
# PRACE PATC Course: Vectorisation & Basic Performance Overview

LRZ, 27.6.- 29.6.2016

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

# Agenda

- Vectorisation & SIMD Instructions
- IMCI Vector Extension
- Hands-on
- Performance overview on the Intel Xeon Phi

# Why should we care about Vectors ?

- Actually no new tricks available
  - The performance must mainly come from: Nodes, Sockets, Cores and Vectors
- If we require performance we need to know whether the compiler does the right thing for us.
- Keep everyone busy at all times.
- Overlap communication and computation.

# Evolution of Intel Vector Instruction Sets

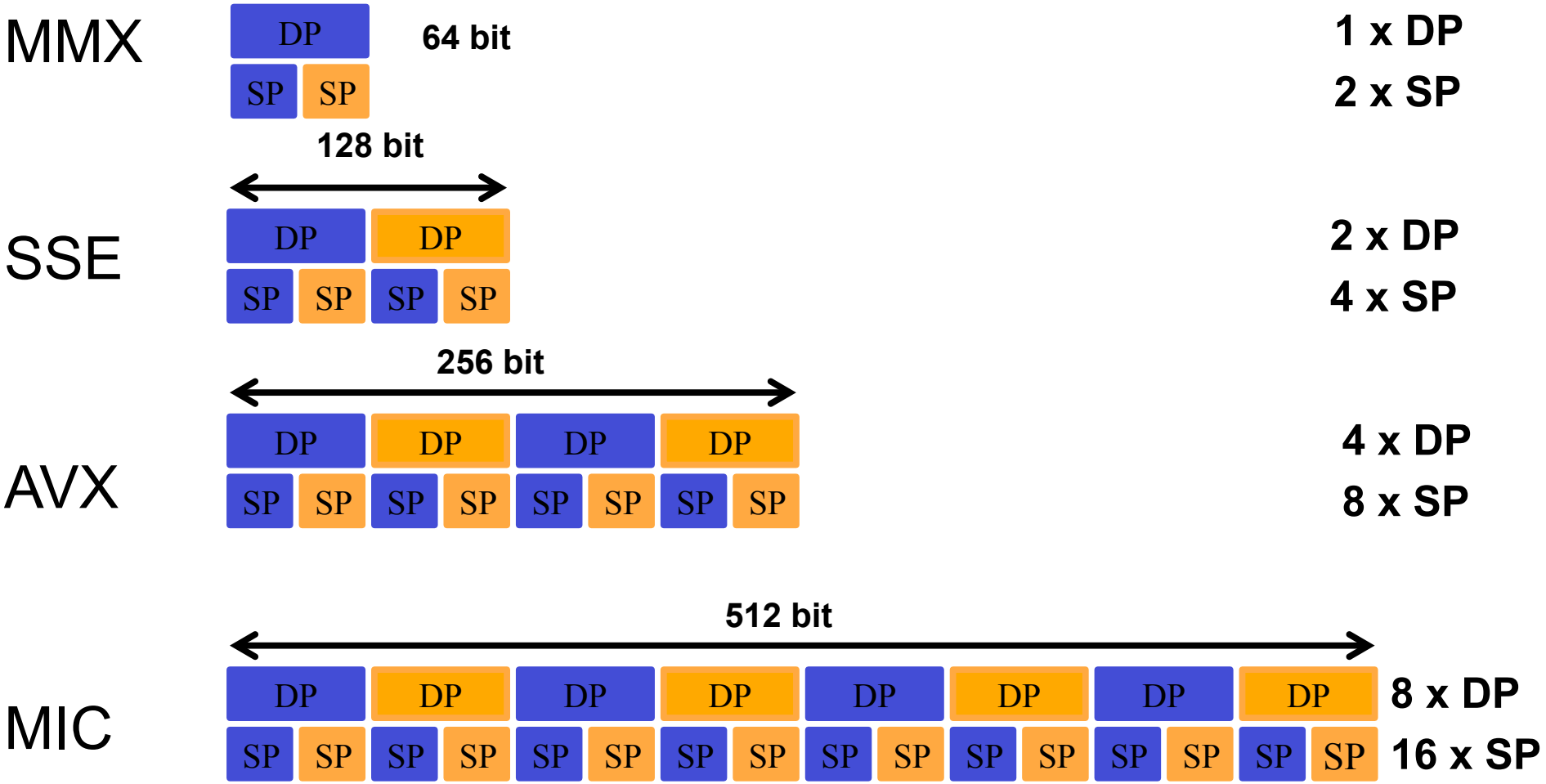
| Instruction Set | Year & Processor | SIMD Width | Data Types |
|---|---|---|---|
| MMX | 1997 Pentium | 64-bit | 8/16/32-bit Int. |
| SSE | 1999 Pentium III | 128-bit | 32-bit SP FP |
| SSE2 | 2001 Pentium 4 | 128-bit | 8-64-bit Int., SP&DP FP |
| SSE3-SSE4.2 | 2004-2009 | 128-bit | Additional instructions |
| AVX | 2011 Sandy-Bridge | 256-bit | SP & DP FP |
| AVX2 | 2013 Haswell | 256-bit | Int. & additional instruct |
| IMCI | 2012 KNC | 512-bit | 32/64-bit Int., SP&DP FP |
| AVX-512 | KNL | 512-bit | 32/64-bit Int. SP&DP FP |

