High-Performance Analysis of Filtered Semantic Graphs

Aydin Buluc
Armando Fox
John Gilbert
Shoaib Ashraf Kamil
Adam Lugowski
Leonid Oliker
Samuel Williams

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-61
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-61.html

May 6, 2012

High-Performance Analysis of Filtered Semantic Graphs

Aydın Buluç[§*]
abuluc@lbl.gov

Armando Fox[‡]
fox@cs.berkeley.edu

John R. Gilbert[†]
gilbert@cs.ucsb.edu

Shoaib Kamil[‡*]
skamil@cs.berkeley.edu

Adam Lugowski[†*]
alugowski@cs.ucsb.edu

Leonid Oliker[§]
loliker@lbl.gov

Samuel Williams[§]
swwilliams@lbl.gov

[§]CRD
Lawrence Berkeley Nat. Lab.
Berkeley, CA 94720

[†] Dept. of Computer Science
University of California
Santa Barbara, CA 93106

[‡]EECS Dept.
University of California
Berkeley, CA 94720

ABSTRACT

High performance is a crucial consideration when executing a complex analytic query on a massive semantic graph. In a semantic graph, vertices and edges carry "attributes" of various types. Analytic queries on semantic graphs typically depend on the values of these attributes; thus, the computation must either view the graph through a *filter* that passes only those individual vertices and edges of interest, or else must first materialize a subgraph or subgraphs consisting of only the vertices and edges of interest. The filtered approach is superior due to its generality, ease of use, and memory efficiency, but may carry a performance cost.

In the Knowledge Discovery Toolbox (KDT), a Python library for parallel graph computations, the user writes filters in a high-level language, but those filters result in relatively low performance due to the bottleneck of having to call into the Python interpreter for each edge. In this work, we use the Selective Embedded JIT Specialization (SEJITS) approach to automatically translate filters defined by programmers into a lower-level efficiency language, bypassing the upcall into Python. We evaluate our approach by comparing it with the high-performance C++ /MPI Combinatorial BLAS engine, and show that the productivity gained by using a high-level filtering language comes without sacrificing performance. We also present a new roofline model for graph traversals, and show that our high-performance implementations do not significantly deviate from the roofline.

1. INTRODUCTION

1.1 Motivation

Large-scale graph analytics are a central requirement of bioinformatics, finance, social network analysis, national security, and many other fields. Going beyond simple searches, analysts use high-performance computing systems to execute complex graph algorithms on large corpora of data. Often, a large semantic graph is built up over time, with the graph vertices representing entities of interest and the edges representing relationships of various kinds—for example, social network connections, financial transactions, or interpersonal contacts.

*Corresponding authors.

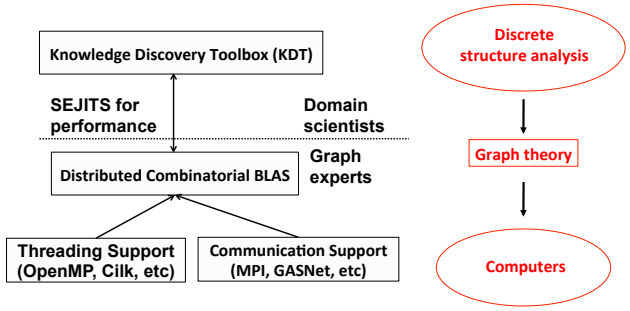


Figure 1: Overview of the high-performance graph-analysis software architecture described in this paper. KDT has graph abstractions and uses a very high-level language. Combinatorial BLAS has sparse linear-algebra abstractions, and geared towards performance.

In a semantic graph, edges and/or vertices are labeled with *attributes* that may represent (for example) a timestamp, a type of relationship, or a mode of communication. An analyst (i.e. a user of graph analytics) may want to run a complex workflow over a large graph, but wish to only use those graph edges whose attributes pass a filter defined by the analyst. For example, in a graph whose vertices represent Twitter users and whose edges represent either "following" or "retweeting" relationships, the analyst may want to search through vertices reachable from a particular user via the subgraph consisting only of "retweet" edges with timestamps earlier than June 30.

The Knowledge Discovery Toolbox [19] is a flexible Python-based open-source toolkit for implementing complex graph algorithms and executing them on high-performance parallel computers. KDT achieves high performance by invoking computational primitives supplied by a parallel C++ /MPI backend, the Combinatorial BLAS [5]. This paper presents new work that allows KDT users to define filters in Python, which act to modify KDT's action based on the attributes that label individual edges or vertices.

Filters raise performance issues for large-scale graph analysis. In many applications it is impossibly expensive to run a filter across an entire graph data corpus, materializing the filtered graph as a new object for analysis. In addition to

the obvious storage problems with materialization, the time spent during materialization is typically not amortized by many graph queries because the user modifies the query (or just the filter) during interactive data analysis. The alternative is to filter edges and vertices "on the fly" during execution of the complex graph algorithm. A graph algorithms expert can implement an efficient on-the-fly filter as a set of primitive Combinatorial BLAS operations coded in C/C++ ; but filters written at the KDT level, as graph operations in Python, incur a significant performance penalty.

Our solution to this challenge is to apply Selective Just-In-Time Specialization techniques from the SEJITS approach [7]. We define a semantic-graph-specific filter domain-specific language (DSL), a subset of Python, and use SEJITS to implement the specialization necessary for filters written in that subset to execute as efficiently as low-level C code.

As a result, we are able to demonstrate that SEJITS technology significantly accelerates Python graph analytics codes written in KDT and running on clusters and multicore CPUs. An overview of our approach is shown in Figure 1.

Figure 2 compares the performance of four filtering implementations on a breadth-first search query in a graph with 8 million vertices and 128 million edges. The chart shows time to perform the query as we synthetically increase the portion of the graph that passes the filter on an input R-MAT [18] graph of scale 23, The top two lines are the methods implemented in the current release v0.2 of KDT [2]: slowest is materializing the subgraph before traversal, and next is on-the-fly filtering in Python. The third, red, line is our new SEJITS+KDT implementation, which shows minimal overhead and comes very close to the performance of native Combinatorial BLAS in the fourth line.

