Parallel Implementation of Orthogonal Matching Pursuit in OpenCL

Amirhossein Jofreh

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Master of Science
in
Electrical and Electronic Engineering

Eastern Mediterranean University
August 2013
Gazimagusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

____
Prof. Dr. Elvan Yılmaz
Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Electrical and Electronics

____
Prof. Dr. Aykut Hocanin
Chair, Department of Electrical and Electronics

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Electrical and Electronics.

____
Prof. Dr. Runyi yu
Supervisor

Examining Committee

1. Prof. Dr. Huseyin Ozkarmanli	______________________
2. Prof. Dr. Runyi Yu	______________________
3. Assoc. Prof. Dr. Hasan Demirel	______________________

ABSTRACT

Orthogonal matching pursuit (OMP) is one of the most effective techniques to recover a sparse signal from limited number of measurements. However, when the number of measurements necessary is very large recovering the sparse signal would a challenge for CPU.

In this thesis we aim to improve the performance of large array reconstruction by using parallel computing technology. We use Open Computing Language (OpenCL) in implementing parallel OMP in CPU and GPU. We also make some modification in pseudoinverse algorithm (i.e. using QR decomposition instead of naive matrix inverse) to improve the robustness of the implementation.

To examine the performance and quality of implementation, we consider signals of four different sizes (i.e. small, medium, large and massive) and evaluate the results. We can obtain better performance (over 2 times faster) for signals of large and massive sizes in terms of the speed and accuracy of the reconstruction.

Thanks to portability of OpenCL, the proposed implementation can be run on all kind of devices such as embedded devices, smart phones, and laptops.

Keywords: Compressive Sensing, Orthogonal Matching Pursuit, OpenCL, Graphic Processing Unit, Central Processing Unit

ÖZ

Dik Eşleştirme Takib tekniği, sınırlı sayıda ölçümlerden bir seyrek sinyal kurtarmak için en cazip tekniklerinden biridir. Ancak, bu sınırlı sayıda ölçümlerin pek çok olduğu zaman, orijinal sinyal kurtarma işi CPU için çok zor olacaktır. Bu tezde önerilen yöntem, CPU tarafından kurtarılması zor olan büyük sayıda olan ölçümler için iyidir.

Bu tezde, Heterojen bilgisayar teknolojisini kullanarak, büyük miktarda olan ölçümlerin hızlıca hesaplanması için yeni bir yöntem öneriyoruz. Bu son teknolojinin gücünü kullanmak için, bize ölçümleri işlemekte tüm kaynakları kullanmak için OpenCL yi kullanıyoruz.

Deneylere göre, işlem hızında hemen hemen üç kat iyileştirme vardır. Ayrıca bu hesaplama deneyi bize küçük bir hata ile çok net bir sonuç verebilir olduğunu gösteriyoruz. Eğer OpenCL yeni atom fonksiyonunu kullanırsak, kata yakın daha hıza ulaşmamız mümkün olacaktır. Ayrıca, en yüksek performans elde etmek için daha hızlı bir donanım kullanmak da mümkündür.

Önerdiğimiz yöntem ile, gömülü cihazlar, akıllı telefonlar ve dizüstü bilgisayarlar gibi her türlü cihazları çalıştırmak için OpenCLyin taşınabilirliğinden yalarlanabiliriz.

Anahtar Kelimeler: Ortogonal Eşleştirme Takip, OpenCL, Grafik İşleme Birimi, Merkezi İşlem Birimi

To My Beloved Family

ACKNOWLEDGMENTS

First of all, I would like to thank Professor Dr. Runyi Yu for helping me in my study of signal processing and also for being my supervisor for the thesis. Without his advice this thesis would have ended as a collection of incoherent work.

The help from other professors in the department during this project are very much appreciated.

I would also like to thank others who help me to achieve the result of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Interface
CPU	Central Processing Unit
CS	Compressed Sensing
GPU	Graphic Processing Unit
LSE	Least Square Error
OMP	Orthogonal Matching Pursuit
OpenCL	Open Computing Language
QR	Right Hand Side and Orthogonal Matrix
RIP	Restricted Isometry Property

Chapter 1

INTRODUCTION

This chapter provides the motivation and aim of our study. It also gives an outline of this thesis.

1.1 Motivation and Thesis Object

Around 2004 Candes, Tao and Donoho found important results to reconstruct the image from deemed insufficient amount of data [1][2].

The phrase of compressed sensing comes from the problem of realization of sparse signal x using a few linear measurements that possess incoherent properties. This technique usually uses tremendous resources to acquire large signals, so it takes a long time to process. The main question then is whether or not we can reduce the time of this process. We have two choices for this purpose.

The first choice to achieve better processing time is to change the method of calculation of reconstructed signal (i.e. orthogonal matching pursuit, matching pursuit,...). This way has its own downside like complexity of codes and formulas. Also, for a large amount of data this would not guarantee that computational device gives us better performance.

The second method is to change the way of processing. It means that instead of using traditional sequential processing, one can use heterogeneous processing (parallel processing) to process the large amount of data. It would be helpful for us to calculate a huge amount of data in a parallel way and also gives us a great performance.

The objective of this research is to improve the performance of large array reconstruction by using the mentioned methods introduced i.e., orthogonal matching pursuit (OMP) in parallel processing. In this study Computing Language (OpenCL) has been used for implementing these methods. They have been modeled over the computational resource of the computer. It is expected to have an improvement in calculation performance of large array signal, especially in term of computation time.

1.3 Thesis Overview

This thesis is consisting of our main chapters: chapter two introducing reconstruction-method and its materials, analyses the existent algorithms and compares the obtained results of implementations. It begins with some basic concepts of compressed sensing(l_0-norm, measurements, restricted isometry property and mutual coherence), it then presents the reconstruction method (OMP) with an example; OpenCL software, and AMD GPU architecture are introduced. Chapter three contains anaïve implementation of OMP algorithm by using QR decomposition instead of computing pseudo-inverse to solving least square problem. This implementation helps improve stability of solving least square problem and reduces the computation time. Moreover, the speed improvement in solving least square problem, speedup of in matrix product for argmax; result in a fast reconstruction process. The ViennaCL library is used in implementing this algorithm on the high performance processors.

Chapter four presents the implementation results for four different sizes of signals (small, medium, large and massive). Based on the experiment results of this thesis, In the case of small and medium size there is no significant difference between CPU and OpenCL implementation. But, for the massive and the large array the improvement two times in speed is observed. Also, results in terms of complexity of calculation and improvement in the algorithm are discussed. At the end conclusion and future works are given.

Chapter 2

BACKGROUND

2.1 Compressed Sensing

Compressive (or compressed) sensing is a framework for signal reconstruction from a measurement vector which is assumed to be smaller in size than the Nyquist-sampled signal vector and that this signal vector is inherently sparse (most of the signal vector components are zero). The measurement acquisition process is described by a matrix multiplication with a fat sensing-matrix and thus the reconstruction problem is an under-determined system of linear equations. The challenge in compressive sensing lies in two main things:

Firstly, how to produce measurement vector in practice and secondly, based on known the measurement vector and measurement matrix, how to find the correct underlying sparse signal vector. The theory behind compressive sensing is based on the observation that many natural signals, such as sound or images can be well approximated with sparse representation in some domain. For example, it turns out that most of the energy in a typical image signal is preserved within the 2% to 4% dominating wavelets [1][2][3].

This chapter is divided into three important parts. In part one: we introduce the compressive sensing problem and formulas. In part two: we will talk about the method of recovery of signal. In part three we introduce OpenCL structure and the relevant information.

To introduce compressed sensing problem first we need to define "l_0-norm"

Definition 1: "l_0-norm" [1]

$$l_0\text{-norm } (\|x\|_0 \triangleq \text{Number of nonzero components of signal x}) \tag{1.1}$$

l_0-norm is not a true norm because it does not have absolute homogeneity ($\forall \alpha \neq 0 \; and \; \forall x \neq 0, \|\alpha x\| = |\alpha| \|x\|$) property. We abuse l_0–norm name to say the vector x is sparse: when the size of x is much larger than itsl_0-norm. Indeed we say x is k-sparse if $\|x\|_0$= k<<N, where N is the size of vector x.

Definition 2: "Measurements" [1]

To find the sparse signal x, we can use measurement y, the measurement-matrix A and under-determined set of equations as[1]

$$y = Ax + e \qquad A \in \mathbb{R}^{MxN}, x \in \mathbb{R}^{N}, y \in \mathbb{R}^{M} \tag{1.2}$$

The Gaussian noise (e) is a member of $\mathbb{R}^M$ which represents some measurement noise. Note that, sparsity level (k) in x is smaller than M (k<M<N).

In this thesis, we will construct A by picking individual elements independently. This is not a common assumption, but it will simplify the notation. Performance of reconstruction of signal x depends on A and e.

To find how good a matrix A is in compressive sensing measurement (to construct and reconstruct), we introduce two fundamental properties of matrix-measurement A, namely, mutual coherence and restricted isometry property (RIP).

Definition 3: "Restricted Isometry Property" [10]

Matrix A fulfills the restricted isometry property with δ_k if

$$(1-\delta_k)\|x\|_2^2 \leq \|Ax\|_2^2 \leq (1+\delta_k)\|x\|_2^2 \tag{1.3}$$

Holds for any k-sparse signal x, where $\delta_k > 0$ is the smallest value to satisfy the inequality mentioned (1.3). Matrix A is the transformation matrix between two spaces of measurements and signal, where the size of signal is much larger than that of the measurements [1]. It characterizes the change of Euclidian norm of x by transformation of A, or if we consider two elements from x RIP can be interpreted as distance change between those two elements by the transformation of A.

Consider two signals x1 and x2, which are transformed noiselessly by transformation matrix to two measurement y1 and y2 as shown in Figure 2.1. By finding the distance of two elements in the metric space, we can find this relation

$$\frac{\|y_1 - y_2\|_2^2}{\|x_1 - x_2\|_2^2} = \frac{\|Ax\|_2^2}{\|x\|_2^2} \tag{1.4}$$

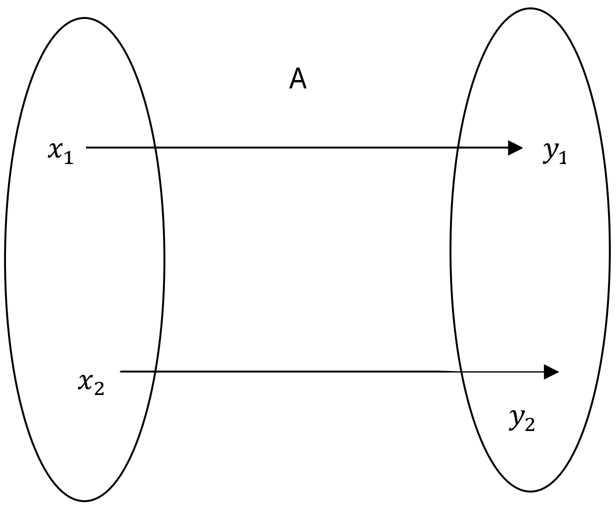


Figure 2.1: Transformation between Two Spaces

Based on (1.3) and (1.4), we determineδ_k. This is an upper- and lower bound of change in Euclidian distance of A.

RIP is used in theory to characterize the recovery performance of compressive sensing, but in practice finding a RIP is challenging. It happens because of the difficulty of findingδ_k. In practice instead of RIP, we can use mutual coherence.

Definition 4: "Mutual Coherence" [3]

Let a_i and a_j be columns of transformation A. Then, we can define mutual coherence by of A by

$$\mu(A) \triangleq \sup\{ \left|\langle a_i, a_j\rangle\right| : \forall i, j, \text{where } i \neq j\} \tag{1.5}$$

2.1.2 Data Reconstruction

In data reconstruction, the problem is to find vector x based on matrix A and measurement vector y [12]. There is no obvious formula to find the answer to this problem. A natural attempt is to solve least squares problem

$$\min_{\hat{x}} \|A\hat{x} - y\|_2^2 \tag{1.6}$$

However, this problem has infinitely many solutions (because of matrix A is of column-rank deficient). Instead, the knowledge of sparsity should be used to find the answer. We can solve the problem based on this constraint

$$\min_{\hat{x}} \|\hat{x}\|_0 \text{ such that } A\hat{x}=\text{y} \tag{1.7}$$

2.1.2 Approaches

There is a different approach to solve this problem l_1-minimization approach and Greedy pursuit.

l_1-minimization

The l_1-minimization approach in most cases based on RIP can recover exact k-sparse vector x. But this approach does not have linear bound (RIP problem) on the runtime, Moreover, the speed is usually not optimal.

Greedy pursuit

Another approach to reconstruct vector x is greedy pursuit. “Greedy pursuit means iterative signal recovery algorithm to calculate the support of signal, and it makes the locally optimal choice at each time to build up an approximation and repeats until the criterion fulfilled" [3]. Orthogonal matching pursuit (OMP) is one of the greedy algorithms for recovering signal. Before introducing of OMP method, we recall the solution to the least square problem in OMP.

Least square estimation

Least square estimation is one of the main tools for many greedy pursuit algorithms. The full compressive sensing problem could not be solved with LSE because that problem represents an under-determined set of linear equations from which it is not clear which solution to choose. However, suppose one accurately detects the true support-set s of x, then it is in the noiseless case straight-forward to see that Ax=$A_s x_s$. Here, y=$A_s x_s$ represents an over-determined set of linear equation, to which least square can be used to find a unique solution. By limiting ourselves to findx_s, we can then reconstruct the full x by padding zeros in the remaining positions.

A least square estimation $\hat{x}_s$ of x is given by the following problem

$$\min_{\hat{x}_s} \|y - A_s \hat{x}_s\|_2^2 \tag{1.8}$$

In this thesis we are only interested in case where A_s has full column rank, in which case the result can be obtained from

$$\hat{x}_s = (A_s)^{+} y = ({A_s}^T A_s)^{-1} {A_s}^T \mathrm{y} \tag{1.9}$$

In practice we don't have access to the true support-set s of x but an estimate$\hat{s}$ instead. Note that by further multiplying the result with A_s gives the orthogonal projection of y, denote by y_p, onto space spanned by columns of A_s.

2.2 Orthogonal Matching Pursuit (OMP)

Orthogonal matching pursuit is a greedy algorithm for recovering signal. Mallat and Zhang[4] proposed this algorithm and Gillbert and Tropp [5] analyzed it. Assume signal vector x is a k-sparse signal, and A is measurement matrix by columns $a_1, a_2, \dots, a_N$.And we have M-dimensional measurement vector y ($y = Ax$). Signal x has only k non-zero component so y can be defined as a linear combination of k columns from A. The most critical part to recover a signal is to find a location of these nonzero components of x. It is critical to determine which column in matrix-measurement A participates in vector y [2]. OMP is a greedy algorithm that picks the columns from matrix A by finding maximum correlation between the columns and the residual of y. In every iteration, for the support of signal x one coordinate would be calculated. When iterations reached the sparsity level(i.e.k), the entire support of signal can be identified.

The OMP algorithm has four steps in each iteration:

(1) Choose the index β_i by finding the largest correlation between $\{a_j\}_1^N$and residual of y.

(2)Unite the chosenβ_i with the index set $S_i = [S_{i-1}\ \beta_i]$, and $a_{\beta_i}$with matrix $A_i = [A_{i-1} a_i]$($a_0$is an empty set).

