C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators Power-Aware Computing – Performance of Applications Kernels

hyPACK-2013

Mode 3 : Intel Xeon Phi Coprocessors

Lecture Topic : Intel Xeon-Phi Coprocessor Compilation & Vectorization – An Overview

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

Intel Xeon-Phi – Vectorization

<u>**Lecture Outline**</u>

Following topics will be discussed

❖ Understanding of Intel Xeon-Phi Architecture

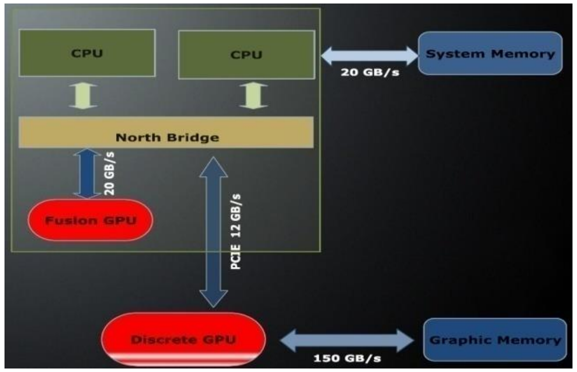
❖ Programming on Intel Xeon-Phi : Compilation and Vectorization

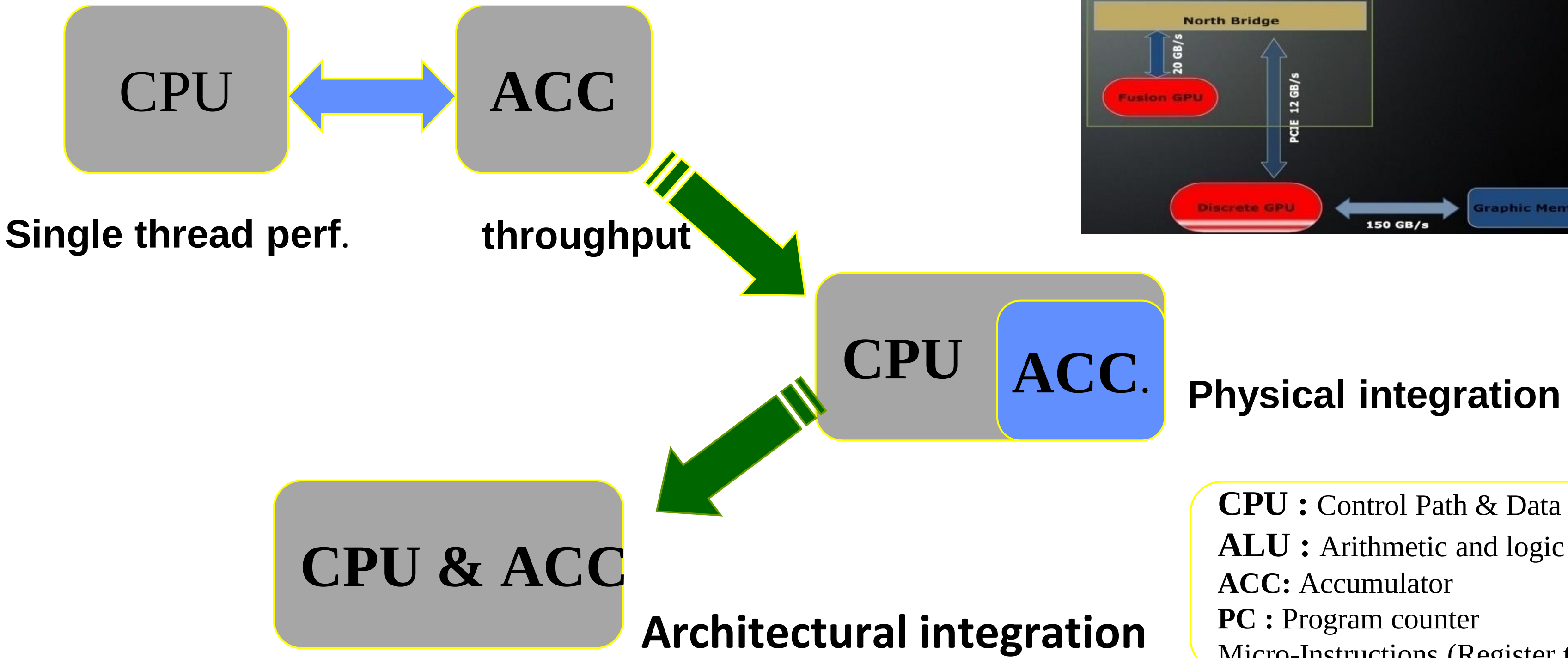
❖ Tuning & Performance

Source : References & Intel Xeon-Phi; http://www.intel.com/

Systems with Accelerators

A set (one or more) of very simple execution units that can perform few operations (with respect to standard CPU) with very high efficiency. When combined with full featured CPU (CISC or RISC) can accelerate the “nominal” speed of a system.



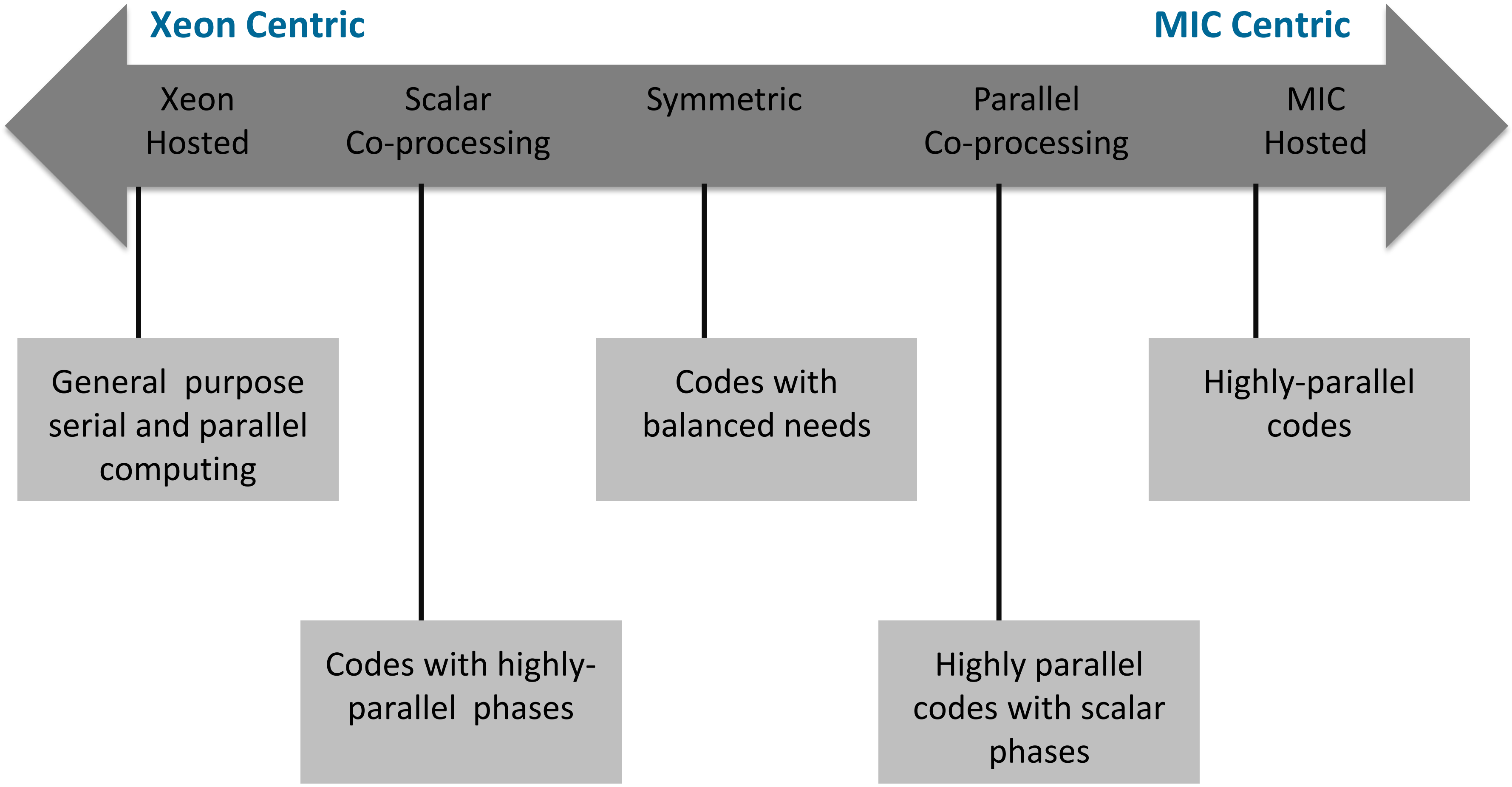


CPU : Control Path & Data Path
ALU : Arithmetic and logic units
ACC: Accumulator
PC : Program counter
Micro-Instructions (Register transfer)

