Power analysis of sorting algorithms on FPGA using OpenCL

Aidan O Mahony
Department of Electrical and Electronic Engineering
University College Cork
Ireland
103837793@umail.ucc.ie

Emanuel Popovici
Department of Electrical and Electronic Engineering
University College Cork
Ireland
E.Popovici@ucc.ie

***Abstract*—With the advent of big data and cloud computing, there is tremendous interest in optimised algorithms and architectures for sorting either using software or hardware. Field Programmable Gate Arrays(FPGAs) are being increasingly used in high end data servers providing a bridge between the flexibility of software and performance benefits of hardware. In this paper we look at implementations of some of the most popular sorting algorithms using OpenCL which take advantage of FPGA architecture. We evaluate these implementations in terms of power consumption which is measured using dedicated server power loggers and execution on Intel Arria 10 hardware. Our experiments show that taking advantage of software FIFOs have a significant impact on power consumption as well as requiring less hardware and memory resources.**

***Index Terms*—Energy efficiency, FPGAs, Acceleration, OpenCL, Sorting, Power Consumption, Radix Sort, Bitonic Sort, Odd/Even Sort, Insertion Sort**

I. Introduction

Sorting plays an important part in everyday life. It is even more relevant to applications which require big data, cloud computing and the internet. Sorting algorithms speed up drastically every database, search engine and indeed certain other computations. In our quest for instant information, hardware is often used to accelerate software applications.

The aim of this paper is to provide a comparison of a number of hardware oriented data sorting algorithms from the perspective of power consumption. The hardware in question is an FPGA (Field-Programmable Gate Array) which allows for more energy efficient hardware when compared to CPU/GPUs [1] for some types of algorithms.

Energy costs of data centers are expected to account for 50% of the total cost of ownership in the short to medium term [2]. Sorting in the data center is widely used, for example search engines such as Google use sorting to implement its PageRanking algorithm. Furthermore, it is estimated that 85% of scientific applications depend on sorting algorithms [3]. When you consider that in 2011 data centers in the United States consumed 100 billion kWh (amounting to a $7.4 billion energy bill) there appears to be scope to save a significant amount of energy if we can improve the energy-efficiency of sorting in the data center.

978-1-5386-6046-1/18/$31.00 © 2018 European Union

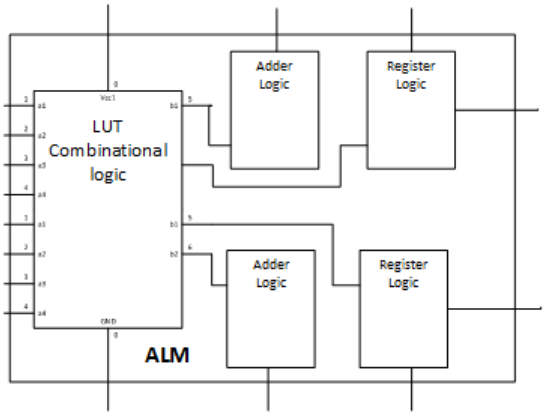

Fig. 1. Adaptive Logic Module

II. Background

In this section we introduce FPGA technology and we will examine the use of the OpenCL framework to FPGA development. Also, we provide some background on sorting, both algorithms and applications.

A. FPGA Technology

An FPGA is a reconfigurable integrated circuit with flexibility approaching general purpose processors while performance is comparable to custom hardware. The first commercially available FPGA, the Xilinx XC2064, had 1200 logic gates however modern FPGAs are massive in scale when in comparison and offer run time programming.

The basic block of an FPGA is an Adaptive Logic Module (ALM) (figure 1) which contains look up tables (LUTs), adders, registers and combinational logic. There is also routing fabric which provides the connectivity between different clusters of logic blocks (also known as logic array blocks (LABs)).

FPGAs are used in a wide variety of industries e.g. automotive (autonomous driving, in-vehicle infotainment), financial, industrial (IoT, manufacturing), and medical (patient monitoring, radiation equipment). Our specific interest in FPGA applications is that of "Big Data" servers.

There already exists a number of applications of FPGA technology for "Big Data" in the commercial space. Microsoft's

Project BrainWave is a *Scalable FPGA-powered DNN Serving Platform* which uses Intel Stratix 10 FPGAs for supporting deep learning frameworks. Amazon is using GPU and FPGA technology to accelerate deep learning, video processing, and many other computations. Baidu are using FPGAs for "Big Data" analysis.

