

OpenCL for Computationally Intensive Applications in Radar Signal Processing on Intel Processors

Savio George

11008328

May 10, 2017

Under the Guidance of:

Dr. Pierre Bayerl

AIRBUS

Submitted to:

Prof. Dr. Achim Gottscheber



**Department of Engineering
SRH Hochschule Heidelberg
Germany**

Abstract

The intention of this paper is to prepare Intel processor based systems capable to run applications using the OpenCL framework on a Linux platform. A basic study of the performance capabilities which could be achieved by making use of potential parallelism through the OpenCL framework on Intel integrated GPU and multi core CPU is also a part of this work. OpenCL implementations of Fast Fourier Transform, and matrix multiplication provided by open source Libraries are made use for this purpose.

Table of Contents

1. Introduction	4
2. OpenCL	5
2.1 What is OpenCL	5
2.2 OpenCL Architecture	5
2.3 OpenCL ICD	10
3. System Preparations	11
3.1 Intel 6th generation processor: i5-6500 CPU @ 3.20 GHz	11
3.2 Intel 4th generation processor: i7-4700EQ CPU @ 2.40 GHz	16
3.3 Intel 3rd generation processor	19
4. Performance Analysis Set up	19
4.1 Architecture	20
4.2 Implementation prerequisites	21
4.3 clFFT and clBLAS	22
5. Algorithms and Evaluation	23
5.1 Fast Fourier Transform	23
5.2 Matrix Multiplication	30
6. Current Status and Future Work	37
7. References	38
8. Software Downloads links	39
9. Appendix	39

1. Introduction

Radar signal processing systems are growing in its capabilities. This growth has also brought in the necessity for increased computational power. In our radar signal processing system, CPUs are employed for handling the signal processing computations. For the next generation radar systems CPUs alone might prove to be insufficient due to its limitations in signal processing capabilities and throughput. This gave birth to the idea of using CPU accelerators, such as graphics processing units (GPUs) to share the heavy processing load with the CPU and provide the necessary throughput. Our current radar signal processing system already has an integrated GPU on board which is left unutilized. The main focus of our work is to make use of this unutilized GPU capable of running radar signal processing applications. Radar signal processing algorithms such as Pulse Compression and Doppler filtering makes use of mathematical algorithms Fast Fourier Transform and matrix multiplication respectively. These mathematical algorithms often require same operation to be carried out on multiple data. To note, GPU have a single instruction multiple data (SIMD) approach and in such cases this could prove to be ideal. The work is divided into three major parts -system preparation, performance analysis setup and algorithms and its evaluation. Under system preparation the software installation procedures necessary to develop and execute OpenCL applications on Intel 4th and 6th generation processors with integrated graphics are discussed, under the performance analysis setup we discuss how we developed an environment to run and test these applications and under algorithms and evaluation we explain in detail about the OpenCL application developed to execute Fast Fourier Transform and matrix multiplication and the evaluation results obtained using the performance analysis setup.

2. OpenCL

2.1 What is OpenCL

Open Computing language also called as OpenCL is an open standard maintained by a non-profit technology consortium Khronos Group. It is a frame work which allows us to write programs that execute across heterogeneous platforms. A heterogeneous platform is nothing but a platform that uses more than one kind of processors such as, Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Field-Programmable Gate Arrays (FPGA) or other hardware accelerators. Heterogeneous computing is all about exploiting the available computer resources to maximize the performance. Thus, OpenCL provides us low level hardware abstraction and a programming model which run our code on various kinds of hardware architectures. In our case we focus on CPUs and GPUs. GPUs are processors essentially designed for graphics processing however its parallel architecture makes is it also suitable for data parallel processes. For developing software using the OpenCL framework the following two tools are required.

- OpenCL Compiler
- OpenCL Runtime Library

These tools typically come along with OpenCL SDK provided by the vendor in our case Intel.

2.2 OpenCL Architecture

The OpenCL framework can be described using the following models:

- Platform Model
- Programming Model

- Memory Model

Platform Model

From a hardware perspective an OpenCL platform requires a host connected to its devices. Typically CPU is the host and the compute devices can be any other kind of processor or a hardware accelerator, in our case it is GPU. Each device will have compute units and is further divided into processing elements as shown in the Figure 1.

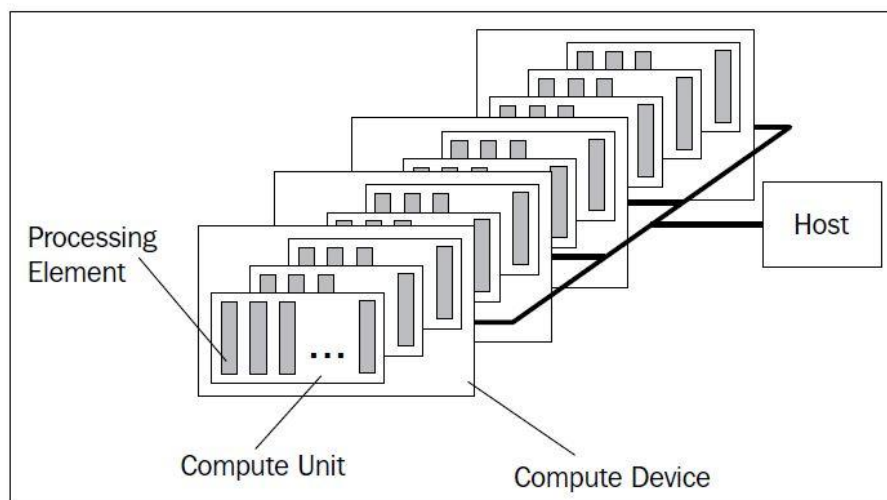


Figure 1: OpenCL Platform Model [1]

The definition of the compute unit is different depending on the hardware vendor. Consider Figure 2 which shows the hardware architecture of an Intel processor graphics.

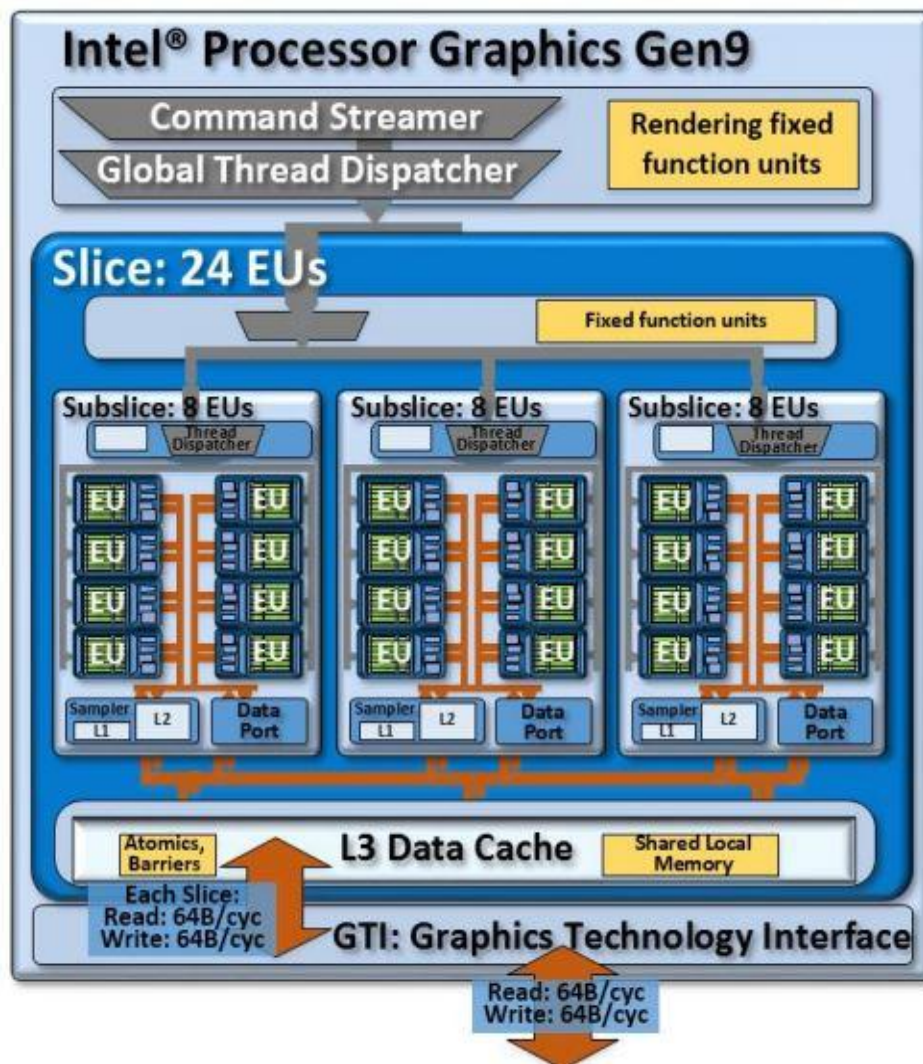


Figure 2: Intel Processor graphics [2]

Here you can find three subslices each with eight execution units and each execution units can handle a maximum of seven SIMD (Single Instruction Multiple Data) threads. Mapping it to the compute device in the Figure 1 the execution units represents the compute units and the seven SIMD threads represent the processing elements (PE) [8], [2]. The actual computation happens on the processing elements which means the kernel gets executed on the processing elements

Programming Model

The kernels and the host program are the two main execution units in an OpenCL program [1]. The host program runs on the Intel CPU and the Kernel program runs on the device.

It is the host program which steers the entire application and decides on which devices the kernel code has to be executed. The host code can be briefly divided into the following steps.

Step 1. Get the available platforms, Select the platform and choose the device.

Step 2. Create a context. A context is the run time link between our device and the platform.

Step 3. Create a command queue. It holds the commands to be executed by the device. Command to execute the kernels, reading and writing to memory objects are. Each command queue will be only for communicating with one device.

Step 4. Compile and load the kernel to execute on our device.

Step 5. Allocate space on the device Memory and transfer the data to this memory on which the device will operate.

Step 6. Execute the kernel on the device.

Step 7. Optional: Fetch the results from the device memory.

A sample OpenCL program which implements a simple vector addition has been provided at the end of the document in the appendix section which shows the above mentioned steps in detail.

Memory Model

When it comes to OpenCL a programmer must have a clear understanding about the memory architecture to achieve the highest possible performance for his/her application. The following are the different types of memory.

Global memory: Typically the largest portion of the memory subsystem. Memory access can be coalesced, to ensure contiguous memory reads and thereby increasing the performance [1]. All the work items belonging to any of the workgroups have access to this memory (work item: it is the individual kernel execution instance; work group: It is group of work items that execute on a single compute unit. The work items belonging to a group typically execute the same kernel) `_global` keyword used in the kernel function identifies this buffer region [1].

