pyMIC: A Python Offload Module for the Intel$^{®}$ Xeon Phi^{TM} Coprocessor

Michael Klemm
Software and Services Group
Intel Corporation
michael.klemm@intel.com

Jussi Enkovaara
High Performance Computing Support
CSC - IT Center for Science Ltd.
jussi.enkovaara@csc.fi

***Abstract*—Python has gained a lot of attention by the high performance computing community as an easy-to-use, elegant scripting language for rapid prototyping and development of flexible software. At the same time, there is an ever-growing need for more compute power to satisfy the demand for higher accuracy simulation or more detailed modeling. The Intel$^{®}$ Xeon Phi^{TM} coprocessor strives to provide additional compute power for floating-point intensive codes, while maintaining the programmability of the traditional Intel$^{®}$ Xeon$^{®}$ platform. In this paper, we present a Python module to handle offloads to the Intel Xeon Phi coprocessor from Python code. It provides an easy-to-use interface to invoke compute kernels on the coprocessor, while handling data transfers in a flexible yet performant way. We discuss how the Python module can be utilized to offload key kernels in the Python-based open source electronic-structure simulation software GPAW. Micro-benchmarks show that our solution imposes only marginal overheads on the kernel invocations. Our performance results for select GPAW kernels show a 2.5 to 6.8 fold performance advantage of offloading over host execution.**

I. INTRODUCTION

Python is one of the most commonly used programming language throughout the computing industry [24]. Python has proven to be an easy-to-use, elegant scripting language that allows for rapid prototyping and development of highly flexible software. In the past years, Python has also gained a lot of attention by the high performance computing (HPC) community. Add-on packages such as Numpy [20] and SciPy [23] provide efficient implementations of key data structures and algorithms, and, as implementing extensions with compiled languages such as C or Fortran is relatively straightforward, the performance aspects no longer prohibit the use of Python as an HPC language.

The need for speed in HPC also drives the need for coprocessor hardware that accelerates the compute-intense floating point operations of typical applications. General-purpose graphics processing units (GPGPUs) and the Intel$^{®}$ Xeon Phi^{TM} coprocessor [5] are examples of discrete extension cards which provide additional compute power on top of traditional CPUs such as the Intel$^{®}$ Xeon$^{®}$ processors.

GPAW is a prominent example of a Python HPC application. It is a versatile open source software package for various electronic structure simulations of nanostructures and materials science research problems [1], [11]. GPAW is based on the density-functional theory and time-dependent density-functional theory, and it offers several options for discretizing the underlying equations, such as uniform real-space grids, plane waves, and localized atomic orbital basis. GPAW can be run on wide variety of HPC systems, and depending on the input data set, simulations can scale up to tens of thousands of traditional CPU cores [22]. GPGPUs have already been shown to be beneficial for speeding up GPAW [14], and the Intel Xeon Phi coprocessor appears as an appealing alternative for GPU-based accelerators. GPAW is implemented as a combination of Python and C, thus the ability to operate on coprocessor in high-level Python code is considered highly beneficial.

In this paper, we present the design, implementation, and the performance of a Python module for offloading compute kernels from a Python program to the Intel Xeon Phi coprocessor. The pyMIC module follows Python's philosophy of being easy-to-use and being widely applicable. We explicitly design our module to blend in with the widely used Numpy package [20] for storing bulk floating-point data. The module is based on the Intel$^{®}$ Language Extensions for Offload [7], but wraps the C/C++ pragmas in a flexible, high-level Python interface. Our micro-benchmarks prove that our approach is not only feasible, but also delivers a low-overhead, efficient offload solution for applications written in Python. We discuss how pyMIC can be utilized in GPAW, and also present results for the offload performance of select key kernels in GPAW.

The paper is structured as follows. Section II surveys the state of the art and related work. Section III briefly introduces the architecture of the Intel Xeon Phi coprocessor. In Section IV, we describe the pyMIC module, its guiding design principles, and its implementation. Section V shows how the pyMIC module can be integrated in the physics application GPAW. Performance results of micro-benchmarks and GPAW kernels are shown in Section VI. Section VII concludes the paper.

II. RELATED WORK

To the best of our knowledge, pyMIC is the first offload infrastructure specifically to target offloading from Python applications to Intel Xeon Phi coprocessors.

The Intel$^{®}$ Manycore Platform Stack (MPSS) [3] ships with a Python version that runs natively on the coprocessor. Through Python's networking APIs it is possible to connect to a Python instance running on the coprocessor and invoke application code there. The Pyro4 [9] or RPyC [12] projects provide remote method invocation of methods by data serialization or by object proxies, respectively. SCOOP [15] provides a distributed, task-based programming model. Whereas the above projects provide a general solution to remote invocation or a parallel programming model, our pyMIC module specifically focuses on offloading to local coprocessors and targets efficient support for Numpy and SciPy.

