KNC Vector Architecture

Knights Corner co-processor
An Initial Encounter

24 April 2012
Sverre Jarp, CERN openlab

CERN openlab

Agenda

- What are MIC and KNC ?
- Vector architecture
- Whiteboard example
  - Putting it all together
- Conclusions

What is KNC, anyway?

- "Knights Corner": First production version of Intel's "Many Integrated Cores" architecture
  - Separate PCIe card
  - Many-core processor with its own memory hierarchy
    - 50 – 60 cores; 4 hw threads per core
    - Each core is a simple processor with the addition of a powerful vector unit
    - Cores are interconnected via ring
    - GDDR5 memory

KNC Block Diagram

INTEL CONFIDENTIAL UNDER CNDA

INTEL MIC - 04.2012 | 82

Knights-Core Block Diagram

MEMORY

L2

L1

Vector/SIMD Part (VPU)

8x 16b Vmask

Data Gather/Broadcast

512b

32x 512b Vreg

T0 T1 T2 T3

4 cycles

*

+

Data Swizzle

Scalar Register

Scalar Units

Scalar Part

Copyright© 2011 Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.

KNC software environment

- Special version of Linux running on the card
- Full Intel software tool suite
  - Composers: C/C++ compiler; Fortran compiler
  - Inspector
  - VTUNE Amplifier
  - Math Kernel Library
  - MPI library
  - etc.

Why vectors?

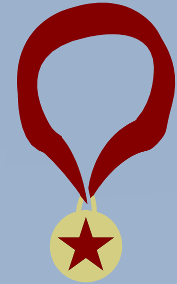
- No new tricks available
  - Performance must mainly come from:
    - Nodes * Sockets(Cards) * Cores * Vectors
- Vectors are energy efficient
  - Marginal power increase when vector size doubles
- Therefore:
  - Design "smart" vector hardware that allows as much work as possible to be done

Why should we care?

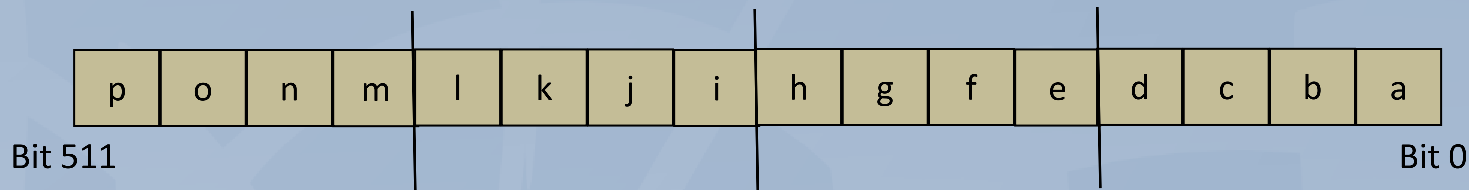
- We still live in a world where, if we require performance, we need to check if the compiler does the right thing for us
  - Recommended actions:
    - Use VTUNE
    - Use "-vec-report", "-opt-report", etc.
    - Use "-S" to generate .s files for manual inspection

The down-side: The ISA manual contains 723 pages !

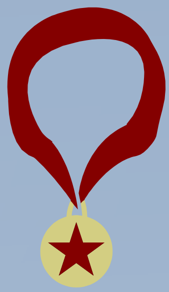
KNC Vector Hardware



- Enhanced register architecture
  - 32 “zmm” registers with 512 bits each
    - Can contain:
      - 16 x 32-bit integer or floats
      - 8 x 64-bit integer or doubles



  - 8 “k” mask registers
    - Contains 16 bits
      - For instance: 0xFFFF, 0xF00F



Smart mask registers

- Mask registers are deployed to control the work:
  - Usage:
    - Produce masks
      - Comparisons
    - Compute with write-masks (or predicates)
      - If mask is off, then the hardware will not
        - » Read the source
        - » Perform the operation
        - » Modify the destination
    - Generate carry/zero flags
      - Arbitrary length arithmetic
    - Flag work in progress
      - Clear corresponding mask when work completed
    - etc.

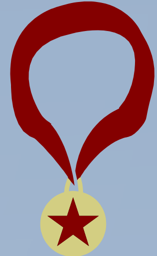
Instruction classes

- Major groupings:
  - Vector arithmetic, logical, shift
  - Vector compare
  - Vector moves (load/store, gather/scatter, etc.)
  - Vector manipulations (blend, broadcast, permute, etc.)
  - Mask operations (logical, moves, etc.)
  - Branch
  - Miscellaneous
- Convention for mnemonics:
  - vxxx(ps|pd): vector FP instructions
  - vpxxx(d|q): vector integer instructions
  - kxxx: mask instruction

