Parallel Sparse Coding for Seafloor Image Analysis

Genlang Chen[1,2], Chenggang Lai[2], Miaoqing Huang[2]
[1]Ningbo Institute of Technology, Zhejiang University, 315100, China
[2]University of Arkansas, Arkansas 72701, USA
cgl@zju.edu.cn, {cl004,mqhuang}@uark.edu

ABSTRACT

Sparse coding has been a popular learning model in machine learning field. However, due to the complexity of the learning model, the high computational cost has seriously hindered its application. Toward this purpose, this paper presents a parallel sparse coding method to improve the performance by exploiting the power of acceleration technologies such as Intel MIC and GPU. We use both parallel programming modes, i.e., the Native model and the Offload model, to parallelize the sparse coding on MIC based computer cluster. Extensive experimental results on the AUV data of the southeast coast of Tasmania have shown that sparse coding can be accelerated significantly on MIC and GPU. When using the same number of threads, the Native model and the Offload model achieve very close performance for sparse coding. In addition, Native model demonstrates better performance scalability than the Offload model. On the other side, parallel implementation on GPU shows the best performance.

1. INTRODUCTION

Due to its capability for learning the hierarchical representations of the unlabeled inputs, sparse coding has become a popular learning model in the machine learning community over the past few years. It has been applied to a number of machine learning applications, including text modeling, image segmentation, image classification, etc [4]. These problems have a common characteristic, i.e., the inputs are highly dimensional and can be learned with many latent variables and layers. In recent research, sparse coding has shown significant advantages over state-of-the-art machine learning methods used in both speech and image processing [5].

Recently, several research groups have developed techniques in order to reduce the high computational cost of sparse coding. Based on the method for iteratively solving two convex optimization problems, i.e., an L_1-regularized least squares problem and an L_2-constrained least squares problem, an efficient sparse coding algorithm has been proposed by Lee *et al.* [7]. Rajat *et al.* developed general principles for massively parallel unsupervised learning tasks using GPUs [9]. However, these improved algorithms mentioned above either need to apply restrictions or the performance is still not satisfying.

In this work we intend to develop sparse coding algorithms that meet the following requirements. First, the learning models should have a good applicability with as few restrictions as possible. Second, the learning models should have a strong scalability by taking more parameters, or more training examples in order to produce very significant performance benefits. Third, the computational cost should be reasonable so that it can be applied to many applications. We leverage the massively parallel technologies such as graphics processing units (GPUs) and Intel Many Integrated Core (MIC) architecture in this work to improve the performance of sparse coding model. Both of them can accommodate hundreds to thousands of threads on a single device with a very marginal scheduling overhead. The rest of the paper is organized as follows. A re-designed parallel learning algorithm and our proposed methods based on MIC and GPU are introduced in Section 2. The experimental results and evaluation using the AUV seafloor image data set are presented in Section 3. We conclude this work in Section 4.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2015) Boston, MA, USA, June 1-2, 2015.

2. PARALLEL SPARSE CODING

2.1 Problem statement

Sparse coding is an algorithm for constructing succinct representations of input vectors such as images [8] using the basis vectors in a dictionary. Given a set of n 2D images, each image can be represented by a 1D array $\overrightarrow{x^{(i)}}$, $i = 1 \dots n$. For example, if an image contains 32×32 pixels, the corresponding $\overrightarrow{x^{(i)}}$ will consist of 1,024 elements. The aim of sparse coding is to find a set of basis vectors $\overrightarrow{d^{(j)}}$ $(j = 1 \dots k)$ such that we can represent an input vector $\overrightarrow{x^{(i)}}$ as a linear combination of these basis vectors: $\overrightarrow{x^{(i)}} = \sum_{j=1}^{k} a_j^{(i)} \overrightarrow{d^{(j)}}$ in which each $a_j^{(i)}$ is a scalar coefficient. If we use D and $\overrightarrow{a^{(i)}}$ to represent $[\overrightarrow{d^{(1)}}, \overrightarrow{d^{(2)}}, \dots \overrightarrow{d^{(k)}}]$ and $[a_1^{(i)}, a_2^{(i)}, \dots a_k^{(i)}]^T$, respectively, we can re-write the equation as $\overrightarrow{x^{(i)}} = D\overrightarrow{a^{(i)}}$. Further, if we use X and A to represent $[\overrightarrow{x^{(1)}}, \overrightarrow{x^{(2)}}, \dots \overrightarrow{x^{(n)}}]$ and $[\overrightarrow{a^{(1)}}, \overrightarrow{a^{(2)}}, \dots \overrightarrow{a^{(n)}}]$, respectively, we want to represent X as $X = DA$. Each column of D (i.e., $\overrightarrow{d^{(j)}}$) is regarded as a base in the directory. Each column of A (i.e., $\overrightarrow{a^{(i)}}$) is the sparse representation of the corresponding input vector $\overrightarrow{x^{(i)}}$ according to the dictionary. Most components of $\overrightarrow{a^{(i)}}$ should be zero's.