The pyCUDA and pyOpenCL projects [16] both support offloading compute-intensive kernels from the host to a GPGPU. Similarly to our approach, they require the programmer to formulate the offloaded algorithms in either CUDA (pyCUDA) or OpenCL (pyOpenCL). Through the Intel® SDK for OpenCL™ Applications [4], pyOpenCL can also support the Intel Xeon Phi coprocessor. Our pyMIC module requires that the compute kernel is implemented in C/C++ instead. The Intel® Parallel Studio XE 2013 for Linux and Windows contains several programming models to facilitate offload programming for the Intel Xeon Phi coprocessor. The Intel® Language Extensions for Offloading (LEO) [7], [8] use C/C++ pragmas or Fortran directives, respectively, to tag code regions for execution on the coprocessor. Intel® Cilk™ Plus [7] defines special keywords to offload function calls. The Intel® Math Kernel Library (MKL) also supports offloading of kernels as part of the BLAS functions [6]

All these offload models have in common that they cannot be directly used from the Python language. One needs to employ the Python C/C++ interface [13] to bridge from Python to C/C++ and to make use of the C/C++ offload models. Our Python module utilizes Intel LEO internally to implement buffer management and offloading, but provides a high-level Python interface that does not require any boilerplate code.

The Intel® Coprocessor Offload Infrastructure (COI) [18] and Heterogeneous Active Messages (HAM) [19] provide a C/C++ API for offloading. COI is the central interface on top of which Intel LEO and Intel Cilk Plus implement their respective offload features. HAM provides an implementation of active messages [26] to send computation and its accompanying data to a remote coprocessor on the same host or on a distant host system. HAM provides a low-latency approach to offloading computation, which makes it interesting for pyMIC and we are investigating how to provide an additional back-end for pyMIC to utilize HAM's API.

III. The Intel Xeon Phi Coprocessor

The Intel Xeon Phi coprocessor [5] is a massively parallel compute device deployed as a PCI* Express (gen 2) extension card. Its foundations in the well-known Intel Architecture (IA) enables a rich choice of programming models to program both Xeon processors and the Xeon Phi coprocessor. The similar architecture keeps the programming environment and tool chain the same across the two platforms.

Fig. 1 shows the Intel Many Integrated Core architecture and the Intel Xeon Phi coprocessor. The coprocessor offers up to 61 general-purpose cores with a base frequency of up to 1238 MHz and a maximum of 16 GB of on-card GDDR5 memory. With turbo mode, the frequency can be increased to a maximum of 1333 GHz. The cores are based on a revamped Intel® Pentium® (P54C) processor design [2]. Each core can execute 64-bit instructions and offers a 512-bit wide Vector Processing Unit (VPU) for SIMD processing. Each of the

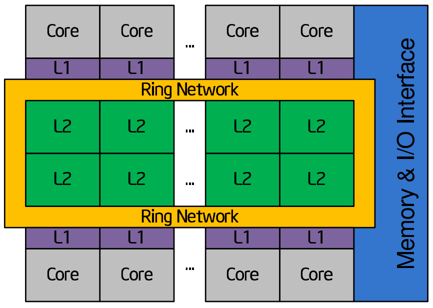

Fig. 1. High-level architecture of the Intel Xeon Phi coprocessor with cores, the ring interconnect, and system interface.

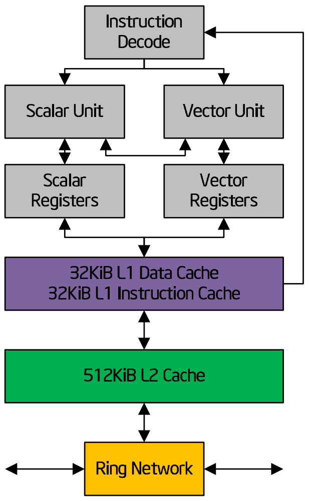

Fig. 2. Schematic view of one coprocessor core with caches, pipelines, and ring hub.

cores supports four hardware threads with four-way round-robin scheduling. In each clock cycle, the decoder stage selects a new instruction stream and feeds its next instruction into the pipeline. The peak floating-point performance for double precision (DP) is 1208 GFLOPS or 2416 GFLOPS for single-precision (SP) computation, respectively.

Each core of the coprocessor owns a private L1 and L2 cache with 32 KB and 512 KB, respectively. With 61 cores, the overall size of the L2 cache is 31.5 MB. Similar to traditional IA, caches are fully coherent and implement the standard coherency protocol across the caches and the card's memory. An on-die network connects all cores and their L2 caches. The ring consists of several bidirectional rings specializing for different kinds of traffic: a 64-byte wide data-block ring (BL), two command and address rings (AD), and two coherency and credit rings (AK). The tag directories that keep track of cache contents are uniformly distributed across the ring network.

Each coprocessor core has two execution pipelines: scalar and vector (see Fig. 2). The scalar pipeline executes all scalar instructions, such as branches, address calculations, and other non-vector instructions of the Intel instruction set. The vector pipeline executes the 512-bit SIMD instructions and can also execute scalar instructions for increased throughput. The decoder stage can decode and feed two instructions simultaneously into the core's pipelines. The SIMD instructions support DP and SP computation as well as integer arithmetic. Each instruction can be masked through a register of *true* and *false* values or can contain a swizzle pattern to rearrange data before an operation is applied to the vector elements. Load and store instructions further support data conversions, such as converting a vector of integer values to a vector of DP or SP values.