Constant Memory: This memory region is initialized by the host. This is similar to creating an OpenCL buffer with `CL_MEM_READ_ONLY` flag. This is the region which remains constant throughout the execution time of the kernel [1].

Local Memory: Memory which is closest to the OpenCL processing element and typically small. This region is shared between work items in a work group. `_local` keyword identifies this region [1].

Private Memory: Memory region allocating local variables in the kernel code. Each processing element will have private memory and is not visible to other work items.

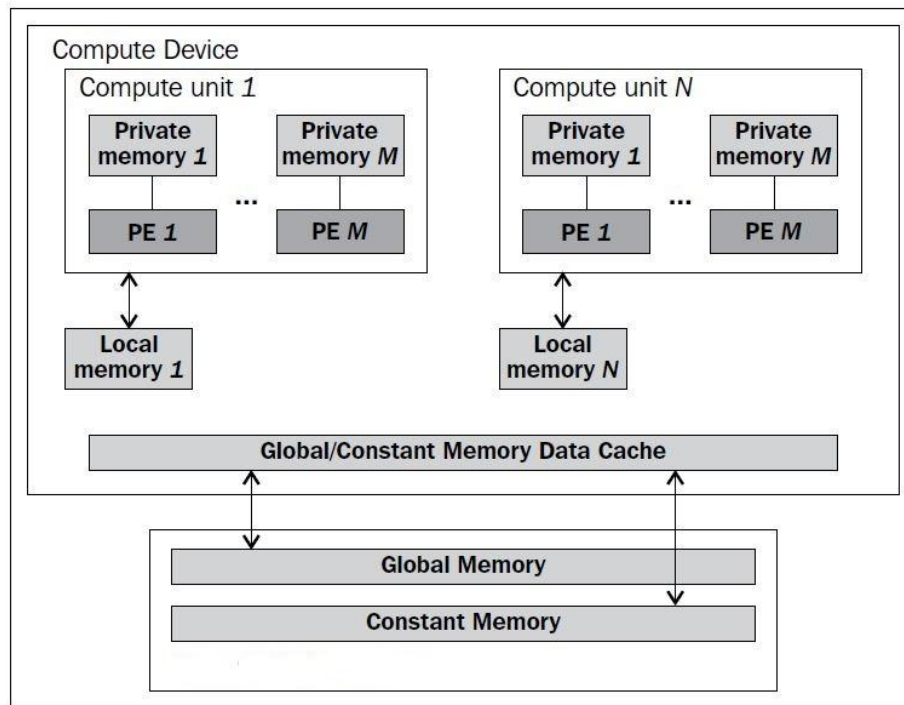


Figure 3: OpenCL Memory Model [1]

The Figure 3 gives us a visual representation of the previously described memory sections in the OpenCL framework. It is up to the Application programmer to decide how to utilize this memory layout to its best to achieve optimal performance suited for the application.

2.3 OpenCL ICD

OpenCL Installable Client Driver (ICD) is a means of allowing multiple OpenCL implementations to co-exist and applications to select between them at the runtime [1]. In other words a system can have more than one implementation of the OpenCL specifications (platforms) provided by different vendors for example: Intel, AMD or any other to be installed and reside safely without any issues. With this feature it is up to the application to choose among the multiple platforms available.

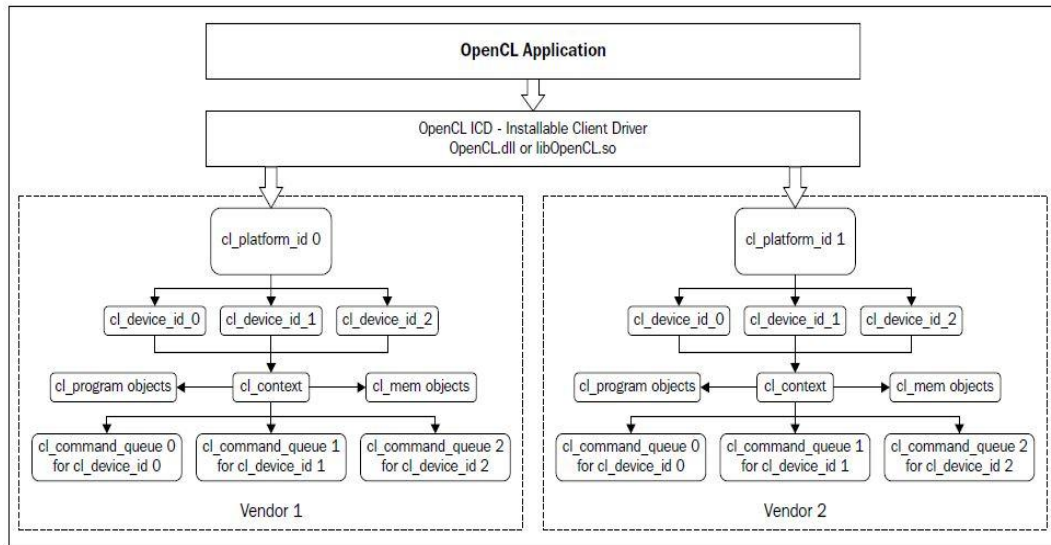


Figure 4: OpenCL ICD and different vendors [1]

At runtime this ICD shared library (OpenCL.dll in Windows and libOpencl.so in Linux) will query the registry and load the appropriate shared library as selected by the application [1]. If an application wants to use both platforms available the application developer can create multiple contexts to do so.

3. System Preparations

In this section we discuss how to prepare an Intel processor based system to be capable to run OpenCL Applications. For this purpose we tried with the possibilities to prepare three different systems with three different processor generations. All the three Intel Processors had integrated graphics.

3.1 Intel 6th generation processor: i5-6500 CPU @ 3.20 GHz

Stage 1. Installing the CentOS Operating System

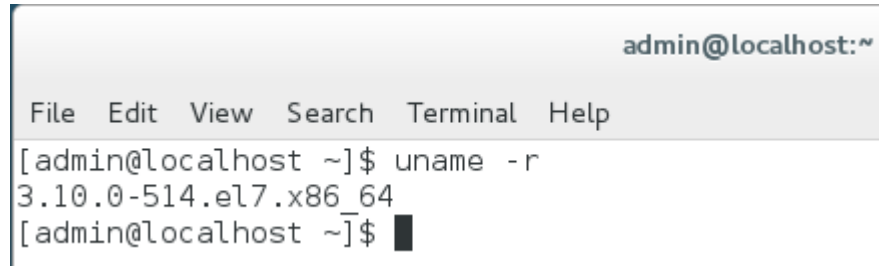
- The operating system installed is CentOS Linux release 7.3.1611(Core).
- The system does not have access to the internet and therefore installations had to be done offline.

- The operating system was downloaded on another computer connected over the internet from the link specified under the software download links [10]. The “everything ISO” was chosen
- The operating system was installed via a USB memory stick made bootable using the dd command.
- Note: For some reason the bootable USB created using Unetbootin was unable to install the CentOS onto the system.

Stage 2. Installing the OpenCL drivers and 4.7 Kernel Build and patch

- Download the Intel OpenCL SDK , the drivers and the library run time packages from the link specified under the software download links [11].
- Follow the “Installation instructions” document.
- For the installation of the drivers follow the steps under **“CentOS 7.2 with Linux 4.7 Kernel-CentOS Installation and preliminary Setup”** in the Installation instructions document.
- For installation of any missing Linux package dependencies please first search the CentOS bootable USB before installing it from anywhere else, refer **“Note: 1”** provided in the appendix for detailed information
- The **“CentOS – Patch and Compile Linux Kernel 4.7”** steps are to be followed.
- Install the package: elfutils-libelf-devel from the CentOS bootable USB (refer **“Note: 1”** for detailed information)
- Also set the clock of the system before following the step 9 in **“CentOS – Patch and Compile Linux Kernel 4.7”** in the “Installation instructions” document.
- To load the new 4.7 Kernel reboot the computer and select the kernel to be loaded appearing on the screen.

- Later you can use the command `uname -r` in the command line as shown in Figure 5 to know which kernel version was loaded.

A terminal window titled 'admin@localhost:~' with a menu bar (File, Edit, View, Search, Terminal, Help). The command `uname -r` is entered and executed, resulting in the output `3.10.0-514.el7.x86_64`. The prompt `[admin@localhost ~]$` is shown again at the end of the line.

```
admin@localhost:~  
File Edit View Search Terminal Help  
[admin@localhost ~]$ uname -r  
3.10.0-514.el7.x86_64  
[admin@localhost ~]$
```

Figure 5: Kernel version loaded

Side Notes:

- An error occurred “no rule to make target” it was because I unknowingly patched the kernel twice by running the step 5 under “**CentOS – Patch and Compile Linux Kernel 4.7**” more than once ; in such a case it is best to delete the extracted files and redo from step 4.
- Another situation is the build of the kernel requires a lot of memory and sufficient disk space.

Stage 3. Installing the Intel OpenCL SDK

- Download the Intel SDK for Linux from the software download links [8].
- Execute the command `./install_GUI.sh`. In case some packages required are missing you may get the following message as shown in Figure 6

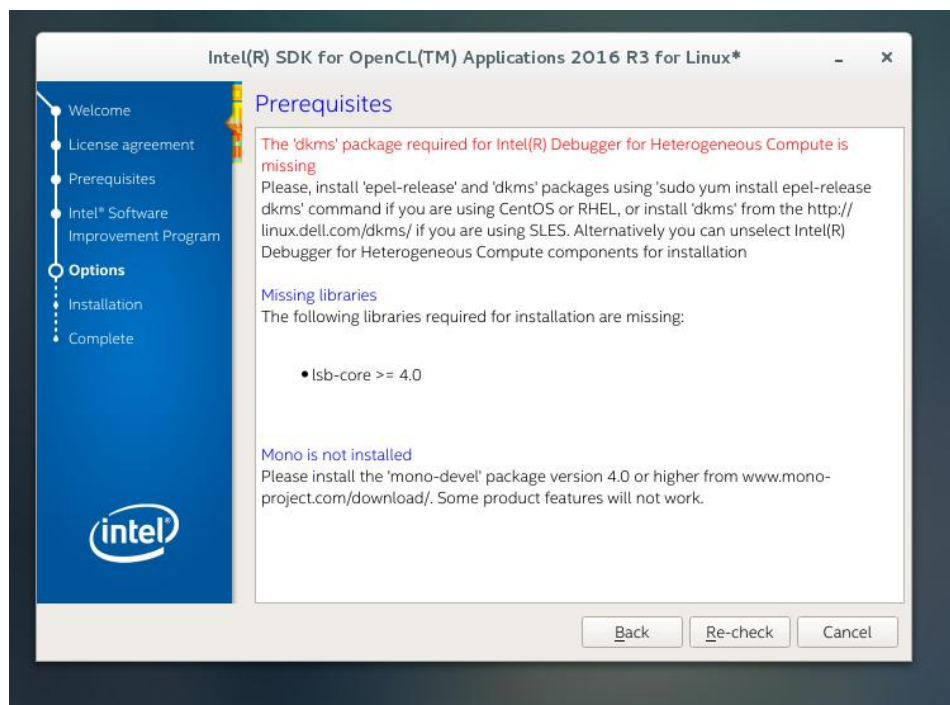


Figure 6: Intel OpenCL SDK dependencies

- Most of the packages required by the Intel SDK are already part of the CentOS repository and is available in bootable USB. So I will strongly recommend installing it from the USB rather than relying on any other external sources (refer “**Note: 1**” in the Appendix for detailed information).
- The packages required are:
 1. Cmake (Available in the Bootable USB)
 2. DKMS (Not available in the Bootable USB). Please download from the software download links [12].
 3. EPEL (Not available in the Bootable USB). Please download from the software download links [13].
 4. Lsbcore (Available in the Bootable USB)
 5. Mono-develop (Not available in the Bootable USB) However it is not essential. Please refer to the conversation [4] under references for more details.

