Full Correlation Matrix Analysis of fMRI Data on Intel® Xeon Phi™ Coprocessors

Yida Wang[1], Michael J. Anderson[2], Jonathan D. Cohen[3], Alexander Heinecke[2], Kai Li[1], Nadathur Satish[2], Narayanan Sundaram[2], Nicholas B. Turk-Browne[3], and Theodore L. Willke[2]

[1]Department of Computer Science, Princeton University

[2]Parallel Computing Lab, Intel Corporation

[3]Princeton Neuroscience Institute, Princeton University

ABSTRACT

Full correlation matrix analysis (FCMA) is an unbiased approach for exhaustively studying interactions among brain regions in functional magnetic resonance imaging (fMRI) data from human participants. In order to answer neuroscientific questions efficiently, we are developing a closed-loop analysis system with FCMA on a cluster of nodes with Intel® Xeon Phi™ coprocessors. Here we propose several ideas for data-driven algorithmic modification to improve the performance on the coprocessor. Our experiments with real datasets show that the optimized single-node code runs 5x-16x faster than the baseline implementation using the well-known Intel® MKL and LibSVM libraries, and that the cluster implementation achieves near linear speedup on 5760 cores.

Keywords

fMRI data, Intel® Xeon Phi™ Coprocessor

1. INTRODUCTION

Neuroscientists use functional magnetic resonance imaging (fMRI) technology to acquire volumes of activity from human brains. Most previous studies focus on offline data analysis to discover neural activity patterns and interactions in different brain regions. Recently two new approaches have shown promise as a means to accelerate discoveries in neuroscience: real-time fMRI with closed-loop feedback[7] and exhaustive study of neural interactions via imaging data[27].

Real-time fMRI (rtfMRI) refers to any process that uses functional information from the scanner in a manner that keeps pace with data acquisition. rtfMRI has been applied to interoperative surgical guidance, brain-computer interfaces, and neurofeedback[25]. A recent study shows that closed-loop neurofeedback can be used to train participants to improve their ability to attend selectively to a stimulus[7]. A system that permits real-time analysis of neural interactions for the entire brain will allow neuroscientists to conduct new scientific and clinical studies at a time when interest in rtfMRI neurofeedback is rising rapidly[25].

Full correlation matrix analysis (FCMA)[30] is a novel attempt to exhaustively study the neural interactions in a brain by applying multivariate pattern analysis (MVPA) methods[21], to the whole-brain correlation matrix, rather than focusing on the instantaneous amplitude of blood-oxygen-level dependent (BOLD) activity, or on correlations in this signal over limited subregions of the brain. However, FCMA is computationally intensive, and has not yet been carried our in real-time. Doing so on cost-effective computational platforms presents a challenge, however it also holds the promise of substantially boosting progress in neuroscience.

To address this challenge, we are developing a closed-loop neuroscience research system with an fMRI scanner and a cluster using Intel ® Xeon Phi™ coprocessors (henceforth referred to as coprocessor), as shown in Fig. 1. The fMRI scanner produces an entire brain's worth of data every 1-2 seconds as a human subject is exposed to stimuli and/or asked to perform tasks. The stream of brain data is sent to a compute cluster with coprocessors that runs FCMA. The FCMA software analyzes the brain data in two ways: performing offline brain interaction analysis after collecting multiple subjects' data; or selecting voxels to train a classifier using one subject's data online to provide real-time feedback as described in the following experiments. The software must achieve satisfactory wall-clock time in both scenarios.

The challenge is to achieve satisfactory performance for both offline analysis and online real-time analysis without substantial computational hardware costs. There are two main design goals for FCMA closed-loop system. The first is a scalable implementation that can achieves linear or near-linear speedup on a large cluster. The second is to ensure that the code running on each individual node fully exploits the hardware capability of the coprocessor.

The input data for FCMA is a stream of 3D human brain data (volumes of voxels) over time. Including the time dimension, the input is a 4D dataset. A separated full correlation matrix (i.e. the temporal correlation in BOLD activity of every voxel in the brain with every other voxel) is com-

SC '15, November 15-20, 2015, Austin, TX, USA

© 2015 ACM. ISBN 978-1-4503-3723-6/15/11...$15.00

DOI: http://dx.doi.org/10.1145/2807591.2807631

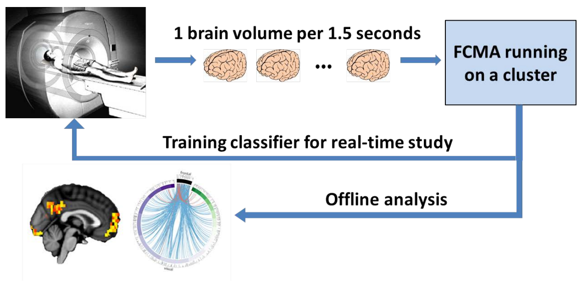
1 brain volume per 1.5 seconds
FCMA running on a cluster
Training classifier for real-time study
Offline analysis

Figure 1: A closed-loop neuroscience research system with an fMRI scanner and a cluster using Intel® Xeon Phi™ coprocessors.

puted for each time epoch of interest during the fMRI scan. Each matrix may then be categorically labeled based on the experimental conditions experienced by the subject during each epoch (e.g., the stimulus or task condition). The goal of FCMA is to make an unbiased determination of which correlations distinguish between conditions, thereby identifying regions of interests (ROIs) that have different patterns of interactions as a function of condition.

The data that FCMA deals with shapes as tall-skinny matrices since the number of voxels (up to 100,000) is much larger than the number of time points (typically dozens to hundreds). Although it is fairly straightforward to parallelize FCMA using cluster-level data partitioning, it is challenging to fully utilize the hardware capabilities of modern architectures, specifically the increasing amounts of thread- and data-level parallelism, and smaller amounts of cache per core. Manycore architectures are leading these trends and hence benefit more from optimizations such as vectorization and blocking. Our baseline implementation using MKL and LibSVM libraries achieved respectable performance, but was found to significantly underutilize the hardware. These libraries, and others like them, use cache conscious algorithms to implement their functions, but they do not co-optimize functions or handle data with special characteristics such as tall-skinny matrices well, things that we believe would help in our application.

This paper describes several optimizations for FCMA on manycore architectures, including blocking tall-skinny matrices for multiplication, retaining L2 cache contents across computation stages, and designing data layout and workflow to be vectorization friendly. We have implemented an optimized version of FCMA that incorporates these ideas, as well as optimized support vector machine (SVM) algorithm.

Our evaluation shows that the optimized implementation on a single coprocessor runs 5x-16x faster than the baseline with MKL and LibSVM libraries. Our optimized SVM runs 10x faster than the popular LibSVM package[6] on a single coprocessor. Our parallel FCMA implementation on a cluster of Intel® Xeon Phi™ coprocessors achieves near linear speedup on up to 96 coprocessors or 5760 cores. Although being applied to FCMA as a case study, our optimizations will also enhance the performance of other applications that involve datasets with similar characteristics on Intel® Xeon Phi™ coprocessors.

We also show that our optimizations for the coprocessor yield a faster implementation on Intel® Xeon® processors. The optimized implementation on an E5-2670 processor runs 1.4x-2.5x faster than the baseline with MKL and LibSVM libraries.