Source : NVIDIA, AMD, SGI, Intel, IBM Alter, Xilinux References

MIC Architecture, System Overview

Compute modes vision

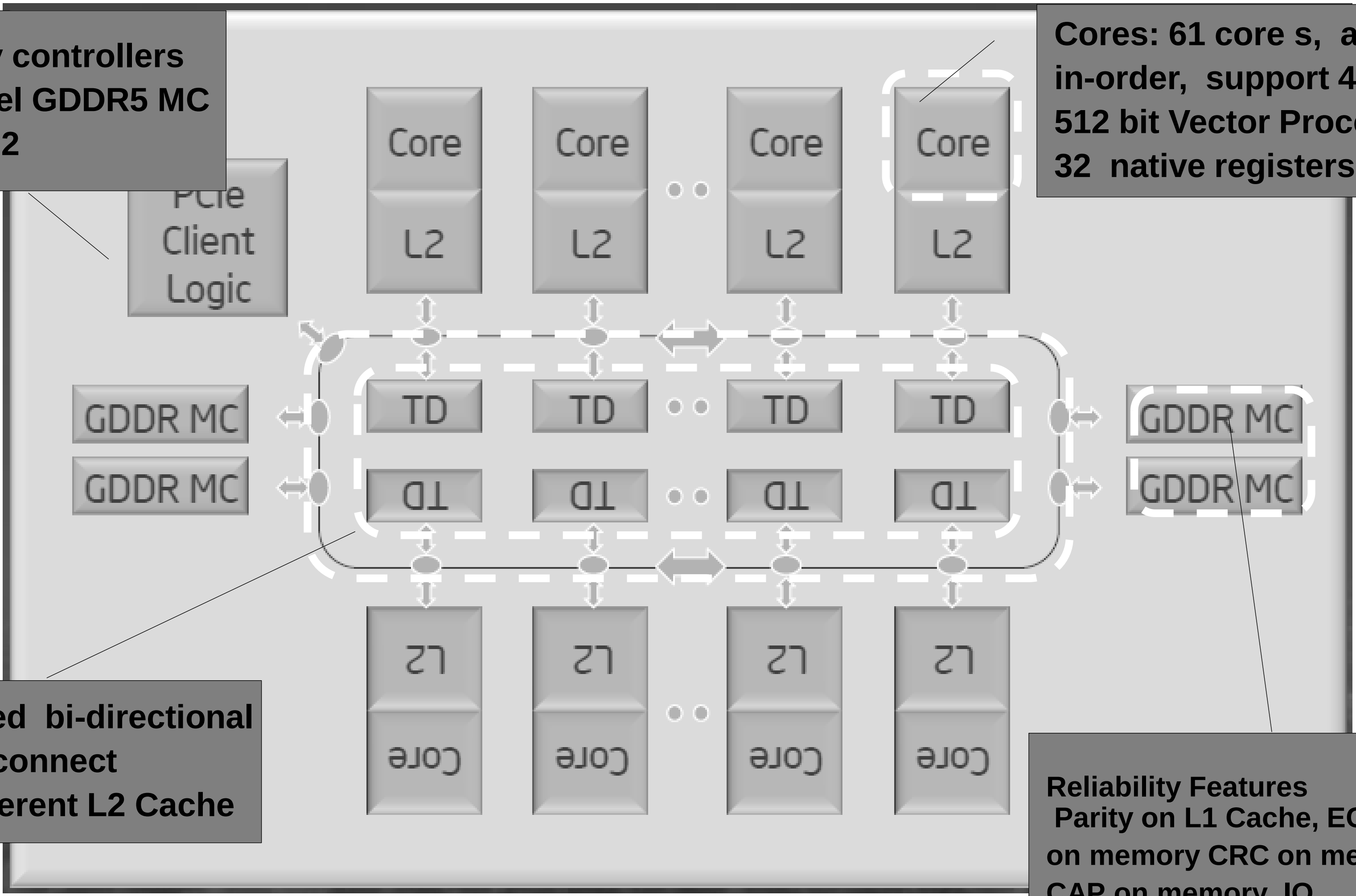

Xeon Centric
MIC Centric
Xeon Hosted
Scalar Co-processing
Symmetric
Parallel Co-processing
MIC Hosted
General purpose serial and parallel computing
Codes with highly-parallel phases
Codes with balanced needs
Highly parallel codes with scalar phases
Highly-parallel codes

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel® Xeon Phi™ Architecture Overview

8 memory controllers
16 Channel GDDR5 MC
PCIe GEN2

Cores: 61 core s, at 1.1 GHz
in-order, support 4 threads
512 bit Vector Processing Unit
32 native registers

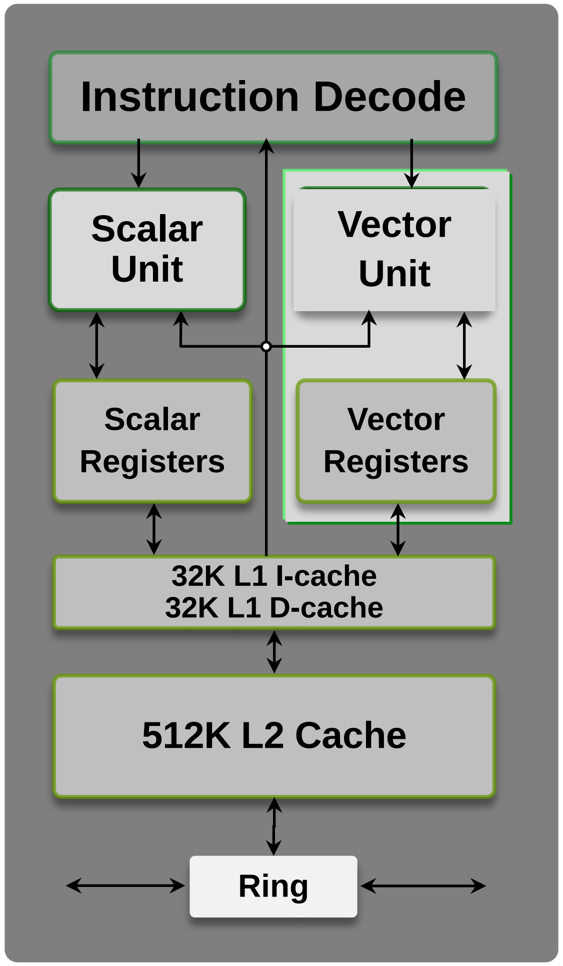


High-speed bi-directional
ring interconnect
Fully Coherent L2 Cache

Reliability Features
Parity on L1 Cache, ECC
on memory CRC on memory IO,
CAP on memory IO

Source : References & Intel Xeon-Phi; http://www.intel.com/

Core Architecture Overview



- 60+ in-order, low power IA cores in a ring interconnect
- Two pipelines
  - Scalar Unit based on Pentium® processors
  - Dual issue with scalar instructions
  - Pipelined one-per-clock scalar throughput
- SIMD Vector Processing Engine
- 4 hardware threads per core
  - 4 clock latency, hidden by round-robin scheduling of threads
  - Cannot issue back to back inst in same thread
- Coherent 512KB L2 Cache per core

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon-Phi Compilation – Vectorization Performance Issues

– Vectorization

What is meant by Vectorization ?

Vectorization is the process of converting an algorithm from a **scalar** implementation to a **vector process**.

Scalar : an operation one pair of operands at a time

Vector : A process in which a single instruction can refer to a vector (series of adjacent values)

- ➢ it adds a form of parallelism to software in which one instruction or operation is applied to multiple pieces of data.
- ➢ Efficient Processing of Data Movement is required to get improvement in performance.

What is meant by Vectorization ?

- ❖ Many general-purpose microprocessors support SIMD (single-instruction-multiple-data) parallelism

- ❖ When the hardware is coupled with C/ C++ compilers that support it, developers have an easier time delivering more efficient, better performing software

- ❖ Types of Vector Computations in Applications
  - ➢ Multi-media Applications
  - ➢ Scientific and Engineering Applications
  - ➢ Graphic Computations
  - ➢ Computational Finance
  - ➢ Information Science Applications

What is meant by Vectorization ?

Compilers :

- ❖ Performance or efficiency benefits from **vectorization** depend on the code structure.

- ❖ Automatic & near automatic techniques (**Auto-Vectorization feature**) introduced below are most productive in delivering improved performance or efficiency.
- ❖ **SIMD** Support
  - ➢Intel C++ Compilers
  - ➢Intel Fortran 90 Compilers
  - ➢Compliers supporting SIMD Instructions

- ❖ Intel Compilers supporting the **Intel Streaming SIMD Extensions** (**Intel SSE**) & Intel Advanced Vector Extensions (**Intel AVX**) on both IA-32 and Intel 64 processors.

Source : References & Intel Xeon-Phi; http://www.intel.com/

What is meant by Vectorization ?

Compilers :

- ❖ **Auto-vectorization :** Performance or efficiency benefits from **vectorization** depend on the both compilers do auto-vectorization, generating Intel SIMD code to automatically vectorize parts of application software when certain conditions are met.

- ❖ **Portability Problems :** Because no source code changes are required to use auto-vectorization, there is no impact on the portability of your application.

- ❖ To take advantage of auto-vectorization, applications must be built at default optimization settings (-O2) or higher. No additional or special switch setting is needed using packed SIMD instructions

Source : References & Intel Xeon-Phi; http://www.intel.com/

What is meant by Vectorization ?

Advantage of Intel MKL and Intel IPP

- ➢ Intel Math Kernel Library (**MKL**)
- ➢ Intel® Integrated Performance Primitives (**IPP**) is another library for C and C++ developers,

- ❖ Another easy way to take advantage of vectorization is to make calls in your applications to the vectorized forms of functions in the Intel® Math Kernel Library. Much of Intel MKL is threaded and supports auto-vectorization to help you get the most of today's multi-core processors. **Intel MKL** functions are also fully thread-safe, so multiple calls for different threads will not conflict with one another.
- ❖ **Intel IPP** offers libraries that can be called for multimedia, data processing, and communications applications

Source : References & Intel Xeon-Phi; http://www.intel.com/

About Vectorization

❖ Whenever possible, instructions on data arrays are processed in an assembly line manner, where several pieces of data are undergoing different parts of an operation simultaneously

Vector Registers

❖ The vector computers get most of their speed through vector operations. This means that a single type of instruction on multiple data. This is uniquely accomplished through the use of vector registers.

Vector Chaining

❖ Vector chaining is a way to decrease vector start-up time. On the C90 a functional unit can begin processing data as soon as the first elements are in the registers.

About Vectorization

- **About Vectorization :**
  - High performance is dependent on the vectorization of long loops. Poor performance can result from the inhibition of this vectorization.
- **Types of Computation in Applications**
  - Loop Not Innermost
  - Vector Dependencies
  - Other Not Vectorizable Constructs
  - Memory Conflicts
  - I/O Optimization

Source : References & Intel Xeon-Phi; http://www.intel.com/

About Vectorization

Loop No nnermost

Problem

- ❖ Only innermost loop can be vectorized at the machine instruction level. However, it may be more efficient to vectorize the operations in the outer loops instead. This could be the case if:
  - ➢ The inner loop is inhibited from vectorization
  - ➢ The outer loop has a longer vector length than the inner loop
  - ➢ The outer loop does more work than the inner loop

Solution

- ❖ The solution is to make the outer loop innermost. Depending on the structure of the loops, there are three ways to do this:
  - ➢ Swap the loops
  - ➢ Split the outer loop
  - ➢ Unwind the inner loop

About Vectorization

Vector Dependencies

Problem : Dependencies occur when each iteration of a loop is dependent on the result of previous iterations.

❖ There are three kinds of dependency:

(1) Result not ready (recurrence or recursion)

(2) Value destroyed (3) Ambiguous subscript

Solution :

Result Not Ready

❖ The solution is to restructure the loop to remove the dependency. Sometimes, this is difficult and requires rethinking the algorithm. Often, however, you can do it by:

➢ Swapping loops **OR** Splitting the dependent work out of the loop

Value Destroyed

❖ You generally do not have to worry about this kind of dependency. The compiler handles it by saving the values in a temporary array.

Ambiguous Subscript

❖ The solution is to use an IVDEP directive to tell the compiler that there is not dependency (if that is in fact the case!)

