# Table of Contents

**Preface**  
4

**Impressions**  
5

**Agenda**  
6

**Acknowledgements**  
8

**Part 1: Lecture Notes**  
9

- Biographies of the tutors  
10
- Welcome & Introduction (Volker Weinberg, LRZ)  
11
- Hardware Overview & Native Execution (Momme Allalen, LRZ)  
29
- KNL MCDRAM and KNC Offloading (Volker Weinberg, LRZ)  
46
- MKL (Momme Allalen, LRZ)  
110
- Vectorisation & Basic Performance Overview (Momme Allalen, LRZ)  
126
- KNL Optimization (Fabio Baruffa, LRZ)  
161
- KNL Tools (Fabio Baruffa, LRZ)  
200
- Many-core Programming with OpenMP* 4.x (Michael Klemm, Intel)  
217
- Advanced MIC Programming (Jan Eitzinger, RRZE)  
244

**Part 2: Plenary Session**  
274

- Biographies of the invited speakers  
275
- Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures (Luigi Iapichino, IPCC@LRZ)  
277
- Extreme-scale Multi-physics Simulation of the 2004 Sumatra Earthquake (Michael Bader, IPCC@TUM) (slides not available yet)  
288
- Development of Intel Xeon Phi Accelerated Algorithms and Applications at IT4I (Vít Vondrák, IPCC@IT4I)  
295
- Application Show Cases on Intel® Xeon Phi™ Processors (Michael Klemm, Intel)  
309
- Evaluation of Intel Xeon Phi "Knights Landing": Initial impressions and benchmarking results (Jan Eitzinger, RRZE)  
317
- Lattice Quantum Chromodynamics on the MIC architectures (Piotr Korcyl, University of Regensburg)  
330
- The experience of the HLST on Europes biggest KNL cluster (Nils Moschüring, IPP)  
345
- Porting the ELPA library to the KNL architecture (Andreas Marek, Max Planck Computing and Data Facility (MPCDF))  

**Part 3: Hands-On Sessions**  
367
Preface

The Leibniz Supercomputing Centre publishes in this booklet the complete material of the Intel MIC programming workshop that took place at LRZ on June 26 – 28, 2017. The workshop discussed Intel’s Many Integrated Core (MIC) architecture and various programming models for Intel Xeon Phi co-/processors. The workshop covered a wide range of topics from the description of the hardware of the Intel Xeon Phi co-/processors through information about the basic programming models as well as information about vectorisation and MCDRAM usage up to tools and strategies how to analyse and improve the performance of applications. The workshop mainly concentrated on techniques relevant for Knights Landing (KNL) based systems. During a plenary session on the last day 8 invited speakers from IPCC@LRZ, IPCC@TUM, IPCC@IT4Innovations, Intel, RRZE, the University of Regensburg, IPP and MPCDF talked about Intel Xeon Phi experience and best practice recommendations. Hands-on sessions were done on the Knights Corner (KNC) based system SuperMIC and two KNL test systems at LRZ.

The workshop was organised as a PRACE Advanced Training Centre (PATC) event by the Czech-Bavarian Competence Centre for Supercomputing Applications (CzeBaCCA) and was combined with a two-day symposium on "HPC for natural hazard assessment and disaster mitigation". The Czech-Bavarian Competence Centre was established in 2016 by the Leibniz Supercomputing Centre (LRZ), the Department of Informatics of the Technical University of Munich (TUM) and the IT4Innovations National Supercomputing Centre of the Czech Republic to foster the Czech-German collaboration in high performance computing. One of the main objectives of the Competence Centre is to organise a series of Intel Xeon Phi specific workshops combined with scientific symposia on topics like optimisation of simulation codes in environmental science.

The successful series of workshops was initiated in February 2016 with an introductory Intel MIC programming workshop concentrating on the Salomon supercomputer at IT4I, the largest European KNC based system, combined with a symposium on “SeisMIC – Seismic Simulation on Current and Future Supercomputers” at IT4Innovations (see inSiDE Vol. 14 No. 1 p. 76ff, 2016). The series was continued with an extended PATC Intel MIC programming workshop concentrating on simulations on the Intel Xeon Phi based SuperMIC cluster at LRZ combined with a scientific workshop on “High Performance Computing for Water Related Hazards” in June 2016 at LRZ (see inSiDE Vol. 14 No. 2 p. 25ff, 2016). The third edition of this workshop series took place on February 7 – 9, 2017 at IT4Innovations, again, and combined a two-day Intel MIC programming workshop with a one-day scientific workshop on “High performance computing in atmosphere modelling and air related environmental hazards” (see inSiDE Vol. 15 No. 1 p. 48ff, 2017).

The present booklet provides the complete material of the 4th Intel MIC programming workshop. Part 1 covers the lectures given during the first two and a half days, part 2 presents the slides of the invited talks of the public plenary session on the last day. Part 3 provides the material for the hands-on sessions.

Volker Weinberg, Momme Allalen
Organisation Committee
Impressions
Intel MIC Programming Workshop @ LRZ

Agenda

Monday, June 26, 2017, Kursraum 2, H.U.010 (course room)

09:00-10:00  Welcome & Introduction (Weinberg)
10:00-10:30  Overview of the Intel MIC architecture (Allalen)
10:30-11:00  Coffee break
11:00-11:30  Overview of the Intel MIC programming models (Allalen)
11:30-12:00  Native mode KNC and KNL programming (Allalen)
12:00-13:00  Lunch break
13:00-14:00  KNL Memory Modes and Cluster Modes, MCDRAM (Weinberg)
14:00-15:30  Offloading (Weinberg)
15:30-16:00  Coffee break
16:00-17:00  MKL (Allalen)

Tuesday, June 27, 2017, Kursraum 2, H.U.010 (course room)

09:00-10:30  Vectorisation and Intel Xeon Phi performance optimisation (Allalen)
10:30-11:00  Coffee break
11:00-12:00  Guided SuperMUC/MIC Tour (Weinberg/Allalen/Leisen)
12:00-13:00  Lunch break
13:00-15:30  KNL code optimisation process (Baruffa)
15:30-16:00  Coffee Break
16:00-17:00  Profiling tools: Intel Advisor (Baruffa)
18:00  GARNIX festival https://www.garnix-festival.de/
**Wednesday, June 28, 2017, 09:00-12:00, Hörsaal, H.E.009 (Lecture Hall)**

09:00-10:30  Many-core Programming with OpenMP 4.x (Michael Klemm, Intel)

10:30-10:45  Coffee Break

10:45-12:00  Advanced KNL programming techniques (Intrinsics, Assembler, AVX-512,...) (Jan Eitzinger, RRZE)

---

**Wednesday, June 28, 2017, 13:00-18:00, Hörsaal, H.E.009 (Lecture Hall)**

**Plenary session with invited talks on MIC experience and best practice recommendations (joint session with the Scientific Workshop "HPC for natural hazard assessment and disaster mitigation"), public session**

13:00-13:30  Luigi Iapichino, IPCC@LRZ: "Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures"

13:30-14:00  Michael Bader, IPCC@TUM: "Extreme-scale Multi-physics Simulation of the 2004 Sumatra Earthquake"

14:00-14:30  Vít Vondrák, IPCC@IT4I: "Development of Intel Xeon Phi Accelerated Algorithms and Applications at IT4I"

14:30-15:00  Michael Klemm, Intel: "Application Show Cases on Intel® Xeon Phi™ Processors"

15:00-15:30  Coffee Break and Group Picture

15:30-16:00  Jan Eitzinger, RRZE: "Evaluation of Intel Xeon Phi "Knights Landing": Initial impressions and benchmarking results"

16:00-16:30  Piotr Korcyl, University of Regensburg: "Lattice Quantum Chromodynamics on the MIC architectures"

16:30-17:00  Nils Moschüring, IPP: "The experience of the HLST on Europes biggest KNL cluster"

17:00-17:30  Andreas Marek, Max Planck Computing and Data Facility (MPCDF), "Porting the ELPA library to the KNL architecture"

17:30-18:00  Q&A, Wrap-up
Acknowledgements

The Czech-Bavarian Competence Centre for Supercomputing Applications is funded by the Federal Ministry of Education and Research. The Intel MIC programming workshop was financially also supported by the PRACE-4IP and PRACE-5IP projects funded by the European Commission’s Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838 and 730913.

We want to thank John Cazes et al. for allowing us to use selected material of their ISC’17 tutorial “Introduction to Manycore Programming”, Texas Advanced Computing Center, 2017.

We also thank all invited speakers for their interesting talks and for allowing us to publish their research in this booklet.
Part 1
Lecture Notes

June 26 – 28, 2017
Leibniz Supercomputing Centre
Garching b. München, Germany
About the Tutors

Dr. Momme Allalen received his Ph.D in theoretical Physics from the University of Osnabrück in 2006. He worked in the field of molecular magnetics through modelling techniques such as the exact numerical diagonalisation of the Heisenberg model. He joined the Leibniz Computing Centre (LRZ) in 2007 working in the High Performance Computing group. His tasks include user support, optimisation and parallelisation of scientific application codes, and benchmarking for characterising and evaluating the performance of high-end supercomputers. His research interests are various aspects of parallel computing and new programming languages and paradigms.

Dr. Fabio Baruffa is HPC Application Specialist at LRZ and member of the Intel Parallel Computing Center (IPCC). He was working as HPC researcher at Max-Planck (MPCDF), Jülich Research Center and Cineca where he was involved in HPC software development. His main research interests are in the area of computational methods and optimizations for HPC systems. He holds a PhD in Physics from University of Regensburg for his research in the area of spintronics.

Dr.-Ing. Jan Eitzinger (RRZE) (formerly Treibig) holds a PhD in Computer Science from the University of Erlangen. He is now a postdoctoral researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE). His current research revolves around architecture-specific and low-level optimization for current processor architectures, performance modeling on processor and system levels, and programming tools. He is the developer of LIKWID, a collection of lightweight performance tools. In his daily work he is involved in all aspects of user support in High Performance Computing: training, code parallelization, profiling and optimization, and the evaluation of novel computer architectures.

Dr.-Ing. Michael Klemm (Intel Corp.) obtained an M.Sc. in Computer Science in 2003 and received a Doctor of Engineering degree (Dr.-Ing.) from the Friedrich-Alexander-University Erlangen-Nuremberg, Germany, in 2008. Michael Klemm works in the Developer Relations Division at Intel in Germany and his areas of interest include compiler construction, design of programming languages, parallel programming, and performance analysis and tuning. Michael Klemm joined the OpenMP organization in 2009 and was appointed CEO of the OpenMP ARB in 2016.

Dr. Volker Weinberg studied physics at the Ludwig Maximilian University of Munich and later worked at the research centre DESY. He received his PhD from the Free University of Berlin for his studies in the field of lattice QCD. Since 2008 he is working in the HPC group at the Leibniz Supercomputing Centre and is responsible for HPC and PATC (PRACE Advanced Training Centre) courses at LRZ, new programming languages and the Intel Xeon Phi based system SuperMIC. Within PRACE-4IP/SIP he took over the leadership to create Best Practice Guides for new architectures and systems.
PRACE PATC Course
Intel MIC Programming Workshop

June, 26-28, 2017, LRZ

LRZ in the HPC Environment

Bavarian Contribution to National Infrastructure

German Contribution to European Infrastructure

PRACE has 25 members, representing European Union Member States and Associated Countries.
LRZ is part of the Gauss Centre for Supercomputing (GCS), which is one of the six
PRACE Advanced Training Centres (PATCs) that started in 2012:
- Barcelona Supercomputing Center (Spain), CINECA
- Consorzio Interuniversitario (Italy)
- CSC – IT Center for Science Ltd (Finland)
- EPCC at the University of Edinburgh (UK)
- Gauss Centre for Supercomputing (Germany)
- Maison de la Simulation (France)

Mission: Serve as European hubs and key drivers of advanced high-quality
training for researchers working in the computational sciences.

http://www.training.prace-ri.eu/

Tentative Agenda: Monday

- Monday, June 26, 2017, Kursraum 2, H.U.010 (course room)
- 09:00-10:00 Welcome & Introduction (Weinberg)
- 10:00-10:30 Overview of the Intel MIC architecture (Allalen)
- 10:30-11:00 Coffee break
- 11:00-11:30 Overview of the Intel MIC programming models (Allalen)
- 11:30-12:00 Native mode KNC and KNL programming (Allalen)

- 12:00-13:00 Lunch break
- 13:00-14:00 KNL Memory Modes and Cluster Modes, MCDRAM (Weinberg)
- 14:00-15:30 Offloading (Weinberg)
- 15:30-16:00 Coffee break
- 16:00-17:00 MKL (Allalen)
Tentative Agenda: Tuesday

- Tuesday, June 27, 2017, Kursraum 2, H.U.010 (course room)
  - 09:00-10:30 Vectorisation and Intel Xeon Phi performance optimisation (Allalen)
  - 10:30-11:00 Coffee break
  - 11:00-12:00 Guided SuperMUC/MIC Tour (Weinberg/Allalen)
  - 12:00-13:00 Lunch break
  - 13:00-15:30 KNL code optimisation process (Baruffa)
  - 15:30-16:00 Coffee Break
  - 16:00-17:00 Profiling tools: Intel Advisor (Baruffa)
  - 18:00 - open end at GARNIX  https://www.garnix-festival.de/

Tentative Agenda: Wednesday

- Wednesday, June 28, 2017, 09:00-12:00, Hörsaal, H.E.009 (Lecture Hall)
  - 09:00-10:30 Many-core Programming with OpenMP 4.x (Michael Klemm, Intel)
  - 10:30-10:45 Coffee Break
  - 10:45-12:00 Advanced KNL programming techniques (Intrinsics, Assembler, AVX-512,...) (Jan Eitzinger, RRZE)
  - 12:00-13:00 Lunch Break
Tentative Agenda: Wednesday

- **Wednesday, June 28, 2017, 13:00-18:00, Hörsaal, H.E.009 (Lecture Hall)**
- Plenum session with invited talks on MIC experience and best practice recommendations (joint session with the Scientific Workshop "HPC for natural hazard assessment and disaster mitigation"), public session
- 13:00-13:30 Luigi Iapichino, IPCC@LRZ: "Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures"
- 13:30-14:00 Michael Bader/Carsten Uphoff, IPCC@TUM: "Extreme-scale Multi-physics Simulation of the 2004 Sumatra Earthquake"
- 14:00-14:30 Vit Vondrak/Branislav Jansik, IPCC@IT4I: "Development of Intel Xeon Phi Accelerated Algorithms and Applications at IT4I"
- 14:30-15:00 Michael Klemm, Intel: "Application Show Cases on Intel® Xeon Phi™ Processors"
- 15:00-15:30 Coffee Break
- 15:30-16:00 Jan Eitzinger, RRZE: "Evaluation of Intel Xeon Phi "Knights Landing": Initial impressions and benchmarking results"
- 16:00-16:30 Piotr Korcyl, University of Regensburg: "Lattice Quantum Chromodynamics on the MIC architectures"
- 16:30-17:00 Nils Moschüring, IPP: "The experience of the HLST on Europe's biggest KNL cluster"
- 17:00-17:30 Andreas Marek, Max Planck Computing and Data Facility (MPCDF), "Porting the ELPA library to the KNL architecture"
- 17:30-18:00 Q&A, Wrap-up

Information

- **Lecturers:**
  - Dr. Momme Allalen, Dr. Fabio Baruffa, Dr. Volker Weinberg (LRZ)
  - Dr.-Ing. Jan Eitzinger (RRZE)
  - Dr.-Ing. Michael Klemm (Intel Corp.)

- **Complete lecture slides & exercise sheets:**
  - [https://www.lrz.de/services/compute/courses/x_lecturenotes/mic_workshop_2017/](https://www.lrz.de/services/compute/courses/x_lecturenotes/mic_workshop_2017/)
  - [http://tinyurl.com/yd6lfweq](http://tinyurl.com/yd6lfweq)

- **Examples under:**
  - [lrz/sys/courses/MIC_Workshop](http://tinyurl.com/yd6lfweq)
Intel Xeon Phi @ LRZ and EU

26.-28.6.2017
Intel MIC Programming Workshop @ LRZ

Intel Xeon Phi and GPU Training @ LRZ

28.-30.4.2014 @ LRZ (PATC): KNC+GPU
27.-29.4.2015 @ LRZ (PATC): KNC+GPU
3.-4.2.2016 @ IT4Innovations: KNC
27.-29.6.2016 @ LRZ (PATC): KNC+KNL
28.9.2016 @ PRACE Seasonal School, Hagenberg: KNC
7.-8.2.2017 @ IT4Innovations (PATC): KNC
26.-28.6.2017 @ LRZ (PATC): KNL
June 2018 @ LRZ (PATC thc.): KNL

http://inside.hlrs.de/
inSiDE, Vol. 12, No. 2, p. 102, 2014
inSiDE, Vol. 15, No. 1, p. 48ff, 2017
Evaluating Accelerators at LRZ

Research at LRZ within PRACE & KONWIHR:

- **CELL programming**
  - IBM announced to discontinue CELL in Nov. 2009.

- **GPGPU programming**
  - Regular GPGPU computing courses at LRZ since 2009.
  - Evaluation of GPGPU programming languages:
    - CAPS HMPP
    - PGI accelerator compiler
    - CUDA, cuBLAS, cuFFT
    - PyCUDA/R

- **Intel Xeon Phi programming**

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

IPCC (Intel Parallel Computing Centre)

- **New Intel Parallel Computing Centre (IPCC) since July 2014:** Extreme Scaling on MIC/x86
- **Chair of Scientific Computing** at the Department of Informatics in the Technische Universität München (TUM) & LRZ
- https://software.intel.com/de-de/ipcc#centers

- **Codes:**
  - Simulation of Dynamic Ruptures and Seismic Motion in Complex Domains: SeisSol
  - Numerical Simulation of Cosmological Structure Formation: GADGET
  - Molecular Dynamics Simulation for Chemical Engineering: ls1 mardyn
  - Data Mining in High Dimensional Domains Using Sparse Grids: SG++
Czech-Bavarian Competence Team for Supercomputing Applications (CzeBaCCA)

- New BMBF funded project that started in Jan. 2016 to:
  - Foster Czech-German Collaboration in Simulation Supercomputing
    - series of workshops will initiate and deepen collaboration between Czech and German computational scientists
  - Establish Well-Trained Supercomputing Communities
    - joint training program will extend and improve trainings on both sides
  - Improve Simulation Software
    - establish and disseminate role models and best practices of simulation software in supercomputing

Intel MIC Programming Workshop @ LRZ
26.-28.6.2017

CzeBaCCA Trainings and Workshops

- Intel MIC Programming Workshop, 3 – 4 February 2016, Ostrava, Czech Republic
- Scientific Workshop: SeisMIC - Seismic Simulation on Current and Future Supercomputers, 5 February 2016, Ostrava, Czech Republic
- PRACE PATC Course: Intel MIC Programming Workshop, 27 - 29 June 2016, Garching, Germany
- Scientific Workshop: High Performance Computing for Water Related Hazards, 29 June - 1 July 2016, Garching, Germany
- PRACE PATC Course: Intel MIC Programming Workshop, 7 – 8 February 2017, Ostrava, Czech Republic
- Scientific Workshop: High performance computing in atmosphere modelling and air related environmental hazards, 9 February 2017, Ostrava, Czech Republic
- PRACE PATC Course: Intel MIC Programming Workshop, 26 – 28 June 2017, Garching, Germany
- Scientific Workshop: HPC for natural hazard assessment and disaster migration, 28 - 30 June 2017, Garching, Germany
CzeBaCCA Trainings and Workshops

1st workshop series: February 2016 @ IT4I

https://www.lrz.de/forschung/projekte/forschung-hpc/CzeBaCCA/
http://www.gate-germany.de/fileadmin/dokumente/Laenderprofile/Laenderprofil_Tschechien.pdf, p. 27

26.-28.6.2017

Intel MIC Programming Workshop @ LRZ

CzeBaCCA Trainings and Workshops

2nd workshop series: June 2016 @ LRZ

https://www.lrz.de/forschung/projekte/forschung-hpc/CzeBaCCA/
http://www.gate-germany.de/fileadmin/dokumente/Laenderprofile/Laenderprofil_Tschechien.pdf, p. 27

26.-28.6.2017

Intel MIC Programming Workshop @ LRZ
CzeBaCCA Trainings and Workshops

3rd workshop series: February 2017 @ IT4I

https://www.lrz.de/forschung/projekte/forschung-hpc/CzeBaCCA/
http://www.gate-germany.de/fileadmin/dokumente/Laenderprofile/Laenderprofil_Tschechien.pdf, p.27

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

Intel Xeon Phi @ Top500 June 2017

- https://www.top500.org/list/2017/06/
- #2: Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P, National Super Computer Center in Guangzhou, China
- #6: Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect, Cray Inc., DOE/SC/LBNL/NERSC, United States
- #7: Oakforest-PACS - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path, Fujitsu, Joint Center for Advanced High Performance Computing, Japan
- #12: Stampede2 - PowerEdge C6320P, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path, Dell, Texas Advanced Computing Center/Univ. of Texas, United States
- #14: Marconi, Intel Xeon Phi - CINECA Cluster, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path, Lenovo, CINECA, Italy
- ... several non European systems ...
- #78: Salomon - SGI ICE X, Xeon E5-2680v3 12C 2.5GHz, Infiniband FDR, Intel Xeon Phi 7120P, HPE, IT4Innovations National Supercomputing Center, VSB-Technical University of Ostrava, Czech Republic

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
The following 4 Best Practice Guides (BPGs) have been written within PRACE-4IP by 13 authors from 8 institutions and have been published in pdf and html format in January 2017 on the PRACE website:

- **Intel® Xeon Phi™ BPG**
  Update of the PRACE-3IP BPG
- **Haswell/Broadwell BPG**
  Written from scratch
- **Knights Landing BPG**
  Written from scratch
- **GPGPU BPG**
  Update of the PRACE-2IP mini-guide

Intel MIC within PRACE: Intel Xeon Phi (KNC) Best Practice Guide

- Created within PRACE-3IP+4IP.
- Written in Docbook XML.
- 122 pages, 13 authors
- Now including information about existing Xeon Phi based systems in Europe: Avitohol @ BAS (NCSA), MareNostrum @ BSC, Salomon @ IT4Innovations, SuperMIC @ LRZ

Intel MIC within PRACE: Knights Landing Best Practice Guide

- Created within PRACE-4IP.
- Written in Docbook XML.
- 85 pages, 3 authors
- General information about the KNL architecture and programming environment
- Benchmark & Application Performance results
Best Practice Guides - Dissemination

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

SuperMIC ∈ SuperMUC @ LRZ
SuperMUC System Overview

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

SuperMUC Phase 2: Moving to Haswell

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
SuperMUC Phase 2: Moving to Haswell

26.–28.6.2017 Intel MIC Programming Workshop @ LRZ

SuperMIC: Intel Xeon Phi Cluster

26.–28.6.2017 Intel MIC Programming Workshop @ LRZ
SuperMIC: Prototype Intel Phi (KNC) System

- 32 compute nodes (diskless)
  - SLES11 SP3
  - 2 Ivy-Bridge host processors E5-2650@2.6 GHz with 16 cores
  - 2 Intel Xeon Phi 5110P coprocessors per node with 60 cores
  - 64 GB (Host) + 2 * 8 GB (Xeon Phi) memory
  - 2 MLNX CX3 FDR PCIe cards attached to each CPU socket

- Interconnect
  - Mellanox Infiniband FDR14
  - Through Bridge Interface all nodes and MICs are directly accessible

- 1 Login- and 1 Management-Server (Batch-System, xCAT, …)
- Air-cooled
- Supports both native and offload mode
- Batch-system: LoadLeveler
SuperMIC Network Access

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

- Description of SuperMIC:
- https://www.lrz.de/services/compute/supermuc/supermic/

- Training Login Information:
- https://www.lrz.de/services/compute/supermuc/supermic/training-login/

- Use course account on paper snippets
First login to Linux-Cluster (directly reachable from the course PCs, use only account a2c06aa!):

```
ssh lxlogin1.lrz.de -l a2c06aa
```

Then:

```
ssh mcct03.cos.lrz.de  or  
ssh mcct04.cos.lrz.de
```

Processor: Intel(R) Xeon Phi(TM) CPU 7210. 64 cores, 4 threads per core. Frequency: 1 - 1.5 GHz

**KNL**: 64 cores x 1.3 GHz x 8 (SIMD) x 2 x 2 (FMA) = 2662.4 GFLOP/s

**Compare with:**

**KNC**: 60 cores x 1 GHz x 8 (SIMD) x 2 (FMA) = 960 GFLOP/s

**Sandy-Bridge**: 2 sockets x 8 cores x 2.7 GHz x 4 (SIMD) x 2 (ALUs) = 345.6 GFLOP/s

---

**Xeon Phi References**

- **Books:**
  - [http://lotsofores.com](http://lotsofores.com); new KNL edition in July 2016

- **Training material by CAPS, TACC, EPCC**

- **Intel Training Material and Webinars**


Acknowledgements

- IT4Innovation, Ostrava.
- Partnership for Advanced Computing in Europe (PRACE)
- Intel
- BMBF (Federal Ministry of Education and Research)
- Dr. Karl Fürlinger (LMU)
- J. Cazes, R. Evans, K. Milfeld, C. Proctor (TACC)
- Adrian Jackson (EPCC)

And now …

Enjoy the course!
Intel MIC Programming Workshop: Hardware Overview & Native Execution
Dr. Momme Allalen (LRZ)
June, 26-28, 2017 @ LRZ

Agenda

- Intro @ accelerators on HPC
- Architecture overview of the Intel Xeon Phi Products (MIC)
- KNL vs KNC
- KNC Programming models
- What you need to know to start your code on KNC
- Native mode KNC and KNL programming
- Hands on
Why do we need “accelerators” on HPC?

- In the past, computers got faster by increasing the clock frequency of the core, but this has now reached its limit mainly due to power requirements.

- Today, processor cores are not getting any faster, but instead the number of cores per chip increases and registers are getting wider.

- On HPC, we need a chip that can provide:
  - higher computing performance
  - @high power efficiency: keep the power/core as low as possible

One solution is a heterogeneous system containing both CPUs and “accelerators”, plus other forms of parallelism such as vector instruction support.

- Two types of hardware options, Intel Xeon Phi (KNC) and Nvidia GPU.

- Can perform many parallel operations every clock cycle.

---

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
Intel Multi-Core Architecture

- Intel Xeon processors are for general purpose
- The current architecture:
  - Haswell and Broadwell
  - Skylake is upcoming

Architectures Comparison (CPU vs GPU)

- Large cache and sophisticated flow control minimise latency for arbitrary memory access.
- Simple flow control
- More transistors for computing in parallel (up to 21 billion on Nvidia Volta GPU)
- (SIMD)

<table>
<thead>
<tr>
<th></th>
<th>Intel Xeon CPU E5-2697v4 “Broadwell”</th>
<th>Nvidia GPU P100</th>
<th>Nvidia GPU V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores @ Clock</td>
<td>2 x 18 cores @ ≥ 2.3 GHz</td>
<td>56SMs @ 1.4 GHz</td>
<td>80SMs@1.4 GHz</td>
</tr>
<tr>
<td>SP Perf./core</td>
<td>≥ 73.6 GFlop/s</td>
<td>up to 166 GFlop/s</td>
<td></td>
</tr>
<tr>
<td>SP peak</td>
<td>≥ 2.6 TFlop/s</td>
<td>up to 10.6 TFlop/s</td>
<td></td>
</tr>
<tr>
<td>Transistors/TDP</td>
<td>2x7 Billion /2x145W</td>
<td>14 Billion/300W</td>
<td>21 Billion/300W</td>
</tr>
<tr>
<td>BW</td>
<td>2 x 62.5 GB/s</td>
<td>510 GB/s</td>
<td>up to 900 GB/s</td>
</tr>
</tbody>
</table>
**Intel Xeon Phi Products**

**Intel Many Integrated Core (MIC) Architecture**

- **Xeon Phi Coprocessor**: first product was released in 2012 named Knights Corner (KNC) which is the first architecture supporting 512 bit vectors.

- **Xeon Phi Processor**: 2nd generation announced at ISC16 in June named Knights Landing (KNL) also support 512 bit vectors with a new instruction set called Intel Advanced Vector Instructions 512 (Intel AVX-512).

**Specialised Platform for high demanding computing application**

**Intel KNC architecture in common with Intel multi-core Xeon CPUs!**

- X86 architecture
- C, C++ and Fortran
- Standard parallelisation libraries
- Similar optimisation methods
- Up to 22 cores/socket
- Up to 3 GHz
- Up to 1.54 TB RAM
- 256 bit AVX vectors
- 2-way hyper-threading

- PCIe bus connection
- IP-addressable
- Own Linux version OS with minimal shell environment
- Full Intel software tool suite
- 8 - 16 GB GDDR5 DRAM (cached)
- Up to 61 x86 (64 bit) in-order cores
- Up to 1 GHz
- 512 bit wide vector registers
- 4 way hyper-threading
- SSE, AVX or AVX2: are **not supported**
- Intel Initial Many Core Instructions (IMCI).
Architectures Comparison

**CPU**
- General-purpose architecture

**MIC**
- Power-efficient Multiprocessor X86 design architecture

**GPU**
- Massively data parallel

---

### System Comparisons

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Hmax (TFlop/s)</th>
<th>Hpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRPC, National Supercomputing Center in Wuxi, China</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-PEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 7250 16C, NUDT, National Super Computer Center in Chongzhou, China</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc, Swiss National Supercomputing Centre (CSCS), Switzerland</td>
<td>361,760</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2,272</td>
</tr>
<tr>
<td>4</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x, Cray Inc, DOD/SC/Oak Ridge National Laboratory, United States</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>5</td>
<td>Sequoia - BlueGene/Q, Power BGC 16C 1.60 GHz, Cusom, IBM DOD/NNSA/LLN, United States</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>6</td>
<td>Cori - Cray XK7, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect, Cray Inc, DOE/SC/BNL/NERSC, United States</td>
<td>622,336</td>
<td>14,014.7</td>
<td>27,880.7</td>
<td>3,939</td>
</tr>
<tr>
<td>7</td>
<td>Oakforest-PACS - PRIMEGRAPES/M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path, Fujitsu, Joint Center for Advanced High Performance Computing, Japan</td>
<td>556,104</td>
<td>13,554.6</td>
<td>24,913.5</td>
<td>2,719</td>
</tr>
<tr>
<td>8</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tsubame interconnect, Fujitsu RIKEN Advanced Institute for Computational Science (AICS), Japan</td>
<td>705,024</td>
<td>10,510.0</td>
<td>11,280.4</td>
<td>12,660</td>
</tr>
</tbody>
</table>

---

Intel MIC Programming Workshop @ Ostrava
Intel Xeon Phi KNL Architecture

- Bootable CPU
- Up to 72 cores based on the Intel Atom cores (Silvermont microarchitecture)
- 4HT running @ 1.3 to 1.5 GHz
- 3+ TFlop/s in DP (FMA)
- 6+ TFlop/s in SP (FMA)
- ~ 384 GB DDR4 (> 90 GB/s)
- 16 GB HBM (MCDRAM)
  > 400 GB/s
- Binary-compatible with Xeon
- Common operating system (SUSE, WINDOWS, RHEL…)

KNC vs KNL

- Co-processor
- Binary incompatible with other architectures
- 61 In-order cores
- 1.1 GHz processor
- up to 16 GB RAM
- 22 nm process
- One 512-bit VPU
- No support for branch prediction and fast unaligned memory access

- No PCIe - Self hosted
- Binary compatible with prior Xeon architectures (no phi)
- up to 72 Out-of-order cores
- 1.4 GHz processor
- Up to 400 GB RAM (with MCDRAM)
- 14 nm process
- Tow 512-bit VPUs
- Support for branch prediction and fast unaligned memory access

The Improvement on the KNL Hardware still not good for non optimised code
## Invocation of the Intel MPI compiler

<table>
<thead>
<tr>
<th>Language</th>
<th>MPI Compiler</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpiicc</td>
<td>icc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiicpc</td>
<td>icpc</td>
</tr>
<tr>
<td>Fortran</td>
<td>mpiifort</td>
<td>ifort</td>
</tr>
</tbody>
</table>

---

**Lab1: Access SuperMIC @LRZ**
Interacting with Intel Xeon Phi Coprocessors

```
user@host~$ micinfo --listdevices

MicInfo Utility Log
Created Thu Jan 7 09:33:20 2016
List of Available Devices

deviceId | domain | bus# | pciDev# | hardwareId
----------|--------|------|---------|----------------
0         | 0      | 20   | 0       | 22508086
1         | 0      | 8b   | 0       | 22508086
```

```
user@host~$ micinfo | grep -i cores

Cores
Total No of Active Cores : 60
Cores
Total No of Active Cores : 60
```

Useful Tools and Files on Coprocessor

- top - display Linux tasks
- ps - report a snapshot of the current processes.
- kill - send signals to processes, or list signals
- ifconfig - configure a network interface
- traceroute - print the route packets take to network host
- mpiexec.hydra – run Intel MPI natively
- /proc/cpuinfo
- /proc/meminfo
Interacting with Intel Xeon Phi Coprocessors

user@host~$ /sbin/lspci | grep -i "co-processor"
20:00.0 Co-processor: Intel Corporation Xeon Phi .... (rev 20)
8b:00.0 Co-processor: Intel Corporation Xeon Phi .... (rev 20)

user@host~$ cat /etc/hosts | grep mic1
user@host~$ cat /etc/hosts | grep mic1-ib0 | wc –l
user@host~$ ssh mic0 or ssh mic1

user@host-mic0~$ ls /
bin boot dev etc home init lib lib64 lrz media mnt proc root sbin sys tmp usr var
• micsmc a utility for monitoring the physical parameters of Intel Xeon Phi coprocessors: model, memory, core rail temperatures, core frequency, power usage, etc.

Intel Fabric integrated on KNL processor

• Intel released KNL-F on Nov. 2016; KNL with a Fabric
  ✓ Intel Omni-Path Architecture
  ✓ High Bandwidth and low latency
  ✓ The Omni-Path technology will allow to build Clusters like: LRZ-KNL Cluster
• Other Xeon Phi based system: Server, Workstation ..
  dap.xeonphi.com
Parallelism on Xeon Phi

C/C++/Fortran, Python/Java …. Porting is easy
Two parallelisation modes are required: Shared memory and vectorisation
Run multiple threads/processes and each thread issues vector instructions (SIMD)

Shared memory parallelism: OpenMP

Victorisation

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de

KNL: Cores and threads

- Up to 36 tiles connected by 2 D Mesh interconnect
  each with 2 physical cores (up 72 cores with out of order instruction execution)
- Distributed L2 cache across a mesh interconnect
KNL Tile (Cores and threads)

- Up to 72 cores with 4 way hyper threading up to 288 logical processors

- Up to 36 MB L2 per KNL

KNL and Vector Instruction Sets

- Binary runs without recompilation
- KNC binary requires recompilation

- Conflict Detection: Improves Vectorisation
- Prefetch: Gather and Scatter Prefetch
- Exponential and Reciprocal Instructions

SNB E5-2600 HSW E5-2600 KNL
x87/MMX x87/MMX x87/MMX
SSE* SSE* SSE*
AVX AVX AVX
AVX2 AVX2 AVX2
AVX-512F AVX-512CD AVX-512PF AVX-512ER
Memory on KNL

- Two levels of memory on KNL:
  1. Main memory
     - KNL has direct access to all of main memory
     - Similar latency and bandwidth as a standard Xeon processors
     - 6 DDR channels
  2. Multi-Channel DRAM or MCDRAM
     - HBM on chip: 16GB
     - Slightly higher latency than main memory
     - 8 MCDRAM controllers, 16 channels

Using MCDRAM on KNL

- At boot time you have to choose one memory mode operation

  **Flat Mode**
  - MCDRAM treated as a NUMA node
  - As separately addressable memory
  - Users control what goes to MCDRAM

  **Cache Mode**
  - MCDRAM treated as a transparent Last Level Cache (LLC)
  - MCDRAM is used automatically

  **Hybrid Mode**
  - Combination of Flat and Cache
  - Ratio can be chosen in the BIOS
Using MCDRAM on KNL

- Flat mode offers the best performance for applications, but require changes to the code (memory allocation) or the execution environment (NUMA node)

```
numactl --membind 0 ./exec                   # DDR4
numactl --membind 1 ./exec                   # MCDRAM
```

Programming Models on KNC

- Native Mode
  - Programs started on Xeon Phi KNC
  - Cross-compilation using –mmic
  - User access to Xeon Phi is necessary
- Offload to MIC (KNC)
  - Offload using OpenMP extensions
  - Automatically offload some routines using MKL
    - MKL Compiled assisted offload (CAO)
    - MKL automatic Offload (AO)
- MPI tasks on Host and MIC
  - Treat the coprocessor like another host
    - MPI only and MPI + X (X may be OpenMP, TBB, Cilk, OpenCL…etc.)
Native Mode on KNC

• First ensure that the application is suitable for native execution.
• The application runs entirely on the MIC coprocessor without offload from a host system.
• Compile the application for native execution using the flag: -mmic
• Build also the required libraries for native execution.
• Copy the executable and any dependencies, such as run time libraries to the coprocessor.
• Mount file shares to the coprocessor for accessing input data sets and saving output data sets.
• Login to the coprocessor via console, setup the environment and run the executable.
• You can debug the native application via a debug server running on the coprocessor.

Native Mode on KNC

• Compile on the Host:

```
~$ $INTEL_BASE/linux/bin/compilervars.sh intel64
~$ icpc -mmic hello.c -o hello.knc
~$ ifort -mmic hello.f90 -o hello.knc
```

• Launch execution from the MIC (KNC):

```
~$ scp hello $HOST-mic0:
~$ ssh $HOST-mic0
~$ ./hello.knc

hello, world
```
Native Mode on KNC: micnativeloadex

The tool automatically transfer the code and dependent libraries and execute from the host:

```
~$ ./hello.knc
-bash: ./hello: cannot execute Binary file
~$ export SINK_LIBRARY_PATH=../intel/compiler/lib/mic
~$ micnativeloadex ./hello.knc
  hello, world
~$ micnativeloadex ./hello.knc -v
  hello, world
  Remote process returned: 0
  Exit reason: SHUTDOWN OK
```
For the Labs:
SuperMIC System Initialisation

- Exercise Sheets + Slides online:
  https://goo.gl/IPBnmK

micinfo and _SC_NPROCESSORS_ONLN

~$ micinfo –listdevices
~$ micinfo | grep -i cores
~$ cat hello.c
#include <stdio.h>
#include <unistd.h>
int main(){
    printf("Hello world! I have %ld logical cores.\n",
            sysconf(_SC_NPROCESSORS_ONLN));
}
~$ icc hello.c –o hello-host && ./hello-host
~$ icc –mmic hello.c –o hello-mic
~$ micnativeloadex ./hello-mic
Lab3: Access KNL test system

Interacting with Intel Xeon Phi Processors

user@host~$ ssh lxlogin1.lrz.de -l Your-User-ID
user@host~$ ssh mcct03.cos.lrz.de -l Your-User-ID
user@host~$ ssh mcct04.cos.lrz.de -l Your-User-ID
user@host~$ module list or module av

user@mcct03~$ numactl -H
node 0 size: 96457 MB
node 0 free: 92947 MB
node 1 cpus:
node 1 size: 16011 MB
node 1 free: 15865 MB
node distances:
node 0 1
 0: 10 31
 1: 31 10

user@mcct04~$ numactl -H
node 0 size: 96341 MB
node 0 free: 79787 MB
node distances:
node 0
 0: 10
Intel MIC Programming Workshop: KNL MCDRAM and KNC Offloading
Dr. Volker Weinberg (LRZ)

June 26-28, 2017, LRZ

KNL Cluster and Memory Modes
MCDRAM

with material from Intel, John Cazes et al. (TACC) and Adrian Jackson (EPCC)
Cores are grouped in pairs (tiles)
- 36 possible tiles
- 2D mesh interconnect
- 2 DDR memory controllers
  - 6 channels DDR4
  - Up to 90 GB/s
- 16 GB MCDRAM
  - Up to 475 GB/s
**Tile**

- Basic unit for replication
- Each tile consists of **2 cores**, **2 vector-processing units (VPU)** per core, a **1 MB L2 Cache** shared between the 2 cores
- **CHA** (caching/home agent)
  - Serves as the point where the tile connects to the mesh
  - Holds a portion of the **distributed tag directory structure**

---

**2D Mesh Interconnect**

- Tiles are connected by a cache-coherent, **2D mesh interconnect**
- Provides a more scalable way to connect the tiles by providing higher bandwidth and lower latency compared to KNC **1D ring interconnect**
- **MESIF** (Modified, Exclusive, Shared, Invalid, Forward) **cache-coherent protocol**
- Cache lines present in L2 caches are tracked using a **distributed tag directory structure**
- Around **700 GB/s total aggregate bandwidth**
- Mesh is organized into rows and columns of half rings that fold upon themselves at the endpoints
- Mesh enforces a **YX routing rule**
- Mesh at fixed frequency of 1.7 GHz
- Single hop: X-direction 2 clocks, Y-direction: 1 clock
KNL Memory Architecture

- **2 Memory Types**
  - MCDRAM (16 GB)
  - DDR4 (96 GB)

- **3 Memory Modes**
  - Cache
  - Flat
  - Hybrid
    - 25% (4 GB)
    - 50% (8 GB)
    - 75% (12 GB)

---

KNL Memory: Overview

- **Memory hierarchy on KNL:**
  - DDR4 (96 GB)
  - MCDRAM (16 GB)
  - Tile L2 (1 MB)
  - Core L1 (32 KB)

- **Tile:** set of 2 cores sharing a 1MB L2 cache and connectivity on the mesh

- **Quadrant:** virtual concept, not a hardware property. Way to divide the tiles at a logical level.

