978-1-5090-3525-0/16/$31.00 ©2016 IEEE

Embedded domain specific language for GPU-accelerated graph operations with automatic transformation and fusion

Stephen T. Kozacik, Aaron L. Paolini, Paul Fox, James L. Bonnett, Eric Kelmelis

EM Photonics
Newark, DE

Dennis W. Prather
Department of Electrical and Computer Engineering
University of Delaware
Newark, DE

***Abstract*—Achieving high performance in large-scale graph analytics using conventional processing systems generally involves excessive runtimes. In response to this, we present a language for graph analytics that allows users to write in a familiar, vertex-centric API while leveraging the computational power of many-core accelerators. Our prototype toolchain automatically employs abstract sparse linear algebra (ASLA) operations using custom semi-rings in order to maximize performance.**

Our implementation presents an API similar to Pregel without requiring the user to write explicit combiners. We map the user's algorithm to an efficient, fused ASLA-based approach at compile time with no runtime overhead. Using this technique, we have implemented several algorithms, including single-source shortest path and PageRank. With a GPU-accelerated linear algebra backend, we can achieve better than 500% speedup over a multi-core ASLA library, using real and synthetic datasets.

Keywords—Graph Analytics, GPU Acceleration, Linear Algebra

I. INTRODUCTION

Graph analytics is a key component in identifying emerging trends and threats in many real-world applications in that it allows the analysis of interconnected systems such as social networks, commercial relationships, computer networks, logistics, and more. Large-scale graph analytics frameworks, such as Apache Giraph and Pregel, provide a convenient means of developing algorithms for distributed analytics on extremely large datasets. The "think like a vertex" programming model used by these frameworks allows algorithm developers to base their software on an intuitive understanding of relationships in the graph. These frameworks allow the user to process large graphs by using many processors and machines; however, to truly scale to exascale systems and achieve low energy usage and run time, graph algorithms need to leverage the computational power of many-core accelerators such as Graphics Processing Units (GPUs). One area that has been able to successfully leverage these systems is sparse linear algebra, which is used in many areas of scientific and high-performance computing. Because of this, a new graph-computing paradigm has emerged for performing graph calculations as sparse matrix calculations using the abstract algebra concept of semirings. This paradigm, known as GraphBLAS, is very powerful, but requires a new way of thinking from analysts.

In this paper we present an embedded domain-specific language in C++ for creating vertex-centric graph algorithms. This language, while vertex-centric, is transformed automatically at compile time into whole-graph operations. These graph operations are then further optimized for many-core acceleration by transforming them into abstract sparse linear algebra operations.

These resultant linear algebra operations can then be run on a GPU, leveraging the existing body of GPU-accelerated sparse linear algebra algorithms. We demonstrate this technique on several algorithms, using both real-world and simulated datasets, and show the benefits of both GPU acceleration and conversion of vertex-centric operations to abstract sparse linear algebra.

II. METHODOLOGY

A. Vertex-Centric EDSL

We have created an embedded domain-specific language (EDSL) for implementing many-core accelerated graph algorithms. With this EDSL, a user can create a graph algorithm from within C++ that is limited only to using those operations supported within the EDSL. This technique offers us the benefit of full insight into and control of any data accesses, allowing us to implement and transform operations as we see fit. This EDSL is based on Google's Pregel [1] system for graph processing.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

As in Pregel, a user defines a vertex that sends and receives messages to define their computation. A user-defined vertex can currently contain one or more properties, as well as methods. In the case of PageRank, a vertex contains a single method to handle incoming messages. Within the user-defined methods, the user can define their operation in terms of graph-specific operations (reduce, send messages, etc.) to form an expression. It should be noted that the internals of this method make use of operations of our own implementation provided by our EDSL for graph computations. These operations are heavily templated, and represent a use of *expression templates* [2], which encode operations in their types. For example, to add an integer and single-precision floating point number in a C++ expression template framework, the resultant type would be something like *Plus<int, float>*, where *Plus* is a class.

```
template <template <typename> class Graph>
struct PageRankVertex {
    using MessageType = double;
    using GraphType = Graph<PageRankVertex>;
    PROPERTIES((double, pageRank))

    auto HandleMessages( Messages<MessageType> msgs) {
        const double r = .85;
        auto update = Reduce(r, (1-r)*msgs);
        auto prop = SetProperty(pageRank, update);
        auto toSend = prop/OutDegree();
        return SendMessages(toSend);
    }
    OutDegree_ OutDegree() {
        return OutDegree_{};
    }
};
```

