the Intel Xeon Phi coprocessor

1. Introduction
   - about the Intel Xeon Phi coprocessor
   - comparing Phi with CUDA
   - the Intel Many Integrated Core architecture

2. Programming the Intel Xeon Phi Coprocessor with OpenMP
   - hello world
   - an interactive session on Stampede
   - matrix-matrix multiplication
Introduction
- about the Intel Xeon Phi coprocessor
- comparing Phi with CUDA
- the Intel Many Integrated Core architecture

Programming the Intel Xeon Phi Coprocessor with OpenMP
- hello world
- an interactive session on Stampede
- matrix-matrix multiplication
the Intel Xeon Phi coprocessor

Intel Many Integrated Core Architecture or Intel MIC:

- 1.01 TFLOPS double precision peak performance on one chip.
- CPU-like versatile programming: *optimize once, run anywhere.*

Optimizing for the Intel MIC is similar to optimizing for CPUs.

- Programming Tools include OpenMP, OpenCL, Intel TBB, pthreads, Cilk.

In the High Performance Computing market, the Intel MIC is a direct competitor to the NVIDIA GPUs.
Is the Intel Xeon Phi coprocessor right for me?

from Intel: https://software.intel.com/mic-developer
the Intel Xeon Phi coprocessor

1 Introduction
- about the Intel Xeon Phi coprocessor
- comparing Phi with CUDA
- the Intel Many Integrated Core architecture

2 Programming the Intel Xeon Phi Coprocessor with OpenMP
- hello world
- an interactive session on Stampede
- matrix-matrix multiplication
similarities between Phi and CUDA

Also the Intel Phi is an accelerator:

- Massively parallel means: many threads run in parallel.
- Without extra effort unlikely to obtain great performance.
- Both Intel Phi and GPU devices accelerate applications in \textit{offload mode} where portions of the application are accelerated by a remote device.
- Directive-based programming is supported:
  - OpenMP on the Intel Phi; and
  - OpenACC on CUDA-enabled devices.
- Programming with libraries is supported on both.
differences between Phi and CUDA

But the Intel Phi is not just an accelerator:

- In *coprocessor native execution mode*, the Phi appears as another machine connected to the host, like another node in a cluster.

- In *symmetric execution mode*, application processes run on both the host and the coprocessor, communicating through some sort of message passing.

- MIMD versus SIMD:
  - CUDA threads are grouped in blocks (work groups in OpenCL) in a SIMD (Single Instruction Multiple Data) model.
  - Phi coprocessors run generic MIMD threads individually. MIMD = Multiple Instruction Multiple Data.
Introduction

- about the Intel Xeon Phi coprocessor
- comparing Phi with CUDA
- the Intel Many Integrated Core architecture

Programming the Intel Xeon Phi Coprocessor with OpenMP

- hello world
- an interactive session on Stampede
- matrix-matrix multiplication
The Intel Xeon Phi coprocessor is connected to an Intel Xeon processor, also known as the “host”, through a Peripheral Component Interconnect Express (PCIe) bus.

Since the Intel Xeon Phi coprocessor runs a Linux operating system, a virtualized TCP/IP stack could be implemented over the PCIe bus, allowing the user to access the coprocessor as a network node.

Thus, any user can connect to the coprocessor through a secure shell and directly run individual jobs or submit batch jobs to it.

The coprocessor also supports heterogeneous applications wherein a part of the application executes on the host while a part executes on the coprocessor.
host and device

from Intel: https://software.intel.com/mic-developer
Each core in the Intel Xeon Phi coprocessor is designed to be power efficient while providing a high throughput for highly parallel workloads.

A closer look reveals that the core uses a short in-order pipeline and is capable of supporting 4 threads in hardware.

It is estimated that the cost to support IA architecture legacy is a mere 2 of the area costs of the core and is even less at a full chip or product level.

Thus the cost of bringing the Intel Architecture legacy capability to the market is very marginal.
cores, caches, memory controllers, tag directories

from Intel: https://software.intel.com/mic-developer
The interconnect is implemented as a bidirectional ring.