B. Sorting

Knuth [4] divides up sorting algorithms into the following categories; insertion, exchanging, selection, merging, and distribution. We will concern ourselves with algorithms that have already been shown as amenable to implementation in hardware such as those that use "Sorting Networks".

In the age of "Big Data" sorting of data is extremely relevant. An example of "Big Data" sorting is found in the Apache Spark cluster computing framework which offers the *Timsort* sorting algorithm for its Sort Shuffle feature. Indeed, it is worth remarking that Apache Spark has set a record as the fastest open source engine for large-scale sorting previously [5]. Also of interest is applying the sorting circuits to massive data sets such as the method presented in [6].

Power consumption of algorithms (and indeed specifically sorting algorithms) has already been examined for other platforms and languages [7].

C. OpenCL

OpenCL (Open Computing Language) is a framework for writing device independent programs. The OpenCL standard is maintained by the Khronos Group and is currently supported by hardware vendors such as Intel (both for CPU and FPGA), Xilinx (FPGA), Qualcomm (GPU), Nvidia (GPU) and Texas Instruments (DSP). OpenCL allows algorithm designers to deploy their code onto many different types of hardware without the hardware language overhead, i.e. a designer can create an algorithm and deploy it to a GPU and an FPGA without translating the algorithm into both CUDA and VHDL/Verilog. This opens up hardware acceleration to a greater audience of developers.

There are a number of applications already taking advantage of the ability to quickly and easily create custom acceleration circuits. Examples of features already accelerated on FPGA using OpenCL include Gzip compression [8], Genome Sequencing [9], graphics [10], and financial mathematics [11].

An OpenCL program is divided into two distinct sections, the host side and the device side. The host device (generally a CPU) is used to launch and interact with the device (which could also be a CPU, or it could be any other OpenCL supported device e.g. GPU/FPGA/DSP). The device side is implemented in *kernel* form. An OpenCL *kernel* is a routine compiled for a specific device. OpenCL kernels are based on the C99 standard [12]. A simple addition kernel for Intel's OpenCL SDK for FPGAs is illustrated in figure 2.

```
__kernel void simpleAdd(__global int A,
                        __global int B,
                        __global int* C)
{
    *C = A + B;
}
```

Fig. 2. OpenCL Kernel for addition

D. State of the art in sorting on FPGA using OpenCL

A comparison of bitonic-sort on FPGA using OpenCL to a GPU implementation in terms of power consumption and time is presented in [13]. The authors of [14] present OpenCL implementations of Radix sort and Bitonic sort for Xilinx FPGAs with a specific interest in the operations generated by the OpenCL compiler. In [15] a comparison of radix sort, bitonic sort and insertion sort on FPGA is presented, however these were implemented in VHDL rather than OpenCL. Odd-even sort and bitonic sort have been analysed in terms of FPGAs and power consumption in [16] from a HDL perspective.

Algorithm 1 Compare Swap Unit

```
COMPARE_SWAP(*a, *b)
if a > b then
    temp = a
    *a = *b
    *b = temp
end if
```

III. Hardware orientated sorting algorithms

The sorting algorithms presented in this section were chosen due to their lack of branches and parallelisable nature.

A. Bitonic Networks & Even-odd networks

The sorting algorithms in this section are taken from Ken Batchers paper *Sorting Networks and their applications* [17]. These types of algorithms are well suited to a hardware implementation however there does exist redundant comparisons due to the *oblivious* sequence of comparisons.

1) Odd-even Merge Sort: If we start at the smallest merging network (a 1x1 merging network) we see it is simply a single comparison unit. A compare and swap unit in pseudocode form is illustrated in algorithm 1. A compare and swap unit compares two values and swaps them if necessary with the aim of guaranteeing a certain order. We can create larger networks by following the iterative rule for odd-even merging networks as described by Ken Batcher [17]. The pseudocode implemented in OpenCL is illustrated in algorithm 2 and is based on the circuit shown in figure 3.

To sort 2^p numbers using this algorithm requires $(p^2 - p + 4)2^{p-2} - 1$ comparison units.

2) Bitonic Sort: This sorting network relies on sorting a *bitonic* sequence. A sequence is called *bitonic* if firstly it is increasing and, after a certain point, is strictly decreasing. There exists an iterative rule (similar to odd-even merge sort) which permits the creation of larger networks. The sorting hardware for an 8 element bitonic sorter is illustrated in figure 4 and figure [13].