Figure 1 - Example PageRank vertex type

```
GraphType g{AdjacencyMatrixCSC};
const int nIterations = 10;

for (int i = 0; i < nIterations; ++i)
{
    // Perform Iteration
    const auto& result = g.Iterate();

    // Fetch results if desired...
}
```

Figure 2 - PageRank example (calling code)

Given a vertex type and input data, the user can instantiate a graph. Once the graph has been constructed, the user then writes their code that performs operations on the graph. If the user wants to access a vertex property before, during, or after iterating, we provide a macro for the user to retrieve this property from the graph. This information is received as a vector of the property for each vertex in the graph and remains resident on the GPU, where it can be manipulated, copied to the host, or set with data from the host.

```
auto& props = GETPROPERTY(g, VertexType::pageRank);
auto mostImportantVertex = MaxElementIndex(props);
```

Figure 3 – Retrieval of property values from graph

When constructing the graph, we automatically invoke a *BuildIteration* function. This function essentially extracts the expression template from the user-defined method, which encodes the intent of the function, and converts the method into a runnable form. Below is the expression template associated with the previously-shown *HandleMessages* function for PageRank.

```
ops::BinaryOp<operators::Divides<
    ops::SetProperty_<
        ops::Property<double, 0ul>,
        ops::BinaryOp<operators::Adds<
            double,
            ops::Reduction<
                ops::BinaryOp<operators::Times<
                    double,
                    ops::Messages<double>
                > >,
                ops::Adds<double, double>
            >
        > >
    >,
    ops::OutDegree_
> >
```

Figure 4 - Expression template derived from PageRankVertex's HandleMessages function

The outermost type is *SendMessages*, which represents the terminating (returned) statement in the *HandleMessages* function of *PageRankVertex*. Internally, the *BuildIteration* function will recursively break down the expression until it is completely decomposed into its fundamental expressions, which is accomplished through the use of function template overloads for each of the fundamental types. Each of these overloads returns a lambda function that invokes the underlying implementation of the function. Expressions that are composed of one or more sub-expressions will be built from their constituent lambda functions that result from independent conversions. In this way, an aggregate function will be built.

B. GPU Accelerating GraphBLAS Operations

We created a GPU-accelerated library for both GraphBLAS-style operations on custom semirings and elementwise operations. We implemented scalar-vector multiplication and vector addition using the *scal*, and *axpy* families of BLAS calls of the cuBLAS library, respectively.

For matrix-vector products we used NVIDIA's CUB library, which has a fast sparse matrix-vector product, which employs a merge-based parallel decomposition [3]. To leverage this feature so as to allow matrix-vector multiplication on custom semirings, we created a wrapper type containing only the original type (e.g. double) that is templated on the addition (+) and multiplication (*) operators. We then used the CUB matrix-vector product with the original matrix and vector types being interpreted as this new type. This results in our semiring addition and multiplication operations being used instead of the standard addition and multiplication operators. We also used CUB's *SegmentedReduce* functions for performing operations such as reductions on messages sent from neighboring vertices.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

