Summer School on PARALLEL COMPUTING

# Intel Xeon Phi: Architecture and Programming

**F. Affinito(f.affinito@cineca.it) V. Ruggiero (v.ruggiero@cineca.it)**
**Roma, 23 July 2015**
**SuperComputing Applications and Innovation Department**

CINECA

## Outline

Architectures

Optimization

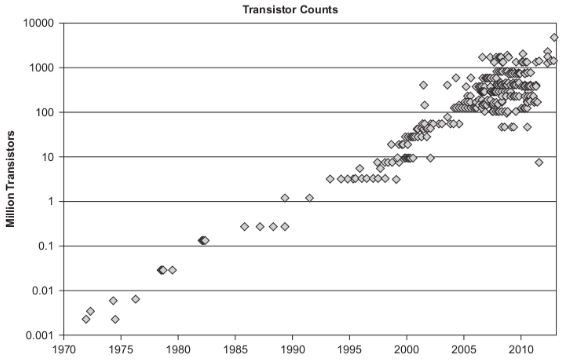
Vectorization

Performance and parallelism

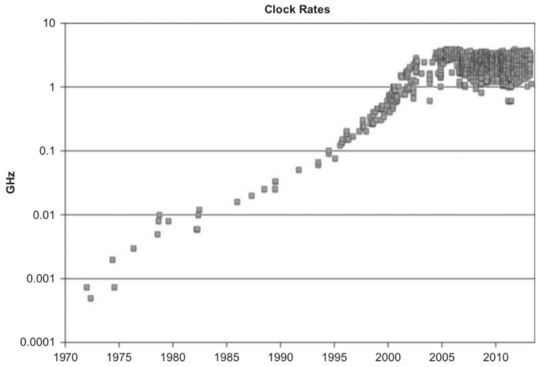
Programmming Models

Profiling and Debugging

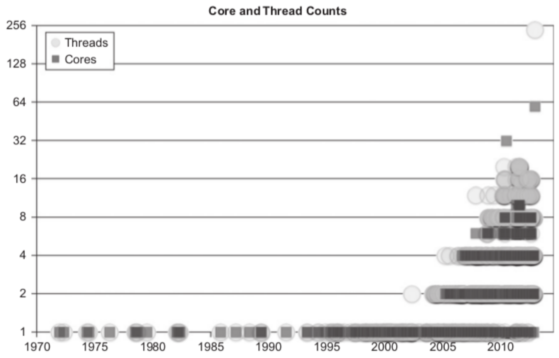
## Trends: transistor...



## Trends: clock rates...



## Trends: core and threads ...



## Trends: summarizing...

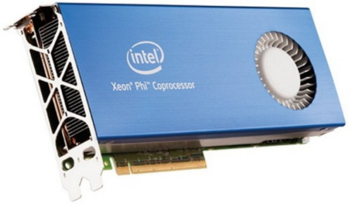
- The number of transistors increases
- The power consumption must not increase
- The density cannot increase on a single chip

Solution :

- Increase the number of cores

## GP-GPU and Intel Xeon Phi..

- Coupled to the CPU
- To accelerate highly parallel kernels, facing with the Amdahl Law



## What is Intel Xeon Phi?

- 7100 / 5100 / 3100 Series available
- 5110P:
  - Intel Xeon Phi clock: 1053 MHz
  - 60 cores in-order
  - 1 TFlops/s DP peak performance (2 Tflops SP)
  - 4 hardware threads per core
  - 8 GB DDR5 memory
  - 512-bit SIMD vectors (32 registers)
  - Fully-coherent L1 and L2 caches
  - PCIe bus (rev. 2.0)
  - Max Memory bandwidth (theoretical) 320 GB/s
  - Max TDP: 225 W

## MIC vs GPU naive comparison

- The comparison is naive

| System | K20s | 5110P |
|---|---|---|
| # cores | 2496 | 60 (*4) |
| Memory size | 5 GB | 8 GB |
| Peak performance (SP) | 3.52 TFlops | 2 TFlops |
| Peak performance (DP) | 1.17 TFlops | 1 TFlops |
| Clock rate | 0.706 GHz | 1.053 GHz |
| Memory Bandwidth | 208 GB/s (ECC off) | 320 GB/s |

## Terminology

- MIC = Many Integrated Cores is the name of the architecture
- Xeon Phi = Commercial name of the Intel product based on the MIC architecture
- Knight's corner, Knight's landing, Knight's ferry are development names of MIC architectures
- We will often refer to the CPU as HOST and Xeon Phi as DEVICE

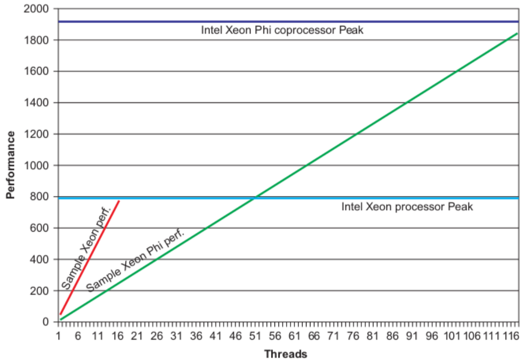
## Is it an accelerator?

- YES: It can be used to "accelerate" hot-spots of the code that are highly parallel and computationally extensive
- In this sense, it works alongside the CPU
- It can be used as an accelerator using the "offload" programming model
- An important bottleneck is represented by the communication between host and device (through PCIe)
- Under this respect, it is very similar to a GPU

## Is it an accelerator? / 2

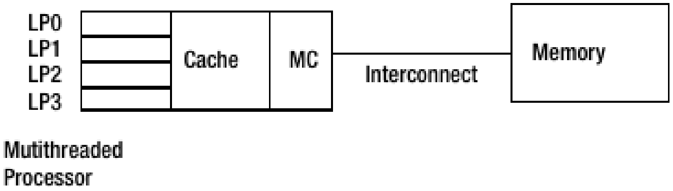
- NOT ONLY: the Intel Xeon Phi can behave as a many-core X86 node.
  - Code can be compiled and run "natively" on the Xeon Phi platform using MPI + OpenMP
- The bottleneck is the scalability of the code
  - Amdahl Law
- Under this respect, the Xeon Phi is completely different from a GPU
  - This is way we often call the Xeon Phi "co-processor" rather than "accelerator"

## Many-core performances



## Architecture key points/1

- Instruction Pipelining
  - Two independent pipelines arbitrarily known as the U and V pipelines
  - (only) 5 stages to cope with a reduced clock rate, e.g. compared to the Pentium 20 stages
  - In-order instruction execution
- Manycore architecture
  - Homogeneous
  - 4 hardware threads per core



## Architecture key points/2

- Interconnect: bidirectional ring topology
  - All the cores talk to one another through a bidirectional interconnect
  - The cores also access the data and code residing in the main memory through the ring connecting the cores to memory controller
- Given eight memory controllers with two GDDR5 channels running at 5.5 GB/s
  - Aggregate Memory Bandwidth = 8 memory controllers x 2 channels x 5.5 GB/s x 4 bytes/transfer = 352 GB/s
- System interconnect
  - Xeon Phi are often placed on PCIe slots to work with the host processors

## Architecture key points/3

- Cache:
  - L1: 8-ways set-associative 32-kB instruction and 32-kB data
  - L1 access time: 3 cycles
  - L2: 8-way set associative and 512 kB in size (unified)Interconnect: bidirectional ring topology

  - TLB cache:
  - L1 data TLB supports three page sizes: 4 kB, 64 kB, and 2 MB
  - L2 TLB
  - If one misses L1 and also misses L2 TLB, one has to walk four levels of page table, which is pretty expensive

## Architecture key points/4

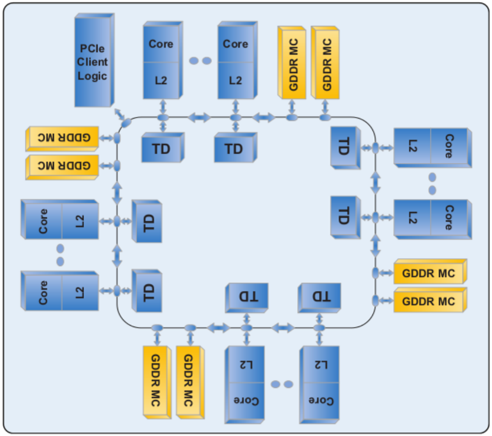
- The VPU (vector processing unit) implements a novel instruction set architecture (ISA), with 218 new instructions compared with those implemented in the Xeon family of SIMD instruction sets.
- The VPU is fully pipelined and can execute most instructions with four-cycle latency and single-cycle throughput.
- Each vector can contain 16 single-precision floats or 32-bit integer elements or eight 64-bit integer or double-precision floating point elements.

## Architecture key points/5

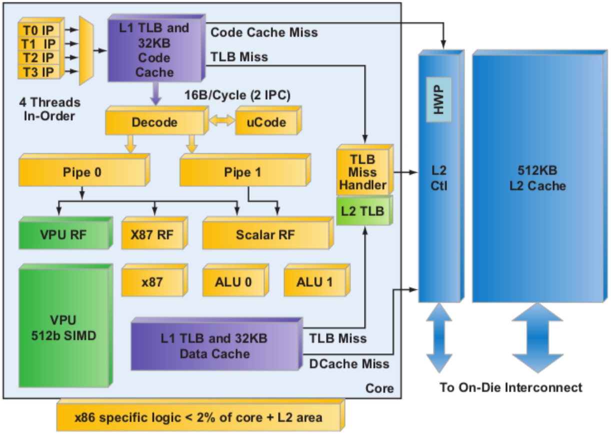
- Each VPU instruction passes through one or more of the following five pipelines to completion:
  - Double-precision (DP) pipeline: Used to execute float64 arithmetic, conversion from float64 to float32, and DP-compare instructions.
  - Single-precision (SP) pipeline: Executes most of the instructions including 64-bit integer loads. This includes float32/int32 arithmetic and logical operations, shuffle/broadcast, loads including loadunpack, type conversions from float32/int32 pipelines, extended math unit (EMU) transcendental instructions, int64 loads, int64/float64 logical, and other instructions.
  - Mask pipeline: Executes mask instructions with one-cycle latencies.
  - Store pipeline: Executes the vector store operations.
  - Scatter/gather pipeline: Executes the vector register read/writes from sparse memory locations.
- Mixing SP and DP computations is expensive!

Summer School on PARALLEL COMPUTING

## Architecture sketch/1



## Architecture sketch/2



## Outline

## Optimizing code step by step

1. Check correctness of your application by building it without optimization using **`-O0`**.
2. Use the general optimization options (**`-O1`**,**`-O2`**,or**`-O3`**) and determine which one works best for your application by measuring performance with each.(Most users should start at **`-O2`** (default) before trying more advanced optimizations. Next, **`-O3`** for loop intensive applications.)
3. Fine-tune performance to target Intel 64-based systems with processor-specific options. (Example is **`-xsse4.2`**. Alternatively, you can use **`-xhost`** which will use the most advanced instruction set for the processor on which you compiled)
4. Add interprocedural optimization **`-ipo`** and/or profile-guided optimization (**`-prof-gen`** and **`-prof-use`**) , then measure performance again to determine whether your application benefits from one or both of them
5. Optimize your application for vector and parallel execution on multi-threaded, multi-core and multi-processor system
6. Use specific tools to help you identify serial and parallel performance "hotspots" so that you know which specific parts of your application could benefit from further tuning

## Compiler: what it can do

- It performs these code modifications
  - Register allocation
  - Register spilling
  - Copy propagation
  - Code motion
  - Dead and redundant code removal
  - Common subexpression elimination
  - Strength reduction
  - Inlining
  - Index reordering
  - Loop pipelining , unrolling, merging
  - Cache blocking
  - . . .
- Everything to maximize performances!!

