# Porting a Neuro-Imaging Application to a CPU-GPU cluster

Reza Sina Nakhjavani[*], Sahel Sharify[*], Ali B. Hashemi[*], Alan W. Lu[*], Cristiana Amza[*], and Stephen Strother[†]
[*]Electrical and Computer Engineering Department, University of Toronto
[†]Rotman Research Institute, Baycrest, Toronto, Ontario
[†]Department of Medical Biophysics, University of Toronto
Email: [*]{sina,sahel,hashemi,luwen,amza}@eecg.toronto.edu, [†]sstrother@research.baycrest.org

***Abstract*—The ever increasing complexity of scientific applications has led to utilization of new HPC paradigms such as Graphical Processing Units (GPUs). However, modifying existing applications to enable them to be executed on GPU could be challenging. Furthermore, the considerable speedup achieved by execution of linear algebra operations on GPUs has added a huge heterogeneity to HPC clusters. In this work, we enabled NPAIRS, a neuro-imaging application, to be executed on GPU with slight modifications to its original code. This important feature of our implementation enables current users of NPAIRS, i.e. non-expert bio-medical scientists, to get benefit from GPU without having to apply fundamental changes to their existing application. As the second part of our research we investigated the efficiency of several scheduling algorithms for a heterogeneous cluster that contains GPU nodes. Experimental results show we achieved 7X speedup for NPAIRS. Moreover, although scheduling does not play an important role when there is no GPU node in the cluster, it can highly improve the makespan for a CPU-GPU cluster. We compared our scheduling results with Torque and MCT, two of the most commonly used schedulers in current HPC platforms. Our results show that the Sufferage scheduling can improve the makespan of Torque and MCT by 47% and 4% respectively.**

## I. Introduction

The demand for high performance computing is increasing everyday. Processing medical images is an example of a CPU-intensive application. This new class of applications is running too slow even on today's multi-core architectures. Although there is no need for most of such applications to be strictly real-time, being able to execute them on the order of a few minutes, instead of hours or days, helps researchers to test and evaluate their new ideas and algorithms quickly. Moreover, some degree of interactivity with the application is important for biomedical researchers using the neuroscience workloads that we are working with in this paper.

This drastic demand for higher performance has led the computer industry to incorporate multi-core and many-core processors in today's HPC platforms. NVIDIA's [1] GPUs and Intel's Xeon Phi are today's most common many-core architectures that are used as co-processors in computationally intensive applications. On the other hand, the emergence of General Purpose computing on GPUs (GPGPU) and their programming languages such as CUDA [2] and OpenCL [3], along with the integration of GPUs into existing multi-core machines has made them a viable solution to accelerate embarrassingly parallel applications. This paradigm has also added heterogeneity in today's desktops and laptops as well as cloud environments targeting HPC workloads.

Scheduling jobs for super computers has been extensively studied. However, the heterogeneous nature of recent modern super computers, as well as CPU and GPU clusters, demands that we revisit the portability and scheduling problem for such systems. GPUs have shown the ability to provide higher peak throughput for a wide range of massively parallel applications. Consequently, compute-intensive applications would prefer to be scheduled on a GPU-enabled server when running on a heterogeneous cluster. This may however lead to GPU device contention and decrease the overall throughput since there is usually a smaller number of GPUs than CPUs in typical clusters used in biomedical settings. Therefore, tasks may finish faster if run on a CPU rather than waiting for a GPU resource to become available. In such cases, scheduling some jobs on GPUs while running others on CPUs results in a better utilization of resources as well as higher peak throughput.

In this paper, we make the following contributions: i) we proposed a technique for porting and scheduling biomedical applications to CPU-GPU clusters with minimal application changes, and ii) we implemented and evaluated scheduling techniques for heterogeneous clusters to shorten execution time and optimize resource utilization.

Our design for portability is particularly important for biomedical applications. Because it allows for separation of concerns between any source code modifications or extensions normally performed by biomedical researchers, and any libraries used for the purpose of providing platform-dependent support. This will isolate the biomedical researchers from such lower-level concerns.

The scheduling algorithms we investigated in this paper range from relatively simple ones, such as shortest (estimated) job first, to more sophisticated ones, such as *Sufferage* scheduling algorithm [4], which tries to optimize the penalty that a task suffers if it cannot be scheduled on its preferred resource.

As our case study we selected NPAIRS, a biomedical application for processing functional Magnetic Resonance Imaging (fMRI) brain images. NPAIRS is used to determine the correlation between brain images of several patients (subjects) while doing a specific task. It is a good example for applications which are both data and CPU intensive. Indeed, NPAIRS performs quite complicated operations (Eigen Value

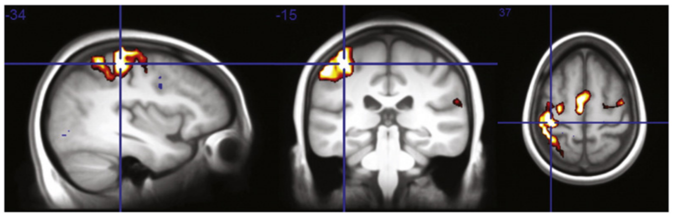

Fig. 1. An example of detected activation areas when the subjects were performing a simple reaction time task [7].

Decomposition) on a large set of input data. These operations are extremely parallelizable and they execute considerably faster on GPU. In this paper, we provide a method to schedule the tasks of our NPAIRS application on a heterogeneous CPU-GPU cluster. However, our scheduling methods are generic and could be used for other applications as well.

## II. Brain Image Processing

Functional Magnetic Resonance Imaging (fMRI) is a non-invasive neuroimaging technique commonly used to study function of human brain by measuring the blood-oxygenation-level-dependent (BOLD) signal [5], [6]. Simply put, activation of a region of the brain causes oxygenated blood to flow to that region. Oxygenated and de-oxygenated blood have different magnetic properties, which can be captured in fMRI images [5], [6]. The imaging data gathered by fMRI are analysed to detect correlations among brain activations in response to a stimulus, *e.g.* a particular motor task.