The coprocessor runs a complete Linux* software stack with support for TCP/IP and OFED/DAPL-based software as well as the typical Linux APIs (e. g., system calls, POSIX* threads, etc.). As an IA-based compute device, it can support a variety of (parallel) programming models that are available on IA. The Intel® Composer compiler suite for the Intel Xeon Phi coprocessor supports C/C++ and Fortran (including Co-Array Fortran). The suite also supports the OpenMP* API version 4.0 [21] and MPI [17] as the traditional HPC models for thread and process parallelism. The VPU can be accessed through the auto-vectorization feature of the Intel Composer, compiler pragmas, OpenMP SIMD constructs, and through low-level intrinsic functions. For each of the programming models, there is the option of using the coprocessors of a cluster as individual compute nodes (native mode) or in a heterogeneous offload mode where a host thread transfers data and control to the coprocessor from time to time [7], [8].

IV. The pyMIC Module

In this section, we present key requirements and design decisions that influenced the interface of the pyMIC module. We then show the integral parts of the pyMIC interface and show how offloading can be used from a Python application. Finally, the section concludes with an introduction into the implementation of the pyMIC module.

A. Design

The key guiding principle of the design of the pyMIC module is to provide an easy-to-use, slim interface at the Python level. At the same time, programmers should have full control over data transfers and offloading, to avoid potential inefficiencies or non-determinism while executing offload code. Because Numpy is a well-known package for dealing with (multi-dimensional) array data, we explicitly designed pyMIC to blend well with Numpy's `ndarray` class and its array operations. As we will see later, `ndarrays` are the granularity of buffer management and data transfer between host and coprocessors.

Although the Intel LEO approach is very generic in that it supports offloading of arbitrary code regions marked by compiler pragmas or directives, a similar approach for Python would require specialized syntax to tag Python code for offloading. While this approach seems the most generic, we avoid its complexity in favor of the simplicity of the pyMIC interface. Instead, we adopt the offload model of Intel Cilk Plus that uses functions as the main granularity for offloading. The rationale is that in the case of Numpy most compute-intensive operations are implemented through function calls. In addition, most HPC codes exhibit their computational hotspots (e.g., solvers) as functions or function calls into libraries such MKL. Whereas Cilk Plus uses virtual shared memory to handle data transfers transparently, we require explicit transfers to avoid hidden transfer overheads that are hard to detect by programmers.

```
import pyMIC as mic

# acquire handle of offload target
dev = mic.devices[0]
offl_a = dev.associate(a)

# load library w/ kernel
dev.load_library("libnop.so")

# invoke kernel
dev.invoke_kernel("nop")
```

Fig. 3. Simplistic offload example to acquire an offload device and invoke a kernel.

```
/* compile with:
 icc -mmic -fPIC -shared -olibnop.so nop.c
*/

#include <pymic_kernel.h>

PYMIC_KERNEL
void nop(int argc, uintptr_t argptr[],
         size_t sizes[]) {
    /* do nothing */
}
```

Fig. 4. Empty kernel implementing the `nop` kernel of Fig. 3.

Finally, we require pyMIC to integrate well with other offload programming models for the Intel Xeon Phi coprocessor, most notably the Intel LEO model. Several HPC applications not only use Python code, but also implement parts of their application logic in C/C++ and/or Fortran. We strive to keep pyMIC flexible, so that advanced programmers can mix Python offloads with C/C++ or Fortran offloads. For instance, one could allocate data in an `ndarray`, transfer it to the coprocessor through the pyMIC interface, and then use the data from an offloaded C/C++ code region in a Python C/C++ extension.

B. Interface

The pyMIC interface consists of two key classes: `offload_device` and `offload_array`. The `offload_device` class provides the interface to interact with offload devices, whereas `offload_array` implements the buffer management and primitive operations on a buffer's data.

Fig. 3 shows a simplistic example of how to invoke a simple kernel with no input arguments. The code first imports the pyMIC module and makes its name space available as `mic`. Once initialized, all available offload devices are enumerated through the module variable `devices` and can be selected through their numerical ID. If the ID does not match an available offload target, an exception is raised.

The two operations implemented by `offload_device` are: `invoke_kernel` and `load_library`. The `invoke_kernel` method can be used to invoke a function

TABLE I. OPERATIONS OF THE `OFFLOAD_ARRAY` CLASS.

Operation	Semantics
`update_host()`	Transfer the buffer from the device
`update_device()`	Transfer the buffer to the device
`fill(value)`	Fill the buffer with the parameter
`fillfrom(array)`	Copy the content `array` to into the offload buffer
`zero()`	Fill the buffer with 0 (equivalent to `fill(0.0)`)
`reverse()`	Reverse the contents of the buffer.
`reshape()`	Modify the dimensions of the buffer; creates a new view.

on the target device once the native code has been loaded by `load_library`. The `invoke_kernel` method requires at least one argument, which has to contain a Python string that corresponds to the function name of the kernel to be invoked. For the example in Fig. 3, a kernel that implements a no-op might look like the C code in Fig. 4.

Kernel functions have to be compiled as a native shared-object library so that they can be loaded by calling `load_library` with the library's filename. The function attribute `PYMIC_KERNEL` ensures that the compiler emits the function symbol with the right visibility, so that the symbol can be found after the library has been loaded.

The function signature of the kernel is required to match the formal parameters as shown in Fig. 4. The first parameter, `argc`, determines the number of actual arguments passed into the kernel through `invoke_kernel`. The `argptr` and `sizes` argument contain pointers to the data and their respective sizes in bytes.

