Intel Xeon Phi: Programming Models
Advantages of the MIC Architecture

- Retains programmability and flexibility of standard x86 architecture.
- No need to learn a new complicated language like CUDA or OpenCL.
- Offers possibilities we always missed on GPUs: Login onto the system, watching and controlling processes via *top, kill* etc. like on a Linux host.
- Allows many different parallel programming models like OpenMP, MPI, Intel Cilk and Intel Threading Building Blocks.
- Offers standard math-libraries like *Intel MKL*.
- Supports whole Intel tool chain, e.g. Intel C/C++ and Fortran Compiler, Debugger & Intel VTune Amplifier.
MIC Programming Models

- Xeon centric
  - Xeon hosted
  - Offload
  - Symmetric
  - MIC hosted

- General purpose serial & parallel computing
- Codes with highly parallel parts
- Codes with balanced needs
- Highly parallel codes

```
main()
foo()
MPI_*()
```

```
main()
foo()
MPI_*()
```

```
main()
foo()
MPI_*()
```

```
main()
foo()
MPI_*()
```

```
foo()
```

```
main()
foo()
MPI_*()
```

```
main()
foo()
MPI_*()
```

```
main()
foo()
MPI_*()
```

27-29 June 2016

Intel MIC Programming Workshop
MIC Programming Models

Parallelisation
- Intel Cilk Plus
- OpenMP
- OpenACC
- MPI
- Intel Threading Building Blocks
- OpenCL
- Pthreads

Ease of use
- MKL
- Auto Vectorisation
- OpenMP 4.0 (#pragma omp simd)
- Vectorisation Pragmas (#pragma vector, ivdep, simd,...)
- Intel Cilk Plus
- Array Notation, Elemental Functions
- C/C++ Vector Classes (F32vec16, F64vec8)

Vectorisation
- Intrinsic/Assembler

Fine control

27-29 June 2016
Intel MIC Programming Workshop
Programming Modes

- Native Mode
  - Programs started on Xeon Phi.
  - Cross-compilation using `--mmic`.
  - User access to Xeon Phi necessary.
  - Necessary to support MPI ranks on Xeon Phi.

- Offload (Accelerator) Mode
  - Programs started on the host.
  - Intel Pragmas to offload code to Xeon Phi.
  - OpenMP possible, but no MPI ranks on Xeon Phi.
  - No user access to Xeon Phi needed.
  - No input data files on Xeon Phi possible.
Offload Modes

- Host and MIC do not share physical or virtual memory in hardware.
- 2 Offload data transfer models are available:
  1. Explicit copy: Language Extensions for Offload (LEO)
     - Syntax: pragma/directive based
     - offload directive specifies variables that need to be copied between host and MIC
     - Example:
       - C: #pragma offload target(mic) in(data:length(size))
       - Fortran: !DIR$ offload target(mic) in(data:length(size))
  2. Implicit Copy: MYO
     - Syntax: keyword extension based
     - shared variables need to be declared, same variables can be used on the host and MIC, runtime automatically maintains coherence
     - Example:
       - C: _Cilk_shared double a; _Cilk_offload func(a);
       - Fortran: not supported
Programming Languages / Libraries

- **OpenMP**
  - Native execution on MIC (cross-compilation with \texttt{--mmic})
  - Execution on host, using offload pragmas / directives to offload code at runtime

- **MPI (and hybrid MPI & OpenMP)**
  - Co-processor only MPI programming model: native execution on MIC using \texttt{mpiexec.hydra} on MIC.
  - Symmetric MPI programming model: MPI ranks on MICs and host CPUs.

- **MKL**
  - Native execution on MIC (compilation with \texttt{--mkl -mmic}).
  - Compiler assisted offload.
  - Automatic Offload (AO): automatically uses both host and MIC, transparent and automatic data transfer and execution management (compilation with \texttt{--mkl, mkl_mic_enable()}/ \texttt{MKL_MIC_ENABLE=1}).
### Distributed vs. Shared Memory

#### Distributed Memory
- **Same program** on each processor/machine (SPMD) or **Multiple programs** with consistent communication structure (MPMD)
- **Program written in a sequential language**
  - all variables process-local
  - no implicit knowledge of data on other processors
- **Data exchange between processes**:  
  - send/receive messages via appropriate library
  - most tedious, but also the most flexible way of parallelization
- **Parallel library discussed here:**  
  - Message Passing Interface, **MPI**

#### Shared Memory
- **Single Program on single machine**
  - UNIX Process splits off **threads**, mapped to CPUs for work distribution
- **Data**
  - may be process-global or thread-local
  - exchange of data not needed, or via suitable synchronization mechanisms
- **Programming models**
  - explicit threading (hard)
  - directive-based threading via **OpenMP** (easier)
  - automatic parallelization (very easy, but mostly not efficient)
MPI vs. OpenMP