Other Non-vectorizable Constructs

Problem

❖ There are a number of other constructs that prevent vectorization. These include:

- ➢ I/O statements (These generate calls no library subroutines)
- ➢ CHARACTER data and functions
- ➢ STOP and PAUSE
- ➢ Assigned GOTO (obsolete, anyway)

Solution

❖ The only solution is to move these constructs out of the loop, either by splitting or by recording so that the constructs are unnecessary

Source : References & Intel Xeon-Phi; http://www.intel.com/

About Vectorization

Typical Vector Computer Features

- ❖ Fast Clock Speed.
- ❖ Segmented, Vector Functional Units
- ❖ Independent Functional Units
- ❖ Register-to-Register Operations
- ❖ Shared, Banked Memory
- ❖ No Virtual Memory Fast I/O

Source : References & Intel Xeon-Phi; http://www.intel.com/

Vectorization and SIMD Execution

- SIMD
  - Flynn’s Taxonomy: Single Instruction, Multiple Data
  - CPU perform the same operation on multiple data elements
- SISD
  - Single Instruction, Single Data
- Vectorization
  - In the context of Intel® Architecture Processors, the process of transforming a scalar operation (SISD), that acts on a single data element to the vector operation that that act on multiple data elements at once(SIMD).
  - Assuming that setup code does not tip the balance, this can result in more compact and efficient generated code
  - For loops in ”normal” or ”unvectorized” code, each assembly instruction deals with the data from only a single loop iteration

Source : References & Intel Xeon-Phi; http://www.intel.com/

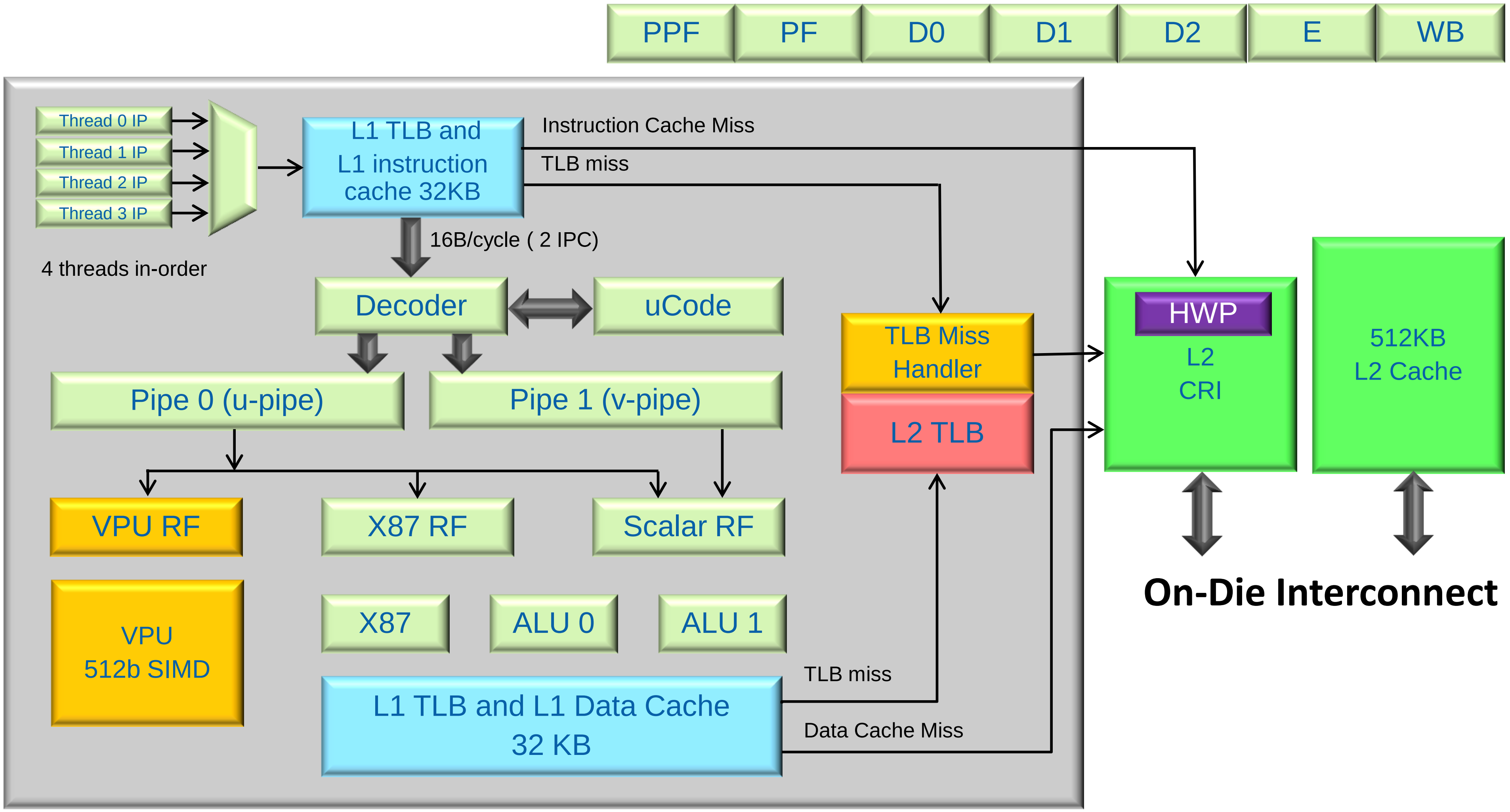
Intel Xeon Phi : Vector Unit

Understand floating point arithmetic Unit

- Vector Processing Unit executing vector FP instruction
- X87 unit also exist can execute FP Instruction as well
- Compiler choose which place to use for FP operation
- VPU is preferred place because of its speed
  - VPU can make the FP results reproducible as well
- Use X87 should be used for two reasons
  - Reproduce the same results 15 years ago, right or wrong
  - Need generate FP exceptions for debugging purpose
- Intel Compiler default to VPU the user can override with
  `–fp-model strict`
- Vectorized, high precision of division, square root and transcendental functions from libsvml
  `-fp-model-precise –no-prec-div –no-prec-sqrt –fast-transcendentals –fimf-precision=high`

Source : References & Intel Xeon-Phi; http://www.intel.com/

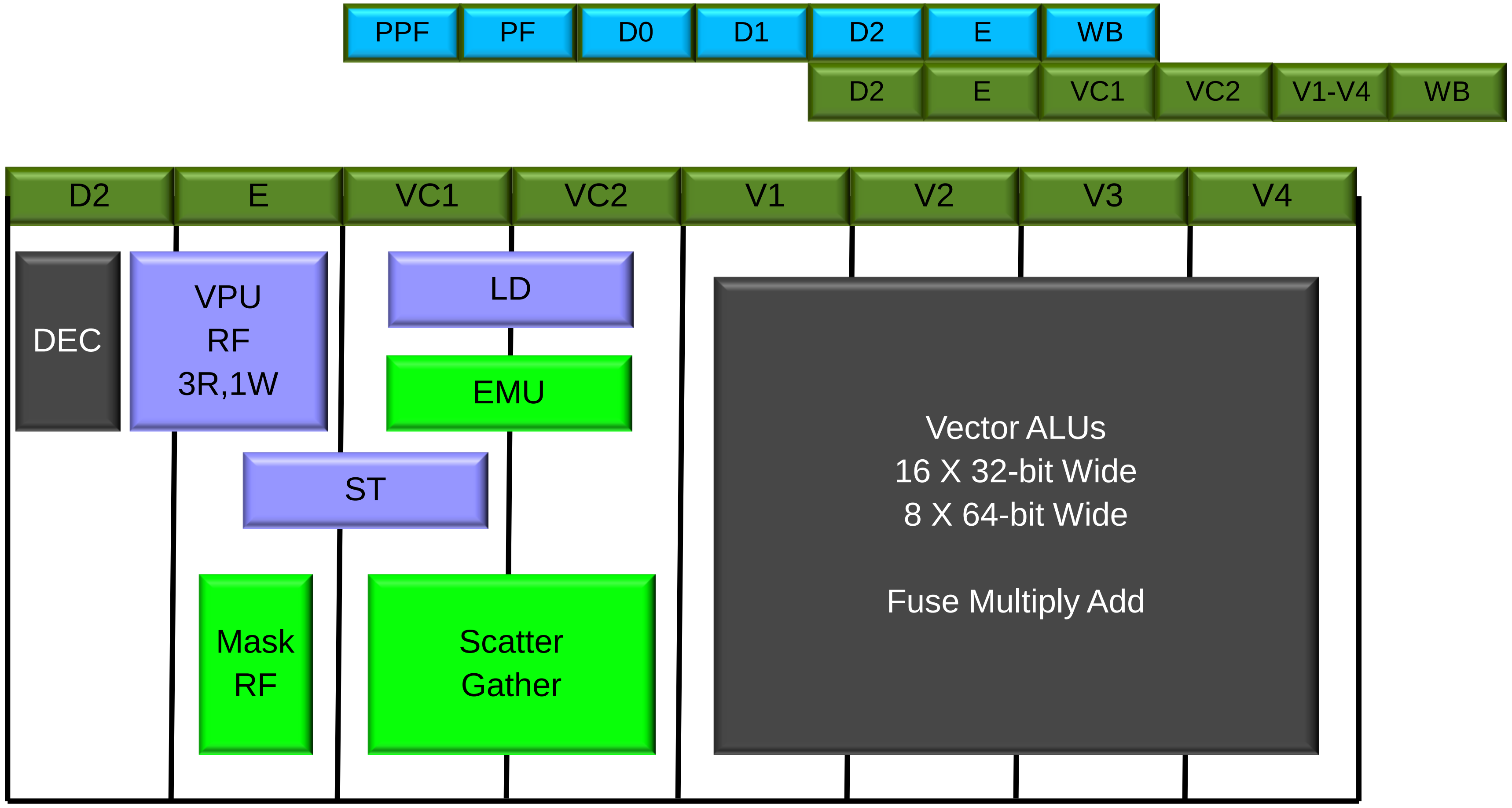
Vector Processing Unit Extends the Scalar IA Core