- **Tag Directory:** tracks cache line locations in all L2 caches. It provides the block of data or (if not available in L2) a memory address to the memory controller.
KNL Memory: MCDRAM

- **High-bandwidth** memory integrated on-package
- 8 MCDRAM devices on KNL, each with 2 GB capacity -> **total 16 GB**
- Connected to EDC memory controller via proprietary on-package I/O: OPIO
- Each device has a separate read and write bus connecting it to its EDC
- **Aggregate Stream Triads Bandwidth** for the 8 MCDRAMS is **over 450 GB/s**
- Slighter higher latency than main memory (~10% slower)

---

KNL Memory: DDR4

- **High-capacity** memory off-package
- KNL has direct access to all of main memory
- 2 DDR4 memory controllers on opposite sides of the chip, each controlling 3 DDR4 channels
- Maximum total capacity is 384 GB
- **Aggregate Stream Triads Bandwidth** from all 6 DDR4 channels is around **90 GB/s**
KNL Memory Modes

- **Cache**:  
  - MCDRAM serves as cache for transactions to DDR4 memory  
  - Direct-mapped memory-side cache with 64-byte cache-lines  
  - Inclusive of all modified lines in L2 cache  
  - Completely transparent to the user

- **Flat**:  
  - Flat address space  
  - Different NUMA nodes for DDR4 and MCDRAM  
  - `numactl` or `memkind` library can be used for allocation

- **Hybrid**:  
  - 20% / 50% / 75% of MCDRAM set up as cache  
  - Potentially useful for some applications
Memory Modes: Comparison

<table>
<thead>
<tr>
<th>Memory Mode</th>
<th>MCDRAM</th>
<th>DDR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat</td>
<td>100% Memory</td>
<td>100% Memory</td>
</tr>
<tr>
<td>Hybrid 25%</td>
<td>75% Memory</td>
<td>100% Memory</td>
</tr>
<tr>
<td>Hybrid 50%</td>
<td>50% Memory</td>
<td>100% Memory</td>
</tr>
<tr>
<td>Hybrid 75%</td>
<td>25% Memory</td>
<td>100% Memory</td>
</tr>
<tr>
<td>Cache</td>
<td>0% Memory</td>
<td>100% Memory</td>
</tr>
</tbody>
</table>

Cluster Modes

- Cluster Modes modify the distance that L2 coherency traffic flows go through the mesh
- 5 Cluster Modes supported:
  - All-to-all
  - Quadrant / Hemisphere
  - 2 Sub-NUMA Cluster modes: SNC-4 / SNC-2
- Regardless of the cluster mode selected, all memory (all MCDRAM and all DDR4) is available to all cores, and all memory is fully cache-coherent.
- What differs between the modes is whether the view of MCDRAM or DDR is UMA (Uniform Memory Access) or NUMA.
Cluster Modes: Overview

Cluster modes modify the distance that coherency traffic flows through mesh!

- A quadrant is a virtual concept and not a hardware property. Divides tiles into 4 groups at a logical level.
- A tag directory is used to track L2 cache line locations and status (which tile and if valid).
- Tag directory is distributed over tiles:
  - Each directory component is responsible for an exclusive portion of address space.
  - Directory indicates where cache line is: a certain tile’s L2 cache or in memory.
Cluster Modes

- A tile’s L2 cache can hold any data
- Tag directory tracks if data is in L2 and which tile’s L2 has data
- Tag directory is distributed across all tiles
  - Each tile has an equal portion of the address space
  - Portion of tag directory in a tile not related to L2 cache in that tile
- Every tile has a Caching-Home Agent (CHA)
  - services queries about it’s portion of the tag directory

Cluster-Modes: UMA vs. NUMA

- Quadrant-Mode:
  - Each memory type is UMA
  - latency from any given core to any memory location in the same memory type (MCDRAM or DDR) is essentially the same
- SNC-4 (2) Mode:
  - Each memory type is NUMA
  - Cores and memory divided into 4 quadrants (2 halves) with lower latency for near memory accesses (within the same quadrant (half)) and higher latency for far memory accesses (within a different quadrant (half))
Cluster-Modes: NUMA Domains

- **Flat-All2All/Quadrant/Hemisphere Mode:**
  - 1 DDR NUMA Domain
  - 1 MCDRAM NUMA Domain

- **Flat-SNC-2:**
  - 2 DDR NUMA Domain
  - 2 MCDRAM NUMA Domain

- **Flat-SNC-4:**
  - 4 DDR NUMA Domain
  - 4 MCDRAM NUMA Domain

- **Cache Mode:**
  - 1 DDR NUMA Domain
  - 0 MCDRAM NUMA Domain

Memory interleaving (technique to spread out consecutive memory access across multiple memory channels in order to parallelise the accesses) differs among the various modes.

---

Cache Coherency Protocol

- For memory loads/stores:
  - Core (requestor) looks in local L2 cache

- If not there it queries Distributed Tag Directory (DTD) for it:
  - Sends message to tile (CHA) containing DTD entry for that memory address (tag owner) to check if *any other tile on the chip* has that address in its caches

- **If it’s not in any cache** then data fetched from memory
  - DTD updates with requestor information

- **If it’s in a tile’s L2 cache** then:
  - Tag owner sends message to tile where data is (resident)
  - Resident sends data to requestor
Cluster Modes: All to All

1. L2 Miss: data not in local L2
2. Directory access: look for tag in directory
3. Memory access: look for data in memory
4. Data: send data to original tile

- **No affinity between tile and tag directory:**
  When there is an L2 miss in a tile the directory tag may be anywhere on the chip

- **No affinity between tag directory and memory:**
  Data associated to a directory tag may be anywhere on the chip

Cluster Modes: Quadrant

1. L2 Miss: data not in local L2
2. Directory access: look for tag in directory
3. Memory access: look for data in memory
4. Data: send data to original tile

- **No affinity between tile and tag directory:**
  When there is an L2 miss in a tile the directory tag may be anywhere on the chip

- **Affinity between tag directory and memory:**
  Data associated to a directory tag will be in the same quadrant that the directory tag is located
Cluster Modes: SNC-4

• **Affinity** between tile and tag directory:
  When there is an L2 miss in a tile the directory tag will be in the same quadrant

• **Affinity** between tag directory and memory:
  Data associated to a directory tag will be in the same quadrant that the directory tag is located

Cluster Modes: Some more remarks

• **All-to-All Mode:**
  – Most general mode. Lower performance than other modes, ignore
  – Only mode that can be used when DDR DIMMS have not identical capacity

• **Quadrant-Mode:**
  – Lower latency and higher bandwidth than all-to-all.
  – Will always give reasonable performance
  – SW transparent, no special NUMA optimisation
    – 1 NUMA region for MCDRAM
    – 1 NUMA region for DDR
  – Specially well suited for MPI applications with 1 rank per KNL

• **SNC-4:**
  – Each Quadrant exposed as a separate NUMA domain (like 4-Socket Xeon)
  – Well suited for MPI applications with 4 or n*4 ranks per KNL
  – SW needs to NUMA optimise to get benefit
  – Good for NUMA-aware code
Cluster-Modes & Memory Modes Combinations

- 5 Flat Memory Mode Variants:
  - Flat-A2A
  - Flat-Quadrant
  - Flat-Hemisphere
  - Flat-SNC4
  - Flat-SNC2

- 5 Cache Memory Mode Variants
  - Cache-A2A
  - Cache-Quadrant
  - Cache-Hemisphere
  - Cache-SNC4
  - Cache-SNC2

- 5 x 3 = 15 Hybrid Variants

Using numactl

- use only DDR (default)
  numactl --membind=0 ./a.out

- use only MCDRAM in flat-quadrant mode
  numactl --membind=1 ./a.out

- use MCDRAM if possible in flat-quadrant mode; else DDR
  numactl --preferred=1 ./a.out

- show numactl settings
  numactl --hardware

- list available numactl options
  numactl --help
Using numactl in various modes

- **Flat-quadrant mode**: use only MCDRAM
  numactl --membind=1 ./a.out
- **Flat-SNC2 mode**: use only MCDRAM
  numactl --membind=2,3 ./a.out
- **Flat-SNC4 mode**: use only MCDRAM
  numactl --membind=4,5,6,7 ./a.out

Changing of Memory and Cluster Modes

- **KNL** = single chip solution that can change the design of a machine at a level that has traditionally unchangeable
- **Operating Systems and applications are not prepared for dynamically changing NUMA distances or changing memory and caching structures**
- → Changing either cluster mode or memory mode requires a rebuild of tag directories
  - Requires reboot
  - Takes 15-20 minutes
Selection of Cluster Modes via Script @ LRZ

mcct03:~ # sudo /usr/local/bin/SetKnlMode
ERROR: A valid mode was not specified.

Usage: SetKnlMode [-fq] -m MODE

- f - Force setting of mode, even if it appears to be already set.
- q - Reboot after 1 minutes, instead of 5 minutes.
- m MODE - Name the mode to change to. Available modes are:
  - CacheQuadrant
  - CacheSNC-4
  - FlatQuadrant
  - FlatSNC-4
  - HybridSNC-4

Uses Supermicro Update Manager BIOS Management:
/usr/local/sbin/sum -c ChangeBiosCfg --file filename

CINECA MARCONI Memory Modes


- Following the suggestions of the Intel experts, we finally adopted one configuration only for all the KNL racks serving the knldebug and knlprod (academic) queues, namely:
  cache/quadrant

- The queues serving the Marconi FUSION partition allow instead the use of nodes in
  flat/quadrant or cache/quadrant modes

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Using the memkind library

- https://github.com/memkind/memkind

The *memkind* library is a **user extensible heap manager** built on top of *jemalloc* (http://jemalloc.net/) which enables control of memory characteristics and a **partitioning of the heap between kinds of memory**.

- The *memkind* library delivers two interfaces:
  - *hbwmalloc.h* - recommended for high-bandwidth memory use cases (stable) → `man memkind`
  - *memkind.h* - generic interface for more complex use cases (partially unstable) → `man hbwmalloc`

### SYNOPSIS
```
#include <hbwmalloc.h>
int hbw_check_available(void);
void* hbw_malloc(size_t size);
void* hbw_realloc(void *ptr, size_t size);
void hbw_free(void *ptr);
int hbw_posix_memalign(void **memptr, size_t alignment, size_t size);
int hbw_posix_memalign_psize(void **memptr, size_t alignment, size_t size,
                              hbw_pagesize_t pagesize);
hbw_policy_t hbw_get_policy(void);
int hbw_set_policy(hbw_policy_t mode);
int hbw_verify_memory_region(void *addr, size_t size, int flags);
```

**Details:** `man hbwmalloc`
Using the memkind library: Policies

`hbw_set_policy()` sets the current fallback policy. The policy can be modified only once in the lifetime of an application and before calling `hbw_*alloc()` or `hbw_posix_memalign()` function. Note: If the policy is not set, than HBW_POLICY_PREFERRED will be used by default.

**HBW_POLICY_BIND**
If insufficient high bandwidth memory from the nearest NUMA node is available to satisfy a request, the allocated pointer is set to NULL and errno is set to ENOMEM. If insufficient high bandwidth memory pages are available at fault time the Out Of Memory (OOM) killer is triggered. Note that pages are faulted exclusively from the high bandwidth NUMA node nearest at time of allocation, not at time of fault.

**HBW_POLICY_PREFERRED**
If insufficient memory is available from the high bandwidth NUMA node closest at allocation time, fall back to standard memory (default) with the smallest NUMA distance.

**HBW_POLICY_INTERLEAVE**
Interleave faulted pages from across all high bandwidth NUMA nodes using standard size pages (the Transparent Huge Page feature is disabled).

Using the memkind library: alloc

**Traditional:**
```c
#include <stdlib.h>
...
double *A;
A = (double*) malloc(sizeof(double) * N);
...
Free(A);
```

**Memkind library:**
```c
#include <hbwmalloc.h>
...
double *A;
A = (double*) hbw_malloc(sizeof(double) * N);
...
hbw_free(A);
```
Using the memkind library: posix_memalign

**Traditional:**
```c
#include <stdlib.h>
...
int ret; double *A;
ret = posix_memalign((void *)A, 64, sizeof(double)*N);
if (ret!=0) //error
...
free(A);
```

**Memkind library:**
```c
#include <hbwmalloc.h>
...
int ret; double *A;
ret = hbw_posix_memalign((void*) A, 64, sizeof(double)*N);
if (ret!=0) //error
...
hbw_free(A);
```

Intel Fortran Extensions for MCDRAM

**Traditional:**
```fortran
real, allocatable :: A(:)
...
ALLOCATE (A(1:1024))
```

**MCDRAM:**
```fortran
real, allocatable :: A(:)
!DIR$ ATTRIBUTES FASTMEM :: A
!DIR$ ATTRIBUTES FASTMEM, ALIGN:64 :: A ! Alternative for alignment
...
ALLOCATE (A(1:1024))
```

**Alternativ:**
```fortran
real, allocatable :: A(:)
!dir# FASTMEM
ALLOCATE (A(1:1024))
```

FOR_SET_FASTMEM_POLICY(...) to change policy.
OpenMP 5.0 MCDRAM Support

- Memory Management Support for OpenMP 5.0
- Support for new types of memory: High Bandwidth Memory, Non-volatile memory etc.


- This Technical Report augments the OpenMP TR 4 document with language features for managing memory on systems with heterogeneous memories.
- To be released approx. Nov. 2018

Stampede2 Supercomputer

- The following slides contain material obtained on the Stampede2 Supercomputer at Texas Advanced Computing Center, The University of Texas at Austin
- [https://portal.tacc.utexas.edu/user-guides/stampede2](https://portal.tacc.utexas.edu/user-guides/stampede2)
- Stampede2 is the flagship supercomputer at the Texas Advanced Computing Center (TACC). It will enter full production in the Fall 2017 as an 18-petaflop national resource that builds on the successes of the original Stampede system it replaces.
- Thanks to TACC for access to Stempede2 during an ISC‘17 Tutorial.
Stampede2 Supercomputer

Model: Intel Xeon Phi 7250
Total cores per KNL node: 68 cores on a single socket
Hardware threads per core: 4
Hardware threads per node: $68 \times 4 = 272$
Clock rate: 1.4GHz

RAM:
96GB DDR4 plus 16GB high-speed MCDRAM. Configurable in two important ways; see Programming and Performance for more info.
All but 508 KNL nodes have a 132GB /tmp partition on a 200GB Solid State Drive (SSD). The 508 KNLs originally installed as the
Local storage: Stampede1 KNL sub-system each have a 58GB /tmp partition on 112GB SSDs. The latter nodes currently make up the flat-quadrant and flat-snc4 queues.
Useful commands and system files

- `/proc/cpuinfo`
- `/proc/meminfo`
- `numastat -H`
- `numastat -m` (includes huge page info, too)
- `numastat -p pid`
- `/sys/devices/system/node/node*/meminfo` and other files
- `/usr/bin/memkind-hbw-nodes`
/proc/meminfo

- **Flat-Quadrant Mode:**
  
  MemTotal: 115218908 kB  
  MemFree: 108756608 kB  
  MemAvailable: 108562240 kB

- **Cache-Quadrant Mode:**
  
  MemTotal: 98696336 kB  
  MemFree: 92462428 kB  
  MemAvailable: 92282108 kB

- **Flat-SNC-4 Mode:**
  
  MemTotal: 115217380 kB  
  MemFree: 109216500 kB  
  MemAvailable: 108983732 kB

numactl –H in Flat-Quadrant Mode

available: 2 nodes (0-1)  
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26  
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54  
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82  
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107  
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128  
129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149  
150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170  
171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191  
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212  
213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233  
234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254  
255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271  
node 0 size: 98207 MB  
node 0 free: 90483 MB  
nodes 1 cpus:  
node 1 size: 16384 MB  
node 1 free: 15723 MB  
nodes distances:  
node 0 1  
0: 10 31  
1: 31 10
numactl –H in Cache Mode

available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149
150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212
213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254
255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
node 0 size: 98199 MB
node 0 free: 90294 MB
node distances:
node   0
0: 10

numactl –H in Flat-SNC4 Mode

available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149
150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212
213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254
255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
node 0 size: 24479 MB
node 0 free: 21852 MB
node 1 cpus: 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78
node 1 size: 24576 MB
node 1 free: 22867 MB
node 2 cpus: 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
110
node 2 size: 24576 MB
node 2 free: 22287 MB
node 3 cpus: 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103
104 105 106 107 108 109 110
node 3 size: 24576 MB
node 3 free: 23144 MB
numactl -H in Flat-SNC4 Mode contnd.

node 4 cpus:
node 4 size: 4096 MB
node 4 free: 3968 MB
node 5 cpus:
node 5 size: 4096 MB
node 5 free: 3976 MB
node 6 cpus:
node 6 size: 4096 MB
node 6 free: 3976 MB
node 7 cpus:
node 7 size: 4096 MB
node 7 free: 3975 MB

Distances:
10 "near" DDR
21 "far" DDR
31 "near" MCDRAM
41 "far" MCDRAM

Affinitization of DDR and MCDRAM to the divisons of the KNL!

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

/usr/bin/memkind-hbw-nodes

Flat-Quadrant:
c455-001.stampede2(17)$ memkind-hbw-nodes
1
c455-001.stampede2(18)$

Flat-SNC4:
c463-001.stampede2(1)$ memkind-hbw-nodes
4,5,6,7
c463-001.stampede2(2)$

Cache:
c403-001.stampede2(2)$ memkind-hbw-nodes
c403-001.stampede2(3)$
### STREAM Benchmark in Cache Mode

**export OMP_NUM_THREADS=1**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>17843.4</td>
<td>0.009051</td>
<td>0.008967</td>
<td>0.009453</td>
</tr>
<tr>
<td>Scale:</td>
<td>14305.0</td>
<td>0.011287</td>
<td>0.011185</td>
<td>0.011523</td>
</tr>
<tr>
<td>Add:</td>
<td>15736.8</td>
<td>0.015452</td>
<td>0.015251</td>
<td>0.015739</td>
</tr>
<tr>
<td>Triad:</td>
<td>15622.9</td>
<td>0.015512</td>
<td>0.015362</td>
<td>0.015851</td>
</tr>
</tbody>
</table>

**export OMP_NUM_THREADS=68**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>263585.5</td>
<td>0.000624</td>
<td>0.000607</td>
<td>0.000640</td>
</tr>
<tr>
<td>Scale:</td>
<td>269297.2</td>
<td>0.000612</td>
<td>0.000594</td>
<td>0.000631</td>
</tr>
<tr>
<td>Add:</td>
<td>325244.9</td>
<td>0.000772</td>
<td>0.000738</td>
<td>0.000798</td>
</tr>
<tr>
<td>Triad:</td>
<td>308499.2</td>
<td>0.000803</td>
<td>0.000778</td>
<td>0.000872</td>
</tr>
</tbody>
</table>

---

### STREAM Benchmark in Flat-Quadrant Mode

**c455-001.stampede2(27)$ export OMP_NUM_THREADS=1**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>18344.3</td>
<td>0.008848</td>
<td>0.008722</td>
<td>0.009068</td>
</tr>
<tr>
<td>Scale:</td>
<td>15184.7</td>
<td>0.010682</td>
<td>0.010537</td>
<td>0.011006</td>
</tr>
<tr>
<td>Add:</td>
<td>16788.1</td>
<td>0.014392</td>
<td>0.014296</td>
<td>0.014504</td>
</tr>
<tr>
<td>Triad:</td>
<td>16712.0</td>
<td>0.014462</td>
<td>0.014361</td>
<td>0.014548</td>
</tr>
</tbody>
</table>

**c455-001.stampede2(30)$ ./stream**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>18377.9</td>
<td>0.008787</td>
<td>0.008706</td>
<td>0.008876</td>
</tr>
<tr>
<td>Scale:</td>
<td>15207.8</td>
<td>0.010655</td>
<td>0.010521</td>
<td>0.010875</td>
</tr>
<tr>
<td>Add:</td>
<td>16677.2</td>
<td>0.014519</td>
<td>0.014391</td>
<td>0.014714</td>
</tr>
<tr>
<td>Triad:</td>
<td>16724.8</td>
<td>0.014447</td>
<td>0.014350</td>
<td>0.014541</td>
</tr>
</tbody>
</table>

**c455-001.stampede2(21)$ numactl -m 0 ./stream**

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>17999.9</td>
<td>0.009024</td>
<td>0.008889</td>
<td>0.009474</td>
</tr>
<tr>
<td>Scale:</td>
<td>14455.0</td>
<td>0.011269</td>
<td>0.011069</td>
<td>0.011690</td>
</tr>
<tr>
<td>Add:</td>
<td>15846.7</td>
<td>0.015331</td>
<td>0.015145</td>
<td>0.016039</td>
</tr>
<tr>
<td>Triad:</td>
<td>15850.0</td>
<td>0.015344</td>
<td>0.015142</td>
<td>0.015800</td>
</tr>
</tbody>
</table>

**c455-001.stampede2(24)$ numactl -m 1 ./stream**

---

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
### STREAM Benchmark in Flat-Quadrant Mode

```bash
c455-001.stampede2(27)$ export OMP_NUM_THREADS=68
c455-001.stampede2(30)$ ./stream
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>82768.7</td>
<td>0.001939</td>
<td>0.001933</td>
<td>0.001945</td>
</tr>
<tr>
<td>Scale:</td>
<td>82901.6</td>
<td>0.001937</td>
<td>0.001930</td>
<td>0.001947</td>
</tr>
<tr>
<td>Add:</td>
<td>88332.1</td>
<td>0.002731</td>
<td>0.002717</td>
<td>0.002741</td>
</tr>
<tr>
<td>Triad:</td>
<td>88425.2</td>
<td>0.002732</td>
<td>0.002714</td>
<td>0.002757</td>
</tr>
</tbody>
</table>

```bash
ac455-001.stampede2(28)$ numactl -m 0 ./stream
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>83334.0</td>
<td>0.001937</td>
<td>0.001920</td>
<td>0.001961</td>
</tr>
<tr>
<td>Scale:</td>
<td>82768.7</td>
<td>0.001938</td>
<td>0.001933</td>
<td>0.001950</td>
</tr>
<tr>
<td>Add:</td>
<td>88131.1</td>
<td>0.002742</td>
<td>0.002723</td>
<td>0.002769</td>
</tr>
<tr>
<td>Triad:</td>
<td>88138.8</td>
<td>0.002745</td>
<td>0.002723</td>
<td>0.002754</td>
</tr>
</tbody>
</table>

```bash
ac455-001.stampede2(29)$ numactl -m 1 ./stream
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>397093.9</td>
<td>0.000434</td>
<td>0.000403</td>
<td>0.000473</td>
</tr>
<tr>
<td>Scale:</td>
<td>426901.2</td>
<td>0.000400</td>
<td>0.000375</td>
<td>0.000428</td>
</tr>
<tr>
<td>Add:</td>
<td>433146.7</td>
<td>0.000578</td>
<td>0.000554</td>
<td>0.000602</td>
</tr>
<tr>
<td>Triad:</td>
<td>345328.6</td>
<td>0.000835</td>
<td>0.000695</td>
<td>0.000944</td>
</tr>
</tbody>
</table>
```

### STREAM Benchmark in Flat-SNC4 Mode

```bash
c455-001.stampede2(27)$ export OMP_NUM_THREADS=68
c455-001.stampede2(30)$ ./stream
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>74773.1</td>
<td>0.002148</td>
<td>0.002140</td>
<td>0.002189</td>
</tr>
<tr>
<td>Scale:</td>
<td>78970.2</td>
<td>0.002031</td>
<td>0.002026</td>
<td>0.002038</td>
</tr>
<tr>
<td>Add:</td>
<td>79156.5</td>
<td>0.003040</td>
<td>0.003032</td>
<td>0.003066</td>
</tr>
<tr>
<td>Triad:</td>
<td>79337.4</td>
<td>0.003032</td>
<td>0.003025</td>
<td>0.003050</td>
</tr>
</tbody>
</table>

```bash
c455-001.stampede2(30)$ numactl -m 0 ./stream
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>36012.3</td>
<td>0.000446</td>
<td>0.000443</td>
<td>0.000475</td>
</tr>
<tr>
<td>Scale:</td>
<td>36093.6</td>
<td>0.000444</td>
<td>0.000443</td>
<td>0.000446</td>
</tr>
<tr>
<td>Add:</td>
<td>38903.7</td>
<td>0.000618</td>
<td>0.000616</td>
<td>0.000621</td>
</tr>
<tr>
<td>Triad:</td>
<td>38942.8</td>
<td>0.000617</td>
<td>0.000613</td>
<td>0.000620</td>
</tr>
</tbody>
</table>

```bash
c455-001.stampede2(30)$ numactl -m 1 ./stream
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Best Rate MB/s</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>36381.3</td>
<td>0.000448</td>
<td>0.000439</td>
<td>0.000480</td>
</tr>
<tr>
<td>Scale:</td>
<td>36430.6</td>
<td>0.000453</td>
<td>0.000439</td>
<td>0.000503</td>
</tr>
<tr>
<td>Add:</td>
<td>39525.4</td>
<td>0.000655</td>
<td>0.000672</td>
<td>0.000799</td>
</tr>
<tr>
<td>Triad:</td>
<td>39519.2</td>
<td>0.000619</td>
<td>0.000607</td>
<td>0.000678</td>
</tr>
</tbody>
</table>
### STREAM Benchmark in Flat-SNC4 Mode

```bash
c455-001.stampede2(27)$ export OMP_NUM_THREADS=68
c455-001.stampede2(27)$ numactl -m 2 ./scratch
Function | Best Rate MB/s | Avg time | Min time | Max time
Copy:    | 34669.0       | 0.004625 | 0.004615 | 0.004635
Scale:   | 34692.3       | 0.004621 | 0.004612 | 0.004630
Add:     | 38199.5       | 0.006288 | 0.006283 | 0.006297
Triad:   | 38215.4       | 0.006287 | 0.006280 | 0.006298
```

```bash
c455-001.stampede2(27)$ numactl -m 3 ./scratch
Function | Best Rate MB/s | Avg time | Min time | Max time
Copy:    | 34535.2       | 0.004650 | 0.004633 | 0.004665
Scale:   | 34601.1       | 0.004644 | 0.004624 | 0.004664
Add:     | 38161.8       | 0.006322 | 0.006289 | 0.006349
Triad:   | 38198.0       | 0.006324 | 0.006283 | 0.006346
```

```bash
c455-001.stampede2(27)$ numactl -m 4 ./scratch
Function | Best Rate MB/s | Avg time | Min time | Max time
Copy:    | 105186.3      | 0.001539 | 0.001521 | 0.001560
Scale:   | 105334.9      | 0.001541 | 0.001519 | 0.001558
Add:     | 101516.0      | 0.002378 | 0.002364 | 0.002396
Triad:   | 102040.8      | 0.002371 | 0.002352 | 0.002398
```

```bash
c455-001.stampede2(27)$ numactl -m 5 ./scratch
Function | Best Rate MB/s | Avg time | Min time | Max time
Copy:    | 107442.9      | 0.001539 | 0.001521 | 0.001560
Scale:   | 107236.9      | 0.001541 | 0.001519 | 0.001558
Add:     | 104206.3      | 0.002312 | 0.002303 | 0.002325
Triad:   | 104672.2      | 0.002295 | 0.002293 | 0.002304
```

```bash
c455-001.stampede2(27)$ numactl -m 6 ./scratch
Function | Best Rate MB/s | Avg time | Min time | Max time
Copy:    | 98834.9       | 0.001627 | 0.001619 | 0.001633
Scale:   | 98762.1       | 0.001628 | 0.001620 | 0.001635
Add:     | 93823.6       | 0.002569 | 0.002558 | 0.002586
Triad:   | 94086.6       | 0.002561 | 0.002551 | 0.002572
```

```bash
c455-001.stampede2(27)$ numactl -m 7 ./scratch
Function | Best Rate MB/s | Avg time | Min time | Max time
Copy:    | 99938.7       | 0.001608 | 0.001601 | 0.001615
Scale:   | 99317.5       | 0.001614 | 0.001611 | 0.001620
Add:     | 94599.5       | 0.002545 | 0.002537 | 0.002550
Triad:   | 94929.6       | 0.002535 | 0.002528 | 0.002543
```

---

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 6000000000
int main(int argc, char **argv)
{
    float *list;
    long long int i;

    list = (float *) malloc(N * sizeof(float));
    if(list != NULL) {printf("memory is reserved\n");
    }else {printf("No memory free.\n");}

    printf("Sizeof float:  %i Bytes\n",sizeof(float));
    fprintf("Sizeof list:  %lli Bytes\n",sizeof(float)*N);
    printf("Sizeof list:  %lli GB\n",sizeof(float)*N/1024/1024/1024);

    for(i=0; i<N; i++) list[i] = i;
    return 0;
}

c455-002.stampede2(14)$ time numactl -m 0 ./alloc
memory is reserved
Sizeof float:  4 Bytes
Sizeof list:  24000000000 Bytes
Sizeof list:  22 GB
real  1m38.960s
user  1m35.385s
sys   0m3.571s
c455-002.stampede2(15)$
Oversubscription of MCDRAM

```
c455-002.stampede2(11)$ time numactl -m 1 ./alloc
memory is reserved
Sizeof float:  4 Bytes
Sizeof list:  24000000000 Bytes
Sizeof list:  22 GB
Killed
real    1m11.186s
user    1m4.327s
sys     0m6.472s
c455-002.stampede2(12)$
```

c455-002.stampede2(15)$ time numactl --preferred=1 ./alloc
memory is reserved
Sizeof float:  4 Bytes
Sizeof list:  24000000000 Bytes
Sizeof list:  22 GB
real    1m38.754s
user    1m35.031s
sys     0m3.714s
c455-002.stampede2(16)$

```
26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
```

Oversubscription of MCDRAM

● Using the memkind library:

```c
#include <hbwmalloc.h>
...
list = (float *) hbw_malloc(N * sizeof(float));
```

c455-001.stampede2(6)$ icc -O0 alloc2.c -lmemkind -o alloc2
c455-001.stampede2(7)$ ./alloc2
```

memory is reserved
Sizeof float:  4 Bytes
Sizeof list:  24000000000 Bytes
Sizeof list:  22 GB
c455-001.stampede2(8)$

```
As default fallback policy is HBW_POLICY_PREFERRED!
```

```
26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
```
Performance

Cluster and Memory Modes Performance Comparison

- Flat-A2A DDR4
- Cache-A2A
- Flat-A2A MCDRAM
- Flat-Quad DDR4
- Cache-Quad
- Flat-Quad MCDRAM
- Flat-SNc4 DDR4
- Cache-SNc4
- Flat-SNc4 MCDRAM

From TACC ISC’17 Tutorial, Baseline = Flat-A2A DDR4

References


- **Tutorial**: "Introduction to Manycore Programming", Texas Advanced Computing Center, 2017. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License. https://creativecommons.org/licenses/by-nc/3.0/

- **Tutorial**: “Introduction to KNL and the ARCHER KNL Cluster”, 2017, Adrian Jackson, EPCC, licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US
Intel Xeon Phi Programming Models: Intel Language Extensions for Offload (LEO) I

- Intel Language Extensions for Offload (LEO)
- OpenMP 4.x Offloading
- “Mine Yours Ours” (MYO) virtual shared memory model
Intel Offload Directives

- Syntax:

  - **C:**
    
    ```
    #pragma offload target(mic) <clauses>
    <statement block>
    ```
    
  - **Fortran:**
    
    ```
    !DIR$ offload target(mic) <clauses>
    <statement>
    ```
    ```
    !DIR$ omp offload target(mic) <clauses>
    <OpenMP construct>
    ```

Intel Offload Directive

- **C:**
  
  Pragma can be before any statement, including a compound statement or an OpenMP parallel pragma

- **Fortran:**
  
  - If OMP is specified: the next line, other than a comment, must be an OpenMP PARALLEL, PARALLEL SECTIONS, or PARALLEL DO directive.
  
  - If OMP is not specified, next line must:
    
    - An OpenMP* PARALLEL, PARALLEL SECTIONS, or PARALLEL DO directive
    - A CALL statement
    - An assignment statement where the right side only calls a function
Intel Offload

- Implements the following steps:
  
  1. Memory allocation on the MIC
  2. Data transfer from the host to the MIC
  3. Execution on the MIC
  4. Data transfer from the MIC to the host
  5. Memory deallocation on MIC

---

Intel Offload: Hello World in C

```c
#include <stdio.h>
int main (int argc, char* argv[]) {

#pragma offload target(mic)
{
    printf("MIC: Hello world from MIC.\n");
}
printf( "Host: Hello world from host.\n");
}
```

---
**Intel Offload: Hello World in Fortran**

```fortran
PROGRAM HelloWorld

!DIR$ offload begin target(MIC)
PRINT *, 'MIC: Hello world from MIC'
!DIR$ end offload

PRINT *, 'Host: Hello world from host'
END
```

**Intel Offload: Hello World in C**

```
lu65fok@login12:~/tests> icpc offload1.c -o offload1
lu65fok@login12:~/tests> ./offload1
offload error: cannot offload to MIC - device is not available

lu65fok@i01r13c01:~/tests> ./offload1
Host: Hello world from host.
MIC: Hello world from MIC.
```
Intel Offload: Hello World in Fortran

lu65fok@login12:~/tests> ifort offload1.f90 -o offload1

lu65fok@login12:~/tests> ./offload1
offload error: cannot offload to MIC - device is not available

lu65fok@i01r13c01:~/tests> ./offload1
Host: Hello world from host.
MIC: Hello world from MIC.

#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[]) {
    char hostname[100];
    gethostname(hostname, sizeof(hostname));

    #pragma offload target(mic)
    {
        char michostname[100];
        gethostname(michostname, sizeof(michostname));
        printf("MIC: Hello world from MIC. I am %s and I have %ld logical cores. I was
called from host: %s \n", michostname, sysconf(_SC_NPROCESSORS_ONLN), hostname);
    }

Intel Offload: Hello World with Hostnames

#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[]) {
    char hostname[100];
    gethostname(hostname, sizeof(hostname));

    #pragma offload target(mic)
    {
        char michostname[100];
        gethostname(michostname, sizeof(michostname));
        printf("MIC: Hello world from MIC. I am %s and I have %ld logical cores. I was
called from host: %s \n", michostname, sysconf(_SC_NPROCESSORS_ONLN), hostname);
    }
Intel Offload: Hello World with Hostnames

lu65fok@login12:~/tests> icpc offload.c -o offload

lu65fok@i01r13c01:~/tests> ./offload

Host: Hello world from host. I am i01r13c01 and I have 32 logical cores.
MIC: Hello world from MIC. I am i01r13c01-mic0 and I have 240 logical cores. I was called from host: i01r13c01

Intel Offload: -offload=optional / mandatory

lu65fok@login12:~/tests> icpc -offload=optional offload.c -o offload

lu65fok@login12:~/tests> ./offload

MIC: Hello world from MIC. I am login12 and I have 16 logical cores. I was called from host: login12
Host: Hello world from host. I am login12 and I have 16 logical cores.

lu65fok@login12:~/tests> icpc -offload=mandatory offload.c -o offload
lu65fok@login12:~/tests> ./offload
offload error: cannot offload to MIC - device is not available
Intel Offload: -none

lu65fok@login12:~/tests> icpc -offload=none offload.c -o offload
offload.c(13): warning #161: unrecognized #pragma
#pragma offload target(mic)
^
lu65fok@login12:~/tests>

lu65fok@i01r13c01:~/tests> ./offload
MIC: Hello world from MIC. I am i01r13c01 and I have 32 logical cores.
I was called from host: i01r13c01
Host: Hello world from host. I am i01r13c01 and I have 32 logical cores.