This paper is organized as follows. Section 2 reviews the coprocessor architecture. We describe FCMA in detail in Section 3.1, followed by a discussion of the baseline implementation in Section 3.2 and its performance analysis in Section 3.3. We propose three optimization ideas in Section 4 and evaluate their performance in Section 5. Section 6 discusses the related work, followed by conclusion and future work in Section 7.

2. INTEL ® XEON PHI™ ARCHITECTURE

The Intel® Xeon Phi™ is a coprocessor based on the Intel Many Integrated Core (MIC) architecture[1]. The coprocessor provides a general-purpose programming environment similar to that of an Intel® Xeon® processor.

Fig. 2 illustrates the high-level architecture of the 5110P coprocessor. This model has 60 CPU cores, each of which runs at a fixed clock rate of 1053MHz and supports up to 4 hardware threads compensating for its in-order instruction execution. Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a 512KB unified L2 cache shared by up to four threads. The L2 caches belonging to different cores are interconnected via a bidirectional ring. Cache coherence is maintained by a global-distributed tag directory.

An L2 cache miss triggered by a core can be satisfied by either a remote cache or the memory with slightly different latencies. A previous empirical study showed that the latency of an L2 cache miss on Xeon Phi takes 250 CPU cycles from a remote L2 cache location and 302 CPU cycles from the main memory[11]. Both L1 and L2 caches use a cache line size of 64B, therefore, a cache miss will bring 16 single precision or 8 double precision floating point numbers into the cache.

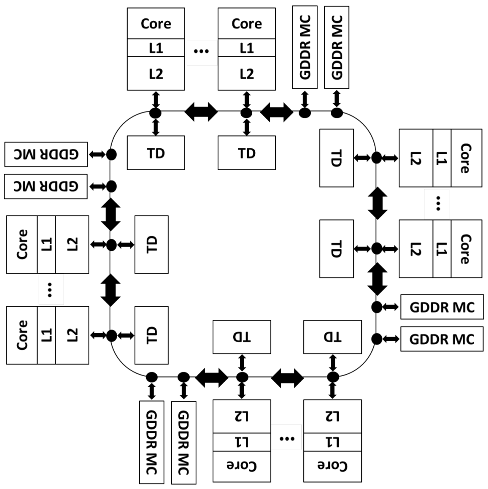

Figure 2: Architecture of the Intel® Xeon Phi™ 5110P coprocessor.

Each core also has a 512-bit wide vector processing unit (VPU), which allows 16 single precision or 8 double precision floating point numbers to be processed in a single CPU

[1]Intel, Xeon and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

cycle. The wide VPU size makes vectorization challenging. If the vectorization intensity, defined as the number of vectorized elements divided by the number of executed VPU instructions, is low, the VPU is not fully utilized.

By using all available threads and VPU in the most efficient way, the peak floating point performance of the coprocessor can reach 2.02 TFLOPS for single precision and 1.01 TFLOPS for double precision.

Each 5110P coprocessor board has 8GB DRAM, out of which ~2GB are dedicated to the operating system, leaving ~6GB memory available for applications.

3. FCMA ALGORITHM

3.1 Overview

FCMA works on fMRI datasets. An fMRI dataset contains the fMRI data from a neuroscientific experiment, often conducted over multiple human subjects. A brain volume is comprised of a number of voxels, depending on the resolution of the fMRI scanner and its scanning speed. An epoch of interest consists of a series of continuous time points during which the subject was doing some specific task. The time epoch can be labeled based on the types of task.

The fMRI scanner used in this research (Siemens Skyra) can be configured in various ways, but a relatively common set of image parameters might be 35,000 voxels every 1.5 seconds. The neuroscience datasets we consider contain fMRI data from tens of subjects, each of whom has dozens of time epochs of interest labeled in two conditions, meaning that hundreds of full correlation matrices must be computed. Although the size of such a dataset is approximately a gigabyte, the size of the corresponding full correlation matrices will be in terabytes.

The most computational intensive part of FCMA involves a three-stage pipeline, as shown in Fig. 3. Before computation, FCMA reads in the preprocessed fMRI data (e.g., corrected for head motion and other noise sources) and the text files specifying the labeled time epochs over which the correlation is to be computed.

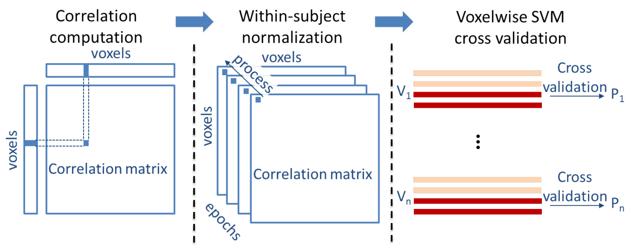

Figure 3: The three-stage pipeline of FCMA.

The first stage of FCMA is *correlation computation*. We operationalize the temporal interaction between two brain voxels during a time epoch of interest using Pearson correlation, which is computed as

$$corr(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \quad (1)$$

where X and Y are two vectors depicting the BOLD activity of voxel x and y in a time epoch, and cov and σ refer to the covariance and the standard deviation of the vectors, respectively. We reduce the computation of the Pearson correlation between voxel pairs to the multiplication of a voxel-by-time matrix and its transpose by normalizing the data within each time epoch[32]. Specifically, the reduction first subtracts the mean value of the time epoch vector, and then divides this mean-centered vector by its root sum of squares as

$$x'_i = \frac{(x_i - \bar{x})}{\sqrt{\sum_{j=1}^{n} x_j^2 - n\bar{x}^2}} \quad (2)$$

where x_i and x_j are elements of time epoch vector X, and $\bar{x}$ depicts its mean value. The Pearson correlation between two of the resulting vectors X' and Y' is their pointwise product

$$corr(X,Y) = X' \cdot Y' \quad (3)$$

which for an arbitrary number of vectors becomes an inner product matrix multiplication. A more detailed derivation can be found in the appendix of [29]. To obtain the full correlation matrix for a given time epoch, $corr(X,X)$ is calculated by taking the product of X' and its transpose. Such reduction simplifies the computation. Suppose there are N voxels in the whole brain (here $N \approx 35,000$), each matrix multiplication yields a $N * N$ full correlation matrix C, in which entry C_{ij} represents the correlation over time in an epoch between voxel i and j.

The second stage is *within-subject normalization*. The full correlation matrices with various labels from the first stage are the input of this stage. Because Pearson correlation coefficients are bounded $[-1,1]$ and not normally distributed near the bounds, we apply the Fisher transformation to every resulting correlation coefficient as

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right) \quad (4)$$

where r is the correlation coefficient between any two voxels. After that, to put the correlation coefficients from different subjects on the same scale for the cross-subject classification, we also apply z-score transformation within each subject

$$\forall z \in P, z' = \frac{z-\mu}{\sigma} \quad (5)$$

where P is the population of Fisher-transformed correlation coefficients within subject, and μ and σ are the mean and standard deviation of the population, respectively. The output of the second stage is then grouped by voxels so that the correlation vectors for the same voxel from all epochs are stored together as the input of the third stage (different labels depicted as dark red and light pink in Fig. 3).

