Impact of CUDA and OpenCL on Parallel and Distributed Computing

Abu Asaduzzaman, Alec Trent, S. Osborne, C. Aldershof
Electrical Engineering and Computer Science Department
Wichita State University
Wichita, Kansas, USA
Abu.Asaduzzaman@wichita.edu

Fadi N. Sibai
Department of Computer Engineering
Prince Mohammad Bin Fahd University
Al-Khobar, Saudi Arabia
fsibai@pmu.edu.sa

2021 8th International Conference on Electrical and Electronics Engineering (ICEEE) | 978-1-6654-4071-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICEEE52452.2021.9415927

***Abstract*—Along with high performance computer systems, the Application Programming Interface (API) used is crucial to develop efficient solutions for modern parallel and distributed computing. Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) are two popular APIs that allow General Purpose Graphics Processing Unit (GPGPU, GPU for short) to accelerate processing in applications where they are supported. This paper presents a comparative study of OpenCL and CUDA and their impact on parallel and distributed computing. Mandelbrot set (represents complex numbers) generation, Marching Squares algorithm (represents embarrassingly parallelism), and Bitonic Sorting algorithm (represents distributed computing) are implemented using OpenCL (version 2.x) and CUDA (version 9.x) and run on a Linux-based High Performance Computing (HPC) system. The HPC system uses an Intel i7-9700k processor and an Nvidia GTX 1070 GPU card. Experimental results from 25 different tests using the Mandelbrot Set generation, the Marching Squares algorithm, and the Bitonic Sorting algorithm are analyzed. According to the experimental results, CUDA performs better than OpenCL (up to 7.34x speedup). However, in most cases, OpenCL performs at an acceptable rate (CUDA speedup is less than 2x).**

Keywords—CUDA, high performance computing, OpenCL, parallel and distributed computing, performance analysis

I. Introduction

Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) are both application programming interfaces (APIs) that allow for cooperative central processing unit (CPU)/graphics processing unit (GPU) parallel program execution through what is known as a kernel, or a section of code compiled with the express purpose of running on a coprocessor such as a GPU [1]. OpenCL is an open standard, being maintained by the Khronos Group, which is composed of members such as AMD, Google, Apple, Qualcomm, and others [2], [3]. CUDA is a proprietary Nvidia property that performs a similar task but with one additional condition: it is exclusive to Nvidia GPU cards only. On an Nvidia GPU, some performance gains are to be expected due to it being a complete in-house solution. OpenCL, by virtue of its name, is designed to be an open, platform agnostic standard. Naturally, this has certain implications within the realm of high performance computing. The questions is: does OpenCL perform to at least an acceptably close standard to CUDA on the same device running the same application?

Both CUDA and OpenCL, kernel-based APIs, have a similar workflow of first moving data from the host to the computation device, then instructing the computing device on what to do through the kernel, then having the GPU perform the computing task in parallel, then finally copying the data back from the computing device's memory into main memory. However, they are somewhat different in their implementation. With CUDA, the four steps listed above are done with functions such as cudaMalloc and cudaMemcpy (analogous to C's malloc and memcpy functions) as well as a new piece of syntax – triple arrowheads <<<x, y>>>, where x, the number of CUDA blocks per kernel grid, represents the total number of Streaming Multiprocessors (SMs) to send the code to and y, the number of CUDA threads per block, is the total number of threads to run on the said SMs. cudaMemcpy works in both ways, from host to computing device and from computing device to host [4].

OpenCL works on similar principles, but it takes more steps to perform the task by virtue of OpenCL being a more platform-agnostic standard. While writing CUDA code can be thought of as a four-step process, writing OpenCL code is more of a six or seven step process. First we must initialize the runtime ourselves as opposed to CUDA where that is done for us automatically. This is done by creating a context, or an object that is "used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context" [5]. The specific command for this is clCreateContextFromType(). CUDA does this automatically, but in this way, OpenCL allows the programmer to use a heterogeneous mix of devices. Once the context is initialized, we obtain OpenCL-compatible device identifications (IDs) with clGetDeviceIDs(). Finally, we create the aforementioned command queue with clCreateCommandQueue with our context and device ID as arguments. From there, we create buffers in much the same way we do on CUDA, with the command clCreateBuffer. The kernel is treated as an object, and is assigned to an object by clCreateProgramWithBinary, and is then encapsulated with clBuildProgram to have all pertinent information for running on the particular computing device. Bringing data to the computing device is done with clEnqueueWriteBuffer. clCreateKernel finalizes the creation of the kernel with arguments such as the name of the kernel and the built program with clBuiltProgram. We then set some kernel arguments such as buffer locations and sizes with clSetKernelArg. Finally, we can run the kernel with clEnqueueNDRangeKernel, which as the name implies, puts the