Standard instruction format

CERN openlab

- Typically a ternary (3 sources) format:
  - *vop zmm1{k1}, zmm2, convert(zmm3/mem)*
  - Target, same as first source
  - Mask register
    - Predicates updates of target
  - Smart conversion of third source
    - See next slide
  - Third source can be register or memory
    - As in [16+rdi+rax*4]



On-the-fly conversion

- Smart way of manipulating the data on the fly
  - Data conversion:
    - uint8, sint16, uint16, float16, etc.
  - Swizzle (inside 128-bit lanes):
    - {aaaa}, {bbbb}, {cccc}, {dddd} to broadcast a value
    - {cdab} , swap inner pairs
    - {badc}, swap with two-away
    - {dacb}, cross-product
  - Broadcast:
    - {1to8}, {4to8}, {8to8} (default for DP)
    - {1to16}, {4to16}, {16to16} (default for SP)

Data element types

CERN openlab

- 4 native packed formats:

	32-bit	64-bit
Integer	dword	qword
Floating-point	float (float32)	double (float64)

- Conversions
  - FP subset:

	Float32	Float64
Float16	Yes	No
Int32	Yes	Yes
Uint32	Yes	Yes
Float64	Yes	No

Floating-point instructions

CERN openlab

- Extensive collection of
  - add, subtract, multiply (separate)
  - Fused add/subtract and multiply
  - Mathematical functions
    - rcp, rsqrt
    - exp, log
  - min, max
  - scale, round
  - pack, unpack
- Some exotic ones
  - Not dealt with today

Standard FP math instructions

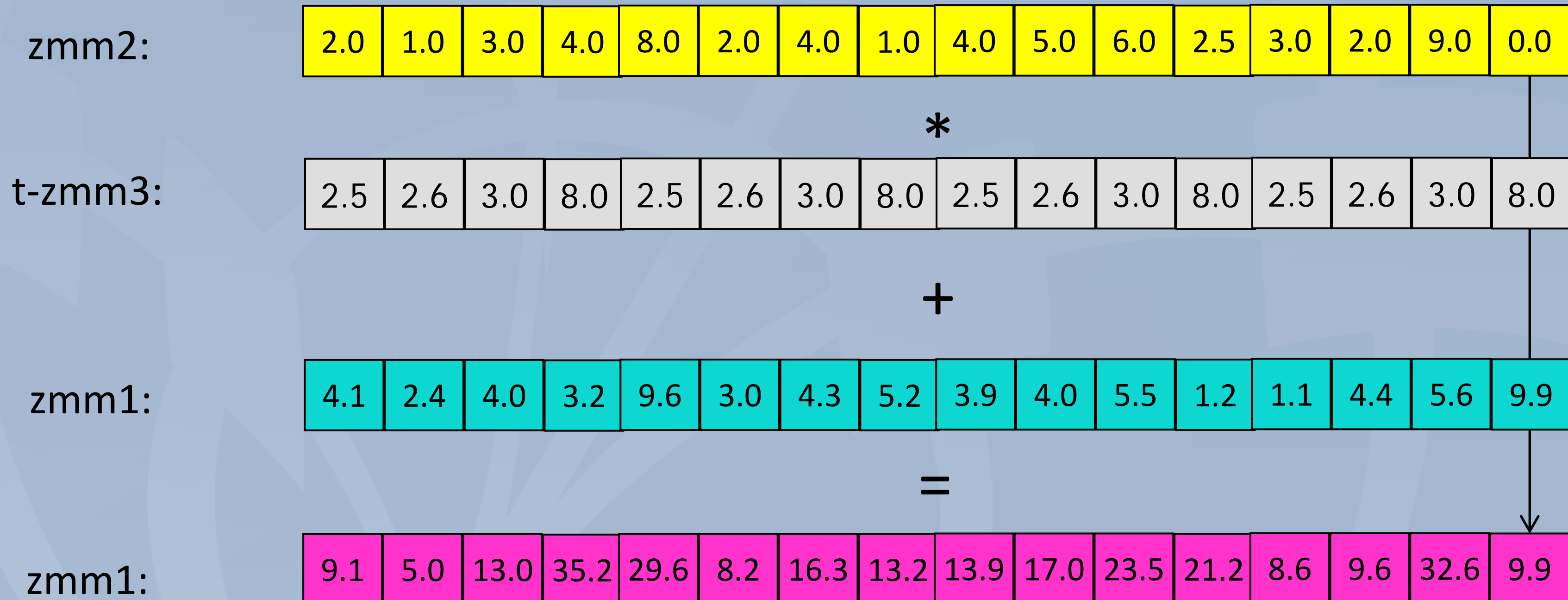
- Standard *vadd, vsub* and *vmul* instructions
  - Packed single, double (ps|pd)
- 24 fused multiply & add/subtract instructions
- Format
  - *v fffff nnn tt*
    - fffff (function): fmadd,fmsub, fnmadd, fnmsub
    - nnn (distribution of source operands): 132, 213, 231
      - Why?
        - » Pre-optimise the register that gets clobbered
        - » Optimise UpLoadConversion
    - tt (type): ps, pd

