Parallel Processing of Filtered Queries in Attributed Semantic Graphs ☆

Adam Lugowski[a,*], Shoaib Kamil[b,*], Aydın Buluç[c,*], Samuel Williams[c], Erika Duriakova[d], Leonid Oliker[c], Armando Fox[e], John R. Gilbert[a]

[a]*Dept. of Computer Science, University of California, Santa Barbara, USA*
[b]*CSAIL, Massachusetts Institute of Technology, Cambridge, USA*
[c]*CRD, Lawrence Berkeley National Laboratory, Berkeley, USA*
[d]*School of Computer Science and Informatics, University College Dublin, Ireland*
[e]*EECS Dept, University of California, Berkeley, USA*

Abstract

Execution of complex analytic queries on massive semantic graphs is a challenging problem in big-data analytics that requires high-performance parallel computing. In a semantic graph, vertices and edges carry *attributes* of various types and the analytic queries 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. Previous investigations have developed Knowledge Discovery Toolbox (KDT), a sophisticated a Python library for parallel graph computations. In KDT, the user can write custom graph algorithms by specifying operations between edges and vertices (*semiring operations*). The user can also customize existing graph algorithms by writing filters. Although the high-level language for this customization enables domain scientists to productively express their graph analytics requirements, the customized queries perform poorly due to the overhead of having to call into the Python virtual machine for each vertex and edge.

In this work, we use the Selective Embedded Just-In-Time 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 that our approach combines the benefits of programming in a high-level language with executing in a low-level parallel environment. We increase the system's flexibility by developing techniques that provide users with the ability to define new vertex and edge types from Python. We also present a new Roofline model for graph traversals and show that we achieve performance that is significantly closer to the bounds suggested by the Roofline. Finally, to further understand the complex interaction with the underlying architecture, we present an analysis using performance counters that quantifies the improvement in hardware behavior in the context our SEJITS methodology. 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 with hundreds of millions of edges and scaling to thousands of processors for graphs.

1. 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.

In a semantic graph, edges and/or vertices are labeled with *attributes* that might represent 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 [32] 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 [13]. Combinatorial BLAS uses broad definitions of matrix and vector operations. The user can define custom callbacks to override the 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 produce ("materialize") a new filtered graph as a temporary 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++and incur a significant productivity hit. Conversely, filters written at the KDT level, as predicate callbacks in Python, are productive, but incur a significant performance penalty.

[☆]This paper is the extended version of the conference paper "High-productivity and high-performance analysis of filtered semantic graphs" that was presented at the 2013 IEEE International Parallel & Distributed Processing Symposium.

[*]Corresponding authors

Email addresses: alugowski@cs.ucsb.edu (Adam Lugowski), skamil@mit.edu (Shoaib Kamil), abuluc@lbl.gov (Aydın Buluç)

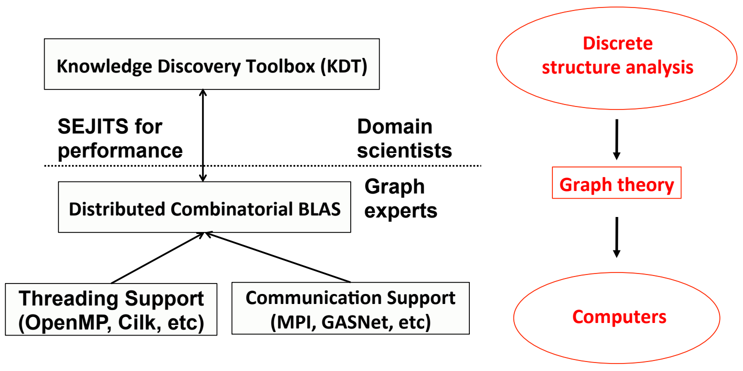

Knowledge Discovery Toolbox (KDT)
SEJITS for performance
Domain scientists
Distributed Combinatorial BLAS
Graph experts
Threading Support (OpenMP, Cilk, etc)
Communication Support (MPI, GASNet, etc)
Discrete structure analysis
Graph theory
Computers

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.

Our solution to this challenge is to apply Selective Just-In-Time Specialization techniques from the SEJITS approach [15]. 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-specific nature. On the other hand, our use of existing SEJITS infrastructure preserves the high-level nature of expressing computations in Python without forcing users to write C++code.

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.

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 [41] 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.

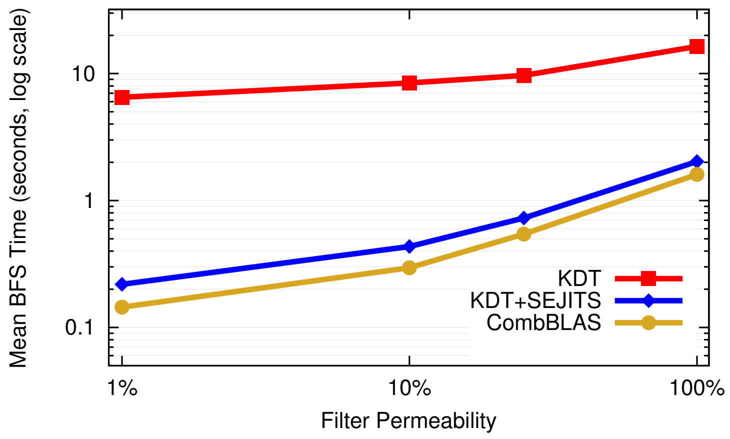


Figure 2: Performance of a filtered BFS query, comparing three methods of implementing custom semiring operations and on-the-fly filters. The vertical axis is running time in seconds on a log scale; lower is better. From top to bottom, the methods are: high-level Python filters and semiring operations 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 [30] graph of scale 22. The top, red, line is the method implemented in the current release v0.2 of KDT [4], 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 2 gives background on the graph-analytical systems our work targets and builds upon. Section 3 is the technical heart of the paper, which describes how we meet performance challenges by using selective, embedded, just-in-time specialization. Section 4 presents Python-defined objects that enable the user to declare their attribute types directly in Python, enabling a much board set of applications. 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 (that is, the percentage of edges that pass the filter). Section 5 gives details about the experimental setting and Section 7 presents our experimental results. In Section 8, we precisely analyze the performance implications of selective just-in translation using hardware performance counters. We survey related work in Section 9. Section 10 gives our conclusions and some remarks on future directions and problems. This paper expands on work first published as a conference paper at IPDPS [11].

2. Background

Running Example: 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 pseudocode for the full BFS implementation is given in Algorithm 1. This is a classical top-down BFS as opposed to the recently developed direction-optimizing algorithm that incorporates a bottom-up step [8, 9].