Table 1: Computation time of four steps.

Image Patch Size	Step	Computation Time (s)
32×32	Load image	5.93
	Optimizing A	14,811.29
	Optimizing D	208.32
	Write results	1.8

Algorithm 1 Learning the dictionary in sparse coding.

```
1: Transfer a large number of images into global memory;
2: Select a group of data sets randomly;
3: Initialize the dictionary $D$ randomly;
4: while (convergence criterion is not satisfied) {
5:   for (each data set in the group) {
6:     Set $A \leftarrow D^T X$, and normalize $A$;
7:     Keep $D$ fixed, optimize over $A$ by solving an $L_1$ regularized least squares problem;
8:     Keep $A$ fixed, optimize over $D$ by using convex optimization techniques;
9:   }
10: }
```

The basic idea of sparse coding is defined as finding two matrices D and A and aims to solve the following optimization problem:

$$\min_{A,D} \sum_{i=1}^{n} \left(\left\| \overrightarrow{x^{(i)}} - \sum_{j=1}^{k} a_j^{(i)} \overrightarrow{d^{(j)}} \right\|^2 + \lambda \sum_{j=1}^{k} |a_j^{(i)}| \right) \quad (1)$$

It has been shown that the optimization problem is not jointly convex in both A and D. But it is convex in either A or D if the other one is kept fixed. An alternating minimization algorithm has bee proposed in [7], as shown in Algorithm 1. When facing large numbers of images and dictionary bases, the step of optimizing over A is particularly time consuming since it involves an uncertain objective function. The overwhelmingly dominant computational effort is spent on optimizing A by solving an L_1 regularized least squares problem. The entire learning process is divided into four steps, including *Load image*, *Optimizing A*, *Optimizing D*, and *Write results*. We conducted a simple test to check the computation time spent on each step. In this test, the source data came from 1,000 images. Then we randomly chose 5 sets of image patches. Each set contains 1,024 patches, each of which is 32×32. The process to form the source data space and choose the data sets is illustrated in Figure 1. Once the 5 sets of image patches are generated, we applied the optimization on them using Algorithm 1. In this simple test, we only carried out 3 iterations of the `while` loop. This implementation was a single-thread implementation written in C and executed on an Intel Xeon E5606 2.13-GHz CPU. The computation times on all steps are shown in Table 1. It is obvious that the step of optimizing over A is the most time-consuming step.

Raina *et al.* have presented a method to optimize the computation in the Equation (1) [9]. Let $\overrightarrow{d_j}$ (i.e., $\overrightarrow{d^{(j)}}$) be the j^{th} column of the dictionary D and set $r_j = \overrightarrow{d_j}^T \overrightarrow{d_j}$, the new optimal value $a_j^{(i)}$ for input $\overrightarrow{x^{(i)}}$ can be calculated as

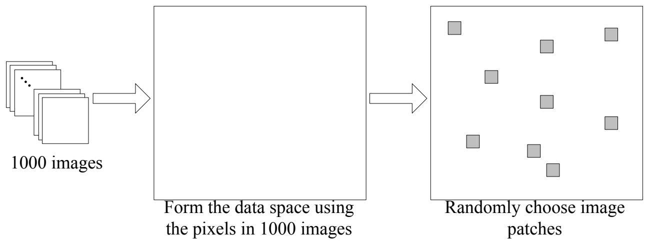
1000 images
Form the data space using the pixels in 1000 images
Randomly choose image patches

Figure 1: The process to form the source data space and randomly choose several data sets.

follows.

