Understanding Performance Differences of FPGAs and GPUs

Jason Cong*, Zhenman Fang*†, Michael Lo*, Hanrui Wang*, Jingxian Xu*, Shaochong Zhang*
* Center for Domain-Specific Computing, UCLA † Xilinx
Email: zhenman@cs.ucla.edu

***Abstract*—This paper aims to better understand the performance differences between FPGAs and GPUs. We intentionally begin with a widely used GPU-friendly benchmark suite, Rodinia, and port 15 of the kernels onto FPGAs using HLS C. Then we propose an analytical model to compare their performance. We find that for 6 out of the 15 ported kernels, today's FPGAs can provide comparable performance or even achieve better performance than the GPU, while consuming an average of 28% of the GPU power. Besides lower clock frequency, FPGAs usually achieve a higher number of operations per cycle in each customized deep pipeline, but lower effective parallel factor due to the far lower off-chip memory bandwidth. With 4x more memory bandwidth, 8 out of the 15 FPGA kernels are projected to achieve at least half of the GPU kernel performance.**

I. Introduction

To improve the performance and energy efficiency of important application domains, different kinds of accelerators have been developed, including GPUs, FPGAs, and ASICs. Compared to ASICs, GPUs and FPGAs gain more popularity by providing better programmability and flexibility. It is natural to ask the question: *when is FPGA better, when is GPU better, and why*?

To answer this question, we intentionally use the GPU-friendly Rodinia benchmark suite [1] for comparison, which is widely recognized in the GPU community. Because Rodinia benchmarks are already optimized for GPUs after several generations of releases, it is fair to use them (the latest Rodinia 3.1 release) in our FPGA and GPU comparison.

We first port 11 Rodinia benchmarks (15 kernels in total) to the FPGA with Vivado HLS (high-level synthesis) C [2] for the kernels and OpenCL for host programs. To achieve reasonable performance on FPGAs, we apply a sequence of optimizations, including caching (tiling), customized pipeline, parallelization, double buffer, and memory coalescing and bursting. To make the porting of other benchmarks easier, we also provide a set of reusable templates and APIs to optimize FPGA accelerator designs, and will open source them at https://github.com/zhenman/rodinia-fpga-hls. These optimizations can be easily understood and mastered by software programmers, and provide 32x to 4,308x speedup compared to the original C code ported for FPGA.

We then propose an analytical model to analyze the performance differences between FPGAs and GPUs. In addition to the clock frequency ($freq$), we introduce three new metrics, including: 1) the algorithm-level operations per cycle achieved for each computing pipeline ($pipe\_OPC$); 2) effective coarse-grained parallelism factor (e_para_factor) achieved taking both computing and memory access limitations into consideration; and 3) extra $overhead\_factor$.

Based on these, we conduct an on-board performance comparison of the 15 Rodinia kernels between a 28nm Xilinx Virtex 7 FPGA and a 28nm Nvidia K40c GPU. We find that for 6 out of the 15 Rodinia kernels, the FPGA can achieve comparable performance or even better performance than the GPU. Meanwhile, on average the FPGA only consumes around 28% of the GPU power. With an in-depth analysis, we find (and confirm) that besides a lower clock frequency than GPUs, FPGAs usually achieve higher $pipe\_OPC$ in each computing pipeline due to its small pipeline initiation interval (II) and large pipeline depth, but lower e_para_factor due to far lower off-chip memory bandwidth. With 4x (or 32x) more memory bandwidth—like ones in the publicly available Amazon F1 instance [3] (or the coming HBM-enabled FPGA [4])—FPGAs become an attractive alternative for GPUs: for 8 (or 9) out of the 15 Rodinia kernels, our FPGA kernels are projected to achieve at least half of the GPU kernel performance, with 3 (or 4) of them faster than the GPU.

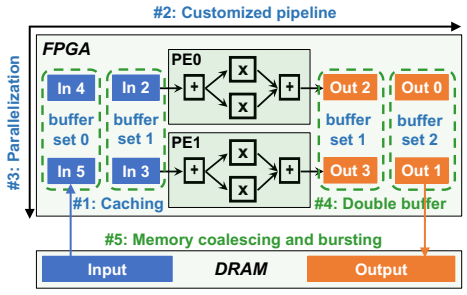

Fig. 1: Overview of FPGA kernel design strategies

II. Overall FPGA Design Strategies for Rodinia

We take the original code from Rodinia and make it HLS C synthesizable on the FPGA, which serves as the FPGA baseline. In addition, we provide a template for the OpenCL interface between CPU and FPGA communication, so that users can only focus on the FPGA kernel design in HLS C.