On-Die Interconnect

Source : References & Intel Xeon-Phi; http://www.intel.com/

Core extension Vector Processing Unit



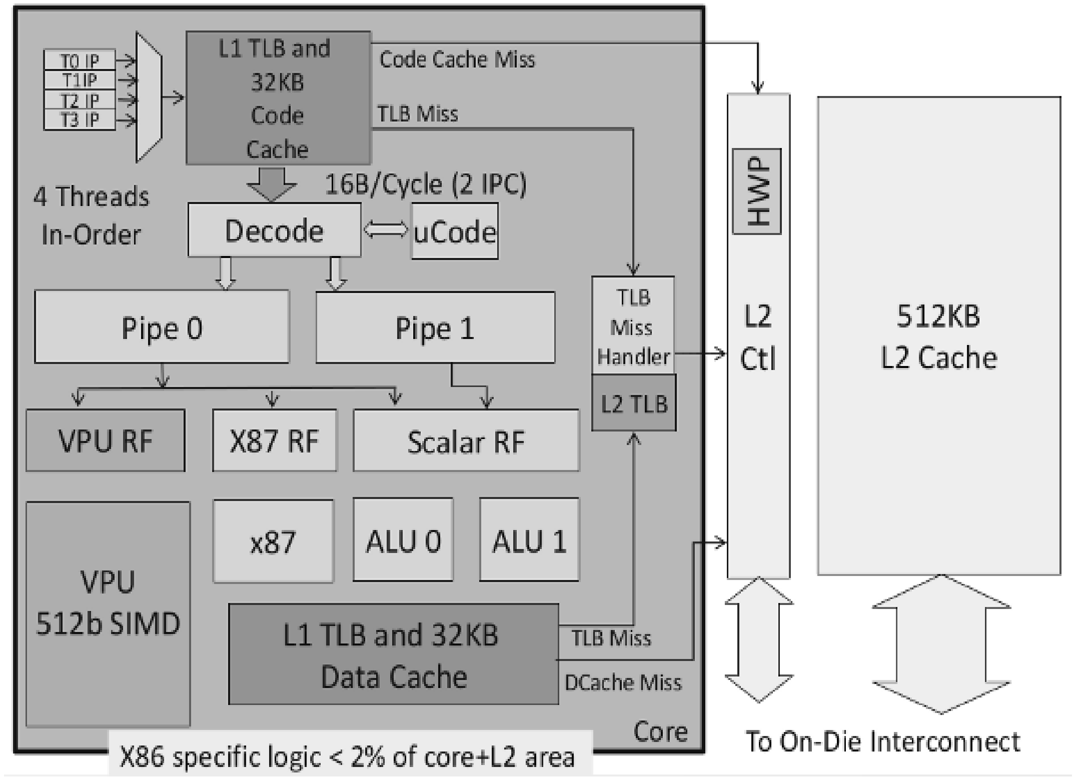
Source : References & Intel Xeon-Phi; http://www.intel.com/

Vector Processing Unit and Intel® IMCI

- Vector Processing Unit Execute Intel® IMCI
  - Intel® Initial Many Core Instructions
- 512-bit Vector Execution Engine
  - 16 lanes of 32-bit single precision and integer operations
  - 8 lanes of 64-bit double precision and integer operations
  - 32 512-bit general purpose vector registers in 4 thread
  - 8 16-bit mask registers in 4 thread for predicated execution
- Read/Write
  - One vector length (512-bits) per cycle from/to Vector Registers
  - One operand can be from the memory free
- IEEE 754 Standard Compliance
  - 4 rounding Model, even, 0, +∞, -∞
  - Hardware support for SP/DP denormal handling
  - Sets status register VXCSR flags but not hardware traps

Source : References & Intel Xeon-Phi; http://www.intel.com/

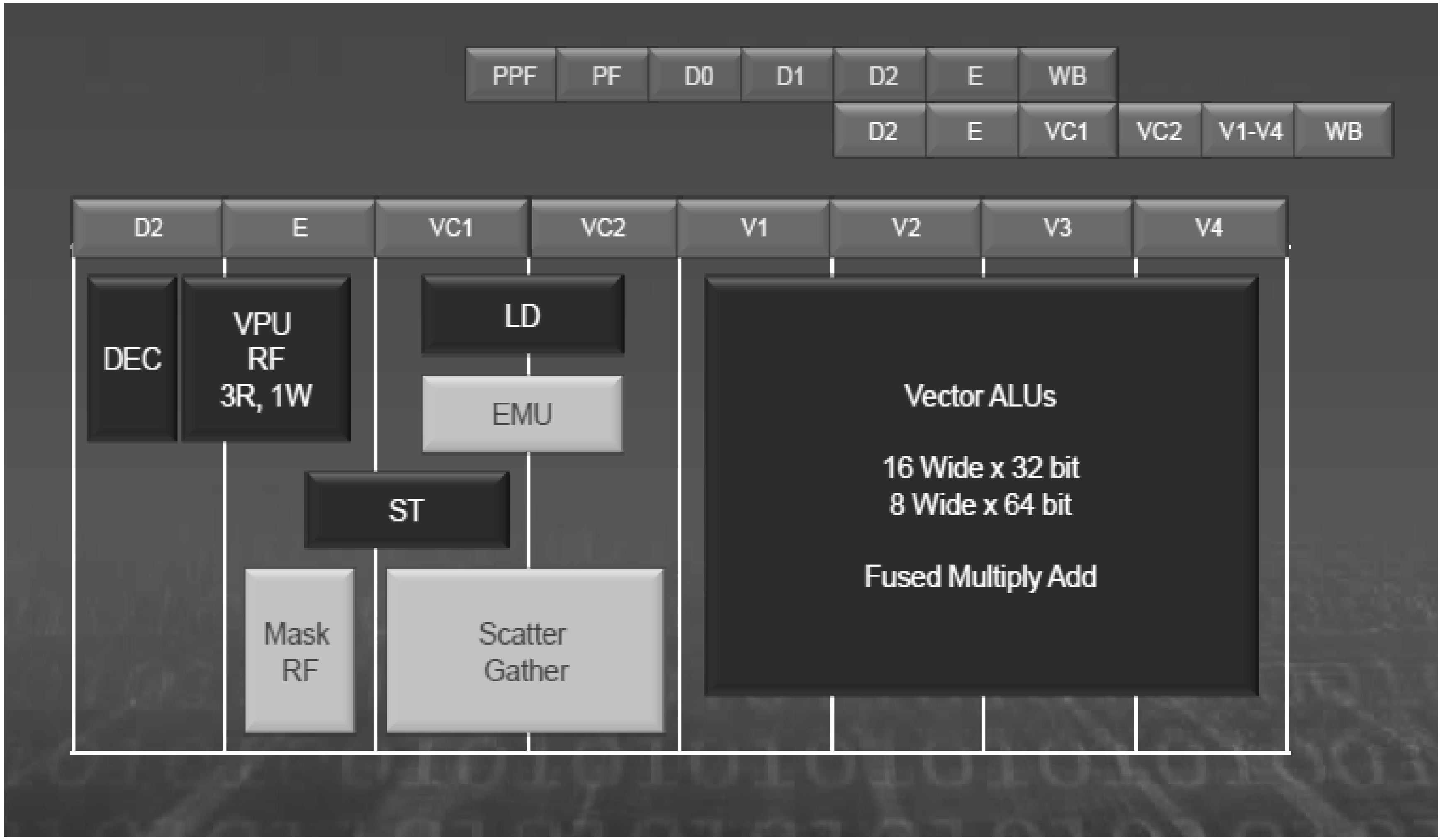
The vector processing unit



- Vector processing unit (VPU) associated with each core.
- This is primarily a sixteen-element wide SIMD engine, operating on 512-bit vector registers.
- Gather / Scatter Unit
- Vector Mask

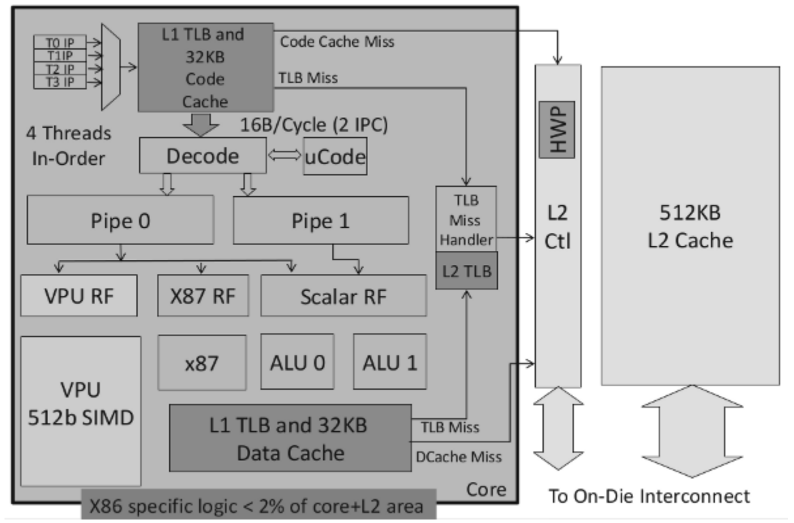
Source : References & Intel Xeon-Phi; http://www.intel.com/

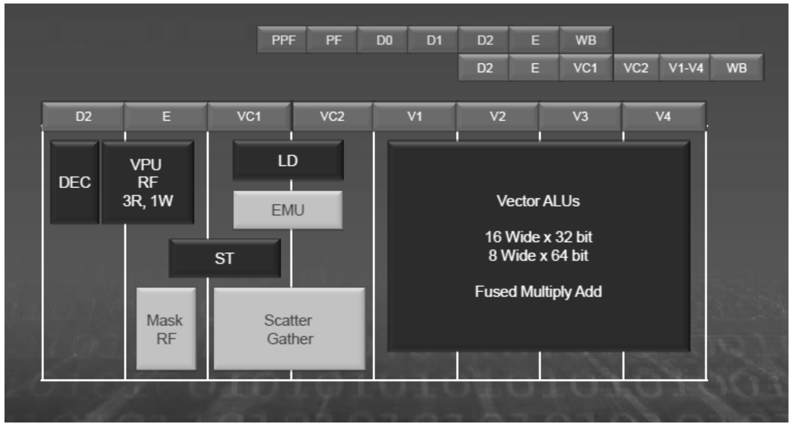
The vector processing unit



Source : References & Intel Xeon-Phi; http://www.intel.com/

Xeon Phi : The Vector Processing Unit





- ❖ Vector processing unit (VPU) associated with each core.
- ❖ This is primarily a sixteen-element wide SIMD engine, operating on 512-bit vector registers.
- ❖ Gather / Scatter Unit
- ❖ Vector Mask
- ❖ Fetches and decodes instructions fr four hardware thread execution contexts
- ❖ Executes the x86 ISA, and Knights Corner vector instructions
- ❖ The core can execute 2 instructions per clock cycle, one per pipe - 32KB, 8-Way set associative L1 Icache & Dcache
- ❖ Core Ring Interface (CRI)
- ❖ L2 Cache

