Implementation of a performance optimized database join operation on FPGA-GPU platforms using OpenCL

Implementation of a Performance Optimized Database Join Operation on FPGA-GPU Platforms Using OpenCL

Mehdi Roozmeh
mehdi.roozmeh@polito.it

Luciano Lavagno
luciano.lavagno@polito.it

Politecnico di Torino
Italy

Abstract—**The growing trend toward heterogeneous platforms is crucial to meet time and power consumption constraints for high-performance computing applications. The OpenCL parallel programming language and framework enable programming CPU, GPU and recently FPGAs using the same source code. This eases software developers to implement applications on various devices supported by heterogeneous HPC platforms.**
This work presents two very different FPGA implementations of a database join operation, one using a direct $O(n^2)$ algorithm, and the other using a bitonic sort network to speed up the join operation. Comparison of performance and energy consumption for both FPGA and GPUs is provided which suggests a 40% performance/watt improvement by using an FPGA instead of a GPU.

Database, Data Center, FPGA, GPU, OpenCL, High-level synthesis, Low-power low-energy computations, Parallel Computing.

I. Introduction

Energy consumption and power dissipation are significant energy costs for modern datacenters. Intel's acquisition of Altera in 2016 and FPGA deployment in datacenters and cloud infrastructure by Microsoft, Baidu and Amazon indicate the ever-growing interest of industry leaders to implement a wide range of workloads on FPGA. This option which previously was not interesting due to the very high design costs implied by HDL-based FPGA design, is now becoming interesting thanks to both FPGA architectural improvements (e.g. in terms of on-chip CPU cores and of external DRAM bandwidth) and design flow improvements, namely the broad adoption of High-Level Synthesis (HLS) tools [1].

Intel, for example, is supplying Systems-In-Package(SIP) including both Xeons and Altera FPGAs. It may also, in the future, integrate them on the same chip. It is also providing its customers with a Software Development Kit (SDK) which supports these heterogeneous SIPs. The SDK allows the programmer to write kernels in OpenCL and than map them uniformly to FPGA resources, in addition to CPUs and GPUs [2].

The OpenCL Programming model has been developed by the Khronos group to overcome the hurdles of programming multi-core and heterogeneous compute platforms[3]. OpenCL enables programmers to develop both close-to-the-metal and portable software. Although, OpenCL is a high-level programming language, it provides a low-level abstraction layer that can expose significant architectural aspects of the target hardware, such as massive parallelism and the memory hierarchy. The CPU/GPU based platforms generally have a fixed architecture. While this makes programming easier and compilation times much faster, it is also a limitation because it reduces both the energy efficiency and the on-chip ("local" in OpenCL terms) memory access bandwidth with respect to an FPGA [4].

SDAccel is a sophisticated toolchain from Xilinx that supports C/C++ and OpenCL for high-level synthesis targeting Xilinx FPGAs. It starts from software simulation, which only verifies OpenCL functionality, proceeds through the generation of high-quality RTL, whose functionality can be verified through RTL simulation, all the way to placement, routing and bitstream generation.

OpenCL defines hierarchical memory model that is common between all vendors and can be applied to all OpenCL applications[5]. Global, local and private memories are the main layers of this hierarchy. SDAccel maps them to the FPGA platform as external DRAMs, BRAMs, and register. SDAccel allows even finer-grained exploitation of the on-chip memory architecture of FPGAs by using directives such as on-chip global memory, multiple AXI buses for kernel global arrays, and partitioned local arrays, which enable a designer to fine-tune the memory architecture and *adapt the RTL architecture to the application, rather than the application to the GPU architecture.*

This paper presents two implementations of the join operation between two database tables, i.e. the creation of a single merged table containing only elements with the same primary key. One of them is ultra-parallel, based on two nested loops which simply apply the join definition to unsorted tables. The other uses a fast bitonic sort algorithm, with lower complexity, followed by a linear join of sorted tables. Note that sorting is very memory bandwidth-intensive. So this application is a sort of worst case when comparing GPU and FPGA platforms.

