Exploring BFS Optimization with Software Programmable FPGAs

***Abstract*—Breadth-first search (BFS) is one of the most important building blocks of graph processing, and it is notoriously difficult to accelerate on FPGAs due to the irregular memory access and low computation-to-memory ratio. To address this problem, we opt to develop the BFS accelerator using high-level design tools such that the resulting accelerator preserves the high level software features including portability, ease of maintenance and use. Based on the BFS memory access patterns, we proposed a set of BFS specific optimizations such as hash filtering, prefetching and caching accordingly. In addition, we also explored some general high level synthesis (HLS) optimizations such as memory bank aware data layout and data path duplication and applied them to the design for higher memory access efficiency. Finally, we developed a design parameter tuning strategy to customize the high-level design options by taking advantage of the inherent software emulation of the HLS design.**

According to the experiments on a set of big graphs on different FPGAs, the optimized HLS based BFS accelerator achieves up to 70X performance speedup compared to the baseline HLS design. When porting the design from Alpha-Data ADM-7v3 to KU115, significant performance speedup is obtained, which demonstrates the portability of the HLS based BFS accelerator. When compared to the state-of-art handcrafted design on similar FPGA cards, the proposed BFS accelerator achieves over 70% of the performance while it possesses the software-like features.

I. Introduction

Breadth-first search (BFS) is the basic building component of many graph algorithms and is thus of vital importance to high-performance graph processing. Nevertheless, it is notoriously difficult for accelerating on FPGAs because of the irregular memory access and the low computation-to-memory ratio. At the same time, BFS on large graphs also involves tremendous parallelisms which indicate great potential for acceleration. With both the challenge and the parallelization potential, BFS has attracted a number of researchers exploring its acceleration on FPGAs [1], [2], [3], [4], [5], [6], [7], [8].

Previous work has shown that BFS accelerators on FPGAs can provide competitive performance and superior energy efficiency when given comparable memory bandwidth. However, these work typically optimize BFS or relatively general graph processing with dedicated circuit design using hardware description language (HDL). The HDL based designs with customized circuits are beneficial to the resulting performance and save resource consumption, but it usually takes a long time for development, upgrade, maintenance and porting to a different FPGA device, which are all important concerns from the perspective of the accelerator developers.

To alleviate this problem, more and more people from both industry and academia opt to use integrated high level synthesis (HLS) design tools for developing accelerators of their target applications namely accelerating with software programmable FPGAs [9], [10]. Basically, the users can focus on the application and program the FPGAs with high level languages such as C/C++ and OpenCL without handling the neither the low-level circuit design nor system drivers. We argue that using software programmable FPGAs for hardware design is not only improving the productivity of the designers, it is also beneficial to the high-level application users when the resulting FPGA designs get continuous and steady FPGA vendor support on software stacks including the drivers and runtime environment. Nevertheless, the HLS based design tools are mostly used for applications with relatively regular memory access patterns and data paths. It remains challenging to accelerate BFS with various memory access patterns and complex data paths.

Under such a context, we choose to build the BFS accelerator using SDAccel which is a standard high level design environment targeting the data-center application acceleration to endow the accelerator with software-like features. Then we further optimize the high level BFS accelerator to achieve high performance. As BFS is known to be memory bandwidth bound, we thus centers the memory access optimization. With intensive experiments on memory access patterns of BFS on large graphs, we proposed a series of high-level design optimizations such as prefetching and caching to the BFS accelerator. Some of the optimizations can be applied to many other similar applications. Finally, we tune the optimization parameters with high-level metrics such as cache hit rate through the convenient HLS based software emulation and obtain optimized BFS accelerator for each graph.

According to the experiments on a set of big graphs on different FPGAs, the optimized high level BFS accelerator achieves up to 70X performance speedup when compared to the baseline design. When compared to the previous optimized HDL based design, the HLS based BFS accelerator achieves over 70% of the performance and gains many software-like features such as ease of maintenance and use. Particularly, we ported the design from Alpha-Data ADM-7v3 on a desktop computer to KU115 on Nimbix Cloud. Significant performance improvement is observed and the portability of the HLS based BFS accelerator is demonstrated.

II. Background and related work

A. High level FPGA design tools

The major FPGA vendors have started to offer high-level programming options such as C/C++ and OpenCL, which

makes it possible for the designers without much low-level circuit design experiences [11], [10], [12] to program the FPGAs efficiently. In addition, the accelerator described with high-level languages preserves many software-like features such as portability, ease of maintenance and use. They are all important concerns from the perspective of the designers. Considering the continuously increasing FPGA resources and stringent time-to-market requirements, the high level FPGA design tools [13] are getting more and more popular.

In line with this trend, we use SDAccel [10] to develop the high level BFS accelerator in this work. SDAccel is a design environment targeting x86-Based server + PCIe based FPGA acceleration card. Compared to the conventional design flow, it provides a number of features including standard OpenCL API, C/C++ style design entry and fast software emulation to make the FPGAs programmable to software designers.

