GBTLX: A First Look

Sanil Rao*, Anurag Kutuluru*, Paul Brouwer*, Scott McMillan†, Franz Franchetti*
*Department of Electrical and Computer Engineering †Software Engineering Institute
{sanilr, anuragku, pbrouwe1, franzf}@andrew.cmu.edu {smcmillan}@sei.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

***Abstract*—We provide a first look at GBTLX, a code generator that translates graph processing programs written using the GraphBLAS Template Library (GBTL) into high-performance C programs that match the performance of hand-tuned implementations. GBTLX refactors code written using GBTL into *problems* that capture the signature of algorithms and *solvers* that capture the semantics (input/output behavior of algorithms. Users provide classes that implement these two aspects using standard GBTL functions and encapsulate the targeted algorithm. GBTLX then performs a sequence of inspection, code generation and high performance execution. First, the user code is traced while running with the original GBTL. Then, the trace is used to define the semantics and signature of the algorithm to be produced in code generation. The SPIRAL system is used to generate high performance C code that implements the user-specified algorithm, specializing the code for algorithm and hardware-dependent optimizations. Finally, the user-provided GBTL-based implementation is replaced by the SPIRAL generated C code. For triangle counting and k-truss enumeration the resulting executables provide performance equivalent to hand-tuned implementations, while the source code is maintainable as it only uses the C++ GBTL library.**

I. Introduction

Graph algorithms have seen increased interest in recent years for a variety of reasons. Whether this be for biology, cybersecurity or social network analysis, researching graph algorithms is a very important task in today's computing landscape [1]. This involves understanding, in detail, how graph algorithms perform on a variety of different types of graphs. As a result, many groups are researching ways to improve graph algorithms and processing across the entire system stack from algorithms and frameworks, all the way down to hardware accelerators for graph applications.

Graph algorithms expressed using a linear algebra formalism [2], as seen through specifications like the GraphBLAS Application Programming Interface (API) [3], [4] or implementations like the GraphBLAS Template Library (GBTL) [5], provide the benefit that the global behavior of the algorithm is easily understood and allow for linear algebra-inspired optimizations. However, writing graph algorithms with matrices often results in temporaries that are huge but normally would not need to be materialized as they will, for example, be reduced in a subsequent algorithmic step. Expressing this in such a C/C++ library is challenging, as this leads to a combinatorial explosion in the API and a large, repetitive code base to capture all cases where optimizations are necessary, across all data formats etc.

To address this issue we are proposing GBTLX, a system that—to the user—looks like a C++ class library based on GTBL, but under the hood is a code generation system based on SPIRAL [6], [7], [8]. GBTLX solves the combinatorial explosion problem by analyzing sequences of multiple GBTL calls to find temporaries that need not be materialized, and specializes code for various data formats and instruction sets and other target platform properties. In this paper we present a first look at GBTLX where the applications are restricted to triangle counting and k-truss enumeration, and we only target multicore CPUs without targeting special instruction sets. We added algorithmic knowledge regarding triangle counting and k-truss to SPIRAL based on previous HPEC Challenge submissions [9], [10]. The resulting performance is on par with the performance reported in these submissions, which shows that GBTLX retains the software abstraction and maintainability of GBTL while providing performance on par with hand-tuned implementations.

Contributions. This paper makes the following contributions:

- It introduces GBTLX, an object oriented inspector/code generator paradigm for GBTL that retains abstraction and maintainability while providing hand-coding level performance.
- GBTL is interpreted as embedded domain specific language (DSL), and tracing of GBTL provides the semantics of user-provided code without the need of a compiler.
- Code written using GBTLX is backwards-compatible with GBTL but can leverage SPIRAL's advanced code generation capabilities to provide future-compatible and performance portable implementations in a true write-once, run-everywhere paradigm.

This paper only provides a first look at GBTLX and focuses on the infrastructure. No in-depth discussion of the SPIRAL code generation module for graphs is provided. Further, we do not claim *performance improvements*, we only claim that high level well-engineered code executes at the speed of hand-tuned code as demonstrated by others.

II. Related Work

There has been a large body of work on graphs and graph algorithms over the past few years. These range from high level frameworks and APIs, to DSLs.