Source : References & Intel Xeon-Phi; http://www.intel.com/

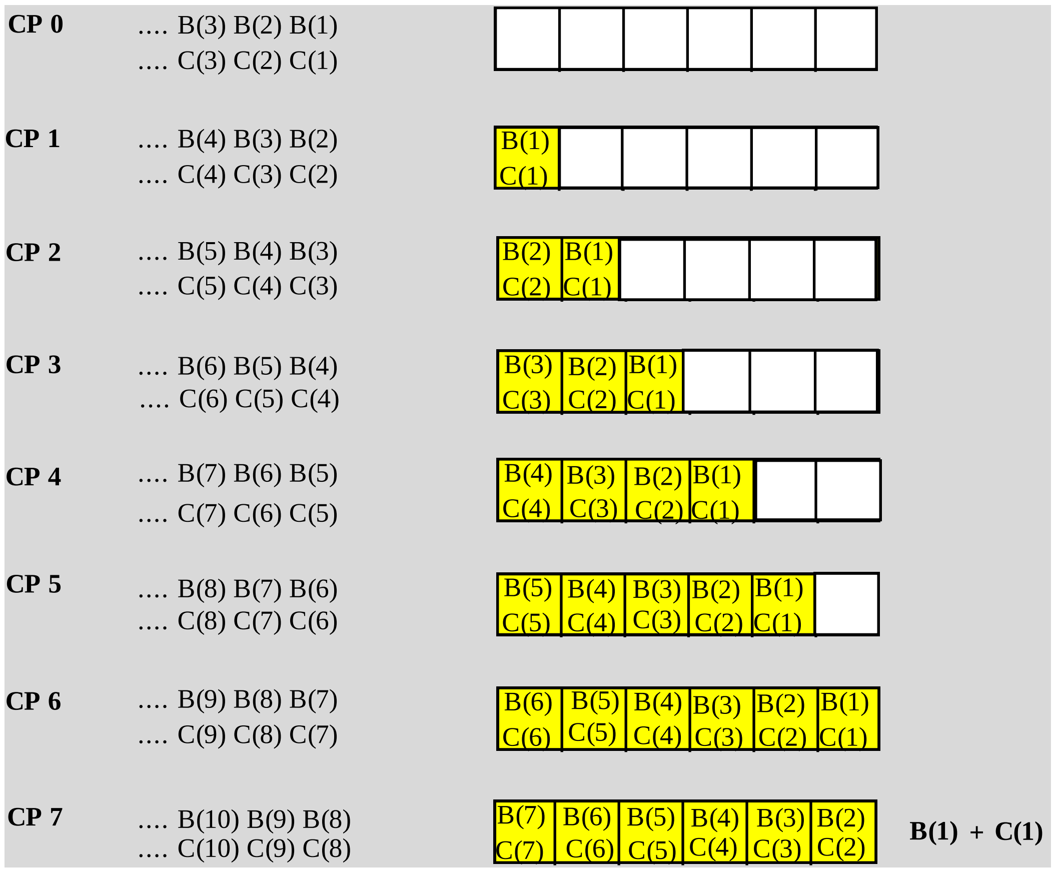
Intel Xeon Phi : Vector Instruction Performance

Vector processing

```
do i = 1, N
   A(i) = B(i)+C(i)
end do
```

$V0 \leftarrow V1 + V2$

Functional Unit Add Floating Point



Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Vector Instruction Performance

- VPU contains 16 SP ALUs, 8 DP ALUs,
- Most VPU instructions have a latency of 4 cycles and TPT 1 cycle
  - Load/Store/Scatter have 7-cycle latency
  - Convert/Shuffle have 6-cycle latency
- VPU instruction are issued in u-pipe
- Certain instructions can go to v-pipe also
  - Vector Mask, Vector Store, Vector Packstore, Vector Prefetch, Scalar

Intel Xeon Phi : Vector Instruction Performance

- Vectorization is key for performance
  - Sandybridge, MIC, etc.
  - Compiler hints
  - Code restructuring
- Many-core nodes present scalability challenges
  - Memory contention
  - Memory size limitations

Source : References & Intel Xeon-Phi; http://www.intel.com/

Demand vectorization by annotation - #pragma simd

- Syntax: `#pragma simd [<clause-list>]`
  - Mechanism to force vectorization of a loop
  - Programmer: asserts a loop ought to be vectorized
  - Compiler: vectorizes the loop or gives an error

Clause	Semantics
No clause	Enforce vectorization of innermost loops; ignore dependencies etc
`vectorlength` *(n_1[, n_2]…)*	Select one or more vector lengths (range: 2, 4, 8, 16) for the vectorizer to use.
`private` *(var_1, var_2, …, var_N)*	Scalars private to each iteration. Initial value broadcast to all instances. Last value copied out from the last loop iteration instance.
`linear` *(var_1:$step_1$, …, var_N:$step_N$)*	Declare induction variables and corresponding positive integer step sizes (in multiples of vector length)
`reduction` *(operator:var_1, var_2,…, var_N)*	Declare the private scalars to be combined at the end of the loop using the specified reduction operator
[`no`*]*`assert`	Direct compiler to assert when the vectorization fails. Default is to assert for SIMD pragma.

Source : References & Intel Xeon-Phi; http://www.intel.com/

SIMD Abstraction – Vectorization/SIMD

```
for (i = 0; i < 15; i++)
    if (v5[i] < v6[i])
        v1[i] += v3[i];
```

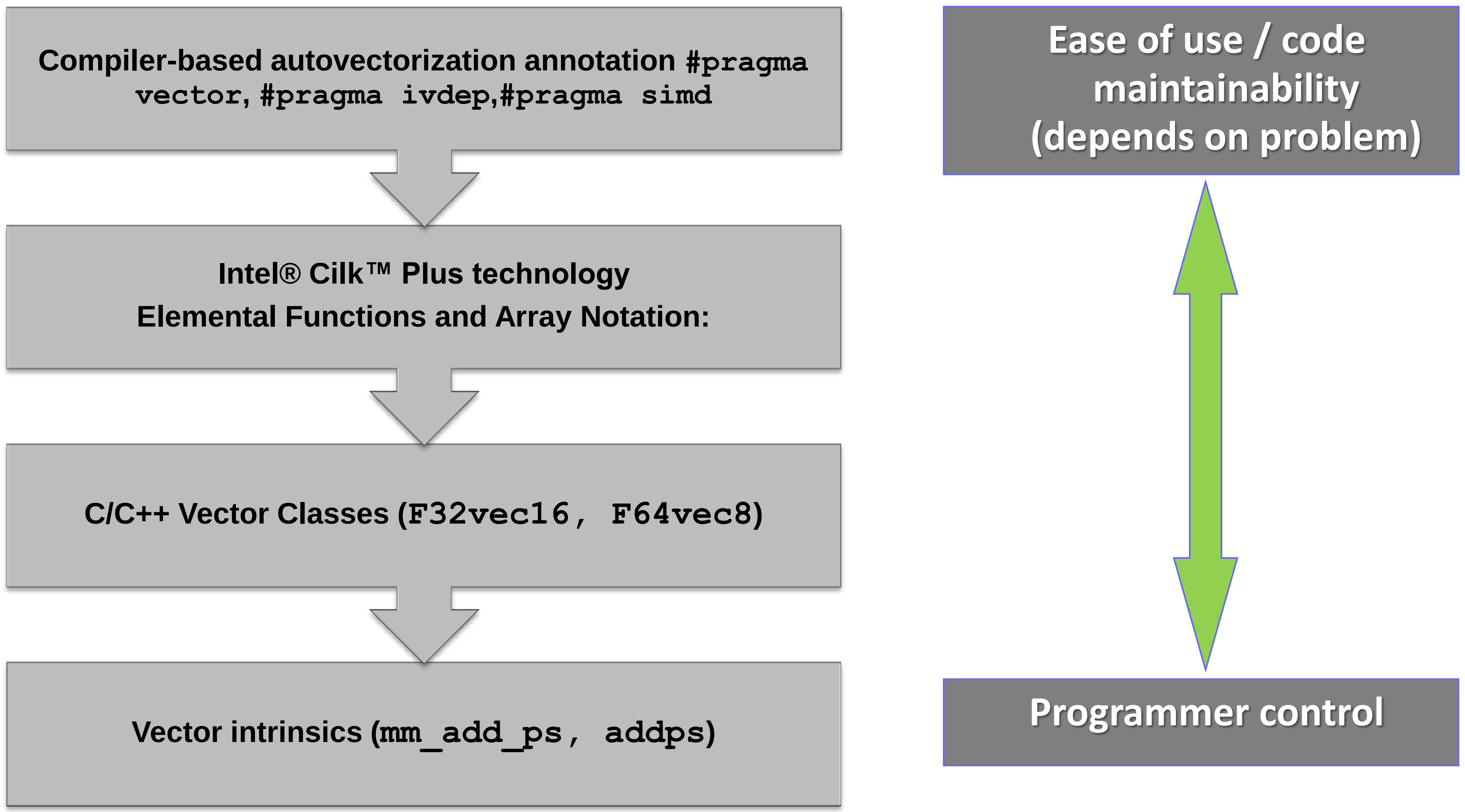
SIMD can simplify your code and reduce the jumps, breaks in program flow control

Note the lack of jumps or conditional code branches

```
v5 = 0 4 7 8 3 9 2 0 6 3 8 9 4 5 0 1
v6 = 9 4 8 2 0 9 4 5 5 3 4 6 9 1 3 0
vcmppi_lt k7, v5, v6
k7 = 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0


v3 = 5 6 7 8 5 6 7 8 5 6 7 8 5 6 7 8
v1 = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
vaddpi v1{k7}, v1, v3
v1 = 6 1 8 1 1 1 8 9 1 1 1 1 6 1 8 1
```

SIMD Abstraction – Options Compared


Compiler-based autovectorization annotation #pragma vector, #pragma ivdep,#pragma simd
Intel® Cilk™ Plus technology
Elemental Functions and Array Notation:
C/C++ Vector Classes (F32vec16, F64vec8)
Vector intrinsics (mm_add_ps, addps)
Ease of use / code maintainability (depends on problem)
Programmer control

Source : References & Intel Xeon-Phi; http://www.intel.com/

Demand vectorization by annotation - #pragma simd

- Syntax: **#pragma simd [<clause-list>]**
  - Mechanism to force vectorization of a loop
  - Programmer: asserts a loop ought to be vectorized
  - Compiler: vectorizes the loop or gives an error

Clause	Semantics
No clause	Enforce vectorization of innermost loops; ignore dependencies etc
vectorlength (n_1[, n_2]…)	Select one or more vector lengths (range: 2, 4, 8, 16) for the vectorizer to use.
private (var_1, var_2, …, var_N)	Scalars private to each iteration. Initial value broadcast to all instances. Last value copied out from the last loop iteration instance.
linear (var_1:$step_1$, …, var_N:$step_N$)	Declare induction variables and corresponding positive integer step sizes (in multiples of vector length)
reduction (operator:var_1, var_2,…, var_N)	Declare the private scalars to be combined at the end of the loop using the specified reduction operator
[no]assert	Direct compiler to assert when the vectorization fails. Default is to assert for SIMD pragma.