(3) Use the LSE (more detail after the algorithm) to find the projection of y on to the range of matrix-measurement column a_i. Thus the residual of y is always orthogonal to A_i.

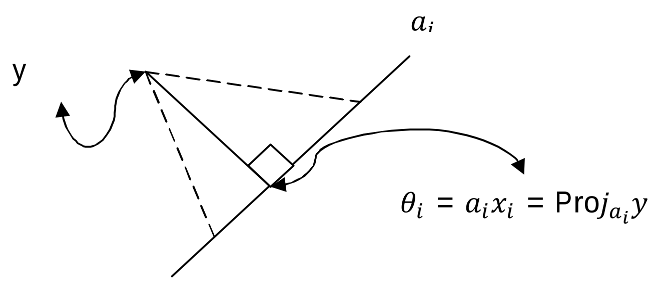


Figure 2.2: Least Square Method

(4) Calculate the new residual of r_i and do this process while we reach to k.

Once we found S (support of signal x), then we can calculate the approximation of signal $\hat{x}$ by $\hat{x} = (A_s)^{+}y$.Table 2.1 gives the algorithm of orthogonal matching pursuit.

Table 2.1Algorithm of OMP

Input: Measurement-matrix A, Measurement y, Sparsity level k of signal vector x

Output: Index set S, Measurement estimate θ_i, residual r_i (i = 0,1,…,k)

```
$r_0 = y$, $S_0 = \emptyset$, $i = 0$
While $i \leq k$ do
    1. $i = i + 1$
    2. $\beta_i = \text{argmax}_{\{j=1,\dots N\}} |\langle r_{i-1}, a_j \rangle|$
    3. $S_i = S_{i-1} \cup \{\beta_i\}$
    4. $A_i = [A_{i-1} a_{\beta_i}]$
5. $x_i = \text{argmin}_x \|A_i x - y\|_2$
    6. $\theta_i = A_i x_i$, $r_i = y - \theta_i$
End while
```

For better understanding the principle of OMP, we provide a simple noiseless example [12]. Assume the following data is given:

$$A=\begin{pmatrix}0.4033 & 0.3257 & -0.0198\\ 0.9150 & 0.9455 & -0.9998\end{pmatrix}, \quad y=\begin{pmatrix}0.9307\\ -0.4271\end{pmatrix}.$$

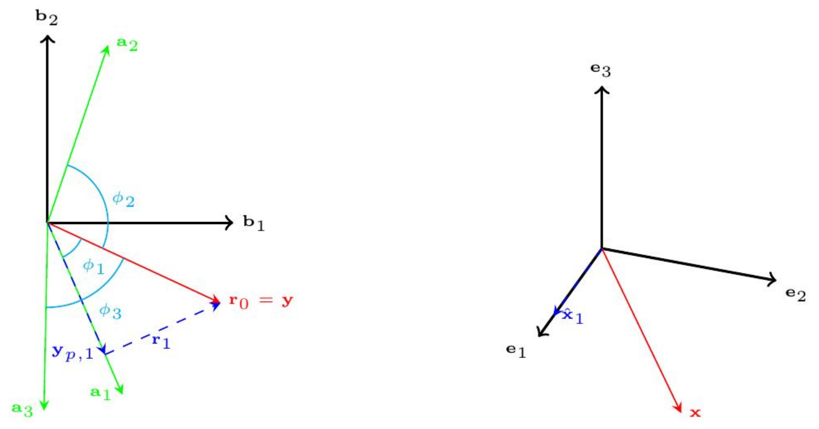
The corresponding signal- and measurement space are shown in Figure 2.3, where in Figure 2.3b, the sought signal x (in gray) is shown as a reference. In Figure 2.3a, the column vectors a_1,a_2 and a_3 from A, the measurement vector y in red is given.

Starting OMP, the initialization phase of the algorithm is executed:

i=0, r_0=y and S_0=∅ **.** It then proceeds to the first iteration:

Step 2, i=1 and in step 3 the residual vector r_0 is correlated with every column-vector in A:

$$A^T r_0 = A^T y = \begin{pmatrix}0.4033 & 0.3257 & -0.0198\\ 0.9150 & 0.9455 & -0.9998\end{pmatrix}^T \begin{pmatrix}0.9307\\ -0.4271\end{pmatrix} = \begin{pmatrix}0.7662\\ -0.1007\\ 0.4086\end{pmatrix}.$$



(a) Measurement Space $\mathbb{R}^2$ (b) Signal Space $\mathbb{R}^3$

Figure 2.3: The Signal and Measurement Space during First Iteration

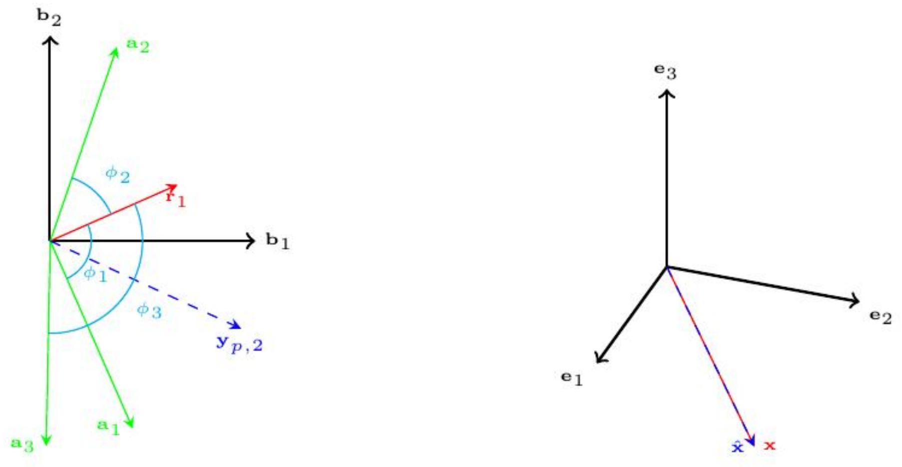
Consequently, the index corresponding to the maximum in amplitude value is chosen by argmax (…) and found to be r=1. We can verify the result by studying Figure 2.3a. where we see that the index corresponding to the vector $a_1$gives the smallest angle φ_1.

Step 4 in algorithm one, the support set become $S_1 = S_0 \cup \{\beta_1\}=\emptyset \cup \{1\}=\{1\}$

In the final step we find new residual-vector by finding LSE: $x_1 = \text{argmin}_x \|a_1 x - y\|_2$, $\theta_1 = a_1 x_1$, $r_1 = y - \theta_1$where $\theta_1=[0.3090\,, -0.7011]^T$ andnew residual is $[0.6217\,, 0.2470]^{T}$which is shown in Figure 2.3a

The first iteration now we can show how$\hat{x}$would look like in signal space if the algorithm stopped here. One point with $\hat{x}_1$ at this stage is that we can verify that OMP found the dominating base vector e_1 of x. We now proceed to the second iteration in figure 2.4.

Step 2 of this iteration i =2.



(a) Measurement Space $\mathbb{R}^2$ (b) Signal Space $\mathbb{R}^3$

Figure 2.4: The Signal and Measurement Space during Second Iteration

Step 3 of this iteration the residual vector r_1 is correlated with every column-vector in A:

$$A^T r_1 = \begin{pmatrix} 0.4033 & 0.3257 & -0.0198 \\ 0.9150 & 0.9455 & -0.9998 \end{pmatrix}^T \begin{pmatrix} 0.9307 \\ -0.4271 \end{pmatrix} = \begin{pmatrix} 0.000 \\ 0.4615 \\ -0.2862 \end{pmatrix}.$$

Now we can see the first element is zero because of r_1 is orthogonal to a_1. Thus argmax gives r=2, which can be verified in Figure 2.4as the index corresponding to the vector a_2. Step 4 of this iteration tells $S_2 = S_1 \cup \{\beta_2\}$=1 ∪ {2}={1,2}. Then, in the last step we find new residual vector via least square $x_2 = \text{argmin}_x \|a_2 x - y\|_2and\theta_2 = a_2 x_2$, $r_2 = y - \theta_2$ where θ_1=[0.9307 , − 0.4271]^T andnew residual is [0.000 , 0.000]^T.

The second iteration of OMP is now finished and we note that the estimated $\hat{x}$ is a perfect recovery of x.

This example gives us a good idea of recovery of signal based on LSE and orthogonal matching pursuit. In the next part we will talk about the structure of OpenCL and how the program executes in OpenCL.

2 .3 OPENCL (Open Computing Language)

Nowadays, computers, handhelds and embedded computer industry often have a highly parallel computing power such as multi core CPUs (central processing unit) and GPUs (graphic processing unit)[5]. This power helps software developers to use full advantage of heterogeneous processing. Indeed, OpenCL attempts to give the developers an ability to use the parallel computing power. It includes libraries, an Application Programming Interface (API), a language and a runtime system to help software development on all OpenCL supported devices. It is a subset of the C99 standard [6].

2.3.1 Platform Models

The platform model of OpenCL is shown in Figure 2.3. As in the figure the host (like CPU) is connected to many OpenCL compute devices (like Multi-GPU). Every compute device contains many Compute Units. Each compute units is divided into many processing elements, and this is where the actual processing takes place.

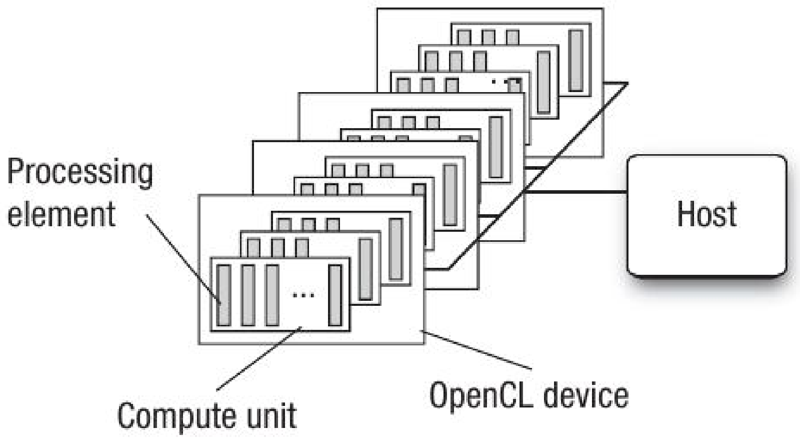


Figure 2.5: OpenCL Platform Models[6]

2.3.2 Execution Model

We can separate execution of OpenCL program to two main parts:1) host code which runs on the host device like CPU, and 2) device code which runs on compute devices. Indeed, the host code defines the context of device code,it also manages the execution of code [5]. Every device code contains some kernels. The kernel is the place of actual processing. OpenCL uses two level hierarchical models to divide the work-items, likes CUDA programming framework. NDRange is an N-dimensional space of workgroups, which defines the execution of a kernel on a device. The number N can be between one to three-dimensional. Each work-group also consists of N dimensional space of work-items. The actual processing happens in the work-item, which is mapped on the processing element.

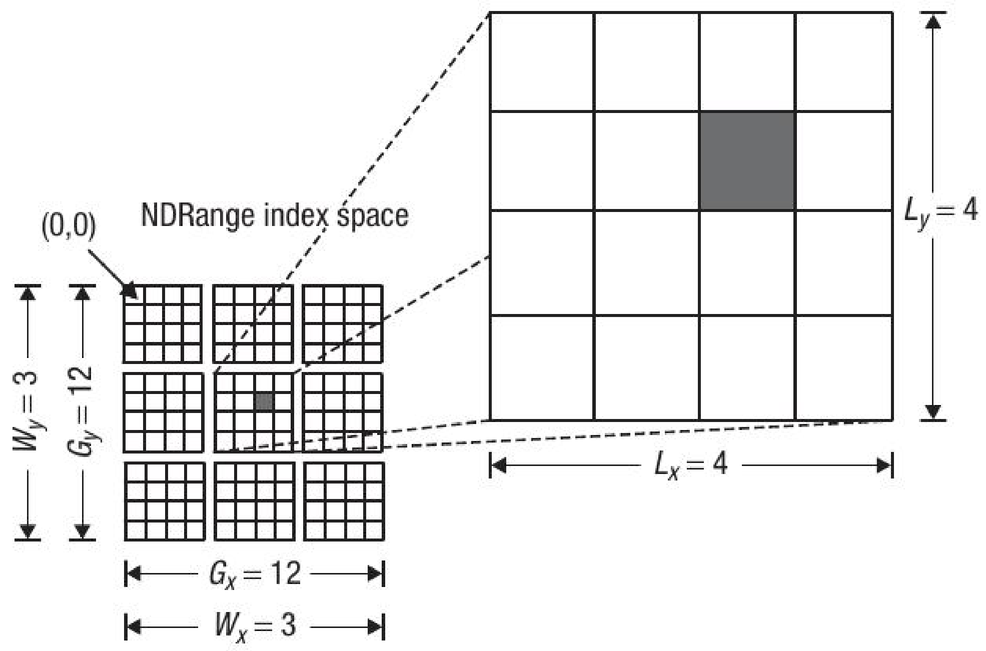


Figure 2.6: OpenCL Execution Model[6]

2.3.3 Memory Model

Figure 2.7shows the memory model used inside a compute device. The execution model is mapped onto this model. Any workgroup is mapped to compute unit, and work-item runs on a Processing Element (PE) [5]. Work-items access to different memory regions like Global memory, Constant memory, Local memory and Private memory. All work-item and work-groups are permitted to read and write from global memory. The Global memory has a Constant memory region, which remains constant during the execution of the kernel. Also, local memory region is just accessible for work-items inside the same work-group. Sometimes, depending on the capabilities of devices, local memory would be mapped onto some dedicated memory region or, if there was no available local memory there, it would be mapped onto the global memory. Each work-item has its own Private memory that is not available to the other work-item.

