Using the Intel XEON PHI

The session is structured as follows:
• XEON PHI – architecture / software stack (7110-P)
• Configuring for use
• “mic 0” what it means ...?
• Native compilation
• Offload Pragmas
• INTEL Cilk
• INTEL MKL / TBB / IPP
• With OPENMP / Hybrid
• With MPI / Hybrid
• With OpenCL
• Debugging
• Tuning and performance analysis
• Questions
THANK YOU INTEL
Provided a Grizzly Pass server with the 7110P coprocessor

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>FORM FACTOR &amp; THERMAL SOLUTION</th>
<th>BOARD TDP (WATTS)</th>
<th>NUMBER OF CORES</th>
<th>FREQUENCY (GHz)</th>
<th>PEAK DOUBLE PRECISION PERFORMANCE (GFLOP)</th>
<th>PEAK MEMORY BANDWIDTH (GB/s)</th>
<th>MEMORY CAPACITY (GB)</th>
<th>INTEL TURBO BOOST TECHNOLOGY</th>
</tr>
</thead>
<tbody>
<tr>
<td>7110P</td>
<td>PCIe, Passive</td>
<td>300</td>
<td>61</td>
<td>1.238</td>
<td>1208</td>
<td>352</td>
<td>16</td>
<td>Peak turbo frequency: 1.33 GHz</td>
</tr>
<tr>
<td>7120X</td>
<td>PCIe, None</td>
<td>300</td>
<td>61</td>
<td>1.238</td>
<td>1208</td>
<td>352</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

CPU – Dual Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz - 32 Logical Processors (Total), 64GB of RAM - OS Centos 6.7 x86_64

INTEL MIC ARCHITECTURE OVERVIEW

- coprocessor consists of up to 61 cores connected by a high performance on-die bidirectional interconnect
- coprocessor runs a Linux operating system and supports all important Intel development tools, like C/C++ and Fortran compiler, MPI and OpenMP, high performance libraries like MKL, debugger and tracing tools like Intel VTune Amplifier XE
- traditional UNIX tools on the coprocessor are supported via BusyBox, which combines tiny versions of many common UNIX utilities into a single small executable
- coprocessor is connected to an Intel Xeon processor - the "host" - via the PCI Express (PCIe) bus
- virtualized TCP/IP stack allows to access the coprocessor like a network node (mic0)
An Intel Xeon Phi coprocessor contains many cores, each with a 512-bit vector arithmetic unit, capable of executing SIMD vector instructions.

- An L1 cache is included in each core (32 KB data + 32 KB instructions).
- An L2 cache is associated with each core (512 KB combined Data and Instr, L1 D cache is inclusive).
- A high-speed interconnect allows data transfer between the L2 caches and the memory subsystem.
- Each core can execute up to four HW threads simultaneously.
- This simultaneous multi-threading helps hide instruction and memory latencies.

<table>
<thead>
<tr>
<th>L1 (Data + Instructions)</th>
<th>Shared L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Size</td>
<td>32 KB + 32 KB</td>
</tr>
<tr>
<td>Miss Latency</td>
<td>15-30 cycles</td>
</tr>
</tbody>
</table>

The Intel Xeon Phi coprocessor includes three levels of memory (GDDR, L2 cache, and L1 cache)

- Accessing the L1 cache involves latency of only one cycle
- Since the Intel Xeon Phi coprocessor is an in-order machine, the latency of memory accesses has significant impact on software performance
- Refetches are one of the tools that can help hide memory latencies
Core
- the processor core (scalar unit) is an in-order architecture (based on the Intel Pentium processor family)
- fetches and decodes instructions from four hardware threads
- supports a 64-bit execution environment, along with Intel Initial Many Core Instructions
- does not support any previous Intel SIMD extensions like MME, SSE, SSE2, SSE3, SSE4.1, SSE4.2 or AVX instructions
- new vector instructions provided by the Intel Xeon Phi coprocessor instruction set utilize a dedicated 512-bit wide vector floating-point unit (VPU) that is provided for each core
- high performance support for reciprocal, square root, power and exponent operations, scatter/gather and streaming store capabilities to achieve higher effective memory bandwidth
- can execute 2 instructions per cycle, one on the U-pipe and one on the Vpipe (not all instruction types can be executed by the V-pipe, e.g. vector instructions can only be executed on the U-pipe)
- contains the L1 Icache and Dcache
- each core is connected to a ring interconnect via the Core Ring Interface (CRI)

Vector Processing Unit (VPU)
- the VPU includes the EMU (Extended Math Unit) and executes 16 single-precision floating point, 16 32-bit integer operations or 8 double-precision floating point operations per cycle. Each operation can be a fused multiply-add, giving 32 single-precision or 16 double-precision floating-point operations per cycle.
- contains the vector register file: 32 512-bit wide registers per thread context, each register can hold 16 singles or 8 doubles
- most vector instructions have a 4-clock latency with a 1 clock throughput

Core Ring Interface (CRI)
- hosts the L2 cache and the tag directory (TD)
- connects each core to an Intel Xeon Phi coprocessor Ring Stop (RS), which connects to the interprocessor core network

Ring
- includes component interfaces, ring stops, ring turns, addressing and flow control
- a Xeon Phi coprocessor has 2 of these rings, one travelling each direction

SBOX
- Gen2 PCI Express client logic
- system interface to the host CPU or PCI Express switch
- DMA engine

GBOX
- coprocessor memory controller
- consists of the FBOX (interface to the ring interconnect), the MBOX (request scheduler) and the PBOX (physical layer that interfaces with the GDDR devices
- There are 8 memory controllers supporting up to 16 GDDR5 channels. With a transfer speed of up to 5.5 GT/s a theoretical aggregated bandwidth of 352 GB/s is provided.

Performance Monitoring Unit (PMU)
- allows data to be collected from all units in the architecture
- does not implement some advanced features found in mainline IA cores (e.g. precise event-based sampling, etc.)
**Xeon Phi Architecture**

- Up to 4 SMT threads per core
- ~60 cores
- Per-core private L1+L2 caches, coherent
- Ring-based interconnect

**How many SMT threads? Varies!**

- Core:
  - at least two to keep pipeline fully occupied
  - more threads can overlap more latency
- Cache:
  - shared L1-I/D and L2, TLBs
  - threads can evict each other’s data
- Memory:
  - more threads keeps more requests outstanding
  - no more gains once bandwidth is saturated

- Per-core thread count on Xeon Phi affects performance in multiple ways
  - Core, cache, main memory effects
- Best thread count varies
  - Across applications, input sets, workload phases
- Dynamically determining best thread count
  - Per application phase (#pragma omp parallel)
  - Aggregate small phases
  - Use performance counters for more insight
**SOFTWARE STACK**

**LINUX or WINDOW HOST**

- **Device Driver:** In kernel space is the coprocessor device driver - responsible for managing device initialization and communication between the host and target devices.

- **Libraries:** on top of the device driver in user and system space - libraries provide basic card management capabilities such as enumeration of cards in a system, buffer management, and host-to-card communication - libraries also provide higher-level functionality such as loading and unloading the user executable onto coprocessor, invoking functions from the executable on the card, and providing a two-way notification mechanism between host and card - the libraries are responsible for buffer management and communication over the PCIe bus.

- **Tools:** Various tools that help maintain the software stack – micinfo, micctrl, micflash, micsmc.

- **Card OS (uOS):** The Linux-based operating system running on the Intel Xeon Phi coprocessor.

