BelRed: Constructing GPGPU Graph Applications with Software Building Blocks

Shuai Che, Bradford M. Beckmann and Steven K. Reinhardt
{Shuai.Che, Brad.Beckmann, Steve.Reinhardt}@amd.com
AMD Research, Bellevue WA

***Abstract*—Graph applications are common in scientific and enterprise computing. Recent research studies used graphics processing units (GPUs) to accelerate graph workloads. These applications tend to present characteristics that are challenging for single instruction multiple data (SIMD) computation. To achieve high performance, prior work studied individual graph problems, and designed device-specific algorithms and optimizations to achieve high performance. However, programmers have to expend significant manual effort, packing data and computation to make such solutions GPU-friendly. Usually, they are too complex for regular programmers, and the resultant implementations may not be portable nor perform well across platforms.**

To address these concerns, we present a library of software building blocks, *BelRed*[1] which allows programmers to build GPGPU graph applications with ease. BelRed is based on the prior research of graph algorithms in linear algebra, and is implemented and optimized for the GPU platform. BelRed currently is built on top of the OpenCL framework. It consists of fundamental building blocks necessary for graph processing. This paper introduces the library and presents several case studies on how to leverage it for a variety of representative graph problems. We evaluate application performance on an AMD GPU and investigate optimization approaches to improve performance.

I. Introduction

Recent research efforts have advanced GPU computing into the areas of SIMD-unfriendly workloads [3], [4], [6], [13], [25], [37]. Graph applications are an emerging set of such irregular workloads. They have been applied in social network analysis, knowledge discovery, business analytics, infrastructure planning and engineering simulation. Graphs with millions of vertices and edges are common. Most prior GPGPU research took problem-specific approaches for parallelization and optimizations of individual problems. These advanced techniques usually are not intuitive for regular programmers because significant effort is needed to restructure SIMD computation and memory accesses. Ideally, programmers need an easy way to program graph applications while achieving high-performance and good portability.

This paper resolves these issues by developing and fine-tuning a set of building blocks shared by many important graph algorithms for GPUs. Inspired by the prior work, we adopt the concept and development of graph algorithm research in the linear-algebra language [2], [16], [23], and optimize them for the GPU. Though there could be alternative abstractions for solving graph problems [20], [21], [34], primitives designed in linear algebra are promising to map relatively well to SIMD architectures with optimizations. Understanding graph building blocks for GPUs and how to use them to construct diverse applications are not sufficiently covered by prior work. This paper makes several contributions:

- We present the BelRed framework and current set of supported API and building blocks (e.g., diverse sparse-matrix and vector operations). We present their GPU parallelization and optimization.
- We demonstrate several representative graph applications and their GPU implementations with BelRed.
- We show performance benefits of our applications on an AMD discrete GPU. We study performance implications of data layouts and access patterns.

II. BelRed

BelRed is a library framework consisting of diverse and optimized functions necessary for programming irregular graph applications on GPUs. The goal is to allow GPU programmers to leverage the API to construct their applications flexibly without significant knowledge of GPU architectural features. The actual API implementation is transparent to programmers. BelRed is based on sparse-matrix and vector operations in linear algebra [16], and supports diverse graph formats and data structure layouts for sparse matrices. The current API is built on top of OpenCL, but can be extended to any programming model (e.g., C++, CUDA, Python, or Matlab). It is not restricted to GPU uses and thus implementations can target any computer architecture or dedicated hardware.

A. API Functions

Table I shows a subset of sample API functions and their descriptions (see Figure 1 for a graphical representation). We use U, $\vec{u}$ and u to denote a 2-D matrix, a vector, and a scalar element respectively. The programmers can call these functions on the host. This is by no means a complete list for graph processing; BelRed is extensible to support other operations, and each function may be extended to different variants with replacement of other operators (e.g., $\times, +, min$). For example, *OuterSum* may have a variant of *OuterProduct*. Similarly, $min.+$ may have a variant of $max.+$. It may also be useful to provide user-defined functions to define specific operations on given matrices or vectors. The works [2], [23] proposed other primitives, which may be useful to include in the future.