1.2 Main contributions

The primary new contributions of this paper are:

1. A system design that allows domain-expert graph analysts to describe filtered semantic graph operations in a high-level language, using KDT v0.2.
2. An domain-specific language implementation that executes flexible filtering without sacrificing performance, using SEJITS selective compilation techniques.
3. Experimental demonstration of excellent performance scaling to graphs with millions of vertices and hundreds of millions of edges.
4. A new roofline performance model [24] for high-performance graph computation, suitable for evaluating the performance of filtered semantic graph operations.
5. A detailed case study of the use of algebraic semiring operations as an alternative low-level approach to filtering, using the Combinatorial BLAS.

1.3 Example of a filtered query

Here we present a simple example of a filtered query in a semantic graph. We will refer to this example through the paper, showing how the different implementations of filters express the query and comparing their performance executing it.

We consider a graph whose vertices are Twitter users, and whose edges represent two different types of relationships between users. In the first type, one user "follows" another; in

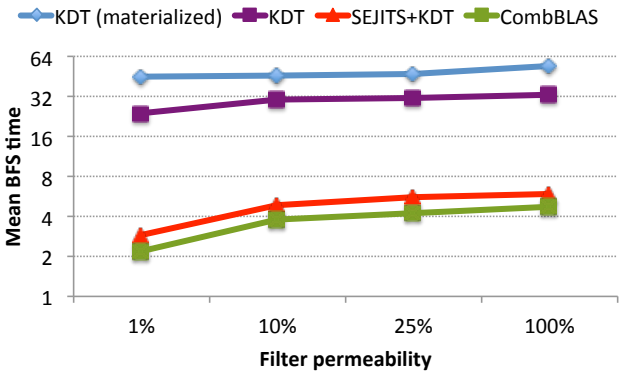


Figure 2: Performance of a filtered BFS query, comparing four methods of implementing custom filters. The vertical axis is running time in seconds on a log scale; lower is better. From top to bottom, the methods are: materializing the filtered subgraph; on-the-fly filtering with high-level Python filters in KDT; on-the-fly filtering with high-level Python filters specialized at runtime by SEJITS+KDT (this paper's main contribution); on-the-fly filtering with low-level C++ filters implemented as customized semiring operations and compiled into Combinatorial BLAS. The graph has 8 million vertices and 128 million edges. The runs use 36 cores (4 sockets) of Intel Xeon E7-8870 processors.

the second type, one user "retweets" another user's tweet. Each retweet edge carries as attributes a timestamp and a count. Figure 3 shows a fragment of such a graph. Our experiments are with several semantic graphs, of various sizes, constructed from publicly available data on tweets during 2009. The largest graph has about 17 million vertices and 720 million edges. Section 7 describes the datasets in more detail.

Our sample query is the one mentioned above: Given a vertex of interest, determine the number of hops required to reach each other vertex by using only retweeting edges timestamped earlier than June 30. The filter in this case is a boolean predicate on edge attributes that defines the types and timestamps of the edges to be used. The query is a breadth-first search in the graph that ignores edges that do not pass the filter.

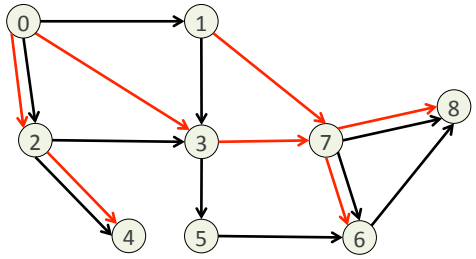


Figure 3: Graph of "following" and "retweeting" relationships. Black edges denote following, and red edges denote retweeting. Red edges are also labelled with counts and timestamps, not shown.

1.4 Outline of the paper

We first survey related work. Then, Section 3 shows how a filter can be implemented below the KDT level, as a user-specified semiring operation in the C++ /MPI Combinatorial BLAS library that underlies KDT. This is a path to high performance at the cost of usability: the analyst must translate the graph-attribute definition of the filter into low-level C++ code for custom semiring scalar operations in Combinatorial BLAS.

Section 4 describes the high-level filtering facility that is new in Version 0.2 of KDT, in which filters are specified as simple Python predicates. This approach yields easy customization, and scales to many queries from many analysts without demanding correspondingly many graph programming experts; however, it poses challenges to achieving high performance.

Section 5 is the technical heart of the paper, which describes how we meet these performance challenges by selective, embedded, just-in-time specialization with SEJITS.

Section 6 proposes a theoretical model that can be used to evaluate the performance of our implementations, giving "roofline" bounds on the performance of breadth-first search in terms of architectural parameters of a parallel machine, and the permeability of the filter.

Section 7 presents our experimental results, and Section 8 gives our conclusions and some remarks on future directions and problems.

2. RELATED WORK

Graph Algorithm Packages

Pegasus [15] is a graph-analysis package that uses MapReduce [9] in a distributed-computing setting. Pegasus uses a primitive called `GIM-V`, much like KDT's `SpMV`, to express vertex-centered computations that combine data from neighboring edges and vertices. This style of programming is called "think like a vertex" in Pregel [21], a distributed-computing graph API. Both of these systems require the application to be written in a low-level language (Java and C++, respectively) and neither has filter support.

Other libraries for high-performance computation on large-scale graphs include the Parallel Boost Graph Library [12], the Combinatorial BLAS [5], Georgia Tech's SNAP [3], and the Multithreaded Graph Library [4]. These are all written in C/C++ and do not include explicit filter support. The first two support distributed memory as well as shared memory while the latter two require a shared address space.

SPARQL [23] is a query language for Resource Description Framework (RDF) [16] that can support semantic graph database queries. The existing database engines that implement SPARQL and RDF support filtering based queries efficiently but they are currently not suitable for running traversal based tightly-coupled graph computations scalably in parallel environments.

The closest previous work is Green Marl [13], a domain specific language (DSL) for small-world graph exploration that runs on GPUs and multicore CPUs without support for distributed machines.