Algorithm 1 Pseudocode of breadth-first search algorithm used in our running example.

```
Input: Graph $G$ with transposed adjacency matrix $G.edges$ and $root$
Output: BFS parent vector $parents$
  $parents \leftarrow$ dense vector length $nvert(G)$, initialized to $-1$
  $frontier \leftarrow$ empty sparse vector length $nvert(G)$
  $parents[root] \leftarrow root$                               ▷ The root is its own parent.
  $frontier[root] \leftarrow root$
  while $frontier$ is not empty do
    $frontier[i] \leftarrow i$
    $frontier \leftarrow G.edges$.SpMV($frontier$, semiring=$SR$)
    prune $frontier[i]$ if $parents[i] \neq -1$           ▷ Remove already discovered vertices from the frontier.
    for all non-null $frontier[i]$ do     ▷ Update the parent vector with vertices discovered in this iteration.
      $parents[i] = frontier[i]$
```

2.1. 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. A semiring consists of a set of 'scalars', and two operations called 'addition' and 'multiplication'. The semiring additive identity (SAID for short) is also the multiplicative annihilator. The addition operation is commutative, and both multiplication and addition are associative. Speaking generally about graph algorithms, the 'scalars' are the edge and vertex data (attributes), 'multiplication' determines how the data on a sequence of edges are combined to represent a path, and 'addition' determines how to combine two or more parallel paths. The scalar multiply function is called

```
struct TwitterBFSSemiring
{
    ParentType multiply( const TwitterEdge & arg1, const ParentType & arg2)
    {
        if (arg1.isRetweet() && arg1.latest(sincedate))
            return arg2;  // unfiltered multiply returns normal value
        else
            return ParentType(); // filtered multiply yields SAID
    }
    ParentType add(const ParentType & arg1, const ParentType & arg2)
    {
        return ((arg2 == ParentType()) ? arg1: arg2); // select non-SAID
    }
    time_t sincedate = stringtotime("2009/06/30");
}
```

Figure 3: An example of a filtered scalar semiring operation in Combinatorial BLAS. This semiring would be used in the SpMV primitive in Algorithm 1. The multiply operation only traverses edges that represent a retweet before June 30, and the add operation returns one of the operands that is not SAID (if any).

```
# define the semiring
class select2nd(kdt.KDTBinaryFunction):
    def __call__(self, x, y):
        return y

SR = kdt.sr(select2nd(), select2nd())
```

Figure 4: An example semiring definition in KDT. This semiring would be used in the SpMV primitive in Algorithm 1. In KDT, the semiring and filter definitions are independent; a filtered semiring operation is achieved by using an unfiltered semiring operation on a graph that has had a filter added to it. A filter is added to a graph in Figure 5.

for each edge examination, making it a suitable candidate to embed the filtering logic. Two fundamental kernels in CombBLAS, sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpGEMM), both use semirings to explore the graph by expanding existing frontier(s) by a single hop.

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. Figure 3 shows the semiring 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 (in this example, the default constructed ParentType object is the additive identity), and the returned object is retained only if it does not match the SAID.

Filters written as semiring operations in C++ can have high performance because the filter itself is a local operation that uses only the data on one edge, and the number of calls to the filter operations is asymptotically the same as the minimum necessary calls to the semiring scalar multiply, which itself is called once per edge examination. The filtered multiply returns SAID if the predicate is not satisfied.

2.2. KDT Filters in Python

The Knowledge Discovery Toolbox 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, invoke the algorithms built by graph-algorithm developers. KDT algorithms are composed in Python from primitives supplied by the CombBLAS. This subsection describes the high-level filtering facility in KDT, in which filters are specified as simple Python predicates [33]. 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.

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.

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. To implement our running example we define the semiring in Figure 4. In Figure 5 we define an edge filter and add it to the graph.

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), the entire graph is traversed and 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 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;

```
# G is a kdt.DiGraph
class TwitterFilter(kdt.KDTUnaryPredicate):
  def __call__(self, e):
    return (e.count > 0 and e.latest < str_to_date("2009/06/30"))

earlyRetweetsOnly = TwitterFilter()

G.addEFilter(earlyRetweetsOnly)
G.e.materializeFilter() # omit this line for on-the-fly filtering

# perform some operations or queries on G, such as BFS

G.delEFilter(earlyRetweetsOnly)
```

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

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 largest of the real twitter datasets. In this comparison, both materialization (an embarrassingly parallel task) and the BFS are run in parallel 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 SAID. 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 [20], and differs from the convention in the CSparse sparse matrix package [16]. 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 small

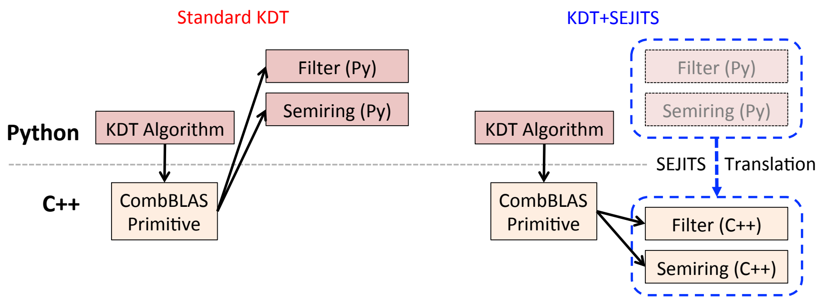


Figure 6: **Left:** Calling process for filter and semiring operations in KDT. For each edge, the C++ infrastructure must upcall into Python to execute the callback. **Right:** Using our DSLs, the C++ infrastructure calls the translated version of the operation, eliminating the upcall overhead.

adapter code in the semiring `multiply()` function that calls the filter predicate on both operands. If the predicate returns false for either operand then the adapter causes `returnedSAID()` to return true. Otherwise the semiring's callback is called and its value returned.

3. SEJITS Translation of Filters and Semiring Operations

Defining semirings and filters in Python results in one or more serialized upcalls from the low-level Combinatorial BLAS into Python for both semiring operations and filtering. In order to mitigate this slowdown, we use the Selective Embedded Just-In-Time Specialization (SEJITS) approach [15]. We define embedded DSLs for semiring and filter operations which are subsets of Python. As shown in Figure 6, 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.

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
```

Figure 7: 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.

3.1. 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. An alternative approach would build a single language but enforce restrictions using typechecking; we forgo the complexity involved in building a typechecker and instead enforce that filters are correct-by-construction. To avoid redundancy, we do share internal abstract syntax tree node types between the two DSLs.

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 and return a boolean.

The example KDT filter from Figure 5 is presented in the filter embedded DSL syntax in Figure 7. 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.

3.2. 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 se-

```
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)
```

Figure 8: Semantic Model for KDT filters using SEJITS.

mantics 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.

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 models are shown in Figures 8 and 9. 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 either a single input or two inputs (either UnaryPredicate or BinaryPredicate). The semantic model for semiring operations ensures

```
UnaryFunction(input=Identifier, body=Expr)

BinaryFunction(inputs=Identifier*, body=Expr)
   check assert len(self.inputs)==2

Expr = Constant
          | Identifier
          | BinaryOp
          | BoolConstant
          | IfExp
          | Attribute
          | FunctionReturn
          | Compare

Identifier(name=types.StringType)

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

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

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

BoolConstant(value = types.BooleanType)

IfExp(test=(Compare | Attribute | Identifier| BoolConstant |
           BinaryOp),
      body=Expr, orelse=Expr)

