High-Performance Computing on the Intel® Xeon Phi™

Lu Xiaowei卢晓伟 lu_xw@inspur.com

18600597385

Inspur(Beijing)Electronic Information Industry Co.,Ltd

Outline

1. MIC Introduction
2. MIC Programming
3. MIC technology innovation and advantage
4. Performance Optimization on MIC
5. Development Process of HPC Application Based on MIC
6. Performance achievements of the Gridding Algorithm

MIC INTRODUCTION

MIC Introduction

What : what is MIC ?

Why : why MIC is used ?

When : when MIC is used ?

Where : where MIC is used ?

Who : WHO use MIC ?

Worldwide MIC work Introduction

What : what is MIC ?

Why : why MIC is used ?

When : when MIC is used ?

Where : where MIC is used ?

Who : WHO use MIC ?

Actual MIC machine

Intel CPU

Intel MIC

What is MIC

MIC Architecture Name

- Intel Many Integrated Core

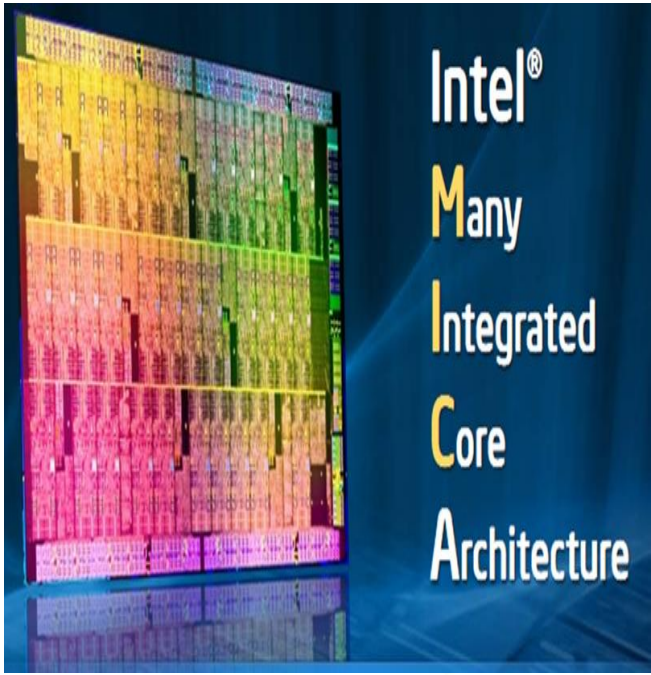
MIC Core’s key components

- General instruction unit on X86 arch
- 512bit vector-bit width VPU
- Support 4Thread/core

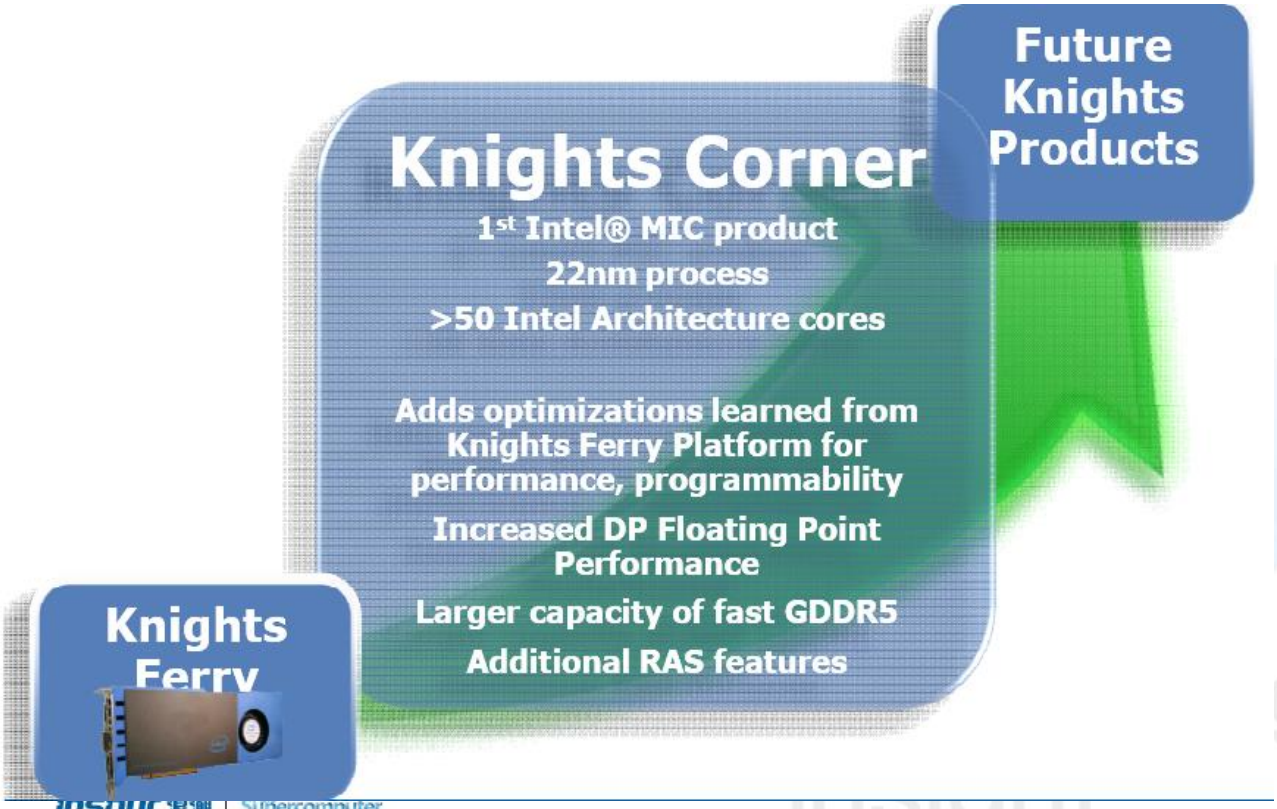
Single core’s capability

- Weaker than CPU Core
- Greater than GPU Core

- Many core, above 50 X86 cores
  - Double precision performance >1TFlops



MIC Product Roadmap



MIC Introduction

What : what is MIC ?

Why : why MIC is used ?

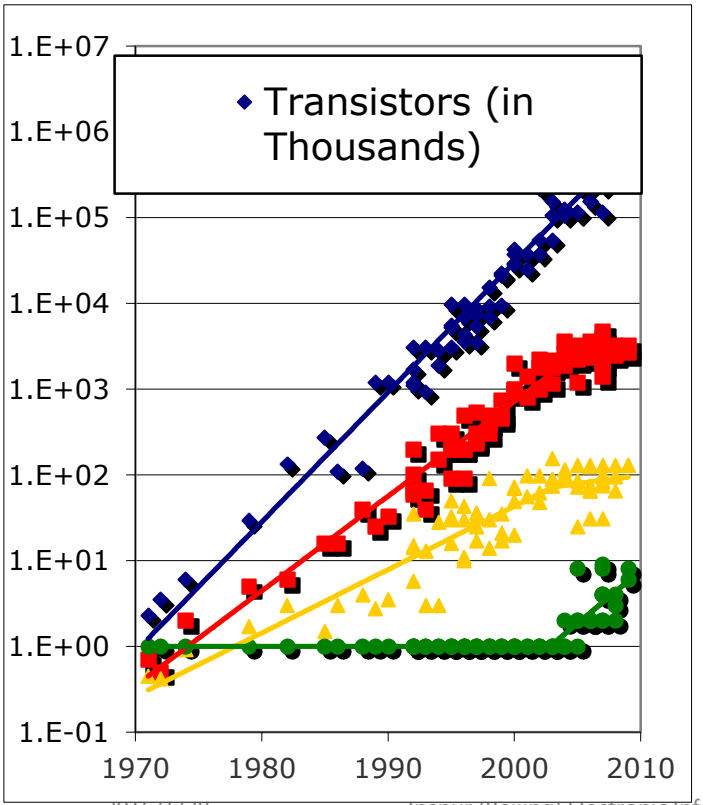
When : when MIC is used ?

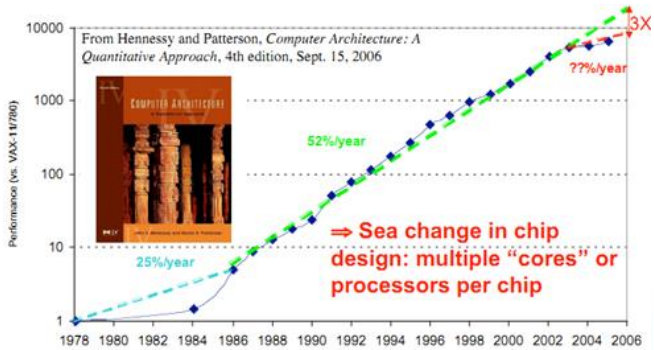
Where : where MIC is used ?

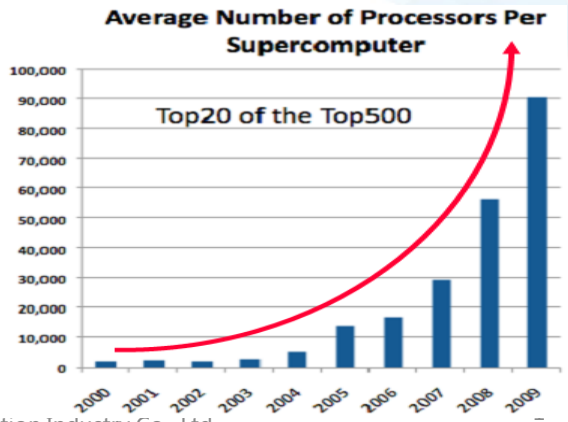
Who : WHO use MIC ?

“…the number of tansistors on a chip will double about evry two years…”

Gordon Moore, Intel co-funder







Moore's Law

...the number of transistors on a chip will double about every two years...

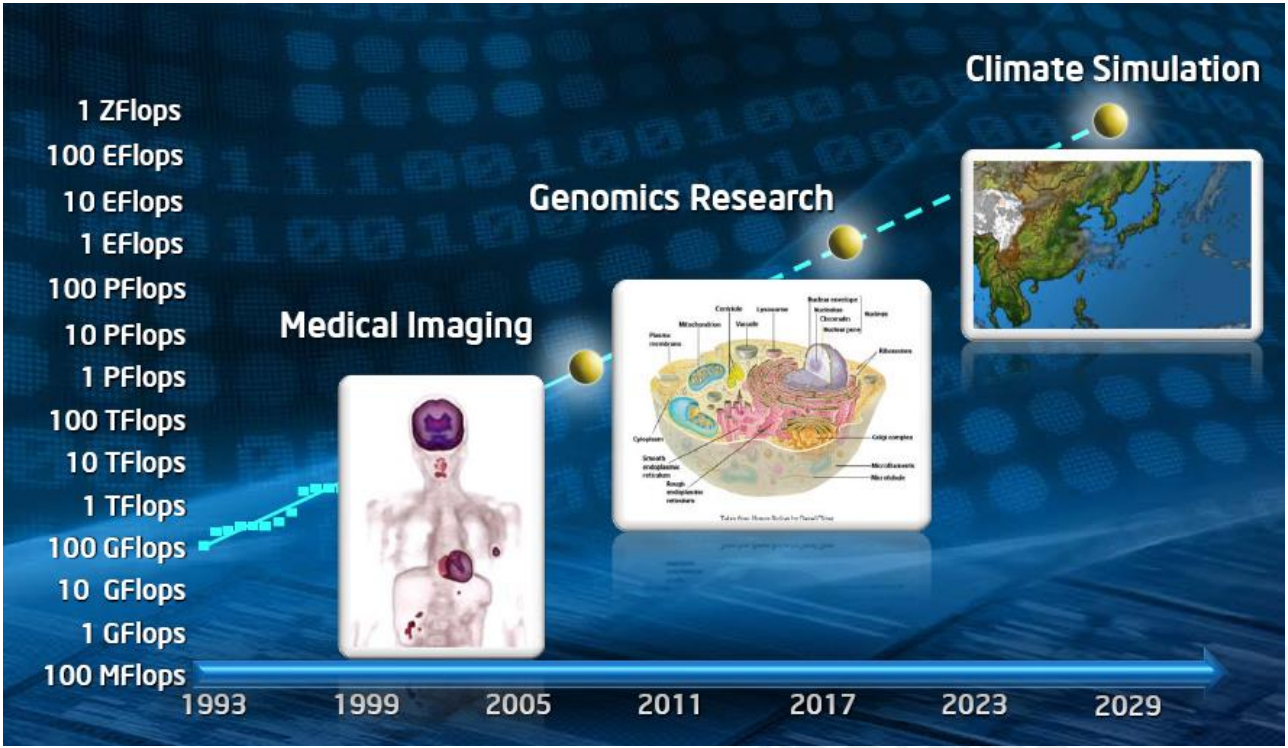
NERSC

Moore's law can continue only by increasing the concurrency

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY
Managed by the University of California for the U.S. Department of Energy

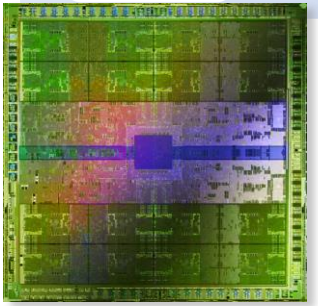
- Clock frequency can't continue to ascend
- challenge of dealing with millions of a concurrent threads(for ExaScale level system)
- deal with the parallelism internal& between the chip

“Requirements of HPC is growing and growing...”

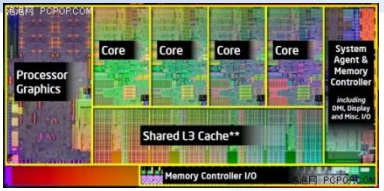


“…Parallel computing is the future…”

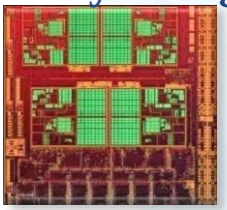
Bill Dally, Chief Scientist & Sr. VP of Research, NVIDIA



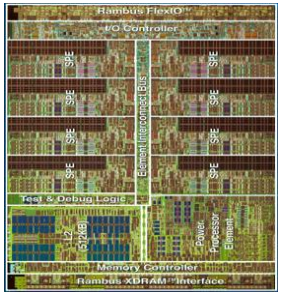
NVIDIA Fermi (512 cores)



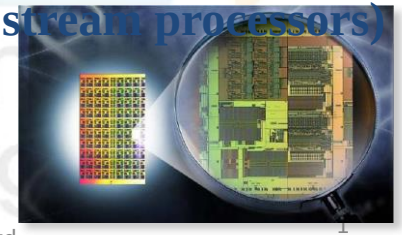
Intel Sandy Bridge



AMD Llano (4 x86 cores + 480 stream processors)



IBM Cell B.E (2PPE + 8 SPE)



MIC Introduction

What : what is MIC ?

Why : why MIC is used ?

When : when MIC is used ?

Where : where MIC is used ?

Who : WHO use MIC ?

HPC development trend--highly parallel

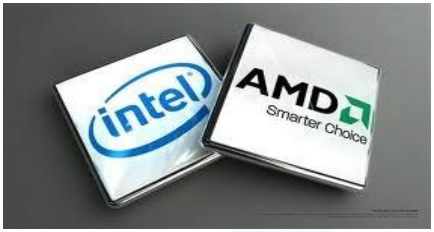
due to the rapid development of multicore / manycore technologies

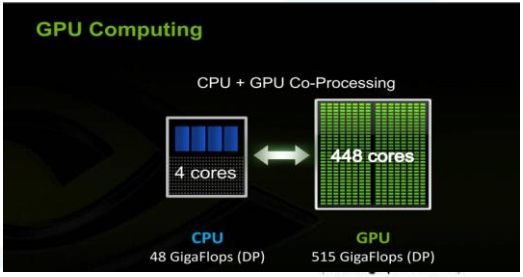
multicore of same architecture

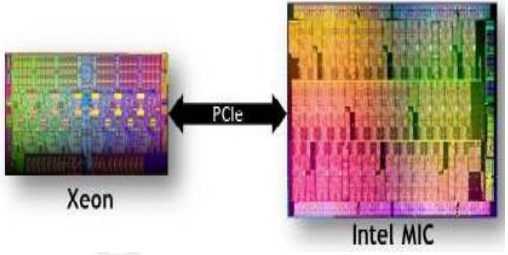
- CPU multicore Parallel computing
  - one single node：OpenMP/pThread
  - cluster：MPI

Heterogeneous manycore

- CPU+GPU cooperation-Computing
  - CPU Primarily responsible for logic operations
  - GPU Primarily responsible for arithmetic operations
- CPU+MIC cooperation-Computing
  - upcoming







Intel-Inspur parallel computing joint-lab

Founded on August 24, 2011

Face to Exascale computing subject

- Research on single MIC card
- Research of cooperative Heterogeneous computing on CPU+MIC
- Research of cooperative heterogeneous computing on CPU+MICs
- Research of heterogeneous computing on CPUs+MICs

- Research mode:
  - Client + Inspur
  - Client of inspur： BGP、CAS、
  - SGRI、University

Worldwide MIC work Introduction

What : what is MIC ?

Why : why MIC is used ?

When : when MIC is used ?

Where : where MIC is used ?

Who : WHO use MIC ?

