Breadth First Search on FPGAs: A Pipelined Execution Approach

***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. Previous work typically accelerate the BFS algorithm through handcrafted circuit design with hardware description language (HDL). Despite the relatively good performance, the HDL based design leads to extremely low design productivity, and incurs high portability and maintenance cost, which dramatically compensates the performance benefit.**

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. Despite the advancements of the high level design tools over the years, it remains challenging to develop high performance BFS accelerator with these tools because of the irregular memory accesses and data paths in BFS algorithm. To obtain higher performance, we analyze the BFS memory access patterns and further propose a set of general high level synthesis (HLS) based optimization techniques such as pipeling, hash filtering, prefetching, and caching without loss of the software-like features. At the same time, we can also rapidly tune the design parameters like hash size, prefetch buffer and cache size with corresponding high level metrics including hash hit rate, prefetch hit rate and cache hit rate taking advantage of the fast software emulation of the HLS design. According to the experiments on a set of big graphs, the optimized HLS based BFS accelerator achieves up to 70X performance speedup compared to the baseline HLS design which has best-effort HLS optimization on the native BFS code. When compared to the state-of-art handcrafted design on similar FPGA cards, the proposed BFS accelerator achieves around 35% of the performance but higher per bandwidth MTEPS in some cases and 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 have 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 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. Another engineering yet non-trivial problem is the high barrier to use the FPGA powered graph processing accelerators in high-level applications such as big data analytics, which is mostly caused by the lack of well-defined high level interface and user-friendly SDK supporting various hardware systems. While improving the ease of using the HDL based accelerators requires a lot of design efforts such as driver and runtime environment support newer devices and diverse computing systems, we believe that this is also one of the key obstacles hindering the widespread adoption of the FPGA accelerators despite the great performance-energy efficiency advantages.

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 observe that there are both considerable random memory access, short sequential read and long sequential read in different parts of the BFS algorithm. For long sequential read, we use the stream model to ensure the design tools to produce efficient burst memory access. At the same time, we also explore the data width optimization and data path

duplication for higher memory bandwidth utilization. For the time-consuming short sequential and random memory access, we squeeze the unnecessary memory access by removing the redundant memory operations first. Then we take advantage of the data locality and develop a specific cache for efficient random vertex status access. Finally, we tune the design parameters with specific high level metrics such as cache hit rate through fast software emulation and obtain optimized BFS accelerator for each graph.

According to the experiments on a set of big graphs, the optimized high level BFS accelerator achieves up to 70X performance speedup when compared to the baseline design which has best-effort HLS optimization on native BFS code. Although there is still a moderate performance gap compared to previous optimized HDL based design in terms of million traverse edges per second (MTEPS), the HLS based BFS accelerator gets competitive per bandwidth MTEPS and gains many software-like features such as portability, ease of maintenance and use.

The major contributions of this work are summarized as follows.

- As far as we know, this is the first HLS based BFS accelerator on FPGAs targeting portability and ease of use on top of performance.
- With intensive experiments, we explore the memory access patterns of the BFS accelerator and exhibit the potential optimization opportunities.
- With the observations of the memory access characteristics of BFS, we develop a series of BFS accelerator optimization strategies such as pipeling, hash filtering, prefetching and caching using a standard HLS tool. The resulting accelerator shows significant performance speed up over a baseline design with best effort HLS optimization on native BFS code.

The rest part of the paper is organized as follows. In Section II, we brief the background of software programmable FPGAs and related work of BFS acceleration especially on FPGAs. In Section III, we analyze the BFS memory access pattern and motivate the BFS accelerator optimizations in this work. In Section IV, we present the overview of the BFS accelerator design using the SDAccel design environment. In Section V, we detail the proposed high level BFS accelerator design and optimizations such as pipiling and caching. In Section VI, we present comprehensive experiments of the BFS accelerator. Finally, we conclude this work in Section VII.

II. Background and related work

In this section, we briefly introduce the high level FPGA design tools. Then we review the existing BFS acceleration work and introduce the widely used BFS algorithm for the BFS accelerator design.

A. High level FPGA design tools

HDL based FPGA design is mostly limited to highly skilled hardware designers and hinders the adoption of FPGAs in more domains of applications. In addition, the HDL based design typically results in low design productivity, large reuse, portability and maintenance cost as well as ease of use challenge. To address this problem, the 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.

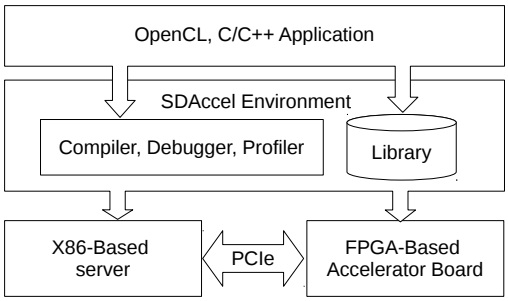

Fig. 1. Xilinx SDAccel design framework [10]

In line with this trend, we use SDAccel [10] to develop the high level BFS accelerator in this work. SDAccel overview is presented in Figure 1. It 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 to make the FPGAs programmable to software designers. First of all, it allows the designers to implement the computing kernel with either C/C++ based HLS or OpenCL. The designers can further optimize the kernel with corresponding pragmas such as loop unrolling and pipelining. Secondly, it allows the designers to perform fast software emulation with which debugging, function verification as well as some preliminary profiling of the kernel can be done rapidly. Finally, the compute kernel is wrapped as an OpenCL thread. With the provided libraries, it can be referenced through the standard OpenCL API in high level applications conveniently. With this reason, the same design can be seamlessly ported to other FPGAs with SDAccel support including the FPGAs in Xilinx Nimbix Cloud [11].

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 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.

