UC Davis
UC Davis Previously Published Works

Title
FPGA versus GPU for Speed-Limit-Sign Recognition

Permalink
https://escholarship.org/uc/item/8ww3d2gg

Authors
Yih, Matthew
Ota, Jeff
Owens, John
et al.

Publication Date
2018-09-09

eScholarship.org Powered by the California Digital Library
University of California

FPGA versus GPU for Speed-Limit-Sign Recognition

Matthew Yih*[1], Jeffrey M. Ota[2], John D. Owens[1], and Pınar Muyan-Özçelik*[3]

***Abstract*— We implement a speed-limit-sign recognition task using a template-based approach on the FPGA using the Intel FPGA SDK for OpenCL. Then we evaluate its throughput, power consumption, accuracy, and development effort against a GPU implementation that is based on a system presented in our previous study. This paper also discusses implementation differences between the FPGA and GPU systems, provides a methodology for translating the GPU system to the FPGA system, and explains optimizations used in the FPGA version. While implementing the FPGA system, we build an efficient FFT engine for image processing on the FPGA which can be utilized by other developers to perform related tasks. In this paper, we also provide our insights on building the FPGA versus GPU system, which we hope can be useful for designing upcoming versions of FPGA-focused OpenCL development environments. We conclude that the FPGA implementation provides better power consumption for the same detection accuracy, while the GPU supports better programmer efficiency.**

***Index Terms*— FPGA, GPU, Autonomous Vehicle, FFT, Speed-Sign detection**

I. INTRODUCTION

As self-driving cars are becoming a reality, we increasingly see a need for efficiently implementing different types of autonomous driving (AD) tasks. Most AD applications require high efficiency (e.g., image processing applications such as road sign recognition). This efficiency can be achieved by heterogeneous computing systems that provide parallel processing. Heterogeneous computing provides different software and hardware alternatives for implementing AD tasks. Hence, understanding advantages and trade-offs of these alternatives is essential for implementing efficient AD applications.

GPUs are frequently used in heterogeneous systems to provide the desired parallel processing power. More recently, FPGAs are proving to be competitive in some aspects, especially in energy efficiency. As high-level programming tools for FPGAs such as OpenCL [11] began to mature, FPGA programming effort may become comparable to that of GPU programming.

This paper presents a system-level evaluation of building an FPGA platform for an autonomous driving task that involves image processing and compares the performance of this FPGA implementation to that of the GPU implementation. Specifically, we have implemented a template-based speed-limit-sign recognition algorithm using OpenCL on an FPGA and compared it with the CUDA [13] implementation of the same algorithm running on a GPU. We discuss the implementation differences between the FPGA and GPU systems that stem from architectural differences as well as the optimizations used in the FPGA, which present understudied areas of research in the field as will be further explained in Section II. While discussing the implementation differences, we also provide the design flow to translate a GPU/CUDA system to an FPGA/OpenCL system. The methodology can be utilized by future studies that require similar translation. Then, we compare the performance and trade-offs of the FPGA and GPU implementations in terms of throughput, power consumption, accuracy, and development effort. We choose these performance indices because throughput represents the computing ability of the real-time AD system; power consumption is an essential attribute for AD tasks as the power on a vehicle is limited; accuracy verifies the correctness of the systems (i.e., with the same approach and same dataset, we should achieve similar accuracy results); and development effort reflects the time and cost required to build such a system. In order to provide a thorough power comparison, we report not only the whole PCIe card power as in previous studies, but also the processor's power which eliminates power overhead from the board. We believe comparing the processor power is meaningful since it scales up proportionally as the design grows larger.

While developing the FPGA implementation, we have realized that currently there is no efficient FFT library for image processing on FPGAs. Hence, we improve Intel Altera's OpenCL FFT design to build an efficient FFT engine. This engine can be utilized by other developers to perform image processing tasks. In this paper, we also provide our insights on building the FPGA versus GPU system, which we hope can be considered while designing upcoming versions of FPGA software development kits.

To sum up, contributions of this study include: comparing the difference between the implementation and performance of the FPGA system and that of the GPU system; developing a methodology to translate a GPU/CUDA system to an FPGA/OpenCL system; building an efficient FFT engine for image processing using OpenCL; and providing our recommendations on the Intel FPGA SDK for OpenCL based on our experience.