Fig. 5 and Fig. 6 show a more sophisticated example of a `dgemm` kernel that uses three Numpy arrays (the matrices) plus scalar data (the matrices' dimensions as well as α and β) to invoke the kernel. The code first creates and initializes three Numpy arrays to store the matrices `a`, `b`, and `c`. It then associates each of the Numpy arrays with a corresponding `offload_array`. The `associate` method hands an array's data over the pyMIC's buffer management and allocates buffer space on the target device. By default it also issues the initial data transfer so that the buffer is initialized properly on the target. If desired this initial transfer can be skipped by passing `False` as the second argument to `associate`.

The code in Fig. 5 passes the associated data buffers as additional arguments to `invoke_kernel` after the kernel's name. It also passes all scalar data (variables `m`, `n`, `n`, `alpha`, and `beta`) to the kernel. The offload infrastructure automatically performs a copy-in operation for these data that are not already present on the device. Because our module cannot safely determine if the data has been changed, it performs a copy-out operation to update the corresponding host data for all non-present array structures. To manually update data on the target or the host, the `offload_array` class offers the methods `update_host` and `update_device` for each respective direction of data transfer.

Table I summarizes the additional operations that are offered by the `offload_array` class. In addition, the `offload_array` implements the full set of array operations for element-wise addition, multiplication, etc. The operations are all performed on the target device and only require a data transfer if the input data of the second operand is not yet present on the target.

Fig. 6 shows what unpacking of data for the kernel in-

```python
import pyMIC as mic
import numpy as np

# size of the matrixes
m = 4096
n = 4096
k = 4096

# create some input data
alpha = 1.0
beta = 0.0
a = np.random.random(m*k).reshape((m, k))
b = np.random.random(k*n).reshape((k, n))
c = np.zeros((m, n))

# load kernel library
dev = mic.devices[0]
dev.load_library("libdgemm.so")

# associate host arrays with target arrays
offl_a = dev.associate(a)
offl_b = dev.associate(b)
offl_c = dev.associate(c)

# perform the offload
dev.invoke_kernel("dgemm_kernel",
                  offl_a, offl_b, offl_c,
                  m, n, k,
                  alpha, beta)

offl_c.update_host()
```

Fig. 5. Using pyMIC to perform a `dgemm` operation on the coprocessor.

```c
#include <pymic_kernel.h>
#include <mkl.h>

PYMIC_KERNEL
void dgemm_kernel(int argc,
                  uintptr_t argptr[],
                  size_t sizes[]) {
    int i;

    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);
}
```

Fig. 6. Kernel in C implementing the offload part of Fig. 5.

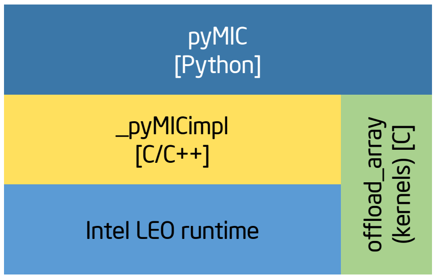

Fig. 7. Architecture of the pyMIC module.

vocation of Fig. 5 looks like on the target. The arguments given to `invoke_kernel` are passed to the C function on the target in the exact same order as they appear at the Python level. For arrays, one can easily unpack them by applying a cast operation to retrieve a pointer from the `argptr` array. Scalar values are also passed by pointers and thus require one dereferencing operation to retrieve the scalar value. After all the unpacking has been performed, the code calls the C interface of MKL's `dgemm` function to run the actual kernel.

C. Implementation

The layered architecture of the pyMIC module is depicted in Fig. 7. The top-level module contains all the high-level logic of the pyMIC module and provides the pyMIC API to be used by the application. Underneath this API module, a Python extension module (`_pyMICimpl`) written in C/C++ interfaces with the offload runtime of the Intel Composer XE and its LEO pragmas. In addition, pyMIC contains a library with standard kernels that implement all the array operations of `offload_array` (see Table I).

We have opted to use C++ to implement the `_pyMICimpl` to make use of STL for improved coding productivity. As a matter of fact, the interface of the extension is exposed to Python with the C calling convention, while the remainder of the code is plain C++. The internal code design of `_pyMICimpl` provides a series of abstractions so that, for instance, the Intel LEO pragmas can easily be replaced by the HAM interface or another offload implementation.

D. Buffer Management

Buffer management is critical when it comes to performance; we need to make sure that we issue as few data transfers as possible.

The high-level Python data management primitives (`associate`, `update_host`, `update_device`) have a direct correspondence with low-level routines in the `_pyMICimpl` module. Fig. 8 shows how the Python code interacts with the low-level interfaces. If, for instance, an update of the device is requested for an array, the `update_device` method invokes the corresponding extension function of the `_pyMICimpl` module and passes the Numpy array and it's size information along. After processing the actual arguments (`PyArg_ParseTuple`), the `_pymic_impl_buffer_update_on_target` function invokes the `buffer_update_on_target` low-level function that is shown in Fig. 9.

```
class offload_array:
  def update_device(self):
    self.device.
      _buffer_update_on_target(self.array)
    return None

class offload_device:
  def associate(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

  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
```

Fig. 8. Example functions for buffer management in pyMIC: `associate` and `update_device`.