#include <stdio.h>
#include <stdlib.h>

int main(){
#pragma offload target (mic)
{
    system("command");
}
}
Intel Offload: system("set")

lu65fok@i01r13c01:~/.tests> ./system
BASH=/bin/sh
BASHOPTS=cmdhist:extquote:force_fignore:hostcomplete:interactive_comments:progc
ptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_EXECUTION_STRING=set
BASH_LINE=()
BASH_SOURCE=()
BASH_VERSION='4.2.10(1)-release'
COI_LOG_PORT=65535
COI_SCIF_SOURCE_NODE=0
DIRSTACK=()
ENV_PREFIX=MIC
EUID=400
GROUPS=()

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

Intel Offload: system("set")

HOSTNAME=i01r13c01-mic0
HOSTTYPE=k1om
IFS=''
LIBRARY_PATH=/lrz/sys/intel/compiler140_144/composer_xe_2013_sp1.2.144/tbb/lib/mic:/lrz/sys/intel/compiler1
40_144/composer_xe_2013_sp1.2.144/tbb/lib/mic:/lrz/sys/intel/compiler140_144/composer_xe_2013_sp1.2.144/tbb/lib/mic:/lrz/sys/intel/compiler140_144/composer_xe_2013_sp1.2.144/tbb/lib/mic
MACHTYPE=k1om-mpss-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/usr/bin:/bin
POSIXLY_CORRECT=y
PPID=37141
PS4="+ "
PWD=/var/volatile/tmp/coi_procs/1/37141
SHELL=/bin/false
SHELLOPTS=braceexpand:hashall:interactive-comments:posix
SHLVL=1
TERM=dumb
UID=400
_=sh

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Intel Offload: system(command)

```
#pragma offload target (mic)
{
    system("hostname");
    system("uname -a");
    system("whoami");
    system("id");
}
```

lu65fok@i01r13c01:~/tests> ./system
i01r13c01-mic0
Linux i01r13c01-mic0 2.6.38.8+mpss3.1.2 #1 SMP Wed Dec 18 19:09:36 PST 2013 k1om GNU/Linux
micuser
uid=400(micuser) gid=400(micuser)

Offload: Using several MIC Coprocessors

- To query the number of coprocessors:
  
  ```
  int nmics = __Offload_number_of_devices()
  ```

- To specify which coprocessor \( n < nmics \) should do the computation:
  
  ```
  #pragma offload target(mic:n)
  ```

- If \( n > nmics \) then coprocessor \( n \% nmics \) is used

- Important for:
  - Asynchronous offloads
  - Coprocessor-Persistent data
Offloading OpenMP Computations

- C/C++ & OpenMP:
  ```
  #pragma offload target(mic)
  #pragma omp parallel for
  for (int i=0;i<n;i++) {
    a[i]=c*b[i]+d;
  }
  ```

- Fortran & OpenMP
  ```
  !$DIR$ offload target(mic)
  !$OMP PARALLEL DO
  do i = 1, n
  a(i) = c*b(i) + d
  end do
  !$omp END PARALLEL DO
  ```

Functions and Variables on the MIC

- C:
  - `__attribute__((target(mic)))` variables / function
  - `__declspec (target(mic))` variables / function
  - `#pragma offload_attribute(push, target(mic))`
    … multiple lines with variables / functions
    `#pragma offload_attribute(pop)`

- Fortran:
  ```
  !$DIR$ attributes offload:mic:: variables / function
  ```
Functions and Variables on the MIC

```c
#pragma offload_attribute(push,target(mic))
const int n=100;
int a[n], b[n], c, d;
void myfunction(int* a, int* b, int c, int d){
    for (int i=0;i<n;i++) {
        a[i]=c*b[i]+d;
    }
}
#pragma offload_attribute(pop)
int main (int argc, char* argv[]){
    #pragma offload target(mic)
    { myfunction(a,b,c,d);
    }
}
```

Intel Offload Clauses

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple coprocessors</td>
<td><code>target(mic[:unit])</code></td>
<td>Select specific coprocessors</td>
</tr>
<tr>
<td>Conditional offload</td>
<td><code>if (condition) / mandatory</code></td>
<td>Select coprocessor or host compute</td>
</tr>
<tr>
<td>Inputs</td>
<td><code>in(var-list modifiers_opt)</code></td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>Outputs</td>
<td><code>out(var-list modifiers_opt)</code></td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td><code>inout(var-list modifiers_opt)</code></td>
<td>Copy host to coprocessor and back when offload completes</td>
</tr>
<tr>
<td>Non-copied data</td>
<td><code>nocopy(var-list modifiers_opt)</code></td>
<td>Data is local to target</td>
</tr>
<tr>
<td>Async. Offload</td>
<td><code>signal(signal-slot)</code></td>
<td>Trigger asynchronous Offload</td>
</tr>
<tr>
<td>Async. Offload</td>
<td><code>wait(signal-slot)</code></td>
<td>Wait for completion</td>
</tr>
</tbody>
</table>
### Intel Offload Modifier Options

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify copy length</td>
<td>length(N)</td>
<td>Copy N elements of pointer’s type</td>
</tr>
<tr>
<td>Coprocessor memory allocation</td>
<td>alloc_if (bool)</td>
<td>Allocate coprocessor space on this offload (default: TRUE)</td>
</tr>
<tr>
<td>Coprocessor memory release</td>
<td>free_if (bool)</td>
<td>Free coprocessor space at the end of this offload (default: TRUE)</td>
</tr>
<tr>
<td>Array partial allocation &amp; variable relocation</td>
<td>alloc (array-slice)</td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
<tr>
<td></td>
<td>in (var-expr)</td>
<td></td>
</tr>
</tbody>
</table>

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

### Intel Offload: Data Movement

- ![](https://via.placeholder.com/150)
  
  #pragma offload target(mic) in(in1,in2,…)
  out(out1,out2,…) inout(inout1,inout2,…)

- **At Offload start:**
  - Allocate Memory Space on MIC for all variables
  - Transfer in/inout variables from Host to MIC

- **At Offload end:**
  - Transfer out/inout variables from MIC to Host
  - Deallocate Memory Space on MIC for all variables

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Intel Offload: Data Movement

- data = (double*)malloc(n*sizeof(double));
- #pragma offload target(mic) in(data:length(n))
- Copies n doubles to the coprocessor, not n * sizeof(double) Bytes
- ditto for out() and inout()

An example for Offloading: Offloading Code

```c
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
{
    #pragma omp parallel for
    for( i = 0; i < n; i++ ) {
        for( k = 0; k < n; k++ ) {
            #pragma vector aligned
            #pragma ivdep
            for( j = 0; j < n; j++ ) {
                //c[i][j] = c[i][j] + a[i][k]*b[k][j];
                c[i*n+j] = c[i*n+j] + a[i*n+k]*b[k*n+j];
            }
        }
    }
}
```
Vectorisation Diagnostics

```bash
classicuser@login12:~/tests> icc -vec-report2 -openmp offloadmul.c -ooffloadmul offloadmul.c: (col. 5) remark: LOOP WAS VECTORIZED
offloadmul.c: (col. 3) remark: loop was not vectorized: not inner loop
offloadmul.c: (col. 2) remark: LOOP WAS VECTORIZED
offloadmul.c: (col. 7) remark: loop was not vectorized: not inner loop
offloadmul.c: (col. 5) remark: loop was not vectorized: not inner loop
offloadmul.c: (col. 9) remark: loop was not vectorized: existence of vector dependence
offloadmul.c: (col. 5) remark: loop was not vectorized: not inner loop
offloadmul.c: (col. 2) remark: *MIC* LOOP WAS VECTORIZED
offloadmul.c: (col. 7) remark: *MIC* loop was not vectorized: not inner loop
offloadmul.c: (col. 5) remark: *MIC* loop was not vectorized: not inner loop
```


Intel Offload: Example

```c
__attribute__((target(mic))) void mxm( int n, double * restrict a, double * restrict b,
double *restrict c )
{
    int i,j,k;
    for( i = 0; i < n; i++ ) {
        ...
    }
}

main(){
    ...
    #pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
    {  
        mxm(n,a,b,c);
    }
}
```

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Offload Diagnostics

u65fok@i01r13c06:~/tests> export OFFLOAD_REPORT=2

lu65fok@i01r13c06:~/tests> ./offloadmul
[Offload] [MIC 0] [File] offloadmul.c
[Offload] [MIC 0] [Line] 50
[Offload] [MIC 0] [Tag] Tag 0
[Offload] [HOST] [Tag 0] [CPU Time] 51.927456(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data] 24000016 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time] 50.835065(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data] 8000016 (bytes)

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

Offload Diagnostics

lu65fok@i01r13c06:~/tests> export H_TRACE=1

lu65fok@i01r13c06:~/tests> ./offloadmul
HOST: Offload function
__offload_entry_offloadmul_c_50mainicc638762473Jnx4JU,
is_empty=0, #varDescs=7, #waits=0, signal=None
HOST: Total pointer data sent to target: [24000000] bytes
HOST: Total copyin data sent to target: [16] bytes
HOST: Total pointer data received from target: [8000000] bytes
MIC0: Total copyin data received from host: [16] bytes
MIC0: Total copyout data sent to host: [16] bytes
HOST: Total copyout data received from target: [16] bytes

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Offload Diagnostics

```bash
lu65fok@i01r13c06:~/tests> export H_TIME=1

lu65fok@i01r13c06:~/tests> ./offloadmul
[Offload] [MIC 0] [File]         offloadmul.c
[Offload] [MIC 0] [Line]         50
[Offload] [MIC 0] [Tag]          Tag 0
[Offload] [HOST] [Tag 0] [CPU Time] 51.920016(seconds)
[Offload] [MIC 0] [Tag 0] [MIC Time] 50.831497(seconds)
```

Environment Variables

- Host environment variables are automatically forwarded to the coprocessor when offload mode is used.
- To avoid names collisions:
  - Set `MIC_ENVIRONMENT_PREFIX=MIC` on the host
  - Then only names with prefix MIC_ are forwarded to the coprocessor with prefix stripped
  - Exception: `MIC_LD_LIBRARY_PATH` is never passed to the coprocessor.
  - Value of `LD_LIBRARY_PATH` cannot be changed via forwarding of environment variables.
Environment Variables on the MIC

```
#include <stdio.h>
#include <stdlib.h>

int main()

#pragma offload target (mic)
{
    char* varmic = getenv("VAR");
    if (varmic) {
        printf("VAR=%s on MIC.\n", varmic);
    } else {
        printf("VAR is not defined on MIC.\n");
    }

    char* varhost = getenv("VAR");
    if (varhost) {
        printf("VAR=%s on host.\n", varhost);
    } else {
        printf("VAR is not defined on host.\n");
    }
}
```

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

Environment Variables on the MIC

```
lu65fok@i01r13c01:~/tests> ./env
VAR is not defined on host.
VAR is not defined on MIC.
lu65fok@i01r13c01:~/tests> export VAR=299792458
lu65fok@i01r13c01:~/tests> ./env
VAR=299792458 on host.
VAR=299792458 on MIC.
lu65fok@i01r13c01:~/tests> export MIC_ENV_PREFIX=MIC
lu65fok@i01r13c01:~/tests> ./env
VAR=299792458 on host.
VAR is not defined on MIC.
lu65fok@i01r13c01:~/tests> export MIC_VAR=3.141592653
lu65fok@i01r13c01:~/tests> ./env
VAR=299792458 on host.
VAR=3.141592653 on MIC.
```

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
The Preprocessor Macro __MIC__

- The macro __MIC__ is only defined in code version for MIC, not in the fallback version for the host.
- Allows to check where the code is running.
- Allows to write multiversioned code.
- __MIC__ also defined in native mode.

```c
#pragma offload target(mic)
{
    #ifdef __MIC__
        printf("Hello from MIC (offload succeeded).\n");
    #else
        printf("Hello from host (offload to MIC failed!).\n");
    #endif
}
```

```
lux5fok@login12:~/tests> icpc -offload=optional offload-mic.c
lux5fok@login12:~/tests> ./a.out
Hello from host (offload to MIC failed!).
lux5fok@i01r13c06:~/tests> ./a.out
Hello from MIC (offload succeeded).
```
Lab: Offload Mode I

Intel Xeon Phi Programming Models: Intel Language Extensions for Offload (LEO) II
Data Traffic without Computation

- 2 possibilities:
  - Blank body of `#pragma offload`, i.e.
    ```c
    #pragma offload target(mic) in (data: length(n))
    {}
    ```
  - Use a special pragma `offload_transfer`, i.e.
    ```c
    #pragma offload_transfer target(mic) in(data: length(n))
    ```

Asynchronous Offload

- Asynchronous Data Transfer helps to:
  - Overlap computations on host and MIC(s).
  - Work can be distributed to multiple coprocessors.
  - Data transfer time can be masked.
Asynchronous Offload

- To allow asynchronous data transfer, the specifiers `signal()` and `wait()` can be used, i.e.

```c
#pragma offload_transfer target(mic:0) in(data : length(n))
signal(data)

// work on other data concurrent to data transfer …
#pragma offload target(mic:0) wait(data) \
noCopy(data : length(N)) out(result : length(N))
{
    ....
    result[i]=data[i] + ...;
}
```

Any pointer type variable can serve as a signal!

- Alternative to the `wait()` clause, a new pragma can be used:

```c
#pragma offload_wait target(mic:0) wait(data)
```

- Useful if no other offload or data transfer is necessary at the synchronisation point.
Asynchronous Offload to Multiple Coprocessors

```c
char* offload0;
char* offload1;
#pragma offload target(mic:0) signal(offload0) \
in(data0 : length(N)) out(result0 : length(N))
{
    Calculate(data0, result0);
}
#pragma offload target(mic:1) signal(offload1) \
in(data1 : length(N)) out(result1 : length(N))
{
    Calculate(data1, result1);
}
#pragma offload_wait target(mic:0) wait(offload0)
#pragma offload_wait target(mic:1) wait(offload1)
```

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ

---

Explicit Worksharing

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            //section running on the coprocessor
            #pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
            {
                mxm(n,a,b,c);
            }
        }
        #pragma omp section
        {
            //section running on the host
            mxm(n,d,e,f);
        }
    }
}
```

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Persistent Data

- #define ALLOC alloc_if(1)
  #define FREE free_if(1)
  #define RETAIN free_if(0)
  #define REUSE alloc_if(0)

- To allocate data and keep it for the next offload:
  #pragma offload target(mic) in (p:length(l) ALLOC RETAIN)

- To reuse the data and still keep it on the coprocessor:
  #pragma offload target(mic) in (p:length(l) REUSE RETAIN)

- To reuse the data again and free the memory. (FREE is the default, and does not need to be explicitly specified):
  #pragma offload target(mic) in (p:length(l) REUSE FREE)
OpenMP 4.0.x Execution Model

- Create and destroy threads,
- create and destroy leagues of thread teams,
- assign / distribute work (tasks) to threads and devices,
- specify which data is shared and which is private,
- specify which data must be available to the device,
- coordinate thread access to shared data.

OpenMP 4.0.x Device Constructs

- Execute code on a target device
  - \texttt{omp target} [clause[,] clause,\ldots]
    structured-block
- Manage the device data environment
  - \texttt{map} ([map-type:] list) // map clause
    map-type := \texttt{alloc} | \texttt{tofrom} | \texttt{to} | \texttt{from}
  - \texttt{omp target data} [clause[,] clause,\ldots]
    structured-block
  - \texttt{omp target update} [clause[,] clause,\ldots]
  - \texttt{omp declare target}
    [variable-definitions-or-declarations]
  - \texttt{omp target enter / exit data} [clause[,] clause,\ldots] (new: OpenMP 4.5)
- Workshare for acceleration
  - \texttt{omp teams} [clause[,] clause,\ldots]
    structured-block
  - \texttt{omp distribute} [clause[,] clause,\ldots]
    for-loops
OpenMP 4.x Offloading Computation

- Use **target** construct to
  - Transfer control from the **host** to the **target device**
  - Map variables between the host and target device data environments
- Host thread waits until offloaded region completed
- Use **nowait** for asynchronous execution

```c
#pragma omp target map(to:b,c,d) map(from:a)
{
    #pragma omp parallel for
    for (i=0; i<count; i++) {
        a[i] = b[i] * c + d;
    }
}
```

OpenMP 4.x Target Construct

- Map variables to a **device data environment** and execute the construct on that device.
- **#pragma omp target** `[clause [...] clause ... ] new-line structured-block`
- where **clause** is one of the following:
  - `if([ target :] scalar-expression)`
  - `device(integer-expression)`
  - `private(list)`
  - `firstprivate(list)`
  - `map([map-type-modifier[,] map-type: ] list)`
  - `is_device_ptr(list)`
  - `defaultmap(tofrom:scalar)`
  - `nowait`
  - `depend(dependence-type: list)`
OpenMP 4.x Data mapping

**map Clause**

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
  int i, j;
  init(v1, v2, N);
  #pragma omp target data
  structured-block
  for (i=0; i<N; i++)
    p[i] = v1[i] * v2[i];
  output(p, N);
}
```

- **On entry to the target region:**
  - Allocate corresponding variables v1, v2, and p in the device data environment.
  - Assign the corresponding variables v1 and v2 the value of their respective original variables.
  - The corresponding variable p is undefined.

- **On exit from the target region:**
  - Assign the original variable p the value of its corresponding variable.
  - The original variables v1 and v2 are undefined.
  - Remove the corresponding variables v1, v2, and p from the device data environment.

- **Map variables to a device data environment for the extent of the region:**
  - `#pragma omp target data clause[[], clause] ... ] new-line structured-block`

- **Alternatively use 2 standalone directives**
  - `#pragma omp target enter data [clause[[], clause]...] new-line`
  - `...`
  - `#pragma omp target exit data [clause[[], clause]...] new-line`

- **Standalone directive to synchronize data**
  - `#pragma omp target update clause[[], clause] ...] new-line`
OpenMP 4.x Teams construct

- The `teams` construct creates a league of thread teams and the master thread of each team executes the region.
- `#pragma omp teams [clause [,] clause] ... ] new-line structured-block`
- where `clause` is one of the following:
  - `num_teams(integer-expression)`
  - `thread_limit(integer-expression)`
  - `default(shared | none)`
  - `private(list)`
  - `firstprivate(list)`
  - `shared(list)`
  - `reduction(reduction-identifier : list)`
- The `teams` construct creates a `league` of thread teams
  - The master thread of each team executes the `teams` region
  - The (max.) number of teams is specified by the `num_teams` clause
  - Each team executes with (max.) `thread_limit` threads
  - Threads in different teams cannot synchronize with each other
OpenMP 4.x Distribute Construct

- The **distribute** construct specifies that the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks. The iterations are distributed across the master threads of all teams that execute the **teams** region to which the distribute region binds.

- **#pragma omp distribute [clause [ , clause] ... ] new-line**
  **for-loops**

- Where **clause** is one of the following:
  - `private(list)`
  - `firstprivate(list)`
  - `lastprivate(list)`
  - `collapse(n)`
  - `dist_schedule(kind[, chunk_size])`

---

Composite constructs and shortcuts in OpenMP 4.5

- 2.10.9  **omp distribute simd**
- 2.10.10  **omp distribute parallel for**
- 2.10.11  **omp distribute parallel for simd**
- 2.11.5  **omp target parallel**
- 2.11.6  **omp target parallel for**
- 2.11.7  **omp target parallel for simd**
- 2.11.8  **omp target simd**
- 2.11.9  **omp target teams**
- 2.11.10  **omp teams distribute**
- 2.11.11  **omp teams distribute simd**
- 2.11.12  **omp target teams distribute**
- 2.11.13  **omp target teams distribute simd**
- 2.11.14  **omp teams distribute parallel for**
- 2.11.15  **omp target teams distribute parallel for**
- 2.11.16  **omp teams distribute parallel for simd**
- 2.11.17  **omp target teams distribute parallel for simd**
OpenMP 4.x Composite constructs and shortcuts

- **omp distribute**
  - omp distribute simd
    - Iterations distributed across the master threads of all teams in a teams region
  - omp distribute parallel for
    - dito + executed concurrently using SIMD instructions
  - omp distribute parallel for simd
    - dito + executed concurrently using SIMD instructions

- **omp teams**
  - omp teams distribute
    - creates a league of thread teams and the master thread of each team executes the region
  - omp teams distribute simd
  - omp teams distribute parallel for
  - omp teams distribute parallel for simd

- **omp target**
  - omp target simd
    - map variables to a device data environment and execute the construct on that device
  - omp target parallel for
  - omp target parallel for simd

- **omp target teams**
  - omp target teams distribute
  - omp target teams distribute simd
  - omp target teams distribute parallel for
  - omp target teams distribute parallel for simd

OpenMP 4.x SuperMIC Test

- **#pragma omp target**
  - u65fok@i01r13c06:~> ./a.out
    
    Hello world from host: I have 32 cores omp_get_default_device=0
    omp_get_num_devices=2 omp_get_num_teams=1 omp_get_team_num=0
    omp_is_initial_device=1
    Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=1

- **#pragma omp target teams**
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=236

- **#pragma omp target teams num_teams(4)**
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=59
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=59
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=59
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=59

- **#pragma omp target teams num_teams(4) thread_limit(2)**
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=2
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=2
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=2
  - Hello world from MIC i01r13c06-mic0: I have 240 cores omp_get_num_threads=2
Intel Xeon Phi Programming Models: Intel “Mine Yours Ours” (MYO) virtual shared memory model

- “Mine Yours Ours” virtual shared memory model.
- Alternative to Offload approach. Only available in C++.
- Allows to share not bit-wise compatible complex data (like structures with pointer elements, C++ classes) without data marshalling. LEO Offload Model only allows offloading of bitwise-copyable data!
- Allocation of data at the same virtual addresses on the host and the coprocessor.
- Runtime automatically maintains coherence.
- Syntax based on the keywords __Cilk_shared and __Cilk_offload.
#define N 10000

_Cilk_shared int a[N], b[N], c[N];

_Cilk_shared void add() {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main(int argc, char *argv[]) {
    ...
    _Cilk_offload add(); // Function call on coprocessor:
    ...
}

**MYO Language Extensions**

<table>
<thead>
<tr>
<th>Entity</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>int _Cilk_shared f(int x){...}</td>
<td>Executable code for both host and MIC; may be called from either side</td>
</tr>
<tr>
<td>Global variable</td>
<td>_Cilk_shared int x = 0</td>
<td>Visible on both sides</td>
</tr>
<tr>
<td>File/Function</td>
<td>static _Cilk_shared int x</td>
<td>Visible on both sides, only code within the file/function</td>
</tr>
<tr>
<td>Class</td>
<td>class _Cilk_shared x {...}</td>
<td>Class methods, members, and operators are available on both sides</td>
</tr>
<tr>
<td>Pointer to shared data</td>
<td>int _Cilk_shared *p</td>
<td>p is local (not shared), can point to shared data</td>
</tr>
<tr>
<td>A shared pointer</td>
<td>int *_Cilk_shared p</td>
<td>p is shared, should only point at shared data</td>
</tr>
<tr>
<td>Offloading a function call</td>
<td>x = _Cilk_offload func(y)</td>
<td>func executes on MIC if possible</td>
</tr>
<tr>
<td>Offloading asynchronously</td>
<td>x = _Cilk_offload_to(n) func</td>
<td>func must be executed on specified (n-th) MIC</td>
</tr>
<tr>
<td>Offloading asynchronously</td>
<td>_Cilk_spawn _Cilk_offload func(y)</td>
<td>Non-blocking offload</td>
</tr>
<tr>
<td>Offload a parallel for-loop</td>
<td>_Cilk_offload _Cilk_for(i=0; i&lt;N; i++) {...}</td>
<td>Loop executes in parallel on MIC</td>
</tr>
</tbody>
</table>
Xeon Phi References

- Books:
- Intel Xeon Phi Programming, Training material, CAPS
- Intel Training Material and Webinars

Acknowledgements

- IT4Innovation, Ostrava.
- Partnership for Advanced Computing in Europe (PRACE)
- Intel
- BMBF (Federal Ministry of Education and Research)
- Dr. Karl Fürlinger (LMU)
- J. Cazes, R. Evans, K. Milfeld, C. Proctor (TACC)
- Adrian Jackson (EPCC)
Thank you for your participation!
Intel MIC Programming Workshop:
MKL
Dr. Momme Allalen (LRZ)

June, 26-28, 2017 @ LRZ

Agenda

- A quick overview of Intel MKL
- Usage of MKL in Accelerator mode (KNC)
  - Compiler Assisted Offload
  - Automatic Offload
  - Native Execution
- Usage of MKL on KNL
- Hands-on & Performance
- Useful links where do we find more information?
What is the Intel MKL?

- Math library for C and Fortran, Includes:
  - BLAS, BLAS95 and Square BLAS
  - LAPACK
  - ScaLAPACK with BLACS
  - FFT and FFTW
  - Vector Math, Vector Statistics Functions
  - ...

- Containing optimised routines
  - For Intel CPUs and MIC architecture
- All MKL functions are supported on Xeon Phi

But optimised at different levels

Execution Models on Intel Xeon Phi Architectures

- Multicore Xeon Or KNL
- Many Core Hosted
  - KNL or Multicore Hosted
  - General purpose serial and parallel computing
- Offload
  - Codes with highly-parallel phases
- Symmetric
  - Codes with balanced needs
- Many Core Hosted (KNC)
  - highly-parallel codes
- MKL Native
- MKL AO & CAO

Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
MKL Usage In Accelerator Mode (KNC)

- **Compiler Assisted Offload**
  - Offloading is explicitly controlled by compiler pragmas or directives.
  - All MKL functions can be inserted inside offload region to run on the Xeon Phi (in comparison, only a subset of MKL is subject to AO).
  - More flexibility in data transfer and remote execution management.

- **Automatic Offload Mode**
  - MKL functions are automatically offloaded to the accelerator.
  - MKL decides:
    - When to offload
    - Work division between host and targets
  - Data is managed automatically

- **Native Execution**
How to use CAO

• The same way you would offload any function call to MIC
• An example in C:

```c
#pragma offload target(mic) \
in(transa, transb, N, alpha, beta) \
in(A:length(matrix_elements)) \
in(B:length(matrix_elements)) \
in(C:length(matrix_elements)) \
out(C:length(matrix_elements) alloc_if(0))
{
    sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,&beta, C, &N);
}
```

How to use CAO

• An example in Fortran:

```fortran
!DEC$ ATTRIBUTES OFFLOAD : TARGET( MIC ) :: SGEMM
!DEC$ OMP OFFLOAD TARGET( MIC ) &
!DEC$ IN( TRANS, TRANSB, M, N, K, ALPHA, BETA, LDA, LDB, LDC ), &
!DEC$ IN( A: LENGTH( NCOLA * LDA )), &
!DEC$ IN( B: LENGTH( NCOLB * LDB )), &
!DEC$ INOUT( C: LENGTH( N * LDC ))

!$OMP PARALLEL SECTIONS
!$OMP SECTION
    CALL SGEMM( TRANSA, TRANSB, M, N, K, ALPHA, &A, LDA, B, LDB BETA, C, LDC )
!$OMP END PARALLEL SECTIONS
```

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
Tips for Using Compiler Automatic Offload

- Use larger (>2MB) pages for data transferring;
  
e.g: ~$export MIC_USE_2MB_BUFFERS=50M

- This means that for any array allocation larger than 50MB, uses huge pages

MKL Usage In Accelerator Mode (KNC)

• Compiler Assisted Offload
  • Offloading is explicitly controlled by compiler pragmas or directives.
  • All MKL functions can be inserted inside offload region to run on the Xeon Phi (In comparison, only a subset of MKL is subject to AO).
  • More flexibility in data transfer and remote execution management.

• Automatic Offload Mode
  • MKL functions are automatically offloaded to the accelerator.
  • MKL decides:
    • When to offload
    • Work division between host and targets
  • Data is managed automatically

• Native Execution
How to Use Automatic Offload

• User does not have to change the code at all
• Either by calling the function mkl_mic_enable() or by setting the following environment variable

```bash
~$export MKL_MIC_ENABLE=1
```

• In Intel MKL 11.0.2 the following functions are enabled for automatic offload:

**Level-3 BLAS functions**
- *GEMM* (for M,N > 2048, k > 256)
- *TRSM* (for M,N > 3072)
- *TRMM* (for M,N > 3072)
- *SYMM* (for M,N > 2048)

**LAPACK functions**
- LU (M,N > 8192)
- QR
- Cholesky

How to Use Automatic Offload

• Blas only: work can be divided between host and device using

```c
mkl_mic_set_workdivision(TARGET_TYPE, TARGET_NUMBER, WORK_RATIO)
```

• What if there doesn’t exist a MIC card in the system?
  • Runs on the host as usual without any penalty !!

• Users can use AO for some MKL calls and use CAO for others in the same program
  • Only supported by Intel compilers
  • Work division must be set explicitly for AO, otherwise, all MKL AO calls are executed on the host
Automatic Offload Mode Example

```c
#include "mkl.h"
err = mkl_mic_enable();

//Offload all work on the Xeon Phi
err = mkl_mic_set_workdivision (MKL_TARGET_HOST, MIC_HOST_DEVICE, 0, 0);

//Let MKL decide of the amount of work to offload on coprocessor 0
err = mkl_mic_set_workdivision(MKL_TARGET_MIC, 0, MIC_AUTO_WORKDIVISION);

//Offload 50% of work on coprocessor 0
err = mkl_mic_set_workdivision(MKL_TARGET_MIC, 0, 0.5);

//Get amount of work on coprocessor 0
err = mkl_mic_get_workdivision(MKL_TARGET_MIC, 0, &wd);
```

Tips for Using Automatic Offload

- AO works only when matrix sizes are right
  - SGEMM: Offloading only when M, N > 2048
  - Square matrices give much better performance

- These settings may produce better results for SGEMM calculation for 60-core coprocessor:
  ```
  export MIC_USE_2MB_BUFFERS=16K
  export MIC_OMP_NUM_THREADS=240
  export MIC_ENV_PREFIX=MIC
  export MIC_KMP_AFFINITY=compact,granularity=fine
  export MIC_PLACE_THREADS=60C,4t
  ```

- Work division settings are just hints to MKL runtime
  Threading control tips:
  Prevent thread migration on the host using:
  ```
  export KMP_AFFINITY=granularity=fine, compact, 1,0
  ```
MKL Usage In Accelerator Mode (KNC)

• Compiler Assisted Offload
  • Offloading is explicitly controlled by compiler pragmas or directives.
  • All MKL functions can be inserted inside offload region to run on the Xeon Phi (In comparison, only a subset of MKL is subject to AO).
  • More flexibility in data transfer and remote execution management.

• Automatic Offload Mode
  • MKL functions are automatically offloaded to the accelerator.
  • MKL decides:
    • When to offload
    • Work division between host and targets
  • Data is managed automatically

• Native Execution

Native Execution

• In order to use Intel MKL in a native application, an additional argument -mkl is required with the compiler option -mmic.

• Native applications with Intel MKL functions operate just like native applications with user-defined functions.

$ icc -O3 -mmic -mkl sgemm.c -o sgemm.exe
Compile to use the Intel MKL

- Compile using \(-\text{mkl}\) flag
  - \(-\text{mkl}=\text{parallel}\) (default) for parallel execution
  - \(-\text{mkl}=\text{sequential}\) for sequential execution

- AO: The same way of building code on Xeon:
  - \texttt{user@host } \$ \texttt{icc -O3 -mkl sgemm.c -o sgemm.exe}

- Native using \(-\text{mmic}\)
  - \texttt{user@host } \$ \texttt{ifort -mmic -mkl myProgram.c -o myExec.mic}

- MKL can also be used in native mode if compiled with \(-\text{mmic}\)

Involving different MKL versions

<table>
<thead>
<tr>
<th>MKL Version</th>
<th>Link flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread, sequential</td>
<td>(-\text{mkl}=\text{sequential})</td>
</tr>
<tr>
<td>Single thread MPI, Sequential</td>
<td>(-\text{mkl}=\text{cluster})</td>
</tr>
<tr>
<td>Multi thread</td>
<td>(-\text{mkl}=\text{parallel} \text{ or } -\text{mkl})</td>
</tr>
</tbody>
</table>
More code examples:

- module show mkl
- $MKLROOT/examples/examples_mic.tgz
  - sgemm         SGEMM example
  - sgemm_f       SGEMM example (Fortran 90)
  - fft           complex-to-complex 1D FFT
  - solverc       Pardiso examples
  - sgaussian     single precision Gaussian RNG
  - dgaussian     double precision Gaussian RNG
  - ... 

Which Model to Choose:

- native execution for
  - Highly parallel code
  - Using MIC as independency compute nodes
- AO if
  - Sufficient Byte/FLOP ratio makes offload beneficial
  - Using Level-3 BLAS functions: GEMM, TRMM, TRSM
- CAO if
  - There is enough computations to offset data transfer overhead
  - Transferred data can be reused by multiple operations

Usage of MKL on KNL

- up to version 2017 MKL automatically try to use MCDRAM
  - with unlimited access to MCDRAM

- In order to restrict how much MCDRAM MKL uses is possible
  - Environment variable (in MB)
    \[ \text{MKL\_FAST\_MEMORY\_LIMIT} = 40 \]
  - Function call (in MB):
    \[ \text{mkl\_set\_memory\_limit(MKL\_MEM\_MCDRAM, 40)} \]

MKL 2018

- Use the “\texttt{intel64}” MKL version
- “\texttt{mic}” version is for KNC only
- Usage:
  - $\text{host: icpc -O3 -xMIC-AVX512 -lmemkind -qopenmp -mkl -qopt-report=0 ...}$
<table>
<thead>
<tr>
<th>Issue ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPD200576142</td>
<td>Improved the performance of functions DORCSD2BY1 routine</td>
</tr>
<tr>
<td>DPD200418144</td>
<td>Fixed the wrong result issue when calling zgstrf by MKL DIRECT CALL</td>
</tr>
<tr>
<td>DPD200592232</td>
<td>Fixed the scaling improvement of 3D r2c and c2r Cluster FFT, for some specific problems size</td>
</tr>
<tr>
<td>DPD200592231</td>
<td>Improved general performance on Intel® Xeon Phi™ product family x200</td>
</tr>
<tr>
<td>DPD200592229</td>
<td>MKL FFT : When 3D FFT scale is not 1, performance drops on KNL but not HSX for the c2r backward transform</td>
</tr>
<tr>
<td>DPD200591316</td>
<td>Fixed the problem with MKL Libraries mkl _*. Dll have to load their dependence on the absolute path of the folder where the DLL itself is located</td>
</tr>
<tr>
<td>DPD200592227</td>
<td>Fixed the problem when MKL FFT selects 1 thread per core on KNL regardless how many threads are requested by environment</td>
</tr>
</tbody>
</table>
Memory Allocation: Data Alignment

- Compiler-assisted offload
  - Memory alignment is inherited from host!

- General memory alignment (SIMD vectorisation)
  - Align buffers (leading dimension) to a multiple of vector width (64 Byte)
    - mkl_malloc, _mm_malloc (_aligned_malloc),
      tbb::scalable_aligned_malloc, ...

```c
void * darray;
int  workspace;
int  alignment = 64;
...
darray = mkl_malloc(sizeof(double) * workspace, alignment);
...
mkl_free(darray);
```
Performance of many Intel MKL routines improves when input and output data reside in memory allocated with 2 MB pages

—> Address more memory with less pages, reduce overhead of translating between host- and MIC address spaces

# Allocate all pointer-based variables with run-time
# length > 64 KB in 2 MB pages:
$ export MIC_USE_2MB_BUFFERS=64K

Native:
KMP_AFFINITY=balanced
OMP_NUM_THREADS=244

Compiler-Assisted Offload:
MIC_ENV_PREFIX=MIC
MIC_KMP_AFFINITY=balanced
MIC_OMP_NUM_THREADS=240
MIC_USE_2MB_BUFFERS=64K

Automatic Offload:
MKL_MIC_ENABLE=1
OFFLOAD_DEVICES=<list>
MKL_MIC_MAX_MEMORY=2GB
MIC_ENV_PREFIX=MIC
MIC_OMP_NUM_THREADS=240
MIC_KMP_AFFINITY=balanced
+ Compiler-Assisted Offload:
OFFLOAD_ENABLE_ORSL=1

Memory Allocation: Page Size

KMP_AFFINITY=
• Host: e.g., compact,1
• Coprocessor: balanced

MIC_ENV_PREFIX=MIC; MIC_KMP_AFFINITY=
• Coprocessor (CAO): balanced

KMP_PLACE_THREADS
• Note: does not replace KMP_AFFINITY
• Helps to set/achieve pinning on e.g., 60 cores with 3 threads each

kmp_* (or mk1_*) functions take precedence over corresponding env. variables

More MKL documentation

• Intel Many Integrated Core Community website:
• Performance charts online:
• Intel MKL forum

• https://software.intel.com/en-us/node/528430
• https://www.nersc.gov/assets/MKL_for_MIC.pdf
Thank you.
Intel MIC Programming Workshop: Vectorisation & Basic Performance Overview
Dr. Momme Allalen (LRZ)  
June, 26-28, 2017 @ LRZ

Agenda

- Basic Vectorisation & SIMD Instructions
- Vector loops - how to write loops in vector format
- Intel and GNU compiler vectorisation flags
- Hands-on (Lab1)
- Intel Tool VTune Amplifier and Adviser
- Hands-on (Lab2)
- Performance overview on the Intel Xeon Phi
## Evolution of Intel Vector Instruction Sets

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year &amp; Processor</th>
<th>SIMD Width</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997 Pentium</td>
<td>64-bit</td>
<td>8/16/32-bit Int.</td>
</tr>
<tr>
<td>SSE</td>
<td>1999 Pentium III</td>
<td>128-bit</td>
<td>32-bit SP FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001 Pentium 4</td>
<td>128-bit</td>
<td>8-64-bit Int., SP&amp;DP FP</td>
</tr>
<tr>
<td>SSE3-SSE4.2</td>
<td>2004-2009</td>
<td>128-bit</td>
<td>Additional instructions</td>
</tr>
<tr>
<td>AVX</td>
<td>2011 Sandy-Bridge</td>
<td>256-bit</td>
<td>SP &amp; DP FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013 Haswell</td>
<td>256-bit</td>
<td>Int. &amp; additional instr</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012 KNC</td>
<td>512-bit</td>
<td>32/64-bit Int., SP&amp;DP FP</td>
</tr>
<tr>
<td>AVX-512</td>
<td>2016 KNL</td>
<td>512-bit</td>
<td>32/64-bit Int. SP&amp;DP FP</td>
</tr>
</tbody>
</table>


## Other Floating-Point Vector

<table>
<thead>
<tr>
<th>Manufactures</th>
<th>Instruction Set</th>
<th>Register Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>VMX</td>
<td>4 way SP</td>
</tr>
<tr>
<td></td>
<td>SPU</td>
<td>2 way DP</td>
</tr>
<tr>
<td></td>
<td>Double FPU</td>
<td>2 way DP</td>
</tr>
<tr>
<td></td>
<td>Power8 has 64 VSR each</td>
<td>2 way DP (64bit) or 4 SP(32)</td>
</tr>
<tr>
<td>Motorola</td>
<td>AltiVec</td>
<td>4 way SP</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow</td>
<td>2 way SP</td>
</tr>
<tr>
<td></td>
<td>3DNow Professional</td>
<td>4 way SP</td>
</tr>
<tr>
<td></td>
<td>AMD64</td>
<td>2 way DP</td>
</tr>
<tr>
<td>ARM 64bit</td>
<td>NEON-v7-A - Cortex-R52</td>
<td>16<em>8b/8</em>16b/4<em>32b/2</em>64b SP</td>
</tr>
<tr>
<td></td>
<td>ARMv8-R</td>
<td>8<em>16bit/4</em>32bit/2*64bit FP</td>
</tr>
</tbody>
</table>
Vectorisation / SIMD instruction sets

MMX
- 64 bit: 1 x DP, 2 x SP
- 128 bit: 1 x DP, 2 x SP

SSE
- 256 bit: 2 x DP, 4 x SP

AVX
- 512 bit: 4 x DP, 8 x SP

MIC
- 1024 bit: 8 x DP, 16 x SP

On KNL each core has 2 Vector Processing Unit (VPU)
- SP => 512 bit registers / 32 bits x 2 VPUs = 32
- DP => 512 bit registers / 64 bits x 2 VPUs = 16
Vectorisation / SIMD instruction sets

Scalar Instructions

\[
\begin{align*}
7 + 2 &= 9 \\
3 + 1 &= 4 \\
0 + 6 &= 6 \\
-5 + 6 &= 1
\end{align*}
\]

Vector Instructions

\[
\begin{align*}
7 &+ 2 & 9 \\
3 &+ 1 & 4 \\
0 &+ 6 & 6 \\
-5 &+ 6 & 1
\end{align*}
\]

Scalar Loop

\[
\text{for } (i = 0; i < n; i++) \\
\]

SIMD Loop

\[
\text{for } (i = 0; i < n; i+=16) \\
A[i:(i+16)] = A[i:(i+16)] + B[i:(i+16)];
\]

Each SIMD add-operation acts on 16 numbers at time

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
KNL and Vector Instruction Sets

- Intel Advanced Vector Extensions 512 (AVX-512)
- 512-bit FP/Integer Vectors
- Gather/Scatter
- Binary compatibility with Xeon
- Supported by non Intel compilers: GCC

Vector Instruction Sets modules on KNL

- AVX-512F: Fundamentals for basic instructions such as: +, -, *, FMA, ... and extension of most AVX2 instructions to 512 vector registers.
- AVX-512CD: Conflict Detection: is set of instructions useful for (application: binning), e.g: is good for Monte Carlo calculations
- AVX-512ER: Exponential Reciprocal calculations, functions like: exp, rcp, and rsqrt in SP and DP.
- AVX-512PF: Prefetch instruction for gather and scatter operation.
Vectorisation on KNL vs KNC

<table>
<thead>
<tr>
<th>Knights Corner</th>
<th>Knights Landing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 VPU: Supports 512 bit vectors</td>
<td>2 VPUs</td>
</tr>
<tr>
<td>16x32-bit floats/integers</td>
<td></td>
</tr>
<tr>
<td>8 x 64-bit doubles</td>
<td></td>
</tr>
<tr>
<td>32 addressable registers</td>
<td>Full support for packed 64-bit integer arithmetic</td>
</tr>
<tr>
<td>Supports masked operations</td>
<td>Support unaligned loads &amp; stores</td>
</tr>
<tr>
<td>Only IMCI sets</td>
<td>Supports SSE/2/3/4, AVX, and AVX2 instruction sets but only on 1 of the 2 vector-units</td>
</tr>
<tr>
<td>In-order core</td>
<td>Other features: Out-of-order core, Improved Gather/Scatter Hardware FP Divide Hardware FP Inverse square root</td>
</tr>
</tbody>
</table>

The Improvement on the KNL Hardware is not good for non optimised code

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de

Difference between In-order and Out-of-order execution

In-order execution:

- Statically scheduled: executes instructions in sequential order.
- It will not execute next instruction until current instruction is completed.
- Have slower execution speed.

Out-of-order execution:

- Dynamically scheduled execution.
- It will execute next instruction without waiting for the previous instructions to finish unless they depend on the result.
- Faster execution speed.
Vectorisation: Approaches

- **Auto vectorisation** → only for loops can be auto-vectorised, you don’t need to do anything!!

- **Guided vectorisation** → using compiler hint and Tuning with directives (pragmas).

- **Explicit or Low level vectorisation** → C/C++ vector classes, Intrinsics /Assembly, “full control over instructions”, **limited portability**
Automatic Vectorisation of Loops

When does the compiler try to vectorise?