[1]BelRed is famous road across Bellevue and Redmond, WA USA.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

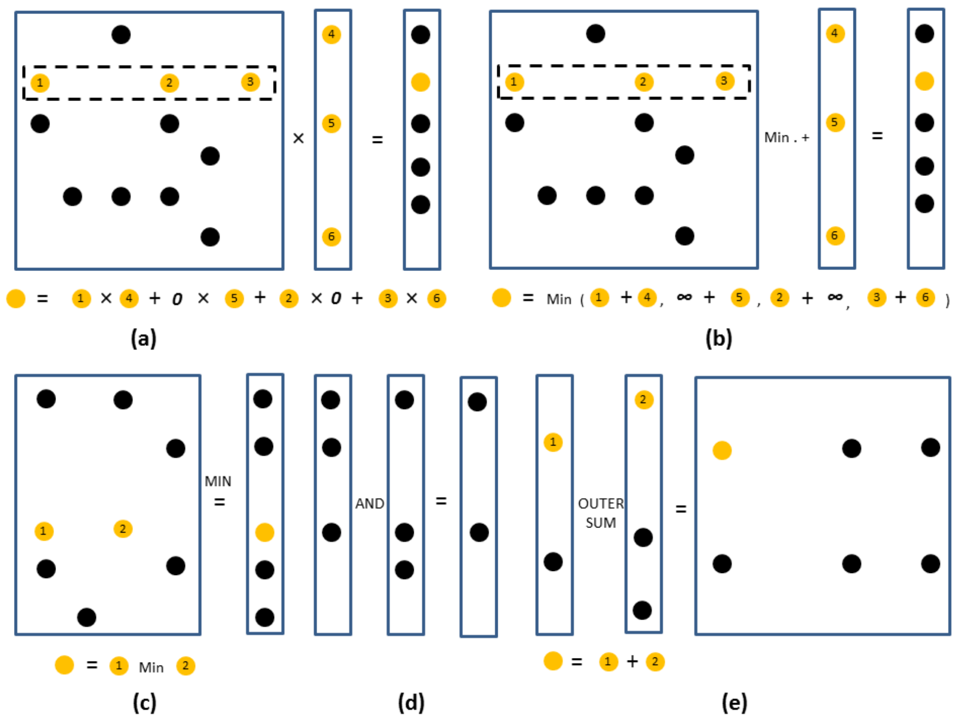

Fig. 1. Sample BelRed operations: (a) matrix-vector multiply, (b) $min.+$, (c) segmented min reduction, (d) vector-vector AND, and (e) outer sum. For each kind of pattern, other operations are supported by the BelRed API. For (b) and (c), the blank regions represent big numbers (i.e., ∞) while for (a), (d) and (e), the blank regions represent zeroes. The 2-D matrices can be stored in a sparse format.

TABLE I
Sample BelRed API functions and descriptions

API	Description
$\vec{u} = SpMV(M, \vec{v})$	sparse-matrix vector multiplication
$\vec{u} = MinDotAdd(M, \vec{v})$	the $min.+$ operation
$\vec{u} = SegReduc\_Op(M)$	segmented reduction. $Op = +, \&, min\ldots$
$U = OuterSum(\vec{v}, \vec{w})$	the outer sum operation
$\vec{u} = ElemWise\_Op(\vec{v}, \vec{w})$	element-wise operations. $Op = +, \&, min, \ldots$

TABLE II
Operations and Current Data Structure Support

BelRed Functions	Data Structure Support
$SpMV$, $min.+$ and $SegReduc\_Op$	CSR, COO and ELL
$OuterSum$ and $ElemWise\_Op$	2-D array and vector

B. Data Structure Layout and Graph Inputs

The BelRed framework supports diverse data structures internally to store graphs for different usages. Different algorithms and traversal patterns may prefer different memory layouts for different graphs. These include the compressed sparse row (CSR), coordinates list (COO) and ELLPACK (ELL) formats to store sparse graph structures (e.g., in Bellman-Ford, Maximal Independent Set, Coloring and PageRank). Table II lists the BelRed functions and the associated data structures they support currently. Future BelRed development will extend the framework to richer data structures (e.g. diagonal and hybrid formats).

BelRed includes support for widely used graph formats: DIMACS Challenge [5], METIS [26] and Matrix Market [22] formats. The library contains utility routines to parse graph inputs stored in these formats. The inputs used in this study were chosen from DIMACS Implementation Challenges [5], University of Florida Sparse Matrix Collection [36]. GT-Graph [28] was used to generate random graphs.

