Literature Review on Graph Acceleration

Cheng Liu

2016-12-11

Chapter 1

Literature Review

1.1 Q100

1.1.1 Highlight

The authors developed an ASIP processor for database query operations [39, 38]. The customized instructions are consist of a series of sub components. They can either be used in temporal computing or spacial computing manner. With temporal computing, each operation consumes minimal hardware taking longer execution time but allowing more parallel operations. With spacial computing manner, each operation can be done in a shorter time leaving less hardware resource for multiple parallel operations. Typically, each instruction is corresponding to a specific primitive query operation and executes in the granularity of an primitive operation, though some of the dependent instructions may work in a stream with less memory access.

The communication bandwidth and off-chip memory bandwidth to the ASIP processor are also discussed. A few interesting observations are presented and may be used to guide the design of accelerators for database query.

- Memory read bandwidth requirement is much larger than that of memory write. In this paper, the former is almost three times larger than the latter.
- On chip interconnection is used to perform the communication between the different computing tiles which are implementations of different instructions.

As the communication pattern is not uniform and the communication infrastructure can be optimized/customized as well.

- Most of the operations are not sensitive to the number of computing tiles, communication bandwidth and off-chip memory bandwidth, while some of them are very sensitive. This complicates the hardware design.

1.1.2 Questions

How does the ASIP instruction access the main memory? There is no load/store instruction specialized for the processor. Probably the instruction is able to access the main memory directly which is similar to the hardware accelerator system.

Why the speedup drops 10X when the dataset is 100X larger? Is there any scalability problem in current design?

1.1.3 Random ideas for improvement

The Q100 and LINQits have diverse design constraints.

LINQits: 1G main memory << Main memory in Q100
LINQits: 1GB/s DDR bandwidth << 5-20GB/s DDR bandwidth in Q100

Will it be a reason for the different design choices? Is it possible to propose new architecture for different design constraints?

1.1.4 Additional questions

1.2 LINQits

1.2.1 Highlight

The authors provide a configurable acceleration framework LINQits to support a domain specific query language LINQ [6]. LINQits divides the collections into smaller partitions and the computing with the granularity of a partition is processed on FPGAs. The type of the partition based computing is relatively limited and regular and thus supported in the proposed hardware template either on pre-processing or post-processing block. Basically, the hardware template reads a

partition, streams it to a queue, processes the stream and then writes back to main memory. The process continues until the end of the partition. All the operations are supported in the template and specific operation can be selected at runtime. In other words, the user-defined blocks are the same and can be replicated together with the queues and the partition readers/writers easily. In this sense, it is more like an ASIC prototype rather than a design making best use of FPGAs.

Although there are not much details about how the partitioned computing eventually complete the overall computing task, I guess it is probably handled on ARM.

Compared to Q100, it explicitly manages the on-chip memory and communication between the processor and the accelerators. Q100 integrates everything including memory access and computing into a specialized instruction. Communication is mostly done through main memory unless two operations are combined as one. In addition, the accelerator design is developed to be homogeneous and can be configured to support various operations. Q100 provides specialized circuits for different operations/instructions.

1.2.2 Questions

How is the multi-pass partition implemented? It seems the partition is done by ARM processor while FPGA just has a partition reader to read the partitioned data from main memory.

Why the author emphasizes on the divide-and-conquer algorithm? How does LINQits benefit from the algorithm?

How does the LINQits coalesce the memory access?

1.2.3 Random ideas for improvement

How much we can do to find the opportunity of fusing the basic computing operations? Is it possible to automatically decide the fused computing operations? Is it possible to develop hardware template for arbitrary fused operations?

As mentioned in the paper, the power consumption of the system can be a major constraints in future system. However, the paper mainly focus on the performance of the system instead of power consumption. Is it possible to develop a

hardware accelerated data query system in the purpose of power efficiency? According to the power consumption distribution presented in the paper, DDR access and on-chip memory (i.e. BRAM) consume the majority of the power consumption which I think should be around 70-80%. of the overall power. Is it possible to develop a more efficient memory architecture for higher power efficiency?

In Q100, the authors mentioned that the performance benefit degrades with larger dataset. Will it be similar in LINQits? If so, how can we scale the performance on such a system for larger database processing?

What can we do to improve the power efficiency of the database query accelerator?

- Interconnection: Packet switched network on chip can be replaced with circuit switched network on chip such that the NoC can be efficient used for communication purpose. Application specifc netwrok based on the traffic characteristic works as well.

- On chip memory: Power off/clock gate when it is not used, improve data reuse through pipelining, improve DDR access efficiency like memory coalescing, make best use of scratch pad memory to reduce the DDR access, on-line data compression

- Hardware Architecture: With sufficient parallelism in the application, use efficient sequential logic instead of highly fine-grained parallel logic to improve the utilization of the computing logic, Serialize corner case logic and parallelize frequent logic or even offload the corner case, ...

- General solutions: adding user specific logic for clock gating, lower the computing precision, dynamically power off the unused part of the system, DVFS on processor when it is not needed.

1.3 C++ Actor Framework Using OpenCL

This paper introduces OpenCL enabled actors to C++ Actor Framework [2, 13]. The design goals are listed as follow:

- Hide complexity of OpenCL management

- Seamless integration into CAF with respect to access transparency as well as location transparency

The concept of the Actor model seems to be interesting. It describes the applications in terms of isolated actor entities which communicate through asynchronous message passing. This abstraction helps to handle complex concurrent processing and can be distributed to larger systems.

Here is a simple explanation about the actor model which I got from Wiki. *An actor is a computational entity that, in response to a message it receives, can concurrently:*

- *send a finite number of messages to other actors*
- *create a finite number of new actors*
- *designate the behavior to be used for the next message it receives*

With this concept, the most straightforward question is whether we can build such an actor framework for FPGA computing system. Although the hardware implementation on FPGAs is relatively limited (for instance, many dynamic behaviors are difficult to be implemented on FPGAs), there is no barrier to support FPGA computing with a similar philosophy.