II. Background and Related Work

This section summarizes related work on the implementation of the join operation using GPU and FPGA accelerators, with the main focus on using Xilinx FPGAs high-level synthesis.

The authors of [6] discuss relative performance of the nested loop and sort-merge join algorithms. However, they do not discuss a specific target platform. Their results confirm that the sort-merge join algorithm outperforms the nested loop join algorithm except for small data sizes. In [7] modern multi-core processors are compared via an extensive analysis of their performance executing of sort-merge join and radix-hash join. Their results indicate that only when very large amounts of data are involved sort-merge join has better performance than radix-hash join.

Two different hardware implementations of the bitonic sorting network were presented in [8]. The best performing design, in that case, utilized a single memory port and a streaming permutation network (SPA), thus resulting in a memory and energy optimized implementation on a Xilinx Virtex-7 platform. A significant performance improvement was also achieved in [9] by proper pipelining of different stages of the sorting network. In [10], the Bitonic sort algorithm was compiled for a GPU-based hardware platform by using CUDA, where optimizations were done mainly to reduce the number of global memory accesses and the number of kernel launches.

Even though GPUs and CPUs have been the main platform for query processing, FPGAs have recently gained interest due to the availability of FPGA-based reconfigurable computing [11]. Implementation of database systems on FPGA is now much easier, as a result of the availability of OpenCL-based and C-based design flows.

In [13], [14] and [15] the authors discuss the usage of an OpenCL-based synthesis framework targeting FPGAs that encourages many software developers to use them as acceleration platforms. Although using OpenCL as a high-level synthesis input language is not yet mature and significant hurdles should be addressed to achieve high-quality RTL generation, the design speed offered by the new flow more than overcomes any limitations [16].

FPGAs are hence considered as a viable option as an accelerator instead of GPUs especially when energy-per-operation is the main concern. As mentioned above, researchers at Baidu are thus considering FPGAs for accelerating their deep learning models for image search [17]. Microsoft's Bing search engine also uses Altera FPGAs as accelerators in combination with traditional microprocessors from Intel [18]. Keeping in view this market trend and the general perception of the complexity of FPGA programming, two of the major FPGA manufacturers, Intel/Altera and Xilinx, have recently introduced tools to enable the designers to program their respective FPGAs directly using C, C++, SystemC and OpenCL code [20], [21]. There is hence a considerable interest on this topic in the design community. This provided us with a motivation to perform this study.

III. Motivation

The OpenCL programming language has been used for a while as the input to a common framework that can target multiple devices with different available levels of parallelism. The OpenCL execution model [23] subdivides an application into multiple kernel executions. Each kernel, capturing task-level parallelism, executes in parallel a number of Work-Groups (WGs), which in turn execute in parallel a number of WorkItems (WIs), in a doubly nested doall loop structure. The memory hierarchy is also modeled explicitly. Kernels communicate via global memory (DRAM). WIs within a WG share a local memory (SRAM), and each WI has its own private memory (registers). WIs within a WG can synchronize by means of barriers, while WGs are fully concurrent.

Despite market availability of tools that allow implementation of OpenCL code on FPGAs, this path still requires a more significant design effort than for a GPU target, especially when both performance and energy are a concern, and resource limitations must be taken into account. This work focuses on optimizing both the code structure and the memory architecture of generated RTL by extensively using the analysis and synthesis capabilities of the SDAccel design environment from Xilinx. Since this is a memory bandwidth-dominated application, the main goal of these optimizations is to increase efficiency of main memory access and bandwidth utilization.

IV. Analysis of Join Algorithm

Join operations are at the core of all relational databases. Their performance on CPU-based platforms has been discussed extensively for several decades. This section introduces and illustrates the pseudo-code of the nested loop and sort-merge join approaches.

A. Nested Loop join Algorithm

The nested loop join, illustrated in Algorithm 1, is a straightforward approach to join two relations. Since each loop iteration of this algorithm is completely independent of the others, it offers a huge level of parallelism, but also requires a huge memory bandwidth. This is because the complexity in terms of the number of both of memory reads and writes, and of comparison operations is proportional to the *product* of the sizes of the tables being merged (i.e. $O(n^2)$ if the have the same size). Hence this case is almost ideal in terms of raw parallelism, but is absolutely brutal in terms of usage of memory bandwidth [22].