III. Graph Building Blocks

In this section, we present some key operations and functions in the BelRed API. Each function is implemented and optimized for the GPU and can be called in a GPU program. Figure 1 shows the graphical representation of sample operations. 2-D matrices in the figure can be represented in compact sparse formats.

A. SpMV

Sparse matrix-vector product (*SpMV*) is a computational kernel that is critical to many scientific and engineering applications. Figure 1 (a) is a graphical view of SpMV. SpMV-type operation is a natural fit for graph applications in which vertices have data exchanges with their neighbors. For example, SpMV can be used to expand the neighbor list (e.g., in breadth-first search [16]). Also, SpMV can be modified with other operators to meet different purposes. For instance *SpVisit* can be defined by substituting $\times$ and $+$ in the inner loop with $|$. There can be multiple GPU implementations for SpMV. The outer loop of SpMV can be unfolded such that each GPU thread is responsible for processing one row. An alternative is that each wavefront is responsible for one row. In this case, in-wavefront parallel reduction is needed to calculate the sum. Prior studies [1], [32] demonstrate approaches to optimize SpMV operations for GPUs. It is one of the useful building blocks for graph processing.

B. *min.+*

$Mmin. + \vec{w}$ is a m-wide vector whose ith element is $min(M(i,j) + \vec{w}(j) : 1 \leq j \leq N)$. The $min.$+ operation is another important programming primitive for graph processing [16]. Comparison of vertex value is a common operation in shortest-path and graph-partitioning algorithms. For each row in the adjacency matrix and a dense vector, $min.$+ sums each pair of two elements in the row and vector, and then performs a reduction with a min operation across all pairs. Figure 1(b) is a graphical representation of the $min.$+ operation. We have three versions of kernel implementations for CSR, COO and ELL. For instance, for ELL each thread is responsible for one row. The ELL matrix is stored in a column-major order with additional padding. This helps improve memory coalescing when multiple threads in a wavefront access data elements in cache simultaneously. Also, there is no divergent branch but with additional redundant computation.

C. *Element-wise Operations*

Element-wise matrix-matrix operations take two matrices and generate a result matrix with an operator (e.g., $\times, +, min$) applied on each pair of elements from two matrices. To implement element-wise GPU operations, each thread is assigned to compute each position in parallel. Element-wise vector-vector operations are also very common in many graph algorithms. For instance, an algorithm may decide which vertices are active or inactive. Figure 1(d) is graphical view of an *AND* operation which will be useful for this case. GPU implementations of these operations seem to be straightforward with threads processing different regions of an array or vector. But the performance will be sub-optimal if inappropriate parameters are chosen, such as OpenCL workgroup size and the amount of work (data chunk in bytes) assigned for each workitem. BelRed relieves programmers from dealing with device-specific tuning with a high-level API.

D. *Segmented Reduction*

Segmented reduction calculates a vector with each element calculated by reducing the elements in a segment of the vector (e.g., each segment can represent a row in a sparse matrix). One useful operation in many graph partitioning problems is to compare the values of all the vertices in a row and calculate the maximum or minimum. For example, given a 2-D adjacency matrix $SegReduc\_Min$ produces a vector with each element being the minimum of the elements in each row (see Figure 1(c)).

E. *Outer Sum*

The *outer sum* operation calculates an $m \times n$ matrix using two 1-D vectors as inputs, with a size of m and n respectively. It is useful for calculating all-to-all relationship. Figure 1(e) shows the *outer-sum* operation in which for a 2-D matrix M and two vectors $\vec{v}$ and $\vec{w}$, the element $M(i,j)$ is the sum of $\vec{v}(i)$ and $\vec{w}(j)$ where $i \in [1,m]$ and $j \in [1,n]$. For a GPU implementation, an $m \times n$ 2-D *ND-Range* is configured to launch an OpenCL kernel. All the points in the computation domain mapped to 2-D indices can be processed concurrently. The Floyd-Warshall algorithm, which computes all-pairs shortest-paths (APSP) for a graph, includes an operation similar to *outer sum*.

F. *Other Building Blocks*

Prior studies investigated other basic building blocks for irregular applications. Our work is complementary to their efforts. One famous example is a set of scan primitives developed by Sengupta [30]. They implemented the classic scan and segmented-scan operations for GPUs. Merrill et al. [25] developed a highly optimized Breadth-First algorithm with prefix-sum. Bolt C++ [18] and Thrust [19] implemented a GPU API similar to the C++ standard template library (STL). Our work is designed uniquely for graph processing.