The remainder of this paper is organized as follows: Section II briefly reviews related work which is followed by the presentation of background in Section III. Section

*Corresponding authors

[1]Matthew Yih and John D. Owens are with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616 {myih, jowens}@ucdavis.edu

[2]Jeffrey M. Ota is with the Autonomous Driving and Sports Research Group, Intel Labs, Santa Clara, CA 95054 jeffrey.m.ota@intel.com

[3]Pınar Muyan-Özçelik is with the Department of Computer Science, California State University, Sacramento, CA 95819 pmuyan@csus.edu

IV provides implementation of our systems and the design flow we use to translate the GPU/CUDA system to the FPGA/OpenCL system. This section also explains optimizations used in the FPGA system and introduces the efficient FFT engine we develop. Next, experimental settings, results, discussion, and insights (Section V), are given. Finally, we conclude in Section VI.

II. Related Work

The template-based speed-limit-sign recognition approach we utilize in this study is based on our previous work [12], which performs a GPU-based real-time recognition with limited hardware and power. By performing a series of correlations in the frequency domain, this approach achieved real-time speed-sign detection and classification on a low-end GPU with over 90% accuracy on 45 minutes of video footage. In this study, we have updated the GPU implementation presented in our previous work to make it compatible with the newer GPU architecture and latest CUDA version. Our FPGA implementation is also using the template-based approach presented in this prior work (which we detail in Section IV).

OpenCL-related development tools for FPGAs are relatively new and are growing to maturity only in recent years. Hence, only a limited number of related studies compare GPU and FPGA programming approaches that use high-level development tools. One such study is presented by Zohouris et al. [15], who present a comparison between GPU and FPGA using the Rodinia benchmark suite [1] programmed in CUDA and OpenCL, respectively. By optimizing OpenCL code for the FPGA, they achieved 3.4 times better power efficacy using an Intel Altera Stratix V in comparison to an NVIDIA K20c. Their focus of power and performance on the FPGA versus GPU is similar to our study. While they used open-source benchmarks designed for heterogeneous computing and focused on numbers, we instead deploy an autonomous vehicle application with contexts and scenarios as benchmarks and address how to construct such a platform in detail.

III. Technical Background

Real-time object recognition is one of the fundamental functions of autonomous vehicle systems as it is the main input to the system to collect information from the real world. In this work, we focus on a template-based speed-limit-sign recognition algorithm and implement it on two different types of platforms. In this section, we provide background on template-based speed-limit-sign recognition and the Intel FPGA SDK for OpenCL platform.

A. Template-Based Real-Time Speed-Limit-Sign Recognition

This speed-sign-recognition approach uses speed-sign images as templates and computes a series of correlations between these templates and input frames in the frequency domain. FFT correlation is done by taking the FFTs of both the template and input frame, then multiplying the complex conjugate of both, then performing inverse FFT on the result. Matching in the frequency domain allows us to compute the result with only one FFT multiplication as opposed to performing matching in the spatial domain, which involves several convolution-type operations. Hence, performing matching in frequency domain is efficient and fits the requirement of low-power and real-time processing. Another benefit of FFT correlation is that it allows us to perform additional processing to improve matching (e.g., *k*th-Law).

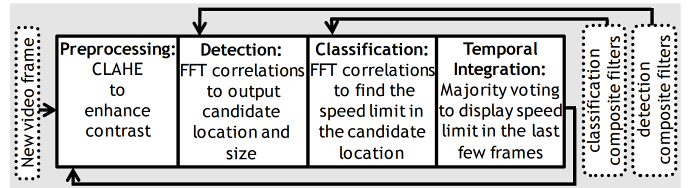

Fig. 1. Speed-sign recognition pipeline

The template-based speed-sign-recognition pipeline presented in our prior study [12] has pre-processing, detection, classification, and temporal integration stages, as illustrated in Fig. 1. The computation-intensive stages are the detection stage and the classification stage. In the detection stage, we use speed-limit signs 00 and 100 to generate a composite template to detect two-digit and three-digit speed signs. The detection stage returns the position of the potential speed-limit sign position. Then, based on that information, the region of the image will be passed to the classification stage. The classification stage has a different set of composite templates consisting of all possible speeds to recognize the specific number. When consecutive frames output the same classification result, the system will conclude that the result is valid.

The template-based pipeline returns a 90% accuracy with no false positives. With video size of 640x240, it yields real-time performance of 18 fps on low-end GPU GeForce 9600M GT.

B. Intel FPGA SDK for OpenCL

OpenCL is a cross-platform API for writing programs for heterogeneous computing devices such as CPUs, GPUs, FPGAs, and the like. Unlike CUDA, which only targets NVIDIA GPUs, OpenCL does not target any specific vendor or hardware type and allows developers to migrate their designs from one type of OpenCL-compatible hardware to another with minimal modification.

The Intel FPGA SDK for OpenCL is a C-based programming suite that compiles OpenCL kernel code to an FPGA bitstream without having to program in a low-level language such as Verilog. The OpenCL execution model for Intel FPGA is illustrated in Fig. 2.