B. Sort Merge Join Algorithm

The main idea behind the sort-merge join algorithm is to that sorts each vector before performing join phase. Then the join process of two sorted vectors can be implemented using one single loop with complexity $O(N + M)$, whose execution time is negligible in comparison with sorting (which is close to $O(NlogN)$).

Bitonic sorting is one of the fastest known sorting networks. In general, the term "sorting network" identifies a sorting algorithm where the sequence of comparisons is not data-dependent, thus making it suitable for parallel hardware implementation. A simple example of sorting network is depicted in Fig. 1, with five comparators and four inputs. The comparators in a layer can work concurrently (i.e. they can be part of a kernel in OpenCL).

Algorithm 1: Nested Loop Join Algorithm

```
Input: vector A[N] and B[N] with Size of N ,M;
Output: A_out[N] ,B_out[M] and Value_out[N*M];
        Output arrays store indices and values of
        input after join operation respectively
for each $i \in \{1 \dots M\}$ do
    for each $j \in \{1 \dots N\}$ do
        if $A[j]$ and $B[i]$ can be joined (have the same
          key) then
            A_out[i*N+j]=j; write current index of A
              into output
            B_out[i*N+j]=i; write current index of B
              into output
            Value_out[i*N+j]=A[j]; write current value
              into output
        end
        else If the join condition is not met mark
          output with holes
        A_out[i*N+j]=-99;
        B_out[i*N+j]=-99;
        Value_out[i*N+j]=-99;
    end
end
```

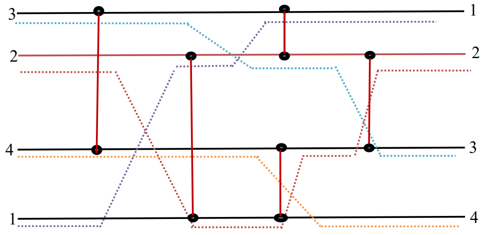

Fig. 1: Illustration of a simple sorting network

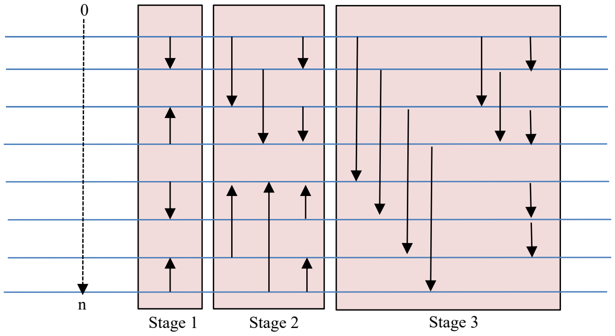

Fig. 2: Bitonic sort network with eight inputs (N=8). It operates in 3 stages, it has a depth of 6 (steps) and employs 24 comparators.

The depth and number of comparators are key parameters to evaluate the performance of a sorting network. The depth of a sorting network is the maximum number of comparators along any path. If all the comparisons in each layer could be done in parallel (i.e. with infinite resources), the depth of the network would be proportional to the total execution time. The bitonic sort network shown in Fig. 2, is one of the fastest comparison sorting networks, where the depth is $D(N) = \frac{log_2 N.(log_2 N+1)}{2}$ and the number of comparators is $C(N) = \frac{N.log_2 N.(log_2 N+1)}{4}$.

Bitonic sorting is a recursive divide-and-conquer algorithm that is based on the notion of bitonic sequence, i.e. a sequence of N elements in which the first K elements are sorted in ascending order, and the last $(N-K)$ elements are sorted in descending order (i.e. the $K-th$ element acts as a divider between two sub-lists, each sorted in a different direction), or some circular shift of such an order.

