Intel® Xeon Phi™ programming

September 22nd-23rd 2015

University of Copenhagen, Denmark

Legal Disclaimer & Optimization Notice

Intel® Xeon Phi™ offloading

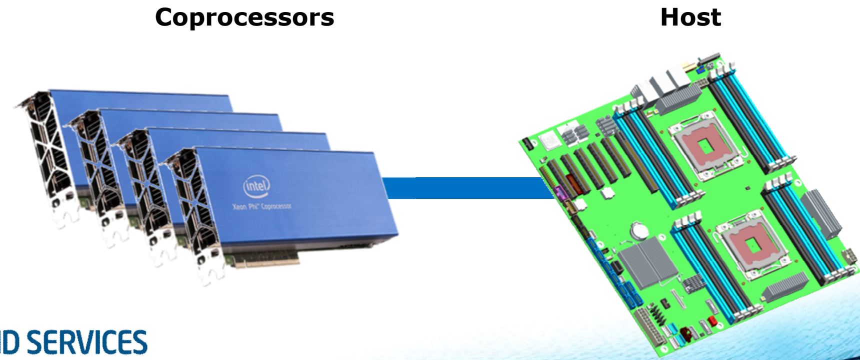
- Usually the most straightforward way efficiently utilize use Intel® Xeon Phi™ coprocessor is to use the *offload* programming model
  - Allows incremental porting of application to the coprocessor
  - Speedup of the computations must offset the data transfer costs!
- Programming models for Intel® Xeon Phi™ offload
  - **OpenMP * 4.0 device constructs**
  - Intel® Language Extensions for Offload (LEO)
  - OpenCL*
  - **pyMIC**

OpenMP 4.0 offloading

Intel® Xeon Phi™, Knights Corner (KNC)

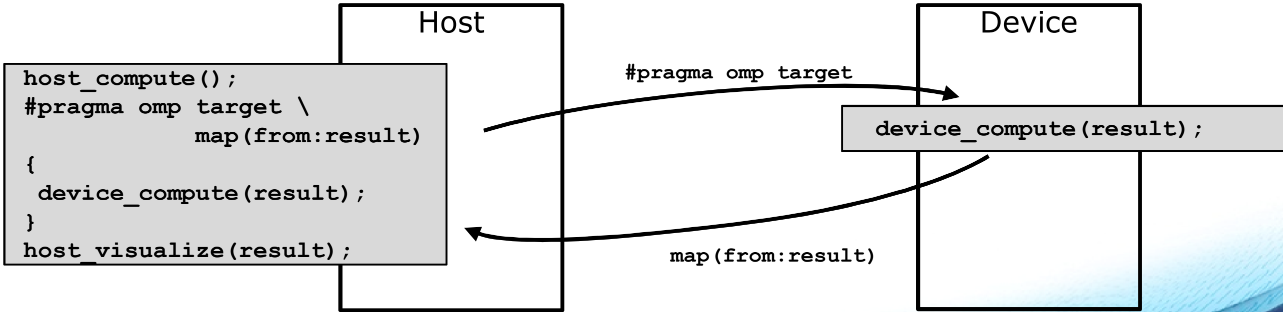
OpenMP 4.0 device model

- OpenMP 4.0 supports accelerators/coprocessors
- Device model
  - One host
  - Multiple accelerators/coprocessors of the same kind



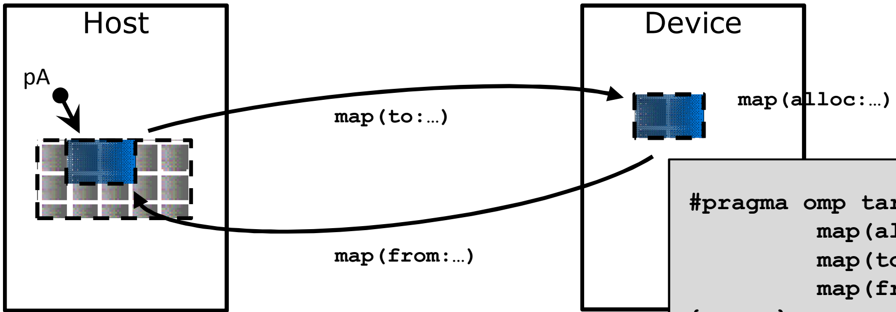
Execution model

- Transfer of control, data movement and parallelism must be defined separately
  - Host-centric execution model with device target regions



Data environment model

- Data environment is lexically scoped
  - Data environment is destroyed at the end of the scope
  - Buffers/data allocated by OpenMP runtime are automatically released




```
#pragma omp target data \
          map(alloc:...) \
          map(to:...) \
          map(from:...)
{ ... }
```

OpenMP target construct

- Create a device data environment **and** execute the construct on the same device
  - Transfer of control is sequential and synchronous
  - The transfer clauses control direction of data flow
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

