FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search

Guohao Dai, Yuze Chi, Yu Wang, Huazhong Yang
Department of Electronic Engineering, Tsinghua University, Beijing, China
Tsinghua National Laboratory for Information Science and Technology
{dgh14, chiyz12}@mails.tsinghua.edu.cn, {yu-wang, yanghz}@mail.tsinghua.edu.cn

ABSTRACT

Large-scale graph processing is gaining increasing attentions in many domains. Meanwhile, FPGA provides a power-efficient and highly parallel platform for many applications, and has been applied to custom computing in many domains. In this paper, we describe FPGP (**FP**GA **G**raph **P**rocessing), a streamlined vertex-centric graph processing framework on FPGA, based on the interval-shard structure. FPGP is adaptable to different graph algorithms and users do not need to change the whole implementation on the FPGA. In our implementation, an on-chip parallel graph processor is proposed to both maximize the off-chip bandwidth of graph data and fully utilize the parallelism of graph processing. Meanwhile, we analyze the performance of FPGP and show the scalability of FPGP when the bandwidth of data path increases. FPGP is more power-efficient than single machine systems and scalable to larger graphs compared with other FPGA-based graph systems.

Categories and Subject Descriptors

B.4.4 [**Hardware**]: Input/Output and Data Communications—*Performance Analysis and Design Aids*; E.1 [**Data**]: Data Structure

Keywords

Large scale graph processing; FPGA framework

1. INTRODUCTION

We are living in a “big data” era with the great explosion of data volume generated and collected from ubiquitous sensors, portable devices and the Internet. However, large graphs are generally hard to deal with. State-of-the-art graph processing systems are generally limited by IO and computation does not take a large part in the total execution time. FPGA can handle the same computation task while providing much better power-efficiency. Researchers have proposed some dedicated solutions for certain algorithms[2, 3] and some generic solutions[11] on FPGA. The formers are restricted to special applications whereas the latter cannot scale to the order of millions because all vertices and edges need to be stored on the FPGA board.

In this work, we present FPGP, a scalable graph processing platform on FPGA that can handle billion-scale graphs with a single FPGA board. Our main contributions are as follows:

- **Enable interval-shard based vertex-centric graph processing on FPGA.** Previous work on FPGA either focuses on certain algorithm only, or supports generic algorithms but limited to less than millions of vertices due to the lack of FPGA resources. By cutting vertices into small intervals and fitting them into small on-chip memory while streaming edges from off-board storage, FPGP can scale to graphs with billions of vertices and even more edges.
- **Analyze the performance bottleneck of generic graph systems on FPGA.** Under FPGP framework, we analyze the impact factors of performance and give an optimized strategy to achieve the maximum performance. Given the analysis, we do a case study of breadth-first search to demonstrate the system performance model when the bandwidth varies.

The following of this paper is organized as follows. Section 2 shows some related graph processing systems. Section 3 describes some background ideas about graph computation tasks and introduces the algorithm. We then present the FPGP framework in Section 4. Then, we model the performance of the FPGP framework in Section 5. We demonstrate and discuss some experimental results in Section 6. Section 7 concludes the paper.

2. RELATED WORK

2.1 Single-machine CPU systems

GraphChi[10] is a disk-based single-machine system following the vertex-centric programming model. GraphChi first introduces the concepts of intervals and shards. GraphChi can handle billions of vertices and edges on a single PC.

TurboGraph[8] is another disk-based graph processing system. It presents the *pin-and-slide* model to handle larger graphs. Edges are managed as pages stored on disk and are pinned into memory when necessary. VENUS[5] is more friendly to hard disks. It introduces the Vertex-centric Streamlined Processing (VSP) model such that graph is loaded in serial but updating kernel is executed in parallel. NXgraph[6] divides edges into smaller sub-shards and sorts edges within each sub-shard, achieving a better performance compared with systems above.

All these systems use interval-shard structure to store graphs, providing the locality of data accessing.

2.2 FPGA systems

Bondhugula et al. presents a parallel FPGA-based all-pairs shortest paths solving system in [4]. This system is

This work was supported by 973 project 2013CB329000, National Natural Science Foundation of China (No. 61373026, 61261160501), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions, Xilinx University Program, Huawei Technologies Co., Ltd, and Tsinghua University Initiative Scientific Research Program.