To achieve reasonable (not necessarily optimal) performance of these ported Rodinia kernels on FPGAs, as summarized in Figure 1, we apply a sequence of optimizations in HLS C that can be easily understood by software programmers. These simple techniques turn out to be very efficient.

1. Caching/Tiling. To enable fast data access by the FPGA processing elements (PE), our first step is to tile the code, transfer a data tile from off-chip DRAM to on-chip BRAM, and cache the data tile on-chip for later PE accesses. On average, it achieves 1.8x speedup over the raw HLS code.

Specifically, users claim local arrays in the code and use $memcpy$ to transport a tile of data to local arrays. After that, future data references are all accessed from local arrays.

2. Customized Pipeline. Customized deep pipeline is a unique technique in FPGAs, which gets rid of the instruction

pipeline overhead in CPUs and GPUs. On average, pipeline achieves an additional 7.8x speedup.

The pipeline initiation interval (II) and depth are two key factors to measure the pipeline performance; they denote the number of cycles 1) between the start times of two consecutive iterations and 2) processing one entire iteration of data. The smaller the II, the higher the pipeline throughput. The larger the depth, the higher the achievable speed.

To implement pipeline, users only need to insert $pipeline$ pragma into target loops. To achieve a small II, users need to perform loop transformations to eliminate harmful data dependencies just like parallel programming in CPUs and GPUs. One unique requirement is that users may have to partition the data arrays using the HLS $array\_partition$ pragma so that all data needed by the pipeline in each II cycles can be concurrently accessed. This is relatively straightforward in most cases except for stencil applications [5]. To make it easier, we implement the reuse buffer techniques proposed in [5] to achieve an II of 1 and provide the templates to users.

3. Parallelization. Like GPUs, coarse-grained parallelism on FPGAs is achieved by duplicating on-chip PEs such that different partitions of data can be processed concurrently. On average, parallelization achieves an additional 2.4x speedup.

In HLS C, this can be achieved by adding $unroll$ pragmas in outer loops. Then, a PE that accelerates inner loops will be duplicated. In addition, BRAMs usually need to be partitioned to satisfy the concurrent data access, i.e., $array\_partition$ pragmas need to be added.

4. Double Buffer. Double buffer aims to overlap the computation and memory access latency between different tasks (tiles). The on-chip BRAMs are duplicated to avoid read-write dependencies so that load, computation, and store can be processed concurrently. On average, double buffer achieves an additional 1.5x speedup.

We provide double buffer templates for users so that they only need to reorganize their code into load, computation, and store functions. Meanwhile, the local arrays need to be duplicated three times to store three concurrent sets of data.

5. Memory Coalescing and Bursting. According to prior characterization [6], memory coalescing (increasing access bit width) and memory bursting (increasing access length) are two key techniques to improve effective off-chip memory access bandwidth. On average, they achieve an additional 6.8x speedup, bringing the final average speedup to 328.9x.

To implement memory coalescing in HLS C, we implement a set of specialized $memcpy$ APIs to bond several pieces of narrow data into a single wider data. Users can directly leverage our APIs to copy wider data types.

III. Comparison Methodology

To perform first-order analysis, we propose Formula 1 to capture the relationship between the runtime and those microarchitectural factors.

$$runtime \approx \frac{total\_ops * overhead\_factor}{pipe\_OPC * e\_para\_factor * freq} \quad (1)$$

freq is the clock frequency of the FPGA and GPU kernels. **total_ops** is the total number of *algorithm-level operations* to be executed by the kernel. For example, we only count multiply and add operations in matrix multiplication and do not count the array indexing operations. Therefore, $total\_ops$ is the same for both FPGA and GPU implementations for a kernel, which makes it easier for comparison.

pipe_OPC is the number of operations that can be processed per cycle for each computing pipeline.

1. For FPGAs, $pipe\_OPC$ is calculated by Formula 2.

$$pipe\_OPC \approx ops\_per\_PE \ / \ II \quad (2)$$

ops_per_PE is the total number of operations done by a single processing element (PE). II is the pipeline initiation interval, which can be estimated by Vivado HLS.