An actor can be an instance of the hardware accelerator and a number of actors can be implemented on FPGA devices. The actors can communicate with each other through a packed-switched interconnection network efficiently with the well-studied NoC systems. Although it is difficult to spawn new actors on FPGAs at run-time, a tightly coupled processor can help with this and new actors can be scheduled to idle accelerator instances.

Eventually, the users need to provide two things **1)the application organized with the actor framework 2)and hardware implementations using either HDLs or HLS tools for all the different actor instances**. When the two aspects are ready, the framework is supposed to provide an efficient parallel execution of the application on a CPU-FPGA computing system.

From a hardware designer's point of view, the system works similar to a data driven computing machine. It is just that the communication granularity is now a message instead of a data and the computing granularity is an larger actor instead of an operation.

Major concerns: There are also similar projects in FPGA community. The researchers tried to integrate a hardware accelerator as well as its drivers as a thread. A lot of threads can be implemented on the same FPGAs. A parallel application can be thus abstracted as many threads and the FPGA is taken as a pool of threads. By scheduling different threads in to the FPGAs at runtime, they can also complete larger parallel applications.

What kinds of applications will fit the Actor Framework? Will the applications be too complex for FPGAs?

It seems the major benefit of the actor framework is to automatically parallelize the application and the scheduler is critical to the performance of the resulting system. Then how are we going to implement the scheduler? Will it be too complicate for hardware? It is possible to put the scheduler on a processor, but it can be a bottleneck as the hardware actor may keep waiting for its response.

Each actor may require specific hardware accelerator, and all the actors to be used in the application should be implemented at compilation time. If all the actors are implemented at the same time, the hardware resource can be a key design constrain. If runtime reconfiguration is supported, the reconfiguration time can be a big challenge and performance may degrade a lot. Basically, the performance of the system may not be as good as expected. Or the usage of the system can be limited to very large FPGA system.

Compared to CPU and GPU, FPGA wins on a small set of computing tasks. Thus many Actor may not work well on FPGAs. The user need to decide what should be implement as an actor to be executed on FPGAs. This can still be a difficult hardware/software partition problem. Abstracting the communication messages to NoC packages is another problem yet to explore.

1.4 xDGP

With the observation that graph partition has deep influence on the performance of graph traversing and optimal partition for a dynamic graph changes with time, thus it is critial to present a dynamic graph partition mechanism. This work [35] adopts vertex migration to achive the dynamic partition. The vertex migration strategy is drived from label propagation algorithm in data mining. Basically a vertex decides the migration based on its neighbors. As the algorthm uses only

local information, it ensures the performance as well as the scalability.

The basic idea of this work is simple and it is developed over google pregel framework. The authors spent most of the pages on experiments.

1.5 Accelerator Design for Graph Analytics

This paper [28] is essentially a graph accelerator implementation framework of Gather-Apply-Scatter model as in GraphLab [27]. It particularly focuses on iterative graph-parallel applications with asynchronous execution and asymmetric convergence. In order to support domain of graph processing, it has a template of hardware for common operations including memory access, synchronization and communication. In order to provide application specific optimization, a design space exploration is also supported like typical domain specific accelerators.

1.5.1 Highlights

Here are the list of highlights of this work.

- It targets graph applications with asynchronous execution and asymmetric convergence which doesn't work well on GPUs. This is also one of the major reasons that contributes to the the high power efficiency.
- It provides a hardware version of GraphLab [27] and maintains sequential consistency model with a synchronous unit (SYU) which essentially follows the edge consistency.
- memory access optimization: It has special cache, load, store units for each data type such as vertex information (VI) and edge information (EI). The cache structure is also configurable to meet the requirements of as VI and EI which have different locality characteristic.
- Graph partition: The framework has each accelerator optimized for fine grained operation level parallelism. And it also replicates the accelerator unit to explore high-level parallelism based on a static graph partition.

1.5.2 Questions

Is it necessary to maintain sequential consistency, will it be possible to loose the consistency model for more parallelism and higher performance?

According to the Graphicionado [**?**], cache may not be a good memory hierarchy for graph processing as the graph problems typically has poor locality. Will a scratch pad memory work better for this design? This work utilize vertex and edge as the basic cache granularity instead of general data type may probably alleviate the problem.

The graph partition is not detailed, how does the partition affect the overall system performance?

1.6 Graphicioando

This paper utilize GraphMat [34] as the graph processing framework. With the observation that graph processing has ineffective usage of both on-chip memory and bandwidth, this work particularly optimizes the on chip memory usage over the baseline hardware accelerator obtained from GraphMat.

According to the pipeline of the baseline accelerator, the on chip memory access characteristics of the different pipeline stages are analyzed and a few optimizations are applied to remove the bottleneck of the accelerator pipeline. Here are the list of the major optimizations.

- It uses on-chip eDRAM as scratchpad memory to alleviate the random destination vertex and edge ID access. For the rest of the sequential memory access, a prefetch scheme is used to hide the memory access latency,

- Instead of replicating the accelerator directly, the authors divide the processing phase into source oriented portion and destination oriented portion. With this strategy, the hardware can be easily partitioned into parts without overlaps. Basically, the scratchpad memory is shared among the vertex processing streams.

- The number of edges are typically much larger than that of the vertex, so the edge access is usually a bottleneck of the accelerator design. This work

uses an array of input queues and output queues connected with a crossbar to access memory and feed data to the downstream processing.

- In order to cope with graphs with larger scratchpad memory requirements, the graph is sliced, though the slicing is a simple one.

1.7 Database query with hardware/software co-design

This work is developed to handle OLTP, but it doesn't show any special design optimization for OLTP system. According to my understanding, the FPGA acceleration for query itself is kind of OLTP support.

Here are the highlights of this paper.