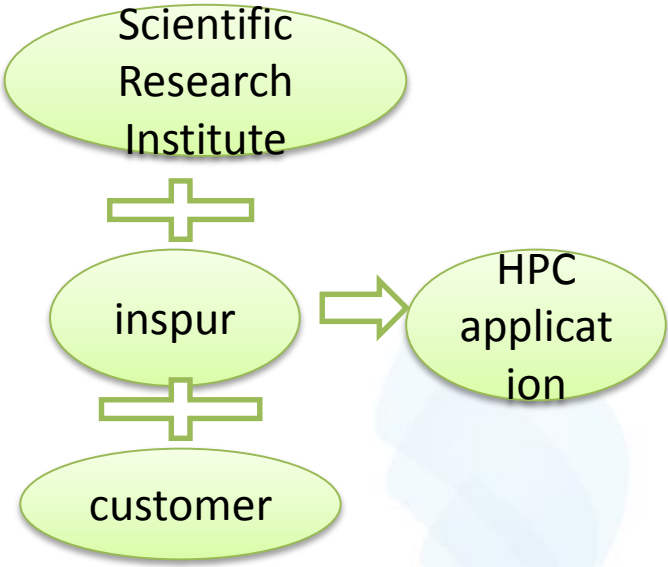
Cooperative R&D model

Researcher

- Cutting-edge technology

operation mode

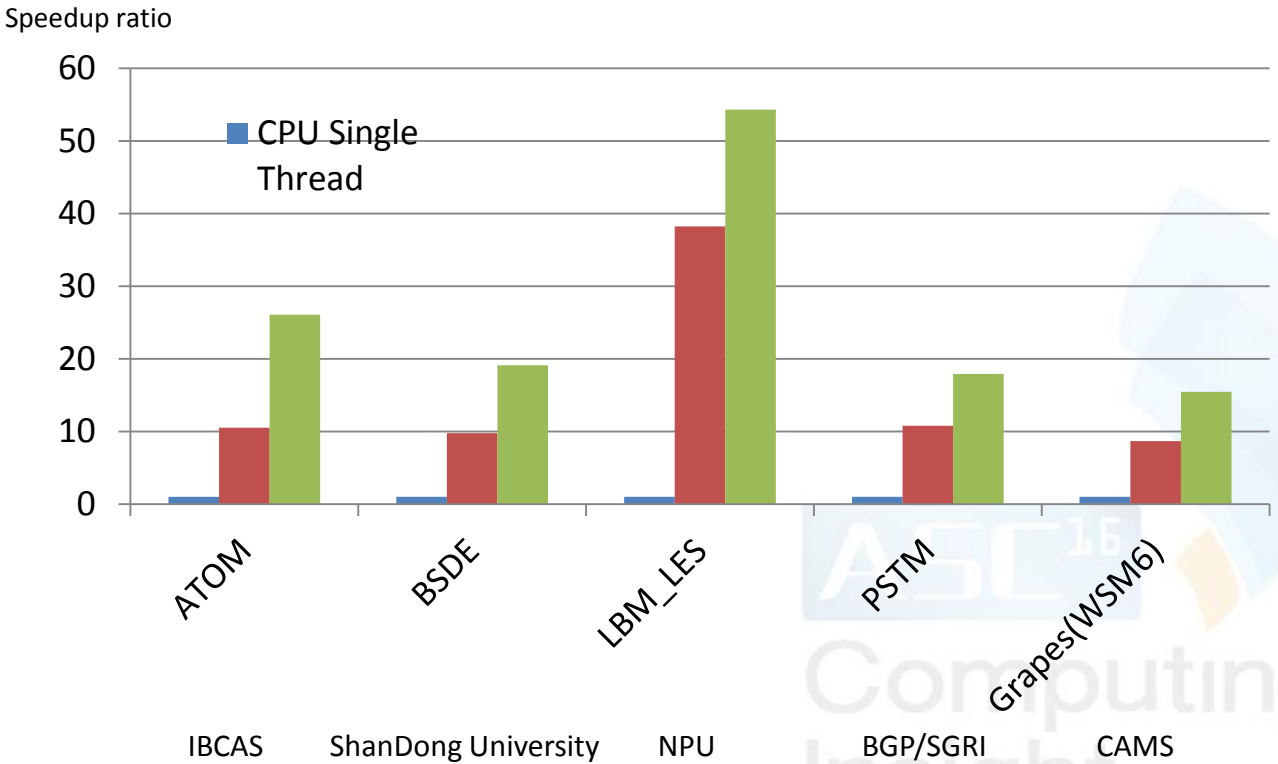
- Own algorithm, but the app can be developed **not necessarily**



- Inspur customer :

China Petroleum BGP、Sinopec SGRI、Chinese Academy of Sciences , some universities

Transplant results to MIC of 5 typical application



MIC PROGRAMMING

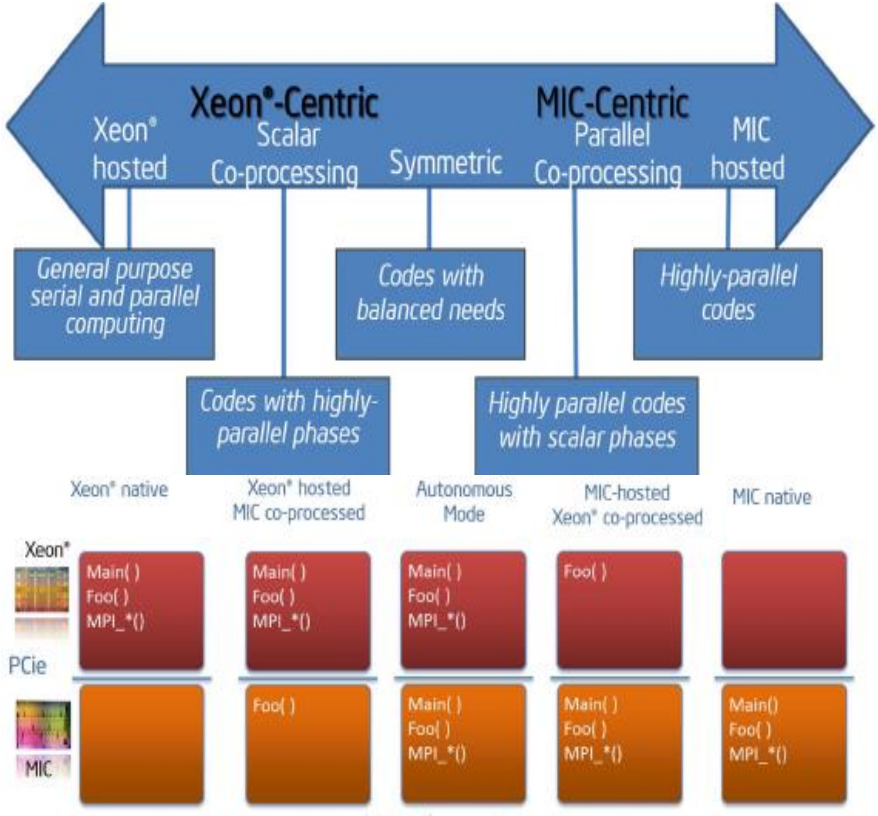
MIC running mode

Perfect combination of MIC and CPU

MIC: the extension of CPU

Five applicatn modes:

- serial+parallel mode
  - Low concurrency
  - CPU hosted
- serial+high parallel mode
  - offload
- Symmetric mode
- High parallel +serial mode
  - MIC hosted
  - CPU co-processed
- High parallel mode
  - MIC hosted



MIC common programming models

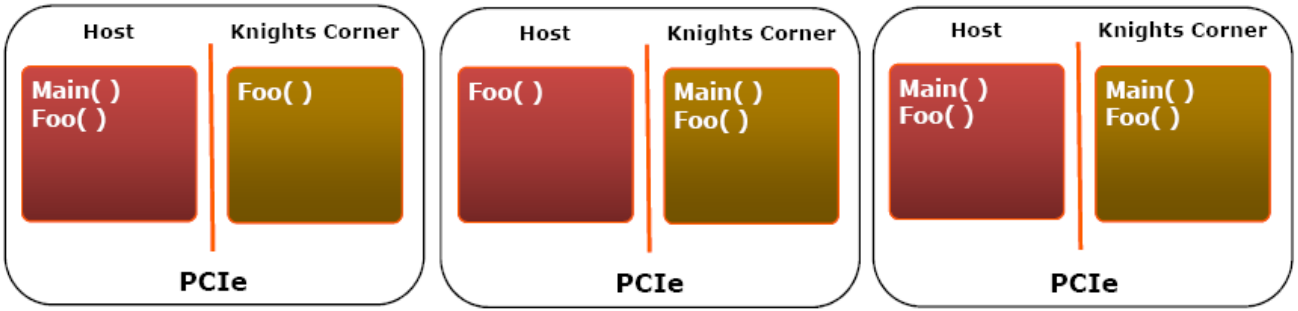
Offload mode (mostly common)

- Through the directive quotation, the code snippets is marked as MIC code, driven by the automatic loading, which run on MIC
- Grammar similar with OpenMP，simple
- The program runs on the CPU, section code will be executed on the MIC
- Set calling device, allocating memory, data transmission in a body

Native mode

- Binary files will be uploaded to the MIC card and run manually

- Symmetrical mode



OFFLOAD MODE

Offload Mode

What is offload mode ?

MIC used as a coprocessor and is in charge of partical computing.

How to use offload mode ?

Modify code, write directives similar to OpenMP.

What are the advantages of Offload mode ?

Flexible and Efficient

What are the limitations of Offload mode ?

Large communication overhead;

Need to modify code

Which case is suitable for Offload Mode?

It applies to serial programs containing a pratical highly parallel program.

Offload concept—host and device

Host—Device

Host：CPU Host Side

Run serial code;

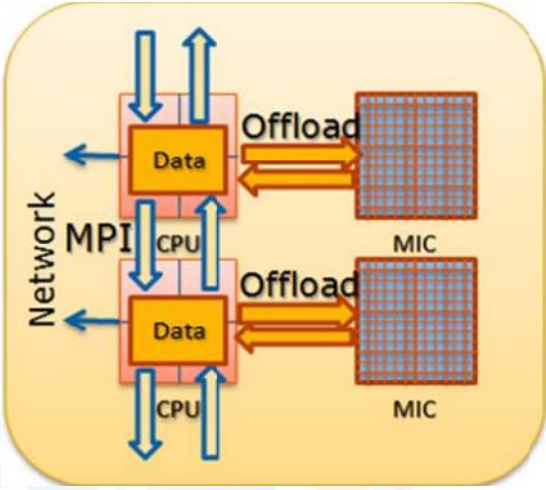
Control computing on CPU and MIC.

Device：MIC Device Side

Coprocessor of CPU;

Separate Device Memory;

Open massive threads to parallel computing.



The Modes of heterogenous computing

Fork-join mode

CPU “offload” MIC function

In MIC: parallel computing

In CPU: get the result after MIC computing

Simple example：MIC Offload

```
1.  #pragma offload target (mic)
2.      for(i=0;i<LEN;++i)
3.      {
4.          A[i]=B[i]+C[i];
5.      }
```

Loop and printf run on the MIC

Now loop is serial！

No data transferring

What is openmp

OpenMP is a API and can provide a simple method for writing multithreaded application, so the programmer doesn't need to make complicate thread creation, synchronous, load balance and destruction，can support Fortran, C and C++.

OpenMP uses quotation to parallel the program, the most common way is loop parallelization (the For loop of C/C++, the Do loop of Fortran).

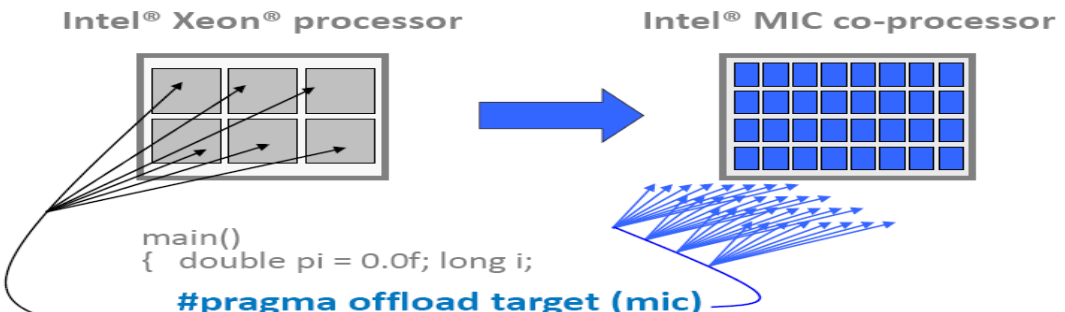
Openmp example

C/C++:

```
1        #pragma omp parallel for
2        for(i=0;i<LEN;++i)
3        {
4            a[i]=b[i]+c[i];
5        }
```

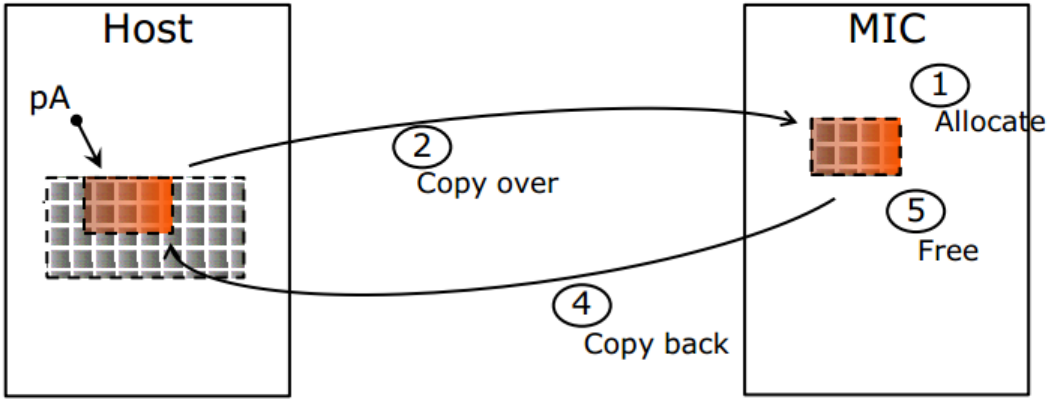
MIC Offload + OpenMP

- Offload+OpenMP
  - Offload in charge of memory allocating , data transferring
  - OpenMP in charge of paralleling
- Combined Offload with OpenMP, making codes running on the MIC
- Code example :
  - #pragma offload target (mic:ID)
  - #pragma omp parallel for …




```
main()
{  double pi = 0.0f; long i;
   #pragma offload target (mic)
   #pragma omp parallel for reduction(+:pi)
   for (i=0; i<N; i++)
   {
      double t = (double)((i+0.5)/N); pi += 4.0/(1.0+t*t);
   }
   printf("pi = %f\n",pi/N);    }
```

MIC Offload process




```
#pragma offload inout(pA:length(n))
{...}
```
3

MIC programming features

Simple

Hide many details

grammar：similar with OpenMP

Flexible

OpenMP、MPI、pThread

Traditional

Succession with CPU

MIC Offload data transferring

MIC Offload data transferring mode

- In : copy CPU to MIC
- Out : copy MIC to CPU
- Inout : copy both
- Nocopy : Data is local to MIC

```
float reduction(float *data, size_t size)
{
   float ret = 0.f;
   #pragma offload target(mic) in(data:length(size))
   {
      for (int i=0; i<size; ++i) {
         ret += data[i];}
   }
   return ret;
}
```

MIC grammar

Variables and pointers restricted to scalars, structs of scalars, and arrays of scalars

Clauses / Modifiers	Syntax	Semantics
Target specification	`target( name[:card_number] )`	Where to run construct
Conditional offload	`if (condition)`	Boolean expression
Inputs	`in(var-list modifiers`$_{opt}$`)`	Copy from host to coprocessor
Outputs	`out(var-list modifiers`$_{opt}$`)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers`$_{opt}$`)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers`$_{opt}$`)`	Data is local to target
Modifiers		
Specify pointer length	`length(element-count-expr)`	Copy N elements of the pointer's type
Control pointer memory allocation	`alloc_if ( condition )`	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	`free_if ( condition )`	Free memory used by pointer if condition is TRUE
Control target data alignment	`align ( expression )`	Specify minimum memory alignment on target

Declarations of Variables and Functions

	C/C++ Syntax	Semantics
Offload pragma	`#pragma offload <clauses>` `<statement>`	Allow next statement to execute on Intel® MIC Architecture or host CPU
Keyword for variable & function definitions	`__attribute__((target(mic)))`	Compile function for, or allocate variable on, both CPU and Intel® MIC Architecture

Entire blocks of code	`#pragma offload_attribute(push, target(mic))`
	`:`
	`#pragma offload_attribute(pop)`
		Mark entire files or large blocks of code for generation on both host CPU and Intel® MIC Architecture

	Fortran Syntax	**Semantics**
Offload directive	`!dir$ omp offload <clause>` `<statement>`	Execute next OpenMP* parallel construct on Intel® MIC Architecture
	`!dir$ offload <clauses>` `<statement>`	Execute next statement (function call) on Intel® MIC Architecture
Keyword for variable/function definitions	`!dir$ attributes offload:<mic> :: <ret-name> OR <var1,var2,…>`	Compile function or variable for CPU and Intel® MIC Architecture

Declaration of variable and function

__declspec(target (mic))

__attribute__ ((target (mic)))

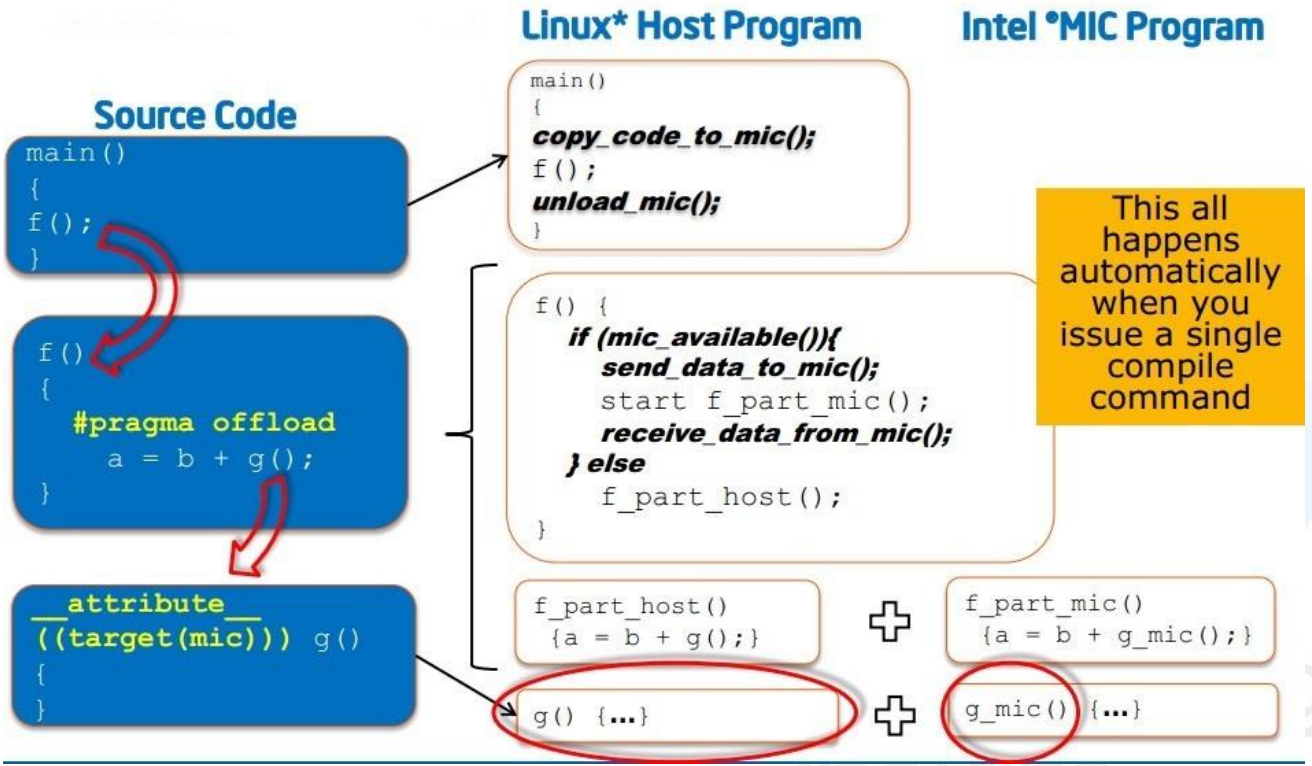
Once declared, functions and variables can be used for both CPU and MIC codes.

