



HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

ADVANCED PROGRAMMING

ASSIGNMENT 2

---

**The Library supports running deep  
learning on Intel Xeon Phi**

---

*Member:*

Nguyen Minh Tri  
Pham Duc Minh Chau  
Le Huynh Duy Thai

*Student ID:*

1770026  
1770316  
1770320

January 30, 2018

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduce</b>   | <b>2</b>  |
| <b>2</b> | <b>Intel Xeon Phi</b>  | <b>3</b>  |
| 2.1      | Introduction . . . . .   | 3         |
| 2.2      | Heterogeneous Computing and Clustering . . . . .               | 4         |
| <b>3</b> | <b>Native model</b>  | <b>4</b>  |
| <b>4</b> | <b>Offload programing model</b>                                | <b>6</b>  |
| 4.1      | Offloading Functions . . . . .                                 | 7         |
| 4.2      | Offloading Scope-Local Data . . . . .                          | 8         |
| 4.3      | Data Transfer without Computation . . . . .                    | 9         |
| 4.4      | Data and Memory Persistence Between Offloads . . . . .         | 9         |
| 4.5      | Offloading Global and Static Variables . . . . .               | 10        |
| 4.6      | Memory retention and data persistence on coprocessor . . . . . | 11        |
| 4.7      | Asynchronous Offload . . . . .                                 | 11        |
| 4.8      | Target-Specific Code . . . . .                                 | 13        |
| 4.9      | Optional and Conditional Offload, Fall-Back to Host . . . . .  | 14        |
| 4.10     | Offload Diagnostics . . . . .                                  | 15        |
| <b>5</b> | <b>The pyMic Module</b>  | <b>15</b> |
| 5.1      | Introduction . . . . .   | 15        |
| 5.2      | Basic Interface . . . . .                                      | 16        |
| 5.3      | Implementation of computing function . . . . .                 | 17        |
| <b>6</b> | <b>Experimental result with Chainer</b>                        | <b>18</b> |
| 6.1      | Chainer . . . . .  | 18        |
| 6.2      | Results . . . . .  | 19        |
| <b>7</b> | <b>Reference</b>   | <b>20</b> |

## 1 Introduce

Deep learning plays a vital role in broad spectrum of scientific fields, such as computer vision, speech recognition, natural language processing, and so on. In order to support deep learning, many frameworks are created with the aim of setting up artificial neural networks as quickly as possible. Such frameworks can be run on systems including either Graphical Processing Unit or Intel Xeon Phi the second generation Knights Landing coprocessor. However, very few deep learning frameworks can be run on legacy systems containing Intel Xeon Phi Knights Corner. For that reason, we propose and develop pyMIC-DL which is a NumPy-like library supporting deep learning frameworks run on such legacy systems.

## 2 Intel Xeon Phi

### 2.1 Introduction

- Intel Xeon Phi coprocessors have been designed by Intel Corporation as supplement to the Intel Xeon processor family. The coprocessors feature the Intel manycore architecture, which enables fast and energy efficient execution of some High Performance Computing(HPC) applications. In most Intel communications, the term “manycore”, refers to the architecture of the Intel Xeon Phi product family, while “multicore” architecture refers to the Intel Xeon family processors.



Figure 1: Manycore Intel Xeon Phi



Figure 2: Multicore Intel Xeon

- The manycore architecture may yield more performance per watt of power and per dollar of setup costs than traditional multi-core CPUs. However, not every application can be accelerated by manycore coprocessors. Intel Xeon Phi coprocessors derive their high performance from multiple cores, dedicated vector arithmetic units with wide vector registers, and cached on board GDDR5. High energy efficiency is achieved using low clock speed x86 cores with lightweight design suitable for parallel HPC applications. Therefore, only highly parallel applications supporting vectorized arithmetic with well-behaved (or negligible) memory traffic will thrive on the manycore architecture.

## 2.2 Heterogeneous Computing and Clustering

Programming models for Intel Xeon Phi coprocessors include native execution and offload-based approaches. These approaches enable developers to design a spectrum of hybrid computing models, ranging from multi-core hosted to multi-core-centric to symmetric and manycore-hosted. The choice of work division between the host and the coprocessor is dictated by the nature of the application. Highly parallel, vectorized workloads can be executed on the co-processor as well as on the host. However, serial segments of an application perform significantly better on Intel Xeon processors, and so do applications with stochastic memory access patterns. The overhead of data transport over the PCIe bus should also be taken into consideration.

