pyMIC2: A Library for Deep Learning Frameworks Run on the Intel® Xeon Phi™ Coprocessor

Anh-Tu Ngoc Tran(✉), Huu-Phu Nguyen, Minh Tri Nguyen, Thanh-Dang Diep, Nguyen Quang-Hung, and Nam Thoai

Faculty of Computer Science and Engineering
HCMC University of Technology, VNUHCM, Vietnam
{51304672,51302990, nmtribk,dang,nqhung,namthoai}@hcmut.edu.vn,

Abstract. Deep learning plays a vital role in a 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. In addition, very few deep learning frameworks can be run on legacy systems containing Intel Xeon Phi Knights Corner. For that reason, we propose and develop pyMIC2 which is a NumPy-like library supporting deep learning frameworks run on such legacy systems. pyMIC2 that is an extension of offloading module pyMIC implements basic functions so that it can be integrated easily into deep learning frameworks. The experimental findings show that pyMIC2 outperforms NumPy in terms of two hardware platforms with the same theoretical performance. Moreover, pyMIC2 shows it is highly effective when integrating it into a well-known deep learning framework Chainer with very impressive performance. Hence, pyMIC2 is a fairly promising NumPy-like library to facilitate deep learning frameworks run on the legacy systems.

Keywords: computational geometry, graph theory, Hamilton cycles

1 Introduction

Deep Learning has proliferated over the last decade because of its very profound impact on a wide range of several application areas, namely computer vision [22,23], speech recognition [17,18], and natural language processing [16]. In order to put research ideas into practice, advances in deep learning frameworks are a considerable need for the implementation of artificial neural networks (ANNs). Therefore, some frameworks are developed to meet the need, such as Caffe[2], TensorFlow[8], or Theano[10], Torch [13], and so on.

Recent years have witnessed a rapid growth of high performance computing field with the advent of new high-end supercomputers which have incredible

computing power [12]. Most of the supercomputers take advantage of accelerators, such as Graphic Processing Unit (GPU) or Intel Xeon Phi coprocessor. In addition, most researchers and programmers run deep learning applications on systems consisting of GPU. There are just a few deep learning frameworks, like Caffe, TensorFlow, Theano and Neon, which can currently be run on infrastructure including Intel Xeon Phi coprocessor. Unfortunately, the frameworks can only be run on systems including Intel Xeon Phi Knights Landing but not for Knights Corner [3]. Consequently, few deep learning frameworks can be run on several legacy systems, such as Tiane-2 (MilkyWay-2), Thunder, cascade and so forth [12]. This leads to the fact that it is difficult to exploit fully computational power of the underlying hardware platform in case it is not in use. For that reason, the need for developing a novel framework which can be run on such legacy systems is very significant. However, in this paper, our goal is to develop a flexible library which can straightforwardly be integrated into deep learning frameworks rather than a new deep learning one.

pyMIC [7,20] is a Python module for offloading compute kernels from a Python program to the Intel Xeon Phi Knights Corner coprocessor. Currently, it can simply be leveraged in several scientific computing applications, such as the Python-based open source electronic-structure simulation software GPAW [20] and the open-source high-order accurate computational fluid dynamics solver for unstructured grids PyFR [21]. In this paper, we present pyMIC2, the next generation of pyMIC with the aim of developing a library supporting Intel Xeon Phi Knights Corner coprocessor for deep learning frameworks. pyMIC2 is expanded from pyMIC by implementing functions similar to NumPy [15] in order to adapt to deep learning frameworks by dint of flexible and layered characteristics of pyMIC.

pyMIC2 mainly provides a myriad of NumPy-like rudimentary functions, like EQ, OR, ABS, MEAN, SUM, and so on. Furthermore, pyMIC2 improves significantly several existing functions of pyMIC in the performance aspect via vectorization and multithreading mechanism. The functions of pyMIC2 can directly be run on Intel Xeon Phi Knights Corner coprocessor by dint of offloading mechanism. Besides, they facilitate high-level functions used as block buildings in establishment of ANNs, such as activate function, loss function, accuracy function, and so forth.

We evaluate pyMIC2 with regard to two aspects: (1) discrepancy in performance in comparison with NumPy and (2) feasibility of pyMIC2 when integrating it into the well-known deep learning framework Chainer [14,24]. Experimental results demonstrate that pyMIC2 outperforms NumPy in terms of computation time when considering them based on two different hardware platforms with same theoretical performance. Given such effectiveness of performance aspect, we conclude that pyMIC2 is a flexible but effective library which is highly integrated to deep learning frameworks.

In brief, this work makes the following contributions:

– We optimize several existing functions of pyMIC so as to improve program performance.

– We implement pyMIC2 which is extended from pyMIC by means of implementing some NumPy-like functions so that it can facilitate greatly several deep learning frameworks run on the legacy systems containing Intel Xeon Phi Knights Corner coprocessor. Therefore, researchers and programmers can closely integrate pyMIC2 into their interesting deep learning framework.
– We try to integrate pyMIC2 into the well-known framework Chainer and evaluate the effectiveness as well as flexibility of pyMIC2.

The remainder of the paper is organized as follows. The next section reviews in detail the underlying architecture of Intel Xeon Phi Knights Corner coprocessor and the structure of pyMIC module. Section 3 describes the implementation of pyMIC2 based on pyMIC. The experimental results so as to evaluate the effectiveness of pyMIC2 compared with NumPy as well as the practical integration of pyMIC2 and Chainer are elaborated in Section 4. Finally, Section 5 gives some conclusions and the further enhancement of the study.

2 Background

2.1 MIC Architecture

From the perspective of a developer, it is important to know features of MIC architecture if you want to optimize your applications so that it can run efficiently on Intel Xeon Phi. As depicted in Fig. 1 [19], all processors and memory controllers are connected to a Core Ring Interconnect (CRI). Moreover, KNC has NUMA (non-uniform memory access) architecture which means that not all processors have equal access time to all memories.

