DOI: 10.15439/2016F327
ACSIS, Vol. 9. ISSN 2300-5963

Energy-efficient FPGA Implementation of the k-Nearest Neighbors Algorithm Using OpenCL

Fahad Bin Muslim, Alexandros Demian, Liang Ma,
Luciano Lavagno
Department of Electronics and Telecommunication
Politecnico di Torino, ITALY

Affaq Qamar
Department of Electrical Engineering
Abasyn University, Peshawar Pakistan

Abstract—**Modern SoCs are getting increasingly heterogeneous with a combination of multi-core architectures and hardware accelerators to speed up the execution of compute-intensive tasks at considerably lower power consumption. Modern FPGAs, due to their reasonable execution speed and comparatively lower power consumption, are strong competitors to the traditional GPU based accelerators. High-level Synthesis (HLS) simplifies FPGA programming by allowing designers to program FPGAs in several high-level languages e.g. C/C++, OpenCL and SystemC.**

This work focuses on using an HLS based methodology to implement a widely used classification algorithm i.e. k-nearest neighbor on an FPGA based platform directly from its OpenCL code. Multiple fairly different implementations of the algorithm are considered and their performance on FPGA and GPU is compared. It is concluded that the FPGA generally proves to be more power efficient as compared to the GPU. Furthermore, using an FPGA-specific OpenCL coding style and providing appropriate HLS directives can yield an FPGA implementation comparable to a GPU also in terms of execution time.

Keywords—kNN; FPGA; High-Level Synthesis; Hardware Acceleration; low-power low-energy computation; Parallel Computing; OpenCL.

I. Introduction

The ever increasing requirement for electronic devices to perform a variety of compute intensive operations has resulted in the evolution of advanced system-on-chip (SoC) designs with heterogeneous system architectures. These heterogeneous systems are essentially multi-core systems offering a substantial gain in performance not only by utilizing additional cores but also by embedding specialized hardware accelerators e.g. Graphics Processing Units (GPUs) and field programmable gate arrays (FPGAs) for accelerating various compute intensive parts of complex applications. A simplified overview of such a system is shown in Fig. 1.

These systems offer substantial gains in execution time as well as energy efficiency [1]. Modern high performance computing (HPC) systems thus rely on such heterogeneous systems consisting of traditional processors for performing the sequential tasks and FPGAs used as accelerators performing tasks concurrently. Modern FPGAs have the ability to provide sufficient processing speed while consuming a fraction of the power consumed by high-end GPUs [2]. This is why several big data companies such as Microsoft, Baidu are exploring FPGA devices as accelerators rather than GPUs [3, 4].

The major limitation while considering such system architectures is the complexity to program the FPGAs which traditionally requires a considerable expertise in register transfer level (RTL) design. This issue is addressed by an approach called high-level synthesis (HLS), which tends to reduce both the verification and design time and effort for an FPGA based application by allowing the designers to program in several higher level languages such as C, C++ and SystemC.

A very promising parallel programming language which is built upon C/C++ and can be used to program an FPGA is the Open Computing Language (OpenCL). The fact that OpenCL is based upon C/C++, makes porting a program from C/C++ to OpenCL quite easy [5]. OpenCL is a programming standard developed by the Khronos group to develop applications being executed on heterogeneous platforms. OpenCL due to its portability holds an edge over the very similar Compute Unified Device Architecture (CUDA) programming framework, which can be used to program NVIDIA GPUs only. Though OpenCL is device portable, yet it does not offer performance portability across multiple devices. OpenCL program support on Xilinx FPGA devices is provided by SDAccel^{TM}, which is a Xilinx development environment for synthesizing OpenCL kernels to be executed on Xilinx FPGA devices [6]. The Xilinx OpenCL high-level synthesis tool, namely Vivado HLS, has been used in this work to implement the k-nearest neighbor (kNN) algorithm onto Xilinx FPGAs.

