CPU-FPGA Co-Scheduling for Big Data Applications

Jason Cong, Zhenman Fang, Muhuan Huang, Libo Wang, Di Wu

University of California, Los Angeles

Due to the limited scaling of general-purpose CPUs, FPGAs have emerged as an attractive alternative to accelerate big data applications due to their low power, high performance and energy efficiency. In this paper we aim to answer one key question: *How should the multicore CPU and FPGA coordinate together to optimize the performance of big data applications?* To address the above question, we conduct a step-by-step case study on CPU and FPGA co-optimization for in-memory Samtool sorting in genomic data processing, which is one of the most important big data applications for personalized healthcare. We find that a straightforward integration of an FPGA accelerator into the Samtool sorting only achieves marginal system throughput improvement over the software baseline running on a 12-core CPU. Therefore we propose a dataflow execution model to effectively orchestrate the computation between the multi-threaded CPU and FPGA, which demonstrates 2.6x speedup in our experiments.

Keywords: Reconfigurable hardware, Scheduling and Task partition, Genomic data processing

I. Introduction

The past decade has witnessed CPU core scaling coming to an end due to dark silicon limitations. At the same time, modern big data processing systems have evolved to an unprecedented scale. Cloud service providers, such as Amazon, Google and Microsoft, are seeking new system solutions to meet the ever-growing processing demands. Customized accelerators, such as GPUs and FPGAs, have gained increasing attention due to their low power, high performance and energy efficiency.

Meanwhile, the problem of efficiently processing such big data has attracted a lot of attention from both academia and industry. To fit the data into memory and to leverage multiple cores and servers, today's big data applications tend to distribute the datasets into multiple partitions where partitions can be processed in parallel [1].

Whereas a data-partition approach can accelerate big data applications on multicore CPUs, using FPGA accelerators is a more attractive solution since it helps to address the limited scaling of general-purpose CPUs and provides energy-efficient accelerators for integration in data centers. However, most prior studies on FPGA acceleration mainly focused on the FPGA accelerator design itself and did not consider efficient CPU and FPGA co-scheduling, which we find can be the key to the performance of such applications.

In this work, we focus on accelerating compression, a widely used routine in data center workloads, perform an in-depth case study on integrating an FPGA compression accelerator into a genomic data processing application, and aim to answer the following question: *How should the multicore CPU and FPGA coordinate together to optimize the performance of big data applications?*

Through our experiments we find that although we can get a high speedup on kernel computation by offloading the computation to FPGAs, the overall application speedup we can achieve may be very limited when comparing with the multi-threaded CPU implementation. The major reason is that the current application execution model fails to fully utilize system resources such as CPU cores and I/O. More specifically, when computation is offloaded to the FPGA, the CPU threads wait for the accelerator to finish and thus are idled.

Therefore, we propose a dataflow execution model and an interval-based scheduling algorithm to effectively orchestrate the computation between multiple CPU cores and the FPGA, which greatly improves the overall system resource utilization. Our experiments show that for pure CPU execution, the dataflow execution model provides a similar performance as the original data-parallel execution model. However, the dataflow execution model outperforms the data-parallel execution model when an FPGA is integrated into the system.

II. CPU-FPGA Co-Scheduling [1]

A. Dataflow Execution Model

To make use of the CPU cycles that are saved from the FPGA acceleration and to better utilize I/O, we propose to use a dataflow execution model. Each application is divided into several stages. Each stage can have multiple tasks that leverage data-level parallelism and all the stages are connected through in-memory data queues and work in a pipelined fashion.

To execute a dataflow program, the number of threads allocated to each stage needs to be decided. Slower stages deserve more CPU threads, while faster stages need fewer CPU threads. Besides the computation complexity of the stage, factors like disk bandwidth and data format play important roles in determining a stage's performance and thus the efficiency of the entire dataflow. For example, SSDs typically provide a higher bandwidth than HDDs; therefore, if the input data resides on SSD instead of HDD, the performance of the read stage will be improved. Finally if computation is offloaded to FPGA, CPU will be less utilized and thus other CPU-sensitive stages (like sort) may run faster. Therefore, it is nontrivial to determine the best thread allocation for a dataflow program.