$$a_j^{(i)} = \begin{cases} 0 & \text{if } |g_j - r_j a_j^{(i)}| \leqslant \beta \\ (-g_j + r_j a_j^{(i)} + \beta)/r_j & \text{if } g_j - r_j a_j^{(i)} > \beta \\ (-g_j + r_j a_j^{(i)} - \beta)/r_j & \text{if } g_j - r_j a_j^{(i)} < -\beta \end{cases} \quad (2)$$

where g_j is the j^{th} component of vector $\overrightarrow{g}$, which is as follows.

$$\overrightarrow{g} = \nabla_a \frac{1}{2} \left\| \overrightarrow{x^{(i)}} - \sum_{j=1}^{k} a_j^{(i)} \overrightarrow{d^{(j)}} \right\|^2 = D^T D \overrightarrow{a^{(i)}} - D^T \overrightarrow{x^{(i)}} \quad (3)$$

In our implementation, each $\overrightarrow{a^{(i)}}$ will be updated for 150 times in one iteration of the optimization of A while D is fixed. In this work, all three matrices, X, D, and A, are all in the dimension of $1,024 \times 1,024$.

2.2 Parallelization using MIC

Because the *Optimizing A* is the most time consuming, we spent the most effort on parallelizing it. On computer clusters equipped with Intel MIC, the parallel sparse coding can be implemented with the following execution modes.

- **Native-1**: i.e., the native model. In this implementation, the MPI process is directly executed on each MIC core. For the 1,024 vectors of $\overrightarrow{a^{(i)}}$'s, they are evenly distributed among MPI processes for computation. Each MPI process is a single-thread process.
- **Native-2**: On top of Native-1 execution mode, we try to take advantage of the internal processing parallelism on each MIC core. Therefore, we launch 4 threads in each MPI process using OpenMP. For each $\overrightarrow{a^{(i)}}$, it will be updated for 150 times. For each update, 4 threads work on the 1,024 components of $\overrightarrow{a^{(i)}}$ in parallel.
- **Offload**: In this mode, the MPI processes are hosted by the CPU cores, which offload the computation including data to the MIC processors. In the sparse coding implementation, the 1,024 $\overrightarrow{a^{(i)}}$'s are first distributed to computer nodes using MPI. Then each subset of $\overrightarrow{a^{(i)}}$'s is offloaded to one MIC card and further distributed among multiple threads using OpenMP.

2.3 Parallelization using GPU

A single modern GPU device can accommodate thousands to millions of threads, which can be scheduled with a very marginal overhead. The hardware architecture consists of dozens of multiprocessors, each of which further contains dozens to hundreds of streaming processors. Based on

Nvidia's CUDA programming model, a grid of threads is broken into thread blocks, each of which will be scheduled to a multiprocessor. Then the threads in a block will be scheduled to the streaming processors in warps.

In order to take advantage of GPU's parallel architecture, based on Algorithm 1 and the Equation (2), the learning task is implemented to fit the two levels of parallelism: blocks and threads. Blocks are used to achieve the data parallelism by working on separate $\overrightarrow{a^{(i)}}$'s. Inside a block, each thread computes just 2 components of the vector $\overrightarrow{a^{(i)}}$ so as to exploit more fine-grained parallelism. In this method, all threads within a block can synchronize with each other by using a small amount of very fast shared memory.

In this work, we did not use multiple GPUs in our implementation. The typical parallel programming model for GPU based computer cluster is offload, which is similar to the offload mode on MIC based computer cluster. For Nvidia GPUs, CUDA is typically used to offload work from CPUs to GPUs.

3. EXPERIMENT AND EVALUATION

In order to evaluate the performance of the parallel methods on Intel MIC and GPU, a series of experiments were carried out on a large and high-quality AUV (Autonomous Underwater Vehicle) data set. In this section we present a brief discussion of our experimental data set, experiment platform, and the results.

3.1 Data set

The data set is comprised of 14 dive missions conducted by the AUV Sirius off the southeast coast of Tasmania in October 2008 [2, 3]. It contains over 100,000 stereo pairs of images. Marine scientists used the CPCe software package [6] to label 50 random points on each image with different class labels, such as biological species (including coral, algae and others), abiotic elements (sand, gravel, rock, shell, etc.), and other unknown data types.