JIT Compilation of DSLs

Embedded DSLs [10] for domain-specific computations have a rich history, including DSLs that are compiled instead of interpreted [17]. Abstract Syntax Tree introspection for such DSLs has been used most prominently for database queries in ActiveRecord [1], part of the Ruby on Rails framework.

The approach applied here, which uses AST introspection combined with templates, was first applied to stencil algorithms and data parallel constructs [7], and subsequently to a number of domains including linear algebra and Gaussian mixture modeling [14].

3. FILTERS AS SCALAR SEMIRING OPS

The Combinatorial BLAS (CombBLAS for short) views graph computations as sparse matrix computations using various algebraic semirings (such as the tropical (min,+) semiring for shortest paths, or the real (+,*) semiring/field for numerical computation). The expert user can define new semirings and operations on them in C++ at the CombBLAS level, but most KDT users do not have the expertise for this.

Two fundamental kernels in CombBLAS, sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpGEMM), both explore the graph by expanding existing frontier(s) by a single hop. The semiring scalar multiply operation determines how the data on a sequence of edges are combined to represent a path, and the semiring scalar add operation determines how to combine two or more parallel paths. In a similar framework, Pegasus [15], semiring multiply is referred to as combine2 and semiring add is referred to as combineAll, followed by an assign operation. However, Pegasus's operations lack the algebraic completeness of CombBLAS's semiring framework.

Filters written as semiring operations in C++ can have high performance because the number of calls to the filter operations is asymptotically the same as the minimum necessary calls to the semiring scalar multiply operation, and the filter itself is a local operation that uses only the data on one edge. The filtered multiply returns a "null" object (formally, the semiring's additive identity or SAID) if the predicate is not satisfied.

For example, Figure 4 shows the scalar multiply operation for our running example of BFS on a Twitter graph. The usual semiring multiply for BFS is `select2nd`, which returns the second value it is passed; the multiply operation is modified to only return the second value if the filter succeeds. At the lowest levels of SpMV, SpGEMM, and the other CombBLAS primitive, the return value of the scalar multiply is checked against SAID, the additive identity of the semiring (in this example, the default constructed ParentType object is the additive identity), and the returned object is retained only if it doesn't match the SAID.

4. KDT FILTERS IN PYTHON

The Knowledge Discovery Toolbox [19, 20] is a flexible open-source toolkit for implementing complex graph algorithms and executing them on high-performance parallel computers. KDT is targeted at two classes of users: domain-expert analysts who are not graph experts and who use KDT primarily by invoking existing KDT routines from Python, and graph-algorithm developers who use KDT primarily by writing Python code that invokes and composes KDT's computational primitives. These computational primitives are supplied by a parallel backend, the Combinatorial BLAS [5], which is written in C++ with MPI for high performance.

4.1 Filter semantics

```
ParentType multiply( const TwitterEdge & arg1,
                     const ParentType & arg2)
{
    time_t end = stringtotime("2009−06−30");
    if (arg1.isRetweet() && arg1.latest(end))
        return arg2;  // unfiltered multiply yields normal value
    else
        return ParentType(); // filtered multiply yields SAID
}
```

Figure 4: An example of a filtered scalar semiring operation in Combinatorial BLAS. This multiply operation only traverses edges that represent a retweet before June 30.

In KDT, any graph algorithm can be performed with an edge filter. A filter is a unary predicate on an edge that returns true if the edge is to be considered, or false if it is to be ignored. The KDT user writes a filter predicate as a Python function or lambda expression of one input that returns a boolean value; Figure 5 is an example.

Using a filter does not require any change in the code for the graph algorithm. For example, KDT code for betweenness centrality or for breadth-first search is the same whether or not the input semantic graph is filtered. This works because the filtering is done in the low-level primitives; user code remains ignorant of filters. Our design allows all current and future KDT algorithms to support filters without any extra effort required on the part of the algorithm designer.

Since filtered graphs behave just like unfiltered ones, it is possible in KDT to add another filter to an already filtered graph. The result is a nested filter whose predicate is a lazily-evaluated logical *and* of the individual filter predicates. Filters are evaluated in the order they are added. This allows both end users and algorithm designers to use filters for their own purposes without having to worry about each other.

4.2 Materializing filters and on-the-fly filters

KDT supports two approaches for filtering semantic graphs:

- **Materializing filter:** When a filter is placed on a graph (or matrix or vector), the entire graph is traversed and a copy is made that includes only the edges that pass the filter. We refer to this approach as *materializing* the filtered graph.
- **On-the-fly filter:** No copy of the graph/matrix/vector is made. Rather, every primitive operation (e.g. semiring scalar multiply and add) applies the filter to its inputs when it is called. Roughly speaking, every primitive operation accesses the graph through the filter and behaves as if the filtered-out edges were not present.

Both materializing and on-the-fly filters have their place; neither is superior in every situation. For example, materialization may be more efficient when a user wants to run many analyses on a well-defined small subset of a large graph. On the other hand, materialization may be impossible if the graph already fills most of memory; and materialization may be much more expensive than on-the-fly filtering for a query

```
# G is a kdt.DiGraph
def earlyRetweetsOnly(e):
    return e.isRetweet() and e.latest < str_to_date("2009−06−30")

G.addEFilter(earlyRetweetsOnly)
G.e.materializeFilter() # omit this line to use on−the−fly filtering

# perform some operations or queries on G

G.delEFilter(earlyRetweetsOnly)
```

Figure 5: Adding and removing an edge filter in KDT, with or without materialization.

whose filter restricts it to a localized neighborhood and thus does not even touch most of the graph. Indeed, an analyst who needs to modify and fine-tune a filter while exploring data may not be willing to wait for materialization at every step of the way.

The focus of this paper is on-the-fly filtering and how to make it more efficient, though our experiments do include comparisons with materializing filters.

4.3 Implementation details

Filtering a semiring operation requires the semiring scalar multiply to be able to return "nothing", in the sense that the result should be the same as if the multiply had never happened. In semiring terms, this means that the multiply operation must return the semiring's additive identity (SAID for short). CombBLAS treats the additive identity SAID the same as any other value. However, CombBLAS uses a sparse data structure to represent a graph as an adjacency matrix—and, formally speaking, SAID is the implicit value of any matrix entry that is not stored explicitly.

