An Improvement of PyMIC for Deep Learning Appication on Intel Xeon Phi

Anh-Tu Ngoc Tran(✉), Phu, Tri, Thanh-Dang Diep[0000−0003−1163−7464], Hung, and Nam Thoai

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

Abstract. The abstract should summarize the contents of the paper using at least 70 and at most 150 words. It will be set in 9-point font size and be inset 1.0 cm from the right and left margins. There will be two blank lines before and after the Abstract. ...
abstract.tex

Keywords: computational geometry, graph theory, Hamilton cycles

1 Introduction, 2 pages

With this chapter, the preliminaries are over, and we begin the search for periodic solutions to Hamiltonian systems. All this will be done in the convex case; that is, we shall study the boundary-value problem
introduction.tex

2 Background, 2 pages

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 your can see in figure 1 [1], 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.

KNC is equiped 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 peformance of KNC is 1200 GFLOPS for double precision and 2400 GFLOPS for single precision.

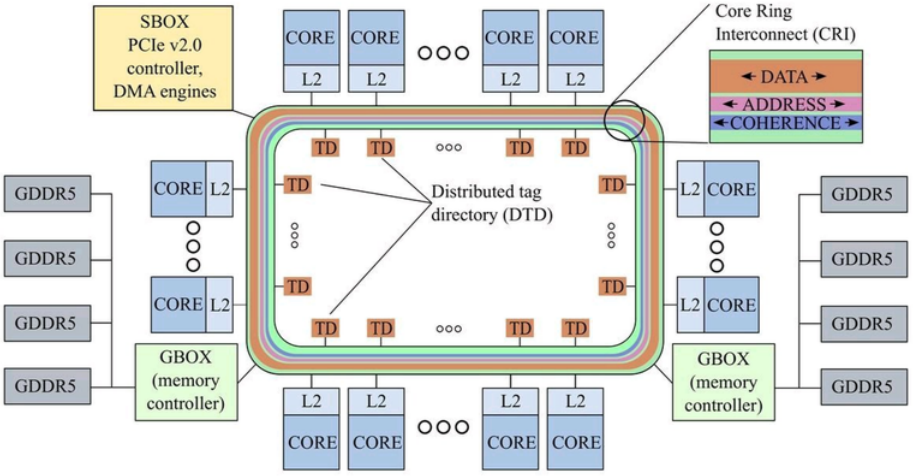


Fig. 1. Knights Corner Architecture

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 coprocess 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 [ ] has helped to connect C code and Python code so that users can write Python application executed on Intel Xeon Phi (figure 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 compatable to Numpy, which is Python-API library for scientific computing. Data structures of Numpy can work with PyMIC without causing any conflict.

To provide an easy-to-use interface with slow overhead and full control over data transfering and offloading. Layered architecture of PyMIC is depicted in figure 3. In the lowest layer, Intel Language Extension for Offloading (Intel LEO) is responsible for directly interacting with coprocesser 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 merchanism. Besides, the key class of PyMIC is PyMIC also contains offload_array class which is totally compatable with class ndarray of Numpy, and several standard kernel that implement array operations.

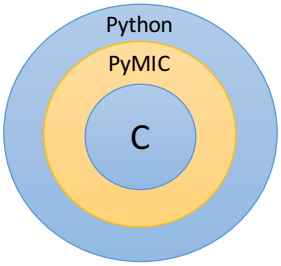


Fig. 2. PyMIC functionality

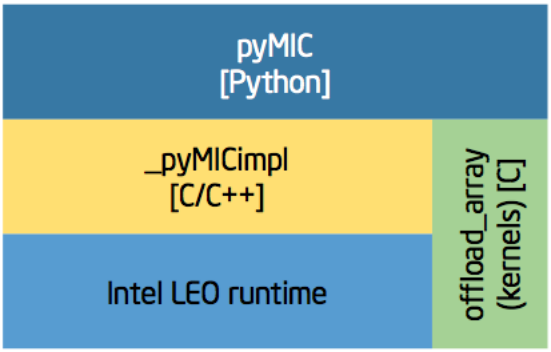


Fig. 3. Architecture of the pyMIC module

To better understand how PyMIC works, we consider the following example. In figure 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 varible **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 (figure 5) will be compile by C compilier (can be Intel compliler 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 transfer to device by enqueueing 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 specifiying **update_device=False**. Transfering data between host and device is very costly; therefore, to achieve better performance, minimizing data movement is necessary. After transfering 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 addtion, all of funtions related to data transfering 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, figure 6 will gives you 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