For the buffer management and data transfer we rely on the Intel LEO pragmas that allow for fine-grained control of each aspect data transfers. As Fig. 9 shows, function `buffer_allocate`, the offload pragma targets a specific coprocessor device indicated by the ID in the variable `device`. To allocate a buffer, we do not need to transfer data (indicated by `nocopy`), but we need to allocate memory on the target (`alloc_if(1)`) and keep it (`free_if(0)`). The size of the buffer is specified through the `size` variable. By using the `align(64)` clause we also ensure that the buffer is aligned for maximum performance of the coprocessor. An alignment of 64 bytes naturally matches the width of the SIMD vectors of the coprocessor cores and thus provides best performance.

For a buffer update, we can safely assume that the buffer has already been allocated by a call to `buffer_allocate` on the target device, so we transfer `size` bytes pointed to by the pointer `data` to the coprocessor (`in`). The fact that the buffer is already allocated is expressed by `alloc_if(0)` and `free_if(0)`, which tells the LEO runtime to not allocate or release any buffer space.

```
 1  std::unordered_map<uintptr_t, uintptr_t>
 2      buffers[PYMIC_MAX_DEVICES];
 3
 4  void buffer_allocate(
 5          int device,
 6          char* data,
 7          size_t size) {
 8      uintptr_t host_ptr = (uintptr_t) data;
 9      uintptr_t dev = 0;
10  #pragma offload target(mic:device)
11          out(dev_ptr)
12          nocopy(data:length(size)
13            align(64)
14            alloc_if(1) free_if(0))
15      {
16          dev_ptr = (uintptr_t) data;
17      }
18      buffers[device][host_ptr] = dev_ptr;
19  }
20
21  void buffer_update_on_target(
22          int device,
23          char* data,
24          size_t size)
25  {
26      uintptr_t host_ptr =
27          reinterpret_cast<uintptr_t>(data);
28      uintptr_t device_ptr = 0;
29  #pragma offload target(mic:device) \
30          out(device_ptr) \
31          in(data:length(size) \
32            align(64)
33            alloc_if(0) free_if(0))
34      {
35          // do nothing
36      }
37  }
```

Fig. 9. Buffer management: data allocation on the device and data transfer to the device.

Intel LEO automatically maintains a mapping between a pointer on the host and the corresponding pointer on the device. The `data` pointer argument of `buffer_allocate` points to the raw storage space of a Numpy array that we associate or update. Once the pointer is used with the offload pragma, LEO establishes the mapping between the pointee and the storage location for it on the coprocessor. Although it is maintained transparently, we need to determine the device pointer (Fig. 9, line 16) and transfer this value back to the host as a integer value. We will later use these integer values as the pointer values passed into the kernel functions.

E. Kernel Invocation

As described above, kernel invocation is performed by calling the `invoke_kernel` method of the `offload_device` class. Before a kernel can be called, its shared-object library needs to be loaded on the target device (`load_library`).

The implementation of `load_library` is straightforward. It enters the extension module and issues an offload region that invokes the `dlopen` system call of the Linux* kernel to load the shared library into the process space on the coprocessor. The handle of the loaded library is then cached in an `stl::unordered_map`, so that the handle can be used to search the library with the `dlsym` function to find the function pointer of a particular kernel function.

Once a library has been loaded, its kernel functions can be used by `invoke_kernel`. It calls `dlsym` to find a function that matches the kernel name and then prepares the `argc`, `argptr`, and `sizes` arguments for the kernel. To avoid expensive search operations in libraries, the function pointer of a kernel is cached once it has been executed for the first time.

For each actual argument that is a Numpy `ndarray`, the `invoke_kernel` automatically allocates a buffer on the device and copies the data to it. The resulting device pointer is then added to the `argptr` array (copy-in). If the actual argument is a scalar value, our implementation creates a Numpy `ndarray` with just a single element and performs the same operations as with other arrays. If the actual argument is an `offload_array`, we assume that the data has been allocated and transferred ahead-of-time. Once the argument list has been processed, the native part of `invoke_kernel` then issues the actual function call of the kernel function on the target device. For all Numpy `ndarrays` that have been copied to the device, the code automatically also performs the copy-out operation to transfer any potential modifications of the array data on the target back on the host.

V. Integration in GPAW

GPAW uses Python to allow for a programmer-friendly implementation of high-level algorithms. C and high-performance libraries are used to obtain high performance for the computational kernels. The main parallelization approach is MPI, which can be invoked from both Python and C code. The development version of code offers also a hybrid OpenMP/MPI implementation.

In traditional HPC systems, the overhead from Python is typically on the order of a few percents [10]. First tests for running GPAW natively on the Intel Xeon Phi coprocessor have indicated that overheads associated with the Python parts can become prohibitively large compared to standard CPU version. Thus, offloading only the most computationally intensive kernels and libraries appears to be the most promising way to use the Intel Xeon Phi coprocessor.

The success of GPAW's GPU implementation is largely due to fact that it is possible to allocate large arrays on the accelerator in the early phase of calculation. These array allocations as well as simple manipulations (additions, multiplications, etc.) can be done at the level of the Python code, which helps keep the data transfers between the host and the device to a minimum.

pyMIC offers a natural way to follow a similar approach also with Intel Xeon Phi coprocessor. As `offload_array` resembles the Numpy `ndarray`, a large part of the high-level algorithms and code can be kept intact. The choice of actual kernels and the placement of data (host for `ndarray` and coprocessor for `offload_array`) can be made at a relatively low level, which allows one to utilize offloading with only a few of code changes. Fig. 10 shows an example of high-level GPAW usage: when creating arrays with `gd.zeros`

```
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)
```

Fig. 10. Example of high level GPAW code utilizing pyMIC.

```
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.associate(array)
          else:
               return array
```

Fig. 11. Example of array creation function in GPAW.

one designates that an array should be an `offload_array` by using an additional argument. Later on, the high-level code remains the same as for NumPy arrays, e.g. calls to `gd.integrate` do not need to be changed, but at a lower level the actual operation is performed on the coprocessor. Fig. 11 shows what the lower level array creation routine looks like.