OpenMP target data construct

- Create a device data environment for the extent of the region
  - Does not include a transfer of control
  - The **map** clauses control the direction of the data flow
- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],…]
structured-block
!$omp end target data
```

OpenMP target [data] clauses

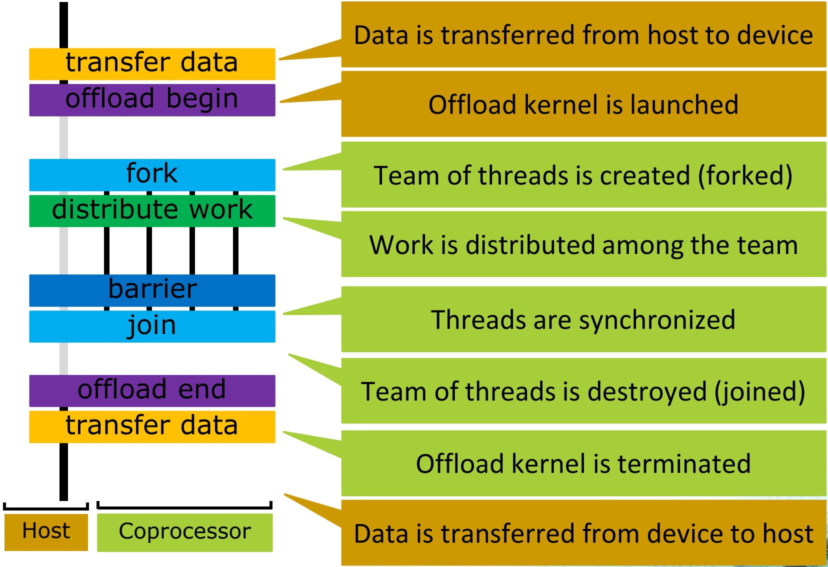
- **map([<alloc|from|to|tofrom>:]list)**
  Map data between the host and the device. Any mapped elements must be *bitwise copyable.* List items are allowed to be array sections
  - **map(alloc: list)**
    On **entry** to the device region, each new corresponding **list** item has an undefined initial value
  - **map(to: list)**
    On **entry** to the device region, each new corresponding **list** item is initialized with the value of the original **list** item
  - **map(from: list)**
    On **exit** from the device region, the value of the corresponding **list** item is assigned to each original **list** item
  - **map(tofrom: list)**
    On **entry** to the device region, behave as in **from**. On **exit** from the device region, behave as in **to**.
    Default if no **map-type** is specified
- **device(n)**
  Execute the target or target data region on device **n**

OpenMP array sections

- An OpenMP array section designates the elements in an array to map
  - Array sections must be contiguous in memory
- Syntax (C/C++)
  `array[lower-bound:length]`
  alternatively
  `array[:length]`, `array[lower-bound:]`, `array[:]`
- Syntax (Fortran)
  `array(start:end)`
  or alternatively any contiguous Fortran array section

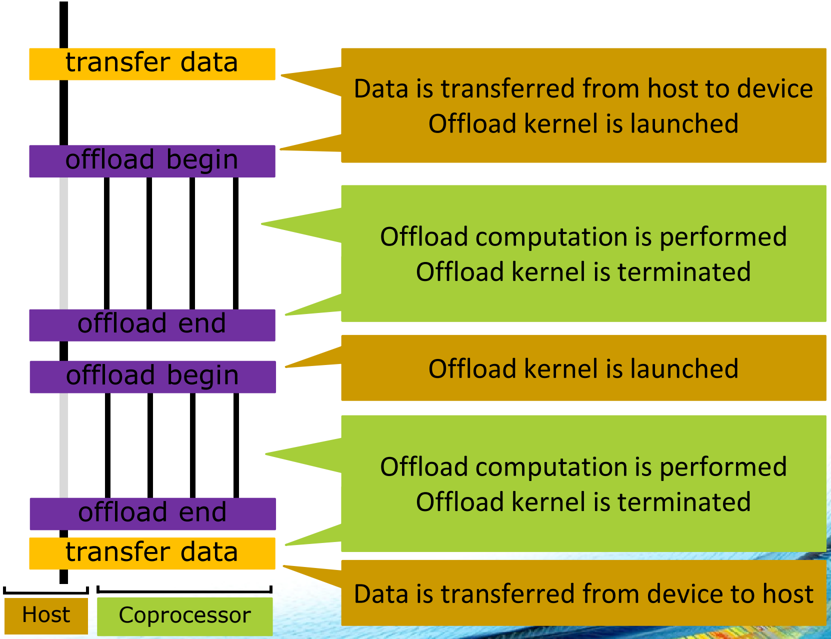
OpenMP target: example

```
#pragma omp target           \
            map(to:a[0:N])   \
            map(from:b[0:N])
{
   #pragma omp parallel for
   for (i = 0; i<N; i++)
   {
     b[i]=a[i]*a[i];
   }
}
```



OpenMP target data: example