BFS algorithm can be formalized as follows. Assume G is a graph with vertex set V and edge set E, BFS finds a shortest path from a source vertex $v_s \in V$ to all the other vertices in the graph G. For each vertex $v \in V$, BFS will output a level value l, indicating its distance from v_s (v can be accessed from v_s by traveling through $l-1$ edges.

A widely used frontier based BFS algorithm implementation is named as level synchronous BFS [1], [2], [14]. The details of the algorithm are presented in Algorithm 1. 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.

Algorithm 1 Level Synchronous BFS Algorithm

```
1:  procedure BFS
2:      level[v_k] ← −1 where v_k ∈ V
3:      level[v_s] ← 1
4:      current_frontier ← v_s
5:      current_level ← 1
6:      while current_frontier not empty do
7:          for v ∈ current_frontier do
8:              S ← n ∈ V|(v, n) ∈ E
9:              for n ∈ S do
10:                 if level[n] == −1 then
11:                     level[n] ← current_level + 1
12:                     next_frontier ← n
13:         current_level ← current_level + 1
14:         Swap current_frontier with next_frontier
```

C. Related work

The growing importance of efficient BFS traverse on large graphs have attracted attentions 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] choosed 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 have 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. Developing the HDL based accelerators takes long time and applying these accelerators on high level applications for software designers still requires a lot of efforts especially when the target computing platforms are different. To that end, 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 negligible 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 can't 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 2. (In FPGA design, longer sequential memory access can be usually taken as burst transfer from the perspective of an internal bus. We use the burst and sequential access interchangeably in this paper.) It can be found that there are a large amount of random and short sequential memory access. Meanwhile, we notice that shorter burst length especially the random memory access results in extremely low memory bandwidth when compared to that of longer sequential memory access. Worse still, more parallel data paths with the typical data width i.e. 32-bit will not improve the memory bandwidth utilization too much due

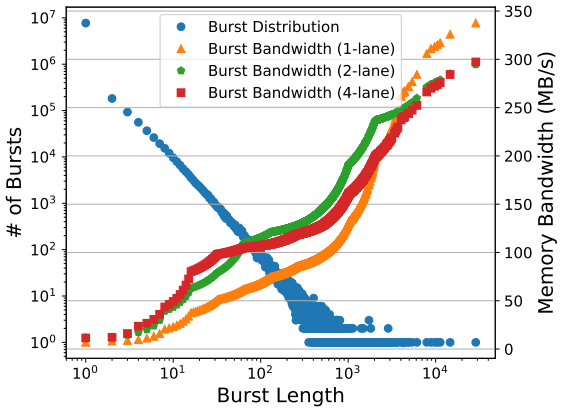

Fig. 2. Burst length distribution in BFS on Youtube Social Network Graph. Random memory access and short sequential memory access take up the the majority of the memory access overhead of BFS. Multiple parallel lanes of data paths improve the memory bandwidth utilization when the burst lenght is relatively large, but they will not help much for short or random memory accesses.

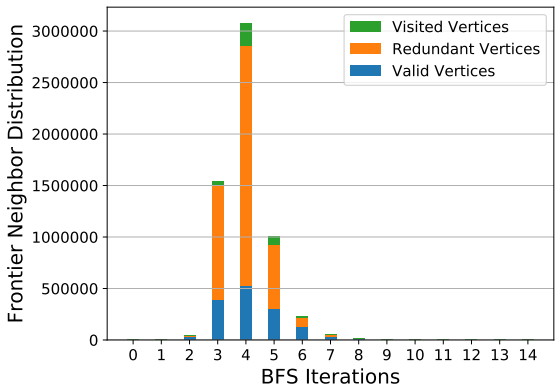

Fig. 3. Unnecessary random vertex status read decomposition. The frontier neighbors consists of three parts including visited vertices in previous BFS iterations (visited vertices), redundant vertices that are repeatedly traversed in one BFS iteration (redundant vertices), and the actual valid vertices that must be traversed (valid vertices). The first two parts of the vertices can be ignored without affecting the correctness of the BFS.

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 separate HLS based memory test on an Alpha Data ADM-PCIE-7v3 FPGA card. We use the test result to estimate the memory 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 status 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 we use the Youtube graph as an example in the experiment as well.) The comparison is shown in Figure 3 and it can be clear that there are a lot of redundant vertices in most of the 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 don't 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 are 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 4. 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 large amount of redundancy in BFS and these memory accesses exhibit very good spatial locality. Taking advantage of

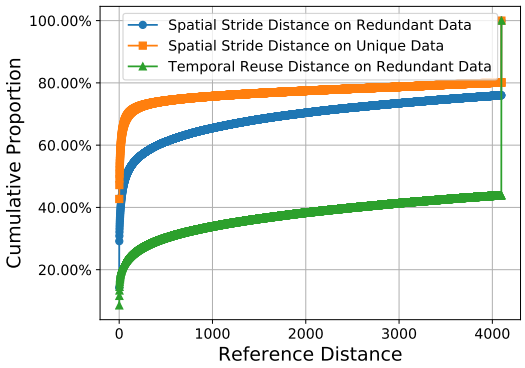

Fig. 4. Cumulative Distribution Function (CDF) of reuse distance and stride distance. They stand for the temporal locality and spatial locality of the BFS vertex status reads respectively. Note that the accesses with reference distance larger than 4000 are combined as they are difficult to be optimized in hardware design.

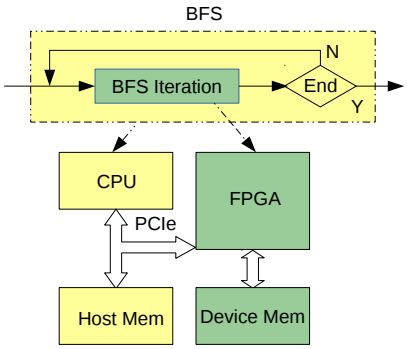

Fig. 5. BFS accelerator overview

the memory access characteristics, we may optimize the BFS memory access efficiency and improve the BFS performance eventually.

