An Agile Approach to the Design of Hardware Accelerators and Adaptable Compilers

Ross Daly, Jackson Melchert, Kalhan Koul, Raj Setaluri, Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Stephen Richardson, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, Keyi Zhang, Priyanka Raina
Stanford University, Email: rdaly525@cs.stanford.edu, praina@stanford.edu

***Abstract*—Application domains for accelerators change at a rapid pace requiring highly adaptable accelerators and corresponding application compilers. This paper describes a non-traditional agile approach to accelerator design with an adaptable application compiler. We start by designing a generic coarse-grained reconfigurable array (CGRA) accelerator and a basic working application compiler targeting the CGRA. Using this experience as a baseline, we develop a hardware and compiler generation framework where changes to feature specifications, written as programs in custom domain-specific languages, automatically propagate to the hardware and the compiler. We showcase this framework with the design of four chips, two of which were taped out, and an adapting application compiler that compiles image processing and machine learning applications written in Halide.**

***Keywords*—accelerators; compilers; co-design; agile**

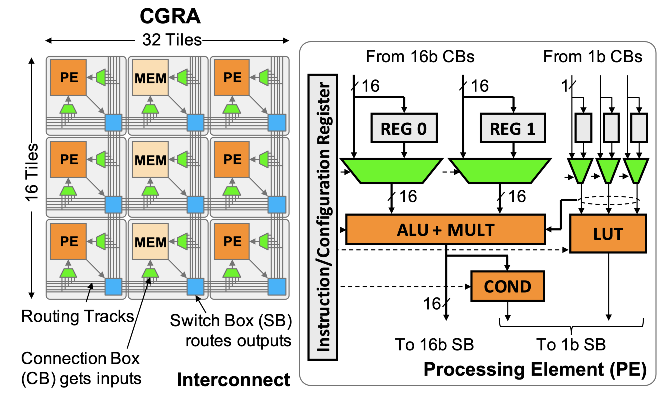

Fig. 1: A baseline CGRA architecture with processing element (PE) tiles, memory (MEM) tiles and a statically-configurable interconnect.

I. Introduction

Due to the slowdown in technology scaling, domain-specific hardware accelerators will play an increasingly important role in improving the performance and energy-efficiency of application execution. In their Turing Lecture, John Hennessy and David Patterson predict that this will lead to a new golden age of computer architecture [1]. Important applications such as image classification, speech recognition, language modeling, recommendation systems, and scientific computing are all changing at a rapid pace. Thus, the accelerators themselves must be sufficiently programmable to account for these future algorithmic advancements. In this paper, we describe our experience creating a series of accelerators targeting such applications in the image processing and machine learning domain.

A. Baseline System

Traditional accelerator design follows the *waterfall* approach, which starts by studying an application and creating a hardware specification, and then continues by going through a number of refinements. The waterfall approach suffers from twin issues of changing application requirements and incomplete knowledge/understanding of the problem, making the resulting system less useful than desired. To avoid these issues we explored an *agile* end-to-end hardware/software design flow. The remainder of this subsection describes our first attempt at achieving such a flow.

We choose a coarse-grained reconfigurable array (CGRA) [2], [3], [4] as our initial accelerator architecture, instead of an application-specific integrated circuit (ASIC), to achieve a good trade-off between programmability and efficiency. CGRAs have a spatial architecture similar to field-programmable gate arrays (FPGAs), with the main difference being that CGRAs use word-level processing elements (PEs) rather than bit-level lookup tables. As shown in Figure 1, PE and memory (MEM) tiles are organized in an island-style array where data is routed between tiles. CGRAs present a good choice for domain-specific accelerators because they can be reconfigured to execute a large variety of applications, similar to an FPGA, and tuning the amount of specialization in the PEs, memories, and routing fabric of the CGRA allows exploring the efficiency space between ASICs and FPGAs.

We choose Halide [5] as the programming language for our accelerator. Halide is a C++-embedded domain specific language (DSL) for image processing and machine learning algorithms. Notably, Halide programs are split into two parts: an algorithm, which specifies the computation that occurs on an input image to produce an output image, and a schedule, which determines the order in which output and intermediate pixels are computed. Schedules are expressed as a composition of scheduling primitives such as split, reorder, tile, and unroll.

‘DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited.