Online guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

# Other Floating-Point Vector

| Manufactures | Instruction Set | Register Width |
|---|---|---|
| IBM | VMX | 4 way SP |
| | SPU | 2 way DP |
| | Double FPU | 2 way DP |
| | Power8 has 64 VSR each | 2 way DP (64bit) or 4 SP(32) |
| Motorola | AltiVec | 4 way SP |
| AMD | 3DNow | 2 way SP |
| | 3DNow Professional | 4 way SP |
| | AMD64 | 2 way DP |
| ARM 64bit | NEON - V7A Cortex-A | 8 SP (16bit) |
| | V6 ARM 11 | Only 32bit (8/16) SP (8bit) |

# Vectorisation / SIMD instruction sets



# Vectorisation is key for performance

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

- Guided vectorisation → using compiler hint and pragmas.

- Low level vectorisation → C/C++ vector classes , Intrinsics / Assembly

# SIMD instruction sets

Scalar Loop

```
for (I = 0; I < n; i++)
    A[i] = A[i] + B[i];
```

SIMD Loop

```
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

# Automatic Vectorisation of Loops

- For C/C++ and Fortran, the compiler detect the loop can be vectorise.
- Enabled using –vec compiler flag (default in –O2 and –O3 optimisation levels) and no source code changes.
- To disable all the autovectorisation use: -no-vec
- How do I know whether a loop was vectorised or not ?
  - use the vector report flags: -qopt-report –qopt-report-phase:vec

~$:more autovec.optrpt

…

*LOOP BEGIN at autovec.cc (14,)*

*Remark #15300: LOOP WAS VECTORIZED [autovec.cc(14,3)]*

*LOOP END*

…..

# Intel specific switches may generate vector extensions

| Functionality | Instructions |
|---|---|
| Optimize for current architecture | -xHOST |
| Generate SSE v1 code | -xSSE1 |
| Generate SSE v2 code | -xSSE2 |
| Generate SSE v3 code (may also emit SSE v1 and SSE v2) | -xSSE3 |
| Generate SSSE v3 code for Atom based processors | -xSSE_ATOM |
| Generate SSSE v3 code (may also emit SSE v1, v2 and SSE v3) | -xSSSE3 |
| Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code) | -xSSE4.1 |
| Generate SSE4.2 code (may also emit (S)SSE v1,v2, v3 and v4 code) | -xSSE4.2 |
| Generate AVX code | -xAVX |
| Generate AVX2 code | -xAVX2 |
| Generate Intel CPUs includes AVX-512 processors code | -xCORE-AVX512 |
| Generate KNL code (and successors) | -xMIC-AVX512 |
| Generate AVX-512 code for newer processors | -axCOMMON-AVX512 |

# Example Automatic Vectorisation