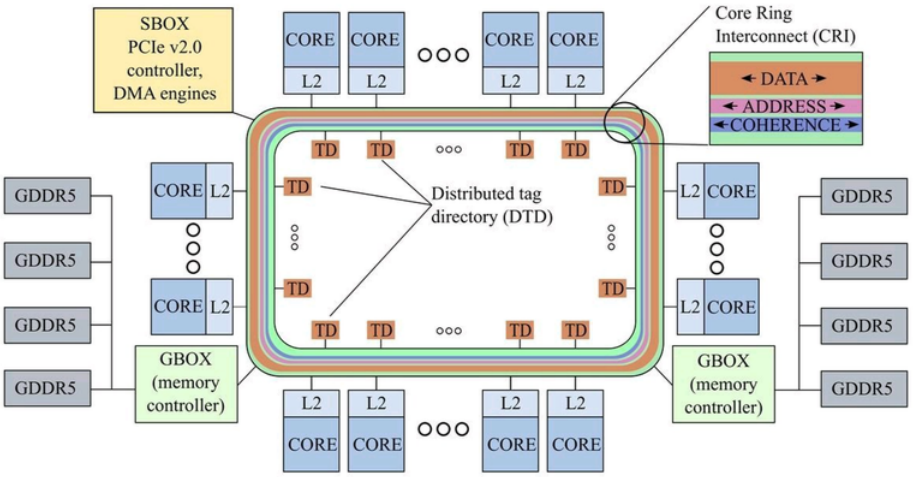


Fig. 1. Knights Corner Architecture

KNC is equipped with 61 cores, each core can launch up to 4 hardware threads run in round-robin order. Vector unit, presents in each core, has 512-bit wide registers (vector registers). This functionality allows SIMD operations on up to 16 single precision floating-point numbers, or up to 8 double precision numbers. In terms of memory, KNC has a two-level cache hierarchy (cache level 1 32KB, level 2 512KB) for each core and from 6GB to 16GB shared memory depending on product series. Furthermore, GDDR5 memory can provide a bandwidth of 325GB/s per coprocessor. Of all hardware specifications mentioned above, theoretical peak performance of KNC is 1200 GFLOPS for double precision and 2400 GFLOPS for single precision.

2.2 PyMIC

For several recent years, Python has emerged as a popular scripting language in HPC thanks to its elegance, ease-to-program and ability to reduce development time and make flexible software. However, its convenience trades off for its performance in comparison in C which is a traditional language used in HPC. In 2012, Intel brought to the market a coprocessor named Intel Xeon Phi with powerful computing ability and energy-efficient feature, but it only supports C/C++ and Fortran language. Therefore, the creation of PyMIC [20] has helped to connect C code and Python code so that users can write Python application executed on Intel Xeon Phi (Fig. 2). To be specific, PyMIC provides its API to invoke functions (also called kernels) written in C to execute on Intel Xeon Phi. Moreover, PyMIC is not only designed to be a bridge between two languages, its API is also compatible to Numpy, which is Python-API library for scientific computing. Data structures of Numpy can work with PyMIC without causing any conflict.

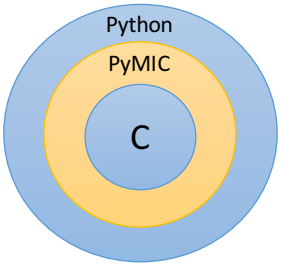


Fig. 2. PyMIC functionality

To provide an easy-to-use interface with slow overhead and full control over data transferring and offloading. The layered architecture of PyMIC is depicted in Fig. 3. In the lowest layer, Intel Language Extension for Offloading (Intel LEO) is responsible for directly interacting with coprocessor through compiler *pragmas* or *directives*. In the higher layer, _pyMICimpl is a Python extension module written in C/C++, and works as a connector between the highest layer

(pyMIC) and the lowest one. To call C functions from pyMIC layer, _pyMICimpl uses Cython mechanism. Besides, the key class of PyMIC is PyMIC also contains offload_array class which is totally compatible with class ndarray of Numpy, and several standard kernels that implement array operations.

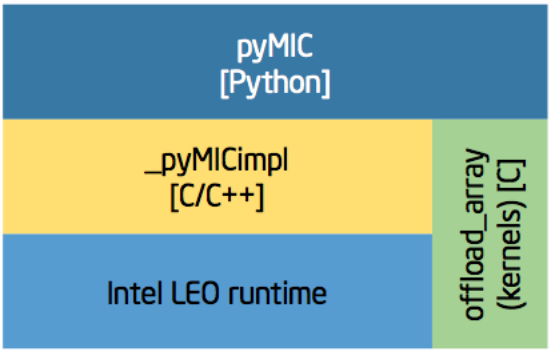


Fig. 3. Architecture of the pyMIC module

To better understand how PyMIC works, we consider the following example. In Fig. 4, first of all to use PyMIC in Python code, we import package **pymic**. PyMIC has In line 5, Xeon Phi card 0 is selected through a global variable **device** to manage all Xeon Phi cards in a node. In the next line, a **stream** from that device is created, it functions as a queue to receive requests from host CPU. Kernel **mydgemm** written in C (Fig. 5) will be compile by C compiler (can be Intel compiler or gcc) into a shared library **libdgemm.so**, it then is offloaded to device by Python API, which is described in line 7. In this example, we demonstrate a general matrix multiplication invoked from Python code. Input matrices are initialized with random values by random function in Numpy package, while output matrix is filled with 0s. However, to do this operation on Xeon Phi, we first have to transfer data to coprocessor. From line 20 to 22, input arrays **a** and **b** are transfered to device by enqueuing a request to **stream**. To be specific, a region of memory will be allocated and data will be copied from host to device. However, with output array **c**, we do not need to copy data to device by specifying **update_device=False**. Transferring data between host and device is very costly; therefore, to achieve better performance, minimizing data movement is necessary. After transferring data to coprocessor, a computing kernel will be invoked by PyMIC API in line 25 and 26. In order to transfer data back to host CPU, member function **update_host()** of **offload_array** object will be called. In addition, all of functions related to data transferring and array manipulation are done completely asynchronously in chronological order of requests put in queue **stream**. Using **sync()** function of **stream** in line 28 to wait until the queue is empty. In summary, Fig. 6 will give an overview of how PyMIC works.

```
1  import pymic
2  import numpy as np
3
4  # select device and load kernel library
5  device = pymic.devices[0]
6  stream = device.get_default_stream()
7  library = device.load_library("libdgemm.so")
8
9  # size of the matrices
10 m, n, k = 4096, 4096, 4096
11
12 # create some input data
13 alpha = 1.0
14 beta = 0.0
15 a = np.random.random(m, k)
16 b = np.random.random(k, n)
17 c = np.zeros(m, n)
18
19 # create offloaded arrays
20 offl_a = stream.bind(a)
21 offl_b = stream.bind(b)
22 offl_c = stream.bind(c, update_device=False)
23
24 # perform the offload and wait for completion
25 stream.invoke(library.mydgemm,
26                offl_a, offl_b, offl_c, m, n, k, alpha, beta)
27 offl_a.update_host()
28 stream.sync()
```