FPGA'16, February 21-23, 2016, Monterey, CA, USA
© 2016 ACM. ISBN 978-1-4503-3856-1/16/02…$15.00
DOI: http://dx.doi.org/10.1145/2847263.2847339

Table 1: Notations of a graph

Notation	Meaning
G	the graph $G = (V, E)$
V	vertices in G
E	edges in G
n	number of vertices in G, $n = \|V\|$
m	number of edges in G, $m = \|E\|$
v_i	vertex i, or its associated attribute value
$e_{i \to j}$	edge e from v_i to v_j
I_i	interval i
S_i	shard i
$SS_{i.j}$	sub-shard $i.j$
P	number of intervals
B_v	(average) size of a vertex on disk
B_e	(average) size of an edge on disk

dedicated for Floyd-Warshall algorithm and is optimized so that a maximum utilization of on-chip resources is achieved while maximum parallelism is enabled. Betkaoui et al. introduces an FPGA-based BFS solution with high efficiency and scalability with optimization on memory access and coarse-grained parallelism. However, neither of them can address general graph computation problems. GraphGen[11] is an FPGA-based generic graph processing system using the vertex-centric model. However, GraphGen stores the whole graph in the on-board DRAM, which significantly limits the scalability of such a system.

3. PRELIMINARY

In this section, a detailed description of a graph computation task is given first, followed by a brief introduction of graph presentation methodology. The algorithm in FPGP is described in Section 3.3. The notations used in this section is listed in Table 1.

3.1 Problem description

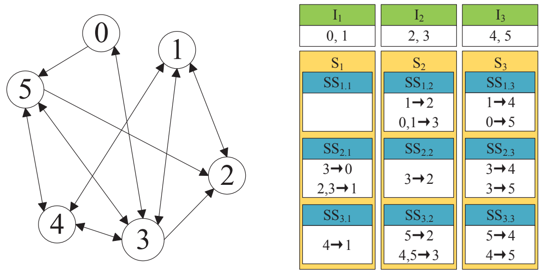
A graph $G = (V, E)$ is composed of its vertices V and edges E. A computation task on G will be updating values on set V according to the adjacency information on set E. By attaching a value to the <*source*, *destination*> pair of each edge, FPGP is applicable to both weighted and unweighted graphs. The update function used in graph computation is usually modeled as vertex-centric. In the vertex-centric model, each vertex carries a mutable attribute value, which can be updated by its in-neighbors during the execution stage. Graph algorithms are usually iterative.

3.2 Graph presentation

Many systems use intervals to store vertex attributes and shards to store edges. Moreover, each shard can be divided into sub-shards. All vertices of $G = (V, E)$, or V, should be divided into P disjoint intervals I_1, I_2, ... , I_P. All edges of G, or E, should be divided into P disjoint shards S_1, S_2, ... , S_P. Each shard S_j is further partitioned into P disjoint sub-shards $SS_{1.j}$, $SS_{2.j}$, ... , $SS_{P.j}$. A sub-shard $SS_{i.j}$ consists of all edges in E whose source vertex resides in the interval I_i and destination vertex resides in the interval I_j. This is shown in Figure 1.

3.3 Algorithm description on FPGP

FPGP adopts the updating strategy described in [6] which supports both weighted and unweighted graphs. Updates are performed in unit of intervals. Within each iteration of traversal, FPGP will iterate each interval as the destination interval and perform updates upon it. For example, in the graph presented in Figure 1, FPGP will first choose I_1 as the destination interval. To calculate the updated I_1, FPGP will first use $SS_{1.1}$ and previous values in I_1 to calculate incremental values to I_1. After that, FPGP will use $SS_{2.1}$ and previous I_2 to calculate another incremental values. Then it'll use $SS_{3.1}$ and I_3, ... , etc. When FPGP finishes iteration over source intervals, it will accumulate all the incremental values as well as optional initial values of I_1. In this way, all edges and vertices are effectively traversed and updates are incrementally performed on the destination.