The third stage is *SVM cross validation*. This stage performs voxel-wise cross validation to identify which voxels are informative in terms of their correlations with other voxels. Note that for each voxel there are only a few hundred correlation vector samples (for cross-subject classification, number of epochs per subject × number of subjects) while each sample has ~35,000 dimensions (corresponding to the number of voxels in the brain), so we use linear SVM to avoid overfitting. Linear SVM handles high-dimensional data better than SVM with other kernels such as polynomial or Gaussian, as well as other classification algorithms such as logistic regression. For each voxel, we extract its corresponding rows from all correlation matrices. Each of these rows contains the normalized correlation values (stage 2) between this voxel and all the other voxels in the brain in a time epoch. These correlation vectors are then labeled with the experimental conditions to which their epochs correspond and fed into linear SVM as samples. Linear SVM runs cross-validation across subjects (leave one subject out at a time) to assign a classification accuracy value to each voxel, quantifying its ability to distinguish between conditions.

3.1.1 *Cluster parallelization framework*

We use a task-based parallel framework on a compute cluster to process this pipeline, in which a master assigns tasks to workers. The master node first distributes brain data to the worker nodes and then sends tasks to the workers to process in parallel. A worker works on one task at a time. When a worker finishes a task, it will receive a new task from the master.

The tasks are defined by partitioning the correlation matrices along their rows. Each task is one run of the three-stage FCMA algorithm for the assigned number of voxels.

3.1.2 *Three-stage algorithm on a worker node*

We then describe the three-stage algorithm on a single worker node after being assigned a number of voxels as a task. In stage one, a worker node computes correlation vectors for all epochs of its given voxels as shown in Fig. 4. The number of given voxels is typically a few hundred or less to permit all correlation data to fit into memory. Since in most cases one time epoch consists of less than 20 time points, the matrix multiplications used to compute correlation have one very small dimension (the k dimension) which limits performance. On the other hand, in order to facilitate the subsequent steps, we arrange the data in memory such that all correlation vectors corresponding to a single voxel are contiguous, which means the result of the correlation matrix multiplications must be interleaved row by row. Fig. 4 illustrates the data layout in different colors, where the activity values of the first and second epochs are colored in dark red and light pink, respectively. The first voxel out of the V voxels computes its correlation vector of the first epoch and places as the first row of the correlation data of the first voxel depicted in dark red; the last voxel computes its correlation vector of the second epoch and places as the second row of the correlation data of the last voxel depicted in light pink.

In stage two, the worker node applies Fisher transformation and within subject z-scoring to the resulting correlation coefficients of its assigned voxels from stage one. As depicted in the bottom half of Fig. 4, a voxel's M correlation vectors are partitioned into E epochs per subject (dashed lines). For within-subject normalization, a sub-column of E values belonging to the same subject (e.g. the vertical black line in the second subject of the first voxel in Fig. 4) is extracted as a population P in formula 5 to process together.

In stage three, the normalized correlation vectors for the assigned voxels with two different labels can then be sent to the linear SVM classifier to determine how well the vectors differentiate the labeled categories via cross validation. The attained accuracies of these voxels are sent back to the master node.

Finally, the master node collects all voxels and sorts them by their resulting accuracies of cross validation. The brain regions constituted by top voxels are identified as ROIs in terms of correlation for following studies.

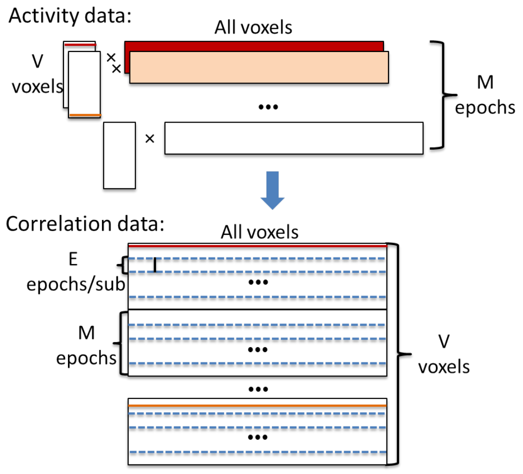

Figure 4: Correlation Computation for V voxels in M epochs, the results are grouped by voxels and then normalized within subject (assuming E epochs per subject).

3.2 Baseline Implementation

We implemented the three-stage algorithm described above in C++. All floating point values are represented in single precision. Our baseline implementation can be viewed as a typical implementation using sound programming discipline and the state-of-the-art libraries.

For correlation computation in the first stage, we allocate continuous aligned space in the memory to store the correlation vectors of all voxels so that the *cblas_sgemm* routine of MKL can be called to compute the correlation vectors epoch by epoch and place the results in an alternate way grouping by voxels via specifying the parameter *ldc* appropriately. The normalization of the second stage is parallelized along all voxels while applying vectorization within z-scoring. The SVM training that takes place in the third stage is essentially an algorithm that repeatedly calculates the matrix-vector product of one voxel's corresponding data matrix with different rows of the matrix itself. Since there are many more voxels (corresponding to the length of a correlation vector) than there are training examples (one time epoch corresponds to one sample), we precomputed all such matrix-vector products (also called the kernel matrix) before beginning SVM cross validation. Given that it is a linear SVM and the kernel function is a dot product, this kernel matrix can be cast as a symmetric matrix multiplication and solved with the *cblas_ssyrk* routine of MKL over one voxel's corresponding data matrix and its transpose. The SVM cross validation works based on the precomputed kernel matrix by applying the sequential minimal optimization (SMO) algorithm implemented in the LibSVM[6] package for training. SMO is an iterative method to solve large quadratic programming problems in the training phase of SVM. It breaks the problem into smallest possible sub-problems and each time solves one analytically until convergence[24]. In this way, one thread takes care of one voxel's kernel matrix computation and cross validation at a time so that different voxels can progress simultaneously.

We have tuned this implementation by carefully designing the data structures to utilize the high performance MKL routines. We have also deliberately precomputed the kernel matrices to avoid duplicate pairwise kernel computation and to keep more frequently used data in the cache. In practice, this results in good performance on a cluster consisting of nodes with Intel® Xeon® processors, where FCMA runs in the master-worker mode communicating via MPI calls, and the master node allocates different sets of voxels to different workers for processing.

3.3 Performance Analysis

This subsection reports our analysis of the baseline implementation on the Intel® Xeon Phi™ coprocessors. All of our measurements were collected by running the baseline implementation on the *face-scene* dataset (more details in Section 5), which contains brain data with 34,470 voxels, in 216 12-time-point epochs with two different labels. The master node assigns 120 voxels to a worker node as a single task for processing. We analyzed the performance of a single worker task.

3.3.1 Low efficiency of matrix multiplication

The first finding is that the matrix multiplications of MKL does not perform efficiently for our tall-skinny matrices on the coprocessor.

At the correlation computation stage, a worker is responsible for 120 voxels. It calls *cblas_sgemm* to perform 216 matrix multiplications between 120×12 and 12×34,470 matrix pairs to obtain correlation coefficients between the assigned 120 voxels and the entire brain over 216 epochs. At the SVM cross validation stage, it computes 120 symmetric kernel matrices between 216×34470 matrices and the corresponding transposes for 120 voxels using *cblas_ssyrk*.

The first row of Table 1 summarizes the performance of the matrix multiplication routines using Intel® vTune™ Amplifier. There are three performance issues. The first issue is that the number of memory references is too high. Our instrumentation shows that there are 34.9 billion memory references, whereas the matrix multiplications for the correlation computation and for SVM cross validation stages should have fewer than 10 billion.

	time	#mem refs	L2 miss	Vector intensity
Matrix multiplication	1830 ms	34.9 billion	709 million	3.6
Normalization	766 ms	6.2 billion	179 million	8.5
LibSVM	3600 ms	23.0 billion	7 million	1.9

Table 1: The instrumentation results of the baseline.