IV. BFS Accelerator Overview

Figure 5 presents the overview of the BFS accelerator. It targets in-memory graphs on a PCIe based high performance FPGA card. The whole graph is stored in FPGA device memory with a standard CSR format. Ideally, the accelerator does the BFS on FPGA without any interference from the host CPU. However, we organize the accelerator following the data flow model of Xilinx HLS and the data flow model doesn't allow feedback from the downstream stages to previous stages. (Detailed BFS data flow will be illustrated in the next section.) As a result, we can only put one BFS iteration on FPGA while leaving the iterative execution control to the host CPU. In each BFS iteration, the BFS kernel on FPGA returns the BFS frontier size to host such that the host will decide if another BFS iteration should be invoked. Although short communication between the host and FPGA in each BFS iteration is needed, the communication cost is negligible compared to the execution time. Hereby, the overall BFS runtime is barely affected.

The BFS algorithm is critical to the BFS accelerator and we explore the existing level synchronous BFS algorithm in detail. We notice that the level synchronous BFS algorithm may have redundant vertices pushed to the frontier queue in a parallel architecture especially when the frontier grows larger. The main reason roots in the large amount of overlapped neighbor vertices among the frontier vertices as mentioned in previous section. When the frontier neighbors are inspected in parallel for faster BFS traverse, these overlapped vertices may be considered as frontiers independently and inserted into the next frontier queue. As a result, there may be redundant vertices put into the frontier queue and it is rather complex to get rid of the redundancy *completely*. Although the redundant vertices in the frontier will not cause any BFS mistakes, they will soon lead to large amount of redundant traverses recursively and degrade the BFS performance.

To address this problem, we analyze the frontier from the vertex status in each BFS iteration to completely cut down the propagation of the redundant frontier vertices as proposed in prior GPU based BFS acceleration [19]. The modified algorithm is described in Algorithm 2. Instead of inspecting on the frontier vertices directly, it starts with vertex status analysis and inspects the frontier in each BFS iteration. Although it seems the additional frontier inspection stage brings more memory access, the inspection processing are complete sequential memory access and can be done efficiently. It is still worth for the overhead when compared to the cost caused by the redundant frontier vertices. The rest part of the algorithm from line 10 to 15 is quite similar to the level synchronous BFS except that the frontier queues are no longer needed.

Algorithm 2 Modified BFS Algorithm

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

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 a series of different angles. First of all, we convert the nested loop structure of the BFS to be a stream 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 characteristics observed in

III. Afterwards, we further apply some general HLS optimizations to the resulting design. Finally, we manually tune the design parameters such as the prefetch buffer size and cache size through the fast software emulation and provide optimized configurations for each graph data set.

A. BFS pipelining

The baseline BFS algorithm is a multi-level nested loop with dynamic memory accesses. It is quite challenging for the HLS tools to produce optimized hardware by adding the HLS pragmas to the native high level code directly because of the following two reasons. First of all, inner most loop and outer loop body can't be pipelined automatically by the HLS tools unless the inner most loop is fully unrolled and pipelined. Nevertheless, the inner most loop in BFS can't be fully unrolled because of the dynamic loop structure. Secondly, the outer loop nests access memory randomly even though these accesses are actually sequential because they must wait for the execution of the inner loop nest which can't be fully pipelined. This will lead to low memory bandwidth utilization. Similar problem also happens when the loop body is complex and different parts of the loop body fail to be pipelined properly. In summary, applying HLS pragmas to the native high level BFS code will produce low efficient design and the performance of the resulting hardware can be far from satisfying.

To address these problems, we divide the BFS algorithm into pipelined sub functions. Basically, there are generally two rules that we can create pipelined functions. First of all, each loop nest can be packaged into a sub function and the dependent sub functions can be pipelined. There are four loop nests in Algorithm 2, but the loop in line 12 actually include two loops as we need to go through both the CSR row and column to obtain a frontier neighbor's index. Thus we actually have five sub functions following this rule. Secondly, complex loop body can be split to more fine-grained sub functions such that the dependent parts can be executed in parallel with additional buffers between them. In BFS inner most loop, we can further divide the *depth* read and write operations to two dependent sub functions. We can do more partitions and create even deeper pipelines, while there is no guarantee for always better performance and more hardware resources are usually required.

Following the two pipelining rules, we create a six-stage pipelined BFS algorithm as detailed in Algorithm 3. 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.

Algorithm 3 Pipelined BFS Algorithm

```
1:  procedure BFS
2:      frontier_size ← 1
3:      level ← 0
4:      while (frontier_size > 0) do
5:          f1(depth, depth_stream)
6:          f2(depth_stream, frontier_stream, level, frontier_size)
7:          f3(frontier_stream, CSR.RPA, RPA_stream)
8:          f4(RPA_stream, CSR.CIA, CIA_stream)
9:          f5(CIA_stream, depth, next_frontier_stream)
10:         f6(depth, next_frontier_stream, level)
11:         level ← level + 1
12:
13: procedure f1(depth, depth_stream)
14:     for v ∈ V do
15:         depth_stream << depth[v]
16: procedure f2(depth_stream, frontier_stream, level, frontier_size)
17:     frontier_size = 0
18:     for v ∈ V do
19:         d[v] ← depth_stream.read()
20:         if (d[v] == level) then
21:             frontier_stream << v
22:             frontier_size + +
23: procedure f3(frontier_stream, CSR.RPA, RPA_stream)
24:     while (!frontier_stream.empty()) do
25:         v ← frontier_stream.read()
26:         RPA_stream << [CSR.RPA[v], CSR.RPA[v + 1]]
27: procedure f4(RPA_stream, CSR.CIA, CIA_stream)
28:     while (!RPA_stream.empty()) do
29:         [begin, end] ← RPA_stream.read()
30:         for v ∈ CSR.CIA(begin, end) do
31:             CIA_stream << v
32: procedure f5(CIA_stream, depth, next_frontier_stream)
33:     while (!CIA_stream.empty()) do
34:         v ← CIA_stream.read()
35:         if (depth[v] == −1) then
36:             next_frontier_stream << v
37: procedure f6(depth, next_frontier_stream, level)
38:     while (!next_frontier_stream.empty()) do
39:         v ← next_frontier_stream.read()
40:         depth[v] ← level + 1
```