[1] Prior work on task and thread scheduling on heterogeneous systems can be found in [2].

B. Proposed Runtime Thread Allocation Strategy

At runtime, we profile the CPU utilization ($util_i$) of stage i every small period of time. It is calculated as the actual thread time spent on each task divided by the total thread time of all tasks in this stage. Time that is spent on reading data from the input queue (dequeue) and writing data to the output queue (enqueue) is not counted into the actual thread time. Therefore, a high $util$ represents that a stage is making high use of its allotted CPU resource, while a small $util$ represents that a stage might be wasting time on dequeue/enqueue and thus is not making full use of its allotted CPU resource.

Our runtime adaptive thread allocation algorithm monitors the CPU utilization of each stage, and makes adjustments in thread allocation every period of time which moves CPU threads from the stages with a lower $util$ to the stages with a higher $util$. Denote the number of threads allocated to the faster stage as n_f, and the CPU utilizations of the faster stage and the slower stage as u_f and u_s. u_f is smaller than u_s since the faster stage should have lower CPU utilization. Empirically by moving threads from the faster stage to the slower stage, we expect that the CPU utilization of the faster stage can increase to $\frac{u_f+u_s}{2}$. Therefore the target number of threads for the faster stage is $\lceil n_f \cdot u_f/(\frac{u_f+u_s}{2})\rceil$. The number of threads that are moved from a faster stage to a slower stage, δ, as follows:

$$\delta = n_f - \lceil n_f \cdot u_f/(\frac{u_f+u_s}{2})\rceil, \tag{1}$$

The interval of thread re-allocation should be long enough so that the current thread allocation policy will take effect. This is due to the fact that when we decrease the number of threads for a stage, we wait for the allocated threads from the previous iteration to finish instead of killing them. Therefore the CPU utilization statistic that is collected right after thread re-allocation may not represent the effectiveness of the new thread allocation policy. Empirically the interval to sample CPU utilization approximately equals to the task time and thread reallocation is executed when tens of tasks have finished in the slowest stage.

III. Case Study

Throughout this section we use an example of the sort routine in Samtools [3] to illustrate the common issues in integrating FPGA solutions into multi-threaded CPU computations, present our observations and experiment results.

A. Samtool In-Memory Sorting

Samtool sorting takes a genomic sequencing file as an input, sorts read alignment by leftmost coordinate or by read name, and finally outputs the sorted read alignments to a compressed file, so that latter processing tools can easily identify the duplicate read alignments in the genome. Note that the size of input sequencing file is typically on the order of several hundreds of gigabytes. Therefore, it cannot fit into the CPU memory and external sorting algorithms are used.

Figure 1 presents an overview of the algorithm used in Samtool sorting [3]. First, it sequentially reads the on-disk file in either normal SAM (Sequence Alignment/Map) text

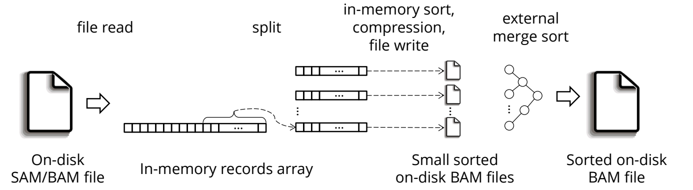

Fig. 1: An overview of the sort routine in Samtools.

or block-compressed BAM (Binary Alignment/Map) format into CPU memory. Second, it splits the read alignments into multiple partitions, so that each partition can be sorted in memory, compressed, and written to a small temporary sorted file in parallel. Compression is applied at this stage for each data block (e.g., 64KB size) within the partition before writing to the disk so as to save storage space and bandwidth. In addition, a cyclic redundancy check (CRC) code is computed on each original uncompressed data block, which will be used to detect file errors when the compressed file is read in the future. Third, all temporary sorted files will be merged together into a single sorted BAM file using external merge sort, which is mainly disk bound. In this paper we focus on the optimization of the first two stages, sequential read and parallel partition processing, which occupy around 50% execution time of the entire Samtool sorting. We call these two stages *in-memory Samtool sorting*. [2]

B. Experiment Setup and Initial Profiling

The software in-memory Samtool sorting [3] runs on a 12-core Intel Xeon CPU E5-2620 (@2.40GHz) with CentOS 7.2. This server has 128GB memory and 500GB SSD. A power meter is attached to the power outlet of the CPU server to measure the system power. The input data samples used in the experiments are the high-coverage exome samples from the 1000 Genome project.[3] For illustration purposes, we use a 27.6 GB SAM file chopped from the first segment of the third exome sample throughout this paper unless otherwise specified. To generate the input SAM files for Samtool sorting, we use `bwa-mem` [4], [5] to align these input exome samples.