For example：

__attribute__ ((target (mic))) int a;

__attribute__ ((target (mic))) void func();

Heterogeneous Compiler – Conceptual Transformation



MIC compiling

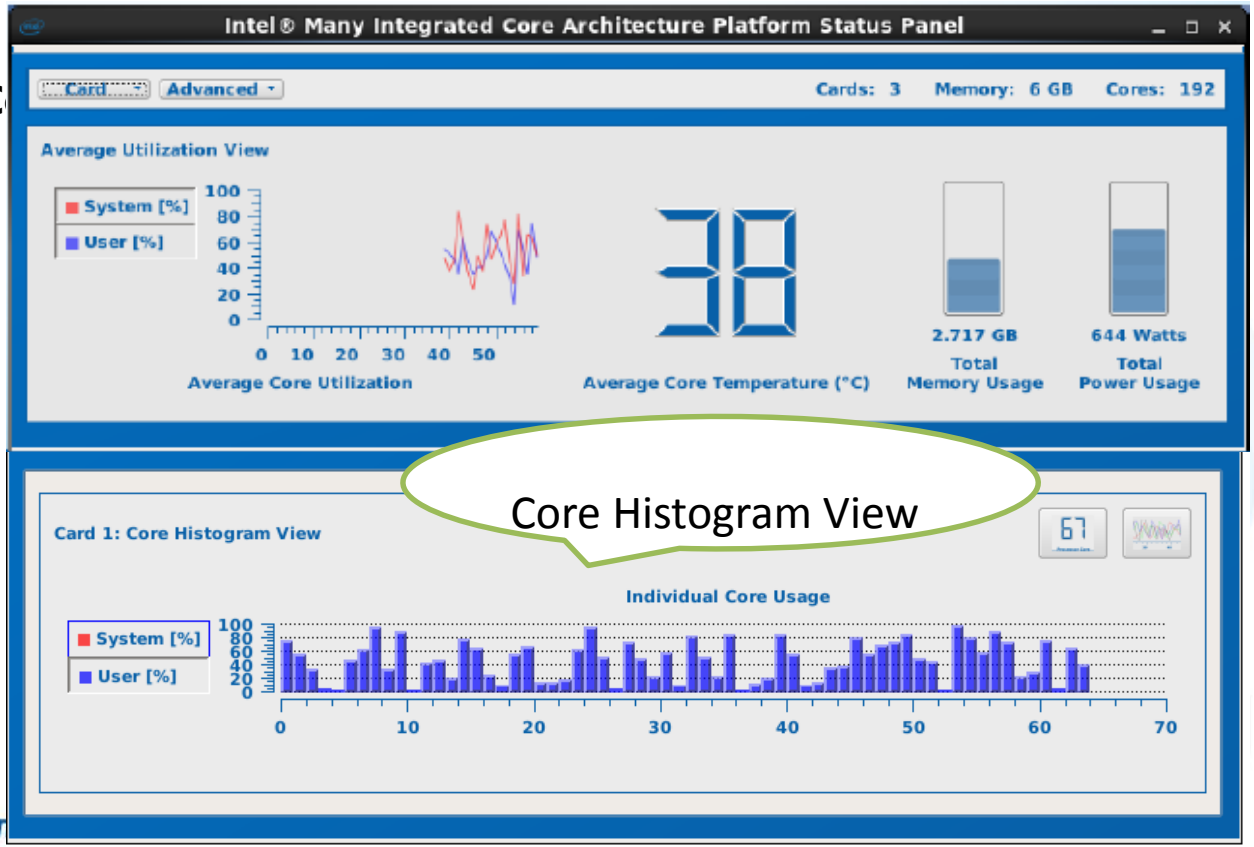
Intel compiler：icc/icpc/ifort

-no-offload

How to compile the MIC sample

```
//C/C++, compilation options：icpc main.cpp –openmp,run:./a.out
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <math.h>
4.  int main()
5.  {
6.      float pi = 0.0f;
7.      int count = 10000;
8.      int i;
9.  #pragma offload target (mic)
10. #pragma omp parallel for reduction(+:pi)
11.     for (i=0; i<count; i++)
12.     {
13.       float t = (float)((i+0.5f)/count);
14.       pi += 4.0f/(1.0f+t*t);
15.     }
16.     pi /= count;
17.
18.    printf("Pi = %f\n", pi);
19. }
```

MIC compile and example



Intel® MIC Architecture Native Compilation

- Purpose:
  - –Build standalone programs for execution directly on coprocessor
  - –Create coprocessor-specific libraries for use by offloaded code sections
- Use:
  - –Invoke the compiler with –mmic rather than –offload-build to generate purely "native" code
- Caveats:
  - –Standalone programs and their data need to be copied manually by the user using scp, ftp or copied using an offload proxy
  - –Shared libraries, such as libiomp5.so (which has no static counterpart) may need to be copied manually, even if you link your program statically.
- Performance analysis procedure is unchanged from offload code
- Debugging requires that you manually attach to the program while it runs on the coprocessor

Example

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <math.h>
4.  int main()
5.  {
6.      float pi = 0.0f;
7.      int count = 10000;
8.      int i;
9.   #pragma offload target (mic)
10. #pragma omp parallel for reduction(+:pi)
11.     for (i=0; i<count; i++)
12.     {
13.       float t = (float)((i+0.5f)/count);
14.       pi += 4.0f/(1.0f+t*t);
15.     }
16.     pi /= count;
17.
18.       printf("Pi = %f\n", pi);
19. }
```

Native compile、execution

Compile: icc pi.c -openmp -mmic -o mic_pi

Upload : scp ./mic_pi 172.31.1.1:/tmp

common upload directory is /tmp , everyone has the authority of writing ,maybe common upload directory is one’ s home path.

Login(MIC card): ssh –x 172.31.1.1

172.31.1.1 is the address of MIC card

Then execute: /tmp/mic_pi

/tmp/mic_pi: error while loading shared libraries: libiomp5.so: cannot open shared object file: No such file or directory

Solution: scp /opt/intel/composer_xe/compiler/lib/mic/libiomp5.so 172.31.1.1:/lib64

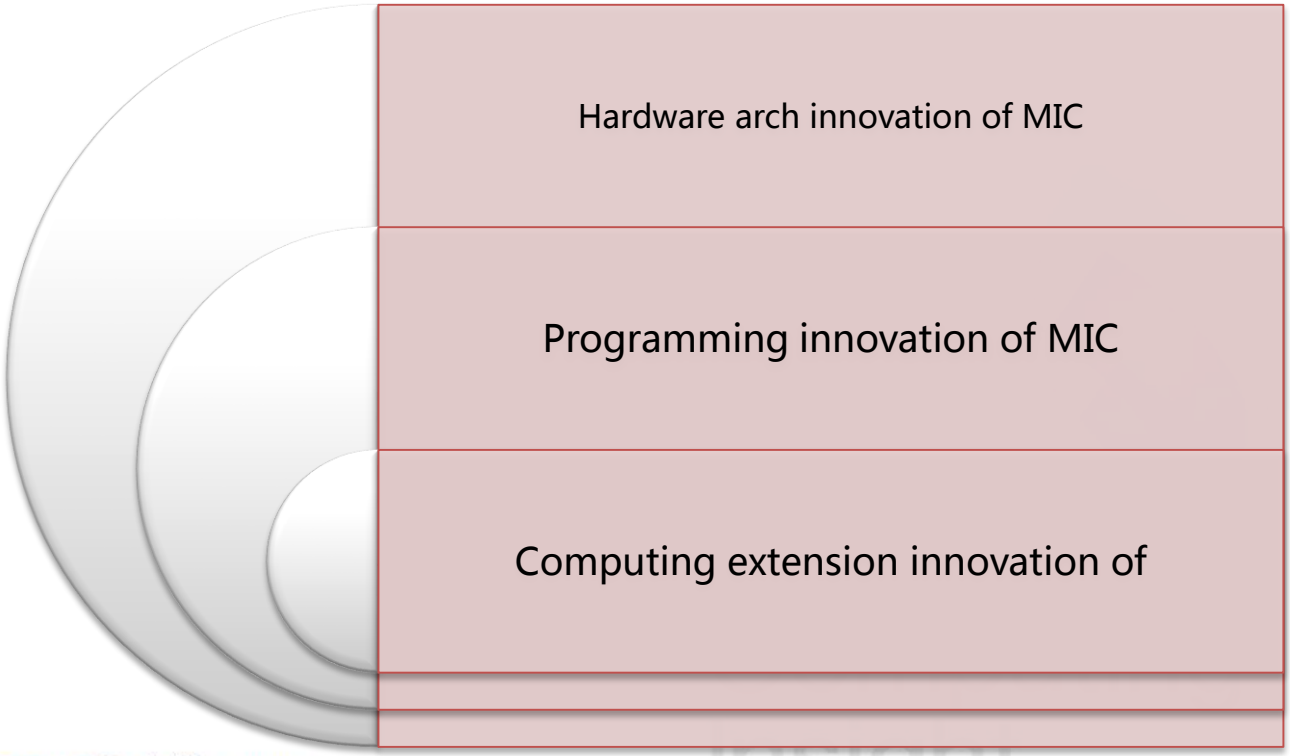
Then ssh –x 172.31.1.1 /tmp/mic_pi

Result Pi = 3.141592

Do not need to add any code based on the OpenMP

MIC TECHNOLOGY INNOVATION AND ADVANTAGE

Three technical innovations of MIC



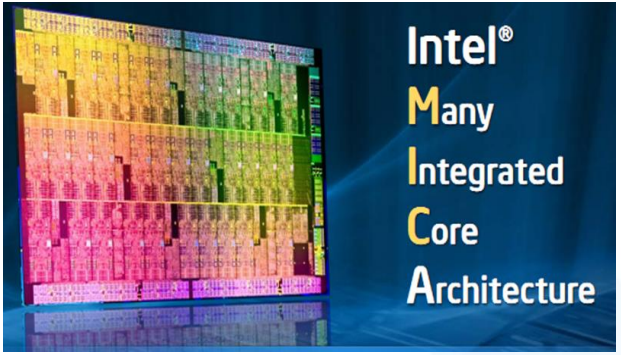
Innovation of MIC hardware Architecture

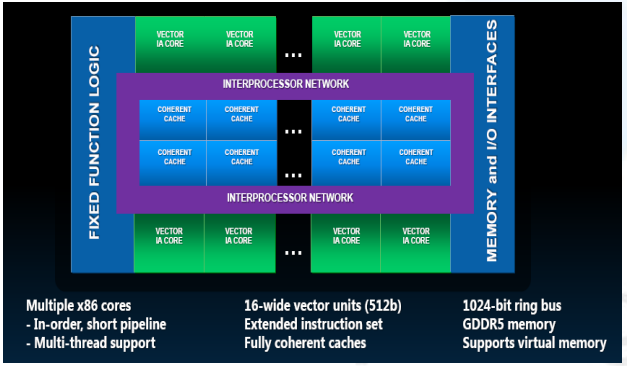
MIC

- Many-core cooperator based on **X86 arch** **(** PCIE card)
- Support 4Thread/core
  - Adjust thread numbers
  - Memory access hided by computing
- Node-management way (Dedicated IP)
  - Not only Host, but also Device
- 512bit vector-bit width
  - Parallel between thread level and instruction level
- Embedded uos
  - Manage resource of card
  - energy consumption control

CPU+MIC： X86 architecture-based heterogeneous

- Many-core X86+Multi-core X86
- **One Code**





MIC programming technology innovation

Convenience in programming

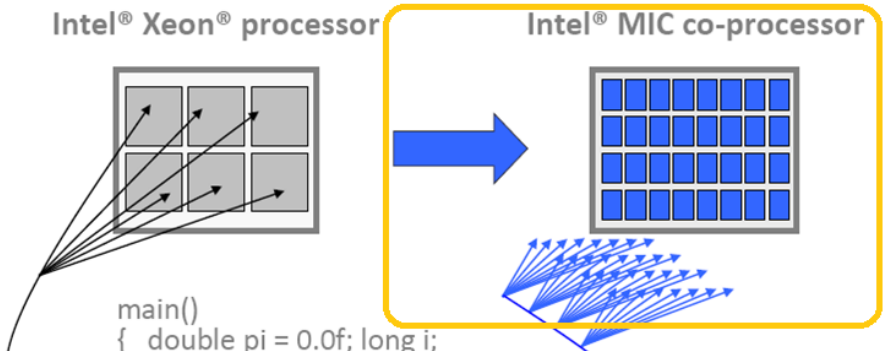
Effective reuse

Programming environment compatibility

Convenience in programming: prophase rapid transplantation

MIC transplantation：Offload mode

- quotation，similar with OpenMP，not need to overwrite the code again
- continue the original programming mode of OpenMP、MPI、OpenCL




```
main()
{  double pi = 0.0f; long i;
   #pragma offload target (mic)
   #pragma omp parallel for reduction(+:pi)
   for (i=0; i<N; i++)
   {
      double t = (double)((i+0.5)/N); pi += 4.0/(1.0+t*t);
   }
   printf("pi = %f\n",pi/N);    }
```

The case of rapid application development

```
SIRT Algorithm
main()
{
      Input...
#pragma offload target in(...) out(...)
{
      Back_project_init();
      for  0~N
      {
            Reproject();
            Backproject();
      }
}
      Output...
}
```

```
Back_project_init()
{
        ......
#pragma omp parallel for
        for i
                for j
                        ......
}
```

```
Reproject()
{
        ......
#pragma omp parallel for
        for i
                for j
                        ......
}
```