## Compiler: what it cannot do

- Global optimization of "big" source code, unless switch on interprocedural analisys (IPO) but it is very time consuming . . .
- Understand and resolve complex indirect addressing
- Strenght reduction (with non-integer values)
- Common subexpression elimination through function calls
- Unrolling, Merging, Blocking with:
  - functions/subroutine calls
  - I/O statement
- Implicit function inlining
- Knowing at run-time variabile's values

## Optimizations: levels

- All compilers have "predefined" optimization levels **`-O<n>`**
  - with **`n`** from 0 a 3 (IBM up to 5)
- Usually :
  - **`-O0`**: no optimization is performed, simple translation (tu use with **`-g`** for debugging)
  - **`-O`**: default value
  - **`-O1`**: basic optimizations
  - **`-O2`**: memory-intensive optimizations
  - **`-O3`**: more aggressive optimizations, it can alter the instruction order
  - Some compilers have **`-fast`** option (**`-O3`** plus more options)

## Intel compiler: -O0 option

- Before doing any optimization you should ensure that the unoptimized version of your code works.
- On very rare occasions optimizing can change the intended behavior of your applications, so it is always best to start from a program you know builds and works correctly.
- Building with Optimisation Disabled
  - Code is not re-ordered
  - Improves visibility when using profiling tools.
    - You should use this option when looking for threading errors!
  - The code is usually much slower
  - The binaries are usually much bigger
  - **`-g`** produce debug information (can be used with **`-O1`**,**`-O2`**.**`-O3`**, etc.)

## Intel compiler: -O1 option

Optimize for speed and size

- This option is very similar to **–O2** except that it omits optimizations that tend to increase object code size , such as the in-lining of functions. Generally useful where memory paging due to large code size is a problem, such as server and database applications.
- Auto-vectorization is not turned on , even if it is invoked individually by its fine grained switch **–vec**. However, at **–O1** the vectorization associated with array notation is enableded.

## Intel compiler: -O2 option

Optimize for maximum speed

- This option creates faster code in most cases.
- Optimizations include scalar optimizations
- inlining and some other interprocedural optimizations between functions/subroutines in the same source file
- vectorization
- limited versions of a few other loop optimizations, such as loop versioning and unrolling that facilitate vectorization.

## Intel compiler: -O3 option

Optimizes for further speed increases

- This includes all the -O2 optimizations, as well as other high-level optimizations
- including more aggressive strategies such as scalar replacement, data pre-fetching, and loop optimization, among others
- It is particularly recommended for applications that have loops that do many floating - point calculations or process large data sets. These aggressive optimizations may occasionally slow down other types of applications compared to **–O2**

## Optimization Report

- The compiler can produce reports on what optimizations were carried out. By default, these reports are disabled

```
-opt-report[n]      n=0(none),1(min),2(med),3(max)
-opt-report-file<file>
-opt-report-routine<routine>
-opt-report-phase<phase>
```

- one or more `*.optrpt` file are generated
- To know the difference phases.

```
icc(ifort,icpc) -qopt-report-help
```

## Optimization Report:example

```
ifort -O3 -opt-report
```

```
....
LOOP BEGIN at mm.f90(44,10)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
....
LOOP BEGIN at mm.f90(65,5)
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 2 3 1 )
....
LOOP BEGIN at mm.f90(66,8)
   remark #25442: blocked by 128   (pre-vector)
   remark #25440: unrolled and jammed by 4   (pre-vector)
....
```

## Different operations, different latencies

For a CPU different operations could present different latencies

- Sum: few clock cycles
- Product: few clock cycles
- Sum+Product: few clock cycles
- Division: many clock cycle ($O(10)$)
- Sin,Cos: many many clock cycle ($O(100)$)
- exp,pow: many many clock cycle ($O(100)$)
- I/O operations: many many many clock cycles ($o(1000 - 10000)$)

## Outline

## What is Vectorization?

- Hardware Perspective: Specialized instructions, registers, or functional units to allow in-core parallelism for operations on arrays (vectors) of data.
- Compiler Perspective: Determine how and when it is possible to express computations in terms of vector instructions
- User Perspective: Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible.

## Processor Specifing Options

- When you use the compiler out of the box (that is, the default behavior), auto-vectorization is enabled, supporting SSE2 instructions.
- You can enhance the optimization of auto-vectorization beyond the default behavior by explicitly using some additional options.
- If you run an application on a CPU that does not support the level of auto-vectorization you chose when it was built, the program will fail to start. The following error message will be displayed:

```
This program was not built to run on the processor in your system}.
```

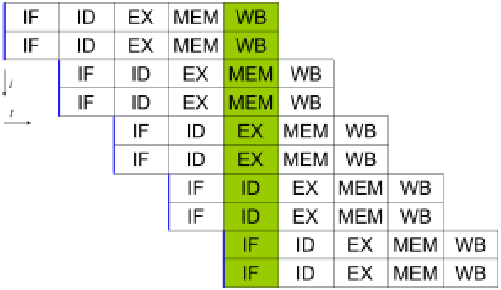
- You can get the compiler to add multiple paths in your code so that your code can run on both lower- and higher-spec CPUs, thus avoiding the risk of getting an error message or program abort

## What Happened To Clock Speed?

- Everyone loves to misquote Moore’s Law:
  - "CPU speed doubles every 18 months."
- Correct formulation:
  - "Available on-die transistor density doubles every 18 months."
- For a while, this meant easy increases in clock speed
- Greater transistor density means more logic space on a chip

## Clock Speed Wasn’t Everything

- Chip designers increased performance by adding sophisticated features to improve code efficiency.
- Branch-prediction hardware.
- Out-of-order and speculative execution.
- Superscalar chips.
- Superscalar chips look like conventional single-core chips to the OS.
- Behind the scenes, they use parallel instruction pipelines to (potentially) issue multiple instructions simultaneously.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| IF | ID | EX | MEM | WB | | | | |
| IF | ID | EX | MEM | WB | | | | |
| | IF | ID | EX | MEM | WB | | | |
| | IF | ID | EX | MEM | WB | | | |
| | | IF | ID | EX | MEM | WB | | |
| | | IF | ID | EX | MEM | WB | | |
| | | | IF | ID | EX | MEM | WB | |
| | | | IF | ID | EX | MEM | WB | |
| | | | | IF | ID | EX | MEM | WB |
| | | | | IF | ID | EX | MEM | WB |

## SIMD Parallelism

- CPU designers had, in fact, been exposing explicit parallelism for a while.
- MMX is an early example of a SIMD (Single Instruction Multiple Data) instruction set.
  - Also called a vector instruction set.
- Normal, scalar instructions operate on single items in memory.
  - Can be different size in terms of bytes, of course.
  - Standard x86 arithmetic instructions are scalar. (ADD, SUB, etc.)
- Vector instructions operate on packed vectors in memory.
- A packed vector is conceptually just a small array of values in memory.
  - A 128-bit vector can be two doubles, four floats, four int32s, etc.
  - The elements of a 128-bit single vector can be thought of as v[0], v[1], v[2], and v[3].

## SIMD Parallelism

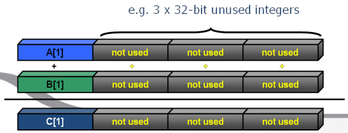
- Vector instructions are handled by an additional unit in the CPU core, called something like a vector arithmetic unit.
- If used to their potential, they can allow you to perform the same operation on multiple pieces of data in a single instruction.
  - Single-Instruction, Multiple Data parallelism.
  - Your algorithm may not be amenable to this...
  - ... But lots are. (Spatially-local inner loops over arrays are a classic.)
- It has traditionally been hard for the compiler to vectorise code efficiently, except in trivial cases.
  - It would suck to have to write in assembly to use vector instructions...

## Vector units

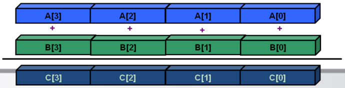
- Auto-vectorization is transforming sequential code to exploit the SIMD (Single Instruction Multiple Data) instructions within the processor to speed up execution times
- Vector Units performs parallel floating/integer point operations on dedicate SIMD units
  - Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX
- Think vectorization in terms of loop unrolling
- Example: summing 2 arrays of 4 elements in one single instruction

```
C(0) = A(0) + B(0)
C(1) = A(1) + B(1)
C(2) = A(2) + B(2)
C(3) = A(3) + B(3)
```

no vectorization



vectorization



## SIMD - evolution

- SSE: 128 bit register (Intel Core - AMD Opteron)
  - 4 floating/integer operations in single precision
  - 2 floating/integer operations in double precision
- AVX: 256 bit register (Intel Sandy Bridge - AMD Bulldozer)
  - 8 floating/integer operations in single precision
  - 4 floating/integer operations in double precision
- MIC: 512 bit register (Intel Knights Corner - 2013)
  - 16 floating/integer operations in single precision
  - 8 floating/integer operations in double precision

## Vector-aware coding

- Know what makes vectorizable at all
  - "for" loops (in C) or "do" loops (in fortran) that meet certain constraints
- Know where vectorization will help
- Evaluate compiler output
  - Is it really vectorizing where you think it should?
- Evaluate execution performance
  - Compare to theoretical speedup
- Know data access patterns to maximize efficiency
- Implement fixes: directives, compilation flags, and code changes
  - Remove constructs that make vectorization impossible/impractical
  - Encourage and (or) force vectorization when compiler doesn’t, but should
  - Better memory access patterns

## Writing Vector Loops

- Basic requirements of vectorizable loops:
  - Countable at runtime
    - Number of loop iterations is known before loop executes
    - No conditional termination (break statements)
  - Have single control flow
    - No Switch statements
    - 'if' statements are allowable when they can be implemented as masked assignments
  - Must be the innermost loop if nested
    - Compiler may reverse loop order as an optimization!
  - No function calls
    - Basic math is allowed: pow(), sqrt(), sin(), etc
    - Some inline functions allowed

## Tuning on Auto-Vectorization

- Auto-vectorization is included implicitly within some of the general optimization options, and implicitly switched off by others.
- It can be further controlled by the auto-vectorization option **`–vec`**.
- Normally the only reason you would use the **`–vec`** option would be to disable(using **`–novec`**) is for the purposes of testing.
  - The general options **`–O2`** , **`–O3`** , and **`–Ox`** turn on auto-vectorization. You can override these options by placing the option **`–novec`** directly on the compiler’s command line.
  - The general options **`–O0`** and **`–O1`** turn off auto-vectorization, even if it is specifically set on the compiler’s command line by using the **`–vec`** option.