```
#pragma omp target data          \
        map(alloc:tmp[0:N])  \
        map(tofrom:a[0:N])   \
{
   #pragma omp target
   #pragma omp parallel for
   for (i = 1; i<N-1; i++)
   {
     tmp[i]=(a[i-1]+a[i]+
                  a[i+1])/3;
   }
   #pragma omp target
   #pragma omp parallel for
   for (i = 1; i<N-1; i++)
   {
     a[i]=(tmp[i+1]+tmp[i])/2;
   }
}
```



OpenMP declare target construct

- Specify that variables and functions are mapped to a device
- Syntax (C/C++)

```
#pragma omp declare target [clause[[,] clause],…]
definition-seq
#pragma omd end declare target
```

- Syntax (Fortran)

  For variables and functions/subroutines

```
!$omp declare target(list)
```

  or for functions/subroutines, in the declarations part

```
!$omp declare target
```

OpenMP declare target construct

- Static data and functions mapped to device exist
  - For the host (like normally)
  - For the the target device (to be referenced to and invoked from the offload code)

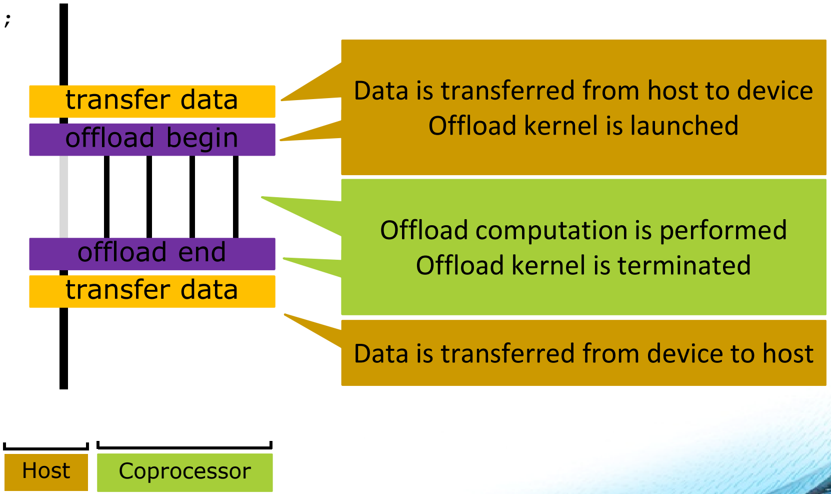
```
#pragma omp declare target
float nested_computation(float value)
{
    /* ... Implementation omitted ... */
}
float computation(float value, int n)
{
    for(int i=0; i<n; i++)
       value = nested_computation(value);
    return value;
}
static float * data;
#pragma omp end declare target
```

```
Host functions
nested_computation:
    ...
computation:
    ...
data:
    ...
```

```
Device functions
__offload_entry_nested_computation_offload_...:
    ...
__offload_entry_computation_offload_...:
    ...
data_8d777f385d3dfec8815d20f7496026dc_...:
    ...
```

OpenMP declare target: example

```
#pragma omp declare target
void computation(float *a, int n);
#pragma omp end declare target

#pragma omp target            \
        map(tofrom:a[0:N])    \
{
   for (int i=0; i<N; i++)
      computation(a, i);
}
```



OpenMP target update directive

- Make the corresponding list items in the device data environment consistent with their original list items
  - Request data transfers from within a target data region
  - *Motion* clauses control direction of data flow
- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],…]
```

- Syntax (Fortran)

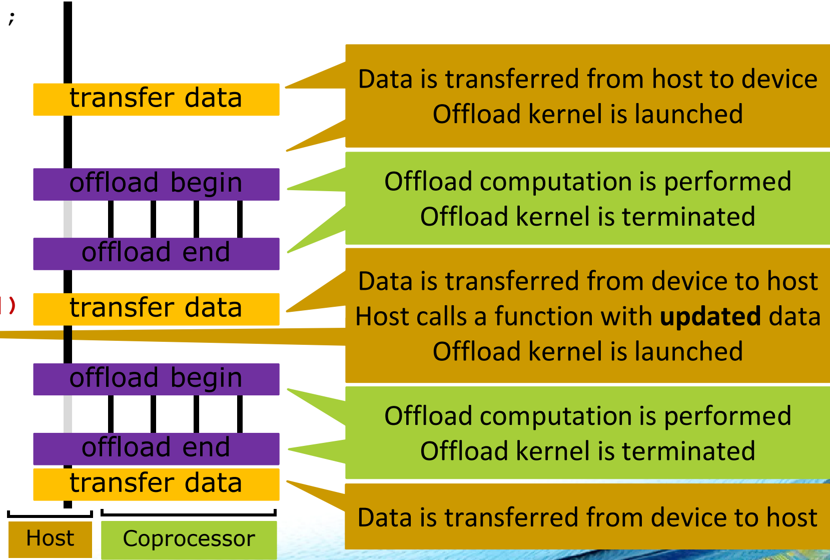
```
!$omp target update [clause[[,] clause],…]
```