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

FunctionReturn(value = Expr)
```

Figure 9: Semantic Model for KDT binary and unary functions, used in semirings and related vector-vector operations.

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 DSL. Similarly, in the filter embedded DSL, the transformation checks whether the body of the return statement is provably a boolean and returns a BooleanReturn node.

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 backend 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.

Table 1: Overheads of using the filtering DSL.

	First Run	Subsequent
Codegen	$0.0545s$	$0.0s$
Compile	$4.21s$	$0.0s$
Import	$0.032s$	$0.032s$

3.3. 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 predicate, 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 1 on an Intel Xeon E7-8870 machine. On first run of a particular specialized operation, the DSL infrastructure translates it to C++ and compiles it; most of the time here is spent calling the external C++ compiler, which is not optimized for speed. CodePy's built-in caching support ensures that subsequent runs only incur the penalty of Python's `import` statement. On a multi-processor machine, only one process performs the compilation; the remaining ones load the cached version when that single compilation finishes.

4. Attributes defined in Python and exposed to C++

4.1. Motivation

The attribute types of vertices and edges should ideally be declared in Python, especially when the application requires several graphs with different edge and/or vertex datatypes. Consider the analysis of multi-modal brain networks (also known as connectomes). In this application, data from multiple modalities, such as fMRI, DTI, EEG, and PET, are collected for the patient's brain. Representing these data sources as graphs and using graph analysis has been instrumental in characterizing neurodegenerative diseases. The co-registration of these modalities requires the application to handle multiple graphs with different edge/vertex types. For example, the temporal and spacial resolution of fMRI and EEG data are incompatible [37], requiring different vertex types. Similarly, the voxel correlations in fMRI and DTI are defined differently, requiring different edge types. The ability to declare edge and vertex types dynamically in Python allows co-analysis of different brain networks and overcomes the limitations of using a single modality [28], and we plan to leverage our described methodology for forthcoming investigations of computational neuropathology.

4.2. Challenge

We wish to enable the user to declare their attribute types in Python. However, in order to obtain high-performance we must meet some CombBLAS and MPI requirements. CombBLAS's architecture requires that all elements of a matrix or vector must have the same type and size. These elements, or Python-Defined Objects (PDOs), must have the following properties:

- Self contained: no external references, object must be able to be copied by value (i.e. with `memcpy`).
- Object is declared and accessed in Python, memory is allocated in C++.
- Python-defined structure must be able to be operated on in C++

KDT 0.3 introduces just such a scheme. We declare a structure in Python that is then placed within a buffer of raw bytes. In other words, we turn C++ objects `Obj1` and `Obj2` into `void*` buffers which are in effect typecast to the Python-defined type at runtime.

4.3. Structure Declaration

Python's `ctypes` interface is used to call into C libraries. Since some C functions operate on `struct` datatypes, `ctypes` includes mechanisms to declare a C `struct` in Python. `ctypes` exposes C primitive datatypes such as `c_int` or `c_double` which can be composed together into a `struct` which is binary compatible with compiled C code on that particular system. We expose a subset of `ctypes`'s datatypes to the KDT user to use to declare a Python-Defined Object's data members.

Python access to the PDO is handled via `ctypes`'s hooks which enable the structure to behave like any Python object. Python operators can be declared using Python's standard operator definition machinery.

A simple example of a custom edge type is a PDO version of the structure in Figure 10, as follows:

```
class TwitterEdge(Structure):
    _fields_ = [("follower", c_bool),
                ("latest", c_uint64), # time_t
                ("count", c_short)]
```

4.4. Memory Handling

CombBLAS is not aware that it is working with a Python-Defined Object; instead, what it sees is a byte buffer of a fixed size. Therefore, all memory for PDOs is allocated by CombBLAS. Pointers to this memory are passed to the callbacks, which then use `ctypes`'s mechanisms to create a Python object backed by the CombBLAS buffer. The PDO is then accessible in the Python callback.

4.5. PDOs and SEJITS

For SEJITS to support the PDO it must be able to access the PDO's memory in the same way as the Python operations would. Luckily `ctypes` declares the structure in precisely such a way. We translate all PDO structs used by a specialized callback into C and add the declarations to the SEJITS-generated C++ module. The callback's parameters are the buffer objects, `Obj1` and/or `Obj2`. We add code to extract the buffers and typecast them to references to the particular structs that the buffers correspond to.

The rest of the C++ specialized callback can now operate on the buffer as if it were a struct.

4.6. Limitations

Our approach has some limitations, particularly that we can only support the intersection of Python and C++ language features. In particular, data members and their types must be declared ahead of time. The declaration is decidedly C-style, and any duck-typed definitions will be lost. The PDO must not contain any pointers or references.

CombBLAS requires that the datatypes must be copyable by value for MPI communication, so no copy constructors are called. In addition, the size of the buffers must be declared at compile time. Our scheme allows an unlimited number of different PDO types to be declared in a single program, but they all must fit into one of a handful different buffer sizes.

5. 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`.

5.1. Algorithms Considered

Our first algorithm is a filtered graph traversal. Given a vertex of interest, it determines the number of hops required to reach every other vertex using only those 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) on 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 [31] implemented in KDT. The filter is the same as in the first query.

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

Figure 10: The edge data structure used for the combined Twitter graph in C++

Table 2: 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

5.2. Test Data Sets

We evaluate our techniques on both algorithms using synthetically-generated graphs and those that are based on real data sets. Our BFS experiments using the synthetic data are generated based on the R-MAT [30] 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 $0.57, 0.19, 0.19, 0.05$ respectively. After generating this non-semantic (boolean) graph, edge payloads are artificially introduced where the timestamp values are generated using the Mersenne Twister pseudo-random number generator [35]. A simple threshold controls filter permeability. We use a fixed seed so that the experiments are reproducible and all codes work on the same problem. The edge type is the same as the Twitter edge type described in the next paragraph in order to be consistent between experiments on real and synthetic data. Our MIS experiments use Erdős-Rényi graphs [18] with an *edgefactor* of 4 because the MIS algorithm on R-MAT graphs completes in very few steps due to high coupling and would otherwise bar us from performing a meaningful performance analysis.

Our real data graphs are based on social network interactions, using anonymized Twitter data [26, 42]. In our Twitter graphs, edges can represent two different types of interactions. The first interaction is the "following" relationship where an edge from vertex v_i to v_j implies 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 implies 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

Table 3: 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

June, the *medium* dataset contains tweets from June and July, the *large* dataset contains tweets dated June through September, and finally the *huge* dataset contains all the tweets from June through 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 2. Unlike the synthetic data, the real twitter data is directed and we only report breadth-first searches 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 3. Processed edge count includes both the edges that pass the filter and the edges that are filtered out.

5.3. 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 at NERSC. 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[36] bandwidth is about 30 GB/s per socket. The machine has 256 GB of DDR3-1066 DRAM. We utilize 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 bandwidth of 12GB/s per chip. Each pair of compute nodes shares one Gemini network chip that 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 of this MPP do not contain a compiler. To remedy this, we ensured

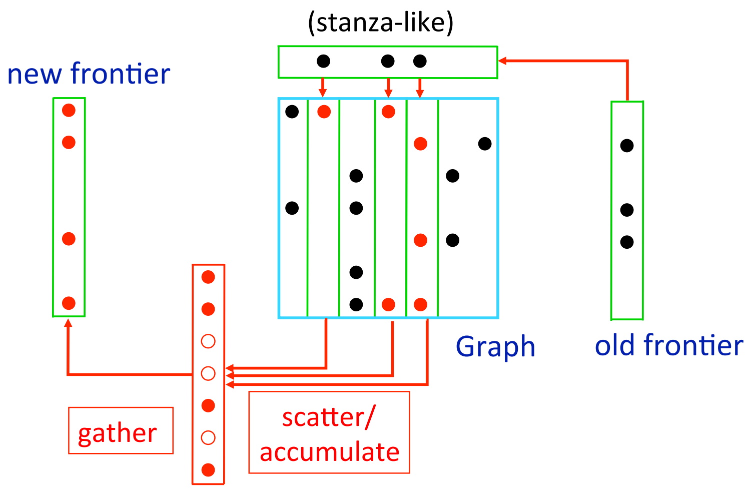


Figure 11: Memory access pattern of one BFS iteration. The graph is represented by the transpose of its sparse adjacency matrix. Each column in the matrix as well as each vector is stored in the compressed form of index-value pairs. In the case of frontier vectors, the pair represents (vertex index, parent's index).

that a compute node with access to the requisite compilers 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.

6. A Roofline model of BFS