- The design has a query control block (QCB) included to support configuration for different queries. Basically, it bridges the gap between the database query and the accelerator.
- This work details how the database query can be transformed to the accelerator configuration which I seldom see in other papers (page 9). The authors argue that SQL statement doesn't include enough information for the accelerator and they transform the query operations based on the output of DBMS transformation instead of AST derived from SQL directly.
- This paper is an extension of previous work. It particularly presents the sorting (Tournament tree sorting) and predicate evaluation implementation.
- When the query can't be mapped to the FPGA accelerator, the query operation can be decomposed to sub-operations and intermediate results will be stored as temporary files. Then it relies the software to merge the temporary files for the result.

1.8 Gunrock

Gunrock [36] is a data-centric graph processing framework which mainly handles graphs that can be expressed with iterative convergent processes. It is also based

on the BSP model. Basically, the graph problems are divided into steps and each step must be synchronized. Advance-filter-compute are the typical primitive steps used in Gunrock. These operations are performed on top of graph frontiers which is used in many graph processing framework. Major optimization strategies used in Gunrock are listed below.

- Kernel fusion: Inspired by the primitive GPU optimization strategy which leverages the producer-consumer locality between operations, Gunrock tries to integrate advance, filter and user-specific functor into a thread to improve performance as well as memory access efficiency.

- Workload balance: Gunrock's advance step which is applied on graph frontier result in severe load balancing problem especially for graphs with power-law distribution just as any other similar framework. To solve this problem, Gunrock roughly divides the vertices in the frontier into three groups based on the size of the associated neighbor lists. Vertices with larger size of neighbor lists will be executed in parallel in a cooperative thread array (CTA). Vertices with medium size will be executed in parallel in a warp. Vertices with small size will be executed in separate threads. When multiple vertices are assigned to a CTA or warp, the vertex with largest size of associated neighbor list will be distributed to all the threads of the CTA or warp and be executed first. Then all the vertices will be executed sequentially with the same logic.

- Idempotent vs. non-idempotent operations: As vertices in current frontier may share the same neighbors, there are duplicate vertices when producing the next frontier. Gunrock removes some of the duplicate vertices with inexpensive heuristics for applications that allows duplicated vertices in frontier. This strategy helps to improve performance. For applications that can't tolerate duplicated vertices, it can also remove duplicate vertices completely.

- Pull and push traversal: This is the same with the top-down and bottom-up traversal strategy used in Ligra.

- Priority queue: Usually all the vertices in the frontier are treated equally in BSP model while Gunrock divides them into two queues based on a criterion to save work.

1.9 Medusa

The goal of Medusa framework in the first place is to ease the general graph applications on GPUs. Here are the major contributions of Medusa.

- Edge-Message-Vertex (EMV) model: it provides 6 APIs which used for fine-grained edge and vertex processing.
- Memory optimization: Graph aware buffer scheme which avoids the message grouping overhead.
- Multi-GPU support: With vertex and edge replication, the graph is equally partitioned. Multi-hop replication is also discussed.

1.10 Dynamic Graph Processing

Problems of processing dynamic graph with its continuous updates in real-time manner

- Graph storage: Modification of graph structure is costly
- Fast response: Widely used global manner cannot achieve real-time processing
- Workload imbalance: Some vertices update more frequently and consume more computing resource in a certain period of time.

Ideas:

- Random-access graph structure: Hash-based graph partition strategy to enable fine-grained graph updates
- Incremental graph processing: Vertex-based incremental graph computing model
- Workload rebalance: Detect hotspots and evaluate their workload, and then rebalance them with greedy algorithms.

1.11 Ligra

Ligra [32] is a light-weight graph processing framework targeting shared-memory multi-core processing system. It simplifies graph traversal algorithm implementation on top of a few parallel libraries including cilkplus and openmp. Given a graph traversal algorithm, the user only needs to provide the implementation of a few half-done routines (i.e. Algorithm specific compute routine, F routines) and the framework will handle the rest. On top of the ease of the parallel graph algorithm development, this work contributes on the following aspects:

- It uses vertexSubset to support the frontier based graph traversal.
- It supports both edgeMap and vertexMap to meet requirements of different algorithms.
- It allows both graph traversal with a bottom-up and top-down approach for the sake of performance. Basically, iterations with larger frontier will prefer the bottom-up approach while the rest will adopt the top-down approach.
- Edge/vertex representations including sparse and dense are both supported.

Communication and synchronization relies on the atomic operation (compare-and-swap) over the shared memory.

1.12 Ligra+

This work [33] is an extension of the Ligra framework. It focuses on the graph compression on top of Ligra.

Inspired by the byte compression used in sparse matrix-vector multiplication, this work proposed a run-length byte encoding method. The basic idea is to encode the edges with variable length instead of the fixed byte block used in previous work. To support the run-length encoding, a head byte is used to specify the lengthy of each encoding block as well as other controlling information. The overall encoding roughly consists of three steps. 1) Edges associated with each vertex are sorted. 2) Difference encoding is applied on the sorted edges. 3) run-length encoding is further used. Due to the variation of edge numbers associated on each vertex, only

verteies with large associated edges are encoded. Then the encoding and decoding are further parallelized.

1.13 Gunrock

Gunrock [36] is a data-centric graph processing framework which mainly handles graphs that can be expressed with iterative convergent processes. It is also based on the BSP model. Basically, the graph problems are divided into steps and each step must be synchronized. Advance-filter-compute are the typical primitive steps used in Gunrock. These operations are performed on top of graph frontiers which is used in many graph processing framework. Major optimization strategies used in Gunrock are listed below.

- Kernel fusion: Inspired by the primitive GPU optimization strategy which leverages the producer-consumer locality between operations, Gunrock tries to integrate advance, filter and user-specific functor into a thread to improve performance as well as memory access efficiency.

- Workload balance: Gunrock's advance step which is applied on graph frontier result in severe load balancing problem especially for graphs with power-law distribution just as any other similar framework. To solve this problem, Gunrock roughly divides the vertices in the frontier into three groups based on the size of the associated neighbor lists. Vertices with larger size of neighbor lists will be executed in parallel in a cooperative thread array (CTA). Vertices with medium size will be executed in parallel in a warp. Vertices with small size will be executed in separate threads. When multiple vertices are assigned to a CTA or warp, the vertex with largest size of associated neighbor list will be distributed to all the threads of the CTA or warp and be executed first. Then all the vertices will be executed sequentially with the same logic.