CombBLAS ensures that SAID is never stored as an explicit value in a sparse structure. (This corresponds to Matlab's convention that explicit zeros are never stored in sparse matrices [11], and differs from the convention in the CSparse sparse matrix package [8].) Note that SAID need not be "zero": for example, in the min-plus semiring used for shortest path computations, SAID is ∞. Indeed, it is possible for a single graph or matrix to be used with different underlying semirings whose operations use different SAIDs.

We benchmarked several approaches to representing, manipulating, and returning SAID values from semiring scalar operations. In the end, we decided that the basic scalar operations would include a `returnedSAID()` predicate, which can be called after the scalar operation, and that KDT would not have an explicit representation of a SAID value.

The result is a clean implementation of on-the-fly filters: filtered semiring operations just require a shim in the `multiply()` function that causes `returnedSAID()` to return true if the value is filtered; the lower-level algorithms call this function after performing the scalar multiply operation.

5. SEJITS AND FILTERS

In order to mitigate the slowdown caused by defining semirings in Python, which results in a serialized upcall into Python for each operation, we opt to instead use the Selective Embedded Just-In-Time Specialization (SEJITS) approach [7]. By defining an embedded DSL for KDT filters, and then translating it to C++ , we can avoid performance

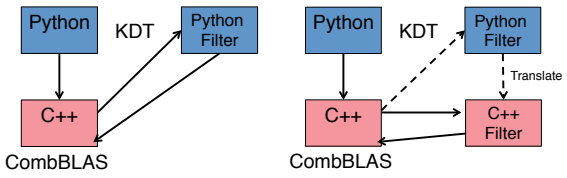


Figure 6: Left: Calling process for filters in KDT. For each edge, the C++ infrastructure must upcall into Python to apply the filter. Right: Using our DSL for filters, the C++ infrastructure calls the translated version for each edge, eliminating the upcall overhead.

penalties while still allowing users the flexibility to specify filters in Python. We use the Asp[1] framework to implement our DSL.

Our approach is shown in Figure 6. In the usual KDT case, filters are written as simple Python functions. Since KDT uses Combinatorial BLAS at the low level to perform graph operations, each operation at the Combinatorial BLAS level must check to see whether the vertex or edge should be taken into account, requiring a per-vertex or per-edge upcall into Python. Furthermore, since Python is not thread-safe, this essentially serializes the computation in each MPI process.

In this work, we define an embedded domain specific language for filters, and allow users to write their filters in this DSL, expressed as a subset of Python with normal Python syntax. Then, at instantiation, the filter source code is introspected to get the Abstract Syntax Tree (AST), and then is translated into low-level C++ . Subsequent applications of the filter use this low-level implementation, sidestepping the serialization and cost of upcalling into Python.

In the next section, we define our domain-specific language and show several examples of filters written in Python.

5.1 Semantic Model for Filters

In our approach, we first define the *semantic model* of filters, which is the intermediate form of our DSL. The semantic model expresses the semantics of filters. After defining this, we then map pure-Python constructs to constructs in the semantic model. It is this pure-Python mapping that users use to write their filters.

In defining the semantic model, we must look at what kinds of operations filters perform. In particular, vertex and edge filters are functions that take in one or two inputs and return a boolean. Within the functions, filters must allow users to inspect fields of the input data types, do comparisons, and perhaps perform arithmetic with fields. In addition, we want to (as much as possible) prevent users from writing filters that do not conform to our assumptions; although we could use analysis for this, it is much simpler to construct the language in a manner that prevents users from writing non-conformant filters. If the filter does not fit into our language, we run it in the usual fashion, by doing upcalls into pure Python. Thus, if the user writes their filters correctly, they achieve fast performance, otherwise the user experience is no worse than before— the filter still runs, just not at fast speed.

The semantic model is shown in Figure 7. We have constructed this to make it easy to write filters that are "correct-by-construction;" that is, if they fit into the semantic model, they follow the restrictions of what can be translated. For example, we require that the return be provably a boolean (by forcing the BoolReturn node to have a boolean body), and that there is either a single input or two inputs (either UnaryPredicate or BinaryPredicate).

[1]URL blinded for submission

```
UnaryPredicate(input=Identifier, body=BoolExpr)

BinaryPredicate(inputs=Identifier*, body=BoolExpr)
    check assert len(self.inputs)==2

Expr = Constant
         | Identifier
         | BinaryOp
         | BoolExpr

BoolExpr = BoolConstant
         | IfExp
         | Attribute
         | BoolReturn
         | Compare

Identifier(name=types.StringType)
Constant(value = types.IntType | types.FloatType)
BoolConstant(value = types.BooleanType)

Compare(left=Expr, op=(ast.Eq | ast.NotEq | ast.Lt | ast.LtE
                       | ast.Gt | ast.GtE), right=Expr)

BinaryOp(left=Expr, op=(ast.Add | ast.Sub), right=Expr)

IfExp(test=BoolExpr, body=BoolExpr, orelse=BoolExpr)

# this if for a.b
Attribute(value=Identifier, attr=Identifier)

# our return type must be provably a boolean
BoolReturn(value = BoolExpr)
```

Figure 7: Semantic Model for KDT filters using SEJITS.

Given the semantic model, now we define a mapping from Python syntax to the semantic model.

5.2 Python Syntax for the Filter DSL

Users of KDT are not exposed to the semantic model. Instead, the language they use to express filters in our DSL is a subset of Python, corresponding to the supported operations. Informally, we specify the language by talking about what a filter can do: namely, a filter takes in one or two inputs (that are of pre-defined edge/vertex types), must return a boolean, and is allowed to do comparisons, accesses, and arithmetic on immediate values and edge/filter instance variables. In addition, to facilitate translation, we require that a filter be an object that inherits from the `PcbFilter` Python class, and that the filter function itself is a member function called `filter`.