I1	I2	I3
0, 1	2, 3	4, 5
S1	S2	S3
SS1.1	SS1.2	SS1.3
	1→2 0,1→3	1→4 0→5
SS2.1	SS2.2	SS2.3
3→0 2,3→1	3→2	3→4 3→5
SS3.1	SS3.2	SS3.3
4→1	5→2 4,5→3	5→4 4→5

(a) Example graph (b) Intervals and sub-shards

Figure 1: An example of intervals and sub-shards

4. FPGP FRAMEWORK

4.1 Framework Overview

The overall framework of FPGP is shown in Figure 2. The Processing Kernels(PKs) provides reconfigurable logic for designers to implement graph updating kernel function on the FPGA chip. On-chip cache stores a portion of graph data using block RAMs(BRAMs) on FPGA, thus PKs can access these data within one or two clock cycles in a random access pattern. Data controller arranges edge or vertex data from either the shared vertex data memory and the local edge data storage. The shared vertex data memory stores the vertex data and can be updated by each PK when executing the graph algorithm. Each local edge data storage stores a portion of edges in the graph and feeds these data for a slice of PKs.

4.2 Reconfigurable Processing Kernel

Designers can implement their own designs for different graph algorithms on PKs. Both the input and output of PKs are from on-chip block RAMs rather than the off-chip memory, thus the long latency caused by the poor locality of graph data is eliminated. Each PK receives an edge e and extracts the value of its source vertex, then performs the $kernel()$ function to update the value for its destination vertex, as shown in Formula (1).

$$e.dst.value = kernel(e.dst.value, e.src.value) \quad (1)$$

4.3 On-chip Cache and Data Arrangement

FPGP provides a dedicated on-chip cache mechanism for graph data and that ensures the locality when executing the graph algorithm on a subset of a graph. As mentioned in Section 3.2, vertices in the graph are divided into P intervals and edges are divided into P^2 sub-shards. Updating is performed in unit of sub-shards. When executing graph algorithm on a sub-shard $SS_{i.j}$, FPGP stores I_i and I_j in the on-chip vertex read cache and vertex write cache, respectively. The values of vertices in I_j are then updated using the values of vertices in I_i while edges in $SS_{i.j}$ are loaded in a streamlined way from external local edge storage. From the perspective of intervals, Formula (1) can be modified into Formula (2).

$$I_j = update(I_j, I_i, SS_{i.j}) \quad (2)$$

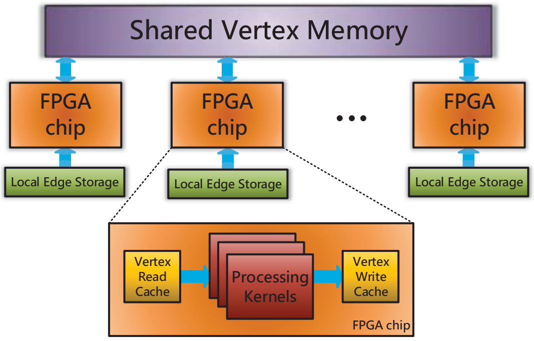


Figure 2: FPGP framework

Algorithm 1 Work flow of FPGP framework[10]

```
Input: $G$
Output: updated $V$ in $G = (V, E)$
 1: for $j = 1$ to $P$ do
 2:     Load $I_j$;
 3:     Load $I_1$;
 4:     for $i = 1$ to $P$ do
 5:         $I_j = update(I_j, I_i, SS_{i.j})$;
 6:         if $i \neq P$ then
 7:             Load $I_{i+1}$;
 8:         end if
 9:     end for
10:     Save $I_j$;
11: end for
```

4.4 Updating Strategy

In the FPGP framework, two kinds of storage are introduced, the shared vertex data memory and the local edge data storage, vertices and edges are stored in them respectively. When performing the vertex-centric updating, edges in the local edge data storage are loaded to FPGA chip sequentially whereas vertices in the shared vertex data memory are manipulated by a controller. Observing that different PKs can update different destination intervals (I_j, I_k, etc.) using the same source interval I_i, I_i can be loaded from the shared vertex data memory and issued to different PKs without increasing the bandwidth requirement of the shared vertex data memory. This storage arrangement and updating strategy makes our FPGP framework scalable as the size of graph increases.

The basic workflow of FPGP is shown in Algorithm 1. FPGP executes the *update* fuction (Line 5 in Algorithm 1) in a streamlined way, as shown in Algorithm 2.

