Introduction to Intel Xeon Phi programming

Fabio AFFINITO
CINECA

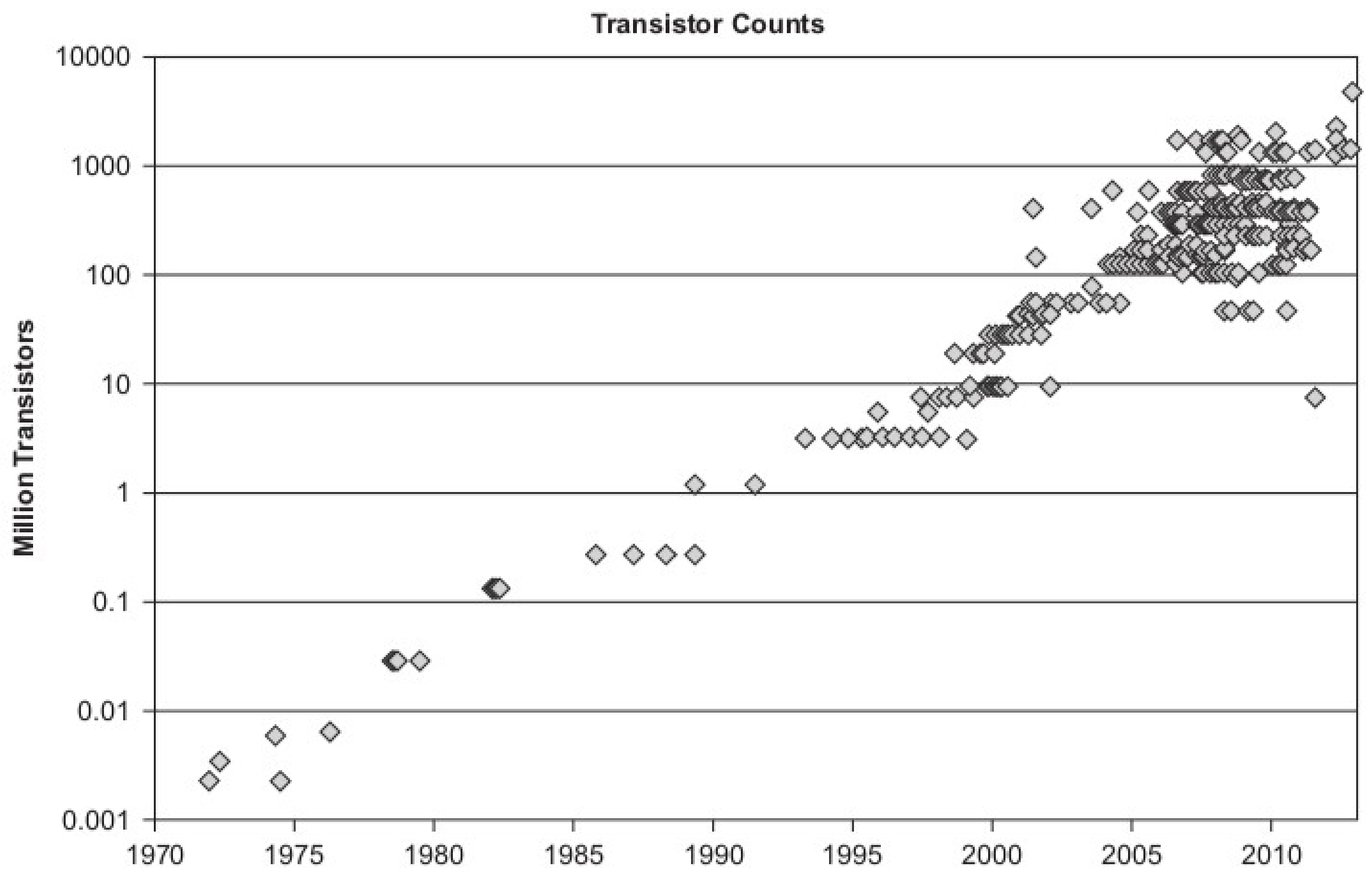
Outline

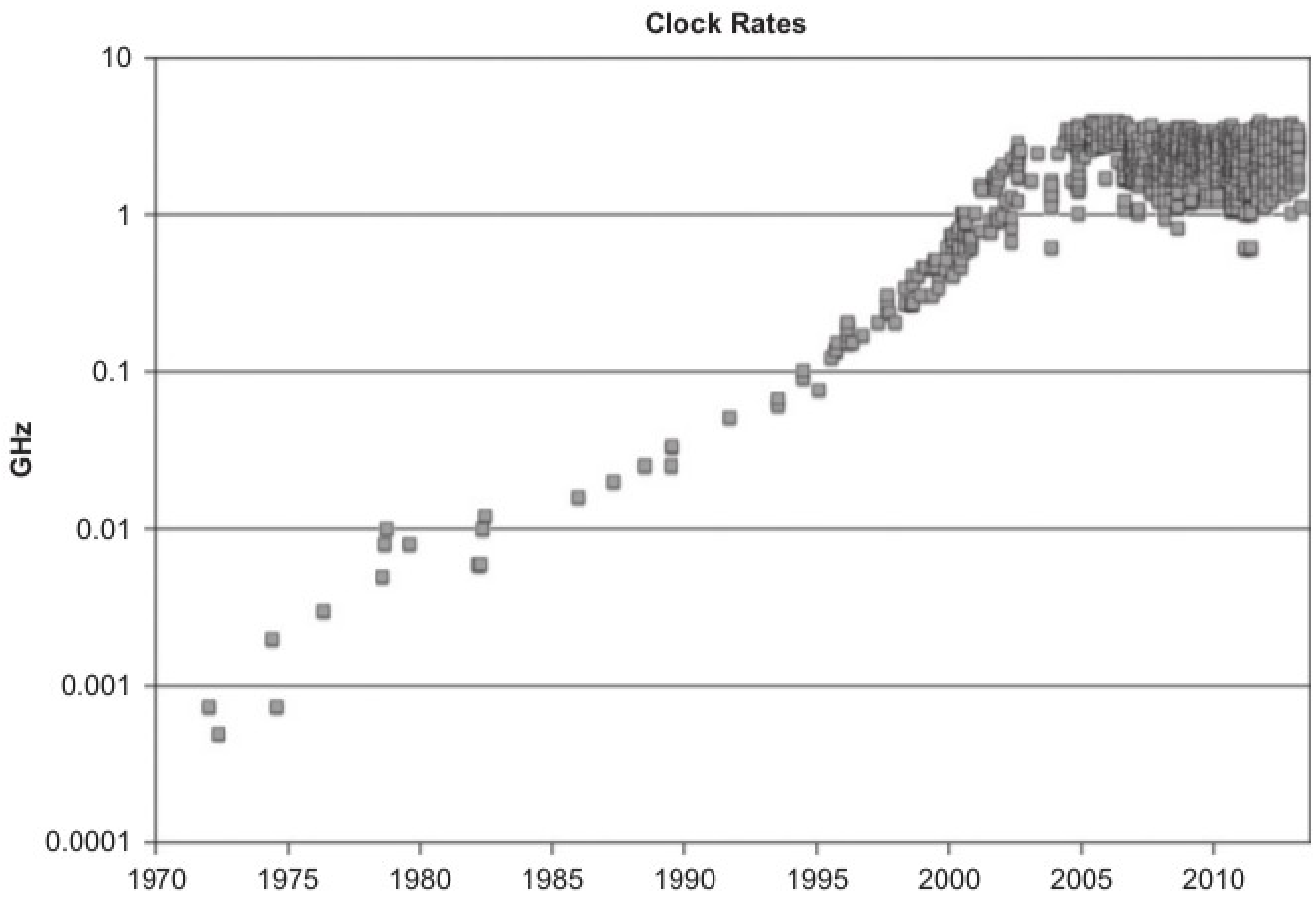
- Overview of Intel Xeon Phi

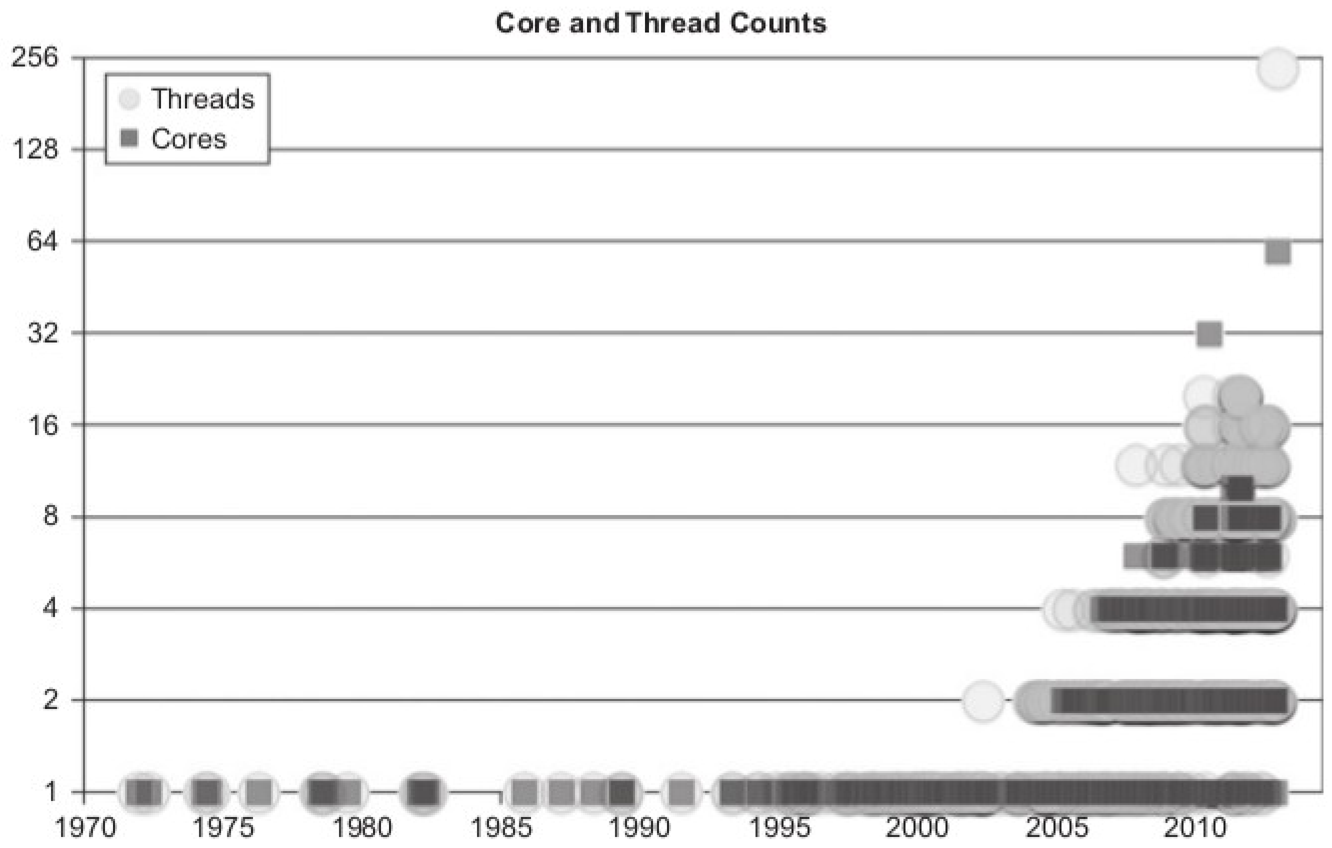
- Program models
    - Offload programming
    - Native programming
    - Symmetric mode

- Vectorization

- MKL Libraries


Transistor Counts
Million Transistors
10000
1000
100
10
1
0.1
0.01
0.001
1970
1975
1980
1985
1990
1995
2000
2005
2010


Clock Rates
GHz
10
1
0.1
0.01
0.001
0.0001
1970
1975
1980
1985
1990
1995
2000
2005
2010


Core and Thread Counts
256
128
64
32
16
8
4
2
1
1970
1975
1980
1985
1990
1995
2000
2005
2010
Threads
Cores

Summarizing:

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