← White space for readability

Example: vfmsub231ps zmm1{k1}, zmm2, zmm3{4to16}

Multiply-add example

- From previous page:
  - *vfmadd231ps zmm1{k1}, zmm2, zmm3{4to16}*



Now the INT math instructions

CERN openlab

- INT32 (dword):
  - Standard instructions:
    - *vpaddd, vpsubd*
  - Also multiply (high or low):
    - *vpmul(h|l)d*
  - One (std) single multiply & add instruction:
    - *vpmadd231d*
  - Special instructions interacting with masks:
    - *vpadcd* (add with carry)
    - *vpaddsetcd* (add and set mask to carry)
    - *vpaddsetsd* (add and set mask to sign)
    - *vpsbbd* (subtract with borrow)
    - *vpsubsetbd* (subtract and set borrow)

- *Surprisingly little support for INT64 (qword) vectors*
  - *But, do we need it?*

Vector shifts

- Vector shift (logical/arithmetic)
  - Shift amount in immediate value or in vector register
  - Example shown (shift i32 vector left logical):
    - *vpsllvd zmm1{k1}, zmm2,* $S_{i32}(zmm3/m_t)$

zmm2:	5	14	13	12	11	10	5	6	2	5	7	9	14	4	4	4
									<<							
zmm3: **(shift amount)**	2	1	0	3	0	0	3	2	1	0	3	1	0	2	1	0
									=							
zmm1:	20	28	13	96	11	10	40	24	4	5	56	18	14	16	8	4

Vector compare instructions

- Syntax:
  - *vcmp(ps|pd)   k2{k1}, zmm1, convert(zmm2/m_t), imm8*
  - *vpcmpd   k2{k1}, zmm1, convert(zmm2/m_t), imm8*
  - imm8:
    - eq, neq
    - lt, nlt
    - le, nle
    - ord,unord (valid for floating-point only)
  - Results are always stored in a mask register (*k2*)
    - Writemasked via *k1*

Mask instructions

CERN openlab

- *kand*
  - Bitwise logical-and
- *kandn*
  - Bitwise logical-and-not
- *kandnr*
  - And-not reverse
- *knot*
  - Bitwise logical-not
- *kor*
  - Bitwise logical-or
- *kxnor*
  - Bitwise logical-xnor
- *kxor*
  - Bitwise logical-xor

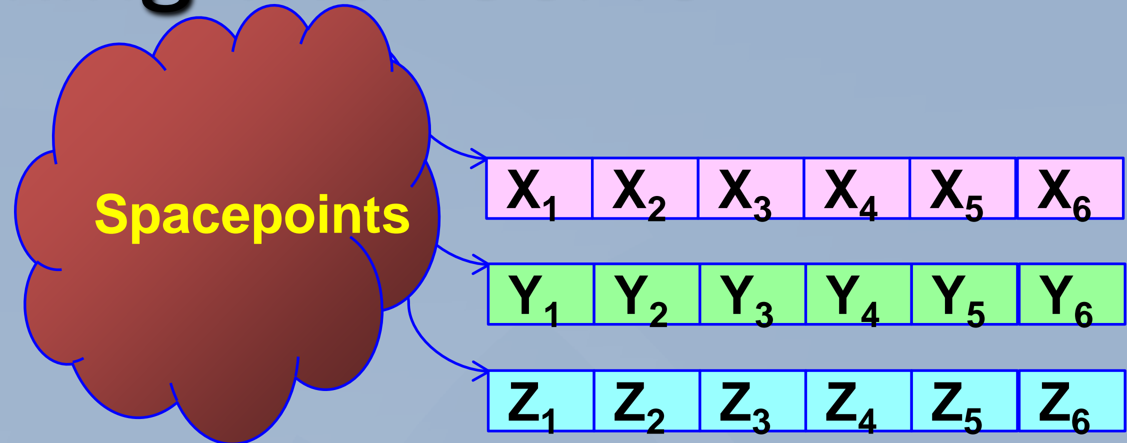
- *kortest*
  - Set ZF if OR results in all '0', CF if all '1'
- *kconcath*
  - Packs two vector masks into r64 (high)
- *kconcatl*
  - Packs two vector masks into r64 (low)
- *kextract*
  - Extracts a vector mask from r64 via imm8
- *kmerge2l1l*
  - Swap and merge low to high byte
- *kmerge2l1h*
  - Swap and merge high byte portion
- *kmov*
  - Move vector masks