Algorithm 2 Odd-even Sorting Kernel

function ODDEVEN SORT(Data)
#Stage 0
compare_swap($Data[0], Data[1]$);
compare_swap($Data[2], Data[3]$);
compare_swap($Data[4], Data[5]$);
compare_swap($Data[6], Data[7]$);
.....
.....
#Stage 5
compare_swap($Data[1], Data[2]$);
compare_swap($Data[3], Data[4]$);
compare_swap($Data[5], Data[6]$);

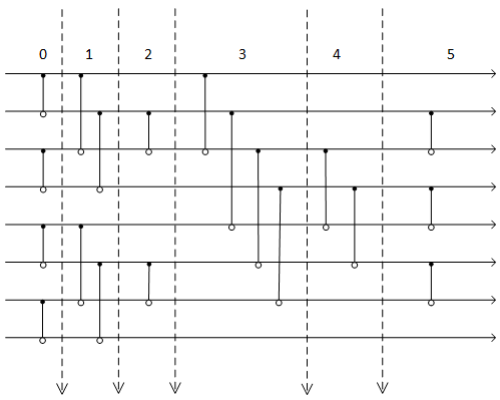

Fig. 3. 8 element odd even Network [18]. Connecting the lines are compare-swap units (illustrated in algorithm 1)

Using bitonic sort we need $(p^2 + p)2^{p-2}$ sorting elements to sort 2^p elements.

B. Merge Sort Trees

Merge Sort trees achieve the sorting computation by arranging compare-swap elements (see algorithm 1) into a tree based structure where each level of the tree sorts a growing number of elements. An example of a sorter tree is illustrated in figure 5. The pseudocode is illustrated in algorithm 4.

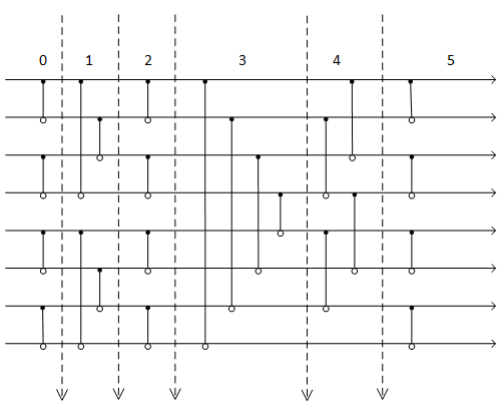

Fig. 4. Bitonic Sorter [13]

Algorithm 3 Bitonic Sorting Kernel

function BITONIC SORT(Data)
#Stage 0
compare_swap($Data[0], Data[1]$);
compare_swap($Data[2], Data[3]$);
compare_swap($Data[4], Data[5]$);
compare_swap($Data[6], Data[7]$);
#Stage 1
compare_swap($Data[0], Data[3]$);
compare_swap($Data[1], Data[2]$);
compare_swap($Data[4], Data[7]$);
compare_swap($Data[5], Data[6]$);
.....
.....
#Stage 5
compare_swap($Data[0], Data[1]$);
compare_swap($Data[2], Data[3]$);
compare_swap($Data[4], Data[5]$);
compare_swap($Data[6], Data[7]$);

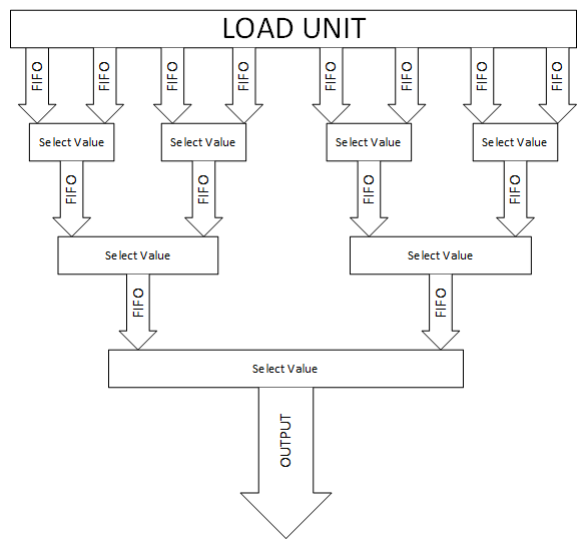

Fig. 5. Merge Sorter Tree [18]

Each level of the tree will merge two times the amount of data than the last level. A requirement is that the input to the lowest level of the tree is already sorted (the choice of sorting algorithm is left to the implementer).