Algorithm 2 Streamlined executing of *update* function

```
Input: $I_i$, $I_j$, $SS_{i.j}$
Output: $I_j$
for $e \in SS_{i.j}$ do
    $e.dst.value = kernel(e.dst.value, e.src.value)$
end for
```

Table 2: Notations of FPGP

Notation	Meaning
N_{pk}	number of PKs on each FPGA chip
N_{chip}	number of FPGA chips
M_{bram}	size of available Block RAMs on each chip
M_v	size of available shared vertex memory
M_e	size of available local edge storage
f	frequency of FPGA
T_{cal}	time spent on PKs
$T_{load.edge}$	time spent on loading edges
$T_{load.vertex}$	time spent on loading vertices
T_{exe}	time spent on the inner loop of Algorithm 1
$T_{total.exe}$	total executing time of each iteration
BW_{share}	bandwidth of the shared vertex memory
BW_{local}	bandwidth of each local edge storage

5. PERFORMANCE ANALYSIS OF FPGP

5.1 FPGP Model

Consider the graph $G = (V, E)$ where $|V| = n, |E| = m$. The size of a vertex value is B_v and the size of an edge is B_e. The total space requirement for vertices and edges are nB_v and mB_e, respectively. Assume all intervals have the same size $\frac{nB_v}{P}$ and all sub-shards have the same size $\frac{mB_e}{P^2}$. Here, P represents the number of intervals in the graph.

Two fundamental requirements for FPGP are that all vertices in the graph can be stored in the shared vertex memory and ping-pong operation can be performed, and all local edge storage space is sufficient to store all edges in the graph, which is shown in Formula (3) and Formula (4). For example, in our FPGP implementation, we use on board DRAM (~GBytes) as shared vertex memory. Assuming $B_v = 4$Bytes, FPGP can support graphs with millions of vertices. Because edge data can be accessed via PCI-e, the local edge storage can be extended to several TBytes.

$$M_v > 2nB_v \tag{3}$$

$$M_e > mB_e \tag{4}$$

5.2 Memory Size

Assuming the number of FPGA chips is N_{chip} and the block RAM size of each chip is M_{bram}, we implement N_{pk} PKs on each FPGA chip. Each PK executes the *kernel* function to update an I_j using the same I_i. Considering the ping-pong interval (I_i and I_{i+1}) on the FPGA chip, the total on-chip BRAM requirement is $(N_{pk} + 2) \cdot \frac{nB_v}{P}$, thus:

$$M_{bram} > (N_{pk} + 2) \cdot \frac{nB_v}{P} \tag{5}$$

P^2 iterations are required and each FPGA chip executes the *update* function $\frac{P^2}{N_{chip}}$ times (Line 5 in Algorithm 1). Assume each FPGA chip runs at the frequency of f, and the bandwidth of local edge storage is sufficient to provide the input tuple $(e.dst, e.src)$, the executing time of calculating the updated value of an interval is $\frac{1}{f} \cdot \frac{m}{P^2}$. Note that the PKs are required to be fully pipelined so that a new edge can be issued every cycle. Thus, the total executing time of calculating the updated value of all vertices V, namely T_{cal}, is shown in Formula (6).

$$T_{cal} = \frac{1}{f} \cdot \frac{m}{P^2} \cdot \frac{P^2}{N_{chip}} \cdot \frac{1}{N_{pk}} = \frac{m}{N_{chip} N_{pk} f} \tag{6}$$

Assuming the total time of loading all edges in a local edge storage is $T_{load.edge}$, Formula (6) is valid when $T_{load.edge} < T_{cal}$. Furthermore, note that calculating the updated value of I_j (Line 5 in Algorithm 1) and loading I_{i+1} are executed concurrently, the total executing time of the inner loop (Line 5 to 8), namely T_{exe}, is

$$T_{exe} = max(T_{cal}, T_{load.edge}, T_{load.vertex}) \tag{7}$$

Formula (7) shows that the total executing time of the inner loop of Algorithm 1 equals to the largest value among T_{cal}, $T_{load.vertex}$ and $T_{load.edge}$. The latter two are closely related to the bandwidth of memory in FPGP, which will be analyzed in Section 5.3.

5.3 Bandwidth

Based on the analysis above, we will show how memory bandwidth affects the performance of FPGP in this section. Assume the bandwidth of the local edge storage is BW_{local}, we can get Formula (8):

$$T_{load.edge} = \frac{mB_e}{N_{chip}BW_{local}} \tag{8}$$

As illustrated in Figure 2, all FPGA chips can access the shared vertex memory, leading to a decrease on the effective bandwidth when the number of FPGA chip N_{chip} becomes larger. However, noticing that we can update different intervals (I_j, I_k, etc.) with the same interval I_i, we can synchronize all FPGA chips before loading an interval. This is shown in Algorithm 3 (Line 3 and Line 8).

