary largely
  - ⇒ do memory first

MPI processes vs. OpenMP threads

Tuning run for optimal per-node config

ranks	64	32	16	8	4	2	1
threads	1	2	4	8	16	32	64
small			X				
medium				X			
large		X					

≤ 2 % off from best

≤ 15 % off from best

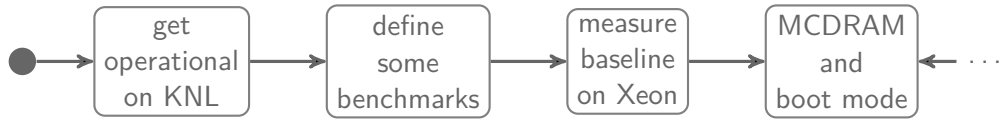
worse

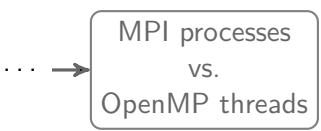
X fastest

Conclusion

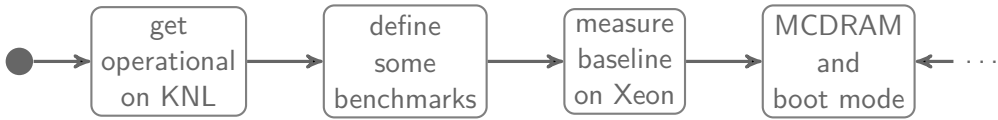
- fastest
  - ⇒ small: 16 ranks × 4 thread
  - ⇒ medium: 8 ranks × 8 threads
  - ⇒ large: 32 × 2 threads
- 16 × 4 performs for all setups
- fastest vs. slowest config: **2.3**×
- very low impact on speedups between MCDRAM usage models
  - absolute numbers vary largely
  - ⇒ do memory first
- always check pinning/affinity

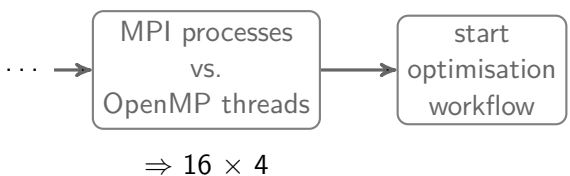
KNL Optimisation Strategy





KNL Optimisation Strategy





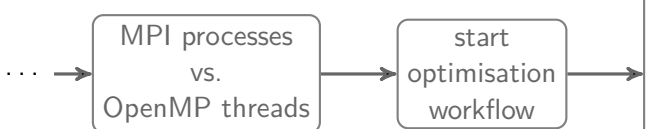
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing

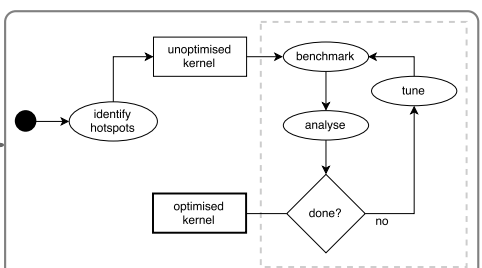
- small
- medium
- large

✓

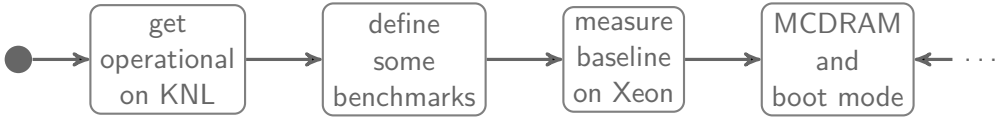
⇒ quad_cache



⇒ 16×4



KNL Optimisation Strategy

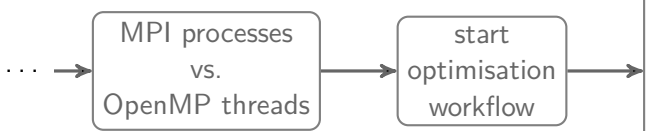


- build scripts
- job scripts
- bug fixing

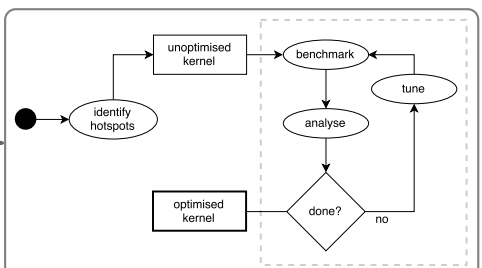
- small
- medium
- large

✓

⇒ quad_cache



⇒ 16×4



- compiler optimisation reports
- Intel VTune Amplifier XE, Advisor XE

First Code Changes

Floating point exception-handling

- prevents vectorisation
- **fp-model-strict** $\rightarrow$ **fp-model-source**
- remove exception handling
- add NaN/Inf-tests
  - $\Rightarrow$ when writing checkpoints

$\Rightarrow$ no significant auto vectorisation

$\Rightarrow$ suboptimal memory layout

First Code Changes

Floating point exception-handling

- prevents vectorisation
- **fp-model-strict** $\rightarrow$ **fp-model-source**
- remove exception handling
- add NaN/Inf-tests
  - $\Rightarrow$ when writing checkpoints

$\Rightarrow$ no significant auto vectorisation

$\Rightarrow$ suboptimal memory layout

Using MKL FFT

- small gain for benchmarks

$\Rightarrow$ might be significant for larger setups

First Code Changes

Floating point exception-handling

- prevents vectorisation
- **fp-model-strict** $\rightarrow$ **fp-model-source**
- remove exception handling
- add NaN/Inf-tests
  - $\Rightarrow$ when writing checkpoints

$\Rightarrow$ no significant auto vectorisation

$\Rightarrow$ suboptimal memory layout

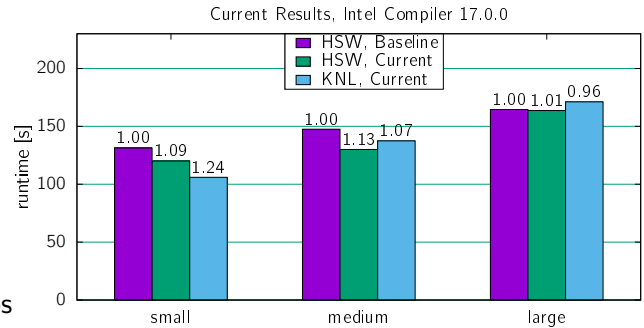
Using MKL FFT

- small gain for benchmarks

$\Rightarrow$ might be significant for larger setups

CONTIGUOUS keyword

- from Fortran 2008
- tell the compiler about contiguously allocated arrays



Cray Specialities

Hardware/Software

- Cray Aries network
- Cray MPI
- Cray Performance Tools
- Cray Compiler

Cray Specialities

Hardware/Software

- Cray Aries network
- Cray MPI
- Cray Performance Tools
- Cray Compiler

Rank Reordering

- instrumented application run
- optimised mapping of MPI ranks to cores and nodes

$\Rightarrow$ no improvement on KNL

Cray Specialities

Hardware/Software

- Cray Aries network
- Cray MPI
- Cray Performance Tools
- Cray Compiler

Rank Reordering

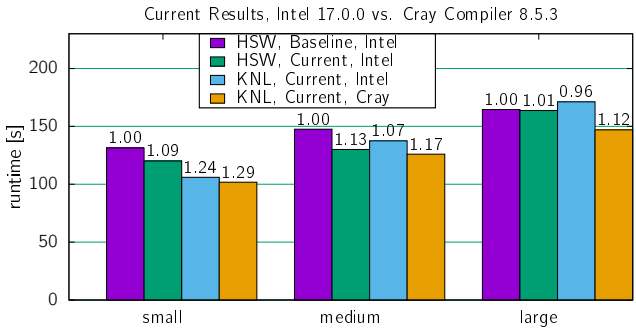
- instrumented application run
- optimised mapping of MPI ranks to cores and nodes