Fig. 4. PyMIC Python code example

```
1  #include <pymic_kernel.h>
2  #include <mkl.h>
3
4  PYMIC_KERNEL
5  void mydgemm(const double *A, const double *B,
6               double *C,
7               const int64_t *m, const int64_t *n,
8               const int64_t *k,
9               const double *alpha,
10              const double *beta) {
11     /* invoke dgemm of MKL's cblas wrapper */
12    cblas_dgemm(CblasRowMajor, CblasNoTrans,
13                CblasNoTrans,
14                *m, *n, *k, *alpha, A,
15                *k, B, *n, *beta, C, *n);
16 }
```

Fig. 5. PyMIC C code example

3 Implementation

Let consider an example, we have an application with four computing functions and only implement 2 kernels written for the first function and the third one to be executed on coprocessor. As described in Fig. 6, to execute the first function, data must be transfered to the coprocessor first and then taken back to host CPU after computation to be input for the second function. This process is repeated for the last two functions. We can see that, this application transfers data to coprocessor two times, and take it back to CPU times. However, data transferring leads to depletion in performance. Therefore, to minimize transferring time in this application, all functions must have corresponding kernels written to be offloaded to coprocessor. With this approach no matter there are how many functions in an application, we only spend one time to send data to coprocessor and one time to take the result back when all functions has finished.

In the previous PyMIC version, it only provides several functions to initialize, change data in array and some simple operations on array such as add, subtract, multiply, and so on. In this paper, we enhance PyMIC by implementing all unit functions that mainly support for deep learning applications with high performance so that users do not have to spend time writing their own kernel. Our kernels are written in C code and parallelized by using OpenMP frameworks to make use of computing power of Intel Xeon Phi. Each unit function is implemented and designed in 3 steps:

– Defining function prototype for C kernel: In this step, we identify parameters needed to be input of a kernel, and their data types. Each parameter is a piece of data transferred to coprocessor; therefore, the number of parameters needs to be chosen wisely to achieve the best performance.

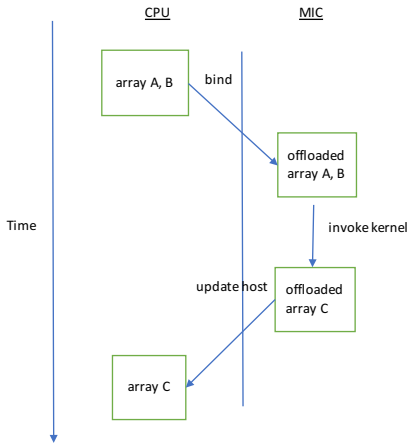


Fig. 6. PyMIC working mechanism

- Implementing function in Python layer: function prototype in Python layer is the same as the one of Numpy so that users can easily use PyMIC in an application using Numpy if they want to execute their application on coprocessor without much modification because their function APIs are the same. The functionality of functions in this layer is to prepare parameters defined in previous step to transfer to kernels in C layer
- Implementing function in C layer: After receiving input data from Python layer, C functions will do computation depending on input data types and return data back to Python layer. These kernels use data parallel algorithms which are implemented using **#pragma omp simd** to take advantage of vector units, and **#pragma omp for** to use up all core in Intel Xeon Phi

4 Evaluation

In this section, we demonstrate the performance of pyMIC2 by comparing functions implemented in pyMIC2 to those of NumPy. We then evaluate MNIST handwritten digit recognition application by using a deep learning framework called Chainer. This application is a classical one in machine learning. The core computing library of Chainer is NumPy, and we want to replace it with pyMIC2 to see how well pyMIC2 works in comparison to NumPy so that we can prove that our approach is practical and efficient enough. Before discussing all of that, we first consider the environment and system on which our experiments are conducted.

4.1 System Setup

In this paper, all of the experiments are conducted on a compute node that has 128 GB RAM, two-way processor Intel Xeon CPU E5-2680 v3 @ 2.50GHz

about 1 TFLOPS and 2 first generation Intel Xeon Phi coprocessors 7120 series connected through PCIe with 1.2 TFLOPS each card.

Furthermore, in order to utilize all computing power of Intel Xeon Phi coprocessor, all threads must be used. However, threads can migrate from one core to another, which is depending on OS scheduling decisions. This leads to performance depletion because migrated threads must fetch data into cache of a new core [25]. Among all of the optimization techniques on Intel Xeon Phi, there is one called thread affinity. We can inhibit thread migration by setting environment variable KMP_AFFINITY to scatter, compact, or several other modes [11], and two of the most popular are scatter and compact. The former is especially good for memory and the latter is for computing.

Moreover, we can improve data transfer performance by using huge memory pages. When data offloaded to coprocessor exceed a threshold value set to MIC_USE_2MB_BUFFERS, memory will be allocated on big 2MB pages, by default the size of pages on Intel Xeon Phi is 4KB. Therefore, we can access more data with less pages, which will decrease page fault rate [5]. There is also a lower allocation cost.