Typically, neuroscientists and physicians start by designing fMRI experiments to answer different questions they have in mind, such as trying to determine which part of the brain is responsible for a specific task. The actual experiment varies depending on the question being investigated. But, in general, it involves choosing subjects, a group of patients or healthy individuals, and choosing the type of stimuli or a task to be performed by the subjects, e.g. a finger tapping task. During the experiment, an MRI scanner collects fMRI data of subjects' brains while the subjects are given the stimulus or perform the requested task.

Generally, it is common to collect data from multiple individuals (subjects) to draw more general conclusion on the brain's function with more statistical power and less noise [6]. After performing the experiment, collected fMRI data needs to be cleansed by applying different preprocessing steps such as motion correction, spatial smoothing, detrending and whitening, and registration to template brains. Finally, statistical analysis is performed on the preprocessed fMRI data to detect correlation between regions of the brain and the task subjects performed during the experiment [6]. Output of the analysis can be presented as a color coded image of brain. For example, Figure 1 show the active region of brain when performing a simple reaction time task [7].

### A. *NPAIRS*

The NPAIRS (Nonparametric, Prediction, Activation, Influence, Reproducibility, re-Sampling) is a neuroimaging software package for analyzing fMRI data [8] [9].

Figure 2 shows the workflow of the NPAIRS application. The NPAIRS program is based on a split-half resampling framework that randomly splits the data into two halves. Then, each half of the data is analyzed individually using a statistical analysis method. Current implementation of the NPAIRS uses principal component analysis (PCA) and Canonical Variate Analysis (CVA) algorithms to do the statistical analysis. Results of the analysis on the two split datasets are used to generate *prediction accuracy (p)* and *reproducibility (r)* metrics. *Prediction accuracy* determines how accurately the values of experimental design parameters, e.g. performance measures, can be predicted in an independent test dataset. *Reproducibility* determines how reliably the parameters in the same test dataset can be reproduced [8]. This resampling loop, which contains process of splitting the data into halves, analysing each half, and computing the evaluation metrics, is repeated by default 100 times or until all the possible disjoint pairs have been tested.

In NPAIRS, in order to control model complexity and reduce data dimensionality, principal component analysis (PCA) is used. PCA determines principal components of the input dataset. Then, only first Q principal components are used to produce linear, multivariate discriminant functions for analysis of the images. The value of Q, i.e. number of selected principal components, significantly affects prediction accuracy and reproducibility of the analysis. To study the effect of number of principal components on the quality of the analysis, an exhaustive search is performed on this hyperparameter.

On each iteration of this exhaustive search, NPAIRS executes the same algorithm with a different value for Q. But, since NPAIRS is a computationally expensive application, each iteration of this search, could take hours, depending on the size of dataset and platform on which NPAIRS is running. Although NPAIRS is written to be executed on a single node, it can either run the algorithm for a specific number of Q or execute it for a set of different values of Q sequentially. Obviously, this exhaustive search is embarrassingly parallel because the evaluation of different values of Q are totally independent. Therefore, it is possible to run several instances of NPAIRS as a separate process (either on a single node or multiple nodes). Each instance will then be sequential and completely independent of all other instances. In this paper, we first improve the performance of NPAIRS on a single node with a GPU with minimal change in the source code. Then, we provide a scheduling framework for parallel execution of NPAIRS on a heterogeneous cluster.

## III. Accelerating NPAIRS

The first step towards accelerating NPAIRS is understanding its execution profile. We instrumented the NPAIRS source

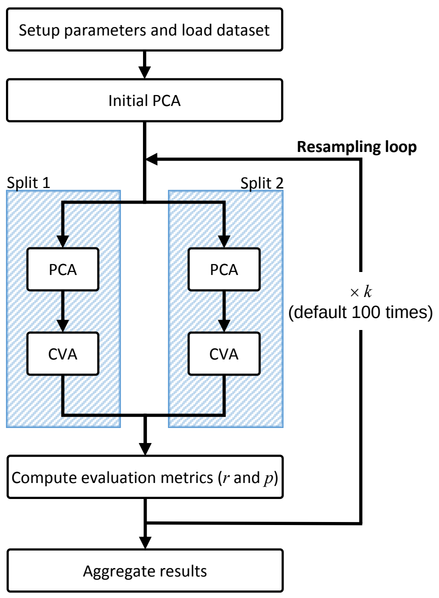

Fig. 2. Highlevel workflow of the NPAIRS application.

code[1] [10].

Based on our instrumentation, we created execution profile of NPAIRS on a dataset of 25 subjects for different number of principal components. Figure 3 illustrates proportion of different parts of NPAIRS (as depicted in Figure 2) on a machine with two Intel Xeon E5-2650 CPUs.

Execution profile of NPAIRS reveals the fact that on average, computation of the PCA algorithm in the resampling loop accounts for 70% of the total execution time. This is expected since PCA is the most complex part of the application. Also, during the execution of NPAIRS, PCA is executed in the resampling loop 200 times (100 times for each split). Therefore, we expect that accelerating PCA significantly reduces NPAIRS execution time.

In NPAIRS, the number of principal components (*#PCs*) can be defined as an input to the application. After PCA function computes eigenvalues and eigenvectors of its input matrix, the first *#PCs* eigenvectors with the highest eigenvalues are selected to be passed to the CVA step (see Figure 2). Most computationally expensive operations in PCA algorithm can be translated into basic linear algebra operations, which are intrinsically parallel. This characteristic makes PCA a suitable candidate to be executed on GPUs, which consist of a large number of weak cores that can efficiently execute small operations in parallel. Thus, to improve the performance of NPAIRS with minimal change in the source code, we move the computation of principal component analysis from CPU to GPU.