## Option -x

| Option | Description |
|---|---|
| CORE-AVX2 | AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions |
| CORE-AVX-I | RDND instr, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions |
| AVX | AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions |
| SSE4.2 | SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel CoreTM i7 processors. SSE4 .1, SSSE3, SSE3, SSE2, and SSE. May optimize for the Intel CoreTM processor family |
| SSE4.1 | SSE4 Vectorizing Compiler and Media Accelerator, SSSE3, SSE3, SSE2, and SSE . May optimize for Intel 45nm Hi-k next generation Intel CoreTM microarchitecture |
| SSSE3_ATOM | MOVBE , (depending on -minstruction ), SSSE3, SSE3, SSE2, and SSE . Optimizes for the Intel AtomTM processor and Intel Centrino AtomTM Processor Technology |
| SSSE3 | SSSE3, SSE3, SSE2, and SSE. Optimizes for the Intel CoreTM microarchitecture |
| SSE3 | SSE3, SSE2, and SSE. Optimizes for the enhanced Pentium M processor microarchitecture and Intel NetBurst microarchitecture |
| SSE2 | SSE2 and SSE . Optimizes for the Intel NetBurst microarchitecture |

## When vectorization fails

- Not Inner Loop: only the inner loop of a nested loop may be vectorized, unless some previous optimization has produced a reduced nest level. On some occasions the compiler can vectorize an outer loop, but obviously this message will not then be generated.
- Low trip count:The loop does not have sufficient iterations for vectorization to be worthwhile.
- Vectorization possible but seems inefficient: the compiler has concluded that vectorizing the loop would not improve performance. You can override this by placing **`#pragma vector always`** before the loop in question
- Contains unvectorizable statement: certain statements, such as those involving switch and printf , cannot be vectorized

## When vectorization fails

- Subscript too complex: an array subscript may be too complicated for the compiler to handle. You should always try to use simplified subscript expressions
- Condition may protect exception: when the compiler tries to vectorize a loop containing an if statement, it typically evaluates the RHS expressions for all values of the loop index, but only makes the final assignment in those cases where the conditional evaluates to TRUE. In some cases, the compiler may not vectorize because the condition may be protecting against accessing an illegal memory address. You can use the **`#pragma ivdep`** to reassure the compiler that the conditional is not protecting against a memory exception in such cases.
- Unsupported loop Structure: loops that do not fulfill the requirements of countability, single entry and exit, and so on, may generate these messages

## When vectorization fails

- Operator unsuited for vectorization: Certain operators, such as the % (modulus) operator, cannot be vectorized
- Non-unit stride used: non-contiguous memory access.
- Existence of vector dependence: vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation

https://software.intel.com/en-us/articles/vectorization-diagnostics-for-intelr-c-compiler-150-and-above

## Strided access

- Fastest usage pattern is "stride 1": perfectly sequential
- Best performance when CPU can load L1 cache from memory in bulk, sequential manner
- Stride 1 constructs:
  - Iterating Structs of arrays vs arrays of structs
  - Multi dimensional array:

  - Fortran: stride 1 on "inner" dimension
  - C / C++: Stride 1 on "outer" dimension

```
do j = 1,n; do i=1,n
  a(i,j)=b
enddo; endo
```

```
for(j=0;j<n;j++)
for(i=0;i<n;i++)
  a[j][i]=b[j][i]*s;
```

## Data Dependencies

- Read after write: When a variable is written in one iteration and read in a subsequent iteration, also known as a flow dependency:

```
A[0]=0;
for (j=1; j<MAX; j++)
A[j]=A[j-1]+1;
// this is equivalent to:
A[1]=A[0]+1; A[2]=A[1]+1; A[3]=A[2]+1; A[4]=A[3]+1;
```

- The above loop cannot be vectorized safely because if the first two iterations are executed simultaneously by a SIMD instruction, the value of A[1] may be used by the second iteration before it has been calculated by the first iteration which could lead to incorrect results.

## Data Dependencies

- write-after-read: When a variable is read in one iteration and written in a subsequent iteration, sometimes also known as an anti-dependency

```
for (j=1; j<MAX; j++)
A[j-1]=A[j]+1;
// this is equivalent to:
A[0]=A[1]+1; A[1]=A[2]+1; A[2]=A[3]+1; A[3]=A[4]+1;
```

- This is not safe for general parallel execution, since the iteration with the write may execute before the iteration with the read. However, for vectorization, no iteration with a higher value of j can complete before an iteration with a lower value of j, and so vectorization is safe (i.e., gives the same result as non- vectorized code) in this case.

## Data Dependencies

- Read-after-read: These situations aren’t really dependencies, and do not prevent vectorization or parallel execution. If a variable is not written, it does not matter how often it is read.
- Write-after-write: Otherwise known as ’output’, dependencies, where the same variable is written to in more than one iteration, are in general unsafe for parallel execution, including vectorization.

## Help to auto-vectoriser

- Change data layout - avoid non-unit strides
- Use the restrict key word (C  C++)
- Use array notation
- Use **`#pragma ivdep`**
- Use **`#pragma vector always`**
- Use **`#pragma simd`**
- Use elemental functions

## Vectorization: arrays and restrict

- Writing "clean" code is a good starting point to have the code vectorized
  - Prefer array indexing instead of explicit pointer arithmetic
  - Use restrict keyword to tell the compiler that there is no array aliasing
- The use of the restrict keyword in pointer declarations informs the compiler that it can assume that during the lifetime of the pointer only this single pointer has access to the data addressed by it that is, no other pointers or arrays will use the same data space. Normally, it is adequate to just restrict pointers associated with the left-hand side of any assignment statement, as in the following code example. Without the restrict keyword, the code will not vectorize.

```
void f(int n, float *x, float *y, float *restrict z, float *d1, float *d2)
{
for (int i = 0; i < n; i++)
z[i] = x[i] + y[i]-(d1[i]*d2[i]);
}
```

## Vectorization: array notation

- Using array notation is a good way to guarantee the compiler that the iterations are independent
  - In Fortran this is consistent with the language array syntax
    a(1:N) = b(1:N) + c(1:N)
  - In C the array notation is provided by Intel Cilk Plus
    a[1:N] = b[1:N] + c[1:N]
- Beware:
  - The first value represents the lower bound for both languages
  - But the second value is the upper bound in Fortran whereas it is the length in C
  - An optional third value is the stride both in Fortran and in C
  - Multidimensional arrays supported, too

## Algorithm & Vectorization

- Different algorithm for the same problem could be vectorazable or not
  - Gauss-Seidel: data dependencies, can not be vectorized

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i][j] = w0 * a[i][j] +
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

  - Jacobi: no data dependence, can be vectorized

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     b[i][j] = w0*a[i][j] +
       w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     a[i][j] = b[i][j];
```

## Optimization & Vectorization

- “coding tricks” can inhibit vectorization
  - can be vectorized

```
for( i = 0; i < n-1; ++i ){
   b[i] = a[i] + a[i+1];
}
```

  - can not be vectorized

```
x = a[0];
for( i = 0; i < n-1; ++i ){
  y = a[i+1];
  b[i] = x + y;
  x = y;
}
```

## Help to auto-vectoriser:directives

- **`#pragma ivdep`**: this tells the compiler to ignore vector dependencies in the loop that immediately follows the directive/pragma. However, this is just a recommendataion, and the compiler will not vectorize the loop if there is a clear dependency. Use **`#pragma ivdep`** only when you know that the assumed loop dependencies are safe to ignore.

```
#pragma ivdep
for(int i = 0;i < m; i++)
a[i] = a[i + k] * c;
```

# Help to auto-vectoriser:directives

- **#pragma vector**: This overrides default heuristics for vectorization of the loop. You can provide a clause for a specific task. For example, it will try to vectorize the immediately-following loop that the compiler normally would not vectorize because of a performance efficiency reason. As another example, **#pragma vector aligned** will inform that all data in the loop are aligned at a certain byte boundary so that aligned load or store SSE or AVX instructions can be used.This directive may be ignored by the compiler when it thinks that there is a data dependency in the loop.
- **#pragma novector**: This tells the compiler to disable vectorizaton for the loop that follows

```
void vec(int *a, int *b, int m)
{
#pragma vector
for(int i = 0; i <= m; i++)
a[32*i] = b[99*i];
}
```

- You can use **#pragma vector always** to override any efficiency heuristics during the decision to vectorize or not, and to vectorize non-unit strides or unaligned memory accesses. The loop will be vectorized only if it is safe to do so. The outer loop of a nest of loops will not be vectorized, even if **#pragma vector always** is placed before it

## Help to auto-vectoriser:directives

- **`#pragma simd`**: This is used to enforce vectorization for a loop that the compiler doesn't auto-vectorize even with the use of vectorization hints such as **`#pragma vector always`** or **`#pragma ivdep`**. Because of this nature of enforcement, it is called user-mandated vectorization. A clause can be accompanied to give a more specific direction (see documentation).

```
#pragma simd private(b)
for( i=0; i<MAXIMUS; i++ )
{
if( a[i] > 0 )
{
b = a[i];
a[i] = 1.0/a[i];
}
if( a[i] > 1 )a[i] += b;
}
```

## Elemental function

- Elemental functions are user-defined functions that can be used to operate on each element of an array. The three steps to writing a function are as follows:
  1. Write the function using normal scalar operations. Restrictions exist on what kind of code can be included. Specifically, you must not include loops, switch statements, goto , setjmp , longjmp , function calls (except to other elemental functions or math library intrinsics).
  2. Decorate the function name with **`__declspec(vector)`** .
  3. Call the function with vector arguments.
- In the following code snippet, the multwo function is applied to each element of array A . At optimization levels -O2 and above, the compiler generates vectorized code for the example.

```
int __declspec(vector) multwo(int i){return i * 2;}
int main()
{
int A[100];
A[:] = 1;
for (int i = 0 ; i < 100; i++)
multwo(A[i]);
}
```

## Consistency of SIMD results

Two issues can effect reproducibility: because the order of the calculations can change

- Alignment
- Parallelism

- Try to align to the SIMD register size
  - MMX: 8 Bytes;
  - SSE2: 16 bytes,
  - AVX: 32 bytes
  - MIC: 64 bytes
- Try to align blocks of data to cacheline size - ie 64 bytes

## Compiler Intrinsics for Alignment

- `__declspec(align(base, [offset]))` Instructs the compiler to create the variable so that it is aligned on an "base"-byte boundary, with an "offset" (Default=0) in bytes from that boundary
- `void* _mm_malloc (int size, int n)` Instructs the compiler to create a pointer to memory such that the pointer is aligned on an n-byte boundary
- `#pragma vector aligned | unaligned` Use aligned or unaligned loads and stores for vector accesses
- `__assume_aligned(a,n)` Instructs the compiler to assume that array a is aligned on an n-byte boundary

## Vectorized loops?