$\Rightarrow$ no improvement on KNL

Cray Compiler

- initial KNL support
- crash with OpenMP

$\Rightarrow$ 64 MPI ranks per KNL

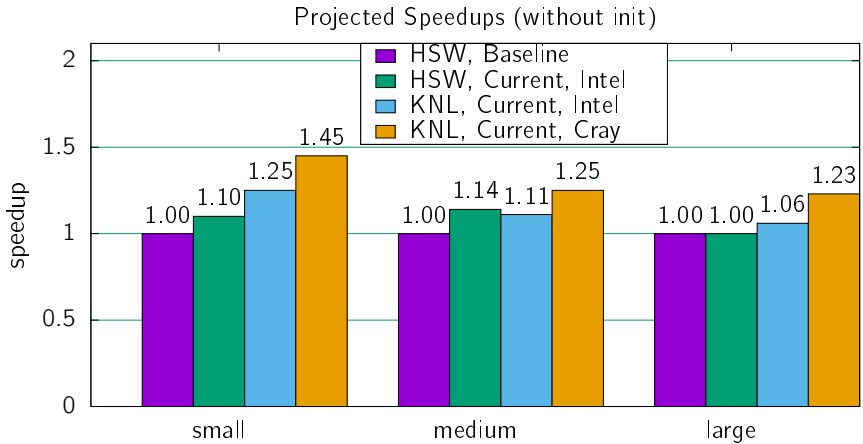


Projected Production Run Performance

- benchmark runs: $\approx$ **5 min**, productions runs: $\approx$ **12 hours**
  - $\Rightarrow$ serial initialisation becomes negligible
  - $\Rightarrow$ plot speedup based on $t_{total} - t_{init}$

Projected Production Run Performance

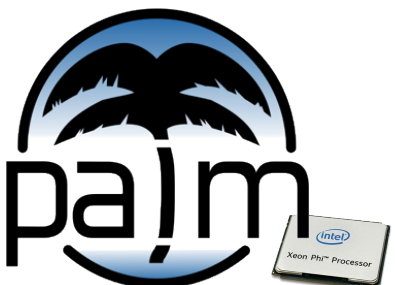
- benchmark runs: $\approx$ **5 min**, productions runs: $\approx$ **12 hours**
  - $\Rightarrow$ serial initialisation becomes negligible
  - $\Rightarrow$ plot speedup based on $t_{total} - t_{init}$



PALM - Final Remarks

Bottom Line

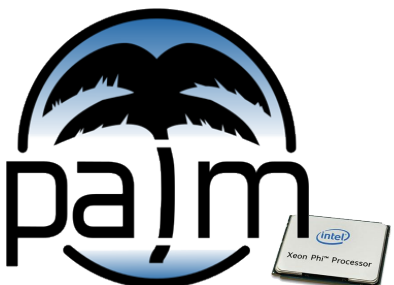
- getting started on KNL was easy
  - $\Rightarrow$ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  - $\Rightarrow$ initialisation
  - $\Rightarrow$ ...



PALM - Final Remarks

Bottom Line

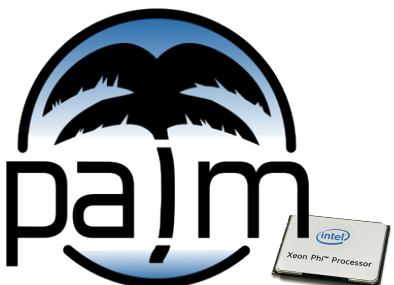
- getting started on KNL was easy
  - ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  - ⇒ initialisation
  - ⇒ ...
- Cray Fortran compiler up to **16.5 %** faster than Intel on KNL



PALM - Final Remarks

Bottom Line

- getting started on KNL was easy
  - ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  - ⇒ initialisation
  - ⇒ ...
- Cray Fortran compiler up to **16.5 %** faster than Intel on KNL
- speedup over dual HSW HLRN node:
  - benchmark: up to **1.29**×
  - production (projected): up to **1.45**×
  - ⇒ even without vectorisation