According to the description of the streamed BFS algorithm, we notice that five sub functions involve external memory access and they have quite different memory access patterns. The memory access patterns are summarized in Figure 6. f3 reads all the vertex status and it has a long sequential memory read. f2 reads the CSR row pointer of the frontier, and it reads two sequential words 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 involves 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 without any specific hints from the designers.

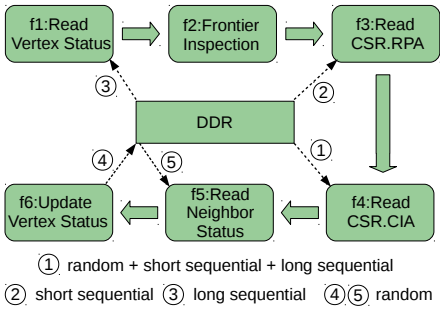

Fig. 6. Streamed BFS Algorithm

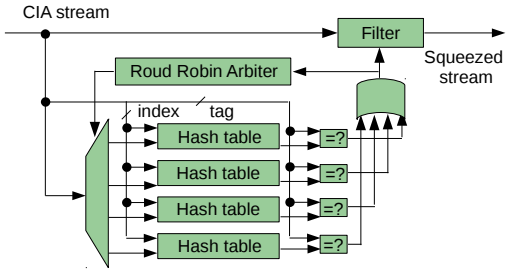

Fig. 7. Redundancy removal based on parallel hash tables.

B. Memory Access Optimization

As observed in Section III, memory access especially the random and short sequential memory accesses are critical to the BFS performance. In this sub section, we will mainly explore the memory access optimization techniques based on the observations on top of the pipelined BFS design.

1) Redundancy Removal: There are many redundant vertices among the frontier neighbors in BFS. They may further cause unnecessary vertex status reads and writes in f5 and f6 respectively. In order to remove the redundant memory access and improve the memory bandwidth utilization, 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. The hash table based redundancy removal structure is shown in Figure 7. We use the lower bits of the data address as the hash function for the sake of better timing. An input data that fails to find a record in any of the hash tables will be considered to be a unique data and put into one of the hash tables to avoid repeated data going through the filter. In order to ensure balanced hash table utilization, we implement a hardware-friendly round robin arbiter to decide the hash table updating order.

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 the observation experiments 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. Each cache line is set to be 512-bit which is equal to the recommended memory access data width and a single memory read or write operation can fulfill the requirement of the cache operations including cache read miss, cache write miss and cache write back. Since the cache can't be shared between different SDAccel data flow functions, a natural cache design is to implement both cache in f5 and f6. Both 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 for the BFS optimization. Cache line size is set to be 64B which fits well with the optimized global memory access port data width. Although set associative cache architectures will achieve higher cache hit rate, the benefit is mostly compromised by the degraded implementation frequency on Alpha Data FPGA board. The trade-off may be different on a more advanced FPGA boards, but the optimization philosophy is pretty much similar.

3) Prefetching: According to the BFS algorithm, the frontier is sequentially inspected. Therefore, the CSR information is also accessed in one direction in f3 and f4, though they are not necessarily sequential. Basically both the column array index and the row column index will increase monotonically in one BFS iteration. The CSR data will not be repeatedly referenced through the BFS. To optimize these memory accesses, a small prefetch buffer is build to improve the memory access efficiency.

Prefetch buffer is also an important design parameter that needs to be tuned. The general design trade-off is complex. A larger prefetch buffer improves the hit rate, but it may incur more memory access cost and waste the memory bandwidth if the prefetched data are not fully used. Meanwhile, larger prefetch buffer adds prefetching cost when there is prefetch buffer miss, which may degrade the performance. A smaller prefetch buffer typically lead to lower hit rate though the prefetched data will be mostly used. Nevertheless, we notice that prefetching data that is smaller than 64B will not have clear advantages on timing nor memory bandwidth utilization compared to 64B prefetch buffer. On the other hand, prefetching more than 64B data doesn't have significant hit rate improvement and even results in performance drop mainly due to the increase of the prefetch cost when there is prefetch buffer miss. In this case, 64B is set to be the prefetch buffer size in the end.

C. General HLS optimization

On top of the pipelining, redundancy removal and caching, there are also many other relatively general design optimizations that can improve the resulting BFS accelerator performance. These optimizations will be briefly introduced in this sub section.

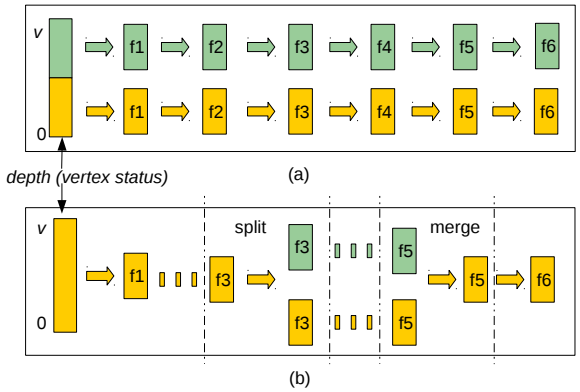

Fig. 8. pipeline duplication. (a) straightforward pipeline duplication (b) optimized pipeline duplication.