```
-vec-report[N]  (deprecated)
-qopt-report[=N]  -qopt-report-phase=vec
```

| N | Diagnostic Messages |
|---|---|
| 0 | No diagnostic messages; same as not using switch and thus default |
| 1 | Tells the vectorizer to report on vectorized loops. |
| 2 | Tells the vectorizer to report on vectorized and non-vectorized loops. |
| 3 | Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences. |
| 4 | Tells the vectorizer to report on non-vectorized loops. |
| 5 | Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized. |
| 6 | Tells the vectorizer to use greater detail when reporting on vectorized and non-vectorized loops and any proven or assumed data dependences. |
| 7 | Tells the vectorizer to emit vector code quality message ids and corresponding data values for vectorized loops. It provides information such as the expected speedup, memory access patterns, and the number of vector idioms for vectorized loops. |

## Vectorization Report:example

```
ifort -O3 -qopt-report=5
```

```
 LOOP BEGIN at matmat.F90(51,1)
      remark #25427: Loop Statements Reordered
      remark #15389: vectorization support: reference C has unaligned access
      remark #15389: vectorization support: reference B has unaligned access
[ matmat.F90(50,1) ]
      remark #15389: vectorization support: reference A has unaligned access
[ matmat.F90(49,1) ]
      remark #15381: vectorization support: unaligned access used inside loop body
[ matmat.F90(49,1) ]
      remark #15301: PERMUTED LOOP WAS VECTORIZED
      remark #15451: unmasked unaligned unit stride stores: 3
      remark #15475: --- begin vector loop cost summary ---
      remark #15476: scalar loop cost: 229
      remark #15477: vector loop cost: 43.750
      remark #15478: estimated potential speedup: 5.210
      remark #15479: lightweight vector operations: 24
      remark #15480: medium-overhead vector operations: 2
      remark #15481: heavy-overhead vector operations: 1
      remark #15482: vectorized math library calls: 2
      remark #15487: type converts: 2
      remark #15488: --- end vector loop cost summary ---
      remark #25015: Estimate of max trip count of loop=28
  LOOP END
```

## Vectorization:conclusion

- Vectorization occurs in tight loops "automatically" by the compiler
- Need to know where vectorization should occur, and verify that compiler is doing that.
- Need to know if a compiler's failure to vectorize is legitimate
  - Fix code if so, use **`#pragma`** if not
- Need to be aware of caching and data access issues
  - Very fast vector units need to be well fed

## Will it vectorize?

Assume a, b and x are known to be independent

## Will it vectorize?

Assume a, b and x are known to be independent

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

## Will it vectorize?

Assume a, b and x are known to be independent

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

Vectorizes if $n \leq 0$; doesn't vectorize if $n > 0$ and small; may vectorize if $n \geq$ number of elements in a vector register

## Will it vectorize?

Assume a, b and x are known to be independent

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

Vectorizes if n$\leq$ 0; doesn't vectorize if n $>$ 0 and small; may vectorize if n$\geq$ number of elements in a vector register

```
for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
```

## Will it vectorize?

Assume a, b and x are known to be independent

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

Vectorizes if n$\leq$ 0; doesn't vectorize if n $>$ 0 and small; may vectorize if n$\geq$ number of elements in a vector register

```
for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
```

Unlikely to vectorize because of non-unit stride (inefficient)

## Will it vectorize?

Assume a, b and x are known to be independent

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

Vectorizes if n$\leq$ 0; doesn't vectorize if n $>$ 0 and small; may vectorize if n$\geq$ number of elements in a vector register

```
for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
```

Unlikely to vectorize because of non-unit stride (inefficient)

```
for (int j=0; j<SIZE; j++) {
  for (int i=0; i<SIZE; i++) b[i] += a[i][j] * x[j];
```

## Will it vectorize?

Assume a, b and x are known to be independent

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

Vectorizes if n$\leq$ 0; doesn't vectorize if n $>$ 0 and small; may vectorize if n$\geq$ number of elements in a vector register

```
for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
```

Unlikely to vectorize because of non-unit stride (inefficient)

```
for (int j=0; j<SIZE; j++) {
  for (int i=0; i<SIZE; i++) b[i] += a[i][j] * x[j];
```

Doesn't vectorize because of non-unit stride, unless compiler can first interchange the order of the loops. (Here, it can)

## Will it vectorize?

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

## Will it vectorize?

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

Doesn’t vectorize because of indirect addressing (non-unit stride), would be inefficient. If x[index[i]] appeared on the LHS, this would also introduce potential dependency (index[i] might have the same value for different values of i)

## Will it vectorize?

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

Doesn't vectorize because of indirect addressing (non-unit stride), would be inefficient. If x[index[i]] appeared on the LHS, this would also introduce potential dependency (index[i] might have the same value for different values of i)

```
for (j=1; j<MAX; j++) sum = sum + a[j]*b[j]
```

## Will it vectorize?

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

Doesn’t vectorize because of indirect addressing (non-unit stride), would be inefficient. If x[index[i]] appeared on the LHS, this would also introduce potential dependency (index[i] might have the same value for different values of i)

```
for (j=1; j<MAX; j++) sum = sum + a[j]*b[j]
```

Reductions such as this will vectorize. The compiler accumulates a number of partial sums (equal to the number of elements in a vector register), and adds them together at the end of the loop.

## Will it vectorize?

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

Doesn't vectorize because of indirect addressing (non-unit stride), would be inefficient. If x[index[i]] appeared on the LHS, this would also introduce potential dependency (index[i] might have the same value for different values of i)

```
for (j=1; j<MAX; j++) sum = sum + a[j]*b[j]
```

Reductions such as this will vectorize. The compiler accumulates a number of partial sums (equal to the number of elements in a vector register), and adds them together at the end of the loop.

```
for (int i=0; i<length; i++) {
 float s = b[i]*b[i]-4.f*a[i]*c[i];
  if ( s >= 0 ) x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
}
```

## Will it vectorize?

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

Doesn’t vectorize because of indirect addressing (non-unit stride), would be inefficient. If x[index[i]] appeared on the LHS, this would also introduce potential dependency (index[i] might have the same value for different values of i)

```
for (j=1; j<MAX; j++) sum = sum + a[j]*b[j]
```

Reductions such as this will vectorize. The compiler accumulates a number of partial sums (equal to the number of elements in a vector register), and adds them together at the end of the loop.

```
for (int i=0; i<length; i++) {
 float s = b[i]*b[i]-4.f*a[i]*c[i];
  if ( s >= 0 ) x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
}
```

This will vectorize. Neither "if" masks nor most simple math intrinsic functions prevent vectorization. But with SSE, the sqrt is evaluated speculatively. If FP exceptions are unmasked, this may trap if s<0, despite the if clause. With AVX, there is a real hardware mask, so the sqrt will never be evaluated if s<0, and no exception will be trapped.

## Interprocedural Optimization

- O2 and O3 activate "almost" file-local IPO (-ip)
  - Only a very few, time-consuming IP-optimizations are not done but for most codes, -ip is not adding anything
  - Switch -ip-no-inlining disables in-lining
- IPO extends compilation time and memory usage
  - See compiler manual when running into limitations
- In-lining of functions is most important feature of IPO but there is much more
  - Inter-procedural constant propagation
  - MOD/REF analysis (for dependence analysis)
  - Routine attribute propagation
  - Dead code elimination
  - Induction variable recognition
  - ...many, many more
- IPO improves auto-vectorization results of the sample application
- IPO brings some new ’tricky-to-find’ auto-vectorization opportunities.

## Profile Guided Optimization

- All the optimization methods described have been static
- Static analysis is good, but it leaves many questions open
- PGO uses a dynamic approach
- One or more runs are made on unoptimized code with typical data, collecting profi le information each time
- This profile information is then used with optimizations set to create a final executable

## Profile Guided Optimization:benefits

- More accurate branch prediction
- Basic code block movements to improve instruction cache behavior
- Better decision of functions to inline
- Can optimize function ordering
- Switch-statement optimizer
- Better vectorization decisions

## Profile Guided Optimization

1. Compile your unoptimized code with PGO

```
icc -prof-gen prog.c
```

   It produces an executable with instrumented information included

2. Make multiple runs with different sets of typical data input; each run automatically produces a dynamic information ( .dyn ) file

```
./a.out
```

   If you change your code during test runs, you need to remove any existing .dyn files before creating others with the new code

3. Finally, switch on all your desired optimizations and do a feedback compile with PGO to produce a fi nal PGO executable