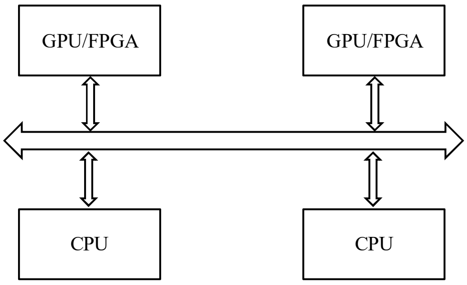

Fig. 1. Typical Heterogeneous System Architecture

The kNN algorithm is used to find the *k* nearest neighbors of a specific point among a set of unstructured data points. It is widely used in a diverse range of domains and applications such as pattern recognition, machine learning, computer vision and coding theory to name a few. The algorithm unfortunately has a significantly large computation cost, since typically training data sets are very large [7]. The algorithm though is highly parallelizable and can be accelerated considerably by exploiting the inherent parallelism of an FPGA device.

This work starts from a parallel implementation of the kNN algorithm from the Rodinia library [8]. Two different implementations of the algorithm are considered and their time, energy/power and cost performance are compared for different hardware platforms i.e. GPUs and FPGAs. We show that, even though GPU and FPGA have similar memory hierarchies, *the best implementation for an FPGA is obtained from OpenCL code that is different from the one leading to the best GPU implementation.* This is because the final selection of the k nearest neighbors is difficult to parallelize with the "doall" strategy implied by OpenCL, but it can still be efficiently pipelined on an FPGA. Furthermore, GPUs have higher DRAM access bandwidth as compared to an FPGA.

The FPGA implementation of our OpenCL code has been obtained by utilizing the SDAccel tool chain from Xilinx, including tools from the Vivado® Design Suite [6, 9]. The algorithm has been implemented on a Virtex-7 FPGA. The GPUs considered for comparison are the GeForce GTX 960 and the Quadro K4200, both by NVIDIA.

The main contributions of this paper are:

- An investigation of the issues encountered when implementing and optimizing an OpenCL code onto a Xilinx FPGA device.
- A performance comparison of the FPGA and GPU implementation in terms of time, energy and power.

The rest of the paper is organized as follows. Section II presents a brief overview of our algorithm and of the OpenCL programming model. A summary of the related work is presented in section III. Section IV describes in detail our adopted methodology. Section V presents the results and the work is concluded in section VI.

II. Overview

This section of the paper presents a brief overview of the kNN algorithm. An overview of OpenCL with a brief description of its platform, execution and memory model is also presented here.

A. kNN Algortihm

Given a set S of n reference (training) data points in a d-dimensional space and a query point q, the k-nearest neighbor algorithm returns the k points in S that are closest to point q. This is illustrated for k = 3 and n = 20 in Fig. 2. The circle represents the query point while the diamonds represent the reference data points.

The algorithm consists of the following main steps:

1- Compute n distances between the query point q and the n reference points of the set S. The distance in our case is the squared Euclidean distance, i.e. for two bi-dimensional points (x_1,y_1) and (x_2,y_2):

$$d = (x_1 - x_2)^2 + (y_1 - y_2)^2 \qquad (1)$$

2- Sort the n distances while preserving their original indices (as specified in S).

3- The k nearest neighbors would be the k points from the set S corresponding to the k lowest distances of the sorted distance array.

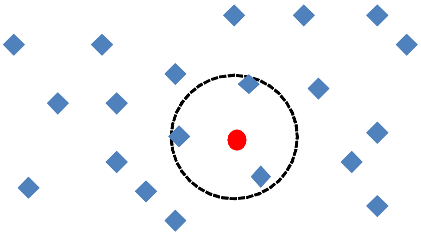

Fig. 2. Illustration of the kNN search algorithm with $k = 3$

B. Overview of OpenCL

OpenCL is an open, industry standard portable framework for writing parallel programs to be executed on heterogeneous platforms consisting of central processing units (CPUs), GPUs, digital signal processors (DSPs) and FPGAs [10]. OpenCL code can be run on a variety of supporting devices by making minimal changes to the host code, hence making it portable. The standard is derived from ISO C99 with additions to support both task-parallel and data-parallel programming models.

At the heart of the OpenCL platform model is the host, which is typically a CPU used to setup the environment for the OpenCL program to run on one or more devices. A device in OpenCL terminology is any hardware platform used to accelerate the compute intensive portions of the application. A piece of code running on the device is called a kernel. The OpenCL device consists of compute units (CU) each further divided into processing elements (PE) as shown in Fig. 3.