3 Implementation, 2 pages

Let consider an example, we have an application with four computing functions and only implemnet 2 kernels written for the first function and the third one to be executed on coprpcessor. As we can see in figure, to execute the first function, data must be transfer to coprocessor first and then taken back to host CPU after computation to be input for the second fucntion. This process is

```
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

repeated for the last two functions. We can see that, this application transfer data to coprocessor two times, and take it back to CPU times. However, data transfering leads to depletion in performance. Therefore, to minimize transfering 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 function to initialize, change data in array and some simple operations on array such as add, substract, multiply, and so on. In this paper, we enhance PyMIC by implementing all unit functions that mainly support for deep learning application 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 transfered to coprocessor; therefore, the number of parameter needs to be chosen wisely to achieve best performance.
- Implementing function in Python layer: function prototype in Python layer is the same as the one of Numpy so that user 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 parameter defined in previous step to transfer to kernels in C layer

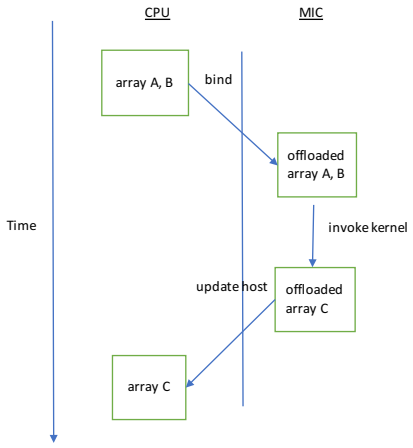


Fig. 6. PyMIC working mechanism

– 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, 10 pages

In this section, we will demonstrate the performance of PyMIC v2 by comparing functions implemented in PyMIC to those of Numpy. We then evaluate hand-written digit recognition application writen by using a deep learning framework called Chainer. This application is a classical one in machine learning. The core computing libray of Chainer is Numpy, and we want replace it with PyMIC to see how well PyMIC works in comparison to Numpy so that we can prove that our approach is practical and effcient 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 computing server 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 coprossers 7120 series connected through PCIe with 1.2 TFLOPS each card.