B. BFS Algorithm

BFS is a widely used graph traversal algorithm and it is the basic building component of many other graph processing algorithms. It traverses the graph by processing all vertices with the same distance from the source vertex iteratively. The set of vertices which have the same distance from the source is defined as the frontier. The frontier that is under analysis in the BFS iteration is named as current frontier while the frontier that is inspected from current frontier is called next frontier. By inspecting only the frontier, BFS can be implemented efficiently and thus the frontier concept is utilized in many BFS implementations.

A widely used frontier based BFS algorithm implementation is named as level synchronous BFS [1], [2], [14]. The basic idea is to traverse the frontier vertices and inspect the neighbors of the current frontier vertices to obtain the frontiers in next BFS iteration. Then the algorithm can start a new iteration with a simple switch of current frontier queue and next frontier queue. The algorithm ends when the frontier queue is empty.

C. Related work

The growing importance of efficient BFS traverse on large graphs have attracted attention of many researchers. In the past few years, many BFS optimization algorithms and accelerators have been proposed on almost all the major computing platforms including multi-core processors, distributed systems, GPUs and FPGAs. In this work, we will particularly focus on the FPGA based BFS acceleration.

The researchers tried to explore BFS acceleration on FPGAs from many various angles. To alleviate the memory bandwidth bottleneck of the BFS accelerators, the authors in [14] explored the emerging Hybrid Memory Cube (HMC) which provides much higher memory bandwidth as well flexibility for BFS acceleration, while the authors in [1] proposed to change the compressed sparse row (CSR) format slightly. Different from the first two work, the authors in [5] chose to perform some redundant but sequential memory access for higher memory bandwidth utilization based on a spare matrix-vector multiplication model. In addition, they particularly took advantage of the hybrid CPU-FPGA architecture offloading only highly parallel BFS iterations for FPGA acceleration while leaving the rest on host CPU.

Most of the BFS accelerators are built on a vertex-centric processing model, while the authors in [8] explored the edge-centric graph processing and demonstrated significant throughput improvement. On top of the single FPGA board acceleration, the authors in [1], [2] also explored BFS acceleration on a FPGA based high-performance computing system with multiple FPGAs and memory instances. There are also work exploring customized soft processors for graph processing and building a distributed solution on top of a group of embedded FPGA boards [15], [16].

Instead of building specialized BFS accelerator, many researchers opted to develop more general graph processing accelerator framework or library recently [7], [6], [3], [17]. They can also be utilized for BFS acceleration despite the lack of specialized optimization for BFS. Meanwhile, this is also a way to improve the ease of use FPGAs for graph processing acceleration.

Prior BFS acceleration work has demonstrated the potential benefits of accelerating BFS on FPGAs. These accelerators were mainly developed for the sake of performance and generality for more graph processing algorithms. However, they were all handcrafted HDL designs. In this work, we opt to develop HLS based BFS accelerators using SDAccel such that the accelerator packed in an OpenCL thread can be easily utilized in high-level applications by a *software designer*. Meanwhile, the same design can be easily ported to other FPGAs with the SDAccel support with little engineering efforts.

III. Observations on BFS memory access

BFS on large graphs is known to be a memory bandwidth bound task. Thus we analyze the memory access of BFS for intensive optimization. With the analysis, three observations are gained and can be used to guide the BFS accelerator design.

Observation 1: BFS memory access pattern involves considerable short sequential memory accesses and random memory accesses which are usually quite expensive. At the same time, it also includes quite some long sequential memory accesses and the overall access time cannot be ignored due to the relatively large memory access amount. We analyze the memory access pattern of the BFS on Youtube Graph. The burst length distribution of the memory accesses is presented in Figure 1. It can be found that a large amount of random and short sequential memory accesses are involved in BFS. Meanwhile, shorter burst length especially the random memory access results in extremely low memory bandwidth. Worse still, more parallel data paths will not improve the memory bandwidth utilization too much due to the memory access conflict, which makes the random and short sequential memory access optimization difficult. Note that the memory bandwidth is obtained from a standalone HLS based memory test on an Alpha Data ADM-7v3 FPGA card. We use the test result to estimate the memory

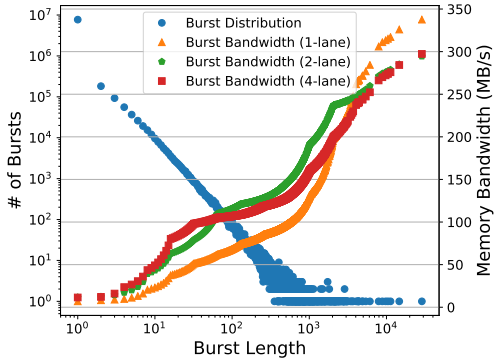

Fig. 1. Burst length distribution in BFS on Youtube Social Network Graph. Random memory access and short sequential memory access take up the the majority of the memory access overhead of BFS. Multiple parallel lanes of data paths improve the memory bandwidth utilization when the burst lenght is relatively large, but they will not help much for short or random memory accesses.

