UC Berkeley
UC Berkeley Previously Published Works

Title
High-productivity and high-performance analysis of filtered semantic graphs

Permalink
https://escholarship.org/uc/item/0vf137bs

ISBN
978-1-4673-6066-1

Authors
Buluç, A
Duriakova, E
Fox, A
et al.

Publication Date
2013-10-07

DOI
10.1109/IPDPS.2013.52

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

High-Productivity and High-Performance Analysis of Filtered Semantic Graphs

Aydın Buluç[1]*, Erika Duriakova[2], Armando Fox[4], John R. Gilbert[3],
Shoaib Kamil[4,5]*, Adam Lugowski[3]*, Leonid Oliker[1], Samuel Williams[1]
[1]*CRD, Lawrence Berkeley National Laboratory, Berkeley, USA*
[2] *School of Computer Science and Informatics, University College Dublin, Ireland*
[3]*Dept. of Computer Science, University of California, Santa Barbara, USA*
[4]*EECS Dept, University of California, Berkeley, USA*
[5]*CSAIL, Massachusetts Institute of Technology, Cambridge, USA*
**Corresponding authors: abuluc@lbl.gov, skamil@mit.edu, alugowski@cs.ucsb.edu*

***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 view the graph through a *filter* that passes only those individual vertices and edges of interest.**

Knowledge Discovery Toolbox (KDT), a Python library for parallel graph computations, is customizable in two ways. First, the user can write custom graph algorithms by specifying operations between edges and vertices. These programmer-specified operations are called *semiring operations* due to KDT's underlying linear-algebraic abstractions. Second, the user can customize existing graph algorithms by writing filters that return true for those vertices and edges the user wants to retain during algorithm execution. For high productivity, both semiring operations and filters are written in a high-level language, resulting in relatively low performance due to the bottleneck of having to call into the Python virtual machine for each vertex and edge.

In this work, we use the Selective Embedded JIT Specialization (SEJITS) approach to automatically translate semiring operations and 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 Combinatorial BLAS engine, and show our approach enables users to write in high-level languages and still obtain the high performance of low-level code. We also present a new roofline model for graph traversals, and show that our high-performance implementations do not significantly deviate from the roofline. Overall, we demonstrate the first known solution to the problem of obtaining high performance from a productivity language when applying graph algorithms selectively on semantic graphs.

I. Introduction

Large-scale graph analytics is a central requirement of bioinformatics, finance, social network analysis, national security, and many other fields that deal with "big data". 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.

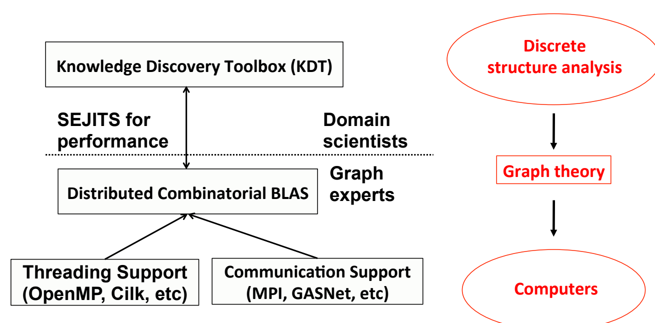


Fig. 1. Overview of the high-performance graph-analysis software architecture described in this paper. KDT has graph abstractions and uses a very high-level language. Combinatorial BLAS has sparse linear-algebra abstractions, and geared towards performance.

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.

The Knowledge Discovery Toolbox [23] is a flexible Python-based open-source toolbox for implementing complex graph algorithms and executing them on high-performance parallel computers. KDT achieves high performance by invoking linear-algebraic computational primitives supplied by a parallel C++/MPI backend, the Combinatorial BLAS [7]. Combinatorial BLAS uses broad definitions of matrix and vector operations: the user can define custom callbacks to override semiring scalar multiplications and additions that correspond to operations between edges and vertices.

Filters act to enable or disable KDT's action (the semiring operations) based on the attributes that label individual edges or vertices. The programmer's ability to specify custom filters and semirings directly in a high-level language like Python is crucial to ensure high-productivity and customizability of graph analysis software. This paper presents new work that allows KDT users to define filters and semirings in Python,

without paying the performance penalty of upcalls to Python.

Filters raise performance issues for large-scale graph analysis. In many applications it is prohibitively expensive to run a filter across an entire graph data corpus, and materialize 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 predicate callbacks 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 [8]. We define two semantic-graph-specific domain-specific languages (DSL): one for filters and one for the user-defined scalar semiring operations for flexibly implementing custom graph algorithms. Both DSLs are subsets of Python, and they use SEJITS to implement the specialization necessary for filters and semirings written in that subset to execute efficiently as low-level C++ code. Unlike writing a compiler for the full Python language, implementing our DSLs requires much less effort due to their domain-specificity and our use of existing SEJITS infrastructure; on the other hand, unlike forcing users to write C++code, we preserve the usability and high-level nature of expressing computations in Python.