978-0-7381-1357-9/21/$31.00 ©2021 IEEE

kernel into a queue to run on the device. After the kernel is done running, we bring the data back into main memory with clEnqueueReadBuffer, and the task is finally done.

CUDA is certainly a more convenient solution for Nvidia GPU cards. However, convenience is not always everything, as it can lead to lazy programming and a lack of understanding of the concrete processes that go into accelerated computing. In this paper, we study both the CUDA and OpenCL APIs to understand whether CUDA API performs substantially better than OpenCL API.

This paper is organized as follows: Section II reviews related work. The selected algorithms and experimental details are discussed in Sections III. Experimental results are presented in Section IV. The paper is concluded in Section V.

II. Related Work

While comparing CUDA and OpenCL APIs, it is useful to consider what software currently supports either or both, as well as any existing performance metrics for either or both. A short list of video editing and other multimedia programs have been listed on the website create.pro, in which the majority of programs either do not list OpenCL support, or the feature set described under OpenCL compatibility is less than what is offered under CUDA (i.e., for Red Redcine-X, CUDA supports two GPU cards whereas OpenCL only supports one) [6]. Final Cut Pro X only supports OpenCL, but given that Final Cut Pro is an Apple property, and that operating system (OS) X is heavily based on OpenCL for its graphics backend (or more accurately Metal, a combination of the OpenCL and OpenGL APIs) [7], this would make sense. It could easily be argued that the prevalence of CUDA's features could simply be a marketing ploy, or it could also be the hitherto seen ease of CUDA coding compared to OpenCL.

In the realm of deep learning, two popular deep learning APIs PyTorch and TensorFlow both only support CUDA for GPU acceleration [8]. Part of the reason for PyTorch only supporting CUDA in the mainline instead of OpenCL seems to be Advanced Micro Devices' (AMD's) push for another computing language API: ROCm (Radeon Open Compute), which PyTorch AMD (an official branch of PyTorch) runs on. The classic XKCD comic regarding standards comes to mind. There was no official statement from Google on why TensorFlow only runs on CUDA for the GPU branch.

A prior study was done utilizing Adiabatic QUantum Algorithm (AQUA), which is "a Monte Carlo simulation of a quantum spin system written in C++" [9]. The Monte Carlo simulation itself is another naturally parallel problem as it is the averaging of many random guesses, so it is a sort of analog to the Mandelbrot set generator to be used in this paper. This paper found that "CUDA performed better when transferring data to and from the GPU" and that "CUDA's kernel execution was also consistently faster than OpenCL, despite the two implementations running nearly identical code" [9].

According to creat.pro, Nvidia GPUs with built in CUDA support as well as strong OpenCL performance are the best options for professionals. They, however, recommend an AMD GPUs when the users are using applications that support OpenCL and have no CUDA option [1]. It is shown that OpenCL's portability does not vitally affect its performance, and OpenCL can be a good alternative to CUDA [10].

Do et al. introduce a benchmark suite that provide code parallelization/optimization techniques in both OpenCL and CUDA for modern GPUs [11]. Rasch et al. introduce a high-level OpenCL/CUDA Abstraction Layer that simplifies the development of host code [12]. Based on our literature review, there are strengths in both CUDA and OpenCL APIs. Therefore, we find it important and interesting to explore CUDA and OpenCL for parallel and distributed computing.

III. Algorithms and Experiments

In this section, we describe the algorithms selected and experimental details to implement the selected algorithms using the CUDA and OpenCL APIs.