OpenMP target update clauses

- `from(list)`
  For each list item in a from clause the value of the corresponding `list` item is assigned to the original `list` item
- `to(list)`
  For each list item in a to clause the value of the original `list` item is assigned to the corresponding `list` item
- `device(n)`
  Use the device `n` to update the list items with

OpenMP target update: example

```
#pragma omp declare target
void computation(float *a, int n);
#pragma omp end declare target

#pragma omp target data     \
        map(tofrom:a[0:N])  \
{
    #pragma omp target
    for (int i=0; i<N; i++)
      computation(a, i);

    #pragma omp update from(a[0:N])
    visualize(a, N);

    #pragma omp target
    for (int i=0; i<N; i++)
      computation(a, i);
}
```

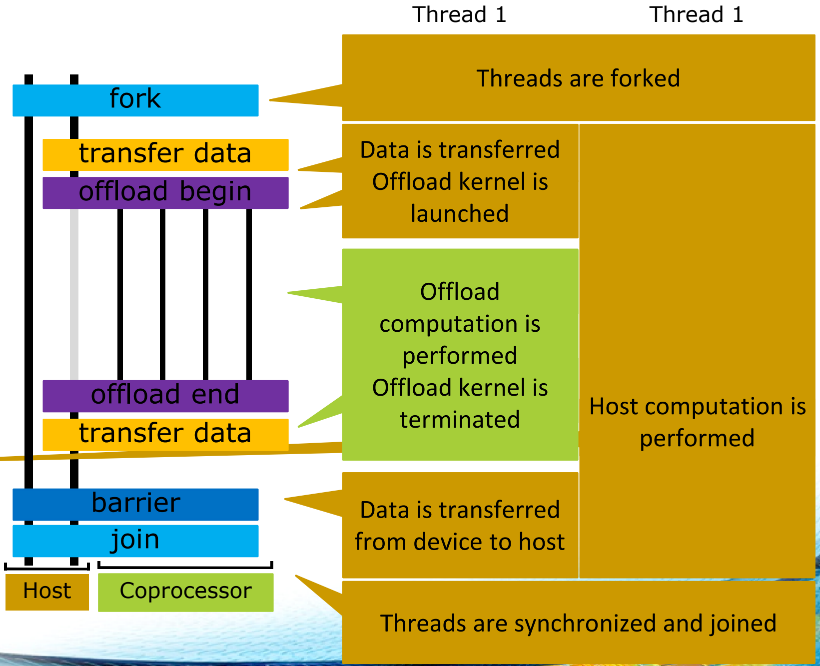


Asynchronous offloading

- With OpenMP 4.0, asynchronous offloading can be implemented by using OpenMP tasks
  - Requires at least 2 host threads to be active
  - Synchronization of tasks with taskwait or barrier
- NOTE: Using tasks works, but does not guarantee simultaneous execution of the offload region

Asynchronous offloading: example

```
#pragma omp parallel sections  \
                 num_threads(2)
{
   #pragma omp task
   {
    #pragma omp target          \
            map(tofrom:a[0:N]) \
    {
      device_compute(a, N);
    }
   }
   #pragma omp task
   {
    host_compute(a, N);
   }
   #pragma omp taskwait
}
```



SOFTWARE AND SERVICES

OpenMP 4.1 offloading

Intel® Xeon Phi™, Knights Corner (KNC)

SOFTWARE AND SERVICES

OpenMP 4.1 for devices

- Transfer control [and data] from the host to the device
- Syntax (C/C++)
  ```
  #pragma omp target [data] [clause[[,] clause],…]
  structured-block
  ```
- Syntax (Fortran)
  ```
  !$omp target [data] [clause[[,] clause],…]
  structured-block
  !$omp end target [data]
  ```
- General clauses (since OpenMP 4.0)
  ```
  device(scalar-integer-expression)
  map([alloc | to | from | tofrom:] list)
  if(scalar-expr)
  ```
- Clauses for asynchronous offloading (also supported by `target update`)
  ```
  nowait
  depend(dependency-type:list)
  ```

Asynchronous offloading

- With OpenMP 4.1, asynchronous offloading can be implemented by using **nowait** clause
  - Current task may resume while the target region executes
  - Synchronization with **barrier**

```
#pragma omp target map(tofrom:a[0:N]) nowait
{
   device_compute(a, N);
}
host_compute(a, N);

#pragma omp barrier
```

Creating and destroying device data

- Manage data without being bound to scoping rules
- Syntax (C/C++)

```
#pragma omp target enter data [clause[[,] clause],…]
#pragma omp target exit data [clause[[,] clause],…]
```