We demonstrate that SEJITS technology significantly accelerates Python graph analytics codes written in KDT, running on clusters and multicore CPUs. An overview of our approach is shown in Figure 1. SEJITS specialization allows our graph analytics system to bridge the gap between the performance-oriented Combinatorial BLAS and and usability-oriented KDT.

Throughout the paper, we will use a running example query to show how different implementations of filters and semiring operations express the query and compare 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 the second type, one user "retweets" another user's tweet. Each retweet edge carries as attributes a timestamp and a count. The example query is a breadth-first search (BFS) through vertices reachable from a particular user via the subgraph consisting only of "retweet" edges with timestamps earlier than June 30.

The primary new contributions of this paper are:

1) A domain-specific language implementation that enables flexible filtering and customization of graph algorithms without sacrificing performance, using SEJITS selective compilation techniques.
2) A new roofline performance model [28] for high-performance graph exploration, suitable for evaluating the performance of filtered semantic graph operations.
3) Experimental demonstration of excellent performance scaling to graphs with tens of millions of vertices and hundreds of millions of edges.
4) Demonstration of the generality of our approach by specializing two different graph algorithms: breadth-first search (BFS) and maximal independent set (MIS). In particular, the MIS algorithm requires multiple programmer-defined semiring operations beyond the defaults that are provided by KDT.

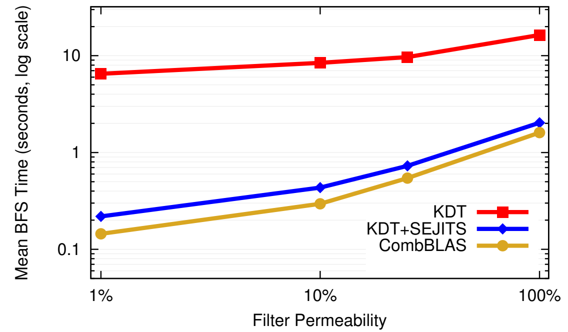


Fig. 2. Performance of a filtered BFS query, comparing three methods of implementing custom semiring operations and on-the-fly filters. The vertical axis is running time in seconds on a log scale; lower is better. From top to bottom, the methods are: high-level Python filters and semiring operations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

Figure 2 summarizes the work implemented in this paper, by comparing the performance of three on-the-fly filtering implementations on a breadth-first search query in a graph with 4 million vertices and 64 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 [21] graph of scale 22. The top, red, line is the method implemented in the current release v0.2 of KDT [2], with filters and semiring operations implemented as Python callbacks. The second, blue, line is our new KDT+SEJITS implementation where filters and semiring operations implemented in our DSLs are specialized using SEJITS. This new implementation shows minimal overhead and comes very close to the performance of native Combinatorial BLAS, which is in the third, gold, line.

The rest of the paper is organized as follows. Section II gives background on the graph-analytical systems our work targets and builds upon. Section III is the technical heart of the paper, which describes how we meet performance challenges by using selective, embedded, just-in-time specialization. Section IV 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 (that is, the percentage of edges that pass the filter). Section V gives details about the experimental setting and Section VI presents our experimental results. We survey related work in Section VII. Section VIII gives our conclusions and some remarks on future directions and problems.

A preliminary version of this paper appeared as a 2-page abstract at PACT [6]. This version is substantially different with a generalized specialization approach that works with both semirings and filters. It includes the MIS algorithm for which the bottleneck was the semiring translation, and a much wider set of experiments.

II. Background

A. Filters as scalar semiring operations

In this section, we show 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.

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 [17], 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 3 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.

```
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
}
```

Fig. 3. An example of a filtered scalar semiring operation in Combinatorial BLAS. This multiply operation only traverses edges that represent a retweet before June 30.

B. KDT Filters in Python

This subsection describes the high-level filtering facility 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.

The Knowledge Discovery Toolbox [23], [24] is a flexible open-source toolkit for complex graph algorithms on high-performance parallel computers. KDT targets two classes of users: domain-expert analysts who are not graph experts, who use KDT by invoking existing routines from Python, and graph-algorithm developers who write Python code that invokes and composes KDT computational primitives. These primitives are supplied by the Combinatorial BLAS [7].

Filter semantics: In KDT, any graph algorithm can be performed in conjunction with an edge filter. A filter is a unary predicate that returns true if the edge is to be considered, or false if it should be ignored. KDT users write filter predicates as Python functions or lambda expressions of one input that return a boolean value; Figure 4 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. Instead, the filtering occurs in the low-level primitives. Our design allows all current and future KDT algorithms to support filters without additional effort on the part of algorithm designers.

It is possible in KDT to add multiple filters to a 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. Multiple filter support allows both end users and algorithm designers to use filters for their own purposes.

Filtering approaches: KDT supports two approaches for filtering semantic graphs:

- *Materializing filter:* When a filter is placed on a graph (or matrix or vector), a copy is made that includes only 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 called. Roughly speaking, every primitive operation

```
# 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 for on−the−fly filtering

# perform some operations or queries on G

G.delEFilter(earlyRetweetsOnly)
```

Fig. 4. Adding and removing an edge filter in KDT, with or without materialization.