=> solution: increase the number of cores

Let me introduce you...

What is the Xeon Phi?

- Intel Xeon Phi clock: 1090 MHz
- 60 cores in-order
- ~ 1 Tflop/s DP peak performance
- 4 hardware threads per core
- 8 GB DDR5 memory
- 512-bit SIMD vectors (32 registers)
- Fully-coherent L1 and L2 caches
- PCIe bus

Terminology

- MIC = Many Integrated Cores is the name of the architecture
- Xeon Phi is the 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 intensive.
In this sense, it works together with the CPU.
It can be used as an accelerator using the “offload” programming model.
The bottleneck is represented by the communication between host and device (through PCIe)

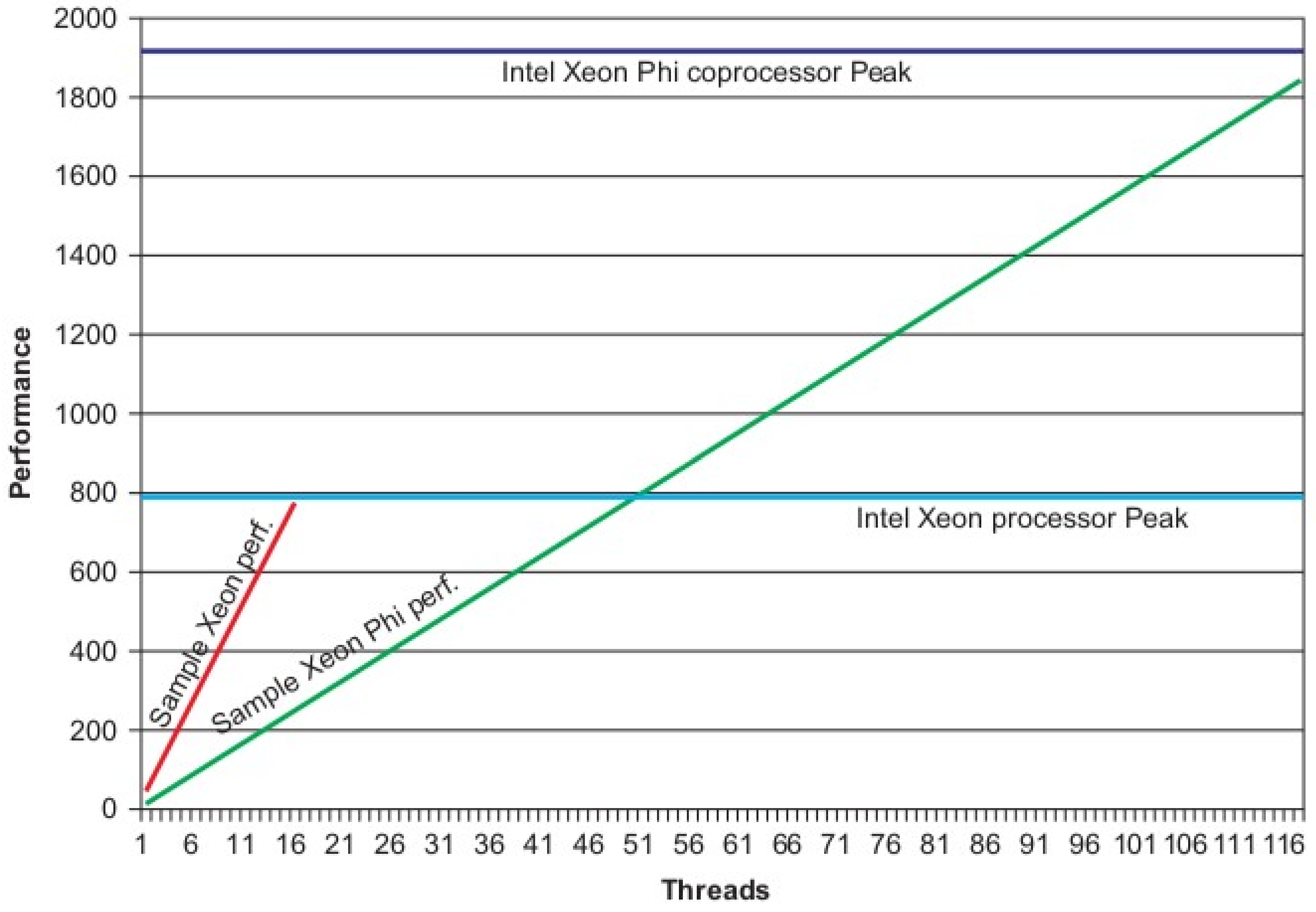
In this sense it is very similar to a GPU.