```
double a[width], b[width];
 for (int i = 0; i < width; i++)
                    a[i]+ = b[i];
```

This loop will be automatically vectorised

# Loop that is not Automatic Vectorisable

- Loops with irregular memory access pattern.
- Calculation with vector dependencies.
- Anything that can not be or very difficult to vectorise.

# Example of a Loop that is not Vectorisable

```
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 -vec-report2 two_exits.cpp

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

# Automatic Vectorisation & Data Dependencies

- SIMD instructions operates on several data elements at once. Vectorisation is only possible if  this change of order does not change the results of the calculation.

- The easiest case is when data element that are stored do not appear in any other interaction of the individual loop. In this case the loop will be safely executed using any parallel method, including vectorisation.

# Vectorisation on Xeon Phi

Vectorisation: Most important to get performance on Xeon Phi.

- Use Intel options –vec-report=n (where n=0..7) to show information about vectorisation.
- Use Intel option -guide-vec to get tips on improvement.
- Help if possible the compiler with Intel pragmas.

# 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 Avec=_mm_load_ps(A+i);
__m128 Bvec=_mm_load_ps(B+i);
Avec=_mm_add_ps(Avec, Bvec);
_mm_store_ps(A+i, Avec);
}
```

IMCI Intrinsics

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

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 implement different instruction set with more than 200 new instructions compared to those implemented on the standard Xeon.

# Data alignment

- Memory address should be a multiple of the vector register width in bytes, i.e.
  - SSE2: 16-Byte alignment
  - AVX: 32-Byte alignment
  - MIC: 64-Byte alignment

# What you need to know about SIMD ?

- SIMD pragma used to guide the compiler to victorise more loops.
- Without explicit vectorlength() and vectorlengthfor() clauses, compiler will choose a vectorlengh using its own cost model.
- Vectorisation improve energy efficient
- Registers are getting wider now, factors to consider in order to improve the memory locality in data accesses:

  Data arrangement: gather, scatter,..

