pyMIC2: A Supporting Deep Learning and Python Offload Module for 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

background.tex

3 Implementation, 2 pages

table of function: name functionality, input, output

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 handwritten 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 coprocessor, 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.

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

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.

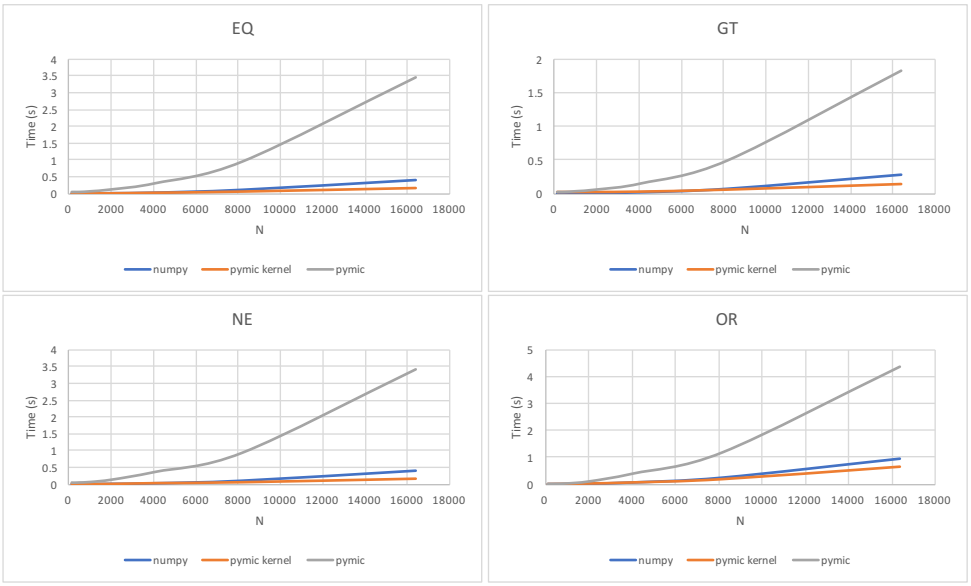
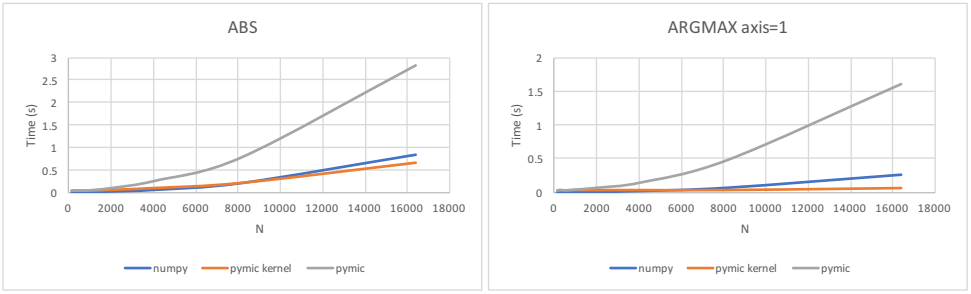
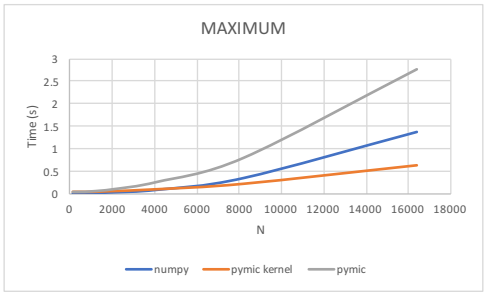
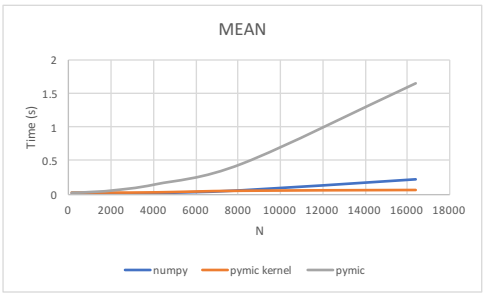