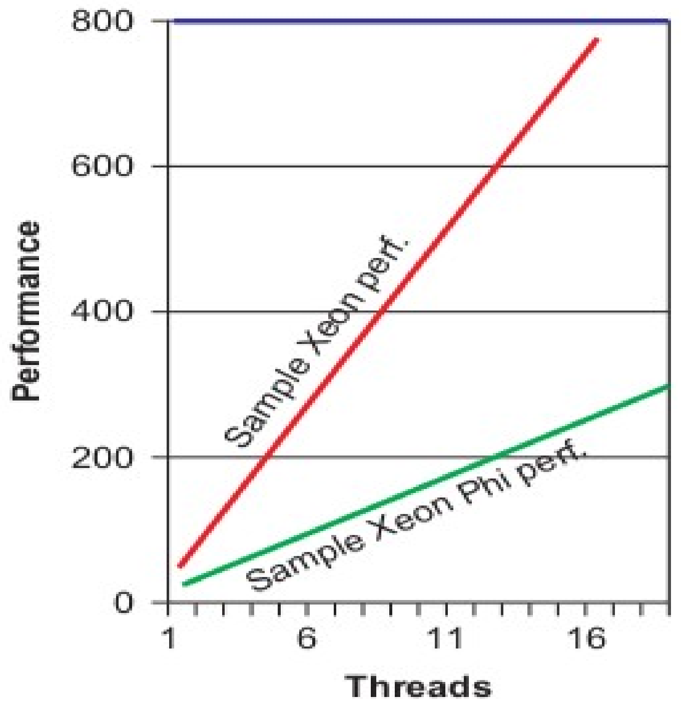
Is it an accelerator?

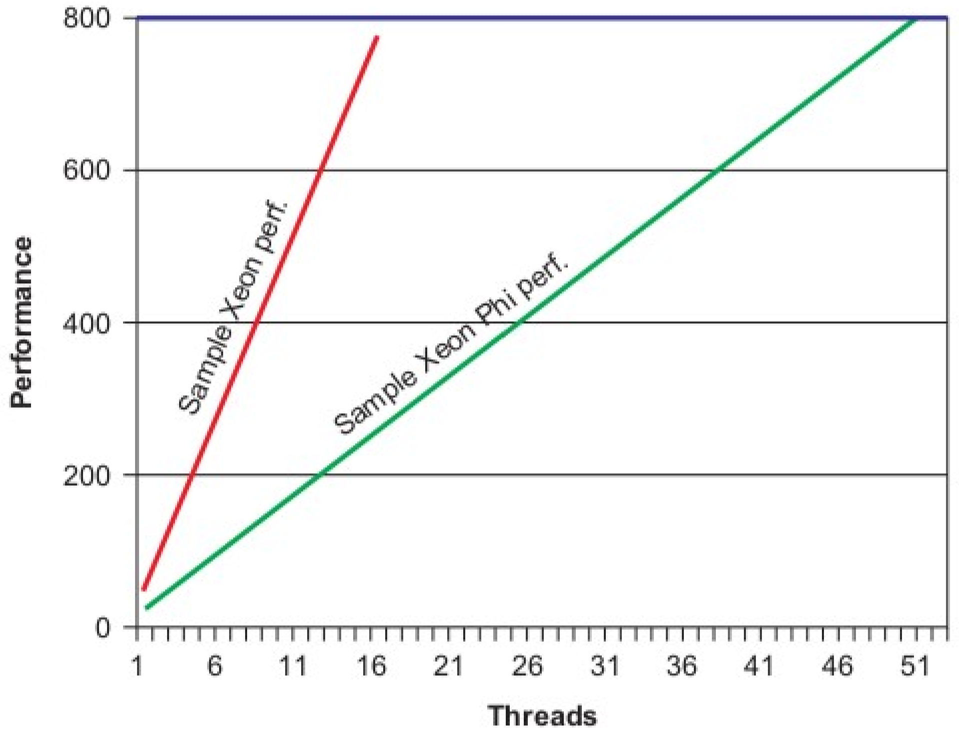
NOT ONLY : the Intel Xeon Phi can behave as a many-core X86 node. Code can be compiled and run “natively” on the Xeon Phi platform using MPI+OpenMP.

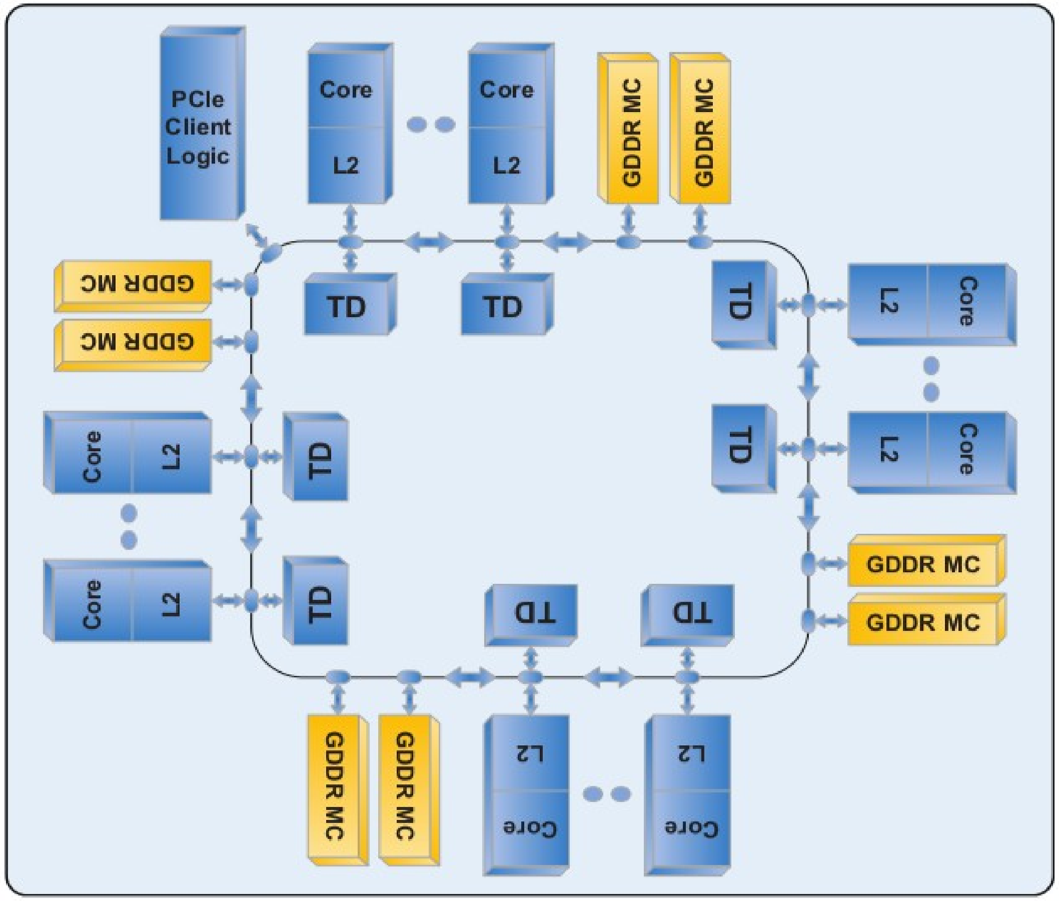
The bottleneck is the Amdahl Law.