Furthermore, in order to utilize all computing power of Intel Xeon Phi co-processor, all threads must be used. However, threads can migrate from one core to another, which is depending on OS scheduling decision. This leads to performance depletion because migrated threads must fetch data into cache of

a new core [colfax]. 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[https://software.intel.com/en-us/node/522691], 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 exceeds 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 memory with less pages, which will decrease page fault rate. There is also a lower allocation cost.

[https://software.intel.com/sites/default/files/Large_pages_mic.pdf]

[https://software.intel.com/en-us/mkl-linux-developer-guide-improving-performance-on-intel-xeon-phi-coprocessors].

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 many time , and any jitter will be removed to guarantee the accuracy of all benchmarks.

4.2 Evaluation of computing functions in PyMIC

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

- The first group is related to logical operations.
- The second group is about arithmetic, exponential and logarithmic functions. It is then divided into three subgroup this will be discussed later.
- The final group only consists of 1 funciton which is matrix multiplication function. This is a one of the important benchmark in LINPACK used to measure the performance for a given machine.

Input data to PyMIC functions can be an 1 or 2 two-dimention arrays or none, which is depending on operations. Size of all 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 numpy execution.
- The orange one is for pymic kernel, which is the time only for computing.
- The grey one is for computing time and transfering data.

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

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