C. Insertion Sorters

Hardware implementations of insertion sort make use of shift registers to store the search keys. The hardware presented in [18] and [19] illustrate this idea (pseudocode taken from [20] and illustrated in algorithm 5). The theory is that for each unsorted element a suitable location in the sorted array is located and the array is *shifted* to accomodate the new value. The value is then inserted into the sorted output array.

Algorithm 4 Merge Sorting Tree

```
MERGETREESORT(Input A, Output B);
FIFO level0[4], FIFO level1[2]
for i = 0 up to 8 do
    if a[i] < a[i+1] then
        level0[i/2].push(a[i])
        level0[(i/2)+1].push(a[i+1])
    else
        level0[(i/2)+1].push[a[i]
        level0[i/2].push(a[i+1])
    end if
end for
for i = 0 up to 4 do
    for j = 0 up to 2 do
        if level0[i].peek() < LEVEL0[i+1].peek() then
            level1[j].insert(level0[i].pop())
            level1[j].insert(level0[i+1].pop())
        else
            level1[j].insert(level0[i+1].pop())
            level1[j].insert(level0[i].pop())
        end if
    end for
    i = i * 2
end for
for i = 0 up to 8 do
    if level1[0].peek() < level1[1].peek then
        B[i] = level1[0].pop()
    else
        B[i] = level1[1].pop()
    end if
end for
```

Algorithm 5 Insertion Sort algorithm

```
INSERTIONSORT(Input A, Output B);
for i = 0 up to size of A do
    new_value = A[i]
    new_pos = 0, found_pos = false
    while notfound_pos do
        new_pos++
        if new_pos == output_size then
            found_pos = true
        else
            if B[new_pos] ≥ new_value then
                found_pos = true
            end if
        end if
    end while
    for index = output_size to new_pos + 1 do
        B[index] = B[index − 1]
    end for
    B[new_pos] = new_value
end for
```

D. Radix Sort

Radix sort is a type of bucket sort which sorts based on the radix (or base) of the data. In the case of integers, this algorithm sorts on each significant position. It requires the idea of a *position* as well as the direction of the sorting (i.e. least significant digit or most significant). The pseudocode of radix sort for integers using base 10 can be seen in algorithm 6.

Algorithm 6 Radix Sort algorithm

```
RADIXSORT(Input A, Number of digits d );
for j = 1 to d do
    count[10] = 0
    for i = 0 to sizeofA do
        count[key of A[i] in pass j] ++
    end for
    for k = 1 to 10 do
        count[k] = count[k] + count[k − 1]
    end for
    for i = n − 1 downto 0 do
        result[count[key of A[i]] = A[j]
        count[key of A[i]] − −
    end for
    for i = 0 to n do
        a[i] = result[i]
    end for
end for
```

IV. Experiment Configuration

The experiment is decomposed into four components; the hardware (FPGA), the development environment (OpenCL), the data to be sorted, and the model used to measure the power consumption. We briefly discuss these in this section.

A. Intel Arria 10

The development board used for this experiment was the Terrasic DE5a-Net [21] which contains the Intel Arria 10 GX[1]. The Arria 10 has 1150K logic elements, 427200 ALMs, 1.7 million registers, 3036 18x19 multipliers and 53 Mb of embedded memory.

B. OpenCL Development Environment

The Intel FPGA SDK for OpenCL supplied (version 16.1) with the Arria 10 supports the OpenCL 1.0 Standard. As FPGA synthesis is normally in the order of hours, Intel also provides an emulation environment which allows the developer to test their designs on a software simulation of the device. This allows the functionality of the design to be tested and debugged without the lengthy intervals between synthesis. Also, the SDK provides profiling tools to optimise the design.

The development flow for implementing the various sorting algorithms is as follows:

1) Create host and kernel code;
2) Iterate between testing on emulator and modifying code;

[1]Intel bought FPGA manufacturer Altera in 2015

3) Run Intel OpenCL Compiler. This step uses Intels Quartus HDL design software to create the final FPGA image (in the form of a *aocx* file);
4) Deploy to the device and execute the host code.

C. Data sets

The data provided to the hardware was generated in a very similar fashion to that discussed in [22]. Specifically, using a data set size of 8, we generated all combinations of 5 sets of data where each set contained 8 8-bit integers. This gave a benchmarking set of $(8!) * 5 = 201600$ data sets.