- **MPI standard**
  - MPI forum released version 2.2 in September 2009
  - MPI version 3.1 in June 2015
  - unified document ("MPI1+2")

- **Base languages**
  - Fortran (77, 95)
  - C
  - C++ binding obsolescent → use C bindings

- **Resources:**
  - http://www.mpi-forum.org

- **OpenMP standard**
  - OpenMP 3.1 (July 2011) released by architecture review board (ARB)
    - feature update (tasking etc.)
  - OpenMP 4.0 (July 2013)
    - SIMD, affinity policies, accelerator support
  - OpenMP 4.5 (Nov 2015)

- **Base languages**
  - Fortran (77, 95)
  - C, C++
    (Java is not a base language)

- **Resources:**
  - http://www.openmp.org
  - http://www.compunity.org
OpenMP Standard

Number of Pages in OpenMP Standard

<table>
<thead>
<tr>
<th>Version</th>
<th>Standard</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 1.0 (1997)</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>Fortran 1.1 (1999)</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>C/C++ 1.0 (1998)</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>Fortran 2.0 (2000)</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>C/C++ 2.0 (2002)</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>3.0 (2008)</td>
<td>318</td>
<td></td>
</tr>
<tr>
<td>3.1 (2011)</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>4.0 (2013)</td>
<td>226</td>
<td>251</td>
</tr>
<tr>
<td>4.0 (2013) + Examples 4.0.1 (2014)</td>
<td>312</td>
<td></td>
</tr>
<tr>
<td>4.0 (2013) + Examples 4.0.2 (2015)</td>
<td>312</td>
<td></td>
</tr>
</tbody>
</table>
Complexity of Recent Standards

![Comparison of Number of Pages in Recent Standards](chart)

- MPI 3.1 (2015)
- OpenMP 4.0 (2013) + Exam.: 4.0.2 (2015)
- OpenACC 2.0 (2013)
- OmpSS (Draft)

27-29 June 2016 Intel MIC Programming Workshop
Productivity Analysis Himeno benchmark

Mont-Blanc Project, D3.6

Performance vs. Lines of Code

- Xeon C static
- Xeon C dynamic
- Xeon F77
- Xeon OpenMP F90
- Xeon OpenMP C, 16 threads
- OpenCL on Tesla M2090
- Xeon MPI C, 16 nodes
- OpenCL + OMPSS en Tesla M2090
- Xeon F90
- OpenCL + OMPSS + MPI on 1 Tesla M2090, 1 node

27-29 June 2016
Intel MIC Programming Workshop
#include <omp.h>

int main() {
    int numth = 1;
    #pragma omp parallel
    {
        int myth = 0; /* private */
        #pragma omp single
            numth = omp_get_num_threads();
        /* block above: one statement */
        myth = omp_get_thread_num();

        printf("Hello from %i of %i\n",\n               myth,numth);
    } /* end parallel */
}

icc -openmp helloopenmp.c
Simple OpenMP Program

lu65fok@login12:~/.mickurs> export OMP_NUM_THREADS=10
lu65fok@login12:~/.mickurs> ./helloopenmp
Hello from 5 of 10
Hello from 2 of 10
Hello from 6 of 10
Hello from 0 of 10
Hello from 8 of 10
Hello from 3 of 10
Hello from 4 of 10
Hello from 9 of 10
Hello from 7 of 10
Hello from 1 of 10
/* C Example */
#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[])
{
    int rank, size;

    MPI_Init (&argc, &argv);    /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);    /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */
    printf("Hello from %i of %i\n", rank, size);
    MPI_Finalize();
    return 0;
}

mpiicc hellompi.c
Simplest MPI Program

```
lu65fok@login12:~/mickurs> mpiicc hellompi.c -o hellompi

lu65fok@login12:~/mickurs> mpirun -n 10 ./hellompi
Hello from 5 of 10
Hello from 6 of 10
Hello from 7 of 10
Hello from 8 of 10
Hello from 9 of 10
Hello from 0 of 10
Hello from 1 of 10
Hello from 2 of 10
Hello from 3 of 10
Hello from 4 of 10
```
Intel Xeon Phi Programming Models: Explicit Offload Model
Intel Offload Directives

- Syntax:
  - **C:**
    ```
    #pragma offload target(mic) <clauses>
    <statement block>
    ```
  - **Fortran:**
    - `!DIR$ offload target(mic) <clauses>`
    - `<statement>`
    - `!DIR$ omp offload target(mic) <clauses>`
    - `<OpenMP construct>`
Intel Offload Directive

- **C:**
  - Pragma can be before any statement, including a compound statement or an OpenMP parallel pragma