2. For GPUs, we refer to a computing pipeline as a single streaming multiprocessor (SM) or computing unit (CU). Unlike FPGAs, it is hard to derive the pipe_OPC from statical analysis. Instead, we execute the application on GPUs and calculate pipe_OPC based on Formula 1. As explained in the following paragraphs, other factors in Formula 1 are straightforward to profile for GPUs.

e_para_factor is the effective number of parallel pipelines. We consider both the computing and off-chip memory access limitations when calculating e_para_factor.

1. For FPGAs, it can be calculated by Formula 3.

$$e\_para\_factor \approx min\{\frac{total\_FPGA\_resource}{resource\_per\_PE}, \frac{total\_memory\_BW}{BW\_per\_PE}\} \quad (3)$$

The first term calculates e_para_factor based on the available computing resources, while the second term calculates based on the available off-chip memory bandwidth (BW). From the Vivado HLS report, one can get the resource utilization—including DSP, LUT, BRAM, and FF—for a single PE. BW_per_PE can be calculated by the total size of off-chip data access of a single PE divided by its runtime (i.e., cycles/frequency).

2. For GPUs, it can be calculated by Formula 4.

$$e\_para\_factor \approx number\_SMs * SM\_activity\% \quad (4)$$

$SM\_activity\%$ is a factor measuring the activity (i.e., utilization) of SMs, which includes the impact of memory access. It is reported by Nvidia's profile tool nvprof.

overhead_factor is specific for FPGAs, and it is simply 1 for GPUs. It measures some inevitable overhead in FPGA kernel designs (Section II), which is the multiplication of:

1. Pipeline overhead cycles to fill the entire pipeline.

$$pipeline\_overhead = 1 + \frac{pipe\_depth}{II * pipe\_iterations} \quad (5)$$

2. Double buffer overhead caused by the idle state of buffer nodes in the first and last few iterations over data tiles.

$$buffer\_overhead \approx \frac{buffer\_iterations}{number\_tiles} \quad (6)$$

IV. Comparison Results and Analysis

A. Experimental Setup

Benchmarks. For a fair comparison, we choose the latest Rodinia release (Ver 3.1, Dec 2015). Shown in Table III, we choose a total of 11 benchmarks (15 kernels), which covers a variety of application domains, including structured grid, unstructured grid, dense linear algebra, and dynamic

Table I: FPGA and GPU comparison breakdown using direct influence factors. "ratio" denotes the speedup of FPGA over GPU.

	Runtime (s)			perf/W	pipe_OPC			e_para_factor			freq (MHz)			overhead_factor		
Kernel	FPGA	GPU	ratio	ratio	FPGA	GPU	ratio	FPGA	GPU	ratio	FPGA	GPU	ratio	FPGA	GPU	ratio
Hotspot	88,593	12,097	0.14	0.59	14.0	6.4	2.18	4.0	14.4	0.28	180	745	0.24	1.07	1	0.93
GICOV	148	438	2.97	7.76	224.0	3.1	72.03	4.0	14.4	0.28	190	745	0.26	1.72	1	0.58
Dilate	234	347	1.48	4.51	18.0	1.3	13.94	8.0	14.6	0.55	180	745	0.24	1.25	1	0.80
MGVF	89,715	11,816	0.13	0.50	45.0	19.6	2.29	4.0	14.1	0.28	160	745	0.21	1.06	1	0.94
SRAD	1,950	1,790	0.92	4.52	38.0	10.7	3.57	16.0	14.6	1.10	190	745	0.26	1.09	1	0.92
BP-1	536	371	0.69	3.10	4.0	0.7	5.62	8.0	14.4	0.56	190	745	0.26	1.15	1	0.87
BP-2	1,995	358	0.18	0.58	63.1	1.4	43.49	0.3	14.4	0.02	200	745	0.27	1.13	1	0.88
StepFactor	4,004	607	0.15	0.58	20.0	3.5	5.67	1.3	14.7	0.09	220	745	0.30	1.00	1	1.00
Flux	145	11	0.08	0.35	3.5	0.4	8.01	1.0	13.1	0.08	200	745	0.27	2.11	1	0.47
LUD	181,055	9,042	0.05	0.17	32.0	14.4	2.22	1.0	9.5	0.11	160	745	0.21	1.01	1	0.99
Kmeans	16,975	3,211	0.19	0.62	515.0	15.0	34.24	0.3	14.9	0.02	200	745	0.27	1.08	1	0.93
KNN	2,538	258	0.10	0.32	384.0	6.5	59.00	0.1	14.4	0.01	240	745	0.32	1.04	1	0.96
SC	15,464	1,187	0.08	0.35	128.4	16.3	7.88	0.5	14.7	0.03	220	745	0.30	1.01	1	0.99
NW	48	362	7.54	19.29	5.0	1.2	4.21	64.0	9.6	6.67	200	745	0.27	1.00	1	1.00
PF	28,750	24,680	0.86	2.85	8.0	6.7	1.20	32.0	12.0	2.67	200	745	0.27	1.00	1	1.00