The second issue is that the cache miss overhead is high. Since empirically the latency of a L2 cache miss on the coprocessor is ∼250 CPU cycles from remote L2 cache and ∼302 CPU cycles from memory[11], and the clock rate of the coprocessor (5110P) is 1053MHz, we can estimate the latency of an L2 cache miss to be ∼300 ns so the total latency of L2 cache misses could be as high as ∼880 ms if not well hidden by other operations, which is significant compared to the total elapsed time (1830 ms).

The third issue is that the vectorization intensity value is only 3.6 while the ideal vectorization intensity is 16. Only 23% of the VPU capability is used during computation.

These three observations show that when computing our tall-skinny matrices, MKL doesn't leverage L2 cache well and does poorly to take advantage of the VPUs of the coprocessor.

3.3.2 Lack of cached data reuse between stages

By manually calculation, we noticed that the second stage of FCMA causes ∼112 million compulsory L2 cache misses. The first stage (correlation computation) generates correlations and writes them into their data structures. The second stage (within-subject normalization) reads the data back to perform Fisher transformation and z-scoring.

Optimization within a function cannot avoid such compulsory L2 cache misses between function calls. When such situations happen between two stages of the processing pipeline, retaining cache contents becomes difficult. To avoid such cache misses, we need to have higher-level optimizations.

In addition, Table 1 shows that the vectorization intensity of within-subject normalization is 8.5, indicating that there are rooms to improve the utilization of the vector unit.

3.3.3 Poor VPU utilization in SVM cross validation

We noticed that the SVM cross validation stage takes a lot of time and most of the time spent in the LibSVM library.

The first reason is that the vectorization intensity of LibSVM is only 1.9 (Table 1), indicating it does not take advantage of the vector unit of the coprocessor well. As we started looking at the code of LibSVM, we found that it stores data in sparse index set instead of dense matrix.

The second reason is LibSVM does unnecessary data type conversions during computation and uses double precision values in the computationally intensive loops.

The third reason is about the memory limitation of the coprocessor. The implementation uses one thread to run cross validation for one voxel. Therefore, the master node needs to assign at least 240 voxels at a time to the coprocessor for fully utilizing its available 240 threads. However, each 5110P coprocessor has only about 6GB memory available to applications. 240 voxels' correlation vectors will consume 8.3GB memory. This forces the master node to only assign a small number of voxels to the coprocessor once, consequently the computing resource is under utilized during the linear SVM cross validation stage.

The poor performance of LibSVM motivated us to optimize the LibSVM. In addition, we implemented *PhiSVM* based on a GPU SVM implementation[5]. We will report the performance of these implementations in Section 5.

4. OPTIMIZATIONS

Based on the optimization opportunities identified above, we came up with three optimization ideas. This section first describes our ideas and then presents how we optimized our implementation.

4.1 Optimization Ideas

We employed three optimization ideas to optimize the FCMA algorithm for a single worker node task:

1) Partitioning tall-skinny matrices for blocking to fit small amount of L2 cache for each thread. The traditional way of blocking is to change the looping structure of the code to process a square block of data in inner loops. However, this approach would exceed the relatively small L2 cache per thread on the coprocessor.

2) Retaining cache contents across stages of the procedure pipeline. Our approach is to look at the contents of L2 cache in the current stage. If the contents will be reused frequently in the next stage, we will consider merging the two stages to avoid cache misses at the next stage. Typically, a cache conscious algorithm of the current stage uses blocked data structure to reduce cache misses. When finishing the computation with the blocks, it will proceed with the next stage computation without waiting for other blocks of the current stage to complete their computations. Obviously, merging stages will reduce modularity of the program. So one needs to be careful when applying this idea.

3) Designing data structures and workflow for vectorization. A vector unit typically requires its data to be layout in memory in consecutive fashion so that they can be moved into and out of its large register file quickly. This optimization ensures that the data structures fit the required layout to maximize the utilization of the vector unit.

4.2 Correlation Computation

As mentioned before, computing correlations are reduced to matrix multiplications. At this stage, a single worker node needs to correlate its assigned V voxels with all brain voxels over all epochs of interest, so the job is essentially doing matrix multiplications between relatively small matrices and tall-skinny matrices as illustrated in the top half of Fig. 5. Our optimization idea #1 is applied to block voxel data as depicted in dark red in Fig. 5. Performing computation within blocks won't trigger unnecessary L2 cache misses. In order to fully utilize the VPUs, we defined the size of blocks to be integral multiples of the vector unit width. Also, we consciously transpose the block of tall-skinny matrix to better utilize the VPUs (idea #3).

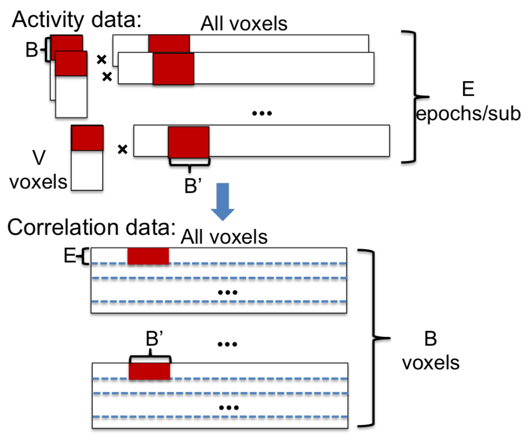

Figure 5: Retain correlations (red blocks) in L2 cache for the normalization stage.

4.3 Within-Subject Normalization

The within-subject normalization stage of FCMA is the second pipeline stage in Fig. 3. It uses the data computed in the correlation computation stage. Applying idea #2 will allow us to avoid many cache misses at this stage.

After computing the correlation coefficients in local blocks, the normalization can be applied to the data before they are written back. What we need to take into consideration, besides the blocking, is that the data necessary for a complete normalization should reside in the same block. It is a cache locality driven job scheduling approach.

Fig. 5 indicates the merging process. Out of V assigned voxels, each thread only takes B voxels to compute their correlations with some other B' voxels for E epochs belonged to one subject, yielding B portions of within-subject correlation coefficients that can be normalized. Note that all blocks in red can be fit into L2 cache, we won't spend additional time to fetch the data between stages. Using different threads to handle different blocks in this way, all correlation data would be computed and immediately normalized then written to the memory. The idea of merging adjacent stages in the procedure pipeline can be generalized to all memory-bound processes where the memory latency cannot be hidden completely by the computation.

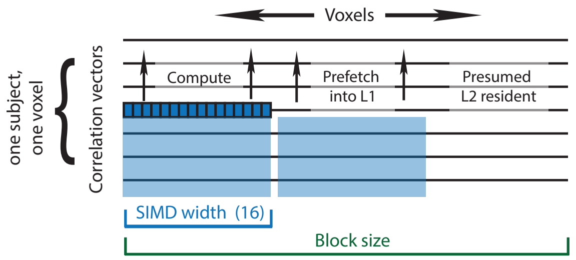

Figure 6: The z-score normalization deals with correlation vectors in L2 cache and processes 16 voxels at a time using SIMD instructions.