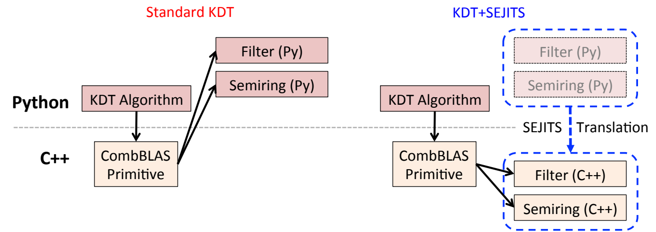


Fig. 5. **Left:** Calling process for filter and semiring operations in KDT. For each edge, the C++ infrastructure must upcall into Python to execute the callback. **Right:** Using our DSLs, the C++ infrastructure calls the translated version of the operation, eliminating the upcall overhead.

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 running 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 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.

A key focus of this paper is on-the-fly filtering and making it more efficient. Our experiments demonstrate that materializing the subgraph can take as much as 18 times the time of performing a single BFS on the real twitter dataset, when using 36 cores of Intel Xeon E7-8870.

Implementation details: Filtering a semiring operation requires the semiring scalar multiply to be able to return “null”, in the sense that the result should be the same as if the multiply never occurred. In semiring terms, the multiply operation must return the semiring's additive identity (SAID for short). CombBLAS treats SAID the same as any other value. However, CombBLAS uses a sparse data structure to represent the graph as an adjacency matrix—and, formally speaking, SAID is the implicit value of any matrix entry 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 [13], and differs from the convention in the CSparse sparse matrix package [9]. 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. It is crucial for usability to allow filters to be ignorant of the semiring they are applied to, therefore returning a SAID needs to be an out-of-band signal. We pair each basic semiring scalar operation with a `returnedSAID()` predicate which is called after the scalar operation. We use a predicate instead of a flag because the predicate can be optimized out by the compiler for unfiltered operations.

The result is a clean implementation of on-the-fly filters: filtering a semiring simply requires a shim in the `multiply()` function that causes `returnedSAID()` to return true if the value (edge) is filtered out; otherwise the semiring's callback is called and its value returned.

III. SEJITS Translation of Filters and Semiring Operations

In order to mitigate the slowdown caused by defining semirings and filters in Python, which results in a serialized upcall into Python for each operation, we use the Selective Embedded Just-In-Time Specialization (SEJITS) approach [8]. We define embedded DSLs for semiring and filter operations which are subsets of Python. As shown in Figure 5, callbacks written in these DSLs are translated at runtime to C++ to eliminate performance penalties while still allowing users the flexibility to specify filters and semirings in Python. We use the Asp[1] framework to implement our DSLs.

In the usual KDT case, semiring operations and filters are written as simple Python functions. Since KDT uses the Combinatorial BLAS at the low level to perform graph operations, each operation at the Combinatorial BLAS level (whether semiring operation or filtering) incurs one or more upcalls into Python per each vertex or edge involved.

We allow users to write their filters and semirings in our embedded DSLs. The languages are defined as proper subsets of Python with normal Python syntax, but they restrict the kinds of operations and constructs that users can utilize in filters and semiring operations. At instantiation, source code of filters and semirings 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.

Although KDT is our target platform in this work, our specialization approach can be used to accelerate other graph processing systems with similar performance challenges. In the next sections, we define our domain-specific languages and show several examples of using them from Python.

[1] Asp is SEJITS for Python, http://sejits.com

```
class MyFilter(PcbFilter):
  def __init__(self, ts):
    self.ts = ts
  def __call__(self, e):
    # if it is a retweet edge
    if (e.isRetweet and
        # and it is before our initialized timestamp
        e.latest < self.ts):
      return True
    else:
      return False
```

Fig. 6. Example of an edge filter that the translation system can convert from Python into fast C++ code. Note that the timestamp in question is passed in at filter instantiation time.

A. Python Syntax for the DSLs

We choose to implement two separate DSLs to clearly express and restrict the kinds of computations that can be done with each; for example, filters require boolean return values, while semiring operations require return values that are one of the vertex or edge types. Separating out the languages and their forms allows us to more easily ensure correctness of each.

Consider the filter embedded DSL. Informally, we specify the language by stating what a filter can do: namely, a filter takes in one input (whose type is pre-defined), 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 use Python's usual interface for callable objects, requiring the class define a function `__call__`.

Binary operations used in semirings and other operations in KDT are similarly defined, but must inherit from the `PcbFunction` class and must return one of the inputs or a numeric value that corresponds to the KDT built-in numeric type. Binary predicates resemble filters but accept two arguments. Future work will extend this to allow creation of new edge and vertex objects, but even with these restrictions a large majority of KDT graph algorithms can be expressed via our embedded DSL.

The example KDT filter from Figure 4 is presented in the filter embedded DSL syntax in Figure 6. It defines a fully-valid Python class that can be translated into C++ since it only uses constructs that are part of our restricted subset of Python.