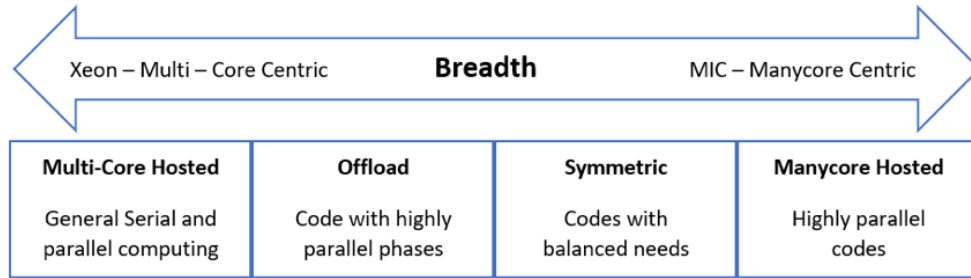


Figure 3: Programming model spectrum

## 3 Native model

Intel Xeon Phi coprocessors run a Linux operating system, with a virtual file system, a multi-user environment, and support for traditional Linux services, including SSH and NFS. These services allow the programmer to run applications directly on an Intel Xeon Phi coprocessor, without the involvement of the host. This does not mean that an application compiled for an Intel Xeon CPU will run on an Intel Xeon Phi coprocessor. Rather, it means that it is possible to compile an application for an Intel Xeon Phi coprocessor from the same source code as the CPU application. Then the executable and its dependent libraries must be transferred to, or shared with, the coprocessor's file system.

To compile a C, C++ code as an executable for the Intel Xeon Phi architecture, Intel compilers must be invoked with the argument `-mmic`.

```
1 #include <stdio>
2 #include <unistd.h>
3 int main(){
4     printf("Hello world! I have %ld logical processors.\n",
5           sysconf(_SC_NPROCESSORS_ONLN ));
6 }
```

Figure 4: This C++ language code (“Native-Hello.cc”) can be compiled for execution on the host as well as on an Intel Xeon Phi coprocessor.

```
user@host% icc hello.c
user@host% ./a.out
Hello world! I have 32 logical cores.
user@host%
```

Figure 5: Compiling and running the “Hello World” code on the host.

We can transfer the executable a.out to the coprocessor and use the shell to run the application on the coprocessor. Running this executable produces the expected “Hello world” output, and the number of logical processors is correctly detected as 240.

```
user@host% icc hello.c -mmic
user@host% scp a.out mic0:~/
a.out 100% 10KB 10.4KB/s 00:00
user@host% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@mic0% ./a.out
Hello world! I have 240 logical cores.
user@mic0%
```

Figure 6: Transferring and running a native application on an Intel Xeon Phi coprocessor.

## 4 Offload programing model

An alternative method is the so-called “offload approach”, where an application begins execution on the host, and at some point it employs the MIC architecture by transferring only some of the data and functions to run on the coprocessor. This process of data and code transfer to the coprocessor is generally called offload, and applications using this procedure are known as offload applications.

- “HelloWorld” in Offload Model

```
1 #include <stdio>
2 #include <unistd.h>
3
4 int main(int argc, char * argv[] ) {
5     printf("Hello World from main()! I see %d logical processors.\n",
6           sysconf(_SC_NPROCESSORS_ONLN ));
7     #pragma offload target(mic)
8     {
9         printf("Hello World from offload! I see %d logical processors.\n",
10              sysconf(_SC_NPROCESSORS_ONLN ));
11     }
12     printf("Bye\n");
13 }
```

Figure 7: “Hello World” in the explicit offload model.

This application must be compiled as a usual host application: no additional compiler arguments are necessary in order to compile offload applications. This code produces the following output:

```
vega@lyra% icpc -o Offload-Hello Offload-Hello.cc
vega@lyra% ./Offload-Hello
Hello World from main()! I see 48 logical processors.
Bye
Hello World from offload! I see 244 logical processors.
```

Figure 8: Output of the execution of Offload-Hello.cc.