A. Algorithms Selected

Three algorithms were chosen for a mixture of ease of implementation as well as an (at the very least perceived) ease of parallelizability – the crux of GPU accelerated computing. First is the generation of the Mandelbrot set, a so-called "embarrassingly parallel" or "naturally parallel" problem in which "almost no communication between the processes [is required]" and that "the computation of the sub-problems is completely independent" [5]. Generating the Mandelbrot set is naturally parallel because we are computing points in a set that are completely independent of one another. Because of this, we can easily maximize the usage of the computing device's computational resources, something that would naturally take substantially longer in the single-threaded paradigm. The two code samples we used were from the same user on Github who at the very least claimed these were "very similar on purpose" [13], [14]. It is worth noting that we had to modify the code to remove some debugging code–the check_succeeded function–that caused it to never run (as well as update the OpenCL header file naming scheme from <OpenCL/OpenCL.h> to the more modern <CL/cl.h>), but neither of these should have any great impact on performance.

The second algorithm we chose is the Marching Squares. This is a way of generating a contour line from a two dimensional (2D) scalar field. Essentially, this line will contain all of the values in the scalar field of the same value. This is another naturally parallel problem since the algorithm can be applied to each unit area in the field separately. Various boundary shapes can be applied to generate said boundary, and can be as a rectangle or as complex as a pentagon. In 3D, this problem is known as the Marching Cubes algorithm. The 3D version in particular is useful for tasks such as mapping and generating solids from collections of 3D points, such as what a Magnetic Resonance Imaging (MRI) machine does to generate its final model. Both of these are direct samples of Nvidia code [15], [16].

Finally, we chose to work with Bitonic sort, which is similar to merge sort in concept, but unlike merge sort, benefits strongly from parallelization. Bitonic sorting algorithm represents distributed computing. Unlike traditional sorting algorithms, Bitonic sort is based on the idea of a sorting network; a series of comparisons that are constructed algorithmically based on the size of the dataset. The sorting network itself is a series of 'n'

lines, each representing each individual data point, and a series of comparisons along those lines constructed algorithmically. This sorting network structure is what allows it to be parallelizable, as each line in the network could be assigned to a thread, which if we consider each value to be sorted as a separate line, the speedup is massive. In fact, Bitonic sort has a worst case complexity of O(log log n), with the tradeoff that it takes O(n log log n) space complexity in the worst case [17]. Considering that merge sort's worst case time is O(n log n) [18], this performance increase is substantial. It is worth noting that this problem is not naturally parallel, as the threads do have to sync up before the comparison is done, otherwise errors such as improper comparisons and overwriting of data could happen. Unlike the prior two cases, these two samples of code were obtained from different sources, so if there is any one test worth being skeptical of, it would be this one [19], [20].

B. Experimental Details

We perform experiments on a Linux-based HPC machine running CUDA (version 9.x) and OpenCL (version 2.x). This computer uses an Intel i7-9700k processor and an Nvidia GTX 1070 GPU. All three programs are run five times to get an average. The time to run the Mandelbrot set and Bitonic sort algorithms is recorded, but for the marching cubes problem, we record the average frames per second after 10 seconds of running the program. This is because the marching cubes program doesn't complete, and is instead a looping simulation. The Mandelbrot set times are obtained by using a 4096x4096 grid and a 25,000x25,000 grid to generate all the points, and the Bitonic sort as done on an array of 2^{20} elements and an array of 2^{24} elements. Our Marching Squares simulation could only be measured in the metric of frames per second, so we are only able to run it in one instance. Such large values for the Mandelbrot set were chosen to determine how longer-term calculations affected performance time, since at first we discover that our tests that seemed rather large only take a few seconds at most.

In more rigorous detail, this experiment is performed on Arch Linux using kernel 5.6.10-arch1-1. Arch Linux is a distribution of Linux that aims to have as up to date packages as reasonable, as well as keeping the packages as close to the upstream release as possible. This gives arguably the best possible chance for these pieces of code to run. The GPU used for this is the EVGA GeForce GTX 1070 SC 8GB GPU. The CPU used is an Intel i7-9700k with stock base frequency (3.60 GHz) and stock turbo frequency (4.90 GHz). System memory is 4x8GB sticks of Corsair Vengeance LPX 3200MHz RAM. As for the rest of the software, Nvidia's closed source driver version 440.82-11 is used, as is the opencl-nvidia package version 440.82-1. CUDA version 10.2.89-5 is used during this experiment. All of these tests are performed in an X server on a two head setup (one 144Hz, one 60Hz), so it is reasonable to assume there is some impact on the magnitude of the results. However, since what is of more concern is the relative standing of the values, this should be negligible. As for what else is running at the time, a fairly minimal setup is used with very few programs running at the time of running. No major changes were done about what is running at the time between each test.