- Idempotent vs. non-idempotent operations: As vertices in current frontier may share the same neighbors, there are duplicate vertices when producing the next frontier. Gunrock removes some of the duplicate vertices with inexpensive heuristics for applications that allows duplicated vertices in frontier.

This strategy helps to improve performance. For applications that can't tolerate duplicated vertices, it can also remove duplicate vertices completely.

- Pull and push traversal: This is the same with the top-down and bottom-up traversal strategy used in Ligra.
- Priority queue: Usually all the vertices in the frontier are treated equally in BSP model while Gunrock divides them into two queues based on a criterion to save work.

1.14 Work-efficient Parallel GPU Methods for SSSP

The algorithm developed in this work [8] is based on delta-stepping algorithm. It consists of the following steps.

- The vertices are divided into one or multiple buckets. Vertices within a bucket is processed in parallel.
- Traverse the vertices in a bucket. Load balancing is the key challenge in this step.
- Decide vertices to be processed in next iteration.

This work comes from the same group of Gunrock and it focuses on SSSP optimization on GPUs. Although the optimization techniques used target SSSP, they are generalized and ported to Gunrock. Thus this work is reviewed for more details about the graph optimization techniques. Here are the highlights of this SSSP optimization techniques.

- load balanced graph traversal:

  Group blocking: The edges of the vertices within a block are stripped from each vertex's edge list and processed by a cooperative thread array (CTA) in parallel. Threads within a block is load-balanced, but load-imbalance may still exist between the blocks. Particularly, when the vertex degree is small, this method is not efficient.

CTA+Warp+Scan [22]: The basic idea is to divide the vertices into three categories based on the size of the edge list. The method is applied in Gunrock as described in last section.

Edge partition: Instead of grouping equal number of vertices in each block, this method organizes the groups of edges with equal length to ensure strict load balance within a block.

- Work organization which essentially decides the vertices to be processed in next iteration. Again three different methods are proposed.

  Workfront sweep: The basic idea of Workfront is to compute on vertex frontier instead of all the vertice in Bellman-Ford. In addition, with the frontier queue and a vertex id table, redundant vertice in the queue are completely removed to minmize the computing work.

  Near-Far Pile: This is used in Ligra and also known as priority queue strategy in Gunrock. The basic idea is to process some of the vertices in the work queue or frontier first which helps to save the work of the computing. Particularly, the Near-Far method divides the vertices in the work queue in two piles based on the distances to the source. The distance threshold is named as delta which can be customized. When the vertices in the near pile are processed, vertices in the far pile will be analyzed to update the new near pile with the threshold increased by anther delta. Meanwhile, duplicated vertices in the far piles will be removed.

  Bucketing with far pile: This method follows the same design philosophy with Near-Far pile. While delta in near-far method may impose diverse number of vertices in near pile in different iterations. The bucketing method selects a determined number of vertices for the near pile when the number of vertices that meets the distance criteria is than the predefined number (i.e. 110 of the active vertices in this work). The rest active vertices are considered as the far pile. In next iteration, the vertices in the far pile are further selected with a increased distance criteria similarly. This method will ensure the number of near pile varies in a relatively small range which fits the GPU hardware.

By adding priority information to the vertices in the active list, we can avoid

updating or computing vertices associated with current vertices with long edges, as they will probably be updated by other paths with smaller edges and more hops. Essentially, this reduces the amount of computing and the overall run-time can be reduced as long as the GPU processing elements remain high utilization.

Here is a summary of relevant SSSP algorithms in reference papers. Some of them are the basis of the algorithms proposed in this work.

Algorithms	**Wk. Complexity**	**Type**	**Parallelism**
Dijkstra	$O(v \log v + e)$	General	Serial
Bellman-Ford	$O(ve)$	High Degree	Parallel
Delta Step	$O(v \log v + e)$	General	Coarse Parallel
PHAST	$O(v \log v + e)$	Low Degree	Preprocessing Parallel

Table 1.1: SSSP Algorithms

Dijkstra algorithm implemented with a priority queue is efficient as a sequential algorithm but expose little parallelism for parallel computing architectures.

PHAST [9]: It has a Dijkstra-like preprocessing step to pre-compute distances to vertices of high-degree. Then the Dijkstra algorithm can start from these highly ranked vertices in parallel. This algorithm works well on low-degree and high-diameter graphs. (More details will be added later.)

Delta Step [23]: Instead of processing one vertex at a time in Dijkstra algorithm, it groups vertices in buckets and process vertices in a bucket in parallel. In delta-stepping, the vertices are grouped into buckets depending on distances of the vertices from the source.

Major challenges for Delta step algorithm on GPUs.

- Delta-stepping's bucket implementation requires dynamic array that can be quickly resized in parallel.
- Fine-grained renaming and moving vertices between buckets are difficult and inefficient.
- GPU memory hierarchy is not well explored.

Bellman-Ford: It is a standard parallel algorithm for SSSP problem. Each vertex maintains the distance to the source and has neighbor vertices information updated iteratively. The algorithm completes when the algorithm converges. This algorithm suffers load imbalance for graphs with power-law distribution. Race condition occurs when the update is parallelized and atomic update is needed.

1.14.1 NXgraph

NXgraph [5] is a graph processing framework on a single machine meaning that the original graph is stored in the disk. It follows a few general optimization rules that have been applied in a few different graph processing frameworks including GraphChi, TurboGraph, VENUS, GridGraph.

- Exploit the locality of the graph data
- Utilize the multi-thread of CPU
- Reduce the amount of data transfer
- Stream disk IO

Here are the highlights of NXgraph centering the above four design principles.