```
icc -prof-use prog.c
```

   In addition to the optimized executable, the compiler produces a pgopti.dpi file. You typically specify the default optimizations, -02 , for phase 1, and specify more advanced optimizations, -ipo , for phase 3. For example, the example shown above used -O2 in phase 1 and -ipo in phase 3

   - -opt-report-phase=pgo Creates a PGO report

## Outline

## Performance and parallelism

- In principle the main advantage of using Intel MIC technology with respect to other coprocessors is the simplicity of the porting
  - Programmers may compile their source codes based on common HPC languages (Fortran/ C / C++) specifying MIC as the target architecture (native mode)
- Is it enough to achieve good performances? By the way, why offload?
- Usually not, parallel programming is not easy
  - A general need is to expose parallelism

## GPU vs MIC

- GPU paradigms (e.g. CUDA):
  - Despite the sometimes significant effort required to port the codes...
  - ...are designed to force the programmer to expose (or even create if needed) parallelism
- Programming Intel MIC
  - The optimization techniques are not far from those devised for the common CPUs
  - As in that case, achieving optimal performance is far from being straightforward
- What about device maturity?

## Intel Xeon Phi very basic features

- Let us recall 3 basic features of current Intel Xeon Phi:
- Peak performance originates from "many slow but vectorizable cores"

  clock frequency x n. cores x n. lanes x 2 FMA Flops/cycle
  1.091 GHz x 61 cores x 16 lanes x 2 = 2129.6 Gflops/cycle
  1.091 GHz x 61 cores x 8 lanes x 2 = 1064.8 Gflops/cycle

- Bandwidth is (of course) limited, caches and alignment matter
- The card is not a replacement for the host processor. It is a coprocessor providing optimal power efficiency

## Optimization key points

In general terms, an application must fulfill three requirements to efficiently run on a MIC

1. Highly vectorizable, the cores must be able to exploit the vector units. The penalty when the code cannot be vectorized is very high
2. high scalability, to exploit all MIC multi-threaded cores: scalability up to 240 processors (processes/threads) running on a single MIC, and even higher running on multiple MIC
3. ability of hiding I/O communications with the host processors and, in general, with other hosts or coprocessors

## Outline

## Introduction

- The programming model and compiler make it easy to develop or port code to run on a system with an Intel Xeon Phi coprocessor
- Full integration into both C/C++ and Fortran
- Enables use of Intel's optimizing compilers on both host and coprocessor
  - Vectorization
  - Parallel programming with TBB, Intel Cilk Plus, OpenMP, MPI, OpenCL
- Enables co-operative processing between host and coprocessor

## Programming models

- An Intel Xeon Phi coprocessor is accessed via the host system, but may be programmed either as a coprocessor(s) or as an autonomous processor.
- The appropriate model may depend on application and context.

- Host only
- Coprocessor only "native"
  - Target Code: Highly parallel (threaded and vectorized) throughout.
  - Potential Bottleneck: Serial/scalar code
- Offload with LEO
  - Target Code: Mostly serial, but with expensive parallel regions
  - Potential Bottleneck: Data transfers.
- Symmetric with MPI
  - Target Code: Highly parallel and performs well on both platforms.
  - Potential Bottleneck: Load imbalance.

## Programming models

- MPI
  - Used for "native" and "symmetric" execution.
  - Can launch ranks across processors and coprocessors.
- OpenMP
  - Used for "native", "offload" and "symmetric" execution.
  - Newest standard (4.0) supports "target" syntax for offloading.
- Many real-life HPC codes use a native MPI/OpenMP hybrid
  - Balance task granularity by tuning combination of ranks/threads. (e.g.16 MPI ranks x 15 OpenMP threads)

## Native mode

PRO

- it is a cross-compiling mode
- just add **`-mmic`**, login and execute
- use well known OpenMP and MPI

CONS

- very slow I/O
- poor single thread performance
- only suitable for highly parallel codes (cfr Amdahl)
- CPU unused

## Native mode

Native model may be appropriate if app:

- Contains very little serial processing
- Has a modest memory footprint
- Has a very complex code structure and/or
- does not have well-identified hot kernels than can be offloaded without substantial data transfer overhead
- Does not perform extensive Input Output

## What is offloading

- Code is instrumented with directives
- Compiler creates a CPU binary and a MIC binary for offloaded code block.
- Loader places both binaries in a single file
- During CPU execution of the application an encountered offload code block is executed on a coprocessor (through runtime), subject to the constraints of the target specifier...
- When the coprocessor is finished, the CPU resumes executing the CPU part of the code.

## The Offload Mechanism

The basic operations of an offload rely on interaction with the runtime to:

- Detect a target phi coprocessor
- Allocate memory space on the coprocessor
- Transfer data from the host to the coprocessor
- Execute offload binary on coprocessor
- Transfer data from the coprocessor back to the host
- Deallocate space on coprocessor

## Offload model

Offload model may be appropriate because:

- Better serial processing
- More memory
- Better file access
- Makes greater use of available resources

## Offload model

Offload model may be appropriate because:

- Better serial processing
- More memory
- Better file access
- Makes greater use of available resources

1. Try offloading compute-intensive section
   If it isn't threaded, make it threaded
2. Optimize data transfers
3. Split calculation & use asynchronous mechanisms

## Offload model

- Intel Xeon Phi supports two offload models:
  - Explicit:
    Data transfers from host to/from coprocessor are initiated by programmer
  - Implicit:
    Data is (virtually) shared (VSHM) between host and coprocessor
- Also called LEO (Language Extensions for Offload)

## Offload model

| | via Explicit Data | via Implicit data |
|---|---|---|
| Meaning ... | Emulate shared data by copying back and forth at point of offload | Maintain coherence in a range of virtual addresses on host and Phi, automatically in software |
| Languaga Support | Fortran, C, C++ | C, C++ |
| Syntax | Pragmas /Directives:<br>!dir$ [omp] offload in Fortran<br>#pragma offload in C C++ | Keywords:<br>_Cilk_shared and<br>_Cilk_offload |
| Used for ... | Offloads that transfer contiguous blocks of data | Offloads that transfer all or parts of complex data structures, or many small pieces of data |

## Directives

- Directives can be inserted before code blocks and functions to run the code on the Xeon Phi Coprocessor (the "MIC").
  - No recoding required. (Optimization may require some changes.)
  - Directives are simple, but more "details" (specifiers) can be used for optimal performance.
  - Data must be moved to the MIC
  - For large amounts of data:
    - Amortize with large amounts of work.
    - Keep data resident ("persistent").
    - Move data asynchronously.

## LEO: pragmas

Offload pragma/directive for data marshalling

- **`#pragma offload <clauses>`** in C/C++
  Offloads the following OpenMP block or Intel Cilk Plus construct or function call or compound statement
- **`!dir$ offload <clauses>`** in Fortran
  Offloads the following OpenMP block or subroutine/function call
- **`!dir$ offload <clauses>..`**
  **`!dir$ end offload`** to offload other block of code
- Offloaded data must be scalars, arrays, bit-wise copyable structs (C/C++) or derived types (Fortran)
  - no embedded pointers or allocatable arrays
  - Excludes all but simplest C++ classes
  - Excludes most Fortran 2003 object-oriented constructs
  - All data types can be used within the target code
  - Data copy is explicit

## Explicit Offload

Explicit offloading requires user to manage data persistence:

- Data/Functions marked as...
  - C C++
    ```
     #pragma offload_attribute(push, target(mic))
    ...
    #pragma offload_attribute(pop))
    _attribute__((target(mic)))
    ```
  - Fortran:
    ```
    !DIR$ OPTIONS /OFFLOAD_ATTRIBUTE_TARGET=mic
    !DIR$ ATTRIBUTES OFFLOAD:mic :: <subroutine>
    ```
    Will exist on both the host and target systems and copied between host and target when referenced.
- Named targets
  - `target(mic)` : target(mic)
  - `target(mic:n)` : explicitly name the logical card number n

## Example: C

```
__attribute__ ((target(mic)))
void foo(){
printf("Hello MIC\n");
}

int main(){
#pragma offload target(mic)
foo();
return 0;
}
```

## Example: Fortran

```
!dir$ attributes &
!dir$ offload:mic ::hello
subroutine hello
write(*,*)"Hello MIC"
end subroutine

program main
!dir$ attributes &
!dir$ offload:mic :: hello
!dir$ offload begin target (mic)
call hello()
!dir$ end offload
end program
```

## Compiler Usage

1. Insert Offload Directive
2. Compile with Intel Compiler
   - How to turn off offloading: use **`-no-offload`** option
   - Activate reporting **`-opt-report-phase:offload`**
   - **`-offload-attribute-target=mic`** flag all global variables and functions for offload
   - For offload models, pass options via **`-offload-option`**

Example

```
icc test.c -O2 -offload-option,mic,compiler,"-O3 -vec-report3"
```

## Offload Performance

- By default when a program performs the first **`#pragma offload`** all MIC devices assigned to the program are initialized
- Initialization consists of loading the MIC program on to each device, setting up a data transfer pipeline between CPU and the device and creating a MIC thread to handle offload requests from the CPU thread
- These activities take time
  - Do not place the first offload within a timing measurement
  - Exclude this one-time overhead by performing a dummy offload to the device
  - Alternatively, use the **`OFFLOAD_INIT=on_start`** environment variable setting to pre-initialize all available MIC devices before starting the main program

## Data Transfer

- Automatically detected and transferred as INOUT
  - Named arrays in static scope
  - Scalars in static scope
- User can override automatic transfer with explicit IN/OUT/INOUT clauses
- Not automatically transferred
  - Memory pointed to by pointers (This also needs a length parameter)
  - Global variables used in functions called within the offloaded construct
  - User must specify IN/OUT/INOUT clauses

https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features

## Explicit Offload

- Pure data transfer:
  - **`#pragma offload_transfer target(mic0)`**
  - **`!DIR$ offload_transfer target(mic0)`**
  - Asynchronous transfers:
    Clauses **`signal(<id>)`** & **`wait(<id>)`**
- Offloading code:
  - **`#pragma offload target(mic0) <code_scope>`**
  - **`!DIR$ offload target(mic0) <code_scope>`**

## Data Transfer

- Programmer clauses for explicit copy:
  in, out, inout, nocopy
- Data transfer with offload region:
  C/C++ **`#pragma offload target(mic)...`**
  **`...in(data:length(size))`**
  Fortran **`!dir$ offload target (mic)...`**
  **`...in(data:length(size))`**
- Data transfer without offload region:
  C/C++ **`#pragma offload_transfer target(mic) ...`**
  **`...in(data:length(size))`**
  Fortran **`!dir$ offload_transfer target(mic)...`**
  **`...in(data:length(size))`**

## Data Transfer:example

C C++

```
#pragma offload target (mic) out(a:length(n)) \
in(b:length(n))
for (i=0; i<n; i++){
a[i] = b[i]+c*d
```

Fortran

```
!dir$ offload begin target(mic) out(a) in(b)
do i=1,n
a(i)=b(i)+c*d
end do
!dir$ end offload
```

## Offload Model:memory

- Memory allocation
  - CPU is managed as usual
  - on coprocessor is defined by in,out,inout,nocopy clauses
- Input/Output pointers
  - by default on coprocessor "new" allocation is performed for each pointer
  - by default de-allocation is performed after offload region
  - defaults can be modified with **`alloc_if`** and **`free_if`** qualifiers
- With **`into(...)`** clause you can specify data to be moved to other variables/memory

## Data Transfer

| | |
|---|---|
| in | The variables are strictly an input to the target region. Its value is not copied back after the region completes. |
| out | The variables are strictly an output of the target region. The host CPU does not copy the variable to the target. |
| inout | The variable is both copied from the CPU to the target and back from the target to the CPU. |
| nocopy | A variable whose value is reused from a previous target execution or one that is used entirely within the offloaded code section may be named in a nocopy clause to avoid any copying. |

## Data Transfer

When are **`alloc_if`** and **`free_if`** clauses needed?

- Needed for pointers or allocatable arrays
  - Default is to always allocate and free memory for pointers that are within the lexical scope of the offload, not otherwise
  - use **`free_if(0)`** if you want to memory and data to persist until next offload
  - Need **`alloc_if(1)`** for globals that are not lexically visible and are NOCOPY
  - Or use **`alloc_if(expression)`** to make dependent on runtime data

## Data Transfer

When are **`alloc_if`** and **`free_if`** clauses needed?

- Not needed for statically allocated data
  - These are statically allocated and persistent on the coprocessor, even for arrays that are not lexically visible or have a **`NOCOPY`** clause.
  - Syntax:

```
#pragma offload nocopy(myptr:length(n):...
...alloc_if(expression))
!DIR$ OFFLOAD IN(FPTR:length(n):...
...free_if(.false.))
```

## Data Transfer:suggestion

For Readability define macros

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

## Data Transfer:suggestion

For Readability define macros

- `#define ALLOC alloc_if(1)`
- `#define FREE free_if(1)`
- `#define RETAIN free_if(0)`
- `#define REUSE alloc_if(0)`
  `#pragma offload target(mic) in( p:length(l))`
- Allocate and do not free
  ```
  #pragma offload target(mic) in (p:length(l)...
   ...ALLOC RETAIN)
  ```
- Reuse memory allocated above and do not free
  ```
  #pragma offload target(mic) in (p:length(l)...
  ... REUSE RETAIN)
  ```
- Reuse memory allocated above and free
  ```
  #pragma offload target(mic) in (p:length(l)...
  ... REUSE FREE)
  ```

## Preprocessor Macros

**`__INTEL_OFFLOAD`**

- Set automatically unless disabled by **`-no-offload`** (or **`-mmic`**)
- Set for the host compilation but not the target (coprocessor) compilation
- Use to protect code on the host that is specific for offload e.g. **`omp_num_set_threads_target()`** family of APIs but must remember to set **`-no-offload`** for host only builds

**`__MIC__`**

- NOT set for host compilation in an offload build
- Set automatically for target (coprocessor) compilation in offload build
- Also set automatically when building native coprocessor application
- Use to protect code that is compiled & executed only on coprocessor e.g. **`_mm512`** intrinsics

## Specific environment variables

| MIC Env Variable | Default Value | Description |
|---|---|---|
| MIC_ENV_PREFIX | none | Environment variables (except those below) are stripped of this prefix and underscore sent to coprocessor. Often set to "MIC". |
| MIC_<card #>_ENV | none | List of environment variables to set on card # |
| MIC_LD_LIBRARY_PATH | set by compiler vars script | Search paths for coprocessor shared libraries |
| MIC_USE_2MB_BUFFERS | Don't use | Use 2MB pages for pointer data where (size $>$ MIC_USE_2M_BUFFERS) |
| MIC_STACKSIZE | 12M | Main thread stack size limit for pthreads |
| OFFLOAD_REPORT | none | Report about Offload activity (0,1,2,3) |

## Preprocessor Macros:example