- Syntax (Fortran)

```
!$omp target enter data [clause[[,] clause],…]
!$omp target exit data [clause[[,] clause],…]
```

- Clauses

```
device(scalar-integer-expression)
map([alloc | delete | to | from | tofrom:] list)
if(scalar-expr)
depend(dependency-type:list)
nowait
```

Creating and destroying device data

```
struct DeviceBuffer {
    // ...
    DeviceBuffer(int dev, size_t sz) {
#pragma omp target enter data device(dev) map(alloc:buffer[:sz])
    }
    ~DeviceBuffer() {
#pragma omp target exit data device(dev) map(delete:buffer[:sz])
    }
}
```

```
void example() {
    DeviceBuffer *buf1 = new DeviceBuffer(0, 1024);
    compute_a_lot_using_offloading(buf1);
    DeviceBuffer *buf2 = new DeviceBuffer(0, 2048);
    compute_some_more_using_offloading(buf1, buf2);
    delete buf1;
    compute_evenmore_using_offloading(buf2);
    delete buf2;
}
```

Offloading from Python* - pyMIC

Intel® Xeon Phi™, Knights Corner (KNC)

SOFTWARE AND SERVICES

Python in HPC

- Python has gained a lot of interest throughout the HPC community (and others):
  - IPython
  - Numpy / SciPy
  - Pandas
- Intel® Xeon Phi™ Coprocessors are an interesting target to speed-up processing of Python codes

pyMIC introduction

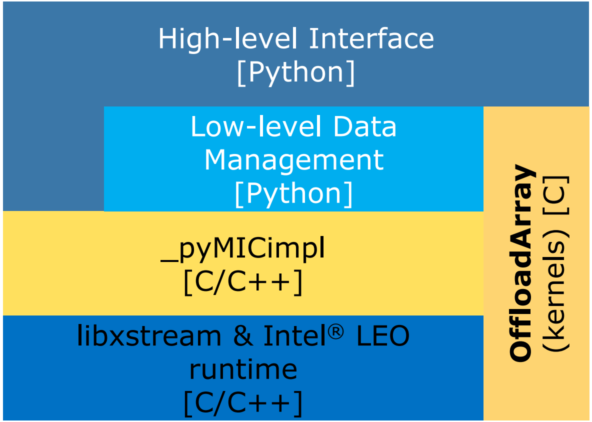
- pyMIC: A Python* Offload Module for the Intel® Xeon Phi™ Coprocessor
- Main developer: Michael Klemm, Intel *michael.klemm@intel.com*
- Available from github: https://github.com/01org/pyMIC

The pyMIC offload infrastructure

- Design principles (pyMIC's 4 "K"s)
  - Keep usage simple
  - Keep the API slim
  - Keep the code fast
  - Keep control in a programmer's hand
- pyMIC facts
  - 3800 lines of C/C++ code;
  - 1100 lines of Python code for the main API;
  - **libxstream** and Intel® LEO for interfacing with MPSS

High-level overview

- libxstream & Intel® LEO: low-level device interaction
  - Transfer of shared libraries
  - Data transfers, kernel invocation
- C/C++ extension module
  - Low-level device management
  - Interaction with LEO
- Low-level API with memcpy-like interface, smart device pointers
- High-level API with offload arrays
- Library with internal device kernels



Example dgemm: the host side…

```python
import numpy as np




m, n, k = 4096, 4096, 4096
alpha = 1.0
beta = 0.0
np.random.seed(10)
a = np.random.random(m * k).reshape((m, k))
b = np.random.random(k * n).reshape((k, n))
c = np.empty((m, n))

am = np.matrix(a)
bm = np.matrix(b)
cm = np.matrix(c)
cm = alpha * am * bm + beta * cm
```

```python
import pymic as mic
import numpy as np

device = mic.devices[0]
stream = device.get_default_stream()
library = device.load_library("libdgemm.so")

m,n,k = 4096,4096,4096
alpha = 1.0
beta = 0.0
np.random.seed(10)
a = np.random.random(m*k).reshape((m, k))
b = np.random.random(k*n).reshape((k, n))
c = np.empty((m, n))

stream.invoke(library.dgemm_kernel,
              a, b, c,
              m, n, k, alpha, beta)
stream.sync()
```

Example dgemm: the host side…

- Get a device handle (numbered from 0 to n-1)
- Load native code as a shared-object library
- Invoke kernel function and pass actual arguments
- Copy-in/copy-out semantics for arrays
- Copy-in semantics for scalars
- Synchronize host and coprocessor

```
import pymic as mic
import numpy as np

device = mic.devices[0]
stream = device.get_default_stream()
library = device.load_library("libdgemm.so")

m,n,k = 4096,4096,4096
alpha = 1.0
beta = 0.0
np.random.seed(10)
a = np.random.random(m*k).reshape((m, k))
b = np.random.random(k*n).reshape((k, n))
c = np.empty((m, n))

stream.invoke(library.dgemm_kernel,
              a, b, c,
              m, n, k, alpha, beta)
stream.sync()
```