Multi-threading is an important part of the OpenCL programming model and it is implemented differently on FPGAs as opposed to GPUs. Because of the nature of FPGAs, it is not realistic to run a massive number of parallel threads on FPGAs in the same way as on GPUs. The Intel FPGA SDK

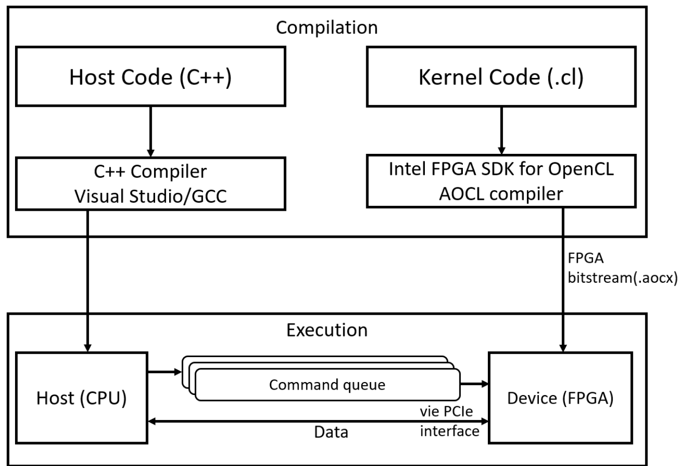

Fig. 2. Intel FPGA SDK for OpenCL execution model

issues the threads iteratively through the kernel pipeline to save hardware resources and achieve pipeline parallelism. Full parallel computing across threads is still possible by making duplicate computing units in the expense of hardware resources. Another major difference of the Intel FPGA SDK from GPU SDKs is the channel function. The Intel FPGA SDK allows kernels to communicate with each other through predefined channels, which eliminates the latency of reading and writing data to global memory. Similar kernel communications are very limited in a GPU implementation.

In summary, writing an OpenCL program based on an existing CUDA program and targeting FPGA is not a straightforward task because these two are running on different execution models at instruction level. Therefore, programs cannot be transformed into OpenCL mechanically. In order to optimize the FPGA's energy consumption and computing efficiency, we need to have smartly written OpenCL kernels that are dedicated to achieve these goals.

More details on multi-threading, channels, and writing FPGA-efficient OpenCL code are presented in Section IV.

IV. APPROACH AND IMPLEMENTATION

We modify the template-based speed-limit sign recognition implementation presented in our previous study [12] to make it compatible with the NVIDIA GTX1060 GPU and CUDA 9.0. We also implement the same approach on an Intel Arria 10 GX FPGA using the Intel FPGA SDK for OpenCL 17.1. Both implementations are developed on Windows OS and used an Intel Core i7 CPU as the host device. For our comparison, we choose to use the Arria 10 and GTX1060 due to their following similarities: they are both mid-range models in their product lines and both PCIe-based. To implement the same approach on the FPGA, we need to translate kernel code from CUDA to OpenCL. If the OpenCL code would also need to be run on the GPU, this translation would be simple and intuitive. However we need to run the OpenCL code on the FPGA, which complicates the translation of the kernel code: due to the architectural differences between the devices, such as the number of processing cores and memory model, optimized code for the GPU will not be efficient on the FPGA and vice versa. Our design and optimization of FPGA kernels are presented in the following subsections. We also provide implementation details of the efficient FFT engine we have developed at the end of this section.

A. Design flow

Our design flow of translating the GPU/CUDA system to FPGA/OpenCL system is shown in Fig. 3. Image processing kernels written in CUDA aim to use massive parallelism, which might be inefficient for the FPGA due to architectural differences. According to Intel's programming guide, because of the FPGA architecture, the FPGA is most suitable for coarse-grained parallelism in complex applications. We rewrite the kernels in a serial fashion for the first step. For example, a parallel reduction for finding a maximum value would be an inefficient algorithm on the FPGA since half of the hardware resources are not used on every new iteration. Thus, the kernel should be rewritten to use multiple *linear search* kernels to better suit the FPGA, as implemented in our design. In addition to reduction, we rewrite other kernels which perform complex point multiplication (used in FFT multiplication), *k*th-Law (used to improve template matching performance), and conversion between char4 and float types (used in the pre-processing stage). We change the kernel execution model from parallel execution to serial execution of multiple copies while the arithmetic operations remain the same.