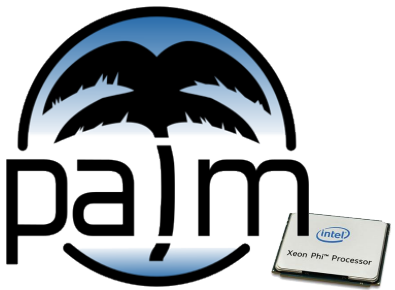
PALM - Final Remarks

Bottom Line

- getting started on KNL was easy
  - ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  - ⇒ initialisation
  - ⇒ ...
- Cray Fortran compiler up to **16.5 %** faster than Intel on KNL
- speedup over dual HSW HLRN node:
  - benchmark: up to **1.29**×
  - production (projected): up to **1.45**×
  - ⇒ even without vectorisation

What's next

- VTune Results:
  - ⇒ increase concurrency
  - ⇒ reduce L2 misses on KNL
- adapt data layout for SIMD
  - ⇒ twice the potential on KNL



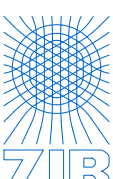
Case study II: Material Science with VASP

VASP – Vienna Ab-Initio Simulation Package

- Plan wave electronic structure code to model atomic scale materials from first principals: $H\psi = E\psi$
- widely used in material sciences
- historically grown, large MPI-only Fortran code base

- different approximations (DFT, hybrid functionals, ...) to tackle the physics
- library dependencies: FFTW, BLAS, scaLAPACK, ELPA

Collaboration of:



SIMD Introduction

SIMD – Single Instruction Multiple Data

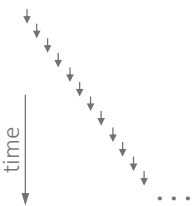
- Multiple words are processed at once sharing 1 program counter
- SIMD registers become increasingly larger: currently 512 bit with AVX-512
  - Slightly increased logic on the chip, but heavily increased arithmetic throughput
  - Xeon Phi KNL w/ and w/o SIMD: **3 TFLOPS** vs. **0.37 TFLOPS**

SIMD Introduction

SIMD – Single Instruction Multiple Data

- Multiple words are processed at once sharing 1 program counter

```
for (i = 0; i < N; ++i)
  y[i] = log(x[i]);
```



SIMD Introduction

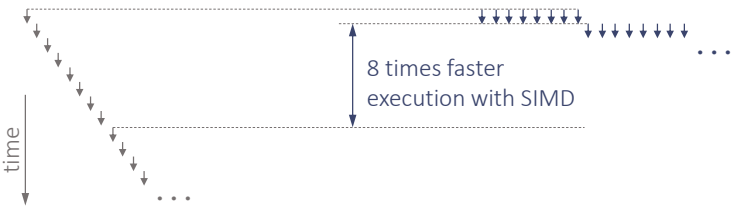
SIMD – Single Instruction Multiple Data

- Multiple words are processed at once sharing 1 program counter

```
for (i = 0; i < N; ++i)
  y[i] = log(x[i]);
```

```
for (i = 0; i < N; i += 8)
  y[i + 0] = log(x[i + 0]);
             ...
  y[i + 7] = log(x[i + 7]);
```

y[i] = vlog(x[i])



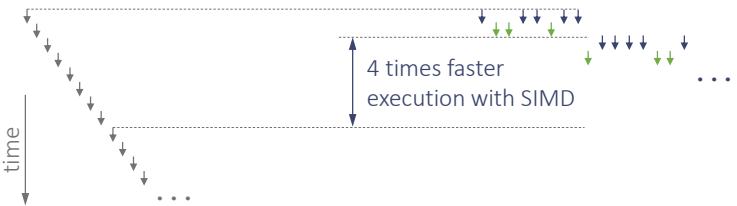
SIMD Introduction

SIMD – Single Instruction Multiple Data

- Multiple words are processed at once sharing 1 program counter
- Control flow divergences can hurt SIMD performance significantly

```
for (i = 0; i < N; ++i)
  if (p[i]) y[i] = log(x[i]);
  else y[i] = exp(x[i]);
```

```
for (i = 0; i < N; i += 8)
  if (m ← p[i]) y[i] = vlog_mask(y[i], m, x[i]);
  else y[i] = vexp_mask(y[i], ~m, x[i]);
```