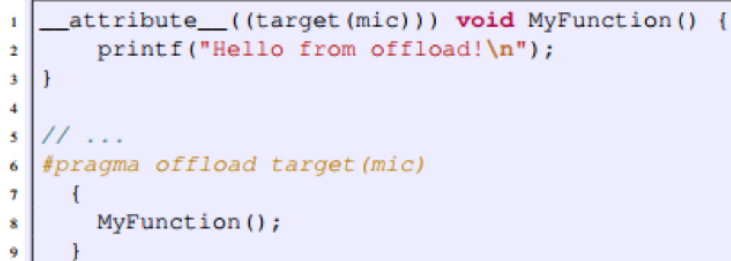
In this context, it is worth mentioning that it is somewhat surprising that the line “Bye” is the second printed line at runtime, even though in the original code it is the third line. At the same time, it should be surprising to the beginner reader that “Hello from offload” was printed at all, because it was output into stdout of

the OS. The output to the coprocessor's stdout is buffered and mirrored in the host console, and the consistency of the order of output is therefore not guaranteed.

In this example, initiating offload with `#pragma offload` is trivial, because all code and all data in the offload region exist only in the scope of `#pragma offload`. However, when functions or data need to be offloaded, offload programming will require more customization of the offload directive.

## 4.1 Offloading Functions

When user defined functions are called in an offload region, they must be declared with the qualifier `__attribute__((target(mic)))`. This qualifier tells the compiler to generate the MIC architecture executable code for the function.



```
1 __attribute__((target(mic))) void MyFunction() {  
2     printf("Hello from offload!\n");  
3 }  
4  
5 // ...  
6 #pragma offload target(mic)  
7 {  
8     MyFunction();  
9 }
```

Figure 9: Offloading a function to an Intel Xeon Phi coprocessor.

If multiple functions must be declared with this qualifier, there is a shorthand way to set and unset this qualifier inside a source file: use `#pragma offload attribute(push/pop)`.



```
1 #pragma offload_attribute(push, target(mic))
2 void MyFunctionOne() { // This function has target(mic) set
3     printf("Hello World from coprocessor!\n");
4 }
5 void MyFunctionTwo() { // The target(mic) attribute is still active
6     fflush(0);
7 }
8 #pragma offload_attribute(pop)
9
10 //...
11 #pragma offload target(mic)
12 {
13     MyFunctionOne();
14     MyFunctionTwo();
15 }
```

Figure 10: Declaring multiple functions with the target attribute qualifier.

## 4.2 Offloading Scope-Local Data

Local scalar variables and arrays of known size are automatically transferred to and from the coprocessor at the start and the end of the offload, respectively. However this behavior can be modified by including them into one of the clauses of the offload pragma: *in*, *out*, *inout*, *nocopy*.

```
1 void MyFunction() {
2     int N = 1000; // Local scalar
3     int data[N]; // Local array of known size
4     #pragma offload target(mic) in(N) out(data)
5     {
6         for (int i = 0; i < N; i++)
7             data[i] = i;
8     }
9 }
```

Figure 11: Offload of local scalars and arrays of known size using `#pragma offload`.

When data is stored in an array referenced by a pointer, the array size is unknown at compile time. In this case, the programmer must indicate the array length in a clause of `#pragma offload`. The length is indicated in array elements and not bytes.

```
1 void MyFunction(const int N, int* data) {  
2   #pragma offload target(mic) in(N) out(data: length(N))  
3   {  
4     for (int i = 0; i < N; i++)  
5       data[i] = 0;  
6   }  
7 }
```

Figure 12: Offload of pointer based arrays of unknown size.

### 4.3 Data Transfer without Computation

If it is necessary to send data to the coprocessor without launching any processing of this data, either the body of the offloaded code can be left blank (i.e., use `{} after pragma offload`), or a special `#pragma offload_transfer` can be used.

```
1 #pragma offload_transfer target(mic) in(N) out(data: length(N))  
2 // The above pragma does not have a body.  
3 // Continuing on the host...
```

Figure 13: Transferring data to the coprocessor without computation.

### 4.4 Data and Memory Persistence Between Offloads

When an array is offloaded to the coprocessor, the following operations are performed:

- (1) Allocate a memory buffer for the array on the coprocessor.
- (2) Copy data from host array to the buffer on the coprocessor.
- (3) Perform offloaded calculations.
- (4) Copy data from coprocessor buffer to the host array.
- (5) Free the memory buffer on the coprocessor.