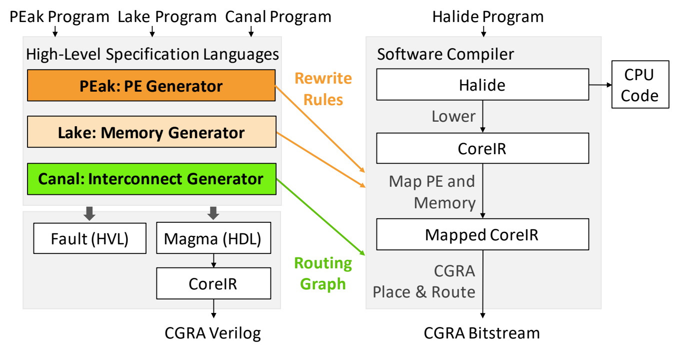

Fig. 2: Hardware-software design flow. Each subsystem is described as a high-level specification which generates both a hardware implementation and compiler collateral. Halide programs can then be automatically compiled to the accelerator.

While the original Halide system compiled these schedules to CPUs and GPUs, [6] showed that the same scheduling primitives could be interpreted for an FPGA target. For example, loop unrolling on an FPGA corresponds to duplicating compute units to achieve data-level parallelism. Because the scheduling primitives have the same interpretation on CGRAs and FGPAs, we were able to reuse parts of this compiler.

Our compiler, which lowered Halide programs to our CGRA, has three main steps. The first step, leveraging [6], produces a dataflow graph of the compute kernels and memory buffers. We chose CoreIR [7] as the intermediate representation (IR) since it is intended to represent nested graphs of operations. The next step, called *mapping* (also commonly referred to as instruction selction), translates compute kernels to specific PE instructions, and a similar translation happens for memory buffers. The final step, place-and-route, selects physical PE and memory tiles on the CGRA for the mapped compute kernels and buffers, respectively, and also chooses how to route data between tiles. Partitioning our compiler this way allows us to more easily optimize the separate components of the compiler, as well as ensure their correctness.

B. Initial Results

Our first chip and compiler were successful in accelerating basic Halide programs. However, we discovered that maintaining both the hardware design and the compiler, and ensuring that they were always in sync, was quite difficult. Each subsystem was designed and maintained by individual students. This resulted in each update to the design requiring significant communication overhead to ensure that the entire system continued to work correctly. In particular, changes to the architecture of the PE or the memory required updating the mapper manually. For example, if a new operation was added to the PE's ALU, then the mapper needed to consider that new operation. Additionally, changes to the interconnect needed to be reflected in the place-and-route tool. We found traditional methods such as documentation or accompanying specifications to be ineffective, due to the extra work required of each student to maintain such collateral.

II. Agile Hardware-Software Design Flow

Our approach to overcoming these challenges was to describe the accelerator using a *single source of truth*, first discussed in [8]. A single source of truth description consists of a high-level specification of a component, from which all necessary design collateral can be generated. To enable this approach, we designed three separate DSLs for the three major design subsystems: PEak for PEs, Lake for memories, and Canal for the interconnect. Rather than separately maintaining a design specification, a register transfer level (RTL) hardware implementation, and a compiler component, these DSLs sought to represent sufficient information (for each domain) to generate all three. This is shown in Figure 2.

A. PEak

The specification of a PE consists of its instruction set architecture (ISA), microarchitectural state, and *single-cycle semantics*. Here, single-cycle semantics refers to the values of any outputs, and next-states of microarchitectural registers, as a function of inputs, similar to Mealy machines. In order to encode this information, PEak represents PEs as simple python programs. Such programs are constrained to be simple classes consisting of only an initializer, in which the microarchitectural state is declared, and a call operator, in which the single-cycle semantics are specified. By leveraging polymorphism, a single PEak program can yield multiple interpretations: (1) Calling PEak programs with simple bit-vector types results in a functional model. (2) Calling PEak programs with magma [9] types yields an RTL (i.e. Verilog) description. (3) Calling PEak programs with SMT bit-vector types from PySMT [10] generates a formal model of the semantics encoded in satisfiability modulo theories (SMT). Interpretations (1) and (2) provide a design specification and an RTL implementation in a straightforward way. The formal model produced by interpretation (3) can be used to automatically synthesize rewrite rules, which can subsequently be used by our compiler to translate CoreIR graphs to PE instructions, thus automating the mapping of compute kernels. The equivalence of CoreIR primitives to SMT bit-vector operations is crucial to achieving this automation.