Example dgemm: the target side…

- Arguments are passed as C/C++ types
- All argument passing is done with pointers to actual data

- Invoke (native) dgemm kernel

```
#include <pymic_kernel.h>

#include <mkl.h>

PYMIC_KERNEL
void dgemm_kernel(const double *A, const double *B,
                  double *C,
                  const int64_t *m, const int64_t *n,
                  const int64_t *k,
                  const double *alpha, const double *beta) {
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                *m, *n, *k, *alpha, A, *k, B, *n,
                *beta, C, *n);
}
```

High-level data structures

`OffloadDevice`

- Interaction with devices
- Loading of shared libraries

`OffloadStream`

- Invocation of kernel functions
- Buffer management

`OffloadArray`

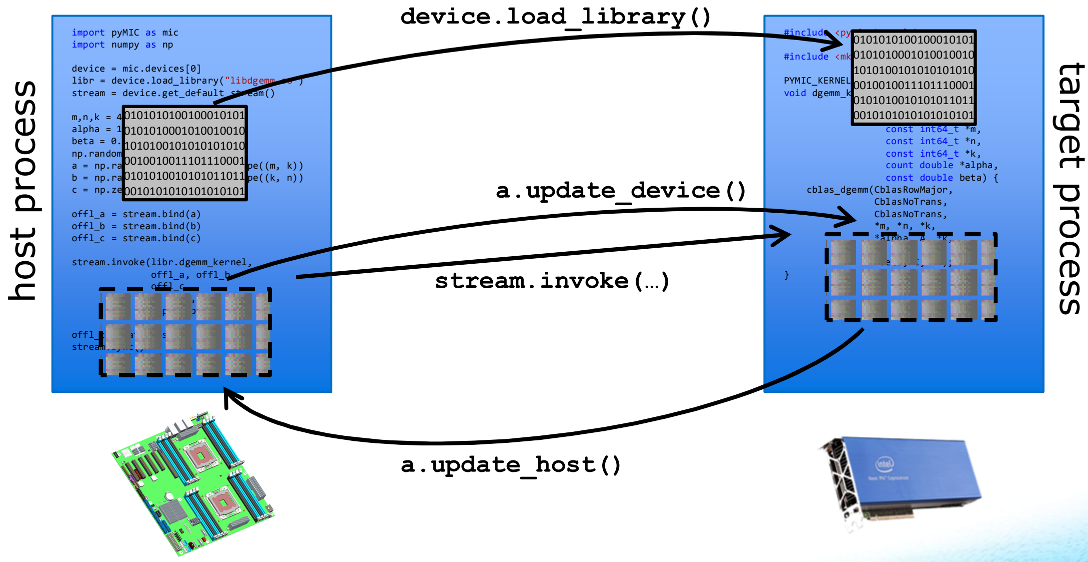
- numpy.ndarray container
- Transfer management
- Simple kernels and operators (fill, +, *)

Optimize offloads with high-level containers

- Get a device handle (numbered from 0 to n-1)
- Load native code as a shared-object library
- Use bind to create an offload buffer for host data
- Invoke kernel function and pass actual arguments
- Update host data from the device buffer

```
import pymic as mic
import numpy as np

device = mic.devices[0]
stream = device.get_default_stream()
library = device.load_library("libdgemm.so")

m,n,k = 4096,4096,4096
alpha = 1.0
beta = 0.0
np.random.seed(10)
a = np.random.random(m*k).reshape((m, k))
b = np.random.random(k*n).reshape((k, n))
c = np.zeros((m, n))

offl_a = stream.bind(a)
offl_b = stream.bind(b)
offl_c = stream.bind(c)

stream.invoke(library.dgemm_kernel,
              offl_a, offl_b, offl_c,
              m, n, k, alpha, beta)

offl_c.update_host()
stream.sync()
```

The high-level offload protocol



SOFTWARE AND SERVICES

Buffer management: buffer creation

```
class OffloadStream:
  def bind(self, array, update_device=True):
    if not isinstance(array, numpy.ndarray):
      raise ValueError("only numpy.ndarray can be associated "
                       "with OffloadArray")

    # detect the order of storage for 'array'
    if array.flags.c_contiguous:
      order = "C"
    elif array.flags.f_contiguous:
      order = "F"
    else:
      raise ValueError("could not detect storage order")

    # construct and return a new OffloadArray
    bound = pymic.OffloadArray(array.shape, array.dtype, order,
False,
                               device=self._device, stream=self)
    bound.array = array

    # allocate the buffer on the device (and update data)
    bound._device_ptr = self.allocate_device_memory(bound._nbytes)
    if update_device:
      bound.update_device()

    return bound
```

```
class OffloadStream:
  def allocate_device_memory(self, nbytes, alignment=64,
sticky=False):
    device = self._device_id
  if nbytes <= 0:
    raise ValueError('Cannot allocate negative amount of '
                     'memory: {0}'.format(nbytes))
  device_ptr = _pymic_impl_stream_allocate(device, self._stream_id,
                                           nbytes, alignment)
  return SmartPtr(self, device, device_ptr, sticky)
```

```
unsigned char *buffer_allocate(int device,
                               libxstream_stream *stream,
                               size_t size,
                               size_t alignment) {
  void *memory = NULL;
  libxstream_mem_allocate(device, &memory, size, alignment);
  return reinterpret_cast<unsigned char *>(memory);
}
```