```
Backproject()
{
        ......
#pragma omp parallel for
        for i
                for j
                        ......
}
```

	Programming Language	Total Lines
CPU	C (since 1978)	~2900
Intel MIC	OpenMP + Comments	~3000 (100 new)

Convenience in programming: Preliminary performance test of MIC

Using Native mode

Using Symmetric mode

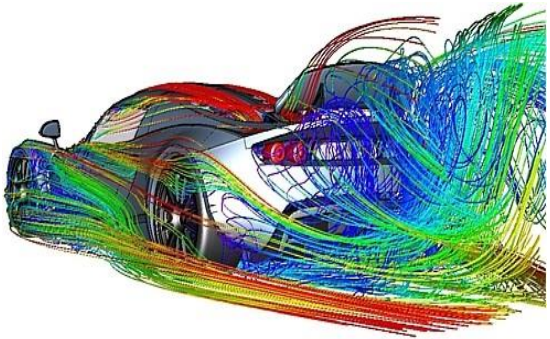
CPU program mode	Computing scale	MIC porogramming mode	Programming work	advantage/disadvantage
OpenMP/Pthread	Single node	Native+OpenMP/PThread	1.—mmic compiling 2.Need to transfer binary program、lib and data from CPU to MIC 3.OpenMP/Pthread thread setting in the MIC	1. relatively short development cycles 2. Logic processing ratio less, performance can be close to optimal performance ● Metocalo ● SIRT 3. feasibility analysis ● The memory space is limited, such as BWA ● More IO and logic processing part directly affect the performance, such as BWA 4、 simple compiling approaches
MPI	Cluster	Symmetric +MPI	1.--mmic compiling 2.Need to transfer binary program、lib and data from CPU to MIC 3.MPI process setting in the MIC	1. relatively short development cycles 2. The performance is difficult to approach the optimal performance 3. feasibility analysis ● The memory space is limited ● Too much MPI process communication too much 5、 complex compiling, may fail
MPI+OpenMP/PThread	cluster	Symmetric+MPI+OpenMP/PThread	1.—mmic compiling 2.Need to transfer binary program、lib and data from CPU to MIC 3.OpenMP thread setting in the MIC	1. relatively short development cycles 2. performance can be close to optimal performance 3、 simple compiling approaches

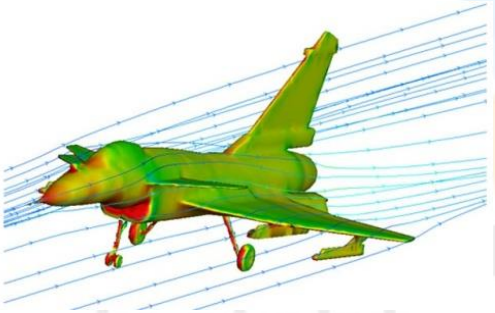
Accomplished code reused

Code reused under the same instruction arch, CPU and MIC combined perfectly.

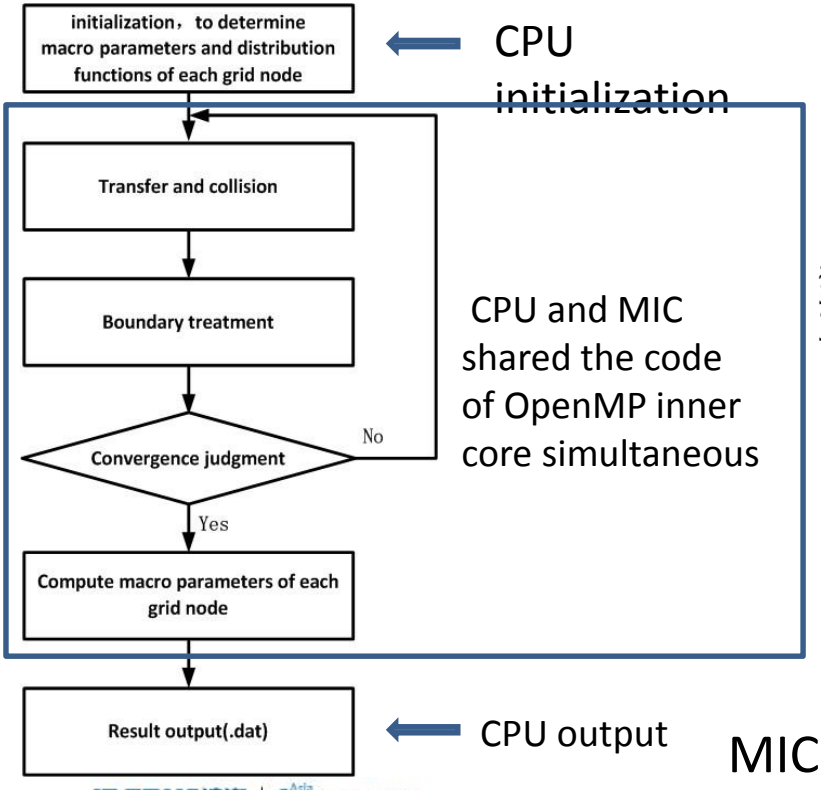
Application case：Inspur cooperated and developed LES（ Large Eddy Simulation ）algorithm with NPC on MIC platform.

Lattice Boltzmann Method can simulate Large Eddy Simulation, this method is the key algorithm of LES，the performance of LES can be significantly increased with MIC acceleration.





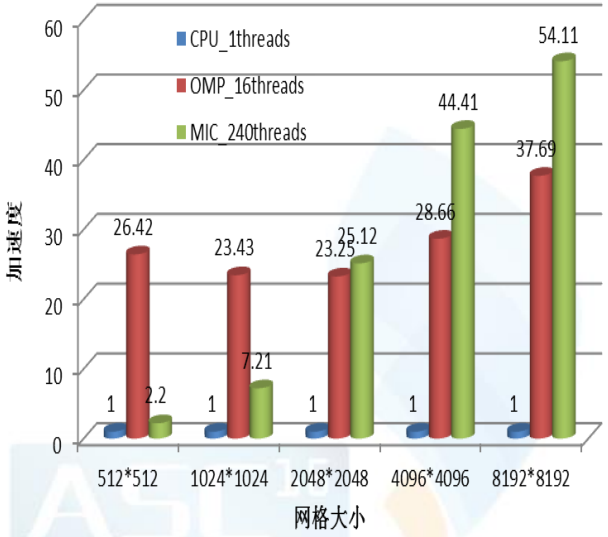
Accomplished code reused



CPU initialization

CPU and MIC shared the code of OpenMP inner core simultaneous

CPU output



MIC = ***1.43****2E52680CPU

Programming environment affinity

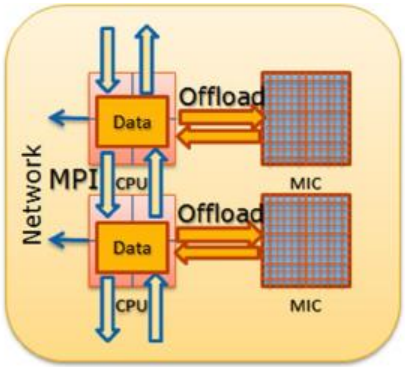
Supporting traditional HPC development language:C/C++/Fortran

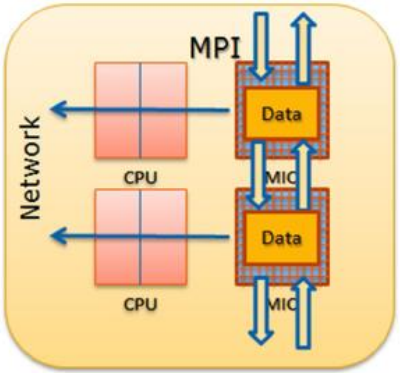
Intel tool chain、lib supporting MIC

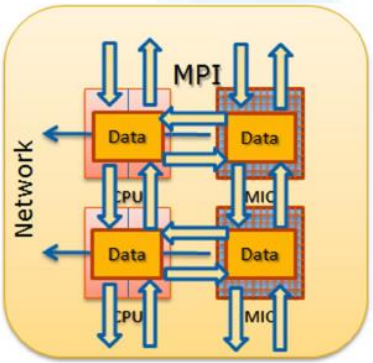
- Compiler icc\icpc\ifort supporting MIC
- Debugging, analysis tool (vtune\MPITrace)
- MKL

Supporting traditional programming module

- OpenMP/Pthread/OpenCL/Clik
- MPI
  - Offload mode
  - Native mode
  - Symmetric mode







MIC programming environment and efficiency analysis

	Intel MIC
Programming language	C/C++/Fortran
Programming module	OpenMP/Pthread/OpenCL/Clik/MPI
compiler	icc/icpc/ifort
Degree of difficulty in coding	Similar with OpenMP directive， convenient in programming
Resue in CPU coding	√
code maintenance, upgrading	easy， setting different thread numbers

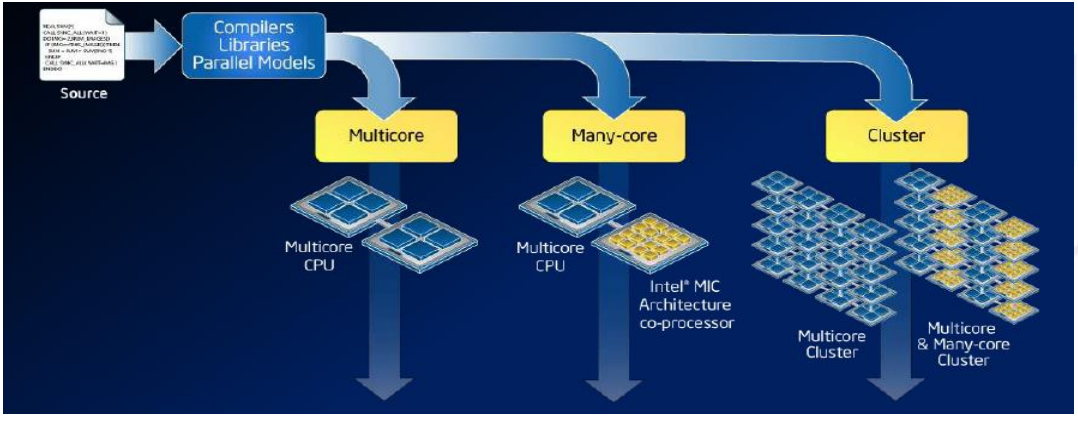
Extension of MIC computing

One code, run everywhere

- CPU extend to MIC
- MIC extend to MIC node (nCPUs+nMICs)
- MIC node extend to MIC cluster

- Can accomplish extension of thread/process level
  - CPU Threads + MIC Threads
  - CPU MPI process+ MIC MPI process

Computing on CPU and MIC simultaneous，make full use of resources usage of cores



Actual user’s MIC application performance

Customer	Application	Performance Increase[1] vs. 2-Socket Intel® Xeon® Processor
Los Alamos	Molecular Dynamics	Up to 2.52x
Acceleware	8th-order isotropic variable velocity	Up to 2.05x
Jefferson Lab	Lattice QCD	Up to 2.27x
Financial Services	Black-Scholes SP Monte Carlo SP	Up to 7x[4] Up to 10.75x[4]
Sinopec	Seismic Imaging	Up to 2.53x[2]
Sandia Labs	miniFE (finite element solver)	Up to 2x[3,4]
Intel Labs	Ray Tracing (incoherent rays)	Up to 1.88x[4]

Inspur-Intel joint Lab

- Up to 2-5x

Intel’s performance of official propaganda

- Up to 1.75-10.75x

Notes:

1. 2-socket Intel® Xeon® processor vs. one Intel® Xeon Phi™ coprocessor (preproduction HW/SW & Application running 100% on coprocessor unless otherwise noted)
2. 2-socket Intel Xeon processor vs. 2-socket Intel Xeon processor + 2 Intel Xeon Phi coprocessor (offload)
3. 8-node cluster, each node with 2-socket Intel Xeon processor (comparison is cluster performance with and without one Intel Xeon Phi coprocessor per node) (Hetero)
4. Intel Measured Oct. 2012

MIC application adaptability -parallel characteristic

	Intel MIC
Intensive computing, highly parallel computing	√
Supporting data paralleling	√
Supporting task paralleling	√
Supporting thread paralleling	√
Supporting process paralleling	√
granularity of parallelism	coarse or fine

MIC application adaptability application characteristic

Good scalability

Intensive computing, high F/B

Good vecterization

Low memory bandwidth

Low memory capacity

MIC technology summary

MIC technology aims

- According to established industry standards, programming habit and programming environment, the development of HPC will be improved

MIC technology advantage

- Solve the bottle of HPC application performance
  - High parallel workload
- Solve HPC application programming, software ecological environmental problems
  - Programming module
  - Programming language
  - compiler
  - Programming tool
  - Programming lib
- Solve the problem of using maximize resources

PERFORMANCE OPTIMIZATION ON MIC

Outline

MIC performance optimization strategy

MIC optimization methods

summary

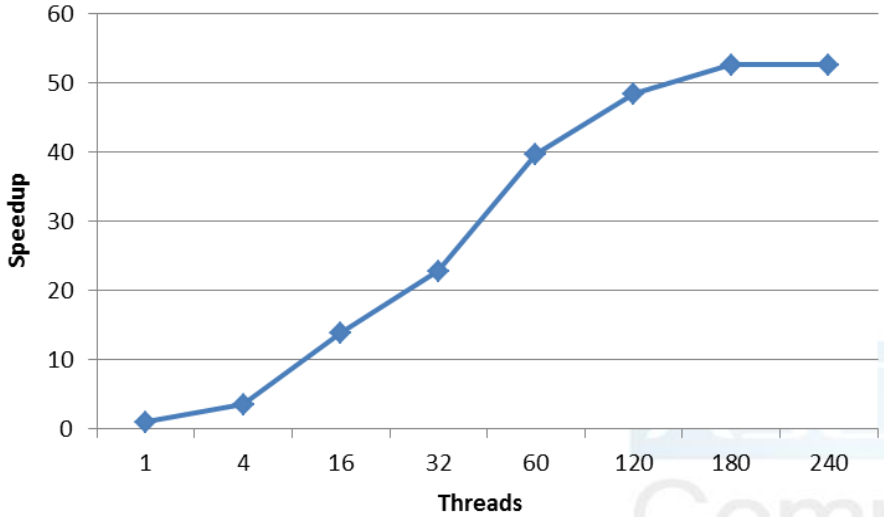
MIC performance optimization strategy

MIC Optimization Methods

- **Parallelism Optimization**
- **Memory Management Optimization**
- **Data Transfer Optimization**
- **Memory Access Optimization**
- **Vectorization Optimization**
- **SIMD optimization**
- **Load Balance Optimization**
- **Extensibility of MIC Threads Optimization**

Parallelism Optimization

guarantee enough parallelism



Parallelism Optimization : Parallel Granularity

The principle of coarse-grained parallelism

- optimize the codes at the top level
- decreases thread management and thread resources

Example of Parallelism Optimization

```
for (i=0; i<M; i++)
{
    for (j=0; j<N; j++)
    {
        …
    }
}
```

```
#pragma omp parallel for
num_threads(THREAD_NUM)
for (i=0; i<M; i++){
    for (j=0; j<N; j++)
    {     …    }
}
```

```
#pragma omp parallel
num_threads(THREAD_NUM)
for (i=0; i<M; i++){
#pragma omp for
    for (j=0; j<N; j++)
    {     …    }
}
```

Example :nested parallelism

```
for (i=0; i<M; i++)
{
   for (j=0; j<N; j++)
   {
      …
   }
}
```

```
#pragma omp parallel for
num_threads(THREAD_NUM)
for (k=0; k<M*N; k++){
     i = k/M;
     j = k%M;
}
```

```
omp_set_nested(true);
#pragma omp parallel for
num_threads(THREAD_NUM1)
for (i=0; i<M; i++){
#pragma omp parallel for
num_threads(THREAD_NUM2)
   for (j=0; j<N; j++)   {          …   }
}
```

M is too small，not satisfy the MIC parallelism

MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

Memory Management Optimization

Size of MIC memory:6GB~16GB

Task partition

- Change parallelism level
  - Parallel inner loop
  - Parallelism based on task->Parallelism based on data
  - Reduce allocation times

MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- **Data Transfer Optimization**
- Memory Access Optimization
- Vectorization Optimization
- SIMD optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

Data Transfer Optimization : Nocopy

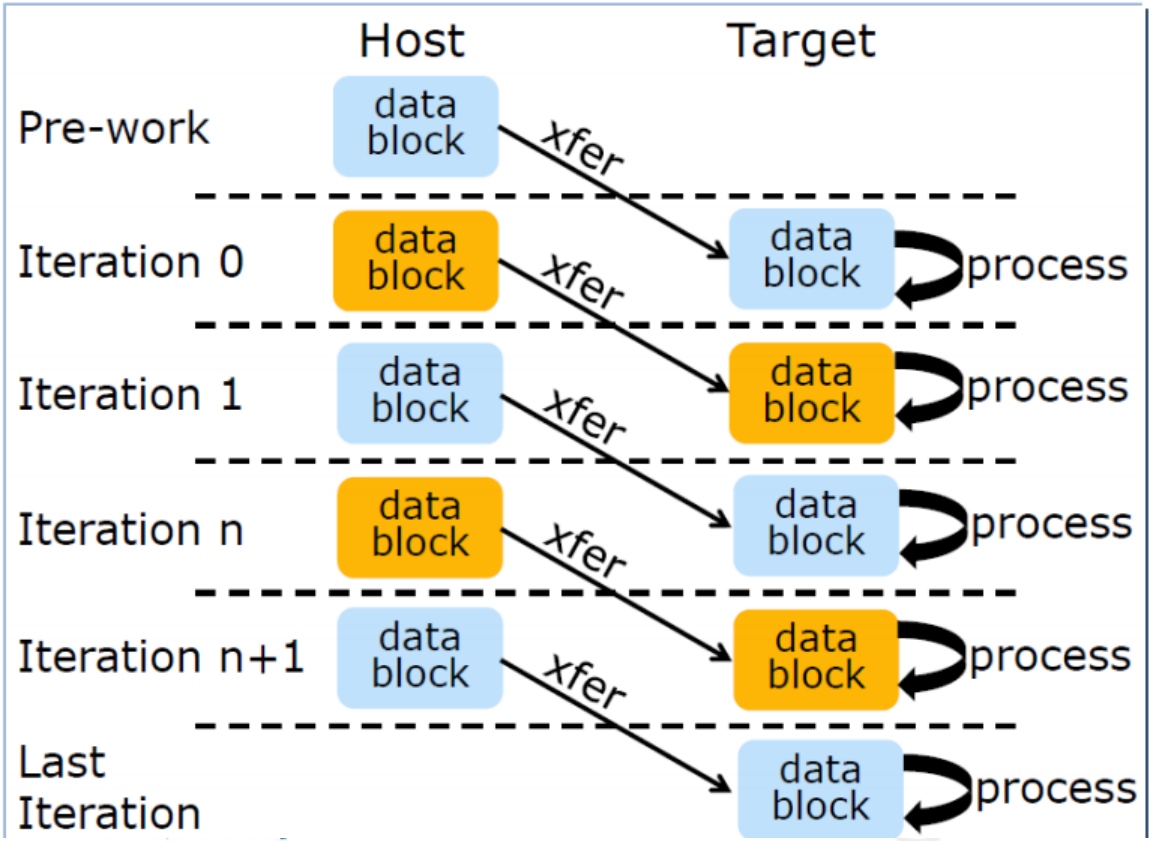
```
p_c = …; // the value p_c doesn’ t change in each iteration
for(i=0; i<steps; i++)//many iterations
 {
    p_in = …; //the value p_in changes in each iteration
#pragma offload target(mic) \
     in(p_in:length(…))  in(p_c: length(…))  out(p_out: length(…))
    {
                    kernel(p_in, p_c, p_out);
    }
}
… = p_out; // p_out is used only when the iterations on CPU are finished
```

Data Transfer Optimization : Nocopy

```
p_c = ...; // the value p_c doesn't change in each iteration
#pragma offload target(mic) \
in(p_c: length(...) alloc_if(1) free_if(0)) \
nocopy(p_in:length(...)alloc_if(1) free_if(0)) \
nocopy(p_out: length(...) alloc_if(1) free_if(0))
{
} //allocate memory without release-pass the value of p_c
for(i=0; i<steps; i++)//many iterations
{
    p_in = ...;
#pragma offload target(mic) \
     in(p_in:length(...) alloc_if(0) free_if(0) ) \
     nocopy(p_c) nocopy(p_out)
     {   kernel(p_in, p_c, p_out); }
}
```

```
#pragma offload target(mic) \
nocopy (p_c: length(...) alloc_if(0) free_if(1)) \
nocopy(p_in:length(...)alloc_if(0) free_if(1)) \
out(p_out: length(...) alloc_if(0) free_if(1))
{
}
... = p_out;
```

Data Transfer Optimization : Asynchronous Transfer



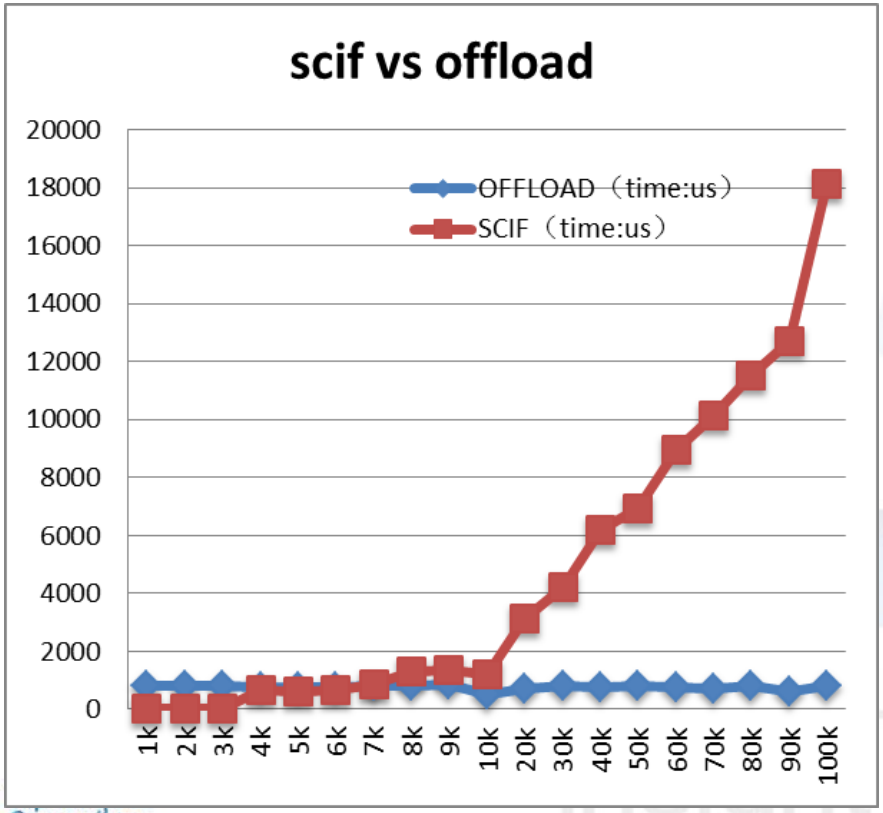
Data Transfer Optimization ： example of asynchronous transfer

```
#pragma offload_transfer target(mic:0) in(in1 : length(count) alloc_if(0) free_if(0) ) signal(in1)
for (i=0; i<iter; i++)
{
      if (i%2 == 0) {
 #pragma offload_transfer target(mic:0) if(i!=iter-1) in(in2 : length(count) alloc_if(0) free_if(0) ) signal(in2)
 #pragma offload target(mic:0) nocopy(in1) wait(in1) out(out1 : length(count) alloc_if(0) free_if(0) )
                  compute(in1, out1);
       } else {
 #pragma offload_transfer target(mic:0) if(i!=iter-1) in(in1 : length(count) alloc_if(0) free_if(0) ) signal(in1)
 #pragma offload target(mic:0) nocopy(in2) wait(in2) out(out2 : length(count) alloc_if(0) free_if(0) )
                  compute(in2, out2);
      }
}
```

Data Transfer Optimization：example of asynchronous transfer

```
int counter;
float *in1;
counter = 10000;
__attributes__((target(mic))) mic_compute;
while(counter>0)
{
#pragma offload target(mic:0) signal(in1)
   {
      mic_compute();
   }
   cpu_compute() //此时本函数与上面的MIC函数并行执行
#pragma offload_wait target(mic:0) wait(in)
   counter--;
}
```

Data Transfer Optimization ： SCIF



MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- SIMD optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

Memory Access Optimization

MIC Hierarchical Memory

The KNC card has eight dual-channel GDDR5 memory controllers with a bandwidth of 5.5 GT/s。

Two levels of cache: L1 and L2。

- Each core of KNC has a 32KB L1 instruction cache and a 32KB L1 data cache，The L1 cache line is 64B, using 8 channels and 8 banks，.. The L1 cache belongs to each core privately, with fast accessing speed。
- KNC has shared L2 Cache，31MB L2 cache is available for memory space

Memory Access Optimization Strategy on MIC

- **Hiding memory latency**
  - Multi-threading
  - Prefetchin
- **Using Cache**
  - Temporal Locality
  - Spatial Locality
- Alignment

Cache Optimization

Code transform

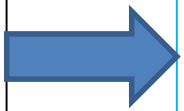
- Loop fusion
- Loop fission
- Loop tiling
- Loop exchange

Data transform

- Data location
- Array reorganization

Loop fusion

```
//original loop:
for(i=0;i<n;i++)
    a[i]=b[i]+1;
for(i=0;i<n;i++)
    c[i]=a[i]/2;
```



```
//Fused loop
for(i=0;i<n;i++)
{
    a[i]=b[i]+1;
    c[i]=a[i]/2;
}
```

Loop fission

```
//original loop:
for(i=1;i<n;i++)
{
    a[i]=a[i]+b[i-1];
    b[i]=c[i-1]*x*y;
    c[i]=1/b[i];
     d[i]=sqrt(c[i]);
}
```