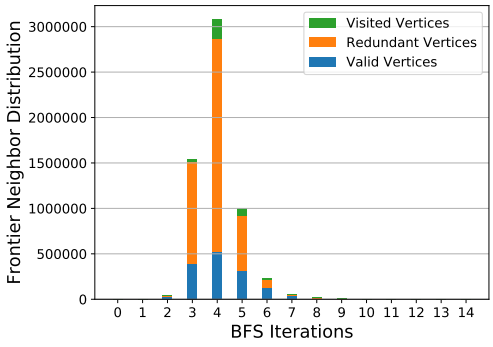

Fig. 2. Unnecessary random vertex status read decomposition. The frontier neighbors consists of three parts including visited vertices in previous BFS iterations (visited vertices), redundant vertices that are repeatedly traversed in one BFS iteration (redundant vertices), and the actual valid vertices that must be traversed (valid vertices). The first two parts of the vertices can be ignored without affecting the correctness of the BFS.

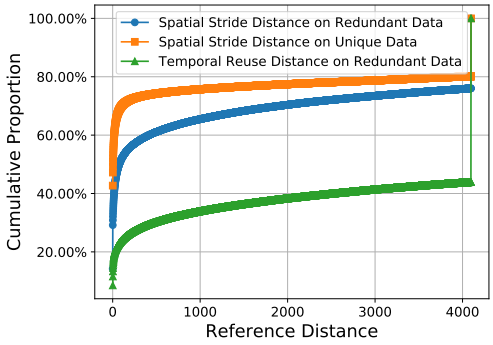

Fig. 3. Cumulative Distribution Function (CDF) of reuse distance and stride distance. They stand for the temporal locality and spatial locality of the BFS vertex status reads respectively. Note that the accesses with reference distance larger than 4000 are combined as they are difficult to be optimized in hardware design.

access time of BFS. In general, it can be concluded that the short sequential and random memory access are the most critical part of the BFS memory accesses.

Observation 2: There are many redundant vertices in the frontier vertex neighbors leading to a lot of redundant memory access in BFS. As random memory access takes up a big portion of the overall memory access overhead, we further investigate the random memory access in BFS. According to the level synchronous BFS algorithm, the random memory access mainly comes from the frontier neighbor vertex status read/write. However, many frontier vertices may have common neighbor vertices and these common vertices lead to many repeated vertex statuses read. To gain insight of the neighbor vertex redundancy, we compare the total amount of frontier neighbor vertices and the amount of unique frontier neighbor vertices in each BFS iteration. (Note that Youtube graph is used in the experiment.) The comparison is presented in Figure 2 and it shows that there are considerable redundant vertices in BFS iterations. The redundancy proportion even goes up to 80% in the BFS iteration with the most frontiers. With proper redundancy removal strategy, the vertex status reads in the following part of the BFS can be reduced dramatically.

On top of the redundant vertices, the visited vertices in previous BFS iterations do not need to be checked by reading the vertex status from memory. According to our experiment, the visited frontiers take up quite some of the total frontier neighbor vertices. However, this is observed in only one or two BFS iterations. Buffering frontier vertices may help to avoid unnecessary random vertex status read, but the benefit may be relatively trivial.

Observation 3: The vertex status reads have good spatial locality especially when the redundant access is removed in advance, but the temporal locality is relative bad meaning that the status reuse distance is quite long. Another potential optimization of random memory access is to use cache architecture, which explores the data locality and improves memory bandwidth utilization accordingly. To ascertain the feasibility of using cache, we analyze both the temporal and spatial locality of the vertex status reads. Since the frontier neighbor vertex redundancy removal affects the locality, we also do the spatial locality analysis on a redundancy-free vertex status read sequence. The data locality based cumulative distribution function (CDF) curve is shown in Figure 3. Clearly, the memory accesses exhibit very good spatial locality. In particular, the spatial locality gets even better and over 70% of the vertices have reference distance less than 200 when the redundancy is squeezed. The temporal locality is not as good and only around 20% of the accesses have short reuse distance. In general, we still believe that a specialized cache is beneficial to the random vertex status reads. Note that we use the stride distance as the metric of spatial locality and reuse distance as the metric of temporal locality [18]. The stride distance of a reference to address A is defined as the minimum distance between A and the memory addresses in a lookback window right before the current memory access. We set the loopback window to be 32 in the experiment. The reuse distance of some reference to address A is equal to the number of unique memory addresses that have been accessed since the last access to A.

In summary, we notice that random and short sequential

memory accesses are the most time-consuming part of the overall BFS memory accesses. It is generally difficult to optimize these memory accesses, With intensive experiments, we observe a large amount of redundancy in BFS and these memory accesses exhibit very good spatial locality. Taking advantage of the memory access characteristics, we may improve the BFS memory access efficiency and performance eventually.