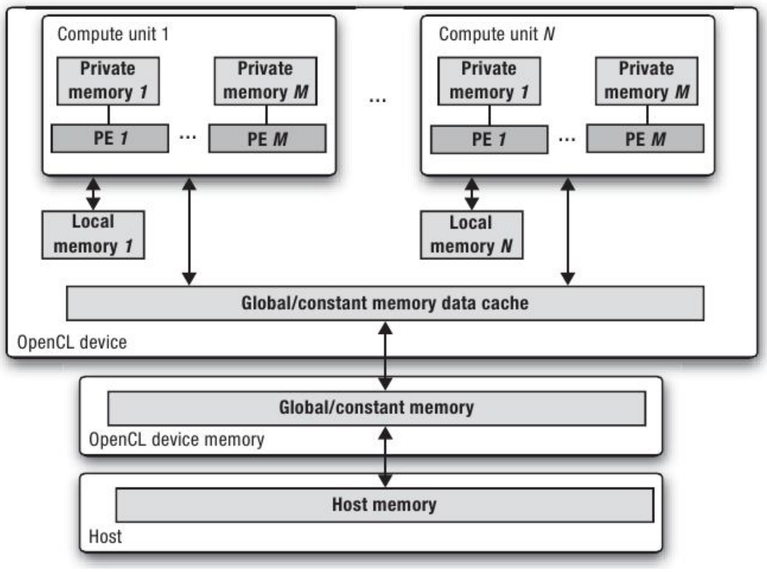
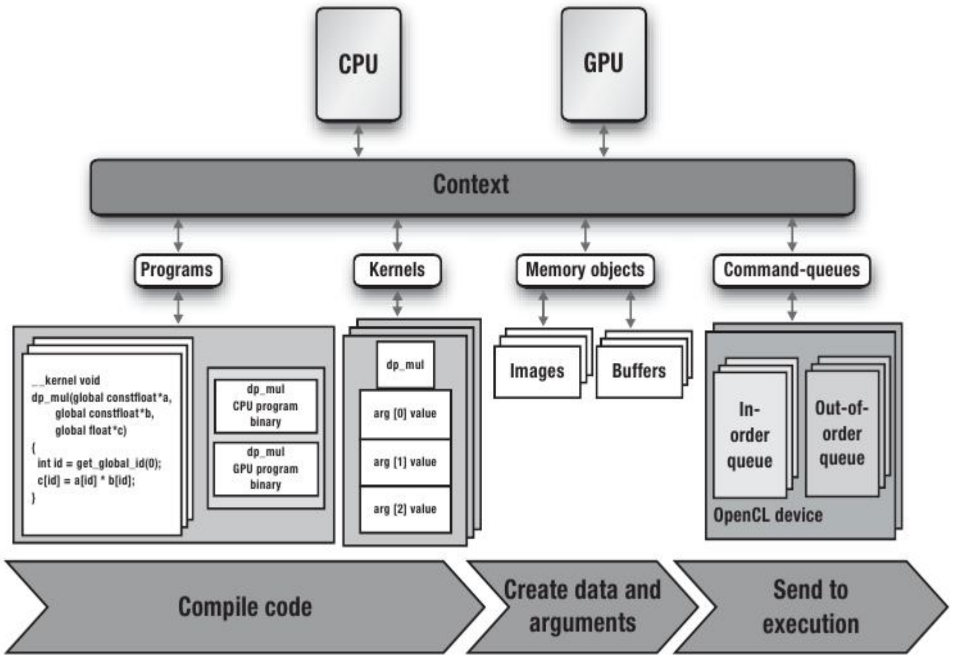


Figure 2.7: OpenCL Memory Models[6]

2.3.4 OpenCL Program Structure

Every program follows these steps to run. At first the host by using OpenCL API, queries the system for OpenCL support. Then, it selects the target device for running OpenCL kernel [5].After that, a context will be created for OpenCL runtime to manage objects like memory, program and kernel objects. Command-queue is part of context which is used for operations on the objects. Then, OpenCL kernel code read and compiled into a binary code file. The OpenCL ICD(installable Client Driver)checks that the kernel is compiled for chosen target. Next step, the data for kernel will be copied to the target device. When this operation happened the host will stall and wait for kernel to be finished. At the end the result will be copied back to the host.



```
__kernel void
dp_mul(global constfloat*a,
       global constfloat*b,
       global float*c)
{
 int id = get_global_id(0);
 c[id] = a[id] * b[id];
}
```

Figure 2.8: OpenCL Program Flow[6]

2.3.5 AMD Architecture

The first cards of the Radeon HD 6850 series were launchedinOctober22, 2010. Performance is differentiated between the GPUs by the number of SIMD arrays each GPU has, the core clock speed, the memory bus width and the number of texture units and Render Output Units(ROP) [21].A GPU consists of 12 compute units (also called SIMD Engines) and each compute unit comprise 80 stream cores, which consists of four processing elements, depending of the GPU model. See Figure 2.9 for a diagram of the GPU architecture and Figure 2.10 for a diagram of a stream core. All the stream cores within a compute unit will perform the same instruction in a lock-step fashion, at each cycle. A VLIW6 is utilized to issue the instructions to the processing elements. All of the processing elements can perform single-precision floating point operations.

Every compute unit has 32 kB of local, on-chip memory called local data share (LDS) and a 8 KB L1 cache. L2 cache is shared by several compute units. The local data share is divided into 32 memory banks, which are four bytes wide and 256 bytes deep [22]. One memory operation can be performed for each bank each cycle, but if more than one operation is map into the same memory bank, a bank conflict occurs and the operations are serialized. A compute unit also has 256 kB of available registers. The register space comprises 16384 general purpose registers, where one register contains four 32-bit values.

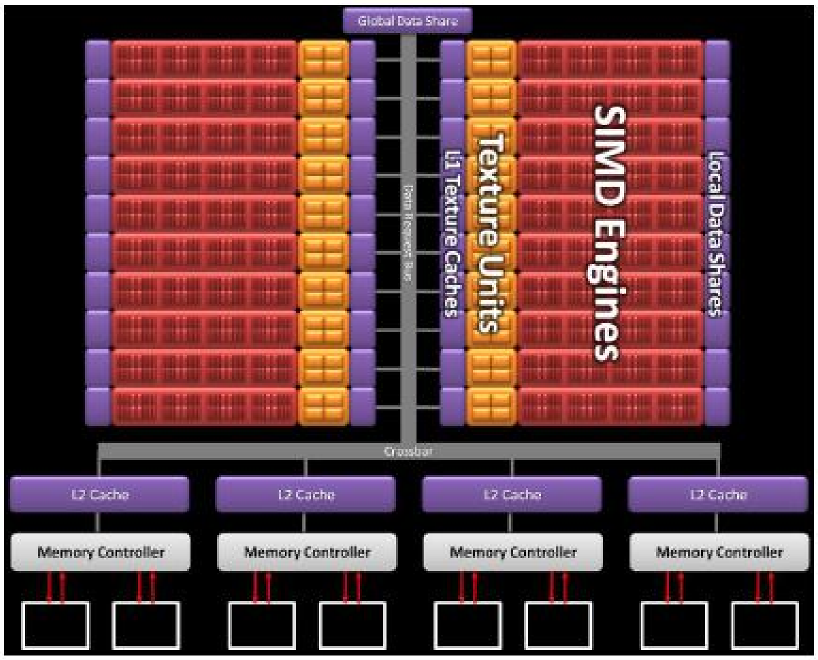


Figure 2.9: Radeon HD6850Architecture. [16]

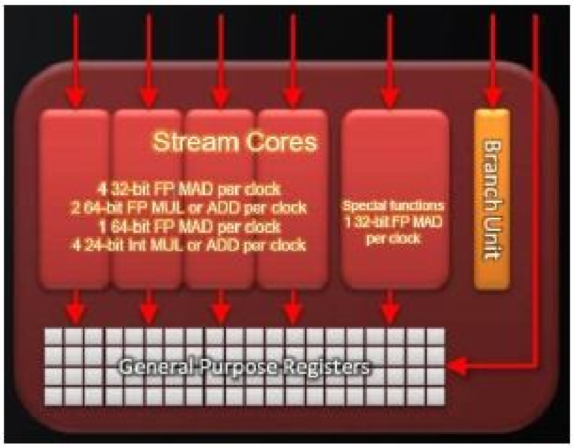

Stream Cores
4 32-bit FP MAD per clock
2 64-bit FP MUL or ADD per clock
1 64-bit FP MAD per clock
4 24-bit Int MUL or ADD per clock
Special functions
1 32-bit FP MAD per clock
Branch Unit
General Purpose Registers

Figure 2.10: Radeon HD6850Architecture. [23]

2.3.6 OpenCL Running on AMD GPU

When work-items are executed on a GPU, they are grouped together in wave fronts. A wave front consists of 64 work-items, that are executed in lockstep on a compute unit. Every work-group is divided into an integer number of wave fronts and to achieve optimal performance, the number of work-items within a work-group should be divisible with the wave front size [22].

As a kernel is being executed, a work-group is assigned to a single compute unit and a work-item runs on a stream-core. Four work-items from the wave front being executed are pipelined on one stream core to hide memory latencies. At each cycle, 16 of the work-items in a wave front execute one instruction. When a wave-front is looked at as a whole, this give the appearance that one instruction is executed every four cycles. If the executions paths of work-items within a wave front diverge, their executions are serialized.

The use of private memory in kernels will map the general purpose registers as long as the capacity allows (Figure 2.10). If more memory is required, the compiler will solve this by generating spill code, and move remaining blocks over to general memory.

2.3.7 ViennaCL

The Vienna Computing Library (ViennaCL) is a scientific computing library written in C++ and based on OpenCL. It allows simple, high-level access to the vast computing resources available on parallel architectures such as GPUs and is primarily focused on common linear algebra operations (BLAS levels 1, 2 and 3) and the solution of large systems of equations by means of iterative methods with optional preconditioned [24]. More relevant information is given in chapter 3.

Chapter 3

PROPOSED APPROACH AND ALGORITHM

3.1 Approach

In any OpenCL application we have two parts. The first part is OpenCL C kernel, which defines the computation for a one instance in the index space, and the second part is C/C++ host program that uses API for configuring and managing behavior of kernel and execution of kernel.

In this thesis, we use ViennaCL to implement OMP over the high performance device. ViennaCL is C++ template which manages the execution of kernel. Also, It selects the high performance device. Indeed ViennaCL is an OpenCL API which do most of process mentioned in chapter 2such as automatic execution of kernel.

In this chapter our main emphasize is to introduce the algorithms and in the net chapter we specifically look on some devices and the performance of implementation.

3.1.1 Function of ViennaCL

The main focus of ViennaCL is on linear algebra operation. Also, ViennaCL uses unified layer to access OpenCL under the hood. To use this library we must know its function and how to use this function. Table 3.1 gives important functions that we want to use in this thesis.

Table 3.1 Table Functions

Function name	**application**
Viennacl	This the main name space for all library
Viennacl::linalg	Namespace of all linear algebra operation
Viennacl::linalg::norm_2	Function to get norm two of vector
Viennacl::linalg::prod	Function to dot product of matrix or vector
Viennacl::linalg::inner_prod	Function to inner product jest to vector
Viennacl::copy	Function to copy data between CPU and GPU
Viennacl::inplace_QR	Function to find RHS
Viennacl::Custom kernel	Use this for optimize performance
Viennacl::inplace_solve	Function to find the least square

3.1.2 Generating Data for OMP

In the compressive sensing experiment, the first thing is to construct sparse signals. So, we create one-dimensional signal by putting few nonzero value coefficient in it. Choosing places of these values happen randomly. These values are generated from normal distribution probability. To implement the signal we use ublas(C++) library. The code is given in Table 3.2 and Figure 3.1 shows a generated signal.

Table 3.2: Generate Sparse Signal

```
std::fill(signal_cpu.begin(),signal_cpuend(),0);

for (inti=0;i<sparse;i++){

intindx=(std::rand()%n_component);

indice(i)=indx;

signal_cpu(indx)= randgauss(-20,20);"////put random number in random place

viennacl::fast_copy(signal_cpu,signal);        ////copy the signal into high
performance
```

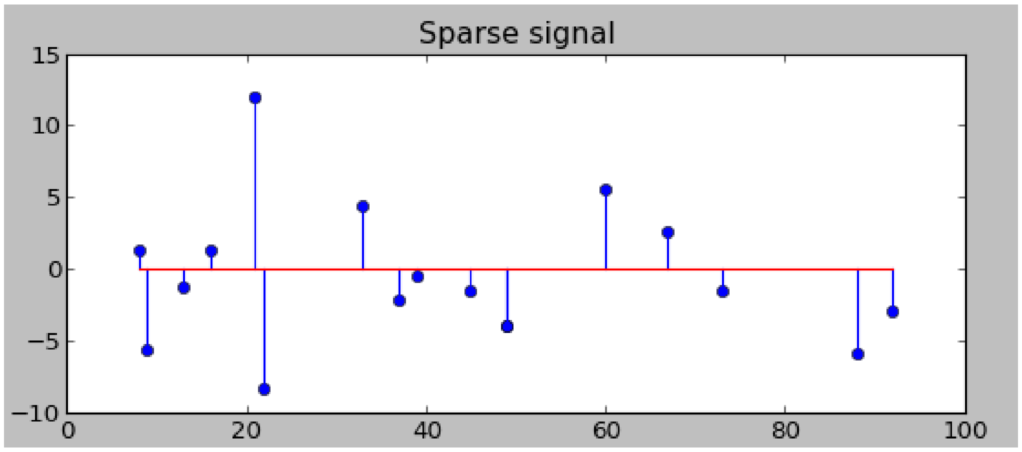


Figure 3.1: Generated Sparse Signal

This was the algorithm that makes a sparse signal. Figure 3.2 gives of process in detail.

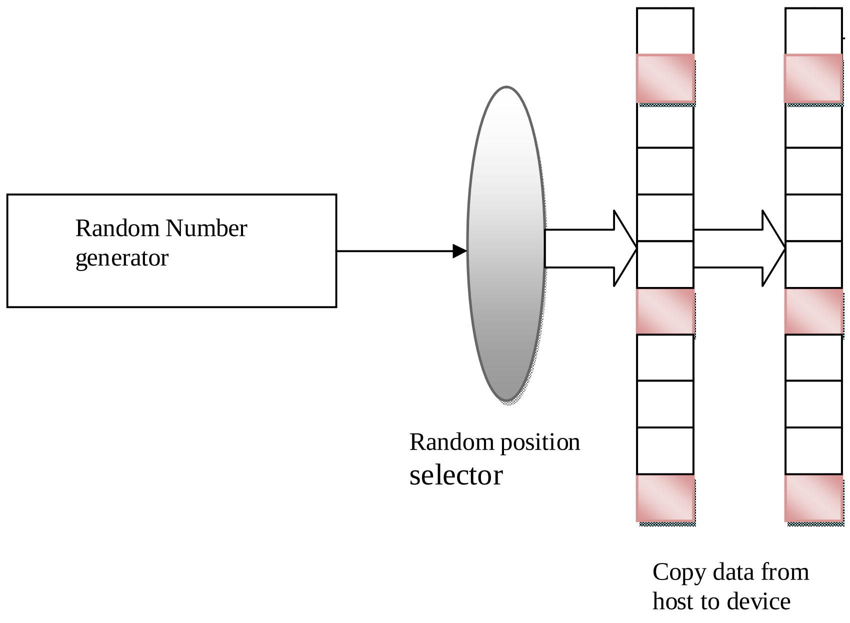


Figure 3.2 Generate and Copy Data

3.1.3 Generate Normalize Dictionary

According to Zhangand Mallat [3] to get stable result the dictionary must be normalized. Thus, we generate dictionary by dimensions of our signal and measurements. After that we extract its columns as vector and divided those by norm 2 of each vector to get normalize vector. Then, vectors must be rejoined together to make a new normalized dictionary. This process can be done by the set of codes in Table 3.3. In Figure 3.2 depicts one generated dictionary

Table 3.3: Generate Normalized Data

```
boost::numeric::ublas::scalar_value<float> h1;

for (int i1 = 0; i1 < n_component;i1++){

v_cpu=(boost::numeric::ublas::column(dictionary_cpu, i1));

h1=(boost::numeric::ublas::norm_2(v_cpu));

v_cpu =(1/h1) * v_cpu;                    ////multiply of vector to inverse

norm

 (boost::numeric::ublas::column(dictionary_cpu, i1))= v_cpu;}

viennacl::copy(dictionary_cpu,dictionary);      ////copy the Dictionary into hp
```

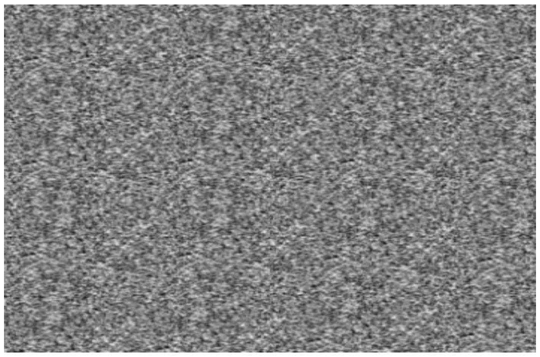


Figure 3.3: Generated Dictionary

The same process as Figure 3.2 happens for generating data. First, the data is generated in host then it is copied to the device.

