Timeline:

2. Cuda
3. Interprocess communication
4. DPTM mockup with LU Decomposition
    a. client <-> server <-> wrapper
    b. started work on ubuntu, then migrated to orodruin
5. DPTM mockup with cuFFT
    a. in relation to LBNE
6. FFTW vs cuFFT vs clFFT
    a. time and gflops comparisons as a function of input size
    b. used pyROOT scripts for presenting data
    c. Somewhere in here stopped doing tests on orodruin, and went to AFSuper
    d. cuFFT has highest performance at large data sizes
7. kernel factory and parameters
    a. Wrapped sdk samples up in to kernel factory framework
    b. used pyROOT scripts for presenting data
    c. time comparison as a function of block sizes

Folder Structure

- **aside** - standalone OpenCL programs used to test system stability
    - **clFFT** - simpler version of 'ClientStandAloneOpenCL' found in 'singleFFT'
    - **matrixMultiplication** - used to test different openCL implementations of matrixMultiplication and determine library linking compatibility
    - **monteCarlo** - unaltered monteCarlo example from Intel SDK
    - **squareArray** - used to compare consisitency of squareArray implemented with OpenCL and CUDA
- **doc** - contains the UML class diagram for the mockup folder
- **mockup** - OpenCL kernel launcher with performance reports
    - **analysis**
        - **Platform - Device** - contains raw data from dpClientSrc with printFile enabled
        - **results**
            - **Platform - Device** - contains graphs for time(x,y,z,mb)
            - **optimal** - contains curves for time(mb) for each kernel on each device
    - **include** - header files
    - **obj** - object files
    - **src** - source files
- **prototype** - cuFFT implementation with a prototype task manager
    - **client.cu** - makes requests to dptm.cu using IPC
    - **wrapper_FFT.cu** - launches kernel on command from dptm.cu
    - **dptm.cu** - takes client.cu requests and combines them to give to wrapper_FFT.cu
- **singleFFT** - FFTW, clFFT and cuFFT comparisons without the use of the task manager
    - **analysis**
        - **oroduruin** - raw data from tests run on the local OSX computer
        - **AFSuper** - raw data from tests run on the local Red Hat computer
    - **conclusion** - contains test results in graphs from pyROOT scripts
- **utilities** - OpenCL programs used to fetch information about the current PC's hardware

Running the Projects:

OpenCL on AFSuper:

There are 3 OpenCL implementations available for use on AFSuper; one from AMD, one from Intel and one from NVIDIA. Each of these implementations has access to all 3 OpenCL platforms available on AFSuper. To see details of the devices on each platform and the names of each platform, try running:

dptm/utilities/ListDevices
dptm/utilities/DeviceDetails

AMD's Firepro cards are not accessible through OpenCL when running on a headless linux machine like AFSuper unless your program is ran with sudo.

Compiling and linking an OpenCL program will require 1 of the following implementations:

-l:/usr/lib64/libOpenCL.so.1 -I/opt/AMDAPP/include (AMD's runtime)
-l:/usr/lib64/libOpenCL.so.1.2 -I/opt/intel/include (Intel's runtime)
-l:/usr/lib64/nvidia/libOpenCL.so.1.0.0 -I/usr/local/cuda/include (Nvidia's runtime)

See the programs under dptm/aside and dptm/utilities for working examples of basic OpenCL programs

CUDA on AFSuper:

You can determine which devices are available for CUDA to use by running '/usr/local/cuda/samples/1_Utilities/deviceQuery'. Other CUDA samples are found in under /usr/local/cuda/samples.

