Università
della
Svizzera
italiana

Faculty
of Informatics

USI Technical Report Series in Informatics

Computational and Parallel Deep Learning Performance Benchmarks for the Xeon Phi

Tim Dettmers, Hanieh Soleimani [1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

Deep learning is a recent predictive modeling approach which may yield near-human performance on a range of tasks. Deep learning models have gained popularity as they achieved state-of-the-art results in many tasks, such as language translation or object recognition, but are computationally intensive thus requiring computers with accelerators and weeks of computation time. Here we test computational and parallel performance of deep learning implementations on the Intel Xeon Phi accelerator. We find that the performance for deep learning algorithms where successive operations have different dimensions is poor. In particular performance of general matrix multiplication and random number generation lead to a reduction about 20% and 7% , respectively, compared to when successive operations are of the same size. While parallelization performance seems to be in line with GPUs, we conclude that the Xeon Phi is unsuitable for deep learning in its current state.

Report Info

Published
July 2016

Number
USI-INF-TR-2016-4

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Deep learning is a field that depends highly on computational power. Graphics processing units (GPUs) were critical in achieving the first major milestones in deep learning [4, 8]. GPUs have been heavily researched as an option for deep learning algorithms and the computational efficiency of convolutions and general matrix multiplications (GEMM) has steadily been improved over the years [1, 9, 10, 13][1]. However, the reliance on a single architecture may hinder progress in computational performance since it discourages research on hardware options which may have better computational performance once their software matures.

In this work, we benchmark our deep learning software on the Intel Xeon Phi's accelerator and make the following contributions:

- We show that computational performance of individual GEMMs, is only efficient when all cores on the Xeon Phi are utilized, but at the same time we show that the full utilization of the cores drops the performance of successive GEMMs which have different dimensions to 20% with respect to successive GEMMs of the same dimensions.
- We show that random number generation for successive matrices with different dimensions results in a performance drop to 10% or less compared to matrices of the same size, and that this is independent of the number of threads used.

[1] Also see https://github.com/soumith/convnet-benchmarks and https://github.com/NervanaSystems/maxas/wiki/SGEMM for additional important contributions.

- We test MPI performance on a simple feed forward neural network and show that the results are similar to results obtained on GPUs.

1.1 Cluster and Hardware

1.1.1 Salomon Cluster

We used the Salomon cluster which consists of 432 nodes equipped with Intel Xeon Phi 7120 accelerators where each node is additionally powered by two Intel Xeon E5-2680v3, 2.5 GHz with 12 cores. The nodes are interconnected in a 7 dimensions enhanced hypercube Infiniband FDR56 network.

1.2 Intel Xeon Phi

The Intel Xeon Phi 7120 is a coprocessor, or accelerator, with 61 1.2 GHz cores and 16 GB, 352 GB/s memory. Although memory latency is high, the latency is usually hidden under massive thread parallelism, where each core can execute up to four threads at a time. The Xeon Phi makes heavy use of SIMD technology and as such is highly suitable for deep learning applications which usually make use of simple operations over large tensors. Unlike GPUs, the architecture of the Xeon Phi is full x86_64 and thus more complex instructions can be executed on Xeon Phis than on GPUs.

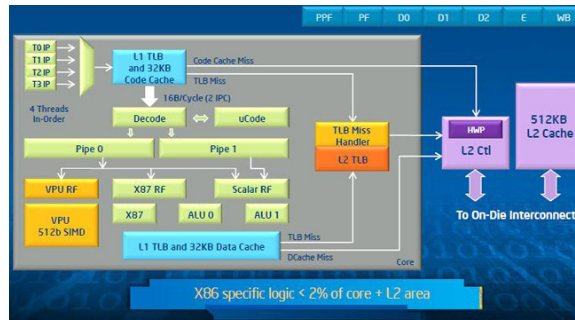

Figure 1. Architecture of the Intel Xeon Phi coprocessor.[2]

An important component of the Intel Xeon Phi coprocessor is its vector processing unit (VPU) which features a 512-bit SIMD instruction set. Thus, the VPU can execute 16 single-precision (SP) operations per cycle. The VPU also supports Fused Multiply-Add (FMA) instructions and hence can execute 32 SP floating point operations per cycle. Figure 2 shows the architecture of a single VPU.