IV. Results and Discussion

First, we discuss the results due to Mandelbrot Set generator that creates complex numbers. We generate five different sets using the CUDA and OpenCL APIs. The time required for the CUDA and OpenCL APIs for generating the 4096x4096 Mandelbrot Set is shown in Table I. The average time for CUDA and OpenCL APIs are 0.387 and 0.631 seconds, respectively. Therefore, the average CUDA speedup with respect to OpenCL is 1.66x. Table II shows the time required by the CUDA and OpenCL APIs for generating the 25,000 x25,000 Mandelbrot sets. Here, the average time for CUDA and OpenCL APIs are 15.367 and 26.371 seconds, respectively. Therefore, the average CUDA speedup with respect to OpenCL is 1.72x.

TABLE I. Results due to Mandelbrot Set 4096x4096 Grid

CUDA Time (s)	OpenCL Time (s)	CUDA Speedup
0.455	0.644	1.42x
0.380	0.580	1.53x
0.361	0.665	1.84x
0.356	0.642	1.80x
0.381	0.623	1.64x

TABLE II. Results due to Mandelbrot Set25,000x25,000 Grid

CUDA Time (s)	OpenCL Time (s)	CUDA Speedup
15.856	28.132	1.77x
13.162	25.326	1.92x
15.814	26.186	1.66x
16.642	25.842	1.55x
15.359	26.369	1.72x

During Mandelbrot set, the CUDA version was able to handle our attempt to run the higher resolution set at 32,768x32,768, but the OpenCL version had a segmentation fault. The performance difference here definitely is pronounced, but this could be simply due to CUDA-based optimizations. As well, perhaps the Mandelbrot set is one of the benchmarks that these CUDA cores were optimized around given its ubiquity as a demonstration as well as its relative ease of coding. It is worth noting that some performance impact could have been because of writing the resulting bitmap image to disk, but at the very least this extends to both the OpenCL and CUDA experiments, so it should hopefully average out. There was no deletion of the resulting image between runs. Since this image gets rather large at the higher end of the scale (approximately two gigabytes for the 25,000 x25,000 case), the performance of the drive being written to could have some impact. However, the hard drive being written to was a solid state device (SSD, a Samsung 840 Pro 256GB SSD with a peak write speed of 550MB/s), so at most that would have added a few seconds of write time. There was an NVMe SSD onboard with peak write speeds of 3.5GB/s, but it was not felt to be urgent enough of an issue to rerun everything (after all, a logical extreme would be writing directly to a RAMdisk or something if the absolute maximum of maximum performance was desired). Something that we didn't have control over was whether or not the same data was being written to the same sectors (since the resulting Mandelbrot set image was the same every time), and whether the disk controller would apply optimizations to speed up the writing of data that was already there on the drive but unassociated to any file. Drive

caching is also another thing we do not have any control over. While the average case should hopefully take care of this, it is at the very least worth pointing out (that the CUDA gains 1.66x speedup over OpenCL).

Second, we discuss the results due to the marching squares algorithm that represent naturally parallelism. The marching squares simulation (i.e., the marching cubes in this study) using CUDA for the default shape provided by Nvidia performed at 1,686 frames per second (FPS) in the first trial. For OpenCL, it performed at 1,239 FPS. FPS due to the CUDA and OpenCL APIs for the marching cubes is shown is Table III. The average FPS for CUDA and OpenCL APIs are 1,708 and 1,291, respectively. Therefore, the average CUDA speedup with respect to OpenCL is 1.32x. It is worth noting that unlike the Mandelbrot Set, the marching cubes is somewhat dependent on the X server performance, and in order to get the actual value that was not restricted by any sort of vertical sync, the argument __GL_SYNC_TO_VBLANK=0 had to be given before the command was executed. This may not be an issue on Windows, but it is on Linux.