3.1.4 Generate Measurements

Based on chapter 2 to create reliable measurement we need to have some idea of the RIP or mutual coherence. To obtain measurement of the signal, dictionary must be multiplied by the signal. This process happens in the high performance device. Linear algebra features of using ViennaCL can easily do the task. See Table 3.4 and Figure 3.3 for codes and outputs.

Table 3.4: Generate Measurement

```
////create measurment

#include <viennacl/vector.hpp>

m = viennacl::linalg::prod(dictionary,signal); ////m is our measurement in hp
```

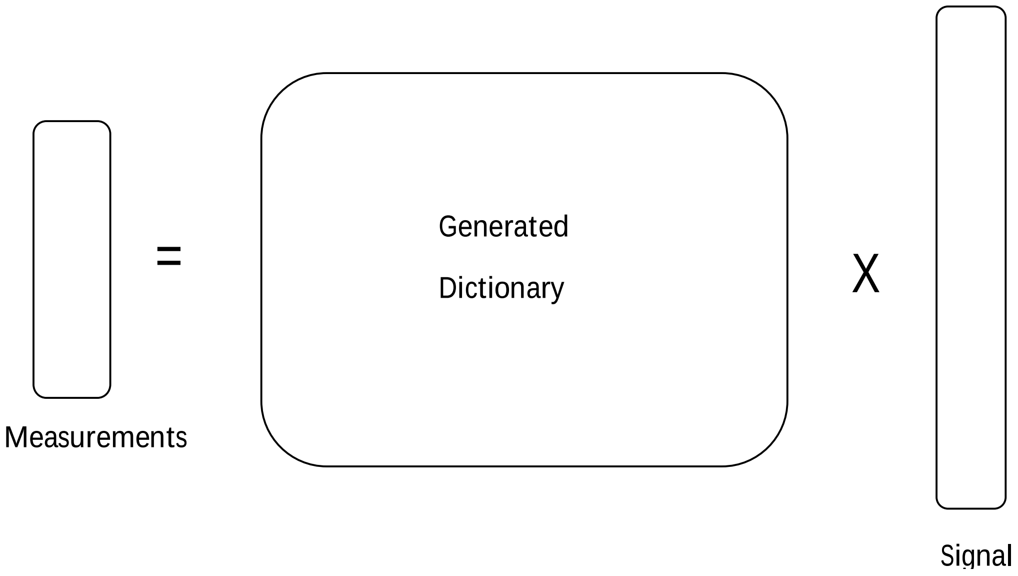


Figure 3.4: Generate Measurements

Figure 3.5 is given to show the process of the matrix vector production on device.

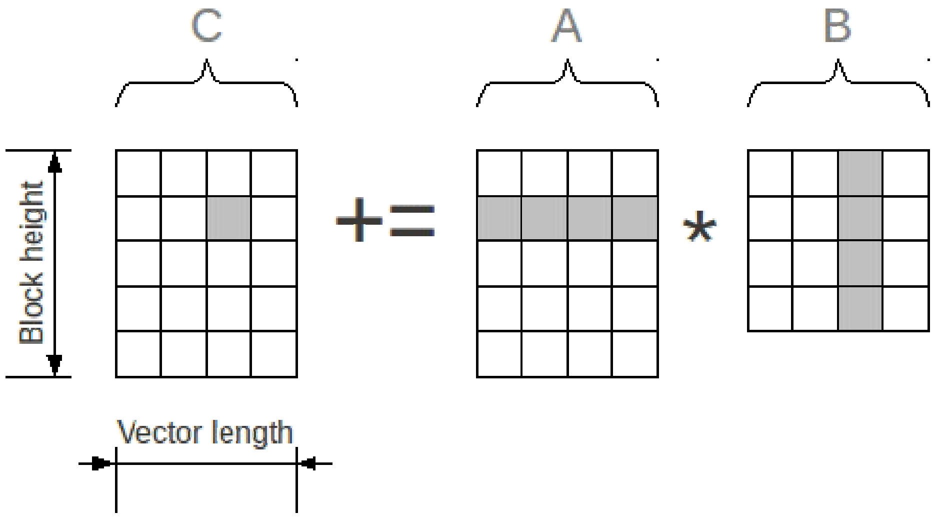


Figure 3.5 Matrix-Matrix multiplications

3.2 Implement of Orthogonal Matching Pursuit

Now we describe the details of the implementation of OMP:

Step1. In step one of implementation of OMP recovery we must set the condition to recovery of the signal. This condition is the level of sparsity (k) of signal or an upper-bound number of the non-zero coefficient of signal. Then, we should find the maximum correlated atom of dictionary. To do so we choose the index β_i by finding the largest correlation between $\{a_j\}_1^N$ and residual of y. in this implementation by using for loop and using indexerwe can to find the argmax. The complexity of this of this step is O($M \times N$)**.**

Table 3.5: Argmax code

```
z=viennacl::linalg::prod(trans(dictionary),residual);
//std::cout<<z<<std::endl;
viennacl::fast_copy(z,killer);
for (int i=0;i<n_component;i++){
killer[i]=fabs(killer[i]);}
float elem=*std::max_element(killer.begin(),killer.end());
//std::cout<<elem<<std::endl;
intpos = std::find(killer.begin(), killer.end(), elem) - killer.begin();
indcol(j)=pos;
viennacl::range          col(pos,pos+1);
viennacl::range          col1(j,j+1);
viennacl::project(new_dictionary,all_row,col1)=viennacl::project(dictionary,all_row,
col);   ////reweight dictionary
```

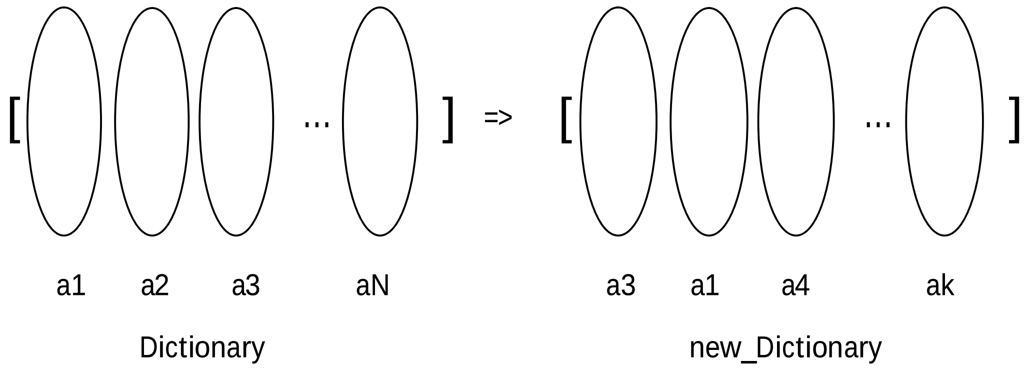


Figure 3.6 New Dictionary after k iteration

The same process as Figure 3.5 is used in this algorithm.

After k iterations we have new dictionary that obtain columns from each of these iteration. See figure 3.4 for an illustration.

Step2. This step is the most important step in the whole recovery of signal. By using the new dictionary and solve optimization problem of below we can estimate the signal. The most difficult thing in this method for the computation of signal is matrix inverse of the dictionary, in solution the LSE problem: $\boldsymbol{x_i} = \textbf{argmin}_x \|\boldsymbol{a_i x} - \boldsymbol{y}\|_2$to solve the above optimization problem we can use pseudoinverse.

$$S_i = ({A_i}^T A_i)^{-1} {A_i}^T y \tag{3.1}$$

this method is computationally expensive and also for large scale of data it tends to be unstable. So to find the answer of this equation we need to find more stable also less expensive method. Onesuch method is called QR decomposition.

Step2.1.QR factorization is a method that uses Gram-Schmidt to make Q and R such that where Q is orthogonal and R is upper triangular matrix [7].

By using QR decomposition we can find the answer of least square problem as:

$$x_i = {R_i}^{-1} {Q^T}_i y \tag{3.2}$$

Just by using upper right hand side of new dictionary we can get the estimate signal to implement it in the OpenCL device. We must define Gram-Schmidt kernel at first. Then, vectorize the matrix as before and put that vector in that kernel then make new right hand side matrix after that. At the end are must inverse this matrix and then do perform of the product to new dictionary with the measurement.

Note that ViennaCL has a very fast and good template to find QR decomposition of matrix. thus, to implement our problem over high performance device we use this template as it is shown in Table 3.6Based on QR decomposition the complexity at this level is O (M^3).Another important thing in calculation of Q and R is setting up block size for parallel computing which is implemented by find the auto_block size code in Table 3.6.

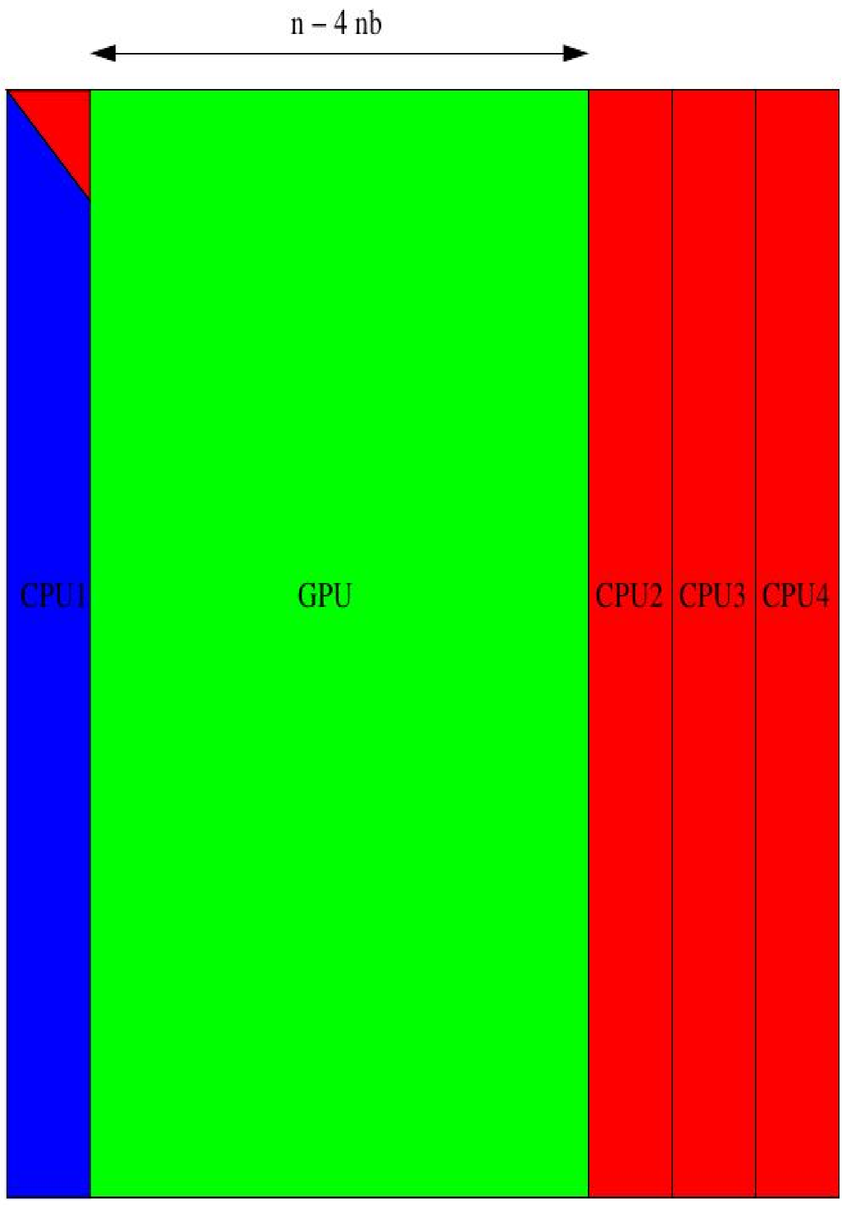


Figure 3.7 QR Decomposition Hybrid Method

Table 3.6 Least Square

```
std::vector<float>hybrid_betas = viennacl::linalg::inplace_qr(help,1024);

//std::cout<<help<<std::endl;

// compute modified RHS of the minimization problem:

// b := Q^T b

viennacl::linalg::inplace_qr_apply_trans_Q(help, hybrid_betas, vcl_b);

viennacl::range                     vcl_range(0,j+1);

viennacl::matrix_range<VCLMatrixType>   vcl_R(help, vcl_range, vcl_range);

viennacl::vector_range<VCLVectorType>   vcl_b2(vcl_b, vcl_range);

// Final step: triangular solve: Rx = b'.

// We only need the upper part of A such that R is a square matrix

viennacl::linalg::inplace_solve(vcl_R, vcl_b2, viennacl::linalg::upper_tag());

gama=viennacl::linalg::prod(new_dictionary,vcl_b2);
```

Step3. The important thing that we get in step 2 is that we find the orthogonal projection of dictionary over measurements, and because of that we call this method orthogonal matching pursuit. At the end we must re-update the residual to find the next correlated column.

Chapter 4

PERFORMANCE AND RESULTS

In this chapter, the experimental results and performance are presented. We first describe the environment of test we then give results of our experiment.

4.1 Environment of Test

This test includes two different high performance devices. Both devices are on the same platform. The first device is GPU from advanced micro devices (AMD),codename BARTS with 12 compute units and 1024 MB global memory and The second device is Intel® core 2 Dou dual Core CPU with 4096 MB RAM size. All programs were compiled in Eclipse IDE by G++ compiler in LINUX operating system. The specifications of both devices are showed in Table 4.1.

Table 4.1: High Performance Device Specification

Property	AMD 6850	Intel core 2 Dou4500
Graphic Bus Technology	PCI-Express16X	NA
Memory(MB)	1024	4096
Core Clock(MHZ)	775	2400
Compute Unit	12	2
Stream processor	128	NA
Memory Bandwidth(MHZ)	134400	6400

4.2 Test Data and Evaluation

Various sparsity levels are chosen, and for each level k, the minimum acceptable measurement number M is decided [7]. Table 4.2 provide all the data used in our experiment.

Table 4.2: Size on Testing Signal

N(signal size)	M (measurement size)	k (Sparsity Level)
Small Signal Size 1024	60	4
	120	8
	240	16
	360	28
Medium Signal Size 2048	240	8
	360	16
	512	24
	768	32
Large Signal Size 4096	512	16
	768	24
	890	32
	2048	64
Massive Signal Size 16384	768	32
	1024	64
	1536	78
	2048	256

The performance of OMP implemented in OpenCL will be evaluated against that of CPU implementation in forms of the time and accuracy.