- For C/C++ and Fortran, the compiler look for vectorisation opportunities and detect whether loop can be vectorised.
- Enabled using --vec compiler flag (or whenever you compile at default optimisation -O2 or higher levels) and no source code changes.
  (Other Intel vec-flags: HSW: -xCORE-AVX2, SKX: -xCORE-AVX512 and KNL: -xMIC-AVX512)
  (GNU: enabled with -ftree-vectorize or -msse/-msse2 and by default at -O3 and -ffast-math)
- To disable all the autovectorisation use: -no-vec
- Sometimes it doesn’t work perfectly and the compiler may need your assistance

How do I know whether a loop was vectorised or not?

- use the vector report flags: -qopt-report=5 –qopt-report-phase=loop,vec
  (GNU: -ftree-vectorizer-verbose=2)

```
~$: more autovec.optrpt
...
LOOP BEGIN at autovec.cc (14,)
Remark #15300: LOOP WAS VECTORIZED [autovec.cc(14,3)]
LOOP END
.....
```

- The vectorisation should improve loop performance in general
Optimisation report phases

- The compiler reports optimisations consists of **9 Phases**:
  - **LOOP**: Loop Nest Optimisations
  - **PAR**: Auto-Parallelisation
  - **VEC**: Vectorisation
  - **OPENMP**: OpenMP
  - **OFFKOAD**: Offload
  - **IPO**: Interprocedural Optimisations
  - **PGO**: Profile Guided Optimisation
  - **CG**: Code Generation Optimisation
  - **TCOLLECT**: Trace Analyser Collection

- Compiler Option for multiple phase reporting:
  - `-qopt-report-phase=VEC,OPENMP,IPO,LOOP`

  - Default is “ALL” phases

Optimisation report levels

- The compiler’s optimisation report have **5 verbosity levels**
- Specifying report verbosity level:

  Compiler Option: `-qopt-report=N`

Example, VEC Phase levels:
  - Level1: reports when vectorisation has occurred
  - Level2: adds diagnostics why vectorisation did not occur
  - Level3: adds vectorisation loop summary diagnostics
  - Level4: adds additional available vectorisation support information
  - Level5: adds detailed data dependency information diagnostics
Example Automatic Vectorisation

double a[vec_width], b[vec_width];
....
//loop
for (int i = 0; i < vec_width; i++)
    a[i] += b[i];

This loop will be automatically vectorised

Loops can be vectorised

- Straight line code, because SIMD instructions perform the same operation on data elements
- Single entry and single exit
- No function calls, only intrinsic math functions such as \( \sin() \), \( \log() \), \( \exp() \), etc., are allowed

Loops that are not vectorisable:
- Loops with irregular memory access patterns
- Calculation with vector dependencies
- Anything that can not be vectorised or is very difficult to vectorise

Example of a Loop that is not Vectorisable

```c
void no_vec(float a[], float b[], float c[])
{
    int i = 0;
    while (i < 100) {
        a[i] = b[i] * c[i];
        // this is a data-dependent exit condition:
        if (a[i] < 0.0)
            break;
        ++i;
    }
}
```

- `icc -c -O2 -qopt-report=5 two_exits.cpp`

  `two_exits.cpp(4) (col. 9): remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.`

Example of Loops that is not Vectorisable

- Existence of vector dependence
  ```c
  for (j=n; j<SIZE; j++) {
      a[j] = a[j] + c * a[j-n];
  }
  ```

- Arrays accessed with stride 2
  ```c
  for (i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
  ```

- Inner loop accesses a with stride SIZE
  ```c
  for (int j=n; j<SIZE; j++) {
      for (int i=0; i<SIZE; i++)
          b[i] += a[i][j] * x[j];
  }
  ```

- Indirect addressing of x using index array
  ```c
  for (i=0; i<SIZE; i+=2) b[i] += a[i] * x[index[i]];
  ```

- It may be possible to overcome these using switches, pragmas, source code changes
  Useful tutorial: Using Auto Vectorisation: https://software.intel.com/en-us/compiler_15.0_vec.c
Data dependencies

Read after Write

```c
a[0]=0;
for (j=1; j<SIZE; j++)
    a[j]=a[j-1] + 1;
// this is equivalent to
```

Write after Read

```c
a[0]=0;
for (j=1; j<SIZE; j++)
    a[j-1]=a[j] + 1;
// this is equivalent to
```

GNU Support for Automatic vectorisation with AVX-512

up to GCC >= 4.9.1 supports AVX-512 instruction set

```bash
host:~/> g++ prog.cc -mavx512f -mavx512er -mavx512cd -mavx512pf
```

For automatic vectorisation support add: `-O3`

```c
//....prog.cc...../
for (int i= 0; i < n; i++)
hosl:~/> g++ -s prog.cc -mavx512f -O3
host:~/> cat prog.s
....
    vmovapd -16432(%rbp,%rax), %zmm0
    vaddpd -8240(%rbp,%rax) , %zmm0, %zmm0
    vmovapd %zmm0, -8240(%rbp,%rax)
```

make sure that the vector operations are operating on the zmm0 registers
Vectorisation: Approaches

- Auto vectorisation → only for loops can be auto-vectorised, you don’t need to do anything !!

- Guided vectorisation → using compiler hint and Tuning with directives (pragmas).

- Explicit or Low level vectorisation → C/C++ vector classes , Intrinsics /Assembly , “full control over instructions”, limited portability

void add_floats(float *a, float *b, float *c, float *d, float *e, int n)
{
    int i;
    #pragma simd
    for (i=0; i<n; i++){
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
#pragma ivdep: Instructs the compiler to ignore assumed vector dependencies.

#pragma loop_count: Specifies the iterations for the for loop.

#pragma novector: Specifies that the loop should never be vectorised.

#pragma omp simd: Transforms the loop into a loop that will be executed concurrently using SIMD instructions. (up to OpenMP 4.0)

## #pragma vector

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td>instructs the compiler to override any efficiency heuristic during the decision to vectorise or not, and vectorise non-unit strides or very unaligned memory accesses; controls the vectorisation of the subsequent loop in the program; optionally takes the keyword assert when vectorising</td>
</tr>
<tr>
<td>aligned</td>
<td>instructs the compiler to use aligned data movement instructions for all array references when vectorising</td>
</tr>
<tr>
<td>unaligned</td>
<td>instructs the compiler to use unaligned data movement instructions for all array references when vectorising</td>
</tr>
<tr>
<td>nontemporal</td>
<td>directs the compiler to use non-temporal (that is, streaming) stores on systems based on all supported architectures, unless otherwise specified; optionally takes a comma separated list of variables. On systems based on Intel® MIC Architecture, directs the compiler to generate clevict (cache-line-evict) instructions after the stores based on the non-temporal pragma when the compiler knows that the store addresses are aligned; optionally takes a comma separated list of variables</td>
</tr>
<tr>
<td>temporal</td>
<td>directs the compiler to use temporal (that is, non-streaming) stores on systems based on all supported architectures, unless otherwise specified</td>
</tr>
<tr>
<td>vecremainder</td>
<td>instructs the compiler to vectorise the remainder loop when the original loop is vectorised</td>
</tr>
<tr>
<td>novecremainder</td>
<td>instructs the compiler not to vectorise the remainder loop when the original loop is vectorised</td>
</tr>
</tbody>
</table>
Example for vectorisation pragmas

```c
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n)) {
#pragma omp parallel for
for( i = 0; i < n; i++ ) {
    for( k = 0; k < n; k++ ) {
#pragma vector aligned
#pragma ivdep
        for( j = 0; j < n; j++ ) {
            //c[i][j] = c[i][j] + a[i][k]*b[k][j];
            c[i*n+j] = c[i*n+j] + a[i*n+k]*b[k*n+j];
        }}}}"
```

Vectorisation: Approaches

- **Auto vectorisation** → only for loops can be auto-vectorised, you don’t need to do anything !!

- **Guided vectorisation** → using compiler hint and Tuning with directives (pragmas).

- **Explicit or Low level vectorisation** → C/C++ vector classes, Intrinsics /Assembly, “full control over instructions”, limited portability
Explicit Vectorisation

double a[vec_width], b[vec_width];
...
//
__m512d a_vec = _mm512_load_pd(a);
__m512d b_vec = _mm512_load_pd(b);
a_vec = _mm512_add_pd(a_vec, b_vec);
_mm512_store_pd(a, a_vec);

This is explicitly vectorised

Vector Intrinsics

- If compiler automatic vectorisation fails...

```c
#include <immintrin.h>
void vecmul(float *a, float *b, float *c, int n) {
    int i;
    __m512 va;
    __m512 vb;
    __m512 vc;
    for (i = 0; i < n; i += 16,
        a += 16, b += 16, c += 16) {
        ___m_prefetch((const char*) (a + 16), _MM_HINT_T0);
        va = _mm512_load_ps(a);
        vb = _mm512_extload_ps(b, _MM_UPCONV_PS_NONE,
                            _MM.Broadcast32_NONE,
                            _MM_HINT_NONE);
        vc = _mm512_mul_ps(va, vb);
        _mm512_store_ps(c, vc);
    }
```

Loop unrolling by 16 (i.e. vector length)
Increment pointers
IMCI Instruction Set

IMCI: Initial Many-Core Instruction set

IMCI is not SSE/or AVX

SSE2 Intrinsics

for (int i=0; i<n; i+=4){
    __m128 A_vec=_mm_load_ps(A+i);
    __m128 B_vec=_mm_load_ps(B+i);
    Avec=_mm_add_ps(A_vec, B_vec);
    _mm_store_ps(A+i, A_vec);
}

IMCI Intrinsics

for (int i=0; i<n; i+=16){
    __m512 A_vec=_mm512_load_ps(A+i);
    __m512 B_vec=_mm512_load_ps(B+i);
    Avec=_mm512_add_ps(A_vec, B_vec);
    _mm512_store_ps(A+i, A_vec);
}

The arrays float A[n] and float B[n] are aligned on 16-bit SSE2 and 64 bit IMCI boundary, where n is a multiple of 4 on SSE and 16 for IMCI.

The vector processing unit on MIC implements a different instruction set with more than 200 new instructions compared to those implemented on the standard Xeon.

Vectorisation procedure on Xeon Phi

Vectorisation: Most important to get performance on Xeon Phi

- The vectoriser for Xeon Phi works just like for the host
  - Enabled by default at optimisation level -O2 and above
  - Data alignment should be 64 bytes instead of 16
  - More loops can be vectorised, because of masked vector instructions, gather/scatter and fused multiply-add (FMA)
  - Try to avoid 64 bit integers (except as addresses)

- Identify a vectorised loops by:
  - Vectorisation and optimisation reports (recommended) -qopt-report=5 -qopt-report-phase=loop,vec
  - Unmasked vector instructions
  - Math library calls to libsvm
Intel Optimisation flags

- **Precision**
  - `fp-model precise`
  - `no-prec-div`
  - `no-prec-sqrt`
  - `fno-alias`

- **Performance**
  - `fp-model fast=2`
  - `-ftz`
  - `-align all`
  - `-march=native`

---

Intel specific switches may generate vector extensions

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize for current architecture</td>
<td><code>-xHOST</code></td>
</tr>
<tr>
<td>Generate SSE v1 code</td>
<td><code>-xSSE1</code></td>
</tr>
<tr>
<td>Generate SSE v2 code</td>
<td><code>-xSSE2</code></td>
</tr>
<tr>
<td>Generate SSE v3 code (may also emit SSE v1 and SSE v2)</td>
<td><code>-xSSE3</code></td>
</tr>
<tr>
<td>Generate SSSE v3 code for Atom based processors</td>
<td><code>-xSSE_ATOM</code></td>
</tr>
<tr>
<td>Generate SSSE v3 code (may also emit SSE v1, v2 and SSE v3)</td>
<td><code>-xSSSE3</code></td>
</tr>
<tr>
<td>Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code)</td>
<td><code>-xSSE4.1</code></td>
</tr>
<tr>
<td>Generate SSE4.2 code (may also emit (S)SSE v1,v2, v3 and v4 code)</td>
<td><code>-xSSE4.2</code></td>
</tr>
<tr>
<td>Generate AVX code</td>
<td><code>-xAVX</code></td>
</tr>
<tr>
<td>Generate AVX2 code</td>
<td><code>-xAVX2</code></td>
</tr>
<tr>
<td>Generate Intel CPUs includes AVX-512 processors code</td>
<td><code>-xCORE-AVX512</code></td>
</tr>
<tr>
<td>Generate KNL code (and successors)</td>
<td><code>-xMIC-AVX512</code></td>
</tr>
<tr>
<td>Generate AVX-512 code for newer processors</td>
<td><code>-xCOMMON-AVX512</code></td>
</tr>
</tbody>
</table>
Intel specific switches may generate vector extensions

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>To generate optimises code for KNL</td>
<td>-xMIC-AVX512</td>
</tr>
<tr>
<td>To generate optimises code for Xeon SKX</td>
<td>-xCORE-AVX512</td>
</tr>
<tr>
<td>Cross platform 2 versions: baseline and KNL</td>
<td>-axMIC-AVX512</td>
</tr>
<tr>
<td>Cross platform 2 versions: baseline and Xeon SKX</td>
<td>-axCORE-AVX512</td>
</tr>
<tr>
<td>3 versions: baseline, KNL and Xeon SKX</td>
<td>-axMIC-AVX512,CORE-AVX512</td>
</tr>
<tr>
<td>Generate AVX-512 code for KNL and SKX</td>
<td>-xCOMMON-AVX512</td>
</tr>
<tr>
<td>Generate code for KNC</td>
<td>-mmic</td>
</tr>
</tbody>
</table>

GNU Support for Automatic vectorisation with AVX-512

Cross Platform (KNL and SKX):

-xCOMMON-AVX512 generate: AVX-512F and AVX-512CD

-mavx512f -mavx512cd

Xeon Processors

-xCORE-AVX512 generate: AVX-512F, AVX512CD, AVX512BW, AVX-512DQ, AVX512VL

-mavx512f -mavx512cd -mavx512bw
-mavx512dq -mavx512vl
-mavx512ifma,
-mavx512ifma, -mavx512vbmi

Xeon Phi

-xMIC-AVX512 generate: AVX-512F, AVX-512CD, AVX-512ER, AVX512FP

-mavx512f -mavx512cd -mavx512er -mavx512pf
Thread Affinity

- Pinning threads is important!

\$ export KMP_AFFINITY="granularity=thread,\textit{x}\"

\textit{x}=\textit{compact}, \textit{scatter}, \textit{balanced}

"See Intel compiler documentation for more information".

```
~$ export KMP_AFFINITY=granularity=thread,compact.
~$ export KMP_AFFINITY=granularity=thread,scatter.
```

Tips for Writing Vectorisable Code

- Avoid dependencies between loop interactions
- Avoid read after write dependencies
- Write straight line code (avoid branches such as switch, goto or return statements,..etc)
- Use efficient memory accesses by aligning your data to
  - 16-Byte alignment for SSE2
  - 32-Byte alignment for AVX
  - 64-Byte alignment for Xeon Phi
Lab 1: Vectorisation 1: nbody problem

Intel VTune Amplifier
Intel Adviser
What is Intel VTune Amplifier XE?

Where is my application:

- Spending Time? functions taking time ..etc
- Wasting Time? find cache misses and other inefficiencies.
- Waiting Too Long? see locks and cpu utilisation during waiting …

What is Intel VTune Amplifier XE?
- Is a performance profiling tool for serial, OpenMP, MPI and hybrid applications
- Helps users to collect timing performance information
- Intel VTune capable to check the threading performance, load balancing, bandwidth, I/O, overhead and much more
- Analysis is simple using a GUI to visualise results of timeline on your source code…
- Capable to look at memory access on KNL: DDR4 and MCDRAM (Flat or Cache)
- Useful for controlling memory allocation using libmemkind library (hpw_malloc)
Usage with command line amplxe-cl

- module load amplifier_xe/2017 or amplifier_xe/2018
- To print all the options type: amplxe-cl -help

amplxe-cl <-action> [-action-option] [-global-option]
    [[- -] target [target options]]

action              : collect, collect-with, report….
[-action-option]  : modify behaviour specific to the action
[-global-option] : modify behaviour in the same manner for all
                  actions, e.g: -q, -quiet to suppress non essential
                  messages
[- -]target        : the target application to analyse
target options     : application options

Using a submission script: amplxe-cl

- Write a submission script based on your resource manager,
  load all the needed modules and set the environment
  variables, and launch your program with:

  amplxe-cl -collect memory-access -knob analyze-mem-objects=true
             -no-summary -app-working-dir . - ./exec

- or to collect only the hotspots on the given target, use:
  - amplxe-cl -collect hotspots - - mpiexec -n 8 ./exec other-options

- To generate the hotspots report for the result directory r00hs
  - amplxe-cl -report hotspots -r r00hs
compile your code with "-g" for source code and add "-lmemkind" for memory analysis

compile, e.g:
  - mpiicc -g -O3 -xHOST -qopenmp -lmemkind source.c
  - ifort -g -O3 -xHOST -qopenmp -lmemkind source.f90

Execute: first load the VTune module and run
  - GUI : amplxe-gui
  - or with a command line: amplxe-cl

Analyse the results with VTune Amplifier
  - GUI: amplxe-gui
  - or command line: amplxe-cl
VTune collections

<table>
<thead>
<tr>
<th>Collections</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotspots</td>
<td>identify the most time consuming sections on the code</td>
</tr>
<tr>
<td>advanced-hotspots</td>
<td>Adds CPI, higher frequency low overhead sampling</td>
</tr>
<tr>
<td>disk-io</td>
<td>Disk IO preview, not working in Stamped (requires root access)</td>
</tr>
<tr>
<td>concurrency</td>
<td>CPU utilisation, threading synchronisation overhead</td>
</tr>
<tr>
<td>memory-access</td>
<td>Memory access details and memory bandwidth utilisation (useful for MCDRAM on KNL)</td>
</tr>
<tr>
<td>hpc-performance</td>
<td>Performance characterisation, including floating point unit and memory bandwidth utilisation</td>
</tr>
</tbody>
</table>

Useful options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-data-limit</td>
<td>Override default maximum data collection size</td>
</tr>
<tr>
<td>-no-summary</td>
<td>Do not produce text summary</td>
</tr>
<tr>
<td>-no-auto-finalize</td>
<td>Do not finalise data analysis after collection</td>
</tr>
<tr>
<td>-finalize</td>
<td>Carry out data analysis after collection</td>
</tr>
<tr>
<td>-start-paused</td>
<td>Start application without profiling</td>
</tr>
<tr>
<td>-resume-after=X</td>
<td>Resume profiling after X seconds</td>
</tr>
<tr>
<td>-duration=Y</td>
<td>Profiling only for Y seconds</td>
</tr>
<tr>
<td>analyse-openmp=true</td>
<td>Determine inefficiencies in OpenMP regions</td>
</tr>
<tr>
<td>analyze-memory-obejcts=true</td>
<td>Determine arrays using most memory bandwidth (highest L2 miss rates)</td>
</tr>
</tbody>
</table>
Intel Adviser

About the Adviser and Capabilities

- Adviser is a vectorisation optimisation and shared memory threading assistance tool for C, C++ and Fortran code
- Adviser supports both serial, threaded, and MPI applications
- Current version is 2017.1.0

Vectorisation:
- Evaluate the efficiency of vectorised code and key SIMD bottlenecks
- Check for loop-carried dependencies dynamically
- Identify memory versus compute balance and provide register utilisation

Threading Advisor:
- find where to add parallelism and identify where the code spends its time.
- Predict the performance you might achieve with the proposed code parallel regions
- Predict the data sharing problems that occur in the proposed parallel code regions
How to use the Adviser

- Load the adviser module and set the environment variables
  - module load advisor_xe/2017 or advisor_xe/2018
- Compile with (-g, -O2, -vec, -simd, -qopenmp, -qopt-report=5,…etc.)
  - ifort -g -xHOST -O2 -qopt-report=5 source.f90
- Collect the information data
  - advixe-cl -c survey - -./exec
- Analyse the data with
  - advixe-gui
### Useful options

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>survey</td>
<td>Helps you detect and select the best places to add parallelism in your code</td>
</tr>
<tr>
<td>Trip Counts</td>
<td>Helps you to collect loop interaction statistics</td>
</tr>
<tr>
<td>Suitability</td>
<td>Helps you predict the likely performance impact of adding parallelism to the selected places</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Helps you predict and eliminate data sharing problems before you add parallelism. 50-500 times slower</td>
</tr>
<tr>
<td>Memory Access Patterns (MAP)</td>
<td>Helps you to collect data on memory access strides 3-20 times slower</td>
</tr>
</tbody>
</table>
Intel Advisor annotations

- Step 1: In the Intel Advisor GUI: build applications and create new project
- Step 2: Display the adviser XE workflow and run the Survey Tool (to discover parallel opportunities)
- Step 3: Display sources in the survey source window and find where to add Intel Advisor parallel site task annotations

```c
#include <advisor-annotate.h>
...
ANNOTATE_SITE_BEGIN();
...
ANNOTATE_ITERACTION_TASK(task1);
...
ANNOTATE_SITE_END();

ANNOTATE_DISABLE_COLLECTION_POP;
...
ANNOTATE_DISABLE_COLLECTION_PUSH;
```

```c
use advisor_annotate
...
CALL ANNOTATE_SITE_BEGIN()
...
CALL ANNOTATE_ITERACTION_TASK("task1")
...
CALL ANNOTATE_SITE_END()

CALL ANNOTATE_DISABLE_COLLECTION_POP()
...
CALL ANNOTATE_DISABLE_COLLECTION_PUSH()
```

Muti-run analysis with Adviser

- Survey
  - advixe-cl -c survey - -search-dir src:./ - - ./exe
- Trip Counts
  - advixe-cl -c tripcounts - -search-dir src:./ - - ./exe
- Suitability (with site annotations in source code)
  - icc -g -xHOST -O2 -qopt-report=5 source.c $ADVISOR_INC $ADVISOR_LIB
  - advixe-cl -c suitability —search-dir src:=./ - - ./exec
- Dependencies
  - advixe-cl -c dependencies -track-stack-variables - -search-dir src:./ - - ./exe
- Memory Access Patterns (MAP)
  - advixe-cl -c map -record-stack-frame -record-mem-allocations - -search-dir src:./ - - ./exe
Vectorisation Procedure

- Quantify performance and baseline measurement
- Define a standard metric for all future improvements
- What system components are stressed during runtime (CPU, memory, disks, network)?
- Find the hotspots using VTune Amplifier
- Identify the loop candidate for adding parallelism using the compiler report flags
- Get advices using Intel Advisor
- Add parallelism in the recommended regions
- Check the results and repeat the previous steps

Lab 2: Vectorisation 2: nbody problem
Step 1: check the not vectorised loops

- **get the vectorisation report**
  - icc -g -O2 -qopt-report=5 -qopt-report-phase=loop,vec -parallel -mmic -qopenmp nobody.c -o exec.mic

vi nbody.optrpt

LOOP BEGIN at nbody.c(66,2) inlined into nbody.c(129,3)

- remark #15542: loop was not vectorized: inner loop was already vectorized
- remark #25018: Total number of lines prefetched=6
- remark #25019: Number of spatial prefetches=6, dist=8
- remark #25021: Number of initial-value prefetches=3
- remark #25139: Using second-level distance 2 for prefetching spatial memory

reference [ nbody.c(87,5) ]

reference [ nbody.c(86,5) ]

reference [ nbody.c(85,5) ]

remark #25015: Estimate of max trip count of loop=10000

Step 1: check the not vectorised loops

- Change in the source code
  - Vectorise the loop with SIMD pragma

```c
int i = 0;
#pragma simd reduction(-: mass_objects)
for (i = 0; i < SIZE; ++i)
{
    x_objects[i] = -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX;
    y_objects[i] = -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX;
    z_objects[i] = -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX;
    vx_objects[i] = -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f;
    vy_objects[i] = -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f;
    vz_objects[i] = -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f;
    ax_objects[i] = 0.0f;
    ay_objects[i] = 0.0f;
    az_objects[i] = 0.0f;
    mass_objects[i] = (float)SIZE + (float)SIZE * rand() / (float)RAND_MAX;
}
```
Step 2: Check SIMD loops

- Check the vectorisation report
  - Vectorise the loop with SIMD pragma

- Check the performance
  - compile again and check the vectorisation report

```
#pragma omp for private(i,j)
for (i = 0; i < SIZE; i++)     // update velocity
{
  #pragma ivdep
  // or #pragma vector always
  for (j = 0; j < SIZE; j++)
  {
    if (i < j || i > j)
    {
      float distance[3];
      float distanceSqr = 0.0f, distanceInv = 0.0f;
    
```

Step 3: Check the Vector length used (typically 8 or 16)

- take a look into the source code and add the IMCI vector extensions on the loops

```
      #pragma omp for private(i,j)
      for (i = 0; i < SIZE; i++)     // update velocity
      {
        #pragma ivdep
        // or #pragma vector always
        for (j = 0; j < SIZE; j++)
        {
          if (i < j || i > j)
          {
            float distance[3];
            float distanceSqr = 0.0f, distanceInv = 0.0f;
          
```

- Check the performance and what do you think!!
Step 4: Check the array alignment status

Alignment status for every array used

```
LOOP BEGIN at nbody.c(154,7) inlined into nbody.c(214,3)
remark #15389: vectorization support: reference x_objects has unaligned access   [ nbody.c(156,7) ]
remark #15389: vectorization support: reference y_objects has unaligned access   [ nbody.c(157,7) ]
remark #15389: vectorization support: reference z_objects has unaligned access   [ nbody.c(158,7) ]
remark #15389: vectorization support: reference vx_objects has unaligned access   [ nbody.c(159,7) ]
remark #15389: vectorization support: reference vy_objects has unaligned access   [ nbody.c(160,7) ]
remark #15389: vectorization support: reference vz_objects has unaligned access   [ nbody.c(161,7) ]
remark #15389: vectorization support: reference ax_objects has unaligned access   [ nbody.c(162,7) ]
remark #15389: vectorization support: reference ay_objects has unaligned access   [ nbody.c(163,7) ]
remark #15389: vectorization support: reference az_objects has unaligned access   [ nbody.c(164,7) ]
remark #15388: vectorization support: reference mass_objects has aligned access   [ nbody.c(165,7) ]
```

```
LOOP BEGIN at nbody.c(73,4) inlined into nbody.c(236,7)
remark #15389: vectorization support: reference x_objects has unaligned access   [ nbody.c(79,5) ]
remark #15389: vectorization support: reference y_objects has unaligned access   [ nbody.c(80,5) ]
remark #15389: vectorization support: reference z_objects has unaligned access   [ nbody.c(81,5) ]
remark #15388: vectorization support: reference mass_objects has aligned access   [ nbody.c(85,5) ]
remark #15388: vectorization support: reference mass_objects has aligned access   [ nbody.c(86,5) ]
remark #15388: vectorization support: reference mass_objects has aligned access   [ nbody.c(87,5) ]
```

Summary

- Concerning the **ease of use and the programmability** Intel Xeon Phi is almost compared to other accelerators like GPGPUs, Mali-GPUs, FPGAs or former CELL processors or ClearSpeed cards.
- Codes using MPI, OpenMP or MKL etc. can be quickly ported. Some MKL routines have been highly optimised for the MIC.
- Due to the large SIMD width of 64 Bytes **vectorisation** is even more important for the MIC architecture than for the actual Intel Xeon based systems.
- It is extremely simple to get a code running on Intel Xeon Phi, but getting performance out of the chip in most cases needs **manual tuning of the code** due to failing auto-vectorisation.
- **MIC programming** enforces programmer to think about SIMD vectorisation.
Xeon Phi References

- Books:

- Intel Xeon Phi Programming, Training material, CAPS
- Intel Training Material and Webinars
- V. Weinberg (Editor) et al., Best Practice Guide - Intel Xeon Phi, http://www.prace-project.eu/Best-Practice-Guide-Intel-Xeon-Phi-HTML and references therein

Acknowledgements

- IT4Innovation, Ostrava.
- Partnership for Advanced Computing in Europe (PRACE)
- Intel
- BMBF (Federal Ministry of Education and Research)
Thank you.
KNL Optimization

Dr. Fabio Baruffa
fabio.baruffa@lrz.de

Porting and optimization methods on KNL

Three steps work... almost!

Optimize performance without coding
(1 week, 2X performance speed-up)

Compile and run for KNL

Advanced optimize performance
(1-3 months, more than 3X performance speed-up)

Optimization methods

- Loop optimization, merging, nesting,...
- Parallel model optimization, such as MPI, OpenMP, hybrid
- Memory access optimization, cache tiling,...
Thank you for your attention!

Optimization process
Optimization process: Few basic guidelines

- Selection of the **best algorithm** for the problem
- Use efficient **library** (why should we reinvent the wheel?)
- **Optimal data layout**
  - **temporal locality**: a resource referred at one point in time will be likely reused in the future
  - **spacial locality**: if a location is referred at a one point in time, its likely that a nearby location will be reused
- Use of compiler optimization **flags**

Performance analysis

- A real life application application has several functions, routines, dependencies,…
- Code optimization and parallelization (shared/distributed memory) is an hard task. The crucial points are:
  - define a good metric for comparisons (timing, flops, memory references,…)
  - define a good representative data setup (non too long, not too short,…)
  - find bottlenecks and critical parts (profiling tools, gprof, Papi, VTune, …)
- Some practical suggestions:
  - there is no general rule
  - use always "realistic" test case to profile the application
  - use always different data sizes for your problem
  - pay attention to input/output
  - use different architectures (when it is possible)
**Common Intel® compiler flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>No optimization. Use in the early stage of dev. and debugging.</td>
</tr>
<tr>
<td>-O1</td>
<td>Optimize for size. Small objects size.</td>
</tr>
<tr>
<td>-O2</td>
<td>Maximize for speed. Includes vectorization.</td>
</tr>
<tr>
<td>-O3</td>
<td>Loop optimization, scalar replacements, efficient cache reuse. Aggressive floating point optimization.</td>
</tr>
<tr>
<td>-g</td>
<td>Create symbols for debugging.</td>
</tr>
<tr>
<td>-ipo</td>
<td>Multi-file inter-procedural analysis</td>
</tr>
<tr>
<td>-qopt-report-phase: name1, ...</td>
<td>All (all phases of the optimization), loop, vec (explicit for vectorization), openmp, ipo, offload,...</td>
</tr>
<tr>
<td>-qopenmp</td>
<td>OpenMP 4.0 support</td>
</tr>
</tbody>
</table>

---

**Code optimization process**

- **Scalar optimization**: compiler flags, data casting, precision consistency.
- **Vectorization**: prepare the code for SIMD, avoid vector dependencies.
- **Memory access**: improve data layout, cache access.
- **Multi-threading**: enable OpenMP, manage scheduling and pinning.
- **Communication**: enable MPI, offloading computation.
Code optimization process

- **Scalar optimization**: compiler flags, data casting, precision consistency.
- **Vectorization**: prepare the code for SIMD, avoid vector dependencies.
- **Memory access**: improve data layout, cache access.
- **Multi-threading**: enable OpenMP, manage scheduling and pinning.
- **Communication**: enable MPI, offloading computation.

Nbody example
Let’s consider a distribution of point masses located at points \( r_1 \ldots r_n \) and have masses \( m_1, \ldots, m_n \).

We want to calculate the position of the particles after a certain time using the Newton law of gravity:

\[
\vec{F}_{ij} = \frac{G m_i m_j}{|\vec{r}_j - \vec{r}_i|^3} (\vec{r}_j - \vec{r}_i)
\]

\[
\vec{F} = m \ddot{\vec{x}} = m \frac{d \vec{v}}{dt} = m \frac{d^2 \vec{x}}{dt^2}
\]

**Particle.hpp:**

```cpp
struct Particle {
  public:
    Particle() { init(); }
    void init() {
      pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
      vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
      acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
      mass = 0.;
    }
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};
```

**GSimulation.cpp:**

```cpp
... for (i = 0; i < n; i++)  // update acceleration
  for (j = 0; j < n; j++)
    real_type distance, dx, dy, dz;
    real_type distanceSqr = 0.0;
    real_type distanceInv = 0.0;
    dx = particles[j].pos[0] - particles[i].pos[0];  //1flop
    dy = particles[j].pos[1] - particles[i].pos[1];  //1flop
    distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;  //6flops
    distanceInv = 1.0 / sqrt(distanceSqr);  //1div+1sqrt
    particles[i].acc[0] += dx * G * particles[j].mass * distanceInv * distanceInv;  //6flops
    particles[i].acc[1] += _    //6flops
...  // update position and velocity
...```

11 Optimization Process

12 Optimization Process
Live-session

- Go to the folder `code/nbody/base`
- Load the appropriate compiler module
- Run `make` from that directory on the login node
- Run the code with `make run`
- Play changing the number of particles
- How does the performance change?

---

Nbody example code

Run the default test case on CPU:

```bash
./nbody.x
```

Initialize Gravity Simulation

```
nPart = 2000; nSteps = 500; dt = 0.1
```

<table>
<thead>
<tr>
<th>s</th>
<th>dt</th>
<th>kenergy</th>
<th>time (s)</th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>701.22</td>
<td>10.099</td>
<td>0.57452</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>956.84</td>
<td>10.098</td>
<td>0.57458</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>1036.6</td>
<td>10.097</td>
<td>0.57461</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1644.9</td>
<td>10.097</td>
<td>0.57462</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>1565.5</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1793.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>1848.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>2304.1</td>
<td>10.097</td>
<td>0.5746</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>3098.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>3324.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
</tbody>
</table>

# Number Threads : 1
# Total Time (s) : 100.97
# Average Performance : 0.57463 +- 1.1331e-05
## Nbody example code

Run the default test case on CPU:

`.nbody.x`

Initialize Gravity Simulation

\[ nPart = 2000; \text{nSteps} = 500; \ dt = 0.1 \]

<table>
<thead>
<tr>
<th>s</th>
<th>dt</th>
<th>kenergy</th>
<th>time (s)</th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>701.22</td>
<td>10.099</td>
<td>0.57452</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>956.84</td>
<td>10.098</td>
<td>0.57458</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>1036.6</td>
<td>10.097</td>
<td>0.57461</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1644.9</td>
<td>10.097</td>
<td>0.57462</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>1565.5</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1793.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>1848.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>2304.1</td>
<td>10.097</td>
<td>0.57462</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>3098.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>3324.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
</tbody>
</table>

# Number Threads : 1

# Total Time (s) : 100.97

# Average Perfomance : 0.57463 +- 1.1331e-05

---

## KNL Basic Architecture

- **Intel® Xeon Phi Processor 7210**: 64 cores at 1.3 GHz in a single socket
- Theoretical \( P_{\text{max}} = (1.3 \times 64 \text{ cores} \times 32 \text{ DP Flops/cycles}) \) GF/s = 2662 GF/s
- \( 32 = 2\text{-VPUs AVXS12} \rightarrow 16 \text{ DP} \) per unit x 2 FMA
- NB: no SIMD code gives 83 GF/s

![KNL Architecture Diagram](image)

- L1d cache: 32KB
- L1i cache: 32KB
- L2 cache: 1024KB
- MCDRAM 16GB

- Hyperthreading
Scalar and general optimization

- The code of part of it can be compiled with more aggressive optimization (-O3) [loop fusion, unroll-and-jam,...]

- Processor specific optimization: -xSSE4.2, -xAVX (E3 and e5 family), -xCORE-AVX2 (v3), -xCORE-AVX512 (Skylake), -xMIC-AVX512 (KNL), -mmic (KNC)

- Floating point semantics: -fp-model=precise, fast=1,2, ...

- Precision of constant and variables: consistent use of single and double precision

<table>
<thead>
<tr>
<th>Type</th>
<th>Decimal Point</th>
<th>Exponent</th>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>no</td>
<td>no</td>
<td>none</td>
<td>0.1,300</td>
</tr>
<tr>
<td>long</td>
<td>no</td>
<td>no</td>
<td>1 or L</td>
<td>0L,1L,10000000000000L</td>
</tr>
<tr>
<td>double</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
<td>0.0,1.0,1.0e100</td>
</tr>
<tr>
<td>float</td>
<td>yes</td>
<td>yes</td>
<td>f or F</td>
<td>0.0F,1.0F,1.0e10F</td>
</tr>
<tr>
<td>long double</td>
<td>yes</td>
<td>yes</td>
<td>1 or L</td>
<td>0.0L,1.0L,1.0e100L</td>
</tr>
</tbody>
</table>

Table 4.4: Conventions for defining literal constants in C and C++.
Scalar and general optimization

- The code of part of it can be compiled with more aggressive optimization (-O3) [loop fusion, unroll-and-jam,...]
- Processor specific optimization: -xSSE4.2, -xAVX (E3 and e5 family ), -xCORE-AVX2 (v3), -xCORE-AVX512 (Skylake), -xMIC-AVX512 (KNL), -mmic (KNC)
- Floating point semantics: -fp-model=precise, fast=1,2, ...
- Precision of constant and variables: consistent use of single and double precision
- Precision of functions: in MKL (scalar arithmetics) there is single and double precision version of the math functions
- Strength reduction: replacing expensive operations with one less expensive (see GSimulation.cpp line: 161)

Optimization report

- -qopt-report[=N]: default level is 2
- -qopt-report-phase=<vec,loop,openmp,...>: default is all
- -qopt-report-file=stdout | stderr | filename
- -qopt-report-filter="GSimulation.cpp,130 - 194"
Optimization report

- `-qopt-report[=N]`: default level is 2
- `-qopt-report-phase=<vec,loop,openmp,...>`: default is all
- `-qopt-report-file=stdout | stderr | filename`
- `-qopt-report-filter="GSimulation.cpp,130 - 194"

Let’s see the report in action!

Scalar and general optimization

Optimization report

- `-qopt-report[=N]`: default level is 2
- `-qopt-report-phase=<vec,loop,openmp,...>`: default is all
- `-qopt-report-file=stdout | stderr | filename`
- `-qopt-report-filter="GSimulation.cpp,130 - 194"

Let’s see the report in action!

- `-qopt-report-phase=vec -qopt-report=5`
- **Level 1**: Reports when vectorization has occurred.
- **Level 2**: Adds diagnostics why vectorization did not occur.
- **Level 3**: Adds vectorization loop summary diagnostics.
- **Level 4**: Adds additional available vectorization support information.
- **Level 5**: Adds detailed data dependency information diagnostics.

Scalar and general optimization
Live-session

- Go to the folder `code/nbody/base`
- Load the appropriate compiler module
- Run `make clean` to remove the old files
- Change the `Makefile` adding the compiler flags to generate the report: `-qopt-report=5`
- Reduce the amount of output:
  `-qopt-report-filter="GSimulation.cpp,130 - 194"` (maybe filter more)
- Change compiler flag to: `-xMIC-AVX512`
- Work on precision consistency

Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
</tbody>
</table>
FP conversions

LOOP BEGIN at GSimulation.cpp(150,7)
remark #25444: Loopnest Interchanged: (1 2) --> (2 1)
remark #15541: loop was not vectorized: inner loop was already vectorized

[ GSimulation.cpp(150,7) ]

LOOP BEGIN at GSimulation.cpp(148,5)
...remark #15417: vectorization support: number of FP up converts: single precision to double
preCISION 1 [ GSimulation.cpp(163,4) ]
remark #15418: vectorization support: number of FP down converts: double precision to single
preCISION 1 [ GSimulation.cpp(163,4) ]
remark #15417: vectorization support: number of FP up converts: single precision to double
preCISION 6
...remark #15452: unmasked strided loads: 6
remark #15453: unmasked strided stores: 3
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 150
remark #15477: vector loop cost: 44.120
remark #15478: estimated potential speedup: 3.28
remark #15487: type converts: 20
remark #15488: --- end vector loop cost summary ---
LOOP END
LOOP END

Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
</tbody>
</table>
Compiler report: ver2

LOOP BEGIN at GSimulation.cpp(137,7)
remark #25085: Preprocess Loops: Moving Out Load and Store
[ GSimulation.cpp(150,4) ]
remark #25085: Preprocess Loops: Moving Out Load and Store
[ GSimulation.cpp(151,4) ]
remark #25085: Preprocess Loops: Moving Out Load and Store
[ GSimulation.cpp(152,4) ]
remark #15415: vectorization support: non-unit strided load was generated for
the variable <this->particles->pos[j][0]>, stride is 10   [ GSimulation.cpp(143,9) ]
remark #15415: vectorization support: non-unit strided load was generated for
the variable <this->particles->pos[j][1]>, stride is 10   [ GSimulation.cpp(144,9) ]
...
remark #15305: vectorization support: vector length 16
remark #15309: vectorization support: normalized vectorization overhead 0.491
remark #15300: LOOP WAS VECTORIZED
remark #15452: unmasked strided loads: 6
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 115
remark #15477: vector cost: 14.500
remark #15478: estimated potential speedup: 7.250
remark #15488: --- end vector cost summary ---
LOOP END

27 Vectorization

Moving out Load/Store

GSimulation.cpp:

...  
for (i = 0; i < n; i++)        // update acceleration
real_type ax_i = particles[i].acc[0];
real_type ay_i = particles[i].acc[1];
real_type az_i = particles[i].acc[2];

for (j = 0; j < n; j++)
{
    real_type distance, dx, dy, dz;
    real_type distanceSqr = 0.0f;
    real_type distanceInv = 0.0f;
    ...
    ax_i += dx * G * particles[j].mass * distanceInv * distanceInv;
    ay_i += ...                //6flops
    az_i += ...                                                            //6flops
}
... // update position and velocity

28 Vectorization
A loop that has been automatically vectorized contains loads from memory locations which are not contiguous in memory → non-unit stride load
The compiler has issued a hardware gather/scatter instructions.