**Intel XEON Phi Coprocessor (Runs LINUX)**

**NOTE:** Linux source for uOS, the device driver, and the low-level SCIF library interface are here [http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss](http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss) - other low level interfaces (COI, MYO) used only by Intel tools are currently available for general use.
THREADED COUNT ACROSS APPLICATIONS

THREADED COUNT ACROSS INPUT SETS (BT)

PER-SECTION TUNING

- Calibration phase: for each section occurrence,
  - Measure hardware performance counters
    • Clock cycles, instruction count, L2 misses
    • More would have been nice but not supported by KNC
    • Use user-mode rdmc for low overhead
    • Try all 4 thread counts in subsequent occurrences
      • Pick best cycle count
      • Instruction count not stable: spin loops
  - Stable phase:
    • Keep watching L2 miss rate, recalibrate on change
    • Signifies data-dependent behavior

- Best vs. per-application setting based on B
Set up your account on host and on coprocessor

- Set up passwordless ssh between your login and mic0 using "ssh-keygen -t rsa" in the host and by logging into mic0
- Copy id_rsa.pub in .ssh of host to authorized_keys in .ssh of mic0
- Check for passwordless ssh as below:

```
[its-rc@its-rc-xeon-phi test1]$ ssh -C mic0 "cat /etc/issue"
Intel MIC Platform Software Stack (Built by Poky 7.0) 3.6 \n \l
[its-rc@its-rc-xeon-phi test1]$ 
```

- Set up .bashrc so that INTEL compiler and other paths are available – logout and login again
- Some commands to add to .bashrc are shown below and check for compiler ...

```
[its-rc@its-rc-xeon-phi test1]$ source /opt/intel/parallel_studio_xe_2016.2.062/bin/psetsvars.sh intel64
mpsvars.sh: Warning: Hardware events collection is disabled by default. To enable it, run mpsvars.sh with --vtune (recommended) or --papi option.
Copyright (C) 2009-2016 Intel Corporation. All rights reserved.
Intel(R) Inspector XE 2016 (build 450824)
Copyright (C) 2009-2015 Intel Corporation. All rights reserved.
Intel(R) VTune(TM) Amplifier XE 2016 (build 444446)
Copyright (C) 2009-2015 Intel Corporation. All rights reserved.
Intel(R) Advisor XE 2016 (build 450722)
[its-rc@its-rc-xeon-phi test1]$ mpsvars.sh --vtune
[its-rc@its-rc-xeon-phi test1]$ compilervars.sh intel64
[its-rc@its-rc-xeon-phi test1]$ 
```

```
[its-rc@its-rc-xeon-phi test1]$ gcc -v
gcc version 4.4.7 (GCC) 20100425 (Red Hat 4.4.7-2)
```

```
[its-rc@its-rc-xeon-phi test1]$ icc -V
icc version 16.0.2 [gcc version 4.4.7 compatibility]
```

```
[its-rc@its-rc-xeon-phi test1]$ icc -V
Intel(R) C and C++ Compiler for applications running on Intel(R) 64, Version 16.0.2.181 Build 20160204
Copyright (C) 1985-2016 Intel Corporation. All rights reserved.

icc: NOTE: The evaluation period for this product ends on 19-mar-2016 UTC.
[its-rc@its-rc-xeon-phi test1]$ 
```

- Set up OPENMP and other varaibles in .bashrc or source as shown below

```
[its-rc@its-rc-xeon-phi test2]$ cat setup_openmp
#export KMP_ALL_THREADS=16
#export MICOMP_NUM_THREADS=128
export MIC_ENV_PREFIX=MIC
export MIC_KMP_AFFINITY=balanced
[its-rc@its-rc-xeon-phi test2]$ ./setup_openmp
```

```
[its-rc@its-rc-xeon-phi test2]$ 
```
NATIVE COMPILATION

```
[its-rc@its-rc-xeon-phi test]s make
  icc -openmp -03 -vec-report=3 -mmic helloflop_xeon.c -o helloflop_xeon_xphi
  icc: command line remark #10411: option '-openmp' is deprecated and will be removed in a future release. Please use the replacement option '-qopenmp'
  icc: command line remark #10010: option '-vec-report=3' is deprecated and will be removed in a future release. See '-help deprecated'
  icc: remark #10397: optimization reports are generated in *.optpt files in the output location
helloflop_xeon.c(32): warning #3639: non-portable spelling of GNU aligned attribute - use 'aligned' instead of 'align' for portability
  float a[FLOPS_ARRAY_SIZE] __attribute__((aligned(64)));

helloflop_xeon.c(34): warning #3639: non-portable spelling of GNU aligned attribute - use 'aligned' instead of 'align' for portability
  float b[FLOPS_ARRAY_SIZE] __attribute__((aligned(64)));

[its-rc@its-rc-xeon-phi test]s ls -la
total 40
  drwxrwxr-x  2 its-rc its-rc  4096 Feb 18 13:41 .
  drwxrwxr-x  3 its-rc its-rc  4096 Feb 18 13:29 ..
  lrwxrwxrwx  1 its-rc its-rc  2593 Feb 18 13:36 helloflop_xeon.c
  lrwxrwxrwx  1 its-rc its-rc  1670 Feb 18 13:41 helloflop_xeon.optpt
  lrwxrwxrwx  1 its-rc its-rc  1972 Feb 18 13:41 helloflop_xeon_xphi
  lrwxrwxrwx  1 its-rc its-rc  631 Feb 18 13:41 Makefile
[its-rc@its-rc-xeon-phi test]s cat Makefile
CFLAGS = -openmp -03 -vec-report=3
CC = icc
FLFLAGS = -align array=4byte -openmp -03 -vec-report=3
MICLIBS = /home/its-rc/downloads/test_nkr/test1
MICLIB = /opt/intel/clk/3.1.2.088/provider/share/common/lib/intel64
hfllops: helloflop_xeon.c
  $(CC) $(CFLAGS) -mmic helloflop_xeon.c -o helloflop_xeon_xphi
all: hfllops
mccopy:
  scp *.xphi nic8:$ (MICLIB)
mccilibcopy:
  scp $(MICLIB)/libiom5.so nic8:$ (MICLIB)
  @echo ""
  @echo "REMEMBER TO export LD_LIBRARY_PATH=$ (MICLIB) ON THE COPROCESSOR (if needed)"
  @echo ""
clean:
  rm -f,./xphi
  rm -f,./xphi
[its-rc@its-rc-xeon-phi test]s make mccopy
mccopy: xphi nic8:/home/its-rc/downloads/test_nkr/test1
hfllops nic8:/home/its-rc/downloads/test_nkr/test1
helloflop_xeon_xphi
[its-rc@its-rc-xeon-phi test]s make mccilibcopy
mccilibcopy: /opt/intel/clk/3.1.2.088/provider/share/common/lib/intel64/libiom5.so nic8:/home/its-rc/downloads/test_nkr/test1
libiom5.so
[its-rc@its-rc-xeon-phi test]s
```

```
The remote process indicated that the following libraries could not be loaded: libiomp5.so
Error creating remote process, at least one library dependency is missing.
Please check the list of dependencies below to see which
one is missing and update the SINK_LD_LIBRARY_PATH
environment variable to include the missing library.

Dependency information for helloflop_xeon_xphi

Full path was resolved as
/home/its-rc/Downloads/test_nkr/test1/helloflop_xeon_xphi

Binary was built for Intel(R) Xeon Phi(TM) Coprocessor
(codename: Knights Corner) architecture

SINK_LD_LIBRARY_PATH =

Dependencies Found:

(None found)

Dependencies Not Found Locally (but may exist already on the coprocessor):

libm.so.6
libiomp5.so
libgcc_s.so.1
libpthread.so.0
libc.so.6
libdl.so.2

[its-rc@its-rc-xeon-phi test1]$ ssh -C mic0 hostname
its-rc-rc-xeon-phi-mic0

[its-rc@its-rc-xeon-phi test1]$ export SINK_LD_LIBRARY_PATH=/home/its-rc/Downloads/test_nkr/test1:$SINK_LD_LIBRARY_PATH
[its-rc@its-rc-xeon-phi test1]$ mnicativeloadex helloflop_xeon_xphi -l

Dependency information for helloflop_xeon_xphi

Full path was resolved as
/home/its-rc/Downloads/test_nkr/test1/helloflop_xeon_xphi

Binary was built for Intel(R) Xeon Phi(TM) Coprocessor
(codename: Knights Corner) architecture

SINK_LD_LIBRARY_PATH = /home/its-rc/Downloads/test_nkr/test1:

Dependencies Found:

/home/its-rc/Downloads/test_nkr/test1/libiomp5.so

Dependencies Not Found Locally (but may exist already on the coprocessor):

libm.so.6
libpthread.so.0
libc.so.6
libdl.so.2
libgcc_s.so.1

[its-rc@its-rc-xeon-phi test1]$ mnicativeloadex helloflop_xeon_xphi
Initializing
Starting Compute on 240 threads
GFlops = 0144.000, Secs = 2.958, GFlops per sec = 2877.383

[its-rc@its-rc-xeon-phi test1]$
// A simple example that gets lots of FLOPS (Floating Point Operations)
// Intel(R) Xeon Phi(TM) co-processors using openmp to scale

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include <sys/time.h>

// dtime
// returns the current wall clock time

double dtime()
{
    double seconds = 0.0;
    struct timeval mytime;
    gettimeofday(&mytime,(struct timezone*)0);
    seconds = (double)(mytime.tv_sec + mytime.tv_usec*1.0e-6);
    return(seconds);
}

#define FLOPS_ARRAY_SIZE (1024*1024)
#define MAXFLOPS_ITERS 1000000000
#define LOOP_COUNT 128

// number of float pt ops per calculation
#define FLOPS_PERCALC 2
// define some arrays -
// make sure they are 64 byte aligned
// for best cache access
float fa[FLOPS_ARRAY_SIZE] __attribute__((align(64)));
float fb[FLOPS_ARRAY_SIZE] __attribute__((align(64)));

// Main program - pedal to the metal...calculate using tons o'flops!
int main(int argc, char *argv[])
{
    int i,j,k;

    int numthreads;
    double tstart, tstop, ttime;
    double gflops = 0.0;
    float a=1.1;

    // initialize the compute arrays

        #pragma omp parallel
        #pragma omp master
        numthreads = omp_get_num_threads();
        printf("Initializing\n");
        #pragma omp parallel for
        for(i=0; i<FLOPS_ARRAY_SIZE; i++)
        {
            fa[i] = (float)i + 0.1;
            fb[i] = (float)i + 0.2;
        }
        
        if(numthreads > 1)
        
        tstart = dtime();
        // scale the calculation across threads requested
        // need to set environment variables OMP_NUM_THREADS and OMP AFFINITY
        
        #pragma omp parallel for private(i,k)
        for(i=0; i<numthreads; i++)
        {
            // each thread will work it's own array section
            // calc offset into the right section
            int offset = i*LOOP_COUNT;

            // loop many times to get lots of calculations
            for(j=0; j<MAXFLOPS_ITERS; j++)
            {
                // scale 1st array and add in 2nd array
                for(k=0; k<LOOP_COUNT; k++)
                {
                    fa[k+offset] = a * fa[k+offset] + fb[k+offset];
                }
            }
        }

        tstop = dtime();
        // # of gigaflops we just calculated
        gflops = (double)((MAXFLOPS_ITERS*LOOP_COUNT) / MAXFLOPS_ITERS*FLOPS_PERCALC);

        // elapsed time
        ttime = tstop - tstart;
        //
        // Print the results
        //
        if ((ttime > 0.0))
        {
            printf("GFlops = %.3lf, Secs = %.3lf, GFlops per sec = %.3lf\n", gflops, ttime, gflops/ttime);
        }

        return( 0 );
}
```c
#include <stdlib.h>
#include <stdio.h>