- **Fortran:**
  - If OMP is specified: the next line, other than a comment, must be an OpenMP PARALLEL, PARALLEL SECTIONS, or PARALLEL DO directive.
  - If OMP is not specified, next line must:
    - An OpenMP* PARALLEL, PARALLEL SECTIONS, or PARALLEL DO directive
    - A CALL statement
    - An assignment statement where the right side only calls a function
Intel Offload Directive

- Offloading a code block in Fortran:

```fortran
!DIR$ offload begin target(MIC)
...
!DIR$ end offload
```

Code block can include any number of Fortran statements, including DO, CALL and any assignments, but not OpenMP directives.
Intel Offload

- Implements the following steps:

1. Memory allocation on the MIC
2. Data transfer from the host to the MIC
3. Execution on the MIC
4. Data transfer from the MIC to the host
5. Memory deallocation on MIC
#include <stdio.h>
int main (int argc, char* argv[]) {

    #pragma offload target(mic)
    {
        printf("MIC: Hello world from MIC.\n");
    }

    printf("Host: Hello world from host.\n");
}

Must be in a new line!
PROGRAM HelloWorld

!DIR$ offload begin target(MIC)
PRINT *,'MIC: Hello world from MIC'
!DIR$ end offload

PRINT *,'Host: Hello world from host'
END
lu65fok@login12:~/tests> icpc offload1.c -o offload1

lu65fok@login12:~/tests> ./offload1
offload error: cannot offload to MIC - device is not available

lu65fok@i01r13c01:~/tests> ./offload1
Host: Hello world from host.
MIC: Hello world from MIC.
Intel Offload: Hello World in Fortran

lu65fok@login12:~/tests> ifort offload1.f90 -o offload1

lu65fok@login12:~/tests> ./offload1
offload error: cannot offload to MIC - device is not available

lu65fok@i01r13c01:~/tests> ./offload1
Host: Hello world from host.
MIC: Hello world from MIC.
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[]) {
    char hostname[100];
    gethostname(hostname, sizeof(hostname));

    #pragma offload target(mic)
    {
        char michostname[100];
        gethostname(michostname, sizeof(michostname));
        printf("MIC: Hello world from MIC. I am %s and I have %ld logical cores. I was called from host: %s \n", michostname, sysconf(_SC_NPROCESSORS_ONLN), hostname);
    }
}
Intel Offload: Hello World with Hostnames

lu65fok@login12:~/tests> icpc offload.c -o offload

lu65fok@i01r13c01:~/tests> ./offload
Host: Hello world from host. I am i01r13c01 and I have 32 logical cores.
MIC: Hello world from MIC. I am i01r13c01-mic0 and I have 240 logical cores. I was called from host: i01r13c01
lu65fok@login12:~/tests> icpc -offload=optional offload.c -o offload

lu65fok@login12:~/tests> ./offload
MIC: Hello world from MIC. I am login12 and I have 16 logical cores. I was called from host: login12
Host: Hello world from host. I am login12 and I have 16 logical cores.

lu65fok@login12:~/tests> icpc -offload=mandatory offload.c -o offload
lu65fok@login12:~/tests> ./offload
offload error: cannot offload to MIC - device is not available
lu65fok@login12:~/tests> icpc -offload=none offload.c -o offload
offload.c(13): warning #161: unrecognized #pragma
   #pragma offload target(mic)
  ^

lu65fok@login12:~/tests>

lu65fok@i01r13c01:~/tests> ./offload
MIC: Hello world from MIC. I am i01r13c01 and I have 32 logical cores.
I was called from host: i01r13c01
Host: Hello world from host. I am i01r13c01 and I have 32 logical cores.
#include <stdio.h>
#include <stdlib.h>

int main(){
    #pragma offload target (mic)
    {
        system("command");
    }
}
lu65fok@i01r13c01:~/tests> ./system
BASH=/bin/sh
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_EXECUTION_STRING=set
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSION='4.2.10(1)-release'
COI_LOG_PORT=65535
COI_SCIF_SOURCE_NODE=0
DIRSTACK=()
ENV_PREFIX=MIC
EUID=400
GROUPS=()
HOSTNAME=i01r13c01-mic0
HOSTTYPE=k1om
IFS=''

MACHTYPE=k1om-mpss-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/usr/bin:/bin
POSIXLY_CORRECT=y
PPID=37141
PS4='+'
PWD=/var/volatile/tmp/coi_procs/1/37141
SHELL=/bin/false
SHELLOPTS=braceexpand:hashall:interactive-comments:posix
SHLVL=1
TERM=dumb
UID=400
_=sh
#pragma offload target (mic)
{
    system("hostname");
    system("uname -a");
    system("whoami");
    system("id");
}

lu65fok@i01r13c01:~/tests> ./system
i01r13c01-mic0
Linux i01r13c01-mic0 2.6.38.8+mpss3.1.2 #1 SMP Wed Dec 18 19:09:36 PST 2013 k1om GNU/Linux
micuser
uid=400(micuser) gid=400(micuser)
Offload: Using several MIC Coprocessors