- In order to explore the locality of the graph processing, vertices of the graph are divided into intervals and the edges are split in to shards. In each interval, the shards are further divided into sub-shards. Graph computation is performed with granularity of a sub-shard.
- To produce the intervals and shards, the graph needs a two-step pre-processing. In the first step, vertex IDs are re-assigned to produce continuous vertex IDs. The continuous vertex IDs facilitate efficient interval generation and pre-shard generation. In the second step, the pre-shard result are further cut into pieces i.e. sub-shards. Note that the edges in each sub-shards are sorted with the destination vertex IDs.
- NXgraph update strategies: The basic idea is to have two intervals of vertices stored in main memory while the sub-shards are streamed from disk. As the vertices are updated in each iteration and new data are needed in next

iteration, vertices must be synchronized. To approach the synchronization cost, each interval has two copies. One copy is used for read and used in current iteration and the other is updated. When the iteration moves forward, the two copies swapped. Thus the synchronization is reduced with the memory cost. In order to support the graphs of which the vertices are large than the main memory, only vertices that are computed will be loaded into main memory. And the order of the sub-shard computation is carefully organized to make sure disk read/write are minimized. Finally, a mixed solution combined the previous strategy is proposed to balance the performance and the memory overhead.

- Finally, the graph processing performance is modeled and bottleneck of the graph processing system can be found from the analytical models.

1.14.2 FPGP

FPGP [7] is the simplified version of NXgraph implemented on FPGAs. The most interesting part of this work is to build specific analytical models targeting CPU-FPGA system. The CPU-FPGA system includes multiple FPGA chips with a big shared memory. Meanwhile, each FPGA board has a private DDR. This work basically has vertices stored on the shared memory while shards of edges stored on private memory. According to the experiment, the shared memory bandwidth is not a clear bottleneck, but the private memory bandwidth is currently the major performance bottleneck. Due to the memory bandwidth, the FPGP doesn't show significant performance speedup over the multi-core counterpart.

1.15 GraphMat

On top of the vertex programming model, this work [34] focuses on transforming graph processing to sparse matrix computing. With the well-studied matrix computing on HPC, GraphMat achieves very good performance and scales on multi-core systems.

With this framework, it is possible to develop graph processing accelerator over spare matrix accelerator which is also well-studied.

Although a few classical graph algorithms including BFS and SSSP can be transformed to spare matrix computing, there is no guarantee for general graph problems.

1.16 In-memory Graph Database for Web-scale Data

In this work [1], Graph Database Engine for Multithreaded System (GEMS) is developed for implementing Resource Description Framework (RDF) database on distributed memory high-performance cluster. In this framework, SPARQL queries will be compiled by SPARQL-to-C++ compiler. With a SGLib, the queries will be eventually converted to graph pattern matching operations. This seems to be a good application of Graph processing on RDF database, but intensive background knowledge needs to be investigated to get better understanding of this work.

1.17 Distributed Graph Engine for Web Scale RDF Data

In this work, Trinity.RDF [40] is developed for a distributed system. Previous work usually relies on join operations for processing SPARQL queries and a large amount of useless data will be generated. To avoid this problem, this work models and stores the RDF data with native graph data. The SPARQL query is represented as a query graph and the SPARQL query processing problem is transformed to be subgraph matching. Trinity.RDF is part of the Trinity project and more details can be found [24].

1.18 Accelerating the Index Traversals for In-Memory Databases

This work [18] developed an hardware accelerator for hash index lookup which is a critical operation for database query. The accelerator was built on top of a RISC

processor architecture. It shares the cache hierarchy and closely coupled with a host processor. Here are the highlights of this accelerator.

- The index operation is divided into three steps i.e. hash, walk traversal and result output. Particularly, hash and walk traversal are decoupled using a set of queues. Eventually, the two operations are pipelined.

- This work allows multiple index operations performed in parallel on the same hash table. Moreover, as the parallel walk traversals typically occurs on different rows of the hash table. These keys can fit in the same cache line thanks to the hash logic, which is also a key factor for the final good performance.

- The accelerator is built based on a RISC processor architecture. There are a few interesting design optimizations here.

  The touch instruction is developed to pre-fetch the data blocks to reduce the long memory access overhead.

  Input buffers and output buffers are nicely integrated in the processor pipeline to decouple the hash and walk traversal operations.

  A few three-operand operations are added to aid the hash operations.

Although the authors claimed that this work handles the pointer chasing based node traversal during the indexing, no optimization is actually done for this. When the node traversal of an entry of a hash table starts, it sequentially does the chasing. The system achieves the performance mainly through the multiple parallel traversals and balanced hashing.

There are few design options that may be further optimized.

- The host processor will be idled when the accelerator starts to work.

- The host processor does the index work when there are exceptions such as TLB miss. It seems that accelerator will be used only when the data set is in the main memory.

- As the cache line is typically determined, the number of the parallel keys and the size of the keys will be limited when fitting the multiple parallel keys in the same cache line.

1.19 Fast and Concurrent RDF Queries

This work [31] presented a distributed graph-based RDF queries. It follows a graph-based design by storing RDF triples as a native graph and leverages graph exploration to handle queries. Unlike previous work, it processes concurrent queries instead a single one. Basically, the clients issues SPARQL queries and then be parsed to sub procedures. The servers includes two layers. The query layer handles sub procedures from the clinets and the graph layer holds a partitioned subgraph of the RDF using RMA technique. To provide better performance, it uses fast network primitives like one-side RDMA as much as possible. A few detailed highlights are listed below.

Graph Based Modeling:

- **Graph partition:** This work adopts differentiated partition algorithms. It introduces index vertices on top of normal RDF, which helps to partition the graph and expose the parallelism as proposed in [3]. The predicate index will not be included in the edge list of normal vertex, which saves the memory space.
- **Full history pruning:** In addition, it also develops a strategy to prune the intermediate result and avoid the final costly aggregation.

Query Processing:

- **Fast and scalable query processing:** It divides a query in to many small sub-queries and processes the independent sub-queries in parallel. It also utilizes the one-side RDMA primitives to efficiently distribute the queries.

1.20 MapGraph

This work [11] is a GPU graph processing framework based on GAS model. The major contribution is a combined load balancing strategy. By taking the size of the frontier as the metric, MapGraph can choose either a dynamic scheduling based balancing or a two-step balancing strategy.