#define SIZE 10

void func(int *p)
{
    int i;
    // Because transfer count is 0, no data is transferred to MIC
    // However, because "in" is used,
    // the pointer value gets initialized on MIC
    #pragma offload target(mic:0) in:p:length(0) REUSE
    { for(i=0; i<SIZE; i++) printf("%3d", p[i]); printf("\n"); }
}

int main()
{
    int i;
    int *a;
    a = (int *)malloc(SIZE*sizeof(int));
    for(i=0; i<SIZE; i++) a[i] = i;
    // Allocate a on MIC only; transfer no data
    #pragma offload_transfer target(mic:0) nocopy(a:length(SIZE) ALLOC)
    // Pick up the memory allocated for a and transfer data into it
    // Transfer count of SIZE is used
    // Each element of a is printed, then incremented
    #pragma offload target(mic) in:a:length(SIZE) REUSE
    { for(i=0; i<SIZE; i++) printf("%3d", a[i]++); printf("\n"); }
    func(a);
    // Fetch data back to CPU, and free a on MIC
    #pragma offload_transfer target(mic:0) out:a:length(SIZE) FREE
    return 0;
}
```

### OFFLOAD PRAGMAS AVAILABLE (more ...)

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Syntax</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C/C++</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offload pragma</td>
<td><code>#pragma offload &lt;clauses&gt; &lt;statement&gt;</code></td>
<td>Allow next statement to execute on coprocessor or host CPU</td>
</tr>
<tr>
<td>Variable/function offload properties</td>
<td><code>_attribute_ ((target (mic)))</code></td>
<td>Compile function for, or allocate variable on, both host CPU and coprocessor</td>
</tr>
<tr>
<td>Entire blocks of data/code defs</td>
<td><code>#pragma offload_attribute (push, target (mic))</code> … <code>#pragma offload_attribute (pop)</code></td>
<td>Mark entire files or large blocks of code to compile for both host CPU and coprocessor</td>
</tr>
<tr>
<td><strong>Fortran</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offload directive</td>
<td><code>!dir$ omp offload &lt;clauses&gt; &lt;statement&gt;</code></td>
<td>Execute OpenMP parallel block on coprocessor</td>
</tr>
<tr>
<td>Variable/function offload properties</td>
<td><code>!dir$ attributes offload: (mic) :: &lt;ret-name&gt; OR &lt;var1, var2, ...&gt;</code></td>
<td>Compile function or variable for CPU and coprocessor</td>
</tr>
<tr>
<td>Entire code blocks</td>
<td><code>!dir$ offload begin &lt;clauses&gt;</code> … <code>!dir$ end offload</code></td>
<td>Mark entire files or large blocks of code to compile for both host CPU and coprocessor</td>
</tr>
</tbody>
</table>

### CLAUSES to CONTROL DATA TRANSFERS

<table>
<thead>
<tr>
<th>Clause</th>
<th>Syntax</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple coprocessors</td>
<td><code>target (mic[:unit])</code></td>
<td>Select specific coprocessors</td>
</tr>
<tr>
<td>Inputs</td>
<td><code>in (var-list modifiers)</code></td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>Outputs</td>
<td><code>out (var-list modifiers)</code></td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td><code>inout (var-list modifiers)</code></td>
<td>Copy host to coprocessor and back when offload completes</td>
</tr>
<tr>
<td>Non-copied data</td>
<td><code>nocopy (var-list modifiers)</code></td>
<td>Data is local to target</td>
</tr>
</tbody>
</table>

### Modifiers (optional)

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Syntax</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify copy length</td>
<td><code>length(N)</code></td>
<td>Copy N elements of pointer’s type</td>
</tr>
<tr>
<td>Coprocessor memory allocation</td>
<td><code>alloc_if (bool)</code></td>
<td>Allocate coprocessor space on this offload (default: TRUE)</td>
</tr>
<tr>
<td>Coprocessor memory release</td>
<td><code>free_if (bool)</code></td>
<td>Free coprocessor space at the end of this offload (default: TRUE)</td>
</tr>
<tr>
<td>Control target data alignment</td>
<td><code>align (N bytes)</code></td>
<td>Specify minimum memory alignment on coprocessor</td>
</tr>
<tr>
<td>Array partial allocation &amp; variable relocation</td>
<td><code>alloc (array-slice) into (var-exp)</code></td>
<td>Enables partial array allocation and data copy into other var &amp; ranges</td>
</tr>
</tbody>
</table>
Explicit Work-sharing