B. Lake

Lake is a DSL used to describe memories with a variety of capabilities. Halide loop nests are mapped to these memories using the streaming memory abstraction in [11]. Programs written in this DSL describe the underlying physical storage in terms of width and capacity along with the quantity and semantics of each port. Each port is additionally characterized with the data width, whether it supports reading, writing, or both, the initiation interval, and the latency. This representation allows one to describe many types of memories including register files or a memory tile with four configurable read and write ports. A custom memory generator compiles Lake programs into optimized RTL. The Lake compiler also generates the required collateral for mapping Halide loop nests to these RTL memories and is used during the mapping stage of compilation.

C. Canal

Canal uses a directed graph to specify an interconnect. A program written in Canal is flexible and can connect together different versions of PE and memory tiles with different numbers of inputs and outputs. Using Canal, a programmer can quickly explore different interconnect designs, including changing the network topology and switchbox design. In the Canal directed graph specification, vertices are hardware endpoints such as tile ports or pipeline registers and edges are directed wired connections. Vertices with more than one incoming edge generate multiplexers. Once the abstract graph is specified, Canal transforms that graph into a routing graph that the compiler uses to place and route applications. Finally, for a given graph and mapping, Canal generates the routing portion of the bitstream that is used to program the CGRA.

III. Design Space Exploration

This agile compilation framework lends itself well to design space exploration. The tight integration between the chip architecture and compiler generation enables quick evaluation of many design points. Given a particular specification in the PEak, Lake, and Canal DSLs, an application compiler can be automatically generated and both hardware and application performance can be measured.

The first generation mechanism we explored was using frequent subgraph mining and analysis techniques to automatically generate PE designs [12]. The goal of this technique is to create a PE specialized for a particular application or set of applications. First, the target applications are lowered into sets of computational graphs. Next, a frequent subgraph mining tool is used to generate common sequences of operations appearing in these applications. As these common sequences are often overlapping in complex ways, maximal independent set analysis is required, in order to refine the set of common sequences. These sequences are then merged together using algorithms originally developed for high level synthesis datapath merging [13]. From this final graph representation, we can generate a PEak program where each instruction corresponds to a unique frequent sequence. This PEak program represents an efficient PE design that is able to execute the most common complex operations in the application set. This frequent subgraph analysis technique was used to generate the specification of the PE in our most recent CGRA accelerator.

IV. Evaluation

Using the agile approach described in Section II, we designed several generations of our chip and compiler. We evaluated our system according to two groups of metrics: chip performance (performance and area), and compiler performance, which measures the performance of applications compiled on to the chip.

A. Specification of Accelerator Generations

Each chip's features are shown in Table I.

Our first chip, Jade, was a standalone CGRA consisting of a 16×16 array of PE and memory tiles and capable of performing 16-bit integer operations and 1-bit logical operations. The memory tiles were configurable to behave as a line buffer, a FIFO, or a typical SRAM.

Garnet improved upon Jade in several ways. First, instead of being a standalone accelerator, the CGRA was integrated into a system on chip (SoC) containing an ARM Cortex-M3 processor, and a large L2 memory called the global buffer. Garnet also increased the size of the CGRA to 16×32 tiles. Additionally, bfloat16 [14] multiplication, addition, and subtraction operations, along with capabilities to approximate division and transcendental functions were added to the PE. Memories were extended with a double buffer mode, and the switchbox topology was further optimized.

In Amber, we added a local register file to each PE tile, acting as an L0 cache. This change allowed for greater data reuse, particularly in neural network applications. The register files can also be used for branch-delay matching in pipelined applications, alleviating pressure on register resources. We replaced the line buffer and double buffer modes in the memories and the compiler with a unified buffer mode [11], which enabled a larger variety of affine memory access patterns to be efficiently scheduled on the array. Finally, we optimized the configuration space of the interconnect to be more space efficient.

The latest generation of our chip, Onyx, is similar to Amber, but leverages the PE specialization discussed in Section III. In the evaluations that follow, the specialized PEs in Onyx do not contain floating point operations.

B. Applications

To benchmark our chip and compiler, we considered the following image processing and machine learning applications:

- Gaussian Small and Gaussian Large: These applications implement an image blurring algorithm consisting of convolutions. Our versions use a 3×3 Gaussian kernel and operate on 16-bit integer types. Gaussian Small and Gaussian Large implement the same algorithm but Gaussian Large contains a higher degree of parallelism.
- Gaussian Float: A version of Gaussian Large operating on bfloat16 types.
- Harris: An implementation of Harris corner detection [15].
- Camera: An end-to-end camera pipeline, representative of the process of transforming raw camera sensor data to a final image [16]. This application includes kernels for hot pixel suppression, demosaicing, and color correction.
- ResNet: A single convolutional layer (conv4_x) of ResNet [17] that uses (3×3) filters and operates on a configurable number of input and output channels.