Next, we discuss selected key kernels of GPAW where we have implemented offloading to the coprocessor. The focus is on the real-space grid mode of GPAW which generally offers the best parallelization prospects by the domain decomposition of a real-space grid. A more detailed description about the algorithms used in GPAW can be found in [10], [11], [22].

Especially in larger systems the most computationally intensive operations (which scales on the order of $O(N^3)$ with the number of atoms) are related to subspace diagonalization and orthonormalization which contain integrals of the form

$$O_{nm} = \int dV \tilde{\psi}_n(\mathbf{r})\hat{O}\tilde{\psi}_m(\mathbf{r}), \tag{1}$$

where the $\tilde{\psi}_i$ are the Kohn-Sham wave functions of the density-functional theory and $\hat{O}$ is either the Hamiltonian or the overlap operator. When using uniform real-space grids the integrals can be evaluated as a sum over grid points G

$$O_{nm} = \sum_G \tilde{\psi}_{nG}(\tilde{O\psi})_{mG} dV \tag{2}$$

i.e., as matrix-matrix products. Other types of matrix-matrix products appearing in subspace diagonalization and orthonormalization are

$$\tilde{\psi}'_{nG} = \sum_m O_{nm}\tilde{\psi}'_{mG}. \tag{3}$$

The matrices $\tilde{\psi}_{nG}$ are very skewed as the matrix dimensions depend on the number of electrons and grid points in the simulation. A typical number of of electrons is $n = 60$ up to 2048, while a common grid size G is between 32^3 and 200^3 ($\approx 32,000 - 8,000,000$). One should note that for small number of electrons and grid points, other parts of the algorithm rather than these O(N^3) operations dominate the computational time, while for very large calculations parallelization over tens or hundreds of nodes is needed. For a single Intel Xeon Phi coprocessor representative matrix dimensions are $n = 512$ and $G = 64^3 (\approx 260000)$.

TABLE II. Latency (in milliseconds) of offload operations.

Operation	Latency
`associate` (8 bytes)	0.13
`update_device` (8 bytes)	0.06
`update_host` (8 bytes)	0.10
`invoke_kernel` (empty kernel)	0.05

VI. Performance

To assess the performance of the Python offload module, we ran a series of micro-benchmarks on a node of the Endeavor cluster at Intel [25]. The node is equipped with two Intel® Xeon® E5-2697 processors with 2.70 GHz (turbo mode and Intel® Hyper-Threading enabled) and a total of 64 GB of DDR3 memory at 1867 MHz. The node is equipped with two Intel Xeon Phi 7120P coprocessors with 61 cores at 1238 MHz (maximum turbo upside 1333 MHz) and 16 GB of GDDR5 memory each. The host runs Red Had Enterprise Linux (RHEL) 6.5 (kernel version 2.6.32-358.6.2) and MPSS 3.3.30726 (coprocessor kernel 2.6.38.8+mpss3.3). We use Intel Composer XE 2013 for C/C++ version 14.0.3.174 to compile the extension module. The Python interpreter is the standard CPython interpreter (version 2.6.6) shipped with RHEL. Numpy is version 1.8.2 and has been setup to use the multi-threaded Math Kernel Library shipped with Intel Composer XE.

We determine the performance on a single Xeon processor package (24 threads) and compare it against one coprocessor (240 threads). We leave one core of the coprocessor empty to handle the operating system load and offload data transfers. Affinity has been set to bind each OpenMP thread of MKL to its own hardware thread. This is in line with most cluster configurations in the Top500 list [25]. A 2:2 ratio of processors to coprocessors improves locality and yields optimal offload performance.

We assess the performance of our pyMIC offload infrastructure by a series of select micro-benchmarks to determine latency and bandwidth achieved for the buffer management functions of Section IV-D. Each of the benchmarks runs for

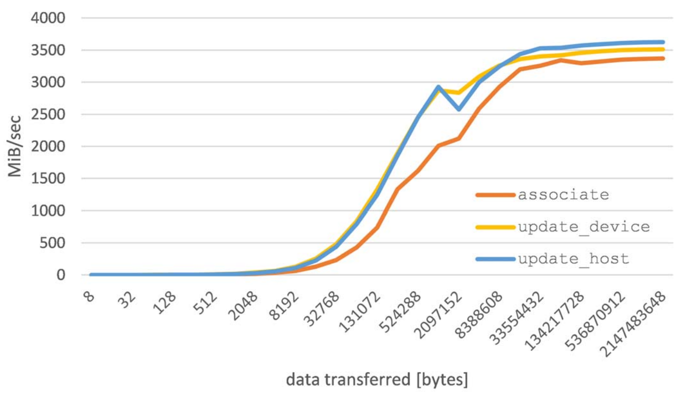

Fig. 12. Bandwidth achieved for copying data to and from the target device.