- To query the number of coprocessors:
  ```c
  int nmics = __Offload_number_of_devices()
  ```

- To specify which coprocessor n< nmics should do the computation:
  ```c
  #pragma offload target(mic:n)
  ```

- If (n > nmics) then coprocessor (n % nmics) is used

- Important for:
  - Asynchronous offloads
  - Coprocessor-Persistent data
Offloading OpenMP Computations

- C/C++ & OpenMP:
  
  ```
  #pragma offload target(mic)
  #pragma omp parallel for
  for (int i=0; i<n; i++) {
    a[i] = c*b[i] + d;
  }
  ```

- Fortran & OpenMP

  ```
  !$DIR$ offload target(mic)
  !$OMP PARALLEL DO
  do i = 1, n
    a(i) = c*b(i) + d
  end do
  !$omp END PARALLEL DO
  ```
Functions and variables on the MIC

- **C:**
  - `__attribute__((target(mic)))` variables / function
  - `__declspec(target(mic))` variables / function
  - `#pragma offload_attribute(push, target(mic))`
  - ... multiple lines with variables / functions
  - `#pragma offload_attribute(pop)`

- **Fortran:**
  - `!DIR$ attributes offload:mic::: variables / function`
Functions and variables on the MIC

```c
#pragma offload_attribute(push,target(mic))
const int n=100;
int a[n], b[n], c, d;
void myfunction(int* a, int*b, int c, int d){
    for (int i=0;i<n;i++) {
        a[i]=c*b[i]+d;
    }
}
#pragma offload_attribute(pop)

int main (int argc, char* argv[]) {
    #pragma offload target(mic)
    {
        myfunction(a, b, c, d);
    }
```
**Intel Offload Clauses**

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple coprocessors</td>
<td><code>target(mic[:unit] )</code></td>
<td>Select specific coprocessors</td>
</tr>
<tr>
<td>Conditional offload</td>
<td><code>if (condition) / manadatory</code></td>
<td>Select coprocessor or host compute</td>
</tr>
<tr>
<td>Inputs</td>
<td><code>in(var-list modifiers_{opt})</code></td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>Outputs</td>
<td><code>out(var-list modifiers_{opt})</code></td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td><code>inout(var-list modifiers_{opt})</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_{opt})</code></td>
<td>Data is local to target</td>
</tr>
<tr>
<td>Async. Offload</td>
<td><code>signal(signal-slot)</code></td>
<td>Trigger asynchronous Offload</td>
</tr>
<tr>
<td>Async. Offload</td>
<td><code>wait(signal-slot)</code></td>
<td>Wait for completion</td>
</tr>
</tbody>
</table>
# Intel Offload Modifier Options

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify copy length</td>
<td>length(N)</td>
<td>Copy N elements of pointer’s type</td>
</tr>
<tr>
<td>Coprocessor memory allocation</td>
<td>alloc_if (bool)</td>
<td>Allocate coprocessor space on this offload (default: TRUE)</td>
</tr>
<tr>
<td>Coprocessor memory release</td>
<td>free_if (bool)</td>
<td>Free coprocessor space at the end of this offload (default: TRUE)</td>
</tr>
<tr>
<td>Array partial allocation &amp; variable relocation</td>
<td>alloc (array-slice) in ( var-expr )</td>
<td>Enables partial array allocation and data copy into other vars &amp; ranges</td>
</tr>
</tbody>
</table>
Intel Offload: Data movement

- #pragma offload target(mic) in(in1,in2,…) out(out1,out2,…) inout(inout1,inout2,…)

- **At Offload start:**
  - Allocate Memory Space on MIC for all variables
  - Transfer in/inout variables from Host to MIC

- **At Offload end:**
  - Transfer out/inout variables from MIC to Host
  - Deallocate Memory Space on MIC for all variables
Intel Offload: Data movement

- `data = (double*)malloc(n*sizeof(double));`
- `#pragma offload target(mic) in(data:length(n))`

- **Copies n doubles** to the coprocessor, not `n * sizeof(double)` Bytes.
- **ditto for out() and inout()**
Allocation of Partial Arrays in C

- int n=1000
- data = (double*)malloc(n*sizeof(double));
- #pragma offload target(mic) in(data[100:200] : alloc(data[300:400])

Host:
- 1000 doubles allocated
- First element has index 0
- Last element has index 999

MIC:
- 400 doubles are allocated
- First element has index 300
- Last element has index 699
- 200 elements in the range data[100], ..., data[299] are copied to the MIC
Allocation of Partial Arrays in Fortran