1.21 Energy Efficient Scalable SpMV on FPGAs

This work [10] developed a sparse matrix-vector multiplication computing engine. It can make good use of the CSC compression format of the sparse data and allows stream processing of the input data. The basic idea is rather simple. Instead of performing the vector dot production for the matrix-vector computing, it take the computing as a weighted vector addition. This is particularly beneficial because of the sequential memory access. According to the experiments on ROACH, the design exhibits great performance speedup with higher bandwidth though.

1.22 Wukong

This work developed a graph database processing framework named Wukong. It utilizes the primitive graph for the query processing on distributed computing system. Particularly, it emphasizes the adoption of native RDMA for better performance. Many other different optimization techniques are integrated in this work and here is a brief list.

1.23 Dynamic Irregularities Elimination

Many parallel applications have dynamic irregular memory access and it causes considerable performance degradation when implemented on GPUs. In order to coalesce the memory access, this work [41] provide a number of strategies such as data reallocation, thread swapping and some adaptive control mechanisms. In particular, these optimization processing can be done on host processor in advance and executed with the GPU threads in a pipelined manner. Here are more details about the dynamic irregular memory access elimination.

- Data reordering: A typical dynamic irregular memory access is A[B[i]]. The data reordering logic is simply to reshape the array A to be A' such that A'[i]=A[B[i]]. The major disadvantage, however, is the same data accessed may be replicated in A' which brings additional non-trivial memory overhead.
- Job swapping: The basic process is to change the sequence of the thread ID of the GPU threads to make sure the threads in the same warp avoid

accessing data in other warps' local memory. As this may also affect the order of the output, the users must make sure the output are reordered to produce the same output with that of the original program.

- Hybrid transformation: It is a mixed data reordering and job swapping method.

- Adaptive efficiency control: In order to hide the cost of the memory coalescing, the run-time optimization process is pipelined with the GPU execution. The adaptive efficiency control is used to carefully adjust the parameters of the memory coalescing to make sure the memory coalescing will not introduce too much overhead and becomes new performance bottleneck.

Irregular memory access is one of the key factor that limits the performance of graph processing on hardware accelerators. So I am thinking if the dynamic memory coalescing strategies can be adopted in graph accelerator design.

1.24 Reorganizing Data to Minimize Non-Coalesced Memory Access

This paper [37] proves that the problem of data reordering for optimal memory coalescing is an NP-complete. Though the data reorganization remains an important way to improve the application performance running on GPUs. The essence for designing an appropriate data reorganization algorithm can be reduced to a trade-off among space, time, and complexity.

After the analysis, the authors proposed two improved optimization algorithms. The first is padding algorithm. The padding algorithm tries to avoid some unnecessary data copies made in the preliminary reordering algorithms. Basically, when two threads from the same warp access the same data elements, then there is no need to duplicate them during the re-ordering stage. A simple padding may only remove a limited amount of data and improved alternatives are discussed in this paper as well.

The second algorithm is sharing algorithm. It uses the shared memory in GPU to enlarge the scope of duplication avoidance. The basic idea is to create a copy

of all the data accessed by a thread block and put them in a consecutive chunk of memory. Then it loads these data in a consecutive manner into shared memory.

1.25 GraphChi

The authors in this work [20] present a disk-based system for computing efficiently on graphs with billions of edges. By using a well-known method to break large graphs into smaller parts, and a novel parallel sliding windows method, GraphChi is able to execute several advanced data mining, graph mining, and machine learning algorithms on very large graphs using a single PC. The most interesting parts of the graph processing framework is the graph partition and parallel sliding windows execution. They will be detailed respectively.

Graph Partition: vertices of the graph are assigned with consecutive integers and then divided into multiple intervals by the vertex index. For instance, a graph with 6 vertices can be divided into three intervals i.e.interval(0, 1), interval (2, 3) and interval (4, 5). The edges of the graph are divided into shards based on the intervals. Basically each shard (i.e.a set of edges) include all the edges whose destinations fall into the corresponding interval. Also the edges of each shard are sorted based on the indices of the source vertices. Then the framework will get to process the big graph with the granularity of the intervals following a vertex processing philosophy. Finally, note that the intervals are necessarily equally distributed in the intervals and they are chosen to balance the number of edges.

Parallel Sliding Window (PSW): the execution mechanism is based on the graph partition and it can be done in the following stages.

- Load the sub-graph from disk: For each interval processing, an associate shard that stores all the input edges will be loaded first. If the out edges is needed, they should be loaded from the rest of the shards.

- Update vertices and edges: With the update function, edge values can be changed. When both vertices of an edge update in parallel, conflict happens. To avoid this problem, the vertices that have edges with both end-points in the same interval are flagged as critical and must be updated in sequential order while the non-critical vertices can be updated safely in parallel. When

the applications that doesn't care about the race conditions, all the vertex update can be done in parallel.

- Write the updated values to disk: Make sure only the cache blocks that are modified are eventually written back to disk.

Another contribution of this work is to support evolving graph. The framework further divides the shard into different sub-shards based on the source of the edge vertices in that interval. Each sub-shard has an edge-buffer allowing new edges to be added. When the size of a shard is too large, the shard will be divided into two with similar number of edges.

GraphChi pre-processing: By computing the prefix sum over the degree sum of the graph, the vertices are divided into multiple intervals with approximately the same number of in-edges. Then the shard will be generated and stored based on the partitioned intervals. Finally, a degree file or data structure that records the in- and out-degree of each vertex is needed for more efficient processing.

Selective scheduling: Often computing converges faster on some parts of a graph than in others, and it is desirable to focus computing only where is is needed. Selective scheduling means an update can flag a neighboring vertex to be updated typically when edge value changes significantly. More details can be found in [4].

1.26 CuSha

It follows the basic idea of GraphChi that divides the big graphs into intervals and shards for both memory access locality and awareness of memory capacity. The major difference is that CuSha [17] allows multiple smaller shards to be assigned to a single GPU warp to improve the hardware utilization as well as graph processing performance.