Several concurrent executions of the kernel body, called work-items, are grouped into work groups, which can be executed in parallel by multiple processing elements. The memory is broadly divided into host (i.e. CPU) memory and device (i.e. GPU or FPGA) memory. The device memory is also explicitly split into private memory (specific to each work-item), local memory (shared by all the work-items in a work group) and a global/constant memory shared by all the work groups. Global memory offers the slowest access but has the largest capacity while private memory is the smallest but the fastest among all. The memory model is also depicted in Fig. 3.

The main difference between OpenCL code executed on a CPU/GPU and an FPGA lies in the way the code is compiled. For CPU/GPU, the code is compiled in a just-in-time manner to exploit the fixed computing architectures of the devices. The intrinsic flexibility of the FPGA architecture on the other hand allows the designer to explore several kernel optimizations and CU combinations. The main caveat is that the generation of these highly optimized compute architectures takes longer than what a just-in-time compilation allows. The OpenCL standard addresses this issue by allowing for an offline compilation of OpenCL code to be implemented on FPGA devices [6].

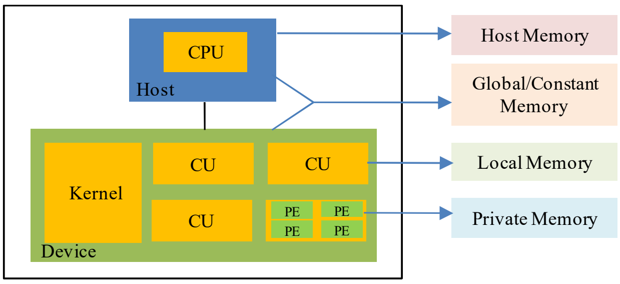

Fig. 3. OpenCL Platform and Memory Model

III. RELATED WORK

The prospect of using an FPGA as an accelerator in modern HPC systems has already been emphasized. This portion of the paper highlights some related work already done in this regard. It also describes some previous research work done in accelerating the kNN algorithm.

A performance comparison of a complex computer vision algorithm used for linear structure detection implemented over a GPU and an FPGA has been presented in [5]. The implementation platforms considered are an AMD Radeon HD6870 GPU and a Xilinx Spartan6 LX150 FPGA. The results show that the FPGA implementation performs better both in terms of power and speed as compared to the GPU. Unlike our case, where we use HLS to automatically generate RTL from an OpenCL code, the authors of this work have written VHDL code manually to implement the algorithm onto the FPGA. The complexity of writing code at RTL is obviously much higher than doing it for the GPU via OpenCL, because: (1) the number of lines of code is much higher at RTL, (2) verification and debugging are much slower, and (3) each RTL model implements one micro-architecture, while a single OpenCL model can generate several micro-architectures by providing different directives to the HLS tool e.g. pipeline or unroll a loop, partition a memory etc.

An accelerated kNN algorithm implemented on an FPGA-based heterogeneous computing system was presented in [11]. Altera's OpenCL compiler was used for compiling the OpenCL code onto the FPGA. The algorithm was implemented onto an Intel Core i7-3770 CPU, an AMD Radeon HD7950 GPU and an Altera's StratixIV 4SGX530 FPGA. The FPGA beats both GPU and CPU in terms of power and energy-per-computation consumption, but the GPU performs better than the FPGA in terms of execution time, most likely because of the higher DRAM access bandwidth of modern GPUs.

A detailed survey on how to parallelize the nearest neighbor algorithm was presented in [12] where, the author advocates both the opportunity and the need to parallelize such algorithms. A GPU based acceleration of a brute force kNN algorithm using CUDA and the CUBLAS library was presented in [7, 13]. It obviously showed a huge speed up with respect to a highly optimized C++ library implementation.

The use of FPGAs for acceleration is hence a widely accepted proposition. This is shown, e.g. by the decision by Baidu to accelerate its deep learning models for image search by using FPGAs [3]. Similarly, Microsoft, after years of research to accelerate its Bing search engine, is now also looking into how to accelerate deep learning models through FPGAs [4]. Considering the market demand, major FPGA manufacturers Altera and Xilinx have also recently introduced tools to program their respective FPGAs directly through OpenCL [6, 14]. One of our main objectives in this project is to explore how various OpenCL programming constructs are handled by the HLS tool and how can we extract maximum performance from the FPGA device.