In summary, to achieve the best performance, our system is configured to KMP_AFFINITY=compact, MIC_USE_2MB_BUFFERS=16K. Additionally, all of our benchmarks are run 100 times.

4.2 Evaluation of computing functions in pyMIC2

NumPy with easy-to-use Python API is a big, high performance computing library and has been developed and updated for a decade. In pyMIC2, we only implement several unit functions that mainly support deep learning. We divide these into three group (Table 1):

- The first group is related to logical operations.
- The second group is about arithmetic, exponential and logarithmic functions.
- The final group only consists of 1 function which is matrix multiplication function. This is a one of the important benchmarks in LINPACK used to measure the performance for a given machine.

Input data to pyMIC2 functions can be an 1 or 2 two-dimension arrays or none, which is depending on operations. Size of all the dimensions is equal to N that increases from 128 to 16384. Furthermore, each element of an array is a 64-bit number which can be integer or float. In each benchmark, there are three lines:

- The blue one represents for execution time of a NumPy function.
- The orange one is for pyMIC2 kernel, which is the time only for computing.
- The grey one is for computing time and transferring data.

Group 1 In the first group, there are 4 functions of logical operations. The pymic line is time-consuming because of data transferring through PCIe. The

Table 1. List of computing functions

Group 1	EQ
	GT
	NE
	OR
Group 2	ABS
	MEAN
	SUM axis=0
	SUM axis=1
	SUM axis=None
	ARGMAX axis=1
	ADD 2 shape-equal arrays
	SUB 2 shape-equal arrays
	MUL 2 shape-equal arrays
	ARANGE
	MAXIMUM
	LOG
	EXP
	ARGMAX axis=0
	ADD 2 shape-different arrays
	SUB 2 shape-different arrays
	MUL 2 shape-different arrays
Group 3	MATRIX MULTIPLICATION

other two nearly take the same amount of time to finish the job, when N is approximately smaller than 9000. However, when N gets bigger, pymic kernel takes less time to finish in comparison to numpy.

Group 2 The second group consists of functions of arithmetic, exponent and logarithm. As usual, pymic line of most functions is the slowest, but when N is getting larger, pymic starts to work more efficient and runs faster than numpy. However, there are still some exceptional functions:

- Function ARGMAX axis=0
- Fucntion ARANGE
- Function EXP and LOG

In function ARANGE, pymic line and numpy are approximately the same because it requires no data transferred to coprocessor. Beside function ARANGE, the numpy line of all functions mentioned above is slower than pymic line which includes transferring time. The results of function EXP and LOG can be explained that for exponential and logarithmic functions, Intel Compiler includes a mathematical software library containing highly optimized and very accurate mathematical functions, commonly used in scientific or graphic applications [6].

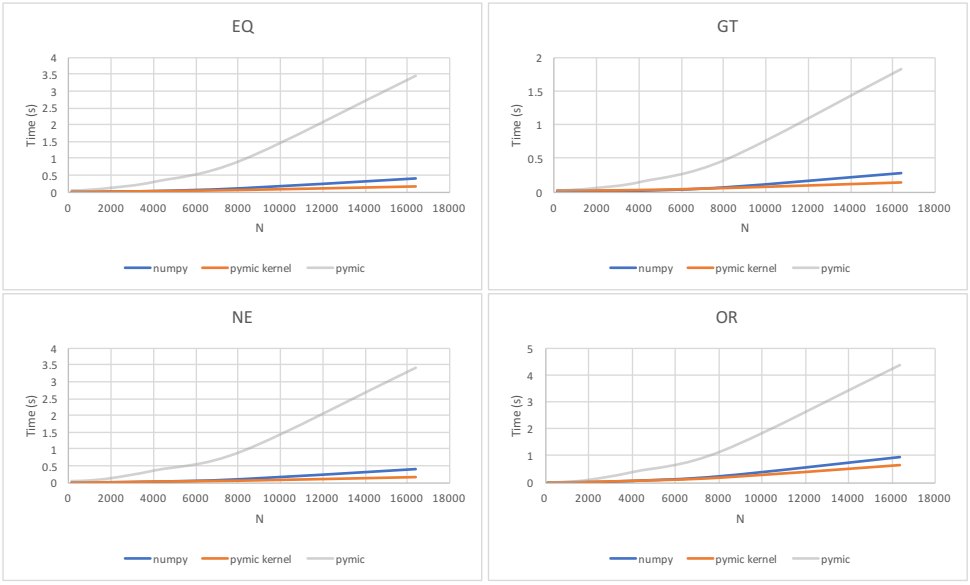
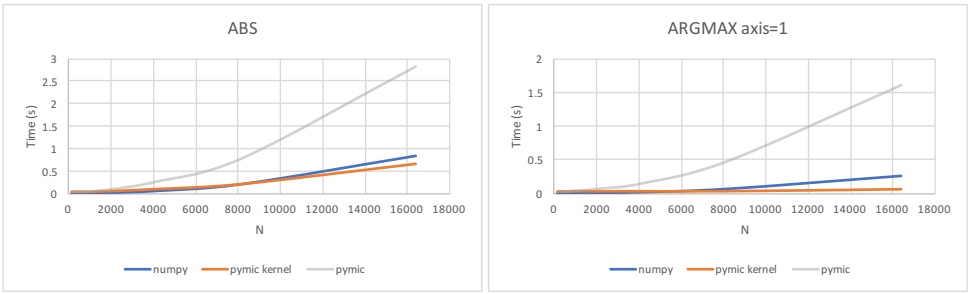
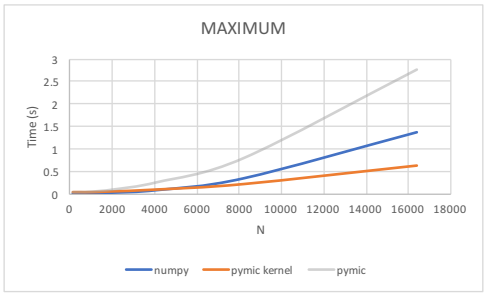
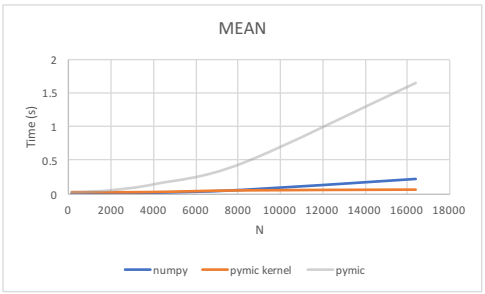
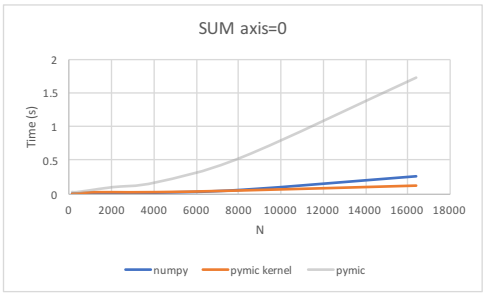