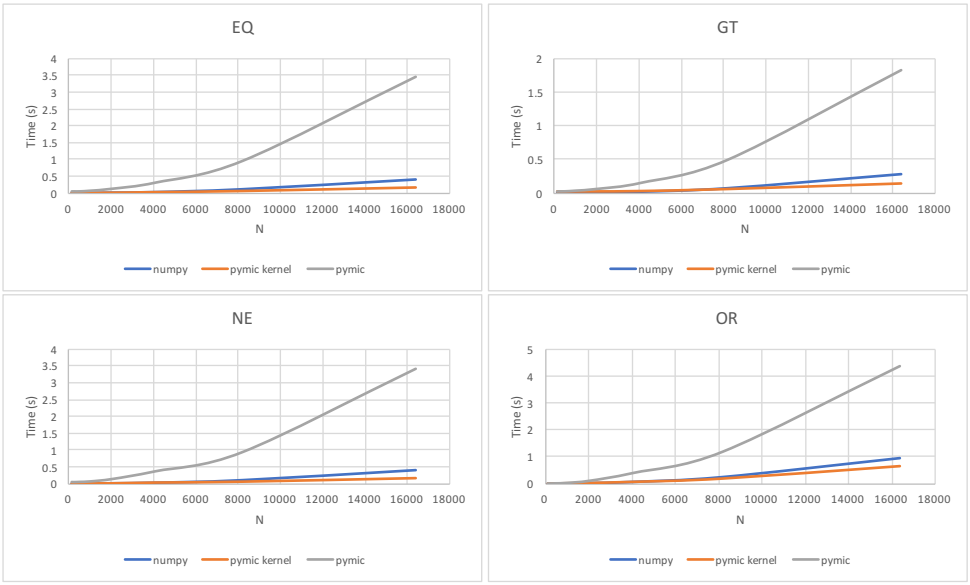
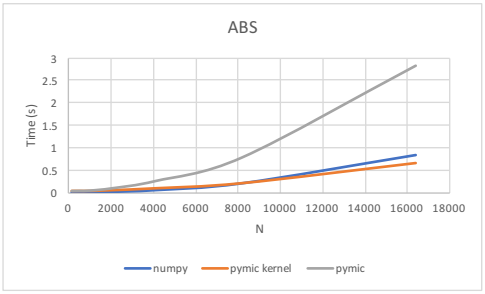
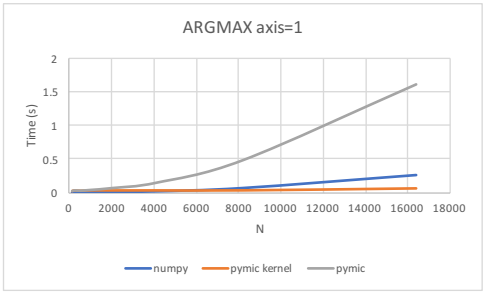
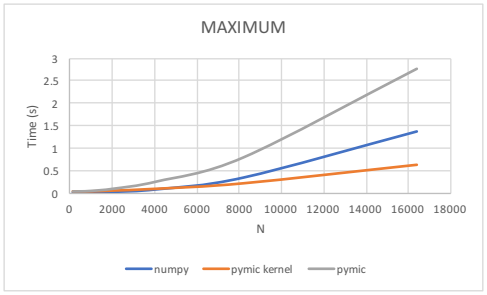


Fig. 7. Benchmark of group 1

Group 2 The second group consists of functions of arithmetic, exponent and logarithm. As usually, 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:





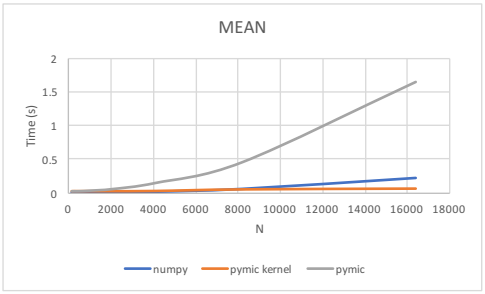


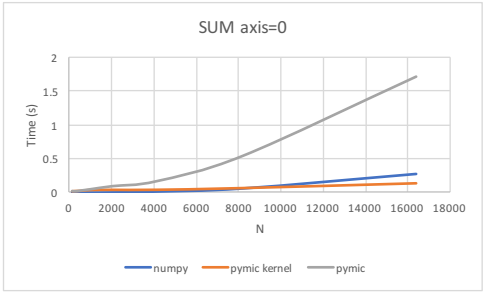
– Function ARGMAX axis=0
– Fucntion ARANGE
– Function EXP and LOG

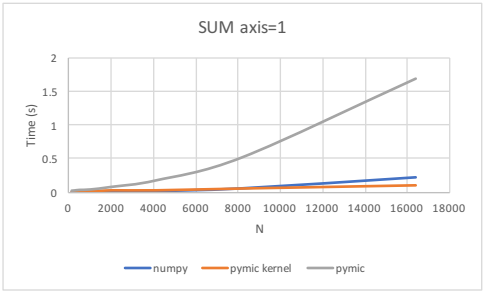
In function arange, pymic line and numpy are approximately the same because it requires no data transfered to coprocessor. Beside function arange, the numpy line of all functions mentioned above is slower than pymic line which includes transfering time. The results of function EXP and LOG can be explained that for functions such as exponent and logarithm, Intel Compiler includes a mathematical software library containing highly optimized and very accurate mathematical functions, commonly used in scientific or graphic applications [https://software.intel.com/en-us/node/522653].

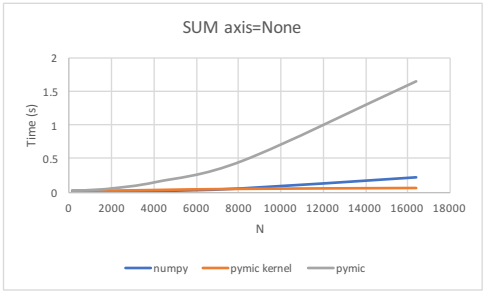
Group 3 In the final group, we use FLOPS (floating point operation per second) for matrix multiplication benchmark instead of time so that we can know how

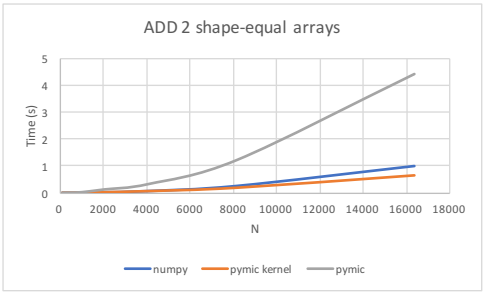
X

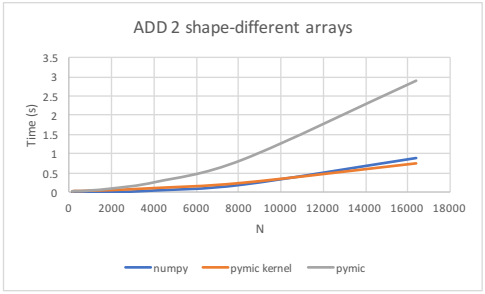

MEAN
Time (s)
N
numpy
pymic kernel
pymic

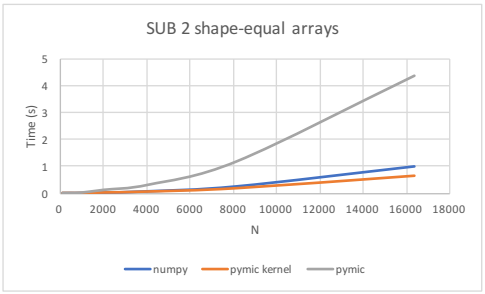

SUM axis=0
Time (s)
N
numpy
pymic kernel
pymic

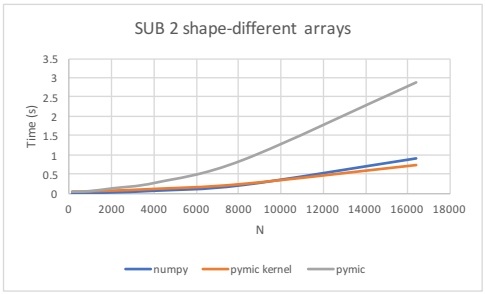

SUM axis=1
Time (s)
N
numpy
pymic kernel
pymic

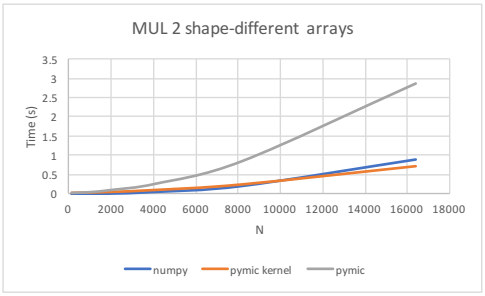

SUM axis=None
Time (s)
N
numpy
pymic kernel
pymic

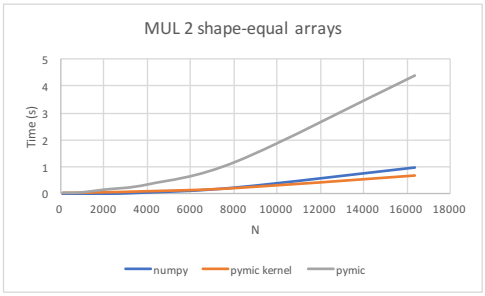

ADD 2 shape-equal arrays
Time (s)
N
numpy
pymic kernel
pymic

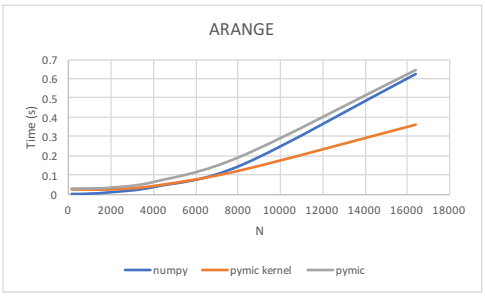

ADD 2 shape-different arrays
Time (s)
N
numpy
pymic kernel
pymic

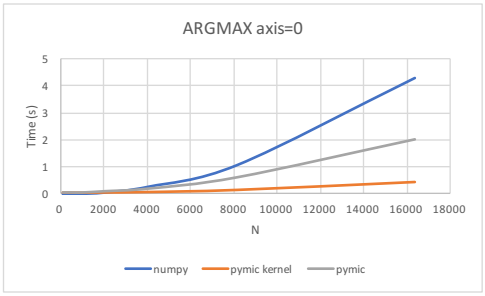

SUB 2 shape-equal arrays
Time (s)
N
numpy
pymic kernel
pymic

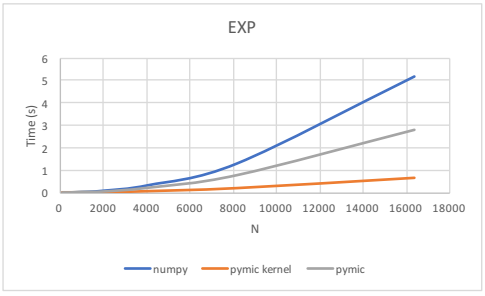