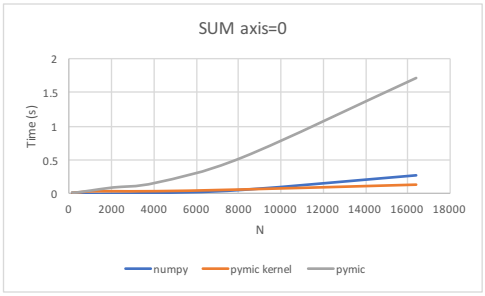
Fig. 1. Benchmark of group 1

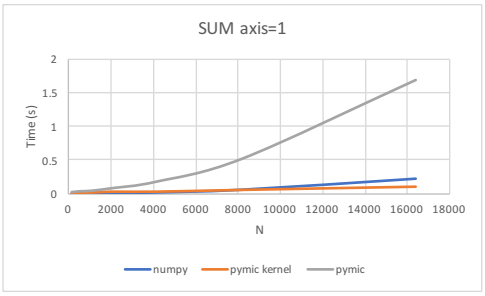
Group 2 The second group consists of functions of arithmetic, exponent and logarithm. As usually, pymic line of most functions is the slowest, but when N is getting larger, pymic starts to work more efficient and runs faster than numpy. However, there are still some exceptional functions:

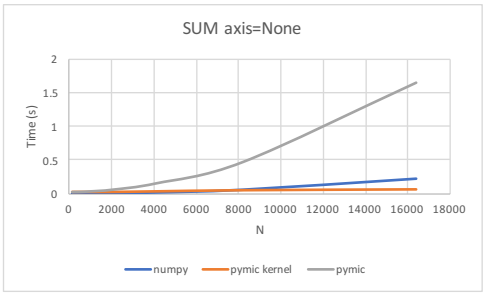


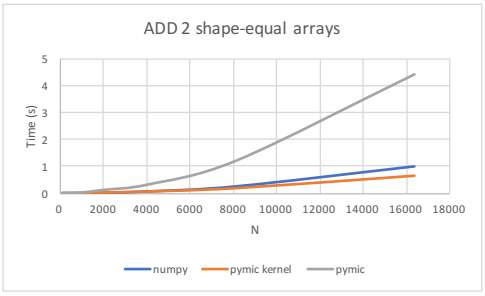

MAXIMUM
Time (s)
N
numpy
pymic kernel
pymic

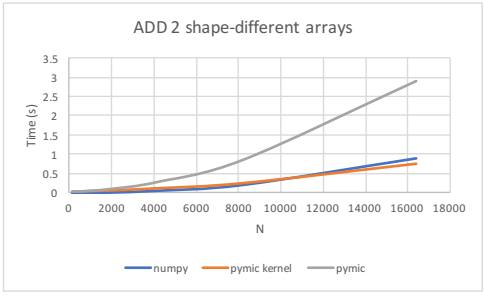

MEAN
Time (s)
N
numpy
pymic kernel
pymic

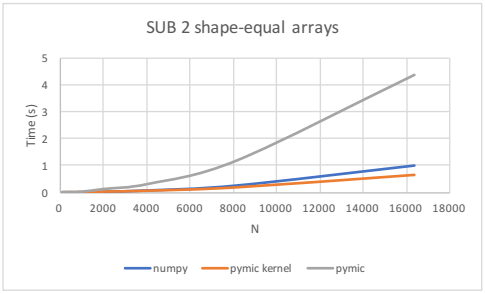

SUM axis=0
Time (s)
N
numpy
pymic kernel
pymic

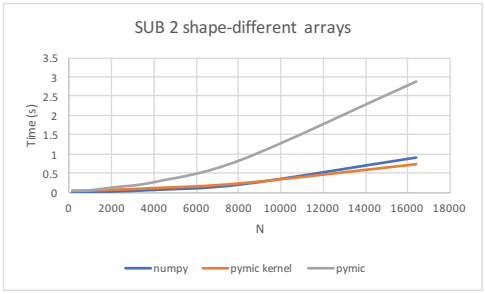

SUM axis=1
Time (s)
N
numpy
pymic kernel
pymic

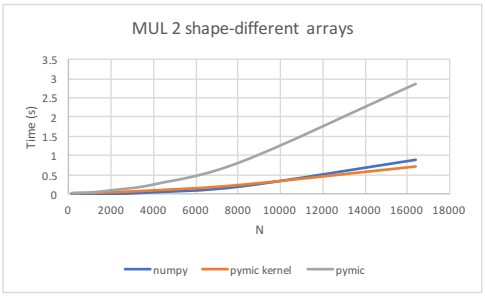