```
#ifdef __INTEL_OFFLOAD
#include <offload.h>
#endif
...
#ifdef __INTEL_OFFLOAD
printf("%d MICS available\n",_Offload_number_of_devices());
#endif
...
int main(){
#pragma offload target(mic)
{
#ifdef __MIC__
printf("Hello MIC number %d\n", _Offload_get_device_number());
#else
printf("Hello HOST\n");
#endif
}
}
```

## Asynchronous Offload

New synchronization clauses **`SIGNAL(&x)`** and **`WAIT(&x)`**

- Argument is a unique address (usually of the data being transferred)

Data:

- **`#pragma offload_transfer target(mic:n) ... ...IN(....) signal(&s1)`**
  - Standalone data offload
- **`#pragma offload_wait target(mic:n) wait(&s1)`**
  - Standalone synchronization, host waits for transfer completion (blocking)

Computation:

- **`#pragma offload target(mic:n) wait(&s1)... ...signal(&s2)`**
  - Offload computation when data transfer has completed
  - Computation on host then continues in parallel
- **`#pragma offload_wait target(mic:n) wait(&s2)`**
  - Host waits for signal that offload computation completed

There is also a non-blocking API to test signal value

## Asynchronous Offload

```
float *T;
int N_OFFLOAD = 2*NTOT/3;
#pragma offload target(mic:0) in(T:length(N_OFFLOAD)) ...
...out(Result:length(N_OFFLOAD)) \
signal (&T)
{
#pragma omp parallel for
for(int opt = 0; opt < N_OFFLOAD; opt++) {
... // do first 2/3 of work on coprocessor
}
}
#pragma omp parallel for
for(int opt = N_OFFLOAD; opt < NTOT; opt++) {
... // do remainder of work on host
}
{
// synchronization before continuing on host using results of offload
#pragma offload_wait target(mic:0) wait (&T)
// easily extended to offload work to multiple coprocessors, using different signals
```

## Offload region:OpenMP

The code section to be executed on accelerators are marked by a target construct.

- A target region is executed by a single thread, called the initial device thread
- Parallelism on accelerator is specified by traditional and extended Openmp-parallel constructs
- The task that encounters the target construct waits at the end of the construct until execution of the region completes
- If a target device does not exist or is not supported by the implementation, the target region is executed by the host device

## Offload region:OpenMP

C

```
#pragma offload target (mic)
#pragma omp parallel for
for (i=0; i<n; i++){
a[i]=b[i]*c+d;
}
```

Fortran

```
!dir$ omp offload target (mic)
!$omp parallel do
do i=1,n
A(i)=B(i)*C+D
end do
!$omp end parallel
```

## Offload region:OpenMP

- The basics work just like on the host CPU
  - For both native and offload models
  - Need to specify `-openmp`
- There are 4 hardware thread contexts per core
  - Need at least 2 x ncore threads for good performance
  - For all except the most memory-bound workloads
  - Often, 3x or 4x (number of available cores) is best
  - Very different from hyperthreading on the host!
  - `-opt-threads-per-core=n` advises compiler how many threads to optimize for
- If you don't saturate all available threads, be sure to set `KMP_AFFINITY` to control thread distribution

## OpenMP defaults

- **`$OMP_NUM_THREADS`** defaults to
  - 1 x ncore for host (or 2x if hyperthreading enabled)
  - 4 x ncore for native coprocessor applications
  - 4 x (ncore-1) for offload applications
  - one core is reserved for offload daemons and OS
- Defaults may be changed via environment variables or via API calls on either the host or the coprocessor

Setting up the environment:

```
OMP_NUM_THREAD = 16
MIC_ENV_PREFIX = MIC
MIC_OMP_NUM_THREADS = 120
```

## Thread Affinity Interface

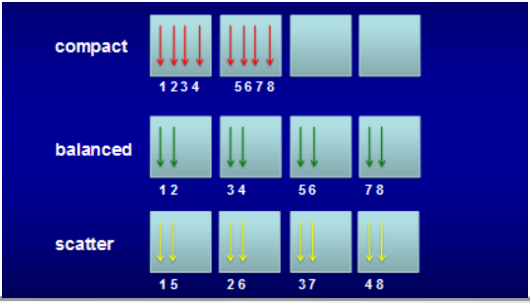
Allows OpenMP threads to be bound to physical or logical cores

- Helps optimize access to memory or cache
- Particularly important if all available h/w threads not used
- else some physical cores may be idle while others run multiple threads

## Thread Affinity

export environment variable **`KMP_AFFINITY=`**

- **`compact`** assign threads to consecutive h/w contexts on same physical core (eg to benefit from shared cache)
- **`scatter`** assign consecutive threads to different physical cores (eg to maximize access to memory)
- **`balanced`** blend of compact & scatter (currently only available for Intel MIC Architecture)



## Support for Multiple Coprocessors

`#pragma offload target(mic [ :coprocessor # ])...`
coprocessor # = <expr> % NumberOfDevices Code must run on coprocessor #, aborts if not available (counts from 0) If -1 , runtime chooses coprocessor, aborts if not available If not present, runtime chooses coprocessor or runs on host if none available

- APIs: `#include offload.h` (C C++)
  `USE MIC_LIB` (Fortran)
  `int _Offload_number_of_devices()` (C C++)
  `result = OFFLOAD_NUMBER_OF_DEVICES(),` (Fortran)
- Returns # of coprocessors installed, or 0 if none
  `int _Offload_get_device_number()` (C C++)
  `result = OFFLOAD_GET_DEVICE_NUMBER()` (Fortran )
- Returns coprocessor number where executed, (-1 for CPU)
- Can use to share work explicitly by coprocessor number

## OFFLOAD_REPORT

Possible values: 1, 2, 3

1. Prints the offload computation time, in seconds.
2. In addition to the information produced with value 1, adds the amount of data transferred between the CPU and the coprocessor, in bytes.
3. In addition to the information produced at value 2, gives additional details on offload activity, including device initialization, and individual variable transfers.

| Line Marker | Descrption |
|---|---|
| [State] | Activity being performed as part of the offload. |
| [Var] | The name of a variable transferred and the direction(s) of transfer. |
| [CPU Time] | The total time measured for that offload directive on the host. |
| [MIC Time] | The total time measured for executing the offload on the target. This excludes the data transfer time between the host and the target, and counts only the execution time on the target. |
| [CPU->MIC Data] | The number of bytes of data transferred from the host to the target. |
| [MIC->CPU Data] | The number of bytes of data transferred from the target to the host. |

## OFFLOAD_REPORT:Example

```
...
[Offload] [MIC 0] [Line]            176
[Offload] [MIC 0] [Tag]             Tag 300
[Offload] [HOST]  [Tag 300] [State]   Start Offload
[Offload] [HOST]  [Tag 300] [State]   Initialize function __offload_entry_compute_forces_
[Offload] [HOST]  [Tag 300] [State]   Create buffer from Host memory
[Offload] [HOST]  [Tag 300] [State]   Create buffer from MIC memory
[Offload] [HOST]  [Tag 300] [State]   Create buffer from Host memory
[Offload] [HOST]  [Tag 300] [State]   Create buffer from MIC memory
...
[Offload] [MIC 0] [Tag 300] [Var]     gammaval  NOCOPY
[Offload] [MIC 0] [Tag 300] [Var]     alphaval  NOCOPY
[Offload] [MIC 0] [Tag 300] [Var]     phase_ispec_inner  NOCOPY
[Offload] [MIC 0] [Tag 300] [Var]     phase_ispec_inner  NOCOPY
[Offload] [MIC 0] [Tag 300] [Var]     var$456_dv_template_V$378  NOCOPY
[Offload] [MIC 0] [Tag 300] [Var]     var$456_dv_template_V$378  NOCOPY
[Offload] [MIC 0] [Tag 300] [Var]     var$191_dv_template_V$162  INOUT
[Offload] [MIC 0] [Tag 300] [Var]     var$191_dv_template_V$162  INOUT
[Offload] [MIC 0] [Tag 300] [State]   Scatter copyin data
 Code running on MIC
[Offload] [MIC 1] [Tag 300] [State]   Gather copyout data
[Offload] [MIC 1] [Tag 300] [State]   MIC->CPU copyout data   0
[Offload] [MIC 0] [Tag 300] [State]   Gather copyout data
[Offload] [MIC 0] [Tag 300] [State]   MIC->CPU copyout data   0
[Offload] [HOST]  [Tag 300] [State]   Scatter copyout data
[Offload] [HOST]  [Tag 300] [CPU Time]        0.128716(seconds)
[Offload] [MIC 1] [Tag 300] [CPU->MIC Data]   17885096 (bytes)
[Offload] [MIC 1] [Tag 300] [MIC Time]        0.031128(seconds)
[Offload] [MIC 1] [Tag 300] [MIC->CPU Data]   29148044 (bytes)
```

## Implicit offloading

- Implicit Offloading: Virtual Shared Memory
- User code declares data objects to be shared:
  - Data allocated at same address on host and target
  - Modified data is copied at synchronization points
  - Allows sharing of complex data structures
  - No data marshaling necessary

## Implicit offloading:example

```
#define N 20000.0
_Cilk_shared float FindArea(float r) \\ Explicitly shared function:
{
For both host & target
float x, y, area;
unsigned int seed = __cilkrts_get_worker_number();
cilk::reducer_opadd<int> inside(0);
cilk_for(int i = 0; i < N; i++) {
x = (float)rand_r(&seed)/RAND_MAX;
y = (float)rand_r(&seed)/RAND_MAX;
x = 2.0 * x - 1.0;
y = 2.0 * y - 1.0;
if (x * x + y * y < r * r) inside++;
}
area = 4.0 * inside.get_value() / N;
return area;
}
```

## Implicit offloading:example