Table II: FPGA and GPU comparison breakdown using more indirect influence factors

	Average power (Watts)			FPGA Pipeline		GPU						
						SM_activity%	occupancy	Number of instructions breakdown			Bandwidth (GB/s)	
Kernel	FPGA	GPU	ratio	II	depth			floating	others	overhead	read	write
Hotspot	22.7	97.3	0.23	1	60	0.96	0.95	35,719,168	175,500,012	5.9	56.1	25.5
GICOV	21.7	56.7	0.38	1	298	0.96	0.58	14,375,000	25,750,000	2.8	0.6	17.4
Dilate	23.0	70.0	0.33	1	27	0.97	0.88	N/A	N/A	N/A	7.8	9.1
MGVF	23.0	88.0	0.26	1	93	0.94	0.87	37,748,736	58,707,972	2.6	63.5	31.1
SRAD	23.3	115.0	0.20	1	121	0.97	0.95	69,206,016	162,094,410	3.3	20.9	80.5
BP-1	19.3	86.7	0.22	8	22	0.96	0.94	2,031,616	66,846,716	33.9	41.5	39.2
BP-2	23.0	74.7	0.31	1	24	0.96	0.96	4,194,352	35,651,744	9.5	60.5	57.0
StepFactor	24.3	92.7	0.26	1	207	0.98	0.92	63,963,136	109,051,904	2.7	82.7	13.7
Flux	20.7	93.3	0.22	13	242	0.87	0.48	290,874,905	443,753,142	2.5	57.3	2.5
LUD	21.0	72.3	0.29	1	317	0.63	0.78	17,548,744	60,750,439	4.5	18.3	22.9
Kmeans	23.7	77.3	0.31	1	297	0.99	0.98	417,792,000	801,365,760	2.9	205.0	1.2
KNN	24.3	77.3	0.31	1	33	0.96	0.84	23,068,660	46,137,352	3.0	120.6	62.4
SC	24.0	108.3	0.22	8	18	0.98	0.89	101,056,512	358,088,704	4.5	176.7	5.9
NW	21.0	53.7	0.39	2	6	0.64	0.03	N/A	N/A	N/A	1.2	1.0
PF	24.3	80.7	0.30	1	5	0.8	0.56	N/A	N/A	N/A	52.9	3.4

Table III: Summary of ported Rodinia benchmarks [1]

Dwarfs	Benchmark	Kernel Names	Domains
Structured Grid: SGrid	HotSpot	Hotspot	Physics Simulation
	Leukocyte	GICOV, Dilate and MGVF	Medical Imaging
	SRAD	SRAD	Image Processing
Unstructured Grid: USGrid	BackPropagation	BP-1, BP-2	Pattern Recognition
	CFDSolver	StepFactor, Flux	Fluid Dynamics
Dense Linear Algebra: DLA	LUDecomposition	LUD	Linear Algebra
	Kmeans	Kmeans	Data Mining
	k-Nearest Neighbors	KNN	Data Mining
	Streamcluster	SC	Data Mining
DP: Dynamic Programming	Needleman-Wunsch	NW	Bioinformatics
	PathFinder	PF	Grid Traversal

programming algorithms. For detailed description and implementation of each kernel, we refer the audience to the Rodinia website [1] and our open source website.

Hardware environment. The FPGA board we use is the Alpha Data ADM-PCIE-7V3 board that has a 28nm Xilinx Virtex 7 XC7VX690T FPGA chip. It has an on-board 16GB DDR3 RAM with a peak bandwidth of 12.8GB/s. The GPU is a 28nm Nvidia Tesla K40c @ 745MHz. It has 15 SMs, each SM has 192 FP32 cores. It has 16GB GDDR5 RAM with a peak bandwidth of 288GB/s. Both FPGA and GPU are connected to an Xeon CPU by PCIe. The average power consumption is measured by a power meter.