In some cases, an offload region is called multiple times with the same shape and size of some or all data structures. In this case, step (1) is necessary only in the first offload instance, and step (5) only in the last one.

- To skip the copy-in stage (2), but perform copy-out (4), use the `out` clause.
- To skip the copy-out stage (4), but perform copy-in (2), use `in`.
- To skip both the copy-in and copy-out, use either the `nocopy` clause, or `in` with a length of 0.

- To preserve a memory buffer allocated on a coprocessor, clauses `alloc_if` and `free_if` may be used.

```
1  double *p=(double*)malloc(sizeof(double)*N);
2
3  // Allocate, but not free memory for array p
4  #pragma offload target(mic) in(N) \
5  inout(p : length(N) alloc_if(1) free_if(0))
6  {
7      // ... perform work
8  }
9
10 // Do not allocate, but free memory for array p
11 #pragma offload target(mic) in(N) \
12 inout(p : length(N) alloc_if(0) free_if(1))
13 {
14     // ... perform work
15 }
```

Figure 14: Illustration of memory buffer retention on coprocessor between offloads.

The latter data persistence instruction must always be combined with the memory-retaining clause `free_if(0)` in the previous offload and `alloc_if(0)` in the current offload.

```
1  // Allocate, but not free memory for array p
2  #pragma offload target(mic) in(N) in(p : length(N) free_if(0))
3  {
4      // ... perform work
5  }
6
7  // Do not allocate memory for p, and re-use the data in it
8  #pragma offload target(mic) in(N) nocopy(p : length(N) alloc_if(0))
9  {
10     // ... perform work - same values in p as in first offload
11 }
```

Figure 15: Illustration data persistence on coprocessor between offloads.

## 4.5 Offloading Global and Static Variables

When an offloaded variable is used in the global scope or with the static attribute, it must be declared with the same qualifier as an offloadable function, `_attribute__((target(mic)))`:

```
1 int* __attribute__((target(mic))) data;  
2  
3 void MyFunction() {  
4     static __attribute__((target(mic))) int N = 1000;  
5     data = new int[N];  
6     #pragma offload target(mic) in(N) out(data: length(N))  
7     {  
8         for (int i = 0; i < N; i++)  
9             data[i] = 0;  
10    }  
11 }
```

Figure 16: Offload of global and static variables.

## 4.6 Memory retention and data persistence on coprocessor

```
1 #pragma offload target(mic) in(p : length(N) alloc_if(1) free_if(0) )  
2 { /* allocate memory for array p on coprocessor, do not deallocate */ }  
3  
4 #pragma offload target(mic) in(p : length(0) alloc_if(0) free_if(0) )  
5 { /* re-use previously allocated memory on coprocessor */ }  
6  
7 #pragma offload target(mic) out(p : length(N) alloc_if(0) free_if(1) )  
8 { /* re-use memory and deallocate at the end of offload */ }
```

Figure 17: Memory retention and data persistence.

By default, memory on coprocessor is allocated before, deallocated after offload. Specifiers `alloc_if` and `free_if` allow to avoid allocation/deallocation. Can be combined with `length(0)` to avoid data transfer. Why bother: data transfer across the PCIe bus is relatively slow (6 GB/s), and memory allocation on coprocessor is even slower (0.5 GB/s).

## 4.7 Asynchronous Offload

Offload pragmas are synchronous. It is also possible to initiate asynchronous offload and data transfer in the explicit offload model. Asynchronous data transfer opens additional possibilities for optimization:

- Data transfer time can be masked.
- The host processor and coprocessor can be employed simultaneously.

- It provides a way to distribute work across multiple coprocessors.

Asynchronous data transfer is initiated by adding the specifier `signal(data)` to the offload pragma. After that, another offload pragma with the wait clause, or `#pragma offload_wait` are used to catch the signal of the end of the offload.

```
1 #pragma offload_transfer target(mic:0) signal(data) \  
2     in(N) in(data: length(N)) \  
3 \  
4 // Execution will not block until transfer is completer. \  
5 // The function below will be run concurrently with data transfer. \  
6 SomeOtherFunction(otherData); \  
7 \  
8 #pragma offload target(mic:0) wait(data) \  
9     in(N) nocopy(data: length(N)) out(result: length(N)) \  
10 { \  
11     //...this offload will be launched after the data is transferred \  
12 }
```

Figure 18: Illustration of asynchronous data transfer and wait clause.