```
//splitted loop:
for(i=1;i<n;i++)
{
    b[i]=c[i-1]*x*y;
    c[i]=1/b[i];
}
for(i=1;i<n;i++)
    a[i]=a[i]+b[i-1];
for(i=1;i<n;i++)
    d[i]=sqrt(c[i]);
```

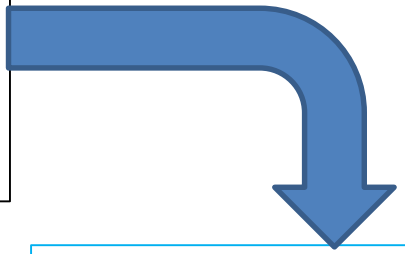
Loop tiling

```
//original loop:
for(i=0;i<n;i++)
   for(j=0;j<m;j++)
       x[i][j]=y[i]+z[j];
```

```
//tiled loop:
for(it=0;it<n;it+=nb)
    for(jt=0;jt<m;jt+=mb)
        for(i=it;i<min(it+nb,n);i++)
            for(j=jt;jt<min(jt+mb,m);j++)
                x[i][j]=y[i]+z[j];
```

Loop exchange

```
//original loop:
for(j=0; j<m; j++)
    for(i=0; i<n; i++)
        c[i][j] = a[i][j] + b[j][i];
```



```
//interchanged loop:
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
      c[i][j] = a[i][j] + b[j][i];
```

Alignment

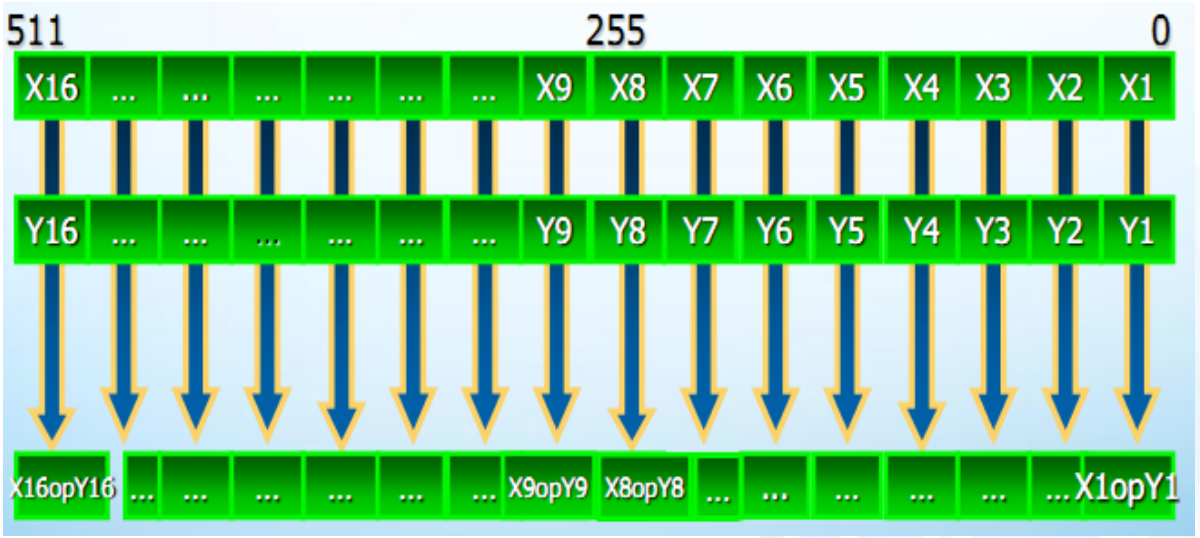
	Windows	Linux
static space （stack）	__declspec(align(N))	__attribute__((aligned(n)))
dynamic space （heap）	_aligned_malloc	_mm_malloc
	_aligned_free	_mm_free

- For example：
  - float A[1000] __attribute__((aligned(64)));
  - float *B= (float*)_mm_malloc(N*sizeof(float), 64);

MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- SIMD optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

Vectorization Optimization



MIC Vectorization Optimization Strategy

Insert quotation automatic vectorization

- without changing the original program structure, automatically vectorizing by inserting a precompiled instruction (quotation).

Adjust program loop and insert quotation automatic vectorization

Write SIMD instruction

Automatic Vectorization : advantage

Improving performance

- vectorization processing can process multiple batches of data simultaneously within a single instruction cycle.

Simplified encoding by writing a single version of code and reducing assembling codes

- less assembling coding reduces reprogramming for a special system, and the program is easily updated for the mainstream system without rewriting the assembling code.

What kind of the loop can be vectorized?

- each statement as being independent or without dependence on the loop ;
- on the innermost loop
- The type of vectorization data should be consistent

This loop can be vectorized

```
for (int i=1; i<1000; i++)
{
        a[i] = a[i-1] * b[i];
}
```

```
for (int i=0; i<1000; i++)
{
        a[i] = b[i] * T + d[i] ;
        b[i] = (a[i] + b[i])/2;
        c = c + b[i];
}
```

```
for (int i=0; i<1000; i++) a[i] = b[i] * T + d[i] ;
for (int i=0; i<1000; i++) b[i] = (a[i] + b[i])/2;
for (int i=0; i<1000; i++) c = c + b[i];
```

automatic vectorization method

a) Compiler vectorization option

-vec (the default vectorization compile options) -no-vec

-vec-report

-vec-report[n]	functionality
n=0	without displaying diagnosed result
n=1	only displaying vectorized loops (default)
n=2	displaying the vectorized and non-vectotorized loops
n=3	displaying the vectorized and non-vectotorized loops, and also the dependent information
n=4	only displaying non-vectorized loops
n=5	displaying the non-vectorized loops, and also the dependent information

automatic vectorization method

The usage of #pragma ivdep and restrict

- #pragma ivdep / vector always / simd
- #pragma vector aligned

```
void foo(int k)
{
#pragma ivdep
        for(int j=0; j<1000; j++)
        {
                a[j] = b[j+k] * c[j];
                b[j] = (a[j] + b[j])/2;
                b[j] = b[j] * c[j];
        }
}
```

```
void foo(float*restrict a, float*restrict b, float*restrict c)
{
        for(int j=0; j<1000; j++)
        {
                a[j] = b[j] * c[j];
                b[j] = (a[j] + b[j])/2;
                b[j] = b[j] * c[j];
        }
```

use the keyword
-restrict

automatic vectorization：optimization method

a) Reorder the nested loops

```
for(j=0; j<N; j++)
#pragma ivdep
for( i=0; i<M; i++)
{
        C[i][j] = A[i][j]+B[i][j];
}
```

```
for( i=0; i<M; i++)
#pragma ivdep
for(j=0; j<N; j++)
{
        C[i][j] = A[i][j]+B[i][j];
}
```

automatic vectorization：optimization method

b) Split the loop

```
for(i=0; i<N; i++)
{
        rand();
        …
        …
}
```

```
for(i=0; i<N; i++)
{
        rand();
}
for(i=0; i<N; i++) //自动向量化
{
        …
        …
}
```

```
float s;
for(i=0; i<N; i++)
{
        …
        s=…;
        for(j=0; j<M; j++)//自动向量化
        {
                if(s>0)
                {
                        …
                }
        }
}
```

```
float s[16];
for(i=0; i<N; i+=16)
{
        T=min(N-1,16);
        for(k=0; k<T; k++) //自动向量化
        {
                …
                s[k]=…;
        }
        for(k=0; k<T; k++)
        {
                for(j=0; j<M; j++) //自动向量化
                {
                        if(s[k]>0)
                        {
                                …
                        }
                }
        }
}
```

automatic vectorization : optimization method

c) Parallelism and Vectorization

Parallelism

Vectorization

```
for(i=0;i<100;i++)
{
        for(j=0;j<1024;j++)
        {
                …
        }
}
```

Low Parallelism

Parallelism

Vectorization

```
for(i1=0;i1<200;i1++)
{
        for(j1=0; j1<512; j1++)
        {
                i = i1/2;
                j = (i1%2)*512+j1;
                …
        }
}
```

MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- SIMD optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

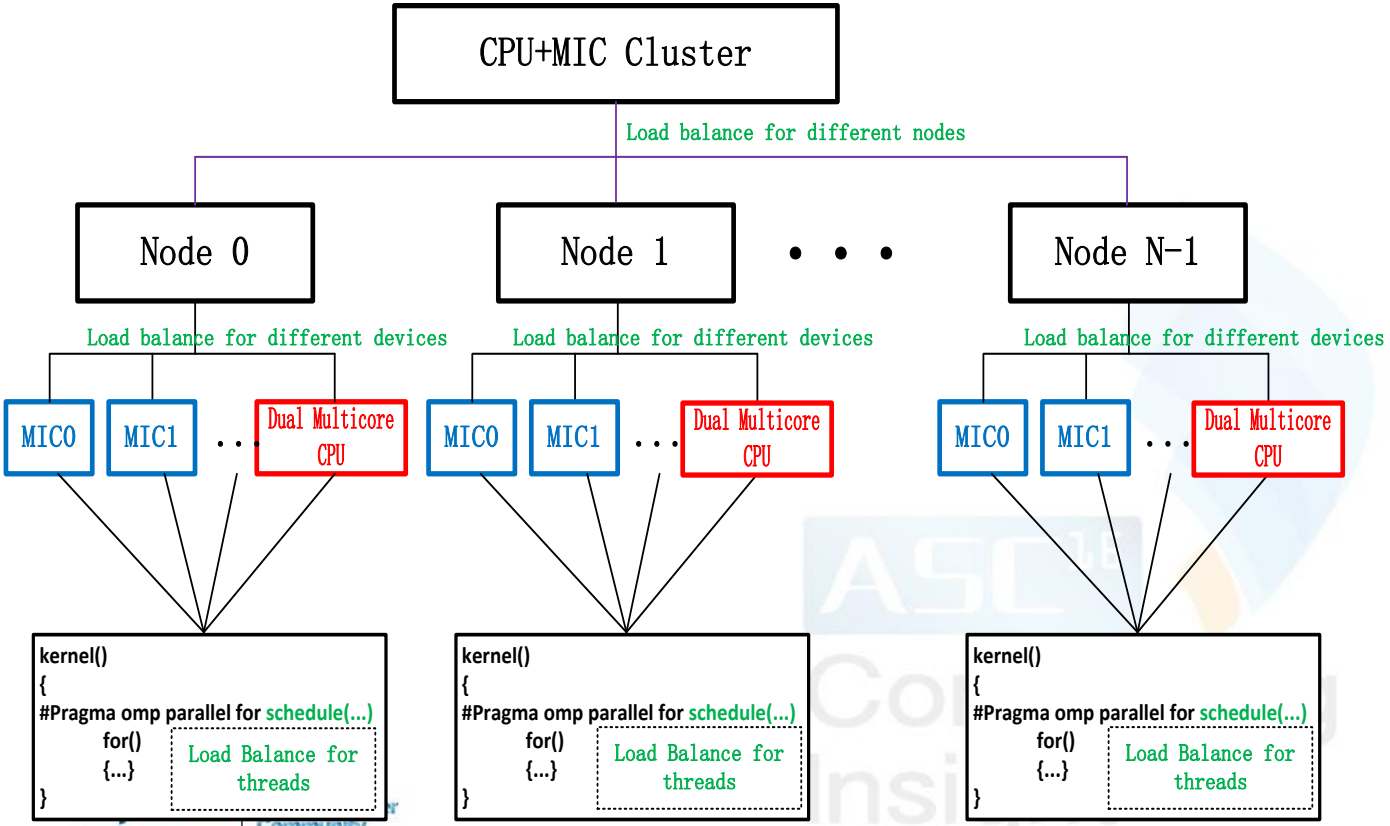
SIMD optimization

```
#include <immintrin.h>
void foo(float *A, float *B, float *C, int N)
{
#ifdef __MIC__
        __M512 _A,_B,_C;
        for(int i=0; i<N; i+=16)
        {
                _A =_mm512_loadunpacklo_ps (_A, (void*)(&A[i]) );
                _A = _mm512_loadunpackhi_ps (_A, (void*)(&A[i +16]) );
                _B = _mm512_loadunpacklo_ps (_B, (void*)(&B[i]) );
                _B = _mm512_loadunpackhi_ps (_B, (void*)(&B[i +16]) );
                _C = _mm512_add_ps(_A,_B);
                _mm512_packstorelo_ps ((void*)(&C[i]) , _C );
                 _mm512_packstorehi_ps ((void*)(&C[i +16]), _C );
        }
#endif
}
```

MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- SIMD optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

Load Balance Optimization



Load balance on computing device

Scheduling strategies	Available scopes
Static	fixed amount of computation, with the fixed iteration workload too
Dynamic	non fixed amount of computation, without the fixed iteration workload neither
Guided	special dynamic scheduling strategy, the guided algorithm can reduce the workload

Load balance between CPU/MIC

Task schedule

dynamic load balance

Data schedule

static load balance

MIC Optimization Methods

- Parallelism Optimization
- Memory Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- SIMD optimization
- Load Balance Optimization
- Extensibility of MIC Threads Optimization

Thread affinity optimization

export KMP_AFFINITY=

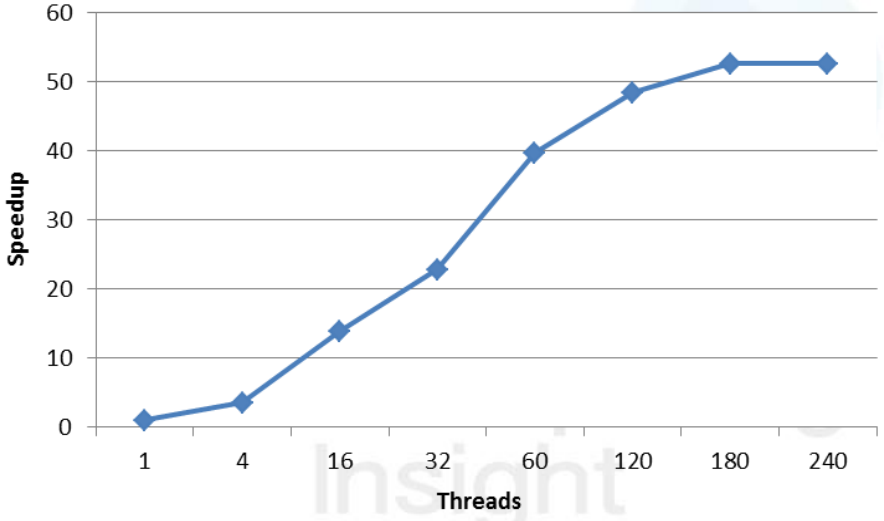
- Scatter
- Compact
- balanced（ unique for MIC ）

How to improve the extension for many threads

Algorithm

Parallelism

Hardware bottleneck



Summary

- Parallelism Optimization
- Ram Management Optimization
- Data Transfer Optimization
- Memory Access Optimization
- Vectorization Optimization
- SIMD Optimization
- Load Balance Optimization
- Extension of MIC Threads Optimization

DEVELOPMENT PROCESS OF HPC APPLICATION BASED ON MIC

MIC development process

Analyze original CPU program

- Hotspot test
- Parallelism analysis 、 vectorization analysis、 MIC Memory analysis

MIC development

- OpenMP parallelism Based on CPU
- Thread Extension Based on CPU platform
- Coordination Parallelism Based on Single Node CPU+MIC Mode
- MIC cluster parallel

MIC optimization

- Vectorization optimization
- MIC memory optimization（L1、L2 Cache optimization）
- IO optimization

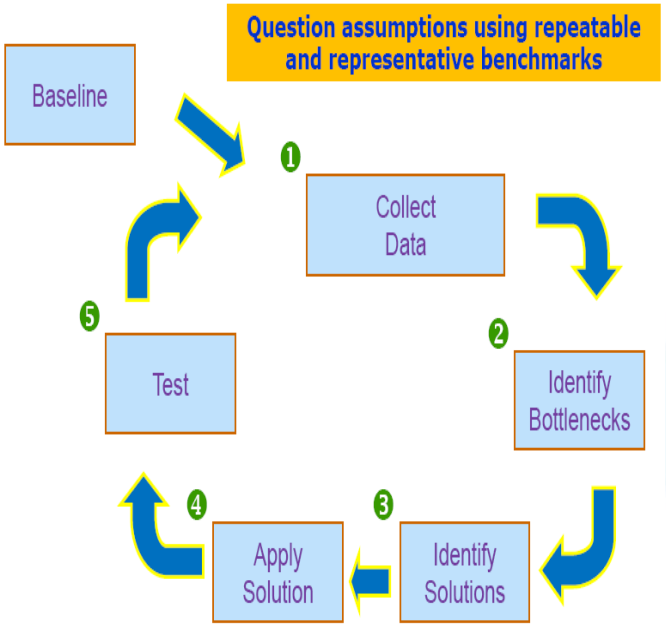
MIC optimization methods

take full advantage of the hardware

- Many cores，many parallel threads
- 512bit vector-bit width VPU
- L1 /L2 cache

MIC optimization process and methods

- Parallelism Optimization
- Ram Management Optimization
- **Vectorization Optimization**
- Extension of MIC Threads Optimization
- Load Balance Optimization
- Memory Access Optimization(L1,L2)
- Data Transfer Optimization



Serial codes of Matrix Multiplication

- C=A*B
  - A is an M*K matrix
  - B is a K*N matrix
  - C is a M*N matrix

```
int main(void)
{
 //allocate and initialize the
   matrix A, B and C and some varibles
 // read the matrix A and B

  kernel computing

 //export matrix C
 //release matrix A, B and C
   …
 return 0;

}
```

P_baseline

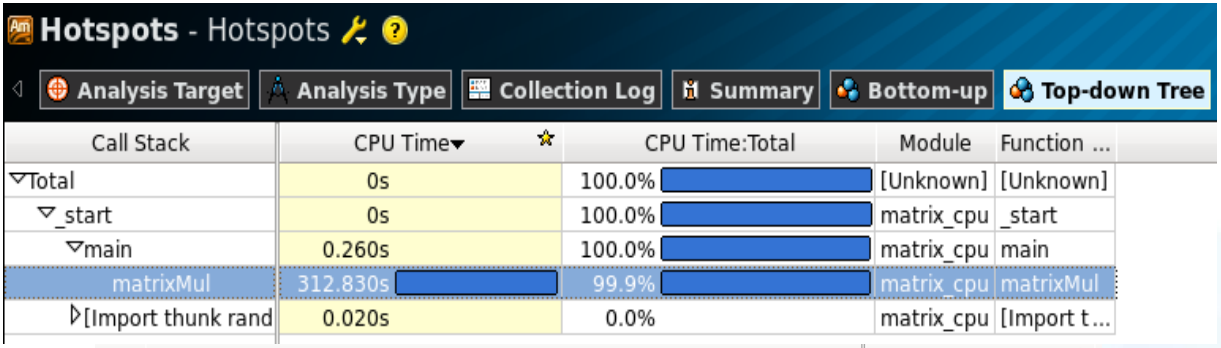
- Test case
  - the size of matrix A, B and C is all 4096*4096.
  - Intel Xeon5675 3.07GHz CPU

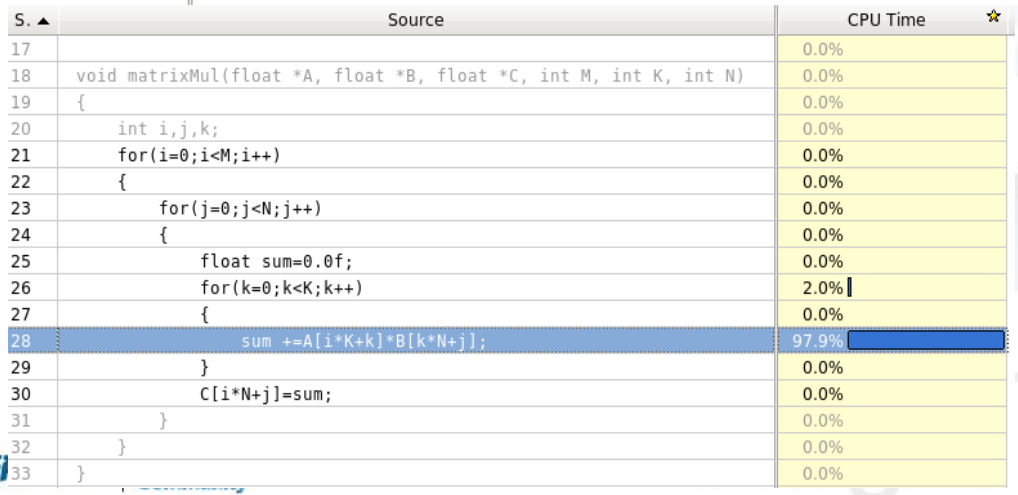
Excute time

- 312.83s