D. Power Measurement Model

There are a number of options available for analysing the power consumption of an FPGA design. Intel provide two software based mechanisms for this purpose. The *Early Power Estimator (EPE)* is a Microsoft Excel based tool which allows a user to input a number of variables and the spreadsheet will provide some approximate power consumption figures. The second software mechanism Intel provides is a part of the *Quartus* suite called *PowerPlay* which is more precise.

If power measurements are required rather than estimations (which is what the EPE and PowerPlay provide) there are a number of options available however they require external hardware. One example is illustrated in [23] where we can see a method for power analysis using an oscilloscope.

Our measurements were taken with a Racktivity PowerManager PM0816-01 power distribution unit. This PDU allows regular power consumption measurements via SNMP (Simple Network Management Protocol). The method we chose involved measuring the power consumption of the host systems plus the static power required to run the development board. Static power is the power consumed by the FPGA when no signals are toggling. Another required measurement is the host power consumption when actively transferring data to the device. Once we have these figures we can finally measure the entire system during sorting computation. The measurements were averaged over the time taken to sort 201600 data sets. This allows us to calculate the dynamic power consumption of the sorting designs. Dynamic power is the additional power consumed through the operation of the device caused by signals toggling.

V. Results

A. Resource Utilisation

The first results of interest are the resource consumption of the sorting algorithms. Table I shows the resources of the Arria 10 required to sort the input data. Note, this table does not include the resource consumption of the board interface as this is constant regardless of the the design. It is worthwhile examining the decisions made by the OpenCL compiler which influenced the resources required.

Algorithm	Resource			Power (mJ)	Fmax (MHz)
	ALUTs	FFs	RAMs		
Bitonic sort	11645	34213	268	36.00	340
Even-odd Network	11531	34119	268	31.15	366
Merge sort Trees	9186	8142	0	17.83	389
Insertion Sort	61805	190245	1170	135.10	242
Radix sort	23713	60651	414	29.87	373
Empty Kernel	1570	1685	0	N/A	412

TABLE I
Resource and Power Utilisation

1) Bitonic Sort & Even-Odd Network: The resources consumed by both of these kernels are very close. As expected this is due to the dependence on the *compare-swap* hardware. The total hardware resources required are almost a simple multiplication of the number of computations by the resource requirement of the *compare-swap* hardware. Approximately 75% of the ALUTs and FFs were actually consumed by the load and store operations used to copy the sorted data to the output buffer.

2) Merge Sort Trees: The HLD FPGA report generated as part of the compilation process provides some explanation as to why the resource usage of this sorter is relatively low. A number of the private variables used were implemented as barrel shifter with registers. Also, unlike the bitonic sort and even-odd network, it was possible to integrate the output buffer into the sorting process which removed the cost of the load and store operations.

3) Insertion Sort: As we can see from the utilisation table, this algorithm consumes more resources than the other algorithms. From the generated report it appears that this algorithm is clustered into 15 distinct blocks which can be run without stalls. Also, this algorithm has a greater iterative design than the other algorithms.

4) Radix Sort: This algorithm is relatively more complex in its approach to sorting data. The private variables were implemented as registers. A large amount of the ALUTs (approximately 16000) and FFs(approximately 45000) were consumed by memory copying.

5) Empty Kernel: The empty kernel was included to illustrate the overhead required for the kernel regardless of the computation involved. As we can see there is some cost in terms of ALUTs and FFs but no memory resources are required.

B. Power Consumption

As discussed earlier, our power measurement approach depends on a base measurement using an empty OpenCL kernel and through sorting the test data set we can calculate the energy required for sorting an eight input data set.

Table I also shows the power consumed by the sorting algorithms. It is immediately apparent that the *Merge sort tree* sorting algorithm requires less power to carry out an eight element sort. As noted previously, the HLD FPGA report states the FIFOs were implemented as barrel registers by the OpenCL compiler. This implementation used no RAMs.

It is also worth remarking on the high cost associated with using insertion sort. This algorithm was chosen due to the fact it should make use of hardware shift-registers however there is no evidence the OpenCL compiler implemented the shifts prior as shift-registers.

VI. Conclusion

In this paper we presented a power consumption comparison of five sorting algorithms using the Intel FPGA SDK for OpenCL and an Arria 10 FPGA. We chose algorithms which have been implemented previously in hardware.

