pyMIC: A Python* Offload Module for the Intel® Xeon Phi™ Coprocessor

Michael Klemm
Software and Services Group
Intel Corporation

Jussi Enkovaara
HPC Support
CSC Finland

Legal Disclaimer & Optimization Notice

Optimization Notice
Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Python in HPC

- Python has gained a lot of interest throughout the HPC community (and others):
  - IPython
  - Numpy / SciPy
  - Pandas

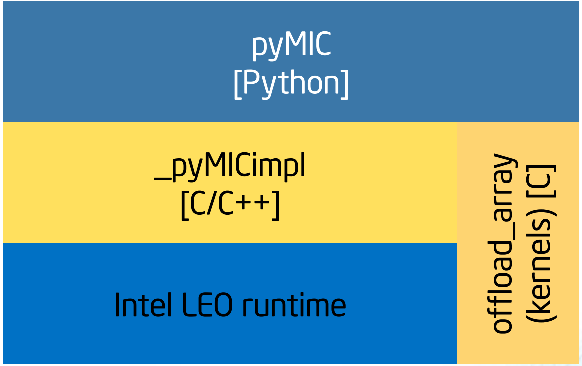
- Intel® Xeon Phi™ Coprocessor is an interesting target to speed-up processing of Python codes

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
  - 650 lines of C/C++ code; Intel® LEO for interfacing with MPSS
  - 450 lines of Python code for the main API

High-Level Overview

- Intel® LEO: low-level device interaction
  - Transfer of shared libraries
  - Data transfers
  - Code invocation
- C/C++ extension module
  - Low-level device management
  - Interaction with LEO
- Slim Python API to hide implementation details
- Library with internal device kernels



Key Data Structures

`offload_device`

- Interaction with devices
- Loading of shared libraries
- Invocation of kernel functions
- Buffer management

`offload_array`

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

Example dgemm: The Host Side…

- 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

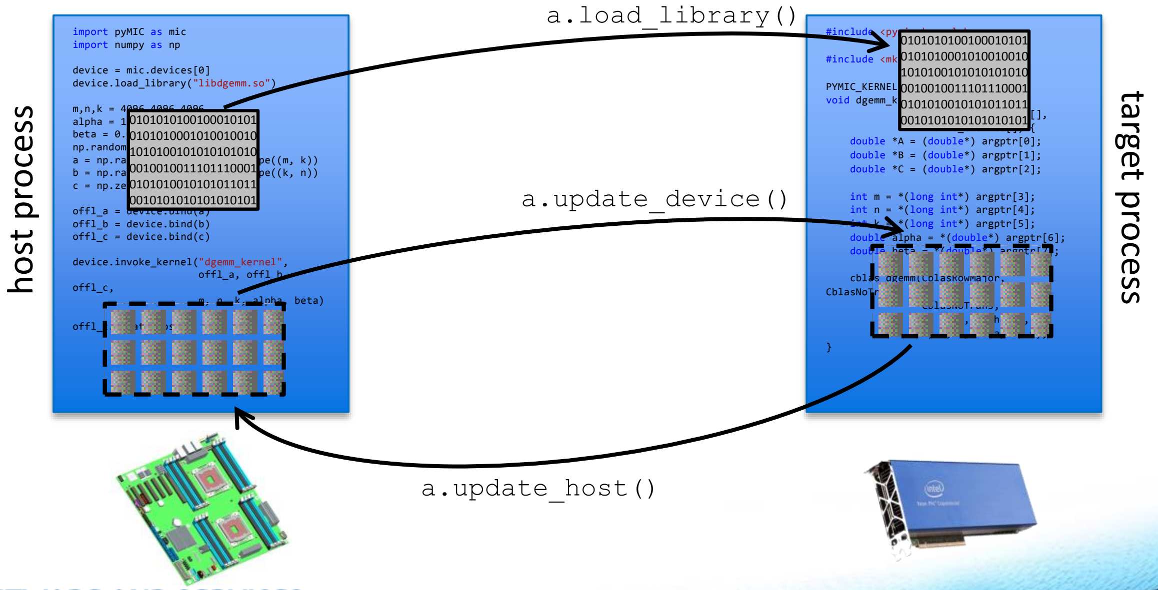
```
import pyMIC as mic
import numpy as np

device = mic.devices[0]
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 = device.bind(a)
offl_b = device.bind(b)
offl_c = device.bind(c)

device.invoke_kernel("dgemm_kernel",
                     offl_a, offl_b, offl_c,
                     m, n, k, alpha, beta)

offl_c.update_host()
```

Example `dgemm`: The Target Side

- Retrieve array pointer by casting `argptr` to target type
- Retrieve scalar arguments by casting and dereferencing
- Invoke (native) dgemm kernel

```
#include <pymic_kernel.h>

#include <mkl.h>

PYMIC_KERNEL
void dgemm_kernel(int argc, uintptr_t argptr[], size_t sizes[]) {
    double *A = (double*) argptr[0];
    double *B = (double*) argptr[1];
    double *C = (double*) argptr[2];

    int m = *(long int*) argptr[3];
    int n = *(long int*) argptr[4];
    int k = *(long int*) argptr[5];
    double alpha = *(double*) argptr[6];
    double beta = *(double*) argptr[7];

    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A, k, B, n, beta, C, n);
}
```