Buffer management: data transfer

```python
class OffloadArray:
  def update_device(self):
    host_ptr = self.array.ctypes.get_data()
    s = self.stream
    s.transfer_host2device(host_ptr,
                           self._device_ptr,
                           self._nbytes)
    return None

  def update_host(self):
    host_ptr = self.array.ctypes.get_data()
    s = self.stream
    s.transfer_device2host(self._device_ptr,
                           host_ptr,
                           self._nbytes)
  return self
```

```c
void buffer_copy_to_target(int device,
                           libxstream_stream *stream,
                           unsigned char *src,
                           unsigned char *dst,
                           size_t size,
                           size_t offset_host,
                           size_t offset_device) {
  unsigned char *src_offs = src + offset_host;
  unsigned char *dst_offs = dst + offset_device;

  libxstream_memcpy_h2d(src_offs, dst_offs,
                        size, stream);
}
```

Example: Singular value decomposition

- Treat picture as 2D matrix M
- Compute SVD
$$M = U\Sigma V^T$$
- Ignore the smallest singular values and singular vectors
- "Optimal" compression of images (in a 2-norm sense)

Example: Singular value decomposition

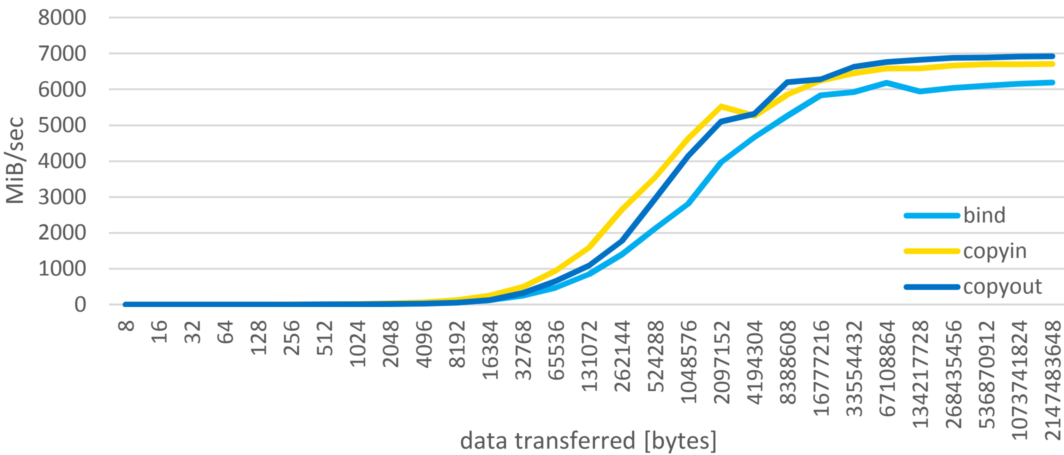
Host code

```
import numpy as np
import pymic as mic
from PIL import Image

def compute_svd(image):
    mtx = np.asarray(image.getdata(band=0),
                     float)
    mtx.shape = (image.size[1], image.size[0])
    mtx = np.matrix(mtx)
    return np.linalg.svd(mtx)

def reconstruct_image(U, sigma, V):
    reconstructed = U * sigma * V
    image = Image.fromarray(reconstructed)
    return image
```