B. Translating User-Defined Filters and Semiring Operations

In the Asp framework for SEJITS embedded DSLs, the most important mechanism for ensuring correct translations is to create an intermediate representation, called the *semantic model*, which defines the semantics of valid translatable objects. AST nodes from parsing Python are translated into this intermediate form as a first step of translation, and most of the logic for checking whether the definition is translatable is executed in this first phase. To be clear, this representation is not the syntax of a language, but rather is the intermediate state that defines semantics based on user-supplied Python syntax.

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

Expr = Constant | Identifier | BinaryOp | BoolExpr

Identifier(name=types.StringType)

BoolExpr = BoolConstant | IfExp | Attribute | BoolReturn |
           Compare | BoolOp

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

BoolOp(op=(ast.And | ast.Or | ast.Not), operands=BoolExpr*)
    check assert len(self.operands)<=2

Constant(value = types.IntType | types.FloatType)

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

BoolConstant(value = types.BooleanType)

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

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

BoolReturn(value = BoolExpr)
```

Fig. 7. Semantic Model for KDT filters using SEJITS. The semantic model for semiring operations is similar, but instead of enforcing boolean return values, enforces that the returned data item be of one of the input return types.

In filters and semirings, the user may wish to inspect fields of the input data types, do comparisons, and perhaps perform arithmetic with fields. Consequently our semantic model allows these operations.

On the other hand, we want to (as much as possible) prevent users from writing code that does not conform to our assumptions; although we could use analysis for this, it is much simpler to construct the languages in a manner that prevents users from writing non-conformant code in either embedded DSL. If the filter or semiring operation does not fit into our language, we run it in the usual fashion, by doing upcalls into pure Python, after outputting a warning. Thus, if the user writes their code correctly, they achieve fast performance, otherwise the user experience is no worse than before— the code still runs, just not at fast speed.

The semantic model for filters is shown in Figure 7. We have defined it to make it easy to write filters and operations 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, for filters, we require that the return be provably a boolean (by forcing the BoolReturn node to have a boolean body), and that there is a single input. The semantic model for semiring operations (not shown for space reasons) is similar, but instead of enforcing boolean return values, it ensures the returned item is one of the inputs or an elemental type understood by KDT.

We define tree transformations that dictate how Python AST nodes are translated into semantic model nodes. For example, the Python function definition for `__call__` is translated into a UnaryPredicate node in the case of the filter embedded

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

TABLE I
OVERHEADS OF USING THE FILTERING DSL.

DSL.

After the code is translated into instances of the semantic model, the rest of the translation is straightforward, utilizing Asp's infrastructure for converting semantic models into back-end code. For many of these transformations, defaults built into Asp are sufficient; for example, we leverage the default translation for constant numbers and therefore do not need to define the transform. The end result of conversion is source code containing the function in a private namespace plus some glue code, described in the next section. This source is passed to CodePy, which compiles it into a small dynamic link library that is then automatically loaded into the running interpreter.

C. Implementation in C++

We modify the C++ portion of KDT's callback mechanism which is based on pointers to Python functions. We add an additional 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. We similarly modify KDT's C++ function objects used for binary operations, which are used to implement semirings. For both kinds of objects, the functions or filters are type-specialized using user-provided information. Future refinements will allow inferred type-specialization.

Compared to Combinatorial BLAS, at runtime we have additional sources of overhead relating to the null check and function pointer call into a shared library, which usually is more expensive than a plain function call. However, these costs are trivial relative to the non-translated KDT machinery, particularly compared to the penalty of upcalling into Python.

Overheads of code generation are shown in Table I for 36 processors on the Intel Xeon E7-8870. On first running using a particular specialized operation, the DSL infrastructure translates and compiles it 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 due to CodePy's built-in caching support.