1) 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 for three reasons. First of all, the vertices in the frontier may not distribute evenly across the graph. As a result, the different pipelines may have unbalanced workloads. Secondly, when the cache is applied to the pipelined data path, the same cache line may have multiple copies stored in f6 cache and this may cause cache coherence problem when they are modified differently and written back to memory independently. Finally, duplicating the non-bottleneck pipeline stages will not be beneficial to the final performance while it incurs more hardware resource consumption including not only the basic FPGA cells but also the global memory ports.

To address the problems, we propose a delicate data path duplication strategy as shown in Figure 8. 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. In this case, we split the 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 a simple yet efficient cache coherence, we merge the output stream of f5 into a single stream before flowing into the last f6 stage with a write cache.

2) Data width optimization: The memory bandwidth utilization is sensitive to the data width setup. According to our experience, sequential memory access with 512-bit data width achieves the optimal memory bandwidth. With this guideline, a lot of design parameters such as the cache line size and prefetch length are set to be 512-bit for higher memory bandwidth utilization. For sequential memory access with smaller data width such as 32-bit, we must ensure that the data is aligned to 512-bit through padding the access. Otherwise, writing unaligned data may corrupt the data in memory.

3) Deadlock removal: Another challenge of the pipelined BFS accelerator design is the unexpected deadlock problem. When a pipeline stage issues a long 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 additional user 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 quite some design parameters such as hash table size and cache design, that need to be explored. However, exploring the design parameters based on the hardware implementation directly requires many lengthy hardware implementations (A typical BFS implementation may take 1 hour to 4 hours to complete.) which is extremely time-consuming.

In this work, we manually tune the design paramters. To obtain the optimized design parameters rapidly, we extract a series of hardware implementation independent metrics such as cache hit rate and hash table hit rate through faster software emulation mode in SDAccel which typically completes in a few minutes. With these metrics, we can roughly decide the prefetch buffer size, cache size and hash table size etc. Afterwards, we go through the lengthy hardware implementation and choose the best parameters based on the final run time. A more systematic design parameter tuning will be helpful, but it is beyond the scope of this work and we will leave it for our future work.

VI. Experiments

In this section, we measure the overall performance of the optimized HLS based BFS accelerator on Alpha Data ADM-PCIE-7v3 using a set of representative graphs. Then we compare the resulting accelerator to both a baseline design which has best-effort HLS optimization applied to native BFS code and existing FPGA based BFS acceleration work. Then we evaluate the major design optimization methods including pipelining, redundancy removal, caching and data path duplication proposed in this work.

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

TABLE I
GRAPH BENCHMARK

Name	# of vertex	# of edge	Type
Youtube [21]	1157828	2987624	Undirectional
LJ [22]	4847571	68993773	Directional
Pokec [23]	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

from social network [21], [22], [23] 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. It gets up to 82.16 MTEPS on the R-MATI graph and achieves 38.83 MTEPS on average. When compared to a baseline HLS based BFS accelerator, the proposed design shows 24.7X to 77.5X performance speedup on the five benchmark graphs. Note that the baseline HLS design refers to a native C/C++ based BFS implementation with best effort HLS pragma optimization but no modification of the source code. 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.

On top of the comparison to the baseline HLS design, we also compare this work to a set of existing BFS accelerators on FPGAs. As the platforms and graph benchmark used in these work are mostly different, it is difficult to make a fair end-to-end comparison. Hereby, we add MTEPS per unit bandwidth (MTEPSPB) as an additional performance metric showing the potential of the BFS accelerators on different hardware platforms. A rough comparison result is listed in Table III. It can be found that the HLS based BFS accelerator proposed in this work achieves around 35% of the performance in [14]. Nevertheless, the memory bandwidth is much lower compared to that in [14] and the per bandwidth MTEPS in this work outperforms especially on the R-MAT graphs. Compared to the general graph processing framework based BFS accelerator in [17] and [24], this work shows competitive per bandwidth MTEPS on either the R-MAT graph or the graph benchmark.

TABLE III
FPGA BASED BFS ACCELERATOR COMPARISON

Work	Platform	Graph	MTEPS	BW(GB/s)	MTEPSPB
[2]	Convey HC-2	R-MAT	1600	80	20
[1]	Convey HC-2	R-MAT	1900	80	24.4
[14]	Micro-AC510	R-MAT	166.2	60	2.8
[24]	VC707 Kit	Twitter	148.6	6.4	9.9
[17]	VC707 Kit	Twitter	12	6.4	1.9
this work	ADM-PCIe-7v3	R-MAT	57.41	10.8	5.3
this work	ADM-PCIE-7v3	Graph in Table I	38.8	10.8	3.6

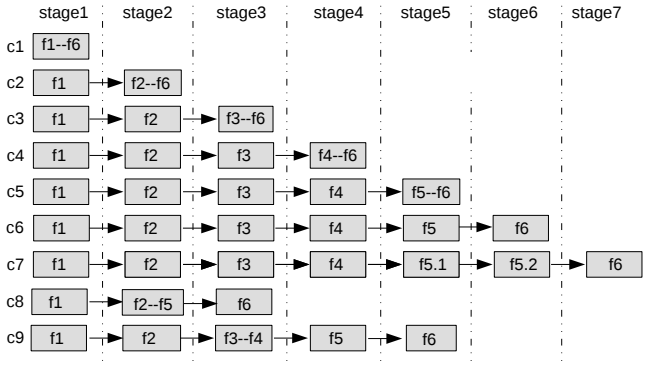

Fig. 9. Pipeline configurations with various combinations.