Algorithm 3 Work flow of FPGP framework with synchronization

```
Input: $G$
Output: updated $V$ in $G = (V, E)$
 1: for $j = 1$ to $P$ do
 2:     Load $I_j$;
 3:     SyncAndIssue();
 4:     Load $I_1$;
 5:     for $i = 1$ to $P$ do
 6:         for each chip do in parallel
 7:             $I_j = update(I_j, I_i, SS_{i.j})$;
 8:         end for
 9:         if $i \neq P$ then
10:             SyncAndIssue();
11:             Load $I_{i+1}$;
12:         end if
13:     end for
14:     Save $I_j$;
15: end for
```

We can get $T_{load.vertex}$ according to Formula (9), assuming the bandwidth of the shared vertex memory is BW_{share}.

$$T_{load.vertex} = \frac{P}{N_{chip}} \cdot P \cdot \frac{nB_v}{P} \cdot \frac{1}{BW_{share}} = \frac{PnB_v}{N_{chip}BW_{share}} \tag{9}$$

Note that according to Formula (5), the lower bound of P is $\frac{(N_{pk}+2)nB_v}{M_{bram}}$. To minimize $T_{load.vertex}$ so that T_{exe} can be minimized, we substitute P with its lower bound. Thus, Formula (7) can be derived into Formula (10):

$$T_{exe} = max(\frac{m}{N_{chip}N_{pk}f}, \frac{mB_e}{N_{chip}BW_{local}}, \frac{(N_{pk}+2)(nB_v)^2}{M_{bram}N_{chip}BW_{share}}) \tag{10}$$

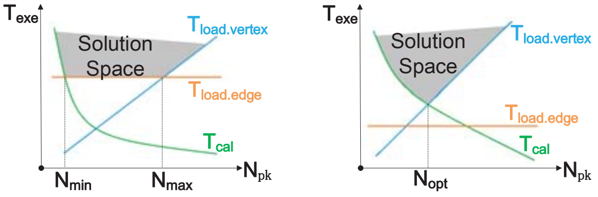
Formula (10) shows the executing time of inner loop in Algorithm 3. The total executing time of one iteration, $T_{total.exe}$, is

$$T_{total.exe} = T_{exe} + \frac{2nB_v}{BW_{share}} \tag{11}$$

The extra item $\frac{2nB_v}{BW_{share}}$ is added because we need to read the new destination interval from the shared vertex memory and write the updated destination interval to the shared vertex memory before changing the subscript j in Algorithm 1 or 3.

5.4 Performance Summary

Implementing more PKs on one FPGA chip may not lead to better performance because larger N_{pk} leads to an increase in the total number of loops, as illustrated in Formula (5). Formula (10) shows that the first item of T_{exe} is inversely proportional to N_{pk}, while the other two is constant and proportional to N_{pk}, respectively. Assuming (N_{cross}, T_{cross}) is the crossover point of T_{cal} and $T_{load.vertex}$ as N_{pk} varies, we can derive that



(a) $T_{opt} < T_{load.edge}$ (b) $T_{opt} > T_{load.edge}$

Figure 3: Solution space: when the bandwidth of the local edge storage increases and $T_{load.edge}$ decreases, the bottleneck turns into on-board processing from off-board data accessing

$$N_{cross} = \sqrt{1 + \frac{mM_{bram}BW_{share}}{f(nB_v)^2}} - 1 \tag{12}$$

$$T_{cross} = \frac{(nB_v)^2}{M_{bram}N_{chip}BW_{share}}(\sqrt{1 + \frac{mM_{bram}BW_{share}}{f(nB_v)^2}} + 1) \tag{13}$$

We can also get N_{min}, the crossover points of T_{cal} and $T_{load.edge}$, and N_{max}, the crossover points of $T_{load.vertex}$ and $T_{load.edge}$. Note that $N_{min} < N_{max}$ if and only if $T_{cross} < T_{load.edge}$.