SIMD Vectorisation in VASP

OpenMP 4.x compiler directives

- portability across compilers
- low code invasiveness
- no SIMD intrinsics for Fortran

- combine OpenMP 4.x SIMD with “high-level vectors” (loop chunking) to increase flexibility and expressiveness

SIMD Vectorisation in VASP - Example

Non-vectorizable loop split into parts to enable SIMD vectorization

```
idx = 0;
for (i = 0; i < ni; ++i) {
  while (“some condition”)
    ++idx;
  d = data[idx];
  for (j = 0; j < nj; ++j)
    res[j] += d * (...);
}
```

C-version of the Codes
(not optimized)

SIMD Vectorisation in VASP - Example

Non-vectorizable loop split into parts to enable SIMD vectorization

```
idx = 0;
for (i = 0; i < ni; ++i) {
  while (“some condition”)
    ++idx;
  d = data[idx];
  for (j = 0; j < nj; ++j)
    res[j] += d * (...);
}
```

`nj` rather small: not a candidate for SIMD vectorization

SIMD Vectorisation in VASP - Example

Non-vectorizable loop split into parts to enable SIMD vectorization

```
idx = 0;
for (i = 0; i < ni; ++i) {
  while (“some condition”)
    ++idx;
  d = data[idx];
  for (j = 0; j < nj; ++j)
    res[j] += d * (...);
}
```

loop iterations are not independent: **idx**

SIMD Vectorisation in VASP - Example

Non-vectorizable loop split into parts to enable SIMD vectorization

```
idx = 0;
for (i = 0; i < ni; ++i) {
  while (“some condition”)
    ++idx;
  d = data[idx];
  for (j = 0; j < nj; ++j)
    res[j] += d * (...);
}
```

Loop-Chunking, e.g. CHUNKSIZE=32

```
idx = 0;
for (i = 0; i < ni; i += CHUNKSIZE) {
  ii_max = min(CHUNKSIZE, ni - i);
  for (ii = 0; ii < ii_max; ++ii) {
    while (“some condition”)
      ++idx;
    vidx[ii] = idx;
  }
  ...
}
```

Compute `idx`-values in advance to enable SIMD vectorization afterwards!

SIMD Vectorisation in VASP - Example

Non-vectorizable loop split into parts to enable SIMD vectorization

```
idx = 0;
for (i = 0; i < ni; ++i) {
  while (“some condition”)
    ++idx;
  d = data[idx];
  for (j = 0; j < nj; ++j)
    res[j] += d * (...);
}
```

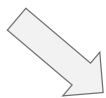
```
idx = 0;
for (i = 0; i < ni; i += CHUNKSIZE) {
  ii_max = min(CHUNKSIZE, ni – i);
  for (ii = 0; ii < ii_max; ++ii) {
    while (“some condition”)
      ++idx;
    vidx[ii] = idx;
  }
  for (ii = 0; ii < ii_max; ++ii)
    vd[ii] = data[vidx[ii]];
  ...
}
```

Load data in a separate loop: leave it to the compiler to vectorize or not

SIMD Vectorisation in VASP - Example

Non-vectorizable loop split into parts to enable SIMD vectorization

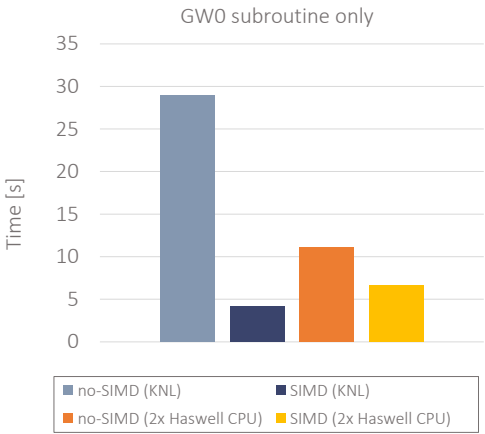
```
idx = 0;
for (i = 0; i < ni; ++i) {
  while (“some condition”)
    ++idx;
  d = data[idx];
  for (j = 0; j < nj; ++j)
    res[j] += d * (...);
}
```