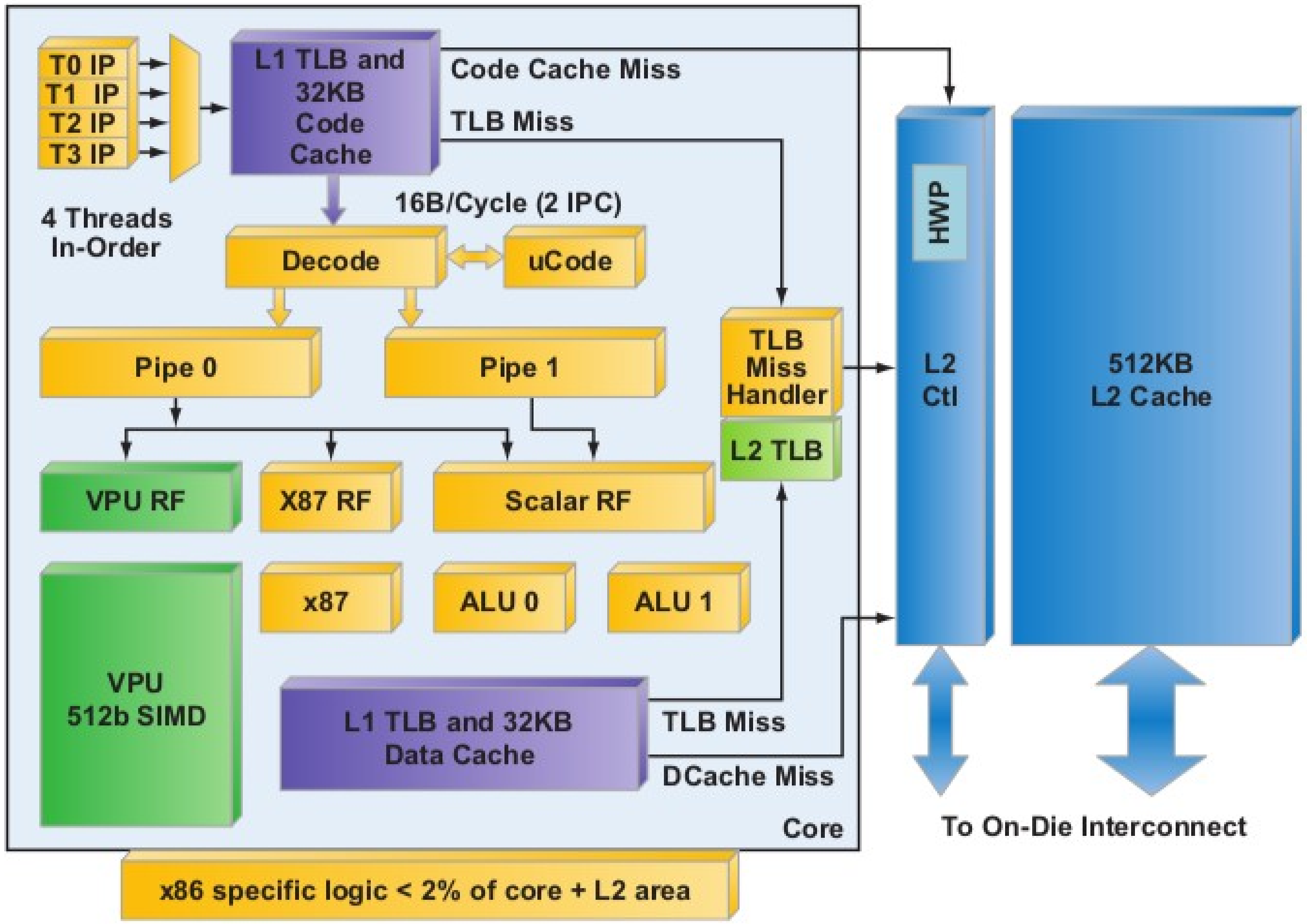
In this sense, the Xeon Phi is completely different from a GPU. This is why we often call the Xeon Phi “co-processor” rather than “accelerator”.

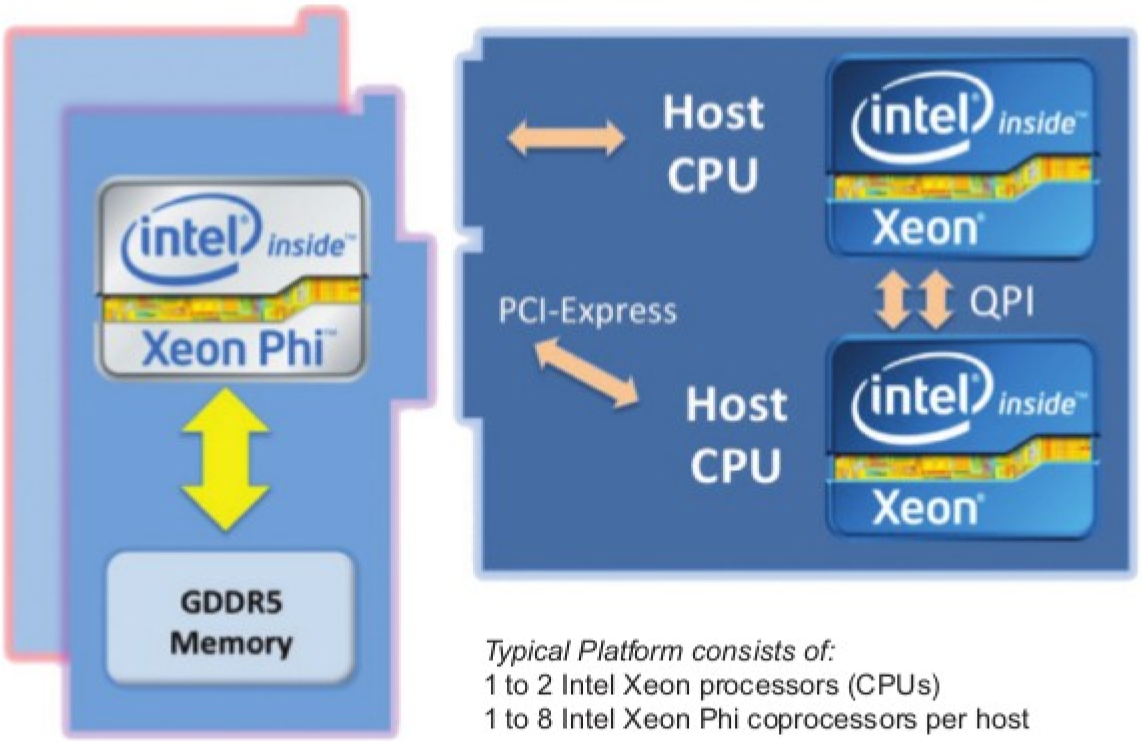

Intel Xeon Phi coprocessor Peak
Intel Xeon processor Peak
Sample Xeon perf.
Sample Xeon Phi perf.
Performance
Threads
0
200
400
600
800
1000
1200
1400
1600
1800
2000
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 106 111 116


800
600
400
200
0
Performance
1
6
11
16
Threads
Sample Xeon perf.
Sample Xeon Phi perf.