Based on our profiling on the single-thread in-memory Samtool sorting, the compression and CRC algorithms, which are well suited for FPGA acceleration, occupy around 45% of the execution time. This motivates us to design an FPGA accelerator for compression and CRC. We design our accelerator with Vivado HLS and SDAccel (v2016.1). The FPGA board is Xilinx Kintex UltraScale KU115.

C. Accelerator Design and Performance

There are already several studies that accelerate compression and CRC on FPGAs [6], [7], [8], [9]. We implement an FPGA compression and CRC accelerator design similar to these studies. The major difference from these studies is

[2] Note that the term "in-memory sorting" here is not to be confused with the concept that refers to embedding the computing cores into memory.

[3] Data can be downloaded from:

- http://www.internationalgenome.org/data-portal/sample/NA12878
- http://www.internationalgenome.org/data-portal/sample/NA12892
- http://www.internationalgenome.org/data-portal/sample/HG01500

that we design our accelerator in HLS, which is portable and maintainable across different Xilinx FPGA platforms.

Our FPGA accelerator takes a byte array as input. It computes the cyclic redundancy check (CRC) code of the input array and produces a compressed byte array. The produced CRC code is the same as the result from Linux $crc32()$ in $zlib.h$. According to HLS report, our FPGA accelerator can process 16 bytes/cycle at 200 MHz, achieving a theoretical peak bandwidth of 3.2 GB/s. The design occupies 12.6% LUT and 4.9% FF on our KU115 board.

The measured performance of our accelerator kernel from Xilinx OpenCL runtime is 2.8 GB/s. The gap between our measured throughput and the theoretical throughput (3.2 GB/s) is because the data transfer between FPGA DRAM and FPGA kernel does not achieve a perfect pipeline initial interval (II) that equals to 1, since each DRAM burst read/write includes non-payload data overhead.

We test the compression ratio of our accelerator under the Calgary Corpus dataset [10]. We are only able to achieve a compression ratio of 1.73 (geometric mean) across the dataset, lower than the previous work, but still at a comparable level. The reason of a lower compression ratio is mainly due to the history string matching loss when there are hash conflicts to the same dictionary. Unlike RTL designs in [6] where double clock frequency is used for hash table, in HLS we do not have the flexibility of using different clocks in a single design.

We compare our compression throughput and ratio to two recent studies [6], [7] and the single-core CPU version in Table I. Although we see room to further optimize our accelerator design (*e.g.,* replacing some modules with RTL designs with doubled frequency), we did not pursue along that direction since its performance is already limited by the CPU-FPGA data transfer bandwidth through the PCIe connection.

TABLE I: FPGA accelerator comparison.

Design	theoretical and measured throughput (GB/s)	compression ratio
This work	3.2 / 2.8	1.73
Altera OpenCL [7]	2.8 / -	2.17
Microsoft RTL [6]	5.6 / -	2.09
CPU [6]	- / 0.05	2.62

To the best of our knowledge, this is the first compression (with CRC) design using Xilinx HLS.

D. FPGA Accelerator Integration with CPU

To handle efficient sharing of the FPGA among multiple application threads, we leverage our Blaze runtime system [11], which is an open-source project that offers accelerator management at node-level and at cluster-level. At node-level, the Blaze runtime system serves as an abstraction layer between the application threads and the underlying FPGAs. This additional layer maintains all information about tasks, kernels, kernel arguments, and device data blocks, enabling more sophisticated management (e.g., task queueing, thread-level fairness, and automatic FPGA reconfiguration) than that of the default OpenCL runtime.