IV. BFS Accelerator Overview

This work targets at in-memory graph processing on a PCIe based high-performance FPGA card. The whole graph is stored in FPGA device memory. Ideally, the accelerator does the BFS on FPGA without any interference from the host CPU. However, the data flow model of Xilinx HLS does not allow feedback from the downstream stages to previous stages. In this case, we actually implement one BFS iteration on FPGA while leaving the host CPU to do the iteration control. The BFS kernel on FPGA returns the BFS frontier size to host such that the host can decide if BFS completes. Although communication between the host and FPGA in each BFS iteration is required, the communication cost is negligible compared to the execution time and the overall BFS runtime is barely affected.

The BFS algorithm is critical to the BFS accelerator and level synchronous BFS is widely used. However, it may have redundant vertices pushed to the frontier queue in a parallel architecture. The redundancy vertices are difficult to get rid of *completely*, while they may lead to a large number of repeated traverses recursively and degrade the BFS performance. To address this problem, we inspect the frontier from the vertex status in each BFS iteration to cut down the propagation of the redundancy. The modified algorithm is described in Algorithm 1. Instead of traversing the frontier vertices directly, it starts with vertex status traverse and inspects the frontier in each BFS iteration. The inspection processing are complete sequential memory access and can be done efficiently and it helps to remove the redundant frontier vertices completely. The rest part of the algorithm is quite similar to the level synchronous BFS except that the frontier queues are no longer needed.

Algorithm 1 Modified BFS Algorithm

```
1:  procedure BFS
2:      level[v_k] ← −1 where v_k ∈ V
3:      level[v_s] ← 0, current_level ← 0, frontier ← v_s
4:      while !frontier.empty() do
5:          for v ∈ V do
6:              if level[v] == current_level then
7:                  frontier ← v
8:          for v ∈ frontier do
9:              S ← n ∈ V|(v, n) ∈ E
10:             for n ∈ S do
11:                 if level[n] == −1 then
12:                     level[n] ← current_level + 1
13:         current_level ← current_level + 1
```

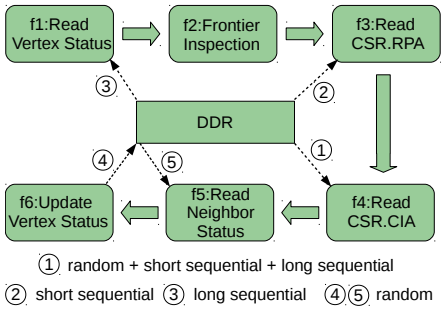

Fig. 4. Streamed BFS Algorithm

V. HLS based BFS optimization

With the observations in Section III and the modified BFS algorithm in Section IV, we start to optimize the BFS accelerator using high-level design tools from different angles. First, we convert the nested loop structure of the BFS to be a streaming manner such that it can be fit into the data flow model in Xilinx HLS for efficient pipelined execution. Then we explore a series of memory optimization techniques based on the BFS memory access patterns observed in III. Afterwards, we apply some general HLS optimizations to the resulting design. Finally, we tune the design parameters such as the prefetch buffer size and cache size through the software emulation and provide optimized configurations for each graph data set.

A. BFS pipelining

We divide the BFS algorithm into pipelined sub functions as summarized in Figure 4. The six sub functions are labeled as *f1* to *f6* respectively. In *f1*, vertex status is read from FPGA DDR memory sequentially through a streaming port. When the vertex status is fetched, *f2* inspects the status flowed from the stream buffer, decides the current frontier and dumps the frontier to the downstream pipeline. With the frontier stream, *f3* can further fetch graph data stored as CSR. CSR includes a row pointer array (RPA) and a column index array (CIA), and they must be sequentially accessed. In *f3*, we combine each pair of RPA entry of the frontier as a construct and pass it to the next stream function *f4*. When *f4* gets the RPA pair, it can read the CIA sequentially through a streaming port. When data in CIA stream which is essentially the potential next frontier vertices are received in *f5*, their vertex status will be checked by reading the vertex status array stored in DDR as well. Only the vertices that are not visited yet will be further forwarded to the *f6*. In *f6*, the vertex status will be updated.

In the streamed BFS implementation, five sub-functions involve external memory access and they have quite different memory access patterns. *f3* reads all the vertex status and it is essentially a long sequential memory read. *f2* reads the CSR row pointer of the frontier. Two sequential words will be read each time. *f1* reads the CSR column index and the burst length depends on the vertex degree which varies in a large range. *f4* and *f5* involve vertex status reads and writes of the next frontier vertices. As these vertices are not sequential, the HLS tools just take them as random access.

B. Memory Access Optimization

In this section, we mainly explore the memory access optimization techniques based on the observations in Section III.

1) Redundancy Removal: There are many redundant vertices among the frontier neighbors in BFS. In order to remove the redundant memory access, we create hash tables to perform the redundancy removal. As the redundant vertices are relatively random, a big hash table can be utilized to squeeze the redundancy. However, big hash table degrades the hardware implementation frequency and eventually lowers the overall performance. Hereby, we build a series of smaller hash tables and apply them in parallel.