4.3 Challenge of Bandwidth

The biggest issue in running the OpenCL software or any high performance language is a time of transfer data between graphic card global memory and the RAM. Because of this problem we decide to introduce new solution. In this solution we decide to run some of functions which have a very poor performance on the GPU on a CPU. This is type of heterogeneous computing. For this reason first of all we load all dictionary, signal and measurement on both CPU and CPU then, We just set indexer and send it to device choose column to process and save it in device then at the end load the data into RAM to view these result.

4.4 Results for Signal of Small Size

We now in this section give test results speed for signal of small size (N=1024):

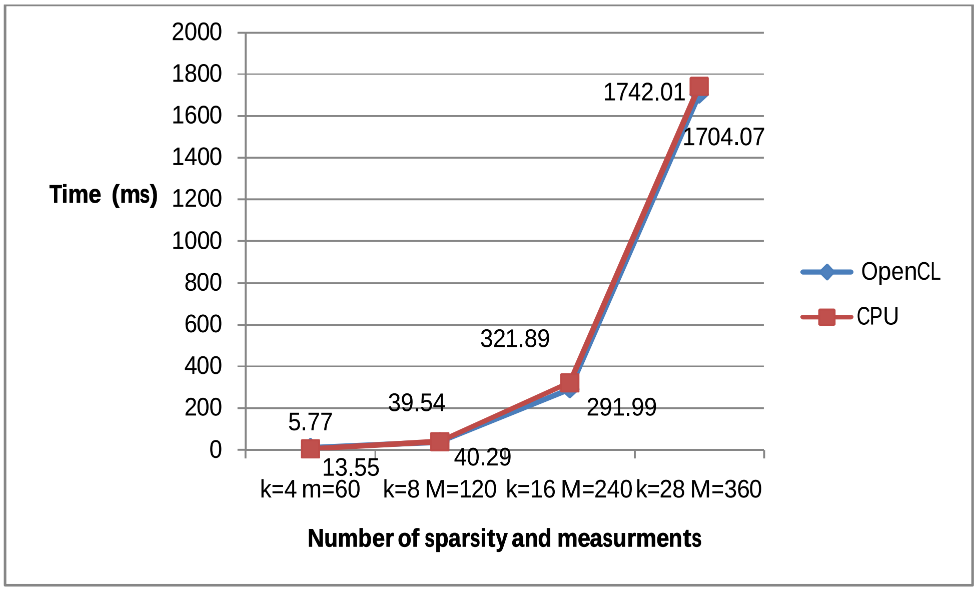


Figure4.2: Execution Time CPU-OpenCL for Small Signal (N=1024)

Figure 4.2shows the execution time for OpenCL and CPU implementation.

We see that for small size array of signals there is little difference between using OpenCL and CPU. Table 4.3 gives information about the ratio in execution time.

Table 4.3: Speedup Ratio Small Signal (N=1024)

M (Measurements Size)	k (Sparse Size)	Ratio in Time OpenCL / CPU
60	4	0.42
120	8	0.92
240	16	1.02
360	28	1.03

To show the results of this recovery input and output and error of signal plotted in Figures: 4.3 and 4.4. Table 4.4 gives performance of estimation. RMSE is calculated according to formula (4.1).

$$\text{RMSE} = \frac{1}{\sqrt{N}} \|x - \hat{x}\| \qquad (4.1)$$

Table 4.4: Recovery Error for Small Sizes Signal (N=1024)

(M) Measurements	(k) Sparse Level	RMSE OpenCL	RMSE CPU
60	4	1.98e(-7)	2.06e(-7)
120	8	4.56e(-7)	3.83e(-7)
240	16	6.15e(-7)	5.06e(-7)
360	28	1.15e(-6)	1.07e(-6)

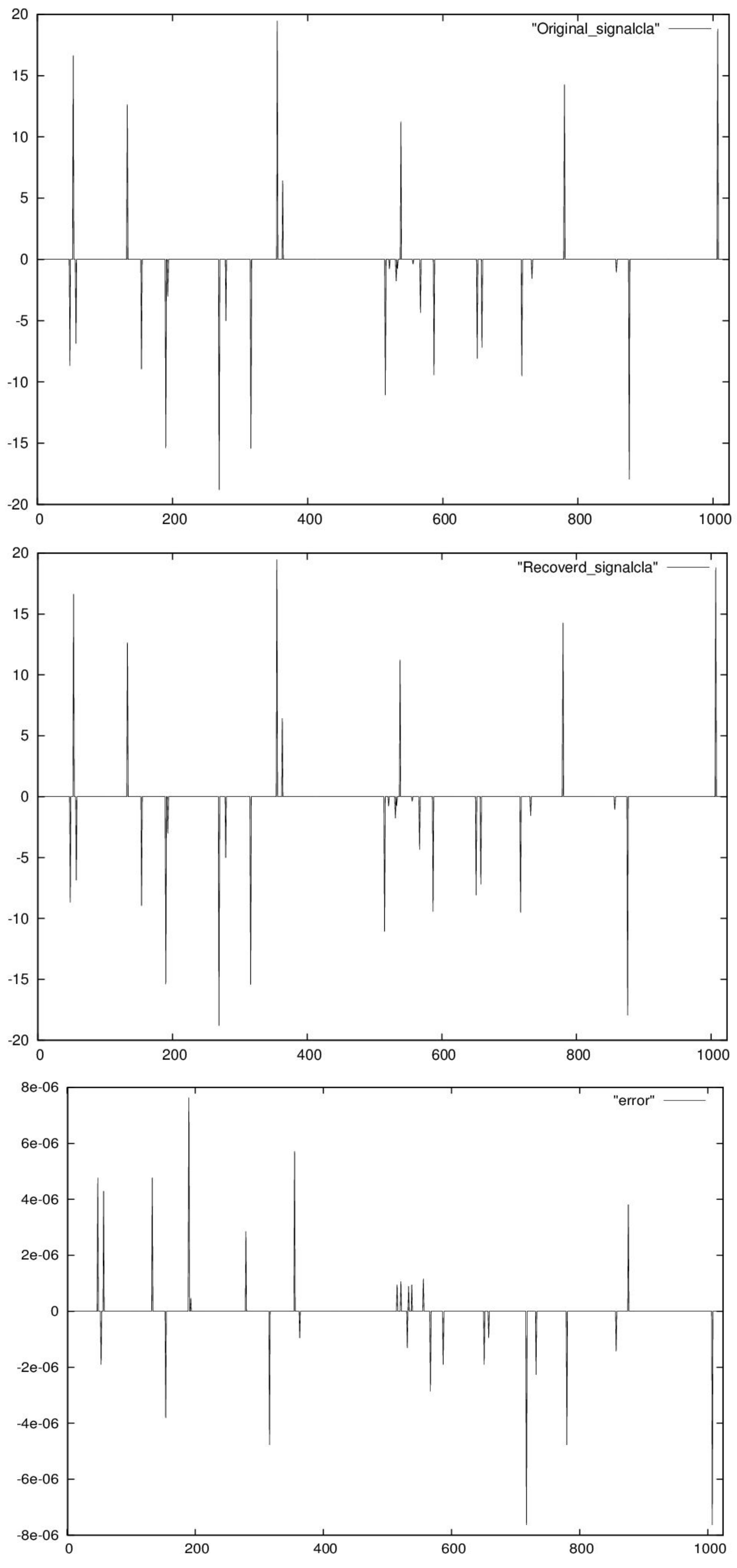


Figure 4.3: Input, Recovered and Error Small Signal OpenCL

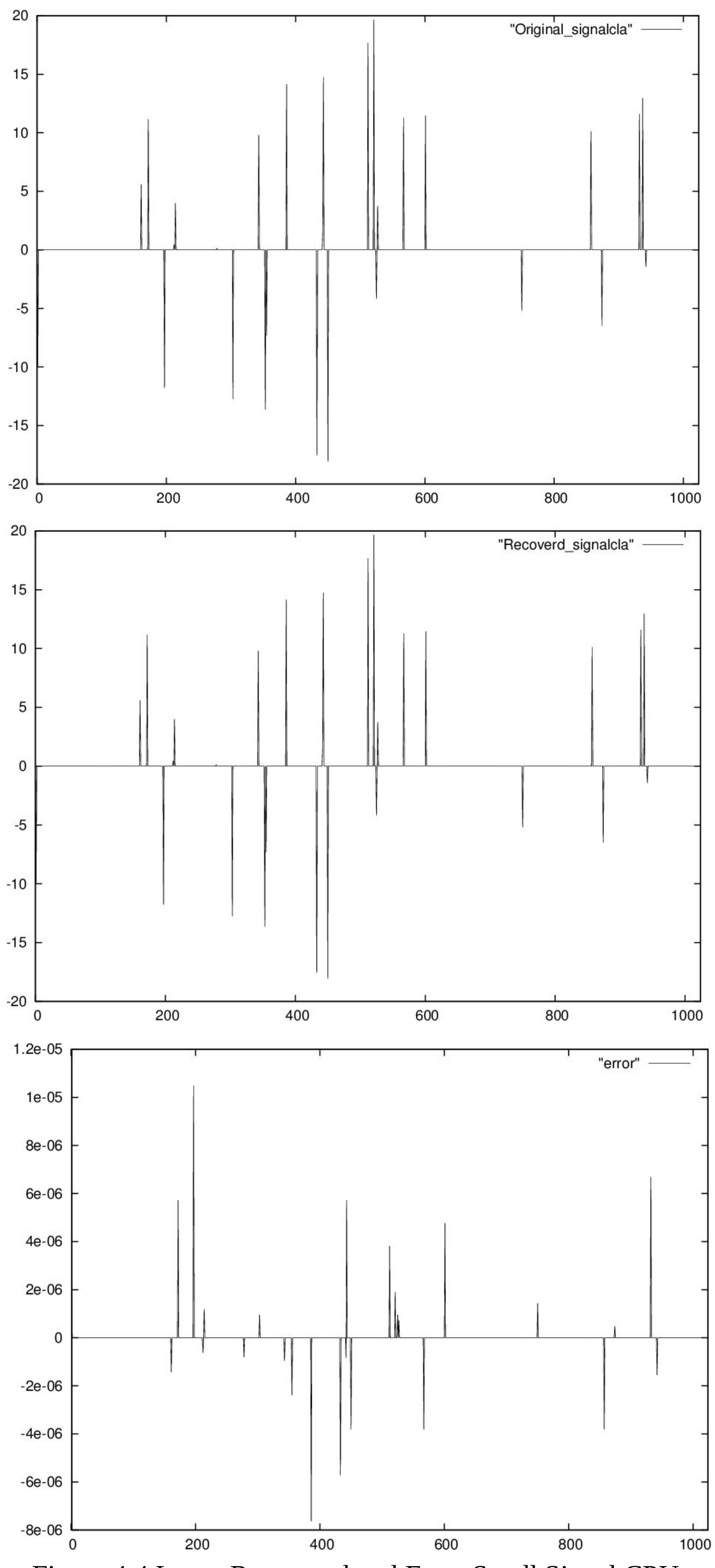


Figure 4.4 Input, Recovered and Error Small Signal CPU

To show the stability of algorithm and implementation we now add some Gaussian noise to the measurement. Figure 4.5 shows the noise by mean 0 and variance 1 N(0,1).

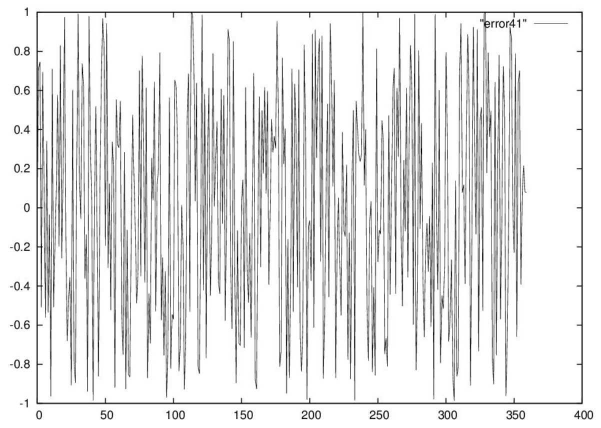


Figure 4.5 Gaussian Noise added to Measurements

Table 4.5 gives the error with presentation of error by implemented method in this thesis, where the SNR is calculated as follows:

$$\text{SNR}=20\log\left(\frac{\|x\|}{\|x-\hat{x}\|}\right) \tag{4.2}$$

Table 4.5: Small Signal Error in Presentation of Noise by Given Implementation

(M) Measurement	(k) Sparse Level	RMSE Percentage	SNR dB
60	4	2.16	67.41
120	8	4.06	61.57
240	16	9.01	56.07
360	28	15.82	45.60

4.5 Results for signal of Medium Size

In this part first condition for medium signal size will be checked.

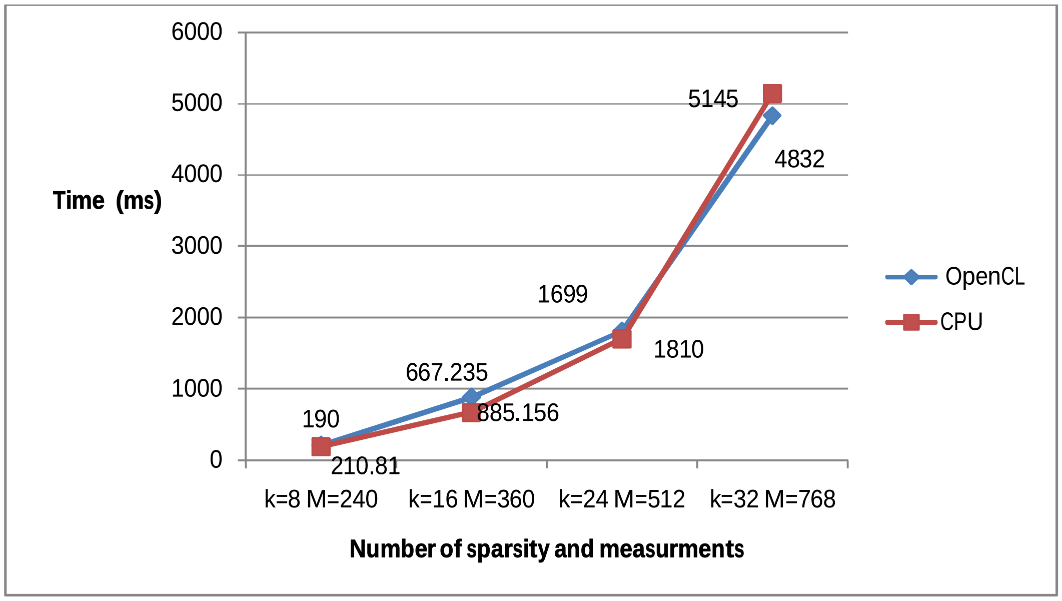


Figure4.6: Execution Time CPU- OpenCL Medium Signal (N=2048)

We see that for medium size array of signals there is a same speed difference between using OpenCL and CPU. Table 4.6 gives information about the ratio in execution time.

Table 4.6: Speed up Ratio Medium Signal (N=2048)

(M) Measurement	(k) Sparse Level	Ratio in Time OpenCL / CPU
240	8	0.62
360	16	0.76
512	24	0.94
768	32	1.06

To show the error of this recovery Figures: 4.7 and 4.8 are plotted. Table 4.7 gives error for all set of medium size signal.