Bitonic sorting first divides the input into pairs of keys and sorts them into a set of bitonic sequences. It then repeatedly merges and sorts pairs of adjacent bitonic sequences, until the entire sequence is sorted [10].

Algorithm 2 & 3 :

Algorithm 2, executed on the host (i.e. the CPU), iterates the execution of three kernels described in Algorithm 3 to complete bitonic sorting in three phases. In the first phase, the algorithm partially sorts an arbitrary input array to obtain a set of bitonic sequences. In the second and third phases respectively, the algorithm merges bitonic sequences repeatedly until a fully sorted array is produced. All the comparisons (i.e. all the WGs and WIs) in each kernel execution can be performed in parallel and independent of each other. This makes the whole algorithm suitable for parallel implementation. Finally, algorithm 4 performs the join algorithm on two sorted vectors within a single linear complexity loop.

Algorithm 2: Host Code for Bitonic Sorting execution

```
Input: A vector of $N$ keys to be sorted;
Output: A sorted vector of the same keys;
Begin
On host:
SORTLOCAL(Input, Output);
for $size = 4 * Work\_Group\_Size$ to $N$ do
    multiply $size$ by 2;
    for $stride = \frac{size}{2}$ to $stride > 0$ do
        divide $stride$ by 2;
        if $stride >= 2 * Work\_Group\_Size$ then
            MERGE LOCAL(Input, Output, size,
              stride);
        end
        else
        MERGE GLOBAL(Input, Output, size, stride);
        end
    end
end
End
```

Algorithm 3: Bitonic Sorting Kernels

```
On device:
Begin
function KERNEL1: SORT LOCAL(Input, Output)
for $local\_id = 0$ to $Work\_Group\_Size - 1$ do
    copy a block of data from global to local memory
      with the size of work_group;
    for $size = 2$ to $size < Work\_Group\_Size$ do
        multiply $size$ by 2;
        for $stride = size/2$ to $stride > 0$ do
            divide $stride$ by 2;
            perform comparison on each pair and swap
              them if they are not sorted;
        end
    end
    for $stride = Work\_Group\_Size$ to $stride > 0$
      do
        divide $stride$ by 2;
        $pos = 2 * local\_id - (local\_id \& (stride - 1));$
        compare and sort each pair;
    end
    write back sorted array to global memory with the
      size of work_group;
end
function KERNEL2: MERGE LOCAL(Input, Output,
  size, stride)
for $local\_id = 0$ to $Work\_Group\_Size - 1$ do
    read one pair in each Work Item;
    perform comparison on each pair and swap them if
      they are not sorted;
    write back sorted pair to the global memory;
end
function KERNEL3: MERGE GLOBAL(Input,
  Output, size, stride)
declare and initialize a private variable $global\_stride$;
for $local\_id = 0$ to $Work\_Group\_Size - 1$ do
    copy a block of data from global to local memory
      with the size of work_group;
    for $stride = global\_stride$ to $stride > 0$ do
        divide $stride$ by 2;
        perform comparison on each pair and swap
          them if they are not sorted;
    end
end
End
```

Algorithm 4: Join Algorithm for Sorted Relations

```
Input: A[N] and B[M] are two sorted vector
A_index[N], B_index[M] contain indices of A and B
before being sorted;
Output: A_out[N+M] ,B_out[N+M]
        ,Value_out[N+M]; If join condition is met
        output arrays store indices and values of
        inputs respectively
Begin
Function Sort_Join
Initialize i,j,k, t_tmp = 0; i,j are indices of inputs , k is
  the outputs index and j_tmp is used to check for
  successive join between array B elements and the
  same element of A
while ($i < N$ and $j < M$)
if A[i]>B[j] then
    j++;
end
else
if A[i]<B[j] then
    i++;
end
else  If the join condition is met write indices and
  values of inputs into output
A_out[k]=A_index[i];
B_out[k]=B_index[j];
Value_out[k]= A[i];
j_tmp=j+1;
k++;
while(A[i]==B[j_tmp])
A_out[k]=A_index[i];
B_out[k]=B_index[j_tmp];
Value_out[k]= A[i];
j_tmp++;
k++;
end
i++;
end
end
end function
End
```