Vector Load/Store instructions

- *vgatherd(ps|pd)*
  - Gather floating-point vector
- *vgatherpf0hintd(ps|pd)*
  - Prefetch vector (in gather form)
- *vscatterd(ps|pd)*
  - Scatter vector
- *vscatterpf0hintd(ps|pd)*
  - Prefetch vector (in scatter from)
- *vmova(ps|pd)*
  - Load/store aligned vector
- *vloadunpack*
  - Load unaligned and expand to vector
- *vpackstore*
  - Compress and store unaligned from vector

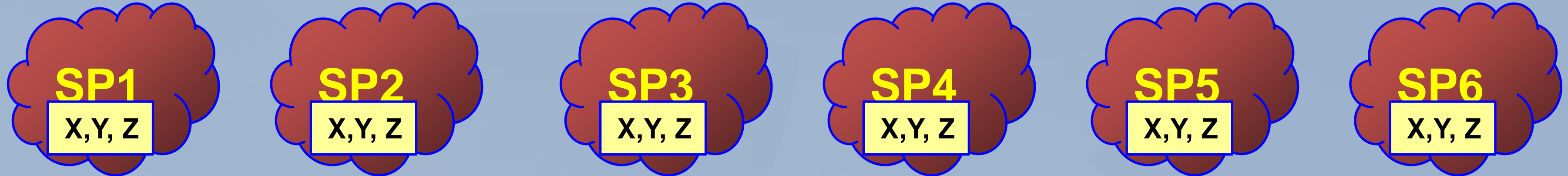
Plus others

Working with SoA's

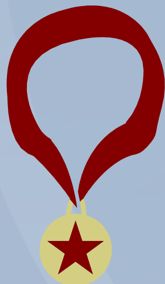


- Typical sequence:
  - Load all data
    - vmovapd zmm1{k1},m_t [Must be: {8to8}]
  - Work
  - Perform tests
  - Mask out irrelevant elements
    - Masking (predication) ensures algorithmic optimization
  - More work
  - Store modified elements
    - vmovapd m_t{k1}, zmm1

Working with AoS's



- Typical sequence:
  - Gather all data
    - `vgatherdpd zmm11{k1}, Noconversion`(mv_t)
  - Work
  - Perform tests
  - Mask out irrelevant elements
  - More work
  - Store modified elements (vscatterdpd)



- Gather and masking (predication) should ensure even better algorithmic optimization

Gather/Scatter loops

- Only one element is guaranteed to complete (when the instruction is executed)
  - Solution: Loop (using mask register for control)

```
..L140: vgatherdps zmm11{k1}, DWORD PTR [12+r15+zmm6*4]
       jkzd      k1, ..L139    # Prob 50%
       vgatherdps zmm11{k1}, DWORD PTR [12+r15+zmm6*4]
       jknzd     k1, ..L140    # Prob 50%
..L139:
```

Conclusions

- MIC ("Knights Corner") pushes performance in several dimensions:
  - Large (double-digit) core count
  - Four threads
  - Long vectors (512b)
- Sophisticated vector instruction set
  - Large number of registers (both data and control)
  - Ternary instructions
  - Smart conversions (on the fly)
  - Native gather/scatter instructions
- Consequently, applications need to expose:
  - Rich regions of:
    - Data and task parallelism

Next goal: Analyse some real loops

Additional Reading and Community

(Extracted from: “An Introduction to Vectorization with the Intel® C++ Compiler”)

- “A Guide to Vectorization with Intel C++ Compilers”, Mario Deilmann, Kiefer Kuah, Martyn Corden, Mark Sabahi, all from Intel.
- “Vectorization with the Intel Compilers (Part 1)”, A.J.C Bik, Intel, Intel Software Network Knowledge base and search the title in the keyword search. This article offers good bibliographical references.
- “The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance”, A.J.C. Bik. Intel Press, June 2004, for a detailed discussion of how to vectorize code using the Intel compiler.
- “Vectorization: Writing C/C++ code in VECTOR Format”, Mukkaysh Srivastav, Computational Research Laboratories (CRL) - Pune, India. Intel Software Network Knowledge base.
- Intel Cilk™ Plus Introductory Information. Overviews, videos, getting started guide, documentation, white papers and a link to the community.
- “Elemental functions: Writing data parallel code in C/C++ using Intel Cilk Plus”, Robert Geva, Intel
- Intel® C++ Composer XE documentation, Includes documentation for the Intel C++ Compiler.
- Intel Software Network, Search for topics such as “Parallel Programming in the “Communities” menu or “Software Forums” or Knowledge Base in the “Forums and Support” menu.
- “Requirements for Vectorizable Loops”, Martyn Corden, Intel
- “The Software Optimization Cookbook. High-Performance Recipes for IA-32 Platforms”, Second Edition, Richard Gerber, Aart J.C. Bik, Kevin B. Smith and Xinmin Tian, Intel Press.