$$N_{min} = \frac{BW_{local}}{fB_e} \tag{14}$$

$$N_{max} = \frac{mB_eBW_{share}M_{bram}}{(nB_v)^2BW_{local}} - 2 \tag{15}$$

We can see that T_{cross} reflects the best performance that on-board resources (FPGA chip, DRAM) can provide, while $T_{load.edge}$ only depends on off-board resources (PCI-e bandwidth). Thus, in Figure 3, when $T_{cross} < T_{load.edge}$, the system gets the best performance as $N_{min} < N_{pk} < N_{max}$, and $T_{exe} = T_{load.edge}$. In this situation, the local edge storage becomes the bottleneck, and the bandwidth of the local edge storage is not sufficient to provide data for processing. When $T_{cross} > T_{load.edge}$, the system gets the best performance as $N_{pk} = N_{cross}$, and $T_{exe} = T_{cross}$. In this situation, the bandwidth of the local edge storage is sufficient to provide data for processing, and a larger N_{pk} can lead to a higher degree of parallelism but also increase the times of replacing on-chip intervals, thus the whole system reaches the best performance when N_{pk} gets the optimal point.

6. EXPERIMENTAL RESULTS

In this section, we will introduce a slice of experimental results of our FPGP framework on real-world graphs. We test the system performance and compare the results with the performance analysis in Section 5. We also compare the executing time of FPGP on real-world graphs with state-of-the-art systems.

6.1 Implementation: A case study of BFS

In this section, we implement the BFS under the FPGP framework. Although BFS does not involve much computation, we can extended the results to more complicated algorithms. In FPGP, data are streamlined loaded to FPGA chip so pipeline is applicable for complicated algorithms. We use one Xilinx Virtex-7 FPGA VC707 Evaluation Kit and choose $N_{pk} = 2$ in the implementation (Figure 4). Thus, 4 interval caches are on the chip, including a ping-pong interval cache (I_i and I_{i+1}), and one result interval I_j for each PK.

In our implementation, we use a FIFO between the local edge storage and on-chip logic. By introducing the FIFO, the FPGP is capable of decoding the compressed sparse format used in the sub-shards. The updating results of I_j from two BFS kernels will be merged before written back to the shared vertex memory.

6.2 Experimental Setup

To test the performance of breadth-first search in the FPGP framework, we implement the breadth-first search algorithm according to the detailed implementation in Section 6.1. We choose Xilinx Virtex-7 FPGA VC707 Evaluation Kit as our FPGA prototype, and the FPGA chip is XC7VX485T-2FFG1761. The VC707 Evaluation Kit has an x8 PCI-e Gen2 interface to a host PC and a 1GB on-board DDR3 memory. Our implementation of FPGP on VC707 runs at the frequency of 100MHz. A personal computer is equipped with a hexa-core Intel i7 CPU running at 3.3GHz, eight 8 GB DDR4 memory and a 1 TB HDD. The computer is used to execute GraphChi as a comparison to FPGP, and we choose the Twitter[9] graph (42 millions of vertices and 1.4 billions of edges) and the YahooWeb[7] graph (1.4 billions of vertices and 6.6 billions of edges) as the benchmark.

6.3 System Performance

In this experiment, we compare the FPGP to GraphChi on the Twitter graph when performing the breadth-first search. GraphChi is a disk-based single-machine system following the vertex-centric programming model. It presents the interval-shard based graph computation model for the first time and many other graph processing systems improve the performance based on GraphChi.

There are 37.08 Mbits BRAM resources in total on the XC7VX485T-2FFG1761 chip, we use 90% of these resources. However, only about 1% LUTs are used. Although LUTs can be used as on-chip memory, it can only provide about 20% extra on-chip memory resources compared with BRAMs, which may not provide significant improvement. Meanwhile, LUTs can be used for other logics in our further implementation to improve the performance of FPGP. $N_{pk} = 2$ in our implementation and thus there are 4 interval caches on the FPGA chip, each has the size of 1MBytes, and the resource cost of our FPGP implementation is shown in Table 3.

Table 3: Resource utilization of BFS

Resource	Utilization	Available	Utilization%
FF	610	607200	0.10
LUT	4399	303600	1.45
BRAM	928	1030	90.09
BUFG	1	32	3.13

As shown in Table 3, the bottleneck of our FPGP implementation is the total block RAM resources on one chip. We simulate the performance of FPGP when $BW_{share} = 6.4$ GB/s and $BW_{local} = 0.8$ GB/s. The executing time on Twitter[9] graph of FPGP is 121.992s (13 iterations), while it takes 148.557s on GraphChi and 76.134s on TurboGraph.