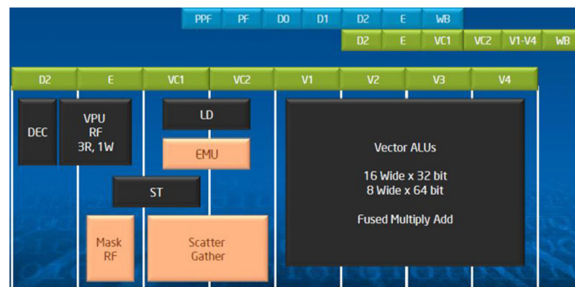

Figure 2. Architecture of a VPU.[3]

2 Deep Learning Language Model

Our initial aim was to accelerate recurrent deep neural networks for language generation as these models are computationally demanding, often taking weeks to train and also because recurrent neural networks lend themselves well to parallelization [6, 12]. The deep learning language model we planned on using

takes a sequence of character and predicts the next character in the sequence (see Figure 3) by using multiple layers of non-linear features, that is input sequences are transformed by a sequence of multiple nonlinear transformations of the form,

$$\mathbf{Output_{t+1}} = \tanh(\mathbf{Input_t W_t} + \mathbf{Output_{t-1} R_t} + \mathbf{Bias}).$$

After a sequence has been processed we calculate the gradient of the error with respect to the weight parameters $\mathbf{W}$ and recurrent weight parameters $\mathbf{R}$ by using the chain rule

$$\mathbf{Y_{i+1}^{t+1}} = \mathbf{A_i^t W_i} + \mathbf{A_{i+1}^{t-1} R_i} + \mathbf{B_i}, \tag{1}$$

$$\mathbf{A_{i+1}^{t+1}} = \sigma(\mathbf{Y_{i+1}^{t+1}}), \tag{2}$$

$$\frac{\partial \mathbf{E}}{\partial \mathbf{W_i}} = \frac{\partial \mathbf{E}}{\partial \mathbf{Y_{i+1}}} \frac{\partial \mathbf{Y_{i+1}}}{\partial \mathbf{W_i}} = \frac{\partial \mathbf{E}}{\partial \mathbf{A_{i+1}}} \frac{\partial \mathbf{A_{i+1}}}{\mathbf{Y_{i+1}}} \frac{\partial \mathbf{Y_{i+1}}}{\partial \mathbf{W_i}}, \tag{3}$$

where $\mathbf{W}$ is the weight matrix, $\mathbf{A}$ is the forward activation, $\mathbf{B}$ is the bias, $\mathbf{Y}$ is the output, and $\mathbf{E}$ is the error matrix. The indices i indicate the layer, while indices t indicate the time step for each iteration. Equations for the recurrent parameters follow analogously. The error here is the cross entropy error $E = \frac{1}{n}\sum_i^n (y \log z_i(1 - z_i) \log(1 - y))$, where y is the target character class and z_i is the predicted character class. The gradients are averaged and we then use stochastic gradient descent to update the parameters. Unlike full gradient descent only a small sample of the data, termed mini batch (128 samples per iteration in our case) are used per iteration since many of the input samples are correlated. This stochastic approach is much more efficient than using the full-batch data as we can find an improvement in the search space even if we only calculate a rough approximation of the full-batch gradient; this is especially true in the beginning of the optimization process. Thus this whole procedure then also entails a chain of small matrix multiplications with a rough dimension of 128 x 1024 times 1024 x 1024 where the dimensions after each layer may vary.

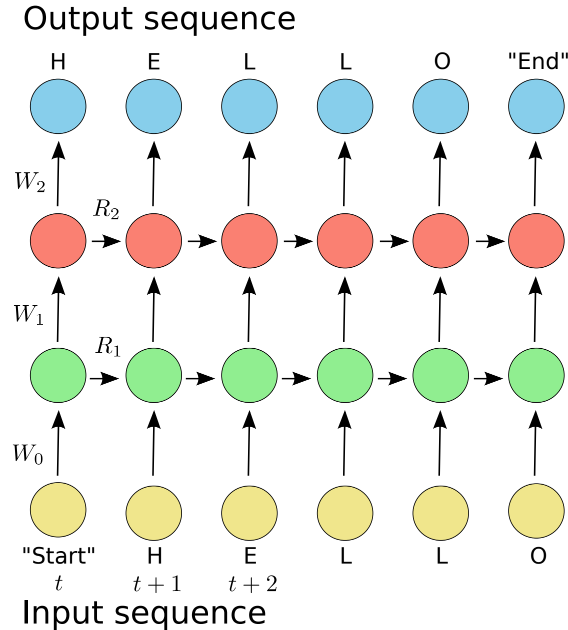

Figure 3. Schematic of the deep learning language model.