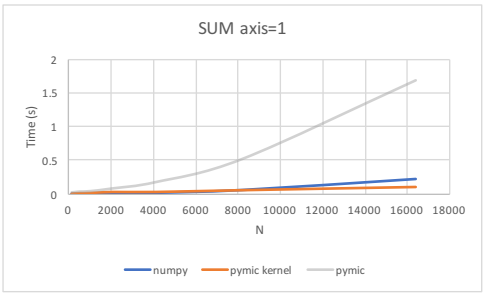
Fig. 7. Benchmark of group 1

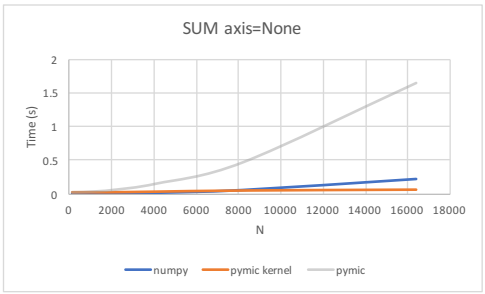


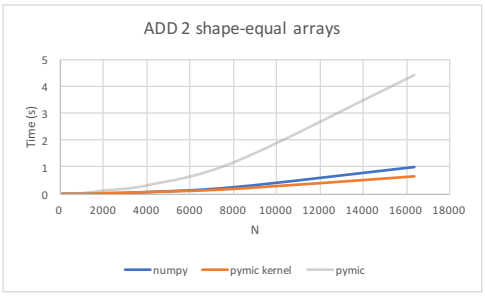


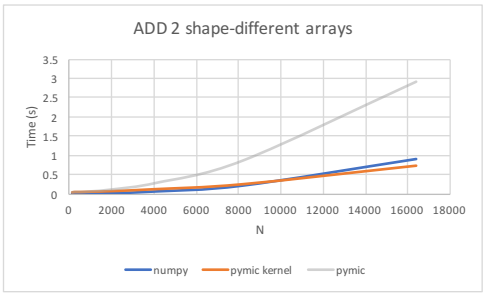

MEAN
Time (s)
N
numpy
pymic kernel
pymic

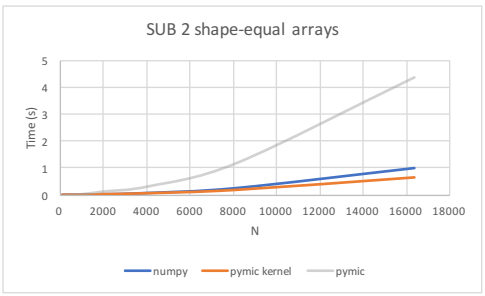

SUM axis=0
Time (s)
N
numpy
pymic kernel
pymic

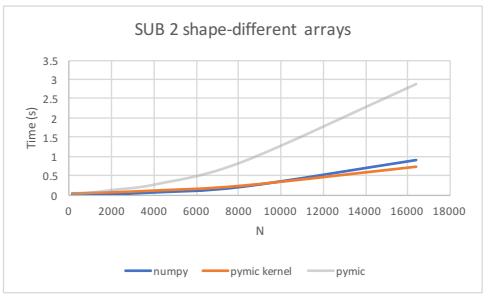

SUM axis=1
Time (s)
N
numpy
pymic kernel
pymic

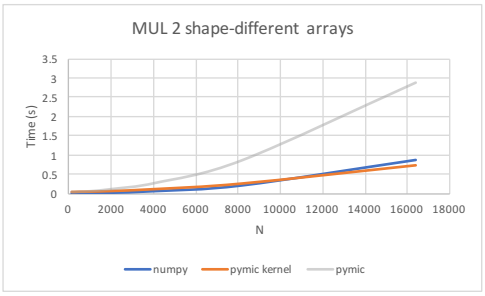

SUM axis=None
Time (s)
N
numpy
pymic kernel
pymic

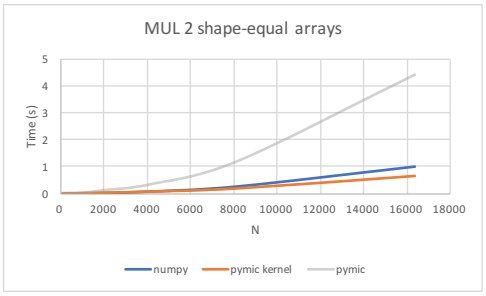

ADD 2 shape-equal arrays
Time (s)
N
numpy
pymic kernel
pymic

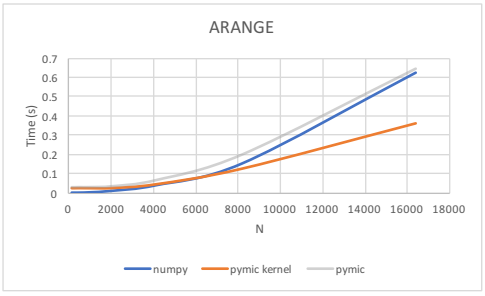

ADD 2 shape-different arrays
Time (s)
N
numpy
pymic kernel
pymic

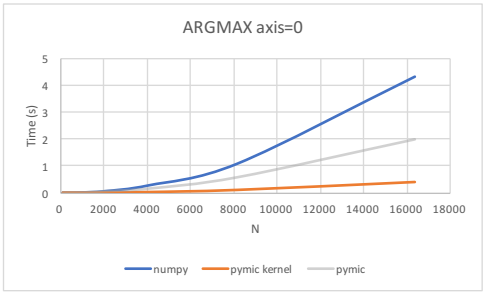

SUB 2 shape-equal arrays
Time (s)
N
numpy
pymic kernel
pymic

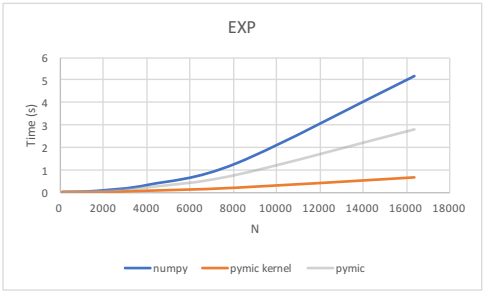

SUB 2 shape-different arrays
Time (s)
N
numpy
pymic kernel
pymic

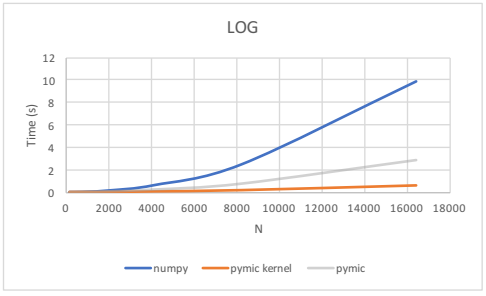

MUL 2 shape-different arrays
Time (s)
N
numpy
pymic kernel
pymic


MUL 2 shape-equal arrays
Time (s)
N
numpy
pymic kernel
pymic


ARANGE
Time (s)
N
numpy
pymic kernel
pymic


ARGMAX axis=0
Time (s)
N
numpy
pymic kernel
pymic


EXP
Time (s)
N
numpy
pymic kernel
pymic


LOG
Time (s)
N
numpy
pymic kernel
pymic

Group 3 In the final group, we use FLOPS (floating point operation per second) for matrix multiplication benchmark instead of time so that we can know how faster this function can achieve in comparison to theoretical performance. This function in pymic will call one of functions of Intel Math Kernel Library (MKL). General matrix multiplication of Intel MKL has been optimized for Intel architecture and proved the efficiency even for small size.

In Fig. 8, we can see that peak performance of numpy, pymic kernel and pymic lines are 574, 849 and 596 GLOPS respectively. For small size matrix, numpy works more efficiently than pymic kernel, but pymic kernel starts to outperform numpy when N is greater than 2500. As pymic kernel, pymic also runs faster than numpy when N is greater than 13000. The computation time of pymic can compensate its transferring time, which leads to better performance than Numpy. Moreover, without including transferring overhead, pymic kernel runs 200 GFLOPS faster than pymic

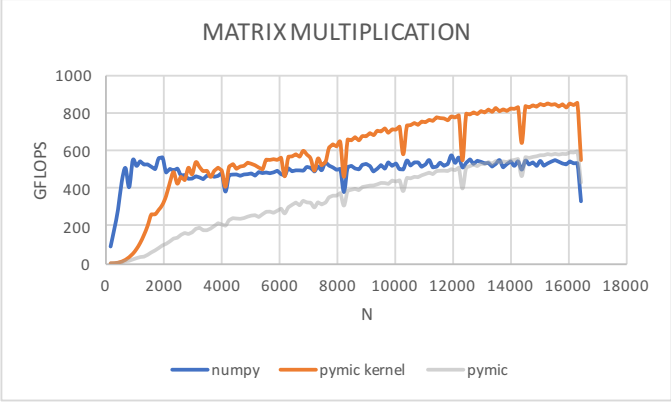


Fig. 8. Matrix Multiplication

4.3 Chainer-MNIST