An example of constructing an optimized Single-Source Shortest Path algorithm using this library is shown below.

```
// The user can read in the matrix from an mtx file
auto adjMat = ReadCsrFromMatrixMarket<double>(
    matrixPath);

auto inf = std::numeric_limits<double>::infinity();

CudaCsrMatrix<double> gpuAdjMat(adjMat);
CudaVector<double> minDistD(adjMat.m(), inf);
minDistD[5] = 0.0;

auto addition = Min<double, double>{};
auto multiplication = Adds<double, double>{};
minDistD = Transform(
    MvProd(trans(gpuAdjMat), minDistD, addition,
        multiplication), minDistD, addition);
```

Figure 5 - Accelerated Single Source Shortest Path Algorithm using our GraphBLAS backend

C. Fusing Graph Operations to Linear Algebra Operations

By recursively breaking down the type constructed using our EDSL described in the above section, we can directly translate vertex-centric operations to matrix and vector operations, where the graph is represented as an adjacency matrix and vertex properties are represented for the whole graph as a vector. In the case of many Pregel functions, a different message is sent to each vertex, resulting in generation of a temporary matrix to hold the messages. Often, however, the receiving vertex is only interested in a reduced form of the messages, such as the minimum in the case of single-source shortest path. In Pregel, this is exploited by having the user define explicit combiners. Because of the nature of our language, we have found that we are able to perform this optimization automatically. Additionally, in performing this operation we are often able to convert the operation into a linear algebra operation on a semiring.

Often, a single graph operation does not directly translate to a linear algebra operation, and so we must fuse several operations to phrase the problem in terms of abstract sparse linear algebra. The act of fusion can result in operations that not only parallelize better, but can also represent an overall reduction in operation count and temporary data. The first step in this process is to transform the EDSL operations into element-wise operations on the graph structure. An example transformation result for our PageRank test is shown in Figure 6.

After this transformation, we iterate through the graph, looking for opportunities to rearrange operations in order to facilitate fusion. We then iterate through the rearranged version, looking for fusion candidates. An example of a fusion candidate is an element-wise product followed by a reduction.
In this case, the operation can be transformed into a matrix-vector product on a semiring where the element-wise product operation becomes the multiplication operation and the reduction operation becomes the addition operation. The fusion of the type in Figure 6 can be seen in Figure 7.

```
ops::BinaryOp<operators::Divides<
    ops::SetProperty_<
    ops::Property<double, 0ul>,
        ops::BinaryOp<operators::Adds<
            double,
            ops::BinaryOp<operators::Times<
                double,
                ops::MvProduct<
                    ops::Adds<double, double>,
                    ops::Times<double, double>,
                    ops::AdjMatTransposed<double>,
                    ops::MessageMember<double>
                >
            > >
        > >
    >,
    ops::OutDegree_
> >
```

Figure 6 - PageRank Iteration type after transformation

```
ops::BinaryOp<operators::Divides<
    ops::SetProperty_<
        ops::Property<double, 0ul>,
        ops::BinaryOp<operators::Adds<
            double,
            ops::BinaryOp<operators::Times<
                double,
                ops::Reduction<
                    ops::ElementWiseProd<
                        ops::AdjMatTransposed<double>,
                        ops::MessageMember<double>
                    >,
                    ops::Adds<
                        double,
                        double>
                    >
                >
            > >
        >
    >,
    ops::OutDegree_
> >
```

Figure 7 - Fused PageRank iteration type

III. EVALUATION

A. Experimental Setup

To test the efficacy of both our frontend language and our backend library, we created implementations of three common graph algorithms. We tested these algorithms on three real-world datasets as well as on synthetic data of 9 different sizes.

B. Experimental Platform

We benchmarked our software on the following hardware:

- CPU: Intel Core i7-5930K CPU @ 3.50GHz (6 Cores, 12 Threads)
- RAM: 16 GB DDR4 2133 MHz
- GPU: Nvidia GeForce GTX Titan X with CUDA 7.5
- Software: Ubuntu 14.04, g++ 5.3, mpich2 3.0.4-6, CombBLAS 1.5.0

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

C. Algorithms

The three algorithms that we created were: Page Rank, Single-Source Shortest Path, and Single-Source Widest Path. Page Rank, which involves finding the most important node in a graph, was chosen due to its utility in several fields and its ease of implementation in several different graph programming paradigms. Next, we chose Single-Source Shortest Path. In addition to its simplicity, this algorithm was chosen because it lends itself easily to direct implementation with abstract sparse linear algebra. Single-Source Widest Path is very similar to Single-Source Shortest Path, but is special in that, when implemented in terms of abstract sparse linear algebra, neither of its operations are standard arithmetic operations.