Device Temperatures:

The Intel Phi's temperature is checked with the command 'sudo mic-info'
The Nvidia devices are checked with 'nvidia-smi'
The AMD devices are not able to return temperatures on a headless linux machine
The CPU's temperature can be checked with 'sensors'

Working in mockup:

This project was to allow profiling different kernels simpler. The profiling of kernels is important for the dptm to have so it can figure out how to schedule device requests. There are two executables which essentially do the same thing but are configured in different ways. 'dpClientSrc' requires editing 'src/mainSrc.cpp' while dpClientShell takes command line arguments or shell redirection as explained in 'ShellInput.txt'. The actual structure of the mockup project is best explained with the UML class diagram found in the 'doc' folder.

The ‘analysis’ folder has raw data from the testing under ‘platform - device’ folders. The three python scripts should be run in the order ‘localSizeLoop.py’, ‘formatOptimal.py’ and then ‘plateauGraphs.py’. The first script uses pyROOT to make plots for every data size for each kernel under ‘results/device-kernel’ and to pick out best time found for each kernel launch found in ‘optimal.csv’. The next script formats all the optimal data to a plain text file ‘optimalFormatted.txt’. The last script plots the optimal points for each kernel under ‘results/optimal’.

Working in singleFFT:
This project was built to compare cuFFT, clFFT and FFTW3 performance. The ‘analysis’ folder has the raw data taken through the measurement process and the script that generates the results. The ‘conclusion’ folder has a plot with the final results on multiple devices. There are 4 types of executables in the singleFFT project. The OpenCL one uses clFFT, the CUDA one uses cuFFT, the CPU and CPUThreadLoop use FFTW3. Each of these clients loops over data size and on their respective devices (see Arguments.txt) and then output their results to the ‘analysis’ folder. This test was repeated on Orodruin and AFSuper and the folder structure reflects which results are from what computer. The ‘ipcsClean.sh’ and ‘IncreaseSharedMemory.sh’ reset shared memory on the operating system in case of failed launches. The ‘RunMultiAllTypes.sh’ and ‘StartSimultaneousTest.sh’ script are used to see how concurrent clients effects global GFLOPs on a device. This is what the DPTM hopes to optimize by providing device management to improve unmanaged performance.

Working in protoype:
This project was meant to explore a toy version of the DPTM. It uses 3 pieces essentially. First is the client, which makes a computation request. Next is the wrapper which is responsible for communicating with the CUDA device. Last is the toy DPTM which is a daemon service that uses select to listen to sockets for new requests from potential clients and to assemble the requests into a new task for the wrapper. You can run this program by using the provided scripts. These scripts launch the dptm first and then the wrapper to secure their connection. Next, the script launches clients which then start making requests to the dptm.

The wrapper, client and dptm are all written in their own source files and are compiled and linked independent of each other. The script ‘IncreaseSharedMemory.sh’, ‘Clean.sh’ and the script ‘clean_ipcs.sh’ deal with making sure the operating system has enough shared memory to support the tests. The script ‘RunScan.sh’ launches ‘RunTest.sh’ with increasing parameters. The ‘setupCUDA.sh’ script sets environment variables for cuda to run.

Working in aside:
This folder was used to do rough estimations and unit checking for the programs found in ‘mockup’, ‘singleFFT’ and ‘protoype’.

‘clFFT’ was used to check how linking the clFFT library from AMD was going to work before combining it with cuFFT in the ‘singleFFT’ folder.

‘matrixMultiplication’ was used to see what the Default LD_LIBRARY_PATH was doing during the linking step in comparison to explicitly linking Intel’s OpenCL implementation.

‘monteCarlo’ was used to check if Intel’s monte carlo sample would run without much tweaking.

‘squareArray’ was the inspiration for the ‘mockup’ project. It was first used to roughly see what the difference in performance of a CUDA and OpenCL square array kernel.

Working in Utilities:
The files in here are used to get OpenCL information for devices on the current system. The first step is to try running the ‘compileAll.sh’ script to compile and link the source code found in the ‘src’ directory. This is a good place to start to get a feel for linking and compiling OpenCL in general. After compiling the executables, they are ready to be launched. Note you need to run them with ‘sudo’ for headless linux systems to get access to AMD cards.