2) Caching: According to the experiments in Section III, there are many random memory accesses and short sequential memory accesses in *f5* and *f6*. It is generally difficult to optimize these memory accesses. Fortunately, the spatial locality analysis shown in previous section implies the great potential of cache-based memory access optimization. Inspired by the observation, we developed an HLS based cache specifically for the vertex status *depth* access.

Since the cache is only used for *depth* array read and write, we choose the *depth* array index instead of its physical address for cache indexing. As the cache cannot be shared between different SDAccel data flow functions, a natural cache design is to implement both cache in *f5* and *f6*. The cache can be relatively simple for supporting only read operations in *f5* and write operations in *f6*. In this work, we choose the directly mapped cache considering the hardware implementation.

3) Prefetching: According to the BFS algorithm, the frontier is sequentially inspected. Therefore, the CSR information is also accessed in a single direction in *f3* and *f4*, though they are not necessarily sequential. Basically both the column array index and the row array index will increase monotonically in each BFS iteration. The CSR data will not be repeatedly referenced through the BFS. To optimize these memory accesses, a small prefetch buffer is built to improve the memory access efficiency. We notice that 64-byte prefetch buffer provides reasonable hit rate and it is used in the experiments.

C. General HLS optimization

On top of the BFS specific optimizations, there are also some other relatively general design optimizations that can improve the resulting BFS accelerator performance. These optimizations will be briefly introduced in this subsection.

1) Memory bank-aware data layout: The data layout affects the memory access efficiency especially for multiple-bank DDR memory, we thus explore the data layout of the graph for higher memory bandwidth utilization. Graph is typically stored as CSR which has two arrays i.e. the row pointer array (RPA) and column index array (CIA). A straightforward way that divides the two arrays into multiple memory banks is inefficient (i.e. using higher bits of the address as the bank selection signal), because a vertex's neighbors may stay in different banks and traversing this vertex requires accessing all the different memory banks. As a result, each processing stage must include all the memory ports attached to the different memory banks and the hardware implementation will be expensive in SDAccel. To solve this problem, we reorganize the CSR data such that the time-consuming BFS pipeline stages including *f3*, *f4*, *f5* and *f6* can be split and parallelized. Each split of the processing can operate on just one memory bank, which makes the hardware design more efficient.

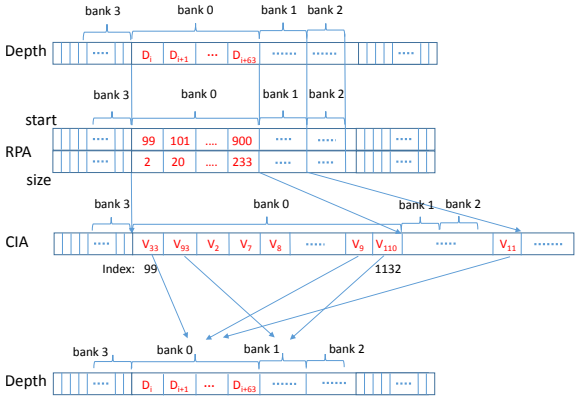

Fig. 5. BFS data layout on multiple-bank DDR memory.

The detailed data layout including both the *depth* and CSR is shown in Figure 5. The *depth* is spread evenly across the memory banks with the granularity of 64 elements. The fine-grained partition granularity ensures the load balance of the processing on different memory banks. The *RPA* of the CSR is partitioned with the same manner. To ensure aligned RPA access, we change the RPA structure with two arrays. One of the arrays stores the start *CIA* index of each vertex and the other array stores the vertex's neighbor size. The *CIA* array is allocated to different memory banks based on the *RPA* partition. Basically, we want to ensure that each vertex's neighbors stay in the same bank with its *RPA*. However, when a frontier's neighbors are read into FPGAs, the *depth* update process i.e. *f6* may still need to refer to multiple *depth* banks. We solve this problem by adding an additional crossbar stage on FPGA and reshaping the *depth* updating streams to independent memory banks. Similar methods can be applied in *f3*, so the partitions of different arrays can be different. In this work, *depth* is divided into two memory banks while CSR data are divided into 4 memory banks considering the amount of memory accesses and the total amount of memory port constrain in SDAccel.

2) Data path duplication: When the DDR memory bandwidth is not saturated, a simple yet efficient optimization method is to duplicate the data paths. With multiple parallel BFS data paths, the accelerator can issue more parallel memory requests pushing higher memory bandwidth utilization. A straightforward way of data path duplication is to split the vertex status into different partitions and each partition is processed by an instance of the same BFS data path. However, this method may not work as good as expected, because the vertices in the frontier may not distribute evenly across the graph. and the different data paths may have unbalanced workloads. Also it requires more global memory ports when

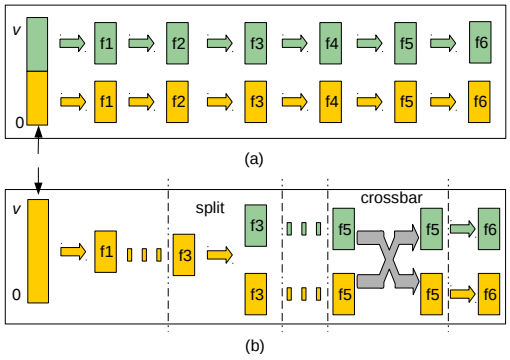

Fig. 6. pipeline duplication. (a) straightforward pipeline duplication (b) optimized pipeline duplication.

the data paths are duplicated.