```
1   for(i=0;i<M;i++)
2   {
3       for(j=0;j<N;j++)
4       {
5           float sum=0.0f;
6           for(k=0;k<K;k++)
7           {
8               sum +=A[i*K +k]*B[k*N +j];
9           }
10          C[i*N +j]=sum;
11      }
12  }
```

Vtune analysis result





P_OMP

Platform

- Intel two-way 6 cores Xeon5675 3.07 GHz CPU

Number of threads

- 24

Execution time

- 170.83s

speedup

- 1.83

```
1   #pragma omp parallel for private(j,k) num_threads(THREAD_NUM)
2    for(i=0;i<M;i++)
3    {
4           for(j=0;j<N;j++)
5           {
6                 float sum=0.0f;
7                 #pragma ivdep
8                 for(k=0;k<K;k++)
9                 {
10                      sum +=A[i*K +k]*B[k*widthN
11                }
12                C[i*N +j]=sum;
13          }
14  }
```

OpenMP

Vector

P_OMP

MIC optimization process

1. **Basic version**
2. **Vectorization optimization**
3. **Advanced optimization**
   a) Cache optimization (Block Matrix Multiplication)
   b) optimization (transfer Asynchronously
   c) CPU+MIC Coordinate Computing

MIC Base version (P_MIC_base)

Platform

- KNC，60cores，1.0GHz

Number of threads

- 240

time

- 174s

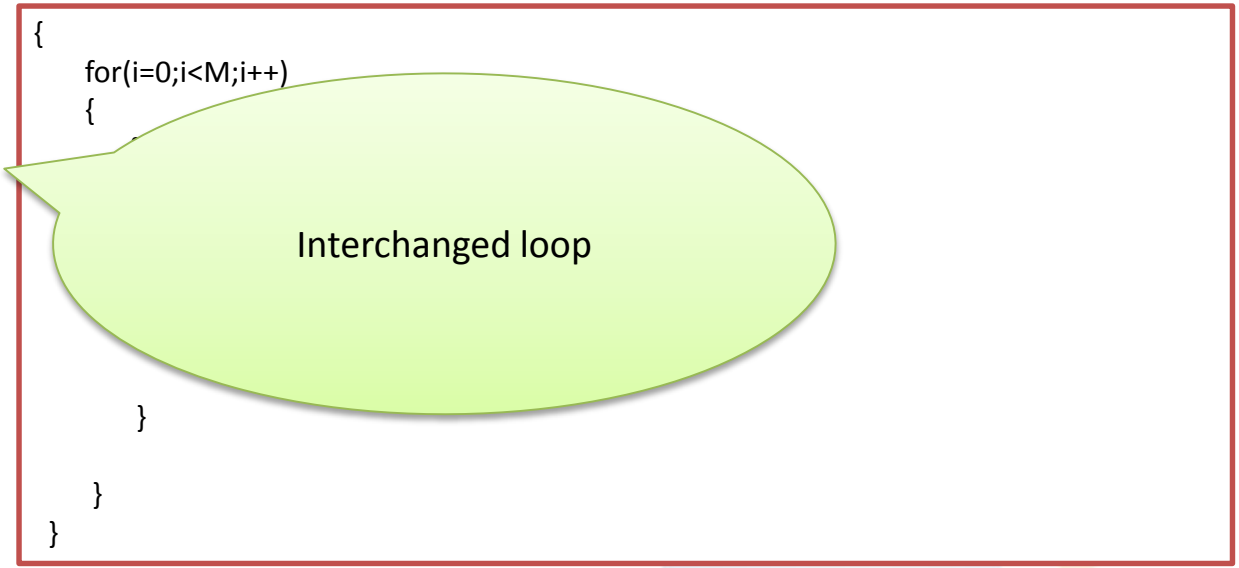
- Speedup（ relative to P_baseline ）
  - 1.80

```
1   #pragma offload target(mic) \
2   in(i, j, k, M,K,N) \
3   in(A: length(M * K)) \
4   in(B: length(K * N)) \
5   out(C: length(M * N))
6   {
7       #pragma omp parallel for private(j, k) num_threads(THREAD_NUM)
8       for(i=0; i<M; i++)
9       {
10          for(j=0; j<N; j++)
11          {
12              float sum=0.0f;
13              #pragma ivdep
14              for(k=0; k<K; k++)
15              {
16                  sum +=A[i*K +k]*B[k*N +j];
17              }
18              C[i*N +j]=sum;
19          }
20      }
21  }
```

MIC offload

Vectorization optimization

1



The serial version after exchaging loop

- Serial （no-vec） execution time is 53.07s

What is vectorization

Intel compiler supports vectorization, which uses vector processor unit to do the batch computation. Loop section can be vectorized by extended instruction set such as MMX, SSE, SSE2, SSE3, SSSE3, AVX, Knights Corner Instructions, etc, then the calculation speed are greatly improved.

Vectorization optimization

2

```
{
    for(i=0;i<M;i++)
    {
        for(k=0;k<K;k++)
       {
                    float temp = A[i*K +k];
                    #pragma ivdep
                     for(j=0;j<N;j++)
                {
                            C[i*N +j] += temp*B[k*N +j];
                     }
         }

     }
 }
```

vector

The serial version after exchaging loop and vectorization

- Serial （no-vec）execution time is 25.00s
- which runs 12.24 times faster than p_baseline

Vectorization optimization

3

```
{
    #pragma omp parallel for private(j,k) num_threads(THREAD_NUM)
    for(i=0;i<M;i++)
    {
        for(k=0;k<K;k++)
      {
                    float temp = A[i*K +k];
                    #pragma ivdep
                     for(j=0;j<N;j++)
                {
                            C[i*N +j] += temp*B[k*N +j];
                     }
         }

     }
 }
```

vector

The parallel version after exchaging loop and vectorization

- execution time is 4 .53s
- which runs 12.24 times faster than p_baseline

Vectorization optimization

4

```
#pragma offload target(mic)  in(i, j, k, M,K,N)  in(A: length(M * K)) in(B: length(K * N)) \
out(C: length(M * N))
{
    #pragma omp parallel for private(j,k) num_threads(THREAD_NUM)
    for(i=0;i<M;i++)
    {
        for(k=0;k<K;k++)
       {
                    float temp = A[i*K +k];
                    #pragma ivdep
                     for(j=0;j<N;j++)
               {
                              C[i*N +j] += temp*B[k*N +j];
                     }
         }

    }
}
```

The MIC version after exchaging loop and vectorization

- execution time is 3.43s
- which runs 7.92 times faster than P_baseline_vec

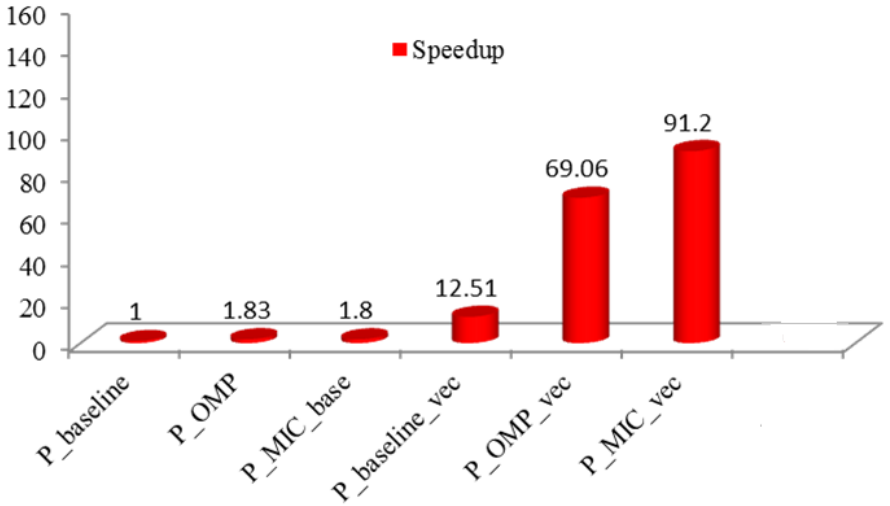
SIMD Instruction Optimization : P_MIC_simd

```
10          #ifdef __MIC__
11          __m512 _A,_B,_C;
12          for(k=0; k<K; k++)
13          {
14              _A = _mm512_set_1to16_ps(A[i*K +k]);
15              for(j=0; j<N/16; j+=16)
16              {
17
18
19
20
21
22
23
24              }
25          }
26          #endif
```

Execution time is 2.00s, this is marked as P_MIC_simd, which runs 12.5 times faster than P_baseline_vec, and 71.5% faster than P_MIC_vec re

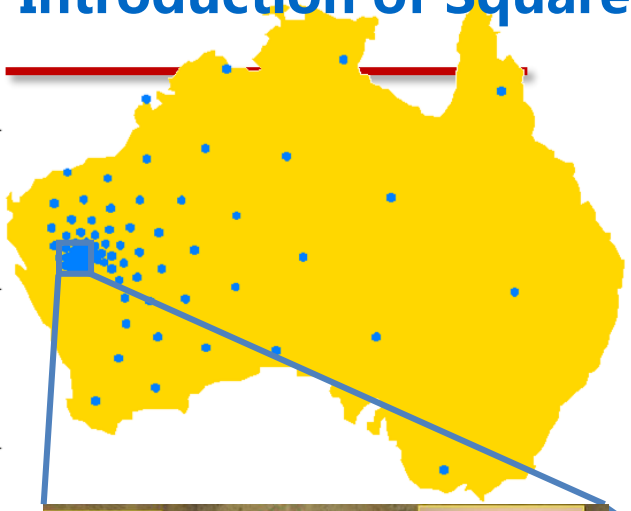
Summary

version	P_baseline	P_OMP	P_MIC_base	P_baseline_vec	P_OMP_vec	P_MIC_vec
time(s)	312.83	170.83	174	25	4.53	3.43

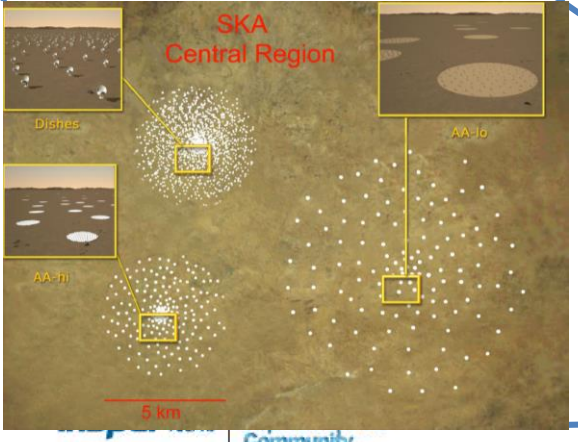


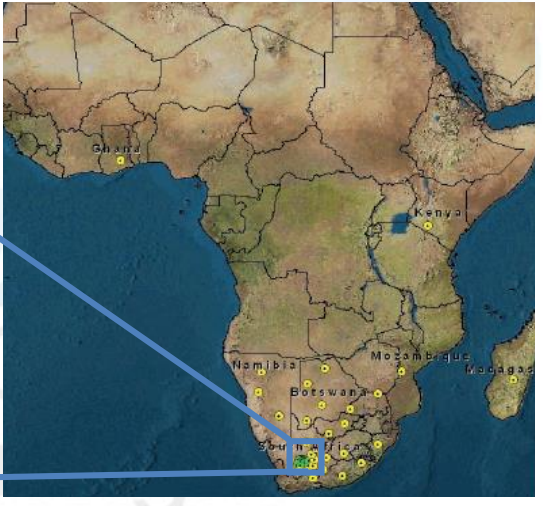
PERFORMANCE ACHIEVEMENTS OF THE GRIDDING ALGORITHM - A TYPICAL CASE STUDY IN THE SKA

Introduction of Square Kilometre Array(SKA)



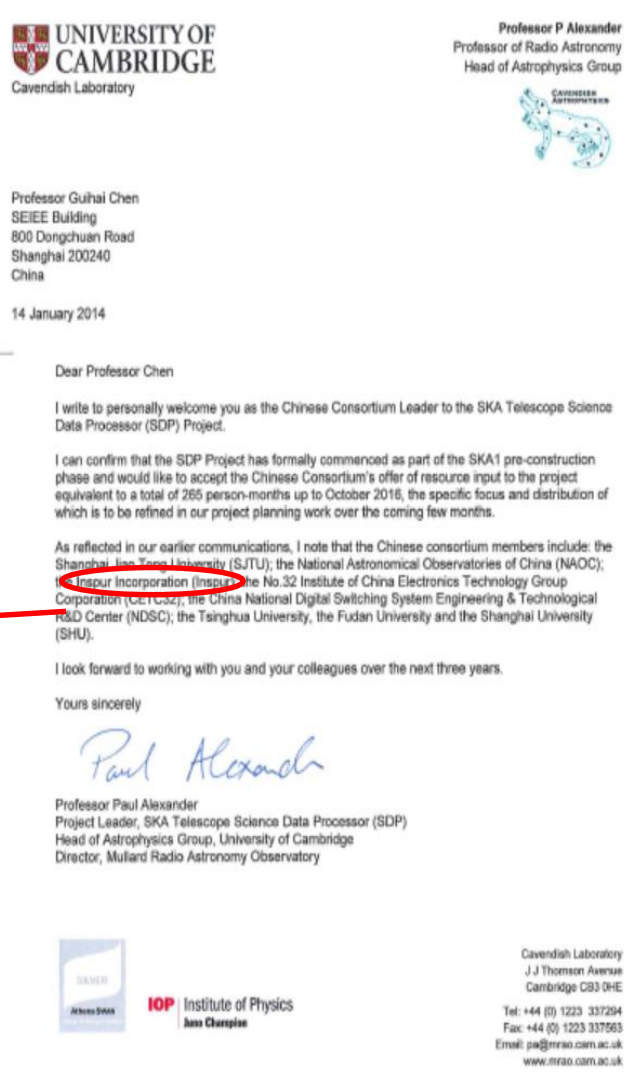
- A Continental sized Radio Telescope
- 11 Countries, 2013-2030
- 10 Work Packages, 11 Consortia
- HPC: 2.6ExaFlops, 5.5 MW





Inspur's role in SKA

- Science Data Processor (SDP)
  - Computing hardware platforms
  - Software
  - Algorithms in data processing
- Inspur
  - Formal member in International SDP Consortium
  - Main member and task leader in Chinese SDP Consortium



UNIVERSITY OF CAMBRIDGE
Cavendish Laboratory

Professor P Alexander
Professor of Radio Astronomy
Head of Astrophysics Group

Professor Guihai Chen
SEIEE Building
800 Dongchuan Road
Shanghai 200240
China

14 January 2014

Dear Professor Chen

I write to personally welcome you as the Chinese Consortium Leader to the SKA Telescope Science Data Processor (SDP) Project.

I can confirm that the SDP Project has formally commenced as part of the SKA1 pre-construction phase and would like to accept the Chinese Consortium's offer of resource input to the project equivalent to a total of 265 person-months up to October 2016, the specific focus and distribution of which is to be refined in our project planning work over the coming few months.

As reflected in our earlier communications, I note that the Chinese consortium members include: the Shanghai Jiao Tong University (SJTU); the National Astronomical Observatories of China (NAOC); the Inspur Incorporation (Inspur); the No.32 Institute of China Electronics Technology Group Corporation (CETC32); the China National Digital Switching System Engineering & Technological R&D Center (NDSC); the Tsinghua University, the Fudan University and the Shanghai University (SHU).

I look forward to working with you and your colleagues over the next three years.

Yours sincerely

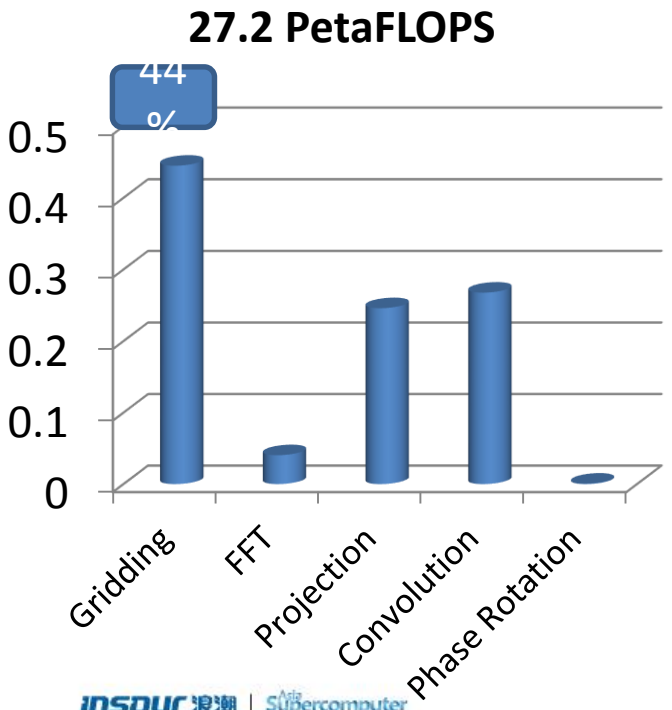
Paul Alexander

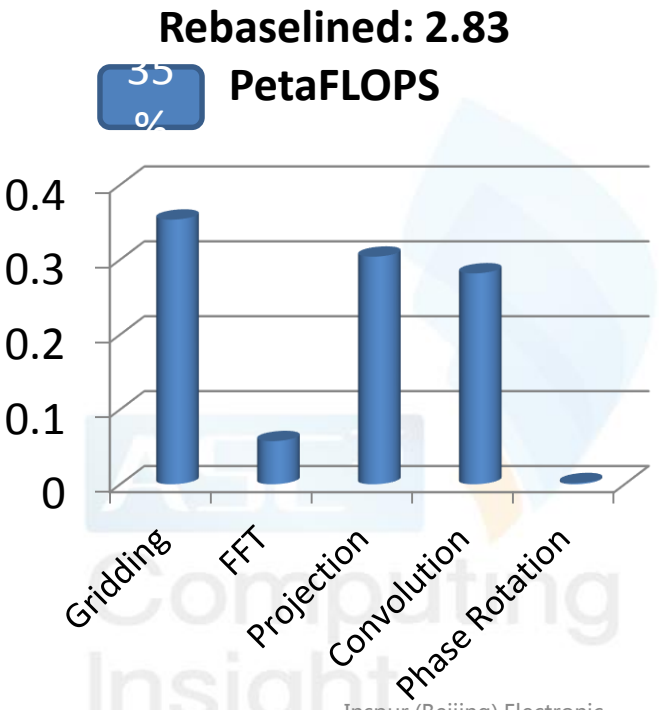
Professor Paul Alexander
Project Leader, SKA Telescope Science Data Processor (SDP)
Head of Astrophysics Group, University of Cambridge
Director, Mullard Radio Astronomy Observatory

Cavendish Laboratory
J J Thomson Avenue
Cambridge CB3 0HE
Tel: +44 (0) 1223 337254
Fax: +44 (0) 1223 337593
Email: pa@mrao.cam.ac.uk
www.mrao.cam.ac.uk

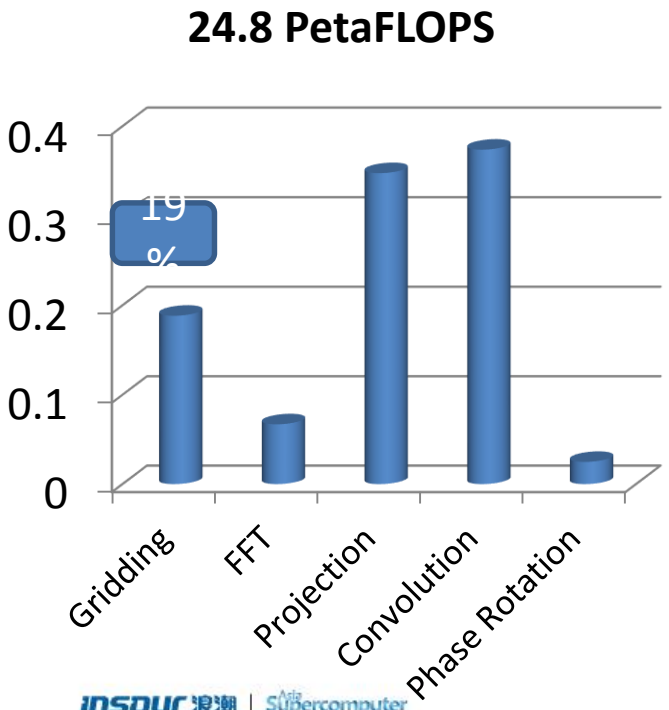
IOP Institute of Physics

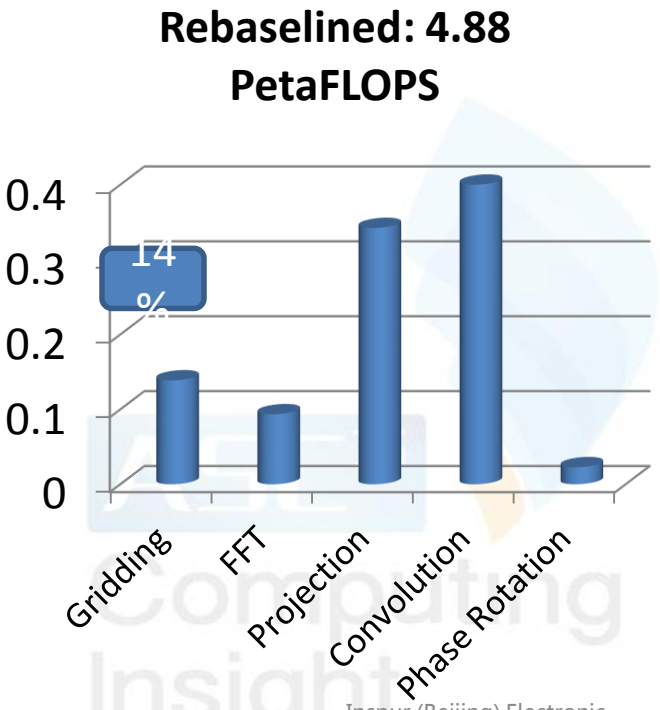
Gridding in the SKA1-Low





Gridding in the SKA1-Mid





Features of the Gridding Algorithm

$$grid(u,v) = \int_{-support}^{support} \int_{-support}^{support} Samples(u-x, v-y) \times C(-x,-y)\, dxdy$$

Given the size of *Samples* is *Nsamples*, the operation number of all the *Samples* is:

$$Nsamples \times (2 \times support + 1) \times (2 \times support + 1)$$

Features of the Gridding Algorithm

$$grid(u,v) = \sum_{x=-\text{support}}^{\text{support}} \sum_{y=-\text{support}}^{\text{support}} Samples(u-x, v-y) \times C(-x,-y)$$

For each *Samples*

- 1 multiplication and 1 addition
- 3 memory read and 1 memory write

Different Benchmark codes

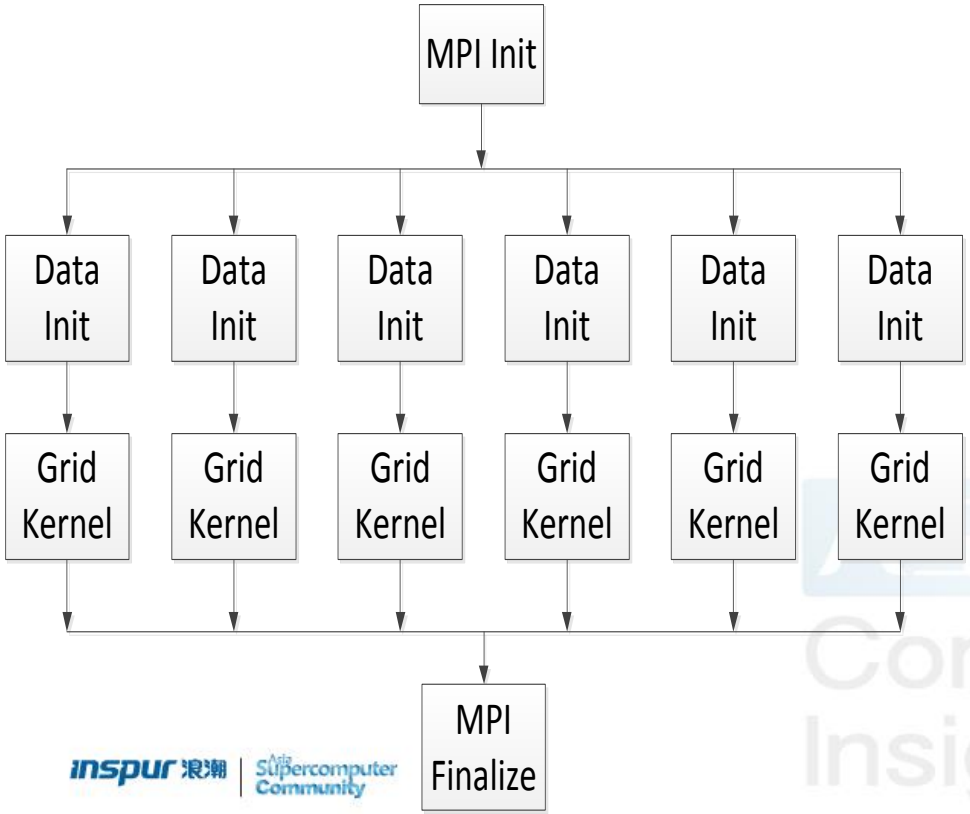
Australian Square Kilometre Array Pathfinder (ASKAP) benchmarks

- MPI
- CUDA

OpenCL gridding code (John Romein, ASTRON)

- Programmed by CUDA
- Highly optimized on GPU and CPU

Flow chart of the ASKAP Gridding code



- The amount of the computing is the same among MPI processes
- There is no MPI communication among processes

Parallel method 1: parallelizing the “dind” loop