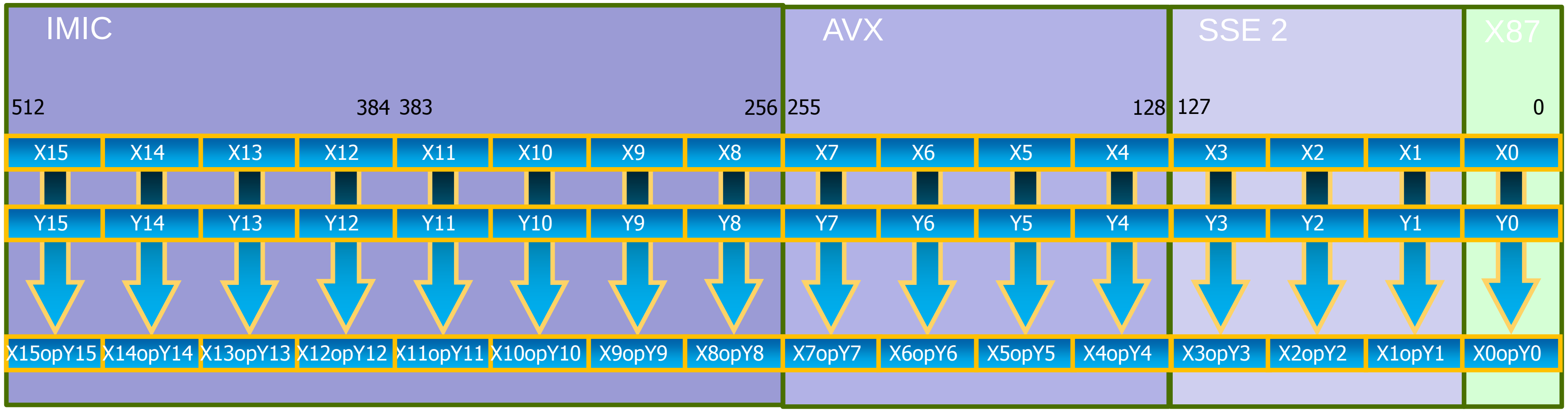
Source : References & Intel Xeon-Phi; http://www.intel.com/

Software Behind the Vectorization

```
float *restrict A, *B, *C;
for(i=0;i<n;i++){
  A[i] = B[i] + C[i];
}
```

Vector (or SIMD) Code computes more than one element at a time.

- [SSE2] 4 elems at a time
  `addps xmm1, xmm2`
- [AVX] 8 elems at a time
  `vaddps ymm1, ymm2, ymm3`
- [IMCI] 16 elems at a time
  `vaddps zmm1, zmm2, zmm3`



IMIC
AVX
SSE 2
X87
512
384 383
256 255
128 127
0
X15 X14 X13 X12 X11 X10 X9 X8 X7 X6 X5 X4 X3 X2 X1 X0
Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
X15opY15 X14opY14 X13opY13 X12opY12 X11opY11 X10opY10 X9opY9 X8opY8 X7opY7 X6opY6 X5opY5 X4opY4 X3opY3 X2opY2 X1opY1 X0opY0

Source : References & Intel Xeon-Phi; http://www.intel.com/

Hardware resources behind Vectorization

- CPU has lot of computation power in form of SIMD unit.

- XMM (128bit) can operate
  - 16x chars
  - 8x shorts
  - 4x dwords/floats
  - 2x qwords/doubles/float complex

- YMM (256bit) can operate
  - 32x chars
  - 16x shorts
  - 8x dwords/floats
  - 4x qwords/doubles/float complex
  - 2x double complex

- Intel® Xeon Phi™ Coprocessor (512bit) can operate
  - 16x chars/shorts (converted to int)
  - 16x dwords/floats
  - 8x qwords/doubles/float complex
  - 4x double complex

Source : References & Intel Xeon-Phi; http://www.intel.com/

Compiler-based Vectorization

Use Compiler Optimization Switches

```
#include <math.h>
void quad(int length, float *a, float *b, float *c,  \
              float *restrict x1, float *restrict x2)
{
    for (int i=0; i<length; i++) {
       float s = b[i]*b[i] - 4*a[i]*c[i];
        if ( s >= 0 ) {
                s = sqrt(s) ;
            x2[i] = (-b[i]+s)/(2.*a[i]);
            x1[i] = (-b[i]-s)/(2.*a[i]);
    }
   else {
          x2[i] = 0.;
         x1[i] = 0.;
    }
 }
}
```

```
>cc -c -restrict -vec-report2 quad.cpp
> quad5.cpp(5) (col. 3): remark: LOOP WAS VECTORIZED.
```

Use Compiler Optimization Switches

Optimization Done	Linux*
Disable optimization	-O0
Optimize for speed (no code size increase)	-O1
Optimize for speed (default)	-O2
High-level loop optimization	-O3
Create symbols for debugging	-g
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program	-fast (same as: -ipo –O3 -no-prec-div -static -xHost)
OpenMP 3.0 support	-openmp
Automatic parallelization	-parallel

Source : References & Intel Xeon-Phi; http://www.intel.com/

Compiler Reports – Optimization Report

Compiler switch:

`-opt-report-phase[=phase]`

`phase` can be:

- **`ipo_inl`** - Interprocedural Optimization Inlining Report
- **`ilo`** – Intermediate Language Scalar Optimization
- **`hpo`** – High Performance Optimization
- **`hlo`** – High-level Optimization
- **`all`** – All optimizations (not recommended, output too verbose)

Control the level of detail in the report:

`-opt-report[0|1|2|3]`

If you do not specify the option, no optimization report is being generated; if you do not specify the level (i.e. -opt-report) level 2 is being used by the compiler.

Source : References & Intel Xeon-Phi; http://www.intel.com/

Get Your Code Vectorized by Intel Compiler

- Data Layout, AOS -> SOA
- Data Alignment (next slide)
- Make the loop innermost
- Function call in treatment
  - Inline yourself
  - inline! Use `__forceinline`
  - Define your own vector version
  - Call vector math library - SVML
- Adopt jumpless algorithm
- Read/Write is OK if it's continuous
- Loop carried dependency

Array of Structures		
S0	X0	T0
S1	X1	T1
…	…	…

Structure of Arrays		
S0	S1	…
X0	X1	…
S0	S1	…

Not a true dependency

```
for(int i = TIMESTEPS; i > 0; i--)
#pragma simd
#pragma unroll(4)
for(int j = 0; j <= i - 1; j++)
 cell[j]=puXDf*cell[j+1]+pdXDf*cell[j];
CallResult[opt] = (Basetype)cell[0];
```

A true dependency

```
for (j=1; j<MAX; j++)
 a[j] = a[j] + c * a[j-n];
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Compiler-Based Autovectorization

- ❖ Compiler recreate vector instructions from the serial Program
- ❖ Compiler make decisions based on some assumption
- ❖ The programmer reassures the compiler on those assumptions
  - ➢ The compiler takes the directives and compares them with its analysis of the code

```
#pragma simd
reduction(+:sum)
for(i=0;i<*p;i++) {
  a[i] = b[i]*c[i];
  sum = sum + a[i];
}
```

- ❖ Compiler checks for
  - ➢ Is "*p" loop invariant?
  - ➢ Are a, b, and c loop invariant?
  - ➢ Does a[] overlap with b[], c[], and/or sum?
  - ➢ Is "+" operator associative? (Does the order of "add"s matter?)
  - ➢ Vector computation on the target expected to be faster than scalar code?

Source : References & Intel Xeon-Phi; http://www.intel.com/

Compiler-Based Autovectorization

- **Compiler checks for**
  - Is “*p” loop invariant?
  - Are a, b, and c loop invariant?
  - Does a[] overlap with b[], c[], and/or sum?
  - Is “+” operator associative? (Does the order of “add”s matter?)
  - Vector computation on the target expected to be faster than scalar code?
- **Compiler Confirms this loop :**
  - “*p” is loop invariant
  - a[] is not aliased with b[], c[], and sum
  - sum is not aliased with b[] and c[]
  - “+” operation on sum is associative (Compiler can reorder the “add”s on sum)
  - Vector code to be generated even if it could be slower than scalar code

Compiler-Based Autovectorization

- Compiler recreate vector instructions from the serial Program
- Compiler make decisions based on some assumption
- The programmer reassures the compiler on those assumptions
  - The compiler takes the directives and compares them with its analysis of the code
- Compiler checks for
  - Is "*p" loop invariant?
  - Are a, b, and c loop invariant?
  - Does a[] overlap with b[], c[], and/or sum?
  - Is "+" operator associative? (Does the order of "add"s matter?)
  - Vector computation on the target expected to be faster than scalar code?