To address the problems, we propose a delicate data path duplication strategy as shown in Figure 6. According to the BFS algorithm, we know that each frontier vertex requires two CSR row pointer read and multiple CSR column index read. Thus the bottleneck pipeline stages may probably start from *f3*. Therefore, we split the frontier stream generated in *f3* into multiple streams. Each sub-stream will be handled independently by a duplicated data path. This also solves the data path load balancing problem naturally. Finally, to ensure both parallel memory access and cache coherence, we reshape the output streams of *f5* into multiple independent streams with a crossbar logic such that data in each stream will not overlap and there is no memory write conflicts.

With this data path duplication strategy, each pipeline stage of a data path typically operates on an independent set of the data. When the design is implemented on an FPGA with multiple memory banks, each data partition can be mapped to a single memory bank. The data path duplication scheme makes it convenient to explore the multiple-bank memory bandwidth. It is fine to create more parallel data paths than the amount of the memory banks, but the total amount of global memory ports is limited to 16 in SDAccel. As a result, limited pipeline stages can be duplicated.

3) Data width optimization: The memory bandwidth utilization is sensitive to the data width setup. Sequential memory access with 512-bit data width achieves the optimal memory bandwidth utilization [19]. With this guideline, the design parameters such as the cache line size and prefetch length are set to be 512-bit. For sequential memory access, they are aligned and accessed with 512-bit memory port.

4) Deadlock removal: Another challenge of the pipelined BFS accelerator design is the unexpected deadlock problem. When a pipeline stage issues a burst request to the memory but gets stalled due to the insufficient read buffer, it has to wait for the downstream pipeline stages to consume the data in the buffer. However, the downstream pipeline stages may also be stalled due to the failure of acquiring the bus that is taken by the upstream pipeline stage. It is difficult to debug and resolve this deadlock. To address this problem, we add an additional local buffer in the pipeline stages with long sequential memory access and split the long sequential memory access into smaller segments such that each segment can be accommodated by the buffer. Although this may cause slightly lower bandwidth utilization, it breaks the deadlock and ensures the correctness of the pipelined design.

D. Parameter Tuning

As presented in previous sub-sections, there are some design parameters such as hash table size and cache configurations that need to be explored. However, exploring all the design parameters based on the hardware implementation directly is extremely time-consuming, because a typical hardware implementation may take hours to complete the compilation.

In this work, we tune the design parameters through the software emulation based analysis. We extract a series of implementation independent metrics such as cache hit rate and hash table hit rate through convenient software emulation of the HLS design. With these metrics, we can decide the design parameters such as prefetch buffer size, cache size, and hash table size etc. Since software emulation allows very fast parameter exploration, we consider all possible combinations of those parameters and choose the optimal configuration that leads to the best performance.

VI. Experiments

We measure the performance of the HLS based BFS accelerator on Alpha Data ADM-7v3 and KU115 using a set of representative graphs and compare them to both a baseline design and previous handcrafted BFS accelerators. The baseline design refers to a design with HLS pragmas added to the native C based level synchronous BFS implementation. Then we briefly evaluate the design optimization methods including pipelining, redundancy removal, caching and data path duplication proposed in this work. Based on the design on ADM-7v3, we further ported the design to KU115 on Nimbix Cloud and explored the portability of the BFS accelerator.

A. Experiment Setup

The graph benchmark used in this work includes three real-world graphs and two synthetic graphs generated using R-MAT model [20] as listed in Table I. The real-world graphs are from social network [21] while the R-MAT graphs are generated using the Graph 500 benchmark parameters ($A = 0.59, B = 0.19, C = 0.19$). To make the presentation easier, the five benchmark graphs are shorted as Youtube, LJ, Pokec, R-MATI, R-MATII respectively. We refer to an R-MAT graph with scale S (2^S nodes) and edge factor E ($E \times 2^S$). In order to avoid trivial search, we only choose vertices from the largest connected component as the BFS starting point.

B. Performance comparison

We use the million traverse per second (MTEPS) as the performance metric. The performance of the proposed BFS accelerator on the graph benchmark is presented in Table II. The implementation on ADM-7v3 achieves up to 82.16 MTEPS on the R-MATI graph. When compared to a baseline HLS based BFS accelerator, the proposed design shows 24.7X

TABLE I
GRAPH BENCHMARK

Name	# of vertex	# of edge	Type
Youtube	1157828	2987624	Undirectional
LJ	4847571	68993773	Directional
Pokec	1632804	30622564	Directional
R-MATI	524288	16777216	Directional
R-MATII	2097152	67108864	Directional

TABLE II
PERFORMANCE SUMMARY

Benchmark	Youtube	LJ	Pokec	RMATI	RMATII
MTEPS	14.35	28.05	36.94	82.16	32.67
Speedup	77.50	36.82	38.83	62.18	24.70

to 77.5X performance speedup on the benchmark. With the comparison, it is clear that straightforward HLS optimizations are far from sufficient and dedicated high level optimizations are critical to the performance of the resulting BFS accelerator.