1,000,000 repetitions to rule out any jitter. Table II lists the latencies of key primitive operations of the pyMIC infrastructure. Associating a Numpy array with one element (8 bytes) on the host with a buffer on the coprocessor takes about 0.13 milliseconds (msec). This time includes the time required on the coprocessor to perform the offload, allocate physical pages on the coprocessor, and transfer 8 bytes (which is negligible). The time of course also includes any overhead of executing our module's Python code and the glue code of the extension module. Similar figures can be observed for the transfer of data to and from the coprocessor. The operations `update_device` and `update_host` are slightly less expensive from a latency perspective because they do not require the allocation of physical memory on the offload device.

Fig. 12 shows the bandwidth achieved for the transfer operations for different data sizes from 8 bytes to 2.1 GiB. For small data sizes, the calling overhead and offload latency dominate the operation and thus the achievable bandwidth stays low. At about 32 MiB of transferred data, bandwidth begins to dominate latency and thus the achieved transfer rate starts to increase. Transfer rates get close to the peak bandwidth that can be achieved with the PCIe bus (gen 2) for large arrays. The `update_device` and `update_host` operations perform slightly better from a bandwidth perspective, because in contrast to `associate` they do not require the allocation of physical pages on the coprocessor.

As our final micro-benchmark, we measured the GFLOP rate of a `dgemm` operation ($C = \alpha AB + \beta C$) in several different settings and different matrix sizes. Without loss of generality, we restrict ourselves to quadratic matrices, because we focus on the offload performance versus host performance. Skewed matrices would expose the same behavior across different sizes owing to the fact that latency and bandwidth are dominating factors for offloading. We measured Numpy's implementation (using MKL) of the `*` operator for its `matrix` class, MKL's `dgemm` routine on the host, and the offloaded version. For all operations, we use $\alpha = 1.0$ and $\beta = 0.0$.

Fig. 13 shows the GFLOPS rates achieved for the matrix-matrix multiplication. Numpy yields approximately 86 GFLOPS for the largest matrices tested. Numpy incurs some additional overhead compared to MKL's `dgemm` operation, because of a temporary matrix that has to be allocated to store the intermediate result $T = AB$ before performing the addition with C. This is effectively avoided by calling the MKL's `dgemm` natively, passing in the floating-points arrays directly. Hence, we can observe that the pure MKL version (219 GFLOPS) performs about twice as well as the Numpy version (86 GFLOPS). The offloaded `dgemm` computation (including transfer overhead) on the Intel Xeon Phi coprocessor achieves 439–538 GFLOPS sustained performance. If data transfers and offload overhead are excluded, performance slightly improves by about 12%. As expected, for very small matrices MKL's multi-threaded `dgemm` on the host gives best results, because the offload and setup overhead cannot be compensated on the target device for these matrix sizes.

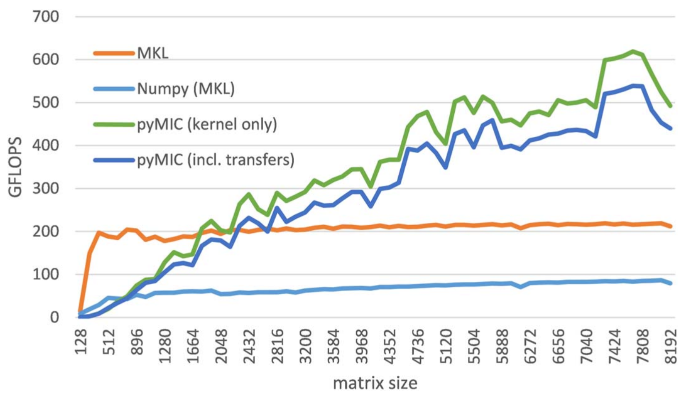

Fig. 13. Performance of the MKL `dgemm` operation.

TABLE III. PERFORMANCE (IN SECONDS) AND SPEEDUP S OF INTEGRATE AND ROTATE OPERATIONS IN GPAW.

	integrate			rotate		
Matrix size	**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

We also investigated the performance of matrix multiplications similar to the operations in GPAW. Table III shows the performance of the integrate operation, Eq. (2), and rotation, Eq. (3), of Section V for problem sizes relevant for single coprocessor usage. Like the pure `dgemm` operation (see Fig. 13), for small data sizes the host performs best, but with larger data sets offloading can offer speedups from 2.5 to 6.8 compared to the host-only execution.

VII. CONCLUSION

In this paper, we have shown the implementation and the performance of a Python module to offload compute kernels from a Python application to the Intel Xeon Phi coprocessor. In line with Python's philosophy of easy-to-use, flexible programming, our module offers a small, yet concise interface to offload kernels to coprocessor devices and to manage data transfers. Our micro-benchmarks indicate that the overhead incurred is low and does not overly affect performance in a negative way. We have also presented how the module can be used in the context of electronic structure simulation software GPAW. For the selected kernels, offloading to a coprocessor with the help of the pyMIC module can speedup the computations by a factor of 2.5-6.8 with large enough data sets. So far, we have implemented offloading only for the two most import kernels of GPAW. As the results are promising, we are also going to implement offloading for other key kernels and investigate how much the full calculation can be accelerated. Furthermore, we plan to utilize multiple coprocessors in the MPI based parallel calculations with GPAW.