3.2 Experiment platform

We conduct our experiments on the NSF sponsored Beacon supercomputer [1] hosted at the National Institute for Computational Sciences (NICS), University of Tennessee. The Beacon system (a Cray CS300-AC Cluster Supercomputer) offers the access to 48 compute nodes and 6 I/O nodes joined by FDR InfiniBand interconnect, which provides a 56 Gb/s bi-directional bandwidth. Each compute node is equipped with 2 Intel Xeon E5-2670 8-core 2.6-GHz processors, 4 Intel Xeon Phi (MIC) coprocessors 5110P, 256 GB of RAM, and 960 GB of SSD storage. Each I/O node provides access to an additional 4.8 TB of SSD storage. Each Xeon Phi 5110P coprocessor contains 60 1.053-GHz MIC cores and 8 GB GDDR5 on-board memory. The compiler used in this work is Intel 64 Compiler XE, Version 14.0.0.080 Build 20130728.

We implemented the GPU implementation on a workstation that contains an Nvidia Tesla K20. The host CPU is Intel Core i7-3820 CPU at 3.60 GHz with 16GB memory. The compiler version is CUDA 5.5.

3.3 Results and discussion

3.3.1 *Comparison among MIC's three implementation modes*

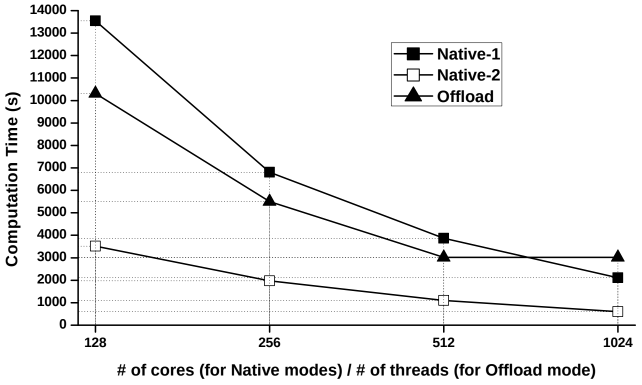

Figure 2: Performance under three parallel execution modes on Beacon computer cluster.

We implemented three parallel execution modes, i.e., Native-1, Native-2, and Offload, on the Beacon computer cluster using multiple nodes. The original image size is 1,024×1,024. Then we randomly chose 5 sets of data as shown in Figure 1, and applied the optimization on them using Algorithm 1. Each data set contains 1,024 image patches of the size 32×32. Only 3 iterations are carried out in the experiment.

In order to show the strong scalability of the parallel implementations, the problem size is fixed for each implementation while the number of participating cores is increased. For the native modes, each MIC core will host one MPI process. We schedule 60 MPI processes to a MIC card. For the offload mode, the MPI process is hosted on the CPU core. We schedule 128 threads to a MIC card using OpenMP from an MPI process in the offload mode.

From Figure 2 we can see that both implementations in Native mode keep the strong scalability, i.e., the computation time halves when the number of processing cores doubles. Because the Native-2 implementation uses 4 times threads as the Native-1 implementation, its performance is approximately 4 times better than the Native-1 mode. The performance of the Offload implementation is typically better than the Native-1 implementation. The 128-core Native-1 implementation occupies 3 MIC cards. As a contrast, the 128-thread Offload implementation only occupy one MIC card. Apparently, the 128-thread Offload implementation has not reached the performance limit of a MIC card because it outperforms the 128-core Native-1 implementation. However, as we schedule more threads in the Offload implementation, it brings more cross-card communication. Eventually the increase in the communication overhead offsets the decrease in the calculation time as the case of 512-thread implementation to 1024-thread implementation.

If we want to compare the performance of Native-2 mode and the Offload mode under the same number of threads, their performances are very close. For example, the 512-thread Offload implementation takes ~3,000 seconds, which is close to the performance of the 128-core (128×4=512) Native-2 implementation.

3.3.2 *Performance comparison of single devices*

In this experiment, we conduct a comparison between two technologies (i.e., MIC and GPU) at the full capacity of a

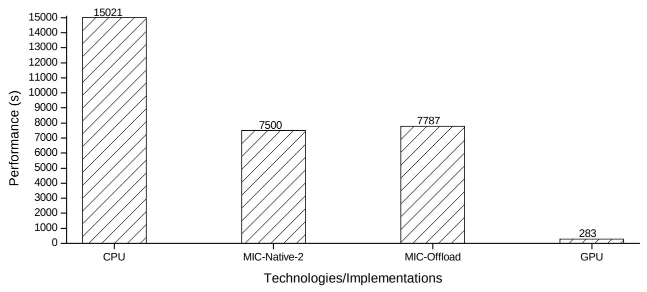

Figure 3: Performance on single devices.

single device. The base case implementation is a single-thread implementation on Intel Xeon E5606 2.13GHz CPU with 12GB memory. The other three implementations on two technologies are as follows.