In addition to merging stages, we also apply our optimization idea #3 to this stage. Our approach to vectorizing the normalization procedure is shown in Fig. 6, which details the processing that takes place for each dark red block shown on the bottom half of Fig. 5. The correlation vectors are placed contiguously into a temporary block of memory. We process the data in chunks of 16 voxels using SIMD instructions and registers (enabled with the *#pragma SIMD* directive). To compute the Fisher transformation, we must perform a $logf$ operation on each data element. On the coprocessor, the $logf$ computation benefits from hardware support for single precision transcendental functions in the extended math unit (EMU). During the first pass through the data, we also compute the mean and standard deviation across correlation vectors. We use the $E[X^2] - E[X]^2$ formulation for variance in order to compute both the mean and standard deviation in one pass. As one block of 16 voxels are processed, we prefetch the next set of voxels into the L1 cache using *_mm_prefetch(_MM_HINT_T0)* We do not do L2 software prefetching because the data are presumed to be L2 resident by our blocking design. After the mean and standard deviation are computed, a second pass through the data subtracts the mean and scales by the inverse of the standard deviation.

4.4 SVM Cross Validation

This stage consists of two parts, a linear kernel matrix precomputation followed by linear SVM cross validation over the precomputed kernel matrices.

SVM kernel matrix precomputation in FCMA is essentially a matrix multiplication between a voxel's $M * N$ data matrix and its transpose, where M is the number of epochs over which the correlation is computed, and N is the number of voxels in the brain, normally $M << N$. We implemented a custom symmetric matrix multiplication function for the coprocessor which attempts to optimize for our particular setting by applying our optimization ideas #1 and #3 similar to the first stage.

Fig. 7 shows the workflow of our optimized implementation. Since one worker node deals with certain number of voxels simultaneously, a number of independent matrix multiplications are running in parallel, one per voxel, as shown in the depth dimension. The size of a data matrix is typically ∼60 MB (400 epochs times 35,000 voxels matrix stored in single precision values), making it possible to consume the ∼6GB of on-board memory of the coprocessor by processing only a 100 voxels' worth of matrices. Therefore the number of independent, concurrently executed matrix multiplications is limited. They cannot saturate the coprocessor, which compels us to split the problems across multiple threads and use OpenMP locks to control access to the C matrices.

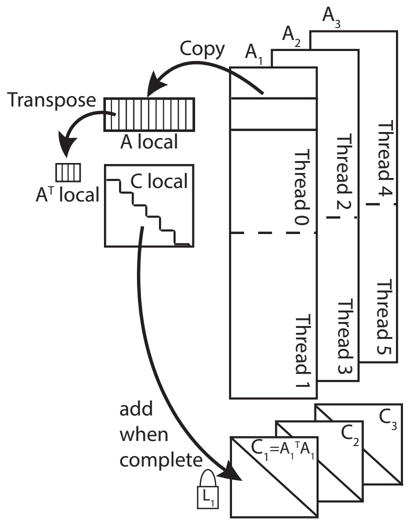

Figure 7: Multiple tall-skinny matrix multiplications (*sryk*) are performed to precompute the SVM kernel matrices.

Each thread proceeds down the long dimension of the matrix in blocks of 96 rows (an integral multiple of VPU length). These blocks are copied into a local buffer (A_local). Smaller blocks of A local are then transposed and copied into a smaller buffer (A^T_local). Once the data is ready, we call an auto-generated 16x9x96 assembly-level matrix multiply routine in the inner loop to generate each block of C_local. We pad A_local with zeros for the last block if A's height is not a perfect multiple of 96, and use vectorized loops for the other dimensions. After the thread completes its portion of the matrix multiply, it takes a lock corresponding to the C matrix and adds its contribution to C [22].

Regarding the SVM cross validation algorithm over the precomputed kernel matrix, in order to circumvent the drawbacks of LibSVM mentioned above, we adopt a fast SVM algorithm[5] implemented for GPUs and rewrote the CUDA code into C++ to run on the coprocessor. Like LibSVM, this fast SVM applies SMO algorithm to solve a SVM training problem. Typically, a single iteration of SMO algorithm involves choosing two rows from the kernel matrix and using them to update information for all other rows. The choice of the two rows is done heuristically. But instead of only using the LibSVM heuristic in the working set selection, our fast SVM adaptively chooses the faster heuristic (either first order[17] or second order[10]) based on the convergence rate on the specific training data. The original CUDA implementation aims at solving huge SVM problems involving tens of thousands of samples by coordinating all GPU cores to work together. In our problem setting, we face a large number of smaller scale SVM problems, one per voxel, each of which only contains a few hundred of samples and has the kernel matrix (linear kernel) precomputed. Therefore, we make it so that a thread takes full responsibility for the cross validation of one voxel. Moreover, using our optimization idea #3, we vectorize the most computationally intensive part of the code for better usage of VPUs. We call the adopted fast SVM algorithm implemented on the coprocessor *PhiSVM*.

Another challenge is finding enough parallelism (i.e. independent SVM problems) to fully utilize the coprocessor during the cross validation stage without exceeding the limits of the on-board memory. This can be solved by redesigning the computing procedure, in which we accumulate a least 240 voxels' kernel matrices before conducting the SVM cross validation. Since a kernel matrix is significantly smaller than a data matrix, reducing to kernel matrices can save a lot of space so that doing SVM cross validation for at least 240 voxels becomes possible, therefore no available computing power will be wasted during the SVM cross validation.

5. EVALUATIONS

To evaluate the proposed optimization ideas, we pursued answers to following questions:

1. What is the system performance for the offline and online data analysis cases?
2. Is the system scalable as we add more coprocessor nodes to the cluster?
3. For a typical dataset, what are the performance contributions of the proposed optimization ideas?
4. How does the resulting system work on general-purpose processors?

To answer these questions, we will first describe our experimental setup and then present our results and analysis.

5.1 Experimental Setup

We ran our experiments on a 48-node cluster, interconnected by an Arista 10GE switch. Each node of the cluster has a motherboard with: 2 Intel® Xeon® E5-2670 processors, both running at a 2.6GHz clock rate, 2 Intel® Xeon Phi™ 5110P coprocessors running at 1053MHz, connected via PCI-e slots, 256GB memory, 8 x 3TB SATA disks, and 1.65TB FusionIO ScaleIO Flash memory card.

The host node runs CentOS 6.3 and each coprocessor runs its on-board Linux (version 2.6.38.8). The software packages used in our experiments include MPSS (version 3.3), Intel MPI (version 5.0.2.044), and Intel compiler (version 15.0.2) [2]. Intel® MKL (version 11.2) and LibSVM (version

[2]Intel's compilers may or may not optimize to the same degree for non-Intel

3.20) libraries were applied to the baseline implementation for comparison. We used Intel® vTune™ Amplifier (version 2015.2.0.39344) to collect the performance data.

Our experiments used two fMRI datasets. The first one is a *face-scene* dataset consisting of fMRI data from 18 subjects who passively viewed either face or scene images as described in [30]. The second one is an *attention* dataset used in [16] consisting of fMRI data from 30 subjects who were asked to look at either left or right images on a screen while being scanned. Detailed information on the datasets is listed in Table 2.

Dataset	Voxels	Subjects	Epochs	Epoch length
Face-scene	34,470	18	216	12
Attention	25,260	30	540	12

Table 2: Datasets used in the experiments.