6. OpenSSL and OpenSSH (Available in the Bootable USB) .

Side Notes:

- If we try to install the Intel complete OpenCL SDK package. This will ask for the kernel 4.7 sources and headers. For this navigate into the folder where you have extracted the kernel patch to build it (According to the installation document it will be ("~/intel-openccl /linux-4.7")) and run the command "**make modules_install**"
- If we install the complete package, two OpenCL platforms will be available as shown in "**Snapshot [2]**" in the appendix.

Stage 4: To compile and link an OpenCL Application

After the successful installation of the SDK while compiling and linking an OpenCL application issues might occur with the linking of the OpenCL API. This can be resolved by the following steps that creates a symbolic link

- \$ sudo ln -s /opt/intel/openccl-1.2-sdk-6.3.0.1904/lib64/libOpenCL.so /usr/lib/libOpenCL.so
- \$ sudo ldconfig

To compile and link a program

- g++ -o DeviceTest DeviceTest.cpp -lOpenCL

If the openccl header files are not being included specify the path to the headers

- g++ -I {path to the include} -o DeviceTest DeviceTest.cpp -lOpenCL

Now all the necessary tools have been installed to build and develop an OpenCL application. However there are some additional tools which are provided by Intel to make OpenCL

development more convenient and for this they suggest the use of Eclipse IDE

The software downloads links [7], and under references [7], [15] will give more information on these tools and how to install and integrate it with the eclipse IDE.

3.2 Intel 4th generation processor: i7-4700EQ CPU @ 2.40 GHz

Since this is a 4th generation based processor (Haswell) we cannot use the same Intel OpenCL SDK package used for the 6th generation as the package does not support the Haswell processors. The idea provided by Concurrent Technologies was to make use of the 2016 version of the Intel Media Server Studio which had OpenCL in it. Concurrent technologies had already tried this by installing OpenCL using the Media Server Studio 2016 and they said they were successful. Similar to this Intel also suggested to follow the same method suggested by concurrent technologies refer to the discussion [3or4] under references. Hence we follow the steps listed down in the document “Installation of Media Server Studio 2016 on the TR B12 v5” provided by concurrent technologies.

However the installation procedures followed here were not exactly the same as provided in the document due to some difficulties faced. We had to make some workarounds to get the OpenCL installed and working.

The workarounds and solutions which are mentioned in this document cannot be claimed to be the best possible methods as it was an experiment to get things running and we are not sure if this could have any future consequences. Nevertheless the workarounds seemed to fit and we were able to successfully install OpenCL packages and develop and run applications.

We list down the issues and the alternate steps taken here:

Stage 1. Installing the CentOS Operating System

- Instead of the Red Hat Enterprise Linux 7.1 operating system we went ahead with the same Centos Linux release 7.3.1611(Core) used for the 6th generation.

Stage 2. Downloading the Media Server Studio and installing the extra packages

- The Media server studio was downloaded as stated in the document provided
- The document from concurrent technologies states about some extra packages to be installed from the internet. Instead we found these packages available within bootable USB created for CentOS and we installed the packages from the USB.

Stage 3. Installing the Media Server Studio

- The following changes were made to the install_sdk_CentOS.sh shell script as shown in the Figure 7 because we use the packages on the USB repository. The highlighted portion is the change.

```
# install prerequisite packages
yum --disablerepo=* --enablerepo=c7-media -y -t groupinstall "Development Tools"
yum --disablerepo=* --enablerepo=c7-media -y -t install kernel-headers kernel-devel bc
yum --disablerepo=* --enablerepo=c7-media -y -t install wget bison ncurses-devel hmaccal
xmlto audit-libs-devel binutils-devel elfutils-devel elfutils-libelf-devel newt-devel nur
```

Figure 7: Changes made to install_sdk_CentOS.sh

- The steps mentioned after the following command were not executed
cct@cctdemo:~/work/MediaServerStudioEssentials2016
/SDK2016Production16.4.4/CentOS\$ sudo
./install_sdk_CentOS.sh

Due to the following error shown in Figure 8

```
[admin@localhost ~]$ cd /home/cct/work/MediaServerStudioEssentials2016/SDK2016Pr
duction16.4.4/CentOS
[admin@localhost CentOS]$ ./build_kernel_rpm_CentOS.sh
Error... Not installed intel-linux-media-devel rpm package!
[admin@localhost CentOS]$
```

Figure 8: Error Message when trying to build the kernel

- We are not sure if not building and patching the kernel could have any future consequences. (Later we also noticed some issues during the testing of the OpenCL application for matrix multiplication. Details of which can be found under “**Snapshot [1]**” in the Appendix.)
- Executing the `./install_sdk_CentOS.sh` script gives us the following error as shown in Figure 9

```
Package python-devel-2.7.5-48.el7.x86_64 already installed and latest version
Package zlib-devel-1.2.7-17.el7.x86_64 already installed and latest version
Nothing to do
error: Failed dependencies:
    libdrm_amdgpu.so.1()(64bit) is needed by (installed) mesa-dri-drivers-11.2.2-2.20160614.el7.x86_64
    pkgconfig(libdrm) >= 2.4.60 is needed by (installed) mesa-libGL-devel-11.2.2-2.20160614.el7.x86_64
    pkgconfig(libdrm) >= 2.4.60 is needed by (installed) mesa-libGL-devel-11.2.2-2.20160614.el7.x86_64
[root@localhost CentOS]#
```

Figure 9: Result of `install_sdk_CentOS.sh`

- We went ahead by not installing the complete package instead only the OpenCL packages and the other packages which did not throw any dependency error.

Stage 4. To compile and link an OpenCL Application

After the successful installation of the openCL packages while compiling and linking an OpenCL application, issues did occur with the linking of the OpenCL API. This can be resolved by the following steps that creates a symbolic link

- `$ sudo ln -s /opt/intel/oneapi/6.3.0.1904/lib64/libOpenCL.so /usr/lib/libOpenCL.so`
- `$ sudo ldconfig`

3.3 Intel 3rd generation processor

We also tried to install the OpenCL SDK for the Intel 3rd generation processor. The installation was successful however Intel had officially removed the support for the older generation processors starting from 4th and below and therefore using the OpenCL we will only have access to the CPU and not to the GPU

If we still require to use these generations of processors we could make use of older versions of Media Server Studio (as you had seen here for the Intel 4th generation processor: i7-4700EQ CPU @ 2.40 GHz) or Beignet which is an open source implementation of the OpenCL specification. The discussion mentioned under the references [4] could be useful piece of information regarding this. To install the Beignet use the software download links [4]. To build Beignet it requires LLVM and the clang compilers the prebuilt binaries could be obtained from software download links [5] or [6]. The necessary steps to build the beignet are provided in its readme file. Beignet also requires a good amount of memory to carry out its build process. We got the Beignet installed onto the system however we did not carry out any tests to see how an OpenCL application performs on it.

4. Performance Analysis Set up

Basically our intention was to first set up the system capable to run an OpenCL application on the GPU. Now here we create an environment using which we can evaluate the OpenCL

application running on the GPU for its Memory read time, write time, Kernel execution time and discrepancy. Intel® HD Graphics 530 which is the integrated graphics on the Intel 6th generation processor: i5-6500 CPU and the Intel® HD Graphics 4600 the integrated graphics in Intel 4th generation processor: i7-4700EQ CPU is our target hardware's.

For these purpose two mathematical algorithms Fast Fourier Transform and the matrix multiplication are our choices. These algorithms are key ingredients in some radar signal processing algorithms such as Pulse Compression and Doppler Filtering.

4.1 Architecture

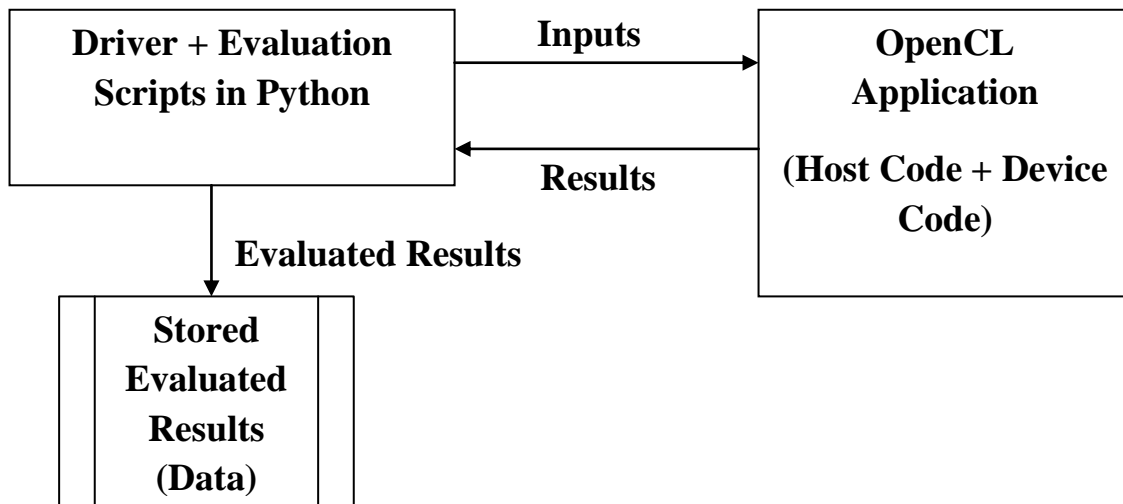


Figure 10: Performance Analysis Set up Stage 1 Block Diagram

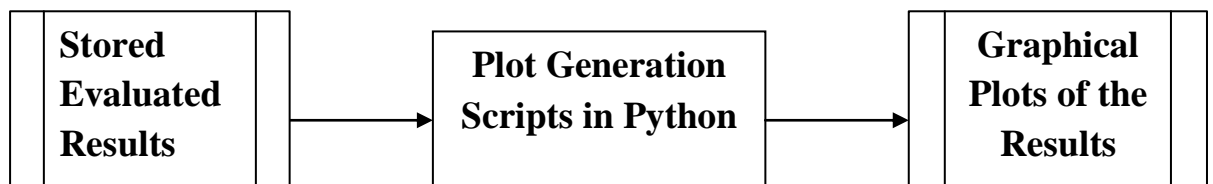


Figure 11: Performance Analysis Set up Stage 2 Block Diagram

The architecture is divided into 2 stages. In stage 1 the “Driver + Evaluation Scripts in Python” starts the “OpenCL Application” by feeding it with the required inputs. The inputs necessary for the OpenCL application to run are generated by the scripts using random number generator function. The OpenCL application does the computation on its device and gives back the results to the “Driver + Evaluation Scripts” block. This block then evaluates the results obtained from the OpenCL application for its discrepancy and the results are then stored. In stage 2 the “Stored Evaluated results” are used by the “Plot Generation Scripts in python” to generate the graphical plots for evaluation.