When compared to design on high-end FPGA computing system, the performance is still much lower while the per bandwidth performance is around 25% of the highly optimized handcraft design and the HLS based design gains the software-like features.

C. Pipelining Optimization

We can convert the BFS algorithm with nested loops to a stream design with six pipeline stages as illustrated in Algorithm 3. To explore pipelining influence, we build a series of implementations with different pipelining configurations. By dividing f5 into two dependent parts i.e. f5.1 and f5.2, we build a seven-stage pipelined BFS accelerator. By combing the pipelining stages, we also create accelerators with less pipelining stages. The different pipeline configurations are listed in Figure 9 and the performance comparison is presented in Figure 10.

According to the experiments, we find that more pipeline stages are typically beneficial to the final performance. Particularly, we observe that the resulting BFS accelerator performance has significant improvement when the number of pipeline stages jumps from 1 to 2 and from 5 to 6. Thus we try to keep the critical pipeline stages and further construct new pipeline configurations i.e. c8 and c9. The performance, however, is not improved as expected. The comparison also indicates that the internal pipeline stages are also important to the overall BFS accelerator performance. It is just that they are dependent and any one of the sub function that fails to be pipelined will stall the overall pipeline and affect the final performance.

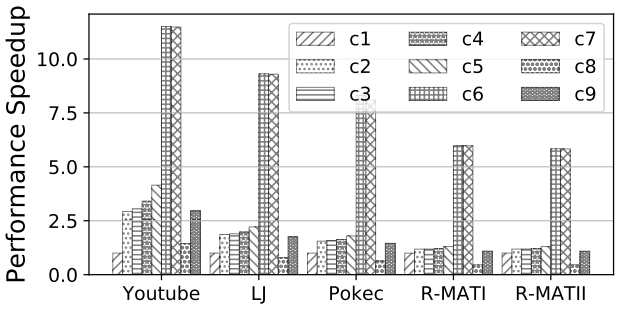

Fig. 10. Performance speedup over a baseline design c1.

TABLE IV
FPGA RESOURCE CONSUMPTION WITH DIFFERENT PIPELINING CONFIGURATIONS

Config.	FF	%	LUT	%	RAMB18K	%
c1	3581	~0	4289	~0	8	~0
c2	4121	~0	5416	~1	10	~0
c3	4208	~0	5746	~1	10	~0
c4	4494	~0	6675	~1	10	~0
c5	4551	~0	7022	~1	10	~0
c6	5546	~0	8126	~1	12	~0
c7	5853	~0	8154	~1	12	~0
c8	5113	~0	7152	~1	12	~0
c9	5408	~0	7843	~1	12	~0

Table IV shows the FPGA resource consumption of the different pipeline configurations. It can be found that deeper pipelined implementations typically consume more hardware resources including LUT and FF for constructing the computing logic and the buffers between the pipeline stages. The RAM blocks in the design are used for buffering data for global memory access and each global memory port requires two BRAM_18K blocks by default. Hereby, it is not relevant to the pipeline depth but relevant to the amount of global memory ports. Since there is almost no computing involved in the BFS accelerator design, only a fractional FPGA resources are consumed.

According to the performance and resource consumption experiments, we find that c6 achieves near optimal performance with reasonable hardware resource overhead. Therefore, it is taken as the optimized pipelining setup in the following experiments.

D. Memory access optimization

We have implemented a series of memory optimizations including redundancy removal, prefetching and caching on top of the pipelined BFS accelerator. These optimizations are tuned based on corresponding metrics obtained through software emulation.

1) Redundancy removal analysis: To squeeze the redundancy in the output stream of f4, we create a hash table based filter as presented in Section V and insert it between f4 and f5. The hash table size affects the redundancy removal rate and the performance eventually. To decide the hash table size rapidly,

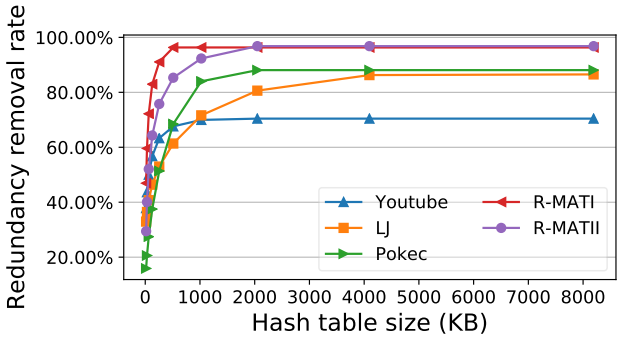

Fig. 11. Correlation between redundancy neighbor vertices removal rate and the hash table size. The redundancy removal rate varies on the different graphs. The optimized hash table size also differs.

TABLE V
HASH TABLE SIZE SETUP

benchmark	Youtube	LJ	Pokec	R-MATI	R-MATII
size (K entries)	256	2048	1024	512	1024

we thus analyze the correlation between the hash table size and the redundancy removal rate with fast software emulation.

Figure 11 shows the redundancy removal rate of different size of hash tables. Basically larger hash table can improve the redundancy removal rate in general, but the improvement gets trivial when the hash table size is large enough. In this work, we keep doubling the hash table and stop until the hash table hit rate (redundancy removal rate) improvement starts to slow down significantly. With this metric, the final hash table setup of the different graphs is given in Table VI.

2) Cache analysis: As described in Section V, we have cache structures for random *depth* reading and writing in f5 and f6 respectively. Similar to the hash table size setup, we decide the cache size based on the cache hit rate analysis while the cache hit rate can be obtained through software emulation of the HLS design.