Host code, cont'd

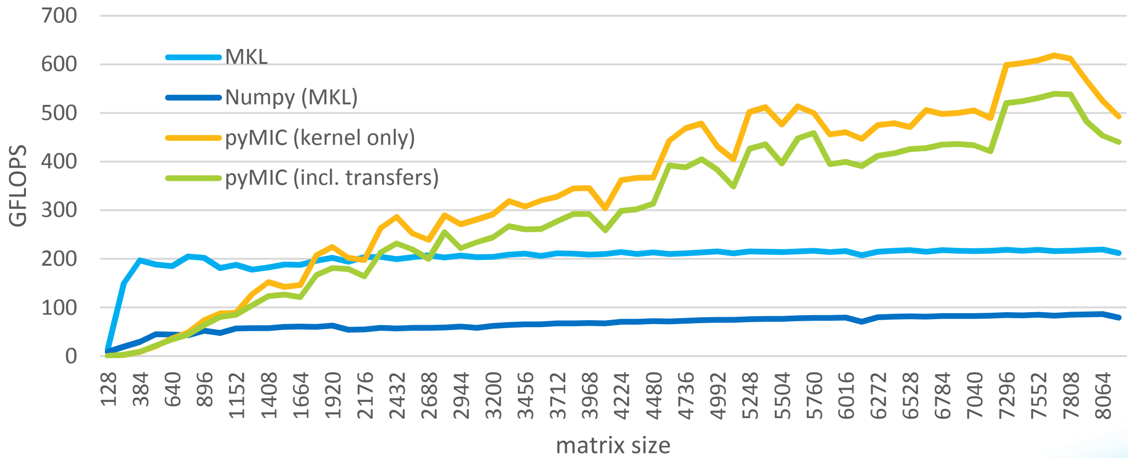
```
def reconstruct_image_dgemm(U, sigma, V):
    offl_tmp    = stream.empty((U.shape[0], U.shape[1]),
                               dtype=float, update_host=False)
    offl_res    = stream.empty((U.shape[0], V.shape[1]),
                               dtype=float, update_host=False)
    offl_U, offl_sigma = stream.bind(U), stream.bind(sigma)
    offl_V      = stream.bind(V)
    alpha, beta = 1.0, 0.0
    m, k, n = U.shape[0], U.shape[1], sigma.shape[1]
    stream.invoke_kernel(library.dgemm_kernel,
                         offl_U, offl_sigma, offl_tmp,
                         m, n, k, alpha, beta)
    m, k, n = offl_tmp.shape[0], offl_tmp.shape[1], V.shape[1]
    stream.invoke_kernel(library.dgemm_kernel,
                         offl_tmp, offl_V, offl_res,
                         m, n, k, alpha, beta)
    offl_res.update_host()
    stream.sync()
    image = Image.fromarray(offl_res.array)
    return image
```

Performance: Data transfer bandwidth



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Intel S2600GZ server with two Intel Xeon E5-2697v2 12-core processors at 2.7 GHz (64 GB DDR3 with 1867 MHz), Red Had Enterprise Linux 6.5 (kernel version 2.6.32-358.6.2) and Intel C600 IOH, one Intel Xeon Phi 7120P coprocessor (C0 stepping, GDDR5 with 3.6 GT/sec, driver v3.3-1, flash image/micro OS 2.1.02.0390), and Intel Composer XE 14.0.3.174. For more complete information visit http://www.intel.com/performance.

Performance: dgemm



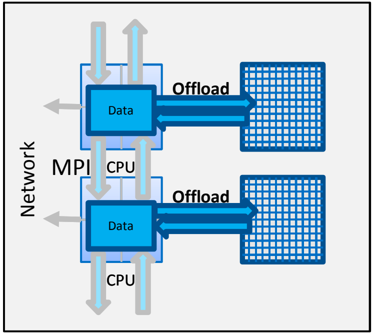
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Intel S2600GZ server with two Intel Xeon E5-2697v2 12-core processors at 2.7 GHz (64 GB DDR3 with 1867 MHz), Red Had Enterprise Linux 6.5 (kernel version 2.6.32-358.6.2) and Intel C600 IOH, one Intel Xeon Phi 7120P coprocessor (C0 stepping, GDDR5 with 3.6 GT/sec, driver v3.3-1, flash image/micro OS 2.1.02.0390), and Intel Composer XE 14.0.3.174. For more complete information visit http://www.intel.com/performance.

Offloading and MPI

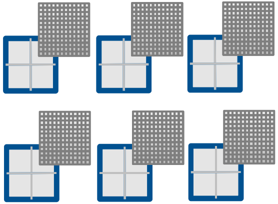
Intel® Xeon Phi™, Knights Corner (KNC)

MPI+Offload programming model

- MPI ranks on Intel® Xeon® processors (only)
- All MPI messages into/out of host CPUs
- Offload models used to accelerate MPI ranks
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* within Intel® Xeon Phi™ coprocessor



Homogenous network of heterogeneous nodes



Build Intel® 64 executable with included offload by using the Intel compiler
Run instances of the MPI application on the host, offloading code onto coprocessor
Advantages of more cores and wider SIMD for certain applications

MPI+Offload programming model

- MPI messaging done by the host
  - To send/receive data from the coprocessor, the data must be copied to/from the host memory and back
- Offloading from a **single** MPI task to a single Intel® Xeon Phi™ is straightforward
- Offloading from **multiple** MPI tasks to a single Intel® Xeon Phi™ is possible, but care must be taken not to overlap threads on the card
  - Use environment variable **`KMP_PLACE_THREADS`** to offset the separate MPI tasks on a single node

Conclusions

- With C/C++ and Fortran, OpenMP target directives can be used to offload computations to Intel® Xeon Phi™
- With Python, pyMIC can be used of offloading the computations
- Offloading can be combined with MPI

intel®