IV. A ROOFLINE MODEL OF BFS

In this section, we extend the Roofline model [28] 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 [26] (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 and semiring 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 amortize any memory system related effects to get the in-core compute limits. We define *permeability* as the percentage of edges that pass the filter. The compute limit decreases with increasing permeability because two operations need to be done for an edge that passes the filter, as opposed to one operation for an edge that does not.

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* 8 byte 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 micro-benchmark 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

TABLE II
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 III
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

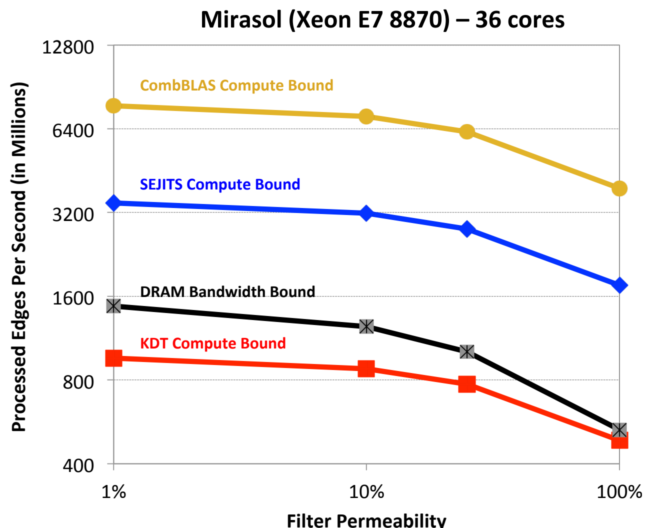


Fig. 8. Roofline-inspired performance model for filtered BFS computations. Performance bounds arise from bandwidth, CombBLAS, KDT, or KDT+SEJITS filter performance, and filter success rate.

statistics from one of the synthetically generated R-MAT graphs that are used in our experiments. These statistics are summarized in Table II. 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 V), in Table III. Combining Tables II and III, 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 8 presents the resultant Roofline-inspired performance model for Mirasol. The plots are upper bounds on the best performance achievable, which also take into account the caching of Python objects. The underlying implementation might incur additional overheads. 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. MPI buffers are also not taken into account.

The Roofline model selects ceilings by optimization, and bounds performance by their minimum. We select a filter implementation (pure Python KDT, KDT+SEJITS, or CombBLAS) and look for the minimum between that filter implementation's limit and the weighted DRAM bandwidth limit.

We observe a pure Python KDT filter will be the bottleneck in a BFS computation as it cannot sustain performance (edges per second) at the rate the processor can move edges on-chip. Conversely, the DRAM bandwidth performance limit is about 5× 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 crucial observation that explains why KDT+SEJITS performance is so close to CombBLAS performance in practice (as shown later in Section V) even though its in-core performance is about 2.3× slower.

V. EXPERIMENTAL DESIGN

This section describes graph algorithms used in our experiments, the benchmark matrices we used to test the algorithms, and the machines on which we ran our tests. KDT version 0.3 is enabled with the SEJITS techniques described in this paper, and is freely available at http://kdt.sourceforge.net.

A. Algorithms Considered

Our first algorithm is a filtered graph traversal. Given a vertex of interest, it determines the number of hops required to reach each other vertex using only retweet edges timestamped earlier than a given date. 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 (BFS) in the graph that ignores edges that do not pass the filter.

Our second query is to find the maximal independent set (MIS) of this graph. MIS finds a subset of vertices such that no two members of the subset are connected to each other and all other vertices outside MIS are connected to at least one member of the MIS. Since MIS is defined on an undirected graph, we first ignore edge directions, then we execute Luby's randomized parallel algorithm [22] implemented in KDT. The filter is the same as in the first query.

B. Test Data Sets

We experiment both with synthetically-generated graphs and those that are based on real data sets. Our BFS runs using the synthetic data are generated based on the R-MAT [21] model that can generate graphs with a highly skewed degree distribution. An R-MAT graph of scale N has 2^N vertices and approximately $edgefactor \cdot 2^N$ edges. In our tests, our $edgefactor$ is 16, and our R-MAT seed parameters a, b, c, and d are $0.59, 0.19, 0.19, 0.05$ respectively. After generating

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

Fig. 9. The edge data structure used for the combined Twitter graph in C++

TABLE IV
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

this non-semantic (boolean) graph, the edge payloads are artificially introduced using a random number generator in a way that ensures target filter permeability. The edge type is the same as the Twitter edge type described in the next paragraph, to be consistent between experiments on real and synthetic data. Our MIS runs use Erdős-Rényi graphs [11] with an $edgefactor$ of 4 because MIS on R-MAT graphs complete in very few steps due to high coupling, barring us from performing meaningful performance analysis.