The example KDT filter from Figure 5 is presented in SEJITS syntax in Figure 8. Note that because a filter cannot call a function, we must use immediate values for checking the timestamp. However, even given our relatively restricted syntax, users can specify a large class of useful filters in our DSL. In addition, if the filter does not fit into our DSL, it is still executed using the slower upcalls to pure Python after

```
class MyFilter(PcbFilter):
  def filter(e):
    # if it is a retweet edge
    if (e.isRetweet and
        # and it is before June 30
        e.latest < JUNE_30_2009):
      return True
    else:
      return False
```

Figure 8: Example of an edge filter that the translation system can convert from Python into fast C++ code.

	First Run	Subsequent
Codegen	0.0545 s	0 s
Compile	4.21 s	0 s
Import	0.032 s	0.032 s

Table 1: Overheads of using the filtering DSL.

issuing a warning to the user.

5.3 Implementation in C++

We modify the normal KDT C++ filter objects, which are instantiated with pointers to Python functions, by adding a function pointer that is checked before executing the upcall to Python. This function pointer is set by our translation machinery to point to the translated function in C++ . When executing a filter, the pointer is first checked, and if non-null, directly calls the appropriate function.

Compared to Combinatorial BLAS, at runtime we have the additional sources of overheads relating to the null check and function pointer call. However, relative to the non-translated KDT machinery, these are trivial costs for filtering, particularly compared to the penalty of upcalling into Python.

Overheads of code generation are shown in Table 1. On first running using a particular filter, the DSL infrastructure translates and compiles the filter in C++ ; most of the time here is spent calling the external C++ compiler, which is not optimized for speed. Subsequent calls only incur the penalty of Python's `import` statement, which loads the cached library.

6. A ROOFLINE MODEL OF BFS

In this section, we extend the Roofline model [24] to quantify the performance bounds of BFS as a function of optimization and filter success rate. The Roofline model is a visually intuitive representation of the performance characteristics of a kernel on a specific machine. It uses bound and bottleneck analysis to delineate performance bounds arising from bandwidth or compute limits. In the past, the Roofline model has primarily been used for kernels found in high-performance computing. These kernels tend to express performance in floating-point operations per second and are typically bound by the product of arithmetic intensity (flops per byte) and STREAM [22] (long unit-stride) bandwidth. In the context of graph analytics, none of these assumptions hold.

In order to model BFS performance, we decouple in-core compute limits (filter performance as measured in processed edges per second) from memory access performance. The in-core filter performance limits were derived by extracting the relevant CombBLAS, KDT, and SEJITS+KDT versions of the kernels and targeting arrays that fit in each core's cache. We run the edge processing inner kernels 10000 times (as opposed to once) to obfuscate any memory system related effects to get the in-core compute limits.

Analogous to arithmetic intensity, we can quantify the average number of bytes we must transfer from DRAM per edge we process — bytes per processed edge. In the following analysis, the indices are 8 bytes and the edge payload is 16 bytes. BFS exhibits three memory access patterns. First, there is a unit-stride *streaming* access pattern arising from access of vertex pointers (this is amortized by degree) as well as the creation of a sparse output vector that acts as the new frontier (index, parent's index). The latter incurs 32 bytes of traffic per traversed edge in write-allocate caches assuming the edge was not filtered. Second, access to the adjacency list follows a *stanza-like* memory access pattern. That is, small blocks (stanzas) of consecutive elements are fetched from effectively random locations in memory. These stanzas are typically less than the average degree. This corresponds to approximately 24 bytes (16 for payload and 8 for index) of DRAM traffic per processed edge. Finally, updates to the list of visited vertices and the indirections when accessing the graph data structure exhibit a memory access pattern in which effectively *random* 64-bit elements are updated (assuming the edge was not filtered). Similarly, each visited vertex generates 24 bytes of random access traffic to follow indirections on the graph structure before being able to access its edges. In order to quantify these bandwidths, we wrote a custom version of STREAM that provides stanza-like memory access patterns (read or update) with spatial locality varying from 8 bytes (random access) to the size of the array (STREAM).

The memory bandwidth requirements depend on the number of edges processed (examined), number of edges traversed (that pass the filter), and the number of vertices in the frontier over all iterations. For instance, an update to the list of visited vertices only happens if the edge actually passes the filter. Typically, the number of edges traversed is roughly equal to the permeability of the filter times the number of edges processed. To get a more accurate estimate, we collected statistics from one of the synthetically generated R-MAT graphs that are used in our experiments. These statistics are summarized in Table 2. Similarly, we quantify the volume of data movement by operation and memory access type (*random*, *stanza-like*, and *streaming*) noting the corresponding bandwidth on Mirasol, our Intel Xeon E7-8870 test system (see Section 7), in Table 3. Combining Tables 2 and 3, we calculate the average number of processed edges per second as a function of filter permeability by summing data movement time by type and inverting.

Figure 9 presents the resultant Roofline-inspired model for Mirasol. Note that these are all upper bounds on the best performance achievable and the underlying implementation might incur additional overheads from internal data structures, MPI buffers, etc. For example, it is common to locally sort the discovered vertices to efficiently merge them later in the incoming processor; an overhead we do not account for as it is not an essential step of the algorithm.

As the Roofline model selects ceilings by optimization, and bounds performance by their minimum, we too may select a filter implementation (pure Python KDT, SEJITS+KDT,

Table 2: Statistics about the filtered BFS runs on the R-MAT graph of Scale 23 (M: million)

Filter permeability	Vertices visited	Edges traversed	Edges processed
1%	655,904	2.5 M	213 M
10%	2,204,599	25.8 M	250 M
25%	3,102,515	64.6 M	255 M
100%	4,607,907	258 M	258 M

Table 3: Breakdown of the volume of data movement by memory access pattern and operation.

Memory access type	Vertices visited	Edges traversed	Edges processed	Bandwidth on Mirasol
Random	24 bytes	8 bytes	0	9.09 GB/s
Stanza	0	0	24 bytes	36.6 GB/s
Stream	8 bytes	32 bytes	0	106 GB/s

or the CombBLAS limit) and the weighted bandwidth limit (in black) and look for the minimum.