SUB 2 shape-different arrays
Time (s)
N
numpy
pymic kernel
pymic

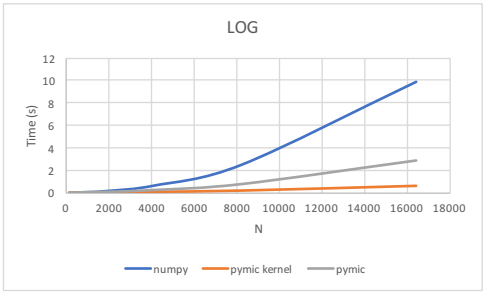

MUL 2 shape-different arrays
Time (s)
N
numpy
pymic kernel
pymic


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

faster this function can achieve in comparison to theoretical performance. This function in pymic will call one 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 figure 8, we can see that peak performance of Numpy, pymic kernel and pymic 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. This also happens the same for pymic. When N is greater than 13000, computation time can compensate transfering time, which leads to better performance than Numpy. Moreover, without including transfering overhead, pymic kernel runs 200 GFLOPS faster than pymic

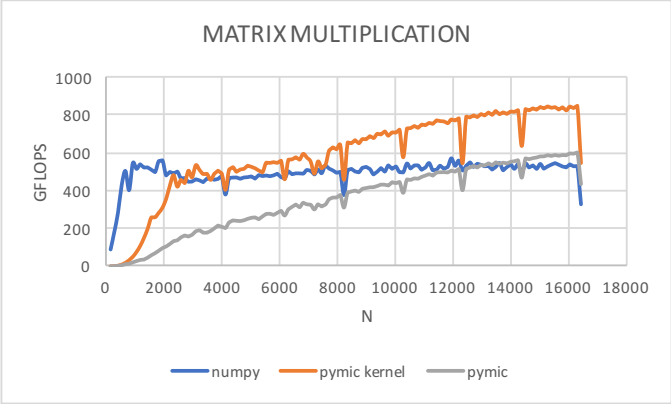


Fig. 8. Matrix Multiplication

4.3 Chainer-MNIST

The intention of this section is to evaluate the performance of a handwritten digit recognition application that uses PyMic 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 PyMic in machine learning and deep learning applications.

Chainer-numpy: Chainer is a well known framework for Artificial Neural Network(ANN). One of its advantages is flexibility, which enable its users to create complex architectures simply and intuitively. Similar to TensorFlow[*], Caffe[*] or Theano[*], Chainer uses back-propagation algorithm [*] 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 make

an network created by Chainer become easier to debug than other frameworks. Therefore, an application written by Chainer integrated Pymic have a faster development time.

Experimental Setup: Our experiments was 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 [*] . The framework was compiled for coprocessor using Intel compiler [*]version as well as OpenMP parallel programming API implementation by Intel and MKL [*]version. All measurements were carried out multiple times and averaged to eliminate variances in the resulting measurements.

Table 2. Technical characteristic of System