5.2 System Performance

In this section, we present the results for both offline and emulated online data analysis. The offline case is time consuming because it involved leave-one-subject-out nested n-fold cross validation, which we elaborate on below, on all n subjects in the dataset. The emulated online data analysis involved selecting voxels to train a classifier using one subject's data on the fly to provide real-time neurofeedback in subsequent experiments.

5.2.1 *Offline analysis performance*

We ran the nested leave-one-subject-out n-fold cross validation on both *face-scene* and *attention* datasets, where n is the total number of subjects in a dataset. In each fold of the outer loop cross validation, a training set consisting of $n-1$ subjects was used for voxel selection by conducting another level of leave-one-subject-out cross validation. Voxels were selected based on their classification accuracies of their correlation vectors, determined by the procedure illustrated in Fig. 3. After voxel selection in each fold, a final classifier can be trained using the correlation patterns of the selected voxels to test on the left out subject of the outer loop to verify the selection. In addition, the selected voxels across different folds can be statistically compared to identify the reliable voxels whose correlation patterns with the rest of the brain are informative[30].

Table 3 shows the elapsed time in seconds as a function of the number of coprocessor nodes. For the *face-scene* dataset, the experiment ran 18 folds of leave-one-subject-out validation. The whole process took 85 seconds using 96 coprocessors. On average, each fold took 4.7 seconds.

For the *attention* dataset, the experiment ran 30 folds of leave-one-subject-out validation. The whole process took 741 seconds using 96 coprocessors. On average, each fold took 24.7 seconds. The *attention* experiment took longer because it involved more subjects and each subject performed more epochs.

We reproduced the results used in [30] and [16].

microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

#nodes	1	8	16	32	64	96
Face-scene	5101	694	385	242	124	85
Attention	54506	6813	3620	2172	1099	741

Table 3: The elapsed times (seconds) of offline data analysis as a function of the number of coprocessors used.

5.2.2 *Emulated online analysis performance*

In closed-loop rtfMRI study of brain interactions, a classifier needs to be trained online using the voxels selected by FCMA based on the whole-brain correlation. This classifier will be used for sending feedback to the subject while being scanned. The voxel selection procedure is similar to the offline analysis, except that instead of taking data from multiple subjects to process in batch, we only use the data received from the subject being scanned, and no nested cross validation is applied.

We performed the emulated online analysis on one subject's data from each of the *face-scene* and *attention* datasets. The voxels for building the classifier were selected using the subject's data. Table 4 shows the elapsed time in seconds as a function of the number of coprocessors. Using 96 coprocessors, voxels were selected within 3 seconds which is fast enough to build an online classifier to provide real-time feedback to the subject in the subsequent experiments.

#nodes	1	8	16	32	64	96
Face-scene	12.00	3.18	2.51	2.26	2.24	2.21
Attention	16.50	3.96	2.97	2.59	2.52	2.51

Table 4: The elapsed time (seconds) of voxel selection for building the online classifier as a function of the number of coprocessors used.

5.3 Speedup

We studied the scalability of our optimized implementation of FCMA by varying the number of coprocessors used in the offline analysis.

Fig. 8 shows the speedup of both datasets as a function of the number of coprocessors. With 96 coprocessors, we achieved a 59.8x on the *face-scene* dataset and a 73.5x on the *attention* dataset. The speedup is greater for the *attention* dataset due to its larger size.

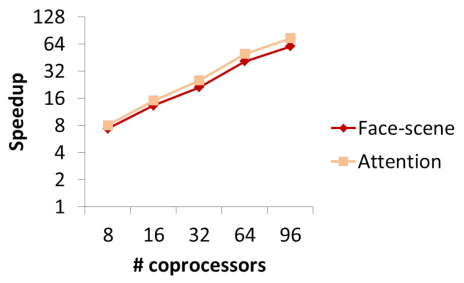

Figure 8: Speedups of the optimized implementation with *face-scene* and *attention* datasets.

5.4 Performance Implications of Optimizations

In this subsection, we focus on the performance of a single coprocessor to demonstrate the contributions of our proposed optimization ideas.

5.4.1 *Performance of a three-stage task*

We first compare the overall performance of a single coprocessor for our optimized implementation and the baseline implementation with MKL and LibSVM libraries. Both *face-scene* and *attention* datasets were used.

As we described in Section 3.1.1, in the parallel FCMA framework, the master node distributes tasks to worker nodes by partitioning all voxels into pieces. In the baseline implementation, due to the memory limitation of the coprocessor, the master only can allocate 120 voxels of the *face-scene* dataset or 60 voxels of the *attention* dataset to a coprocessor for processing, respectively. As a result, the third stage of the baseline implementation of the FCMA pipeline cannot fully exploit the available hardware capability of the coprocessor, since one thread takes care of only one voxel's cross validation. This constraint is largely relaxed in the optimized implementation in which a coprocessor can take more voxels (e.g. 240) by reducing the large correlation data into much smaller precomputed kernel matrices portion by portion to guarantee the consumed memory size doesn't go beyond the available on-board memory of the coprocessor.

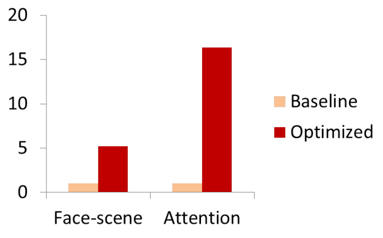

Figure 9: Improvement of the optimized implementation over the baseline for a single worker task. The baseline performance is normalized to 1.

Since the number of voxels one coprocessor can take differs between the baseline and optimized implementations, we normalized the performance to processing time per voxel. Fig. 9 shows the speedup of the optimized implementation over the baseline. The performance of the baseline was set to 1 for both datasets. Our optimized implementation runs 5.24x and 16.39x faster than the baseline, respectively. For *attention* dataset, the fraction of time spent in SVM computation is significantly larger, and hence benefits much more from our optimizations.

Next, we break down the contributions based on the proposed optimizations. We compared the performance of processing a task of 120 voxels from the *face-scene* dataset in a single coprocessor.

5.4.2 *Blocking skinny matrices (vs. MKL)*

There are two matrix multiplications in two stages of the FCMA processing pipeline: in correlation computation and in SVM cross validation. For offline analysis on *face-scene* dataset, at the correlation computation stage, FCMA performs 216 (epochs) multiplications of matrix of A[120,12] with B[12,34,470] and writes results to matrix C[120,34,470] (illustrated in Fig. 4). 21.443 billion floating-point operations and 4.136 million memory writes are performed.

In the SVM cross validation stage, FCMA performs a multiplication of matrix A[204,34,470] with its transpose A^T and writes results to matrix C[204,204] (illustrated in Fig. 7). Since A is multiplied with its transpose, only upper or lower triangle of the resulting matrix needs to be computed. This matrix multiplication performs 172.14 billion floating point operations but only 20,088 memory writes.

Table 5 reports the elapsed times and Giga FLoating Operations/Seconds (GFLOPS) for both cases. Our optimized matrix multiplication achieved 126 GFLOPS in the correlation computation stage and 430 GFLOPS in the SVM cross validation stage. The matrix multiplication using MKL achieved 93 GFLOPS for the correlation computation stage, and 108 GFLOPS for the SVM cross validation stage.

The matrix multiplication in the correlation computation stage produced many more writes than the SVM cross validation stage did, explaining why the latter reached 3.4x higher GFLOPS.