SUM axis=None
Time (s)
N
numpy
pymic kernel
pymic

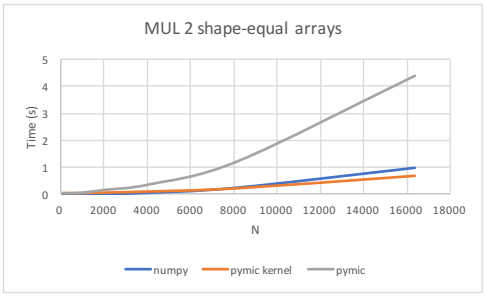

ADD 2 shape-equal arrays
Time (s)
N
numpy
pymic kernel
pymic


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



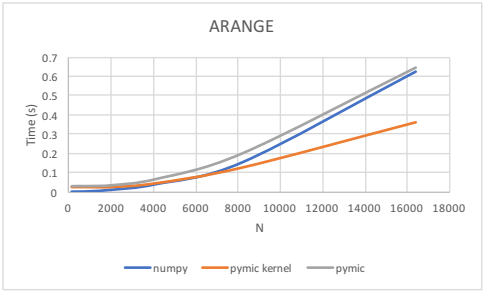


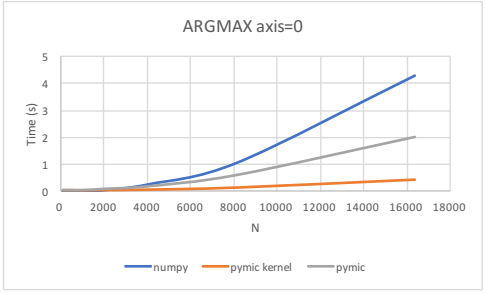


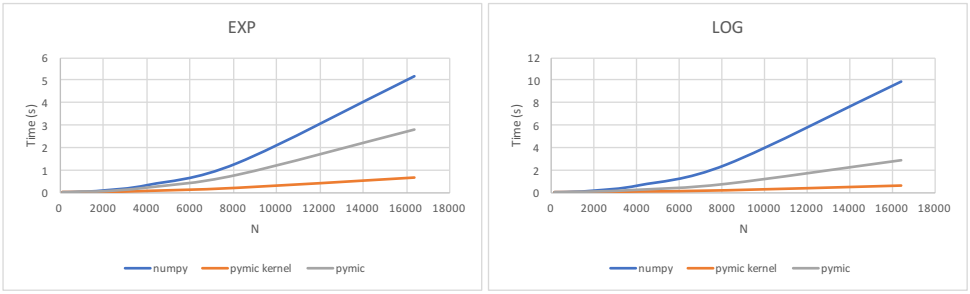


– Function ARGMAX axis=0
– Fucntion ARANGE
– Function EXP and LOG

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 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 2, 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

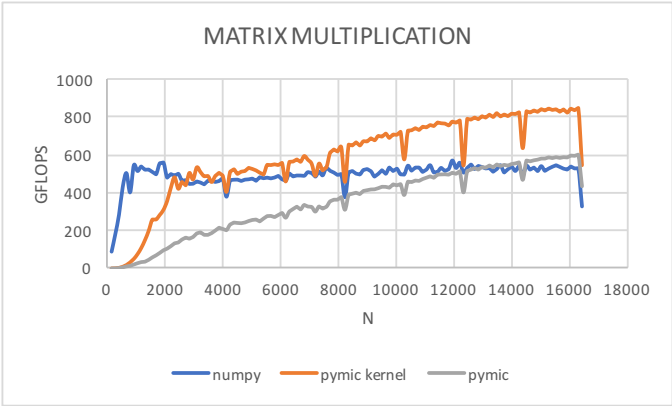


Fig. 2. Matrix Multiplication

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 origin version and the other is PyMic integrated version, called Chainer-XP , to execute on Intel Xeon Phi coprocessor.