4.2 Implementation prerequisites

For the realization of the above software architecture we require many additional packages installed on the system.

1. Cmake Version 2.8 (Available in the bootable USB)
2. Python 2.7(Available in the bootable USB)
3. Numpy(Available in the bootable USB)
4. C++ boost Libraries Version 1.63.0 from software download links [9].

Please make sure that the numpy is installed or else the boost numpy will not build and the numpy libraries will not be created.

To build and install the C++ boost libraries:

1. Run “./bootstrap.sh”
2. Run “./b2”
3. Run “./b2 install --includedir=/usr/include --libdir=/usr/lib

4.3 clFFT and clBLAS

We have made use of the clFFT and the clBLAS open source libraries for the performance analysis. clFFT library is the OpenCL implementation of the FFT algorithm. Its routines are optimized for the AMD graphics. However it is functional across other CPU and GPU devices [3]. Similarly clBLAS is also an OpenCL implementation of the BLAS libraries we make use of the matrix multiplication sub routines included in it.

Library implementation was chosen as our first choice for the performance analysis test because it naturally relieves the user/programmer from several tasks such as writing, maintaining and optimizing the code themselves. Apparently the best performing software will be the ones written specifically in regards to the underlying hardware because we could optimize the kernel accordingly to fit our hardware specification and needs. But here we needed to how well these open source libraries could function on our hardware.

Both the libraries are available on the Github as sources as well as the prebuilt binaries for Linux and Windows. Either we could download the sources and build it from our system or we could just use the prebuilt binaries. For our work on the 6th generation system the clFFT libraries were built from the sources while the clBLAS libraries were obtained as prebuilt binaries (We were not able to build the binaries from the sources for more details view the conversation provided in reference [5]). On the 4th generation system both the clFFT and the clBLAS libraries we used the prebuilt binaries (for convenience, did not try building it from the sources). The necessary links are provided in the software download links [1], [2] and [3].

5. Algorithms and Evaluation

The algorithm followed by the OpenCL Applications developed to carry out the Fast Fourier transform and the matrix multiplication using the clFFT and the clBLAS libraries respectively are explained here. The evaluation results obtained by executing the application on the Intel 6th and 4th generation processors are shown in the form of graphical plots.

The results obtained from the OpenCL device is compared with results from “Numpy” for the evaluation of Discrepancy. “Numpy” is a library in python programming language for scientific computing. It has math functions to compute fft, matrix operations and much more.

For the graphical representation of the evaluation results we also made use of the box plots, which represents the samples in quartiles (first quartile (Q1) is defined as the middle number between the smallest number and the median of the data set. The second quartile (Q2) is the median of the data. The third quartile (Q3) is the middle value between the median and the highest value of the data set [10]. The other samples which do not belong to these quartiles are plotted as individual points and are called as outliers).

5.1 Fast Fourier Transform

Algorithm

Step 1: Set up the OpenCL environment- get the platform, choose the device as GPU, create the context and the command queue

Step 2: Set up the clFFT using the API `clfftInitSetupData()` and `clfftSetup()`

Step 3: Allocate the specified amount of memory on the device to transfer the data to the device

Step 4: Create a default plan for a complex FFT, Set plan parameters and Bake the plan.

Step 5: Transfer the data to the device and execute the FFT on the OpenCL device.

Step 6: Wait for the device to finish the computations. Fetch the result from the device. Compare the results from the OpenCL device to Numpy results and store the discrepancy. Also store the write and the read time (Time taken to transfer the data to the device memory and to read back the results)

Step 7: Repeat the steps 5 and 6 for a certain number of times to compute the average result.

Step 8: Destroy the plan created for the clFFT, and de-allocate the allocated memory resources. Do not forget to de-allocate and free the memory or else it may result in a error message (CL_OUT_OF_HOST_MEMORY,CL_OUT_OF_RESOURCES) and the application might hang

Step 9: Start again from step 3 now for a larger set of data.

Step 10: The application remains running until it is tested for all the set of input data's. After the application exits store the resultant evaluation data onto a file.

Evaluation Results for the Fast Fourier Transform

The application was tested for 5 different sets of data of 2^n (where $n=10, 11, 12, 13, 14$). Each set of data was tested for a 5000 times with random float values.

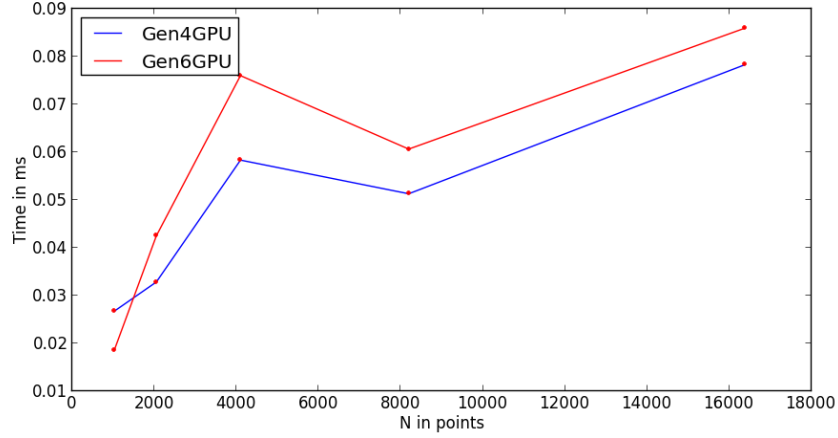


Figure 12: Kernel Execution Time

The Figure 12 is the average time taken by the device to compute the FFT. The red dots represent the 5 sets of data and the horizontal axis gives the number of elements in each set. The Gen4GPU represents the 4th generation processor and the Gen6GPU represents the 6th generation processor. We found the clFFT library is optimized to work on AMD graphics cores and perhaps due to the same reason these optimizations may not show similar performance on Intel. Please view references [16] for more details.

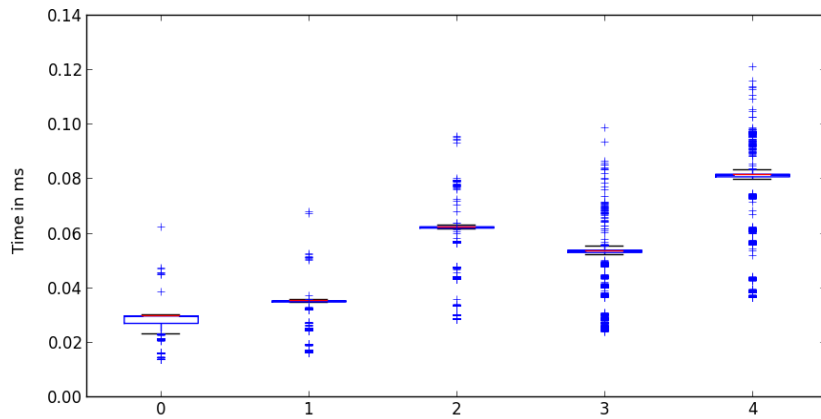


Figure 13: Kernel Execution Time 4th Generation Processor

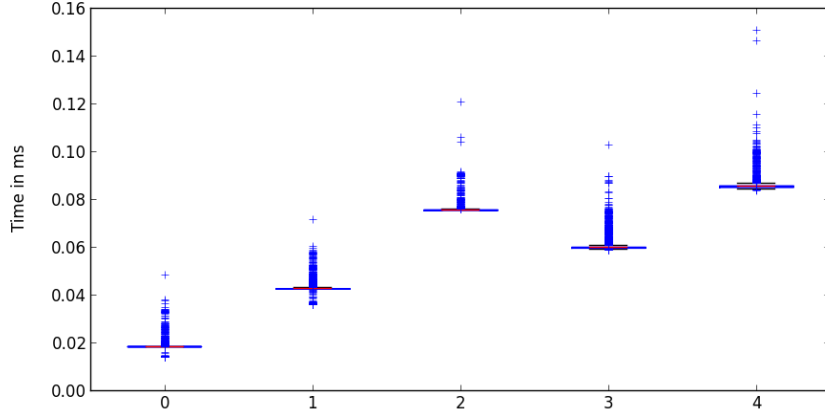


Figure 14: Kernel Execution Time 6th Generation Processor

Figure 13 and 14 are the Box Plot representation of the time taken by the device to compute the FFT on the 4th and the 6th generation processors respectively. The horizontal axis represents the 5 sets of data (2^n , where $n=10, 11, 12, 13, 14$). Each set of data was tested for a 5000 times with random float values.

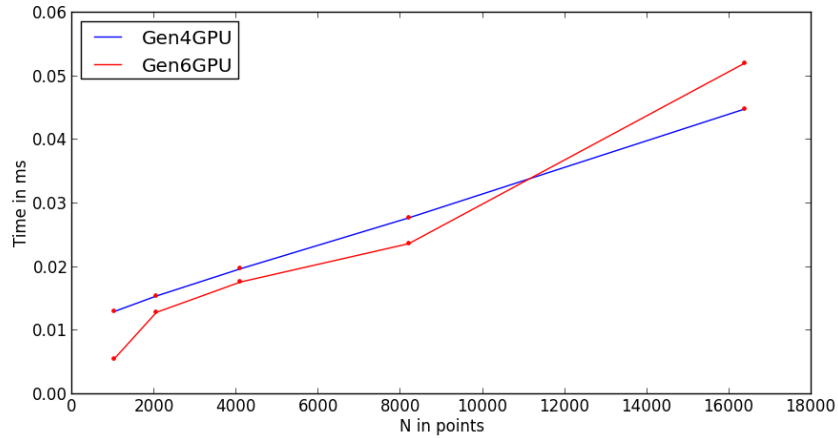


Figure 15: Read Time

Figure 15 is the average time taken by the host to read the results from the device memory. The red dots represent the 5 sets of data and the horizontal axis gives the number of elements in each set.