IV. Graph Algorithms with BelRed

In this section, we present several case studies to show the usage of BelRed to program diverse graph applications. We choose these applications because they are critical routines for many larger graph applications and it is easy to demonstrate the use of the API.

A. *PageRank*

PageRank (PRK) is an algorithm to calculate probability distributions representing the likelihood that a person randomly clicking on links arrives at any particular page. In PageRank, each GPU thread can expand the neighbor list of a vertex and atomically adds the associated PageRank amount $\frac{PgRankVal}{num\_outgoing\_edges}$ to each neighbor in the neighbor list [6]. With the BelRed API, this step can be substituted with a SpMV operation on the GPU. A transposition is first performed so that the ith row of the new matrix represents the information for all the vertices which have an outgoing edge to the ith vertex. Then the algorithm goes into the main loop, launching SpMV, which is followed by an *update* kernel to calculate PageRanks. The *update* kernel is written by the user and finalizes the *PageRank* value for each vertex at the end of an iteration. This follows the equation $PageRank[tid] = (1-d)/num\_vertices + d * PageRank\prime[tid]$, where d is a damping factor (0.85 in our study) and *tid* is the GPU thread id.

B. *Bellman-Ford*

The SSSP algorithm is a common subroutine in various graph applications. Given a user-specified source vertex, the algorithm searches the shortest path between the source vertex and all the other vertices in the graph. In this study, we use the Bellman-Ford algorithm [3], [11] to solve the single-source shortest path problem. The core of the algorithm is performing the $min.$+ operation on a matrix M and a vector

$\vec{w}$ on the GPU. The vector $\vec{w}$ stores the shortest distances of all the vertices to the source. The source vertex i in $\vec{w}$ is initialized to 0 (i.e., zero distance to itself), while all the other vertices are initialized to a ∞ value (i.e., a big integer). $\vec{w}' = Mmin. + \vec{w}$ produces the shortest distances for all the vertices directly connected to the source at the end of the first iteration. Similarly, $\vec{w}'' = Mmin. + \vec{w}'$ produces the shortest distances after traversing two layers of vertices from the source. The same process repeats until convergence. A *check_convergence* GPU kernel following *min.+* in the main loop is used to determine program termination.

C. Coloring

Graph coloring (CLR) partitions the vertices of a graph such that no two adjacent vertices share the same color. We use an algorithm similar to that used in prior work [15]. In the initialization step, each vertex is labeled with a random integer value. Each iteration labels a set of vertices with one color. For each vertex, a GPU thread compares its vertex value with that of its neighbors. If the value of a given vertex happens to be the largest (or smallest) among its neighbors, it labels itself with the color of the current iteration. The algorithm converges when all vertices are colored. In fact, the step of determining whether a vertex is a local maximum or minimum can leverage the BelRed API $SegReduc\_Min$ operation spanned across rows.

D. Maximal Independent Set

An independent set are vertices in a graph in which none are adjacent. In an Maximal Independent Set (MIS), it is not possible to add another vertex to the set without violating that property. In this study, we use the algorithm described by Shah et al. [24] to demonstrate the use of the library. In our implementation, the program first selects a subset of graph vertices, $select$, randomly as an initial set. It conducts SpMV (or SpVisit) with a graph M and the vector $select$ as inputs. The algorithm marks the touched neighbors as "visited" vertices, and subsequently determines whether the expanded neighbors and selected vertices overlap. This can be achieved through an *AND* operation (i.e., $select\&SpMV(M, select)$). For the result, it evaluates the overlapped vertices and keeps only those with higher degrees, $final\_select$, which is implemented with a specific kernel. Then we add them to the set mis. This step is achieved through an *OR* operation (i.e., $mis|final\_select$). All neighbors of $final\_select$ are discarded; the following iterations process the remaining vertices, until all the vertices are evaluated.

E. Floyd-Warshall

Floyd-Warshall (FW) solves the all-pairs shortest paths (APSP) problem. This problem can be solved in a manner of dynamic programming in 2-D matrix operations [13]. In our implementation, a 2-D distance array is used to keep track of the shortest distances from all possible sources to all possible destinations. Floyd-Warshall can be broken into two major BelRed operations. The first is an *outer sum* on the GPU calculating an $n \times n$ matrix with two n-wide vectors as inputs. The second is a parallel element-wise min operation on the GPU applied on each pair of elements from two 2-D matrices $M1$ and $M2$. Each element in the result matrix is be calculated with $min(M1(i,j), M2(i,j))$, ($i, j \in [1, n]$).