TABLE III. RESULTS DUE TO MARCHING SQUARES

CUDA (FPS)	OpenCL (FPS)	CUDA Speedup
1,686	1,239	1.36x
1,688	1,332	1.27x
1,785	1,303	1.37x
1,658	1,284	1.29x
1,724	1,297	1.33x

Finally, we discuss the results due to the Bitonic sort algorithm that represents distributed computing. The Bitonic sort is performed first on a set of 2^{20} (i.e., 1,048,576) elements. The time required to execute the Bitonic sort implementations in CUDA and OpenCL is shown in Table IV. The average time for CUDA and OpenCL are 1.08 and 7.93 mseconds, respectively. As a result, the average CUDA speedup with respect to OpenCL is 7.34x. Finally, the Bitonic sort on a set of 2^{24} (i.e., 16,777,216) elements is performed. The time required by CUDA and OpenCL is shown in Table V. The average time for CUDA and OpenCL are 92.20 and 174.21 mseconds, respectively. As a result, the average CUDA speedup with respect to OpenCL is 1.89x. It is noticed that the performance of the OpenCL API in the case of 2^{20} elements is really bad, and for the case of 2^{24} elements it is much better. There are a few arguable reasons for this. First, perhaps the OpenCL code simply works better at larger scales. The performance of Bitonic sort is O(log (log n)), so perhaps this particular implementation simply has a running time function closer to the worst case function compared to the CUDA implementation. This is the only one in which the implementations had two different authors.

TABLE IV. RESULTS DUE TO BITONIC SORT 2^20 ELEMENTS

CUDA Time (ms)	OpenCL Time (ms)	CUDA Speedup
1.25	8.07	6.46x
1.01	7.89	7.82x
1.02	7.89	7.74x
1.03	7.87	7.64x
1.07	7.85	7.43x

TABLE V. RESULTS DUE TO BITONIC SORT 2^24 ELEMENTS

CUDA Time (ms)	OpenCL Time (ms)	CUDA Speedup
92	174.178	1.88x
95	174.376	1.84x
89	174.113	1.96x
93	174.180	1.87x
92	174.202	1.89x

Another argument would be that perhaps a particularly intensive process was running at the time and was coloring the results. This one is at the very least easy to rebuke, as we run it multiple times and still got the same results. Perhaps the most compelling argument would be that it is simply down to data pipeline speeds from main memory to GPU memory and GPU memory to main memory. It was mentioned in the paper that performed analysis on the AQUA algorithm that "CUDA performed better when transferring data to and from the GPU" [9], after all. Since it is such a fast experiment, perhaps the relative slowness of OpenCL's data pipeline becomes more evident. This is strengthened by the fact that the 2^{24}-element Bitonic sort experiment performs much in line with the past tests. While the actual amount of data being transferred in both cases in the 2^{20}-element Bitonic sort experiment was 32 Megabytes (512 Megabytes for the 2^{24} Bitonic sort experiment), it could be a matter of the time required to create that pipeline making that performance difference in the smaller case. This is important to look into, as applications which require many small kernels may have their performance impacted by a potentially much slower data pipeline. It is worth noting as well that the OpenCL implementation performed its operations on an array of unsigned integers while the CUDA implementation performed its operations on floating point values. At the very least for the larger scale Bitonic sort, both the CUDA and OpenCL performance was more in line with what was expected.

The average CUDA speedup with respect to OpenCL due to five different implements of Mandelbrot Set generation, Marching Squares algorithm, and Bitonic Sort is summarized in Table VI. It is noticed that for the Mandelbrot set generation, the CUDA speedup over OpenCL increases as the grid size increases from 4096x4096 to 25,000x25,000. However, for the Bitonic Sort algorithm, the CUDA speedup over OpenCL decreases as the element size increases from 2^{20} to 2^{24}.