The major challenges for graph processing on GPUs are listed here.

- Non-Coalesced Memory Access
- GPU under-utilization and Intra-warp divergence

To solve these challenges, the authors reorganize the shards with an additional mapper to be concatenated Windows such that each window may include relatively balanced number of edges for processing.

In the end, this paper also present a rough model for deciding the interval size and shard size. According to the experiments, all the setup near the optimized points show similar good performance.

1.27 Dictionary Encoding of RDF Datasets

The authors in [26] proposed a RDF dictionary encoding that employs a parallel RDF parser and a distributed dictionary data structure exploiting RDF-specific optimizations. This is part of general graph engine for multithreaded system (GEMS) and it maybe a good candidate for hardware acceleration.

1.28 Data Compression for In-memory Column Databases

This work [21] investigates the commonly used data compression schemes in memory column database. The authors believe that it is still beneficial to apply data compression for in-memory database. A clear benefit is that compression can save memory requirement and some of the compression algorithms including dictionary encoding, run-length encoding and bitmap can directly answer queries over compressed data, yielding high performance.

Some further questions are

- Will it be more efficient to have the algorithms implemented on FPGA or GPU?
- How to provide application specific compression support for the database?
- It is possible to provide different compression schemes for different columns of data in the database?

1.29 Algorithms for FPGA Implementation of SpMM

In this work [15], the authors studied the sparse matrix-matrix multiplication. Compared to previous work, higher throughput can be achieved by separation of the indices comparison and floating point arithmetic modules. As in sparse matrix multiplication, most of the computation is the indices comparison. The proposed design offers significant speedup over CPU and GPGPU when the matrices sparsity is unstructured and randomly distributed.

When the sparse matrix design is employed on FPGA for graph processing, the matrices transformed from the graph are typically unstructured and thus they will be fit perfectly for graph processing on FPGA through SpMM.

1.30 GEMS: Graph Engine for Multi-threaded System

This paper [25] presents GEMS, a software infrastructure that enables large-scale, graph database on commodity clusters. Unlike current approaches, GEMS implements and explores graph database by employing graph-based methods. This is reflected in different layers of the software stack.

- parallelism and space efficiency of graph data is explored.
- The proposed graph-based methods introduce irregular, fine-grained data access with poor spatial and temporal locality.

1.31 PHAST

This work[9] presents an algorithm to solve the non-negative single-source path (NSSP) problem particularly for low highway dimension graphs such as road networks. The paper includes a large number of research advancements on accelerating NSSP problem and some of them can actually be generalized in many graph problems. Interesting summaries and algorithms will be put in this survey.

Efficient implementations which typically rely on fast priority queues of the Dijkstra's algorithm are listed below (Note that the graph has n vertices and m edges):

- Binary heaps runs in $\mathcal{O}(m \log n)$ time.
- k-heaps
- Fibonacci heaps
- Bucket based implementation when the edges are integers within a small domain.
- Multi-level buckets
- smart queues can run in $\mathcal{O}(m + n \log C)$ time

The performance of the Dijkstra's algorithm improves if the vertices are ordered such that neighboring vertices tend to have similar IDs. The authors observed that reordering the vertices according to a depth-first search order, Dijkstra's algorithm runs faster in many cases.

Contraction Hierarchies (CH) [12] is the start point of the PHAST. It essentially reduces the NSSP problem to a traversal of shallow acyclic graph and is mainly used in road networks. The basic idea of CH is to gradually replace two connected edges with a single equal virtual edge i.e. (s, m) + (m, t) is replaced by a virtual edge (s, t) when (s, m) + (m, t) is the only shortest path from s to t. At the same time, the rank of vertex m is defined based on this. Note that rank(m) ¡ rank(s) and rank(m) ¡ rank(t).

In order to improve memory access efficiency, the graph is represented as a cache-efficient manner. Basically, the graph is stored in two different format. In the first format, the edges and the vertices are stored in separate arrays. The edge array is sorted by tail ID and this ensures the outgoing edges are accessed consecutively. The other is the vertex ID array which denotes the start position of the outgoing edges. In the other format, the storing format is about the same except that in edges are used and sorted. With the two graph representation, the level of the vertices are determined. When the vertices are processed following the level of the vertices, performance can be improved.

PHAST also supports parallel tree searching and most important is that the pre-processing can be shared as well.

The two-phase PHAST approach:

- pre-process network to compute auxiliary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space and query times)

Read the presentation together with the paper. It does help a lot, though it is still difficult to fully understand the paper.

1.32 Zedwulf

This work [16] prototyped a 32-node Zynq SoC cluster and did communication optimization for sparse-graph access on the Zynq cluster. An optimized graph-oriented global scatter technique using message passing interface (MPI) library is developed.

1.33 High-throughput and Energy-efficient Graph Processing on FPGA

Interesting statements mentioned in the paper [42]. Vertex-centric graph processing randomly accesses edges through pointers stored with vertices while edge-centric graph processing directly accesses edges from external memory in a stream fashion. For graphs with the property that edge set is much larger than the vertex set, edge-centric graph processing is often advantageous compared to vertex-centric graph processing.

This is a edge-centric graph processing framework. In this work, vertices are partitioned. Each partition has an edge list and a message list. The edge list stores all the edges whose source vertices are in the partition's vertex set. The message list stores all the messages whose destination vertices are in the partition's vertex set. Despite the memory coalescing, the design still need a permutation network to write back to main memory. With the observation that main memory consumes

a large amount of power under random access, this work developed a permutation network to make sure the write operations be aware of the row activation, which is beneficial to the overall power efficiency.

1.34 Mobile Routing Planning

There are many data ordering strategies that are beneficial to the overall shortest path calculation in this work [29]. Here is a list of them summarized in the paper.