```
int main(int argc, char **argv)
{
// Get r1 & r2 from user...
Offload to target (big area)
Area1 = cilk_spawn _Cilk_offload FindArea(r1);
\\While target runs, compute other area on host (small area)
Area2 = FindArea(r2);
\\ Wait for host & target to complete
cilk_sync;
float Donut = Area1 - Area2;
float PI = 3.14159265;
float AreaR = PI * (r2 * r2 - r1 * r1);
float Accuracy = 100 * (1 - fabs(Donut - AreaR)/AreaR);
printf("Area1=%lf, Area2=%lf\n", Area1, Area2);
printf("Donut =%lf, Accuracy = %lf\n", Donut, Accuracy);
}
```

## Symmetric mode

- Using MPI you can make work together the executable running on the host and the one running on the device (compiled with `-mmic`)
- Load balancing can be an issue
- Tuning of MPI and OpenMP on both host and device is crucial
- Dependent on the cluster implementation (physical network, MPI implementation, job scheduler..)

## Symmetric mode

Compile the program for the host

```
mpiicc -openmp -o test test.c
```

Compile the program for the coprocessor

```
mpiicc -mmic -openmp -o test.mic test.c
```

Set the environment

```
export I_MPI_MIC=enable
export I_MPI_MIC_POSTFIX='.mic'
```

- I_MPI_MIC enable the Intel Xeon Phi coprocessor recognition
- I_MPI_MIC_POSTFIX specify a string as the postfix of an Intel Xeon Phi coprocessor file name. The default value is an empty string

## Symmetric mode

Run

```
mpirun -machinefile hostfile ./test
```

Where hostfile is(for example):

```
node142-mic0:4
node142-mic1:4
node142:8
```

## MKL Libraries

- Intel released a version for Xeon Phi of the MKL mathematical libraries
- MKL have three different usage models
  - Automatic offload (AO)
  - Compiler assisted offload (CAO)
  - Native execution

## MKL Libraries

- Offload is automatic and transparent
- The library decides when to offload and how much to offload (workdivision)
- Users can control parameters through environment variables or API
- You can enable automatic offload with

```
MKL_MIC_ENABLE=1
```

or

```
mkl_mic_enable()
```

- Not all the MKL functions are enabled to AO. Level 3 BLAS: xGEMM, xTRSM, xTRMM
  LAPACK xGETRF, xPOTRF, xGEQRF
- Always check the documentation for updates

## MKL Libraries

- MKL functions can be offloaded as other "ordinary" functions using the LEO pragmas
- All MKL functions can take advantage of the CAO
- It’s a more flexible option in terms of data management (you can use data persistence or mechanisms to hide the latency...)

C C++

```
#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

```
!dir$ attributes offload : mic : sgemm
!dir$ offload target(mic) &
!dir$ in (transa, transb, m, n, k, alpha, beta, lda, ldb, ldc), &
!dir$ in (a:length(ncola*lda)), in (b:length(ncolb*ldb)) &
!dir$ inout (c:length(n*ldc))
CALL sgemm (transa, transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
```

## MKL Libraries

- MKL libraries are also available when using the native mode.
  - Use all the 240 threads:

```
MIC_OMP_NUM_THREADS=240
```

  - Set the thread affinity:

```
MIC_KMP_AFFINITY = ...
```

## Outline

Architectures

Optimization

Vectorization

Performance and parallelism

Programmming Models

Profiling and Debugging

## Loop Profiler (no parallel version code)

Identify Time Consuming Loops/Functions

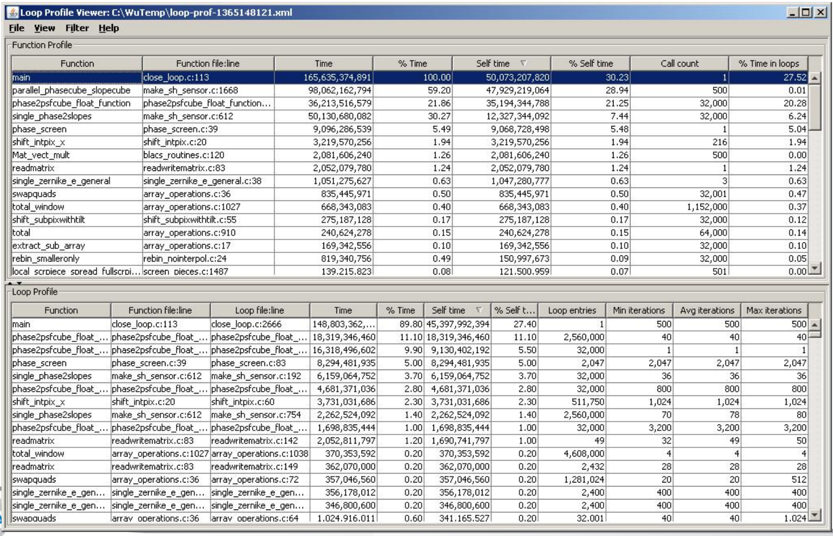
- Compiler switch
  - **`-profile-functions`**
    - Insert instrumentation calls on function entry and exit points to collect the cycles spent within the function.
  - **`-profile-loops= <inner|outer|all>`**
    - Insert instrumentation calls for function entry and exit points as well as the instrumentation before and after instrument able loops of the type listed as the option's argument.
- You can read the results from the .dump text file
- GUI-based data viewer utility
  - Input is generated XML output file, named loop_prof_<timestamp>.xml

```
loopprofileviewer.sh <datafile>
```

## Loop Profiler:dump file

```
time(abs)  time(%) self(abs)    self(%) call_count exit_count loop_ticks(%) function file:line
8432294327 100.00 70546995207  67.66       1         1          0.00        MAIN__ fem.F:1
2607725174 30.93  134933586349 16.00      715       715         0.00        s_par_ s_par.F:3
2579776614 14.92  25797766140  14.92     5728      5728         0.00        sol_ sol.F:4
541913956   0.64    4499167955  0.53       1         1          0.00        cost_ costr.F:1
320566224   0.38    3205662248  0.38       1         1          0.00        pres_  presol.F:2
164528333   0.20    1645283337   0.20     715       715         0.00        funz_  funz.F:1
122051340   0.14    1220513408   0.14      1         1          0.00        prel_ fem.F:3301
...
```

## Loop Profiler:xml file


Loop Profile Viewer: C:\WuTemp\loop-prof-1365148121.xml

Function Profile

| Function | Function file:line | Time | % Time | Self time | % Self time | Call count | % Time in loops |
|---|---|---|---|---|---|---|---|
| main | close_loop.c:113 | 165,635,374,891 | 100.00 | 50,073,207,820 | 30.23 | 1 | 27.52 |
| parallel_phasecube_slopecube | make_sh_sensor.c:1668 | 98,062,162,794 | 59.20 | 47,929,219,064 | 28.94 | 500 | 0.01 |
| phase2psfcube_float_function | phase2psfcube_float_function… | 36,213,516,579 | 21.86 | 35,194,344,788 | 21.25 | 32,000 | 20.28 |
| single_phase2slopes | make_sh_sensor.c:612 | 50,130,680,082 | 30.27 | 12,327,344,092 | 7.44 | 32,000 | 6.24 |
| phase_screen | phase_screen.c:39 | 9,096,286,539 | 5.49 | 9,068,728,498 | 5.48 | 1 | 5.04 |
| shift_intpix_x | shift_intpix.c:20 | 3,219,570,256 | 1.94 | 3,219,570,256 | 1.94 | 216 | 1.94 |
| Mat_vect_mult | blacs_routines.c:120 | 2,081,606,240 | 1.26 | 2,081,606,240 | 1.26 | 500 | 0.00 |
| readmatrix | readwritematrix.c:83 | 2,052,079,780 | 1.24 | 2,052,079,780 | 1.24 | 1 | 1.24 |
| single_zernike_e_general | single_zernike_e_general.c:38 | 1,051,275,627 | 0.63 | 1,047,280,777 | 0.63 | 3 | 0.63 |
| swapquads | array_operations.c:36 | 835,445,971 | 0.50 | 835,445,971 | 0.50 | 32,001 | 0.47 |
| total_window | array_operations.c:1027 | 668,343,083 | 0.40 | 668,343,083 | 0.40 | 1,152,000 | 0.37 |
| shift_subpixwithtilt | shift_subpixwithtilt.c:55 | 275,187,128 | 0.17 | 275,187,128 | 0.17 | 32,000 | 0.12 |
| total | array_operations.c:910 | 240,624,278 | 0.15 | 240,624,278 | 0.15 | 64,000 | 0.14 |
| extract_sub_array | array_operations.c:17 | 169,342,556 | 0.10 | 169,342,556 | 0.10 | 32,000 | 0.10 |
| rebin_smalleronly | rebin_nointerpol.c:24 | 819,340,756 | 0.49 | 150,997,673 | 0.09 | 32,000 | 0.05 |
| local_scrpiece_spread_fullscrpi… | screen_pieces.c:1487 | 139,215,823 | 0.08 | 121,500,959 | 0.07 | 501 | 0.00 |

Loop Profile

| Function | Function file:line | Loop file:line | Time | % Time | Self time | % Self t… | Loop entries | Min iterations | Avg iterations | Max iterations |
|---|---|---|---|---|---|---|---|---|---|---|
| main | close_loop.c:113 | close_loop.c:2666 | 148,803,362,… | 89.80 | 45,397,992,394 | 27.40 | 1 | 500 | 500 | 500 |
| phase2psfcube_float_… | phase2psfcube_float_… | phase2psfcube_float_… | 18,319,346,460 | 11.10 | 18,319,346,460 | 11.10 | 2,560,000 | 40 | 40 | 40 |
| phase2psfcube_float_… | phase2psfcube_float_… | phase2psfcube_float_… | 16,318,496,602 | 9.90 | 9,130,402,192 | 5.50 | 32,000 | 1 | 1 | 1 |
| phase_screen | phase_screen.c:39 | phase_screen.c:83 | 8,294,481,935 | 5.00 | 8,294,481,935 | 5.00 | 2,047 | 2,047 | 2,047 | 2,047 |
| single_phase2slopes | make_sh_sensor.c:612 | make_sh_sensor.c:192 | 6,159,064,752 | 3.70 | 6,159,064,752 | 3.70 | 32,000 | 36 | 36 | 36 |
| phase2psfcube_float_… | phase2psfcube_float_… | phase2psfcube_float_… | 4,681,371,036 | 2.80 | 4,681,371,036 | 2.80 | 32,000 | 800 | 800 | 800 |
| shift_intpix_x | shift_intpix.c:20 | shift_intpix.c:60 | 3,731,031,686 | 2.30 | 3,731,031,686 | 2.30 | 511,750 | 1,024 | 1,024 | 1,024 |
| single_phase2slopes | make_sh_sensor.c:612 | make_sh_sensor.c:754 | 2,262,524,092 | 1.40 | 2,262,524,092 | 1.40 | 2,560,000 | 70 | 78 | 80 |
| phase2psfcube_float_… | phase2psfcube_float_… | phase2psfcube_float_… | 1,698,835,444 | 1.00 | 1,698,835,444 | 1.00 | 32,000 | 3,200 | 3,200 | 3,200 |
| readmatrix | readwritematrix.c:83 | readwritematrix.c:142 | 2,052,811,797 | 1.20 | 1,690,741,797 | 1.00 | 49 | 32 | 49 | 50 |
| total_window | array_operations.c:1027 | array_operations.c:1038 | 370,353,592 | 0.20 | 370,353,592 | 0.20 | 4,608,000 | 4 | 4 | 4 |
| readmatrix | readwritematrix.c:83 | readwritematrix.c:149 | 362,070,000 | 0.20 | 362,070,000 | 0.20 | 2,432 | 28 | 28 | 28 |
| swapquads | array_operations.c:36 | array_operations.c:72 | 357,046,560 | 0.20 | 357,046,560 | 0.20 | 1,281,024 | 20 | 20 | 512 |
| single_zernike_e_gen… | single_zernike_e_gen… | single_zernike_e_gen… | 356,178,012 | 0.20 | 356,178,012 | 0.20 | 2,400 | 400 | 400 | 400 |
| single_zernike_e_gen… | single_zernike_e_gen… | single_zernike_e_gen… | 346,800,600 | 0.20 | 346,800,600 | 0.20 | 2,400 | 400 | 400 | 400 |
| swapquads | array_operations.c:36 | array_operations.c:64 | 1,024,916,011 | 0.60 | 341,165,527 | 0.20 | 32,001 | 40 | 40 | 1,024 |

## Auto-Parallelizer

- The Intel compiler has an auto-parallelizer that can automatically add parallelism to loops.
- By default, the auto-parallelizer is disabled, but you can enable it with the -parallel option.
- Use this feature to give hints on where best to parallelize their code.