With asynchronous offload, `SomeOtherFunction()` will be executed concurrently with data transport. In the second `#pragma` statement, the specifier `wait(data)` indicates that the offloaded calculation should not start until the data transport signaled by data has been completed.

In this following code, two coprocessors are employed simultaneously using asynchronous offloads, the host code execution will wait at this pragma until the transport signaled by data has finished. This pragma is useful when it is not necessary to initiate another offload or data transfer at the synchronization point.

```
1 char* offload0;
2 char* offload1;
3
4 #pragma offload target(mic:0) signal(offload0) \
5   in(N) in(data0 : length(N)) out(result0 : length(N))
6 { // Offload will not begin until data is transferred
7   Calculate(data0, result0);
8 }
9
10 #pragma offload target(mic:1) signal(offload1) \
11   in(N) in(data1 : length(N)) out(result1 : length(N))
12 { // Offload will not begin until data is transferred
13   Calculate(data1, result1);
14 }
15
16 #pragma offload_wait target(mic:0) wait(offload0)
17 #pragma offload_wait target(mic:1) wait(offload1)
```

Figure 19: Illustration of asynchronous offload to different coprocessors.

## 4.8 Target-Specific Code

The values of tuning parameters in HPC algorithms may depend on the amount of memory, cache size, and other parameters, which are different between the host and the coprocessor platforms. These tuning parameters can be multi-versioned using `__MIC__`.

```
1 #ifdef __MIC__
2 const int tileSize = 32; // Use the value 32 for MIC
3 #else
4 const int tileSize = 64; // Use the value 64 for CPU
5 #endif
```

Figure 20: Tuning parameters can be multi-versioned using `__MIC__`.

```
1  #ifdef __MIC__
2  for (int i = 0; i < n; i += 16) { // Intrinsics on MIC
3      __m512 AVec = _mm512_load_ps(A+i);
4      __m512 BVec = _mm512_load_ps(B+i);
5      AVec = _mm512_add_ps(AVec, BVec);
6      _mm512_store_ps(A+i, AVec);
7  }
8  #else
9  for (int i = 0; i < n; i++) { // Same code with automatic
10     A[i] = A[i] + B[i];        // vectorization on the CPU
11 }
12 #endif
```

Figure 21: `__MIC__` can protect functions unavailable on either the host, or the target platform.

## 4.9 Optional and Conditional Offload, Fall-Back to Host

If no coprocessors are found in the system, the offload code can be executed anyway, using the host processor instead of the coprocessor. This can be achieved by adding the clause `optional` to the offload pragma.

```
1  #include <cstdio>
2  #include <unistd.h>
3  int main(int argc, char * argv[] ) {
4      printf("Hello World from main()! I see %d logical processors.\n",
5             sysconf(_SC_NPROCESSORS_ONLN ));
6      #pragma offload target(mic) optional
7      {
8          #ifdef __MIC__
9              printf("Hello from offload on MIC with %d logical processors.\n",
10                     sysconf(_SC_NPROCESSORS_ONLN ));
11          #else
12              printf("Hello from offload on CPU with %d logical processors.\n",
13                     sysconf(_SC_NPROCESSORS_ONLN ));
14          #endif
15      }
16  }
```

Figure 22: Offload-Fallback.cc: handling fall-back to host when offload fails.

This clause is `if`, and it takes one argument. If the argument evaluates to a non-zero value or boolean “true”, the offload will be sent to a coprocessor. If it evaluates to 0 or boolean “false”, the calculation in the scope of the offload pragma will fall back to host CPU execution. `#pragma offload target(mic) if (N>1000)`

Conditional offload can be used, for example, to prevent offload of a problem that is too small to pay off for the offload overhead or to distribute work between coprocessors and the CPU.

## 4.10 Offload Diagnostics

This can be generate diagnostic output for offload applications in the Linux environment variable *OFFLOAD\_REPORT* or the function `_Offload_report`.