We incorporate additional two considerations during acceleration integration with CPU. The first consideration is

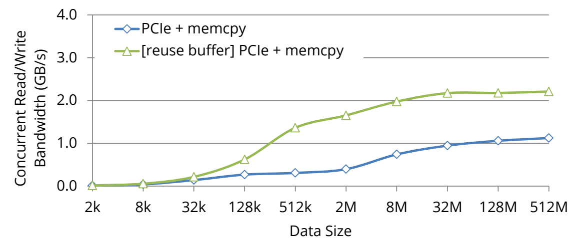

Fig. 2: Concurrent read/write bandwidth between CPU and FPGA through PCIe.

the data size for each CPU-FPGA communication through the PCIe bus. Similar to [12], we measure the PCIe read and write bandwidth using OpenCL APIs. The results are presented in Figure 2 where we can see that the concurrent read/write bandwidth increases as the payload size increases. The original BAM file is in a block-compressed format, where each uncompressed block has the maximum size of 64 KB at which size the data transfer rates are below 1.0 GB/s. To solve this problem, we redefined the maximum size of uncompressed blocks in BAM format to 32 MB; at this size the data transfer rate is much higher.

The second consideration is to reuse OpenCL memory objects. At OpenCL runtime, an OpenCL memory object is first created and allotted to the kernel. Figure 2 demonstrates that reusing OpenCL memory objects can increase the bandwidth between host CPU and FPGAs. Therefore, we implement a runtime OpenCL block reuse mechanism: Instead of releasing the OpenCL memory objects from previous runs, we maintain these objects in a lookup table as long as we do not run out of device memory space. New memory object allocations will first perform table lookups to see if there are already allotted objects that are large enough to hold the current ones; failure to find pre-allocated objects results in new memory objects being allocated, and old memory objects being released if we run out of space.

Compared to the CPU compression (with CRC) implementation, our FPGA implementation achieves 17.2x speedup under the single-thread scenario, and achieves 3.3x speedup in the 12-threaded scenario.

E. Performance of Samtool Sorting

Finally, Figure 3 summarizes the runtime breakdown for in-memory Samtool sorting with and without FPGA acceleration. During the in-memory sorting phase, data is first read into memory. Then parallel threads are launched; each thread sorts a chunk of data, compresses it and writes the data to a file. When multiple threads are used, we do not add extra synchronization among threads after sorting or compression. Therefore, we cannot tell the exact time that is spent on sorting, compression or file write; instead, the total time of these three steps is reported in the figure.

Looking at Figure 3, we can see that in the single-thread scenario, we achieve 17.2x speedup on the compression kernel by using the FPGA accelerator, 2.3x speedup on ‘sort+compress+write’, and 67% overall performance improvement. Note that file write time increases slightly since

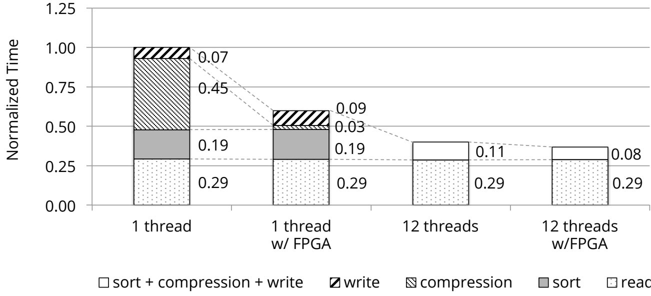

Fig. 3: Normalized time for in-memory Samtool sorting.

our FPGA compression ratio is smaller than that of the CPU baseline.

However, comparing the two rightmost columns in Figure 3 where 12 threads are used in the CPU baseline, we can only observe 1.4x speedup on 'sort+compress+write'. This is because 'sort' and 'compress' are well-parallelized in this baseline. The application performance is now limited by the sequential read stage and there is a marginal of 8% overall performance improvement by integrating the FPGA accelerator into the system.

F. Parallelizing the Read Stage

Following today's trends in big data processing, we first partition our input SAM files into multiple smaller files so that file read can be parallelized and IO bandwidth can be better-utilized. Our experiments show that parallelizing the read stage can improve the SSD performance by up to 3x in the SAM file read stage.