CUDA code that runs on the GPU and OpenCL code that runs on the FPGA can be called by the same host and executed on both devices simultaneously. This allows us to conduct the conversion from CUDA to OpenCL one kernel at a time, which simplifies the development process. Frequent modification may be needed during the early phase, and compiling for the FPGA may take hours each time. Thus, we use the emulator provided by the SDK to test functional correctness. The emulator emulates the FPGA's behavior on the CPU, which allows us to check functional correctness without waiting hours for compilation of FPGA bitstreams. However, the emulator provided in the SDK version that we use has several potential shortcomings: it cannot emulate concurrent memory accesses, the *autorun* function does not work correctly, and an *NDRange*-launched kernel will not release memory. We need to run the program on the FPGA, instead of the emulator, to handle these problems.

B. Kernel optimization

The FPGA has limited hardware resources for kernels w.r.t. the GPU. For a real-time application, run-time reconfiguration is not an option since it takes several seconds to finish. Thus, hardware resources should be assigned to each kernel with a balanced load. In order to get the best performance out of limited resources, we should improve the performance of the bottleneck kernel.

There are two major techniques to improve performance of FPGA kernels, namely *vectorization* and *multiple com-*

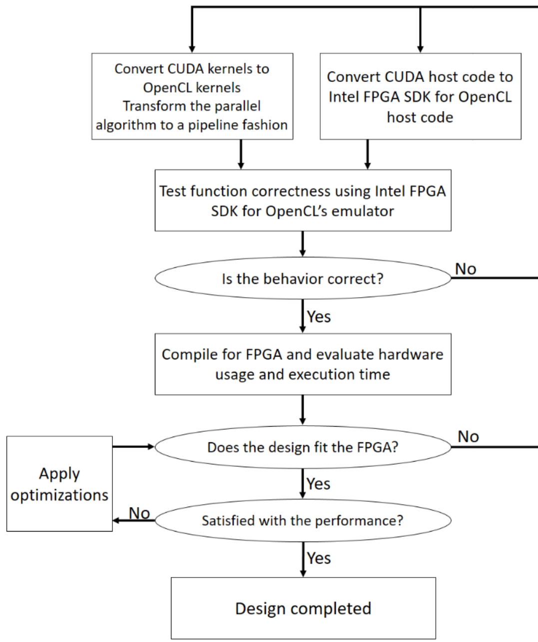

Fig. 3. Converting CUDA/GPU design to OpenCL/FPGA

pute units. Vectorization is done by specifying attribute "num_simd_work_items", which allows the compiler to treat the datapath and arithmetic operations as vector operations and thus perform computation in SIMD fashion. This should be done whenever it is possible as this can increase performance with minimal hardware usage. Multiple computing units can be achieved by specifying attribute "num_compute_units" or manually writing multiple kernel copies under different names. By making multiple kernel copies, the computation runs in parallel, and can have a speed-up of N for N kernel copies before the bandwidth reaches the limit.

In our FPGA system, vectorization does not benefit our kernels much because a single task kernel is more efficient in our application, and vectorization requires an NDRange kernel. On the other hand, two copies of complex-multiplication kernels allow the throughput to increase by 96% and four copies of linear search kernels allow the throughput to increase by 280%.

There are several other techniques programmers can do to optimize performance and resource usage, such as loop unrolling, avoiding nested loops, eliminating data dependencies when possible, utilizing on-chip memory, and taking advantage of dedicated hardware like accumulators (available in Arria 10 and newer Intel FPGAs).

C. 2D FFT engine for Image processing

FFT processing is widely used in CPU/GPU systems; thus, several libraries support CPUs/GPUs such as FFTW [5], cuFFT [3], and clFFT [2]. Because GPUs mostly deal with image processing, these libraries naturally have optimized algorithms for image-processing-related FFT operations. On the other hand, the existing FFT engines for the Intel FPGA

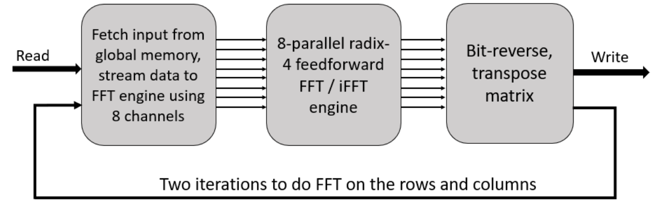

Fig. 4. Intel design example 2D FFT

that come from Intel Altera IP cores and are used in OpenCL design examples [8] are not specifically designed for 2D real-time image processing. Though clFFT is an FFT library designed for OpenCL, it targets GPUs only and is not targetable to an FPGA. We addressed this problem by building an FFT engine that performs efficient FFT for image processing with reasonable hardware usage and comparable throughput to GPU implementations.