Our real data graphs are based on social network interactions, using anonymized Twitter data [18], [29]. In our Twitter graphs, edges can represent two different types of interactions. The first 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 created 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 from 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 sets of 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.

More details for these four different (small-huge) combined graphs are listed in Table IV. Unlike 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 V. Processed edge count includes both the edges that pass the filter and the edges that are filtered out.

TABLE V
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

C. Architectures

To evaluate our methodology, we examine graph analysis behavior on 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 realize a flat MPI programming modeling using 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 6-core chips, and so a node can be viewed as a four-chip compute configuration with strong NUMA properties. Each Opteron chip contains six super-scalar, 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[26] 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 KDT+SEJITS filters, since the on-the-fly compilation mechanism requires at least one MPI process be able to call the compilation toolchain.

VI. EXPERIMENTAL RESULTS

In this section we use [semiring implementation]/[filter implementation] notation to describe the various implementation combinations we compare. For example, Python/SEJITS means that only the filter is specialized with SEJITS but the semiring is in pure Python (not specialized).

A. Performance Effects of Permeability

Figure 10 shows the relative distributed-memory performance of four methods in performing breadth-first search

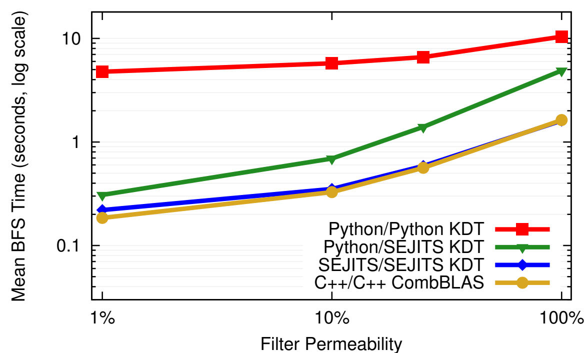


Fig. 10. Relative breadth-first search performance of four methods on synthetic data (R-MAT scale 25). Both axes are in log scale. The experiments are run using 24 nodes of Hopper, where each node has two 12-core AMD processors.

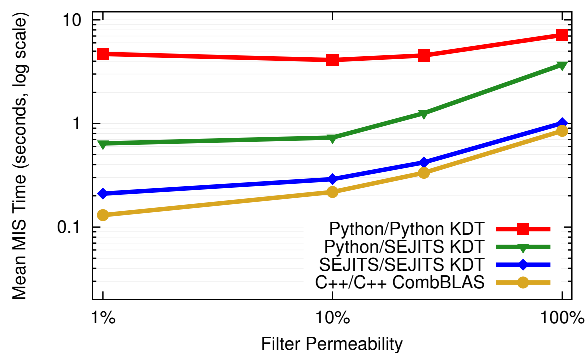


Fig. 11. Relative maximal independent set performance of four methods on synthetic data (Erdős-Rényi scale 22). y-axis uses a log scale. The runs use 36 cores of Intel Xeon E7-8870 processors.

on a graph with 32 million vertices and 512 million edges, with varying filter permeability. 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. The figure shows that the SEJITS/SEJITS KDT implementation (blue line) closely tracks CombBLAS performance (gold line), with the gap between it and the Python/Python KDT implementation (red line) shrinking as permeability increases. This is expected because as the permeability increases, both implementations approach the bandwidth bound regime as suggested by the roofline model in Section IV.

A similar but more condensed figure, showing the performance effects of permeability on Mirasol (Figure 2) exists in the introduction. There, KDT+SEJITS is the same as SEJITS/SEJITS. The effects of permeability on the MIS performance is shown in Figure 11 and reflect the BFS findings.

Since low permeability (1-10%) cases incur less memory traffic, Python overheads (KDT algorithms are implemented in Python) as well as the function pointer chasing of the SEJITS approach leave a noticeable overhead over CombBLAS. This is not the case for high-permeability filters where the extra memory traffic largely eliminates CombBLAS's edge, as observable from the shrinking gap between the blue and the gold lines in Figures 10 and 11 as permeability increases.

B. Performance Effects of Specialization

Since SEJITS specializes both the filter and the semiring operations, we discuss the effects of each specialization separately in this section.

All of the performance plots show that the performance of SEJITS/SEJITS (where both the filter and the semiring is specialized with SEJITS) is very close to the CombBLAS performance, showing that our specialization approach successfully bridges the performance gap between Python and the low-level CombBLAS. The Python/SEJITS case is typically slower than the SEJITS/SEJITS case, with the gap depending on the permeability. More selective filters make semiring specialization less relevant because as the permeability increases, more edges pass the filter and more semiring operations are performed, making Python based semiring operations the bottleneck. In the BFS case, shown in Figure 12, Python/SEJITS is $3-4\times$ slower than SEJITS/SEJITS when permeability is 100% due to the high number of semiring operations, but only $20-30\%$ slower when permeability is 1%. By going from 1% (Figure 12(a)) to 100% (Figure 12(d)), the green line separates from the other blue and gold lines and approaches the red line.

The performance of the MIS case, shown in Figure 13, is more sensitive to semiring translation, even for low permeabilities. The semiring operation in the MIS application is more computationally intensive, because each vertex needs to find its neighbor with the minimum label as opposed to just propagating its value as in the BFS case. Therefore, specializing semirings becomes more important in MIS.