## A. Implementing PCA on GPU

NPAIRS uses the covariance method to compute principal components of prepossessed fMRI images [11]. In order to be

[1]NPAIRS is an open source software package under GNU GPL v2 License.

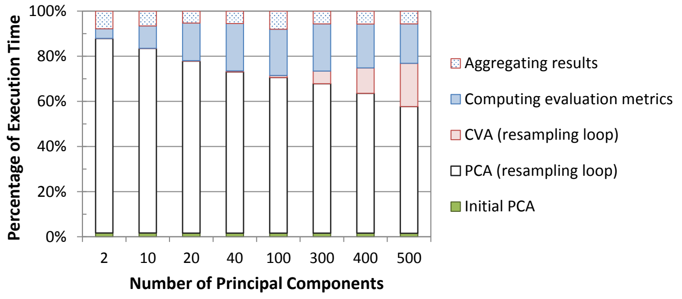

Fig. 3. Execution profile of NPAIRS for different number of principal components on a dataset of 25 subjects. The computations of PCA in the resampling loop accounts for 70% of total computation time on average.

consistent with the existing implementation, we implemented PCA algorithm on GPU using the same method. The covariance method for computing principal components works as follows.

First, the input matrix ($M$), which contains the prepossessed fMRI images, is normalized by subtracting average of each column from the elements of that column, as eq. 1.

$$\forall i,j \quad 0 \le i < m, 0 \le j < n \quad \hat{M}_{ij} = M_{ij} - \frac{1}{m}\sum_{i=0}^{m} M_{ij} \tag{1}$$

Then, sum of squares and products (SSP) is computed by multiplying the transpose of normalized matrix with itself (eq. 2).

$$SSP = \hat{M}^T \times \hat{M} \tag{2}$$

Next, eigenvalues and eigenvectors of the symmetric SSP matrix are computed. Finally, the normalized input matrix is multiplied by a matrix whose columns are eigenvectors computed in the previous step.

$$PC_{score} = \hat{M} \times EigenVectors(SSP) \tag{3}$$

The outputs of PCA which are used in NPAIRS are $PC_{score}$, eigenvalues, and eigenvectors.

There are two main framework for programming on GPUs, Compute Unified Device Architecture (CUDA) [2] and OpenCL [3]. The former has been introduced by NVIDIA, one of the pioneer manufactures of GPUs. The latter is implemented by Khronos group, an industry consortium for creating open standards for the authoring and acceleration of parallel computing, graphics, etc. Some studies show that CUDA shows better performance than OpenCL in many applications [12]. Moreover, our target GPU is NVIDIA Titan and CUDA is well suited for NVIDIA GPUs. Additionally CUDA is more mature in terms of available matrix operation libraries, e.g. CUBLAS, CUDA). Considering all these advantages we decided to use CUDA as our framework for programming on GPUs. Moreover, we used CUDA Basic Linear Algebra Subroutines (CUBLAS) library [13] for matrix computations and CULA library [14] for eigenvalue decomposition.

## *B. Invoking CUDA from Java*

Since NPAIRS is implemented in Java and we used CUDA to implement PCA on GPU, integration of these two pieces of code could be challenging. Moreover, because developers and users of NPAIRS application are biomedical researchers, it is necessary to do the integration with minimal changes to the NPAIRS original source code. Generally, there are two methods for interfacing a non-Java code with a Java application: *in-process* and *inter-process*. The former in which the interfacing is done in a single process has less performance overhead. However, the latter in which the interfacing is done in multiple processes is more portable. Java Native Interface (JNI) and sharing data for example by writing on disk are the best candidates in terms of minimal changes for in-process and inter-process communication, respectively. However, transferring data through writing on disk or ramdisk is inefficient due to its high I/O overhead. Since the goal is to improve performance of NPAIRS, in-process communication is a better choice for integrating PCA code on GPU with the NPAIRS source code. Java Native Interface (JNI) [15] is the most commonly used method for in-process communication in Java. JNI enables a Java application, running in a Java Virtual Machine (JVM), to call an external library function implemented in other languages such as C and CUDA. We used JNI to connect NPAIRS with the PCA implementation on GPU.

We created a C library from our CUDA implementation of PCA algorithm. This library contains a function named `PCA_on_GPU` to perform PCA on GPU. When an instance of NPIARS calls `PCA_on_GPU` function, first, CPU initializes the GPU. Then, our library transfers the CUDA implementation of PCA and the input matrix to GPU. Finally, after GPU compute PCA function, our library retrieves the results from GPU and passes them to the NPAIRS instance.

In order to execute PCA on GPU, we need to only add a few lines in the NPAIRS original source code to: 1) check availability of GPU 2)setup temporary variables to receive the results from GPU and store them in the corresponding variables in NPAIRS 3) call PCA function on GPU . A snapshot of the NPAIRS source code which supports execution of PCA on GPU is depicted in figure 4. In this figure, line 4 to 16 reflects the only necessary modifications in the original NPAIRS source code. Note that NPAIRS application consists of more than 100,000 lines of code. Therefore, the modification of the source code is negligible.

Transferring data between the JVM and the native library can affect performance of the application. This issues is escalated in NPAIRS because `PCA_on_GPU` function in the native library is executed multiple times (by default 200 times in the resampling loop). To alleviate the overhead of data transfer between JVM and native library, we keep the data in memory space of JVM and only send pointers of those memory locations to the native library.

```
 1 class PCA {
 2 native void PCA_on_GPU(double[] M, long nRows,
       long nColc, double[] PC_score, double[]
       evec_tmp, double[] eigenvalues);
  ...
 3 computePCA(Matrix M, boolean normalizeBySD) {
    ...
 4  if(isGPUAvailable()) {
 5    System.loadLibrary("PCA_GPU_lib");

 6    double[] pca_tmp = new double[nRows*nCols];
 7    double[] evec_tmp= new double[nCols*nCols];

 8    PCA_on_GPU(M,nRows,nCols,pca_tmp,evec_tmp,
         eigenvalues);

 9    for (int i = 0; i < nRows; i++)
10      for (int j = 0; j < nCols; j++)
11        PCscore.set(i,j, pca_tmp[j*nCols+i]);

12    for (int i = 0; i < nCols; i++)
13      for (int j = 0; j < nCols; j++)
14        eigenvectors.set(i,j,evec_tmp[j*nCols+i]);
15  }
16  else{
      ... /* Original CPU implemention of PCA */
17  }
18 }
  ...
19 }
```

Fig. 4. To enable NPAIRS to compute PCA on a GPU, we need to add just a few lines to PCA class in NPAIRS source code, which has more than 100,000 lines of code.