	Function	Time	GFLOPS
Our blocking	correlation matrix computation	170 ms	126
	SVM kernel matrix computation	400 ms	430
MKL	correlation matrix computation	230 ms	93
	SVM kernel matrix computation	1600 ms	108

Table 5: The performance results of matrix multiplication routines used in both correlation computation and SVM cross validation stages.

Table 6 shows the total number of memory references, the number of L2 cache misses, and the vectorization intensities of the matrix multiplication routines in our optimized implementation and in MKL. These are the combined results of both stages. The vectorization intensity measured from vTune for our optimized implementation was close to the theoretical peak value 16, whereas that for MKL was 3.6. The results show that our implementation took full advantage of the vector unit, whereas MKL only utilized 23% of the hardware capability. Our implementation had 5.82x fewer L2 cache misses than MKL (121.8 vs. 708.9 million). These results show that MKL performed relatively poorly because of its large number of L2 cache misses and low vectorization intensity.

Our measurements also indicate that MKL made 3.49x more memory references than our implementation (34,858.37 vs. 9,974.87 millions).

	#memory refs	L2 miss	Vector intensity
Our blocking	9,974,870,500	121,800,000	16
MKL	34,858,368,500	708,900,000	3.6

Table 6: Memory references, L2 misses and vector intensity of the matrix multiplication routines.

5.4.3 *Retaining cache contents*

To understand the performance impact of modifying the FCMA algorithm to retain cache contents across the correlation computation and normalization stages, we implemented two cases: separated and merged.

As we discussed in Section 4.3, the merged implementation performed normalization on correlations as soon as they were computed, without waiting for the entire correlation computation stage to finish. The data in the L2 cache was retained for within-subject normalization without writing out to memory and reading back in again. Conversely, the separated implementation finished all correlation computation before moving forward to the normalization stage.

Table 7 shows the results of elapsed times, number of memory references, and number of L2 cache misses for both implementations. The merged implementation had a fewer number of memory references (1.93 vs. 4.35 billion) and a fewer number of L2 cache misses (67.5 vs. 188.1 million), resulting in a 24% reduction in elapsed time.

Method	Time	#memory refs	L2 miss
merged	320 ms	1,925,806,500	67,500,000
separated	420 ms	4,347,490,500	188,100,000

Table 7: Performance comparisons of retaining L2 cache contents (merged stages vs separated stages).

5.4.4 *Vectorization for SVM*

To determine the performance impact of vectorization in the coprocessor, we compared LibSVM, our optimized LibSVM, and our optimized PhiSVM.

In both optimized implementations, we reorganized the data layout and workflow for the computationally intensive loops in order to better utilize the VPUs of the coprocessor. Since single precision floating point numbers are accurate enough for our application, we used *float* type in PhiSVM. For fair comparison, we also converted all *double* type values in LibSVM to *float* type so that the VPU can process an equal number of values in a single *SIMD* instruction.

Table 8 shows the elapsed times and vectorization intensities for all three implementations. Optimized PhiSVM took 390 ms whereas optimized LibSVM and the single precision LibSVM took 1,150 ms and 3,600 ms, respectively. PhiSVM outperformed LibSVM even after its careful vectorization because of the advances in algorithm and data structure as described in section 3.3.3.

	Time	Vector intensity
LibSVM	3600 ms	1.9
Optimized LibSVM	1150 ms	8.9
PhiSVM	390 ms	9.8

Table 8: The performance of SVM cross validation.

5.5 Performance on Intel® Xeon® Processors

To determine how well our optimizations for the coprocessor would work on a general-purpose processor, we compared our optimized implementation with the baseline implementation on a single E5-2670 processor in one node of our cluster, described in Section 5.1. This experiment was identical to what is described in Section 5.4.1.

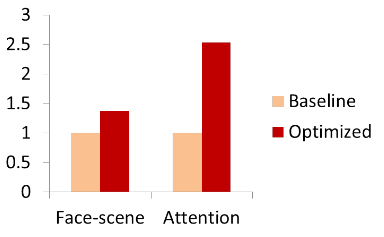

Figure 10: The performance improvement of our optimized version over the baseline implementation on Intel® Xeon® E5-2670 processor. The baseline performance is normalized to 1.

Figure 10 shows that our optimizations for the coprocessor worked quite well for the processor. Our optimized implementation ran 1.4x and 2.5x faster than the baseline for the *face-scene* and *attention* datasets, respectively.

The performance improvements on the processor were significant but less dramatic than on the coprocessor for several reasons. First, the processor has a relatively large Last Level Cache (LLC) per CPU core or per thread. It has 8 CPU cores and 16 hyperthreads and 20MB LLC. On average, each thread has 1.28MB LLC per thread, which is an order of magnitude larger than that for the coprocessor. The large cache allows fewer LLC cache misses, making the performance tuning for L2 cache misses less important.

Second, the width of vector registers on the processor is 256-bit, only half of that on the coprocessor. The narrower vector unit makes the effect of vectorization less significant.

Third, the processor supports 2 hyperthreading threads, whereas the coprocessor supports 4 per core. The total number of concurrent threads on the processor is 16 versus 240 on the coprocessor. Therefore, the thread starvation issue presenting during SVM cross validation of the baseline implementation on the coprocessor doesn't exist on the processor.

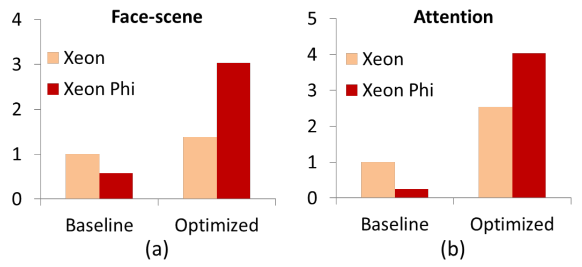

Figure 11: The performance comparisons between Xeon® E5-2670 processor and Xeon Phi™ 5110P coprocessor (a) *face-scene* dataset; (b) *attention* dataset. The baseline performance of E5-2670 processor is normalized to 1.

We also compared the baseline and the optimized implementation on the processor and coprocessor. Fig. 11 shows that the optimized implementation on the coprocessor outperformed the same code running on the processor for both *face-scene* and *attention* datasets.

6. RELATED WORK

Many scientific computing applications in multiple disciplines such as physics and chemistry have taken advantage of manycore architectures such as the Intel® Xeon Phi™ coprocessors[3, 15]. To the best of our knowledge, this paper presents the first neuroscience application that adopts the coprocessor for achieving performance goals (for both offline and online analysis). We have improved upon previously published FCMA runtime results [30]. Even our baseline implementation is 9.7X faster than [30]. Overall, using the coprocessor we achieve a 50.8X speedup. While some of the performance difference can be attributed to the different processor generations used, much of it is due to the algorithm and performance optimization.