Based on the radix-2^k FFT architecture proposed by Garrido et al. [6], the Intel OpenCL design example performs 2D FFT by taking FFTs of the horizontal rows, transposing the output and repeating the horizontal FFTs, and then, transposing the output for a second time to get the final output, as shown in Fig. 4. This approach yields an acceptable 30 GFLOPS with less than 40% of Arria 10 FPGA resources for a 1024x1024 complex input. As stated by Parker et al. [14], it is possible to achieve higher throughput with multiple cores and more hardware usage.

However, for image processing, the input data consists of real numbers, and this complex-to-complex FFT algorithm wastes half of its throughput and resources on the all-zero complex part. To address this issue, we apply the Hermitian-symmetric reduction approach to build a real-to-complex and complex-to-real FFT engine [7]

Since an FFT input with all-zero imaginary data would yield a Hermitian-symmetrical output (the second half of the output's real part is symmetrical to the first half and imaginary part is conjugated), we can process the $2N$-point FFT input with an N-point FFT engine and express the output with $N+1$ numbers. First, we pack the $2N$-real-input $x[n]$ into an N-complex-sequence $y[n]$, shown in Equation (1). We take the N-point Fast Fourier transform of $y[n]$ to get the temporary output $Y[k]$, shown in Equation (2). Then, as Jones proved [10], an equation for the 2N-point FFT output of $x[n]$ can be written as shown in Equations (3) and (4), where $Xr[k]$ and $Xi[k]$ stand for the real and imaginary part of $X[k]$, $Re()$ stands for *real part of* and $Im()$ stands for *imaginary part of*.

$$y[n] = x[2n] + ix[2n+1] \tag{1}$$

$$Y[k] = \sum_{n=0}^{N-1} y[n] e^{-j(\frac{2\pi}{N})nk} \quad (k = 0, 1 \ldots N-1) \tag{2}$$

$$X_r[k] = \frac{1}{2}Re(Y[k] + Y[N-k]) + \frac{1}{2}\cos(k\pi/N) \times Im(Y[k] + Y[N-k]) - \frac{1}{2}\sin(k\pi/N) \times Re(Y[k] - Y[N-k]) \quad (3)$$

$$X_i[k] = \frac{1}{2}Im(Y[k] - Y[N-k]) + \frac{1}{2}\sin(k\pi/N) \times Im(Y[k] + Y[N-k]) - \frac{1}{2}\cos(k\pi/N) \times Re(Y[k] - Y[N-k]) \quad (4)$$

The N-th data is the singular point in the symmetrical output sequence, which is computed by Equation (5).

$$X(N) = Re(Y(0)) - Im(Y(0)) \quad (5)$$

The rest of the output can be computed by Equations (6) and (7).

$$X_r(2N-k) = X_r(N) \quad (6)$$

$$X_i(2N-k) = -X_i(N) \quad (7)$$

Equations (3) and (4) give us the result of the first N outputs of a $2N$-point FFT. Combining equations (5), (6) and (7), we can compute the output of a $2N$-point FFT with only one N-point FFT transformation. We implement this approach on the horizontal row FFTs. On the other hand, vertical column FFTs remain the same as those are complex-to-complex FFTs. Implementation of the above equations requires the FFT engine to finish the N-point transformation as it involves $Y[N]$ and $Y[N-k]$ simultaneously. Thus, we implement the extra computation in the *transpose* kernel. Implementing the computation in the *transpose* kernel also avoids extra global memory bandwidth usage. To reduce run-time computation, we use pre-computed factors containing the $\cos$ and $\sin$ components. The dataflow of our design is shown in Fig. 5. This design fits the FPGA because the modularized kernels improve the resource usage as the modules can be used several times in one 2D-transform and inverse-FFT.

One difficulty of constructing this method is that we need to write Equation (5) as an extra output in the *transpose* kernel as the original design has store units only enough for Equations (1) and (2). If we use the normal write-to-global method, it will cause the number of store units inside the if-then-else branch for Hermitian-symmetric reduction and the branch of normal transpose to be different. Moreover, the compiler will then try to combine the store units of both branches, and the extra unit will be constructed in concatenated form, as illustrated in Fig. 6. This will result in a large amount of extra latency. With a lack of control over the compiler, and without a way to balance the number of store units in both branches, we addressed this problem by passing the extra data through the channel function to another kernel. This solution makes the latency negligible.