- integer :: n=1000
- double precision, allocatable :: data(:)
- allocate(data(n) )
- !C: #pragma offload target(mic) in(data[100:200] : alloc(data[300:400]))
- !DIR$ offload target(mic) in(data(100:299) : alloc(data(300:699)))
- **Host:**
  - 1000 doubles allocated
  - First element has index 0
  - Last element has index 999
- **MIC:**
  - 400 doubles are allocated
  - First element has index 300
  - Last element has index 699
  - 200 elements in the range data[100], …, data[299] are copied to the MIC
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
{
    #pragma omp parallel for
    for( i = 0; i < n; i++ ) {
        for( k = 0; k < n; k++ ) {
            #pragma vector aligned
            #pragma ivdep
            for( j = 0; j < n; j++ ) {
                //c[i][j] = c[i][j] + a[i][k]*b[k][j];
                c[i*n+j] = c[i*n+j] + a[i*n+k]*b[k*n+j];
            }
        }
    }
}
Vectorisation Diagnostics

lu65fok@login12:/tests> icc -vec-report2 -openmp offloadmul.c -ooffloadmul
offloadmul.c(35): (col. 5) remark: LOOP WAS VECTORIZED
offloadmul.c(32): (col. 3) remark: loop was not vectorized: not inner loop
offloadmul.c(57): (col. 2) remark: LOOP WAS VECTORIZED
offloadmul.c(54): (col. 7) remark: loop was not vectorized: not inner loop
offloadmul.c(53): (col. 5) remark: loop was not vectorized: not inner loop
offloadmul.c(8): (col. 9) remark: loop was not vectorized: existence of vector dependence
offloadmul.c(7): (col. 5) remark: loop was not vectorized: not inner loop
offloadmul.c(57): (col. 2) remark: *MIC* LOOP WAS VECTORIZED
offloadmul.c(54): (col. 7) remark: *MIC* loop was not vectorized: not inner loop
offloadmul.c(53): (col. 5) remark: *MIC* loop was not vectorized: not inner loop

__attribute__((target(mic)))) void mxm( int n, double * restrict a, double * restrict b, double *restrict c ){

    int i,j,k;
    for( i = 0; i < n; i++ ) {
        ...
    }
}

main(){
    ...
    #pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n))
    {
        mxm(n,a,b,c);
    }
}
u65fok@i01r13c06:~tests> export OFFLOAD_REPORT=2

lu65fok@i01r13c06:~tests> ./offloadmul

[Offload] [MIC 0] [File] offloadmul.c
[Offload] [MIC 0] [Line] 50
[Offload] [MIC 0] [Tag] Tag 0
[Offload] [HOST] [Tag 0] [CPU Time] 51.927456(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data] 24000016 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time] 50.835065(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data] 8000016 (bytes)
Offload Diagnostics

lu65fok@i01r13c06:~/tests> export H_TRACE=1

lu65fok@i01r13c06:~/tests> ./offloadmul
HOST: Offload function
__offload_entry_offloadmul_c_50mainicc638762473Jnx4JU, is_empty=0, #varDescs=7, #waits=0, signal=none
HOST: Total pointer data sent to target: [24000000] bytes
HOST: Total copyin data sent to target: [16] bytes
HOST: Total pointer data received from target: [8000000] bytes
MIC0: Total copyin data received from host: [16] bytes
MIC0: Total copyout data sent to host: [16] bytes
HOST: Total copyout data received from target: [16] bytes
lu65fok@i01r13c06:~/tests>
lu65fok@i01r13c06:~/tests> export H_TIME=1

lu65fok@i01r13c06:~/tests> ./offloadmul
[Offload] [MIC 0] [File] offloadmul.c
[Offload] [MIC 0] [Line] 50
[Offload] [MIC 0] [Tag] Tag 0
[Offload] [HOST] [Tag 0] [CPU Time] 51.920016(seconds)
[Offload] [MIC 0] [Tag 0] [MIC Time] 50.831497(seconds)

*******************************************************************************
timer data       (sec)
*******************************************************************************

27-29 June 2016 Intel MIC Programming Workshop
Environment Variables

- Host environment variables are automatically forwarded to the coprocessor when offload mode is used.
- To avoid names collisions:
  - Set `MIC_ENVIRONMENT_PREFIX=MIC` on the host
  - Then only names with prefix MIC_ are forwarded to the coprocessor with prefix stripped
  - **Exception**: MIC LD_LIBRARY_PATH is never passed to the coprocessor.
  - Value of LD_LIBRARY_PATH cannot be changed via forwarding of environment variables.
Environment Variables on the MIC