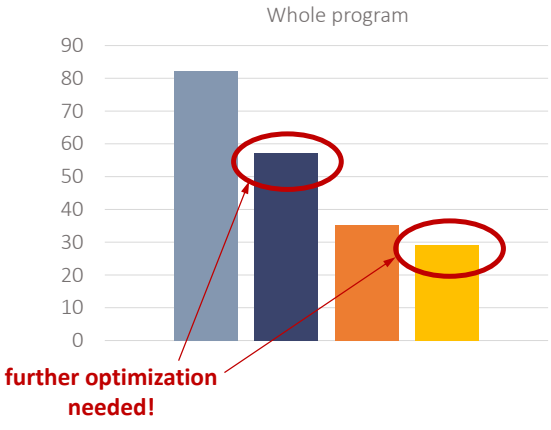


```
idx = 0;
for (i = 0; i < ni; i += CHUNKSIZE) {
  ii_max = min(CHUNKSIZE, ni - i);
  for (ii = 0; ii < ii_max; ++ii) {
    while (“some condition”)
      ++idx;
    vidx[ii] = idx;
  }
  for (ii = 0; ii < ii_max; ++ii)
    vd[ii] = data[vidx[ii]];
  for (j = 0; j < nj; ++j)
    #pragma omp simd
    for (ii = 0; ii < ii_max; ++ii)
      res[j] += vd[ii] * (...);
}
```

SIMD Vectorisation in VASP - Results

Non-vectorizable loop split into parts to enable SIMD vectorization





Xeon Phi nodes: quadrant mode, all data in MCDRAM

KNL Summary

Xeon Phi (KNL) has a low entry barrier...

- no offloading
- well-known CPU toolchains and workflows

KNL Summary

Xeon Phi (KNL) has a low entry barrier...

- no offloading
- well-known CPU toolchains and workflows

...but getting performance is challenging

- ease of use can be misleading towards quick fixes
- code needs to be re-thought and re-written for SIMD
- effort pays off with significant speed-up for hot-spots
  $\Rightarrow$ Xeon benefits from Xeon Phi optimisations as well
- overall application performance suffers from low single-thread performance
  $\Rightarrow$ working on a few hotspots is not sufficient

KNL Summary

Xeon Phi (KNL) has a low entry barrier...

- no offloading
- well-known CPU toolchains and workflows

...but getting performance is challenging

- ease of use can be misleading towards quick fixes
- code needs to be re-thought and re-written for SIMD
- effort pays off with significant speed-up for hot-spots
  $\Rightarrow$ Xeon benefits from Xeon Phi optimisations as well
- overall application performance suffers from low single-thread performance
  $\Rightarrow$ working on a few hotspots is not sufficient

With AVX-512 in Xeon and Xeon Phi, SIMD can no longer be neglected.

Overall Conclusions

A deep knowledge of each hardware platform is necessary to fully exploit its computing power.

Overall Conclusions

A deep knowledge of each hardware platform is necessary to fully exploit its computing power.

Code modernisation for KNL within IPCCs world-wide is just one example of the effort it takes to keep the huge amount of legacy code in HPC usable.

Overall Conclusions

A deep knowledge of each hardware platform is necessary to fully exploit its computing power.

Code modernisation for KNL within IPCCs world-wide is just one example of the effort it takes to keep the huge amount of legacy code in HPC usable.

Within the more and more diverse HPC hardware landscape, larger shares of HPC centre's budgets will have to be allocated for code modernisation work in order to utilise future machines efficiently.

Overall Conclusions

A deep knowledge of each hardware platform is necessary to fully exploit its computing power.

Code modernisation for KNL within IPCCs world-wide is just one example of the effort it takes to keep the huge amount of legacy code in HPC usable.

Within the more and more diverse HPC hardware landscape, larger shares of HPC centre's budgets will have to be allocated for code modernisation work in order to utilise future machines efficiently.

- EoP