V. Experimental Setup

In this paper, the experimental results are measured on real hardware using an AMD Radeon HD 7950 (Tahiti) discrete GPU. The AMD Radeon HD 7950 features 28 GCN CUs with 1792 processing elements running at 800 MHz with 3 GB of device memory. We compare the results to those obtained from four CPU cores on an AMD A8-5500 with a 3.2-GHz clock rate and 2 MB L2 cache. We use AMD APP SDK 2.8 with OpenCL 2.1 support. AMD APP Profiler v2.5 is used to collect profiling results. In addition, this study is restricted to cases when the working sets of applications do not exceed the capacity of the GPU device memory. We leave graph partitioning for multi-node processing for future work.

VI. Results

A. Performance Improvement

We calculate performance speedups by running applications on a AMD Radeon HD 7950 discrete GPU, and comparing them to running applications on four CPU cores of an AMD A8-5500. We focus on the main computation part, and exclude I/O and graph parsing. For the AMD Radeon HD 7950, we include the PCI-E data-transfer overhead when calculating speedups. Figure 2 shows the speedups for all the applications using different inputs. The arithmetic mean of speedups for all the program-input pairs is approximately $4\times$. Application performance is also input-dependent (i.e., graph structures). Maximal Independent Set achieves only a 10% performance improvement due to combined factors of in-loop CPU computation and kernel-call overhead. It also has opportunity for additional GPU offloads. Future work will compare BelRed performance to the hand-tuned application implementations.

In addition programming of these applications become easier with BelRed library functions. The benefit mainly comes from the fact that programmers are relieved from developing GPU kernels optimized for a particular device (40–200 lines of code for our kernels for different applications).

1) Kernel Launching: Our BelRed API encapsulates common functionality into well-defined functions for reuse, productivity and maintainability. However, the cost of maintaining modularity is the possibly resultant overhead of GPU kernel call invocation and extra device-memory accesses. Therefore, an application implemented with BelRed may launch more kernel calls than a version otherwise implemented with cross-function optimizations and thus fewer kernels. For instance, the two BelRed kernels in Floyd-Warshall actually can be merged into a big kernel instead of using the BelRed API. This leads to about $2\times$ slow-down of the BelRed version. Thus, GPU vendors may focus on reducing the cost associated with kernel launching. Recently AMD proposed Heterogeneous System Architecture (HSA) [14], supporting low-overhead

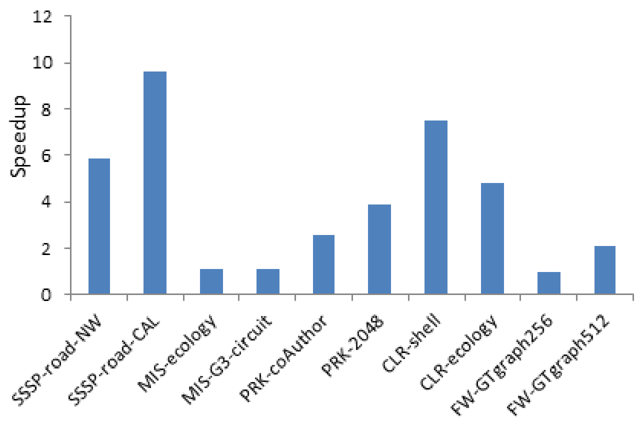

Fig. 2. The performance speedups of running graph applications on the GPU compared to the CPU

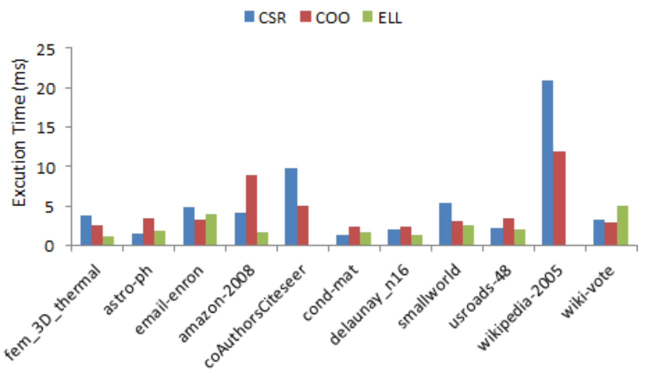

Fig. 3. Performance of the SpMV operation over different inputs and graph formats

user-space queuing on the GPU side; HSA may help address this concern, benefiting library designers. In addition, because of kernel division, extra global memory write and read may be needed between kernels to pass the intermediate results from one kernel to another.