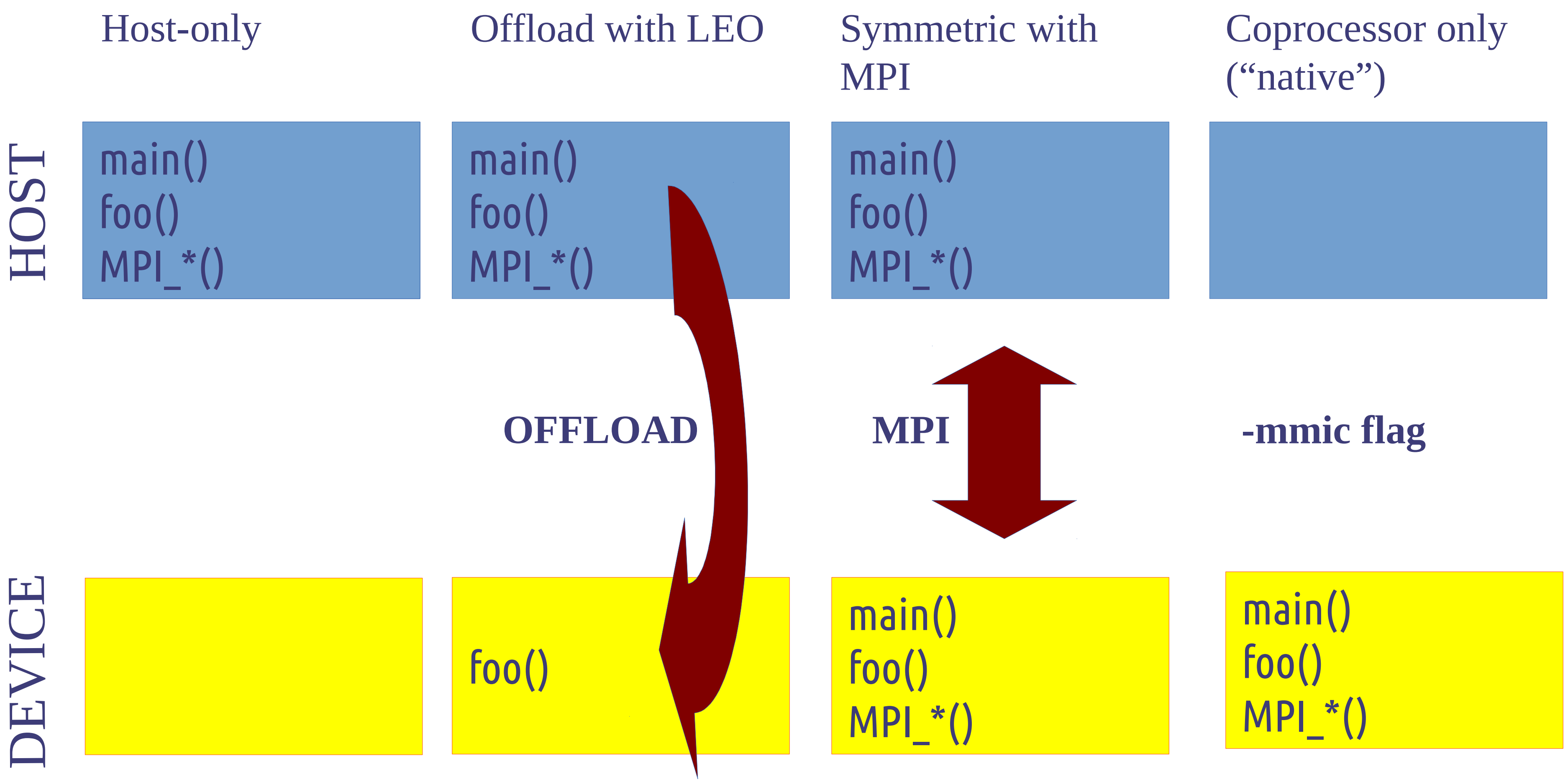

800
600
400
200
0
Performance
1
6
11
16
21
26
31
36
41
46
51
Threads
Sample Xeon perf.
Sample Xeon Phi perf.


PCIe Client Logic
Core
L2
Core
L2
GDDR MC
GDDR MC
GDDR MC
GDDR MC
TD
TD
TD
L2
Core
L2
Core
Core
L2
Core
L2
TD
TD
TD
TD
TD
GDDR MC
GDDR MC
GDDR MC
GDDR MC
L2
Core
L2
Core


T0 IP
T1 IP
T2 IP
T3 IP
L1 TLB and 32KB Code Cache
Code Cache Miss
TLB Miss
4 Threads In-Order
16B/Cycle (2 IPC)
Decode
uCode
Pipe 0
Pipe 1
TLB Miss Handler
L2 TLB
VPU RF
X87 RF
Scalar RF
VPU 512b SIMD
x87
ALU 0
ALU 1
L1 TLB and 32KB Data Cache
TLB Miss
DCache Miss
Core
HWP
L2 Ctl
512KB L2 Cache
To On-Die Interconnect
x86 specific logic < 2% of core + L2 area



Typical Platform consists of:
1 to 2 Intel Xeon processors (CPUs)
1 to 8 Intel Xeon Phi coprocessors per host


Host-only
Offload with LEO
Symmetric with MPI
Coprocessor only ("native")
HOST
DEVICE
main()
foo()
MPI_*()
main()
foo()
MPI_*()
main()
foo()
MPI_*()
OFFLOAD
MPI
-mmic flag
foo()
main()
foo()
MPI_*()
main()
foo()
MPI_*()

Coprocessor only (“native”)

PRO:

- it is a cross-compiling mode
- very simple
    - just add -mmic, login and execute
- use well known OpenMP and MPI (or pthreads or OpenCL)

CONS:

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

Offload (1)

Intel provides a set of directives to the compiler named “LEO”: Language Extension for Offload.

These directives manage the transfer and execution of portions of code to the device.

C/C++

```
#pragma offload target (mic:device_id)
```

Fortran

```
!dir$ offload target (mic:device_id)
```

Offload (2)

Variable and function definitions

C/C++

```
__attribute__ ((target(mic)))
```

Fortran

```
!dir$ attributes offload:mic :: <function/var name>
```

It compiles (allocates) variables on both the host and device

For entire files or large blocks of code (C/C++ only)

```
#pragma offload_attribute (push, target(mic))
#pragma offload_attribute (pop)
```

Offload (3)

Since host and device don't have physical or virtual shared memory, variable must be copied in an explicit or in an implicit way.

Implicit copy is assumed for
- scalar variables
- static arrays

Explicit copy must be managed by the programmer using clauses defined in the LEO

Offload (4)

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))`

Offload (5) - 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 (6) – Example: compiling function for CPU and Xeon Phi

C/C++

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

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

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
```

Offload (7)

Memory allocation

- CPU is managed as usual
- on coprocessor is defined by `in,out` and `inout` 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

Offload (8)

Using memory qualifiers

free_if(0)
free_if(.false.) retain target memory

alloc_if(0)
alloc_if(.false.) reuse data in subsequent offload

alloc_if(1)
alloc_if(.true.) allocate new memory

free_if(1)
free_if(.true.) deallocate memory

Offload (9)

note: specify the device_id when using more than one device
target(mic:0)
target(mic:1) ...

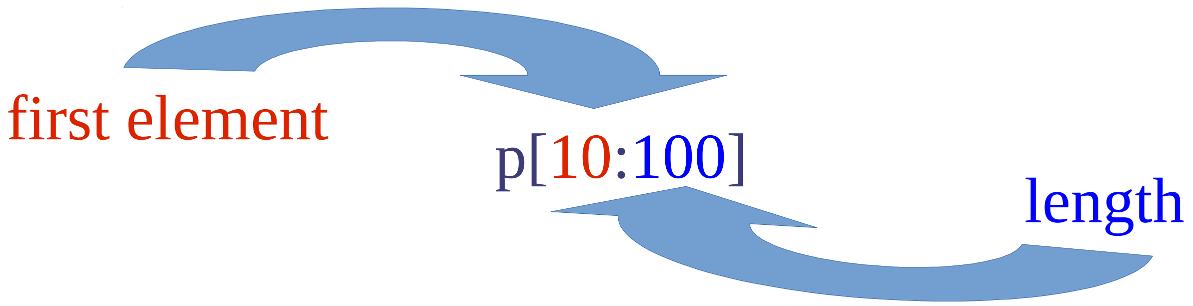
```
#define ALLOC alloc_if(1)
#define FREE free_if(1)
#define RETAIN free_if(0)
#define REUSE alloc_if(0)

#allocate the memory but don't de-allocate
#pragma offload target(mic:0) in(a:length(8)) ALLOC RETAIN)
...