Software environment. The FPGA synthesis environment is Xilinx SDAccel 2016.4, which includes Vivado HLS to program kernels in C/C++ and OpenCL runtime to manage CPU-FPGA communication. We use a CUDA 9.0 environment to run GPU kernels. To profile metrics proposed in Section III, we leverage the Vivado HLS report, event timers from OpenCL, and metrics profiled by Nvidia's nvprof.

B. Overall On-Board Results

The overall on-board execution results are summarized in the "Runtime" columns of Table I. In all 15 Rodinia kernels, FPGA wins for 3 of them (green color); GPU wins for 9 of them (red color); FPGA has comparable performance for the remaining 3 (black color). In terms of performance per power ("Perf/W ratio" column of Table I), FPGA wins for 6 kernels (in green color). FPGA kernels typically consume 20% to 39% of the GPU power, as detailed in Table II.

C. In-Depth Analysis

To gain a deeper understanding, we measure and compare the metrics proposed in Section III. The breakdown results of direct influence factors, with the improvement of FPGA over GPU for each factor, are shown in Table I.

freq. Typically, large FPGA accelerator designs in HLS C can only get around 200MHz frequency, while GPUs can run at 745MHz or even higher frequency. Our FPGA kernels only achieve 21% to 32% clock frequency of the GPU.

overhead_factor. As shown in Table I, the FPGA has a small $overhead\_factor$, which is not a key factor.

pipe_OPC. In all of our cases, FPGA kernels have a larger $pipe\_OPC$ value than GPU kernels, ranging from 1.2x to 72.0x. This means that the computing capability of a single FPGA PE is stronger than that of a single GPU SM.

The large $pipe\_OPC$ of FPGAs can be explained by the small pipeline II (II=1 in most cases) and large pipeline depth, shown in Table II. Some kernels have irregular memory access patterns, e.g., BP-1, Flux, and SC, and (multiple) data need to be fetched in different cycles. Therefore, they have a higher II and lower $pipe\_OPC$.

For GPUs, the peak $pipe\_OPC$ should be 192 as there are 192 SIMD cores in each SM. However, the actual $pipe\_OPC$ rarely exceeds 20. There are several reasons for that gap. First, for some kernels, not all the threads in the SM are fully working, as shown in the "occupancy" column in Table II. Second, there are lots of "extra" instructions added to the "effective functional units" to support the program

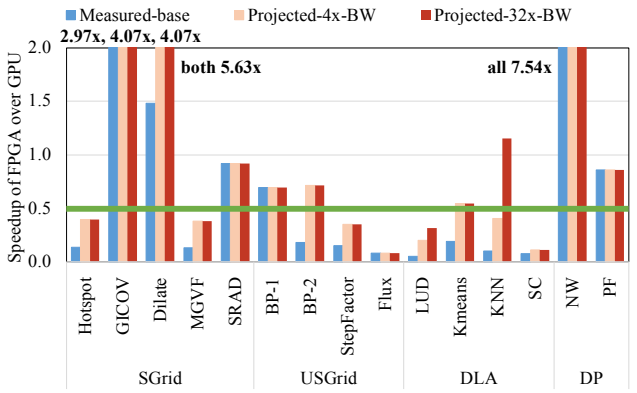

Fig. 2: Projected FPGA speedup with 4x and 32x more mem BW

execution, e.g., data indexing, load and store. As shown in Table II, only float instructions are useful, and the "extra" instructions can be 2.6x to 33.9x. Third, the instruction pipeline itself may have some intrinsic overhead.

e_para_factor. As shown in Table I, GPU kernels usually have a much higher e_para_factor than FPGA kernels, which means GPUs are better at coarse-grained parallelism.

In our setup, the GPU has 15 SMs and there are two factors potentially affecting the e_para_factor. One is the SM $activity\%$ (utilization) shown in Table II, which is high in most of the cases. The other is the memory read and write bandwidths, which are well below the peak bandwidth, as shown in Table II. This means that the memory system is not a bottleneck in the GPU.

Meanwhile, the e_para_factor in most of our FPGA kernels is low. There are two things that limit their coarse-grained parallelism. For computation-bound kernels—including SRAD, NW, and PF—the parallelism is limited by the on-chip FPGA logic resource. For memory-bound kernels—all the remaining kernels excluding BP-1, Flux, and SC (which have irregular memory accesses)—the parallelism is limited by our off-chip memory bandwidth (BW).