- DFS order
- Assign priority using Contraction Hierarchy(CH) and then reorder the vertices with the priority. Some people proposed to divide the nodes into chunks where vertices with similar priorities should be put together for better memory access locality.
- The data structure is also important to the memory access, and a block based data structure is proposed in this work. In addition, the block is the basic processing granularity and dedicated replacement mechanism is used to swap blocks between on-chip memory and external memory.

1.35 Efficient Execution of Memory Access Phases Using Dataflow Specialization

The authors observed [14] that OOO processors perform much better than the in-order processor for many compute kernels including the natural phases and induced phases. These kernels commonly have abundant instruction-level parallelism and memory-level parallelism. They have much dynamic behavior in the cache hits/misses, some control flow and commonly have a small static-code footprint of 10 to 300 instructions. With this observation, the authors developed a OOO style memory access accelerators which helps to improve the performance of in-order processors and application accelerators. However, it also consumes substantial power. This work separates the memory operations and the computing operations. Particularly, the memory parallel operations are extracted and implemented on a flexible hardware fabricate i.e. MAD. The same design philosophy

can be applied in many out of order processors and hardware accelerators. It is an interesting paper needed more detailed investigation and hopefully it can be adopted in FPGA accelerator design.

1.36 SQRL: Hardware Accelerator for Collecting Software Data Structure

This work [19] proposed SQRL, a hardware accelerator that integrates with the last-level-cache and enables energy-efficient iterative computation on data structures. SQRL integrates a data-structure-specific LLC refill engine with a lightweight compute array that executes the compute kernel. The collector runs ahead of the PEs in a decoupled fashion and gathers data objects into the LLC. Since data structure traversals are structured, the collector does not require memory-disambiguation hardware and address generation logic like conventional processors; furthermore, it can exploit the parallelism implicit in the data structures.

1.37 Stochastic Optimization of Floating-Point Programs with Tunable Precision

The aggressive optimization [30] of floating point computation is an important problem in high-performance computing. Unfortunately, floating-point instruction sets have complicated semantics that often force compilers to preserve programs as written. This work present a method that treats floating point optimization as a stochastic search problem. The authors demonstrate the ability to generate reduced precision implementations of Intel's handwritten C numeric library which are up to 6 times faster than the original code and achieve end-to-end speedups over 30% on a direct numeric simulation and a ray tracer by optimizing kernels that can tolerate a loss of precision while still remaining correct. In particular, the authors present a stochastic search techniques for characterizing maximum error. The techniques comes with an asymptotic guarantee and provides strong evidence of correctness.

Bibliography

[1] V. G. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. In-memory graph databases for web-scale data. *Computer*, 48(3):24–35, Mar 2015.

[2] Dominik Charousset, Raphael Hiesgen, and Thomas C Schmidt. Caf-the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 15–28. ACM, 2014.

[3] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[4] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98. ACM, 2012.

[5] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: An efficient graph processing system on a single machine. *CoRR*, abs/1510.06916, 2015.

[6] Eric S Chung, John D Davis, and Jaewon Lee. Linqits: Big data on little clients. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 261–272. ACM, 2013.

[7] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong 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, FPGA '16, pages 105–110, New York, NY, USA, 2016. ACM.

[8] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, Washington, DC, USA, 2014. IEEE Computer Society.

[9] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[10] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 161–170. ACM, 2014.

[11] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[13] Raphael Hiesgen, Dominik Charousset, and Thomas C Schmidt. Manyfold actors: extending the c++ actor framework to heterogeneous many-core machines using opencl. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 45–56. ACM, 2015.

[14] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 118–130. ACM, 2015.

[15] Ernest Jamro, Tomasz Pabiś, Paweł Russek, and Kazimierz Wiatr. The algorithms for fpga implementation of sparse matrices multiplication. *Computing and Informatics*, 33(3):667–684, 2015.

[16] Nachiket Kapre and Pradeep Moorthy. A case for embedded fpga-based socs in energy-efficient acceleration of graph problems. *Supercomputing frontiers and innovations*, 2(3), 2015.

[17] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM.

[18] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 468–479. ACM, 2013.

[19] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. Sqrl: hardware accelerator for collecting software data structures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 475–476. ACM, 2014.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[21] Chunbin Lin, Jianguo Wang, and Yannis Papakonstantinou. Data compression for analytics over large-scale in-memory column databases. *arXiv preprint arXiv:1606.09315*, 2016.

[22] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[23] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[24] Microsoft. Trinity. https://www.microsoft.com/en-us/research/project/trinity/, 2013. [Online; accessed 10-November-2016].

[25] Alessandro Morari, Vito Giovanni Castellana, Oreste Villa, Jesse Weaver, Gregory Todd Williams, David J Haglin, Antonino Tumeo, and John Feo. Gems: Graph database engine for multithreaded systems., 2015.

[26] Alessandro Morari, Jesse Weaver, Oreste Villa, David Haglin, Antonino Tumeo, Vito Giovanni Castellana, and John Feo. High-performance, distributed dictionary encoding of rdf datasets. In *2015 IEEE International Conference on Cluster Computing*, pages 250–253. IEEE, 2015.

[27] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28. IEEE, 2014.

[28] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 166–177. IEEE, 2016.

[29] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile route planning. In *European Symposium on Algorithms*, pages 732–743. Springer, 2008.

[30] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices*, 49(6):53–64, 2014.

[31] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 317–332, 2016.

[32] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[33] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Proceedings of the 2015 Data Compression Conference*, DCC '15, pages 403–412, Washington, DC, USA, 2015. IEEE Computer Society.

[34] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[35] Luis Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *arXiv preprint arXiv:1309.1049*, 2013.

[36] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.

[37] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM.

[38] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. Q100: the architecture and design of a database processing unit. *ACM SIGPLAN Notices*, 49(4):255–268, 2014.

[39] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. The q100 database processing unit. *IEEE Micro*, 35(3):34–46, 2015.

[40] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the VLDB Endowment*, volume 6, pages 265–276. VLDB Endowment, 2013.

[41] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGPLAN Not.*, 46(3):369–380, March 2011.

[42] 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)*, pages 103–110, May 2016.