## *C. Experimental Results*

In order to evaluate the efficiency of our proposed GPU implementation, we executed the GPU-assisted NPAIRS on three different resources, i.e. two CPU nodes Fat (32 cores) , Light (16 cores), and one GPU node, described in Section V. In this experiment we varied number of principal components from 2 to 500 and evaluated NPAIRS on two datasets with 25 and 31 subjects. Results of this experiments is depicted in Figure 5. Obviously, the larger data set needs more time to be processed. In addition, the execution time increases as the number of principal components increases. This is due to the nature of NPAIRS application, in which the number of principal components directly affects size of input for the CVA, thus affects the execution time of NPAIRS. Also, the results show that the difference between execution time of the original NPAIRS running on CPU and the GPU assisted implementation is larger for the dataset with 31 subjects compared to the dataset with 25 subjects. This confirms the suitability of PCA to be executed on GPU. Although larger data size imposes data movement overhead for GPU, the result show that the achieved speedup mitigates it.

Execution profile of our proposed GPU implementation of NPAIRS is depicted in Figure 6. This figures shows that in our proposed the GPU assisted implementation, the PCA computation accounts for about 20% of the execution time of NPAIRS. Where as, when running NPAIRS on a CPU node, this proportion is close to 70% on average (Figure 3). It should be noted that execution of PCA as a stand-alone

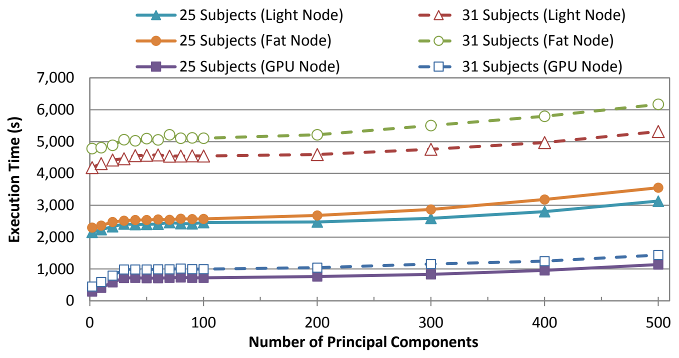

Fig. 5. Execution profile of NPAIRS for different numbers of principal components on two datasets with 25 and 31 subjects for three different nodes: a GPU node, a Light node, and Fat node.

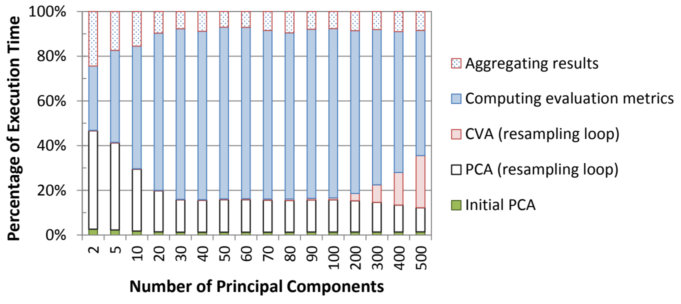

Fig. 6. Execution profile for different numbers of principal components on a dataset of 25 subjects running on a GPU.

application on a GPU can be up to 12 times faster than the CPU implementation of PCA. However, since the PCA computation accounts for about 70% of the execution time of NPAIRS, using our proposed the GPU assisted implementation can speed up NPAIRS 3 to 7 times.

## IV. Scheduling On a Heterogeneous Cluster

We show that GPU-assisted implementation of NPAIRS can speed up execution of NPAIRS 3 to 7 times. However, in a real-world research cluster, not all nodes are equipped with a GPU. But, those CPU-only nodes, though are much slower than GPU nodes, can still contribute in an exhaustive search on the number of principal components for NPAIRS. This way, instances of NPAIRS, each of which evaluates a different value for the number of principal components, can be executed on a CPU node or a GPU node in parallel. However, since the performance of NPAIRS on GPU and CPU nodes are significantly different, to execute NPAIRS on a heterogeneous cluster, we need to carefully schedule instances of NPAIRS, each evaluating a different value for the number of principal components. In this section, we study performance of well-known scheduling algorithms, which can be used regardless of the cluster management platform. We show that execution time of the exhaustive search on the number of principal components for NPAIRS on a cluster significantly varies for different scheduling algorithms.

There are several performance metrics to evaluate scheduling algorithms such as latency, throughput, and makespan. In this application, the goal is to minimize execution time of multiple instances of NPAIRS, each evaluating a different number of principal components, running in parallel on a cluster. This task is achieved when all instances of NPAIRS successfully finish their execution. It should be noted that instances of NPAIRS are independent, have no deadline, and can be executed in parallel on any resource in the cluster. Considering all these characteristics, we use *makespan* of the NPAIRS jobs, i.e. the time taken to execute all instances of NPAIRS, to evaluate performance of different scheduling algorithms.

### A. *Overview of the Scheduling Algorithms*

Although finding an optimal schedule with the minimum makespan is a NP-hard problem, there are two different approaches to find a near-optimal schedule.

The first approach is using machine learning techniques to explore the whole solution space, i.e. all possible schedules. In this approach, the exploration stops when some predefined constraints about the solution quality are satisfied or the algorithm execution time exceeds some threshold. Genetic Algorithms [16], [17] and Bee Colony Optimization [18], [19] are examples of this approach. Although this approach may result in a better solution, it is computing intensive and suffers from poor scalability.