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 evaluated our approach using the MNIST [*] dataset of handwritten digits including training set of 60,000 examples, and 10,000 test images, each image contains 28x28 pixel grey levels. The testing ANNs architecture have 1000, 1500, 2000, 2500 ... 5500 unit, each unit represents a node in the network's hidden layer. In order to simply the performance evaluation process, we just modify the ANNs with one hidden layer, that mean 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[*].

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 intend to perform entire evaluation on docker in order to easily customize the constrains of system resources. A simple experiment was conducted to ensure that the execution of the ANNs in docker is similar to physical machine and we obtained the expected result. Thereafter, we started to evaluate the performance application, written by two version of Chainer, one is original version and the other is PyMic integrated version, called Chainer-XP , to execute on Intel Xeon Phi coprocessor.

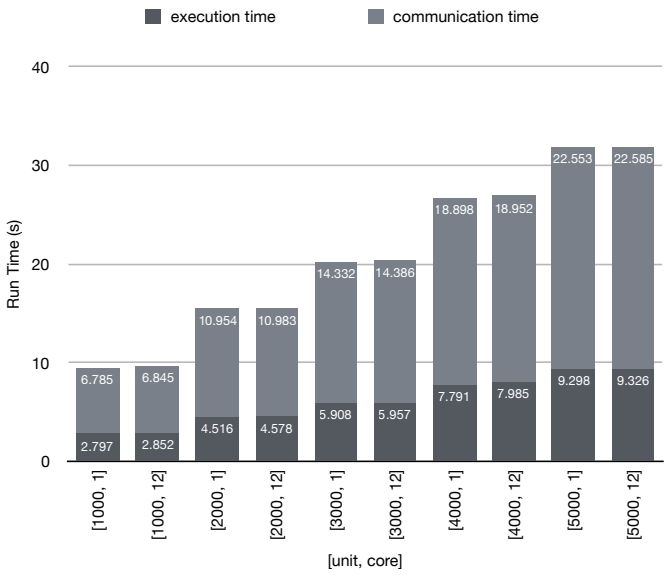


Fig. 9. Chainer-XP 1 core and full core

Results presented for Chainer-XP in Figure [*] show that all experiments of our ANNs training corresponding to number of Units: 1000, 2000, 3000, 4000 and 5000, were executed in roughly equal time periods without being influenced by the number of CPU cores. That also means most of the computational function have been offloaded into Inter Xeon Phi coprocessor. Moreover, it can be seen that the data communication occupies most of execution time of training process, overhead is caused by the synchronization between CPU(host) and coprocessor(device) during the calculation. Overall, PyMic are an potential framework which support its user 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 intend to evaluate PyMic's computational capabilities in Intel Xeon Phi. The first ANN (ANN-1) written by original Chainer was 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 contains three layers and 1000 units in hidden layer, the