```cpp
struct Particle
{
    public:
        real_type pos[3];
        real_type vel[3];
        real_type acc[3];
        real_type mass;
};

struct ParticleSoA
{
    public:
        real_type *pos_x,*pos_y,*pos_z;
        real_type *vel_x,*vel_y,*vel_z;
        real_type *acc_x,*acc_y,*acc_z
        real_type *mass;
};
```

### SoA: unit stride access

- The **Particle** structure has **strided** access: the distance between 2 consecutive position for different particles is 10 elements.

```cpp
struct Particle
{
    public:
        real_type pos[3];
        real_type vel[3];
        real_type acc[3];
        real_type mass;
};
Particle *particles;
```

```cpp
void GSimulation :: start()
{
    //allocate particles
    particles = new Particle[get_npart()];
    init_pos();
    init_vel();
    init_acc();
    init_mass();
    ...
}
```

- The **ParticleSoA** structure has **unit-stride** access: the distance between 2 consecutive position for different particles is 1 element.

```cpp
struct ParticleSoA
{
    public:
        real_type *pos_x,*pos_y,*pos_z;
        real_type *vel_x,*vel_y,*vel_z;
        real_type *acc_x,*acc_y,*acc_z
        real_type *mass;
};
ParticleSoA *particles;
```

```cpp
void GSimulation :: start()
{
    //allocate particles
    particles = new ParticleSoA[get_npart()];
    particles->pos_x = new real_type[get_npart()];
    particles->pos_y = new real_type[get_npart()];
    particles->pos_z = new real_type[get_npart()];
    particles->vel_x = new real_type[get_npart()];
    particles->vel_y = new real_type[get_npart()];
    particles->vel_z = new real_type[get_npart()];
    ...
}
Live-session

- Go to the folder `code/nbody/ver3`
- Load the appropriate compiler module
- Run `make clean` to remove the old files
- Change the Makefile adding the compiler flags to generate the report: `-qopt-report=5`
- Does the compiler automatically vectorize the inner loop? Why?

Data dependencies

```
LOOP BEGIN at GSimulation.cpp(143,20)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at GSimulation.cpp(146,4)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at GSimulation.cpp(149,6)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed ANTI dependence between this->particles->pos_x[j] (155:3) and this->particles->acc_z[i] (164:3)
  remark #15346: vector dependence: assumed FLOW dependence between this->particles->acc_z[i] (164:3) and this->particles->pos_x[j] (155:3)
  LOOP END
  LOOP END

LOOP BEGIN at GSimulation.cpp(171,4)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between this line 173 and this line 185
  remark #15346: vector dependence: assumed ANTI dependence between this line 173 and this line 185
  LOOP END
  LOOP END
```
Data dependencies

Vectorization changes the order of the operation inside a loop, since each SIMD instruction operates on several data at once. Vectorization is only possible if this does not change the results.

**ANTI** dependence: write-after-read (WAR). Statement i precedes j, and i uses a value that j computes: 2 → 3

**FLOW** (true) dependence: read-after-write (RAW). Statement i precedes j, and i uses a value that j computes: 1 → 2, 2 → 4

1: \( x = 1; \)  
2: \( y = x + 2; \)  
3: \( x = z - w; \)  
4: \( x = y / z; \)

for (i=0; \( i < N-1; \) i++)  
\( a[i] = a[i+i] + b[i]; \)

for (i=0; \( i < N; \) i++)  
\( a[i] = a[i-i] + b[i]; \)

Vectorization: How much we can gain!

Today’s CPUs have different levels of parallelism (see previous slides).

Vectorization is the process of converting a scalar algorithm to one which works on multiple elements in one step.

**SIMD** instructions operate on multiple data elements (128-bits registers). Intel® during the years has increased the number and the size of that registers.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SSE2</strong></td>
<td></td>
<td>a[1]</td>
<td></td>
<td></td>
<td>2 doubles</td>
</tr>
</tbody>
</table>
Requirements for Auto-Vectorization

To be vectorizable, loops **must** meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

```c
void no_vec(float a[], float b[], float c[]) {
    int i = 0.;
    while (i < 100) {
        a[i] = b[i] * c[i];
        // this is a data-dependent exit condition:
        if (a[i] < 0.0)
            break;
        ++i;
    }
}
```

remark: loop was not vectorized:
nonstandard loop is not a vectorization candidate.
Requirements for Auto-Vectorization

To be vectorizable, loops **must** meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

3. **Straight-line code**: the code must not branch inside the loop; do not break the SIMD operation on consecutive data.

4. **The innermost loop of a nest**: the only exception is in the case of prior optimization, like loop unrolling, exchange,...
Requirements for Auto-Vectorization

To be vectorizable, loops **must** meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

3. **Straight-line code**: the code must not branch inside the loop; do not break the SIMD operation on consecutive data.

4. **The innermost loop of a nest**: the only exception is in the case of prior optimization, like loop unrolling, exchange,…

5. **No function call**: the two major exception are for intrinsic math functions and inlined functions.

---

**Intel® compiler directives**

<table>
<thead>
<tr>
<th>Directive</th>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>always</td>
<td>Force vectorization even when it might be not efficient.</td>
</tr>
<tr>
<td></td>
<td>[un]aligned</td>
<td>Use [un]aligned data movement instructions for all array vector references.</td>
</tr>
<tr>
<td></td>
<td>[non]temporal(var1,…)</td>
<td>Do or do not generate non-temporal (streaming) stores for the given array variables. On Intel® MIC architecture, generates a cache-line-evict instruction when the store is known to be aligned.</td>
</tr>
<tr>
<td></td>
<td>[no]vecremainder</td>
<td>Do (not) vectorize the remainder loop when the mail loop is vectorized.</td>
</tr>
<tr>
<td></td>
<td>[no]mask_readwrite</td>
<td>Enables/disables memory speculation causing the generation of [non-]masked loads and stores within conditions.</td>
</tr>
<tr>
<td>simd</td>
<td>vectorlength(n1,…)</td>
<td>Assume safe vectorization for the given vector length values or data type.</td>
</tr>
<tr>
<td></td>
<td>vectorlengthfor(dtype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>private(var1,…)</td>
<td>Which variables are private to each iteration; firstprivate, initial value is broadcasted to all private instances; lastprivate, last value is copied out from the last instance.</td>
</tr>
<tr>
<td></td>
<td>firstprivate(var1,…)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lastprivate(var1,…)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>linear(var1:step1,…)</td>
<td>Letting know the compiler that var1 is incremented by step1 on every iteration of the original loop.</td>
</tr>
<tr>
<td></td>
<td>reduction(opr:var1,…)</td>
<td>Which variables are reduction variables with a given operator.</td>
</tr>
<tr>
<td></td>
<td>[no]assert</td>
<td>Warning or error when vectorization fails.</td>
</tr>
<tr>
<td></td>
<td>[no]vecremainder</td>
<td>Do (not) vectorize the remainder loop when the mail loop is vectorized.</td>
</tr>
</tbody>
</table>

---

From presentation: M. Fernandez, Bayncore
Best practices for vectorization
Vectorization via: #pragma simd

The compiler helps you to vectorize your code performing a series of tests to determine if the vectorization is possible and efficient. With `pragma simd` you inform the compiler to not do these tests and to vectorize.

```
simd-example.c: icc -qopt-report=2 -c simd-example.cpp

void add_floats(float *a, float *b, float *c, float *d, float *e, int n) {
    int i;
    for (i=0; i<n; i++){
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
```

LOOP BEGIN at simd-example.c(3,2)
remark #15344: loop was not vectorized: vector dependence prevents vectorization.
First dependence is shown below. Use level 5 report for details
remark #15346: vector dependence: assumed FLOW dependence between line 4 and line 4
remark #25439: unrolled with remainder by 4
LOOP END
Vectorization via: #pragma simd

The compiler helps you to vectorize your code performing a series of tests to determine if the vectorization is possible and efficient. With `pragma simd` you inform the compiler to not do these tests and to vectorize.

```c
void add_floats(float *a, float *b, float *c, float *d, float *e, int n) {
  int i;
  for (i=0; i<n; i++){
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
  }
}
```

The user can enforce vectorization with `pragma simd`.

What is happening with `pragma vector`? It is still under the discretion of the compiler.
Data shared: #pragma simd reduction

The **pragma simd** gives the developer the *full control* on the vectorization but...

> With great power comes great responsibility!

In case of data shared which needs to be reduced:

```c
double return_sum(float *a, float *b, float *c, int n) {
    double sum=0;
    #pragma simd reduction(+:sum)
    for (int i=0; i<n; i++)
        sum += a[i] + b[i] * c[i];
    return sum;
}
```

Since the loop runs effectively in parallel by doing two (or four, or eight, etc.) operations simultaneously, the variable `sum` is updated by different iterations and then a **race condition** occurs and the results can be wrong. With *reduction* the compiler generate code to work on private copies of `sum` and then gather together to get the correct answer.

---

### Live-session

- Go to the folder `code/nbody/ver3`
- Run `make clean` to remove old files
- Try to use `#pragma omp simd`
Solution in the folder code/nbody/ver3
GSimulation.cpp:

```c
... 
for (i = 0; i < n; i++)        // update acceleration
#pragma omp simd
for (j = 0; j < n; j++)
{
    real_type distance, dx, dy, dz;
    real_type distanceSqr = 0.0f;
    real_type distanceInv = 0.0f;
    ...
    particles->acc_x[i] += dx * G * particles->mass[j] * distanceInv * distanceInv * distanceInv;  //6flops
    particles->acc_y[i] += _
    particles->acc_z[i] += _
}
... // update position and velocity
```

Run the default test case on CPU:

```
./nbody.x
```

```
===============================
Initialize Gravity Simulation
nPart = 2000; nSteps = 500; dt = 0.1
------------------------------------------------
s       dt      kenergy     time (s)    GFlops
------------------------------------------------
  50        5    1.9722e+06  0.37769     15.354
 100       10    2.4651e+06  0.37759     15.358
 150       15    2.2521e+07  0.37747     15.363
 200       20    1.2662e+08  0.37706     15.38
 250       25    4.5866e+08  0.3772     15.374
 300       30    1.2487e+09  0.37717     15.375
 350       35    2.2531e+09  0.37714     15.376
 400       40    3.5807e+09  0.37733     15.368
 450       45    5.2184e+09  0.37712     15.377
 500       50    7.3094e+09  0.37712     15.377

# Number Threads : 1
# Total Time (s)  : 3.7734
# Average Perfomance : 15.374 +- 0.0051804
```
Run the default test case on CPU:
./nbody.x

Initialize Gravity Simulation
nPart = 2000; nSteps = 500; dt = 0.1

<table>
<thead>
<tr>
<th>s</th>
<th>dt</th>
<th>kenergy</th>
<th>time (s)</th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>1.9722e+06</td>
<td>0.37769</td>
<td>15.354</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>2.4651e+06</td>
<td>0.37759</td>
<td>15.358</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>2.2521e+07</td>
<td>0.37747</td>
<td>15.363</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1.2662e+08</td>
<td>0.37706</td>
<td>15.38</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>4.5866e+08</td>
<td>0.3772</td>
<td>15.374</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1.2487e+09</td>
<td>0.37717</td>
<td>15.375</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>2.2531e+09</td>
<td>0.37714</td>
<td>15.376</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>3.5807e+09</td>
<td>0.37733</td>
<td>15.368</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>5.2184e+09</td>
<td>0.37712</td>
<td>15.377</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>7.3094e+09</td>
<td>0.37712</td>
<td>15.377</td>
</tr>
</tbody>
</table>

# Number Threads   : 1
# Total Time (s)   : 3.7734
# Average Performace: 15.374 +- 0.0051804

It is easy to get the wrong results faster!!!
Live-session: solution

- Solution in the folder code/nbody/ver4

GSimulation.cpp:

```c
... for (i = 0; i < n; i++)         // update acceleration
    real_type ax_i = particles[i].acc[0];
    real_type ay_i = particles[i].acc[1];
    real_type az_i = particles[i].acc[2];
    #pragma simd reduction(+:ax_i,ay_i,az_i)
    for (j = 0; j < n; j++)
        { 
            real_type distance, dx, dy, dz;
            real_type distanceSqr = 0.0f;
            real_type distanceInv = 0.0f;
            ...
            ax_i += dx * G * particles[j].mass * distanceInv * distanceInv * distanceInv;      //6flops
            ay_i += ...                //6flops
            az_i += ...                                                            //6flops
        }
... // update position and velocity
```

Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
</tbody>
</table>
Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
</tbody>
</table>

Unaligned access

```
LOOP BEGIN at GSimulation.cpp(159,2)
  <Peeled loop for vectorization>
LOOP END

LOOP BEGIN at GSimulation-nodep.cpp(159,2)
  ...
  remark #15389: vectorization support: reference this->particles->pos_x[j] has unaligned access
  [Gsimul...{168,6}]
  ...
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #1539: vectorization support: normalized vectorization overhead 1.026
  remark #15300: PEEL LOOP WAS VECTORIZED
  remark #1542: entire loop may be executed in remainder
  remark #15454: masked aligned unit stride loads: 1
  remark #15456: masked unaligned unit stride loads: 5
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 189
  remark #15477: vector loop cost: 29.370
  remark #15488: estimated potential speedup: 5.160
  remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at GSimulation.cpp(159,2)
  <Reminder loop for vectorization>
LOOP END
```
Data alignment

The compiler cannot know if your data is aligned to a multiple of the vector register width. This could effect the performance.

A pointer \( p \) is aligned to a memory location on a \( n \)-byte boundary if:

\[(\text{size\_t})p \% n == 0\]

For AVX, alignment to 32byte boundaries (4 DP words) allows a single reference to a cache line for moving 4 DP words into the registers.

Single Cache access for 4 DP words

- cache line 0
- cache line 1
- cache line 2

Load 4DP words

Across Cache line access for 4 DP words

- cache line 0
- cache line 1
- cache line 2

2 Loads 4DP words

32byte Aligned

Non-Aligned

Data alignment

On the Stack: for declared variables the Intel® C/C++ compiler aligned the data naturally:

- \( \text{float } f; //4\text{-byte aligned} \)
- \( \text{double } d; //8\text{-byte aligned} \)

For array data an attribute is necessary:

- \( \text{float array}[N] \_\text{attribute}__()\_\text{aligned}(32); //32\text{-byte aligned} \)

On the Heap: the array can be allocated/deallocate with special functions:

- \#include <malloc.h>
- \( \text{float } *\text{array} = (\text{float}*) \_\text{mm}\_\text{malloc}(N*\text{sizeof(float)}, 32); \)
- \( \_\text{mm}\_\text{free}(\text{array}); \)
Data alignment

SSE: works better with **16 bytes** alignment

Why?: the **XMM** registers are 16 bytes (i.e. 128 bits)

Penalties:
  - **Unaligned** access vs aligned access (but still in the same cache line) 40% worse.
  - **Unaligned** access vs aligned access (but split over cache line) 500% worse.

Rule of thumb: Try to align to the SIMD register size
  - **MMX**: 8 Bytes; **SSE2**: 16 Bytes; **AVX**: 32 Bytes; **AVX512/MIC**: 64 Bytes.

Also try to align blocks of data to **cacheline** size – i.e. 64 Bytes.

---

Peel and reminder loop

The compiler can generate a **Peel** and **Reminder** loop in case where:
- The loop trip count is known only during runtime
- The alignment is not known during compilation

Then the compiler generates a check in code at the beginning of the loop to verify its assumptions. This could cause inefficiency, since every time enters the loops, it does these checks.

```c
for(j = 0; j < N; j++) array[j] = ...  
```

---

The diagram shows the Peel and Reminder loop with vector iterations and cache line boundaries.
Live-session

- Go to the folder code/nbody/ver4
- Run make clean to remove old files
- Try to use the compiler report
- Replace the new/delete statements with the memory alignment functions
- Does the compiler report say what you expect?
- Solution: replace Gsimulation.cpp with Gsimulation-align.cpp

Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4a</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
</tbody>
</table>
Final remarks on vectorization

- Not always the compiler does what we want, we need to give suggestions: `__assume_aligned(...)`. Without this step, the compiler will not detect the optimal alignment for accesses using such arrays. Alignment is generally unknown at compile time.
  - We have changed not too much 😊 in the code.
  - We can give hints if the compiler does not vectorize as we expect. (#pragma vector, #pragma omp simd)
  - Very good speedup: \( \sim 10.2x \)

Enabling vectorization

- Auto Vectorization
- Compiler options
- Guided Vectorization
- Report hints
  - Adding #pragmas
  - Change few lines of code
- Low-level Vectorization
- Vector intrinsics
  - ASM code

Easy to use

Programmer control
Intel® intrinsic instructions

Intrinsics are like library functions, but directly understood by the compiler. They are almost translated into assembly code and are **hardware specific**.

---

Array notation using Intel® Cilk™ Plus

Intel Cilk Plus includes extensions to C and C++ that allows for parallel operations on arrays. The intent is to allow users to express high-level **vector parallel** array operations. This helps the compiler to effectively vectorize the code. Array notation can be used for both static and dynamic arrays.

It is supported in C/C++ Intel compiler and GCC 4.9.

The vectorization become explicit: **array-expression[lower-bound : length : stride]**

```c
int main(int argc, char **argv)
{
    cost int array_size = 10;
    int a[array_size];
    int b[array_size];

    // Initialize array using for loop
    for (int i = 0; i < array_size; i++)
        a[i] = 5;

    // Initialize the array using Array Notation. Since the array is
    // statically allocated, we can use default values for the start index (0)
    // and number of elements (all of them).
    b[] = 5;
    ...
```
Past, present and future of Intel® SIMD

Current Intel® Xeon® Processors
- Multimedia Extensions (MMX) (1997)
- Streaming SIMD Extensions (SSE) (1999)
- Advanced Vector Instructions (AVX) (2008)
- AVX2 (2013)
- AVX-512 (2015)

Future Intel® Co/Processors (including Knights Landing)
- Exponential & Reciprocal (ERI)
- Prefetch Instructions (PFI)
- Foundation Instructions (FI)
- Conflict Detection (CDI)
- Byte and Word Instructions (BWI)
- Double-/Quad- Word (DQI)
- Vector Length Extensions (VLE)

Future Intel® Xeon® Processors

Current Intel® Xeon® Phi Coprocessors (Knights Corner)

Intel® Many Core Instructions (IMCI)

Multi-threading: OpenMP introduction

Shared memory system: the RAM can be accessed by several different CPUs.

OpenMP is designed for multi-processor/core, shared memory machine in UMA/NUMA architecture.

OpenMP webpage: www.openmp.org

- A set of compiler directives and API for multithreading applications
- An explicit (not automatic) programming model, offering the programmer full control over parallelization
Fork-join model

- **OpenMP** uses the fork-join model for parallel execution

- The program starts as a single process: the master thread

- During the execution the master thread creates a team of parallel threads → FORK

- The subsequent part is executed in parallel

- When all the threads terminate the execution, they synchronize and terminate → JOIN

---

Live-session: Nbody example

code/nbody/ver4/GSimulation.cpp:
Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4a</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
<tr>
<td>ver5</td>
<td>OpenMP / 128 threads</td>
<td>1414.1 GFs</td>
</tr>
</tbody>
</table>

Additional compiler optimization

The -fp-model switch lets you choose the floating point semantics at coarse granularity

- fast [=1] allows value unsafe optimizations (default)
- fast=2 allows additional optimizations
- precise value-safe optimizations only

More informations: https://www.nccs.nasa.gov/images/FloatingPoint_consistency.pdf

We all have to think to the 3 points:

- Accuracy
- Reproducibility
- Performance
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4a</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
<tr>
<td>ver5</td>
<td>OpenMP / 128 threads</td>
<td>1414.1 GFs</td>
</tr>
<tr>
<td>ver6</td>
<td>-fp-model fast=2 / 128 threads</td>
<td>1702.9 GFs</td>
</tr>
</tbody>
</table>

### Loop tiling
**KNL memory model**

- On-package high-bandwidth memory (HBM) – MCDRAM
- Optimized for arithmetic performance and bandwidth (not latency)

---

**Cache optimization**

**Original:**
```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        ... *b[i][j];
```

**Tiled:**
```
for (ii=0; ii<m; ii+=TILE)
    for (j=0; j<n; j++)
        for (i=ii; i<ii+TILE; i++)
            ... *b[i][j];
```

- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

**Cache size:** 4  
**TILE=4**

(must be tuned to cache size)

Cache hit rate without tiling: 0%  
Cache hit rate with tiling: 50%

---

**Loop Tiling**
Live-session: Nbody example

code/nbody/ver6/GSimulation.cpp:

Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4align</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
<tr>
<td>ver5</td>
<td>OpenMP / 128 threads</td>
<td>1414.1 GFs</td>
</tr>
<tr>
<td>ver6</td>
<td>-fp-model fast=2 / 128 threads</td>
<td>1702.9 GFs</td>
</tr>
<tr>
<td>ver6tile</td>
<td>loop tiling /128 threads / 65536 particles</td>
<td>2322.2 GFs</td>
</tr>
</tbody>
</table>
Learn how we did it in IPCC


**MC² SERIES**

**LEARN HOW THEY DID IT**

Performance Optimization of Smoothed Particle Hydrodynamics Algorithms for Multi/Many-Core Architectures

March 7 - 1-Hour Webinar

Register Now

Dr. Fabio Soares, Sr. HPC Application Specialist
Leibniz Supercomputing Centre

*MC² series - experts in computational disciplines share their experience with performance optimization methods used in real-life applications.*
KNL tools

Dr. Fabio Baruffa
fabio.baruffa@lrz.de

Profiling tools
Which tool do I use? A roadmap to optimization

We will focus on tools developed by Intel, available to users of the LRZ systems.

Again, we will skip the MPI layer.

VTune is a very rich tool, we will touch it only quickly.

We will dedicate more time (and hands-on) to Advisor.

Profilong with Intel® VTune Amplifier XE

- Powerful tool for analyzing the node-level performance
  - Multiple programming languages (C/C++, Fortran, .NET, Java, Assembly)
  - Support for all latest Intel® processors (incl. Intel® MIC / Broadwell micro-architectures)
- Performance analysis at different levels
  - High-level (code analysis, parallelization efficiency), no special rights needed
  - Low-level (inspection of all architectural components), module driver is required
  - Processor-specific analysis (e.g., utilisation of vector units on Intel® MIC)
- Minimal execution time overhead
  - No recompilation or special linking needed
  - H/W counter sampling and multiplexing → all interesting events gathered once
- Multiplatform (Windows/Linux, 32/64-bit) + complete command-line interface
- Can produce very large traces (∼400MB per min. of exec. time)
Hot-spot guided optimization

Typical workflow

1. Compile code with -g -O2 or -g -O3
2. Set the environment variables or use a wrapper script
3. Tweak code input for a short representative run

VTune
find top hotspots

Optimize
eliminate issues, reduce hotspot time

Compiler
identify issues in optimization report

Performance overview

Wall-clock time
Cumulative CPU time
Performance bottlenecks are highlighted in red

Overall CPU usage
Threads behaviour

Function level profiling

Time line of the application

7 Profiling tools

Threads behaviour: locks and waits

Threads are spinning!

Threads sleeping

Useful work

Concurrency

Synchronization

8 Profiling tools
Source code view

Closing remarks

The tool is useful and can be used to find:

- Hotspots in the code and possible bottlenecks
- Characterization of the parallelization efficiency
- Possible locks and spinning threads in the application

- More advanced profiling is provided using special kernel modules (memory bandwidth, hardware event-based sampling,
  
  - Instrumenting the code for reducing the amount of profiling part in the application
Roofline model

The **roofline model** allows to understand the performance limit of an application, based on operational intensity (algorithm specific) and on hardware specifics (memory bandwidth).

The **expected performance** is defined as:

\[
P = \min(P_{\text{max}}, l \cdot b_s)
\]

- \(P_{\text{max}}\) = **Applicable peak performance** of a loop, assuming that data comes from L1 cache (this is not necessary Ppeak)
- \(l\) = **Computational/arithmetic intensity** ("work" per Byte transferred) over the slowest data path utilized ("the bottleneck")
- \(b_s\) = **Applicable peak bandwidth** of the slowest data path utilized
Roofline model

Peak performance of 2-socket Ivy-Bridge node

Peak: 448 GFlops/s

Stream BW: 78.5 GB/s
Roofline model

Peak performance of 2-socket Ivy-Bridge node

Peak:
448 GFlops/s

Stream BW:
78.5 GB/s

Memory-bound

Core-bound

15 Roofline model

16 Roofline model

Peak:
448 GFlops/s

Stream BW:
78.5 GB/s

Core Performance:
22.4 GFlops/s
**Arithmetic intensity**

The core parameter behind the Roofline model is **Arithmetic Intensity.** Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes). A **BLAS-1** vector-vector increment (\( x[i] += y[i] \)) would have a very low arithmetic intensity of 0.0417 (N FLOPS / 24N Bytes) and would be independent of the vector size.

---

**Roofline model: example daxpy.cpp**

- **DAXPY:** \( y[i] = a \times x[i] + y[i] \), double precision, \( i = 0, \ldots, N-1 \)

- **2 Flops** for each element of \( x \) and \( y \)
  - well balanced: 1 multiply, 1 add
  - need to load \( x[i] \) and \( y[i] \) for each ‘i’: 2x8 = 16 bytes (\( a \) is the register)
  - need to write out \( y[i] \): another 8 bytes
  - **Arithmetic intensity:** 2 FLOPS / 24 Bytes = 1/12 = 0.083
  - Speed of light performance (working from main memory)
    - on Ivy-Bridge with mem bw of 38 GB/s: 3.6 GFlops/s
      - even the socket peak is 166.4 GFlops/s
    - If \( x \) and \( y \) fit into **cache**, higher cache BW → higher performance
Intel® Advisor XE

Profiling with Intel® Advisor XE

- Modern HPC processors explore different levels of parallelism: between the cores (multi-threading), within a core (vectorization)

- Adapting applications to take advantage of so high parallelism is defined often as code modernization

- The Intel® Advisor XE is a software tool for vectorization optimization and thread prototyping

- The tool guides the software developer to resolve issues during the vectorization process
Creating a new project via GUI

Setting up the application

Profiling tools
Vectorization analysis workflow

1. Run Survey
2. Check the Trip-counts (Mark-up Loops)
3. Check Dependencies
4. Check Memory Access Patterns
5. Deeper-dive analysis

Start → Edit & compile → Take Snapshot

5 Steps to efficient vectorization

1. Compiler diagnostics + Performance Data + SIMD efficiency information
2. Guidance: detect problem and recommend how to fix it
3. "Accurate" Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads
4. Loop-Carried Dependency Analysis
5. Memory Access Patterns Analysis

Profiling tools
Profiling with Intel® Advisor XE

- How to improve performance
- ISA
- Hot-spots
- What prevents vectorization
- Report from the loop

Profiling tools

- Vectorization informations
- Number of vector registers
- Traits
- Application intensity

Profiling with Intel® Advisor XE
Profiling with Intel® Advisor XE

Useful suggestion

Recommendations to enable vectorization

Loop vectorized (ver2)

Vector length

Loop analytics

Vectorization efficiency
Loop vectorized and improved efficiency

Memory access pattern

Stride distribution
Roofline analysis with Advisor - base

Roofline analysis with Advisor - ver4
Closing remarks: 6 steps vectorization methodology

1. Measure baseline release build performance: define a metric which makes sense for the code

2. Determine hotspots using Intel® VTune: most-time consuming functions in the application


4. Get advise using Intel® Advisor: use the vectorization analysis capability of the tool

5. Implement vectorization recommendations

more informations: https://software.intel.com/en-us/articles/vectorization-toolkit
Many-core Programming with OpenMP* 4.x

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group
(michael.klemm@intel.com)

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015 Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

SOFTWARE AND SERVICES
Outline

• Very brief Introduction to OpenMP
• OpenMP for Many-core Processors
• OpenMP SIMD Constructs
• Task-generating loops
• Affinity Control

Brief Introduction to OpenMP
**OpenMP API**

- De-facto standard, OpenMP 4.0 out since July 2013
- API for C/C++ and Fortran for shared-memory parallel programming
- Based on directives (pragmas in C/C++)
- Portable across vendors and platforms
- Supports various types of parallelism

**OpenMP History**

In spring 2019, vendors and the DOE agree on the spelling of parallel loops and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released.

First hybrid applications with Fortran and OpenMP appear.

C++/OpenMP, the group of OpenMP users, is formed and organizes workshops on OpenMP in North America, Europe, and Asia.

The merge of Fortran and C/C++ specifications begins.

Unified Fortran and C/C++: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendors.

Incorporates task parallelism: A hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.

Supports offloading execution to accelerators and coprocessor devices, SIMD parallelism, and more. Repurposes OpenMP beyond traditional boundaries.

OpenMP supports tasklets, task priorities, dynamic loops, and hints for load balancing. Offloading now supports asynchronous execution and dependencies to host execution.

**SOFTWARE AND SERVICES**

- Permanent ARB
- Auxiliary ARB
OpenMP Platform Features

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Group of computers communicating through fast interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessors/Accelerators</td>
<td>Special compute devices attached to the local node through special interconnect</td>
</tr>
<tr>
<td>Node</td>
<td>Group of processors communicating through shared memory</td>
</tr>
<tr>
<td>Socket</td>
<td>Group of cores communicating through shared cache</td>
</tr>
<tr>
<td>Core</td>
<td>Group of functional units communicating through registers</td>
</tr>
<tr>
<td>Hyper-Threads</td>
<td>Group of thread contexts sharing functional units</td>
</tr>
<tr>
<td>Superscalar</td>
<td>Group of instructions sharing functional units</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Sequence of instructions sharing functional units</td>
</tr>
<tr>
<td>Vector</td>
<td>Single instruction using multiple functional units</td>
</tr>
</tbody>
</table>

OpenMP 3.0 in Three Slides

```c
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++)
    {
        ...
    }

    #pragma omp for
    for (i = 0; i < N; i++)
    {
        ...
    }
```
OpenMP 3.0 in Three Slides /2

double a[N];
double l, s = 0;
#pragma omp parallel for reduction(+:s) private(l) \ schedule(static,4)

for (i = 0; i < N; i++)
{
    l = log(a[i]);
    s += l;
}

Software and Services

OpenMP 3.0 in Three Slides /3

#pragma omp parallel
#pragma omp single
for(e = l->first; e; e = e->next)
    #pragma omp task
    process(e);

Software and Services
OpenMP for Many-core Processors

Intel Xeon Phi Tile Architecture

Software and Services
SIMD on Intel® Architecture

**Good News and Bad News!**

**Goods news are:**
- OpenMP has already everything you need to get to good many-core performance
- You can combine individual OpenMP features to match your code with the underlying hardware structure

**Bad news are:**
- Simplistically adding pragmas to loops does no longer cut it!
- You need to think about parallelism at much higher levels
- My above statement was a bit of a lie...
  - Memory allocation still missing from OpenMP
  - Still requires use of non-OpenMP APIs
OpenMP SIMD Constructs

Why Auto-vectorizers Fail

• Data dependencies
• Other potential reasons
  • Alignment
  • Function calls in loop block
  • Complex control flow / conditional branches
  • Loop not “countable”
  • E.g. upper bound not a runtime constant
  • Mixed data types
  • Non-unit stride between elements
  • Loop body too complex (register pressure)
  • Vectorization seems inefficient
• Many more … but less likely to occur
Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - Control-flow dependence
  - Data dependence
  - Dependencies can be carried over between loop iterations

- Important flavors of data dependencies

**FLOW**

\[
\begin{align*}
\text{s1: } a &= 40 \\
\text{b} &= 21 \\
\text{s2: } c &= a + 2
\end{align*}
\]

**ANTI**

\[
\begin{align*}
\text{b} &= 40 \\
\text{s1: } a &= b + 1 \\
\text{s2: } b &= 21
\end{align*}
\]

Loop-Carried Dependencies

- Dependencies may occur across loop iterations
  - Loop-carried dependency
- The following code contains such a dependency:

```c
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

- Some iterations of the loop have to complete before the next iteration can run
  - Simple trick: Can you reverse the loop w/o getting wrong results?
Loop-Carried Dependencies

• Can we parallelize or vectorize the loop?
  • Parallelization: no
    (except for very specific loop schedules)
  • Vectorization: yes
    (if vector length is shorter than any distance of any dependency)

Example: Loop not Countable

• “Loop not Countable” plus “Assumed Dependencies”

```c
typedef struct {
    float* data;
    size_t size;
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```
In a Time before OpenMP 4.0

• Programmers had to rely on auto-vectorization...
• ... or to use vendor-specific extensions
  • Programming models (e.g., Intel® Cilk™ Plus)
  • Compiler pragmas (e.g., #pragma vector)
  • Low-level constructs (e.g., _mm_add_pd())

```c
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

OpenMP SIMD Loop Construct

• Vectorize a loop nest
  • Cut loop into chunks that fit a SIMD vector register
  • No parallelization of the loop body

• Syntax (C/C++)
  ```
  #pragma omp [for] simd [clause[,] clause],...
  for-loops
  ```

• Syntax (Fortran)
  ```
  !$omp [do] simd [clause[,] clause],...
  do-loops
  ```
Example

```c
void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp simd reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
```

Example: Worksharing w/ SIMD

```c
void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
```
Data Sharing Clauses

- **private(var-list):**
  Uninitialized vectors for variables in *var-list*
  
  ![Diagram](42) ![Uninitialized Vectors](42 ? ? ? ?)

- **firstprivate(var-list):**
  Initialized vectors for variables in *var-list*
  
  ![Diagram](42) ![Initialized Vectors](42 42 42 42)

- **reduction(op:var-list):**
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct
  
  ![Diagram](12 5 8 17) ![Reduction](42)

SIMD Loop Clauses

- **safelen (length)**
  - Maximum number of iterations that can run concurrently without breaking a dependence
  - in practice, maximum vector length

- **linear (list[:linear-step])**
  - The variable’s value is in relationship with the iteration number
    
    \[ x_i = x_{\text{orig}} + i \times \text{linear-step} \]

- **aligned (list[:alignment])**
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture

- **collapse (n)**
Be Careful What You Wish For...

void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum) \  
        schedule(static, 5)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}

• You should choose chunk sizes that are multiples of the SIMD length
  • Remainder loops are not triggered
  • Likely better performance

• In the above example ...
  • and AVX-512, the code will only execute the remainder loop!
  • and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 Simplifies SIMD Chunks

void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum) \  
        schedule(simd: static, 5)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}

• Chooses chunk sizes that are multiples of the SIMD length
  • First and last chunk may be slightly different to fix alignment
    and to handle loops that are not exact multiples of SIMD width
  • Remainder loops are not triggered
  • Likely better performance
SIMD Function Vectorization

float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
    #pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}

• Declare one or more functions to be compiled for calls from a SIMD-parallel loop

• Syntax (C/C++):
  
  #pragma omp declare simd [clause[[],] clause],...]
  [#pragma omp declare simd [clause[[],] clause],...]]
  [...]  
  function-definition-or-declaration

• Syntax (Fortran):
  
  !$omp declare simd (proc-name-list)
SIMD Function Vectorization

```c
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
    #pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```

```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}

vec8 distsq_v(vec8 x, vec8 y) {
    return (x - y) * (x - y);
}
```

```c
vd = min_v(distsq_v(va, vb), vc)
```

SIMD Function Vectorization

- **simdlen (length)**
  - generate function to support a given vector length
- **uniform (argument-list)**
  - argument has a constant value between the iterations of a given loop
- **inbranch**
  - function always called from inside an if statement
- **notinbranch**
  - function never called from inside an if statement
- **linear (argument-list[:linear-step])**
- **aligned (argument-list[:alignment])**
- **reduction (operator:list)**
inbranch & notinbranch

```c
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
    #pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}

for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}

SIMD Constructs & Performance

![Graph showing relative speed-up for various benchmarks with different speed-up rates.]

Task-generating Loops

Issues with Traditional Worksharing

- Worksharing constructs do not compose well
- Pathological example: parallel `dgemm` in MKL

```c
void example() {
#pragma omp parallel
{
    compute_in_parallel(A);
    compute_in_parallel_too(B);
    // dgemm is either parallel or sequential
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
               m, n, k, alpha, A, k, B, n, beta, C, n);
}
}
```

- Writing such code either
  - oversubscribes the system,
  - yields bad performance due to OpenMP overheads, or
  - needs a lot of glue code to use sequential `dgemm` only for sub-matrixes
Issues with Traditional Worksharing /2

- Worksharing constructs do not compose well
- Pathological example: load imbalance

```c
void load_imbalance() {
    long_running_task() // can execute concurrently

    for (int i = 0; i < N; i++) { // can execute concurrently
        for (int j = 0; j < M; j++) {
            loop_body(i, j);
        }
    }
}
```

- Writing such code requires
  - nested parallelism,
  - manual, non-portable fine-tuning, and
  - a lot of care to get the load balance right.

Ragged Fork/Join

- Traditional worksharing can lead to ragged fork/join patterns

```c
void example() {
    compute_in_parallel(A);

    compute_in_parallel_too(B);

    cblas_dgemm(..., A, B, ...);
}
```
Example: Sparse CG

```c
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * bnrm2;
    if (sc->residual <= sc->tolerance)
        break;
    rho_old = rho;
}

void matvec(Matrix *A, double *x, double *y) {
    // ...
    #pragma omp taskloop 
        private(i,j,is,ie,j0,y0) 
        schedule(static)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    } 
    // ...
}
```

The taskloop Construct

- Parallelize a loop using OpenMP tasks
  - Cut loop into chunks
  - Create a task for each loop chunk

- Syntax (C/C++)
  ```
  #pragma omp taskloop [simd] [clause[[,] clause],...]
  for-loops
  ```

- Syntax (Fortran)
  ```
  !$omp taskloop[simd] [clause[[,] clause],...]
  do-loops
  !$omp end taskloop [simd]
  ```
Clauses for taskloop Construct

• Taskloop constructs inherit clause both from worksharing constructs and the task construct
  • shared, private
  • firstprivate, lastprivate
  • default
  • collapse
  • final, untied, mergeable

• grainsize(grain-size)
  Chunks have at least grain-size and max 2*grain-size loop iterations

• num_tasks(num-tasks)
  Create num-tasks tasks for iterations of the loop

Example: task and taskloop

```c
void load_imbalance() {
  #pragma omp taskgroup
  {
    #pragma omp task
    long_running_task() // can execute concurrently

    #pragma omp taskloop collapse(2) grainsize(500) nogroup
    for (int i = 0; i < N; i++) { // can execute concurrently
        for (int j = 0; j < M; j++) {
            loop_body(i, j);
        }
    }
  }
}
```
Example: Sparse CG, taskloop

```c
#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * bnrm2;
    if (sc->residual <= sc->tolerance)
        break;
    rho_old = rho;
}
```

```c
void matvec(Matrix *A, double *x, double *y) {

    // ...

    #pragma omp taskloop private(j,is,ie,j0,y0) grain_size(500)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

Performance of Sparse CG w/ Tasks

X. Teruel, M. Klemm, K. Li, X. Martorell, S.L. Olivier, and C. Terboven. A Proposal for Task-Generating Loops in OpenMP. In A.P. Rendell et al., editor, International Workshop on OpenMP, pages 1-14, Canberra, Australia, September 2013: LNCS 8122
NUMA is here to Stay…

- (Almost) all multi-socket compute servers are NUMA systems
  - Different access latencies for different memory locations
  - Different bandwidth observed for different memory locations
- Example: Intel® Xeon E5-2600v2 Series processor

Thread Affinity – Why It Matters?