Data locality optimizing algorithms for improving the efficiency of accessing data residing in memory hierarchies have been well studied for a long time[18, 31]. Several optimizations have been performed to fully exploit the processor architecture in order to achieve high performance of linear algebra operations especially matrix multiplication[13], based on which different versions of BLAS routines such as MKL and GotoBlas were implemented. There are studies about optimizing matrix multiplication on manycore architectures such as GPUs[19, 26, 28] and some recent work on Intel® Xeon Phi™ coprocessors[12, 14]. Most of the optimizations for *GEMM* focus on coordinating multiple threads to conquer huge, nearly-square matrices. Our application, on the other hand, requires a single thread to work on one matrix multiplication between matrices with one small dimension. Dense matrix multiplication involving tall-skinny matrices is known to be difficult to optimize[8]. Tall-skinny matrix operations appear in other contexts as well, such as QR decomposition or eigenvalue problems [1, 2, 4, 20]. Cache locality optimizations are among the most important optimizations required for tall-skinny problems. Cache locality driven thread scheduling (e.g. [23]) is a general way to block data efficiently. We have implemented similar ideas (Section 4.2.1 and 4.2.2) in our pipeline for better L2 cache reuse.

While other techniques for solving linear SVM exist (such as [9]), we use *PhiSVM* (which is derived from SMO-based techniques such as [5] and [10]) as it is fast and efficient for small SVM problems that we solve for FCMA. PhiSVM is also usable in other applications that require an efficient coprocessor-based SVM library.

The optimization ideas presented in this paper (such as tall-skinny matrix multiplication) are relevant and generalizable to a lot of other applications as well e.g. [20].

7. CONCLUSIONS

This paper describes an emerging neuroscience application FCMA and its optimization on Intel® Xeon Phi™ coprocessors. Our optimized implementation for a single node task on the coprocessor runs 5x-16x faster than an optimized baseline version with MKL and LibSVM libraries for two different datasets. Our optimization also improves the performance on the E5-2670 processor by a factor 2. In addition, we show that our parallel code achieves near linear speedup on 5760 coprocessor cores. This work reduces the previously intractable timescale of computing and analyzing the full correlation matrix in an fMRI dataset of the human brain to minutes for offline analysis, and seconds for online analysis. This latter finding makes plugging FCMA into established closed-loop rtfMRI studies possible.

Due to increasing amounts of parallelism as we move from multicore to manycore architectures, optimizations to exploit these hardware features become increasingly important. This paper optimized FCMA code by redesigning matrix multiplication for tall-skinny matrices, merging adjacent memory-bound stages in the procedure pipeline, and rewriting more vectorization-friendly SVM algorithms, in the consideration of making efficient use of the available hardware (cache and VPUs). The optimizations described in this paper can be generalized as independent components that have many other applications. We also showed that most of the optimizations done on the coprocessor works as well on the processor, although to a lesser degree, due to similar memory hierarchical structure and vectorization techniques. In addition, we believe our implementation can be migrated on to the next generation of Intel® Xeon Phi™ (KNL) with moderate effort.

Our future work will pursue two directions, computational and neuroscientific. Computationally, we plan to develop a more general framework for efficiently running a variety of applications on the Intel® Xeon Phi™ coprocessor. Neuroscientifically, we are using these new tools to explore other applications of the FCMA approach that will benefit from advanced high performance computing devices such as the Intel® Xeon Phi™ coprocessor.

8. ACKNOWLEDGMENTS

We would like to thank our shepherd Mihai Anitescu and the anonymous reviewers of the SC program committee for their valuable comments which improved the paper a lot. We are also grateful to Yungang Bao, Guangming Tan and Linpeng Tang for helpful discussion. This work was supported in part by Intel Corporation, the J. Insley Blair Pyne fund at Princeton University, the John Templeton Foundation, the National Science Foundation (MRI BCS1229597), and the National Institutes of Health (R01 EY021755).

9. REFERENCES

[1] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1213–1222, May 2014.

[2] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding qr decomposition for gpus. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 48–58, May 2011.

[3] E. Aprà, M. Klemm, and K. Kowalski. Efficient implementation of many-body quantum chemical methods on the intel® xeon phi™ coprocessor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 674–684, Nov 2014.

[4] T. Auckenthaler, T. Huckle, and R. Wittmann. A blocked qr-decomposition for the parallel symmetric eigenvalue problem. *Parallel Comput.*, 40(7):186–194, 2014.

[5] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on

graphics processors. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 104–111, Jul 2008.

[6] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):1–27, 2011.

[7] M. T. deBettencourt, J. D. Cohen, R. F. Lee, K. A. Norman, and N. B. Turk-Browne. Closed-loop training of attention with real-time brain imaging. *Nature Neuroscience*, 18(3):470–475, 2015.

[8] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Poster: Beating mkl and scalapack at rectangular matrix multiplication using the bfs/dfs approach. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1370–1370, Nov 2012.

[9] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, 2008.

[10] R.-E. Fan, P.-H. Chen, and C.-J. Lin. Working set selection using second order information for training support vector machines. *J. Mach. Learn. Res.*, 6:1889–1918, 2005.

[11] J. Fang, A. L. Varbanescu, H. J. Sips, L. Zhang, Y. Che, and C. Xu. An empirical study of intel xeon phi. *arXiv preprint arXiv:1310.5842*, 2013.

[12] P. Gepner, V. Gamayunov, D. L. Fraser, E. Houdard, L. Sauge, D. Declat, and M. Dubois. Evaluation of dgemm implementation on intel xeon phi coprocessor. *Journal of Computers*, 9(7):1566–1571, 2014.

[13] K. Goto and R. A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

[14] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi^{TM} coprocessor. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, pages 126–137, May 2013.

[15] S. Heybrock, B. Joó, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, and P. Dubey. Lattice qcd with domain decomposition on intel® xeon phi^{TM} co-processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 69–80, Nov 2014.

[16] J. Hutchinson, Y. Wang, and N. Turk-Browne. Decoding the locus of attention from the full correlation matrix of the human brain. In *Society for Neuroscience*, SfN '14, Nov 2014.

[17] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural Computation*, 13(3):637–649, 2001.

[18] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[19] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for gpus. In *Computational Science - ICCS 2009*, pages 884–892. Springer, 2009.

[20] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer. The elpa library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21):213201, 2014.

[21] K. A. Norman, S. M. Polyn, G. J. Detre, and J. V. Haxby. Beyond mind-reading: multi-voxel pattern analysis of fmri data. *Trends in cognitive sciences*, 10(9):424–430, 2006.

[22] H. Pabst. Libxsmm. https://github.com/hfp/libxsmm.

[23] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 60–71, Oct 1996.

[24] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, Apr 1998.

[25] J. Sulzer, S. Haller, F. Scharnowski, N. Weiskopf, N. Birbaumer, M. L. Blefari, A. Bruehl, L. Cohen, R. Gassert, R. Goebel, et al. Real-time fmri neurofeedback: progress and challenges. *NeuroImage*, 76:386–399, 2013.

[26] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, Nov 2011.

[27] N. B. Turk-Browne. Functional interactions as big data in the human brain. *Science*, 342(6158):580–584, 2013.

[28] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Nov 2008.

[29] Y. Wang, J. D. Cohen, K. Li, and N. B. Turk-Browne. Full correlation matrix analysis of fmri data. Technical report, Princeton Neuroscience Institute, 2014.

[30] Y. Wang, J. D. Cohen, K. Li, and N. B. Turk-Browne. Full correlation matrix analysis (fcma): An unbiased method for task-related functional connectivity. *Journal of Neuroscience Methods*, 251:108–119, 2015.

[31] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991.

[32] K. J. Worsley, J.-I. Chen, J. Lerch, and A. C. Evans. Comparing functional connectivity via thresholding correlations and singular value decomposition. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 360(1457):913–920, 2005.