The experiment results demonstrate that using an OpenCL implementation of merge sort trees has a significant benefit in both more efficient resource utilisation and reduced energy consumption than the other algorithms. Indeed, OpenCL merge sort trees consume 40% less power than the closest sorting algorithm (radix sort) when sorting 8 element data sets.

There are a number of different experiments left which are of interest. From an algorithm perspective broadening the number of sorting algorithms, or choosing more efficient implementations of the sorting algorithms (such as that presented in [24]), or designing power aware sorting algorithms are possible avenues. From a hardware point of view increasing the number of inputs to the sorting circuits, or improving the accuracy of the power consumption measurement approach, comparing OpenCL implementations with Verilog implementations [25], or augmenting other sorting algorithms with FIFOs to determine if a FIFO approach can improve power consumption are potentially interesting approaches.

VII. Acknowledgments

This paper has been supported in part by Intel Programmable Solutions Group, Racktivity, and SFI INSIGHT Centre for Data Analytics.

References

[1] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 94–101.

[2] Y. Yue, B. He, L. Tian, H. Jiang, F. Wang, and D. Feng, "Rotated logging storage architectures for data centers: Models and optimizations," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 203–215, 2016.

[3] I. Zecena, Z. Zong, R. Ge, T. Jin, Z. Chen, and M. Qiu, "Energy consumption analysis of parallel sorting algorithms running on multicore systems," in *2012 International Green Computing Conference (IGCC)*, June 2012, pp. 1–6.

[4] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[5] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," *International Journal of Data Science and Analytics*, vol. 1, no. 3, pp. 145–164, Nov 2016.

[6] B. Lopez and N. Cruz-Cortes, "On the usage of sorting networks to big data," in *Advances in Big Data Analytics: The 2014 WorldComp International Conference Proceedings. Mercury Learning and Information*, 2014, pp. 102–108.

[7] M. Rashid, L. Ardito, and M. Torchiano, "Energy consumption analysis of algorithms implementations," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2015, pp. 1–4.

[8] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, 2014, p. 4.

[9] A. Sirasao, E. Delaye, R. Sunkavalli, and S. Neuendorffer, "Fpga based opencl acceleration of genome sequencing software," *System*, vol. 128, no. 8.7, p. 11, 2015.

[10] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating computer vision algorithms using opencl framework on the mobile gpu-a case study," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 2629–2633.

[11] V. M. Morales, P.-H. Horrein, A. Baghdadi, E. Hochapfel, and S. Vaton, "Energy-efficient fpga implementation for binomial option pricing using opencl," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[12] N. Trevett, "Opencl introduction," *Khronos Group*, 2013.

[13] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[14] D. Connors, E. Grover, and B. Caldwell, "Exploring alternative flexible opencl (flexcl) core designs in fpga-based mpsoc systems," in *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:8.

[15] C. Grozea, Z. Bankovic, and P. Laskov, "Fpga vs. multi-core cpus vs. gpus: hands-on experience with a sorting application," in *Facing the multicore-challenge*. Springer, 2010, pp. 105–117.

[16] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, Feb. 2012.

[17] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[18] D. Koch and J. Torresen, "Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 45–54.

[19] R. Marcelino, H. Neto, and J. Cardoso, "Sorting units for fpga-based embedded systems," *Distributed Embedded Systems: Design, Middleware and Resources*, pp. 11–22, 2008.

[20] K. Ø. Arisland, A. C. Aasbø, and A. Nundal, "Vlsi parallel shift sort algorithm and design," *INTEGRATION, the VLSI journal*, vol. 2, no. 4, pp. 331–347, 1984.

[21] "Terasic de5a-net arria 10 fpga development kit (2017)."

[22] N. Zeinolabedini, G. Qin, D. Vasudevan, M. Schellekens, and E. Popovici, "Static average-case power analysis for sorting applications," in *Microelectronics (MIEL), 2012 28th International Conference on*. IEEE, 2012, pp. 397–400.

[23] F.-X. Standaert, L. v. O. tot Oldenzeel, D. Samyde, and J.-J. Quisquater, "Power analysis of fpgas: How practical is the attack?" in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 701–710.

[24] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 240–249.

[25] P. C. Petrut, A. Amaricai, and O. Boncalo, "Configurable fpga architecture for hardware-software merge sorting," in *Mixed Design of Integrated Circuits and Systems, 2016 MIXDES-23rd International Conference*. IEEE, 2016, pp. 179–182.