We observe a pure Python KDT filter will result in a performance bound more than an order of magnitude lower than the bandwidth limit. Conversely, the bandwidth limit is about 25× lower than the CombBLAS in-core performance limit. Ultimately, the performance of a SEJITS specialized filter is sufficiently fast to ensure a BFS implementation will be bandwidth-bound. This is a very important observation that explains why SEJITS+KDT performance is so close to CombBLAS performance in practice (as shown later in Section 7) even though its in-core performance is 4× slower.

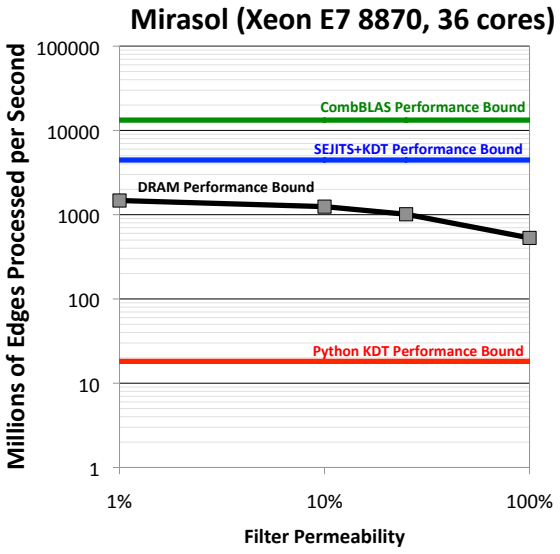


Figure 9: Roofline-inspired model for filtered BFS computations. Performance bounds arise from bandwidth, CombBLAS, KDT, or SEJITS+KDT filter performance, and filter success rate.

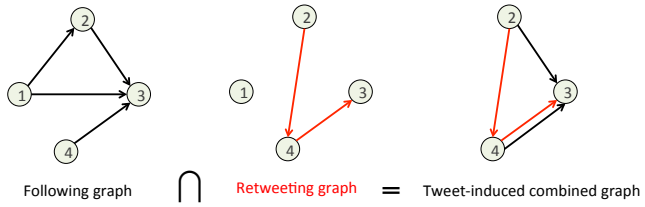


Figure 10: Toy example illustrating the process of building a combined graph induces by the tweets

7. EXPERIMENTS

7.1 Methodology

To evaluate our methodology, we examine graph analysis behavior on an Mirasol, an Intel Nehalem-based machine, as well as the Hopper Cray XE6 supercomputer. Mirasol is a single node platform composed of four Intel Xeon E7-8870 processors. Each socket has ten cores running at 2.4 GHz, and supports two-way simultaneous multithreading (20 thread contexts per socket). The cores are connected to a very large 30 MB L3 cache via a ring architecture. The sustained stream bandwidth is about 30 GB/s per socket. The machine has 256 GB 1067 MHz DDR3 RAM. We use OpenMPI 1.4.3 with GCC C++ compiler version 4.4.5, and Python 2.6.6.

Hopper is a Cray XE6 massively parallel processing (MPP) system, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. In reality, each socket (multichip module) has two dual hex-core chips, and so a node can be viewed as a four-chip compute configuration with strong NUMA properties. Each Opteron chip contains six superscalar, out-of-order cores capable of completing one (dual-slot) SIMD add and one SIMD multiply per cycle. Additionally, each core has private 64 KB L1 and 512 KB low-latency L2 caches. The six cores on a chip share a 6MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average STREAM [22] bandwidth of 12GB/s per chip. Each pair of compute nodes shares one Gemini network chip, which collectively form a 3D torus. We use Cray's MPI implementation, which is based on MPICH2, and compile our code with GCC C++ compiler version 4.6.2 and Python 2.7. Complicating our experiments, some compute nodes do not contain a compiler; we ensured that a compute node with compilers available was used to build the SEJITS+KDT filters.

7.2 Test data sets

For most of our parallel scaling studies, we use synthetically-generated R-MAT [18] graphs with a very skewed degree distribution. An R-MAT graph of scale N has 2^N vertices and approximately $edgefactor * 2^N$ edges. In our tests, our $edgefactor$ is 16, and our R-MAT seed paratemeters a, b, c, and d are $0.59, 0.19, 0.19, 0.05$ respectively. After generating this non-semantic (boolean) graph, the edge payloads are artificially introduced using a random number generator in a way that ensures targer filter permeability. The edge type is the same as the Twitter edge type described below, to be consistent between experiments on real and synthetic data.

We also use graphs from real social network interactions, from anonymized Twitter data. In our Twitter graphs, edges can represent two different types of interactions. The first

```
struct TwitterEdge
{
        bool follower;
        time_t latest; // set if count>0
        short count; // number of tweets
};
```

Figure 11: The edge data structure used for the tweet-induced combined (tweeting+following) graph in C++ (methods are omitted for brevity)

Table 4: Sizes (vertex and edge counts) of different combined twitter graphs.

Label	Vertices (millions)	Edges (millions)		
		Tweet	Follow	Tweet&follow
Small	0.5	0.7	65.3	0.3
Medium	4.2	14.2	386.5	4.8
Large	11.3	59.7	589.1	12.5
Huge	16.8	102.4	634.2	15.6

interaction is the "following" relationship where an edge from v_i to v_j means that v_i is following v_j (note that these directions are consistent with the common authority-hub definitions in the World Wide Web). The second interaction encodes an abbreviated "retweet" relationship: an edge from v_i to v_j means that v_i has mentioned v_j at least once in their tweets. The edge also keeps the number of such tweets (count) as well as the last tweet date if count is larger than one.

The tweets occurred in the period of June-December of 2009. To allow scaling studies, we creates subsets of these tweets, based on the date they occur. The *small* dataset contains tweets from the first two weeks of June, the *medium* dataset contains tweets that happened in June and July, the *large* dataset contains tweets dated June-September, and finally the *huge* dataset contains all the tweets from June to December.

These partial tweets are then induced upon the graph that represents the follower/followee relationship. If a person tweeted someone or has been tweeted by someone, then the vertex is retained in the tweet-induced combined graph.