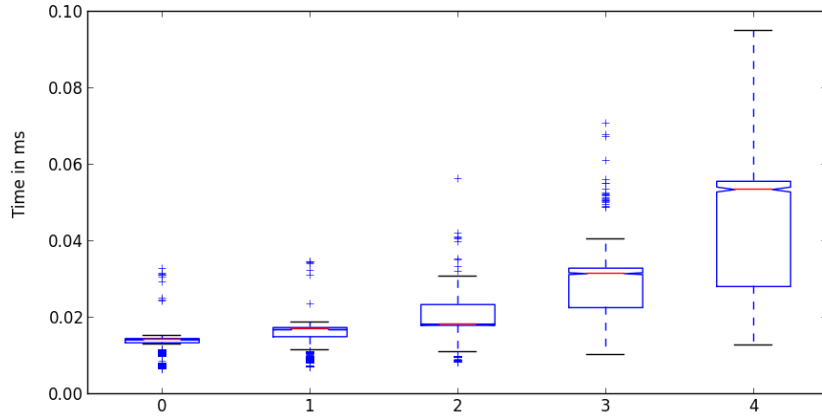


Figure 16: Read Time 4th Generation Processor

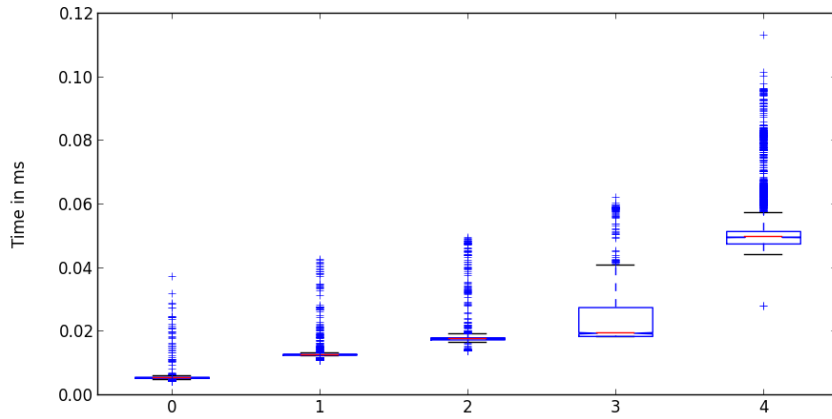


Figure 17: Read Time 6th Generation Processor

Figure 16 and 17 are the Box Plot representation of the time taken by the host to read the results from the device memory on the 4th and 6th generation processors respectively. The horizontal axis represents the 5 sets of data. The horizontal axis represents the 5 sets of data (2^n , where $n=10, 11, 12, 13, 14$). Each set of data was tested for a 5000 times with random float values.

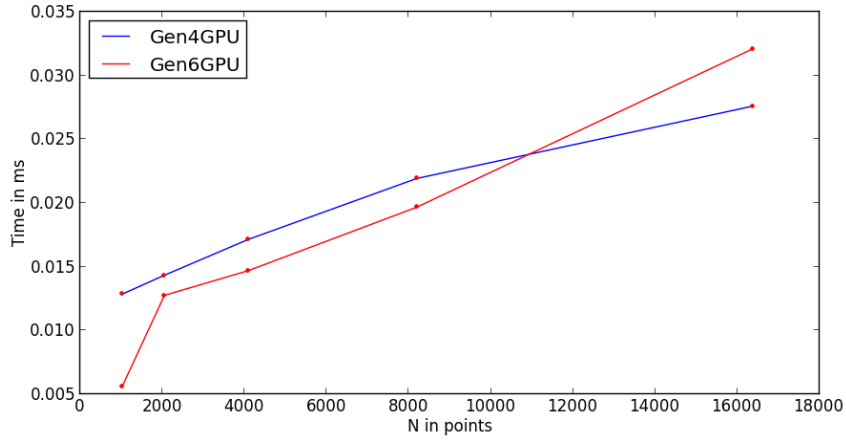


Figure 18: Write Time

Figure 18 is the average time taken by the host to transfer the data onto the device memory. The red dots represent the 5 sets of data and the horizontal axis gives the number of elements in each set.

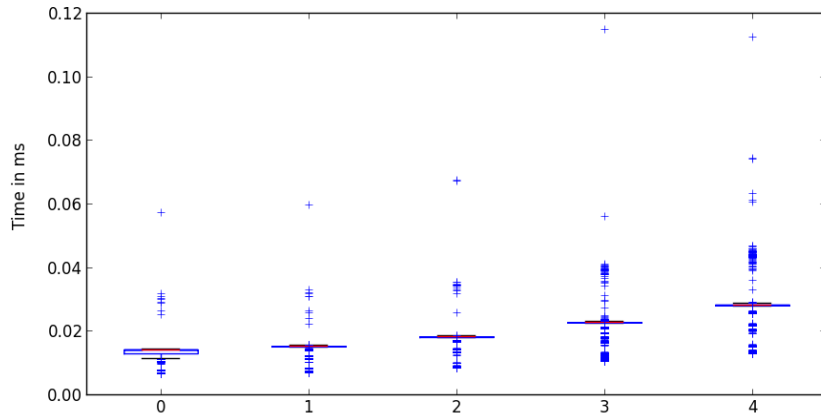


Figure 19: Write Time 4th Generation Processor

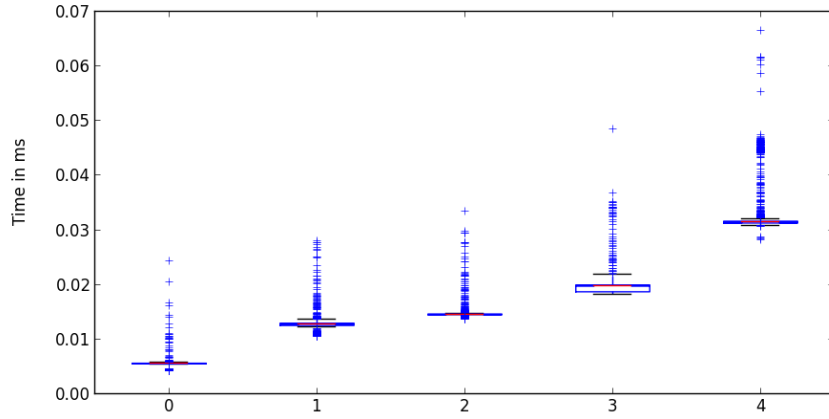


Figure 20: Write Time 6th Generation Processor

Figure 19 and 20 are the Box Plot representation of the time taken by the host to transfer the data onto the device memory on the 4th and 6th generation processors respectively. The horizontal axis represents the 5 sets of data. The horizontal axis represents the 5 sets of data (2^n , where $n=10, 11, 12, 13, 14$). Each set of data was tested for a 5000 times with random float values.

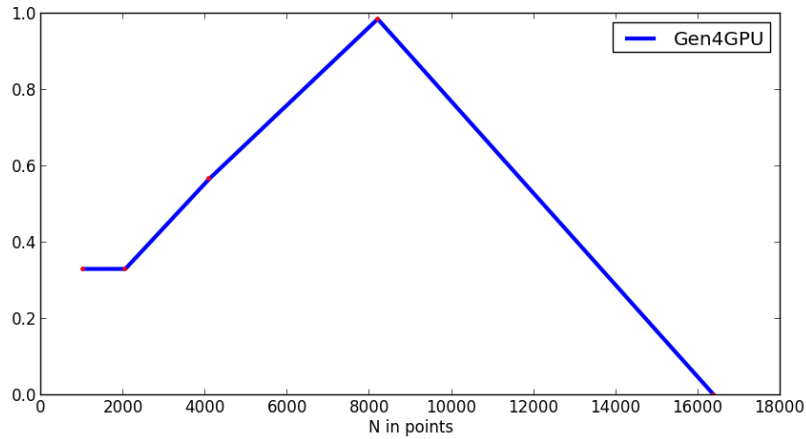


Figure 21: Discrepancy on 4th Generation Processor

Figure 21 gives us the average Discrepancy in the values computed by the 4th generation processor GPU. The discrepancy is very high and abnormal. The problem has to be

investigated.

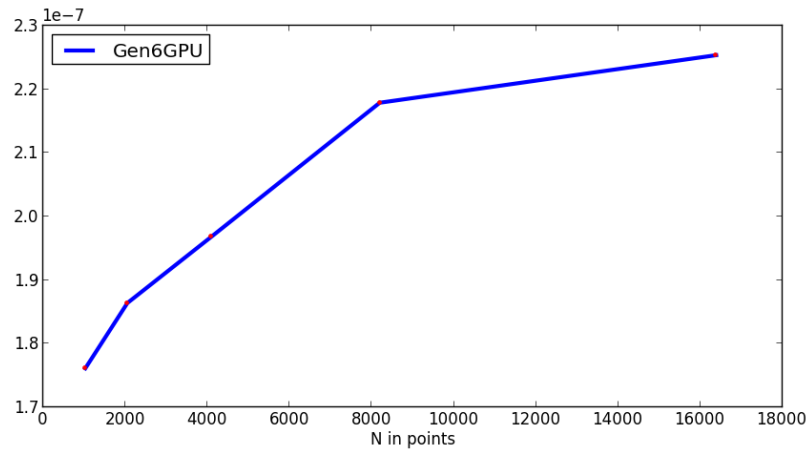


Figure 22: Discrepancy on 6th Generation Processor

Figure 22 gives us the Discrepancy in the values computed by the 6th generation processor GPU.

Note: The OpenCL implantation of the FFT application executed on both the generation processors were same. However the 4th Generation processor shows almost hundred per cent discrepancy at certain point and this has to be investigated.

5.2 Matrix multiplication

Algorithm

Step 1: Set up the OpenCL environment- get the platform, choose the device as GPU, create the context and the command queue

Step 2: Set up the clBLAS

Step 3: Allocate the specified amount of memory on the device to transfer the Input Matrices and to store the resultant Matrix

Step 5: Transfer the Matrix data to the device and execute the matrix multiplication on the OpenCL device.