- If *OFFLOAD\_REPORT* is unset, no diagnostic output is produced (this is the default behavior).
- *OFFLOAD\_REPORT*=1 prints information on the offload locations (lines of code) and times.
- *OFFLOAD\_REPORT*=2, in addition, produces information regarding the amount of data traffic.
- *OFFLOAD\_REPORT*=3 gives additional details: device initialization and individual variable transfers.

## 5 The pyMic Module

### 5.1 Introduction

The key guiding principle of the design of the pyMIC module is to provide an easy-to-use, slim interface at the Python level. 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.

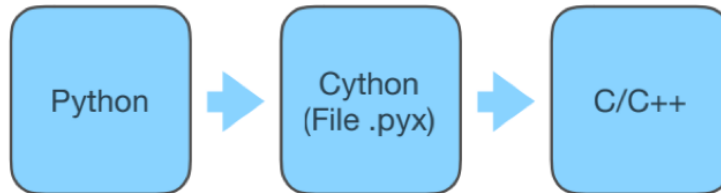


Figure 23: Cython Mechanism.

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.

```

1  import pyMIC as mic
2
3  # acquire handle of offload target
4  dev = mic.devices[0]
5  offl_a = dev.associate(a)
6
7  # load library w/ kernel
8  dev.load_library("libnop.so")
9
10 # invoke kernel
11 dev.invoke_kernel("nop")

```

Figure 24: Simplistic offload example to acquire an offload device and invoke.

```

1  /* compile with:
2     icc -mmic -fPIC -shared -olibnop.so nop.c
3  */
4
5  #include <pymic_kernel.h>
6
7  PYMIC_KERNEL
8  void nop(int argc, uintptr_t argptr[],
9          size_t sizes[]) {
10     /* do nothing */
11 }

```

Figure 25: Empty kernel implementing the “nop” kernel.

## 5.2 Basic 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.

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

Figure 26: Operations of the offload array class.

### 5.3 Implementation of computing function

The layered architecture of the pyMIC module is depicted above. 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`.

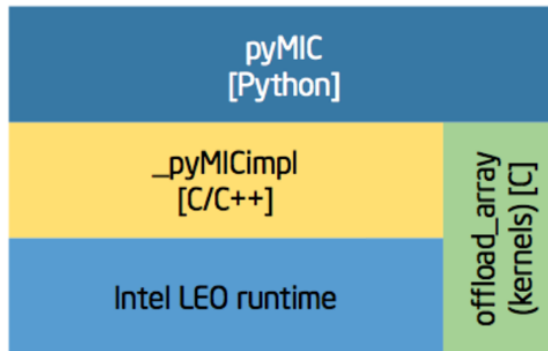


Figure 27: pyMIC Architecture.

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.

|         |                                     |   |
|---------|-------------------------------------|---|
| Group 1 | <i>eq</i>                           | check the equality of two arrays that have the same shapes    |
|         | <i>gt</i>                           | compare whether left value is greater than right value        |
|         | <i>ne</i>                           | check the inequality of two arrays that have the same shapes  |
|         | <i>or</i>                           | to bitwise or each element of two arrays                      |
| Group 2 | <i>abs</i>                          | get absolute values of each element in an array               |
|         | <i>mean</i>                         | get the average of the array elements                         |
|         | <i>sum axis=0</i>                   | return sum of array elements over axis 0                      |
|         | <i>sum axis=1</i>                   | return sum of array elements over axis 1                      |
|         | <i>sum axis=None</i>                | return sum of all array elements                              |
|         | <i>argmax axis=0</i>                | return the indices of the maximum values along axis 0         |
|         | <i>argmax axis=1</i>                | return the indices of the maximum values along axis 1         |
|         | <i>arange</i>                       | return evenly spaced values within a given interval           |
|         | <i>maximum</i>                      | get maximum of array elements                                 |
|         | <i>log</i>                          | get natural logarithm of all elements in an array             |
|         | <i>exp</i>                          | calculate the exponential of all elements in the input array. |
|         | <i>add 2 shape-equal arrays</i>     | add two arrays with the same shapes                           |
|         | <i>add 2 shape-different arrays</i> | add two arrays with different shapes                          |
|         | <i>mul 2 shape-equal arrays</i>     | mul two arrays with the same shapes                           |
|         | <i>mul 2 shape-different arrays</i> | mul two arrays with different shapes                          |
|         | <i>sub 2 shape-equal arrays</i>     | sub two arrays with the same shapes                           |
|         | <i>sub 2 shape-different arrays</i> | sub two arrays with different shapes                          |
| Group 3 | <i>dot</i>                          | matrix multiplication   |

Figure 28: pyMIC Kernels.