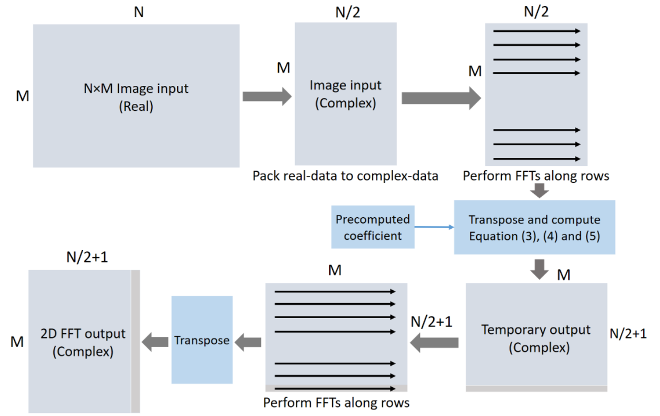

Fig. 5. Data flow of 2D real-to-complex FFT

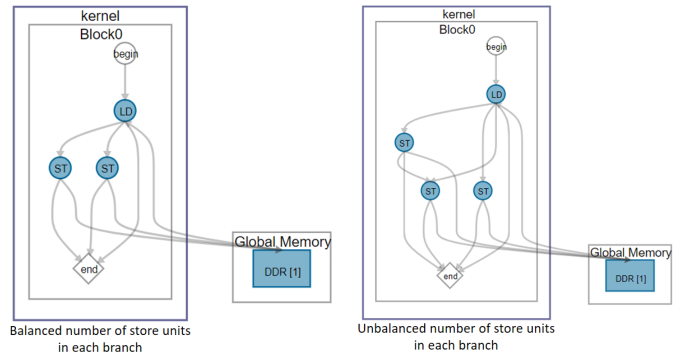

Fig. 6. Load/store unit diagrams provided by the System Viewer tool of the Intel FPGA SDK for OpenCL

Using this Hermitian-symmetric reduction, we achieve approximately two times speed-up over the original complex-to-complex FFT design with no loss in accuracy.

V. Experimental Settings, Results, and Discussion

Our system configuration is given in Table I. To compare the FGPA and GPU systems, we use the video clips from the LISA Traffic Sign Dataset [9] and from the EU dataset that was utilized in our previous study [12], which presented the initial GPU implementation of the template-based speed-limit-sign recognition approach. The LISA dataset includes 604x326 video captured from on-vehicle cameras. We extract 30 clips with 828 frames of road image containing 38 US-speed-limit signs. The EU dataset includes gray-scale 640x240 video from different types of European roads (e.g., highway, city, and country) under varieties of weather conditions (e.g., sunny, foggy, rainy, and snowy). It has 69 clips totaling 45 minutes of driving video that includes 120 EU-speed-limit signs.

A. Comparing the FPGA to the GPU

We measure the execution time by calculating the number of frames processed per second based on the timing information of each frame and observe that both GPU and FPGA systems have a similar throughput. Power consumption of

TABLE I
SYSTEM CONFIGURATION

	Host	GPU	FPGA
Hardware	Intel Core i7	NVIDIA GTX1060	Intel Arria 10 GX
Software	Visual Studio 15.2	CUDA 9.0	Intel FPGA SDK for OpenCL 17.1

TABLE II
PERFORMANCE COMPARISON OF THE FPGA AND THE GPU

	GPU	FPGA
Average FPS	33	30
Board and processor power	24 W	17 W
Processor power	19 W	12.5 W
Joules per frame	0.72 J	0.56 J
Accuracy on EU dataset	90%	90%
Accuracy on LISA dataset	92%	92%

the FPGA is estimated using *PowerMonitor.exe* provided by Quartus 17.1 and that of the GPU is estimated using NVIDIA NVML library [4], where both tools report the power consumption based on on-board sensors. In order to provide thorough comparisons between the two systems, we include power comparison between the whole PCIe cards' power, and the comparison between the processors' power, which eliminates power overhead from the boards. The processor power comparison is meaningful because it scales as the design grows larger. The FPGA processor power consumption is calculated by subtracting the idle power before configuration from the computing status power. On the other hand, the GPU processor power is calculated by subtracting the idle power from the computing status power. Our results show that the FPGA system provides a better power consumption than the GPU system. In addition, we observe that the difference between the values of the FFT outputs computed using NVIDIAs CUFFT library on the GPU and using our efficient FFT engine on the FPGA is negligible (i.e., about 10^{-8}). Thus, the GPU and FPGA systems have the same computation accuracy results on both datasets, which indicate the percentage of correctly classified speed-limit signs. As expected, the accuracy of our FPGA and GPU systems on EU signs are the same as that of our previous study [12] since they use the same template-based approach and video clips from the same EU dataset. In addition, accuracy of our systems on US-speed-limit signs that are extracted from LISA dataset is close to that of on EU-speed-limit signs, which indicates the scalability of the template-based approach across different types of road signs. Throughput (i.e., average number of frames per second [FPS]), power consumption, and accuracy (i.e., correctness of the detected position and the speed-limit number) of the GPU and FPGA systems are provided in Table II.