Table 4.7: Recovery Error for Medium Sizes Signal (N=2048)

(M) Measurement	(k) Sparse Level	RMSE OpenCL	RMSE CPU
240	8	1.37e(-7)	1.25e(-7)
360	16	6.33e(-7)	5.92e(-7)
512	24	7.45e(-7)	4.38e(-7)
768	32	2.44e(-6)	1.83e(-6)

When a Gaussian noise is added to the measurement. Table 4.8 gives output error in the presentation of noise.

Table 4.8: Medium Signal Error in Presentation of Noise by Given Implementation (N=2048)

(M) Measurement	(k) Sparse Level	RMSE Percentage	SNR dB
240	8	2.23	60.78
360	16	6.86	55.31
512	24	7.37	51.43
768	32	13.05	43.38

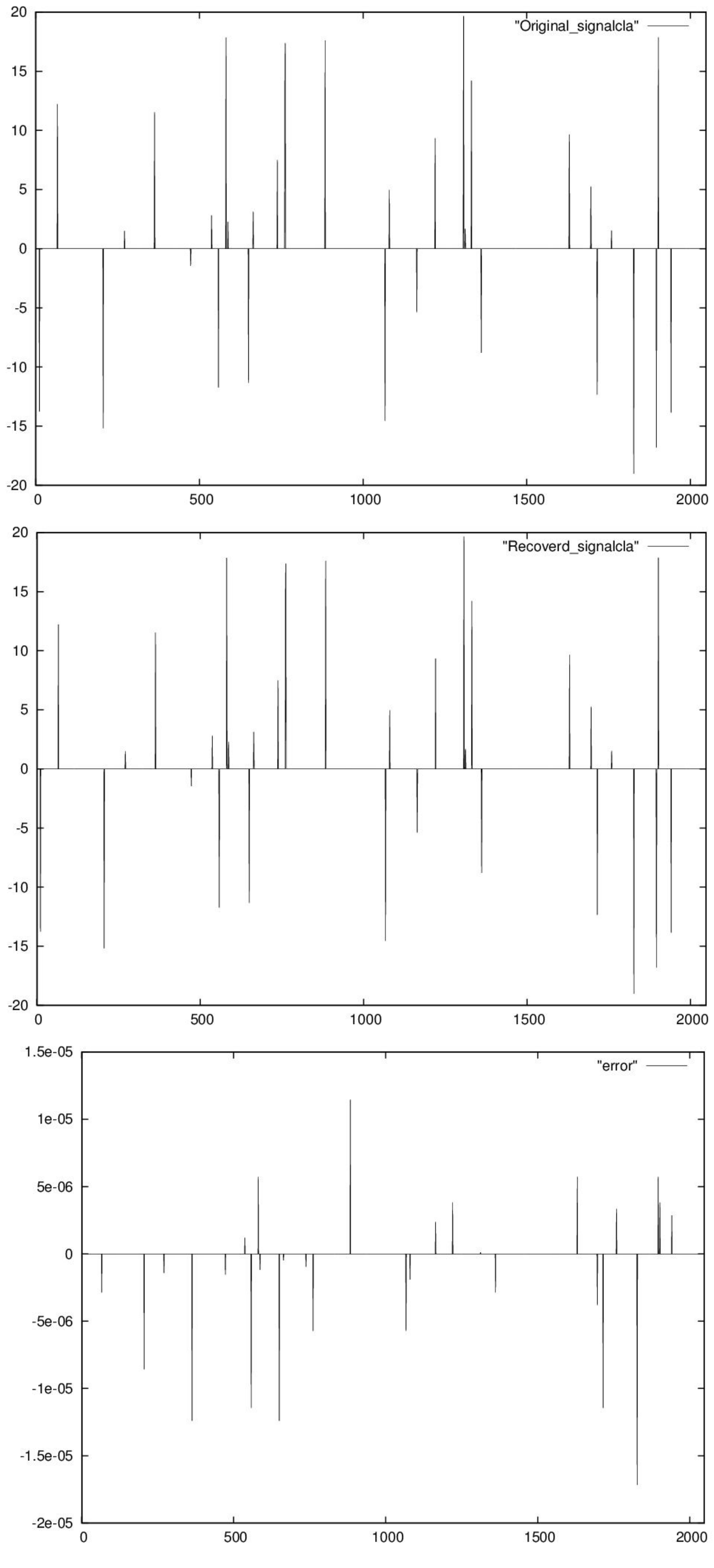


Figure 4.7: Input, Recovered and Error Medium Signal OpenCL

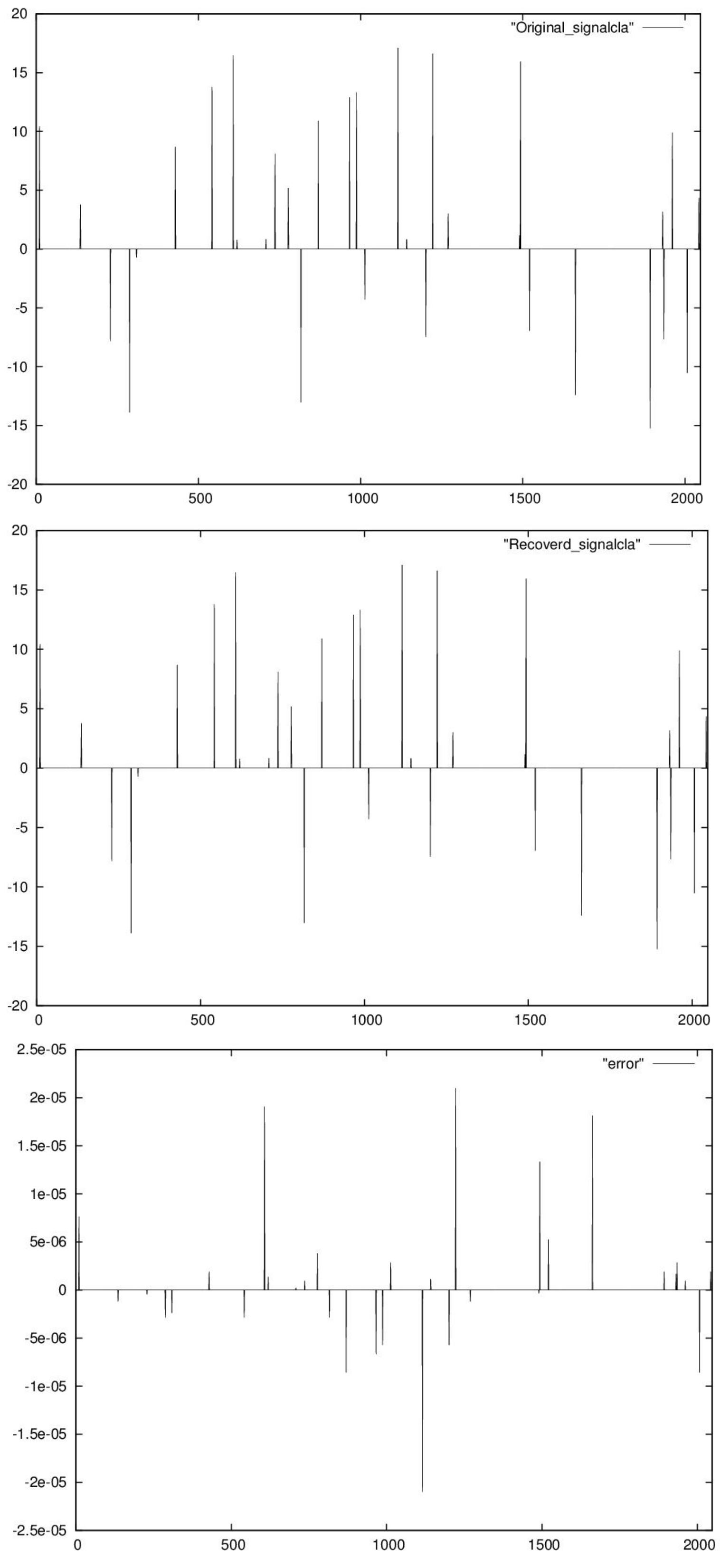


Figure 4.8: Input, Recovered and Error Medium Signal CPU

4.6 Results for Signal of Large Size

Time result for large signal size showed in Figure 4.9.

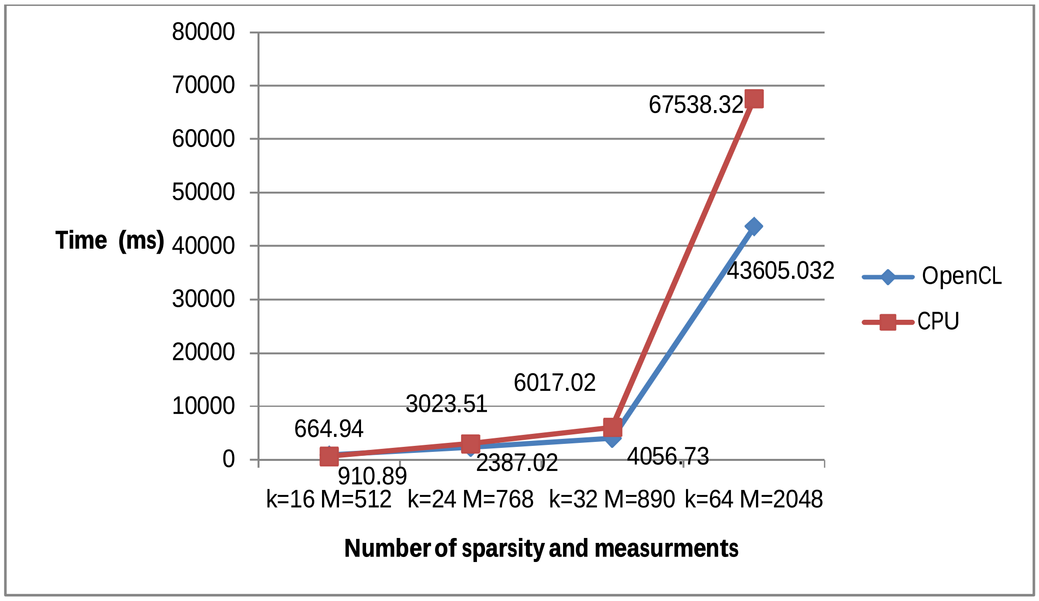


Figure4.9: Execution Time CPU-OpenCL Large Signal (N= 4096)

Figure 4.9 shows strength of parallel computing for large array of signal. It shows when computational complexity goes higher OpenCL will give a better performance. Table 4.9 shows the ratio of speedup.

Table 4.9: Table 4.6: Speed up Ratio Large Signal (N=4096)

(M) Measurement	(k) Sparse Level	Ratio in Time OpenCL / CPU
512	16	0.72
768	24	1.24
890	32	1.48
2048	64	1.54

To show the error Figures: 4.10 and 4.11 are plotted. Table 4.10 gives error ratio for all large signals.

Table 4.10: Large Signal Size Error Ratio (N=1024)

(M) Measurements	(k) Sparse Level	RMSE OpenCL	RMSE CPU
512	16	2.12e(-7)	2.83e(-7)
768	24	4.83e(-7)	4.15e(-7)
890	32	5.22e(-7)	4.48e(-7)
2048	64	4.27e(-6)	2.13e(-6)

A Gaussian noise is added to the measurement to show the robustness for large size signal. Table 4.11 gives output error in presentation of noise in dB.

Table 4.11: Large Signal Error in Presentation of Noise by Given Implementation (N=4096)

(M) Measurements	(k) Sparse Level	RMSE Percentage	SNR dB
512	16	2.86	66.09
768	24	6.24	52.18
890	32	7.02	50.90
2048	64	7.56	50.71