#don't allocate or deallocate the memory
#pragma offload target(mic:0) in(a:length(8) REUSE RETAIN)

#don't allocate the memory but de-allocate
#pragma offload target(mic:0) in(a:length(8) REUSE FREE)
```

Offload (10)

Partial offload of arrays

```
int *p;
#pragma offload ... in (p[10:100] : alloc(p(5:1000))
{...}
```

It allocates 1000 elements on coprocessor; first usable element has index 5, last has index 1004; only 100 elements are tranferred, starting from index 10.

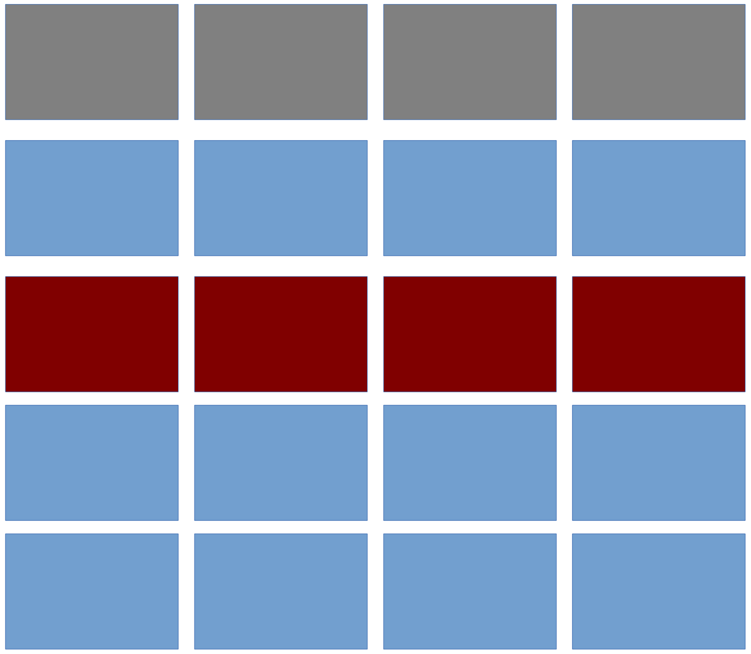


Offload (11)

Allocation of multidimensional arrays

```
int [4][4] *p;
#pragma offload ... in ( (*p)[2][:] : alloc(p[1:4][:]))
{...}
```

- `alloc p[1:4][:]` allocates 16 elements in a 5x4 shape
- first row is not allocated
- only row 2 is transferred



Offload (12)

Copy from a variable to another one

It permits to copy data from the host to a different array allocated on the device

```
integer :: p(1000), p1(2000)
integer :: rank1(1000), rank2(10,100)

!dir$ offload ... (p(1:500) : into (p1(501:1000)))
```

Offload (13)

Using OpenMP in an offload region:

optional, if defined, it must be immediately followed by a openmp directive

C/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
```

Setting up the environment:

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

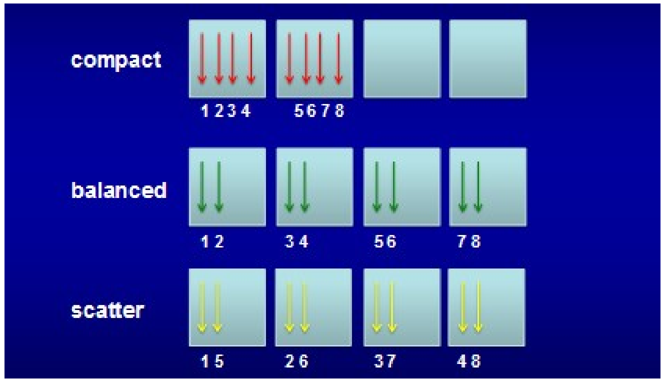
Offload (14)

Tuning up OpenMP

- the coprocessor has 4 hardware threads per core; at least 2 should be used;
- for many-cores systems (and hence for the Xeon Phi) binding threads to the cores and choosing an affinity are crucial factor that affect performance

```
MIC_ENV_PREFIX = MIC
MIC_KMP_AFFINITY  =
```

- compact
- scattered
- balanced



Offload (15)

LEO compiler options

- `-opt-report-phase:offload` activate reporting
- `-no-offload` disable offload
- `-offload-attribute-target=mic` build ALL functions for both host and device

Offload (16)

Static libraries

- xiar can be used to create libraries containing offloaded code
- specify -qoffload-build that forces xiar to create both a library for the host (xxx.a) and a library for the device (xxxMIC.a)
- use the same options that you would use for ar
- use normally the linker options (-L.. -lxxx.a) and the compiler will automatically include the coprocessor library

Offload (17)

Managing multiple devices

Including offload.h `#include <offload.h>` you can use a few API:

```
_offload_number_of_device()
_offload_get_device_number()
```

or use runtime environmente variables:

```
OFFLOAD_DEVICES=0,1
```

always remember to specify the target device `target(mic:1)`

Offload (18)

MIC specific MACROs

```
#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");
  }
}
```

Offload (19)

I/O from offloaded regions

- Buffered printf are available (use only to debugging purposes!)
- Always use fflush(0) on the coprocessor

- Files I/O is possible only through a proxy filesystem

Offload (20): I/O example

```
#pragma offload_attribute (push, target(mic))
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#pragma offload_attribute(pop)
int main(){
  FILE *fp;
  char buffer[7];
  #pragma offload target(mic) nocopy(fp)
  {
    fp=fopen(“./proxyfs/myfile.txt”,”wb”);
    if(fp == NULL)
    {
      fprintf(stderr,”failed to open myfile.txt\n”);
      exit(1);
    }
  fwrite(“Hello\n”,1,7,fp);
  fclose(fp);
  }
return 0;
}
```

Offload (20)

Asynchronous computation

By default, offload forces the host to wait for completion

- Asynchronous offload starts the offload and continues on the next statement just after the offload region
- Use the `signal` clause to synchronize with a `offload_wait` statement
- Always refer to a specific mic target

Offload (21)

Example

```
char signal_var;
do {
  #pragma offload target(mic:0) signal(&signal_var)
  {
     long_running_mic_compute();
  }
  concurrent_cpu_computation();
  #pragma offload_wait target(mic:0) wait(&signal_var)
} while(1);
```

Offload (22)

Reporting

Use `OFFLOAD_REPORT` or the variable `_Offload_report` with a verbosity from 1 to 3.
`OFFLOAD_REPORT=1` only provides timing

Conditional offload

Only offload if it is worth

```
#pragma offload target (mic) in (b:length(size)) \
                             out (a:length(size) \
                             if(size>100)
```

Native mode

```
icc -mmic mycode.c -o mycode.x
scp mycode.x mic0:.
ssh mic0
export OMP_NUM_THREADS=240
./mycode.x
```

Simple... but not always successful

Symmetric mode (1)

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..)