```c
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
{
   //section running on the coprocessor
   #pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
   {
      mm(n,a,b,c);
   }
}
#pragma omp section
{
   //section running on the host
   mm(n,d,e,f);
   
}
}
}
```

- To explicitly share work between the coprocessor and the host one can use OpenMP sections to manually distribute the work

Persistent Data on the Coprocessor

- Main bottleneck of accelerator based programming are data transfers over the slow PCIe bus between host and accelerator - minimize data transfers - keep the data on the coprocessor between computations using the same data

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

one can simply use the following notation:

- to allocate data and keep it for the next offload
  ```c
  #pragma offload target(mic) in (p:length(1) ALLOC RETAIN)
  ```
- to reuse the data and still keep it on the coprocessor
  ```c
  #pragma offload target(mic) in (p:length(1) REUSE RETAIN)
  ```
- to reuse the data again and free the memory. (FREE is the default, and does not need to be explicitly specified)
  ```c
  #pragma offload target(mic) in (p:length(1) REUSE FREE)
  ```
```c
#include <stdio.h>

#define SIZE 10
#define ALLLOC alloc_if(1) free_if(0)
define REUSE alloc_if(0) free_if(0)
#define FREE alloc_if(0) free_if(1)

// Example of Non-Bitwise Object Transfer, All Data Elements Needed
typedef struct {
    int ni;
    int *m2;
} nbws;

decisper(target(mic); nbws struct1;

void send_inputs() {
    int n;
    int *m2;
    // Initialize the struct
    struct1.m1 = 10;
    struct1.m2 = (int *)malloc(SIZE * sizeof(int));
    for (int i=0; i<SIZE; i++)
    {    struct1.m2[i] = i;
    }
    // In this offload data is transferred
    m1 = struct1.m1;
    m2 = struct1.m2;
    #pragma offload target(mic:0) in(m1) in(m2[0:SIZE] : ALLLOC) nocopy(struct1)
    {
        struct1.m1 = m1;
        struct1.m2 = m2;
        printf("MIC offload: struct1.m2[0] = %d, struct1.m2[SIZE-1] = %d\n", struct1.m2[0], struct1.m2[SIZE-1];
    }
    [SIZE-1];
    fflush(0);
}
}

void use_the_data() {
    // In this offload data is used and updated
    #pragma offload target(mic:0) nocopy(struct1)
    {
        for (int i=0; i<SIZE; i++)
        {    struct1.m2[i] += 1;
        }
        printf("MIC offload2: struct1.m2[0] = %d, struct1.m2[SIZE-1] = %d\n", struct1.m2[0], struct1.m2[SIZE-1];
    }
    [SIZE-1];
    fflush(0);
}

void receive_results() {
    int *m2;
    // In this offload data is used, freed on MIC and brought back to the CPU
    m2 = struct1.m2;
    #pragma offload target(mic:0) out(m2[0:SIZE] : FREE) nocopy(struct1)
    {
        for (int i=0; i<SIZE; i++)
        {    struct1.m2[i] += 1;
        }
        printf("MIC offload3: struct1.m2[0] = %d, struct1.m2[SIZE-1] = %d\n", struct1.m2[0], struct1.m2[SIZE-1];
    }
    [SIZE-1];
    fflush(0);
}

int main()
{
    send_inputs();
    use_the_data();
    receive_results();
    return 0;
}
```
#define ROWCHUNK 96
#define COLCHUNK 96

#pragma omp parallel for collapse(2) private(i,j,k)
    for(i = 0; i < n; i+=ROWCHUNK ) {
        for(j = 0; j < n; j+=ROWCHUNK ) {
            for(k = 0; k < n; k+=COLCHUNK ) {
                for (ii = i; ii < i+ROWCHUNK; ii+=6) {
                    for (kk = k; kk < k+COLCHUNK; kk++) {
                        #pragma ivdep
                        #pragma vector aligned
                        for ( jj = j; jj < j+ROWCHUNK; jj++){
                            c[(ii*n)+jj] += a[(ii*n)+kk]*b[kk*n+jj];
                            c[((ii+1)*n)+jj] += a[((ii+1)*n)+kk]*b[kk*n+jj];
                            c[((ii+2)*n)+jj] += a[((ii+2)*n)+kk]*b[kk*n+jj];
                            c[((ii+3)*n)+jj] += a[((ii+3)*n)+kk]*b[kk*n+jj];
                            c[((ii+4)*n)+jj] += a[((ii+4)*n)+kk]*b[kk*n+jj];
                            c[((ii+5)*n)+jj] += a[((ii+5)*n)+kk]*b[kk*n+jj];
                        }
                    }
                }
            }
        }
    }
}
intel cilk plus parallel extensions

- shared-memory model (cilk_shared/cilk_offload) can be used for both data and task parallelism on a xeon phi coprocessor
- complex data structures can be handled by virtual shared memory
- same virtual address space is used on both the host and the coprocessor, enabling a seamless sharing of data
- there are also cilk functions to specify offloading of functions and cilk_for loops
**Intel MKL (Math Kernel Library)**

- Intel Xeon Phi coprocessor is supported since MKL 11.0
- As of Intel MKL 11.0 Update 2 the following functions are highly optimized for the Intel Xeon Phi coprocessor
  1. BLAS Level 3, and much of Level 1 & 2
  3. LAPACK routines (LU, QR, Cholesky)
  4. Fast Fourier Transformations
  5. Vector Math Library
  6. Random number generators in the Vector Statistical Library

**MKL Usage Modes**

The following 3 usage models of MKL are available for the Xeon Phi:

1. **Automatic Offload**
2. **Compiler Assisted Offload**
3. **Native Execution**

**Automatic Offload:**

- To enable automatic offload either the function `mkl_mic_enable()` has to be called within the source code or the environment variable `MKL_MIC_ENABLE=1` has to be set. If no Xeon Phi coprocessor is detected the application runs on the host without penalty.
- By default, the MKL library decides when to offload and also tries to determine the optimal work division between the host and the targets (MKL can take advantage of multiple coprocessors).
- In case of the BLAS routines the user can specify the work division between the host and the coprocessor by calling the routine `mkl_mic_set_Workdivision(MKL_TARGET_MIC,0,0.5)` or by setting the environment variable `MKL_MIC_0_WORKDIVISION=0.5`.
- Both examples above specify to offload 50% of computation only to the 1st card (card #0).
- Example functions one can use for auto offload: Level 3 BLAS functions - ?GEMM (for m,n > 2048, k > 256), ?TRSM (for M,N > 3072), ?TRMM (for M,N > 3072), ?SYMM (for M,N > 2048), LAPACK functions – LU (M,N > 8192, QR, Cholesky).

**Compiler Assisted Offload:**

- In this mode of MKL the offloading is explicitly controlled by compiler pragmas or directives. In contrast to the automatic offload mode, all MKL function can be offloaded in CAO-mode.
- It allows for data persistence on the device.
- For Intel compilers it is possible to use AO and CAO in the same program, however the work division must be explicitly set for AO in this case. Otherwise, all MKL AO calls are executed on the host.
- Setting larger pages by the environment setting `MIC_USE_2MB_BUFFERS=16K` usually increases performance.
Example of CAO for MKL's `sgemm` routine shown below:

```c
#pragma offload target(mic) \
    in(transa, transb, N, alpha, beta) \
    in(A:transa, N) \
    in(B:transb, N) \
    in(C:N) \
    out(C:N) \
alloc_if(0)
{
    sgemm(&transa, &transb, N, N, N, &alpha, A, &N, B, &N, &beta, C, &N);
}
```

Native Execution:
- In this mode of MKL the Intel Xeon Phi coprocessor is used as an independent compute node. It allows for data persistence on the device.
- `icc -O3 -mkl -mmic file.c -o file`

Automatic Offload MKL DGEMM: `export MKL_MIC_ENABLE=1` / `icc -O3 -mkl dgemm_with_timing.c -o dgemm_with_timing`

```
```

This example measures performance of Intel(R) MKL function `dgemm`
computing real matrix C=A*B+beta*C, where A, B, and C are matrices and alpha and beta are double precision scalars.

Initializing data for matrix multiplication C=A*B for matrix A(12000x200) and matrix B(200x1000).

Allocating memory for matrices aligned on 64-byte boundary for better performance.

Initializing matrix data.
Making the first run of matrix product using Intel(R) MKL `dgemm` function via BLAS interface to get stable run time measurements.

Measuring performance of matrix product using Intel(R) MKL `dgemm` function via BLAS interface.

Matrix multiplication using Intel(R) MKL `dgemm` function completed.

Deallocating memory.

It is highly recommended to define LOOP COUNT for this example on your computer at 3 to have total execution time about 1 second for reliability of measurements.

Example completed.
```
Native Execution MKL DGEMM: icc -O3 -mkl -mmic dgemm_with_timing.c -o dgemm_with_timing

Dependency information for dgemm_with_timing:

Full path was resolved as:
/home/its-rc/Downloads/test_nkr/test5/src/dgemm_with_timing

Binary was built for Intel(R) Xeon Phi(TM) Coprocessor
(codename: Knights Corner) architecture


 Dependencies Found:
 /opt/intel/compilers_and_libraries_2016.2.101/linux/mkl/lib/intel64_lin_mик/libmkl_intel_lp64.so
 /opt/intel/compilers_and_libraries_2016.2.101/linux/mkl/lib/intel64_lin_mик/libmkl_intel_thread.so
 /opt/intel/compilers_and_libraries_2016.2.101/linux/mkl/lib/intel64_lin_mик/libmkl_core.so
/home/its-rc/Downloads/test_nkr/test5/src/libomp5.so

 Dependencies Not Found Locally (but may exist already on the coprocessor):
 libpthread.so.0
 libc.so.6
 libdl.so.2
 libm.so.6
 libgcc_s.so.1

This example measures performance of Intel(R) MKL function dgemm computing real matrix C=alpha*A*B+beta*C, where A, B, and C
are matrices and alpha and beta are double precision scalars.

Initializing data for matrix multiplication C=A*B for matrix
A(2066x2066) and matrix B(2066x1000).

Allocating memory for matrices aligned on 64-byte boundary for better performance.

Initializing matrix data.

Making the first run of matrix product using Intel(R) MKL dgemm function
via CBLAS interface to get stable run time measurements.

Measuring performance of matrix product using Intel(R) MKL dgemm function
via CBLAS interface.

== Matrix multiplication using Intel(R) MKL dgemm completed ==
== at 3.67674 milliseconds ==

Deallocating memory.

It is highly recommended to define LOOP_COUNT for this example on your computer at 326 to have total execution time about 1 second for reliability of measurements.

Example completed.
TBB: Intel Threading Building Blocks

- Intel TBB library is a template based runtime library for C++ / opensource
- Allows us to fully utilize the scaling capabilities within our code by increasing the number of threads and supporting task oriented load balancing
- icc -mmic -tbb foo.cpp
- As a rule of thumb an application must scale well past one hundred threads on Intel Xeon processors to profit from the possible higher parallel performance offered with the Intel Xeon Phi coprocessor
- To check if the scaling would profit from utilising the highly parallel capabilities of the MIC architecture, you should start to create a simple performance graph with a varying number of threads (from one up to the number of cores)

TBB: Using Natively
• Paralleization with the task construct:
The class tbb:task represents a low-level task with low overhead. It can be used by higher-level constructs like parallel_for or task_group.
• Loop parallelization:
The TBB template parallel_reduce provides the basic implementation of subdividing a range to iterate over into several subranges.
• Vectorization and Cache-Locality
• Work Stealing v/s Work Sharing:
Using work-stealing for task scheduling means that idle threads will take over tasks from other busy threads (there should be a certain amount of tasks per thread available in order to compensate load imbalances).

Work-sharing as method of the task scheduling is worthwhile for well-balanced workloads. Work-sharing is typically implemented as task pool and achieves near optimal occupancy for balanced workload.
IPP: The Intel Integrated Performance Primitives

- Intel Integrated Performance Primitives (IPP) is a software library, which provides a large collection of functions for signal processing and multimedia processing
- Signal processing, including FFT, FIR, Transformations, Convolutions etc
- Image & frame processing, including Transformations, filters, color manipulation
- General functionality, including Statistical functions, Vector, matrix and linear algebra for small matrices
- The functions in this library are optimised to use advanced features of the processors like SSE and AVX instruction sets. Many functions are internally parallelised using OpenMP
- `icc ipptest.cpp -o ipptest -I$IPPROOT/include -L$IPPROOT/lib/intel64 -lippi -lipps -lippcore`
OPENMP / Hybrid

• XEON PHI with a heavy load OPENMP or MPI run
// An OpenMP simple matrix multiply

void doMult(int size, float (* restrict A)[size],
            float (* restrict B)[size], float (* restrict C)[size])
{
    #pragma offload target(mic:MIC_DEV) \
        in(A[0]:size*size) in B[0]:size*size) \
        out(C[0]:size*size)
    
    // Zero the C matrix
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            C[i][j] = 0.0;

    // Compute matrix multiplication.
    #pragma omp parallel for default(none) shared(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];

    int main(int argc, char *argv[])
    {
        if (argc != 4) {
            fprintf(stderr, "Usage: %s size nThreads aIter\n", argv[0]);
            return -1;
        }

        int L, H;
        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);

        // Fill the A and B arrays
        #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);
            }
        }

        // warm up
        doMult(size, A, B, C);
        double startTime, endTime, runtime;
        float time; 
        /*
         * time = time - minTime
         */

        for (i = 0; i < nIter; i++)
        {
            double startTime = dsecnd(1);
            doMult(size, A, B, C);
            double endTime = dsecnd(1);
            double runtime = endTime - startTime;
            minTime = minTime < runtime ? minTime : runtime;
            maxTime = maxTime > runtime ? maxTime : runtime;
            runtime = runtime / nIter;
        }

        printf("Average time: %f
", runtime);
        free(A); free(B); free(C);
    }

    int main(int argc, char *argv[])
    {
        if (argc != 4) {
            fprintf(stderr, "Usage: %s size nThreads aIter\n", argv[0]);
            return -1;
        }

        int L, H;
        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);