#include <stdio.h>
#include <stdlib.h>

int main()
{
    #pragma offload target (mic)
    {
        char* varmic = getenv("VAR");
        if (varmic) {
            printf("VAR=%s on MIC.\n", varmic);
        } else {
            printf("VAR is not defined on MIC.\n");
        }
    }
    char* varhost = getenv("VAR");
    if (varhost) {
        printf("VAR=%s on host.\n", varhost);
    } else {
        printf("VAR is not defined on host.\n");
    }
}
Environment Variables on the MIC

lu65fok@i01r13c01:~tests> ./.env
VAR is not defined on host.
VAR is not defined on MIC.
lu65fok@i01r13c01:~tests> export VAR=299792458
lu65fok@i01r13c01:~tests> ./.env
VAR=299792458 on host.
VAR=299792458 on MIC.
lu65fok@i01r13c01:~tests> export MIC_ENV_PREFIX=MIC
lu65fok@i01r13c01:~tests> ./.env
VAR=299792458 on host.
VAR is not defined on MIC.
lu65fok@i01r13c01:~tests> export MIC_VAR=3.141592653
lu65fok@i01r13c01:~tests> ./.env
VAR=299792458 on host.
VAR=3.141592653 on MIC.
The Preprocessor Macro __MIC__

- The macro __MIC__ is only defined in code version for MIC, not in the fallback version for the host.
- Allows to check where the code is running.
- Allows to write multiversioned code.
- __MIC__ also defined in native mode.
The Preprocessor Macro __MIC__

```c
#pragma offload target(mic)
{
    #ifdef __MIC__
        printf("Hello from MIC (offload succeeded).\n");
    #else
        printf("Hello from host (offload to MIC failed!).\n");
    #endif
}
```

```
lu65fok@login12:~/tests> icpc -offload=optional offload-mic.c
lu65fok@login12:~/tests> ./a.out
Hello from host (offload to MIC failed!).
lu65fok@i01r13c06:~/tests> ./a.out
Hello from MIC (offload succeeded).
```
Lab: Offload Mode I
Intel Xeon Phi Programming Models: Explicit Offload Model II
Proxy Console I/O

- stderr and stdout on MIC are **buffered and forwarded (proxied) to the host console.**
- Forwarding is done by the *coi_daemon* running on the MIC.
- Output buffer should be flushed with *fflush(0)* of the *stdio*-Library.
- Proxy console input not supported.
- Proxy I/O is **enabled by default.**
- Can be switched off using *MIC_PROXY_IO=0.*
#include <stdio.h>
#include <unistd.h>
__attribute__((target(mic))) extern struct _IO_FILE *stderr;
int main (int argc, char* argv[]){
    char hostname[100];
    gethostname(hostname,sizeof(hostname));

    #pragma offload target(mic) {
        char michostname[100];
        gethostname(michostname, sizeof(michostname));
        printf("MIC stdout: Hello world from MIC. I am %s and I have %ld logical cores. I was called from host: %s \n", michostname, sysconf(_SC_NPROCESSORS_ONLN), hostname);
        fprintf(stderr,"MIC stderr: Hello world from MIC. I am %s and I have %ld logical cores. I was called from host: %s \n", michostname, sysconf(_SC_NPROCESSORS_ONLN), hostname);
        fflush(0);
    }
    printf( "Host stdout: Hello world from host. I am %s and I have %ld logical cores.\n", hostname, sysconf(_SC_NPROCESSORS_ONLN));
    fprintf(stderr, "Host stderr: Hello world from host. I am %s and I have %ld logical cores.\n", hostname, sysconf(_SC_NPROCESSORS_ONLN));
Proxy Console I/O

lu65fok@i01r13c01:~/tests> ./proxyio 1>proxyio.out 2>proxyio.err

lu65fok@i01r13c01:~/tests> cat proxyio.out
MIC stdout: Hello world from MIC. I am i01r13c01-mic0 and I have 240 logical cores. I was called from host: i01r13c01
Host stdout: Hello world from host. I am i01r13c01 and I have 32 logical cores.

lu65fok@i01r13c01:~/tests> cat proxyio.err
MIC stderr: Hello world from MIC. I am i01r13c01-mic0 and I have 240 logical cores. I was called from host: i01r13c01
Host stderr: Hello world from host. I am i01r13c01 and I have 32 logical cores.
lu65fok@i01r13c01:~/tests>
Proxy Console I/O

```
lu65fok@i01r13c01:~tests> export MIC_PROXY_IO=0

lu65fok@i01r13c01:~tests> ./proxyio 1>proxyio.out 2>proxyio.err

lu65fok@i01r13c01:~tests> cat proxyio.out
Host stdout: Hello world from host. I am i01r13c01 and I have 32 logical cores.

lu65fok@i01r13c01:~tests> cat proxyio.err
Host stderr: Hello world from host. I am i01r13c01 and I have 32 logical cores.
```

27-29 June 2016
Intel MIC Programming Workshop
Data Traffic without Computation

- 2 possibilities:
  - Blank body of `#pragma offload`, i.e.
    ```
    #pragma offload target(mic) in (data: length(n))
    {}
    ```
  - Use a special pragma `offload_transfer`, i.e.
    ```
    #pragma offload_transfer target(mic) in(data: length(n))
    ```
Asynchronous Offload

- Asynchronous Data Transfer helps to:
  - Overlap computations on host and MIC(s).
  - Work can be distributed to multiple coprocessors.
  - Data transfer time can be masked.
Asynchronous Offload

- To allow asynchronous data transfer, the specifiers `signal()` and `wait()` can be used, i.e.

```
#pragma offload_transfer target(mic:0) in(data : length(n))
signal(data)
// work on other data concurrent to data transfer …
#pragma offload target(mic:0) wait(data) \nocospy(data : length(N)) out(result : length(N))
{
     ....
     result[i]=data[i] + ...;
}
```

Any pointer type variable can serve as a signal!