The Roofline model [41] 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 and has been demonstrated to show that performance of many HPC kernels is well-correlated with STREAM bandwidth. Unfortunately, the traditional HPC application characteristics (massive parallelism, streaming memory access) and even metrics (flops per byte) are often antithetical to the computational challenges found in linear algebra-based graph algorithms. To remedy this, we extend the Roofline model to quantify the performance bounds of BFS as a function of optimization and filter success rate. Doing so allowed us to separate the effects of computation from data movement and express performance as a function of *Filter Permeability* — the percentage of edges that pass the filter — and thus explain the performance benefit of the technology demonstrated in this paper.

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 applying them to arrays that fit in each core's cache. We run the edge processing inner kernels 10,000 times (as opposed to once) to amortize any memory system related effects to get the in-core compute limits.

The compute limit decreases with increasing permeability because two operations must be performed for an edge that passes the filter as opposed to the 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. To do so, we must not only estimate data movement, but also effective bandwidth for each operation. In the following analysis, the indices are 8 bytes and the edge payload is 16 bytes. BFS exhibits three memory access patterns which are illustrated in Figure 11. First, there is a unit-stride *streaming* access pattern arising from access of the vertex pointers (this is amortized by degree) as well as the creation of the sparse output vector that acts as the new frontier (the gather step in Figure 11). 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 mean degree, due to two reasons. The first reason is the heavy-tailed degree distribution that is characteristic of many real world graph instances, which applies to both sequential and parallel settings regardless of the data decomposition. In heavy-tailed distributions, the median is smaller than mean. The second reason only applies to the parallel setting and it is due to per-processor subgraphs being sparser than the full graph for the 2D decomposition (also called hypersparsity [12]). The stanza related traffic 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 (the scatter/accumulate step in Figure 11) 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 which we expect to be quite different than STREAM, 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 (i.e. asymptotically the STREAM benchmark).

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 4. 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 5), in Table 5. Combining Tables 4 and 5, we calculate the average number of processed edges per

Table 4: Statistics about the filtered BFS runs on the R-MAT graph of Scale 23 (M: million)

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

Table 5: 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

second as a function of filter permeability by summing data movement time by type and inverting.

Figure 12 presents the resultant Roofline-inspired performance model for Mirasol. The plots are upper bounds on the achievable performance and also include the effects of 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. Neither access to MPI buffers nor MPI performance were 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 7) even though its in-core performance is about 2.6× slower.

This Roofline model serves as an excellent surrogate for the performance we observe in practice in Figure 2 and generally in Section 7. Specifically, it methodologically explains the smaller (≈40×) gap we observe between SEJITS and pure Python KDT performances for BFS as opposed to over 140× suggested by the in-core compute limits. The actual predicted performance difference is the gap between the DRAM bandwidth limit and the KDT in-core compute limit because the SEJITS Roofline is the lower of the bandwidth-bound and in-core compute-bound lines. Due to the aforementioned data movement effects that we did not account

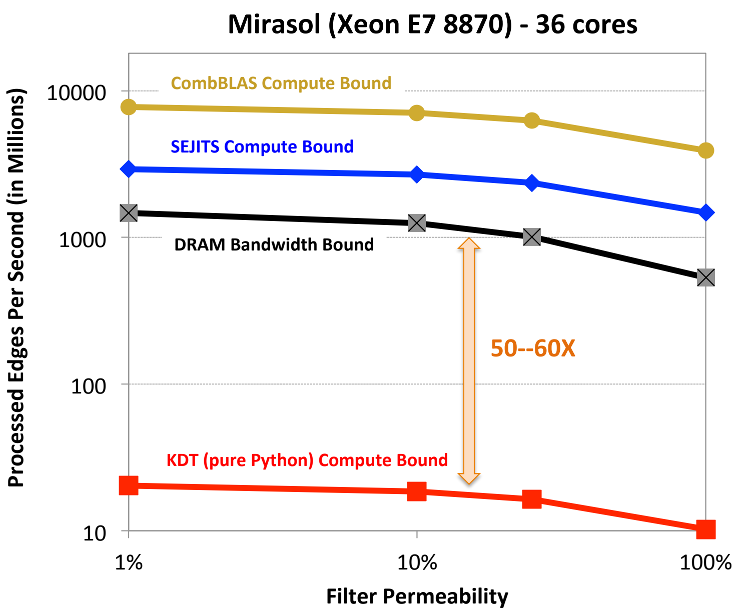


Figure 12: Roofline-inspired performance model for filtered BFS computations. Performance bounds arise from bandwidth, CombBLAS, KDT, or KDT+SEJITS filter performance, and filter success rate. The performance axis is in log-10 scale.

for (such as sorting and MPI buffers), the model suggests a slightly higher bandwidth-bound line, hence a slightly bigger gap than what is observed in practice.

7. 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).

7.1. Performance Effects of Permeability

Figure 13 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 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 6.

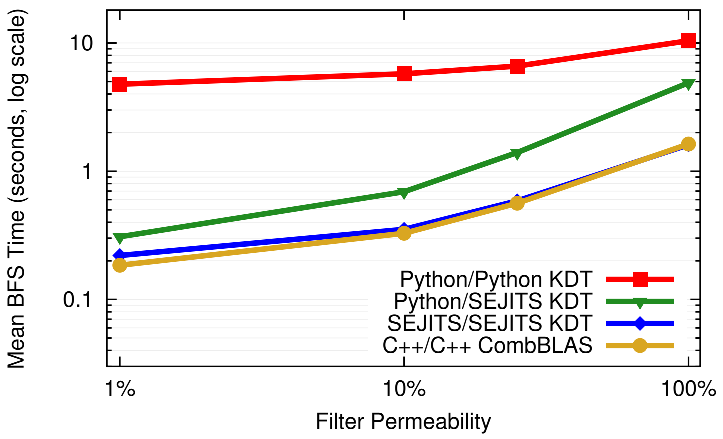


Figure 13: Relative breadth-first search performance of four methods on synthetic data (R-MAT scale 25). Both axes are in log scale. The experiments are run using 24 nodes of Hopper, where each node has two 12-core AMD processors. Time is mean of 16 BFS runs from different starting vertices. Notation: [semiring implementation]/[filter implementation].

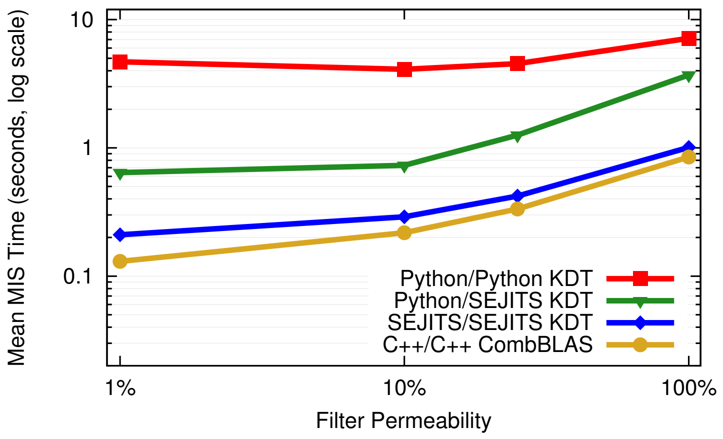
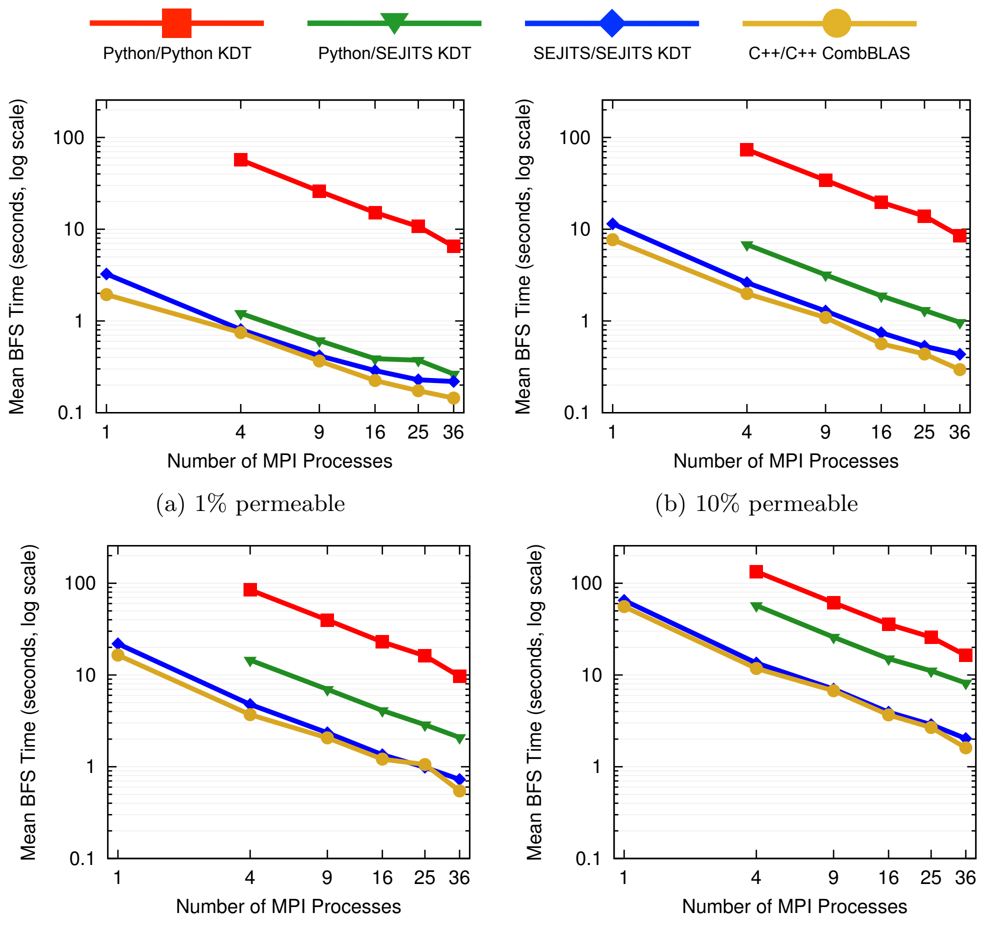


Figure 14: Relative maximal independent set performance of four methods on synthetic data (Erdős-Rényi scale 22). y-axis uses a log scale. The runs use 36 cores of Intel Xeon E7-8870 processors. Time is mean of 16 runs. Notation: [semiring implementation]/[filter implementation].