The Offload Protocol



Buffer Management: Buffer Creation

```python
class offload_device:
  def bind(self, array, update_device=True):
    if not isinstance(array, np.ndarray):
      raise ValueError("...")

    # construct a new offload_array
    ass = offload_array(array.shape, array.dtype,
                        order)
    ass.array = array

    if update_device:
      # allocate & copy
      self._copy_to_target(ass.array)
    else:
      # only allocate
      self._buffer_allocate(ass.array)

    return ass
```

```c
void buffer_allocate(int device, char* data,
                     size_t size) {
    uintptr_t host_ptr = (uintptr_t) data;
    uintptr_t dev = 0;
#pragma offload target(mic:device) \
      out(dev_ptr) \
      nocopy(data:length(size) \
             align(64) alloc_if(1) free_if(0))
    {
        dev_ptr = (uintptr_t) data;
    }
    buffers[device][host_ptr] = dev_ptr;
}
```

Buffer Management: Data Transfer

```python
class offload_device:
  def _buffer_update_on_target(self,
                                 *arrays):
    if len(arrays) == 0:
      raise
         ValueError("no argument")
    if type(arrays[0]) == tuple:
      arrays = arrays[0]
    for array in arrays:
      nbytes = int(array.nbytes)
      _pymic_impl_buffer_update_on_target(
        self.map_dev_id(), array, nbytes)
    return None
```

```c
void buffer_update_on_target(int device,
                             char* data,
                             size_t size)
{
    uintptr_t host_ptr =
        reinterpret_cast<uintptr_t>(data);
#pragma offload target(mic:device) \
        in(data:length(size) \
          align(64)
          alloc_if(0) free_if(0))
    {
        // do nothing
    }
}
```

Example: Singular Value Decomposition

- Treat picture as 2D matrix
- Decompose matrix:

$$M = U \times \Sigma \times V^T$$

- Ignore some singular values
- Effectively compresses images

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
```

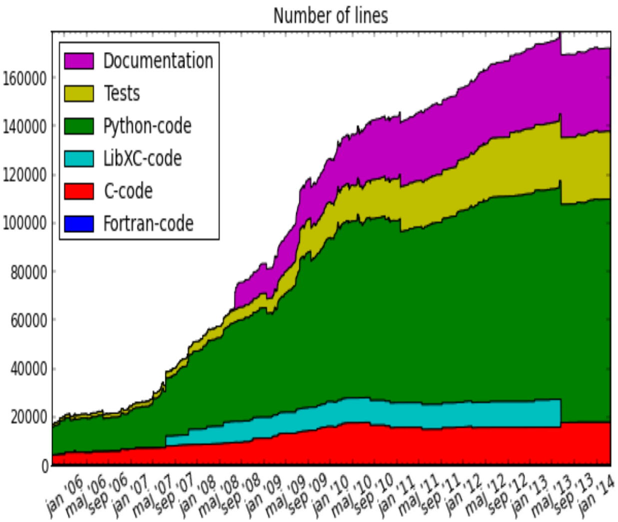
Host code, cont'd

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

Integration with GPAW

- GPAW is an open source software package for various quantum mechanical simulations at atomic scale
  - http://wiki.fysik.dtu.dk/gpaw
  - Few hundred users all over the world
- Implemented as a combination of Python and C
  - High-level algorithms in Python
  - Computational kernels in C (or in libraries)
  - Massively parallelized (with MPI)
  - Key operation: matrix-matrix multiplication

Integration with GPAW



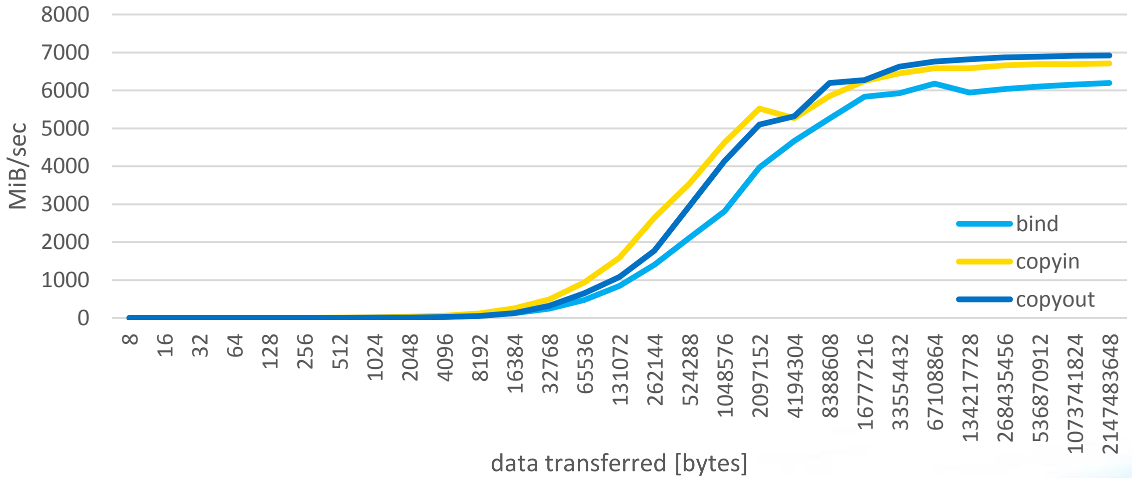
- Control the flow of large data (NumPy arrays) in Python level
- Offload heavy computations to coprocessor

Integration in GPAW