```
#pragma simd reduction(+:sum)
      for(i=0;i<*p;i++) {
        a[i] = b[i]*c[i];
        sum = sum + a[i];
             }
```

- Compiler Confirms this loop :
  - "*p" is loop invariant
  - a[] is not aliased with b[], c[], and sum
  - sum is not aliased with b[] and c[]
  - "+" operation on sum is associative (Compiler can reorder the "add"s on sum)
  - Vector code to be generated even if it could be slower than scalar code

Source : References & Intel Xeon-Phi; http://www.intel.com/

Hints to Compiler for Vectorization Opportunities

#pragma	Semantics
#pragma ivdep	Ignore vector dependences unless they are proven by the compiler
#pragma vector always [assert]	If the loop is vectorizable, ignore any benefit analysis If the loop did not vectorize, give a compile-time error message via assert
#pragma novector	Specifies that a loop should never be vectorized, even if it is legal to do so, when avoiding vectorization of a loop is desirable (when vectorization results in a performance regression)

Source : References & Intel Xeon-Phi; http://www.intel.com/

Hints to Compiler for Vectorization Opportunities

#pragma	**Semantics**
#pragma vector aligned / unaligned	instructs the compiler to use aligned (unaligned) data movement instructions for all array references when vectorizing
#pragma vector temporal / nontemporal	directs the compiler to use temporal/non-temporal (that is, streaming) stores on systems based on IA-32 and Intel® 64 architectures; optionally takes a comma separated list of variables

Source : References & Intel Xeon-Phi; http://www.intel.com/

Hints to Compiler for Vectorization

#pragma	Semantics
#pragma ivdep	Ignore vector dependences unless they are proven by the compiler
#pragma vector always [assert]	If the loop is vectorizable, ignore any benefit analysis If the loop did not vectorize, give a compile-time error message via assert
#pragma novector	Specifies that a loop should never be vectorized, even if it is legal to do so, when avoiding vectorization of a loop is desirable (when vectorization results in a performance regression)
#pragma vector aligned / unaligned	instructs the compiler to use aligned (unaligned) data movement instructions for all array references when vectorizing
#pragma vector temporal / nontemporal	directs the compiler to use temporal/non-temporal (that is, streaming) stores on systems based on IA-32 and Intel® 64 architectures; optionally takes a comma separated list of variables

Source : References & Intel Xeon-Phi; http://www.intel.com/

Compiler VEC report

- Indicates whether each loop is vectorized
  - Vectorized ≠ efficient
- Different levels
  - -vec-report1, for high-level triage of large code
  - -vec-report2, when you want reasons for not vectorizing
  - -vec-report6, for even more detail, e.g. misalignment
- Indicates reasons for not vectorizing
  - Unsupported datatype → rewrite to use 32b indices vs. 64b
- Line numbers may not be what you expect
  - Inlining
  - Loop distribution, interchange, unrolling, collapsing

Source : References & Intel Xeon-Phi; http://www.intel.com/

Compiler OPT report - contents

- Control over static reports
  - -opt-report [n=0-3] enables varying levels of detail
  - -opt-report-phase=[several options] enables specific detail
- Reveals info on various compiler optimization
  - Offloaded variables, –opt-report-phase=offload
  - Inlining, Vectorization
  - OpenMP parallelization, auto-parallelization
  - Loop permutations, loop distribution, loop distribution
  - Multiversioning of loops performed by compiler
    - Dynamic dependence checking, unit-stride for assumed shape arrays, trip-count checks, etc.
  - Prefetching
  - Blocking, unrolling, jamming
  - Whole-program optimization

Source : References & Intel Xeon-Phi; http://www.intel.com/

Prefetch on Intel Multicore and Manycore

- **Objective:** Move data from memory to L1 or L2 Cache in anticipation of CPU Load/Store
- More import on in-order Intel Xeon Phi Coprocessor
- Less important on out of order Intel Xeon Processor
- Compiler prefetching is on by default for Intel® Xeon Phi™ coprocessors at –O2 and above
- Compiler prefetch is not enabled by default on Intel® Xeon® Processors
  - Use external options `-opt-prefetch[=n] n = 1.. 4`
- Use the compiler reporting options to see detailed diagnostics of prefetching per loop
  - Use `-opt-report-phase hlo -opt-report 3`

Source : References & Intel Xeon-Phi; http://www.intel.com/

Automatic Prefetches

Loop Prefetch

- Compiler generated prefetches target memory access in a future iteration of the loop
- Target regular, predictable array and pointer access

Interactions with Hardware prefetcher

- Intel® Xeon Phi™ Comprocessor has a hardware L2 prefetcher
- If Software prefetches are doing a good job, Hardware prefetching does not kick in
- References not prefetched by compiler may get prefetched by hardware prefetcher

Source : References & Intel Xeon-Phi; http://www.intel.com/

Explicit Prefetch

- **Use Intrinsics**
  - `_mm_prefetch((char *) &a[i], hint);`
    See xmmintrin.h for possible hints (for L1, L2, non-temporal, …)
  - But you have to specify the prefetch distance
  - Also gather/scatter prefetch intrinsics, see zmmintrin.h and compiler user guide, e.g. _mm512_prefetch_i32gather_ps
- **Use a pragma / directive (easier):**
  - #pragma prefetch a [:hint[:distance]]
  - You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.
- **Use Compiler switches:**
  - `-opt-prefetch-distance=n1[,n2]`
  - specify the prefetch distance (how many iterations ahead, use n1 and prefetches inside loops. n1 indicates distance from memory to L2.

Source : References & Intel Xeon-Phi; http://www.intel.com/

Memory Alignment

- Allocated memory on heap
  - `_mm_malloc(int size, int aligned)`
  - `scalable_aligned_malloc(int size, int aligned)`
- Declarations memory:
  - `__attribute__((aligned(n))) float v1[];`
  - `__declspec(align(n)) float v2[];`
- Use this to notify compiler
  - `__assume_aligned(array, n);`
- Natural boundary
  - Unaligned access can fault the processor
- Cacheline Boundary
  - Frequently accessed data should be in 64
- 4K boundary
  - Sequentially accessed large data should be in 4K boundary

Instruction	Length	Alignment
SSE	128 Bits	16 Bytes
AVX	256 Bits	32 Bytes
IMCI	512 Bits	64 Bytes

Source : References & Intel Xeon-Phi; http://www.intel.com/

Streaming Store

- Avoid read for ownership for certain memory write operation
- Bypass prefetch related to the memory read
- Use `#pragma vector nontemporal(v1,…)` to drop a hint to compiler
- Without Streaming Stores 448 Bytes read/write per iteration
  - With Streaming Stores, 320 Bytes read/write per iteration
  - Relief Bandwidth pressure; improve cache utilization
  - –vec-report6 displays the compiler action

```
bs_test_sp.c(215): (col. 4) remark: vectorization support:
streaming store was generated for CallResult.
bs_test_sp.c(216): (col. 4) remark: vectorization support:
streaming store was generated for PutResult.
```

```
for (int chunkBase = 0; chunkBase < OptPerThread; chunkBase +=
CHUNKSIZE)
{
#pragma simd vectorlength(CHUNKSIZE)
#pragma simd
#pragma vector aligned
#pragma vector nontemporal (CallResult, PutResult)
      for(int opt = chunkBase; opt < (chunkBase+CHUNKSIZE); opt++)
      {
         float CNDD1;
         float CNDD2;
         float CallVal =0.0f, PutVal  = 0.0f;
         float T = OptionYears[opt];
         float X = OptionStrike[opt];
         float S = StockPrice[opt];
                       ……

         CallVal  = S * CNDD1 - XexpRT * CNDD2;
         PutVal  = CallVal  +  XexpRT - S;
         CallResult[opt] = CallVal ;
         PutResult[opt] = PutVal ;
      }
}
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Data Blocking

- ❖ Partition data to small blocks that fits in L2 Cache
  - ➢ Exploit data reuse in the application.
  - ➢ Ensure the data remains in the cache across multiple uses
  - ➢ Using the data in cache remove the need to go to memory
  - ➢ Bandwidth limited program may execute at FLOPS limit
- ❖ Simple case of 1D
  - ➢ Data size DATA_N is used WORK_N times from 100s of threads
  - ➢ Each handles a piece of work and have to traverse all data

Without Blocking

- ➢ 100s of thread pound on different area of DATA_N
- ➢ Memory interconnet limit the performance

```
#pragma omp parallel for
for(int wrk = 0; wrk < WORK_N; wrk++)
{
     initialize_the_work(wrk);
     for(int ind = 0; ind < DATA_N; ind++)
     {
        dataptr  datavalue = read_data(dataind);
        result = compute(datavalue);
        aggregate = combine(aggregate, result);
     }
     postprocess_work(aggregate);
}
```

With Blocking

- ➢ Cacheable BSIZE of data is processed by all 100s threads a time
- ➢ Each data is read once kept reusing until all threads are done with it

```
for(int BBase = 0; BBase < DATA_N; BBase += BSIZE)
{
#pragma omp parallel for
    for(int wrk = 0; wrk < WORK_N; wrk++)
    {
         initialize_the_work(wrk);
         for(int ind = BBase; ind < BBase+BSIZE; ind++)
         {
            dataptr  datavalue = read_data(ind);
            result = compute(datavalue);
            aggregate[wrk] = combine(aggregate[wrk], result);
          }
          postprocess_work(aggregate[wrk]);
    }
}
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Offload Code Examples

❖ C/C+ Offload Pragma

```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
 for (i = 0; i<count; i++) {
      float t = (float) (i+0.5/count);
      pi += 4.0/(1.0t*t);
 }
pi/ = count;
```

❖ C/C++ Offload Pragma

```
#pragma offload target(mic)
      in(transa, transb, N, alpha, beta) \
      in(A:length(matrix_elements)) \
      in(B:length(matrix_elements)) \
      inout(C:length(matrix_elements))
         sgemm(&transa, &transb, &N, &N, &N,
& alpha, A, &N, B, & N, &beta, C &N);
```

❖ Fortran Offload Directives

```
!dir$ omp offload target(mic)
!$omp parallel do
     do i = 1, 10
                A(i) = B(i) * C(i)
     enddo
```

❖ C/C++ Language Extension

```
class_Cilk_Shated common {
    int data1;
    int *data2;
    class common *next;
    void process();
}
_Cilk_Shared class common obj1, obj2;
_Cilk_spawn _offload obj1.process();
_Cilk_spawn  _offload  obj2.process();
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Summary: Tricks for Performance

- ❖ Use asynchronous data transfer and double buffering offloads to overlap the communication with the computation
- ❖ Optimizing memory use on Intel MIC architecture target relies on understanding access patterns
- ❖ Many old tricks still apply: peeling, collapsing, unrolling, vectorization can all benefit performance

Conclusions

- An Overview of Intel Xeon-Phi Compilation & Vectorisation techniques are   discussed

References & Acknowledgements

References :

1. Theron Voran, Jose Garcia, Henry Tufo, University Of Colorado at Boulder National Center or Atmospheric Research, TACC-Intel Hihgly Parallel Computing Symposium, Austin TX, April 2012
2. Robert Harkness, Experiences with ENZO on the Intel R Many Integrated Core (Intel MIC) Architecture, National Institute for Computational Sciences, Oak Ridge National Laboratory
3. Ryan C Hulguin, National Institute for Computational Sciences, Early Experiences Developing CFD Solvers for the Intel Many Integrated Core (Intel MIC) Architecture, TACC-Intel Highly Parallel Computing Symposium April, 2012
4. Scott McMillan, Intel Programming Models for Intel Xeon Processors and Intel Many Integrated Core (Intel MIC) Architecture, TACC-Highly Parallel Comp. Symposium April 2012
5. Sreeram Potluri, Karen Tomko, Devendar Bureddy , Dhabaleswar K. Panda, Intra-MIC MPI Communication using MVAPICH2: Early Experience, Network-Based Computing Laboratory, Department of Computer Science and Engineering The Ohio State University, Ohio Supercomputer Center, TACC-Highly Parallel Computing Symposium April 2012
6. Karl W. Schulz, Rhys Ulerich, Nicholas Malaya ,Paul T. Bauman, Roy Stogner, Chris Simmons, Early Experiences Porting Scientific Applications to the Many Integrated Core (MIC) Platform ,Texas Advanced Computing Center (TACC) and Predictive Engineering and Computational Sciences (PECOS) Institute for Computational Engineering and Sciences (ICES), The University of Texas at Austin ,Highly Parallel Computing Symposium ,Austin, Texas, April 2012
7. Kevin Stock, Louis-Noel, Pouchet, P. Sadayappan ,Automatic Transformations for Effective Parallel Execution on Intel Many Integrated, The Ohio State University, April 2012
8. http://www.tacc.utexas.edu/
9. Intel MIC Workshop at C-DAC, Pune April 2013
10. First Intel Xeon Phi Coprocessor Technology Conference iXPTC 2013 New York, March 2013
11. Shuo Li, Vectorization, Financial Services Engineering, software and Services Group, Intel ctel Corporation;
12. Intel® Xeon Phi™ (MIC) Parallelization & Vectorization, Intel Many Integrated Core Architecture, Software & Services Group, Developers Relations Division