The second option is using greedy algorithms to optimize a partial solution, iteratively aiming to find a near-optimal final solution. In this approach, scheduling algorithms have polynomial execution time and are more scalable than the machine learning based scheduling algorithms. Moreover, if implemented efficiently, its results can be competitive with the results of machine learning techniques. We implemented and evaluated the following well-known algorithms.

*1) Shortest Job First (SJF) and Longest Job First (LJF):* In SJF [20], first, submitted jobs are sorted in ascending order of their estimated execution time on GPU. Then, the shortest unscheduled job will be assigned to the fastest available node. If all nodes of the cluster are busy, the shortest unscheduled job will wait until a node will be available. LJF [20] is similar to SJF except that jobs are sorted descendingly based on their execution time and then will be scheduled with the same order.

*2) Min-Min and Max-Min:* The Min-Min algorithm [21], [22] schedules the submitted jobs iteratively. At each iteration, finish time of all unscheduled jobs on each resource, i.e. node, are computed using eq. 4.

$$f_{jr} = e_{jr} + avl_r \tag{4}$$

where $f_{jr}$ and $e_{jr}$ are the finish time and estimated execution time of the job $j$ on resource $r$, respectively, and $avl_r$ is the earliest time that resource $r$ becomes available. Then, for each job the resource with the earliest finish time is chosen as the selected resource (also known as first *min*). Finally, the job with earliest finish time will be scheduled on its selected resource (also known as second *min*). The algorithm iterates until all jobs are scheduled.

The Max-Min algorithm [21], [22] is similar to Min-Min in its first *min* step. But, it selects the job with maximum finish time in its second step.

*3) Sufferage:* Sufferage algorithm [4], [22] is an extension of Min-Min. It defines *Sufferage* of a job as the difference between the Earliest Finish Time (*EFT*) of the job and its Second Earliest Finish Time (*SEFT*). The goal of this algorithm is to minimize the Sufferage value of all submitted jobs. This is achieved by prioritizing jobs that are competing on the same resource. Similar to Min-Min, Sufferage is an iterative algorithm. But, on the contrary to Min-Min, in which at each iteration only one job is assigned to one resource, Sufferage may assign multiple jobs to their preferred resources. The Sufferage algorithm works as follows.

At the beginning of each iteration, finish time of unscheduled jobs on all resources are computed. Then, the unscheduled jobs are sorted in ascending order of their *EFT*s on all resources. Starting with an unscheduled job with the *EFT*, job *i* selects its preferred resource $r_i^*$, i.e. the resource which finishes the job earlier than any other resource in the cluster. If the preferred resource of job *i* has not been assigned to any job in the current iteration, job *i* will be scheduled on its preferred resource. But, if in the current iteration, the preferred resource of the job *i* has already been assigned to another job *j*, then job *i* should compete with job *j* for this resource. Among job *i* and job *j*, the job with greater Sufferage value will be assigned to resource $r_i^*$. The other job will be put back in the unscheduled job queue to be scheduled in the next iterations. An iteration of Sufferage algorithm ends with traversing all unscheduled jobs as described above. Detail of the Sufferage algorithm is presented in Algorithm 1.

**Algorithm 1:** Sufferage Algorithm

```
while there is an unscheduled job do
    foreach unscheduled job $j$ do
        foreach resource $r$ in cluster do
            $f_j^r \leftarrow$ finish time of job $j$ on resource $r$;
            if $f_j^r <$ EFT of job $j$ then
                Assign $j$'s EFT to its SEFT;
                Assign $f_j^r$ to EFT of job $j$;
                $r_j^* \leftarrow r$;
            else if $f_j^r <$ SEFT of job $j$ then
                Assign $f_j^r$ to SEFT of job $j$
        $sufferage_j \leftarrow$ SEFT of job $j$ - EFT of job $j$;
    Sort jobs ascendingly based on EFT;
    Mark all resources as unassigned;
    foreach unscheduled job $j$ do
        if $r_j^*$ is unassigned then
            Schedule job $j$ on its preferred resource $r_j^*$;
            Set status of $r_j^*$ to assigned;
        else if sufferage of job $j$ is greater than sufferage of
        currently scheduled job on $r_j^*$ then
            Return the job currently scheduled on $r_j^*$ to the
            unscheduled job queue;
            Schedule job $j$ on $r_j^*$;
```

## V. Evaluation of Scheduling Algorithms

To evaluate the scheduling algorithms introduced in the previous section, first we create execution profiles of NPAIRS with different values for the number of principal components. Then, we use our in-house simulator to evaluate the scheduling algorithms. Inputs for the simulator are as follows: i) Execution profiles of NPAIRS application with different values for the number of principal components on all available resources. ii) A list of available resources in the cluster. iii) A list of submitted jobs to the cluster with their input parameters, the number of principal components for NPAIRS.

The execution profiles for the NPAIRS jobs are obtained by running individual execution of each job on each node of our in-house heterogeneous cluster which consists of three types of resources:

- **Fat node:** Four Intel Xeon E5-4620 CPUs with 512GB of memory.
- **Light node:** Two Intel Xeon E5-2650 CPUs with 32GB of memory.
- **GPU node:** One Intel Core i7-3770K CPU with 16GB of memory, equipped with a Nvidia GeForce GTX TITAN GPU.

In the following experiments, unless otherwise mentioned, we use a cluster of 3 Fat nodes, 3 Light nodes, and 2 GPU nodes. We use the same fMRI dataset of 25 subjects for all experiments. Also, for all experiments, submitted workload to the cluster is a batch of 99 independent NPAIRS jobs each with a unique value for the number of principal components ranging from 2 to 100. We assume all NPAIRS job are part of an exhaustive search process on the number of principal components, as described in section II, and are submitted to the cluster at the same time. All the scheduling algorithms that we evaluate, schedule the batch of NPAIRS jobs on their submission. We use makespan of the submitted batch of NPAIRS jobs to evaluate efficiency of different clusters and scheduling algorithms.