G. Dataflow-Samtools

We model Samtool in-memory sorting using the proposed dataflow execution model. We divide the entire application into three stages: $read$, $sort$ and $compress + write$. Data between stages are organized using multi-input and multi-output queues. In the experiment, 12 CPU threads are allocated to these three stages and thread allocations are adjusted dynamically according to the strategy in Section II-B. We name our dataflow implementation of Samtool sorting as **Dataflow-Samtool** sorting.

1) Evaluation on Runtime Thread Allocation

To evaluate the effectiveness of runtime adaptive thread allocation, we compare it to the static thread allocations, where we explore the application's latency under different static thread allocations. We found that application latency of the best static thread allocation and that of the worst static thread allocation can differ by up to 5x. Compared to the best static thread allocation that achieves the shortest application latency, we found that the runtime adaptive thread allocation can provide a performance very close (93%) to the best static allocation on the CPU platform. Moreover, it can even outperform the best static allocation on the CPU-FPGA platform. This is because our runtime thread allocation strategy provides a more flexible thread configuration than static allocations. For example, when a stage finishes, its threads can be reallocated to other stages.

2) Comparison of Different Optimizations

We compare the performance of different optimizations over the original 12-thread in-memory Samtool sorting in Figure 4 which is denoted as *SAM on SSD*:

1. **parallel read** parallelizes the read stage using 6-threads, which achieves and then executes the sort+compression+write stage using 12 threads;
2. **parallel read + FPGA** is based on parallel read but uses FPGA to perform compression;
3. **dataflow** is the 12-threaded CPU implementation of Dataflow-Samtool;
4. **dataflow + FPGA** is the 12-threaded CPU implementation of Dataflow-Samtool with FPGA.

There are several observations. First, dataflow does not outperform parallel read in the pure CPU case. The major reason is that there is additional memory consumption in maintaining data queues in the dataflow model which slows down the memory system.

Second, dataflow + FPGA performs better than parallel read + FPGA. The major reason is that in dataflow + FPGA mode, we are executing different stages of the job simultaneously with both computation-intensive tasks and I/O intensive tasks running in parallel. Therefore, I/O and compute resources can be better utilized. While in parallel read + FPGA mode, we are executing the tasks in a stage-by-stage fashion. CPU can be idled during read stage and I/O can be idled during compute stage, which is less efficient.

Finally, among all four optimizations, CPU-FPGA co-optimized dataflow-Samtool achieves the best performance, which is 2.64x faster than the original 12-thread Samtool sorting.

IV. More case studies

In this section we perform more experiments to evaluate our dataflow execution model.

A. Performance for Different Datasets

We evaluate the application performance on three large datasets, whose file sizes are 54.5 GB, 56.7 GB and 235.2 GB respectively. The Samtools in memory sorting takes about half an hour to finish on the CPU platform for the largest dataset. Our dataflow Samtools with FPGAs can achieve speedup of 2.8x, 2.6x, and 2.5x on these three datasets respectively.

B. Changing Input format

We change the input format from SAM format to BAM format where data is stored as compressed binary file. This change indicates that input file size is reduced and decompression is needed during read stage. Reading files in BAM format is more computation-intensive than reading files in SAM format. The results are presented in Figure 4 as *BAM on SSD*. In this case dataflow + FPGA achieve the best performance which is 2.62x better than the 12-threaded CPU baseline.

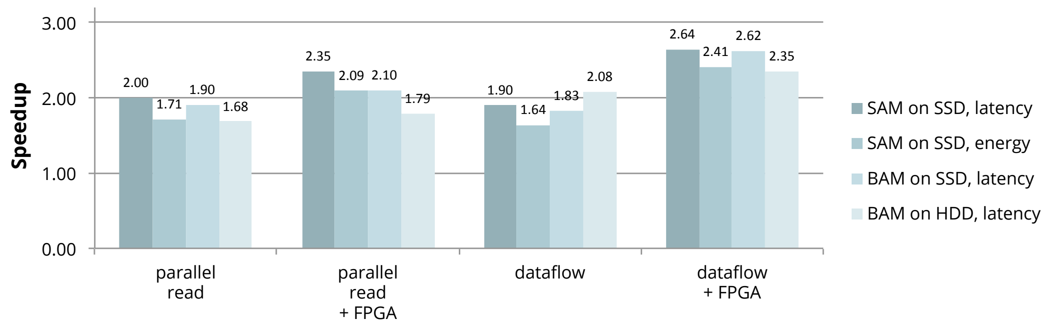

Fig. 4: Overall speedup and energy savings of different optimizations over original 12-thread Samtool in-memory sorting.