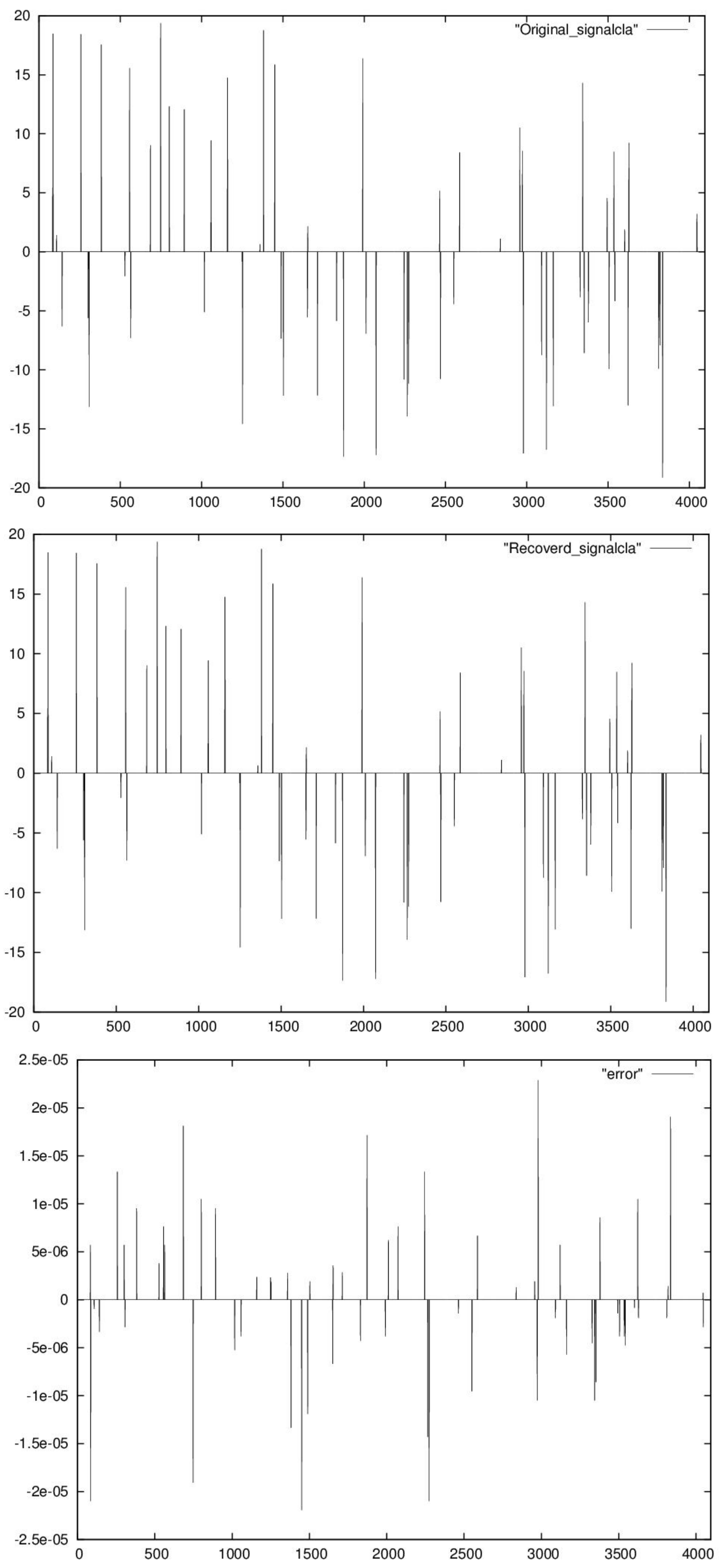


Figure 4.10: Input, Recovered and Error Large Signal OpenCL

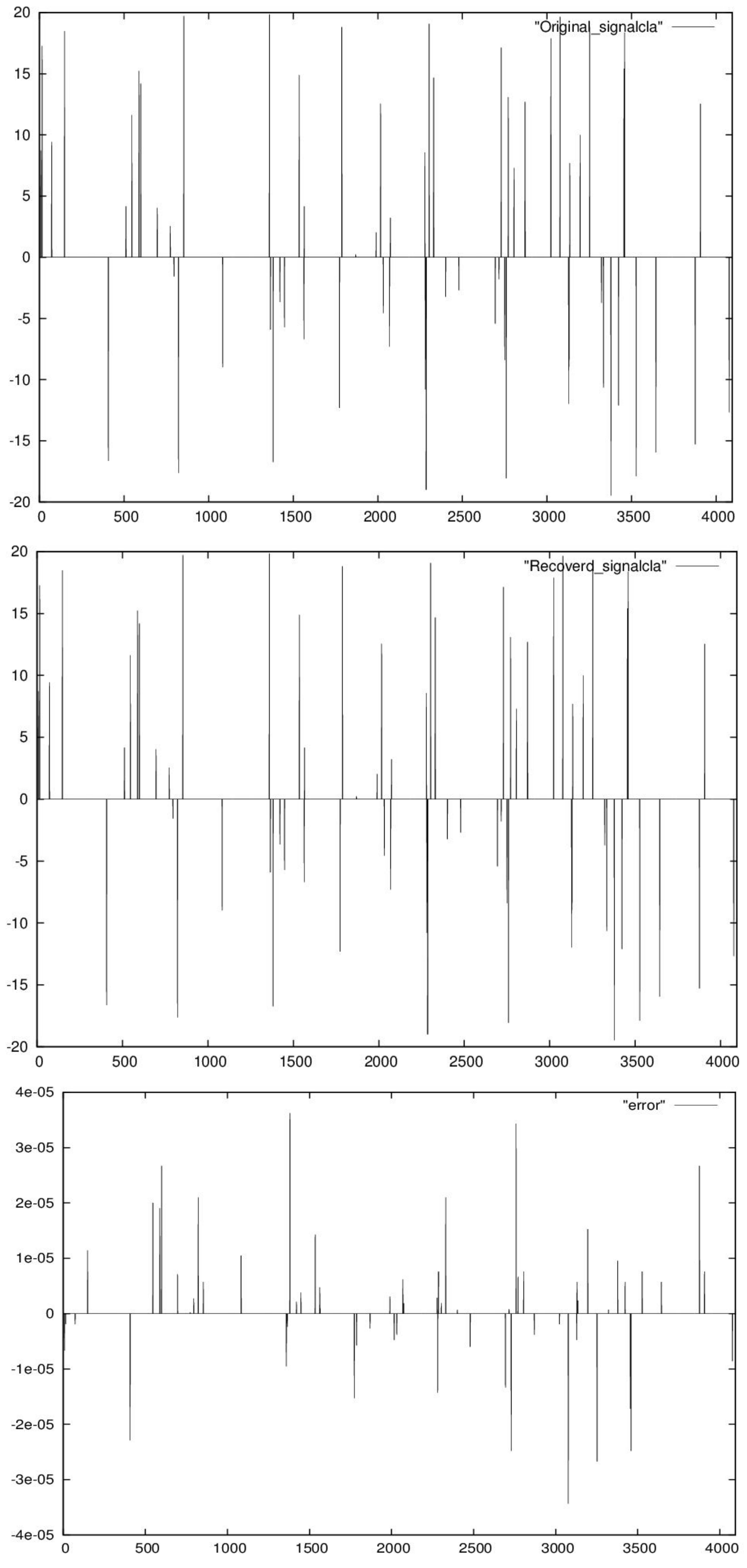


Figure 4.11: Input, Recovered and Error Large Signal CPU

4.7 Results for Signal of Massive Size

Massive size of signal compared results shown in Figure 4.12.

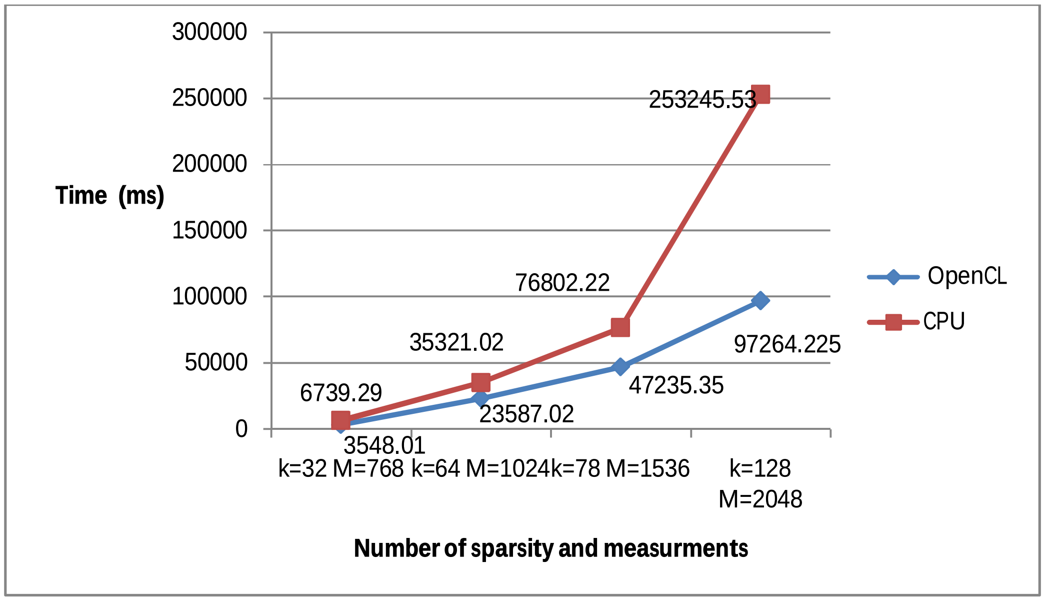


Figure4.12: Execution Time CPU-OpenCL Massive Signal (N=16384)

Figure 4.12 showed that OpenCL completely outperform CPU in calculation. Table 4.12 gives the ratio of speedup.

Table 4.12: Massive size Signal Speed up Ratio (N=16384)

(M) Measurements	(k) Sparse Level	Ratio in Time OpenCL / CPU
768	32	1.89
1024	64	1.49
1536	78	1.48
2048	128	2.60

To show the error Figures: 4.13 and 4.14 are plotted. Table 4.13 gives error ratio for all massive signals.

Table 4.13: Massive Signal Size Error Ratio (N=16384)

(M) Measurements	(k) Sparse Level	RMSE OpenCL	RMSE CPU
768	32	1.92e(-7)	1.35e(-7)
1024	64	3.63e(-7)	3.04e(-7)
1536	78	7.11e(-7)	7.04e(-7)
2048	128	5.42e(-6)	5.08e(-6)

Then a Gaussian noise added to the measurement to show the stability of algorithm in massive signal size. Table 4.14 gives output error in presentation of noise.

Table 4.14: Massive Signal Error in Presentation of Noise by Given Implementation (N=16384)

(M) Measurements	(k) Sparse Level	RMSE Percentage	SNR dB
768	32	4.46	50.67
1024	64	8.43	43.81
1536	78	7.36	47.82
2048	128	9.56	46.91

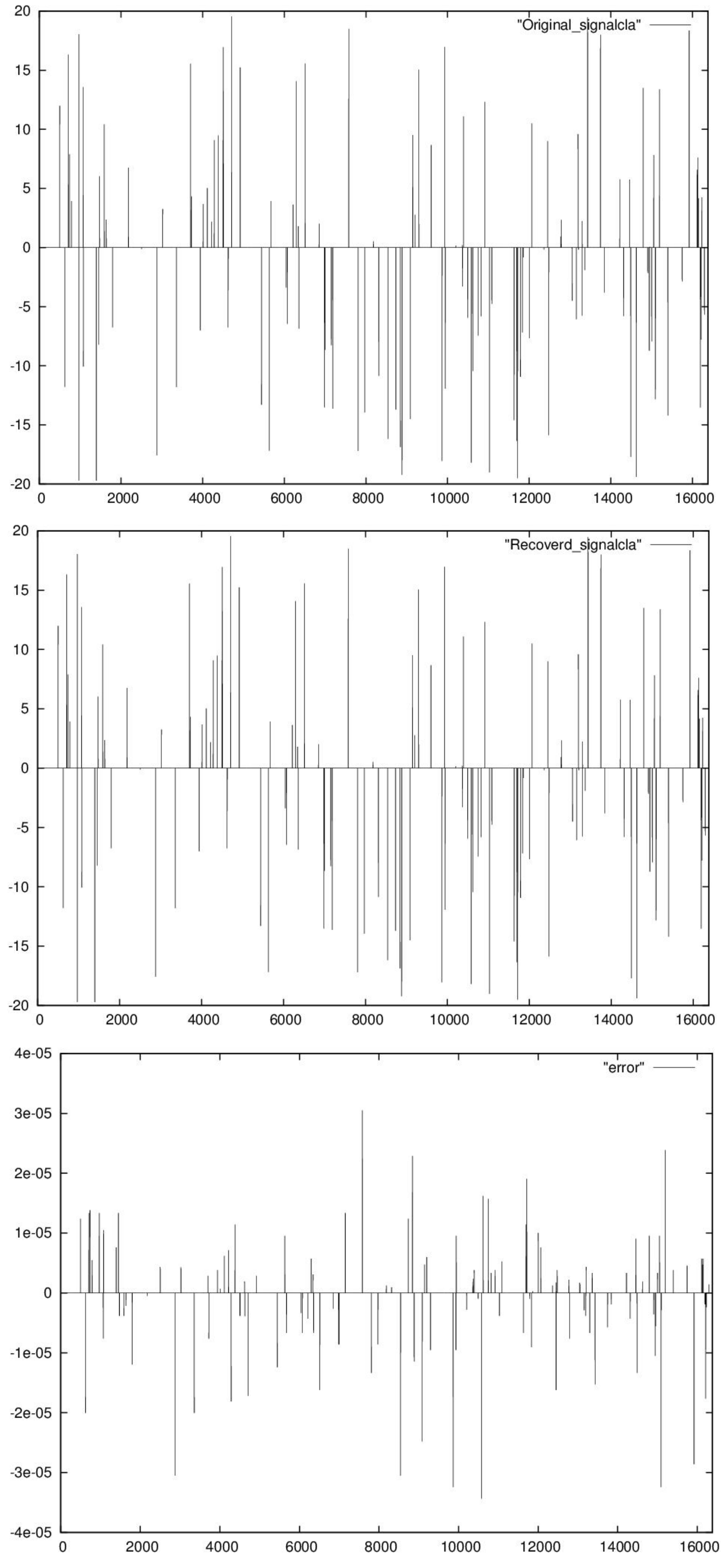


Figure 4.13: Input, Recovered and Error Massive Signal OpenCL

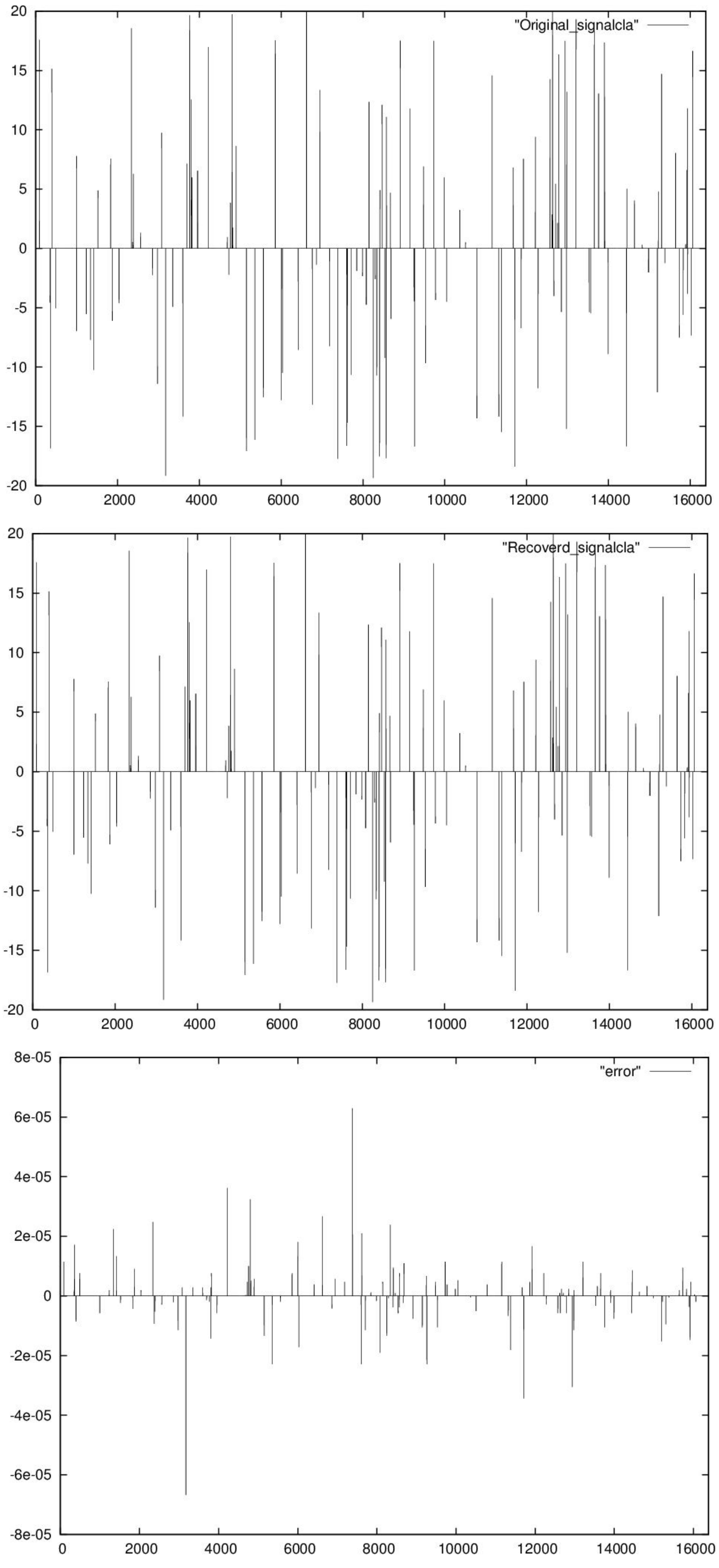


Figure 4.14: Input, Recovered and Error Massive Signal CPU

4.8 Discussions

First we recall computational complexity of operation in argmax and QR decomposition. Complexity of the argmaxis O($M \times N$) and the QR complexity is O(M^3).

The results are evaluated in terms of speed and accuracy. The purpose is to show performance and quality of this implementation.

In the case of small size, OpenCL has a little improvement in calculation time (Table 4.3). And the accuracy of reconstruction is very good with or without noise.

In the case of medium size the results give the similar evaluation to those for small size signals.

The situation begins to change for large sized signals. That is the reason why we use parallel processing. We see an improvement (over two times in speed) of OpenCL over the CPU implementation (Table 4.9). This has no change in error (Table 4.10, 4.11).

The effect of parallel implementation becomes apparent for massive size of signals. More improvement in speed at a cost of slightly more error achieved (Table 4.10, 4.11, 4.12).