Device number must be specified!
Asynchronous Offload

- Alternative to the `wait()` clause, a new pragma can be used:
  
  ```
  #pragma offload_wait target(mic:0) wait(data)
  ```

- Useful if no other offload or data transfer is necessary at the synchronisation point.
Asynchronous Offload to Multiple Coprocessors

```c
char* offload0;
char* offload1;
#pragma offload target(mic:0) signal(offload0) \
         in(data0 : length(N)) out(result0 : length(N))
{
    Calculate(data0, result0);
}
#pragma offload target(mic:1) signal(offload1) \
         in(data1 : length(N)) out(result1 : length(N))
{
    Calculate(data1, result1);
}
#pragma offload_wait target(mic:0) wait(offload0)
#pragma offload_wait target(mic:1) wait(offload1)
```
#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))
            {
                mxm(n,a,b,c);
            }
        }
        #pragma omp section
        {
            //section running on the host
            mxm(n,d,e,f);
        }
    }
}
Persistent Data

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

- To allocate data and keep it for the next offload:
  `#pragma offload target(mic) in (p:length(l) ALLOC RETAIN)`
- To reuse the data and still keep it on the coprocessor:
  `#pragma offload target(mic) in (p:length(l) REUSE RETAIN)`
- To reuse the data again and free the memory. (FREE is the default, and does not need to be explicitly specified):
  `#pragma offload target(mic) in (p:length(l) REUSE FREE)`
Intel Xeon Phi Programming Models: Intel Cilk Plus
Cilk Plus

- http://www.cilkplus.org/
- “The easiest, quickest way to harness the power of both multicore and vector processing.”
- Extension to the C and C++ languages to support data and task parallelism.
- Only 3 new keywords to implement task parallelism
- Serial semantics make understanding and debugging the parallel program easier
- Array Notations provide a natural way to express data parallelism.
The Three Cilk Keywords

- Intel Cilk Plus adds three keywords to C and C++ to allow developers to express opportunities for
  - \textit{cilk\_for} - Parallelize for loops
  - \textit{cilk\_spawn} - Specifies that a function can execute in parallel with the remainder of the calling function
  - \textit{cilk\_sync} - Specifies that all spawned calls in a function must complete before execution continues
The C/C++ language extensions for array notations are Intel-specific language extensions that are part of the Intel® Cilk™ Plus feature.

**Major benefits:**
- Allows you to use array notation to program vector operations in a familiar language
- Achieves predictable performance based on mapping parallel constructs to the underlying SIMD hardware
- Enables compiler vectorization with less reliance on alias and dependence analysis

**Tutorial:** [http://www.cilkplus.org/tutorial-array-notation](http://www.cilkplus.org/tutorial-array-notation)
Intel Cilk Plus C/C++ Extensions for Array Notations

- The array notations work at all optimization levels. Specifying -O0 compiler options serializes the array operations into sequential loops and helps to debug applications using array notations.

  - \(a[lb:len:str]\) is an array section with \(len\) elements: \(a[lb], a[lb + str], a[lb + 2*str], \ldots, a[lb + (len-1)*str]\).

- Examples:
  - \(A[:] += B[:];\)
  - \(A[0:16] += B[32:16];\)

Number of elements, not upper bound!
Intel Xeon Phi Programming Models: MYO
MYO

- “Mine Yours Ours” virtual shared memory model.
- Alternative to Offload approach.
- Only available in C++.
- Allows to share not bit-wise compatible complex data (like structures with pointer elements, C++ classes) without data marshalling.
- Allocation of data at the same virtual addresses on the host and the coprocessor.
- Runtime automatically maintains coherence.
- Syntax based on the keywords __Cilk_shared and __Cilk_offload.
MYO: Example

#define N 10000
_Cilk_shared int a[N], b[N], c[N];

_Cilk_shared void add() {
for (int i = 0; i < N; i++)
c[i] = a[i] + b[i];
}

int main(int argc, char *argv[]) {
...
_Cilk_offload add(); // Function call on coprocessor:
...
#pragma offload_attribute(push, _Cilk_shared)
#include <math.h>
void function_1();
void function_2();
#pragma offload_attribute(pop)
void function_3();

int main(){
    _Cilk_offload function_1();
    function_3();
    _Cilk_offload function_2();
}