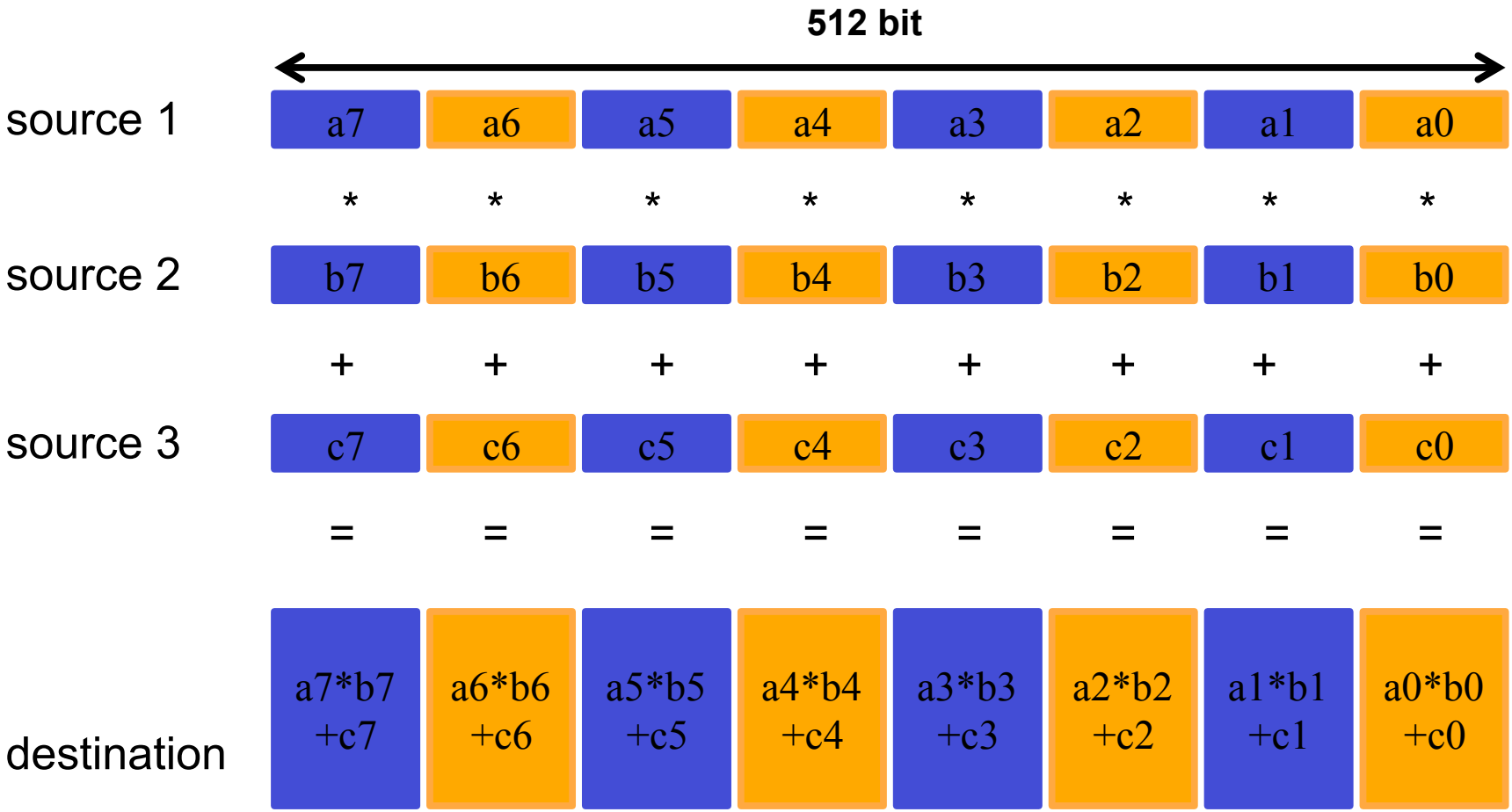
  Memory alignment: avoid cache-to-register

  Caches: maybe non-coherent

  Prefetching: MIC needs extra user intervention, (Intel compiler automaticaly prefetches data for vectorized loops)

# SIMD Fused Multiply Add (FMA)

Instruction set to perform fused multiply-add operations.






https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf

allalen@lrz.de

# Intel SIMD

#pragma SIMD

Force SIMD operation on loops

Array notation ; data arrangement appropriate for SIMD

| | |
|---|---|
| a[index:count] | start at index with count element |
| | (also: index:count:stride) |
| a[i:n]=b[i-1:n]+b[i+1] | (think single line SIMD) |
| e[:]=f[:]+g[:] | (entire array, heap or stack) |
| r[:]=s[i[:]], r[i[:]]=s[:] | (gather, scatter) |
| func(a[:]) | (scalar/simd-enabled=by element/SIMD) |
| if(5==a[:]) result[:]=0 | (works with conditionals) |

SIMD Enabled functions: element → vector function

# Example with: #pragma simd

```
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];
  }
}
```

Function uses too many unknown pointers for the compiler's automatic runtime independence. check optimisation to kick -in

# Intel-Specific Vectorisation Pragmas

- #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 Single Instruction Multiple Data (SIMD) instructions. (up to OpenMP 4.0)

# #pragma vector

| | |
|---|---|
| **always** | 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 |
| **aligned** | instructs the compiler to use aligned data movement instructions for all array references when vectorising |
| **unaligned** | instructs the compiler to use unaligned data movement instructions for all array references when vectorizing |
| **nontemporal** | 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.<br>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 |
| **temporal** | directs the compiler to use temporal (that is, non-streaming) stores on systems based on all supported architectures, unless otherwise specified |
| **vecremainder** | instructs the compiler to vectorise the remainder loop when the original loop is vectorised |
| **novecremainder** | instructs the compiler not to vectorise the remainder loop when the original loop is vectorised |

# Example for vectorisation pragmas