TABLE VI. AVERAGE CUDA SPEEDUP

Application	CUDA Speedup
Mandelbrot Set 4096x4096 grid	1.66x
Mandelbrot Set 25,000x25,000 grid	1.72x
Marching Squares	1.32x
Bitonic Sort 2^20 elements	7.34x
Bitonic Sort 2^24 elements	1.89x

V. CONCLUSION

CUDA and OpenCL both APIs have their merits and conveniences while performing parallel and distributed computing. In this work, we study OpenCL and CUDA, conduct experiments on Nvidia GPU cards using the Mandelbrot Set generation, the Marching Squares algorithm, and the Bitonic

Sorting algorithm and compare the performance of CUDA and OpenCL. The Mandelbrot Set generator produces complex numbers (in a naturally parallel method), the Marching Squares algorithm exemplifies embarrassingly parallelism, and the Bitonic Sorting algorithm characterizes distributed computing. Experimental results show that CUDA outperforms OpenCL (speedup up to 7.34x) on Nvidia hardware for all tests, something makes sense (Nvidia hardware is optimized for CUDA). It is observed that OpenCL performs at an acceptable rate; most cases the CUDA speedup is less than 2x. The results may differ if a non-Nvidia GPU card such as an AMD GPU card is used.

CUDA is best for maximum performance with Nvidia GPU cards and a homogenous device environment, whereas OpenCL is good for a heterogeneous and more portable environment. CUDA is relatively easy to code for compared to some of the overhead with OpenCL and juggling various devices. For its broad-reaching and powerful extensibility, OpenCL has its own value in parallel and distributed computing.

REFERENCES

[1] "OpenCL vs. CUDA: Which Has Better Application Support?" 2017. https://create.pro/opencl-vs-cuda/

[2] J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," Computing in Science & Engineering, Vol. 12, No. 3, pp. 66-73, 2010. DOI: 10.1109/MCSE.2010.69

[3] "Khronos Members," The Khronos Group, 2020. https://www.khronos.org/members/list

[4] "CUDA C Programming Guide," NVIDIA Developer Documentation, 2019. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html# device-memory

[5] "clCreateContext," 2020. https://www.khronos.org/registry/OpenCL /sdk/1.0/docs/man/xhtml/clCreateContext.html

[6] "OpenCL vs CUDA," 2020. https://create.pro/opencl-vs-cuda/

[7] "Metal (API)," Wikipedia, 2020. https://en.wikipedia.org/wiki/ Metal_(API)

[8] N. Dimolarov, "On the state of Deep Learning outside of CUDA's walled garden," 2019. https://towardsdatascience.com/on-the-state-of-deep-learning-outside-of-cudas-walled-garden-d88c8bbb4342

[9] K. Karimi, N. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," 2010. https://arxiv.org/abs/1005.2581

[10] J. Fang, A. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," International Conference on Parallel Processing. IEEE, Taipei City, Taiwan, 2011. DOI: 10.1109/ICPP.2011.45

[11] Y. Do, H. Kim, P. Oh, D. Park, and J. Lee, "SNU-NPB 2019: Parallelizing and Optimizing NPB in OpenCL and CUDA for Modern GPUs," IEEE International Symposium on Workload Characterization (IISWC), 2019.

[12] A. Rasch, M. Wrodarczyk, R. Schulze, and S. Gorlatch, "OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA," IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2018.

[13] "GitHub - benhiller/opencl-mandelbrot: A very simple OpenCL program to generate an image of the Mandelbrot set," 2020. https://github.com/benhiller/opencl-mandelbrot

[14] "GitHub - benhiller/cuda-mandelbrot: A very simple CUDA program to generate an image of the Mandelbrot set, very similar to the OpenCL program on purpose," 2020. https://github.com/benhiller/cuda-mandelbrot"

[15] OpenCL – Nvidia Developer," 2020. https://developer.nvidia.com/opencl

[16] "CUDA Zone - Nvidia Developer," 2020. https://developer.nvidia.com/ cuda-zone

[17] D. Vrajitoru, "Embarrassingly Parallel Algorithms," 2020. https://www.cs.iusb.edu/~danav/teach/b424/b424_23_embpar.html

[18] "Bitonic sorter," 2020. https://en.wikipedia.org/wiki/Bitonic_sorter

[19] "GitHub – Gram21/GPUSorting: Implementation of a few sorting algorithms in OpenCL," 2020. https://github.com/Gram21/GPUSorting

[20] "Bitonic Sort on CUDA," 2020. https://gist.github.com/mre/1392067