CGRA	Jade	Garnet	Amber	Onyx
CGRA Features	16×16 array Re	SoC with a 16×32 array	SoC with a 16×32 array	SoC with a 16×32 array
PE	16-bit and 1-bit integer ops	16-bit and 1-bit integer ops + Floating point ops	16-bit and 1-bit integer ops + Floating point ops + PE L0 cache	16-bit and 1-bit integer ops + Specialized instructions + PE L0 cache
Memory	Line Buffer, FIFO, SRAM modes	Line buffer, FIFO, SRAM modes + Double buffer mode	Unified buffer, FIFO, SRAM modes	Unified buffer, FIFO, SRAM modes
Interconnect	Configurable interconnect	Configurable interconnect + Imran switchbox topology	Configurable interconnect + Imran switchbox topology + Optimized configuration space	Configurable interconnect + Imran switchbox topology + Optimized configuration space

TABLE I: Accelerator feature additions across CGRA generations.

	CGRA Generation	Jade	Garnet	Amber	Onyx
	Maximum PE Frequency (MHz)	1428	1428	1428	1000
	PE Tile Area (μm^2)	3605.7	4523.0	8806.7	6662.1
Harris	Clock Frequency (MHz)	435	435	540	609
	Resource Utilization (PE/Mem/Reg)	91/12/84	91/12/84	91/12/86	81/12/111
Camera Pipeline	Clock Frequency (MHz)	X	190	190	139
	Resource Utilization (PE/Mem/Reg)	X	280/68/192	280/68/192	170/68/171
ResNet Layer	Clock Frequency (MHz)	X	145	145	261
	Resource Utilization (PE/Mem/Reg)	X	136/162/345	136/162/345	112/162/196
Gaussian Small	Clock Frequency (MHz)	411	411	471	386
	Resource Utilization (PE/Mem/Reg)	160/16/208	160/16/208	160/16/187	112/16/272
Gaussian Large	Clock Frequency (MHz)	X	502	502	386
	Resource Utilization (PE/Mem/Reg)	X	280/28/277	280/28/277	196/28/384
Gaussian Float	Clock Frequency (MHz)	X	348	348	X
	Resource Utilization (PE/Mem/Reg)	X	280/28/277	280/28/277	X

TABLE II: Comparison of clock frequency, resource utilization, and PE area for different applications for each generation of our CGRA. An X indicates that an application cannot be mapped to a given accelerator design because of hardware constraints.

C. Chip Performance

Table II measures clock frequency and resource utilization (in terms of number of PEs, memories, and registers) across each generation of the chip. Maximum PE frequency and PE area are obtained using a commercial synthesis flow using a 16 nm technology. In this experiment, only the PE is updated, not the memory tile or interconnect. Although we have been developing and improving both the chip and compiler simultaneously, to isolate the effect of the chip design changes, in these experiments we use the same compiler across each generation. Each application is aggressively pipelined on each generation of the chip, and Amber leverages the L0 caches contained within each PE tile when beneficial. Clock frequency is estimated using a custom static timing analysis (STA) tool specifically designed for our chip.

As seen in Table II, our adaptable compiler successfully compiles applications to all four generations of the chip, with only a few exceptions. Camera, ResNet, and Gaussian Large can not be placed-and-routed on Jade due to insufficient PE resources, but can be placed-and-routed on subsequent generations. Gaussian Float cannot be mapped to Jade and Onyx due to their lack of support for bfloat16 operations.

The compiler achieves a high clock frequency (between 200 - 500 MHz) on all applications for each generation of the chip. Often times the critical path occurs between two pipeline registers that were placed far apart. Using the L0 caches to co-locate pipeline registers removes these critical paths and results in a higher clock frequency as seen for both Harris and Gaussian Small applications on Amber compared to Garnet.

Finally, we study the effect of PE specialization. The number of PEs utilized is the same across Jade, Garnet, and Amber, but decreases on Onyx across all applications due to its specialized PEs.