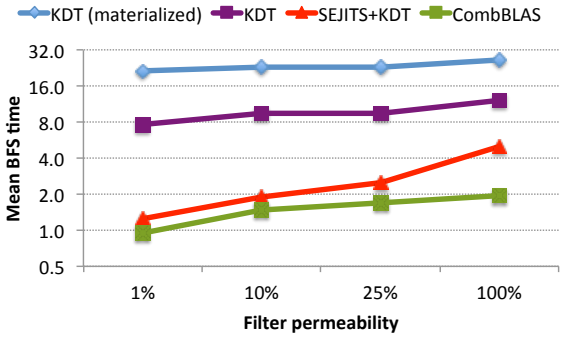


Figure 12: Relative breadth-first search performance of four methods. y-axis uses a log scale. The experiments are run using 24 nodes of Hopper, where each node has two 12-code AMD processors.

Table 5: Statistics about the largest strongly connected components of the twitter graphs

	Vertices	Edges traversed	Edges processed
Small	78,397	147,873	29.4 million
Medium	55,872	93,601	54.1 million
Large	45,291	73,031	59.7 million
Huge	43,027	68,751	60.2 million

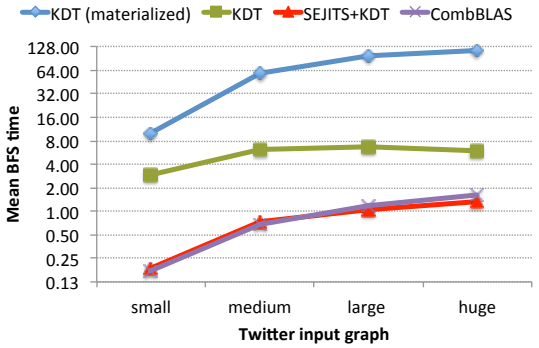


Figure 13: Relative filtered breadth-first search performance of four methods on real Twitter data. The y-axis is in seconds on a log scale. The runs use 36 cores (4 sockets) of Intel Xeon E7-8870 processors.

A simple example shows this process in Figure 10. Node 1 is eliminated because it has not tweeted about anyone, nor has it been tweeted by someone. All the follower/followee edges connected to Node 1 are also eliminated. In the combined graph, three edges remain: $e(2,3)$, $e(2,4)$, and $e(4,3)$. Each edge can potentially encode both the following relation and the tweeted relationship, however both fields are needed only in the case of $e(4,3)$. The data structure for edges in the combined graph is shown in Figure 11.

More details for these four different (small-huge) combined graphs is listed in Table 4. Contrary to the synthetic data, the real twitter data is directed and we only report BFS runs that hit the largest strongly connected component of the filter-induced graphs. More information on the statistics of the largest strongly connected components of the graphs can be found in Table 5. Processed edge count includes both the edges that pass the filter and the edges that are filtered-out. After symmetrization, our huge graph requires approximately 45GB of memory, not accounting for space for vectors, MPI buffers, and other auxiliary data structures.

7.3 Experimental results

Synthetic data set: Figure 12 shows the relative distributed-memory performance of four methods in performing breadth-first search on a graph with 32 million vertices and 512 million edges, with varying filter selectivity. The structure of the input graph is an R-MAT of scale 25, and the edges are artificially introduced so that the specified percentage of edges pass the filter. These experiments are run on Hopper using 576 MPI processes with one MPI process per core. A similar figure (Figure 2) for Mirasol exists in the introduction. The SEJITS+KDT implementation closely tracks CombBLAS performance, except for the 100% filter; the performance hit here is mostly due to anomalous

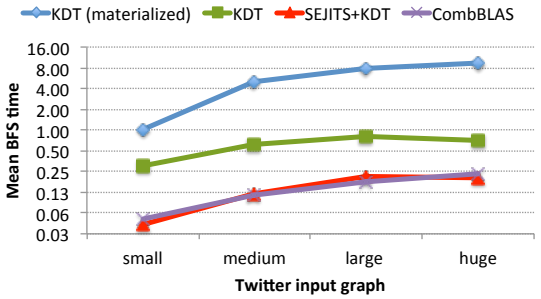


Figure 14: Relative filtered breadth-first search performance of four methods on real twitter data; y-axis is in seconds on a log scale. The experiments are run using 24 nodes of Hopper, where each node has two 12-code AMD processors.

performance variability on the test machine.

Twitter data set: The filter used in the experiments with the Twitter data set is to keep edges whose latest retweeting interaction happened by June 30, 2009, and is explained in detail in Section 1.3. Figure 13 shows the relative performance of four systems in performing breadth-first search on real graphs that represent the twitter interaction data on Mirasol. Figure 14 shows the same graph Hopper using 576 MPI processes. SEJIT+KDT's performance is identical to the performance of CombBLAS in these data sets, showing that for real-life inspired cases, our approach is as fast as the underlying high-performance library.

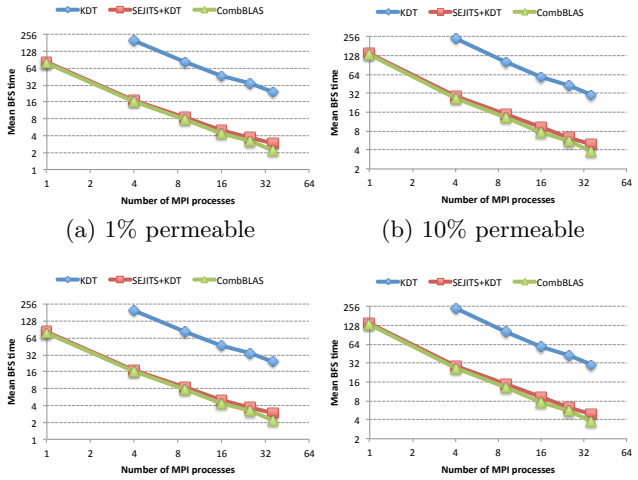
Parallel scaling: The parallel scaling of our approach is shown in Figure 15 for lower concurrencies on Mirasol. CombBLAS achieves remarkable linear scaling with increasing process counts (34-36X on 36 cores), while SEJITS+KDT closely tracks its performance and scaling. Single core KDT runs did not finish in a reasonable time to report. We do not report performance of materialized filters as they were previously shown to be the slowest.