As a baseline for the scheduling algorithms, we evaluate a basic First Come First Serve (FCFS) scheduling algorithm. Since makespan of a schedule produced by FCFS depends on the arrival order of the jobs and we assume that all jobs are submitted in a batch at the same time, we repeat FCFS algorithm 200 times on the same batch of jobs, each time with a different order. and report average, the best, and the worst makespan.

In addition, we evaluate two basic scheduling algorithms, Torque [23] and Minimum Compilation Time (MCT) [24]. Torque schedules all jobs on their fastest resource(s) in the cluster. MCT algorithm [24], follows a greedy strategy and assigns each job to a node that can finish the job sooner,

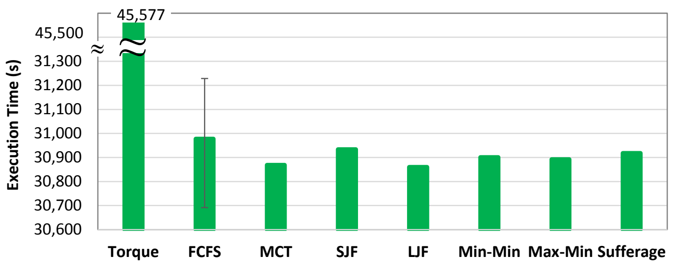

Fig. 7. Execution time of the batch of NPAIRS jobs with different scheduling algorithms on a CPU cluster of 3 Fat nodes and 5 Light nodes.

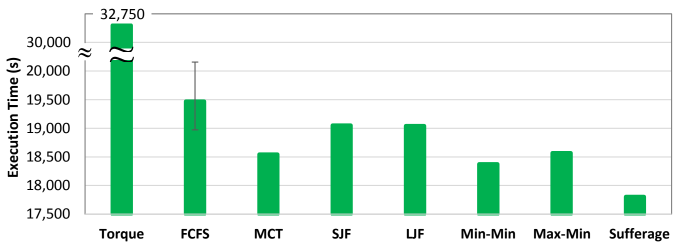

Fig. 8. Execution time of the batch of NPAIRS jobs with different scheduling algorithms on a heterogeneous cluster of 3 Fat nodes, 3 Light nodes, and 2 GPU nodes .

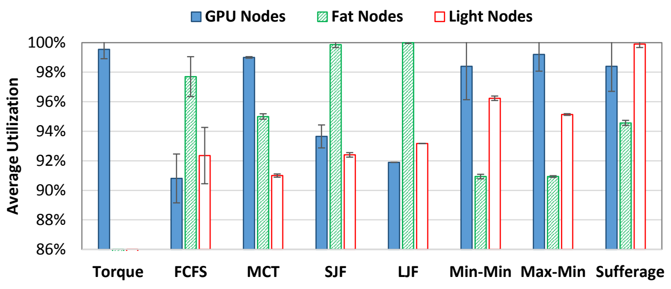

Fig. 9. Resource utilization of different scheduling algorithms on a heterogeneous cluster of 3 Fat nodes, 3 Light nodes, and 2 GPU nodes.

considering jobs that have been already scheduled on the nodes.

In order to ensure a fair comparison between our CPU-GPU framework for NPAIRS and its original CPU implementation, we evaluate different scheduling algorithms on a CPU cluster with 5 Light nodes and 3 Fat nodes. We use the best results obtained from this cluster to compare with the results of heterogeneous CPU-GPU cluster. Figure 7 depicts the makespan of the NPAIRS batch for all tested scheduling algorithms on the CPU cluster. The results indicate that in a CPU cluster, where execution of NPAIRS tasks do not differ significantly on different nodes, the difference between makespan of tested scheduling algorithms is less than 0.25%. The only exception is the Torque scheduling algorithm [23], in which all jobs are scheduled only on their fastest resources, which are the Light nodes in this cluster (Figure 5).

As we demonstrated in Section III, NPAIRS application can get a significant performance boost by utilizing a GPU. To show the effect of a few GPU nodes on the performance of the NPAIRS workload, we build a heterogeneous cluster by replacing two Light nodes in the above-mentioned CPU cluster with two GPU nodes. In the heterogeneous cluster, since the execution time of NPAIRS on different nodes significantly varies, on the contrary to the homogeneous CPU cluster, makespan of a batch of NPAIRS job depends on the scheduling algorithm.

The makespan of the batch of NPAIRS jobs in the heterogeneous cluster with different scheduling algorithms is illustrated in Figure 8. The results show that all algorithms, except Torque, perform better than the average of FCFS algorithm. This confirms hypothesis about the importance of having a scheduling algorithm for a homogeneous cluster.

Torque schedules jobs on the fastest resource in the cluster, the two GPU nodes in this cluster, and does not utilize other nodes. Therefore, the makespan of Torque is higher than the other scheduling algorithms.

SJF and LJF have slightly higher makespan (less than 0.5%) compared to the best FCFS case. Min-Min and Max-min, which can be considered more intelligent versions of SJF and LJF, outperform them by 3.5% and 2.5%, respectively. The difference between performance of Min-Min and Max-Min is due to their different approaches for incrementally finding the final scheduling solution. Min-Min, at each iteration, tries to keep the load of the resource balanced by assigning a job to a recourse which leads to the minimum possible increase of current overall finish time. This strategy postpones allocation of longer jobs and fails to create a schedule of good quality when few number of very long jobs remain to be scheduled at the end of algorithm. In this case, all resources, except those which are running the long remaining jobs, have approximately same finish time and remain idle while few resources are busy executing the long jobs. On the other hand, Max-Min gives higher priority to longer jobs. It fails when a long job that is scheduled on a powerful resource steals the opportunity from many shorter jobs. The NPAIRS application does not contain any job with extremely longer execution time. For this reason, Min-Min outperforms Max-Min by 1.5%.