## 6 Experimental result with Chainer

### 6.1 Chainer

Chainer is an open source framework designed for efficiently developing deep learning algorithms. Most existing frameworks construct a computational graph in advance of training. This approach is fairly useful, especially for implementing fixed and layer-wise neural networks like convolutional neural networks. However, with the advent of new applications and performance requirements, such as recurrent or stochastic neural networks, we need some kinds of tips to reduce the development efficiency and maintainability of the code for these complex networks. Chainers approach is unique for this by building the computational graph "on-the-fly" during training. It allows users to change the graph at each iteration or for each sample, which depends on conditions. This gives much greater flexibility in the implementation of complex neural networks, which leads in turning to faster iteration, and greater ability to quickly realize cutting-edge deep learning algorithms.

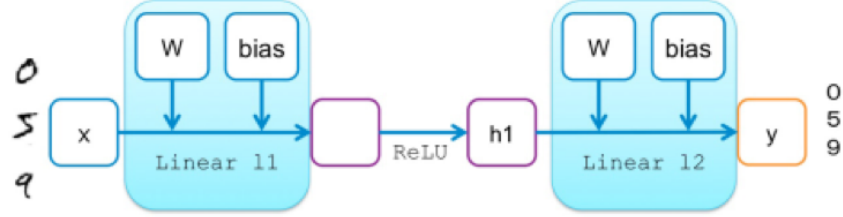


Figure 29: Define neural network for MNIST digit classification.

## 6.2 Results

This library is an extension of original pyMIC that is a Python offloading module for Intel coprocessor. The experimental results show that pyMIC not only outperforms compared with NumPy when considering them on two distinct hardware platforms with nearly the same theoretical peak performance, but also can be highly integrated into one popular deep learning framework Chainer with convincing performance.

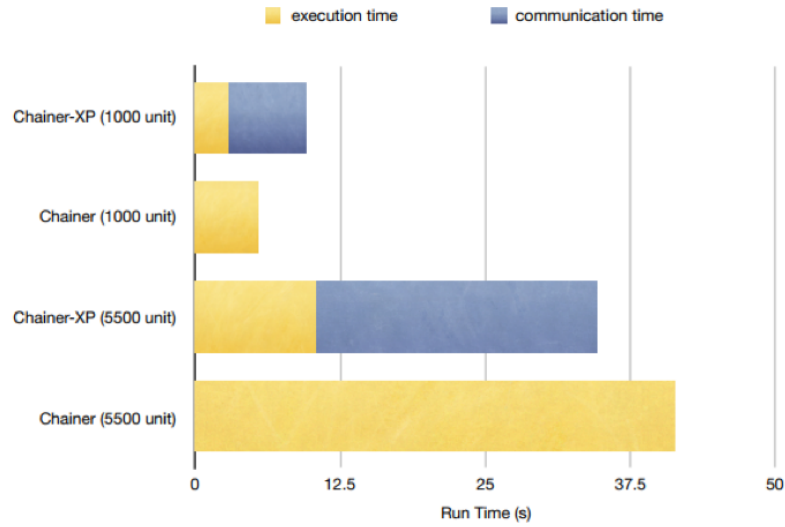


Figure 30: Comparing Chainer running on CPU and Xeon Phi.

## 7 Reference

- Klemm, M., Enkovaara, J.: **pymic: A python offload module for the intel xeon phi coprocessor**. Proceedings of PyHPC (2014)
- Klemm, M., Witherden, F., Vincent, P.: **Using the pymic offload module in pyfr**. arXiv preprint arXiv:1607.00844 (2016)
- Vladimirov, A., Asai, R., Karpusenko, V.: **Parallel Programming and Optimization with Intel Xeon Phi Coprocessors: Handbook on the Development and Optimization of Parallel Applications for Intel Xeon Processors and Intel Xeon Phi Coprocessors**. Colfax International (2015)
- Backpropagation Algorithm - <http://deeplearning.stanford.edu/wiki/index.php/>
- Numpy - Broadcasting - <http://www.numpy.org/>
- Cython Extension - <http://cython.org/>