```
const int sSize = 2 * support + 1;
for (int dind = 0; dind < int(samples.size()); ++dind) {
    int gind = samples[dind].iu + gSize * samples[dind].iv - support;
    int cind = samples[dind].cOffset;
    for (int suppv = 0; suppv < sSize; suppv++) {
        const Value d = samples[dind].data;
        for (int suppu = 0; suppu < sSize; suppu++) {
            grid[gind+suppu ] += d *C[cind+suppu ] ;
        }//end for suppu
        gind += gSize;
        cind += sSize;
    }//end for suppv
}//end for dind
```

Grid Kernel

Difficulties in the parallelizing and optimizing

Data dependency

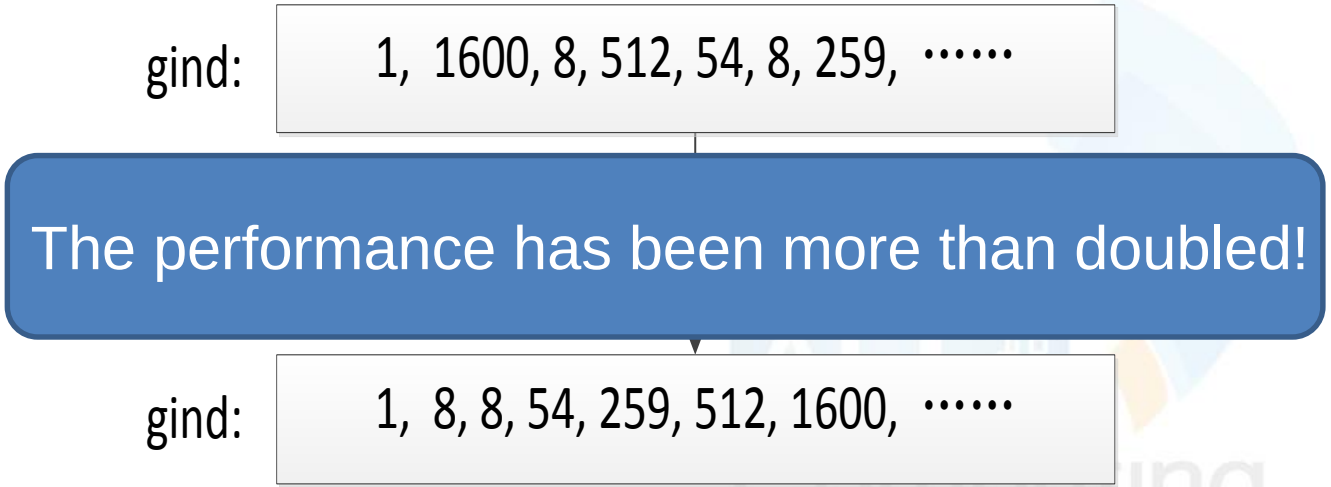
- The randomness of “gind” will make memory conflict that will lead the computing failure.
- Also make it impossible for the vectorization.

Indirect array index

- Because the index “gind” is random, it will increase cache misses and reduce performance.

The optimization method

Quick sort



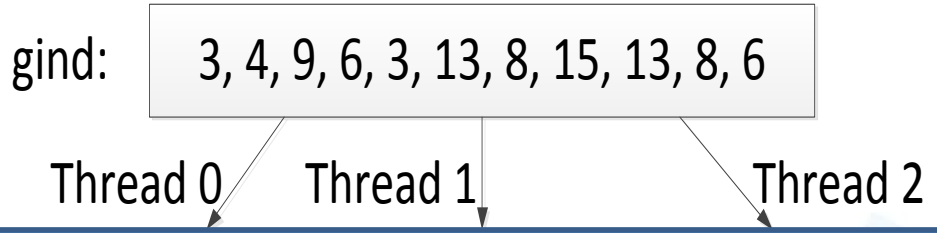
The optimization method

Fix the data dependency

- Sorting decrease the possibility of memory conflict.
- Using a barrier after the clause “gind+=sSize ”.

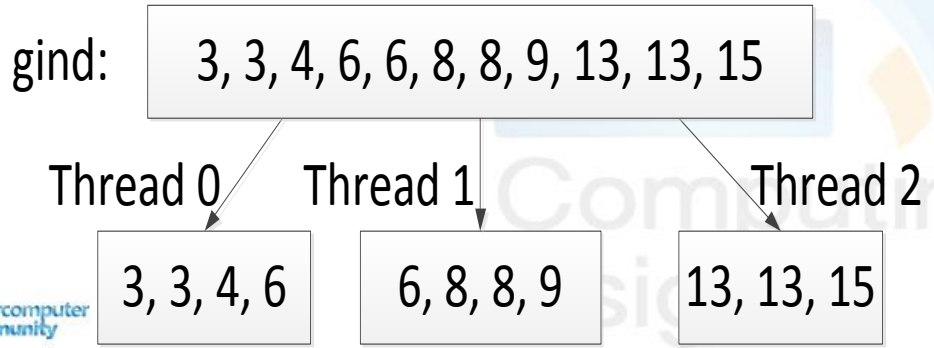
Vectorization

- Add the ”pragma simd” before the innermost loop to direct compiler vectorizes this loop automatically.



Why the sorting decreases the possibility of memory conflict?

A

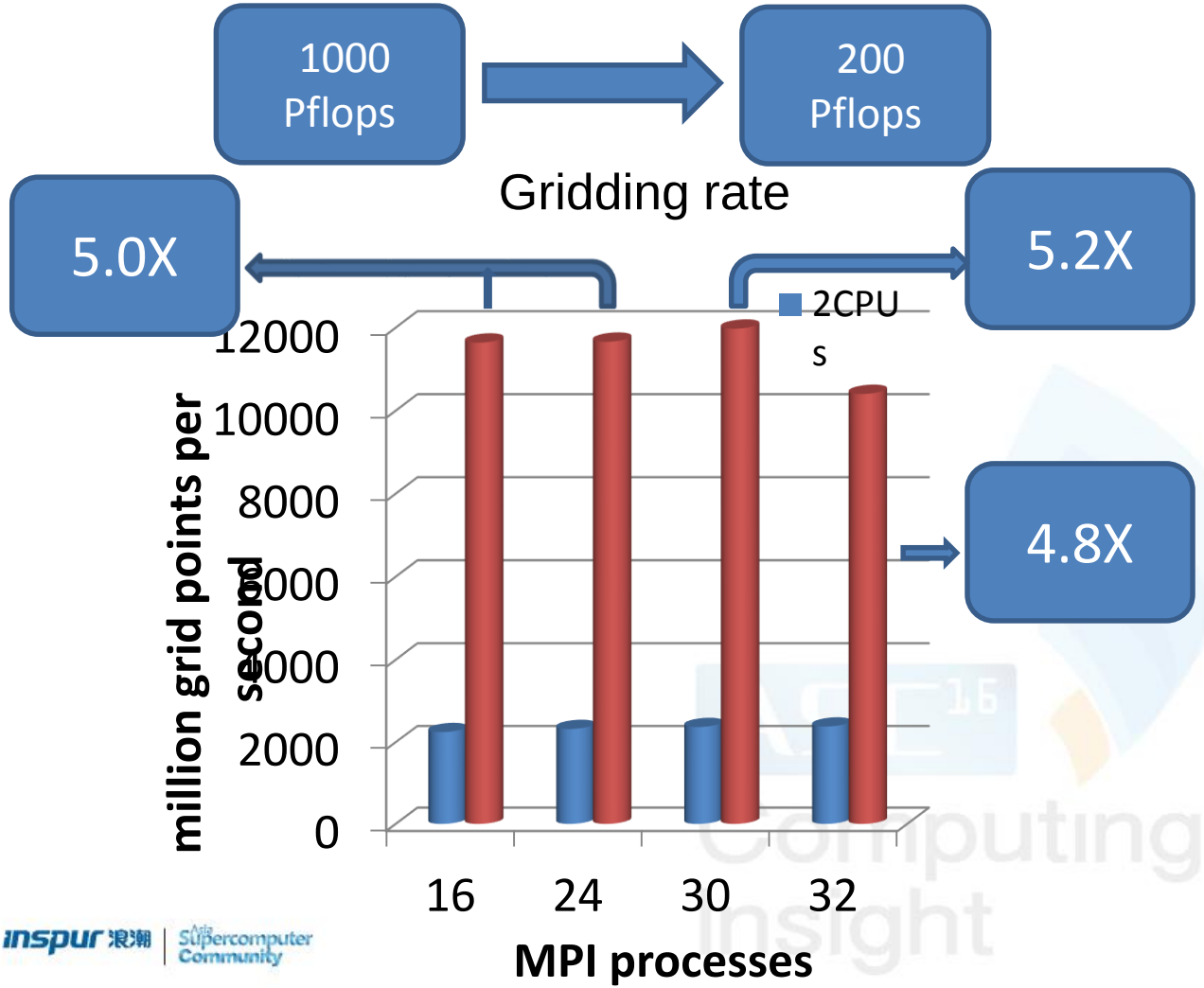


Testing Platform

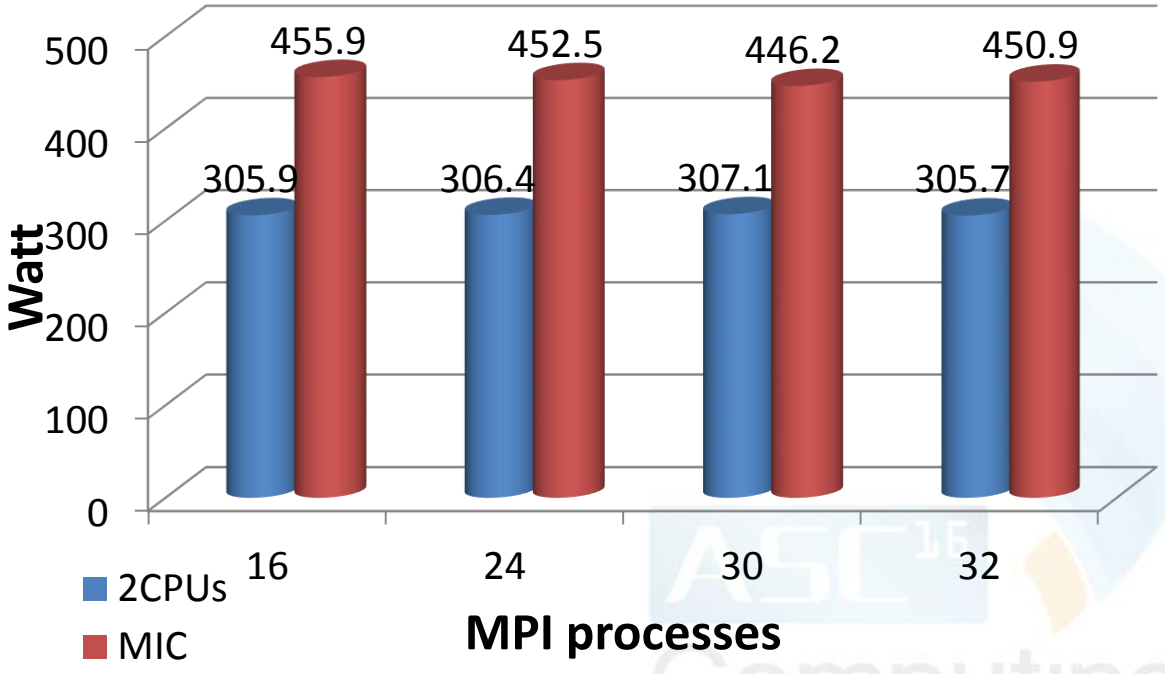
CPU		Intel(R) Xeon(R) E5-2650 v2, 8cores, 2.6GHz
Memory		128GB memory， 1333MHz
MIC		Intel Xeon Phi 7120P， 61cores，Frequency: 1.25GHz， GDDR Speed: 5.5GT/s， 16GB memory
Network		FDR Infiniband 56Gb/s
OS		Redhat 6.4, 2.6.32-358.el6.x86_64
MIC driver		MPSS： 3.2.1-1， Flash Version: 2.1.02.0390
Compiler	icpc	Intel(R) 64, Version 14.0.2.144 Build 20140120
	mpi	Intel(R) MPI Library for Linux* OS, Version 4.1 Update 3 Build 20140226

Testing cases

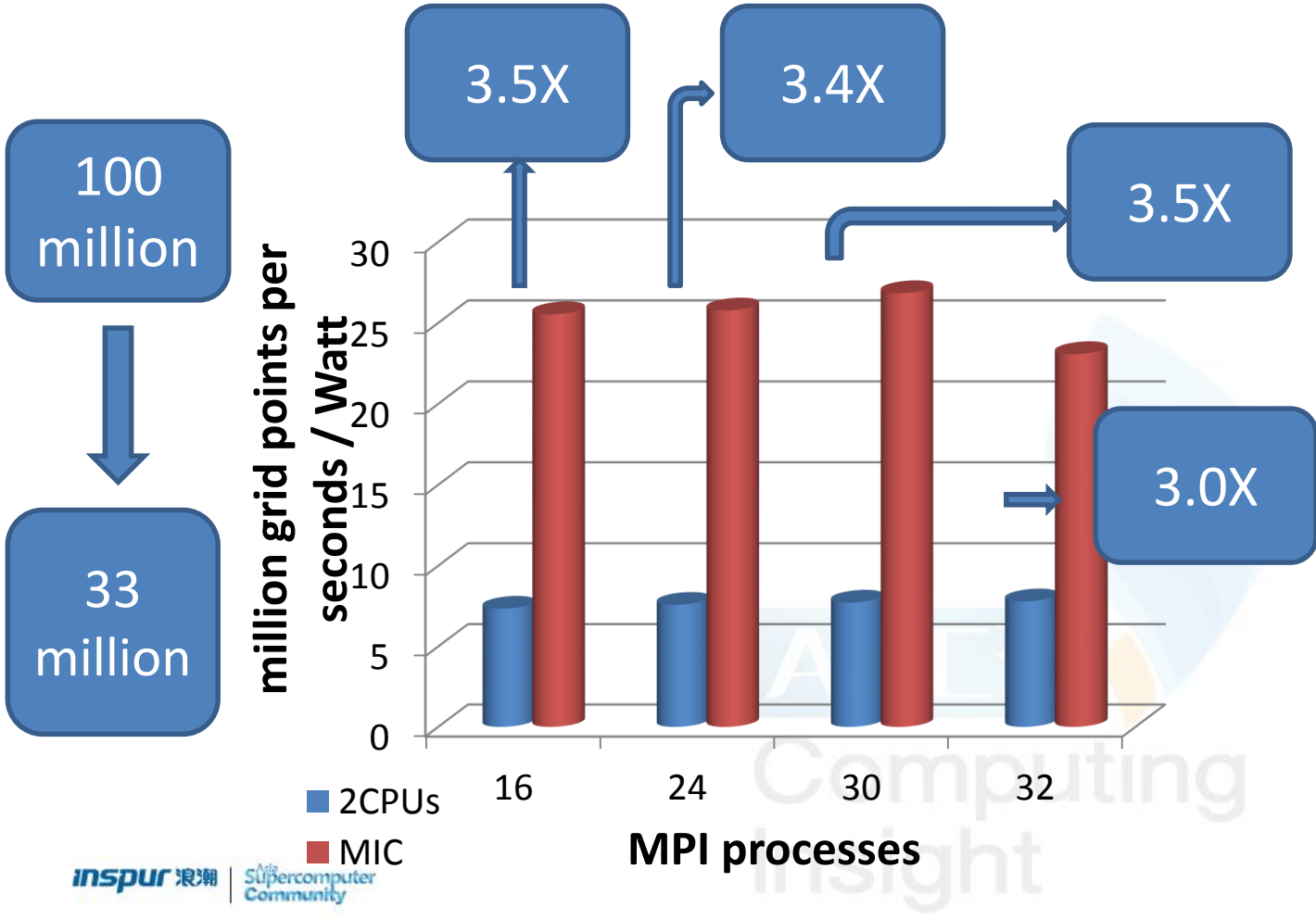
CPU	np=16,nt=1	np=24,nt=1	np=30,nt=1	np=32,nt=1
MIC	np=16,nt=15	np=24,nt=10	np=30,nt=8	np=32,nt=7


1000 Pflops
200 Pflops
Gridding rate
5.0X
5.2X
4.8X
2CPUs
12000
10000
8000
6000
4000
2000
0
million grid points per second
16
24
30
32
MPI processes

Energy consumption (Watt)



- The idle state of CPU+MIC is: 159.7 Watt
- The idle state of CPU (removing the MIC) is ：136.2 Watt
- The idle state of MIC：23.5 Watt


100 million
33 million
3.5X
3.4X
3.5X
3.0X
million grid points per seconds / Watt
30
25
20
15
10
5
0
16
24
30
32
2CPUs
MIC
MPI processes

With the help from ASC15 Student Supercomputer Challenge

152 teams, 5 continents

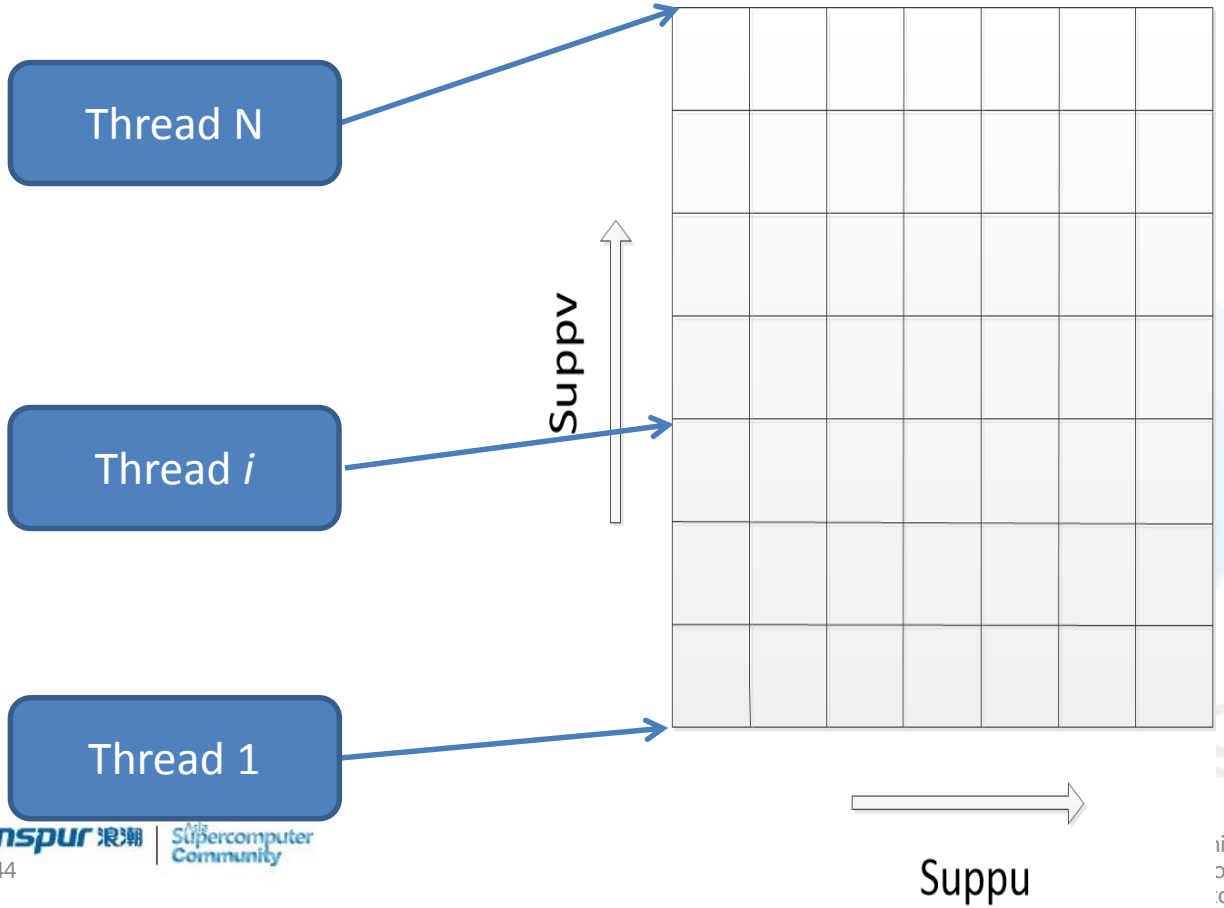
135 Universities: Tsinghua University, MIT, …

Parallel method 2: parallelizing the “suppv” loop