C. Changing Storage Type

We also test another case where the input resides on HDD rather than SSD. Since parallel read on HDD is slower than parallel read on SSD, the application is more disk-bounded. Therefore overall we observe less speedup by using FPGA accelerators. However among all the execution models, dataflow execution with FPGA accelerator still performs the best (2.35x speedup) as shown in *BAM on HDD* in Figure 4.

V. Conclusion

In this paper we conduct a case study on in-memory Samtool sorting and aim to find the right strategy to coordinate today's multicore CPU and FPGA together to optimize the performance of big data applications. To improve resource utilization and system performance, we proposed a dataflow execution model that combined data-level parallelism on multicore CPU, hardware specialization on FPGA, and pipeline parallelism between CPU cores and FPGA. Accordingly, we developed an adaptive runtime to effectively orchestrate the computation between multiple cores and FPGA. Our experimental results demonstrated that the dataflow execution model is more accelerator-friendly than the data-parallel execution model. Overall our CPU-FPGA coordination through the dataflow execution model can achieve an average of 2.6x system performance speedup over the original 12-thread in-memory Samtool sorting.

VI. Acknowledgments

This work is partially supported by the Center for Domain-Specific Computing under the NSF InTrans Award CCF-1436827; funding from CDSC industrial partners including Baidu, Fujitsu Labs, Google, Huawei, Intel, IBM Research Almaden and Mentor Graphics; C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; and UCLA Institute for Digital Research and Education Postdoc Fellowship.

References

[1] Dhruba Borthakur. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, page 39, 2008.

[2] Related work on CPU-FPGA co-scheduling. http://vast.cs.ucla.edu/~mhhuang/dt2017-coscheduling/related-work.pdf. Accessed: 2017-06-10.

[3] Samtools. http://www.htslib.org/doc/samtools.html. Accessed: 2016-12-10.

[4] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[5] Burrow-wheeler aligner for pairwise alignment between DNA sequences. https://github.com/lh3/bwa. Accessed: 2016-12-10.

[6] Jeremy Fowers et al. A scalable high-bandwidth architecture for lossless compression on FPGAs. In *FCCM*, pages 52–59. IEEE, 2015.

[7] Mohamed S. Abdelfattah et al. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, 2014.

[8] M. Walma. Pipelined cyclic redundancy check (crc) calculation. In *Computer Communications and Networks, Proceedings of 16th International Conference on*, Aug 2007.

[9] Y. Huo et al. High performance table-based architecture for parallel crc calculation. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, April 2015.

[10] Calgary corpus dataset. http://corpus.canterbury.ac.nz/descriptions/#calgary. Accessed: 2016-12-10.

[11] Muhuan Huang et al. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *SOCC*. ACM, 2016.

[12] Young-kyu Choi et al. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *DAC*, pages 109:1–109:6, New York, NY, USA, 2016. ACM.

Jason Cong is Chancellors Professor and director of the Center for Domain-Specific Computing (CDSC) at the University of California, Los Angeles (UCLA). His research interests include synthesis of VLSI circuits and systems, programmable systems and novel computer architectures. He has a PhD in computer science from the University of Illinois at Urbana Champaign. He is a Fellow of the ACM and IEEE.

Zhenman Fang is a postdoc researcher at UCLA. His research interests include computer architecture, customized computing and cloud computing. He has a PhD in computer science from Fudan University, China. He is a member of the ACM and IEEE.

Muhuan Huang is a software engineer at Google. Her research interests include scalable system design, customized computing and big data computing infrastructures. She has a PhD degree in computer science from UCLA.

Libo Wang is a software engineer at Google. His research interests include test automation, data visualization and scalable system design. He has a master degree in computer science from UCLA.

Di Wu is a staff engineer at Falcon Computing Solutions. His research interests include accelerator design, large scale deployment for machine learning, cognitive computing and high-throughput genome data analysis. He has a PhD degree in computer science from UCLA.