Chapter 5

CONCLUSIONS

5.1 Conclusion

In this thesis, we have implemented OMP algorithm on high performance devices for both CPU and GPU. With respect to the obtained results and outputs we have the following conclusions:

First, the fast signal recovery of OMP can be achieved by parallel implantation, when appropriate devices are chosen. It is particularly faster when the size of the signal is large.

Second, in view of OpenCL portability, It is possible to run this implementation over multi-platforms. It is also possible to use all computing resources available in the system.

This study also demonstrates the need of heterogonous computing for reconstruction of large size signal, as they require expensive computation.

5.2 Future Work

This work in the thesis can be further improved by following:

- Implementing batch OMP by HP device.
- Online dictionary learning
- Multi-dimensional data recovery in Compressive sensing
- Optimize the OpenCL in the kernel of Linux
- Implementing QR OMP in OpenCL

REFERENCES

[1] D.L. Donoho, "*Compressed Sensing*," IEEE Transaction on Information Theory, vol. 52, pp. 1289-1306, Apr. 2006.

[2] E.J. Candes, J. Rombergand T. Tao, "*Robust uncertainty principles:exact signal reconstruction from highly incomplete frequency information*," IEEE Transaction Information Theory, vol. 52, pp. 489-509, Feb. 2006.

[3] S. Mallat and Z. Zhang, "*Matching Pursuits with time-frequency dictionaries*," IEEE Transaction on Signal Processing, vol. 41, pp. 3397-3415, Jul. 1993.

[4] R.G. Baraniuk and M.F. Durate, "*Model-based compressive sensing*," IEEE Transactions on Information Theory, vol. 56, pp. 297-312, Apr. 2010.

[5] K. O. W. Group, *The OpenCL Specification*, Khronos Group, Jun. 2011.

[6] A. Mushi, B. Gaster, and T.G. Mattson, *OpenCL Programming Guide*, Jul. 2011.

[7] A. Majumdar, N. Krishnan, and S.B. Pillai, "*Extinctions to orthogonal Matching Pursuit for Compressed Sensing*," Indian Institute of Technology, Signal Processing, Jun. 2010.

[8] M. Elad, Sparse and Redundant Representations: *From Theory to*

Applicationsin Signaland Image Processing, Springer, Haifa, Israel, 2010.

[10] E. J. Candes, "*The restricted isometry property and its implications for compressed sensing,*" Comptes Rendus Mathematique, vol. 346, pp.589–592, May 2008.

[11] E.J. Candes and T. Tao, "*Near-optimal signal recovery from randomprojections: universal encoding strategies,*" IEEE Transactions on Information Theory, vol. 52, pp. 5406–5425, Dec. 2006.

[12] E.J. Candes and T. Tao, "*Decoding by linear programming,*" IEEE Transactions on Information Theory, vol. 51, pp. 4203–4215, Dec. 2005.

[13] D.L. Donoho and X. Huo, "*Uncertainty principles and ideal atomic decomposition,*" IEEE Transactions on Information Theory, vol. 47, pp.2845–2862, Nov. 2001.

[14] Justin Romberg and Michael Wakin, "*Compressed Sensing: A Tutorial,*" IEEE Statistical Signal Processing Workshop, Aug. 2007.

[15] M. Stojnic, F. Parvaresh, and B. Hassibi, "*On the reconstruction of block-sparse signals with an optimal number of measurements,*" IEEE Transaction Signal Processing, vol. 57, no. 8, pp. 3075–3085,Aug. 2009.

[17] Y. Eldar and M. Mishali, "*Robust recovery of signals from a structured union*

of subspaces, ”IEEE Transactions on Information Theory, vol. 55, no. 11, pp. 5302–5316, Nov. 2009.

[18] J. Tropp and A.C. Gilbert, *“Signal recovery from partial information via orthogonal matching pursuit,”* IEEE Transactions on Information Theory, vol. 53, no. 12, pp. 4655-4666, 2007.

[19] T. Blumensath and M. E. Davies, “*Sampling theorems for signals from the union of finite-dimensional linear subspaces,”* IEEE Transactions on Information Theory, vol. 55, no. 4, pp. 1872–1882, Apr. 2009.

[20] AMD Inc., *http://developer.amd.com/sdks/AMDAPPSDK/assets/*

AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf,AMD Accelerated Parallel Processing OpenCL Programming Guide.

[21] B.R. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012.

[22] AMD,http://developer.amd.com/libraries/appmathlibs/pages/default.aspx, Accelerated Parallel Processing Math Libraries, 2012.

[23] Vienna University of Technology, *http://viennacl.sourceforge.net*,ViennaCL

APPENDIX

OMP OPENCL CODE

```
//#include <boost/numeric/mtl/mtl.hpp>

#include <iostream>

#include <cmath>

#include <cstdlib>

//#include <vector>

#include <viennacl/scalar.hpp>

#include <viennacl/vector.hpp>

#include <viennacl/matrix.hpp>

#include <viennacl/matrix_proxy.hpp>

#include <viennacl/linalg/norm_2.hpp>

#include <viennacl/linalg/norm_1.hpp>

#include <viennacl/linalg/prod.hpp>

#include <viennacl/linalg/inner_prod.hpp>

#include <viennacl/linalg/qr.hpp>

#include "viennacl/linalg/lu.hpp"

#include "viennacl/traits/size.hpp"

#include <boost/numeric/ublas/vector.hpp>

#include <boost/numeric/ublas/matrix.hpp>

#include <boost/numeric/ublas/io.hpp>

#include <boost/numeric/ublas/storage.hpp>

#include <boost/numeric/ublas/matrix_proxy.hpp>

#include "viennacl/ocl/device.hpp"

#include <viennacl/ocl/forwards.h>

#include "viennacl/ocl/backend.hpp"

#include "viennacl/ocl/program.hpp"

#include "viennacl/ocl/context.hpp"

#include "viennacl/ocl/kernel.hpp"

#include "CL/cl.hpp"

#include <chrono>

#include "examples/benchmarks/benchmark-utils.hpp"
```

```
floatrandgauss(float min, float max)
{
float r = (float)rand() / (float)RAND_MAX;
return min + r * (max - min);
}
const char * argmax =
"__kernel void argmax(               \n"
"         __global  float * mat1,\n"
"         __global  float * vec2,\n"
"         __global  float * newmat,\n"
"         __global  float * indcol,\n"
"            unsignedint j,    \n"
"            unsignedint component,    \n"
"            unsigned       int feature)   \n"
" { float y=0; \n"
"  float v3=0;  \n"
"unsignedconstintgid= get_global_id(0);\n"
"for(unsigned int z=0;z<component;z++){\n"
"  for (unsigned inti= gid; i< feature; i += get_global_size(0)){\n"
"     v3 += vec2[i]*mat1[(z*feature)+i];}\n"
"       if(y<fabs(v3))\n"
"      {      y=v3;     \n"
"                        indcol[gid+(j)]=z;\n"
" for (unsigned int n= gid; n < feature; n += get_global_size(0)){\n"
"      newmat[((j*feature)+n)]=mat1[((z*feature)+n)];}}\n"
"}};\n";
//typedefstd::vector<viennacl::ocl::platform >platforms_type;
//typedefstd::vector<viennacl::ocl::device>devices_type;
//typedefstd::vector<cl_device_id>cl_devices_type;
//randgauss function for dictionary
```

```
int main(){

viennacl::ocl::set_context_device_type(0, viennacl::ocl::gpu_tag());

std::vector<viennacl::ocl::device> devices = viennacl::ocl::current_context().devices();

viennacl::ocl::current_context().switch_device(devices[0]);

Timer timer;

//std::cout<<viennacl::memory_types()<<std::endl;

std::cout<<viennacl::ocl::current_device().info() <<std::endl;

//evices_.push_back(devices[0]);

//intlast_nf=128;

intstart_sparse=2;

int sparse=0;

for (int benchmark=0;benchmark<1;benchmark++){

constintn_feature=1024;/*2*(last_nf); */                    //measurments

sparse+= start_sparse;

start_sparse=sparse;

constintn_component=8192;

srand (time(NULL));

typedefviennacl::matrix<float, viennacl::column_major>VCLMatrixType;

typedefviennacl::vector<float>VCLVectorType;

boost::numeric::ublas::vector<float>
        landa(sparse),indice_cpu(sparse),m_cpu(n_feature),signal_cpu(n_component),v_cpu(n_featu
re);

boost::numeric::ublas::vector<int>                              indcol(sparse);

viennacl::scalar<float>

thetha(0),eps(0),scalarh(1),alpha(0);

VCLVectorType                                                         indcol_gpu(sparse),/*

tuple(n_feature)*/realgama(sparse),error(n_component),vcl_b(n_feature),z(1),indice(sparse),m(n_feat
ure),signal(n_component),v(n_feature),residual(n_feature),gama(n_feature);

VCLMatrixType  help(0,n_component),dictionary(n_feature,n_component),
new_dictionary(0,n_component),tuple(n_feature,1);

boost::numeric::ublas::matrix<float>
        dictionary_cpu(n_feature,n_component), new_dictionary_cpu(0,n_component);
```

```
//generate dictionary
for (int i1 = 0; i1 < n_feature;i1++) {
for (int i2 = 0; i2 < n_component;i2++) {
        dictionary_cpu( i1, i2)= randgauss(-10,10);
        }
        }
        v.clear();
        boost::numeric::ublas::scalar_value<float> h1;
        for (int i1 = 0; i1 < n_component;i1++){
        v_cpu=(boost::numeric::ublas::column(dictionary_cpu, i1));
        h1=(boost::numeric::ublas::norm_2(v_cpu));
        v_cpu=(1/h1) * v_cpu;
        (boost::numeric::ublas::column(dictionary_cpu, i1))= v_cpu;
        }
        timer.start();
        viennacl::copy(dictionary_cpu,dictionary);
        viennacl::ocl::get_queue().finish();    //wait for copy operations to finish.
        //std::cout<<timer.get() <<std::endl;
        viennacl::backend::finish();
        //std::cout<<"Dictionary:"<<dictionary<<std::endl;


        //dictionary end
        //generate sparse signal
        std::fill(signal_cpu.begin(),signal_cpu.end(),0);
        for (inti=0;i<sparse;i++){
        intindx=(std::rand()%n_component);
        indice(i)=indx;
        signal_cpu(indx)= randgauss(-20,20);
        }
        viennacl::fast_copy(signal_cpu,signal);
```

```
viennacl::backend::finish();

//std::cout<<"Random signal:"<<signal_cpu<<std::endl<<"indices:"<<indice<<std::endl;

//end generate signal

//generate measurment

m = viennacl::linalg::prod(dictionary,signal);

//m.switch_memory_domain(viennacl::MAIN_MEMORY);

//std::cout<<"measurments:"<<m<<std::endl;

//end generate measurment

//orthogonal matching pusuit

// Solves [1] min || D * gamma - x ||_2 subject to || gamma ||_0 <= m

// or     [2] min || gamma ||_0         subject to || D * gamma - x || <= eps

// Parameters

// ----------

//   D, array of shape n_features x n_components

//   x, vector of length n_features

//   m, integer <= sparsity level

//   eps, float (supersedes m)


//residual

residual=m;

std::fill(indcol.begin(),indcol.end(),-1);                                          // idx

viennacl::range all_col(0,n_component);

viennacl::range all_row(0,n_feature);

viennacl::fast_copy(indcol,indcol_gpu);

eps=0;

std::cout<<"==========================="<<std::endl;

std::cout<<"OMP START"<<std::endl;

std::cout<<"==========================="<<std::endl;

auto start = std::chrono::high_resolution_clock::now();

for (int j=0;j<sparse;j++){
```

```
//viennacl::traits::resize(new_dictionary,n_feature,j+1);

new_dictionary.resize(n_feature,j+1);

//viennacl::traits::resize(help,n_feature,j+1);

help.resize(n_feature,j+1);

viennacl::scalar<float>  theta(0);

for (inti=0;i<n_component;i++){

viennacl::range    row(i,i+1);

viennacl::range    col(i,i+1);

tuple=viennacl::project(dictionary,all_row,col);



tuple,static_cast<cl_uint>(i),static_cast<cl_uint>(direction),static_cast<cl_uint>(tuple.size())
));

//std::cout<<tuple<<std::endl;

tuple=viennacl::project(dictionary,all_row,col);

// timer.start();

z=viennacl::linalg::prod(trans(tuple),residual);

//std::cout<<timer.get() <<std::endl;

//timer.start();

alpha=viennacl::linalg::norm_1(z);

if (theta<alpha){

theta=alpha;

//landa(j)=alpha;

indcol(j)=i;

viennacl::range    col(j,j+1);

viennacl::project(new_dictionary,all_row,col)=tuple;

tuple,static_cast<cl_uint>(j),static_cast<cl_uint>(direction),static_cast<cl_uint>(tuple.size())
));

}

}

//viennacl::ocl::program &my_prog = viennacl::ocl::current_context().add_program(argmax,
"argmax");
```

```
//viennacl::ocl::kernel &my_kernel = my_prog.add_kernel("argmax");

//viennacl::ocl::enqueue(my_kernel(dictionary,
residual,new_dictionary,indcol_gpu,static_cast<cl_uint>(j),static_cast<cl_uint>(n_componen
t),static_cast<cl_uint>(n_feature)));

//least square slove

vcl_b = m;

help=new_dictionary;

//std::cout<<"help"<<help<<std::endl;

std::vector<float>hybrid_betas = viennacl::linalg::inplace_qr(help,256);

// compute modified RHS of the minimization problem:

// b := Q^T b

viennacl::linalg::inplace_qr_apply_trans_Q(help, hybrid_betas, vcl_b);

viennacl::range                                          vcl_range(0,j+1);

viennacl::matrix_range<VCLMatrixType>              vcl_R(help,          vcl_range,
vcl_range);

viennacl::vector_range<VCLVectorType>              vcl_b2(vcl_b, vcl_range);



// Final step: triangular solve: Rx = b'.

// We only need the upper part of A such that R is a square matrix



viennacl::linalg::inplace_solve(vcl_R, vcl_b2, viennacl::linalg::upper_tag());

//new residual

gama=viennacl::linalg::prod(new_dictionary,vcl_b2);

residual=m-gama;

/*scalarh=viennacl::linalg::inner_prod(residual,residual);

if(eps>=scalarh){

break;

}

eps=scalarh.

realgama(j)=vcl_b2(j);

}
```

```
std::cout<<"============================"<<std::endl;

std::cout<<"OMP STOP"<<std::endl;

std::cout<<"============================"<<std::endl;

//error of method

auto finish = std::chrono::high_resolution_clock::now();

std::cout<<   std::chrono::duration_cast<std::chrono::nanoseconds>(finish-start).count()   <<
"ns\n";

//unsparse the measurment

//error.clear();

for (inti=0;i<sparse;i++){

error(indcol(i))=realgama(i);

}

std::cout<<"Dimension="<<n_feature<<"*"<<n_component<<std::endl;

//std::vector<float>error_cpu(n_component);

//viennacl::copy(error,error_cpu);

signal-=error;

alpha = viennacl::linalg::norm_2(signal);

std::cout<<"error in percent="<<(alpha)<<std::endl;

std::cout<<"check"<<indcol<<std::endl<<indice;

}

return EXIT_SUCCESS;

}
```