Parallel scaling at higher concurrencies is done on Hopper, using the scale 25 synthetic R-MAT data set. Figure 16 shows the comparative performance of KDT on-the-fly filters, SEJITS+KDT, and CombBLAS, with 10% and 25% filter permeability.

Finally, we show weak scaling results on Hopper using 1% filter permeability (other cases experienced similar performance). In this run, shown in Figure 17, each MPI process is responsible for approximately 11 million original edges (hence 22 million edges after symmetricization). More concretely, 121-concurrency runs are obtained on a scale 23 R-MAT graph, 576-concurrency runs are obtained on scale 25 R-MAT graph, and 2025-concurrency runs are obtained on scale 27 R-MAT graph (1 billion edges). KDT curve is mostly flat (only 9% deviation) due to its in-core computational bottlenecks, while SEJITS+KDT and CombBLAS shows higher deviations (54% and 62%, respectively) from the perfect flat line. However, these deviations are expected on a large scale BFS run and experienced on similar architectures [6].

8. CONCLUSION

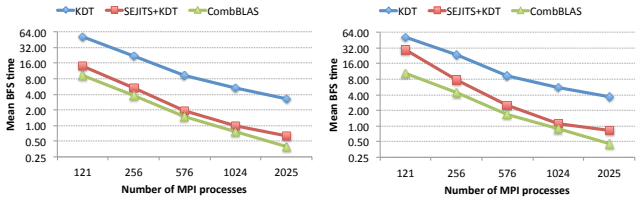
The KDT graph analytics system achieves customizability through user-defined filters, high performance through the use of a scalable parallel library, and conceptual simplicity



(a) 1% permeable (b) 10% permeable

(c) 25% permeable (d) 100% permeable

Figure 15: Parallel 'strong scaling' results of filtered BFS on Mirasol, with varying filter permeability on a synthetic data set (R-MAT scale 23). Both axes are in log-scale, time is in seconds.



(a) 10% permeable (b) 25% permeable

Figure 16: Parallel 'strong scaling' results of filtered BFS on Hopper, with varying filter permeability on a synthetic data set (R-MAT scale 25). Both axes are in log-scale, time is in seconds.

through appropriate graph abstractions expressed in a high-level language.

We have shown that the performance hit of expressing filters in a high-level language can be mitigated by Just-in-Time Specialization. In particular, we have shown that our embedded DSL for filters can enable Python code to achieve comparable performance to a pure C++ implementation. A roofline analysis shows that the specializer enables filtering to move from being compute-bound to memory bandwidth-bound. We demonstrated our approach on both real-world data and large generated datasets. Our approach scales to graphs on the order of hundreds of millions of edges, and machines with thousands of processors.

In future work we will further generalize our DSL to support a larger subset of Python, as well as expand SEJITS support beyond filtering to cover more KDT primitives. An open question is whether CombBLAS performance can be pushed closer to the bandwidth limit by eliminating internal data structure overheads.

Acknowlegements

This work was supported in part by National Science Foundation grant CNS-0709385. Portions of this work were performed at the UC Berkeley Parallel Computing Laboratory

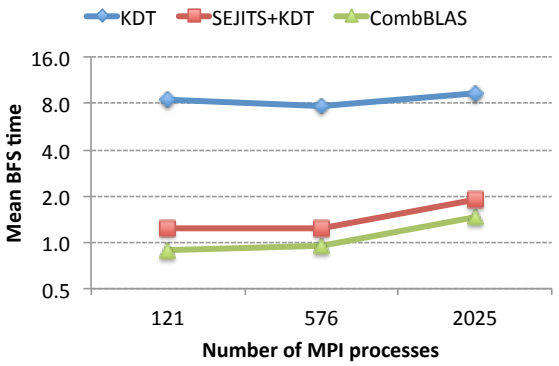


Figure 17: Parallel 'weak scaling' results of filtered BFS on Hopper, using 1% percent permeability. y-axis is in log scale, time is in seconds.

(Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the Universal Parallel Computing Research Centers (UPCRC) awards from Microsoft Corp. (Award #024263) and Intel Corp. (Award #024894), with matching funds from the UC Discovery Grant (#DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05-CH-11231. Authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05-CH-11231.

9. REFERENCES

[1] Active Record - Object-Relation Mapping Put on Rails. http://ar.rubyonrails.org, 2012.

[2] Knowledge Discovery Toolbox. http://kdt.sourceforge.net, 2012.

[3] D. A. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS'08: Proceedings of the 2008 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12. IEEE Computer Society, 2008.

[4] J.W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. In *Proc. Workshop on Multithreaded Architectures and Applications*. IEEE Press, 2007.

[5] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(4):496–509, 2011.

[6] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Supercomputing*, 2011.

[7] B. Catanzaro, S.A. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K.A. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA)*, 2009.

[8] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th Symposium on Operating System Design and Implementation*, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association.

[10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[11] J.R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1992.

[12] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *Proc. Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'05)*, 2005.

[13] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 349–362, New York, NY, USA, 2012. ACM.

[14] S. Kamil, D. Coetzee, S. Beamer, H. Cook, E. Gonina, J. Harper, J. Morlan, and A. Fox. Portable parallel performance from sequential, productive, embedded domain specific languages. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2012.

[15] U. Kang, C.E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[16] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, February 1999.

[17] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Conference on Domain-Specific Languages - Volume 2*, DSL'99, pages 9–9, Berkeley, CA, USA, 1999. USENIX Association.

[18] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *PKDD*, pages 133–145. Springer, 2005.

[19] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In *Proceedings of the Twelfth SIAM International Conference on Data Mining (SDM12)*, pages 930–941, April 2012.

[20] A. Lugowski, A. Buluç, J. Gilbert, and S. Reinhardt. Scalable Complex Graph Analysis with the Knowledge Discovery Toolbox. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*,

2012.

[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 International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[22] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[23] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF (working draft). Technical report, W3C, March 2007.

[24] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.