To gain further insights, in addition to executing all kernels together that constitute the systems, we also execute each kernel individually on the FPGA and the GPU and compare

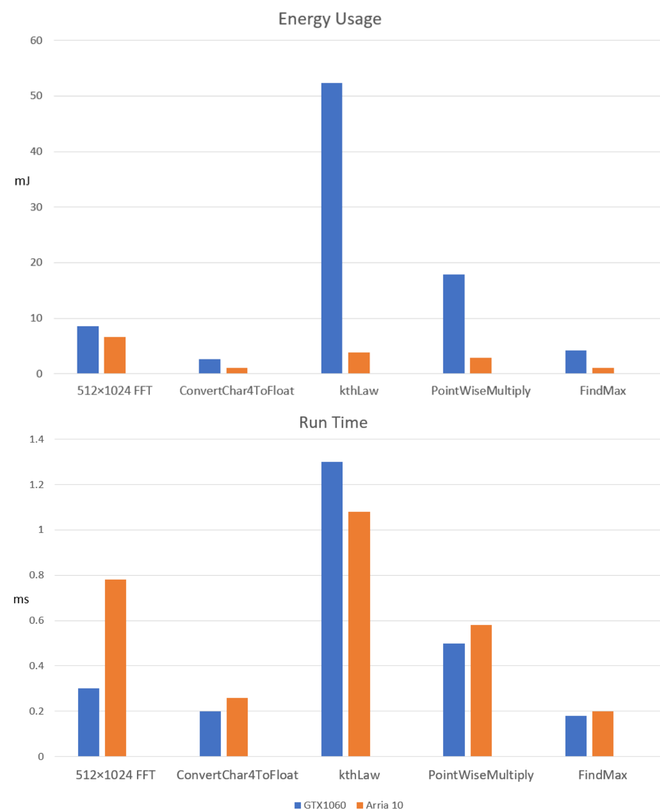

Fig. 7. Energy and run time comparison of individual kernels

their performance. The energy usage and latency of the kernels for computing a single frame are given in Fig. 7. Energy usage is computed by multiplying the power and the run time of the kernel. In order to do a fair comparison, while executing the kernels individually, the kernel on the FPGA is not allowed to utilize all FPGA resources. The kernel is set to the same configuration that is used in the execution of all kernels that constitute the system.

Results show that FPGA kernels for a single workload can achieve better power efficiency and similar throughput. Although each kernel has significantly better power efficiency, the FPGA has 34% better power efficiency over the GPU due to the overhead of the static current that supports the configured FPGA.

B. Insights on building the FPGA versus the GPU system

As new and diverse workloads are emerging around new applications, FPGAs have a major advantage over ASIC chips in terms of development effort and scalability. Moreover, with current high-level programming tools such as OpenCL, FPGAs are becoming comparable to general purpose computing units like CPUs and GPUs. This section presents our insights to the differences between developing an FPGA system and a GPU system.

Nowadays, computing device complexity has become so large, it is not efficient to design them in an RTL language. OpenCL for FPGA is designed to mitigate this inefficiency.

However, since available OpenCL libraries are usually designed for GPUs, it is not efficient or even not possible to apply these libraries to FPGAs. In addition, the available APIs are much more complete for GPUs. Although FPGA programming has been around for a while, it is mostly used for chip prototyping or specific DSP products. Hence, the FPGA development support is not as advanced as that of GPU, which features an ecosystem of popular general-purpose computing tools that mostly evolved in the past decade. Intel Altera has a collection of RTL design resources called Altera IP cores, and it is possible to combine them with an OpenCL development environment. But it is by no means an easy task for a software programmer who has limited background in this area because it requires knowledge about RTL code and interface. Thus, for software programmers new to this area, finding pre-built resources or libraries that fit their needs is difficult, as we have experienced in this study with our search for the FFT support.

Another factor that differentiates GPU programming from FPGA programming is the optimization process. As GPU compilers and schedulers are getting more mature, the programmer can now focus on the algorithmic side of programming. On the other hand, to optimize the performance of FPGA systems, the programmer must know the strengths and weaknesses of the FPGA, because the generated hardware structure greatly affects the performance. For example, for an image processing kernel on an FPGA that processes pixels with interactions with neighboring pixels, using a shift-register as a sliding window is much more efficient than processing the pixels in parallel.

One of the difficulties we encountered is the lack of flexibility in choosing the structure of the load-and-store unit of a kernel. Intel's FPGA SDK for OpenCL does most of the optimization part for low-level details, but many times the compiler would choose a structure that is not the most efficient alternative. For example, the compiler will use a *burst-coalesced* mode load unit for accessing global memory when it deems that it is large enough to benefit from this mode, but this mode is not as efficient as a *prefetch mode* load unit. This is because that the compiler has no information about the host code and data. In the current version of the SDK, the programmer has no control over the load-and-store unit mode.