```
const int sSize = 2 * support + 1;
for (int dind = 0; dind < int(samples.size()); ++dind) {
   int gind = samples[dind].iu + gSize * samples[dind].iv - support;
   int cind = samples[dind].cOffset;
   for (int suppv = 0; suppv < sSize; suppv++) {
      const Value d = samples[dind].data;
      for (int suppu = 0; suppu < sSize; suppu++) {
         grid[gind+suppu ] += d *C[cind+suppu ] ;
      }//end for suppu
      gind += gSize;
      cind += sSize;
   }//end for suppv
}//end for dind
```

Grid Kernel

Avoid memory write conflict



A defect in the parallel method 2

The scalability is limited by the parameter *sSize*

If *sSize* is small,

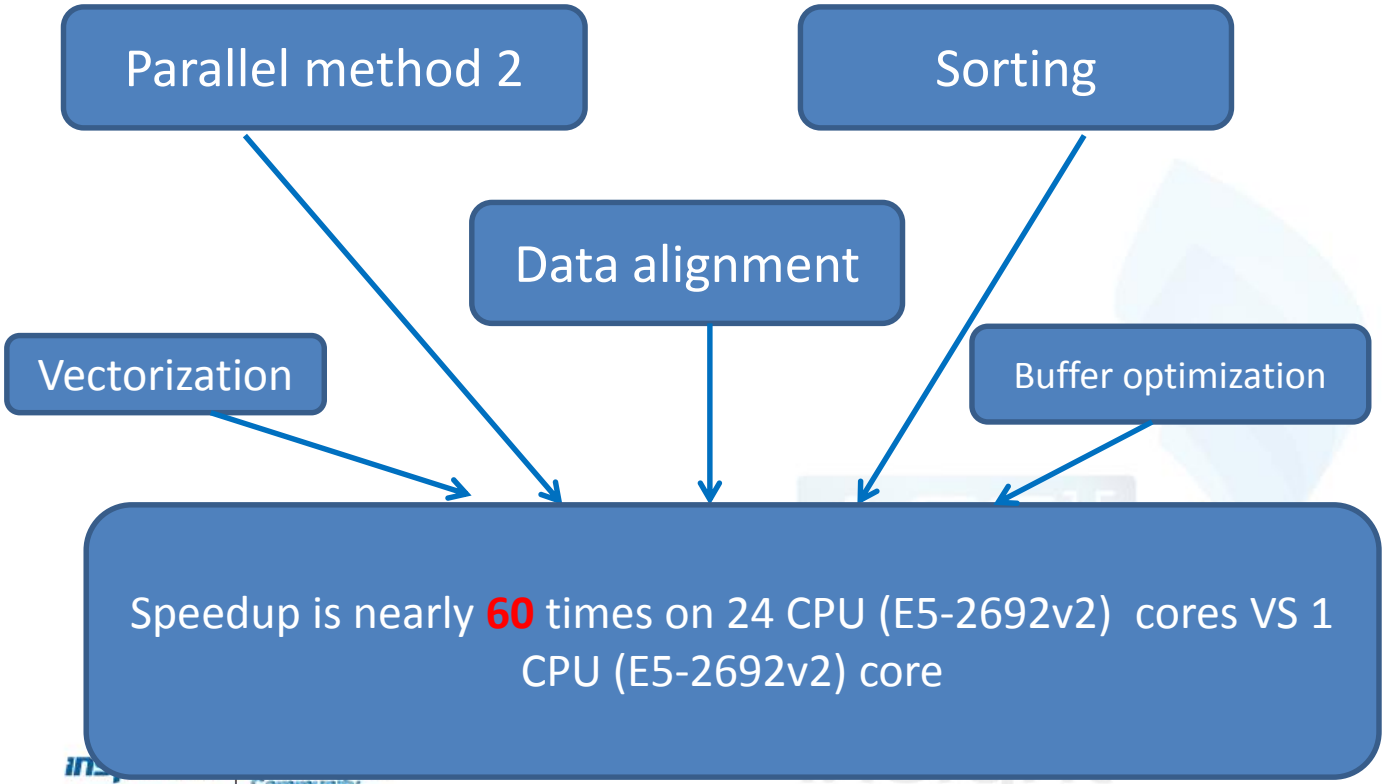
- *sSize*=129 in the preliminary

The maximum number of threads is no more than *sSize*

- 129 < 240, the maximum number of threads of the MIC

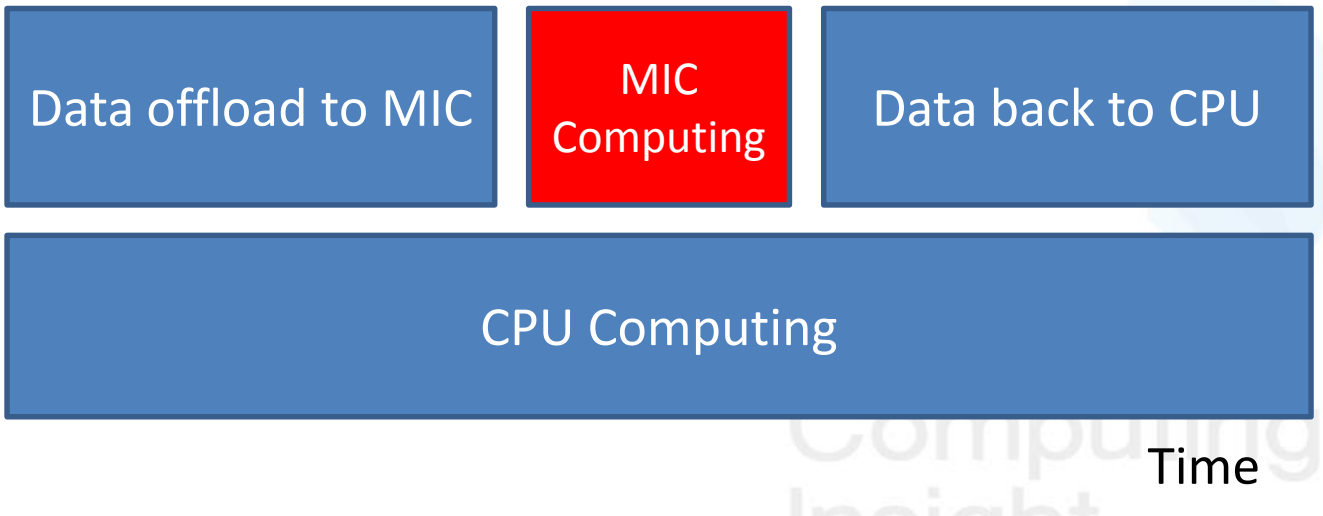
The smaller of the *sSize*, the worse of the performance

The best performance in the ASC15 preliminary stage



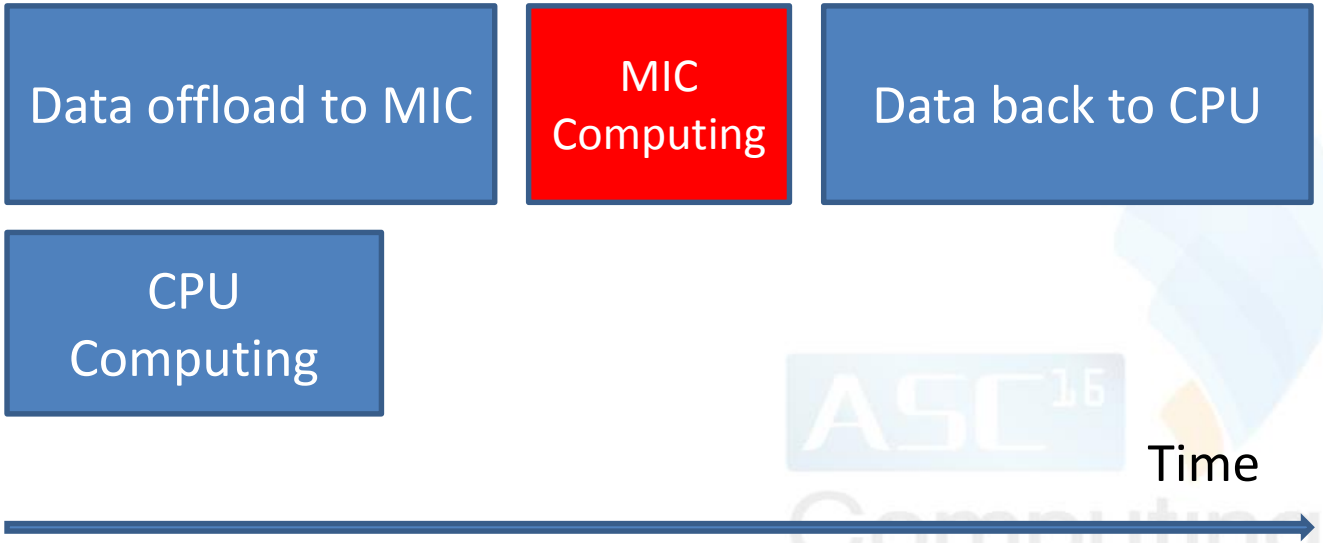
CPU+MIC collaborative computing

The number of memory accessing is in the same level as the number of float operation



CPU+MIC collaborative computing

For the worst case (the workload is small enough):



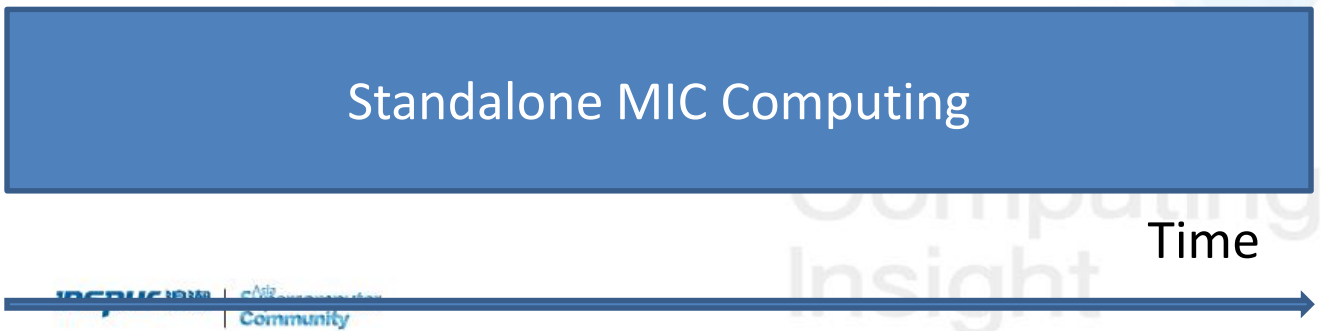
The Knights Landing Xeon Phi

Work as a host processor

Out of order cores

14nm, 3 Teraflops double-precision

16 GB DRAM on Chip (near memory), 5 times the performance of DDR4



Future plans

Further optimizing the Gridding Algorithm

- Based on the methods presented in the ASC15

Test the performance of the Gridding on Knights Landing Xeon Phi

Output a report including:

- Optimization methods on Knight Landing Xeon Phi
- Performance and energy consumption

THANKS