## Auto-Parallelizer

- Finds loops that could be candidates for making parallel
- Decides if there is a sufficient amount of work done to justify parallelization
- Checks that no loop dependencies exist
- Appropriately partitions any data between the parallelized code
- The auto-parallelizer (at the time of writing) uses OpenMP.

## Profiling Steps

1. Compile the sources with the **`-parallel`** option. To get superior results, it's always best to enable interprocedural optimization ( **`-ipo`** ). The option **`-par-report2`** instructs the compiler to generate a parallelization report, listing which loops were made parallel.
2. Look at the results from the compiler and make a note of any lines that were successfully parallelized.
3. Add your own parallel constructs to the identified loops. If you add OpenMP directives to have more control, the option **`-openmp-report`** instructs the compiler to generate a OpenMP parallelization report.
4. Rebuild the application without the **`-parallel`** option.

## Report:example

```
presolutore_parallelo.F(34): (col. 2) remark:
DISTRIBUTED LOOP WAS AUTO-PARALLELIZED
..
solutore_parallelo.F(43): (col. 10) remark: loop was not parallelized:
existence of parallel dependence
solutore_parallelo.F(43): (col. 10) remark: loop was not parallelized:
insufficient computational work
...
solutore_parallelo.F(40): (col. 7) remark:
OpenMP DEFINED REGION WAS PARALLELIZED
...
solutore_parallelo.F(67): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED
solutore_parallelo.F(84): (col. 7) remark: OpenMP multithreaded code
generation for SINGLE was successful
```

## VTune

- The Intel VTuneTM Performance Analyzer is a powerful software-profiling tool available on both Microsoft Windows and Linux OS.
- VTuneTM helps you understand the performance characteristics of your software at all levels: system, application and microarchitecture.
- The main features of VTuneTM are sampling, call graph and counter monitor.
- For details on all features and how to use the tool, see the VTuneTM documentation

https://software.intel.com/sites/products/documentation/doclib/iss/2013/amplifier/lin/ug_docs/

## VTune

- Hot Spot Analysis (Statistical Call Graph)
  Where is the application spending time and how did it get there?
- Hardware Event-based Sampling (EBS)
  Where are the tuning opportunities? (e.g., cache misses)
  - Pre-defined tuning experiments
- Thread Profiling
  Where is my concurrency poor and why?
- Thread timeline visualizes thread activity and lock transitions
  - Integrated EBS data tells you exactly what's happening and when

## VTune

- Timeline correlates thread and event data
  - See what active threads are doing
  - Filter profile results by selecting a region in the timeline
- Advanced Source / Assembler View
  - See event data graphed on the source / assembler
  - View and analyze assembly as basic blocks
- Collect System Wide Data & Attach to Running Processes
  - EBS collects system wide data, filter it to find what you need
  - Hot Spot and Concurrency Analyses can attach to a running process
- GUI & Command Line
  - Stand-alone GUI, Command Line
  - GUI makes setup and analysis easy
  - Command line for regression analysis and collection on remote systems

## VTune

Set the environment

```
source /cineca/prod/compilers/intel/cs-xe-2013/binary/vtune_amplifier_xe_2013/amplxe-vars.sh
```

Display a list of available analysis types and preset configuration levels

```
amplxe-cl -collect-list
```

Run Hot Spot analysis on target myApp and store result in r001hs directory

```
mpirun -np 2 amplxe-cl -collect hotspots  -r r001hs myApp
```

Analyze the result in directory r001par

```
amplxe-cl -report summary  -r <path_r001hs>
```

Run the standalone graphical interface

```
amplxe-gui
```

## VTune:EBS

To known supported memory load events in your platform

```
amplxe-runss -event-list
```

To run the application monitoring the events.

```
amplxe-cl -collect-with runsa-knc -knob event-config="CPU_CLK_UNHALTED,
INSTRUCTIONS_EXECUTED" -r hs0001 mpirun -np 2 my_application
```

Description of the events for Intel paltform named Knights Corner

http://www.hpc.ut.ee/dokumendid/ips_xe_2015/vtune_amplifier_xe/documentation/en/help/reference/knc/

## CPI

Cycles Per Instruction (CPI), a standard measure, has some special kinks

- Threads on each Intel XeonTM Phi core share a clock
- If all 4 HW threads are active, each gets 1/4 total cycles
- Multi-stage instruction decode requires two threads to utilize the whole core - one thread only gets half
- With two ops/per cycle (U-V-pipe dual issue):

| Threads per Core | Minimum Best CPI per Core | Minimum Best CPI per Thread |
|---|---|---|
| 1 X | 1.0 | =1.0 |
| 2 X | 0.5 | =1.0 |
| 3 X | 0.5 | =1.5 |
| 4 X | 0.5 | =2.0 |

- To get thread CPI, multiply by the active threads

## Efficiency Metric

- Changes in CPI absent major code changes can indicate general latency gains/losses

| Metric | Formula | Investigate if |
|---|---|---|
| CPI per Thread | CPU_CLK_UNHALTED/ INSTRUCTIONS_EXECUTED | $>$ 4 or increasing |
| CPI per Core | (CPI per Thread) / Number of hardware threads used | $>$ 1 or increasing |

- Note the effect on CPI from applied optimizations
- Reduce high CPI through optimizations that target latency
  - Better prefetch
  - Increase data reuse through better blocking

## Efficiency Metric

Compute to Data Access Ratio

- Measures an application's computational density, and suitability for Intel Xeon PhiTM coprocessors

| Metric | Formula | Investigate if |
|---|---|---|
| Vectorization Intensity | VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED | |
| L1 Compute Data Access | VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE | <Vectorization Intensity |
| L2 Compute Data Access | VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_WRITE_MISS | < 100x L1 Compute to Data Access Ratio |

- Increase computational density through vectorization and reducing data access (see cache issues, also, DATA ALIGNMENT!)

## L1 Cache Usage

- Significantly affects data access latency and therefore application performance

| Metric | Formula | Investigate if |
|---|---|---|
| L1 Misses | DATA_READ_MISS_OR_WRITE_MISS + L1_DATA_HIT_INFLIGHT_PF1 | |
| L1 Hit Rate | (DATA_READ_OR_WRITE - L1 Misses) / DATA_READ_OR_WRITE | $<$ 95 % |

- Tuning Suggestions:
  - Software prefetching
  - Tile/block data access for cache size
  - Use streaming stores
  - If using 4K access stride, may be experiencing conflict misses
  - Examine Compiler prefetching (Compiler-generated L1 prefetches should not miss)

## Data Access Latency

| Metric | Formula | Investigate if |
|---|---|---|
| Estimated Latency Impact | (CPU_CLK_UNHALTED - EXEC_STAGE_CYCLES - DATA_READ_OR_WRITE) / DATA_READ_OR_WRITE_MISS | $> 145$ |

- Tuning Suggestions:
  - Software prefetching
  - Tile/block data access for cache size
  - Use streaming stores
  - Check cache locality - turn off prefetching and use CACHE_FILL events - reduce sharing if needed/possible
  - If using 64K access stride, may be experiencing conflict misses

## TLB Usage

- Also affects data access latency and therefore application performance

| Metric | Formula | Investigate if |
|---|---|---|
| L1 TLB miss ratio | DATA_PAGE_WALK/DATA_READ_OR_WRITE | $>$ 1 % |
| L2 TLB miss ratio | LONG_DATA_PAGE_WALK LONG_DATA_PAGE_WALK | $>$ .1 % |
| L1 TLB misses per L2 TLB miss | DATA_PAGE_WALK / | $>$ 100x LONG_DATA_PAGE |

- Tuning Suggestions:
  - Improve cache usage & data access latency
  - If L1 TLB miss/L2 TLB miss is high, try using large pages
  - For loops with multiple streams, try splitting into multiple loops
  - If data access stride is a large power of 2, consider padding between arrays by one 4 KB page

## VPU Usage

- Indicates whether an application is vectorized successfully and efficiently

| Metric | Formula | Investigate if |
|---|---|---|
| Vectorization Intensity | VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED | $<$8 (DP), $<$16(SP) |

- Tuning Suggestions:
  - Use the Compiler vectorization report!
  - For data dependencies preventing vectorization, try using Intel CilkTM Plus #pragma SIMD (if safe!)
  - Align data and tell the Compiler!
  - Restructure code if possible: Array notations, AOS->SOA

## Memory Bandwidth

- Can increase data latency in the system or become a performance bottleneck

| Memory | Formula | Investigate if |
|---|---|---|
| Bandwidth | (UNC_F_CH0_NORMAL_READ + UNC_F_CH0_NORMAL_WRITE+ UNC_F_CH1_NORMAL_READ + UNC_F_CH1_NORMAL_READ + UNC_F_CH1_NORMAL_WRITE) X 64/time | < 80GB/sec (practical peak 140GB/sec) (with 8 memory controllers) |

- Tuning Suggestions:
  - Improve locality in caches
  - Use streaming stores
  - Improve software prefetching

## Bibliography

- **Optimizing HPC Applications with Intel Cluster Tools**
  A. Supalov.A Semin, M. Klemm and C. Dahnken
  *Apress open* 2014.
- **High-Performance Computing on the Intel Xeon Phi**
  E. Wang, Q. Zhang, B. Shen, G.Zhang, X. Lu, Q. Wu and Y. Wang
  *Springer* 2012.
- **High Performance Parallelism Pearls**
  J. Reinders and J. Jeffers
  *Springer* 2012.
- **Intel Xeon Phi Coprocessor Architecture and Tools. The Guide for Application Developers**
  R. Rahman
  *Apress open* 2013.
- **Intel Xeon Phi Coprocessor High Performance Programmming**
  J. Jeffers and J. Reinders
  *Morgan Kaufmann* 2013.
- **Parallel Programming with Intel Parallel Studio XE**
  S. Blair-Chappell and A. Stokes
  *John Wiley and Sons, Inc* 2012.
- https://software.intel.com