STREAM Triad, Intel® Xeon E5-2697v2

- compact, par
- scatter, par
- compact, seq
- scatter, seq
Thread Affinity – Processor Binding

Binding strategies depends on machine and the app

- Putting threads far, i.e. on different packages
  - (May) improve the aggregated memory bandwidth
  - (May) improve the combined cache size
  - (May) decrease performance of synchronization constructs

- Putting threads close together, i.e. on two adjacent cores which possible share the cache
  - (May) improve performance of synchronization constructs
  - (May) decrease the available memory bandwidth and cache size (per thread)
Thread Affinity in OpenMP* 4.0

- OpenMP 4.0 introduces the concept of **places**...
  - set of threads running on one or more processors
  - can be defined by the user
  - pre-defined places available:
    - threads: one place per hyper-thread
    - cores: one place exists per physical core
    - sockets: one place per processor package

... and affinity **policies**...
- spread: spread OpenMP threads evenly among the places
- close: pack OpenMP threads near master thread
- master: collocate OpenMP thread with master thread

... and means to control these settings
- Environment variables `OMP_PLACES` and `OMP_PROC_BIND`
- clause `proc_bind` for parallel regions

Thread Affinity Example

- Example (Intel® Xeon Phi™ Processor):
  Distribute outer region, keep inner regions close

```c
OMP_PLACES=cores(8); OMP_NUM_THREADS=4,4
#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)
```

![Diagram showing thread affinity example](image)
We’re Almost Through

- There are so many things in OpenMP today
  - Can’t cover all of them in 90 minutes!

- OpenMP 4.0 and 4.5 have more to offer!
  - Improved Fortran 2003 support
  - User-defined reductions
  - Task dependencies
  - Cancellation
  - “doacross” Loops

- We can chat about these features in 1:1s, FTFs, phone calls, or in emails 😊

The last Slide...

- OpenMP 4.5 is not only a bugfix release
  - Task-generating loops
  - Locks with hints
  - Improved support for offloading (if it matters to you)

- Work on OpenMP 5.0 has already been started
  - Expected release during Supercomputing 2018
  - OpenMP 5.0 Beta is scheduled for Supercomputing 2017
  - Features being discussed:
    - Bugfixes 😊
    - Futures
    - Error handling
    - Transactional memory
    - Extensions to tasking
    - Fortran 2008 support
    - C++1x support
    - Data locality and affinity
Performance Engineering Tasks: Software side

Optimizing software for a specific hardware requires to align several orthogonal targets.

**Software side:** Reduce algorithmic and processor **work**

1. Reduce algorithmic work
2. Minimize processor work

Processor work consists of:
- **Instruction execution**
- **Data transfers**
Performance Engineering Tasks: Hardware

Parallelism: Horizontal dimension

3 Distribute work and data for optimal utilization of parallel resources

Memory

L3
L3
L3
L3

L3
L3
L3
L3

L2
L2
L2
L2

L2
L2
L2
L2

L1
5

Use most effective execution units on chip

Avoid bottlenecks

Data paths: Vertical dimension

Technologies Driving Performance

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP SIMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td>13</td>
<td>200</td>
<td>1.1</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ILP **Obstacle**: Not more parallelism available
- Clock **Obstacle**: Heat dissipation
- Multi-Manycore **Obstacle**: Getting data to/from cores

Flavors of improvements
- Pure speed increase: Clock
- Transparent solutions
- Explicit solutions

Strategies
- Parallelism
- Specialisation
History of Intel hardware developments

The real picture
Finding the right compromise

Maximum DP floating point (FP) performance

\[ P_{\text{core}} = n_{\text{super}}^{FP} \cdot n_{\text{FMA}} \cdot n_{\text{SIMD}} \cdot f \]

<table>
<thead>
<tr>
<th>uArch</th>
<th>[n_{\text{super}}^{FP}]</th>
<th>[n_{\text{FMA}}]</th>
<th>[n_{\text{SIMD}}]</th>
<th>[n_{\text{cores}}]</th>
<th>Release</th>
<th>Model</th>
<th>[P_{\text{core}}] [GF/s]</th>
<th>[P_{\text{chip}}] [GF/s]</th>
<th>[P_{\text{serial}}] [GF/s]</th>
<th>TDP</th>
<th>GF/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Q1/2012</td>
<td>E5-2680</td>
<td>11.7</td>
<td>173</td>
<td>7</td>
<td>130</td>
<td>1.33</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>Q3/2013</td>
<td>E5-2690-v2</td>
<td>24</td>
<td>240</td>
<td>7.2</td>
<td>130</td>
<td>1.85</td>
</tr>
<tr>
<td>KNC</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>61</td>
<td>Q2/2014</td>
<td>7120A</td>
<td>10.6</td>
<td>1210</td>
<td>1.33</td>
<td>300</td>
<td>4.03</td>
</tr>
<tr>
<td>Haswell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td>Q3/2014</td>
<td>E5-2695-v3</td>
<td>21.6</td>
<td>425</td>
<td>6.6</td>
<td>120</td>
<td>3.54</td>
</tr>
<tr>
<td>Broadwell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>22</td>
<td>Q1/2016</td>
<td>E5-2699-v4</td>
<td>17.6</td>
<td>704</td>
<td>7.2</td>
<td>145</td>
<td>4.85</td>
</tr>
<tr>
<td>Pascal</td>
<td>1</td>
<td>2</td>
<td>32</td>
<td>56</td>
<td>Q2/2016</td>
<td>GP100</td>
<td>36.8</td>
<td>4700</td>
<td>1.5</td>
<td>300</td>
<td>15.67</td>
</tr>
<tr>
<td>KNL</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>72</td>
<td>Q4/2016</td>
<td>7290F</td>
<td>35.2</td>
<td>2995</td>
<td>3.4</td>
<td>260</td>
<td>11.52</td>
</tr>
<tr>
<td>Skylake</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>26</td>
<td>Q3/2017</td>
<td>8170</td>
<td>23.4</td>
<td>1581</td>
<td>7.6</td>
<td>165</td>
<td>9.58</td>
</tr>
</tbody>
</table>
HARDWARE OPTIMIZATIONS FOR SINGLE-CORE EXECUTION

- SIMD
- SMT
- Memory hierarchy

KNL architecture

Core can retire 2 instructions per cycle
Core details: Simultaneous multi-threading (SMT)

Recommendations for data structure layout

- Promote temporal and spatial locality
- Enable packed (block wise) load/store of data
- Memory locality (placement)
- Avoid false cache line sharing
- Access data in long streams to enable efficient latency hiding

Above requirements may collide with object oriented programming paradigm: array of structures vs structure of arrays
Comparison memory hierarchies

<table>
<thead>
<tr>
<th></th>
<th>Intel Broadwell-EP</th>
<th>Intel Xeon Phi KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 D-Cache</td>
<td>32 kB</td>
<td>32 kB</td>
</tr>
<tr>
<td>L2</td>
<td>256 kB</td>
<td>1 MB shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 MB total</td>
</tr>
<tr>
<td>L3</td>
<td>18 x 2.5 MB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>45 MB total (shared)</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>4 channels DDR4-2400</td>
<td>6 channels DDR4-2133</td>
</tr>
<tr>
<td>Secondary Memory</td>
<td>-</td>
<td>16 GB MCDRAM</td>
</tr>
<tr>
<td>Peak Bandwidth</td>
<td>76.8 GB/s</td>
<td>102 GB/s, 450 GB/s</td>
</tr>
<tr>
<td>Update Bandwidth</td>
<td>98 GB/s (81%)</td>
<td>168 GB/s (53%)</td>
</tr>
</tbody>
</table>

Further differences:
- LLC on Xeon Phi is not shared
- Different MCDRAM modes are available: cache, flat, hybrid
- Latency DDR ca. 125ns and MCDRAM ca. 150ns

PARALLEL RESOURCES
SIMD
SIMD processing – Basics

- **Single Instruction Multiple Data (SIMD)** operations allow the **concurrent execution** of the **same operation** on “**wide**” registers.
- x86 SIMD instruction sets:
  - AVX-512: register width = 512 Bit → 8 DP floating point operands
  - AVX: register width = 256 Bit → 4 DP floating point operands
- Adding two registers holding double precision floating point operands

---

Data types in 32-byte SIMD registers

- Supported data types depend on actual SIMD instruction set

- double: double: double: double
- int: int: int: int: int: int: int: int
- Scalar slot
SIMD processing – Basics

- Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n; i++)
    C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n; i+=4){
    C[i]  =A[i]  +B[i];
    C[i+1]=A[i+1]+B[i+1];
}
```

//remainder loop handling

Load 256 Bits starting from address of A[i] to register R0

Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]

//remainder loop handling

LABEL1:
VLOAD R0 ← A[i]
VLOAD R1 ← B[i]
V64ADD[R0,R1] → R2
VSTORE R2 → C[i]
i←i+4
i<(n-4)? JMP LABEL1

//remainder loop handling
SIMD processing – Basics

- No SIMD vectorization for loops with data dependencies:

```c
for(int i=0; i<n; i++)
    A[i] = A[i-1] * s;
```

- “Pointer aliasing” may prevent SIMDification

```c
void scale_shift(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
        C[i] = A[i] + B[i];
}
```

- C/C++ allows that \( A \rightarrow \& C[-1] \) and \( B \rightarrow \& C[-2] \)
  \( \rightarrow C[i] = C[i-1] + C[i-2] \): dependency \( \rightarrow \) No SIMD

- If “pointer aliasing” is not used, tell it to the compiler, e.g. use
  `-fno-alias` switch for Intel compiler \( \rightarrow \) SIMD

Why and how?

Why check the assembly code?

- Sometimes the only way to make sure the compiler “did the right thing”
  - Example: “LOOP WAS VECTORIZED” message is printed, but Loads & Stores may still be scalar!

- Get the assembler code (Intel compiler):

  ```
  icc -S -O3 triad.c -o a.out
  ```

- Disassemble Executable:

  ```
  objdump -d ./a.out | less
  ```

The x86 ISA is documented in:

- Intel Software Development Manual SDM
- Intel Architecture Instruction Set Extensions Programming Reference

253
Basics of the x86-64 ISA

- Instructions have 0 to 3 operands (4 with AVX-512)
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two assembler syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: A[i] equivalent to *(A+i) (a pointer has a type: A+i*8)

```assembly
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```assembly
401b9f: 0f 29 5c c7 30 movaps %xmm3, 0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08 add $0x8,%rax
401ba8: 78 a6 js 401b50 <triad_asm+0x4b>
```

Basics of the x86-64 ISA with extensions

16 general Purpose Registers (64bit):
rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15
alias with eight 32 bit register set:
eax, ebx, ecx, edx, esi, edi, esp, ebp
8 opmask registers (16bit or 64bit, AVX512 only):
k0–k7
Floating Point SIMD Registers:
xmm0–xmm15 (xmm31) SSE (128bit) alias with 256-bit and 512-bit registers
ymm0–ymm15 (xmm31) AVX (256bit) alias with 512-bit registers
zmm0–zmm31 AVX-512 (512bit)

SIMD instructions are distinguished by:
VEX/EVEX prefix: v
Operation: mul, add, mov
Modifier: nontemporal (nt), unaligned (u), aligned (a), high (h)
Width: scalar (s), packed (p)
Data type: single (s), double (d)
ISA support on KNL

KNL supports all legacy ISA extensions:

- MMX, SSE, AVX, AVX2

Furthermore KNL supports:

- AVX-512 Foundation (F), KNL and Skylake
- AVX-512 Conflict Detection Instructions (CD), KNL and Skylake
- AVX-512 Exponential and Reciprocal Instructions (ER), KNL
- AVX-512 Prefetch Instructions (PF), KNL

AVX-512 extensions only supported on Skylake:

- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

ISA Documentation:

*Intel Architecture Instruction Set Extensions Programming Reference*

---

Architecture specific issues KNC vs. KNL

**KNC architectural issues**

- Fragile single core performance (in-order, pairing, SMT)
- No proper hardware prefetching
- Shared access on segmented LLC costly

**KNL fixes most of these issues and is more accessible!**

**Advices for KNL**

- 1 thread per core is usually best, sometime two threads per core
- Large pages can improve performance significantly (2M,1G)
- Consider the *-no-prec-div* option to enable AVX-512 ER instructions
- Aggressive software prefetching is usually not necessary
- MCDRAM is the preferred target memory (try cache mode first)
- Alignment restrictions and penalties are similar to Xeon. We experienced a benefit from alignment to page size with the MCDRAM.
Example for masked execution

Masking for predication is very helpful in cases such as e.g. remainder loop handling or conditional handling.

Gather instruction interface on KNC and Haswell

KNC:

\[ \text{kxnor } k2, k2 \]

..L100:

\[ \text{vgatherdps } zmm13\{k2\}, \left[ \text{rdi } + \text{ zmm17 } \ast \text{ 4} \right] \]
\[ \text{jkzd } k2, \text{..L101} \]
\[ \text{vgatherdps } zmm13\{k2\}, \left[ \text{rdi } + \text{ zmm17 } \ast \text{ 4} \right] \]
\[ \text{jknzd } k2, \text{..L100} \]

..L101:

Haswell:

\[ \text{vpcmpeqw } ymm7, ymm7, ymm7 \]
\[ \text{vgatherdps } ymm15, \left[ \text{rdi } + \text{ ymm11 } \ast \text{ 4} \right], ymm7 \]
Gather microbenchmarking results

<table>
<thead>
<tr>
<th></th>
<th>Knight Corner</th>
<th>Haswell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Cache</td>
<td>L2 Cache</td>
</tr>
<tr>
<td></td>
<td>Instruction</td>
<td>Loop</td>
</tr>
<tr>
<td>16 per CL</td>
<td>9.0</td>
<td>13.6</td>
</tr>
<tr>
<td>8 per CL</td>
<td>4.2</td>
<td>9.4</td>
</tr>
<tr>
<td>4 per CL</td>
<td>3.7</td>
<td>9.1</td>
</tr>
<tr>
<td>2 per CL</td>
<td>2.9</td>
<td>8.6</td>
</tr>
<tr>
<td>1 per CL</td>
<td>2.3</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Serialization for loading several items per CL

No working prefetching for gather on KNC

Case Study: Simplest code for the summation of the elements of a vector (single precision)

```c
float sum = 0.0;
for (int j=0; j<size; j++){
    sum += data[j];
}
```

Instruction code:

```
401d08:   f3 0f 58 04 82
401d0d:   48 83 c0 01
401d11:   39 c7
401d13:   77 f3
```

To get object code use `objdump -d` on object file or executable or compile with `-s`
Summation code (single precision): Optimizations

1:
addss xmm0, [rsi + rax * 4]
add rax, 1
cmp eax,edi
js 1b

3 cycles add pipeline latency

Unrolling with sub-sums to break up register dependency

1:
addss xmm0, [rsi + rax * 4]
addss xmm1, [rsi + rax * 4 + 4]
addss xmm2, [rsi + rax * 4 + 8]
addss xmm3, [rsi + rax * 4 + 12]
add rax, 4
cmp eax,edi
js 1b

SSE SIMD vectorization

SIMD processing – The whole picture

SIMD influences instruction execution in the core – other runtime contributions stay the same!

Comparing total execution time:

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SSE</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>AVX</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total runtime with data loaded from memory:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>24</td>
</tr>
<tr>
<td>SSE</td>
<td>12</td>
</tr>
<tr>
<td>AVX</td>
<td>10</td>
</tr>
</tbody>
</table>

SIMD only effective if runtime is dominated by instructions execution!
Summation code with AVX-512 (single core)

1:

\[
\begin{align*}
\text{vaddps } & \text{zmm0, zmm0, } [\text{rsi + rax } \times 4] \\
\text{vaddps } & \text{zmm1, zmm1, } [\text{rsi + rax } \times 4 + 64] \\
\text{vaddps } & \text{zmm2, zmm2, } [\text{rsi + rax } \times 4 + 128] \\
\text{vaddps } & \text{zmm3, zmm3, } [\text{rsi + rax } \times 4 + 192] \\
\text{add rax, 64}
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX-512 plain</td>
<td>12942 MFlops/s</td>
<td>12977 MFlops/s</td>
<td>2256 MFlops/s</td>
</tr>
<tr>
<td>SMT1</td>
<td>1.60 cycles/CL</td>
<td>1.60 cycles/CL</td>
<td>9.21 cycles/CL</td>
</tr>
<tr>
<td>AVX-512 plain</td>
<td>18101 MFlops/s</td>
<td>12894 MFlops/s</td>
<td>2976 MFlops/d</td>
</tr>
<tr>
<td>SMT2</td>
<td>1.14 cycles/CL</td>
<td>1.61 cycles/CL</td>
<td>6.98 cycles/CL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMCI plain</td>
<td>11863 MFlops/s</td>
<td>1411 MFlops/s</td>
<td>740 MFlops/s</td>
</tr>
<tr>
<td>SMT2</td>
<td>1.41 cycles/CL</td>
<td>11.85 cycles/CL</td>
<td>22.64 cycles/CL</td>
</tr>
<tr>
<td>IMCI plain</td>
<td>10052 MFlops/s</td>
<td>2730 MFlops/s</td>
<td>904 MFlops/d</td>
</tr>
<tr>
<td>SMT4</td>
<td>1.66 cycles/CL</td>
<td>6.14 cycles/CL</td>
<td>18.52 cycles/CL</td>
</tr>
</tbody>
</table>

Pushing the limits: L1 performance

A common technique to hide instruction latencies and loop overhead is deeper unrolling.

<table>
<thead>
<tr>
<th></th>
<th>KNC SMT2</th>
<th>KNL SMT1</th>
<th>KNL SMT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-way unrolled</td>
<td>11863 MFlops/s</td>
<td>12942 MFlops/s</td>
<td>18101 MFlops/s</td>
</tr>
<tr>
<td></td>
<td>1.41 cycles/CL</td>
<td>1.60 cycles/CL</td>
<td>1.14 cycles/CL</td>
</tr>
<tr>
<td>8-way unrolled</td>
<td>1.28 cycles/CL</td>
<td>24188 MFlops/s</td>
<td>22981 MFlops/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.86 cycles/CL</td>
<td>0.91 cycles/CL</td>
</tr>
<tr>
<td>16-way unrolled</td>
<td>1.21 cycles/CL</td>
<td>29076 MFlops/s</td>
<td>27609 MFlops/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.71 cycles/CL</td>
<td>0.75 cycles/CL</td>
</tr>
</tbody>
</table>

Peak is 1.3 Ghz * 2 instr/cycle * 16 Flops/instr = 41.6 Gflops/s (70%)
Pushing the limits: L2 performance

1:
vprefetch0 [rsi + rax * 4 + 256]
vpaddps zmm0, zmm0, [rsi + rax * 4]
add rax, 16
cmp rax, rdi
jl 1b

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-way unrolled</td>
<td>1.49 cycles/CL</td>
<td>6.03 cycles/CL</td>
<td>18.56 cycles/CL</td>
</tr>
<tr>
<td>SMT4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 prefetching</td>
<td>3.20 cycles/CL</td>
<td><strong>3.13 cycles/CL</strong></td>
<td>38.82 cycles/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 prefetching</td>
<td>3.37 cycles/CL</td>
<td>3.85 cycles/CL</td>
<td>38.93 cycles/CL</td>
</tr>
<tr>
<td>SMT4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNL 16-way unrolled</td>
<td>0.71 cycles/CL</td>
<td>1.53 cycles/CL</td>
<td>10.29 cycles/CL</td>
</tr>
<tr>
<td>SMT1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The software prefetching interferes with the hardware prefetcher.

Shared L2 cache scalability

The L2 cache is shared by two cores.

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 cores Shared L2</th>
<th>2 cores Private L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNL 16-way unrolled</td>
<td>53870 MFlops/s</td>
<td><strong>77598 MFlops/s</strong></td>
<td>107644 MFlops/s</td>
</tr>
<tr>
<td>SMT1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pushing the limits: Memory performance

1:
vaddps zmm0, zmm0, [rsi + rax * 4]  
vprefetch1 [rsi + rax * 4 + 4096]  
vaddps zmm0, zmm0, [rsi + rax * 4 + 64]  
vprefetch0 [rsi + rax * 4 + 1024]  
vaddps zmm2, zmm2, [rsi + rax * 4 + 128]  
vprefetch1 [rsi + rax * 4 + 4160]  
vaddps zmm3, zmm3, [rsi + rax * 4 + 192]  
vprefetch0 [rsi + rax * 4 + 1088]  
vaddps zmm4, zmm4, [rsi + rax * 4 + 256]  
vprefetch1 [rsi + rax * 4 + 4224]  
vaddps zmm5, zmm5, [rsi + rax * 4 + 320]  
vprefetch0 [rsi + rax * 4 + 1152]  
vaddps zmm6, zmm6, [rsi + rax * 4 + 384]  
vprefetch1 [rsi + rax * 4 + 4288]  
vaddps zmm7, zmm7, [rsi + rax * 4 + 448]  
vprefetch0 [rsi + rax * 4 + 1216]  
vprefetch1 [rsi + rax * 4 + 4352]  
vprefetch0 [rsi + rax * 4 + 1280]  
vprefetch1 [rsi + rax * 4 + 4416]  
vprefetch0 [rsi + rax * 4 + 1344]  
vprefetch1 [rsi + rax * 4 + 4480]  
vprefetch0 [rsi + rax * 4 + 1408]  
vprefetch1 [rsi + rax * 4 + 4544]  
vprefetch0 [rsi + rax * 4 + 1472]  
add rax, 128  
cmp rax, rdi  
jl lb

float sum=0.;  
int i;

#pragma vector aligned  
for(i = 0; i < length; i++)  
{
    sum += A[i];
}

return sum;

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-way unrolled</td>
<td>1.49 cy/CL</td>
<td>6.03 cy/CL</td>
<td>18.56 cy/CL</td>
</tr>
<tr>
<td>SMT4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 prefetching</td>
<td>3.20 cy/CL</td>
<td>3.13 cy/CL</td>
<td>38.82 cy/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory prefetching</td>
<td>3.05 cy/CL</td>
<td>4.98 cy/CL</td>
<td>14.17 cy/CL</td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNL 16-way unrolled</td>
<td>0.71 cy/CL</td>
<td>1.53 cy/CL</td>
<td>10.92 (11.76) cy/CL</td>
</tr>
<tr>
<td>SMT1 (MCDRAM)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add rax, 128</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summation code (full device)

<table>
<thead>
<tr>
<th></th>
<th>KNC</th>
<th>Single core</th>
<th>Full device</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 prefetching</td>
<td>1727 MB/s</td>
<td>90219 MB/s</td>
<td></td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM prefetching</td>
<td>4687 MB/s</td>
<td>170754 MB/s</td>
<td></td>
</tr>
<tr>
<td>SMT1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM prefetching</td>
<td>4731 MB/s</td>
<td>175158 MB/s</td>
<td></td>
</tr>
<tr>
<td>SMT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM prefetching</td>
<td>4740 MB/s</td>
<td>176347 MB/s (62%)</td>
<td></td>
</tr>
<tr>
<td>SMT4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>KNL</th>
<th>Single core</th>
<th>Full device</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMT1 DDR</td>
<td>8078 MB/s</td>
<td>78413 MB/s (76% Peak)</td>
<td></td>
</tr>
<tr>
<td>SMT1 MCDRAM</td>
<td>7072 MB/s</td>
<td>345198 MB/s</td>
<td></td>
</tr>
<tr>
<td>SMT2 MCDRAM</td>
<td>9443 MB/s</td>
<td>339352 MB/s</td>
<td></td>
</tr>
<tr>
<td>SMT4 MCDRAM</td>
<td>12363 MB/s</td>
<td>334483 MB/s</td>
<td></td>
</tr>
</tbody>
</table>

MCDRAM
- LLC on Xeon Phi is not shared
- Different MCDRAM modes are available: cache, flat, hybrid
How to leverage SIMD

Alternatives:
- The **compiler** does it for you (but: aliasing, alignment, language)
- Compiler directives (**pragmas**)
- Alternative **programming models** for compute kernels (OpenCL, cilk+, OpenMP 4, Intel ispc)
- C++ Vector classes
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembler**

To use **intrinsics** the following headers are available:
- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX, AVX-512)
- `x86intrin.h` (all instruction set extensions)
- See next slide for an example

---

**Example: array summation using C intrinsics**

(SSE, single precision)

```c
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 = _mm_setzero_ps();
sum1 = _mm_setzero_ps();
sum2 = _mm_setzero_ps();
sum3 = _mm_setzero_ps();
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
sum0 = _mm_add_ps(sum0, sum1);
sum0 = _mm_add_ps(sum0, sum2);
sum0 = _mm_add_ps(sum0, sum3);
_mm_store_ss(&scalar_sum, sum0);
```

summation of partial results

core loop (bulk)
Example: array summation from intrinsics, instruction code

```assembly
14:   0f 57 c9                 xorps %xmm1,%xmm1
17:   31 c0                   xor %eax,%eax
19:   0f 28 d1                movaps %xmm1,%xmm2
1c:   0f 28 c1                movaps %xmm1,%xmm0
1f:   0f 28 d9                movaps %xmm1,%xmm3
22:   66 0f 1f 44 00 00       nopw 0x0(%rax,%rax,1)
28:   0f 10 3e                movups (%rsi),%xmm7
2b:   0f 10 76 10             movups 0x10(%rsi),%xmm6
2f:   0f 10 6e 20             movups 0x20(%rsi),%xmm5
33:   0f 10 66 30             movups 0x30(%rsi),%xmm4
37:   83 c0 10                add    $0x10,%eax
3a:   48 83 c6 40             add    $0x40,%rsi
3e:   0f 58 df addps %xmm7,%xmm3
41:   0f 58 c6                addps %xmm6,%xmm0
44:   0f 58 d5                addps %xmm5,%xmm2
47:   0f 58 cc                addps %xmm4,%xmm1
4a:   39 c7                   cmp %eax,%edi
4c:   77 da                   ja 28 <compute_sum_SSE+0x18>
51:   0f 58 c3                addps %xmm3,%xmm0
54:   0f 58 c2                addps %xmm2,%xmm0
57:   f2 0f 7c c0             haddps %xmm0,%xmm0
5b:   c3                      retq
```

Example: array summation using C intrinsics (IMCI, single precision)

```c
float scalar_sum;
__m512 t0, t1, t2, t3;
__m512 sum0 = _mm512_setzero_ps();
__m512 sum1 = _mm512_setzero_ps();
__m512 sum2 = _mm512_setzero_ps();
__m512 sum3 = _mm512_setzero_ps();

for(i = 0; i < length; i+=64)
{
    t0 = _mm512_load_ps(data+i);
    t1 = _mm512_load_ps(data+i+16);
    t2 = _mm512_load_ps(data+i+32);
    t3 = _mm512_load_ps(data+i+48);
    sum0 = _mm512_add_ps(sum0, t0);
    sum1 = _mm512_add_ps(sum1, t0);
    sum2 = _mm512_add_ps(sum2, t2);
    sum3 = _mm512_add_ps(sum3, t3);
}
```
Example: array summation from IMCI intrinsics, instruction code

```plaintext
..B2.3:
vaddps (%rdi,%rdx,4), %zmm3, %zmm3
vprefetch1 1024(%rdi,%rdx,4)
vaddps 64(%rdi,%rdx,4), %zmm2, %zmm2
vprefetch0 512(%rdi,%rdx,4)
vaddps 128(%rdi,%rdx,4), %zmm1, %zmm1
incl %ecx
vaddps 192(%rdi,%rdx,4), %zmm0, %zmm0
addq $64, %rdx
cmpl %eax, %ecx
jb ..B2.3
```

Loop body

```plaintext
..B2.5:
vaddps %zmm2, %zmm3, %zmm2
vaddps %zmm1, %zmm2, %zmm1
vaddps %zmm0, %zmm1, %zmm3
nop
vpermf32x4 $238, %zmm3, %zmm4
vaddps %zmm4, %zmm3, %zmm5
nop
vpermf32x4 $85, %zmm5, %zmm6
vaddps %zmm6, %zmm5, %zmm7
nop
vaddps %zmm7{badc}, %zmm7, %zmm8
nop
vaddps %zmm8{cdab}, %zmm8, %zmm9
nop
vpackstorelps %zmm9, -8(%rsp)
```

Vectorization and the Intel compiler

- Intel compiler will try to use SIMD instructions when enabled to do so
  - “Poor man's vector computing”
  - Compiler can emit messages about vectorized loops (not by default):

  ```plaintext
  plain.c(11): (col. 9) remark: LOOP WAS VECTORIZED.
  ```

- Use option `-vec_report3` to get full compiler output about which loops were vectorized and which were not and why (data dependencies!)
- Some obstructions will prevent the compiler from applying vectorization even if it is possible

- You can use source code directives to provide more information to the compiler
Rules for vectorizable loops

1. Countable
2. Single entry and single exit
3. Straight line code
4. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple inner loops with unit stride
2. Minimize indirect addressing
3. Align data structures (SSE 16 bytes, AVX 32 bytes)
4. In C use the restrict keyword for pointers to rule out aliasing

Obstacles for vectorization:
- Non-contiguous memory access
- Data dependencies

x86 Architecture:
SIMD and Alignment

- Alignment issues
  - Alignment of arrays with AVX (IMCI) should be on 32-byte (64-byte) boundaries to allow packed aligned loads and NT stores (for Intel processors)
  - Modern x86 CPUs have less (not zero) impact for misaligned LD/ST, but Xeon Phi relies heavily on it!
- How is manual alignment accomplished?
- Dynamic allocation of aligned memory (align = alignment boundary):

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr,
                    size_t align,
                    size_t size);
```
**Interlude: Software Prefetching on Xeon Phi**

- Compiler will issue a massive amount of prefetch instructions starting with –O2
- This includes all intrinsic load and stores

- This is a reasonable compromise to deal with the shortcomings of the overall architecture

- To turn off software prefetching by the compiler:
  - Global option `-no-opt-prefetch`
  - Loop local pragma `#pragma noprefetch`

- To be sure always check the assembly code, especially with Intrinsic code.

---

**Microbenchmarking for Architectural Exploration**

- Probing of the memory hierarchy
- Saturation effects in cache and memory
- Typical OpenMP overheads

---

266
LLC performance on Xeon Phi (1 core)

LLC performance on SandyBridge-EP (1 core)
LLC bandwidth scaling Xeon Phi

LLC bandwidth scaling SandyBridge-EP
Memory bandwidth saturation on Xeon Phi

Memory bandwidth saturation on SandyBridge-EP
### Thread synchronization overhead on IvyBridge-EP

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>599</td>
<td>425</td>
</tr>
<tr>
<td>SMT threads</td>
<td>612</td>
<td>423</td>
</tr>
<tr>
<td>Other socket</td>
<td>1486</td>
<td>1067</td>
</tr>
</tbody>
</table>

- Strong topology dependence!

### Full domain

<table>
<thead>
<tr>
<th>Socket (10 cores)</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node (20 cores)</td>
<td>1934</td>
<td>1301</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>4999</td>
<td>7783</td>
</tr>
</tbody>
</table>

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

- Overhead grows with thread count

---

### Thread synchronization overhead on Intel Xeon Phi KNC (60-core)

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th>2 threads on distinct cores: 1936</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMT1</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>One core</td>
</tr>
<tr>
<td>Full chip</td>
</tr>
</tbody>
</table>

That does not look bad for 240 threads!

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Sandy Bridge node

3.75 x cores (16 vs 60) on Phi
2 x more operations per cycle on Phi
2.7 x more barrier penalty (cycles) on Phi

2MB == 512 cy
30MB == 25000 cy

7.5 x more work done on Xeon Phi per cycle

One barrier causes 2.7 x 7.5 = 20x more pain 😞.
Thread synchronization overhead on Xeon Phi KNL 7210 (64-core)
Barrier overhead in CPU cycles (Intel C compiler 16.03)

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>One core</td>
<td>n/a</td>
<td>963</td>
<td>1580</td>
<td>2240</td>
</tr>
<tr>
<td>Full chip</td>
<td>5720</td>
<td>8100</td>
<td>9900</td>
<td>11400</td>
</tr>
</tbody>
</table>

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Ivy Bridge node

3.2x cores (20 vs 64) on Phi
4x more operations per cycle per core on Phi

\[ 4 \cdot 3.2 = 12.8 \times \text{more work done on Xeon Phi per cycle} \]

1.9x more barrier penalty (cycles) on Phi (11400 vs. 6000)

\[ \Rightarrow \text{One barrier causes } 1.9 \cdot 12.8 \approx 24 \times \text{more pain 😞}. \]

Configuration complexity

- **Cluster modes:** lower the latency and increase the bandwidth
  - All-to-all
  - Quadrant mode (default)
  - Sub-numa-clustering (SNC), best performance but explicit

- **Memory modes:**
  - Cache mode (default)
  - Flat mode (explicit)
  - Hybrid

- **Mapping** of application on hardware:
  - Use SMT or not. How many SMT threads?
  - Use all cores?
  - MPI+X. How exactly?

- **Memory configuration:** Alignment and page size choices
Specific issues with Xeon Phi

- **MCDRAM** adds additional complexity
- Configuration of system and mapping of application on hardware gets more critical
- The compromise and strategy made with KNL will soon be outdated
- KNL as a hosted cluster system is probably too specialized for a general purpose academic cluster

But

- Xeon Phi implements features which are not available anywhere else:
  - High degree of parallelism
  - Multiple memory types and explicit memory control
  - Mesh type on-die topology

- It allowed a glimpse in the future on real hardware
Documentation


- Intrinsic guide as interactive webpage

Part 2

Plenary Session

June 28, 2017

Leibniz Supercomputing Centre
Garching b. München, Germany

Joint session with the Scientific Workshop "HPC for natural hazard assessment and disaster mitigation".
Plenary Session – Biographies
Wednesday, June 28, 2017, 13:00-18:00, Hörsaal, H.E.009 (Lecture Hall)

Session 1 (Chairman: Volker Weinberg)

13:00-13:30 Luigi Iapichino, IPCC@LRZ: "Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures"

Luigi Iapichino is a scientific computing expert at LRZ and is member of the Intel Parallel Computing Center (IPCC). His main tasks are code modernisation for many-core systems, and HPC support. He got a PhD in physics from TU München, working at the Max Planck Institute for Astrophysics. Before moving to LRZ in 2014, he worked at the Universities of Würzburg and Heidelberg, involved in research projects related to computational astrophysics.

13:30-14:00 Michael Bader, IPCC@TUM: "Extreme-scale Multi-physics Simulation of the 2004 Sumatra Earthquake"

Prof. Bader studied computer science and earned his PhD in 2001 at TUM. He subsequently acted as a coordinator of the elite master’s program in computational engineering (as part of the Elite Network Bavaria) and of the Munich Center of Advanced Computing. From 2009 to 2011, before assuming the position of professor at TUM, he worked as an assistant professor at the SimTech Cluster of Excellence at the University of Stuttgart. Michael works on hardware-aware algorithms in computational science and engineering and in high performance computing.

14:00-14:30 Vít Vondrák, IPCC@IT4I: "Development of Intel Xeon Phi Accelerated Algorithms and Applications at IT4I"

Vít Vondrák is Scientific Director at the IT4Innovations National Supercomputing Centre and Associate Professor at the Technical University of Ostrava. His expertise is in numerical linear algebra, optimisation methods, and high performance computing. He is a member of the PRACE Council representing the Czech Republic in this pan-European HPC infrastructure. He is also the principal investigator of the IPCC project funded by Intel and co-investigator of the TEP project by European Space Agency.

14:30-15:00 Michael Klemm, Intel: "Application Show Cases on Intel® Xeon Phi™ Processors"

Michael Klemm holds a PhD from the Friedrich-Alexander-University Erlangen-Nuremberg. He works in the Developer Relations Division at Intel in Germany. His areas of interest include compiler construction, design of programming languages, parallel programming, performance analysis and tuning. Michael Klemm joined the OpenMP organization in 2009 and was appointed CEO of the OpenMP ARB in 2016.

15:00-15:30 Coffee Break
15:30-16:00 Jan Eitzinger, RRZE: "Evaluation of Intel Xeon Phi "Knights Landing": Initial impressions and benchmarking results"

Jan Eitzinger holds a PhD in Computer Science from the University of Erlangen. He is now a postdoctoral researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE). His current research involves architecture-specific and low-level optimization for current processor architectures, performance modeling on processor and system levels, and programming tools. He is the developer of LIKWID, a collection of lightweight performance tools.

16:00-16:30 Piotr Korcyl, University of Regensburg: "Lattice Quantum Chromodynamics on the MIC architectures"

Piotr did his PhD in Theoretical Physics at the Jagiellonian University about 'Dynamics of Supersymmetric Quantum Mechanics with Gauge Symmetry'. He later worked at Jagiellonian University Krakow, DESY Zeuthen, the Columbia University New York and the University of Regensburg. He is currently working as Associate Professor at the Jagiellonian University, Krakow. Currently he is on scientific leave till October 2017. Top #1 SuperMIC user at LRZ running production QCD code on the whole machine.

16:30-17:00 Nils Moschüring, IPP: "The experience of the HLST on Europes biggest KNL cluster"

Nils did his PhD in theoretical physics at the chair for computational and plasma physics of Prof. Ruhl at the LMU. He started working at the Max Planck Institute for Plasmafysik in February this year as part of the High Level Support Team. The team supports fusion scientists in performing optimized simulations on supercomputers. He could gain lots of experience on the KNL partition of the MARCONI supercomputer in Bologna.

17:00-17:30 Andreas Marek, Max Planck Computing and Data Facility (MPCDF), "Porting the ELPA library to the KNL architecture"

Andreas is a senior HPC and applications specialist at the Max Planck Computing and Data Facility (MPCDF). He did his PhD at the Max Planck Institute for Astrophysics in 2007. From 2003 - 2016 he was the lead developer of the VERTEX code. With this code he also got the winner of the Leibniz Extreme scaling award at LRZ in 2016. Since 2010 he is involved in the ELPA project as main developer and project leader.

17:30-18:00 Q&A, Wrap-up
Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures

Dr. Luigi Iapichino

Dr. Fabio Baruffa

Leibniz Supercomputing Centre

Intel MIC Programming Workshop & Scientific Workshop "HPC for natural hazard assessment and disaster mitigation", LRZ, June 28th, 2017

Work contributors

Dr. Fabio Baruffa
Sr. HPC Application Specialist
Leibniz Supercomputing Centre

- Member of the Intel Parallel Computing Center (IPCC) @ LRZ/TUM
- Expert in performance optimization and HPC systems

Dr. Luigi Iapichino
Scientific Computing Expert
Leibniz Supercomputing Centre

- Member of the Intel Parallel Computing Center (IPCC) @ LRZ/TUM
- Expert in computational astrophysics and simulations
Outline of the talk

• Overview of the code: P-Gadget3 and SPH.
• Challenges in code modernization approach.
• Multi-threading parallelism and scalability.
• Enabling vectorization through:
  Data layout optimization (AoS → SoA).
  Reducing conditional branching.
• Performance results, takeaways from our KNL experience.

Gadget intro

• Leading application for simulating the formation of the cosmological large-scale structure (galaxies and clusters) and of processes at sub-resolution scale (e.g. star formation, metal enrichment).
• Publicly available, cosmological TreePM N-body + SPH code.
• First developed in the late 90s as serial code, later evolved as an MPI and a hybrid code.
• Good scaling performance up to O(100k) Xeon cores (SuperMUC@LRZ).
### Smoothed particle hydrodynamics (SPH)

- **SPH** is a Lagrangian particle method for solving the equations of fluid dynamics, widely used in astrophysics.

- It is a mesh-free method, based on a particle discretization of the medium.

- The local estimation of gas density (and all other derivation of the governing equations) is based on a *kernel-weighted summation* over neighbor particles:

\[
\rho_i = \rho(r_i) = \sum_j m_j W(|r_i - r_j|, h_j)
\]

### Optimization strategy

- We isolate the representative code kernel `subfind_density` and run it in as a stand-alone application, avoiding the overhead from the whole simulation.

- As most code components, it consists of two sub-phases of nearly equal execution time (40 to 45% for each of them), namely the neighbour-finding phase and the remaining physics computations.

- Our physics workload: \(\sim 500k\) particles. This is a typical workload per node of simulations with moderate resolution.