3 Parallelization

We can use data parallelism to accelerate learning on multiple nodes, where we split the data into equal parts for each accelerator and keep the model constant for each node. When we update the parameters after each iteration we need to synchronize them. The best way to do this is to split the gradients of each model into N parts, where N is the number of nodes and let each node aggregate one of the N parts, average them and then distribute them among all nodes. This operation can be done by an MPI SCATTER followed by an MPI Allgather. For the recurrent model we have a lot of computation and little communication so that the bottleneck is not too big. However, in this case we tested our parallelization procedure on a simple neural network model, which is the same model as above but without any recurrent connections. We do this because our implementation was too slow to test MPI performance on recurrent neural networks.

4 Xeon Phi performance problems

We attempted to benchmark our deep learning model, but soon we found that this was rather infeasible as one pass through our test data which was the fairly small Brown corpus would have taken many weeks and full training up to one year. It was clear to us that either something in our implementation was wrong or that there were performance problems associated with the Xeon Phi software or hardware. We did end up checking our code for vectorization and benchmarked most procedures against the GPU. For vectorized and the few procedures that could not be vectorized easily we found that the performance of the Xeon Phi was on par with GPUs after normalizing with theoretical FLOPS of each accelerator. However, after having a closer look at GEMM and random number generation operations we found that the bottlenecks lie here.

4.1 GEMM problems

We found that GEMM operations are slow when they are chained one after each other and the dimensions of successive GEMMs change (see Figure 4). We isolated our GEMM procedure from our library and created a minimal benchmark to test the performance of the GEMM as implemented in the Intel MKL (which is optimized for Xeon Phis).

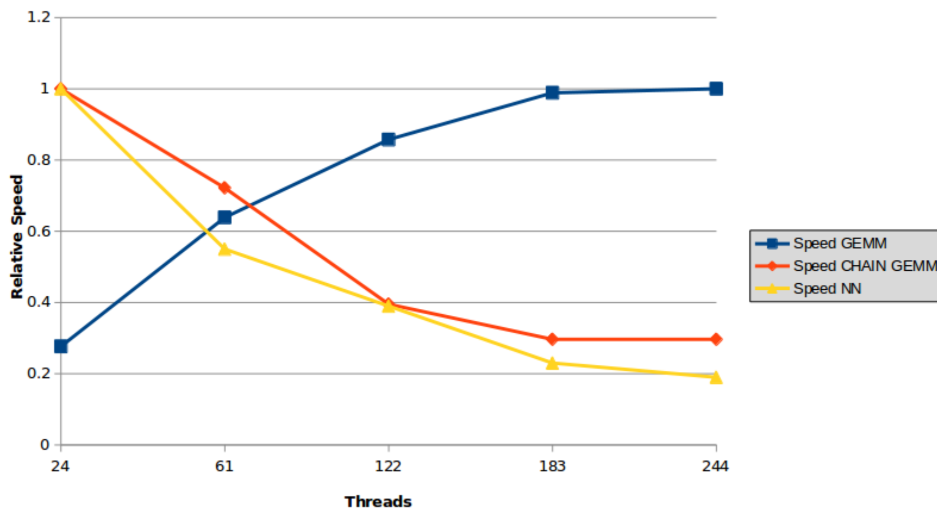

Figure 4. GEMM performance in isolated environments (GEMM and Chain GEMM) and in our neural network library on the MNIST data set.

More specifically we found the following.

- GEMM is fast if many threads are used (about 70% peak performance) and slow if few threads are used (25% of peak performance).

- GEMM is slow if many threads are used and the successive matrix dimensions differ (typical for a neural network; 10% of peak performance) but faster for different sizes and few threads (20% of peak performance).
- It follows that neither many, nor few threads are efficient for feedforward and recurrent neural networks.