Sufferage is an extension on Min-Min algorithm and reduces makespan of Min-Min by 3%. This is because all the NPAIRS jobs have the same preferred resources, GPU nodes, and Sufferage algorithm assigns GPU nodes to jobs that get the most advantages of running on them, which are the jobs that suffer the most from running on other nodes. Overall, Sufferage results in the minimum makespan among all evaluated scheduling algorithms. Compared to Torque, FCFS, and MCT algorithms, Sufferage reduces makespan of the NPAIRS jobs by 47%, 9%, and 4%, respectively. This is a considerable improvement which has been achieved with a very low scheduling overhead.

To have a more complete analysis of the evaluated scheduling algorithms, resource utilization for each types of nodes in the heterogeneous cluster is presented in Figure 9. As expected, Sufferage has the most balanced utilization among all

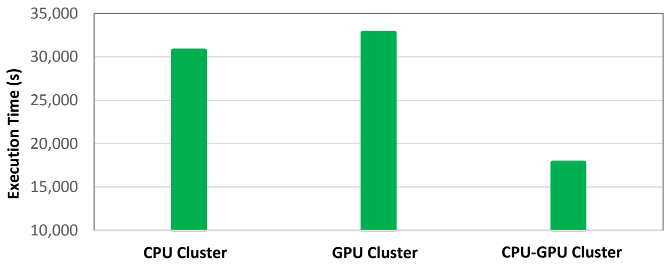

Fig. 10. Execution time of the batch of NPAIRS jobs on three different clusters: a CPU cluster of 3 Fat nodes and 5 Light nodes, a GPU cluster of 2 GPU nodes, and a heterogeneous cluster of 3 Fat nodes, 3 Light nodes, and 2 GPU nodes.

algorithms. The least utilized resources when using Sufferage algorithm are the Fat nodes. The low utilization of Fat nodes in Sufferage is a key factor in the success of Sufferage algorithm because efficiency of Fat nodes on executing NPAIRS jobs is less than the GPU nodes and Light nodes (see Figure 5).

On the contrary to Sufferage, FCFS, SFJ, and LJF result in the most utilization of FAT nodes, which explains their high makespan (Figure 8). Although Min-Min and Max-Min try to increase utilization of the fastest nodes, i.e. GPU nodes, and decrease the slowest nodes, i.e. Fat nodes, they are not as successful as Sufferage in utilizing Light nodes more than Fat nodes.

To show that in the presence of powerful GPU nodes, we can still benefit from available CPU nodes, we compared the performance of a CPU-only cluster of 8 nodes, a GPU-only cluster of 2 nodes, and a heterogeneous cluster of 6 CPU nodes and 2 GPU nodes (Figure 10). The shortest makespan for the CPU cluster belongs to one single execution of FCFS algorithm, which is less than 1% shorter than the makespan produced by Sufferage algorithm. The shortest makespan for the heterogeneous cluster is produced by the sufferage algorithm. Also, to show that in presence of powerful GPU nodes, we can still benefit from available CPU nodes, we provide make span of a cluster of 2 GPU nodes (Figure 10). The results support our approach, which is utilizing a few powerful GPU nodes with already available CPU nodes in a heterogeneous cluster.

## VI. Related work

Since the advent of GPGPU, several researches have been performed to efficiently use the computational power of GPUs for scientific applications. Using GPU's computational power in medical image processing has been widely investigated [25]–[27] such as BROCCOLI [25] which is an OpenCL implementation of an fMRI analysis software package. However, the method we presented in this paper benefits from GPUs while applying minimal changes to the existing CPU code of the fMRI application.

GPUs have also been widely used in a wide range of other scientific applications [28]–[30]. Authors in [28] have implemented a numerical weather prediction algorithm on a GPU and integrated it into a weather forecasting application. They achieved 7x speedup for the GPU version of the algorithm, Whereas they got only 2x speedup after integrating it into the whole application. However, we got 14x speedup for the PCA and 7x speedup when integrating it into NPAIRS. This implies that our implementation is as efficient that even with the presence of data transfer cost, we still have a reasonable speedup.

In [31], a chemistry application has been scheduled on a HPC platform. However, the benefit of using GPU is not investigated in this work since porting to GPU requires a fundamental upgrade to that application. On the other hand, our suggested method applies minimal changes to NPAIRS with the benefit of achieving almost 7x speedup.

There are multiple works on studying job scheduling for heterogeneous clusters. StarPU [32] is a scheduling framework for heterogeneous multi-core architectures. Several basic strategies has been implemented in StarPU. However, its main target is to provide the load balancing among all available resources. Authors in [33], have extended Hadoop to perform the task scheduling for GPU-based heterogeneous clusters. Their main goal is to minimize the execution time. However, Hadoop framework is not appropriate for small jobs with execution times of less than a minute. Therefore, in cases where jobs are being executed too fast on a GPU, it is not reasonable to use Hadoop due to its significant overhead. Ravi *et al.* [24] schedule a set of well-known applications on a cluster of CPU-GPU nodes. They have developed a set of simple scheduling schemes. They test their method both for single-node and multi-node applications. Likewise, the whole scheduling schema is for independent jobs. Torque [23] is a resource manager which is being widely used to manage heterogeneous clusters. The scheduling strategy of Torque is based on OpenPBS and it is fairly simple. The idea is that the user will specify the job that needs to be executed. Once a resource is selected for a job, Torque does not consider the possibility of running that job on another resource type. Considering the fact that user will always ask for the fastest resource to execute the job, this schema will impose a high load imbalance to the system.