The intention of this section is to evaluate the performance of a handwritten digit recognition application that uses pyMIC2 to perform parts of computation on Intel Xeon Phi coprocessor. Our experimental evaluation focuses on two aspects. The first one is the different between CPU and coprocessor computing capabilities, the other one is the potential of using pyMIC2 in deep learning applications.

Chainer-numpy: Chainer is a well-known framework for Artificial Neural Network(ANN). One of its advantages is flexibility, which enables its users to create complex architectures simply and intuitively. Similar to TensorFlow, Caffe or

Theano, Chainer uses backpropagation algorithm [1] for training data and adjusting network's parameters. However, by using "Define-by-Run" scheme, the network is defined on-the-fly while running actual forward computation, it makes an network created by Chainer become easier to debug than other frameworks. Therefore, an application written by Chainer integrated pyMIC2 has a faster development time.

Experimental Setup: Our experiments are performed on a combination of CPU Intel Xeon E5 and Intel Xeon Phi 7120P coprocessor. The technical specifications of our system are described in Table 2 . The framework is compiled for coprocessor using Intel compiler version 17.0.4 as well as OpenMP parallel programming API implementation by Intel and MKL [*]version. All measurements are carried out 100 times and averaged to eliminate variances in the resulting measurements.

Table 2. Hardware specifications

Codename	**CPU**	**Coprocessor**
Model	Intel Xeon E5-2680V3	Intel Xeon Phi 7120P
Microarchitecture	Sandy Bridge EP	Intel Many Integrated Core
Clock frequency	2.50/3.30 GHz	1.24/1.33 GHz
Memory Size	512 GB	16 GB
Cache	30.0 MB SmartCache	30.5 MB L2
Max Memory Bandwidth	68 GB/s	352 GB/s
Core/Threads	12/24	61/244

We evaluate our approach using the MNIST [9] dataset of handwritten digits including training set of 60,000 examples, and 10,000 test images, each image contains 28x28 pixel grey levels. The architecture of testing ANNs has 1000, 1500, 2000, 2500 ... 5500 units, each unit represents a node in the network's hidden layer. In order to simplify the performance evaluation process, we just modify the ANNs with one hidden layer, which means our ANNs have only 3 layers which are input, output and hidden. Detailed information of the ANNs used in our evaluation are shown in Table 3.