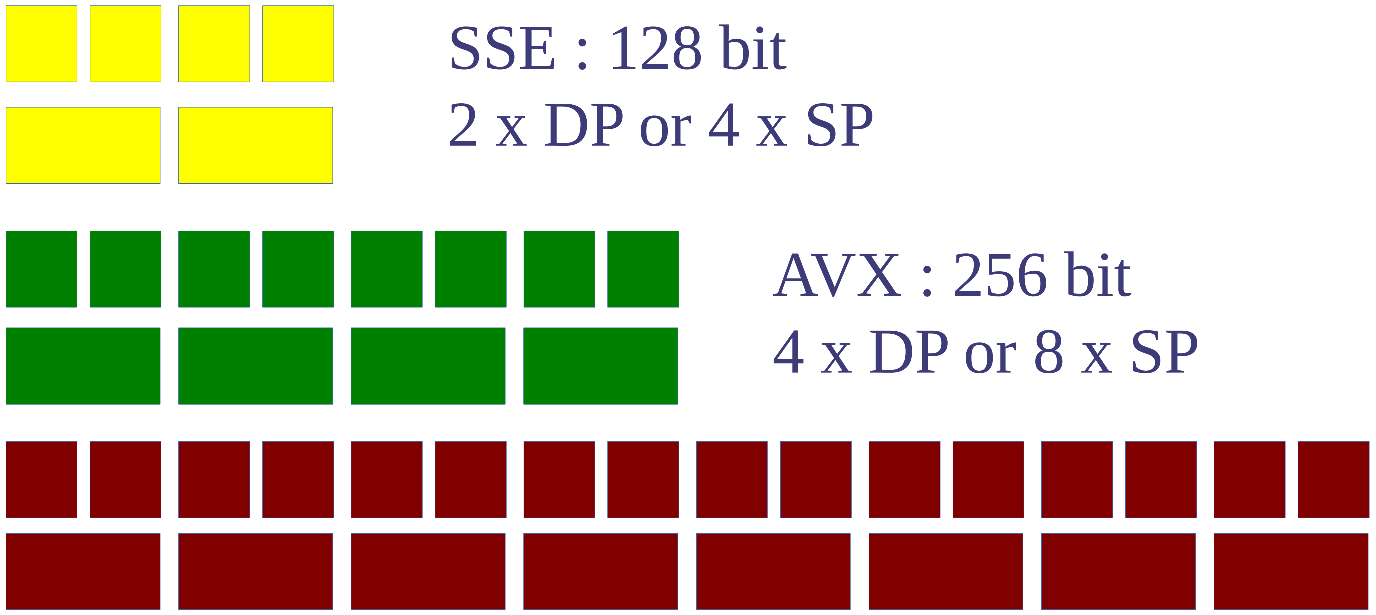
Symmetric mode (2)

Example

```
# compile the program for the coprocessor (-mmic)
mpiicc -mmic -o test.MIC test.c
# compile the program for the host
mpiicc -mmic -o test test.c

#copy the executable to the coprocessor
scp test.MIC mic0:/tmp/test.MIC

#set the I_MPI_MIC variable
export I_MPI_MIC=1

#launch MPI jobs on the host knf1 and on the coprocessor mic0
mpirun -host knf1 -n 1 ./test : -n 1  -host mic0 /tmp/test.MIC
```

Vectorization (1)

Lots of cores but also... large registers!



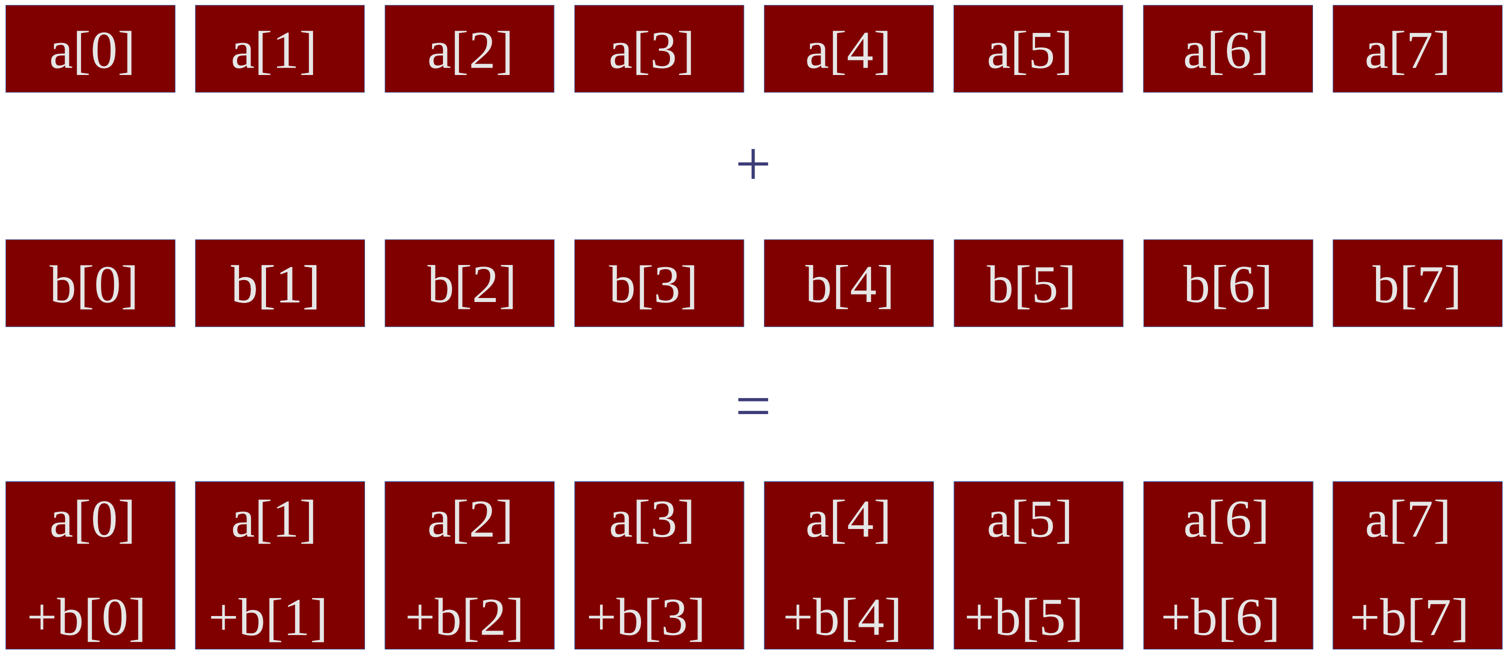
SSE : 128 bit
2 x DP or 4 x SP

AVX : 256 bit
4 x DP or 8 x SP

MIC : 512 bit
8 x DP or 16 x SP

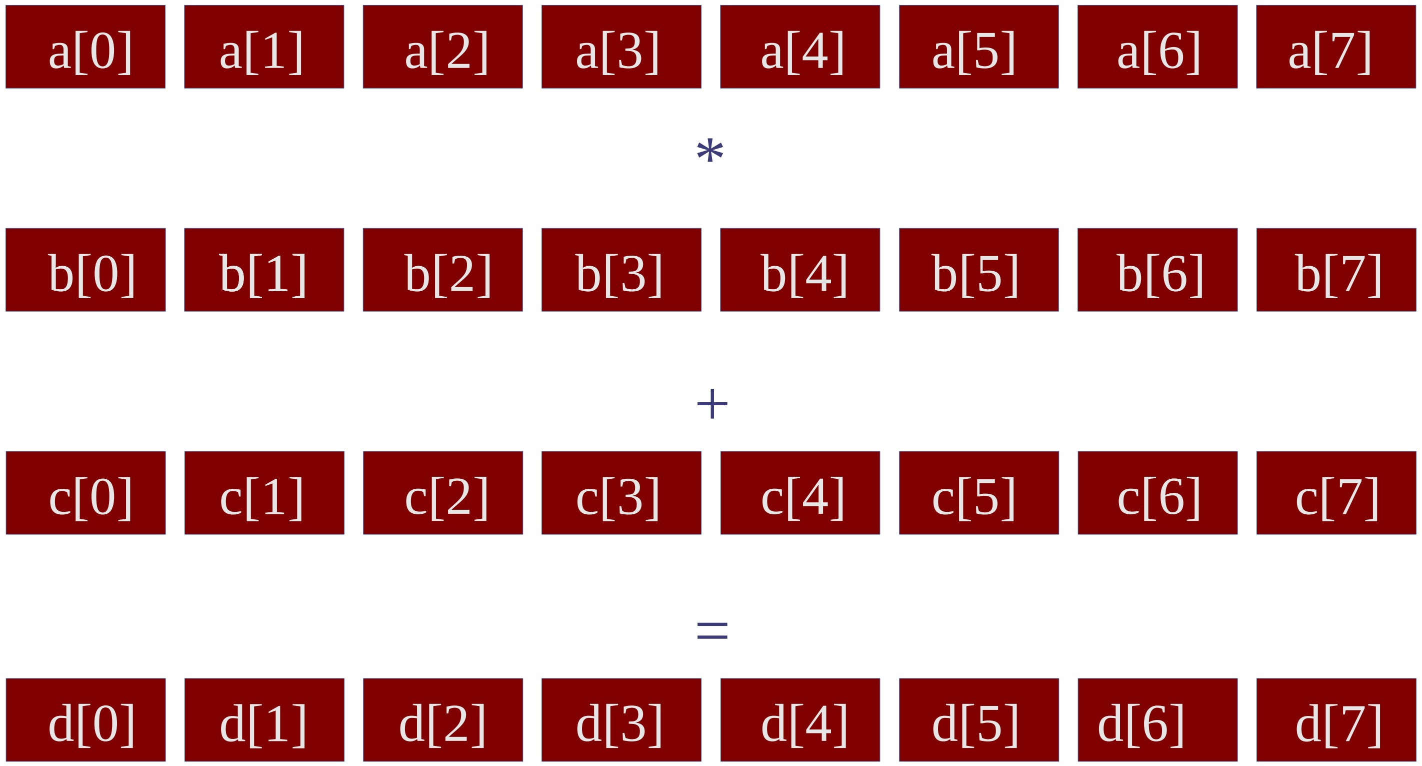
Vectorization (2)