We also compare this work to a set of existing BFS accelerators on FPGAs. As the platforms and graph benchmarks used in these work are mostly different, it is difficult to make a complete fair end-to-end comparison. Here we just use R-MAT graph for the comparison. We provide two implementations on Alpha-Data ADM-7v3 and KU115 respectively. A rough comparison result is listed in Table VI-B. The best HLS based BFS implementation on KU115 is getting close to that in [14], though the peak memory bandwidth is relatively higher. When compared to design on high-end FPGA computing system such as Convey HC-2 with highly optimized memory sub systems, the performance is still much lower.

Since different FPGAs may have diverse memory bandwidth, we also measure the per bandwidth BFS performance i.e. MTEPS/GB. According to the experiments, we can see that the HLS based BFS accelerator on Alpha-Data achieves higher MTEPS/GB. The comparison shows that the memory bandwidth on KU115 is not fully explored. This is mainly caused by the fact that only 16 global memory ports are allowed to be implemented in the SDAccel design and limited parallel data paths can be instantiated on the FPGAs as mentioned in previous section. We believe the performance of the proposed design can be further improved given more parallel data paths.

C. Design Configuration and Resource Overhead

With the software emulation based tuning, we can decide the design configurations rapidly. The graph specific configuration

TABLE III
FPGA BASED BFS ACCELERATOR COMPARISON ON R-MAT

Work	Platform	Graph	MTEPS	BW(GB/s)	MTEPS/GB
[2]	Convey HC-2	R-MAT	1600	80	20
[1]	Convey HC-2	R-MAT	1900	80	23.8
[14]	Micro-AC510	R-MAT	166.2	60	2.8
this work	ADM-7v3	R-MAT	57.41	10.8	5.3
this work	KU115	R-MAT	120.84	76.8	1.57

TABLE IV
MEMORY OPTIMIZATION PARAMETER SETUP ON ADM-7V3

Benchmark	Hash Table	Cache Size	Prefetch Buffer
Youtube	256K	16K × 64B	64B
LJ	**512K**	32K × 64B	64B
Pokec	1024K	16K × 64B	64B
R-MATI	512K	8K × 64B	64B
R-MATII	**512K**	32K × 64B	64B

TABLE V
FPGA RESOURCE CONSUMPTION ON ADM-7V3

Config.	FF	%	LUT	%	RAMB18K	%
Youtube	65244	7	108810	25	1515	51
LJ	65266	7	108829	25	2784	94
Pokec	65262	7	108812	25	2155	73
R-MATI	65244	7	108808	25	1217	41
R-MATII	65266	7	108829	25	2784	94

of the BFS accelerator targeting ADM-7v3 is summarized in Table IV. The hash tables for LJ and R-MATII as highlighted in the table are shrunk to fit for the on-chip memory constraints. Note that the *depth* read and write cache are set to be the same and the cache size in the table refers to the capacity of one cache size.

The corresponding FPGA resource consumption is presented in Table V. FF and LUT consumption do not change much with the different design configurations and they take up only a small portion of the total FPGA resources. Block RAMs turns out to be the major resource bottleneck, and it leads to the adoption of sub optimal design configurations.

D. Optimization evaluation

In this section, we evaluate the performance of the BFS accelerators with the different optimizations. Basically we start from the baseline design and add the optimizations including pipelining, hash redundancy removal, prefetching, caching and data path duplication in order. The performance improvement can be found in Figure 7. In general, the performance of the BFS accelerator improves significantly when more optimization techniques are applied. Particularly, pipelining and data path duplication enhance the performance most. The performance improvement brought by the hash table based filtering seems to be trivial, but it actually boosts the performance by over 20% on average. In addition, it also affects the cache efficiency as observed in Section III and is thus critical to the overall accelerator performance.

There is only one memory bank available in ADM-7v3, so we evaluate the memory bank-aware data layout strategy by porting the design to KU115. Without the bank-aware layout optimization, porting the design from ADM-7v3 to KU115 achieves less performance improvement despite the much larger memory bandwidth on KU115. When the optimization is applied, the multiple-bank memory on KU115 can be utilized. The performance of the BFS accelerator on KU115 improves significantly as shown in Table VI especially for the graphs with more edges. This experiment also demonstrated the portability of the proposed HLS based BFS accelerator.