Table 4: Executing time of BFS on Twitter

System	GraphChi	Turbograph	FPGP
Time(s)	148.557	76.134	121.992

As we can see, FPGP achieves **1.22x** speedup to GraphChi when $BW_{share} = 6.4$ GB/s and $BW_{local} = 0.8$ GB/s. Note that the VC707 Evaluation Kit can provide up to 12.8 GB/s of BW_{share} (by using DDR3 on the board) and up to 2 GB/s of BW_{local} (by using PCI-e). Meanwhile, our FPGP system requires less power that the CPU system.

We also test the performance on YahooWeb graph[7], which consists of 1.4 billions of vertices and 6.6 billions of edges. It takes 635.44s on FPGP while takes 2451.62s on GraphChi, thus FPGP achieves **3.86x** speedup to GraphChi. FPGP is scalable to graphs with billions of vertices.

6.4 Bandwidth

As analyzed in Section 5, both bandwidth and memory size have influence on the whole system performance. To get the best performance of FPGP, we almost use all of on-chip BRAM resources and do not change the BRAM size by implementing FPGP on different FPGA chips. The size of DRAM on VC707 board and off-board DRAM via PCI-e is able to meet Formula (3) and Formula (4). Thus in our experiment we just change the bandwidth, BW_{local} and BW_{share}.

Because the data access pattern is organized into a streamlined way in FPGP, we can use the simulated results to demonstrate the actual bandwidth of actual implementation. According to the analysis in Section 5.4, both the bandwidth of the shared vertex memory and local edge memory influence the system performance. Simply improve the bandwidth of either of them will approach a bottleneck when the bandwidth of another is limited. In this section, we simulate the performance of FPGP when the bandwidth of local edge storage and shared vertex memory varies. We choose $N_{pk} = 2$ and the FPGP runs at the frequency of 100 MHz.

BW_{share}. We vary the BW_{share} from 0.4 GB/s to 51.2 GB/s, and $BW_{local} = 0.4$ GB/s and 0.8 GB/s respectively. We test the executing time of FPGP when performing the BFS algorithm per iteration, and the result is shown in Figure 5. As we can see, the performance of FPGP will hardly be improved when BW_{share} is more than 12.8 GB/s, for BW_{local} has become the bottleneck in this situation.

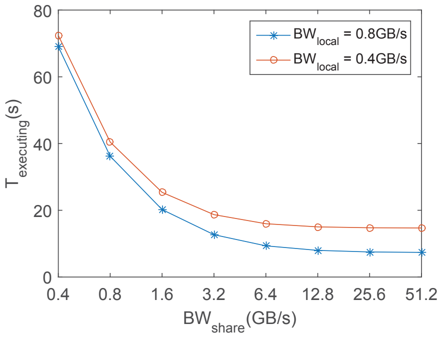


Figure 5: Performance when BW_{share} varies

BW_{local}. The local edge storage typically requires a memory space about 10 to 100 GB to store part of edges in a graph and can be carried out by using disk or via PCI-e. The typical bandwidth of these devices is about 0.5 GB/s. Thus, we simulate the performance of FPGP by varying the BW_{local} from 0.1 GB/s to 0.8 GB/s. The result is shown in Figure 6. When BW_{local} increases, the $T_{load.edge}$ line in Figure 3 continuously decline and the situation changes from Figure 3(a) to Figure 3(b). However, due to the platform limitation, we cannot get a larger bandwidth than ∼GBytes via PCI-e, the real situation of our FPGP implementation is more corresponding to Figure 3(a). Thus, improving the bandwidth of local edge storage will enhance the performance of FPGP markedly compared with improving the bandwidth of shared vertex memory, which can be inferred from our experiment results in Figure 6.