Projected performance. In fact, the memory BW limitation for FPGAs is being solved: Amazon F1 instance [3] already put four DDR-4 channels in a single FPGA (roughly 4x more BW), and an HBM-enabled FPGA will soon be available (roughly 32x more BW) [4]. Therefore, we also project the performance of our FPGA kernels based on our analytical model, assuming the performance scales ideally when there are 4x and 32x more memory BWs and the same on-chip FPGA resource. The normalized speedups are shown in Figure 2. Except for KNN and LUD, 4x more BW is good enough to improve the e_para_factor and thus throughput for the above memory-bound kernels. With 4x more memory BW, FPGAs become an attractive alternative for GPUs: for 8 out of the 15 Rodinia kernels, our FPGA kernels can achieve at least half of the GPU kernel performance, with 3 of them faster than the GPU. With 32x more memory BW, these numbers increase to 9 and 4, respectively.

V. Related Work

Most prior studies only compared GPU and FPGA performance using a single application. And they did not provide general insights, with a few exceptions as discussed below.

In [7] Che et al. conducted a performance study of Gaussian elimination, DES and NW from Rodinia on a CPU, GPU and FPGA. Their comparison is primarily qualitative and limited to only three benchmarks, while our work provides quantitative analysis between FPGA and GPU.

In [8] Krommydas et al. presented their implementation of OpenDwarfs benchmark suite in OpenCL, which is also derived from Berkeley dwarfs. However, the paper neither aggressively optimized the FPGA kernels nor provided the breakdown of performance comparison.

In [9] Zohouri et al. evaluated the performance and power of six Rodinia benchmarks on FPGAs using OpenCL, including NW, Pathfinder, Hotspot, SRAD, LUD and CFD. All the kernels ran slower on their FPGA over the GPU, while our work proved that FPGAs can achieve better performance in certain applications. Also, no step-by-step FPGA kernel optimization guideline was provided in [9]. In addition, they did not perform a quantitative analysis to explain the FPGA-GPU performance differences.

VI. Conclusion

In this paper we proposed an analytical model, with key performance metrics like $pipe\_OPC$ and e_para_factor, to better understand the performance differences between FPGAs and GPUs. We conducted an in-depth on-board performance comparison and analysis on the 28nm Xilinx Virtex 7 FPGA and Nvidia K40c GPU.

We found that besides a lower clock frequency than GPUs, the FPGA usually achieves a higher number of operations per cycle ($pipe\_OPC$) in each computing pipeline due to its small pipeline II and large pipeline depth, but lower effective parallel factor (e_para_factor)—largely due to far lower off-chip memory bandwidth. On our current 28nm FPGA and GPU, 6 out of 15 Rodinia kernels can achieve at least half of the GPU kernel performance, with an average of 28% of the GPU power. This number is projected to be 8 (or 9) if there is more memory bandwidth like Amazon F1 instance (or HBM-enabled FPGA).

Acknowledgments

This work is funded by the Center for Domain-Specific Computing and Center for Future Architectures Research. We also thank Falcon Computing for open-sourcing the memory coalescing APIs.

References

[1] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC'2009*, pp. 44–54. [Online]. Available: http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php
[2] J. Cong *et al.*, "High-level synthesis for fpgas: From prototyping to deployment," *TCAD'2011*, vol. 30, no. 4, pp. 473–491.
[3] "Amazon EC2 F1 Instances," https://aws.amazon.com/ec2/instance-types/f1/, accessed: 2018-01-21.
[4] G. Singh *et al.*, "Xilinx 16nm datacenter device family with in-package hbm and ccix interconnect," in *HotChips'2017*.
[5] J. Cong *et al.*, "An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers," in *DAC'2014*, pp. 1–6.
[6] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD'2016*, pp. 1–8.
[7] S. Che *et al.*, "Accelerating compute-intensive applications with gpus and fpgas," in *SASP'2008*, pp. 101–107.
[8] K. Krommydas *et al.*, "On the characterization of opencl dwarfs on fixed and reconfigurable platforms," in *ASAP'2014*, pp. 153–160.
[9] H. R. Zohouri *et al.*, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *SC'2016*, pp. 35:1–35:12.