IV. TEST CASE IMPLEMENTATIONS

In this paper, we compare two different OpenCL implementations of the kNN algorithm, and we show that the OpenCL code that leads to the best implementation on an FPGA is very different from the one that leads to the best GPU results.

A. Implementation I

The first implementation is a direct implementation of the most easily parallelizable part of the kNN algorithm, namely the distance calculation task, while both sorting and nearest neighbors identification are performed by the host. The implementation uses global memory only, and thus it is mainly a measure of global memory access bandwidth. The distance calculation for each point in the reference data set is completely independent of the other points making the algorithm extremely parallelizable. This implementation is illustrated in implementation I. It makes sense as an "acceleration" of kNN only if the dimensionality *d* of each point is high, and hence distance computation (which is $O(d*n)$) dominates over finding the *k* smallest distances (which is $O(k*n)$).

Implementation I. Distance calculation on device & neighbors on host

```
Input: A query point q and S, a set of reference points;
Output: Indices of the k reference points with the smallest distance from q;
Begin
On device:
for each reference point s ∈ S do
    compute all distances between q and all points s ∈ S;
end for
On host:
for i = 0 to k-1 do
    print the index in S of the i-th smallest element of the distance vector;
end for
End
```

B. Implementation II

This implementation uses two separate kernels, and streams data between them. The first kernel is used to calculate the distances as in the previous case. The second kernel finds the *k* smallest distances and returns their indices at the end of its execution.

This implementation is meant to utilize a streaming memory optimization technique offered by SDAccel, which automatically maps global arrays, used merely for inter-kernel communication, to the on-chip block RAMs. The pseudo code is given in implementation II.

Implementation II. kNN on device using multiple kernels

```
Input: A query point q and S, a set of reference points;
Output: k smallest distances with their respective indices per work-group;
Begin
On device:
declare global distance array “dist” for inter-kernel communication;
                    Kernel1: distance calculation
for each reference point s ∈ S do
    declare local arrays for each point s ∈ S;
    copy each point s ∈ S into the local memory;
    compute the distances between q and points s ∈ S and save in “dist”;
end for
                    Kernel2: find k smallest distances
for i = 0 to k-1 do
    print the index in S of the i-th smallest element of the distance vector;
end for
End
```

V. RESULTS

We performed a comparison of our implementations on GPUs as well as an FPGA. The code has been optimized for the FPGA by using a variety of optimization options offered by the HLS tool from Xilinx, e.g. loop unrolling and pipelining, and we report the best results for each implementation on each platform. The experimental setup along with the results from our experiments is presented here.

A. Experimental setup

The experimental setup consists of three target devices shown in Table. I. The first device is an NVIDIA GeForce GTX960 GPU with 1024 cores and a maximum operating frequency of 1178MHz. The device has about 2GB GDDR5 of global memory, with 112GB/s of memory bandwidth. It is accessible from the host through a PCIe 3.0 interface with 16 lanes. The second device is an NVIDIA Quadro K4200 GPU with 1344 CUDA cores and a maximum clock frequency of 784MHz. The device has about 4GB of GDDR5 global memory, with 172.8GB/s of memory bandwidth. It is accessible from the host through a PCIe Gen2 interface with 16 lanes. The third device is an Alpha data ADM-PCIE-7V3 FPGA board with a Virtex-7 690t. The global memory consists of two DDR3 memories with 21.3GB/s of bandwidth. The host can access it through a PCIe Gen3 interface with 8 lanes.

TABLE I. TARGET PLATFORM COMPARISON

Device	Global memory size	Global Memory Bandwidth (GB/s)	Bus interface
GTX 960	2GB GDDR5	112.0	PCIe 3.0 x16
K4200	4GB GDDR5	172.8	PCIe 2.0 x16
FPGA	Two 8GB SODIMMs	21.3	PCIe 3.0 x8

The dataset used for experimenting with our kNN algorithm is from [15]. It contains locations of various hurricanes and is used by our algorithm to specify the *k* nearest hurricanes in the vicinity of a given query point. *k* is usually small in comparison to the number of points in the reference data set and we have fixed it to 5 in our experiments. The number of reference data points used is about 300,000.