```
#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];
      }}}}
```

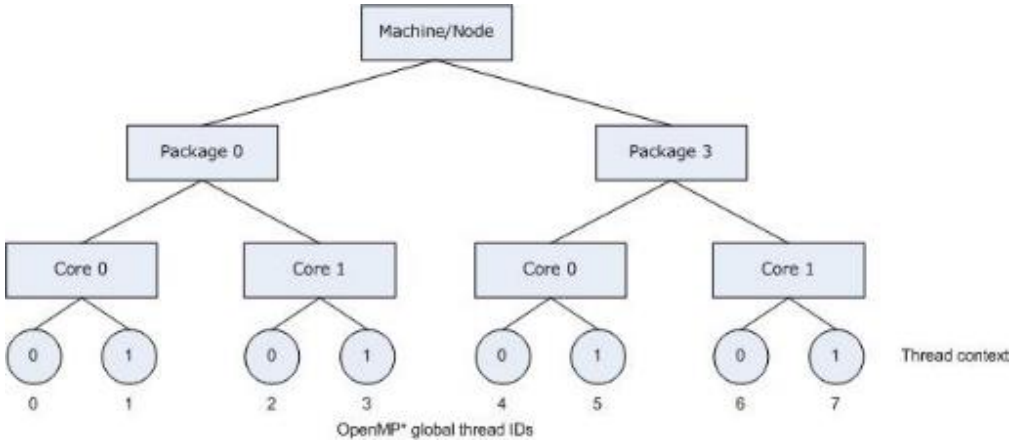
# Thread Affinity

- Pinning threads is important!

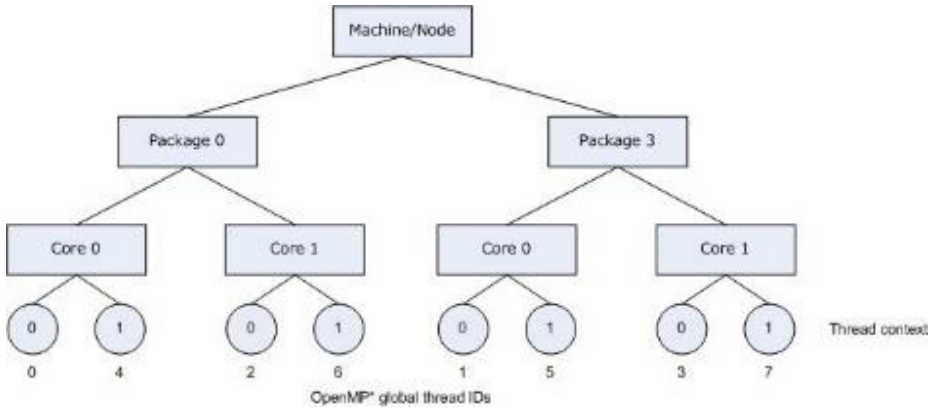
  *~$ export KMP_AFFINITY="granularity=thread,x"*

  *x=compact, scatter, balanced*

  "See Intel compiler documentation for more information".




~$export KMP_AFFINITY=granularity=thread,compact.




~$export KMP_AFFINITY=granularity=thread,scatter.

# Lab 3:
# nbody problem


allalen@lrz.de

# 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

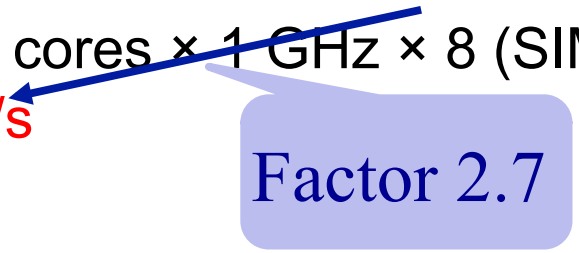
# Performance overview on Xeon Phi


allalen@lrz.de

# Performance Comparison

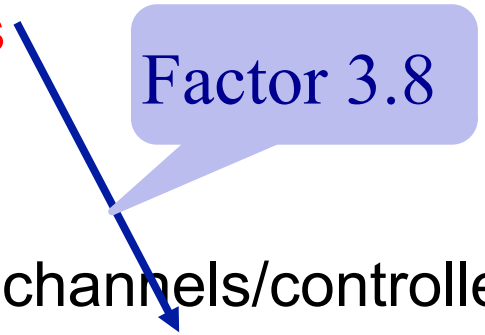
- Sandy-Bridge-EP: 2 sockets ×8 cores @ 2.7 GHz.
- Xeon Phi: 60 cores @1.0 GHz.

- **DP FLOP/s:**
  - Sandy-Bridge: 2 sockets × 8 cores × 2.7 GHz × 4 (SIMD) × 2 (ALUs) = 345.6 GFLOP/s
  - Xeon Phi: 60 cores × 1 GHz × 8 (SIMD) × 2 (FMA)= 960 GFLOP/s



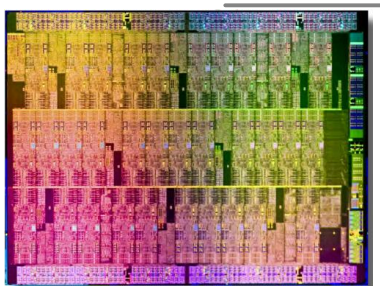
# Memory Bandwidth Comparison

- Sandy-Bridge:
  2 sockets × 4 memory channels × 6.4 GT/s × 2 bytes per channel = 102.4 GB/s