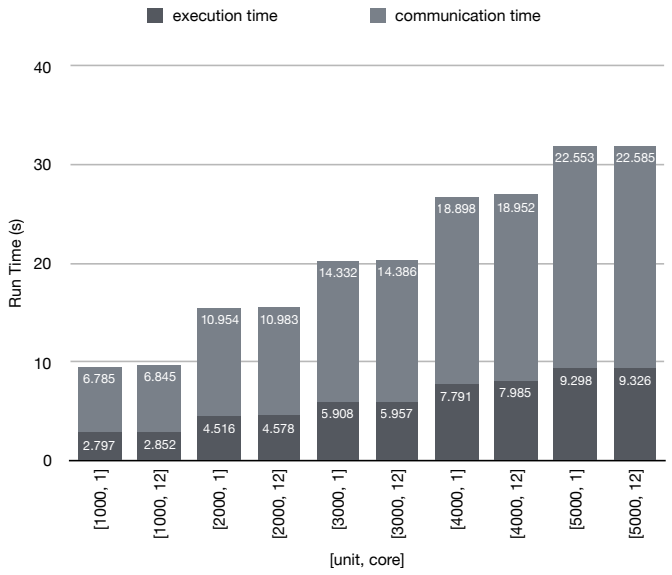


Fig. 3. Chainer-XP 1 core and full core

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 origin 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.

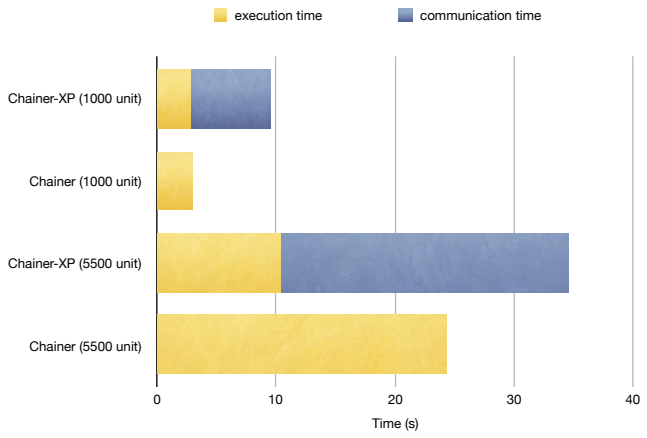


Fig. 4. Chainer-XP 1 core and 12 core

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 ration between execution time on coprocessor an 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.

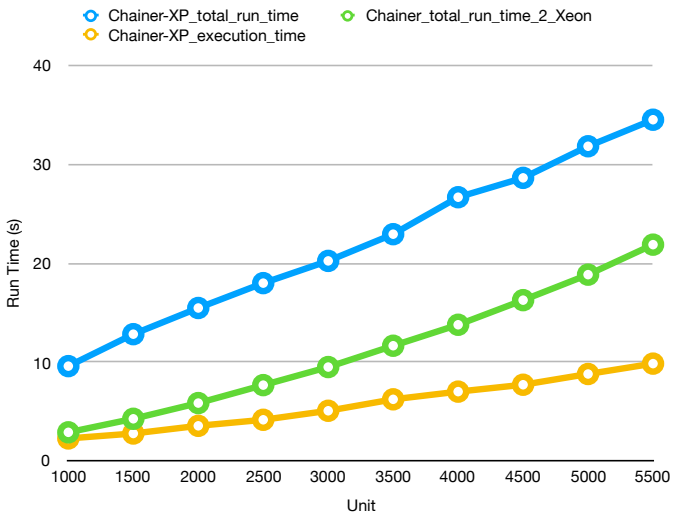


Fig. 5. Chainer-XP vs Chainer

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. Klemm, M., Enkovaara, J.: pymic: A python offload module for the intel xeon phi coprocessor. Proceedings of PyHPC (2014)
2. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of molecular biology 147(1), 195–197 (1981)