B. Performance Analysis

The parallel architecture of the FPGA has been exploited by exposing parallelism in the kernels through several HLS optimization directives offered by SDAccel. The loop unroll attribute in SDAccel could be used to expose concurrency to the compiler by either fully or partially unrolling the loops in OpenCL kernels. However, fully unrolling loop iterations that access global memory, as in our case, does not ensure the best throughput, since only a few global memory ports are available. So we unroll only enough to match the available maximum number of global memory access ports. Throughput can be further improved by using the loop pipeline attribute, which can pipeline any explicit loop in the kernel as well as the work-item loop within a work group, and better match the limited number of memory ports.

The work group size in the OpenCL standard can be specified by the `(reqd_work_group_size)` attribute which indicates the size of the problem space that can be handled by a single invocation of the kernel compute unit. This attribute is highly recommended in the case of FPGA implementation, because it allows performance optimization during the custom logic generation for the kernel, by informing the synthesis tool about the iteration count of the loop over work items.

Several memory access optimizations are also offered by SDAccel which are critical to performance enhancements on an FPGA. For instance, 2-element vector data types improve the memory access throughput, as compared to using C structs, when reading in two-dimensional data points. One of the optimizations offered by SDAccel, namely “on-chip global memories”, was exploited in implementation II to achieve a very significant speed up in execution. This optimization utilizes the block RAMs in the FPGA to create memory buffers that are visible only to the kernels accessing them, while inter-kernel buffers in “standard” OpenCL are allocated in the external, slower, DRAM.

The performance analysis for implementation I is presented in Table. II. The resource utilization in case of FPGA implementation is also shown. The frequency reported by Vivado HLS is 240MHz. The sorting in this implementation is done on the host. Hence, the sorting time in all the cases is also included as a part of the total execution time of the kNN algorithm. The devices in this implementation are used only to calculate all the distances between the query point and all the reference data points. This process is fully parallelizable with no loop dependencies. Work-item pipelining has been used here for FPGA implementation and the data is read in bursts from the global memory. Both the GPUs perform faster than the FPGA due to their higher DRAM access bandwidth. FPGA implementation however out-performs both the GPUs in terms of power and energy consumption. The power analysis for the FPGA implementation was done using the power estimation capabilities of Vivado. The GPU power on the other hand was estimated based on the datasheet, which from our earlier experiments was very close to the one reported by GPU profiler tools e.g. GPU-Z.

TABLE II. PERFORMANCE ANALYSIS OF IMPLEMENTATION I

Parameters/Devices		**FPGA**	**GTX 960**	**K4200**
Device time		1.24ms	0.04ms	0.05ms
Sort time (Host)		4ms	3ms	3ms
Total execution time		5.24ms	3.04ms	3.05ms
Power (Device)		0.422W	120W	108W
Energy (Device)		0.523mJ	4.4mJ	5.6mJ
Resource Utilization	BRAMs	0	N/A	N/A
	DSPs	12 (0.33%)		
	FFs	3109 (0.36%)		
	LUTs	2006 (0.46%)		

The performance analysis for implementation II is given in Table. III. This implementation also has a clock frequency of 240MHz. It exploits the "on-chip global memories" option, offered by SDAccel for streaming data between kernels. A global memory buffer "dist" is used for inter-kernel communication which gets mapped to the on-chip Block RAMs and is visible only to the kernels that uses it. This explains the increased power consumption in comparison to the other case, where the BRAMs were kept powered down.

TABLE III. PEFORMANCE ANALYSIS OF IMPLEMENTATION II

Parameters/Devices		**FPGA**	**GTX 960**	**K4200**
Total execution time		1.23ms	0.93s	3.11s
Power		3.136W	120W	108W
Energy		0.0039J	111.6J	335.88J
Resource Utilization	BRAMs	512 (34.83%)	N/A	N/A
	DSPs	12 (0.33%)		
	FFs	23892(2.78%)		
	LUTs	11838 (2.76%)		

This implementation is considerably faster on the FPGA than both the GPUs, yet it still consumes both less power and less total energy in comparison. This speed up occurs at the cost of about 7x increase in the power consumption as compared to the FPGA implementation of implementation I. The best case GPU implementation (GTX960) in this case is about 756x slower than the FPGA implementation, since the multiple kernels execute sequentially on the GPU and share only global memory.