Each direction is comprised of three independent rings:

1. The first, largest, and most expensive of these is the data block ring. The data block ring is 64 bytes wide to support the high bandwidth requirement due to the large number of cores.

2. The address ring is much smaller and is used to send read/write commands and memory addresses.

3. Finally, the smallest ring and the least expensive ring is the acknowledgement ring, which sends flow control and coherence messages.
the interconnect

from Intel: https://software.intel.com/mic-developer
architecture of one core

from Intel: https://software.intel.com/mic-developer
An important component of the Intel Xeon Phi coprocessor’s core is its Vector Processing Unit (VPU).

The VPU features a novel 512-bit SIMD instruction set, officially known as Intel Initial Many Core Instructions (Intel IMCI).

Thus, the VPU can execute 16 single precision (SP) or 8 double precision (DP) operations per cycle.

The VPU also supports Fused Multiply Add (FMA) instructions and hence can execute 32 SP or 16 DP floating point operations per cycle. It also provides support for integers.

Theoretical peak performance is 1.01 TFLOPS double precision:

\[ 1.053 \text{ GHz} \times 60 \text{ cores} \times 16 \text{ DP/cycle} = 1,010.88 \text{ GFLOPS}. \]
the Vector Processing Unit (VPU)

*from Intel:* https://software.intel.com/mic-developer
architectural difference between Phi and GPU

Applications that process large vectors may benefit and may achieve great performance on both the Phi and GPU, but notice the fundamental architectural difference:

- The Intel Phi has 60 cores each with a vector processing unit that can perform 16 double precision operations per cycle.

- The NVIDIA K20C has 13 streaming multiprocessors with 192 CUDA cores per multiprocessor, and threads are grouped in warps of 32 threads.

  The P100 has 56 streaming multiprocessors with 64 cores per multiprocessor.
Introduction
- about the Intel Xeon Phi coprocessor
- comparing Phi with CUDA
- the Intel Many Integrated Core architecture

Programming the Intel Xeon Phi Coprocessor with OpenMP
- hello world
- an interactive session on Stampede
- matrix-matrix multiplication
The TACC Stampede System

For this class, we have a small allocation on the Texas Advanced Computing Center (TACC) supercomputer: Stampede.

- Stampede is a 9.6 PFLOPS (1 petaflops is a quadrillion flops) Dell Linux Cluster based on 6,400+ Dell PowerEdge server nodes.

- Each node has 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor.

- The base cluster gives a 2.2 PF theoretical peak performance, while the co-processors give an additional 7.4 PF.
the program **hello.c**

```c
#include <omp.h>
#include <stdio.h>

#pragma offload_attribute (push, target (mic))

void hello ( void )
{
    int tid = omp_get_thread_num();
    char *s = "Hello World!";
    printf("%s from thread %d\n", s, tid);
}

#pragma offload_attribute (pop)

int main ( void )
{
    const int nthreads = 1;
    #pragma offload target (mic)
    #pragma omp parallel for
    for(int i=0; i<nthreads; i++) hello();
    return 0;
}
```

**Introduction to Supercomputing (MCS 572)**
specific pragmas

In addition to the OpenMP constructs:

- To make header files available to an application built for the Intel MIC architecture:
  
  ```
  #pragma offload_attribute (push, target (mic))
  
  #pragma offload_attribute (pop)
  ```

- The programmer uses
  
  ```
  #pragma offload target (mic)
  ```

to mark statements (offload constructs) that should execute on the Intel Xeon Phi coprocessor.
1 Introduction
- about the Intel Xeon Phi coprocessor
- comparing Phi with CUDA
- the Intel Many Integrated Core architecture