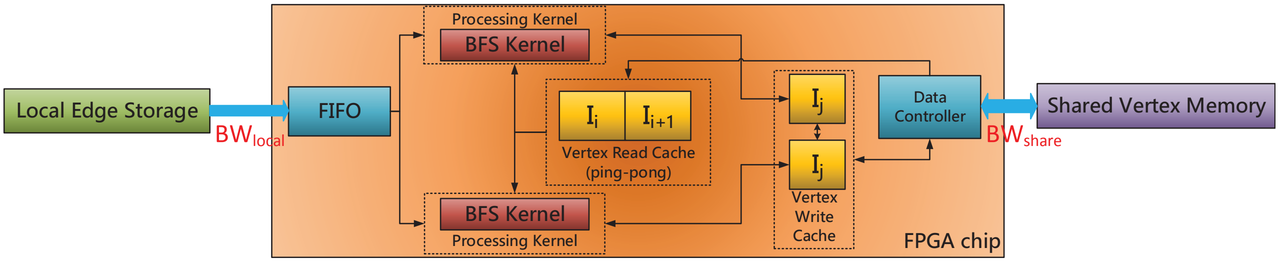


Figure 4: Detailed implementation of Breadth-First Search in FPGP

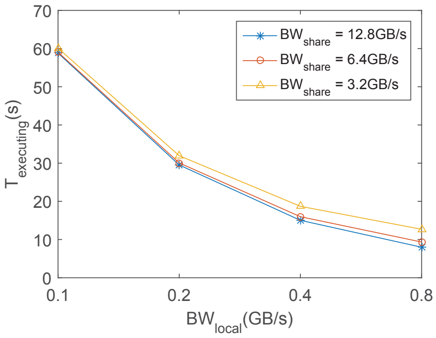


Figure 6: Performance when BW_{local} varies

6.5 Discussion

FPGP provides a framework for implementing different graph algorithms on FPGA. Compared with state-of-the-art CPU-based systems, such as TurboGraph[8] and NXgraph[6], FPGP doesn't achieve better performance on current FPGA chips.

We compare an Intel Haswell architecture CPU (i7-4770) with our Xilinx XC7VX485T-2FFG1761. The i7-4770 has 4 cores and a 256KBytes L2 cache for each core, each core runs at 3.4GHz and the latency of fetch data in the L2 cache is around 10 cycles, thus each core is equivalently running at around 300MHz when fetching data in L2 cache. In our implementation, we have 2 PKs and each PK is attached with a 1MBytes BRAM, running at 100MHz. State-of-the art systems use more powerful CPUs which have more cores, larger L2 cache size, and run at a higher frequency. Meanwhile, CPU systems store edge data in the main memory and have larger bandwidth compared with the PCI-e implementation in FPGP.

As analyzed above, to improve the performance of FPGP, one way is to improve the bandwidth of both shared vertex memory and local edge storage. However, current implementation uses PCI-e to access edge data and on-board DRAM to access vertex data, FPGP is not as competitive as CPU systems. Thus, using FPGAs with larger on-chip memory becomes a possible way. Larger on-chip memory can provide a higher degree of parallelism. Meanwhile, it can also enlarger the interval size on the FPGA chip, so that the number of intervals can be reduced and the times of replacing intervals can be lowered.

A recent work[1] shows that in-memory-computing is effective to improve the performance of graph processing. "By putting computation units inside main memory, total memory bandwidth for the computation units scales well with the increase in memory capacity."[1] Thus, allocating more memory resources to each graph processing unit is important for improving performance of graph processing, the point of view corresponds to our conclusion that an FPGA chip with lager on-chip memory resources can lead to a better performance in FPGP.

7. CONCLUSION

In this paper, we present an power-efficient large-scale interval-shard based graph processing framework based on FPGA, FPGP, which can handle generic graph algorithms on graphs with billions of edges in several seconds. Meanwhile, we model the performance of FPGP and analyze the bottleneck on a specific hardware platform. In our future work, we will support more graph algorithms under the FPGP framework.

8. REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117, 2015.

[2] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *FPT*, pages 1–8, 2011.

[3] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *ASAP*, pages 8–15, 2012.

[4] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel fpga-based all-pairs shortest-paths in a directed graph. In *IPDPS*, pages 112–121, 2006.

[5] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. Venus : Vertex-centric streamlined graph computation on a single pc. In *ICDE*, pages 1131–1142, 2015.

[6] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang. Nxgraph: An efficient graph processing system on a single machine. *arXiv preprint arXiv:1510.06916*, 2015.

[7] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. In *KDD Cup*, pages 8–18, 2012.

[8] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *SIGKDD*, page 77, 2013.

[9] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *IW3C2*, pages 1–10, 2010.

[10] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc disk-based graph computation. In *OSDI*, pages 31–46, 2012.

[11] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *FCCM*, pages 25–28, 2014.