Dynamical Allocation of virtual shared Objects

- Special MYO functions to de/allocate shared memory
  - **Allocation:**
    - `void * _Offload_shared_malloc(size_t size)`
    - `void * _Offload_sharedAligned_malloc(size_t size, size_t alignment)`
  - **Deallocation:**
    - `__Offload_shared_free(void *p)`
    - `__Offload_sharedAligned_free(void *p)`
Dynamical Allocation of virtual shared Objects

- 1) `double*_Cilk_shared data;
- 2) `data = (double*_Cilk_shared) Offload_shared_malloc(N*sizeof(double));
- 3) `_Offload_shared_free(data);

- `&data[0] is the same on host and coprocessor
- Mind:
  - A) Shared pointer to shared data: `double*_Cilk_shared data
  - B) Unshared pointer to shared data: `double _Cilk_shared*

- B) in 1) data does not work:
  `myo2.c(13): error: *MIC* illegal to reference a non-shared global variable in a _Cilk_shared context
Virtual Shared Classes

- Offload Model only allows offloading of bitwise-copyable data.
- Sharing complicated structures with pointers or C++ classes is only possible via MYO
class _Cilk_shared Person {
  public:
    int id;
    char name[10];
  Person() { id=0; name[0]=\0; }
  void Set(_Cilk_shared const char* name0, const int id0) {
    id = id0;
    strcpy(name, name0);
  }
};

_Cilk_shared Person someone;
_Cilk_shared char who[100];

int main(){
  strcpy(who, "Mike");
  _Cilk_offload someone.Set(who, 2);
}
• To query the number of coprocessors:

\[
\text{int \ nmics} = \_\_\text{Offload\_number\_of\_devices}()
\]

• To specify which coprocessor \( n < \text{nmics} \) should do the computation:

\[
\text{\_Cilk\_offload\_to}(n) \ \text{func}();
\]

• Asynchronous offload:

\[
\text{\_Cilk\_spawn \_Cilk\_offload\_to}(i) \ \text{func}();
\]

Within func() also OpenMP, pthreads etc. can be used.
int _Cilk_shared *response;
void _Cilk_shared Respond(int _Cilk_shared & a) { a=1;}

response = (int _Cilk_shared *) _Offload_shared_malloc(n_d*sizeof(int));

int n_d = _Offload_number_of_devices();
response[0:n_d] = 0;

for (int i = 0; i < n_d; i++) {
    _Cilk_spawn _Cilk_offload_to(i) Respond(response[i]);
}
_Cilk_sync;

Needed only for asynchronous offload.
Offload: Using several MIC Coprocessors

- To query the number of coprocessors:
  
  ```c
  int nmics = __Offload_number_of_devices()
  ```

- To specify which coprocessor \( n < \) \( nmics \) should do the computation:
  
  ```c
  #pragma offload target(mic:n)
  ```

- If \( n > nmics \) then coprocessor \( (n \mod nmics) \) is used

- **Important for:**
  - Asynchronous offloads
  - Coprocessor-Persistent data
## MYO Language Extensions

<table>
<thead>
<tr>
<th>Entity</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>int _Cilk_shared f(int x){…}</td>
<td>Executable code for both host and MIC; may be called from either side</td>
</tr>
<tr>
<td>Global variable</td>
<td>_Cilk_shared int x = 0</td>
<td>Visible on both sides</td>
</tr>
<tr>
<td>File/Function static</td>
<td>static _Cilk_shared int x</td>
<td>Visible on both sides, only to code within the file/function</td>
</tr>
<tr>
<td>Class</td>
<td>class _Cilk_shared x {...}</td>
<td>Class methods, members, and operators are available on both sides</td>
</tr>
<tr>
<td>Pointer to shared data</td>
<td>int _Cilk_shared *p</td>
<td>p is local (not shared), can point to shared data</td>
</tr>
<tr>
<td>A shared pointer</td>
<td>int *_Cilk_shared p</td>
<td>p is shared, should only point at shared data</td>
</tr>
<tr>
<td>Offloading a function call</td>
<td>x = _Cilk_offload func(y)</td>
<td>func executes on MIC if possible</td>
</tr>
<tr>
<td></td>
<td>x = _Cilk_offload_to(n) func</td>
<td>func must be executed on specified (n-th) MIC</td>
</tr>
<tr>
<td>Offloading asynchronously</td>
<td>_Cilk_spawn _Cilk_offload func(y)</td>
<td>Non-blocking offload</td>
</tr>
<tr>
<td>Offload a parallel for-loop</td>
<td>_Cilk_offload _Cilk_for(i=0; i&lt;N; i++) {...}</td>
<td>Loop executes in parallel on MIC</td>
</tr>
</tbody>
</table>
Lab: Offload Mode II
Thank you!