B. Data Layouts and Accesses

We consider three popular data layouts for sparse-graph storage– COO, CSR, and ELL. Each structure has their unique advantages when solving different problems. COO and CSR are more general and flexible formats than ELL. Compared to ELL, they also require less memory capacity to store graphs and sparse matrices. On the other hand, for GPU processing, the ELL format (similar to the diagonal format) generally is more efficient, especially for structured or semi-structured graphs. For unstructured graphs, sometimes COO or CSR is preferable given the limited GPU memory capacity. Discussions on different data structures and their implications to GPU performance can be found in prior research [1], [32].

We develop multiple versions of GPU kernels for SpMV, $min.+$ and other operations to process graphs stored in different data structures. For instance, we apply some techniques from prior work [1], [32] to improve performance of sparse-matrix operations. Also, for $min.+$ and segmented reduction, we optimize codes to take advantage of the per-SIMD scratchpad memory (local data share in AMD GPUs) to perform parallel addition and min operations. We also organize the data layout of ELL into column-major layout with padding to remove divergence. This organization allows better memory coalescing without load imbalance but at the cost of redundant computation. We use the SpMV operation as an example for result demonstration. Figure 3 reports execution times in CSR, COO, and ELL over a variety of graph inputs. As shown in the figure, performance depends on different layout-input pairs.

Though ELL is good for many graph inputs, for *coAuthor* and *Wikipedia*, the points for the ELL format are missing; the available OpenCL buffer is not big enough to hold the working sets for these two problems, because ELL requires much padding capacity for unstructured graphs.

VII. Related Work

Combinatorial BLAS [2] is graph library offering a powerful set of linear algebra primitives for graph analytics. Graph BLAS [23] is a recent effort of defining standard linear-algebra building blocks for graph processing. The area of research on graph algorithms in linear algebra is documented in the work by Kepner and Gilbert [16]. Our work is unique in that we target a practical implementation for the GPU architecture also with several workloads as case studies.

Prior benchmarking efforts (e.g. the Rodinia [7], Parboil [33], and SHOC [9] frameworks) included and evaluated only a few graph or tree-based algorithms. Burtscher et al. [3] performed a quantitative study of irregular programs on GPUs. Pannotia [6] is a recent work on developing and characterizing graph algorithms specifically on the GPU platform. Other works studied how to accelerate graph algorithms efficiently on GPUs. Harish et al. [13] parallelized several graph algorithms on the GPU. Merrill et al. [25] proposed an optimized breadth-first search implementation with *prefixsum* on both single and multiple GPU nodes. Burtscher et al. [4] presented an efficient CUDA implementation of a tree-based *Barnes-Hut* n-body algorithm. Other works studied GPU acceleration for connected component labeling [29], minimum-spanning tree [37], B+tree searches [8] and so on. All these works are problem-specific solutions. In contrast, we focus on programming abstractions and GPU library construction.

Several researchers built libraries of parallel graph algorithms for CPUs. These include the Parallel Boost Graph Library [35], the SNAP library [31], and the Multi-threaded Graph Library [27]. Other works studied graph algorithms on big machine clusters and followed a “think-like-a-vertex model”. For example, Pregel [21] is a system to process large-scale graphs. Giraph [34] is an open-source implementation of Pregel. The GraphLab [20] framework is a MapReduce-like API and framework designed for data mining with optimizations for graph algorithms. GraphChi [17] is a disk-based system for processing large graphs by breaking graphs into small parts and using a parallel-sliding window method. Galois [12] is system which can be used to build graph applications. STINGER [10] is a framework designed for high-performance streaming graph analysis.

VIII. Conclusions and Future Work

Graph applications are a set of emerging workloads used in data analytics, social networks and web analysis. It is important to understand and improve their performance on GPUs and other manycore architectures. Programming irregular graph algorithms tends to be a challenge for GPU programmers. This paper presents a library, BelRed including GPU implementations of common linear-algebra building blocks for graph processing. We demonstrate that diverse graph algorithms can be accelerated on the GPU built with BelRed. In addition, applications maintain good code portability across platforms. One benefit is that programmers are relieved from writing error-prone, device-specific GPU kernels. Instead, programmers can focus on high-level algorithm design.