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 14 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 13 and 14 as permeability increases.

7.2. Performance Effects of Specialization



(a) 1% permeable

(b) 10% permeable

(c) 25% permeable

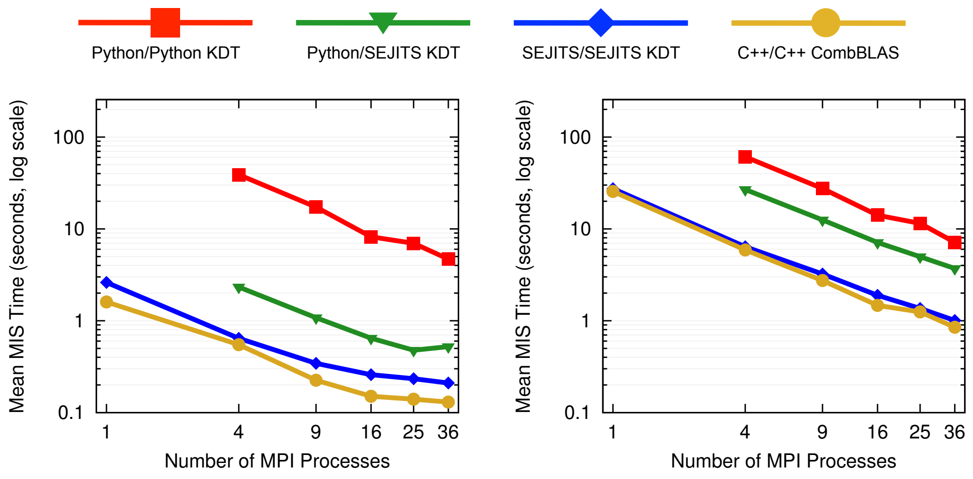
(d) 100% permeable

Figure 15: Parallel 'strong scaling' results of filtered BFS on Mirasol, with varying filter permeability on a synthetic data set (R-MAT scale 22). Both axes are in log-scale, time is in seconds (mean of 16 runs from different starting vertices). Single core Python/Python and Python/SEJITS runs did not finish in a reasonable time to report. Notation: [semiring implementation]/[filter implementation].

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 15, 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 15a) to 100% (Figure 15d), 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 16, 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.

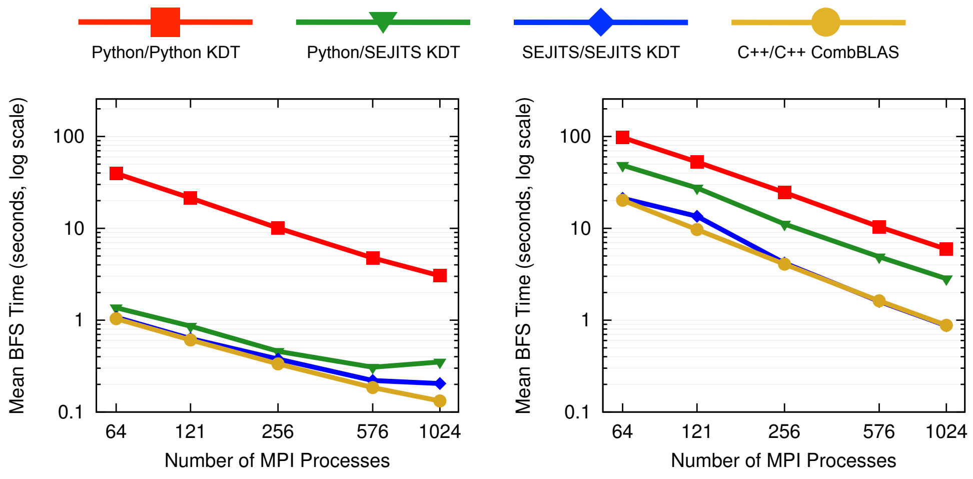


(a) 1% permeable (b) 100% permeable

Figure 16: 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 (mean of 16 runs). Notation: [semiring implementation]/[filter implementation].

7.3. *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 15 and 16 for lower concurrencies on Mirasol. CombBLAS



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

Figure 17: 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 (mean of 16 runs from different starting vertices). Notation: [semiring implementation]/[filter implementation].

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 17 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. The SEJITS/SEJITS result tracks CombBLAS closely, except for the largest core counts with 1% permeability. This difference is because the BFS time is so short that the small fixed overhead of importing the SEJITS-compiled filter predicates and semirings is not amortized.

Finally, we show weak scaling results on Hopper using 1% filter permeability (other cases experienced similar performance). In this run, shown in Figure 18, 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). The 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 a perfect flat line. However, these deviations are expected on a large scale BFS run and are experienced on similar architectures [14]. The results demonstrate that our SEJITS approach does not impede scalability to thousands of processors, compared to an high-performance library like CombBLAS.

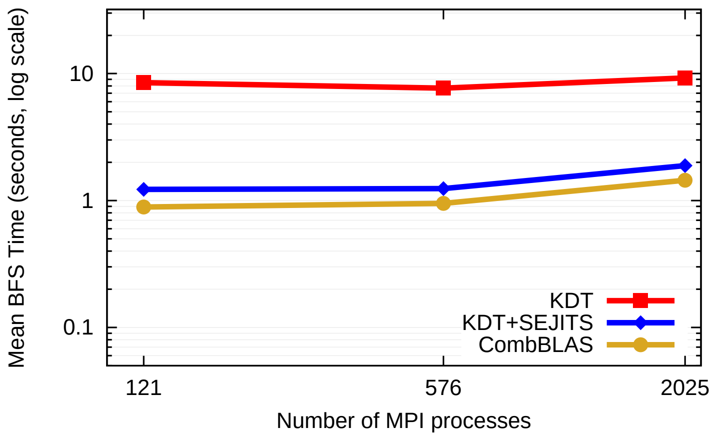


Figure 18: Parallel 'weak scaling' results of filtered BFS on Hopper, using 1% percent permeability. y-axis is in log scale, time is in seconds. From top to bottom, the methods are: high-level Python filters and semiring operations in KDT; high-level Python filters and semiring operations specialized at runtime by KDT+SEJITS; low-level C++ filters implemented as customized semiring operations and compiled into Combinatorial BLAS.

7.4. 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 5.1. Figure 19 shows the relative performance of three systems in performing breadth-first search on real 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.

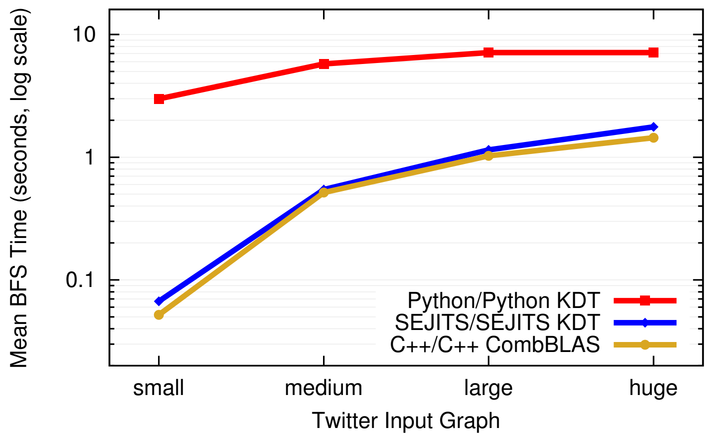


Figure 19: Relative filtered breadth-first search performance of three methods on real Twitter data. The y-axis is in seconds on a log scale. The runs use 16 cores of Intel Xeon E7-8870 processors.

8. Performance Results Introspection via Hardware Performance Counters

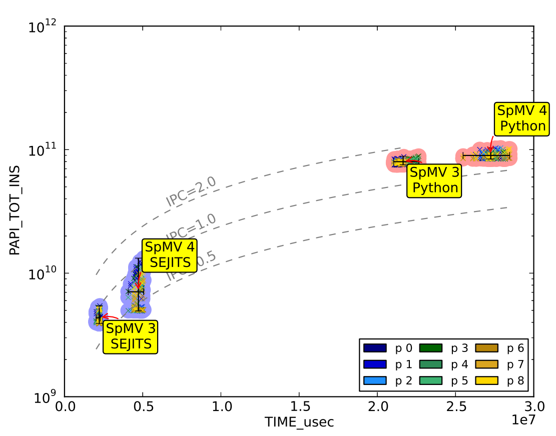
The Performance Application Programming Interface (PAPI) [5] library provides direct access to low-level performance counters. These counters can measure performance attributes of a particular program execution. For example, PAPI counters can be used to measure the total number of instructions executed, or the total number of cache misses (L1 or L2, data or instruction).

Our study incorporates several PAPI performance counters to gain a detailed analysis of the performance benefits of KDT+SEJITS over Python KDT. We are particularly interested in the `PAPI_TOT_INS` (total instructions completed), `PAPI_L1_ICM` (number of L1 instruction cache misses), `PAPI_L1_DCM` (number of L1 data cache misses), and `PAPI_L2_TCM` (number of L2 total cache misses). Additional experiments with L3 cache misses did not provide any additional insights that were not already captured in the L2 cache analysis.

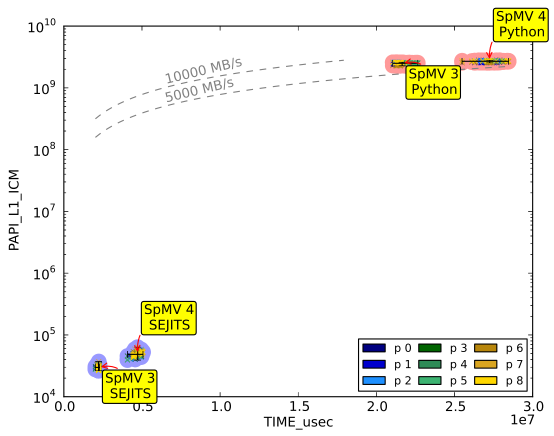
Table 6: PAPI measurements for 100% filter, showing (Time_usec) total time, (TOT_INS) total instructions, (L1_ICM) L1 instruction cache misses, (L1_DCM) L1 data cache misses, and (L2_TCM) total L2 misses. All values are the mean of 96 points (9 processes × 16 repeats). Figure 20 is a visual representation of this data.

	Time_usec	TOT_INS	L1_ICM	L1_DCM	L2_TCM
SpMV 3 Python	$2.16e+07$	$7.99e+10$	$2.50e+09$	$4.31e+08$	$8.94e+07$
SpMV 3 SEJITS	$2.22e+06$	$4.36e+09$	$2.98e+04$	$1.18e+08$	$6.91e+07$
SpMV 4 Python	$2.73e+07$	$8.97e+10$	$2.69e+09$	$6.98e+08$	$2.38e+08$
SpMV 4 SEJITS	$4.73e+06$	$7.07e+09$	$4.85e+04$	$2.54e+08$	$2.08e+08$
Other Python	$6.20e+04$	$3.13e+08$	$1.27e+06$	$2.90e+05$	$1.05e+05$
Other SEJITS	$4.39e+04$	$1.18e+08$	$3.93e+04$	$2.14e+05$	$1.00e+05$

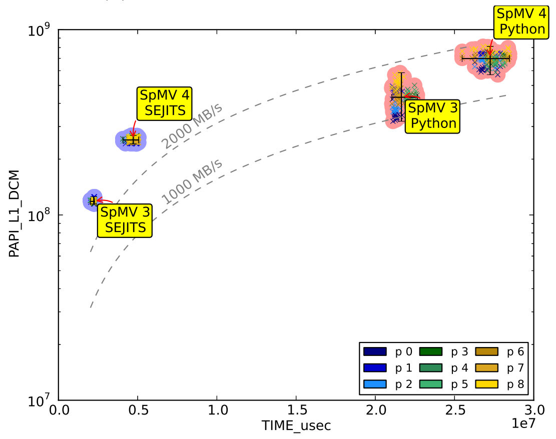
Performance counters were examined for our breadth-first search program on a scale 22 RMAT graph as described in Section 5, using both 10% and 100% permeable on-the-fly filters, repeating a BFS from a single



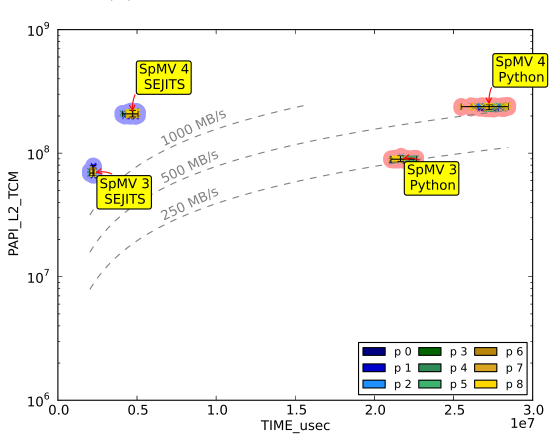
(a) Total Instructions Completed



(b) L1 Instruction Cache Misses



(c) L1 Data Cache Misses

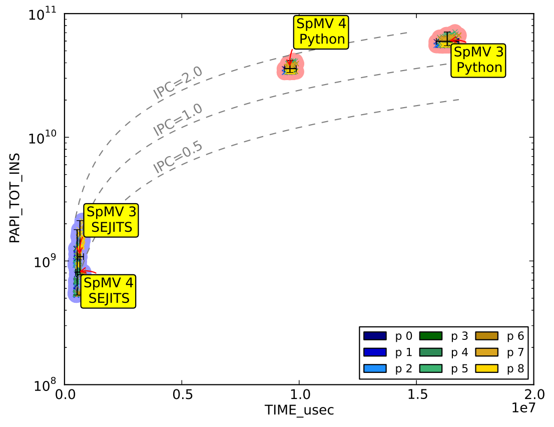


(d) L2 Total Cache Misses