- We focus on node-level performance, through minimally invasive changes.

- We use tools from the Intel® Parallel Studio XE (VTune Amplifier and Advisor).
Target architectures for our project

Intel® Xeon processor

- E5-2650v2 Ivy-Bridge (IVB) @ 2.6 GHz, 8-cores / socket.
  TDP: 95W, RCP (03/2017): $1116.
- AVX.

Intel® Xeon Phi™ coprocessor 1st generation

- Knights Corner (KNC) coprocessor 5110P @ 1.1GHz, 60 cores.
  TDP: 225W, RCP: N/D.
- Native / offload computing.
- Directly login via ssh.
- SIMD 512 bits.

Further tested architectures

Intel® Xeon processors

- E5-2697v3 Haswell (HSW) @ 2.3 GHz, 14-cores / socket.
- AVX2, FMA.
- E5-2699v4 Broadwell (BDW) @ 2.2 GHz, 22-cores / socket.
- AVX2, FMA.

Intel® Xeon Phi™ processor 2nd generation

- Knights Landing (KNL) Processor 7250 @ 1.4 GHz, 68 cores.
- Available as bootable processor.
- Binary-compatible with x86.
- High bandwidth memory.
- New AVX512 instructions set.
Initial profiling

- Severe shared-memory parallelization overhead
- At later iterations, the particle list is locked and unlocked constantly due to the recomputation
- Spinning time 41%

Multi-threading parallelism

Improved performance

- Lockless scheme: lock contention removed through “todo” particle list and OpenMP dynamic scheduling.
- Time spent in spinning only 3%

Multi-threading parallelism
Improved speed-up

- On IVB @ 8 threads
  - speed-up: 1.8x
  - parallel efficiency: 92%

- On KNC @ 60 threads
  - speed-up: 5.2x
  - parallel efficiency: 57%

Multi-threading parallelism

Obstacles to efficient auto-vectorization

\[
\text{for}(n = 0, n < \text{neighboring} \_\text{particles}, n++ \{} \\
\quad j = \text{ngblist}[n]; \\
\quad \text{if} \ (\text{particle} \ n \ \text{within} \ \text{smoothing} \_\text{length})\{} \\
\quad \quad \text{inlined} \_\text{function1}(\ldots, \&w); \\
\quad \quad \text{inlined} \_\text{function2}(\ldots, \&w); \\
\quad \quad \rho \ += \text{P. AoS}[j].\text{mass} \times \text{w}; \\
\quad \quad \text{vel}_x \ += \text{P. AoS}[j].\text{vel}_x; \\
\quad \quad \ldots \\
\quad \quad v2 \ += \text{vel}_x \times \text{vel}_x + \ldots \text{vel}_z \times \text{vel}_z; \\
\quad \}\}
\]

for loop over neighbors
check for computation
computing physics
Particles properties via AoS (cache unfriendly!)

\[
\rho_i = \rho(r_i) = \sum_j m_j W(|r_i - r_j|, \eta_j)
\]
AoS to SoA: performance outcomes

- Gather-scatter overhead at most 1.8% of execution time. → intensive data-reuse

- Performance improvement:
  - on IVB: 13%, on KNC: 48%

- Xeon/Xeon Phi performance ratio: from 0.15 to 0.45.

- The data structure is now vectorization-ready.

Vectorization: improvements from IVB to KNL

- Vectorization through localized masking (if-statement moved inside the inlined functions).

- Vector efficiency:
  perf. gain / vector length

  on IVB: 55%
  on KNC: 42%
  on KNL: 83%
**Node-level performance comparison between HSW, KNC and KNL**

Features of the KNL tests:
- KMP Affinity: scatter;
- Memory mode: Flat;
- MCDRAM via numacl;
- Cluster mode: Quadrant.

Results:
- Our optimization improves the speed-up on all systems.
- Better threading scalability up to 136 threads on KNL.
- Hyperthreading performance is different between KNC and KNL.

---

**Performance comparison: first results including KNL and Broadwell**

- Initial vs. optimized including all optimizations for *subfind_density*

- IVB, HSW, BDW: 1 socket w/o hyperthreading.
- KNC: 1 MIC, 240 threads.
- KNL: 1 node, 136 threads.

- Performance gain:
  - Xeon Phi: **13.7x** KNC, **19.1x** KNL.
  - Xeon: **2.6x** IVB, **4.8x** HSW, **4.7x** BDW.

---

Performance results on Knights Landing

Performance results
Code optimization on KNL: lessons learnt (so far...)

Optimization for KNL as a three-step process:

<table>
<thead>
<tr>
<th>Step</th>
<th>Effort</th>
<th>Expected performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation &quot;out of the box&quot;</td>
<td>1 hour</td>
<td>Lower than Haswell (~1.5x)</td>
</tr>
<tr>
<td>Optimization without coding (use of AVX512,</td>
<td>1 week</td>
<td>Up to 2x over previous</td>
</tr>
<tr>
<td>explore configuration, MCDRAM, MPI/OpenMP)</td>
<td></td>
<td>step</td>
</tr>
<tr>
<td>Optimization with coding (this project and</td>
<td>1-3 months</td>
<td>Up to the level of</td>
</tr>
<tr>
<td>beyond)</td>
<td>(IPCC: 2</td>
<td>Broadwell</td>
</tr>
<tr>
<td>years)</td>
<td>years)</td>
<td></td>
</tr>
</tbody>
</table>

Freely adapted from Leijun Hu, Inspur @ ISC 2017

Some more KNL wisdom

- Quad-cache is a good starting point, quad-flat with allocation on MCDRAM is worth being tested, SNC modes are for very advanced developers.
- It is unlikely to gain performance with more than 2 threads/core.
- Vectorize whenever possible, use compiler reports and tools to exploit low-hanging fruits.
- Know where your data are located and how they move.
- If optimizations are portable, the effort pays off!
Summary and outlook

- Code modernization as the iterative process for improving the performance of an HPC application.
- Our IPCC example: P-Gadget3. Threading parallelism Data layout Vectorization
  {Key points of our work, guided by analysis tools.}
- This effort is (mostly) portable! Good performance found on new architectures (KNL and BDW) basically out-of-the-box.
- For KNL, architecture-specific features (MCDRAM, large vector registers and NUMA characteristics) are currently under investigation for different workloads.
- Investment on the future of well-established community applications, and crucial for the effective use of forthcoming HPC facilities.

This work: https://arxiv.org/abs/1612.06090 (IEEE Xplore, accepted)

Acknowledgements

- Research supported by the Intel® Parallel Computing Center program.
- Project coauthors: Nicolay J. Hammer (LRZ), Vasileios Karakasis (CSCS).
- P-Gadget3 developers: Klaus Dolag, Margarita Petkova, Antonio Ragagnin.
- Research collaborator at Technical University of Munich (TUM): Nikola Tchipev.
- TCEs at Intel: Georg Zitzlsberger, Heinrich Bockhorst.
- Thanks to the IXPUG community for useful discussion.
- Special thanks to Colfax Research for granting access to their computing facilities.
todo_partlist = partlist;

while(partlist.length){
    error=0;
    #pragma omp parallel for schedule(dynamic)
    for(auto p:todo_partlist){
        if(something_is_wrogl) error=1;
        ngblist = find_neighbours(p);
        sort(ngblist);
        for(auto n:select(ngblist,K))
            compute_interaction(p,n);
    }
    //...check for any error
    todo_particles = mark_for_recomputation(partlist);
}

// Data layout

struct ParticleAoS
{
    float pos[3], vel[3], mass;
}
Particle_AoS *P_AoS;
P_AoS = malloc(N*sizeof(Particle_AoS));

void gather_Pdata(struct Particle_SoA *dst, struct Particle_AoS *src, int N )
for(int i = 0, i < N, i++ ){
    dst -> pos_x[i] = src[i].pos[1]; dst -> pos_y[i] = src[i].pos[2]; ...
}

rho   += P_AoS[j].mass*w;
vel_x += P_AoS[j].vel_x;

struct ParticleSoA
{
    float *pos_x, ..., *vel_x, ..., mass;
}
Particle_SoA P_SoA;
P_SoA.pos_x = malloc(N*sizeof(float));
...
DEVELOPMENT OF INTEL XEON PHI ACCELERATED ALGORITHMS AND APPLICATIONS AT IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTRE

Vít Vondrák, Lubomír Říha, Michal Merta, Jan Zapletal, Milan Jaroš, …

Introduction

IT4Innovations National Supercomputing Centre – www.it4i.cz

- Salomon Supercomputer - 1008 computational nodes of which 576 are regular compute nodes and 432 accelerated nodes (two Intel Xeon Phi 7120P co-processors per node)

Selected Applications from Parallel Algorithms Research Lab at IT4Innovations

- IPC Center at IT4Innovations – acceleration of the ESPRESO library and its integration into ELMER and OpenFOAM community codes (http://ipcc.it4i.cz, http://espreso.it4i.cz)
- BEM4I – development of efficient Boundary Element Method (BEM) assembler for multi- and many-core architectures with wide SIMD units (http://bem4i.it4i.cz)
- Blender – parallelization of the Blender Cycles rendering engine by MPI and OpenMP with support for Intel Xeon Phi accelerators (http://ipcc.it4i.cz/files/Download/)
ESPRESO Massively Parallel Sparse Linear Solver

- FETI domain decomposition based solver supporting massively parallel solution of structural mechanics problems
- Supports acceleration by Intel Xeon Phi processors – KNC and KNL
- Provides general API for other open source libraries (ELMER, OpenFOAM)

### ORNL Titan Full Scale Weak Scalability Test Solving up to 124 billion DOF on 17576 Compute Nodes

<table>
<thead>
<tr>
<th>Problem size [billion DOF]</th>
<th>Number of compute nodes [-]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>116</td>
<td>512</td>
</tr>
<tr>
<td>1000</td>
<td>1728</td>
</tr>
<tr>
<td>2094</td>
<td>4096</td>
</tr>
<tr>
<td>4688</td>
<td>5832</td>
</tr>
<tr>
<td>8000</td>
<td>10648</td>
</tr>
<tr>
<td>13824</td>
<td>17576</td>
</tr>
</tbody>
</table>

- Designed to solve real world problems generated by open-source (ELMER and OpenFOAM) and commercial tools (Ansys Workbench)

- KNL - Intel(R) Xeon Phi (TM) processors (previously called Knights Landing)
- KNC - Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)

ESPRESO Advanced Features

- Within the IPCC project at IT4Innovations the library was accelerated by offloading to Intel Xeon Phi (KNC) co-processors
- Algorithmic changes have been made to replace actions with sparse FEM data structures with dense objects to fully utilize the coprocessor
- Acceleration of two key routines: (1) system matrix processing and (2) preconditioner application
- Supports offload to multiple coprocessors and dynamic CPU/co-processor load balancing
- Scalability tested on the Salomon supercomputer with up to 864 Xeon Phi 7120P co-processors
- Improved scalability of community codes (ELMER)

- KNL - Intel(R) Xeon Phi (TM) processors (previously called Knights Landing)
- KNC - Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)
Offloading Approaches in ESPRESO

Matrixpack offload
- One thread on host offloads all matrices within one array (MatrixPack)
- One parallel OpenMP region on device

Matrix semi-pack offload
- Each host thread processes one matrix pack
- Multiple OpenMP regions on device

Individual offload
- Each thread on host offloads its matrices individually
- No OpenMP region on device

The Future of ESPRESO – Porting to KNL

- Tests were performed on Intel Xeon Phi 7210 processors at Intel Endeavor cluster
- Slower preprocessing phase, faster solve – suitable for problems requiring large number of iterations

Note: Comparison of iterative solver runtime on different Xeon architectures using sparse/dense matrix structures
BEM4I – Xeon Phi Accelerated Library

Boundary element method
• alternative to FEM
• discretization of boundary only
• singular surface integrals
• dense matrices (classic BEM)

BEM4I (bem4i.it4i.cz)
• solver developed at IT4Innovations
• optimized for Intel Xeon and Xeon Phi processors

Performance (threading)
• 2 x Xeon E5-2680v3 (Haswell) processor with 24 cores
  • speedup 24 w.r.t. single threads
• Xeon Phi 7120P (Knights Corner) co-processor with 61 cores
  • speedup 61+
• Xeon Phi 7210, (Knights Landing) processor with 64 cores
  • speedup 64+

Performance (SIMD)
• speedup 8+ with AVX-512 w.r.t. scalar code


KNC – Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)
Massively parallel version: BETI = BEM4I + ESPRESO

- Boundary Element Tearing and Interconnecting (alternative to FETI originally implemented in ESPRESO)
- massively parallel – SIMD + OpenMP + MPI + offload to Intel Xeon Phi (KNC) co-processor

KNC - Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)

Acceleration of Blender Cycles Rendering Engine
CyclesPhi – Accelerated Blender Cycles

- Blender is an open source 3D creation suite. It has two render engines: Blender Internal and Cycles.
- Cycles is a raytracing based render engine with support for interactive rendering, shading node system, and texture workflow.
- We have modified the kernel of the Blender Cycles rendering engine and then extended its capabilities to support the HPC environment. We call this version the CyclesPhi. It supports following technologies:
  - OpenMP
  - MPI
  - Intel Xeon Phi (KNC) processor with Offload
  - Intel Xeon Phi (KNC) processor with Symmetric mode
  - And their combinations

KNC - Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)

MPI Parallelization of Blender Cycles

Supports both interactive and offline rendering – with almost linear strong scalability

Tests on Salomon supercomputer: 2x E5-2680v3 (HSW) and 2x7120p (KNC) processors per node

KNC - Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)
Acceleration of Blender Cycles by Xeon Phi Processor

- 1x Xeon Phi 7250 (KNL)
- 2x HSW + 2x Xeon Phi 7120p (KNC)
- 2x 12core E5-2680v3 (HSW)
- 1x Xeon Phi 7120p (KNC)

<table>
<thead>
<tr>
<th>Scene (see figures)</th>
<th>Classroom</th>
<th>Fishy cat</th>
<th>Koro</th>
<th>Pabellon Barcelona</th>
<th>Victor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x Xeon Phi 7250 (KNL)</td>
<td>18 s</td>
<td>110 s</td>
<td>120 s</td>
<td>11 s</td>
<td>1680 s</td>
</tr>
<tr>
<td>1x Xeon Phi 7120p (KNC)**</td>
<td>3 s</td>
<td>12 s</td>
<td>14 s</td>
<td>4 s</td>
<td>120 s</td>
</tr>
<tr>
<td>2x HSW + 2x Xeon Phi 7120p (KNC)**</td>
<td>6 s</td>
<td>14 s</td>
<td>19 s</td>
<td>4 s</td>
<td>150 s</td>
</tr>
<tr>
<td>2x 12core E5-2680v3 (HSW)</td>
<td>1 s</td>
<td>8 s</td>
<td>10 s</td>
<td>1 s</td>
<td>90 s</td>
</tr>
</tbody>
</table>

** preprocessing is done by the 2x 12core E5-2680v3 (HSW) and includes data transfer over PCIe bus

- Salomon supercomputer: 2x Intel Xeon E5-2680v3 CPUs + 2x Intel Xeon Phi 7120P (KNL) processors
- Intel Xeon Phi 7250 - HLRN-III Cray System

KNL - Intel(R) Xeon Phi (TM) processors (previously called Knights Landing)
KNC - Intel(R) Xeon Phi (TM) co-processor (previously called Knights Corner)

Thank you
(TWO) APPLICATION SHOW CASES ON INTEL® XEON PHI™ PROCESSORS

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright© 2017, Intel Corporation. All rights reserved. Intel, the Intel logo, Atom, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Intel® Xeon Phi™ Processor Architecture

GTC-P
Tokamak plasma physics particle-in-cell (PIC) code

Work by:
Jason Sewall (Intel)
Princeton Gyrokinetic Toroidal Code

- Plasma turbulence simulation
  - Motion of ions through Tokamak
  - Vlasov-Poisson equation using particle-in-cell (PIC)
  - Well-studied in HPC
  - Many ‘leadership-class' runs and ports

Algorithm

**Charge**
- Particles deposit charge onto grid \(O(\text{Particles})\)

**Poisson**
- Solve Poisson equation over grid \(O(\text{Grid})\)

**Field**
- Reconstruct electric field over grid \(O(\text{Grid})\)

**Smooth**
- Filter grid fields \(O(\text{Grid})\)

**Push**
- Transfer field to particles \(O(\text{Particles})\)
- Move particles (in phase space) \(O(\text{Particles})\)

**Shift**
- Move particles between MPI ranks \(O(\text{Particles})\)

Particles >>> Grid (for this code)
Optimizations (BASELINE)

- B-1rank-half problem:
  - Run with 1 rank and 400 particles
  - KNL results from Xeon Phi 7250
  - BDW results from 2x Xeon E5-2698 v4

Optimizations to help vectorization

- Avoid excessive memoization
  - Gathers expensive, can be avoided sometimes

```c
im = ii;
im2 = ii + 1;

tdumtmp = pi2_inv * (tflr - zetatmp * qInv(in)) + 10.0;
tdumtmp2 = pi2_inv * (tflr - zetatmp * qInv(in2)) + 10.0;
tdum = (tdumtmp - (int)tdumtmp) * delt[im];
tdum2 = (tdumtmp2 - (int)tdumtmp2) * delt[im2];
j00 = abs_min_int(mtheta[im] - 1, (int)tdum);
j01 = abs_min_int(mtheta[im2] - 1, (int)tdum2);
jtion0tmp = igrid[im] + j00;
jtion1tmp = igrid[im2] + j01;
```

- Minimize type conversions

```c
const real im_r = ii_r;
const real im2_r = ii_r + 1.0;

const real mth_im_r = poloidal_mtheta(im_r, mtheta_a, mtheta_b, mthetamax_r);
const real mth_im2_r = poloidal_mtheta(im2_r, mtheta_a, mtheta_b, mthetamax_r);

const real pgrid_base = igrid[(int) im_r];
const real pgrid_next = pgrid_base + mth_im_r + 1.0;

const real qinv_m = poloidal_qtinv(im_r, q0, q1, q2, ainv, a0, deltar, mth_im_r);
const real qinv_m2 = poloidal_qtinv(im2_r, q0, q1, q2, ainv, a0, deltar, mth_im2_r);

const real tdumtmp = tflr - zetatmp_pi2 * qInv_m + 10.0;
const real tdumtmp2 = tflr - zetatmp_pi2 * qInv_m2 + 10.0;

const real tdum = fmod(tdumtmp, 1.0) * mth_im_r;
const real tdum2 = fmod(tdumtmp2, 1.0) * mth_im2_r;

const real j00 = abs_min_real(mth_im_r - 1.0, floor(tdum));
const real j01 = abs_min_real(mth_im2_r - 1.0, floor(tdum2));

const int jtion0tmp = (int) (pgrid_base + j00);
const int jtion1tmp = (int) (pgrid_base + j01);
```
Optimizations (PUSH)

- Large ‘diagnostic’ branch in code
  - Only active for certain iterations
  - Multiversion code so extra code not in ‘normal’ loop
- Strip-mining loop can help alignment
- Narrowing masks from whole-loop to just write-masking
- Marking reductions essential for correctness

```c
#pragma omp for nowait
for (int mo = 0; mo < mi; mo += 16) {
    real *__restrict__ z0mo  = particle_data->z0 + mo;
    real *__restrict__ z1mo  = particle_data->z1 + mo;
    real *__restrict__ z2mo  = particle_data->z2 + mo;
    ....
    #pragma omp simd aligned(z0mo, z1mo, z2mo, ... : 64) \ simdlen(16) \ 
        reduction(+ : particles_energy_a, ...) 
    for (int v = 0; v < 16; v++) {
        const real zion2m = z2mo[v];
        const int valid = v + mo < mi && !gtc_hole(zion2m);
        <lots of code>
        if (valid) {
            chargei_update(ij1, densityi_part, wz0 * wt00);
            chargei_update(ij1 + 1, densityi_part, wz1 * wt00);
            chargei_update(ij1 + mzeta + 1, densityi_part, wz0 * wt10);
            chargei_update(ij1 + mzeta + 2, densityi_part, wz1 * wt10);
            chargei_update(ij2, densityi_part, wz0 * wt01);
            chargei_update(ij2 + 1, densityi_part, wz1 * wt01);
            chargei_update(ij2 + mzeta + 1, densityi_part, wz0 * wt11);
            chargei_update(ij2 + mzeta + 2, densityi_part, wz1 * wt11);
        }
    }
}
```

Optimizations (Charge)

```c
#pragma omp for
for (m = 0; m < mi; m++) {
    zetatmp = z2[m];
    if (zetatmp == HOLEVAL) {
        continue;
    }
    <later>
    densityi_part[ij1] += d1;
    densityi_part[ij1 + 1] += d2;
    densityi_part[ij1 + mzeta + 1] += d3;
    densityi_part[ij1 + mzeta + 2] += d4;
    densityi_part[ij2] += d5;
    densityi_part[ij2 + 1] += d6;
    densityi_part[ij2 + mzeta + 1] += d7;
    densityi_part[ij2 + mzeta + 2] += d8;
}
```

- Strip-mining loop can help alignment
- Narrowing masks from whole-loop to just write-masking helpful
- Write-conflicts can be helped with ordered simd
  - Or vconflict + scatter

3.2x speedup for Push
1.6x speedup overall
Optimizations (Charge)

GTC-P: Charge Optimisations

- **Runtime (s)**
  - BDWx2 Baseline: 18.1
  - BDWx2 Push Opts: 18.12
  - BDWx2 Charge Opts: 17.49
  - KNL Baseline: 16.44
  - KNL Push Opts: 18.4
  - KNL Charge Opts: 7.7

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. * Other names and brands may be claimed as the property of others. All products, dates, and figures are preliminary and are subject to change without any notice. Copyright © 2017, Intel Corporation.

2.3x speedup for Charge
1.6x speedup overall

Optimizations (Sorting)

GTC-P: Sorting Optimisations

- **Runtime (s)**
  - BDWx2 Baseline: 18.1
  - BDWx2 Push Opts: 18.12
  - BDWx2 Charge Opts: 17.49
  - KNL Baseline: 16.44
  - KNL Push Opts: 18.4
  - KNL Charge Opts: 7.7

1.2x speedup for Sort
KNL now ~2x faster than 2xBDW
On KNL optimisations deliver ~2.6x cumulative speedup

Unnecessary pressure on TLB:

```c
#pragma omp for
for (m = 0; m < mi_new; m++) {
    z0[m] = z00[m];
    z1[m] = z01[m];
    z2[m] = z02[m];
    z3[m] = z03[m];
    z4[m] = z04[m];
}
```

Use vectors, alignment, and copy 1 at a time:

```c
#pragma omp for simd schedule(static:simd) aligned(z0,z00:64) nowait
for (m = 0; m < mi_new; m++)
    z0[m] = z00[m];
```

```
#pragma omp for simd schedule(static:simd) aligned(z1,z01:64) nowait
for (m = 0; m < mi_new; m++)
    z1[m] = z01[m];
```

```
#pragma omp for simd schedule(static:simd) aligned(z2,z02:64) nowait
for (m = 0; m < mi_new; m++)
    z2[m] = z02[m];
```

```
#pragma omp for simd schedule(static:simd) aligned(z3,z03:64) nowait
for (m = 0; m < mi_new; m++)
    z3[m] = z03[m];
```

```
#pragma omp for simd schedule(static:simd) aligned(z4,z04:64) nowait
for (m = 0; m < mi_new; m++)
    z4[m] = z04[m];
```
**NWChem AIMD**

NWChem Ab-initio Molecular Dynamics

Work by:
E. Bylaska (PNNL), Matthias Jacquelin (LBL), Bert de Jong (LBL), Michael Klemm (Intel)

---

**Introduction: Plane Wave Methods**

- 100-1000 atoms, uses plane wave basis
- Many FFTs and DGEMM operations
- "Meaty": Lots of FLOPs, but also bandwidth sensitive

\[
(-1/2)\nabla^2 \Psi + V_{\text{ext}} \Psi + V_{\text{H}} \Psi + V_{\text{xc}} \Psi + V_{\text{N_{\text{elect}}}} \Psi = E \Psi
\]

\[
\langle \Psi | \Psi \rangle = \delta_{ij}
\]

- \( N_e \): number of atoms
- \( N_e \): number of electrons
- \( N_{\text{grid}} \): size of FFT grid

---

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other names and brands may be claimed as the property of others. All products, dates, and figures are preliminary and are subject to change without any notice. Copyright © 2017, Intel Corporation.
Strong Scaling is Key

- 20 psec of simulation time \( \approx \) 200,000 steps
  - 1 sec/step = 2-3 days simulation time
  - 10 sec/step = 23 days simulation time
  - 13 sec/step = 70 days simulation time
- Mesoscale phenomena at longer time scales
  - Assume 1 sec/step
  - 100 psec = 10-15 days simulation time
  - 1 nsec = 100 - 150 days simulation time
- Strong scaling required to reduce time per time step as much as possible
  - At least below 1sec/step
3D FFTs – Pipelined Implementation

- Performed at each step
  - 2 Ne 3D FFTs for DFT
  - Plus (Ne+1)*Ne 3D FFTs for hybrid DFT
- In reciprocal space, sphere of radius Ecut is stored
- 3D FFTs are pipelined
  - Overlap communication and computation
  - Latency reduction
  - N^2 1D FFTs per stage execute in parallel

Lagrange Multiplier

- Sequence of matrix products of shape F or M
  - F: \( N_{\text{pack}} \times N_e \) or \( N_e \times N_{\text{pack}} \) matrix (tall & skinny)
  - M: \( N_e \times N_e \) matrix
  - In general: \( N_{\text{pack}} \sim N_e \)
Lagrange Multiplier – Parallelization

Experimental Setup – NERSC Cori

- **“Haswell”, HSW**
  - Cray* XC40
  - 2S Intel® Xeon® E5-2698v3 processors
  - 32 cores, no Hyper-Threading
  - 2.3 GHz clock frequency
  - 128 GB of DDR4 at 2133 MHz
  - Cray* Aries* w/ Dragonfly

- **“Knights Landing”, KNL**
  - Cray* XC40
  - Intel® Xeon Phi™ 7250 processors
  - 68 cores w/ 4 hardware threads
  - 1.4 GHz clock frequency
  - 96 GB of DDR4 at 2400 MHz
  - Cache mode
  - Quadrant cluster mode
  - Cray* Aries* w/ Dragonfly
Experimental Setup – Benchmarks

- water64:
  - 64 water molecules in a box
  - test intra-node strong scaling
- water256:
  - 256 water molecules
  - test cluster strong scaling
  - $N_e=2056$
  - $N_g=5,832,000\ (180^3)$
  - $N_{pack}=437,000$

Intra-node Performance

- Insight into performance without fabric effects
- Xeon node saturates at about 16 cores, reaching memory bandwidth limits
- Xeon Phi node keeps strong scaling due to the on-package cache memory
- 1.8x speed-up of KNL over HSW node

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Cray* XC40 system, 2S Intel® Xeon® E5-2698v3 processor, Intel® Hyper-Threading technology disabled, 128 GB of DDR4 (8x 16 GB, 2133 MHz), Cray* Aries interconnect with Dragonfly topology; Cray* XC40 system Intel® Xeon Phi™ 7250 processors, 96 GB of DDR4 (8x 16GB, 2400 MHz), quadrant cluster mode, MCDRAM in cache mode, Cray* Aries interconnect with Dragonfly topology.
Performance

Relative Performance – HSW vs KNL

- Strong scaling regime
- Interconnect latency becomes visible
- Less occupancy of the network
- KNL seems to suffer from this more than HSW does

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Cray* XC40 system, 2S Intel® Xeon® E5-2698v3 processor, Intel® Hyper-Threading technology disabled, 128 GB of DDR4 (8x 16 GB, 2133 MHz), Cray® Aries interconnect with Dragonfly topology; Cray® XC40 system Intel® Xeon Phi™ 7250 processors, 96 GB of DDR4 (6x 16GB, 2400 MHz), quadrant cluster mode, MCDRAM in cache mode, Cray® Aries interconnect with Dragonfly topology.
Performance – Effect of the Processor Grid

- Processor grid is a tradeoff
- 2D processor grid: \( N_p = N_{pi} \times N_{pj} \)
- Large \( N_{pj} \) favors FFTs and non-local pseudopotentials
- Lagrange multiplier suffers from large \( N_{pi} \)
- Balancing \( N_{pi} \) and \( N_{pj} \) is required
  - problem size
  - number of ranks

![Graph showing run times of AIMD on 256 Water molecules]
Summary –

- Much of Knights Landing's throughput comes from parallelism:
  - Codes will need to be modernized to fully exploit the features of the chip
  - Usually: thread-parallel and SIMD-parallel execution key to performance

- Optimizations for Knights Landing usually also pay off on Xeon processors

- Plain library approaches are not good enough at times due to special requirements of application kernels
Evaluation of Intel Xeon Phi "Knights Landing": Initial impressions and benchmarking results

J. Eitzinger

PRACE PATC, 28.6.2017

History of Intel hardware developments
The real picture

Finding the right compromise

Core complexity

Intel Skylake-EP

Intel KNL

Nvidia GP100

SIMD

# cores

Frequency
Maximum DP floating point (FP) performance

\[ P_{core} = n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f \]

<table>
<thead>
<tr>
<th>uArch</th>
<th>(n_{super}^{FP})</th>
<th>(n_{FMA})</th>
<th>(n_{SIMD})</th>
<th>(n_{cores})</th>
<th>Release</th>
<th>Model</th>
<th>(P_{core}) [GF/s]</th>
<th>(P_{chip}) [GF/s]</th>
<th>(P_{serial}) [GF/s]</th>
<th>TDP</th>
<th>GF/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Q1/2012</td>
<td>E5-2680</td>
<td>11.7</td>
<td>173</td>
<td>7</td>
<td>130</td>
<td>1.33</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>Q3/2013</td>
<td>E5-2690-v2</td>
<td>24</td>
<td>240</td>
<td>7.2</td>
<td>130</td>
<td>1.85</td>
</tr>
<tr>
<td>KNC</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>61</td>
<td>Q2/2014</td>
<td>7120A</td>
<td>10.6</td>
<td>1210</td>
<td>1.33</td>
<td>300</td>
<td>4.03</td>
</tr>
<tr>
<td>Haswell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td>Q3/2014</td>
<td>E5-2695-v3</td>
<td>21.6</td>
<td>425</td>
<td>6.6</td>
<td>120</td>
<td>3.54</td>
</tr>
<tr>
<td>Broadwell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>22</td>
<td>Q1/2016</td>
<td>E5-2699-v4</td>
<td>17.6</td>
<td>704</td>
<td>7.2</td>
<td>145</td>
<td>4.85</td>
</tr>
<tr>
<td>Pascal</td>
<td>1</td>
<td>2</td>
<td>32</td>
<td>56</td>
<td>Q2/2016</td>
<td>GP100</td>
<td>36.8</td>
<td>4700</td>
<td>1.5</td>
<td>300</td>
<td>15.67</td>
</tr>
<tr>
<td>KNL</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>72</td>
<td>Q4/2016</td>
<td>7290F</td>
<td>35.2</td>
<td>2995</td>
<td>3.4</td>
<td>260</td>
<td>11.52</td>
</tr>
<tr>
<td>Skylake</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>26</td>
<td>Q3/2017</td>
<td>8170</td>
<td>23.4</td>
<td>1581</td>
<td>7.6</td>
<td>165</td>
<td>9.58</td>
</tr>
</tbody>
</table>

**Chebyshev Filter Diagonalization on KNL and P100**

DFG SPPEXA Essex 2 project
Basic ChebFD scheme

1. Filter $n_s$ search vectors
2. Orthogonalize $n_s$ search vectors
3. Go to 1 if not converged

- $n$: matrix/vector dimension
- $n_p$: polynomial degree (defined by application)
- $n_s$: number of search vectors (defined by application)

Test systems
- **Piz Daint** (Switzerland): 5320 Node Cray XC50
- **Oakforst-PACS** (Japan): 8208 Node Fujitsu PRIMERGY

Node-level Performance

![Graph showing performance comparison between KNL and P100](image-url)
Scaling Results

Fujitsu PRIMERGY

Cray XC50

ERLANGEN REGIONAL COMPUTING CENTER

System configuration challenge
Configuration complexity

- **Cluster modes**: lower the latency and increase the bandwidth
  - All-to-all
  - Quadrant mode (default)
  - Sub-numa-clustering (SNC), best performance but explicit
- **Memory modes**:
  - Cache mode (default)
  - Flat mode (explicit)
  - Hybrid
- **Mapping** of application on hardware:
  - Use SMT or not. How many SMT threads?
  - Use all cores?
  - MPI+X. How exactly?
- **Memory configuration**: Alignment and page size choices

---

Impact of QPI snoop mode and CoD on latency

- Starting with HSW, QPI snoop mode can be set via BIOS
  - Early Snoop
  - Home Snoop
  - Home Snoop + Opportunistic Snoop Broadcast (BDW only)
  - Directory (CoD)

<table>
<thead>
<tr>
<th></th>
<th>SNB</th>
<th>IVB</th>
<th>HSW (CoD)</th>
<th>BDW (CoD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>L2</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>L3</td>
<td>40</td>
<td>40</td>
<td>37</td>
<td>47 (non-CoD)</td>
</tr>
<tr>
<td>Mem</td>
<td>230</td>
<td>208</td>
<td>168</td>
<td>280 (HS), 248 (ES), 190 (HS+OSB), 176 (DIR)</td>
</tr>
</tbody>
</table>

Graph 500 (v2.1.4), full chip w/ SMT, Turbo
Xeon E5-2697v4 (BDW)  μarch comparison

Cache/Memory Latency [cycles]
Uncore Frequency: Bandwidth and Energy Consumption

Intel HPCG (16.0.3), n=256, full chip (no SMT), Xeon E5-2697 v4 (BDW)

Intel HPL (16.0.3), N=60.000, full chip (no SMT), Xeon E5-2697 v4 (BDW)

Uncore Frequency: LINPACK Performance
Specific issues with Xeon Phi

- **MCDRAM** adds additional complexity

- **Configuration** of system and **mapping** of application on hardware gets more critical

- The compromise and made with KNL will soon be outdated

- KNL as a hosted cluster system is probably too specialized for a general purpose academic cluster

---

But

- Xeon Phi implements features which are not available anywhere else:
  - High degree of chip level parallelism
  - Multiple memory types and explicit memory control
  - Mesh type on-die topology

- It allowed a glimpse in the future on real hardware
# Lattice Quantum Chromodynamics on the MIC architectures

Piotr Korcyl

Universität Regensburg

Intel MIC Programming Workshop @ LRZ
28 June 2017

---

## Regensburg’s Lattice QCD group

<table>
<thead>
<tr>
<th>Software/Hardware</th>
<th>Physics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilo Wettig</td>
<td>Andreas Schäfer</td>
</tr>
<tr>
<td>Simon Heybrock</td>
<td>Gunnar Bali</td>
</tr>
<tr>
<td>Jacques Bloch</td>
<td>Sara Collins</td>
</tr>
<tr>
<td>Stefan Solbrig</td>
<td>Enno Scholz</td>
</tr>
<tr>
<td>Nils Meyer</td>
<td>Wolfgang Söldner</td>
</tr>
<tr>
<td>Robert Lohmayer</td>
<td>postdocs + PhD students</td>
</tr>
<tr>
<td>Peter Georg</td>
<td></td>
</tr>
<tr>
<td>Daniel Richtmann</td>
<td></td>
</tr>
</tbody>
</table>

---

317
QFT describes the strong interaction at the fundamental level.

Distinctive features
- asymptotic freedom: interaction becomes weaker at shorter distances;
- quarks and gluons confined into colourless bound states (hadrons);
- spontaneous symmetry breaking determines low-energy dynamics.

Figure: Particle spectrum measured by experiment and calculated by BMW Collaboration.
Continuum Quantum Chromodynamics

\[ \mathcal{S}_{\text{QCD}} = \int d^4x \, \text{Tr} \left\{ \frac{1}{2 g_0^2} F_{\mu \nu} F_{\mu \nu} + \sum_{i,j} \bar{\psi}_i \left[ i \left( \gamma^\mu \mathcal{D}_\mu \right)_{ij} - m_0 \delta_{ij} \right] \psi_j \right\} \]

\[ Z(g_0^2, m_0) = \int [A_\mu(x)] [\bar{\psi}(x)] [\psi(x)] \exp \{ i \mathcal{S}_{\text{QCD}}(g_0^2, m_0) \} \]

\[ \langle \mathcal{O} \rangle = \frac{1}{Z(g_0^2, m_0)} \int [A_\mu(x)] [\bar{\psi}(x)] [\psi(x)] \mathcal{O} \exp \{ i \mathcal{S}_{\text{QCD}}(g_0^2, m_0) \} \]

Lattice Quantum Chromodynamics

\[ \mathcal{S}_{\text{QCD}} = a^4 \sum_\mathbf{x} \text{Tr} \left\{ \frac{1}{2 g_0^2} \sum_{\mu < \nu} \text{Re}(1 - U_{\mu \nu}) - \text{tr} \log \det D(U) \right\} \]

\[ \langle \mathcal{O} \rangle = \frac{1}{N} \sum_{\text{configuration}} [\text{set of configurations } (g_0^2, m_0)] \mathcal{O}(\text{configuration}) \]

distributed with Boltzmann probability weight \( \propto \exp \{-\mathcal{S}_{\text{QCD}}(g_0^2, m_0)\} \).
Algorithms: Markov chain

We need set of configurations \( \{ \mathbf{g}_0, m_0 \} \) generated with a probability 
\( \propto \exp \left\{ -\mathcal{S}_{\text{QCD}} (\mathbf{g}_0, m_0) \right\} \) where 
\[
\mathcal{S}_{\text{QCD}} (\mathbf{g}_0, m_0) = a^4 \sum_x \text{Tr} \left\{ \frac{1}{8g_0^2} \sum_{\mu<\nu} \text{Re}(1 - U_{\mu\nu}) + \phi^\dagger (D^\dagger D)^{-1} \phi \right\}
\]

where \( D \) is the Wilson-Dirac operator \( D_{\alpha \beta}^{AB} (x|y) \):

- sparse matrix of size \((12TL^3) \times (12TL^3)\), typically \(10^9 \times 10^9\).

Hybrid Monte Carlo algorithm

- introduce associated momenta with a gaussian probability distribution
- integrate Hamilton's EoM for MC-time interval \( \tau = 2.0 \):
  - advance gauge and pseudofermion variables: \( \dot{Q} = \frac{\partial H}{\partial P} = P \)
  - advance momenta: \( \dot{P} = -\frac{\partial H}{\partial \dot{Q}} = -\frac{\partial S}{\partial Q} \)
- perform accept/reject step with a probability \( \exp \left( -\Delta S \right) \)

Gauge field configurations

Coordinated Lattice Simulations (CLS) ensembles with 2+1 dynamical fermions

Cost (so far)

area \( \propto \) cost

\( (1 \text{ superMUC core-h} = 3.7 \text{ Juqueen core-h}) \)

\( \Rightarrow \) total of 165 Mcore-h superMUC-equivalent, 350 TB of data
Machines

Some of the computer time allocations

Dedicated machines:
- QPACE - PowerXCell 8i multi-core processors
- QPACE 2 - Xeon Phi 7120X accelerators (KNC), 64 nodes, 4 KNC each
- QPACE 3 - Xeon Phi 7210 accelerators (KNL), 320 nodes, 1 KNL each

General purpose machines:
- LRZ superMIC
- CINECA Marconi KNL cluster
- LRZ superMUC
- JUQUEEN @ Juelich
- Cray XC40 at Warsaw University

Solving the Dirac equation

State-of-the-art linear solver

Inversions of the Dirac operator contribute significantly to the total numerical cost. Typically an iterative solver with a suitable preconditioner to control the condition number of the Dirac operator is used.