D. Compiler Feature Comparison

In Table III we measure the effect of several compiler optimizations for each of the benchmark applications. The baseline compiler performs the basic steps of scheduling, mapping, packing, placement, routing, and bitstream generation. First we consider pipelining. Pipeline registers are introduced during the place-and-route phase and placed on to register resources available in the routing fabric. We also consider utilizing the L0 caches as pipeline registers when no register resources are available. All experiments are measured on the Amber.

Pipelining drastically increases the clock frequency of each application. Harris and ResNet achieve over a 16× per-

	Compiler Generation	**Baseline**	**With Pipelining**	**With L0 Cache**
Harris	Clock Frequency (MHz)	26	435	540
	Resource Utilization (PE/Mem/Reg)	91/12/30	91/12/84	91/12/86
Camera Pipeline	Clock Frequency (MHz)	17	190	264
	Resource Utilization (PE/Mem/Reg)	281/68/8	281/68/192	281/68/76
ResNet	Clock Frequency (MHz)	9	145	142
	Resource Utilization (PE/Mem/Reg)	136/162/0	136/162/345	136/162/311
Gaussian Small	Clock Frequency (MHz)	103	411	471
	Resource Utilization (PE/Mem/Reg)	160/16/136	160/16/208	160/16/187
Gaussian Large	Clock Frequency (MHz)	68	502	492
	Resource Utilization (PE/Mem/Reg)	280/28/158	280/28/277	280/28/242
Gaussian Float	Clock Frequency (MHz)	79	348	492
	Resource Utilization (PE/Mem/Reg)	280/28/154	280/28/277	280/28/242

TABLE III: Comparison of clock frequency and resource utilization across each generation of our compiler. Measured using the Amber generation.

formance improvement, Camera an 11× improvement, and Gaussian Small a 4× improvement. However, this improved performance comes at the cost of an increase in the number of utilized registers. Furthermore, pipelining too aggressively could saturate the available register resources and cause placement to fail. To overcome this, the compiler can place pipeline registers on to the L0 caches introduced in Amber. Harris, Camera, Gaussian Small, and Gaussian Float all improve in clock frequency when using this feature. The other applications, while having a slight decrease in the clock frequency, do use fewer registers with the L0 cache. Over several generations of our chip, we have dramatically increased the capability and efficiency of our accelerator, and over several generations of our compiler, we have added the features to leverage our design and significantly improve application performance.

V. Conclusion

With the slowdown of Moore's law, hardware specialization is the most promising technique for continued improvement of computing systems. The lack of a structured approach for evolving the software stack, as the underlying hardware becomes more specialized, has been one of the biggest impediments to its adoption. As we discovered, our approach provides a systematic way of thinking about accelerators as specialized CGRAs and employs a combination of new programming languages and formal methods to automatically generate the accelerator hardware and its compiler from a single source of truth. Furthermore, it enables the creation of DSE frameworks that automatically generate accelerator architectures that approach the efficiencies of hand-designed ones, with a significantly lower design effort. This has the potential to massively improve productivity of hardware-software engineering teams and enable quicker customization and deployment of complex accelerator-rich computing systems.

References

[1] J. Hennessy and D. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3282307

[2] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[3] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[4] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[5] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[6] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–25, 2017.

[7] R. Daly and L. Truong, "Invoking and linking generators from multiple hardware languages using coreir," in *WOSET*, 2018.

[8] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang, "Creating an agile hardware design flow," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[9] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," in *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA*, ser. LIPIcs, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 136. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 7:1–7:21. [Online]. Available: https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

[10] M. Gario and A. Micheli, "Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms," in *SMT workshop*, vol. 2015, 2015.

[11] Q. Liu, D. Huff, J. Setter, M. Strange, K. Feng, K. Sreedhar, Z. Wang, K. Zhang, M. Horowitz, P. Raina *et al.*, "Compiling halide programs to push-memory accelerators," *arXiv preprint arXiv:2105.12858*, 2021.

[12] J. Melchert, K. Feng, C. Donovick, R. Daly, C. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, "Automated design space exploration of cgra processing element architectures using frequent subgraph analysis," *arXiv preprint arXiv:2104.14155*, 2021.

[13] N. Moreano, E. Borin, Cid de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale

machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[15] C. Harris, M. Stephens *et al.*, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15, no. 50, 1988, pp. 10–5244.

[16] R. Kimmel, "Demosaicing: Image reconstruction from color ccd samples," *IEEE Transactions on image processing*, vol. 8, no. 9, pp. 1221–1228, 1999.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.