Table 3. ANN architecture

ANN Configuration	
Layer	3
Batch Size	60000
Connetion Function	Linear
Activation Function	Relu
Loss Function	Softmax Cross Entropy
Gradient Method	Stochastic gradient descent

Result: We perform entire evaluation on Docker [4] in order to easily customize the constraints of system resources. A simple experiment is conducted to ensure that the execution of the ANNs in Docker is similar to physical machine and we obtain the expected result. Thereafter, we start to evaluate the performance of applications, written by two versions of Chainer, one is original version and the other is pyMIC2 integrated version, called Chainer-XP, to execute on Intel Xeon Phi coprocessor.

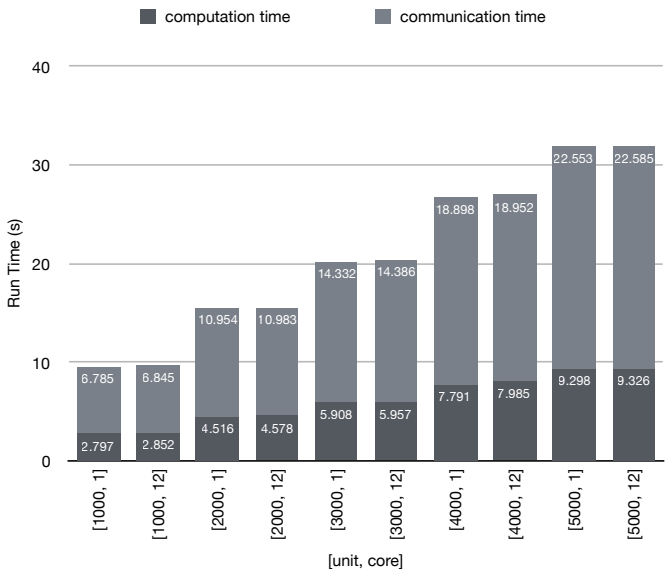


Fig. 9. Chainer-XP sigle core and full core

Results presented for Chainer-XP in Figure 9 show that all experiments of our ANNs training corresponding to the number of units: 1000, 2000, 3000, 4000 and 5000, are executed in roughly equal time periods without being influenced by the number of CPU cores. That also means most of the computational functions have been offloaded into Intel Xeon Phi coprocessor. Moreover, it can be seen that the data communication occupies most of the execution time of training process, overhead is caused by the synchronization between CPU(host) and coprocessor(device) during the calculation. Overall, pyMIC2 is a potential framework which supports its users run ANNs entirely on Intel Xeon Phi. However, the synchronization occupies a prominent role throughout the training process, if it is not tightly controlled, the performance can be devastatingly reduced.

In this experiment, we evaluate pyMIC2's computational capabilities in Intel Xeon Phi. The first ANN (ANN-1) written by original Chainer is executed on a Docker that used full core of a CPU Intel Xeon E5, and the other ANN(ANN-2) written by Chainer-XP on a Docker used only one core combined with Intel

Xeon Phi coprocessor. In detail, both ANNs contain three layers and 1000 units in hidden layer, the network's configuration details are described in the Table 3. Then we measure thoroughly the computation time of all functions during training process and find out that with the second ANN calculation time on coprocessor occupies just approximately 29.2% total run time. More specifically, with a mentioned ANN, the calculation is mainly executed in *dot* function, about 90.2% of computation time as it is called three times each training.

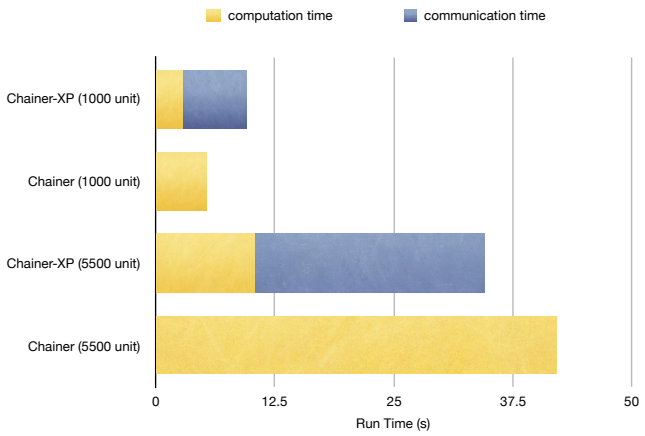


Fig. 10. Chainer-XP single core and 12 core

The biggest matrix size that the *dot* function must perform during the execution of the above network is [784,60000] x [60000,1000], which corresponds to 70.8% total run time used to synchronize data between host and device. Although the purely computation time of ANN-2 on the coprocessor is approximately 1.93 times faster, the total training time was about 1.78 times slower than ANN-1 on the full core CPU. The reason for this drop in performance is transmitting data between hosts and devices repeatedly leading to overlap and redundancy. However, the cause of this result is that the pyMIC2 integration into Chainer has not been optimized, therefore, we can significantly improve the result as pyMIC2 provides objects and functions that strictly support memory allocation and deallocation on Intel Xeon Phi coprocessor. In other words, we must fully understand how Chainer organizes the data so that we can reduce the communication between host and device and increase the performance of training process. As mentioned above, in this paper we just focus on the computing capabilities; thus, in the rest of the experiment, the number of units is raised to 5500 with the aim of increasing the size of the largest matrix to [784,60000] x [60000,5500]. The observed result shows that the ratio between computation time on coprocessor and total training time is almost unchanged at about 30%. In addition, when the size of one of the two *dot* operands expanded 5.5 times,

the computation time on the Intel Xeon Phi increased only 3.8 times, while the CPU run time is grown up to 7.8 times, and total training time achieve 1.2 times faster than CPU. Thus, with several medium configuration CPUs, there is the possibility that we can speed up with Intel Xeon Phi for large-scale computing in ANNs.