V. Optimization of Kernel Implementations for FPGAs

SDAccel provides designers with an extensive set of OpenCL attributes that allows the designer to fully control the micro-architecture of the synthesized RTL.Optimization is performed in two phases, a micro-architecture optimization and a macro-architecture optimization. In the first stage of optimization for both test cases, each kernel is optimized only with respect to its internal structure, by using (1) Work Item pipelining, (2) loop unrolling and (3) array partitioning synthesis-specific attributes (i.e. synthesis directives). In the next stage of optimization, multiple WGs of each kernel are instantiated on an FPGA, each with its own global memory access port for each OpenCL kernel argument, in order to fully utilize the off-chip memory band-width and increase memory transfer efficiency.

VI. Performance AND Power Analysis

This section compares the power and performance analysis of the two join algorithms on two GPUs and an FPGA (the Virtex UltraScale VU440). Table I reports the specification of the two GPUs [25] and of the FPGAs that we target in this work. Although the relative performance of each device can vary from one test case to another, the GTX960 often outperforms the K4200 in our experiments. The higher number of cores and higher memory bandwidth of the K4200 are not as effective as one could hope, most likely because of higher core speed (37%) and the use of a second generation Maxwell architecture, with a very large cache, for the GTX960.

Even though companies like Microsoft may not disclose their data-center infrastructure specification in detail, reports suggest that a typical data center can consume about 30 MW and include about 50,000 servers, with one or two GPUs on each card. For example, the Microsoft Azure cloud service offers Tesla K80 cards with two GK 210 GPUs on each card. A Tesla GK 210 has a similar specification to our K4200 GPU in terms of core frequency (562MHz), architecture(Kepler) and double precision support.

Figures 3 and 4 compare the performance of the two discussed test cases on the GPUs and the FPGA, with increasing input table sizes. For both applications, the most advanced VU440 FPGA has better performance than both GPUs. In this experiment, our FPGA implementation uses a 200 MHz clock frequency that results in lower dynamic power consumption and better overall performance per watt (i.e. better energy consumption) than both GPU platforms. Tables II and III present performance, resource usage and power analysis for the two discussed algorithms, using always the same data size (8192 items). In the FPGA case, we instantiated a number of WGs that uses at most about 60% of the on-chip resources, to ensure that the design can be placed and routed[1].

Moreover, for both applications a fully-optimized implementation on both FPGAs consumes less energy than both GPUs to perform the same amount of computation. This is due to the smaller power consumption, and in case of the VU440 also to a lower execution time.

TABLE I: Specification of tested Platforms

Params/Devices	GTX960	K4200	Virtex7 Series	VU440
Architecture	Maxwell GM206	Kepler GK104	Virtex7	Virtex US
Process	28nm	28nm	28nm	20nm
Cuda Cores	1024	1344	-	-
Core Speed	1127 MHz	706 MHz	-	-
Memory Interface	GDDR5	GDDR5	DDR3	DDR4
Memory Bandwidth	112.2GB/sec	172.8GB/sec	200 GB/s	300 GB/s
On-chip memory	1 MB	0.5 MB	6 MB	11 MB
Maximum Power	120W	108 W	-	-
Double Precision	NO	YES	YES	YES
Price	350 $	900 $	3000 $	37000 $

TABLE II: Performance and energy analysis of Nested_Loop Join

Params/Devices	Virtex7	VU440	GTX960	K4200
Device time	29 ms	3.145 ms($t_{clk} = 5ns$)	135 ms	253ms
WGs	65	400	256	256
Band-Width Utilization	3.2 %(6.5 GB/s)	20 %(60 GB/s)	100% (172 GB/s)	59%($66GB/s$)
Device power	35 W	81.7 W	95 W	105 W
Energy	1 J	0.256 J	12.8 J	26.5 J
Utilization	BRAMs = 65(4.4%)	400(16%)	NA	
	DSPs = 260(7.2%)	1600(28%)		
	FFs= 279500(32%)	1720000(33%)		
	LUTs = 260000(60%)	1600000(63%)		