```
from gpaw.grid_descriptor
   import GridDescriptor

gpts = (64, 64, 64)
nbands = 512
cell = (8.23, 8.23, 8.23)
gd = GridDescriptor(gpts, cell)

psit_nG = gd.zeros(nbands, mic=True)
vt_G = gd.zeros(mic=True)
# Initialize psit_nG and vt_G
htpsit_nG = gd.zeros(nbands, mic=True)

for n in range(nbands):
    htpsit_nG[n] = vt_G * psit_nG[n]

H_nn = gd.integrate(psit_nG, htpsit_ng)
```

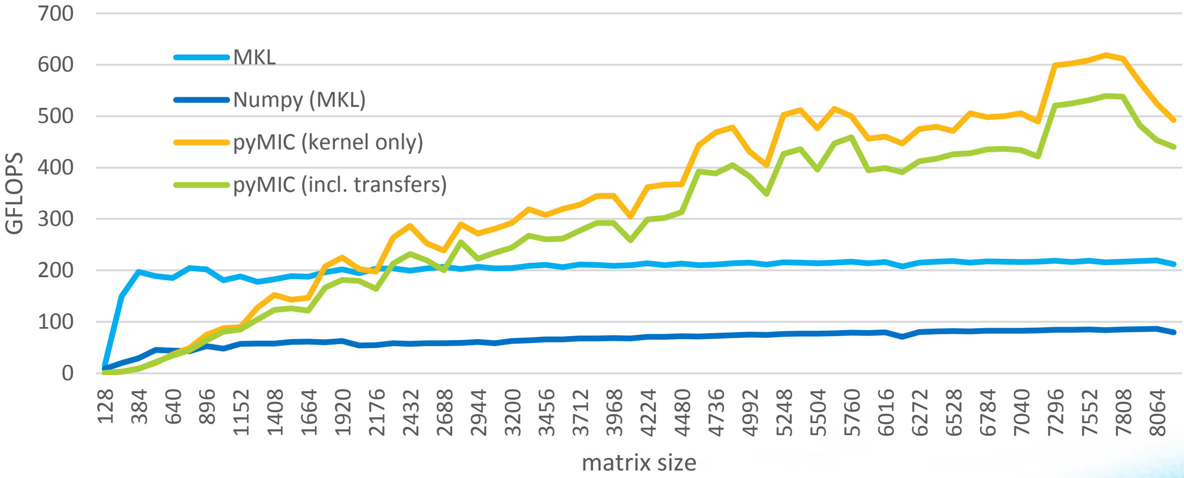
```
import pyMIC as mic

device = mic.devices[0]

...
    def zeros(self, n=(), dtype=float,
              mic=False):

        array = self._new_array(n, dtype)
        if mic:
            return device.bind(array)
        else:
            return array
```

Performance: Bandwidth of Data Transfers



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.

SOFTWARE AND SERVICES

Performance: GPAW integrate and rotate

Matrix size	integrate			rotate		
	Xeon	Xeon Phi	S	Xeon	Xeon Phi	S
n=256, $G=48^3$	0.10	0.11	0.91x	0.10	0.04	2.50x
n=256, $G=64^3$	0.25	0.25	1.00x	0.26	0.10	2.60x
n=256, $G=86^3$	0.61	0.55	1.11x	0.55	0.17	3.24x
n=256, $G=96^3$	0.78	0.79	0.99x	1.59	0.31	5.13x
n=512, $G=48^3$	0.30	0.12	2.50x	0.35	0.11	3.18x
n=512, $G=64^3$	0.74	0.27	2.74x	0.91	0.28	3.25x
n=512, $G=86^3$	1.75	0.57	3.07x	1.89	0.50	3.70x
n=512, $G=96^3$	2.53	0.97	2.61x	6.28	0.92	6.83x

Summary & Future Work

- pyMIC
  - A slim, easy-to-use offload interface for Python
  - Native kernels on the target devices
  - Almost negligible extra overhead for Python integration
- Future versions will likely bring:
  - Offloading of full Python code
  - Asynchronous offloading and data transfers
- Download pyMIC at https://github.com/01org/pyMIC.
- Mailinglist at https://lists.01.org/mailman/listinfo/pymic

intel®