VI. CONCLUSIONS AND FUTURE WORK

This paper explores both kernel implementation changes and a variety of HLS directives to optimize the synthesis of an OpenCL application to be implemented on an FPGA platform. Two fairly different implementations of the kNN classification algorithm have been considered as our test cases. The FPGA is found to offer a better power/energy performance as compared to the GPU in all the algorithm implementations. By carefully analyzing the algorithm characteristics, we managed to find an OpenCL implementation of kNN that also results in better overall execution time on an FPGA than on a GPU, and is thus pareto-optimal with respect to GPU implementations with respect to performance, power and energy. It exploits on-chip global memory implementation and data streaming options that are more readily and more frequently available on an FPGA than on a GPU. Future work includes using our findings to enhance HLS tools to improve the level of automation starting from a non hardware specific OpenCL model.

ACKNOWLEDGMENT

The authors would like to extend their gratitude to Xilinx, Inc. for their support while carrying out this work. This work is also supported in part by the European Commission through the ECOSCALE project (H2020-ICT-671632).

REFERENCES

[1] Mavroidis, I., Papaefstathiou, I., Lavagno, L., Nikolopoulos, D. S., Koch, D., Goodacre, J., ... & Palomino, M. (2016, March). ECOSCALE: Reconfigurable computing and runtime system for future exascale systems. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (pp. 696-701). IEEE.

[2] Ovtcharov, K., Ruwase, O., Kim, J. Y., Fowers, J., Strauss, K., & Chung, E. S. (2015). Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, *2*.

[3] Ouyang, J., Lin, S., Qi, W., Wang, Y., Yu, B., & Jiang, S. (2014, August). Sda: Software-defined accelerator for largescale dnn systems. In *Hot Chips*(Vol. 26).

[4] http://www.nextplatform.com/2015/08/27/microsoft-extends-fpga-reach-from-bing-to-deep-learning/ [Accessed: 26 April 2016].

[5] Struyf, L., De Beugher, S., Van Uytsel, D. H., Kanters, F., & Goedemé, T. (2014, January). The battle of the giants: a case study of GPU vs FPGA optimisation for real-time image processing. In *Proceedings PECCS 2014*(Vol. 1, pp. 112-119). VISIGRAPP.

[6] User Guide, "SDAccel Development Environment User Guide v2015.1", Xilinx, 2015.

[7] Garcia, V., Debreuve, E., Nielsen, F., & Barlaud, M. (2010, September). K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing* (pp. 3757-3760). IEEE, http://dx.doi.org/10.1109/ICIP.2010.5654017.

[8] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., & Skadron, K. (2009, October). Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (pp. 44-54). IEEE, http://dx.doi.org/10.1109/IISWC.2009.5306797.

[9] User Guide, "Vivado Design Suite User Guide High-Level Synthesis v2015.1", Xilinx, 2015.

[10] Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, *12*(1-3), 66-73, http://dx.doi.org/10.1109/MCSE.2010.69.

[11] Pu, Y., Peng, J., Huang, L., & Chen, J. (2015, May). An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on* (pp. 167-170). IEEE, http://dx.doi.org/10.1109/FCCM.2015.7.

[12] Aydin, B. E. R. K. A. Y. (2014). Parallel algorithms on nearest neighbor search. *Survey paper, Georgia State University*.

[13] Garcia, V., Debreuve, E., & Barlaud, M. (2008, June). Fast k nearest neighbor search using GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on* (pp. 1-6). IEEE, http://dx.doi.org/10.1109/CVPRW.2008.4563100.

[14] Singh, D. (2011). Implementing FPGA design with the OpenCL standard.*Altera whitepaper*.

[15] http://weather.unisys.com/hurricane/atlantic/2012/index.php [Accessed: 30 June 2016].