Future directions for BelRed include: adding support for more API functions; evaluating these building blocks for more graph applications; supporting more graph formats and data structures; optimizing, tuning and characterizing BelRed (e.g., SIMD utilization and memory efficiency) on various platforms including APUs, GPUs and other types of architectures; studying the extension of this framework to multiple machine nodes; and understanding the cost of programming effort vs. performance benefit.

Acknowledgements

We would like to thank the reviewers for their constructive comments and suggestions.

References

[1] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec 2008.

[2] A. Buluc and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4), November 2011.

[3] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization*, Nov 2012.

[4] M. Burtscher and K. Pingali. An efficient cuda implementation of the tree-based Barnes Hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Morgan Kaufmann, 2011.

[5] The DIMACS Implementation Challenge. Web resource. http://dimacs.rutgers.edu/Challenges/.

[6] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph algorithms. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sept 2013.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[8] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in B+ tree searches on an APU. In *SC Companion*, pages 240–247, 2012.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable HeterOgeneous computing (SHOC) benchmark suite. In *Proceedings of Third Workshop on General-Purpose Computation on Graphics Processing Units*, Mar 2010.

[10] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *Proceedings of IEEE High Performance Extreme Computing Conference*, Sept 2012.

[11] J. T. Fineman and E. Robinson. *Graph Algorithms in the Language of Linear Algebra*, chapter Fundamental Graph Algorithms. Society for Industrial and Applied Mathematics, 2011.

[12] GALOIS. Resource. http://iss.ices.utexas.edu/?p=projects/galois.

[13] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of 2007 International Conference on High Performance Computing*, Dec 2007.

[14] Heterogeneous System Architecture (HSA). Web resource. http://hsafoundation.com/.

[15] J. Cohen and P. Castonguay. Efficient graph matching and coloring on the gpu. http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0332-GTC2012-Graph-Coloring-GPU.pdf.

[16] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, January 2011.

[17] A. Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, Oct 2012.

[18] Bolt C++ Template Library. Advanced Micro Devices. https://github.com/HSA-Libraries/Bolt.

[19] The Thrust library. Web resource. http://code.google.com/p/thrust/.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[21] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010.

[22] Matrix Market Format. Web resouce. http://math.nist.gov/MatrixMarket/formats.html.

[23] T. Mattson, D. A. Bader, J. W. Berry, A. Bulu, Dongarra J, C. Faloutsos, J. Feo, J. R. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. E. Leiserson, A. Lumsdaine, D. A. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *Proceedings of IEEE High Performance Extreme Computing Conference*, Sept 2013.

[24] Maximal Independent Set. Presentation slides. http://acts.nersc.gov/events/para06/Shah.pdf.

[25] D. G. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2012.

[26] METIS File Format. Web resource. http://people.sc.fsu.edu/~jburkardt/data/metis_graph/metis_graph.html.

[27] MultiThreaded Graph Library. Web resource. https://software.sandia.gov/trac/mtgl.

[28] GTGraph: A Suite of Synthetic Random Graph Generators. Web resource. http://www.cse.psu.edu/~madduri/software/GTgraph/index.html.

[29] V. M. A. Oliveira and R. A. Lotufo. A study on connected components labeling algorithms using GPUs. In *Proceedings of the 23rd SIBGRAPI Conference on Graphics, Patterns and Images*, Aug 2010.

[30] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware*, Aug 2007.

[31] SNAP: Small-world Network Analysis and Partitioning. Web resource. http://snap-graph.sourceforge.net/.

[32] B. Su and K. Keutzer. clSpMV: a cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the international conference on Supercomputing*, Jan 2012.

[33] Parboil Benchmark suite. Web resource. http://impact.crhc.illinois.edu/parboil.php.

[34] The Apache Giraph. Web resource. https://giraph.apache.org/.

[35] The Parallel Boost Graph Library. Web resource. http://osl.iu.edu/research/pbgl/.

[36] The University of Florida Sparse Matrix Collection. Web resource. http://www.cise.ufl.edu/research/sparse/matrices/.

[37] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics*, July 2009.