2 Programming the Intel Xeon Phi Coprocessor with OpenMP
- hello world
- an interactive session on Stampede
- matrix-matrix multiplication
an interactive session on Stampede

login4$ srun -p development -t 0:10:00 -n 1 --pty /bin/bash -l

Welcome to the Stampede Supercomputer

--> Verifying valid submit host (login4)...
--> Verifying valid jobname...
--> Enforcing max jobs per user...
--> Verifying availability of your home dir (/home1/02844/janv)...
--> Verifying availability of your work dir (/work/02844/janv)...
--> Verifying availability of your scratch dir (/scratch/02844/janv)...
--> Verifying valid ssh keys...
--> Verifying access to desired queue (development)...
--> Verifying job request is within current queue limits...
--> Checking available allocation (TG-DMS140008)...

srun: job 3206316 queued and waiting for resources
srun: job 3206316 has been allocated resources
compiling and running

We use the Intel C++ compiler icc:

c559-402$ icc -mmic -openmp -Wno-unknown-pragmas -std=c99 hello.c

c559-402$ ls -lt

total 9
-rwx------ 1 janv G-815233 11050 Apr 22 16:33 a.out
-rw------- 1 janv G-815233 331 Apr 22 16:21 hello.c

c559-402$ ./a.out

Hello World! from thread 0

c559-402$

Because of the pragma offload_attribute, the header files are wrapped and we can compile also without the options -mmic and -openmp:

```bash

Because of the `pragma offload_attribute`, the header files are wrapped and we can compile also without the options `-mmic` and `-openmp`:

```bash

c559-402$ icc -std=c99 hello.c

c559-402$ ./a.out

Hello World! from thread 0

c559-402$ more hello_old.c

c559-402$ exit

logout

login4$
```
Introduction

- about the Intel Xeon Phi coprocessor
- comparing Phi with CUDA
- the Intel Many Integrated Core architecture

Programming the Intel Xeon Phi Coprocessor with OpenMP

- hello world
- an interactive session on Stampede
- matrix-matrix multiplication
matrix-matrix multiplication

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

void MatMatMult ( int size,
    float (* restrict A)[size],
    float (* restrict B)[size],
    float (* restrict C)[size] )
{
    #pragma offload target(mic) \
    in(A:length(size*size)) in(B:length(size*size)) \
    out(C:length(size*size))
    {
        #pragma omp parallel for default(none) shared(C, size)
        for(int i=0; i<size; i++)
            for(int j=0; j<size; j++) C[i][j] = 0.0f;

        #pragma omp parallel for default(none) shared(A, B, C, size)
        for(int i=0; i<size; i++)
            for(int j=0; j<size; j++)
                for(int k=0; k<size; k++)
                    C[i][j] += A[i][k] * B[k][j];
    }
}
```
```c
int main ( int argc, char *argv[] )
{
    if(argc != 4) {
        fprintf(stderr,"Use: %s size nThreads nIter\n", argv[0]);
        return -1;
    }
    int i, j, k;
    int size = atoi(argv[1]);
    int nThreads = atoi(argv[2]);
    int nIter = atoi(argv[3]);
    omp_set_num_threads(nThreads);
    float (*restrict A)[size] = malloc(sizeof(float)*size*size);
    float (*restrict B)[size] = malloc(sizeof(float)*size*size);
    float (*restrict C)[size] = malloc(sizeof(float)*size*size);

    #pragma omp parallel for default(none) shared(A,B,size) private(i,j,k)
    for(i=0; i<size; i++)
        for(j=0; j<size; j++)
        {
            A[i][j] = (float)i + j;
            B[i][j] = (float)i - j;
        }
}```
double avgMultTime = 0.0;
MatMatMult(size, A, B, C); // warm up

double startTime = dsecnd();
for(int i=0; i < nIter; i++) MatMatMult(size, A, B, C);
double endTime = dsecnd();

avgMultTime = (endTime-startTime)/nIter;

#pragma omp parallel
#pragma omp master
printf("%s nThreads %d matrix %d %d runtime %g GFlop/s %g",
  argv[0], omp_get_num_threads(), size, size,
  avgMultTime, 2e-9*size*size*size/avgMultTime);

#pragma omp barrier
  free(A); free(B); free(C);
return 0;
}
references


- Intel Xeon Phi Coprocessor Developer’s Quick Start Guide. Whitepaper available at https://software.intel.com/mic-developer