Domain Decomposition Adaptive Algebraic Multigrid Solver (DD-$\alpha$AMG) is the state-of-the-art solver developed and implemented by mathematicians and physicists from Univ. of Wuppertal and Regensburg.
Solving the Dirac equation

**Typical setups**

- lattices: \(32^3 \times 96, \ldots, 64^3 \times 192 \Rightarrow \text{MPI-ranks: 512, \ldots, 12288}\)

**State-of-the-art linear solver: overview of implementation details**

- global lattice divided into local lattice via MPI (or dedicated library pMR)
- even-odd decomposition allows to work simultaneously on each local lattice
- use preconditioning: choose a matrix \(M^{-1} \approx D^{-1}\) and solve
  \[DM^{-1}Mu = DM^{-1}v = f\]
- domain decomposition for the preconditioner reduces communication
  - inversions done on each block separately, ideally from cache
  - boundary exchanges not so frequent
- work on local lattices done by threads (persistent openMP threads)

**Details**

- ‘Lattice QCD with Domain Decomposition on Intel Xeon Phi Co-Processors’
  S. Heybrock *et al.*, arXiv:1412.2629

**KNC/KNL architectures**

- fuse identical components of fields from different sites (site-fusing) allows to exploit large vector units
- overlap computations and communication to hide communication latency
- half-precision for the preconditioner to reduce memory bandwidth requirements and memory footprint
- multiple right-hand-sides
- real and imaginary parts kept separated in registers

**Three examples**

- implementation of the domain decomposition preconditioner and the changes in the data layouts \(\rightarrow\) SIMD registers
- sophisticated boundary exchanges \(\rightarrow\) hide communication latency
- implementation of the three-point functions measurement code and the changes in parallelization \(\rightarrow\) load balance
Code development: three examples

Domain-decomposition preconditioner: SIMD layouts, SOA and AOS

Simon Heybrock’s figure

Code development: three examples

Domain-decomposition preconditioner: SIMD layouts, SOA and AOS

Simon Heybrock’s figure
Code development: three examples

Domain-decomposition preconditioner: SIMD layouts, SOA and AOS

Simon Heybrock's figure

Code development: three examples

Domain-decomposition preconditioner: SIMD layouts, SOA and AOS

Simon Heybrock's figure
Code development: three examples

Domain-decomposition preconditioner: SIMD layouts, SOA and AOS

Simon Heybrock's figure

Communication latency hiding for DD

boxes represent domains, numbers represent order of execution, small letters represent order of communication

bad:

Tilo Wettig’s slide from QCDNA16 workshop.
Code development: three examples

Three-point correlation functions: load balancing

Description:
- each ellipse corresponds to $N_x N_y N_z \times 12 \times 12$ matrix multiplications
- each star corresponds to $N_x N_y N_z \times 12 \times 12$ matrix multiplications
- we need to repeat for different positions of the central ellipse and all stars in between

Data redistribution:
- redistribute data along the $\tau$ direction and compute in parallel
**Code development: three examples**

**Three-point correlation functions: load balancing**

Yet another data redistribution:
- redistribute data along the $t$ direction and compute in parallel

**Practical details**

**KNC/KNL**
- KNC: we run with 2 or 4 threads/core
  - 2 threads: cache misses
  - 4 threads: cache eviction
- KNL/OmniPath: we run with > 4 MPI ranks / KNL

**Example wall-clock timings on QPACE 2 (in seconds)**

<table>
<thead>
<tr>
<th>Lattice size</th>
<th>MDA</th>
<th>BDA</th>
<th>Bar.spec.</th>
<th>H-Bar.spec.</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$32^3 \times 96$</td>
<td>308</td>
<td>777</td>
<td>33</td>
<td>90</td>
<td>LHA</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>76</td>
<td>13</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>$32^3 \times 128$</td>
<td>379</td>
<td>1011</td>
<td>51</td>
<td>127</td>
<td>LHA</td>
</tr>
<tr>
<td></td>
<td>53</td>
<td>112</td>
<td>26</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>$48^3 \times 128$</td>
<td>370</td>
<td>910</td>
<td>47</td>
<td>101</td>
<td>LHA</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>102</td>
<td>19</td>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>
Solver single-node scaling

Results: Single KNC

- almost perfect scaling (except for load imbalance):
  - cores can work independently during MR inversion
  - almost no competition for memory access since MR runs from cache

Tilo Wettig’s slide from QCDNA16 workshop.

Physical results

Spin content of the nucleon presented at Lattice2017 in Granada

$\gA$: continuum limit at $M_N \approx 420$ MeV

$\gA$: continuum limit at $M_N \approx 420$ MeV
Conclusions

Quantum Chromodynamics and HPC

Lattice QFT & High Performance Computing combined with high precision experiments is the only way to check the correctness of our understanding of the Standard Model and therefore to discover New Physics behind it.

QCD and new architectures

- adapt the application to both processor and interconnect
- choose an appropriate algorithm
- vectorization: data layout!
- load balancing: make sure all cores are busy
- many other ways of enforcing vectorization
- avoiding intrinsics - code portability between KNC and KNL

Thank you for your attention!

Piotr Korczył
The experience of the HLST on Europe’s biggest KNL cluster

Tamás Fehér, Serhiy Mochalskyy, Nils Moschüring
Roman Hatzky
Intel MIC Programming Workshop
LRZ

Marconi – KNL at CINECA, Bologna

Total number of KNL nodes: 3600

Partition dedicated to the EUROfusion community: 392 (144 flat / 248 cache mode)

-> about 1 Pflop/s

Photo: F.Pierantoni
Overview

• Memory Bandwidth benchmarks
• Latency benchmarks
• OpenMP Benchmarks
• Code Performance
• Summary

STREAM and IMB

MEMORY BANDWIDTH BENCHMARKS
STREAM Memory Bandwidth
flat mode MCDRAM

Alignment also important for cache mode
STREAM Memory Bandwidth

Array size

- Flat MCDRAM 490 GiB/s
- 339 GiB/s
- Cache
- Flat 90 GiB/sec
- 59 GiB/sec

Cache versus flat

78 GB total arrays size

Marconi Stream Triad scatter KNL 68 tasks flat hemisphere
Marconi Stream Triad scatter KNL 68 tasks flat quadrant
Marconi Stream Triad scatter KNL 68 tasks flat snc2
Marconi Stream Triad scatter KNL 68 tasks cache hemisphere
Marconi Stream Triad scatter KNL 68 tasks cache quadrant
Marconi Stream Triad scatter KNL 68 tasks cache snc2
STREAM Memory Bandwidth over time

linpack_knl_cache_105080.r064u06s01 at 2017-04-06_23-39-26

linpack_knl_cache_130796.r064u06s01 at 2017-04-14_16-10-05
Intel MPI benchmark - bandwidth

LATENCY BENCHMARKS
Latency

Broadwell

- 32k L1 cache
- 265k L2 cache
- 45M L3 cache

DDR ~ 90ns

Vector size (B)

- 1k
- 10k
- 100k
- 1M
- 10M
- 1G
- 10G

Latency (ns)

Nils Moschüring
Intel MIC Programming Workshop, June 28th, 2017

KNL

- 32k L1 cache
- 1M L2 cache

MCDRAM ~ 170 ns

DDR ~ 155 ns

Vector size (B)

- 1k
- 10k
- 100k
- 1M
- 10M
- 1G
- 10G

Latency (ns)

IMB – Ping Pong Test - Latency

Intra node Marconi

Broadwell

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>0.61</td>
<td>1.09</td>
</tr>
</tbody>
</table>

Latency (µs)

Knights Landing

<table>
<thead>
<tr>
<th></th>
<th>KNL0</th>
<th>node0</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>0.85</td>
<td></td>
</tr>
</tbody>
</table>

Intra node HELIOS

Sandy Bridge

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>0.25</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Latency (µs)

Knights Corner

<table>
<thead>
<tr>
<th></th>
<th>KNC0</th>
<th>node0</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>2.7</td>
<td></td>
</tr>
</tbody>
</table>

Latency (µs)

Nils Moschüring
Intel MIC Programming Workshop, June 28th, 2017
### IMB – Ping Pong Test - Latency

**Inter node Marconi**

<table>
<thead>
<tr>
<th>Node</th>
<th>CPU</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>CPU0</td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td>node1</td>
<td></td>
</tr>
</tbody>
</table>

**Broadwell**

**Inter node HELIOS**

<table>
<thead>
<tr>
<th>Node</th>
<th>CPU</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>CPU0</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>node1</td>
<td></td>
</tr>
</tbody>
</table>

**Sandy Bridge**

**Knights Landing**

<table>
<thead>
<tr>
<th>Node</th>
<th>KNL</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>KNL0</td>
<td>3.99</td>
</tr>
<tr>
<td></td>
<td>node1</td>
<td></td>
</tr>
</tbody>
</table>

**Knights Corner**

<table>
<thead>
<tr>
<th>Node</th>
<th>KNC</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>KNC0</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td>node1</td>
<td></td>
</tr>
</tbody>
</table>

---

**OPENMP BENCHMARKS**

Nils Moschüring

Intel MIC Programming Workshop, June 28th, 2017
OpenMP overhead

- KNL overhead ≈ 2x larger:
  - more threads
  - lower CPU frequency
- Exception: ATOMIC 5x longer, use CRITICAL instead
- Using EPCC OpenMP Microbenchmarks J.M. Bull et. al

OpenMP overhead + hyper-threading
CODE PERFORMANCE

Roofline Model

Full node vector
Full node scalar
Single core vector
Single core scalar

KNL Broadwell
MCDRAM 490 GB/s
Cache 60-340 GB/s
DDR4 90.7 GB/s

GFlop/sec

1/16 1/8 1/4 1/2 1 2 4 8 16 32 64

1 2 4 8 16 32 64

arithmetic intensity (flop/byte)
HPCG benchmark

- HPCG: sparse 3D problem with multigrid preconditioned conjugate gradient solver.
- The Intel optimized version of the HPCG benchmark was executed in one node.

Gysela execution time

- Test case: 127 x 256 x 64 x 63 (Nr x Ntheta x Nphi x Nvar, Nmu=0)
- 1 node, 4 MPI tasks, 8 threads (Broadwell) / 16 threads (KNL)
Nils Moschüring Intel MIC Programming Workshop, June 28th, 2017

UTL_TRIDIAG_R

- Solve a tridiagonal system
  - forward elimination
  - back substitution
- Not vectorizable

\[
\begin{bmatrix}
  b_1 & c_1 & 0 & & \\
  a_2 & b_2 & c_2 & \ddots & \\
  & a_3 & b_3 & \ddots & \ddots \\
  & & \ddots & \ddots & c_{n-1} \\
  0 & & & a_n & b_n
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  \vdots \\
  x_n
\end{bmatrix}
= \begin{bmatrix}
  d_1 \\
  d_2 \\
  d_3 \\
  \vdots \\
  d_n
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>instruction</th>
<th>instruction</th>
<th>Broadwell latency</th>
<th>Broadwell throughput</th>
<th>KNL latency</th>
<th>KNL throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>3N</td>
<td>FMA</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>2N-1</td>
<td>DIVSD</td>
<td>10-14</td>
<td>1/5-1/4</td>
<td>42</td>
<td>1/42</td>
</tr>
<tr>
<td>2N-1</td>
<td>VDIVPD</td>
<td>19-23</td>
<td>1/16</td>
<td>32</td>
<td>1/32</td>
</tr>
</tbody>
</table>

Summary

• Good stuff:
  – MPI latency and OpenMP overhead comparable
  – KNL can match Broadwell performance without extensive tuning for most codes
  – Optimization on KNL helps on Broadwell and vice versa

• Bad Stuff:
  – Cache mode operation can be dubious
  – Peak performance hard to reach
  – Hyperthreading rarely useful
Summary

• KNL is equal to Broadwell if your code either
  – Has very good scalability (to make use of increased core count)
  – Has very good vectorization (to make use of more vector units)
  – Effectively uses only 16 GB (to make use of higher bandwidth)
• If more than one holds, you will probably get more performance than on Broadwell
• Memory mode Quadrant seems to be the best
Porting the ELPA library to the KNL architecture

MIC Programming Workshop @ LRZ

Dr. Andreas Marek

together with

Dr. Hermann Lederer, Dr. Pavel Kus, Dr. Lorenz Hüdepohl

Outline

- What is the ELPA library
- A roofline model for KNL
- Writing AVX-512 kernels
- Going to many nodes
- KNL experiences
- Conclusions
The ELPA library: Eigenvalue solvers for Petaflop applications

- ELPA is a high-performance library for the massively parallel solution of dense, symmetric (hermitian) eigenvalue problems
- replacement for Scalapack routines (pdsyevd, pzheevd, pdsyevr, pzheevr)
- widespread used, e.g. in electronic structure codes
- opensouce (see https://elpa.mpcdf.mpg.de)
- available with many linux distributions (Debian, Fedora, Suse etc.)
- supports a large variety of platforms (X86, X86_64, OpenPower, BG P/Q, GPUs, KNL)
- achieved already 0.3 PFLOPS/s on 294k cores (BG/P FZJ Juelich in 2011)

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

=> used in many HPC centers worldwide (Juelich, Cineca, ORNL, LLNL ...)

=> nowadays used in most material science codes (FHI-aims, Quantum Esspresso, VASP, OpenMX...)

=> Intel is interested in integrating ELPA in it`s MKL library

The ELPA Library - Scalable Parallel Eigenvalue Solutions for Electronic Structure
Theory and Computational Science: A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang,
T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer,

Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations: T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni,

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
The ELPA library: Eigenvalue solvers for Petaflop applications

=> used in many HPC centers worldwide (Juelich, Cineca, ORNL, LLNL …)

=> nowadays used in most material science codes (FHI-aims, Quantum Espresso, VASP, OpenMX…)

=> Intel is interested in integrating ELPA in it’s MKL library

The ELPA Library - Scalable Parallel Eigenvalue Solutions for Electronic Structure
Theory and Computational Science. A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang,
T. Auckenthaler, A. Heincke, H.-J. Bungartz, and H. Lederer,

Parallel solution of partial symmetric eigenvalue problems from electronic
structure calculations. T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni,

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

Development of the ELPA library

- The development of the ELPA library started under the lead of MPCDF in 2008; it has been funded by a BMBF project 01H08007 from Dec. 2008- Nov. 2011

- ELPA is maintained and developed at MPCDF

- Since Feb. 2016 a new BMBF project 01H15001 is funding the latest developments of ELPA (including the porting to KNL)

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
The algorithm in a nutshell - I

1. step: - transformation of matrix to banded matrix  
   - store information for later back-transform

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

The algorithm in a nutshell - II

2. step: - transformation to tridiagonal matrix  
   - store information for later back-transform

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
The algorithm in a nutshell - III

3. step: - solve eigenvalue problem

The algorithm in a nutshell - IV

4. step: - back-transformation to banded matrix
The algorithm in a nutshell - V

5. step: - back-transformation to full matrix

The backtransformation is computationally expensive and relies on optimized kernels

The algorithm in a nutshell - VI
Prior to porting ELPA (and especially judging the performance) we want to have a feeling on what you can expect/get on a new architecture

=> roofline model is helpful

very nice work of NERSC in the NESAP project

Typical arithmetic intensities in BLAS routines:

- BLAS Level 2 (Matrix-Vector): \( \sim O(1) \)
- BLAS Level 3 (Matrix-Matrix): \( \sim O(n) \)


Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
we have used Intel’s „Software Development Emulator“ (http://software.intel.com/en-us/articles/intel-software-development-emulator) on a standard Haswell node to develop the AVX-512 kernels

Porting the kernels to AVX-512 - using Intel’s SDE

- we have used Intel’s „Software Development Emulator“ (http://software.intel.com/en-us/articles/intel-software-development-emulator) on a standard Haswell node to develop the AVX-512 kernels


=> this allowed us to start with the development of the AVX-512 kernels before the hardware was available

=> this lead to ~4000 lines of AVX-512 intrinsics code

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

Porting the kernels to AVX-512 - I

- Depending on the setup, 30% to 75% of runtime is spend in the back-transformation kernels

Example code:

```c
__mm512d h1, h2, q1, x1, y1, q2, x2, y2;

for(i = 2; i < nb; i++)
{
    h1 = __mm512_set1_pd(hh[i-1]);
    h2 = __mm512_set1_pd(hh[hh+i]);

    q1 = __mm512_load_pd(&q[i*ldq]);
    x1 = __mm512_FMA_pd(q1, h1, x1);
    y1 = __mm512_FMA_pd(q1, h2, y1);
    q2 = __mm512_load_pd(&q[i*ldq+8]);
    x2 = __mm512_FMA_pd(q2, h1, x2);
    y2 = __mm512_FMA_pd(q2, h2, y2);

    ...
```

A problem turned out to be a compiler independent alignment of the data (and removing the effort from the users that build the ELPA library)

=> we had to use „posix_memalign“ For the data allocation and to ensure correct striding

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
Porting the kernels to AVX-512 - II

Programming AVX-512 intrinsics showed a little surprise:

- KNL and SkyLake server architecture share a large set of instructions
  - but sets are not identical
- Subsets are represented by individual feature flags (CPUID)

On KNL some instructions are missing (compared to "old" Xeons and upcoming SkyLake)

=> more code needed to program around
=> higher CPI

Reminder: AVX-512 – KNL and SKX

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

Porting the kernels to AVX-512 - III

Programming AVX-512 intrinsics showed a little surprise:

- KNL and SkyLake server architecture share a large set of instructions
  - but sets are not identical
- Subsets are represented by individual feature flags (CPUID)

On KNL some instructions are missing (AVX-512DQ of normal xeon line)

=> more code needed to program around
=> higher CPI

Reminder: AVX-512 – KNL and SKX

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
Porting the kernels to AVX-512 - IV

Theoretical limit:
1.3 GHz * 32 Flops/cycle * #cores
= 2.6 TFLOPS/s @ 64 cores

Speedup AVX2 → AVX512 ~ 1.5x - 1.6x

ELPA @ 64 cores: ~520 GFLOPS/s

double precision values!

- ELPA measurements are ~ ¼ of theoretical value (ELPA is memory bound)
- we still work on improving this value; new version of Intel Advisor allows to measure the roofline directly (not yet done)

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

Porting the kernels to AVX-512 - V

Theoretical limit:
1.3 GHz * 32 Flops/cycle * #cores
= 2.6 TFLOPS/s @ 64 cores

Speedup AVX2 → AVX512 ~ 1.5x-1.6x

memory bound
=> we can not expect peak


- ELPA measurements are ~ ¼ of theoretical value (ELPA is memory bound)
- we still work on improving this value; new version of Intel Advisor allows to measure the roofline directly (not yet done)

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
Porting the kernels to AVX-512 - VI

Kernel ~ 44% of r.t.

Kernel ~ 20% of r.t.

Linear scaling: \(1/1.8^n\)

~ 1.3x

~ 1.5x - 1.6x for kernels AVX2 → AVX512

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

Going to many nodes...

- In a collaboration between the ELPA-AEO and the ELSI project (http://wordpress.elsi-interchange.org) it was possible to do first test of the ELPA library on the KNL system „THETA“ of the Argonne National Lab

- All benchmarks have been performed by Vazquez-Mayagoit Alvaro (Argonne National Lab)

  => successful runs on up to 200,000 cores and a matrix size >1,000,000 could be demonstrated

  => we are still trying to improve the scaling on KNL

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
Going to many nodes...

**Preliminary results**

ELPA2 double-precision real, KNL AVx-512 optimized

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH1135

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

**KNL experiences - Summary I**

General look and feel:

- it was straightforward to compile and run ELPA on KNL; first results after a few minutes
- architecture and software feels like a good old friend
- some initial surprises:
  a) compiler performance: in some setups GNU (gfortran + gcc) 6.1.0 was faster (up to 20%) than Intel (17.0.098); We have not checked again with newer versions of Intel 2017 or Intel 2018 beta. Probably this is not the case anymore
  b) documentation: it was not easy to find details like latency of MCDRAM (~190 ns), DRAM (~120 ns), ...
  c) missing instructions on AVX-512 intrinsics

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
KNL experiences – Summary II

Tools:

- **Intel SDE**: fantastic tool, enabled writing and testing of AVX-512 kernels, before having access to real hardware
- **Intel Advisor**: very useful tool (and has improved a lot recently). Latest ability to create roofline plots helps extremly (we could only use that up to now on Haswell systems)
- **Intel Vtune**: is a also very useful with interesting metrics (on Xeon processors)
  We found the advanced metrics on KNL less useful
- we percieve an „information gap“ between Vtune (high-level code analysis, low-level hardware information) and Advisor (detailed code analysis, most relevant metrics for advanced performance tuning):

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

KNL experiences – Summary III

It would have been helpful if we could have easily obtained the following Information:

- how much percentage of code is vectorized (total + subroutine level)
- what is the performance in GFlop/s (total + subroutine level)
- finding the arithmetic intensity (total + subroutine level)
- memory bandwidths used and data amount transferred (total + subroutine level)
- which parts of the code are memory bound or compute bound on KNL

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
KNL experiences – Summary III

It would have been helpful if we could have easily obtained the following information:

- how much percentage of code is vectorized (total + subroutine level)
- what is the performance in GFlop/s (total + subroutine level)
- finding the arithmetic intensity (total + subroutine level)
- memory bandwidths used and data amount transferred (total + subroutine level)
- which parts of the code are memory bound or compute bound on KNL

Good perspective: with the new version of Intel Advisor (which we sadly do not yet have on our KNL system) a lot of these points are addressed!

=> In the near future we will be able to answer these questions for ELPA on KNL systems

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)

Conclusions

- As always: roofline analysis extremely helpful for performance optimisations
- We have ported the ELPA library to run on KNL clusters
- Writting AVX-512 intrinsic kernels proved to be necessary
- Successfull runs on up to ~200,000 cores have been demonstrated
- We are still working on improving the performance
  - detailed roofline analysis with new Advisor
  - cache blocking
  - but also trying libxsmm (dgemm sizes get small in ELPA during iteration)
- Some of the Intel tools (SDE, Advisor) were really helpfull for working on KNL
- The work on KNL is not yet finished ...

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
Questions?

Thank you for your attention

Intel MIC Programming Workshop @ LRZ, Dr. Andreas Marek (MPCDF)
PRACE PATC Course: Intel MIC Programming Workshop & Scientific Workshop: HPC for natural hazard assessment and disaster mitigation, 26 - 30 June 2017, LRZ

● Czech-Bavarian Competence Team for Supercomputing Applications (CzeBaCCA)

● BMBF funded project that started in Jan. 2016 to:
  - Foster Czech-German Collaboration in Simulation Supercomputing
    - series of workshops
  - Establish Well-Trained Supercomputing Communities
    - joint training program
  - Improve Simulation Software
    - establish and disseminate role models and best practices of simulation software in supercomputing
CzeBaCCA Trainings and Workshops

- Intel MIC Programming Workshop, 3 – 4 February 2016, Ostrava, Czech Republic
- Scientific Workshop: SeisMIC - Seismic Simulation on Current and Future Supercomputers, 5 February 2016, Ostrava, Czech Republic
- PRACE PATC Course: Intel MIC Programming Workshop, 27 - 29 June 2016, Garching, Germany
- Scientific Workshop: High Performance Computing for Water Related Hazards, 29 June - 1 July 2016, Garching, Germany
- PRACE PATC Course: Intel MIC Programming Workshop, 7 – 8 February 2017, Ostrava, Czech Republic
- Scientific Workshop: High performance computing in atmosphere modelling and air related environmental hazards, 9 February 2017, Ostrava, Czech Republic
- PRACE PATC Course: Intel MIC Programming Workshop, 26 – 28 June 2017, Garching, Germany
- Scientific Workshop: HPC for natural hazard assessment and disaster mitigation, 28 - 30 June 2017, Garching, Germany

26.-28.6.2017 Intel MIC Programming Workshop @ LRZ
Agenda: Wednesday

- **Wednesday, June 28, 2017, 13:00-18:00, Hörsaal, H.E.009 (Lecture Hall)**
- Plenum session with invited talks on MIC experience and best practice recommendations (joint session with the Scientific Workshop "HPC for natural hazard assessment and disaster mitigation"), public session
- 13:00-13:30 Luigi Iapichino, IPCC@LRZ: "Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures"
- 13:30-14:00 Michael Bader/Carsten Uphoff, IPCC@TUM: "Extreme-scale Multi-physics Simulation of the 2004 Sumatra Earthquake"
- 14:00-14:30 Vit Vondrak/Branislav Jansik, IPCC@IT4I: "Development of Intel Xeon Phi Accelerated Algorithms and Applications at IT4I"
- 14:30-15:00 Michael Klemm, Intel: "Application Show Cases on Intel® Xeon Phi™ Processors"
- 15:00-15:30 Coffee Break
- 15:30-16:00 Jan Eitzinger, RRZE: "Evaluation of Intel Xeon Phi "Knights Landing": Initial impressions and benchmarking results"
- 16:00-16:30 Piotr Korcyl, University of Regensburg: "Lattice Quantum Chromodynamics on the MIC architectures"
- 16:30-17:00 Nils Moschüring, IPP: "The experience of the HLST on Europes biggest KNL cluster"
- 17:00-17:30 Andreas Marek, Max Planck Computing and Data Facility (MPCDF), "Porting the ELPA library to the KNL architecture"
- 17:30-18:00 Q&A, Wrap-up
- 19:00 Informal Dinner at Gasthof Neuwirt

PRACE PATC Evaluation Form

Please fill out the course evaluation form before leaving this room to help us and PRACE to increase the quality of PATC trainings.

[https://events.prace-ri.eu/event/609/evaluation/evaluate](https://events.prace-ri.eu/event/609/evaluation/evaluate)

http://tinyurl.com/yabw6zzzw

You will get a USB memory stick + certificate for free :-)

Thank you!!
Upcoming courses @ LRZ

- **Scientific Workshop: HPC for natural hazard assessment and disaster mitigation**
  Wednesday, June 28, 2017 13:00 - Friday, June 30, 2017 around noon

- **Using R at LRZ**
  Thursday, July 4, 2017, 9:00-17:00

- **Workshop - HPC mit COMSOL Multiphysics am LRZ**
  Thursday, July 13, 2017, 9:30 - 16:00

- **Compact Course: Iterative Linear Solvers and Parallelization**
  Monday, September 4 - Friday, September 8, 2017, 08:30 (Mo) - 15:30 (Fr)

- **CLC training: NGC data analysis**
  Thursday, Sep 7, 2017 9:30 - 17:00

- **PRACE PATC Course: Advanced Fortran Topics**
  Monday, September 11 - Friday, September 15, 2017, 09:00 - 18:00

- **PRACE PATC Course: Node-Level Performance Engineering**
  Thursday, November 30 - Friday, December 1, 2017, 9:00 - 17:00

- **PRACE PATC Course: Introduction to hybrid programming in HPC**
  Thursday, January 18, 2018, 10:00 - 17:00

---

PATC Course Curriculum @ LRZ

- **Intel MIC Programming Workshop**
  3-days, 3 LRZ + 1 RRZE + 1 Intel lecturers + 8 invited speakers
  June 2017 & 2018

- **Advanced Fortran Topics**
  2 days, 2 LRZ lecturers, September 2017 & 2018

- **Node-Level Performance Engineering**
  2 days, 2 RRZE lecturers, December 2017 & 2018

- **Introduction to hybrid programming in HPC**
  1-day, 1 HLRS & 1 RRZE lecturers, January 2018

- **Advanced Topics in High Performance Computing**
  4-days, 3 LRZ + 2 RRZE lecturers, March 2018

- **VI-HPS Tuning Workshop**
  5-days, 15 lecturers, April 2016 & 2018

- **HPC Code Optimisation Workshop**
  1 day, 2 IPCC @ LRZ lecturers, May 2018
HPC Courses

- Information on further HPC courses:
  - by LRZ: http://www.lrz.de/services/compute/courses/
  - by the Gauss Centre of Supercomputing (GCS): http://www.gauss-centre.eu/training
  - by the PRACE Advanced Training Centres (PATCs): http://www.training.prace-ri.eu/
  - by IT4Innovations: http://prace.it4i.cz/kurzy-it4innovations/

Acknowledgements

- IT4Innovation, Ostrava.
- Partnership for Advanced Computing in Europe (PRACE)
- Intel
- BMBF (Federal Ministry of Education and Research)
- Dr. Karl Fürlinger (LMU)
- J. Cazes, R. Evans, K. Milfeld, C. Proctor (TACC)
- Adrian Jackson (EPCC)
Thanks for joining the Intel MIC Programming Workshop!
Part 3

Hands-On Sessions

June 26 – 28, 2017

Leibniz Supercomputing Centre
Garching b. München, Germany
MIC-Native and Offload-lab: Running simple C Programs in Native and Offload Mode

In this lab you run simple programs in native and offload mode. We then go on to offload a matrix-matrix multiplication and perform a scaling analysis.

Appropriate Environment

Start 3 xterm windows:
- 1 xterm with a shell on the login node supermic.smuc.lrz.de
- 1 xterm with a shell on a compute node i01r13??? (submit a job and look at llq to figure out the hostname of the allocated compute node)
- 1 xterm with a shell on the associated MIC i01r13???-mic0

Attention:
- Compile on supermic.smuc.lrz.de
- Run on Compute nodes i01r13c?? for Offload and MPI
- Run on MICs i01r13c??-mic0/1 for Native Mode

Lab 1: Running MIC binaries natively

- Compile the program hello.c for MIC using
  
  icpc -mmic hello.c -o hello-mic

- Try to launch the program on the host.

- Login to the MIC and execute the program under your home directory
  /home/hpc/a2c06/a2c06??

- Execute the program on the host using micnativeloadex. Look at the output of micnativeloadex program -l.

- Get information about the number of cores on a MIC by using the tools micinfo, micinfo -listdevices, micsmc -a on the host.
• Login to the MIC and get information about the cores, memory etc. by inspecting files like /proc/cpuinfo, /proc/meminfo or using tools like top.

• Modify the hello world program, so that also the number of logical cores is printed out. Run the program on the host and on the MIC.

• Compile the program pthreadspin.c using "icpc -mmic -O0 -lpthread" for the MIC architecture. Run the program using micnativeloadex. Login to the MIC and watch the CPU load using top and ps. Look on the threads using ps -eLF.

**Lab 2: Offloading simple code to Intel Xeon Phi**

• Add a new code block which prints “MIC: Hello world from MIC” to the hello world program. Add an offload pragma for the MIC architecture.

• Run the code on the login node vs. the compute nodes.

• Extend the “hello world” functions to print out the hostname and the numbers of cores of the MIC and the host.

• Compile using one of the compiler options -offload=optional, -offload=mandatory (Default) and -offload=none. Run each time on the login node and a compute node.

• Try to figure out more about the environment under which offloaded code is running. Offload system(“cmd”) calls to get info from commands like set, hostname, uname -a, whoami, id etc.

**Lab 3: Offloading simple numerical code to Intel Xeon Phi**

• Use the exercises c1.c and c2.c.

• Include appropriate Intel Offload pragmas.

• Compile using icc -restrict. How many threads are executing the binary?

• Parallelise using the appropriate OpenMP worksharing construct. To set the number of threads on the MIC you can use:
  
  o export MIC_ENV_PREFIX=MIC
  o export MIC_OMP_NUM_THREADS=...

• Export OFFLOAD_REPORT=2 and rerun the 2 programs. Dito for H_TRACE=1 and H_TIME=1.

• Compile the OpenMP parallelised program for MIC and run in natively. How many threads run per default?

• Natively set number of threads to 1, 2, 244 and figure out the number of threads running.
Lab 4: Offloading MxM code to Intel Xeon Phi

- Parallelize the matrix-matrix multiplication matrixmul.cpp using OpenMP.
- Compile using icpc -mmic -vec-report3 [-offload=optional] -openmp
- Run the program on the MIC natively or via micnativeloadex.
- Watch the program again on the MIC and via micsmc -a.
- Add an appropriate offload target(mic) pragma around the region with the for-loops.
- Add a function call offload_check(void) to the Offload region which checks if the code is really running on the Coprocessor. The routine should print out where it is running depending on the value of __MIC__.
- Also print out the number of current / max OMP threads (omp_get_num_threads(), omp_get_max_threads).
- Test the strong scaling of the code. Run the code with different numbers of threads, but with same matrix size 2000. Write a small script that exports OMP_NUM_THREADS and starts the program for the following sizes.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Runtime(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
<tr>
<td>236</td>
<td></td>
</tr>
</tbody>
</table>

- Write the data into a file and plot it, e.g. with gnuplot.
- Repeat for larger matrix sizes.
- Compare with the native Host / Xeon Phi performance.
KNL MCDRAM Usage Lab

In this lab you run simple programs and the stream benchmark to use MCDRAM and DDR on 2 KNLs with different memory/cluster mode configuration.

Appropriate Environment

Open 3 xterms. In 2 xterms first login to the Linux-Cluster (directly reachable from the course PCs, use only account a2c06aa!)

```
ssh lxlogin1.lrz.de -l a2c06aa
```

Then login in one xterm: `ssh mcct03.cos.lrz.de` and in the other xterm: `ssh mcct04.cos.lrz.de`

Login to the SuperMIC login node in the third xterm.

Lab 1: First steps

- Figure out the number of physical cores, DDR and MCDRAM memory size on both KNLs. Compare with stampede2 shown on the slides.
- Which memory mode is used on mcct03 and mcct04?
- Which cluster modes could be configured?

Lab 2: Testing compatibility on KNL

- Compile `hello.c` using “icc hello.c” and run on the KNLs and the login node.
- Recompile `hello.c` using “icc -xmic-avx512 hello.c” and run on the KNLs and the login node.
- Compile using “icc -xsse2 -axmic-avx512 hello.c” and compare again on the KNLs and the SuperMIC login node.
- You can also test versions compiled with “-mmic” on KNLs and the above versions on the KNCs.
Lab 3: Measuring stream bandwidth on KNL

- Compile the stream benchmark using
  `icc -gopenmp -O2 -xMIC-AVX512 stream.c`
- Measure on both KNLs.
- Compare performance on both KNLs using `OMP_NUM_THREADS=1` and `OMP_NUM_THREADS=x`, where \( x \) is the number of physical cores
- Use
  `numactl -m 0 ./stream`
  and
  `numactl -m 1 ./stream`
  and compare the performance both with `OMP_NUM_THREADS=1` and `OMP_NUM_THREADS=x`
- Copy `stream.c` in `stream-hbw.c`. Change the code to dynamically allocate the arrays \( a, b, c \) using `hbw_malloc`.
- Run the code on the KNL with flat memory mode using both
  `numactl -m 0 ./stream-hbw`
  and
  `numactl -m 1 ./stream-hbw`
  and compare the performance both with `OMP_NUM_THREADS=1` and `OMP_NUM_THREADS=x`. Which memory is allocated in these cases, MCDRAM or DDR?
MKL-lab1 : SGEMM with Automatic Offload

In this example we use the SGEMM function from Intel MKL, to offloading computation to Intel MIC architecture coprocessor.

1. Objectives and learning goals

- Open in an editor the source code
- Set up the build environment
- Replace the WORK FOR YOU comments with MKL calls
- Build the program (for KNC and KNL, see the slides)
- Run the program with various matrix sizes (e.g. 1000 and 3000)
- Set the following environment variables and repeat the tests:

  On KNC:
  ```bash
  export MIC_ENV_PREFIX=MIC
  export MIC_USE_2MB_BUFFERS=16K
  export MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-240:1]
  export KMP_AFFINITY=granularity=fine,compact,1,0
  ```

  On KNL:
  ```bash
  export KMP_AFFINITY=granularity=fine,compact,1,0
  export OMP_NUM_THREADS=256
  ```

Try now to understand the performance numbers observed for the host and AO execution.
MKL-lab2 : FFT with pragma offload and Native Execution

In this example we learn how to offload MKL function calls using offload pragmas.

1. Objectives and learning goals

- To control memory allocation on MIC
- Understand data transferring and data persistence
- Set up the build environment
- Compile the program on the host without modifying the original code
- icc –no-offload –mkl mkl_fft.c –o mkl_fft
- Run the program: ./mkl_fft
- Code to offload, replace the LRZ WORK FOR YOU comments with MKL calls
- Compile the program for offload: icc –mkl mkl_fft.c –o mkl_fft
- Run the program: ./mkl_fft
- Check the performance results

- What about the performance:
- Compile the program for Native execution on KNC:
  - icc –mmic –mkl_fft.c –o mkl_fft.mic
  - add this setting:
    export KMP_AFFINITY=explicit,granularity=fine,proclist=[1-240:1]
    and run again.
- On KNL use: icc –qopenmp –O3 –xMIC-AVX512 –mkl_fft.c –o mkl_fft.mkl

Try now to understand the performance numbers.
MKL Fortran example:

- **Objectives and learning goals:** compile a program for coprocessor only execution; make use of automatic offload; make use of compiler-assisted offload; Also, learn to adjust the affinity settings for Intel OpenMP*, and experiment with large memory pages – an option that is offered by the Linux* µOS on the coprocessor.

1. On the host, cross-compile `getting_started.f90` for coprocessor execution using:
   ```
   ifort -qopenmp -mkl -mmic getting_started.f90 -o get_start-knc
   ```
   On KNL: `ifort -qopenmp -mkl -xMIC-AVX512 getting_started.f90 -o get_start-knl`

1. For KNC copy the generated executable to the coprocessor:
   ```
   scp get_start-knc hostname-mic0:~/
   ```
   Login to the coprocessor and run the program `./get_start-knc`
   If the program reports any missing libraries, copy the necessary files from `$/MKL_BASE/lib/mic` and `$/IFORT_BASE/compiler/lib/mic` on the host to a directory on the coprocessor. Set the `LD_LIBRARY_PATH` environment variable to point to that directory, then rerun the executable. Alternately, you can use micnativeloadex utility.

2. Next compile `getting_started.f90` to use automatic offload. On the host, open `getting_started.f90` and add the line `call mkl_mic_enable()` near the beginning of the `getting_started` function before the execution proceeds to SGEMM or DGEMM. Alternatively, you can set the environment variable `MKL_MIC_ENABLE=1`.
   Compile and execute the program on the host. Some of the work will automatically be offloaded to the coprocessor:
   ```
   ifort -qopenmp -mkl getting_started.f90 -o getting_started
   ./getting_started
   ```

4. Now compile `offload.f90` using the Language Extensions for Offload (LEO) to offload the entire `run` function to the coprocessor. Open `offload.f90` and add a `!dir$ offload` directive before each call to the `run` function. Specify which data is going into the offload section and which is coming out. For example:
   ```
   !dir$ offload target(mic) in(a:length(n))
   ```
   in front of a function copies in the array `a`. Compile and run the program:
   ```
   ifort -qopenmp -mkl offload.f90 -o offload
   ./offload
   ```
   The Intel compiler does not require an option in order to enable compiler-assisted offload. LEO can be disabled even when an offload directive is found, using `-no-offload`.

#### Compare the execution models for Intel Xeon Phi
Vectorisation labs: with SIMD and OpenMP pragmas

Objectives and learning goals

In this example we learn how to vectorise and parallelise regions using SIMD and OpenMP pragmas

- To enable the compiler to generate diagnostic information
- Understand the vectorisation performance
- Understand vectorisation reports
- To control memory allocation on Xeon Phi

1. Lab 1

- Compile the program without modifying the original code.
- Use the –no-vec flag to turn off the vectorisation:
  - $icc –no-vec –qopenmp –mmic nBody.c –o nbody.knc
  - $icc –qopenmp –xMIC-AVX512 nBody.c –o nbody.knl
- run the program with
  - for KNC: $micnativeloadex ./nbody.knc
  - On KNL: ./nbody.knl
and record execution time.
- Add the vector report flags: -qopt-report=5
- Display the optimisation report file “nbody.optrpt” and try to understand the vectorised regions.
- Remove the –no-vec and –qopt-report flags and repeat the execution step above to record the execution time in the end. Check the performance results.
- Display the source code and switch on the parallelisation lines.
- Compile the program only with:
  - $icc –qopenmp –mmic nBody.c –o nbody.knc (KNC)
  - $icc –qopenmp –xMIC-AVX512 nBody.c –o nbody.knl (KNL)
  - and repeat the execution line above.
- Check the performance results.
- Change the environment variable: OMP_NUM_THREADS to: 20, 80,…up to 256 and run again.

What about the performance.
2. Lab 2

- Display nBody.c code and replace the LRZ WORK FOR YOU comments with SIMD and OpenMP calls.
- Display the Makefile to add the flag `-qopt-report=5 -qopt-report-phase:loop,vec` or just compile the code with the vector report flags.
- Display the output reports and try to understand the vectorised regions, and remove unnecessary type conversions.
- Use a faster floating point semantics by adding the flag: `-fp-model fast=2`
- Compile again / run and check the performance results.
- Set the following environment:
  ```
  export MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-236:1]
  export KMP_AFFINITY=granularity=fine,compact,1,0
  ```

Try now to understand the performance numbers observed for the host and native execution. And between the KNL and Standard Xeon.