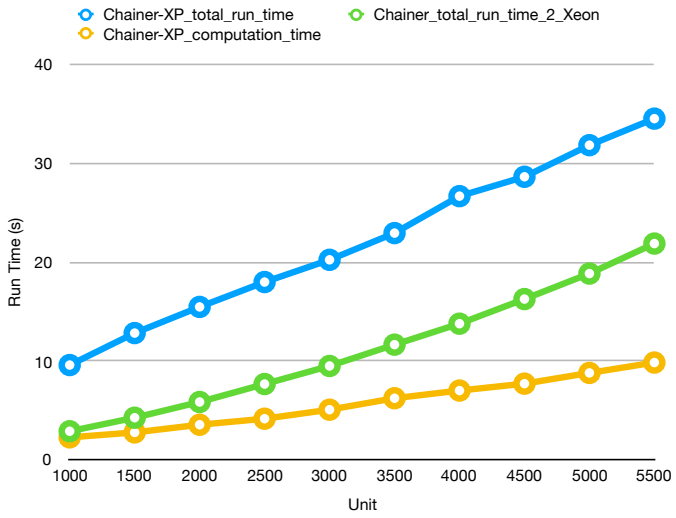


Fig. 11. Chainer-XP vs Chainer

The comparison between two versions of Chainer is illustrated in Figure 11. In order to obtain this experience, The ANN-1 is executed entirely on CPU with a Docker containing dual-processor Intel Xeon E5, which has computing capability approximately 960 GFLOP/s, while the peak performance of an Xeon Phi coprocessor is approximate 1 TFLOP/s **??** in double precision. Although they almost have the similar processing power, computation time of an ANN integrated pyMIC2 is getting faster and faster than this ANN which runs in CPU. Hence, It can be seen that pyMIC2 has used the Xeon Phi's SIMD mechanism, as well as Vectorization technique of Intel Xeon Phi quite effectively in calculating sequential elements.

5 Discussione

discussion.tex

6 Conclusions and Future Work

In this article, we propose and develop pyMIC2 which is a NumPy-like library for deep learning frameworks run on Intel Xeon Phi Knights Corner coprocessor.

This library is an extension of pyMIC that is a Python offloading module for Intel Xeon Phi Knights Corner coprocessor. The experimental results show that pyMIC2 not only outperforms compared with NumPy when considering them on two distinct hardware platforms with the theoretical performance, is but also able to be highly integrated into one popular deep learning framework Chainer with convincing performance.

However, pyMIC2 still contains several limitations. First, we initially implement some fundamental functions which are sufficient to integrate it into Chainer in order to run ANNs with MNIST dataset. There are still enormous functions that need to be implemented to be capable of running other networks. As a result, we intend to implement other NumPy-like functions so as to apply to other deep learning networks. Second, pyMIC2 can currently be run on at most one Intel Xeon Phi Knights Corner coprocessor and hence, it does still not exploit fully systems containing more than one Intel Xeon Phi Knights Corner coprocessor. For that reason, we intend to enhance pyMIC2 so that it is able to be run such systems. Eventually, computation loads are performed completely on only Intel Xeon Phi coprocessor when using pyMIC2. Central Processing Units (CPUs) have still not been utilized to share the burden of on computation. Therefore, we intend to improve pyMIC2 so that it can be run on both CPUs and the coprocessor simultaneously to boost program performance.

References

1. Backpropagation algorithm. http://deeplearning.stanford.edu/wiki/index.php/Backpropagation_Algorithm
2. Caffe. http://caffe.berkeleyvision.org
3. Deep learning frameworks for intel xeon phi. https://software.intel.com/en-us/ai-academy/frameworks
4. Docker: Build, ship, and run any app, anywhere. https://www.docker.com
5. How to use huge pages to improve application performance on intel xeon phi coprocessor. https://software.intel.com/sites/default/files/Large_pages_mic.pdf
6. Overview: Intel math library. https://software.intel.com/en-us/node/522653
7. pymic: A python offload module for the intel(r) xeon phi(tm) coprocessor. https://software.intel.com/en-us/articles/pymic-a-python-offload-module-for-the-intelr-xeon-phitm-coprocessor
8. Tensorflow. https://www.tensorflow.org
9. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist
10. Theano. http://deeplearning.net/software/theano
11. Thread affinity interface. https://software.intel.com/en-us/node/522691
12. Top500 supercomputing sites. https://www.top500.org
13. Torch: A scientific computing framework for luajit. http://torch.ch
14. Chainer: a deep learning framework. https://chainer.org (2017)
15. Numpy. http://www.numpy.org (2017)
16. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. Journal of Machine Learning Research 12(Aug), 2493–2537 (2011)

17. Ding, W., Wang, R., Mao, F., Taylor, G.: Theano-based large-scale visual recognition with multiple gpus. arXiv preprint arXiv:1412.2302 (2014)
18. Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al.: Deep speech: Scaling up end-to-end speech recognition. arXiv preprint arXiv:1412.5567 (2014)
19. Jeffers, J., Reinders, J.: Intel xeon phi coprocessor high performance programming, 2013. Journal of Computer Science & Technology
20. Klemm, M., Enkovaara, J.: pymic: A python offload module for the intel xeon phi coprocessor. Proceedings of PyHPC (2014)
21. Klemm, M., Witherden, F., Vincent, P.: Using the pymic offload module in pyfr. arXiv preprint arXiv:1607.00844 (2016)
22. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
23. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge. International Journal of Computer Vision 115(3), 211–252 (2015)
24. Tokui, S., Oono, K., Hido, S., Clayton, J.: Chainer: a next-generation open source framework for deep learning. In: Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS). vol. 5 (2015)
25. 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)