Step 6: Wait for the device to finish the computations. Fetch the result from the device. Compare the results from the OpenCL device to Numpy results and store the discrepancy. Also store the write and the read time (Time taken to transfer the data to the device memory and to read back the results)

Step 7: Repeat the steps 5 and 6 for a certain number of times to compute the average result.

Step 8: De-allocate the allocated memory resources. Do not forget to de-allocate and free the memory or else it may result in error message (CL_OUT_OF_HOST_MEMORY, CL_OUT_OF_RESOURCES) and the application might hang.

Step 9: Repeat again from step 3 now for different input matrix sizes.

Step 10: The application remains running until it is tested for all the set of inputs. After the application exits store the resultant evaluation data onto a file

Evaluation Results for the matrix multiplication

The application was tested for different sizes of input matrix. Each combination of the input matrices was tested for 1000 times with random float value. The evaluated results shown here are for 7 output matrix sizes- 160x160, 320x320, 480x480, 640x640, 800x800, 960x960 and 1120x1120.

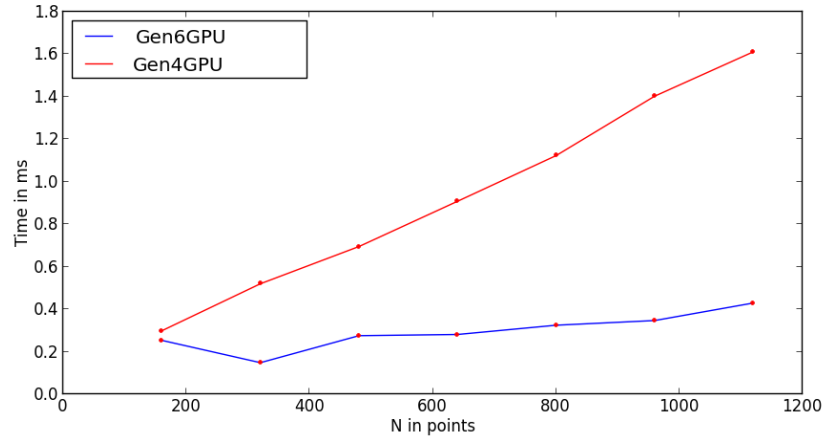


Figure 23: Kernel Execution Time

The Figure 23 represents the average time taken by the device to compute the output matrix. The red dots represent the 7 output matrix and the horizontal axis represents the size of the output matrix. The Gen4GPU represents the 4th generation processor and the Gen6GPU represents the 6th generation processor.

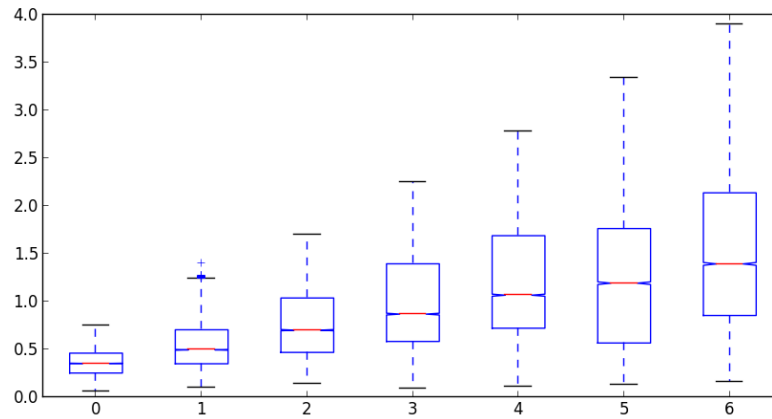


Figure 24: Kernel Execution Time 4th Generation Processor

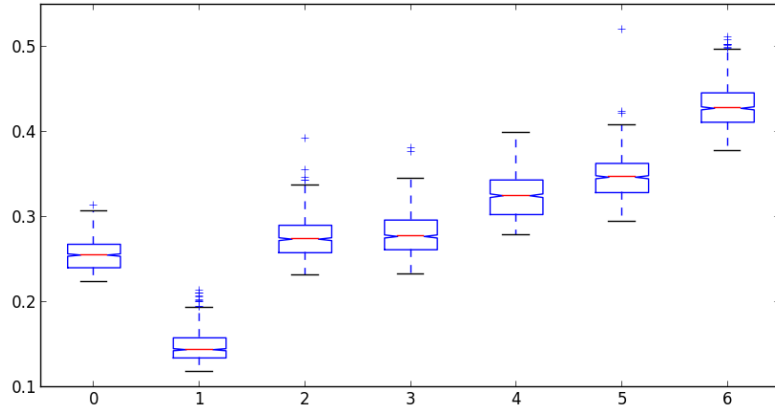


Figure 25: Kernel Execution Time 6th Generation Processor

Figure 24 and 25 are the Box Plot representation of the time taken by the device to compute the output matrix on the 4th and the 6th generation processors respectively. The horizontal axis represents the 7 output matrices. Each output matrix was computed for 1000 times with random float values.

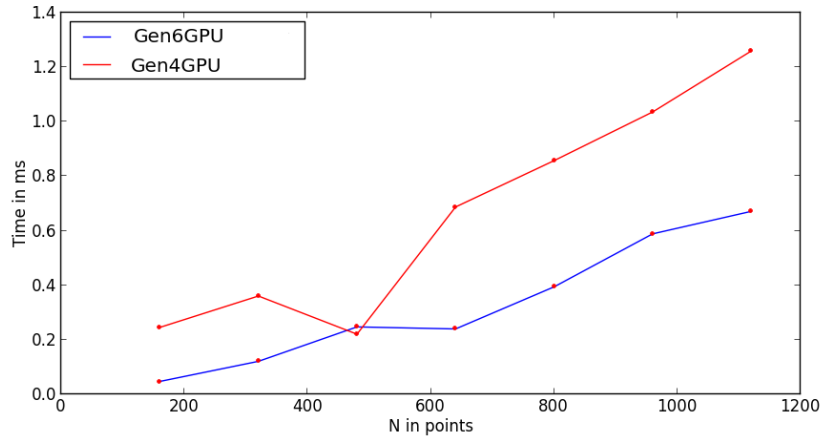


Figure 26: Read Time

Figure 26 is the average time taken by the host to read the results from the device memory. The red dots represent the 7 output matrices and the horizontal axis represents the size of the output matrix.

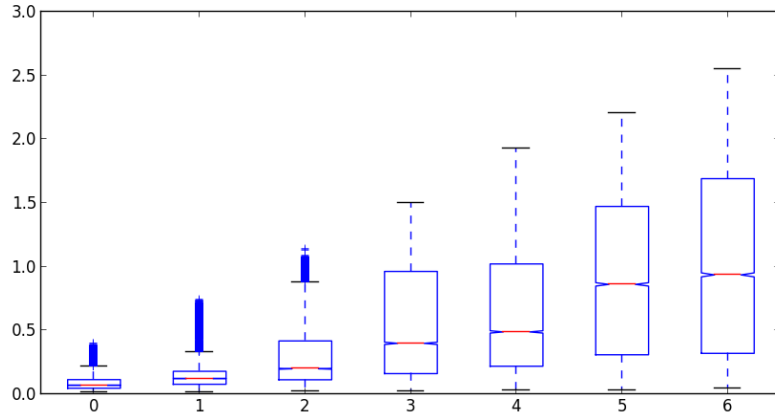


Figure 27: Read Time 4th Generation Processor

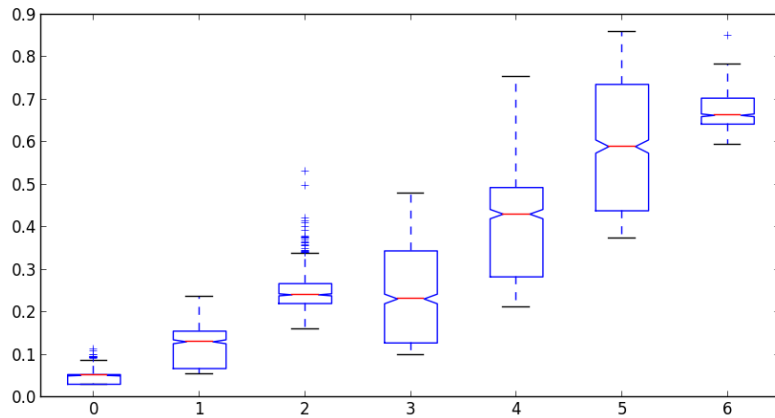


Figure 28: Read Time 6th Generation Processor

Figure 27 and 28 are the Box Plot representation of the time taken by the host to read the results from the device memory on the 4th and 6th generation processors respectively. The horizontal axis represents the 7 output matrices. Each output matrix was computed for 1000 times with random float values.

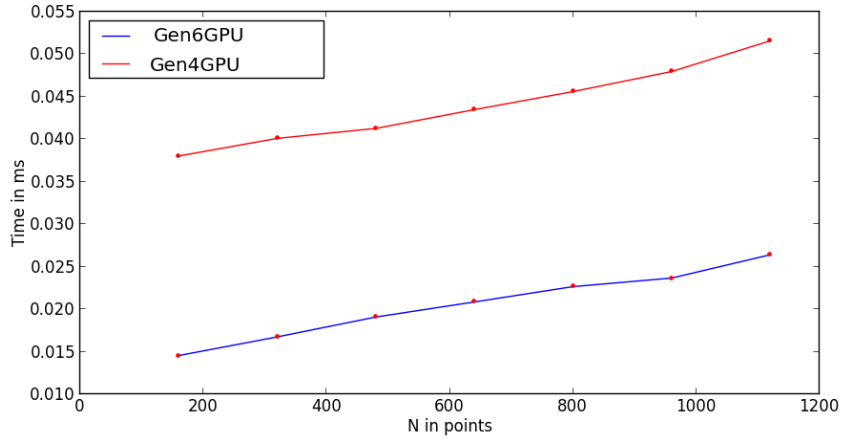


Figure 29: Write Time

Figure 29 is the average time taken by the host to transfer the input matrices onto the device memory. The red dots represent the 7 sets of input matrices.