D. Reference Implementation

CombBLAS was chosen as the reference as it was shown by Satish et al. [4] to have the highest performance among graph frameworks for the chosen algorithms. To make the fairest possible comparison, we chose high-end CPU hardware and used MPI to parallelize the CPU operations. Nine MPI processes were used because CombBLAS requires a square number of processes.

Table 1 - Datasets used for benchmarking

Data Set	Vertices	Edges
Wikipedia	3,566,907	45,030,389
LiveJournal	4,847,571	68,993,773
cage15	5,154,859	99,199,551

E. Real World Datasets

We chose the LiveJournal and Wikipedia datasets as these were used by Satish et al. and represented real-world networks that were realistic applications for our algorithms (e.g. PageRank). The cage15 dataset was chosen as it is a large, directed, weighted graph which we believed would yield interesting results for both Single-Source Shortest Path and Single-Source Widest Path. For benchmarking purposes, we transformed this into a binary adjacency matrix for use with our PageRank implementation.

F. Synthetic Datasets

To analyze how our performance scales as the number of edges increases, we generated synthetic data using an approach similar to Satish et al. [4]. We used the Recursive-Matrix (R-MAT) model [5] with the "a" parameter set to .57 and the "b" and "c" parameters set to .19. Using this model, we generated directed graphs with edge counts from 2^{20} through 2^{28}, where there were 16 edges per vertex. To generate these datasets we used the PaRMAT multithreaded RMAT graph generator [6] with generation of duplicate edges disabled.

G. Results

Table 2 - PageRank Runtimes

	Data Set		
Platform	**Wikipedia**	**LiveJournal**	**cage15**
CPU	0.160325	0.262824	0.129189
GPU	0.0837797	0.12686	---
GPU w/Fusion	0.0266712	0.0264352	0.0100554

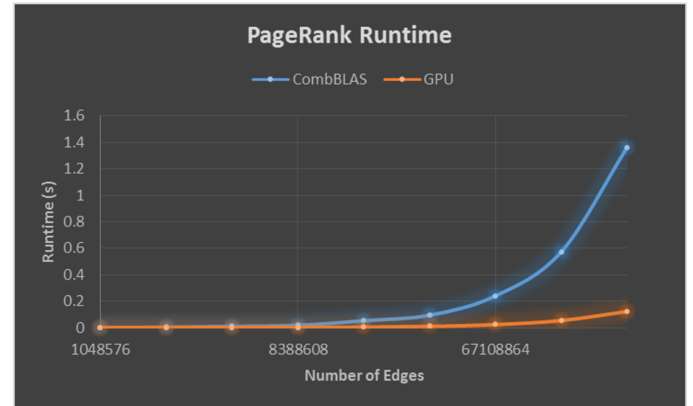

Figure 8 – PageRank runtimes on synthetic data generated using the R-MAT model

Figure 8 compares the runtime of our PageRank algorithm versus the CombBLAS runtime for each graph size.

Table 3 – Single-Source Shortest Path Runtimes

	DataSet		
Platform	**Wikipedia**	**LiveJournal**	**cage15**
CPU	0.201485	0.279654	0.15647
GPU	0.0701992	0.0916699	---
GPU w/Fusion	0.0244974	0.0234983	0.00801

A similar plot for the Single Source Shortest Path algorithm is shown in Figure 9. Note that the x axis uses a logarithmic scale so we observe linear scaling with the number of edges. Using the same benchmark setup as previously discussed, we are able to outperform CombBLAS by nearly 1000% in many cases.