        // Fill the A and B arrays
        #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);
            }
        }

        // warm up
        doMult(size, A, B, C);
        double startTime, endTime, runtime;
        float time; 
        /*
         * time = time - minTime
         */

        for (i = 0; i < nIter; i++)
        {
            double startTime = dsecnd(1);
            doMult(size, A, B, C);
            double endTime = dsecnd(1);
            double runtime = endTime - startTime;
            minTime = minTime < runtime ? minTime : runtime;
            maxTime = maxTime > runtime ? maxTime : runtime;
            runtime = runtime / nIter;
        }

        printf("Average time: %f
", runtime);
        free(A); free(B); free(C);
    }

    int main(int argc, char *argv[])
    {
        if (argc != 4) {
            fprintf(stderr, "Usage: %s size nThreads aIter\n", argv[0]);
            return -1;
        }

        int L, H;
        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);

        // Fill the A and B arrays
        #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);
            }
        }

        // warm up
        doMult(size, A, B, C);
        double startTime, endTime, runtime;
        float time; 
        /*
         * time = time - minTime
         */

        for (i = 0; i < nIter; i++)
        {
            double startTime = dsecnd(1);
            doMult(size, A, B, C);
            double endTime = dsecnd(1);
            double runtime = endTime - startTime;
            minTime = minTime < runtime ? minTime : runtime;
            maxTime = maxTime > runtime ? maxTime : runtime;
            runtime = runtime / nIter;
        }

        printf("Average time: %f
", runtime);
        free(A); free(B); free(C);
    }

    int main(int argc, char *argv[])
    {
        if (argc != 4) {
            fprintf(stderr, "Usage: %s size nThreads aIter\n", argv[0]);
            return -1;
        }

        int L, H;
        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);

        // Fill the A and B arrays
        #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);
            }
        }

        // warm up
        doMult(size, A, B, C);
        double startTime, endTime, runtime;
        float time; 
        /*
         * time = time - minTime
         */

        for (i = 0; i < nIter; i++)
        {
            double startTime = dsecnd(1);
            doMult(size, A, B, C);
            double endTime = dsecnd(1);
            double runtime = endTime - startTime;
            minTime = minTime < runtime ? minTime : runtime;
            maxTime = maxTime > runtime ? maxTime : runtime;
            runtime = runtime / nIter;
        }

        printf("Average time: %f
", runtime);
        free(A); free(B); free(C);
    }

    int main(int argc, char *argv[])
    {
        if (argc != 4) {
            fprintf(stderr, "Usage: %s size nThreads aIter\n", argv[0]);
            return -1;
        }

        int L, H;
        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);

        // Fill the A and B arrays
        #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);
            }
        }

        // warm up
        doMult(size, A, B, C);
        double startTime, endTime, runtime;
        float time; 
        /*
         * time = time - minTime
         */

        for (i = 0; i < nIter; i++)
        {
            double startTime = dsecnd(1);
            doMult(size, A, B, C);
            double endTime = dsecnd(1);
            double runtime = endTime - startTime;
            minTime = minTime < runtime ? minTime : runtime;
            maxTime = maxTime > runtime ? maxTime : runtime;
            runtime = runtime / nIter;
        }

        printf("Average time: %f
", runtime);
        free(A); free(B); free(C);
Hello world: rank 0 of 2 running on its-rc-xeon-phi-mic0
Hello world: rank 1 of 2 running on its-rc-xeon-phi-mic0

Dependency information for /tmp/mpi_test.mic

Full path was resolved as
/tmp/mpi_test.mic

Binary was built for Intel(R) Xeon Phi(TM) Coprocessor
(codename: Knights Corner) architecture

SILK_LD_LIBRARY_PATH =

Dependencies Found:
(none found)

Dependencies Not Found Locally (but may exist already on the coprocessor):
libmpi.so.12
libd.so.12
libd.so.2
libt.so.1
libthread.so.0
libm.so.6
libgcc_s.so.1
libc.so.6

[root@its-rc-xeon-phi test9]#
for (i = 0; i < num_procs; i++)
    
    distributed_seed = rand();
    
    buff[i] = distributed_seed;
}

// Broadcast the seed to all processes
MPI_Bcast(buff, num_procs, MPI_INT, MASTER, MPI_COMM_WORLD);

// At this point, every process (including rank 0) has a different seed. Using their seed,
// each process generates n points randomly in the interval [0,1; 1]
startprocess = MPI_Wtime();

rand (buff[i]);

unsigned int point count = 0;
unsigned int rand MAX = 120000;
temp x, y, z;
float temp2, temp3, pi;

double result;
unsigned int inside = 0; total inside = 0;

for (point = 0; point < local_points; point++)
    
    temp = (float) point / num_procs; // id belongs to 0, num_procs
    
    p.x = temp / rand MAX;
    
    temp2 = (float) point / num_procs; // id belongs to 0, num_procs
    
    p.y = temp2 / rand MAX;
    
    temp3 = (float) point / num_procs / rand MAX;
    
    result = p.x * p.x + p.y * p.y - p.z * p.z;

    if (result < 1)
    
        inside++;

    }

double elapsed = MPI_Wtime() - startprocess;

MPI_Reduce((void*) &inside, &total inside, 1, MPI_UNSIGNED, MPI_SUM, MASTER, MPI_COMM_WORLD);

if (id == MASTER)

    printf("%d points into the sphere! id: %d, inside: %d
", inside, id, total inside);

else

    MPI_Send((void*) &inside, 1, MPI_INT, MASTER, TAG HELLO, MPI_COMM_WORLD);

    MPI_Send((void*) &total inside, 1, MPI_INT, MASTER, TAG HELLO, MPI_COMM_WORLD);

    MPI_Send((void*) &total inside, 1, MPI_INT, MASTER, TAG HELLO, MPI_COMM_WORLD);

}

// Rank 0 distributes seed randomly to all processes.
double startprocess, elapsed;

int distributed seed = 0;

int *buff = (int *) malloc(num_procs * sizeof(int));

unsigned int MAX_NUM_POINTS = pow(2, 32) - 1;

unsigned int num local points = MAX_NUM_POINTS / num_procs;

if (id = MASTER)

    rand (time(NULL));

free (buff);

MPI_Finalize();

return 0;
OPENCL on the XEON PHI

OpenCL – Portable Heterogeneous Computing

- OpenCL = Two APIs and Two Kernel languages
  - C Platform Layer API to query, select and initialize compute devices
  - OpenCL C and (soon) OpenCL C++ kernel languages to write parallel code
  - C Runtime API to build and execute kernels across multiple devices
- One code tree can be executed on CPUs, GPUs, DSPs, FPGA and hardware
  - Dynamically balance work across available processors
g++ cmxoptions.cpp gemm.cpp ../common/basic.cpp ../common/cmdparser.cpp ../common/ocobject.cpp -I../common -lOpenCL -oGEMM -std=gnu++0x
../common/ocobject.cpp: In member function 'void OpenCLBasic::createCommandQueue(clCommandQueueProperties)':
../common/ocobject.cpp:439: warning: 'cl command queue = clCreateCommandQueue(cl context*, cl device id*, cl command queue properties, cl int*)' is deprecated (declared at /usr/include/CL/cl.h:1359)
../common/ocobject.cpp:439: warning: 'cl command queue = clCreateCommandQueue(cl context*, cl_device_id*, cl_command_queue_properties, cl_int*)' is deprecated (declared at /usr/include/CL/cl.h:1359)