- MIC-Native-2: the Native-2 implementation on a single Xeon 5110P MIC card. 60 MPI processes execute on 60 cores. Each MPI process contains 4 threads.
- MIC-Offload: the Offload implementation on a single Xeon 5110P MIC card. We scheduled 180 threads to the MIC card in the offload mode. We also found that the performance actually dropped when we scheduled more threads (e.g., 240-thread implementation took ∼9,600 senconds).
- GPU: the GPU implementation on a single Nvidia K20 GPU device. In the GPU implementation, we scheduled a 1-dimensional grid consisting of 1,024 blocks. Each thread block is 1-dimensional and contains 512 threads.

From Figure 3 we can find that both programming models achieve the similar performance on a single MIC card with full capacity. For this particular application, the performance of a single MIC card is approximately 2 times of the single-thread implementation on the Intel Xeon E5606 CPU. On the other hand, it can be found that GPU has tremendous performance advantage for the sparse coding application. A single K20 device can achieve more than 50 times speedup than the CPU implementation and more than 25 times speedup than the MIC. The main reason is the significant number of cores in a single K20 GPU. Because the main calculation in the learning dictionary process is the matrix multiplication, which can be parallelized efficiently among processing cores. The more cores the better performance in general. A single K20 GPU device contains 2,496 cores. By contrast, an Intel Xeon Phi 5110P only contains 60 cores. When we use 1,024 MIC cores across multiple MIC cards in Native-2 execution mode, the performance is ∼600 seconds, which is at the same magnitude of the performance of a K20 GPU.

4. CONCLUSIONS

In this work, we conduct a detailed study regarding the performance and scalability of parallel sparse coding on both Intel MIC processors and GPU. On MIC processors, in order to improve the performance, three execution modes in two programming models are leveraged to achieve the parallelism of sparse coding, including Native-1, Native-2, and Offload. The results show that it is very important to schedule multiple threads in an MPI process for the Native mode to achieve the best performance. When using the same number of threads, the Native model and the Offload model achieve very close performance for sparse coding. In addition, Native model demonstrates better performance scalability than the Offload model. On the other side, parallel implementation on GPU shows the best performance. However, if we use the same number of cores for both technologies, they will achieve very close performance.

5. ACKNOWLEDGMENTS

This research used resources of the Beacon supercomputer supported by the NSF and the State of Tennessee. The authors would also thank Nvidia for GPU donations.

This work is supported in part by the Special Scientific Research Fund for Public Welfare Industry of the Chinese State Ocean Administration (Grant No.: 201305026), by NSF (Grant No.: CNS-1219062), and by NASA (Grant No.: 12-2012EPSCoR-0026). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and NASA.

6. REFERENCES

[1] `http://www.jics.tennessee.edu/aace/beacon`.

[2] Barrett, N., Meyer, L., Hill, N., and Walsh, P. Methods for the processing and scoring of auv digital imagery from south eastern tasmania.

[3] Bewley, M., Douillard, B., Nourani-Vatani, N., Friedman, A., Pizarro, O., and Williams, S. Automated species detection: An experimental approach to kelp detection from sea-floor auv images. In *Proceedings of Australasian Conference on Robotics and Automation, Victoria University of Wellington, New Zealand* (2012).

[4] Bo, L., Ren, X., and Fox, D. Hierarchical matching pursuit for image classification: Architecture and fast algorithms. In *Advances in neural information processing systems* (2011), pp. 2115–2123.

[5] Boureau, Y.-L., Bach, F., LeCun, Y., and Ponce, J. Learning mid-level features for recognition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on* (2010), IEEE, pp. 2559–2566.

[6] Kohler, K. E., and Gill, S. M. Coral point count with excel extensions (cpce): A visual basic program for the determination of coral and substrate coverage using random point count methodology. *Computers & Geosciences 32*, 9 (2006), 1259–1269.

[7] Lee, H., Battle, A., Raina, R., and Ng, A. Y. Efficient sparse coding algorithms. In *Advances in neural information processing systems* (2006), pp. 801–808.

[8] Olshausen, B. A., et al. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature 381*, 6583 (1996), 607–609.

[9] Raina, R., Madhavan, A., and Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning* (2009), ACM, pp. 873–880.