Figure 12 shows the correlation between the cache size and the resulting cache hit rate. According to the experiment in the figure, the cache size influence varies dramatically on different graphs. Particularly, we find that R-MATI reaches nearly optimal hit rate when the cache size is 8K×64B i.e. 8K entries with 64B cache line. For the Youtube and Pokec graphs, cache hit rate is satisfactory when the cache size goes up to 16K×64B. Live Journal graph requires much higher cache size and 64K×64B cache is an optimized setup. However, it exceeds the on-chip RAM blocks on the target FPGA board. As a result, we set it to be 32K×64B in this work.

TABLE VI
CACHE SIZE SETUP

benchmark	Youtube	LJ	Pokec	R-MATI	R-MATII
size (K entries)	16	32	16	8	32

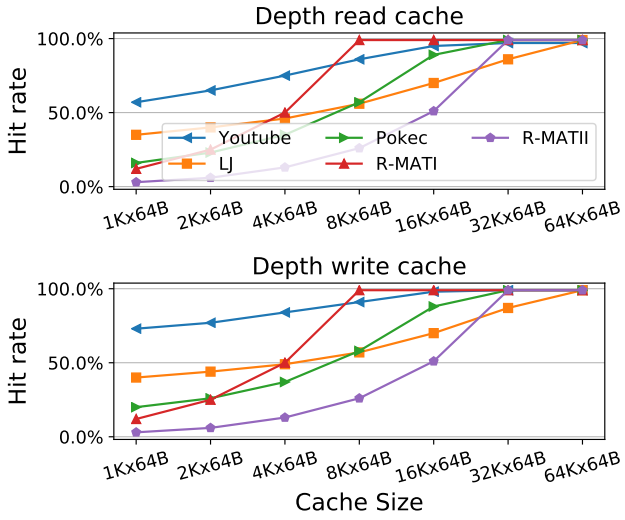

Fig. 12. Cache configurations have significant influence on the cache hit rate. The influence varies on different graph data set, but the trend is similar.

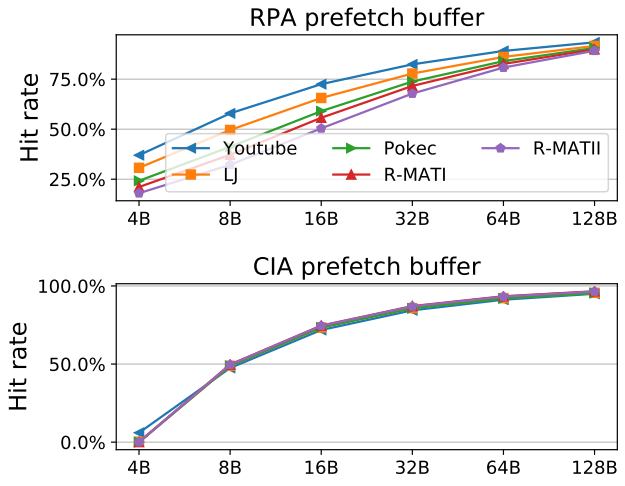

Fig. 13. A small prefetch buffer can already achieve high hit rate. Particularly the prefetch buffer size influence on different graph data set is similar.

3) Prefetch buffer analysis: We explore the correlation between prefetch buffer size and buffer hit rate through the software emulation as well. The result is shown in Figure 13. Unlike the cache in BFS accelerator, the influence of the prefetch buffer varies slightly on different graph benchmark and 64B prefetch size achieves satisfactory hit rate. 64B is also the optimized global memory access data width and a single read operation completes the prefetch at buffer miss simplifying the HLS initialization interval optimization. With these reasons, we choose 64B as the prefetch buffer size for all the different graphs. The prefetch buffer consumes negligible hardware resources, so the resource overhead is skipped to save the space.

4) Parameters of the memory optimizations: According to the experiments, both hash table and cache require large amount of block RAMs. As a result, optimized setup can't be fulfilled at the same time. Considering that cache can also avoid redundant memory access and cache hit rate is more sensitive to the cache size, we opt to provide block RAMs to cache first while leaving the rest for hash table. Prefetch buffer is much smaller compared to cache and hash table, so we set prefetch buffer to be 64B directly. With this strategy, the design parameter configurations of the memory optimization strategies are summarized and presented in Table VII. 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.

TABLE VII
MEMORY OPTIMIZATION PARAMETER SETUP

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 VIII
FPGA RESOURCE CONSUMPTION

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

The corresponding FPGA resource consumption is presented in Table VIII. FF and LUT consumption don't 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.

E. General HLS optimizations

There are many HLS optimization techniques that can be applied to the general high level designs for better performance. We mainly explore the data path replication strategies in this sub section. The rest of the optimization techniques such as data width extension, loop unrolling, and loop pipelining are already well documented in Xilinx HLS user guide [25]. We just adopt them as necessary and will not dwell in this paper.

As mentioned in Section V, we have two data path duplication strategies i.e. a straightforward duplication strategy and an optimized duplication strategy. To compare the two strategies, we analyze the percentage of frontier vertices that go through each duplicated data path in each BFS iteration. This frontier distribution over the data paths can be obtained through fast software emulation. To make the figure clear, we just put the frontier distribution of LJ in Figure 14. It can be seen that the frontier distribution varies in a large range in

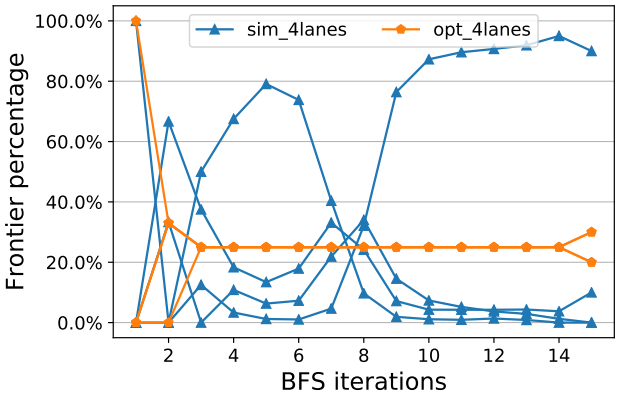

Fig. 14. Workload distribution on each data path. We take the percentage of the frontier vertices in each BFS iteration as the workload.