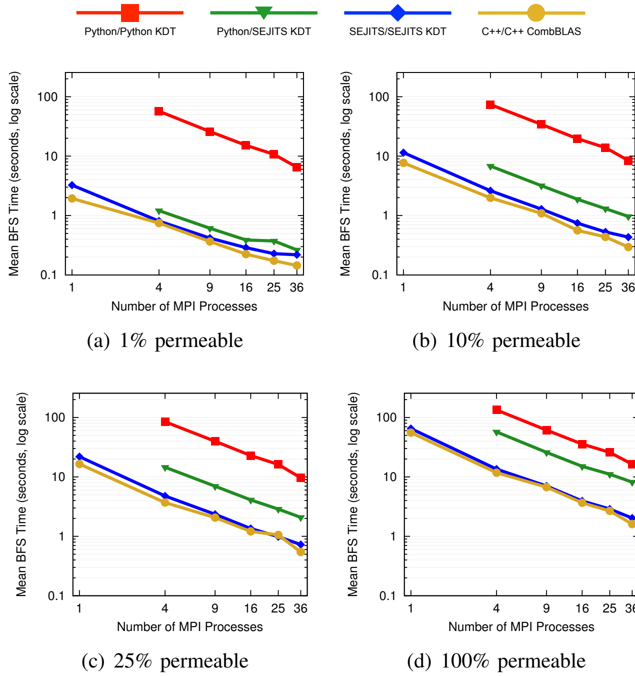
C. Parallel Scaling

Parallel scalability is key to enabling analysis of very large graphs in a reasonable amount of time. The parallel scaling of our approach is shown in Figures 12 and 13 for lower concurrencies on Mirasol. CombBLAS achieves remarkable scaling with increasing process counts, while SEJITS translated filters and semirings closely track its performance and scaling.

Parallel scaling studies of BFS at higher concurrencies is run on Hopper, using the scale 25 synthetic R-MAT data set. Figure 14 shows the comparative performance of KDT on-the-fly filters (Python/Python), SEJITS filter translation only (Python/SEJITS), SEJITS translation of both filters and semirings (SEJITS/SEJITS), and CombBLAS, with 1% and 100% filter permeability. Good scaling to thousands of cores makes it possible for domain scientists to utilize large-scale clusters and supercomputers.

D. Performance on the Real Data Set

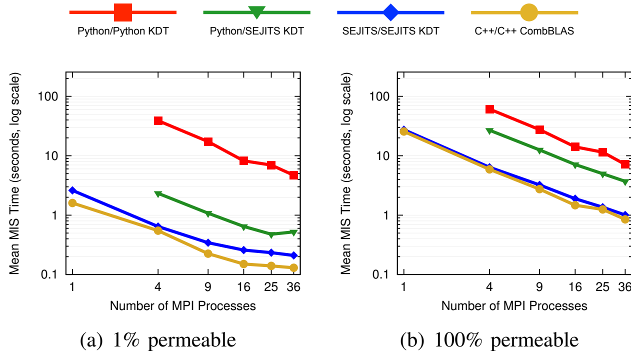
The filter used in the experiments with the Twitter data set considers only edges whose latest retweeting interaction happened before June 30, 2009, and is explained in detail in Section V-A. Figure 15 shows the relative performance of three systems in performing breadth-first search on real



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

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

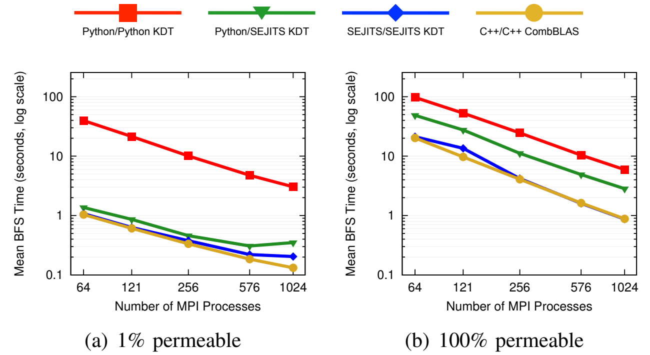
Fig. 12. Parallel 'strong scaling' results of filtered BFS on Mirasol, with varying filter permeability on a synthetic data set (R-MAT scale 22). Both axes are in log-scale, time is in seconds. Single core Python/Python and Python/SEJITS runs did not finish in a reasonable time to report.



(a) 1% permeable (b) 100% permeable

Fig. 13. Parallel 'strong scaling' results of filtered MIS on Mirasol, with varying filter permeability on a synthetic data set (Erdős-Rényi scale 22). Both axes are in log-scale, time is in seconds.

graphs that represent the twitter interaction data on Mirasol. We chose to present 16 core results because that is the concurrency in which this application performs best, beyond which synchronization costs start to dominate due to the large diameter of the graph after the filter is applied. Since the filter to semiring operations ratio is very high (on the order of 200 to 1000), SEJITS translation of the semiring operation does not change the running time. Therefore, we only include a single SEJITS line to avoid cluttering the plot. SEJITS/SEJITS performance is identical to the performance of CombBLAS in these data sets, showing that for real-world usage, our approach is as fast as the underlying high-performance library without forcing programmers to write low-level code.



(a) 1% permeable (b) 100% permeable

Fig. 14. 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.

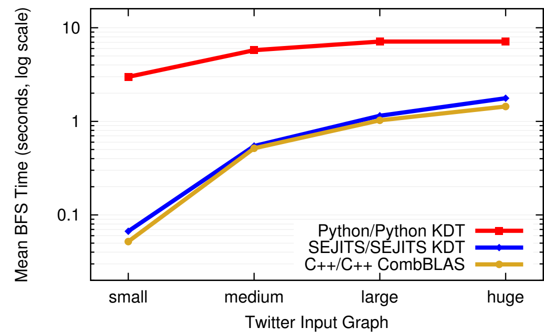


Fig. 15. Relative filtered breadth-first search performance of three methods on real Twitter data. The y-axis is in seconds on a log scale. The runs use 16 cores of Intel Xeon E7-8870 processors.

VII. Related Work

Graph Algorithm Packages: Pegasus [17] is a graph-analysis package that uses MapReduce [10] 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 [25], a distributed-computing graph API. Both Pegasus and Pregel require the application to be written in a relative low-level language (Java and C++, respectively) and neither supports filtering.

Other libraries for high-performance computation on large-scale graphs include the Parallel Boost Graph Library (PBGL) [14], the Combinatorial BLAS [7], Georgia Tech's SNAP [4], and the Multithreaded Graph Library [5]. These are all written in C/C++ and with the exception of the PBGL 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. PBGL also supports a FilteredGraph concept. Since PBGL is written in C++ with heavy use of template mechanisms, it is not conceptually simple to use by domain scientists. By contrast, our approach targets usability by specializing algorithms from a high-productivity language.

SPARQL [27] is a query language for Resource Description Framework (RDF) [19], which supports semantic graph database queries. The existing database engines that implement SPARQL and RDF handle filtering based queries efficiently but they are not as effective for running traversal based tightly-coupled graph computations scalably in parallel environments.

The closest previous work is Green Marl [15], a domain specific language (DSL) for small-world graph exploration that runs on GPUs and multicore CPUs without support for distributed machines (though such support is planned). Green Marl supports a very different programming model than KDT. In Green Marl, programmers iterate over nodes/edges or access them in specific traversal orders; work can be accomplished within a traversal or iteration step. KDT's underlying linear algebra abstraction allows graph algorithms to be implemented by customizing generic high-performance primitives of CombBLAS. In addition, the approach of Green Marl is to use an external DSL that has a different syntax and compiler than the rest of an application; KDT allows users to write their entire application in Python.