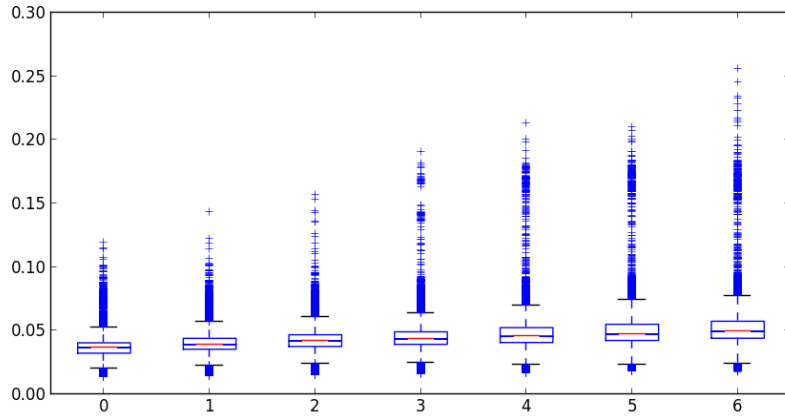


Figure 30: Write Time 4th Generation Processor

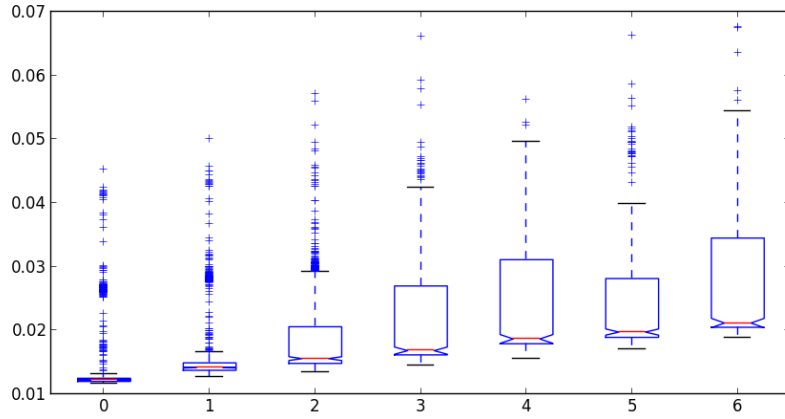


Figure 31: Write Time 6th Generation Processor

Figure 30 and 31 are the Box Plot representation of the time taken by the host to transfer the data onto the device memory on the 4th and 6th generation processors respectively. The horizontal axis represents the 7 output matrices. Each output matrix was computed for 1000 times with random float values.

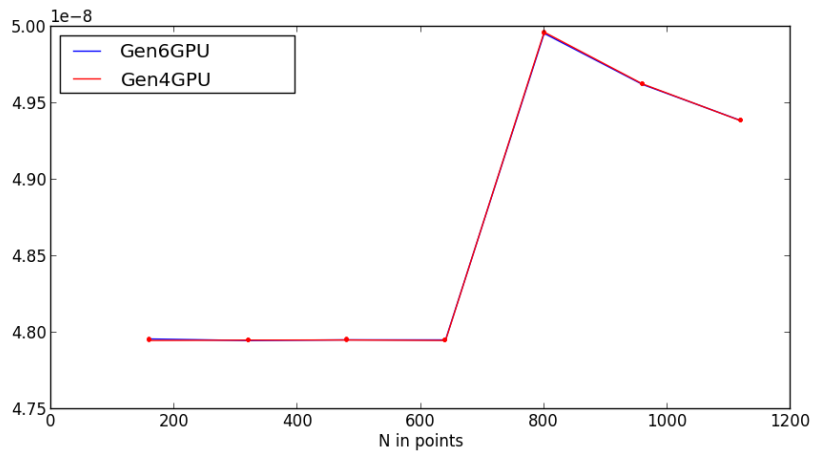


Figure 32: Discrepancy

Figure 32 gives us the average Discrepancy in the values computed. The red dots represent the 7 output matrices and the horizontal axis represents the size of the output matrix

6. Current Status and Future Work

The 4th and the 6th generation processor based systems are currently capable of running OpenCL applications. However the 4th generation processor showed stability issues with the graphic display while running the OpenCL application. This has been noted and is to be investigated (**Snapshot [1]**, note that there were also issues while installing the OpenCL drivers on the system. Details of which has already been mention under “**system preparation**”). Moreover the OpenCL application to compute the FFT using the clFFT library showed very high discrepancy on the 4th generation processor graphics which also has to be investigated. The 6th generation processor is comparatively stable and there are currently no known issues.

The intention of this work was only to develop and create a platform capable of running and testing OpenCL applications and therefore we think there is an immense possibility to optimize these applications for better performance [9]. Intel has a runtime generated FFT for intel processor graphics [11]. This article discusses on techniques to optimize the FFT, specifically for Intel graphics. Similarly, Intel also has a work published for matrix multiplication [12]. The clFFT and clBLAS libraries should also be looked into in detail for further optimization possibilities in terms of performance and memory usage.

7. References

- [1] Ravishekhar Banger, Koushik Bhattacharyya “OpenCL Programming by Exa and the horizontal axis represents the size of the output matrix mple”
- [2] The Compute Architecture of Intel Processor Graphics Gen9
- [3] <https://software.intel.com/en-us/forums/opencl/topic/681561>
- [4] <https://software.intel.com/en-us/forums/opencl/topic/393041>
- [5] <https://github.com/clMathLibraries/clBLAS/issues/315>
- [6] <https://wiki.archlinux.org/index.php/GPGPU>
- [7] <http://www.digit.in/apps/getting-started-with-opencl-code-builder-25940.html>
- [8] <https://software.intel.com/en-us/forums/opencl/topic/401195>
- [9] <https://software.intel.com/en-us/node/540421>
- [10] https://en.wikipedia.org/wiki/Box_plot
- [11] <https://software.intel.com/en-us/articles/genFFT>
- [12] <https://software.intel.com/en-us/articles/sgemm-for-intel-processor-graphics>
- [13] <https://github.com/clMathLibraries/clFFT/issues/186>
- [14] http://registrationcenter-download.intel.com/akdlm/irc_nas/11396/intel-opencl-4.1-release-notes.pdf
- [15] <https://software.intel.com/en-us/articles/getting-started-with-opencl-code-builder>
- [16] <http://developer.amd.com/community/blog/2015/08/24/accelerating-performance-acl-clfft-library/>

8. Software downloads links

- [1] <https://github.com/clMathLibraries/clBLAS/releases>
- [2] <https://github.com/clMathLibraries/clFFT>
- [3] <https://github.com/clMathLibraries/clBLAS>
- [4] <https://01.org/beignet>
- [5] Clang for x86_64 Ubuntu 14.04 (.sig)
- [6] <http://releases.llvm.org/download.html>
- [7] <https://software.intel.com/en-us/articles/intel-code-builder-for-opengl-api>
- [8] <https://software.intel.com/en-us/intel-opengl>
- [9] <http://www.boost.org/>
- [10] <https://www.centos.org/download/>
- [11] <https://software.intel.com/en-us/articles/opengl-drivers>
- [12] <http://fr2.rpmfind.net/linux/rpm2html/search.php?query=dms>
- [13] <https://fedoraproject.org/wiki/EPEL>

Note: Software's were already downloaded and is saved in a USB memory stick

9. Appendix

A sample program to demonstrate vector addition in OpenCL

```
#include <iostream>
#include <CL/cl.hpp>
#include <time.h>

long const LENGTH = 11111111;
double A[LENGTH];
double B[LENGTH];
```

```

double C[LENGTH];
void initialize()
{
    for (long i = 0; i < LENGTH; i++)
    {
        A[i] = rand() / (double)RAND_MAX;
        B[i] = rand() / (double)RAND_MAX;
    }
}

double get_event_exec_time(cl::Event event)
{
    cl_ulong start_time, end_time;
    /*Get start device counter for the event*/
    event.getProfilingInfo<cl_ulong>(CL_PROFILING_C
OMMAND_START, &start_time);
    //clGetEventProfilingInfo(event,
    //    CL_PROFILING_COMMAND_START,
    //    sizeof(cl_ulong),
    //    &start_time,
    //    NULL);
    /*Get end device counter for the event*/
    event.getProfilingInfo<cl_ulong>(CL_PROFILING_C
OMMAND_END, &end_time);
    //clGetEventProfilingInfo(event,
    //    CL_PROFILING_COMMAND_END,
    //    sizeof(cl_ulong),
    //    &end_time,
    //    NULL);
    /*Convert the counter values to milli seconds*/
    double total_time = (end_time -
start_time) * 1e-6;
    return total_time;
}

int main(){

//Step1: Get the available Platforms//
std::vector<cl::Platform> all_platforms;
//The function get all the platforms available
//Platforms are nothing but the OpenCL SDK installed
//Therefore you may find more than one platform availa
ble
//The following link might help

```



```

//https://streamcomputing.eu/blog/2015-08-14/opengl-
basics-multiple-opengl-devices-with-the-icd/
cl::Platform::get(&all_platforms);
//If no platforms are found installed
if(all_platforms.size()==0){
std::cout<<" No platforms found. Check OpenGL installa
tion!\n";
exit(1);
}

//Select the Platform and choose the device//
//I have 2 platforms here one from the Intel and the o
ther from the AMD
//I use the AMD here because then I have access to the
AMD Radeon Graphics Card(GPU)
cl::Platform AMD_platform=all_platforms[1];
std::cout << "Using platform: " <<
AMD_platform.getInfo<CL_PLATFORM_NAME>() << "\n";

//get all the devices under this platform
std::vector<cl::Device> all_devices;
AMD_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devic
es);
//If no devices are found
if(all_devices.size()==0){
std::cout<<" No devices found. Check OpenGL installati
on!\n";
exit(1);
}
//Get the first device which is the AMD GPU
cl::Device GPU_device=all_devices[0];
std::cout << "Using device: " << GPU_device.getInfo<CL
_DEVICE_NAME>() << "\n";

```

//Step1: Get the available Platforms-//

//Step2: Create the Context//

```

//The OpenGL context is created for one or more device
s
//Like GPU, CPU, or DSP or Accelerator and so on.
//Imagine the Context as the runtime link to the our d
evice and platform
cl::Context context( GPU_device );
//Step2: Create the Context//

```

```

//Step3: Creation of Command Queue//
//create queue to which we will push commands for the
device.
//Each Command queue points to a single device within
a context
cl::CommandQueue queue(context, GPU_device, CL_QUEUE_P
ROFILING_ENABLE);
//Step3: Creation of Command Queue//

```

```

//The step 4: Compile and load the
//kernel to
//execute on our device//

```

```

cl::Program::Sources sources;

//The actual source of our program is the Kernel//
//kernel calculates for each element C=A+B
std::string kernel_code =
"    void kernel simple_add(global const int* A, global const int
* B, global int* C){
"        C[get_global_id(0)]=A[get_global_id(0)]+B[get_global_id(
0)];
"    }";

//The actual source of our program is the Kernel//
sources.push_back({kernel_code.c_str(),kernel_code.length()});

//Build the kernel to be executed on the device -//
//The build of the kernel can only be handled during
//run time because we do not know the device
//before
cl::Program program(context,sources);
//When the program is build it is also checked for errors
if (program.build({ GPU_device }) != CL_SUCCESS){
std::cout << " Error building: " <<
program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(GPU_device) << "\n";
exit(1);
}
//Build the kernel to be executed on the device //