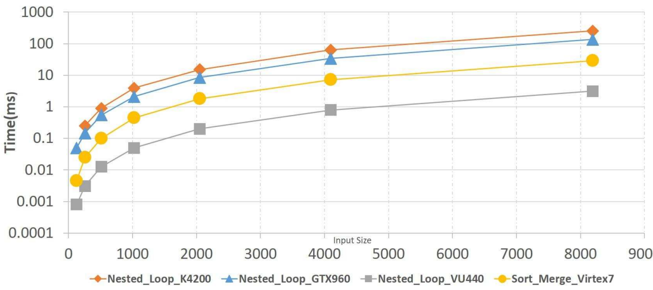

Fig. 3: Performance comparison of nested-loop join versus data size

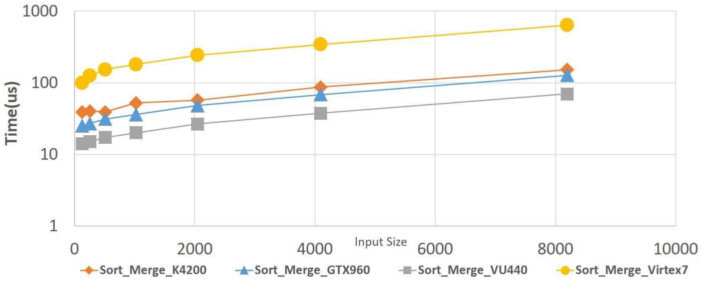

Fig. 4: Performance comparison of sort-merge join versus data size

TABLE III: Performance and energy analysis of Sort_Merge Join

Params/Devices	Virtex7	VU440	GTX960	K4200
Device time	638 us	70 us ($t_{clk} = 5ns$)	127 us	152 us
WGs	20	122	256	256
Band-Width Utilization	5% (10 GB/s)	30% (90 GB/s)	100 % (172 GB/s)	42% (47 GB/s)
Device power	13.2 W	75 W	90 W	100 W
Energy	8.4 mJ	5.2 mJ	11.7 mJ	15.2 mJ
Utilization	BRAMs = 140 (9.5 %)	923 (37%)	NA	
	DSPs = 300(8.3%)	2023 (36%)		
	FFs = 268000(30%)	1634800 (32%)		
	LUTs = 254800 (58%)	1554400 (61%)		

VII. Conclusion

This paper compares performance and energy consumption of two well-known join algorithm implementations on GPU and FPGA devices. Nested-loop and sort-merge join algorithms are memory-intensive computations that require careful optimization to be efficiently implemented on FPGAs. Our experiment suggests that a significant amount of speed up can be achieved by properly using all the optimization techniques offered by SDAccel. Note that, even though sorting is a memory-intensive application, our best implementation makes such effective use of the available DRAM bandwidth that it has better better performance than a GPU. Moreover, thanks to the lower power consumption of the FPGA, its overall energy consumption per operation is significantly better than that of a GPU.

VIII. Acknowledgment

We express our gratitude to Xilinx Inc. for their precious help while carrying out this research activity. This work was supported by the European Commission through the ECOSCALE project (H2020-ICT-671632).

[1] The current version of SDAccel from Xilinx also limits the maximum number of WGs that can be instantiated on an FPGA to 10. We did not consider this limitation since it is tool-dependent, rather than resource-dependent, and will most likely be lifted in future versions of the tool.

REFERENCES

[1] Nicole Hemsoth, Timothy Prickett Morgan. "FPGA frontiers: new applications in reconfigurable computing" Published by Next Platform Press, 2017 edition (January 16, 2017)

[2] "Intel FPGA RTE for OpenCL : Getting Started Guide" UG-OCL005,2016.10.31

[3] Khronos OpenCL Working Group, "The OpenCL Specification", Version 2.0, October 17, 2014 (khronos.org/registry/cl/sdk/2.0/docs/man/xhtml)

[4] "SDAccel Development Environment Methodology Guide: Performance Optimization" UG1207 (v1.0) February 16, 2016

[5] "SDAccel Development Environment: User Guide" , UG1023, September 2015

[6] Chen, Mingxian, and Zhi Zhong. "Block Nested Join and Sort Merge Join Algorithms: An Empirical Evaluation." International Conference on Advanced Data Mining and Applications. Springer International Publishing, 2014.