[0] Intel(R) OpenCL [Selected]

[0] Intel(R) Many Integrated Core Acceleration Card [Selected]

[1] Intel(R) Xeon(R) CPU E5-2680 8 @ 2.70GHz

Build program options: "-DTILE=8 -DTILE_SIZE=4 -DTILE_GROUP_N=10 -DTILE_SIZE_N=128 -DTILE_GROUP_N=1 -DTILE_SIZE_K=8"

Running gemm on kernel with matrix size: 2900x3000

Memory row stride to ensure necessary alignment: 15872 bytes

Size of memory region for one matrix: 62980896 bytes

Using alpha = 0.37599 and beta = 0.872412

Host time: 0.355203 sec.
Host perf: 351.866 GFLOPS
Host time: 0.258869 sec.
Host perf: 469.101 GFLOPS
Host time: 0.257623 sec.
Host perf: 485.143 GFLOPS
Host time: 0.251048 sec.
Host perf: 496.871 GFLOPS
Host time: 0.257487 sec.
Host perf: 485.399 GFLOPS
Host time: 0.257053 sec.
Host perf: 484.523 GFLOPS
Host time: 0.255225 sec.
Host perf: 489.781 GFLOPS
Host time: 0.257576 sec.
Host perf: 485.231 GFLOPS
Host time: 0.258809 sec.
Host perf: 404.305 GFLOPS
Host time: 0.257841 sec.
Host perf: 404.733 GFLOPS

[root@its-rc-xeon-phi GEMM]# pwd
/home/its-rc/Downloads/test/nkr/test10/GEMM

[root@its-rc-xeon-phi GEMM]# 
Mapping the OpenCL constructs to Intel Xeon Phi coprocessor

- At initialization time, the OpenCL driver creates 240 SW threads and pins them to the HW threads (for a 60-core configuration).
- Then, following a clEnqueueNDRange() call, the driver schedules the work groups (WG) of the current NDRRange on the 240 threads.
- A WG is the smallest task being scheduled on the threads - calling clEnqueueNDRange() with less than 240 WGs, leaves the coprocessor underutilized.
- The OpenCL compiler creates an optimized routine that executes a WG. This routine is built from up to three nested loops, as shown in the following pseudo code:
  __Kernel ABC(…)
  For (int i = 0; i < get_local_size(2); i++)
    For (int j = 0; j < get_local_size(1); j++)
      For (int k = 0; k < get_local_size(0); k++)
        Kernel_Body;
- Note that the innermost loop is used for dimension zero of the NDRRange. This directly impacts the access pattern of your performance critical code. It also impacts the implicit vectorization efficiency.
- The OpenCL compiler implicitly vectorizes the WG routine based on dimension zero loop, i.e., the dimension zero loop is unrolled by the vector size. So the WG code with vectorization looks like:
  __Kernel ABC(…)
  For (int i = 0; i < get_local_size(2); i++)
    For (int j = 0; j < get_local_size(1); j++)
      For (int k = 0; k < get_local_size(0); k += VECTOR_SIZE)
        Vector_Kernel_Body;
- The vector size of Intel Xeon Phi coprocessor is 16, regardless of the data types used in the kernel.
Debugging

- The GNU debugger (gdb) has been enabled by Intel to support the Intel Xeon Phi coprocessor. The debugger is now part of the recent MPSS release and does not have to be downloaded separately any more.
- There are 2 different modes of debugging supported: native debugging on the coprocessor or remote cross-debugging on the host.

Native debugging with gdb

- Run gdb on the coprocessor
  
  ssh -t mic0 /usr/bin/gdb

- One can then attach to a running application with process ID pid via
  
  (gdb) attach pid

- or alternatively start an application from within gdb via
  
  (gdb) file /path/to/application
  (gdb) start

Remote debugging with gdb

- Run the special gdb version with Xeon Phi support on the host
  
  /usr/linux-kлом-4.7/bin/x86_64-kлом-linux-gdb

- Start the gdbserver on the coprocessor by typing on the host gdb
  
  (gdb) target extended-remote| ssh -T mic0 gdbserver -multi -

- Attach to a remotely running application with the remote process ID pid
  
  (gdb) file /local/path/to/application
  (gdb) attach pid

- It is also possible to run an application directly from the host gdb
  
  (gdb) file /local/path/to/application
  (gdb) set remote exec-file /remote/path/to/application
TUNING

• A single Xeon Phi core is slower than a Xeon core due to lower clock frequency, smaller caches and lack of sophisticated features such as out-of-order execution and branch prediction.
• To fully exploit the processing power of a Xeon Phi, parallelism on both instruction level (SIMD) and thread level (OpenMP) is needed.

Single core optimization

Memory alignment

• Xeon Phi can only perform memory reads/writes on 64-byte aligned data.
• Any unaligned data will fetched and stored by performing a masked unpack or pack operation on the first and last unaligned bytes. This may cause performance degradation, especially if the data to be operated on is small in size and mostly unaligned
• Most common ways to let the compiler to align the data or to assume that the data has been aligned

<table>
<thead>
<tr>
<th>C/C++</th>
<th>n/a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Align arrays: \texttt{align array64byte}</td>
</tr>
<tr>
<td>Compiler directives for alignment</td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>Align variable var:</td>
</tr>
<tr>
<td>float var[100] _attribute_((aligned(64)))</td>
<td></td>
</tr>
<tr>
<td>Inform the compiler of the alignment of variable var:</td>
<td></td>
</tr>
<tr>
<td>_assume_aligned(var, 64)</td>
<td></td>
</tr>
<tr>
<td>Declare a loop to be aligned:</td>
<td></td>
</tr>
<tr>
<td>#pragma vector aligned</td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td>Align variable var:</td>
</tr>
<tr>
<td>real var(100)</td>
<td></td>
</tr>
<tr>
<td>$ \texttt{attributes align:64::var}</td>
<td></td>
</tr>
<tr>
<td>Inform the compiler of the alignment of variable var:</td>
<td></td>
</tr>
<tr>
<td>$ assume_aligned var:64</td>
<td></td>
</tr>
<tr>
<td>Declare a loop to operate on aligned data:</td>
<td></td>
</tr>
<tr>
<td>$ vector aligned</td>
<td></td>
</tr>
</tbody>
</table>

Allocation of aligned dynamic memory

C/C++

Use \_mm\_malloc() and \_mm\_free() to allocate and free memory. These take the desired byte-alignment as a second input argument. When using a aligned variable var, use \_assume\_aligned(var, 64) to inform the compiler about the alignment.

Fortran

Use \_align\_array64byte compiler flag to enforce aligned heap memory allocation.
SIMD optimization

- Each Xeon Phi core has a 512-bit VPU unit which is capable of performing 16SP flops or 8 DP flops per clock cycle.
- VPU units are also capable of Fused Multiply-Add (FMA) or Fused Multiply-Subtract (FMS) operations which effectively double the theoretical floating point performance.
- Intel compilers have several directives to aid vectorization of loops. These are listed in the following in short.

Let the compiler know there are no loop carried dependencies, but only affect compiler heuristics.

C/C++ #pragma IVDEP
Fortran !DIR$ IVDEP

Let the compiler know the loop should be vectorized, but only affect compiler heuristics.

C/C++ #pragma VECTOR
Fortran !DIR$ VECTOR

Force the compiler vectorize a loop, independent of heuristics.