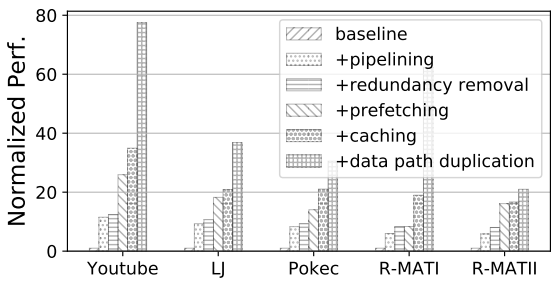

Fig. 7. BFS accelerator optimization technique evaluation. The performance on all the graphs improves when more optimizations including pipelining, redundancy removal, prefetching, caching, and data path duplication are gradually applied to the design.

TABLE VI
MEMORY-BANK AWARE DATA LAYOUT OPTIMIZATION INFLUENCE ON THE BFS ACCELERATOR PERFORMANCE (MTEPS)

Benchmark	Youtube	LJ	Pokec	RMATI	RMATII
ADM-7v3	14.35	28.05	36.94	82.16	32.67
KU115 with bank opt.	18.69	61.49	68.04	122.48	119.2
KU115 no bank opt.	14.15	41.74	40.82	85.18	63.31

VII. Insights and Conclusions

In this section, we summarize the insights and findings from this study, and conclude this paper.

A. Insights

From this study, we have obtained the following insights on HLS based graph processing. 1) The high level design tools are able to produce competitive hardware implementations for irregular graph algorithms. Given more efforts, we can create a library of HLS based graph processing algorithms in which each algorithm can be highly customized for higher performance. In this case, this can be a competitive alternative to the RTL based graph processing framework which also sacrifices the performance for the design productivity. 2) Random memory access and short sequential memory access are frequent memory access patterns. They can be optimized using the common wisdom such as caching and prefetching. However, building these logic using the HLS is error-prone and inefficient. If they are integrated into the memory access port and allows the designers to decide when it can be utilized. The HLS design tools can be beneficial to more applications with irregular memory access. 3) Optimizing the HLS design for higher performance is non-trivial. It is not so much friendly to software programmers as expected. Hardware design experiences are important for optimizing the HLS based design. We believe that those insights are helpful for further research on FPGA-based graph processing, and shed light on the design and implementation of future graph accelerators.

B. Conclusions

Handcrafted HDL based BFS accelerators usually suffer high portability and maintenance cost as well as ease of use problem despite the relatively good performance. HLS based BFS accelerator can greatly alleviate these problems, but it is difficult to achieve satisfactory performance due to the inherent irregular memory access and complex nested loop structure. In this work, we developed a series of HLS based optimizations such as redundancy removal, prefetching, caching and data path duplication. With the optimizations, BFS performance can be greatly improved. According to the experiments on a representative graph benchmark, the resulting HLS based BFS accelerator achieves up to 70X speedup compared to a baseline HLS design. When compared to the existing HDL based BFS accelerators on similar FPGA cards, the proposed HLS based BFS accelerator on KU115 gets over 70% of the MTEPS while it still preserves the software-like features including portability and ease of use and maintenance.

References

[1] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "Cygraph: A reconfigurable architecture for parallel breadth-first search," in *IPDPSW*. IEEE, 2014, pp. 228–235.

[2] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *ASAP*. IEEE, 2012, pp. 8–15.

[3] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *FPGA*. New York, NY, USA: ACM, 2017, pp. 217–226.

[4] X. Ma, D. Zhang, and D. Chiou, "Fpga-accelerated transactional execution of graph workloads," in *FPGA*. ACM, 2017, pp. 227–236.

[5] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform," in *FPL*. IEEE, 2015, pp. 1–8.

[6] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *FPGA*. ACM, 2016, pp. 111–117.

[7] N. Engelhardt and H. K.-H. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," in *FPL*. IEEE, 2016, pp. 1–4.

[8] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on fpga," in *FCCM*. IEEE, 2016, pp. 103–110.

[9] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*. Springer, 2016.

[10] Xilinx, "SDAccel," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2017.

[11] Nimbix, "SDAccel on the Nimbix Cloud," https://www.nimbix.net/xilinx/, 2017.

[12] Intel, "Intel FPGA SDK for OpenCL," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, 2017.

[13] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *TCAD*, vol. 35, no. 10, pp. 1591–1604, 2016.

[14] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search," in *FPGA*, 2017, pp. 207–216.

[15] N. Kapre, "Custom fpga-based soft-processors for sparse graph acceleration," in *ASAP*. IEEE, 2015, pp. 9–16.

[16] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multi-softcore architecture on fpga for breadth-first search," in *FPT*. IEEE, 2010, pp. 70–77.

[17] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: graph processing framework on FPGA A case study of breadth-first search," in *FPGA*, 2016, pp. 105–110.

[18] J. Weinberg, *The Chameleon framework: Practical solutions for memory behavior analysis*. University of California, San Diego, 2008.

[19] L. Kalms and D. Ghringer, "Exploration of opencl for fpgas using sdaccel and comparison to gpus and multicore cpus," in *FPL*, 2017, pp. 1–4.

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *ICDM*. SIAM, 2004, pp. 442–446.

[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.