The pyMIC project is continuing work and we have a roadmap of features that will be added to the module in the future. All data-transfer operations and kernel invocations are synchronous, that is, the host thread waits for the operation to complete and to return control back to the host thread. We plan to add support for asynchronous operations by returning a handle from the pyMIC operations; the Python code can then use a method of the handle object to wait for completion of the operation. One optimization we would like to implement in the future is to avoid unnecessary copy-in and copy-out operations when Numpy data is passed to `invoke_kernel` as arguments without associating it with `offload_array`. One solution could be to add Fortran-like intent descriptions to the arguments, such that `invoke_kernel` can determine if copy-in, copy-out, or both is requested by the programmer.

Finally, we are working to lift the restriction of having to use C/C++ or Fortran code to implement the offload kernels. We have already started to investigate how to execute fully functional Python code as part of the kernel invocation.

Acknowledgments

Parts of this work were supported by an Intel® Parallel Computing Center Grant.

** 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 "Grizzly Pass" server 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.

References

[1] https://wiki.fysik.dtu.dk/gpaw.

[2] Intel Corporation. Pentium® Processor 75/90/100/120/133/150/166/200, 1997. Document number 241997-010.

[3] Intel Corporation. Intel® Manycore Platform Software Stack (MPSS), 2014. https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[4] Intel Corporation. Intel® SDK for OpenCL™ Applications, 2014. http://software.intel.com/en-us/intel-opencl.

[5] Intel Corporation. Intel® Xeon Phi™ Coprocessor System Software Developers Guide, 2014. Document number 328207-003EN.

[6] Intel Corporation. Reference Manual for Intel® Manycore Math Kernel Library 11.2, 2014. https://software.intel.com/en-us/mkl_11.2_ref.

[7] Intel Corporation. User and Reference Guide for the Intel® C++ Compiler 14.0, 2014. Document number 328222-002US.

[8] Intel Corporation. User and Reference Guide for the Intel® Fortran Compiler 14.0, 2014. Document number 328223-002US.

[9] I. de Jong. Pyro—Python Remote Objects, 2014. http://pythonhosted.org/Pyro4/.

[10] J. Enkovaara, N. A. Romero, S. Shende, and J.J. Mortensen. GPAW - Massively Parallel Electronic Structure Calculations with Python-based Software. *Procedia Computer Science*, 4(0):17 – 25, 2011.

[11] J. Enkovaara, C. Rostgaard, J.J. Mortensen, J. Chen, M. Dułak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H.A. Hansen, H.H. Kristoffersen, M. Kuisma, A.H. Larsen, L. Lehtovaara, M. Ljungberg, O. Lopez-Acevedo, P.G. Moses, J. Ojanen, T. Olsen, V. Petzold, N.A. Romero, J. Stausholm-Møller, M. Strange, G.A. Tritsaris, M. Vanin, M. Walter, B. Hammer, H. Häkkinen, G.K.H. Madsen, R.M. Nieminen, J.K. Nørskov, M. Puska, T.T.Rantala, J.Schiøtz, K.S. Thygesen, and K.W. Jacobsen. Electronic Structure Calculations with GPAW: a Real-space Implementation of the Projector Augmented-wave Method. *J. Phys.: Cond. Matter*, 22(25), 2010.

[12] T. Filiba. RPyC—Transparent, Symmetric Distributed Computing, June 2014. http://rpyc.readthedocs.org/en/latest/.

[13] Python Software Foundation. Extending Python with C or C++, 2014. https://docs.python.org/2/extending/extending.html.

[14] S. Hakala, V. Havu, J. Enkovaara, and R. Nieminen. Parallel Electronic Structure Calculations Using Multiple Graphics Processing Units (GPUs). In Pekka Manninen and Per Öster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, page 63. Springer Berlin Heidelberg, 2013.

[15] Y. Hold-Geoffroy, O. Gagnon, and M. Parizeau. Once you SCOOP, no Need to Fork. In *Proc. of the 2014 Annual Conf. on Extreme Science and Engineering Discovery Environment*, Atlanta, GA, July 2014.

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Runtime Code Generation. *Parallel Computing*, 38(3):157–174, March 2012.

[17] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 2012. available at: http://www.mpi-forum.org.

[18] C.J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire. Offload Runtime for the Intel® Xeon Phi™ Coprocessor. Technical report, Intel Corporation, March 2013. Available at https://software.intel.com/en-us/articles/offload-runtime-for-the-intelr-xeon-phitm-coprocessor.

[19] M. Noack, F. Wende, F. Cordes, and T. Steinke. A Unified Programming Model for Intra- and Inter-Node Offloading on Xeon Phi Clusters. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, November 2014. To appear.

[20] Numpy Developers. NumPy, 2014. http://www.numpy.org/.

[21] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.0, 2013. http://www.openmp.org/.

[22] N.A. Romero, C. Glinsvad, A.H. Larsen, J. Enkovaara, S. Shende, V.A. Morozov, and J.J. Mortensen. Design and Performance Characterization of Electronic Structure Calculations on Massively Parallel Supercomputers: a Case Study of GPAW on the Blue Gene/P Architecture. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.

[23] SciPy Developers. SciPy, 2014. http://www.scipy.org/.

[24] TIOBE Software BV. TIOBE Index for September 2014, September 2014. http://www.tiobe.com/.

[25] top500.org. Top500 Supercomputing Sites, June 2014. http://www.top500.org/.

[26] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *ACM SIGARCH Comp. Arch. News*, 20(2):256–266, May 1992.