C/C++ #pragma SIMD or #pragma vector always
Fortran !DIR$ SIMD or !DIR$ VECTOR ALWAYS

In order to aid vectorization, Intel compilers can report details on whether vectorization is successful or not. The reports can be generated with -vec-reportN compiler flag, where N denotes the report level. The vector report levels are:

Level Description

N=0 No diagnostic information.
N=1 Report vectorized loops.
N=2 Report vectorized and non-vectorized loops.
N=3 Report vectorized and non-vectorized loops with any proven or assumed data dependencies.
N=4 Report non-vectorized loops.
N=5 Report non-vectorized loops with a reason why they were not vectorized.
N=6 Use greater detail on reporting vectorized and non-vectorized loops with any proven or assumed data dependencies.
N=7 Very detailed report, not intended to be human readable. Python script for gathering information and annotating the vector report with source code available from Intel.
OpenMP optimization

OpenMP thread affinity

- Each Xeon Phi card has a shared-memory environment with ~ 60 physical cores - divided into 4 logical cores each - node topology.
- Each memory bank resides closer to some of the cores in the topology and therefore access to data laying in a memory bank attached to another socket is generally more expensive.
- Non-uniform memory access (NUMA) can create performance issues if threads are allowed to migrate from one logical core to another during their execution.
- In order to extract maximum performance, consider binding OpenMP threads to logical and physical cores across different sockets on a single Xeon Phi card. The layout of this binding in respect to the node topology has performance implications depending on the computational task and is referred as thread affinity.
- The thread affinity interface of the Intel runtime library can be controlled by using the KMP_AFFINITY environment variable or by using a proprietary Intel API.

```
KMP_AFFINITY=[modifier,...]<type>[,<permute>][,[<offset>]]
```

**modifier**
- default=noverbose, respect, granularity=core
  - granularity={fine, thread, core}, norespect, noverbose, nowarnings,
    - proc_list={<proc-list>}, respect, verbose, warnings.

**type**
- default=noverbose, respect, granularity=core
  - balanced, compact, disabled, explicit, none, scatter

In most cases it is sufficient only to specify the affinity and granularity. The most important affinity types supported by Intel Xeon Phi are:

- **balanced**
  - Thread affinity balanced is a mixture of scatter and compact affinities. Threads from <1> to <n_p> will be spread across the topology as evenly as possible in the granularity context, where <n_p> denotes the number of physical cores. For thread <k> from threads <n_p>=1 to <n> will be assigned as close as possible to thread <k+1>.

- **compact**
  - Thread <k+1> will be assigned as close as possible to thread <k> in the granularity context according to which the threads are placed.

- **none**
  - Threads are not bound to any contexts. Use of affinity none is not recommended in general.

- **scatter**
  - Threads from <1> to <n> will be spread across the topology as evenly as possibly in the granularity context according to which the threads are placed.

The most important granularity types supported by Intel Xeon Phi are:

- **core**
  - Threads are bound to a single core, but allowed to float within the context of a physical core.

- **fine/thread**
  - Threads are bound to a single context, i.e., a logical core.
Other considerations:

- **OpenMP thread placement**: KMP_PLACE_THREADS defines the node topology for KMP_AFFINITY
- **Multiple parallel regions and barriers**: Whenever an OpenMP parallel region is encountered, a team of threads is formed and launched to execute the computations. Whenever the parallel region is ended, threads are joined and the computation proceeds with a single thread (FORK JOIN). Between different parallel regions it is up to the OpenMP implementation to decide whether the threads are shut down or left in an idle state. Intel OpenMP -library leaves the threads in a running state for a predefined amount of time before setting them to sleep. The time is defined by KMP_BLOCKTIME and KMP_LIBRARY environment variables.
- **False sharing**: Let $v$ be a vector with real entries and size $n=1E+08$. Let $f(x)$ denote a function which counts the number of entries in $v$ which are smaller than zero.

In the first implementation, each thread counts the number of negative entries it has found in $v$ to a globally shared array. To avoid race conditions, each thread uses its own entry in the shared array, uniquely determined by thread id. When a thread has finished its portion of vector, a global counter is atomically incremented.

The second implementation is practically equivalent to the first one, except that each thread has its own private array for counting the data.

- **Memory limitations**: Available memory per core on Xeon Phi is very limited. When an application is run using all the available threads, approximately 30Mb of memory is available per thread assuming none of the data is shared. Excessive memory allocation per thread is therefore highly discouraged. Care should be also taken when assigning private variables in order to avoid unnecessary data duplication among threads.
OPENMP NESTED PARALLELISM

- Enabling OpenMP nested parallelism is done by setting environment variable OMP_NESTED=TRUE or with an API call to omp_set_nested function.
- The number of nested threads within each OpenMP parallel region is done by setting the environment variable OMP_NUM_THREADS=n_1, n_2, n_3,..., where n_j refers to the number of threads on the jth level.
- The number or threads within each nesting level can be also set with an API call to omp_set_num_threads function.

OPENMP LOAD BALANCING

Different schedule kinds supported by OpenMP runtime on a Xeon Phi are

<table>
<thead>
<tr>
<th>Schedule Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>With the static scheduling policy, iteration indices are divided into chunks of chunk_size and distributed to threads in a round-robin fashion. If chunk_size is not defined, iteration indices are divided into chunks that are roughly equal in size.</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>With the dynamic scheduling policy, iteration indices are assigned to threads in chunks of chunk_size. Threads request and process new chunks with assigned iteration indices until the whole index range has been processed.</td>
</tr>
<tr>
<td>GUIDED</td>
<td>With the guided scheduling policy, iteration indices are assigned to threads in chunks of size chunk_size at minimum. In the beginning of the iteration, the chunk_size actually assigned to be processed is proportional to the number of unassigned iteration indices versus the number of available threads. The assigned chunk_size and can be larger than the minimum.</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>The scheduling policy and chunk size will be decided at runtime based on the OMP_SCHEDULE environment variable.</td>
</tr>
</tbody>
</table>
Scalasca

- Scalasca is a scalable automatic performance analysis toolset designed to profile large scale parallel Fortran, C and C++ applications that use MPI, OpenMP and hybrid MPI+OpenMP parallelization models.
- It is portable on Intel Xeon Phi architecture in native, symmetric and offload models. Version 2.x uses the Score-P instrumenter and measurement libraries.

Instrumentation of the code to be profiled is done with the command $skim mpiifort -O -openmp *.f$

This produces the instrumented executable that will be executed on the host, while

$skim mpiifort -O -openmp -mmic *.f$

produces an executable for the coprocessor. Further, measurement is then performed with the $scan$ command:

$scan mpiexec -n 2 a.out.cpu // on the host$

% $scan mpiexec -n 1 a.out.cpu // on the device$

It can also be launched on more than one node on the host:

$scan mpiexec.hydra -host node0 -n 1 a.out.cpu : -host node1 -n 1 a.out.cpu$

and on more than one coprocessor, if available:

$scan mpiexec.hydra -host mic0 -n 30 a.out.mic : -host mic1 -n 31 a.out.mic$

For symmetric execution one can use:

$scan mpiexec.hydra -host node0 -n 2 a.out.cpu : -host mic0 -n 61 a.out.mic$

Finally, the collected data can be analyzed with the $square$ command. The $scan$ output would look something like $epik_a_2x16_sum$ for the runs performed on the host and $epik_a_mic61x4_sum$ for those performed on the coprocessor. For data collected on the host and on the device we have:

$square epik_a_2x16_sum$

$square epik_a_61x4_sum$

respectively. To analyze the data collected in a run from a symmetric execution type:

$square epik_a_2x16+mic61x4_sum$. 
SOURCE CODE IN THIS PRESENTATION CAN BE DOWNLOADED FROM
THANK YOU – NILAY ROY