SIMD arithmetic



Vectorization (3)

SIMD Fused Multiply Add


a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
*
b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7]
+
c[0] c[1] c[2] c[3] c[4] c[5] c[6] c[7]
=
d[0] d[1] d[2] d[3] d[4] d[5] d[6] d[7]

Vectorization (4)

How to vectorize my code?

First option: **believe** in your compiler.

However a compiler is pretty much conservative and could require to be supported.

Use `-vec-report [n]` where n=1,5 to get info about what the compiler has done (and why it didn't vectorize something)

If you're using C/C++ take care of *aliasing*: use `restricted` keyword

Vectorization (5)

If the compiler is too shy, you can help with pragma...

`#pragma vector` `!dir$ vector`	tells to the compiler that the loop should be vectorized
`#pragma simd` `!dir$ simd`	enforces the vectorization of the innermost loop

SIMD support will be added in the OpenMP 4.0

Vectorization (6)

Alignment

Data properly aligned facilitate access in memory

For better performance align data to

- 16 byte for SSE instructions
- 32 byte for AVX instructions
- 64 byte for MIC instructions

Use

```
#pragma offload in (a:length(size) align(64))
```

In conclusion... is everything so simple???

Remember:

- in-order cores
- limited pre-fetching
- low clock frequency (~1GHz)
- Small caches
- Amdahl law
- Small main memory
- PCIe is a bottleneck
- MPI latency on network ring
- ...

Intel MKL libraries on Intel Xeon Phi

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

Automatic offload (AO) (1)

- 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()`

Automatic offload (2)

Not all the MKL functions are enabled to AO.

In MKL 11.0.1:

- *Level 3 BLAS*: xGEMM, xTRSM, xTRMM
- *LAPACK* xGETRF, xPOTRF, xGEQRF

Always check the documentation for updates

Compiler assisted offload (1)

- 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...)

Compiler assisted offload (2)

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)
```

Native execution

MKL libraries are also available when using the native mode.

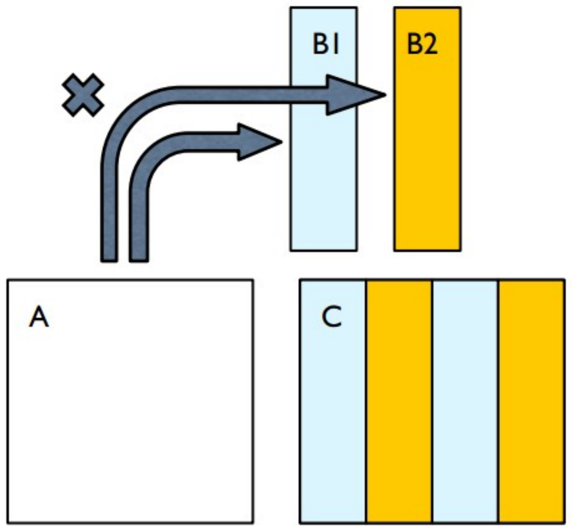
Tips:

Use all the 240 threads: MIC_OMP_NUM_THREADS=240

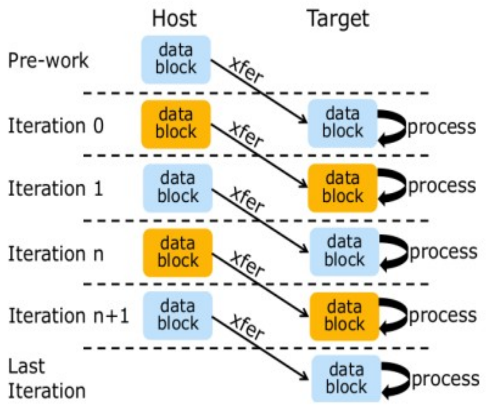
Set the thread affinity: MIC_KMP_AFFINITY = ...

Backup slides

An example:
hiding the latency with the “double-buffer” technique



- partition B and C matrices
- overlap communication with computation



Preparing the buffers..

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

#pragma offload target(mic:0) \
        nocopy( A[0:M*K] : align(Align) ALLOC RETAIN) \
        nocopy( C1[0:M*N2] : align(Align) ALLOC RETAIN) \
        nocopy( C2[0:M*N2] : align(Align) ALLOC RETAIN) \
        nocopy( B1[0:M*N2] : align(Align) ALLOC RETAIN) \
        nocopy( B2[0:M*N2] : align(Align) ALLOC RETAIN)
```

Even iterations

```
#pragma offload_transfer target(mic:0) if(ii!=BLOCKS-1) \
    in(B([(ii+1)*K*N2:K*N2]:align(Align)REUSE RETAIN \
    into(B2[0:K*N2])) \
    in(Cg[(ii+1)*M*N2:M*N2]:align(Align) REUSE RETAIN \
    into (C2[0:M*N2])\
    signal(C2)
```

Transfer

```
#pragma offload target(mic:0) \
    nocopy(A[0:M*K]:align(Align) REUSE RETAIN) \
    nocopy(B1[0:K*N2]:align(Align) REUSE RETAIN) \
    out(C1[0:M*N2]:align(Align) REUSE RETAIN \
    into(Cg[ii*M*N2:M*N2])) \
    wait(C1)
{   gemm(A,B1,C1);   }
```

Compute

Odd iterations

```
#pragma offload_transfer target(mic:0) if (ii !=
BLOCKS-1 )\
    in(B[(ii + 1)*K*N2:K*N2]:align(Align) REUSE RETAIN \
    into(B1[0:K*N2])) \
    in(Cg[(ii + 1)*M*N2:M*N2]:align(Align) REUSE RETAIN \
    into(C1[0:M*N2])) \
    signal(C1)
```

Transfer

```
#pragma offload target(mic:0) \
    nocopy( A[0:M*K]:align(Align) REUSE RETAIN) \
    nocopy(B2[0:K*N2]:align(Align) REUSE RETAIN) \
    out(C2[0:M*N2]:align(Align) REUSE RETAIN \
    into(Cg[ii*M*N2:K*N2])) \
    wait(C2)
    { gemm(A, B2, C2); }
```

Compute

Deallocating buffers...

```
#pragma offload target(mic:0) \
   nocopy(0A[0:M*K]0:align(Align) REUSE FREE)\
   nocopy(C1[0:M*N2]:align(Align) REUSE FREE)\
   nocopy(C2[0:M*N2]:align(Align) REUSE FREE)\
   nocopy(B1[0:M*N2]:align(Align) REUSE FREE)\
   nocopy(B2[0:M*N2]:align(Align) REUSE FREE)
```

Bibliography:

Jeffers, Reinders: **Intel Xeon Phi Coprocessor High-Performance programming.** Morgan Kauffman Publishers

R. Rahman: **Intel Xeon Phi Coprocessor architecture and tools.** Apress Open