```

```

//The step 4: Compile and load the
//kernel to
//execute on our device//

```

```
//Step5: To allocate space on the device  
//memory and transfer the data to the memory//
```

```
// create buffers on the device to handle the arrays  
//To allocate space on the device-//  
cl::Buffer buffer_A(context, CL_MEM_READ_ONLY, sizeof(  
double)*LENGTH);  
cl::Buffer buffer_B(context, CL_MEM_READ_ONLY, sizeof(  
double)* LENGTH);  
cl::Buffer buffer_C(context, CL_MEM_READ_WRITE, sizeof  
(double)*LENGTH);
```

```
//To allocate space on the device-//
```

```
initialize();  
//Write to buffer//  
//After creating the buffers move the arrays into the  
device memory  
//write arrays A and B to the device  
queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, sizeof(  
double)* LENGTH, A);  
queue.enqueueWriteBuffer(buffer_B, CL_TRUE, 0, sizeof(  
double)* LENGTH, B);  
//Write to buffer//
```

```
//Step5: To allocate space on the device  
//memory and transfer the data to the memory//
```

```
//Step6:Execute the Kernel on the device//
```

```
//Create the kernel object  
//simple_add is nothing but the kernel name  
cl::Kernel kernel_add=cl::Kernel(program,"simple_add")  
;  
kernel_add.setArg(0,buffer_A);  
kernel_add.setArg(1,buffer_B);  
kernel_add.setArg(2,buffer_C);  
//The number 10 is the number of threads we want to ru  
n  
//Our array size is 10  
std::vector<cl::Event> write_event;
```

```

cl::Event kernel_event;
queue.enqueueNDRangeKernel(kernel_add, cl::NullRange,
cl::NDRange(LENGTH), cl::NullRange, &write_event, &kernel_event);
queue.finish();

```

//Step6:Execute the Kernel on the device//

```

double exec_time;
exec_time = get_event_exec_time(kernel_event);
printf("Time taken to execute the SAXPY kernel = %lf ms\n", exec_time);
clock_t end = clock();
double runtime = (double)(end -
begin) / CLOCKS_PER_SEC;

```

**//Step7: Fetch the results from the device
//memory//**

```

//read result C from the device to array C
queue.enqueueReadBuffer(buffer_C, CL_TRUE, 0, sizeof(double)*LENGTH, C);

```

**//Step7: Fetch the results from the device
//memory//**

```

printf("Runtime: %lfms\n", runtime);

return 0;
}

```

Note: 1

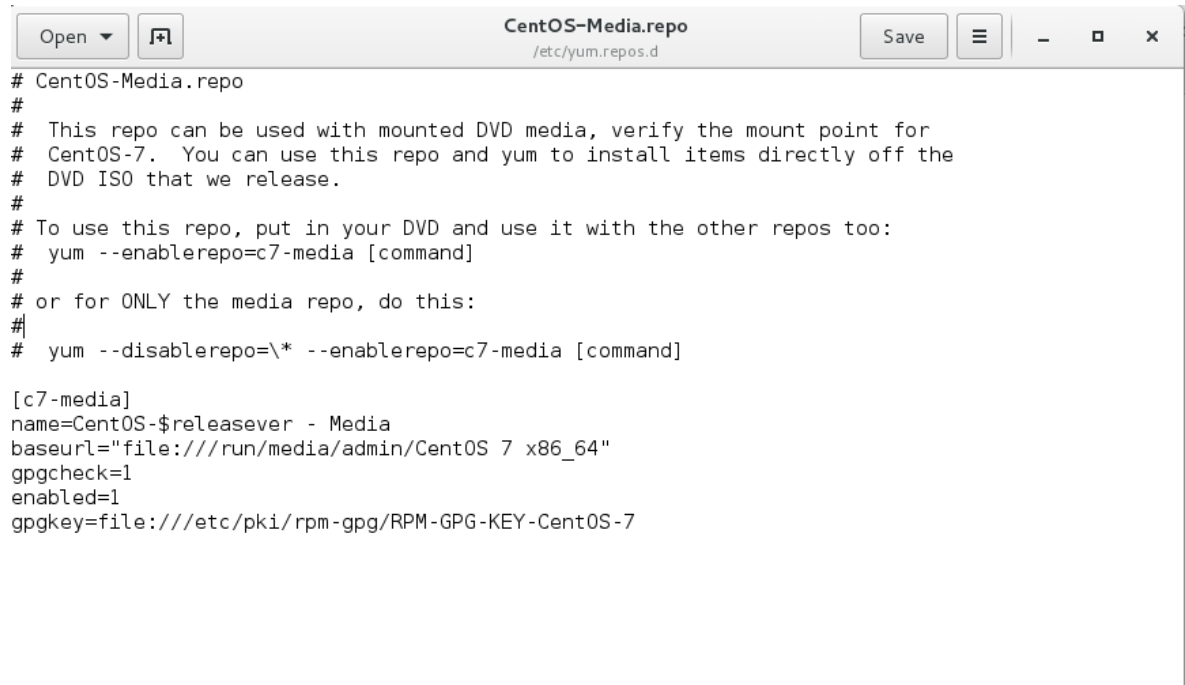
To install the packages from the USB repository it is necessary to change the URL path because for us the repository is the USB memory stick.

Go to the following directory
/etc/yum.repos.d

And edit the following sections in the file
“CentOS-Media.repo”

baseurl = <to your path of the USB memory stick>

The df command will list out the path on which your usb is present copy the path and paste it at the base url as shown



```
# CentOS-Media.repo
#
# This repo can be used with mounted DVD media, verify the mount point for
# CentOS-7. You can use this repo and yum to install items directly off the
# DVD ISO that we release.
#
# To use this repo, put in your DVD and use it with the other repos too:
# yum --enablerepo=c7-media [command]
#
# or for ONLY the media repo, do this:
#
# yum --disablerepo=* --enablerepo=c7-media [command]

[c7-media]
name=CentOS-$releasever - Media
baseurl="file:///run/media/admin/CentOS 7 x86_64"
gpgcheck=1
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
```

And also change the enabled to 1. All the other paths which were present earlier under baseurl was deleted.

From now to install the packages using the yum command please use the command provided in the screenshot. For example

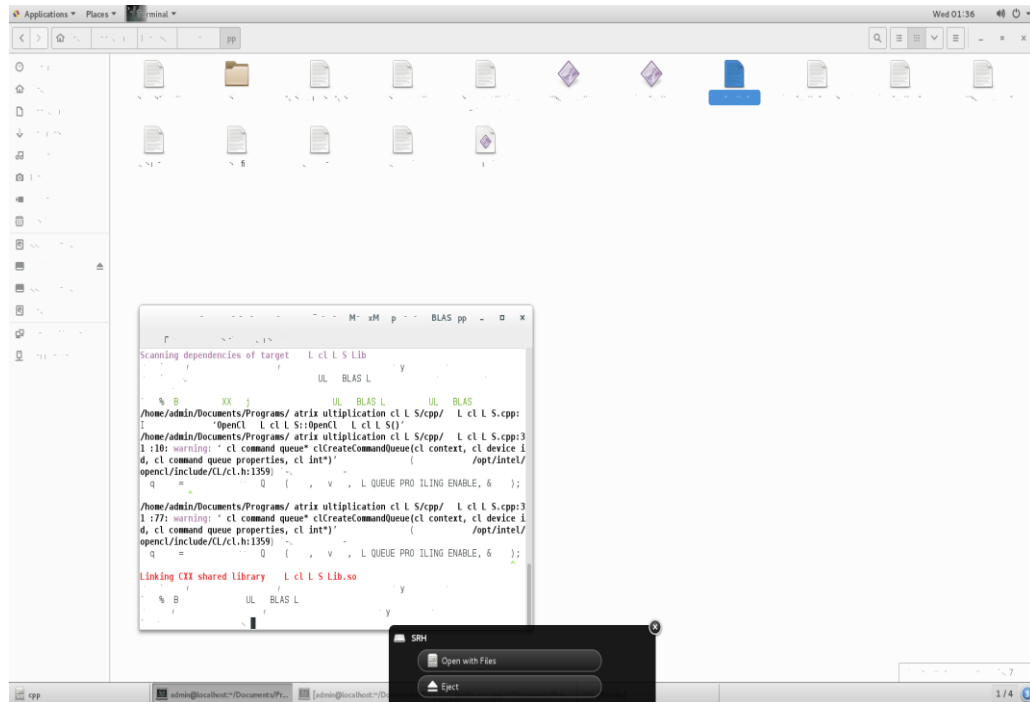
To install

```
yum --disablerepo=* --enablerepo=c7-media install boost-devel
```

To search a package

```
yum --disablerepo=* --enablerepo=c7-media search [package name]
```

Snapshot [1]



The above snapshot was taken during the testing of OpenCL application for matrix multiplication on the 4th generation processor. The gnome user interface had problems after this and we had to restart the system to restore the graphics to its normal state. This problem could be because we did not build and patch the kernel. The following discussions could be helpful

<https://software.intel.com/en-us/forums/opencl/topic/681561>

<https://software.intel.com/en-us/forums/opencl/topic/393041>

Snapshot [2]

Number of OpenCL plaforms: 2

```
-----
Platform: Intel(R) OpenCL
  Vendor: Intel(R) Corporation
  Version: OpenCL 2.0

  Number of devices: 2
  -----
    Name: Intel(R) HD Graphics
    Version: OpenCL C 2.0
    Max. Compute Units: 24
    Local Memory Size: 64 KB
    Global Memory Size: 6270 MB
    Max Alloc Size: 3135 MB
    Max Work-group Total Size: 256
    Max Work-group Dims: (256 256 256)
  -----
    Name: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
    Version: OpenCL C 2.0
    Max. Compute Units: 4
    Local Memory Size: 32 KB
    Global Memory Size: 7848 MB
    Max Alloc Size: 1962 MB
    Max Work-group Total Size: 8192
    Max Work-group Dims: (8192 8192 8192)
  -----

-----
Platform: Experimental OpenCL 2.1 CPU Only Platform
  Vendor: Intel(R) Corporation
  Version: OpenCL 2.1 LINUX

  Number of devices: 1
  -----
    Name: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
    Version: OpenCL C 2.0
    Max. Compute Units: 4
    Local Memory Size: 32 KB
    Global Memory Size: 7848 MB
    Max Alloc Size: 1962 MB
    Max Work-group Total Size: 8192
    Max Work-group Dims: (8192 8192 8192)
  -----

-----
```