Another example is that the compiler will try to combine the load/store units in different branches, and add a second layer of units concatenated to the shared units if either branch has extra load/store units, as depicted in Fig. 6. This will ultimately lead to longer latency than not sharing the units at all. This happened with our 2D real-to-complex FFT kernel and we had to work around it by passing the extra units to another kernel using channel functions. Both problems can potentially be solved if the SDK provides more options to let the user control the design structures. However, more options and controls over the design means developer would be required to acquire more knowledge. This may be difficult for new programmers to deal with and conflicts with the purpose of a simple high-level tool like OpenCL. Nevertheless, since optimizations must be done when possible, it is still desirable to let the programmers have options than let them try to work around it.

VI. CONCLUSION

In this paper, we presented our experience in translating a template-based speed-limit-sign recognition application running on the GPU to its counterpart running on the FPGA. In addition, we explained the techniques we have used to design and optimize the computing kernels on the FPGA and introduced an efficient FFT engine which can also be utilized by other image processing applications running on the FPGA. Then, we compared the performance and power efficiency of the Intel Arria 10 FPGA and an NVIDIA GTX1060 GPU for performing the same approach by presenting a methodology for comparing FPGA and GPU results. Finally, we shared our insights of using Intel FPGA SDK for OpenCL.

Our results indicate that GPUs are, in general, better suitable for fine-grained parallelism without data sharing whereas FPGAs are most suitable for coarse-grain parallelism. Our study shows that FPGAs can achieve throughput and accuracy comparable to GPUs with a significantly lower power consumption. Moreover, with the introduction of advanced programming tools, building FPGA systems for autonomous vehicles and other applications are becoming more efficient. Hence, the improved FPGA programming environment can speed up the development process compared to FPGA systems programmed in RTL languages.

The future plan for our study includes comparing more FPGA board models and more optimization techniques, such as full-channel utilization between more kernels and approximate computing using lower-precision floating point computing (not supported by Arria 10 yet) to reach theoretical throughput limits. Also, we plan to evaluate other road sign detection approaches such as the ones that involve deep-learning-based algorithms and develop hybrid systems that combine template-based approaches (for performing detection) and deep-learning-based approaches (for performing classification) running on both FPGAs and GPUs. With these additional efforts, we plan to further investigate the performance and programmability of FPGAs.

ACKNOWLEDGMENT

Thanks to Intel Labs for both funding this project and providing technical support for it.

REFERENCES

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[2] clFFT. A software library containing FFT functions written in OpenCL, 2014. [Online; accessed 28-December-2017].

[3] NVIDIA Corporation. CUDA CUFFT Library, 2017.

[4] NVIDIA Corporation. NVML API Reference Guide, 2017. [Online; accessed 28-December-2017].

[5] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[6] Mario Garrido, J. Grajal, M. A. Sánchez, and Oscar Gustafsson. Pipelined Radix-2K Feedforward FFT Architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(1):23–32, January 2013.

[7] Texas Instrument. Efficient FFT Computation of Real Input, 2014. [Online; accessed 28-December-2017].

[8] Intel. OpenCL 2D Fast Fourier Transform Design Example, 2016. [Online; accessed 28-December-2017].

[9] Morten Bornø Jensen, Mark Philip Philipsen, Andreas Møgelmose, Thomas Baltzer Moeslund, and Mohan Manubhai Trivedi. Vision for looking at traffic lights: Issues, survey, and perspectives. *IEEE Transactions on Intelligent Transportation Systems*, 17(7):1800–1815, 2016.

[10] Keith Jones. *The Regularized Fast Hartley Transform: Optimal Formulation of Real-Data Fast Fourier Transform for Silicon-Based Implementation in Resource-Constrained Environments*. Springer Publishing Company, Incorporated, 2012.

[11] Aaftab Munshi. *The OpenCL Specification Version: 1.2*. Khronos OpenCL Working Group, 19 edition, 11 2012.

[12] Pınar Muyan-Özçelik, Vladimir Glavtchev, Jeffrey M. Ota, and John D. Owens. Real-time speed-limit-sign recognition on an embedded system using a GPU. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 1, chapter 32, pages 497–516. Morgan Kaufmann, February 2011.

[13] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. Version 9.1.

[14] M. Parker, S. Finn, and H. S. Neoh. Multi-GSPS FFTs using FPGAs. In *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, pages 430–436, July 2016.

[15] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 35:1–35:12, Piscataway, NJ, USA, 2016. IEEE Press.