4.2 Random number generation

We isolated the random number generator for uniform random numbers which has Intel Xeon Phi support and performed our test for this algorithm. The random number generator for Xeon Phi makes uses of the Mersenne twister algorithm which is able to generate random numbers quickly in a parallel fashion.

However, in our case we found that filling matrices with random numbers is only fast if the matrices have the same dimension but slow if the dimensions differ. Unlike the GEMM operation this does occur for any number of threads. We also found that performance does not improve if some operations are between successive random number generation calls, for example, when we are using dropout for every hidden layer [7, 11], the performance does not improve. The slow down is significant with our neural network training being hit by a performance reduction of 13.5 times. This makes it infeasible to use dropout regularization with the Xeon Phi.

4.3 Discussion of performance problems

We do not know why the performance decreased so significantly in our case as it was difficult to analyze the situation further beyond isolating the algorithms. One hypothesis is that the scheduling of threads on the Xeon Phi works sub optimally when confronted with rapidly changing sizes for the memory buffers (in our case matrices) so that most of the time is spend on scheduling threads rather than fetching memory or doing computations. This hypothesis is to be explored in further studies.

5 Parallel

We initially set out to parallelize the language model introduced above, but the performance was so poor that even with large scale parallelization over many nodes it would still have taken weeks to perform. We thus benchmarked the MPI performance of the Xeon Phi for a simple neural network task on the MNIST dataset for two epochs, that is two passes through the dataset. We expected normal performance since the parallelization bottleneck is situated in the network and PCIe interfaces that connect the Xeon Phi with the network interface and not the Xeon Phi itself [5]. The performance can be seen in Table 1 and is in line with our expectations for the performance of simple neural network parallelization.

Table 1. Parallelization performance of simple neural network.

Features	Seconds per epoch	Speedup
1 node base line	25	1.0x
1 node, halved data set	14	1.8x
MPI 2 nodes	20	1.25x
MPI 4 nodes	17	1.5x
MPI 8 nodes	13.5	1.9x
MPI 16 nodes	11	2.3x

6 Conclusion

Here we analyzed the computational and parallelization performance of the Xeon Phi accelerator for simple deep learning tasks such as classification of handwritten digits and the generation of language.

We found that the parallel performance on Xeon Phis is about equivalent to the parallel performance on GPUs. We also found that matrix multiplication and random number generation on the Xeon Phi are slow if one queues these operations one after each other with differently sized buffers. Since we isolated these operations to minimally working code, we hypothesized that these performance problems take root in their internal implementation or that this is due to thread scheduling for the Xeon Phi. However, we do not have any further insights into these hypotheses and further work is required to unravel the root of these performance problems.

References

[1] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cudnn: Efficient primitives for deep learning*, arXiv preprint arXiv:1410.0759, (2014).

[2] G. Chrysos, *Intel xeon phi x100 family coprocessor - the architecture*. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[3] ———, *Intel xeon phi x100 family coprocessor - the architecture*. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[4] G. E. Dahl, D. Yu, L. Deng, and A. Acero, *Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition*, Audio, Speech, and Language Processing, IEEE Transactions on, 20 (2012), pp. 30–42.

[5] T. Dettmers, *8-bit approximations for parallelism in deep learning*, International conference on learning representations, (2016).

[6] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al., *Deep speech: Scaling up end-to-end speech recognition*, arXiv preprint arXiv:1412.5567, (2014).

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors*, arXiv preprint arXiv:1207.0580, (2012).

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in Advances in neural information processing systems, 2012, pp. 1097–1105.

[9] A. Lavin, *Fast algorithms for convolutional neural networks*, arXiv preprint arXiv:1509.09308, (2015).

[10] ———, *maxdnn: An efficient convolution kernel for deep learning with maxwell gpus*, arXiv preprint arXiv:1501.06633, (2015).

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting*, The Journal of Machine Learning Research, 15 (2014), pp. 1929–1958.

[12] I. Sutskever, J. Martens, and G. E. Hinton, *Generating text with recurrent neural networks*, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, pp. 1017–1024.

[13] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, *Fast convolutional nets with fbfft: A gpu performance evaluation*, arXiv preprint arXiv:1412.7580, (2014).