[7] Balkesen, Cagri, et al. "Multi-core, main-memory joins: Sort vs. hash revisited." Proceedings of the VLDB Endowment 7.1 (2013): 85-96.

[8] Chen, Ren, Sruja Siriyal, and Viktor Prasanna. "Energy and memory efficient mapping of bitonic sorting on FPGA." Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2015.

[9] Mueller, Rene, Jens Teubner, and Gustavo Alonso. "Sorting networks on FPGAs." The VLDB Journal. The International Journal on Very Large Data Bases 21.1 (2012): 1-23.

[10] Mu, Qi, Liqing Cui, and Yufei Song. "The implementation and optimization of Bitonic sort algorithm based on CUDA." arXiv preprint arXiv:1506.01446 (2015).

[11] Boncz, Peter A., Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." CIDR. Vol. 5. 2005.

[12] Wang, Zeke, et al. "Relational query processing on OpenCL-based FPGAs." Field Programmable Logic and Applications (FPL), 2016 26th International Conference on. IEEE, 2016.

[13] Abdelfattah, Mohamed S., Andrei Hagiescu, and Deshanand Singh. "Gzip on a chip: High performance lossless data compression on fpgas using OpenCL." Proceedings of the International Workshop on OpenCL 2013 & 2014. ACM, 2014.

[14] Denisenko, Dmitry. "OpenCL Compiler Tools for FPGAs." Proceedings of the 4th International Workshop on OpenCL. ACM, 2016.

[15] Wang, Kui, and Jari Nurmi. "Using OpenCL to rapidly prototype FPGA designs." Nordic Circuits and Systems Conference (NORCAS), 2016 IEEE. IEEE, 2016.

[16] Krommydas, Konstantinos, Ruchira Sasanka, and Wu-chun Feng. "Bridging the FPGA programmability-portability Gap via automatic OpenCL code generation and tuning." Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on. IEEE, 2016.

[17] Ouyang, Jian, et al. "SDA: Software-defined accelerator for large-scale DNN systems." Hot Chips 26 Symposium (HCS), 2014 IEEE. IEEE, 2014.

[18] "Microsoft knows where exactly Intel's future is" http://www.wired.com/2015/06/Microsoft-knows-exactly-Intel's-future/, 2015, [Online; accessed 12-July-2016].

[19] "Dominate FPGA event," "http://www.eetimes.com/author.asp?section_id=36&doc_id=1330431", 2016, [Online;accessed 12-December- 2016].

[20] Xilinx, SDAccel Development Environment User Guide, Xilinx.

[21] D. Singh, "Implementing FPGA design with the OpenCL standard," Altera white paper, 2011.

[22] Cilardo, Alessandro, and Luca Gallo. "Interplay of loop unrolling and multidimensional memory partitioning in HLS." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015. IEEE, 2015.

[23] Munshi, Aaftab. "The OpenCL specification." Hot Chips 21 Symposium (HCS), 2009 IEEE. IEEE, 2009.

[24] Xilinx UltraScale Architecture for High-Performance,Smarter Systems - WP434 (v1.2) October 29, 2015

[25] http://gpuboss.com/

[26] http://datacenterfrontier.com/inside-amazon-cloud-computing-infrastructure/

[27] http://www.datacenterknowledge.com/

[28] "TESLA K80 GPU ACCELERATOR" BD-07317-001_v05 | January 2015- Board Specification

[29] "Fundamentals of Azure" - Microsoft Azure Essentials- Michael Collier -Robin Shahan

[30] "UltraScale FPGA", *Product Tables and Product Selection Guides*

[31] Vivado Design Suite User Guide "Synthesis" , UG901 (v2016.3) October 21, 2016