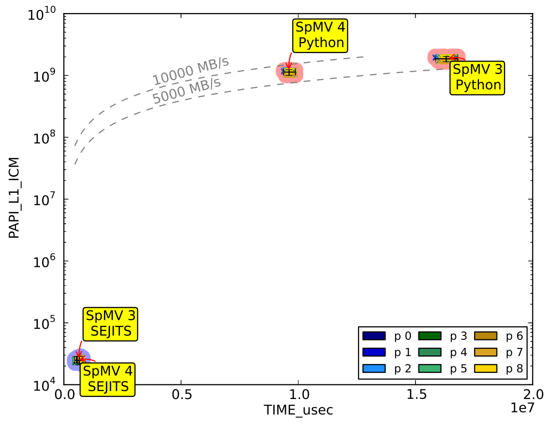
Figure 20: PAPI performance counters vs. time (in μs), showing (a) total instructions, (b) L1 instruction cache misses, (c) L1 data cache misses, and (d) total L2 misses. BFS on Scale 22 graph with 100% permeable filter, repeated 16 times from starting vertex 1726462. P=9 on Mirasol. Each point is a counter value for a single process in a single BFS iteration. Table 6 offers a summary of the same data in tabular form.

starting vertex 16 times. These tests were conducted on Mirasol using 9 MPI processes, ensuring that all processes are placed on a single socket. Note that each part of each BFS iteration is measured separately. The parts include the loop condition check, the SpMV, the frontier update and the parents vector update. Due to the small diameter of the input graph, nearly all the time is spent in two SpMV calls, during which the majority of the graph is explored.

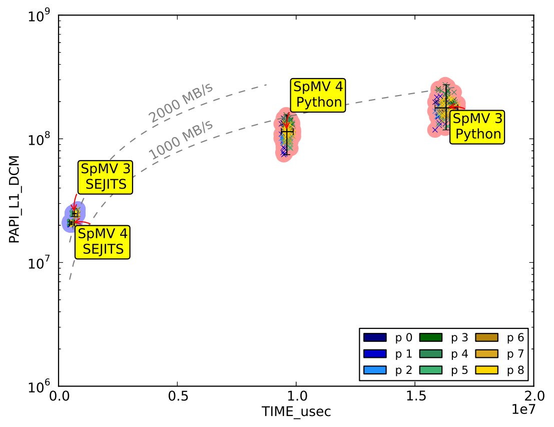
Figures 20 and 21 present the performance counters data relative to runtime for the 100% and 10% filters, respectively. Only the most time-consuming SpMV calls are presented to keep from cluttering the plots. Each MPI process is represented by a different color as suggested by the legend. For a given colored dot, each occurrence in these plots correspond to a different BFS exploration (out of 16 repeats), totaling 96 dots per operation. We clustered all the points corresponding to a particular operation into a point cloud



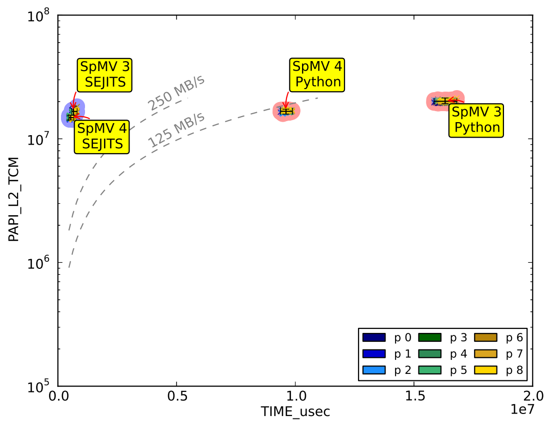
(a) Total Instructions Completed



(b) L1 Instruction Cache Misses



(c) L1 Data Cache Misses



(d) L2 Total Cache Misses

Figure 21: PAPI performance counters vs. time (in μs). BFS on Scale 22 graph with 10% permeable filter, repeated 16 times from starting vertex 1291427. P=9 on Mirasol. Each point is a counter value for a single process in a single BFS iteration. Table 7 offers a summary of the same data in tabular form.

for ease of visualization.

In addition, Table 6 and Table 7 provide the same information in tabular form for 100% and 10% filters, but only showing the mean of the 96 points. The tables also include an "Other" category that combines all overheads except the two SpMVs, which account for a small overall fraction of runtime (two orders of magnitude less time than SpMVs). Note that we shorten "KDT+SEJITS" to "SEJITS" and "Python/Python" to "Python" for brevity.

These figures clearly underscore the dramatic performance benefits of the SEJITS approach. In the 100% filter run (Figure 20), the SEJITS versions incur over an order of magnitude fewer total instruction completions, as well as four orders of magnitude fewer L1 instruction cache misses. The Python callbacks require a wrapper object, while their SEJITS counterparts do not. This results in a further half-order of magnitude fewer L1 data cache misses. The benefits are even larger for the 10% filter run, as shown in

Table 7: PAPI measurements for 10% filter, showing (Time_usec) total time, (TOT_INS) total instructions, (L1_ICM) L1 instruction cache misses, (L1_DCM) L1 data cache misses, and (L2_TCM) total L2 misses. All values are the mean of 96 points (9 processes × 16 repeats). Figure 21 is a visual representation of this data.

	Time_usec	TOT_INS	L1_ICM	L1_DCM	L2_TCM
SpMV 3 Python	$1.63e+07$	$5.97e+10$	$1.86e+09$	$1.78e+08$	$2.01e+07$
SpMV 3 SEJITS	$6.58e+05$	$1.08e+09$	$2.46e+04$	$2.48e+07$	$1.66e+07$
SpMV 4 Python	$9.61e+06$	$3.59e+10$	$1.13e+09$	$1.14e+08$	$1.67e+07$
SpMV 4 SEJITS	$5.39e+05$	$8.13e+08$	$2.47e+04$	$2.09e+07$	$1.48e+07$
Other Python	$4.32e+04$	$1.73e+08$	$3.03e+06$	$3.68e+05$	$8.57e+04$
Other SEJITS	$3.96e+04$	$1.03e+08$	$3.60e+04$	$1.67e+05$	$8.75e+04$

Figure 21. The L2 and L3 (not shown) total cache misses are nearly same for both approaches, indicating that the majority of the performance impact between SEJITS and pure Python approaches is captured within the L1 cache. This can be visually observed in the L2 total cache misses (d) plots where the SEJITS and Python clusters for SpMV 3 and SpMV 4 are on the same horizontal line.

The only performance cost for SEJITS is the JIT compilation, incurred on the first execution of a kernel which uses SEJITS callbacks. All our kernels use a SEJITS callback, which amounts to about 20 seconds for just the first iteration. Since the results of the JIT compilation are cached even between independent jobs, this cost is only paid the first time the script is run.

9. Related work

Graph Algorithm Packages

Pegasus [25] is a graph-analysis package that uses MapReduce [17] in a distributed-computing setting. Other cloud-based graph analysis systems include GPS [40], and Apache Hama [3], and Giraph [2]. Redekopp et al. [39] recently studied performance optimizations for such cloud-based graph platforms.

Pegasus [25], uses a generalized matrix-vector multiplication primitive called `GIM-V`, much like KDT's `SpMV`, to express vertex-centered computations that combine data from neighboring edges and vertices. In Pegasus, the semiring multiply is referred to as combine2 and the semiring add is referred to as combineAll, followed by an assign operation.