- Xeon Phi:
  8 memory controllers × 2 channels/controller × 6.0 GT/s ×4 bytes per channel = 384 GB/s.
- For complicated memory access patterns: memory latency / cache performance is important. Xeon Phi caches less powerful then Xeon caches (e.g. no L1 prefetcher etc.) .

# Comparing Accelerators



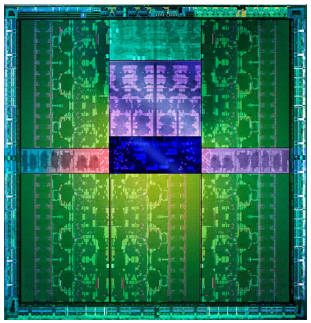
## Intel Xeon Phi

- 60 cores each with 512 Bit **SIMD** FMA→ 480/960 SIMD DP/SP tracks
- ~3 B Transistor count (22 nm)
- ~1 GHz clock speed
- ~250 W power consumption
- ~1,2 TF/s Peak performance (DP)

- 250 GB/s (GDDR5) Memory bandwidth

- 60 – 240 threads to execute
- Fortran, C/C+, MPI/OpenMP, **SIMD**, etc. Programming languages

## NVIDIA GPU: K20

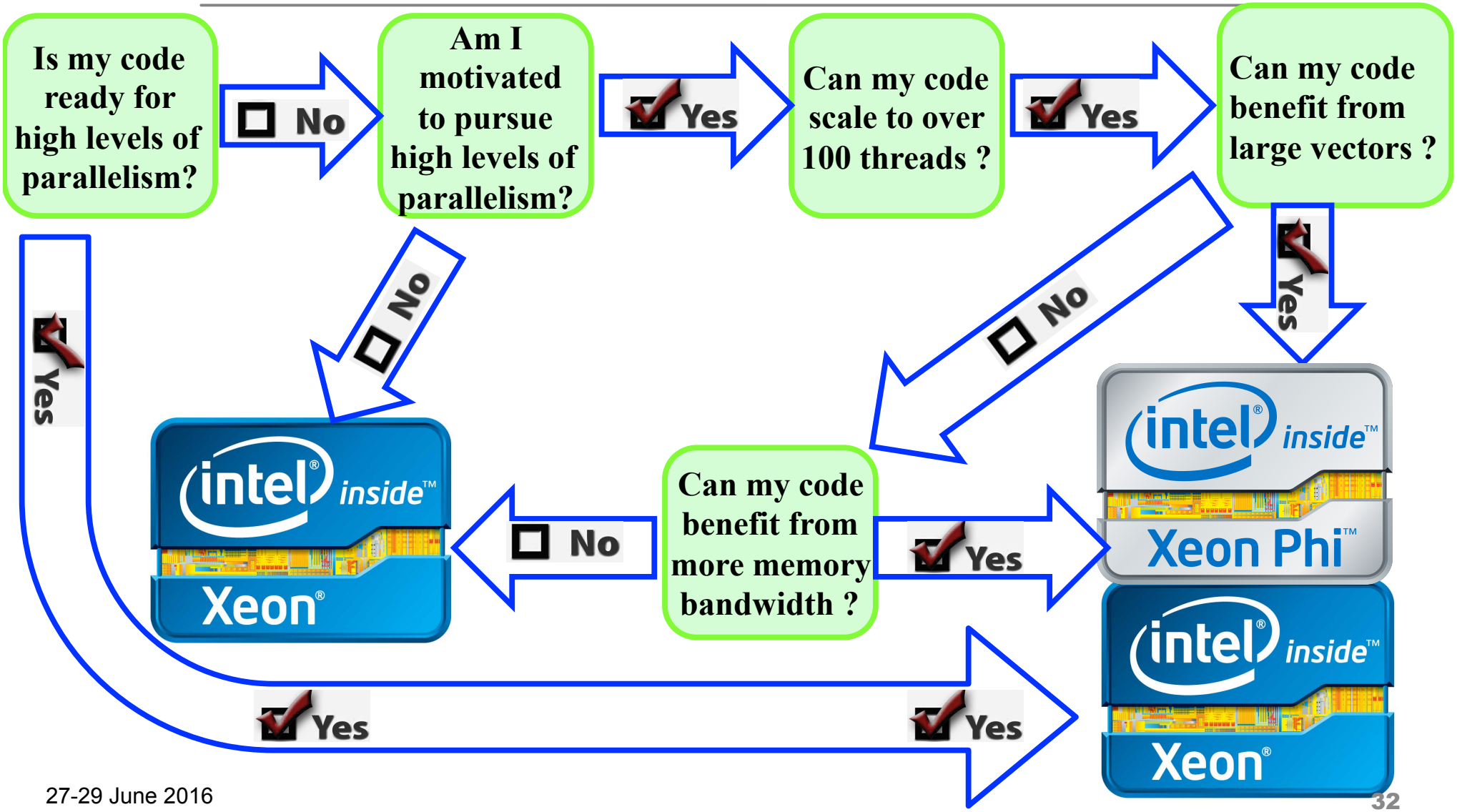


- 15 Streaming Mutiprocessors (SMX) units each:192 cores
- 960/2880 DP/SP "cores"
- 7.1 B Transistor count (28 nm)
- ~700 MHz clock speed
- ~250 W power consumption
- ~1.3 TF/s Peak performance (DP)

- 250 GB/s (GDDR5) Memory bandwidth

- > 10,000 Threads to execute
- CUDA, OpenACC and OpenCL Programming languages

# Is Intel Xeon Phi the Right Coprocessor for my Code ?



# To Achieve a Good Performance on Xeon Phi

- Data should be aligned to 64 Bytes (512 Bits) for the MIC architecture, in contrast to 32 Bytes (256 Bits) for AVX and 16 Bytes (128 Bits) for SSE.
- Vectorisation is very important for the MIC architecture due to the large SIMD width of 64 Bytes
- Use the new instructions like gather/scatter, FMA..etc. which allow more loops to be parallelised on the MIC than on an Intel Xeon based host.
- Use pragmas like #pragma ivdep, #pragma vector always, #pragma vector aligned, #pragma simd etc. to achieve autovectorisation.
- Autovectorisation is enabled at default optimisation level -O2 .
- Let the compiler generate vectorisation reports using the compiler option -vecreport2 to see if loops were vectorised for MIC (Message "*MIC* Loop was vectorised" etc).
- The options -opt-report-phase hlo (High Level Optimiser Report) or -opt-report-phase ipo_inl (Inter procedural Optimiser, inline expansion of functions) may also be useful.
- Explicit vector programming is also possible via the new **SIMD** constructs from OpenMP 4.1

# Thank you.