JIT Compilation of DSLs: Embedded DSLs [12] for domain-specific computations have a rich history, including DSLs that are compiled instead of interpreted [20]. 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 [8], and subsequently to a number of domains including linear algebra and Gaussian mixture modeling [16].

Finally, general JIT approaches for Python such as PyPy [3] does not offer the advantages of embedded DSLs such as domain-specific optimizations and the lack of need to perform detailed domain analysis.

VIII. 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 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 Selective Embedded Just-in-Time Specialization. In particular, we have shown that our embedded DSLs for filters and semirings enable Python code to achieve comparable performance to a pure C++ implementation. A roofline analysis shows that specialization enables filtering to move from being compute-bound to memory bandwidth- bound. We demonstrated our approach on both real-world data and large synthetic datasets. Our approach scales to graphs on the order of hundreds of millions of edges, and machines with thousands of processors, suggesting that our methodology can be applied to even more computationally intensive graph analysis tasks in the future.

In future work we will further generalize our DSL to support a larger subset of Python, as well as expand SEJITS support beyond filtering and semiring operations 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.

Acknowledgments

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 (Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the 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 DOE Office of Science 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. The authors from UC Santa Barbara were supported in part by NSF grant CNS-0709385 by a contract from Intel Corp., by a gift from Microsoft Corp. and by the Center for Scientific Computing at UCSB under NSF Grant CNS-0960316.

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] PyPy. http://pypy.org, 2012.
[4] 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 *Proc. IEEE Int. Symposium on Parallel&Distributed Processing*, pages 1–12, 2008.
[5] J.W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. In *Workshop on Multithreaded Architectures and Applications*. IEEE, 2007.
[6] A. Buluç, A. Fox, J. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams. High-performance analysis of filtered semantic graphs. PACT '12, pages 1–2, 2012.
[7] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, 2011.
[8] 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 *PMEA*, 2009.
[9] T.A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2006.
[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. Symposium on Operating System Design and Implementation*, pages 137–149, Berkeley, CA, 2004. USENIX.
[11] Paul Erdős and Alfréd Rényi. On random graphs. *Publicationes Mathematicae*, 6(1):290–297, 1959.
[12] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.
[13] J.R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1992.

[14] 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.

[15] 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, 2012. ACM.

[16] 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 *PPoPP'12*, 2012.

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

[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proc. Int. Conference on World Wide Web*, pages 591–600, New York, NY, 2010. ACM.

[19] O. Lassila and R.R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, Feb 1999.

[20] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proc. Conference on Domain-Specific Languages*, DSL'99, pages 9–9, Berkeley, CA, 1999. USENIX.

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

[22] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proc. ACM symposium on Theory of computing*, STOC '85, pages 1–10, New York, NY, 1985. ACM.

[23] 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 *SDM'12*, pages 930–941, April 2012.

[24] A. Lugowski, A. Buluç, J. Gilbert, and S. Reinhardt. Scalable Complex Graph Analysis with the Knowledge Discovery Toolbox. In *Int. Conference on Acoustics, Speech, and Signal Processing*, 2012.

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

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

[27] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF (working draft). Technical report, W3C, March 2007.

[28] 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.

[29] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *Proc. ACM Int. Conference on Web search and Data Mining*, WSDM '11, pages 177–186, New York, NY, USA, 2011. ACM.