References & Acknowledgements

References :

13. Intel® Xeon Phi™ (MIC) Programming, Rama Malladi, Senior Application Engineer, Intel Corporation, Bengaluru India April 2013
14. Intel® Xeon Phi™ (MIC) Performance Tuning, Rama Malladi, Senior Application Engineer, Intel Corporation, Bengaluru India April 2013
15. Intel® Xeon Phi™ Coprocessor Architecture Overview, Dhiraj Kalamkar, Parallel Computing Lab, Intel Labs, Bangalore
16. Changkyu Kim,Nadathur Satish ,Jatin Chhugani ,Hideki Saito,Rakesh Krishnaiyer ,Mikhail Smelyanskiy ,Milind Girkar, Pradeep Dubey, Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology , Technical Report Intel Labs , Parallel Computing Laboratory , Intel Compiler Lab, 2010
17. Colfax International Announces Developer Training for Intel® Xeon Phi™ Coprocessor, Industry First Training Program Developed in Consultation with Intel SUNNYVALE, CA, Nov, 2012
18. Andrey Vladimirov Stanford University and Vadim Karpusenko , Test-driving Intel® Xeon Phi™ coprocessors with a basic N-body simulation Colfax International January 7, 2013 Colfax International, 2013 http://research.colfaxinternational.com/
19. Jim Jeffers and James Reinders,Intel® Xeon Phi™ Coprocessor High-Performance Programming by Morgann Kauffman Publishers Inc, Elsevier, USA. 2013
20. Michael McCool, Arch Robison, James Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufman Publishers Inc, 2013.
21. Dan Stanzione, Lars Koesterke, Bill Barth, Kent Milfeld by Preparing for Stampede: Programming Heterogeneous Many-Core Supercomputers. TACC, XSEDE 12 July 2012
22. John Michalakes, Computational Sciences Center, NREL, & Andrew Porter, Opportunities for WRF Model Acceleration, WRF Users workshop, June 2012
23. Jim Rosinski , Experiences Porting NOAA Weather Model FIM to Intel MIC, ECMWF workshop On High Performance Computing in Meteorology, October 2012
24. Michaela Barth, KTH Sweden , Mikko Byckling, CSC Finland, Nevena Ilieva, NCSA Bulgaria, Sami Saarinen, CSC Finland, Michael Schliephake, KTH Sweden, Best Practice Guide Intel Xeon Phi v0.1, Volker Weinberg (Editor), LRZ Germany March 31 ,2013

References & Acknowledgements

References :

25. Barbara Chapman, Gabriele Jost and Ruud van der Pas, Using OpenMP, MIT Press Cambridge, 2008
26. Peter S Pacheco, An Introduction Parallel Programming, Morgann Kauffman Publishers Inc, Elsevier, USA. 2011
27. Intel Developer Zone: Intel Xeon Phi Coprocessor,
28. http://software.intel.com/en-us/mic-developer
29. Intel Many Integrated Core Architecture User Forum,
30. http://software.intel.com/en-us/forums/intel-many-integrated-core
31. Intel Developer Zone: Intel Math Kernel Library, http://software.intel.com/en-us
32. Intel Xeon Processors & Intel Xeon Phi Coprocessors – Introduction to High Performance Applications Development for Multicore and Manycore – Live Webinar, 26.-27, February .2013,
33. recorded http://software.intel.com/en-us/articles/intel-xeon-phi-training-m-core
34. Intel Cilk Plus Home Page, http://cilkplus.org/
35. James Reinders, Intel Threading Building Blocks (Intel TBB), O'REILLY, 2007
36. Intel Xeon Phi Coprocessor Developer's Quick Start Guide,
37. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide
38. Using the Intel MPI Library on Intel Xeon Phi Coprocessor Systems,
39. http://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems
40. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors,
41. http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf
42. Programming and Compiling for Intel Many Integrated Core Architecture,
43. http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture
44. Building a Native Application for Intel Xeon Phi Coprocessors,
45. http://software.intel.com/en-us/articles/

References & Acknowledgements

References :

46. Advanced Optimizations for Intel MIC Architecture, http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture
47. Optimization and Performance Tuning for Intel Xeon Phi Coprocessors - Part 1: Optimization Essentials, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeonphi-coprocessors-part-1-optimization
48. Optimization and Performance Tuning for Intel Xeon Phi Coprocessors, Part 2: Understanding and Using Hardware Events, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding
49. Requirements for Vectorizable Loops,
50. http://software.intel.com/en-us/articles/requirements-for-vectorizable-
51. R. Glenn Brook, Bilel Hadri, Vincent C. Betro, Ryan C. Hulguin, Ryan Braby. Early Application Experiences with the Intel MIC Architecture in a Cray CX1, National Institute for Computational Sciences. University of Tennessee. Oak Ridge National Laboratory. Oak Ridge, TN USA
52. http://software.intel.com/mic-developer
53. Loc Q Nguyen , Intel Corporation's Software and Services Group , Using the Intel® MPI Library on Intel® Xeon Phi™ Coprocessor System,
54. Frances Roth, System Administration for the Intel® Xeon Phi™ Coprocessor, Intel white Paper
55. Intel® Xeon Phi™ Coprocessor, James Reinders, Supercomputing 2012 Presentation
56. Intel® Xeon Phi™ Coprocessor Offload Compilation, Intel software

References

1. Andrews, Grogory R. (2000), Foundations of Multithreaded, Parallel, and Distributed Programming, Boston, MA : Addison-Wesley
2. Butenhof, David R (1997), Programming with POSIX Threads , Boston, MA : Addison Wesley Professional
3. Culler, David E., Jaswinder Pal Singh (1999), Parallel Computer Architecture - A Hardware/Software Approach , San Francsico, CA : Morgan Kaufmann
4. Grama Ananth, Anshul Gupts, George Karypis and Vipin Kumar (2003), Introduction to Parallel computing, Boston, MA : Addison-Wesley
5. Intel Corporation, (2003), Intel Hyper-Threading Technology, Technical User's Guide, Santa Clara CA : Intel Corporation Available at : http://www.intel.com
6. Shameem Akhter, Jason Roberts (April 2006), Multi-Core Programming - Increasing Performance through Software Multi-threading , Intel PRESS, Intel Corporation,
7. Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell (1996), Pthread Programming O'Reilly and Associates, Newton, MA 02164,
8. James Reinders, Intel Threading Building Blocks – (2007) , O'REILLY series
9. Laurence T Yang & Minyi Guo (Editors), (2006) High Performance Computing - Paradigm and Infrastructure Wiley Series on Parallel and Distributed computing, Albert Y. Zomaya, Series Editor
10. Intel Threading Methodology ; Principles and Practices Version 2.0 copy right (March 2003), Intel Corporation
11. William Gropp, Ewing Lusk, Rajeev Thakur **(1999),** Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press..
12. Pacheco S. Peter, **(1992),** Parallel Programming with MPI, , University of Sanfrancisco, Morgan Kaufman Publishers, Inc., Sanfrancisco, California
13. Kai Hwang, Zhiwei Xu, (**1998**), Scalable Parallel Computing (Technology Architecture Programming), McGraw Hill New York.
14. Michael J. Quinn (**2004**), Parallel Programming in C with MPI and OpenMP McGraw-Hill International Editions, Computer Science Series, McGraw-Hill, Inc. Newyork
15. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Progrmaming, Boston, MA : Addison-Wesley

References

16. SunSoft. Solaris multithreaded programming guide. SunSoft Press, Mountainview, CA, **(1996)**, Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill,
17. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, **(2001)**,Parallel Programming in OpenMP San Fracncisco Moraan Kaufmann
18. S.Kieriman, D.Shah, and B.Smaalders **(1995),** Programming with Threads, SunSoft Press, Mountainview, CA. 1995
19. Mattson Tim, **(2002),** Nuts and Bolts of multi-threaded Programming Santa Clara, CA : Intel Corporation, Available at : http://www.intel.com
20. I. Foster **(1995,** Designing and Building Parallel Programs ; Concepts and tools for Parallel Software Engineering, Addison-Wesley (1995
21. J.Dongarra, I.S. Duff, D. Sorensen, and H.V.Vorst **(1999),** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) SIAM, 1999
22. OpenMP C and C++ Application Program Interface, Version 1.0". **(1998),** OpenMP Architecture Review Board. October 1998
23. D. A. Lewine. *Posix Programmer's Guide:* **(1991),** Writing Portable Unix Programs with the Posix. 1 Standard. O'Reilly & Associates, 1991
24. Emery D. Berger, Kathryn S McKinley, Robert D Blumofe, Paul R.Wilson, *Hoard : A Scalable Memory Allocator for Multi-threaded Applications* ; The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). Cambridge, MA, November (**2000**). Web site URL : http://www.hoard.org/
25. Marc Snir, Steve Otto, Steyen Huss-Lederman, David Walker and Jack Dongarra, (**1998**) *MPI-The Complete Reference: Volume 1, The MPI Core, second edition* [MCMPI-07].
26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (**1998**) *MPI-The Complete Reference: Volume 2, The MPI-2 Extensions*
27. A. Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill, **(1996)**
28. OpenMP C and C++ Application Program Interface, Version 2.5 (**May 2005**)", From the OpenMP web site, URL **: http://www.openmp.org/**
29. Stokes, Jon 2002 Introduction to Multithreading, Super-threading and Hyper threading *Ars Technica*, October **(2002)**

References

30. Andrews Gregory R. 2000, Foundations of Multi-threaded, Parallel and Distributed Programming, Boston MA : Addison – Wesley (**2000)**
31. Deborah T. Marr , Frank Binns, David L. Hill, Glenn Hinton, David A Koufaty, J . Alan Miller, Michael Upton, "Hyperthreading, Technology Architecture and Microarchitecture", Intel (**2000-01)**
32. http://www.erc.msstate.edu/mpi
33. http://www.arc.unm.edu/workshop/mpi/mpi.html
34. http://www.mcs.anl.gov/mpi/mpich
35. The MPI home page, with links to specifications for MPI-1 and MPI-2 standards : http://www.mpi–forum.org
36. Hybrid Programming Working Group Proposals, Argonne National Laboratory, Chiacago (2007-2008)
37. TRAC Link : https://svn.mpi-forum.org/trac/mpi-form-web/wiki/MPI3Hybrid
38. Threads and MPI Software, Intel Software Products and Services 2008 - 2009
39. Sun MPI 3.0 Guide November 2007
40. Treating threads as MPI processes thru Registration/deregistration –Intel Software Products and Services 2008 – 2009
41. Intel MPI library 3.2 - http://www.hearne.com.au/products/Intelcluster/edition/mpi/663/
42. http://www.cdac.in/opecg2009/
43. PGI Compilers http://www.pgi.com

Thank You

Any questions ?