networks configuration details are described in the Table [*]. Then we measured thoroughly the execution time of all function during training process and found 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 execute in *dot* function, about 90.2% of all time execution as it is called three times each training.

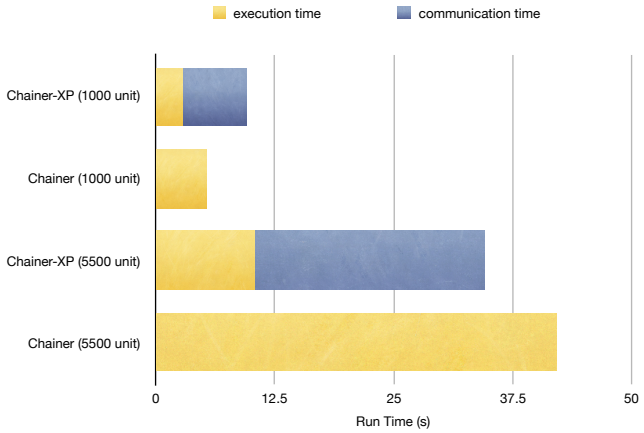


Fig. 10. Chainer-XP 1 core and 12 core

The biggest matrix size that the *dot* function must perform during the execution of the above network is [784,60000] x [60000,1000], which corresponding to 70.8% total run time were use to synchronize data between host and device. Although the purely computing time on the coprocessor was approximately 1.93 times faster, the total training time was about 1.78 times slower than the full CPU. The reason for this drop in performance is data was transmitted between hosts and devices repeatedly leading to overlap and redundancy. However, the caused of this result is the PyMic integration into Chainer has not been optimized, so that the we can significantly improve the result as PyMic provides objects and functions that strictly support allocate and deallocate memory 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, in this paper we just focus on the computing capabilities so in the rest of the experiment, the number of units was raised to 5500 with the aim of increasing the size of the largest matrix to [784,60000] x [60000,5500]. The observed result show that the ratio between execution time on coprocessor and total training time was 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 grown up to 7.8 times, and total

training time achieved 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.

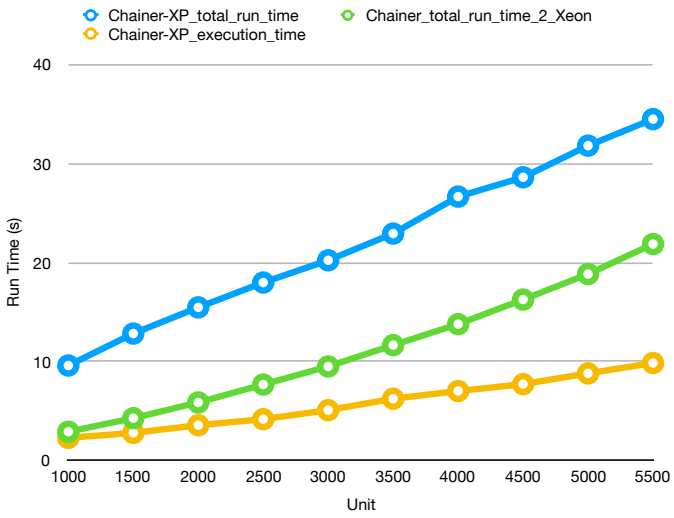


Fig. 11. Chainer-XP vs Chainer

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

5 Discussion, 1page

discussion.tex

6 Conclusions and Future Work, 1 page

conclusion.tex

References

1. Jeffers, J., Reinders, J.: Intel xeon phi coprocessor high performance programming, 2013. Journal of Computer Science & Technology