TABLE IX
FPGA RESOURCE CONSUMPTION WITH DATA PATH DUPLICATION

Config.	FF	%	LUT	%	RAMB18K	%
opt-DPD2	110150	12	185478	42	1893	64
opt-DPD4	194707	22	331870	70	809	84

most BFS iterations of all graphs, while the optimized data path duplication strategy addresses the problem efficiently. Note that sim_4lanes represents the straightforward data path duplication with 4 lanes. Similarly, opt_4lanes stands for the optimized data path duplication with 4 lanes.

SDAccel allows at most 16 global memory ports in a hardware thread while the amount of global memory ports increases when the data path is duplicated. Six global memory ports are needed in the basic six-stage pipelined design. Using the straightforward data path duplication, the accelerator allows at most 2 lanes of data paths. Using the optimized data path duplication strategy proposed in Sec V, 4 lanes of data paths can be implemented.

According to the above analysis, we can confirm that the optimized data path duplication strategy has clear advantage on load balance and allows more parallel data paths in the accelerator. Hereby, it is adopted in this work. When the data path duplication strategy is decided, the cache size and hash table size must be adjusted accordingly to fit for the total block ram resource constraint. Here we divide the hash table and depth read cache into each data path equally. Depth write cache stays unchanged in the merged data path. While the prefetch buffers consume little resources and they are duplicated as necessary.

The FPGA resource consumption of the accelerator with optimized data path duplication strategies are presented in Table IX. The accelerator with data path duplication incurs more logic resources including FF and LUT. Although the block RAM consumption doesn't increase too much, it remains the resource bottleneck mostly because of the large cache requirement.

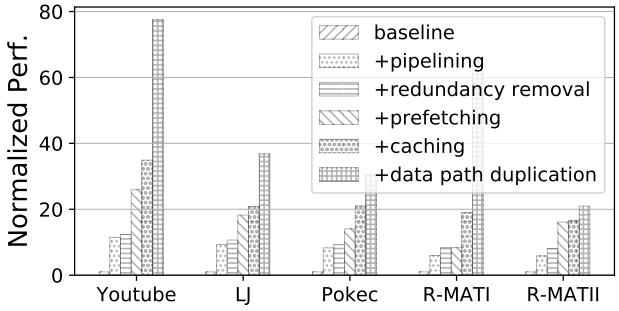

Fig. 15. BFS accelerator optimization techqniue evaluation. The performance on all the graphs improves when more optimizations including pipelining, redundancy removal, prefetching, caching, and data path duplication are gradually applied to the design.

F. Optimization evaluation

As discussed in previous sub sections, we need hardware implementation to tune the pipelining depth while we can roughly decide the rest design parameters through software emulation. This ensures the HLS based BFS accelerator can be rapidly customized for each graph data set.

After tuning the design parameters, we evaluate the performance of the BFS accelerators with the optimization. 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 with these optimizations can be found in Figure 15. 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 significantly. 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 as well.

VII. 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 stream the basic BFS algorithm and develop a series of HLS based optimizations such as redundancy removal, prefetching, caching and data path duplication. 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 with best-effort HLS pragma optimization. When compared to the existing HDL based BFS accelerators on similar FPGA cards, the proposed HLS based BFS accelerator gets around 35% of the MTEPS, but it preserves the nice software-like features including portability and ease of use and maintenance, and

achieves higher per bandwidth MTEPS in some cases mostly because of the graph specific optimization techniques.

REFERENCES

[1] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "Cygraph: A reconfigurable architecture for parallel breadth-first search," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. 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 *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*. 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 *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 217–226. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/3020078.3021739

[4] X. Ma, D. Zhang, and D. Chiou, "Fpga-accelerated transactional execution of graph workloads," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 227–236. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/3020078.3021743

[5] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. IEEE, 2015, pp. 1–8.

[6] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 111–117.

[7] N. Engelhardt and H. K.-H. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016, pp. 1–4.

[8] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on fpga," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 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, [Online; accessed 1-Sep-2017].

[11] ——, "SDAccel on the Nimbix Cloud," https://www.nimbix.net/xilinx/, 2017, [Online; accessed 1-Sep-2017].

[12] Intel, "Intel FPGA SDK for OpenCL," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, 2017, [Online; accessed 1-Sep-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," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 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 *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, 2017, pp. 207–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=3021737

[15] N. Kapre, "Custom fpga-based soft-processors for sparse graph acceleration," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. 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 *Field-Programmable Technology (FPT), 2010 International Conference on*. 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 *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, 2016, pp. 105–110. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847339

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

[19] H. Liu and H. H. Huang, "Enterprise: Breadth-first Graph Traversal on GPUs," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 68:1—-68:12, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807594

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[21] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *2012 IEEE 12th International Conference on Data Mining*, Dec 2012, pp. 745–754.

[22] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[23] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference and International Workshop Present Day Trends of Innovations*, vol. 1, no. 6, 2012.

[24] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martnez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[25] Xilinx, "Vivado Design Suite User Guide: High Level Synthesis," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf, 2017, [Online; accessed 1-Sep-2017].