## VII. Conclusion

The computational power of GPU has recently been used to accelerate scientific applications. Neuroimaging applications mostly consist of complex linear algebra operations which are intrinsically parallel. Thus, they are one of the best candidates to be accelerated by GPUs. In this paper, we efficiently ported NPAIRS, a neuroimaging application, to GPU. This is done by adding only a few lines of code to the original NPAIRS code. This minimal change of the code is so important from the view point of NPAIRS' non-expert users and developers, i.e. bio-medical scientists. Because they do not have to suffer from any fundamental change in the application.

As the second part of our research, we investigated the efficiency of different scheduling algorithms for running NPAIRS on a heterogeneous cluster. Experimental results show that when running NPAIRS original code on a homogeneous cluster of CPU nodes, the scheduler does not have a considerable impact on overall execution time. However, by replacing a quarter of CPU nodes with GPU nodes and utilizing the Sufferage scheduling algorithm, we improved the application performance by 44%. We also compared our results with Torque's basic scheduler and MCT, two of the commonly used schedulers in current HPC platforms. The Sufferage algorithm improves Torque and MCT by 47% and 4% respectively.

Our results show that by applying minimal changes to the original code and adding a few GPUs, each costs only 25% of a CPU node, significant performance improvement is achieved.

## References

[1] Nvidia corporation. [Online]. Available: http://www.nvidia.com/

[2] (2007) Compute unified device architecture programming guide. [Online]. Available: docs.nvidia.com/cuda

[3] Khronos group. [Online]. Available: https://www.khronos.org/opencl/

[4] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–131, 1999.

[5] N. K. Logothetis, "What we can do and what we cannot do with fMRI," *Nature*, vol. 453, no. 7197, pp. 869–878, 2008.

[6] A. Eklund, "Computational medical image analysis: With a focus on real-time fMRI and non-parametric statistics," 2012.

[7] A. H. Andersen, D. M. Gash, and M. J. Avison, "Principal component analysis of the dynamic response measured by fMRI: a generalized linear systems framework," *Magnetic Resonance Imaging*, vol. 17, no. 6, pp. 795–815, 1999.

[8] S. C. Strother, J. Anderson, L. K. Hansen, U. Kjems, R. Kustra, J. Sidtis, S. Frutiger, S. Muley, S. LaConte, and D. Rottenberg, "The quantitative evaluation of functional neuroimaging experiments: The NPAIRS data analysis framework," *NeuroImage*, vol. 15, no. 4, pp. 747–771, 2002.

[9] S. Strother, A. Oder, R. Spring, and C. Grady, "The NPAIRS computational statistics framework for data analysis in neuroimaging," in *Proceedings of COMPSTAT'2010*, 2010, pp. 111–120.

[10] The PLS (partial least squares) and NPAIRS (nonparametric, prediction, activation, influence, reproducibility, re-sampling) neuroimaging software package. [Online]. Available: https://code.google.com/p/plsnpairs/

[11] S. Strother, S. L. Conte, L. K. Hansen, J. Anderson, J. Zhang, S. Pulapura, and D. Rottenberg, "Optimizing the fMRI data-processing pipeline using prediction and reproducibility performance metrics: I. a preliminary group analysis," *NeuroImage*, vol. 23, Supplement 1, no. 0, pp. S196–S207, 2004, mathematics in Brain Imaging.

[12] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *ArXiv e-prints*, 2010.

[13] (2008) cuBLAS library. [Online]. Available: https://developer.nvidia.com/cublas

[14] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *SPIE Defense, Security, and Sensing*, 2010.

[15] Java Native Interface. Accessed: 2014-03-01. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[16] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8–22, 1997.

[17] S. Song, K. Hwang, and Y.-K. Kwok, "Risk-resilient heuristics and genetic algorithms for security-assured grid job scheduling," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 703–719, 2006.

[18] M. Arsuaga-Rios, M. Vega-Rodriguez, and F. Prieto-Castrillo, "Multi-objective artificial bee colony for scheduling in grid environments," in *Swarm Intelligence (SIS), 2011 IEEE Symposium on*, 2011, pp. 1–7.

[19] T. Davidovic, M. Selmic, and D. Teodorovic, "Scheduling independent tasks: Bee colony optimization approach," in *Control and Automation, 2009. MED '09. 17th Mediterranean Conference on*, 2009, pp. 1020–1025.

[20] A. Streit, "On job scheduling for hpc-clusters and the dynp scheduler," in *High Performance ComputingHiPC 2001*, 2001, pp. 58–67.

[21] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[22] T. D. Braun, H. J. Siegel, N. Beck, L. L. Blni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[23] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 8.

[24] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling concurrent applications on a cluster of CPU-GPU nodes," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2262–2271, 2013.

[25] A. Eklund, P. Dufort, M. Villani, and S. LaConte, "BROCCOLI: Software for fast fMRI analysis on many-core CPUs and GPUs," *Frontiers in Neuroinformatics*, vol. 8, p. 24, 1900.

[26] L. Shi, W. Liu, H. Zhang, Y. Xie, and D. Wang, "A survey of GPU-based medical image computing techniques," *Quantitative imaging in medicine and surgery*, vol. 2, no. 3, pp. 188–206, 2012.

[27] A. R. F. da Silva, "Cudabayesreg: parallel implementation of a bayesian multilevel model for fMRI data analysis," *Journal of Statistical Software*, vol. 44, no. 4, pp. 1–24, 2011.

[28] W. Vanderbauwhede and T. Takemi, "An investigation into the feasibility and benefits of GPU/multicore acceleration of the weather research and forecasting model," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, 2013, pp. 482–489.

[29] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "Fast outlier detection using a GPU," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, 2013, pp. 143–150.

[30] W. Liu, B. Schmidt, and W. Muller-Wittig, "CUDA-BLASTP: accelerating BLASTP on CUDA-enabled graphics hardware," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 8, no. 6, pp. 1678–1684, 2011.

[31] R. Warrender, J. Tindle, and D. Nelson, "Job scheduling in a high performance computing environment," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, 2013, pp. 592–598.

[32] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[33] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for GPU-based heterogeneous clusters," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, pp. 733–740.