Powergraph [21] advocates a similar GAS (gather-apply-scatter) abstraction for graph-parallel computations. This style of programming is called "think like a vertex" in Pregel [34], a distributed-computing graph API. In traditional scientific computing terminology, these are all BLAS-2 level operations; none of these aforementioned systems currently include KDT's BLAS-3 level `SpGEMM` "friends of friends" primitive. BLAS-3 operations are higher level primitives that enable more optimizations and generally deliver superior performance. Both Pegasus and Powergraph 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) [22], the Combinatorial BLAS [13], Georgia Tech's SNAP [7], and the Multithreaded Graph Library (MTGL) [10]. These are all written in C/C++ and with the exception of the PBGL and MTGL 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 and MTGL provides generic filter support via visitor functions. PBGL also supports an explicit FilteredGraph concept. Since PBGL and MTGL are written in C++ with heavy use of template mechanisms, they are not conceptually simple to use by domain scientists. By contrast, our approach targets usability by specializing algorithms from a high-productivity language.

SPARQL [38] is a query language for Resource Description Framework (RDF) [27], 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 [23], 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 [19] for domain-specific computations have a rich history, including DSLs that are compiled instead of interpreted [29]. 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 [15], and subsequently to a number of domains including linear algebra and Gaussian mixture modeling [24].

Finally, general JIT approaches for Python such as PyPy [6] do not offer the advantages of embedded DSLs such as domain-specific optimizations and the lack of need to perform detailed domain analysis.

10. 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 impact of expressing filters in a high-level language like Python 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. In addition, we provide users with the ability to define new vertex and edge types from Python, yet still obtain the same high performance. A Roofline analysis shows that specialization enables filtering to move from being compute-bound to memory bandwidth-bound. Further performance counter-based analysis shows that the SEJITS performance gains are due to a combination of executing fewer instructions with the ability to avoid data movement for object wrappers during computation. 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 to machines with thousands of processors, suggesting that our methodology can be applied to even more computationally intensive graph analysis tasks in the future. Ultimately, the ability to both attain high performance and scale to thousands of cores for most cases makes it possible for domain scientists to efficiently utilize large-scale clusters and supercomputers.

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 is supported by the Computer Science Program, the Applied Mathematics Program, and the Early-Career Research Program within the Office of Science Advanced Scientific Computing Research of the U.S. Department of Energy under contract No. DE-AC02-05CH11231. 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. The authors from UC Santa Barbara were supported in part by NSF grant CNS-0709385 by a contract from Intel Corp. (#618442525-57661), by a gift from Microsoft Corp. by the Center for Scientific Computing at UCSB under NSF Grant CNS-0960316, and contract #8-482526701 from the DOE Office of Science.

References

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

[2] Apache Gigraph. http://giraph.apache.org, 2013.

[3] Apache Hama. http://hama.apache.org, 2013.

[4] Knowledge Discovery Toolbox. http://kdt.sourceforge.net, 2013.

[5] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/, 2013.

[6] PyPy. http://pypy.org, 2013.

[7] 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.

[8] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3):137–148, 2013.

[9] S. Beamer, A. Buluç, K. Asanovic, and D. Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1618–1627. IEEE Computer Society, 2013.

[10] 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.

[11] A. Buluç, E. Duriakova, A. Fox, J. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams. High-productivity and high-performance analysis of filtered semantic graphs. In *Proceedings of the IPDPS*. IEEE Computer Society, 2013.

[12] A. Buluç and J. R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *Proc. IPDPS*, April 2008.

[13] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, 2011.

[14] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. IEEE/ACM Conference on Supercomputing*, 2011.

[15] 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.

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

[17] 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.

[18] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6(1):290–297, 1959.

[19] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

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

[21] J E Gonzalez, Y Low, H Gu, D Bickson, and C Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] X Lei, X Ostwald, J Hu, C Qiu, C Porcaro, A P Bagshaw, and D Yao. Multimodal functional network connectivity: an EEG-fMRI fusion in network space. *PloS one*, 6(9):e24642, 2011.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

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

[37] A. Meyer-Lindenberg. From maps to mechanisms through neuroimaging of schizophrenia. *Nature*, 468(7321):194–202, 2010.

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

[39] M. Redekopp, Y. Simmhan, and V.K. Prasanna. Optimizations and analysis of bsp graph processing models on public clouds. In *27th IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 203–214, 2013.

[40] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, page 22, 2013.

[41] 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.

[42] 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.