Table 4 – Single-Source Widest Path Runtimes

	DataSet		
Platform	**Wikipedia**	**LiveJournal**	**cage15**
CPU	0.220781	0.332494	0.135978
GPU	0.0882223	---	---
GPU w/Fusion	0.0271287	0.02976	0.0138946

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

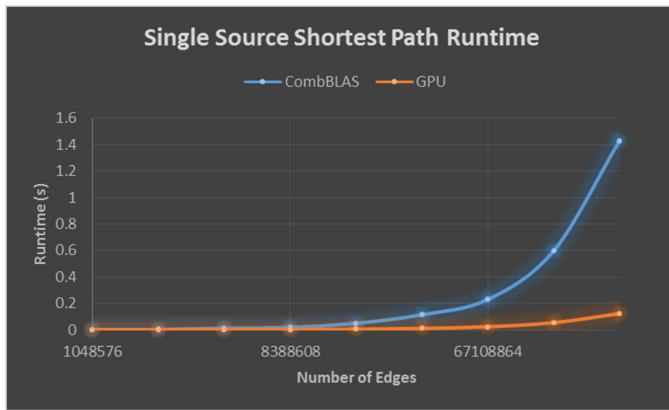

Figure 9 – Single-Source Shortest Path runtimes on synthetic data generated using the R-MAT model

For testing the Single-Source Widest Path algorithm on synthetic data we generated random edge weights distributed uniformly from 0.0 to 1.0. In this case again we see linear scaling with the number of edges and are able to outperform CombBLAS by nearly 1000%.

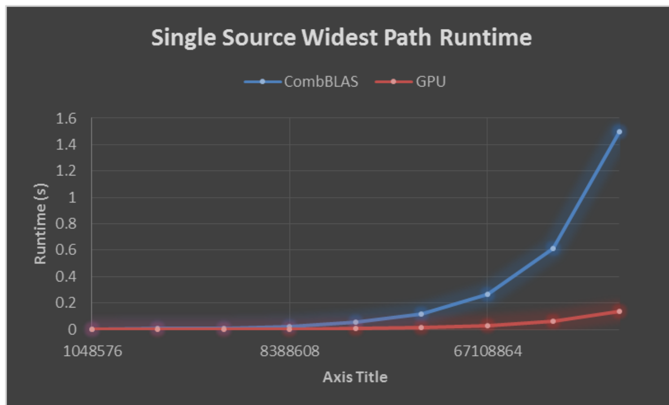

Figure 10 – Singe-Source Widest Path runtimes on synthetic data generated using the R-MAT model

IV. Conclusion

We have created GPU-accelerated graph analytics toolchain based on a C++ EDSL that allows users to implement a variety of algorithms. Even without abstract sparse linear algebra-backed operations, our GPU path is significantly faster than the CPU runtime; however, by automatically employing fusion and ASLA-based techniques within our tool, we achieve significant additional speedup. Additionally, we anticipate that further optimization will continue to improve performance.

References

[1] G. Malewicz, M. H. Austern, A. J. . 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*, New York, NY, USA, 2010, pp. 135–146.

[2] T. Veldhuizen, “C++ Gems,” S. B. Lippman, Ed. New York, NY, USA: SIGS Publications, Inc., 1996, pp. 475–487.

[3] D. Merrill and M. Garland, “Merge-based Sparse Matrix-vector Multiplication (SpMV) Using the CSR Storage Format,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2016, p. 43:1–43:2.

[4] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, “Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2014, pp. 979–990.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A Recursive Model for Graph Mining,” in *Proceedings of the 2004 SIAM International Conference on Data Mining*, 0 vols., Society for Industrial and Applied Mathematics, 2004, pp. 442–446.

[6] “farkhor/PaRMAT,” *GitHub*. [Online]. Available: https://github.com/farkhor/PaRMAT. [Accessed: 06-May-2016].