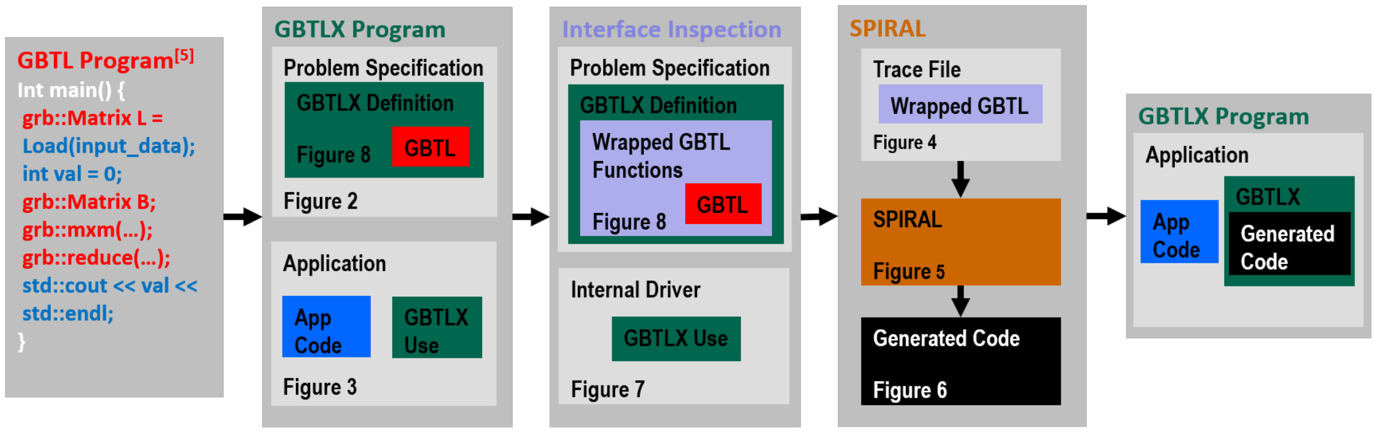

Fig. 1. System overview of GBTLX from source C++ application to generated high-performance application. An orginial GBTL program is modified into a GBTLX program. That program is inspected through an interface, generating a trace file for the SPIRAL backend to generate a high-performance algorithm.

GraphBLAS. One library specification for writing graph algorithms is called GraphBLAS [3] and its C++ implementation, GBTL [5]. This API exposes the basic building blocks of many graph algorithms in a linear algebraic context. These building blocks provide developers easy to use operations for writing graph applications using a linear algebraic approach. It further allows for a common language between those who are not familiar with computing but are familiar with the underlying mathematics governing the algorithm.

GraphIt. One challenge with using a framework or API is that the implementation might not be beneficial for a variety of inputs. In addition it might not always lead to the best performance. In this case, the use of a Domain Specific Language would be beneficial. GraphIt [11] is a DSL designed for graph algorithms that generates fast implementations. In GraphIt, the graph algorithm is written using a high level abstraction, a language for the DSL. Once expressed, a schedule is specified where optimizations can be applied, including data layout and parallelization.

Galois. Galois [12] works in a similar fashion to GraphIt, focusing on improving scheduling. Rather than schedule computation through coordination scheduling, scheduling such that all operations finish before then next can begin, it might be more beneficial to schedule as the data becomes available. This is especially true for fine-grained parallel graph applications. In addition, Galois can take implementations from other DSLs and achieve much better performance through their system.

SPIRAL and FFTX. SPIRAL [6], [7], [8] is a code generation system that originally targeted FFTs and other signal processing applications. Its scope has been broadened significantly over the last several years. SPIRAL is now available as open source tool under a permissive BSD license [13], and recent efforts focus on how to expose SPIRAL to users. A paradigm that has proven successful is to wrap a part of SPIRAL's functionality as embedded domain specific language (DSL), using C/C++. The FFTX [14] project is one such effort, and the machinery used for GBTLX is closely related to FFTX. In either case, a domain-specific C++ library is used as the frontend and SPIRAL is used as the backend code generator tool. The front end has implied delayed execution semantics and object oriented design patterns are used to ensure this behavior.

III. System Overview

We show a full end-to-end example of our system. This example highlights the template that will be used for any problem relating to graph processing.

A very common, easily expressible graph algorithm is counting the exact number of triangles present in a given input graph $\mathcal{G}$. Through the language of linear algebra, triangle counting can be formulated as

$$\Delta = ||L \; .\otimes \; (L \; \oplus.\otimes \; L)||$$

where L is the lower triangular portion of the adjacency matrix representation of $\mathcal{G}$, $\oplus.\otimes$ is the semiring used for matrix multiplication, $.\otimes$ is the point-wise multiplication operator, and Δ is the exact number of triangles [15]. This formulation makes triangle counting a great target application for a linear-algebra based library like GBTL. However, while easy to write, the resulting GraphBLAS operations are quite expensive resulting in poor performance when executed.

We demonstrate the use of GBTLX for the triangle counting problem, showing how to get better performance while writing a linear algebra-based application. We begin with the structure of a GBTLX triangle counting application.

User Code. Figures 2 and 3 illustrate a GBTL reference triangle counting application modified for GBTLX. The first file is the problem specification file. This file include the header `gbtlx.hpp`, containing all the types, macros and functions necessary to use GBTLX. This file also consists of two derived objects, `TriangleProblem` and `TriangleCounter`. In `TriangleProblem`, the user defines a method `randomProblemInstance`, creating a representative input for their application via a graph generator. This method is called when generating the program's trace file. Also, `TriangleProblem`, captures the initial input and final output data structures for the application, encapsulated in the `Signature` class. `TriangleCounter`, contains GBTL

```
#include <graphblas/graphblas.hpp>
#include <gbtlx.hpp>

class TriangleProblem: public GBTLXProblem {
public:
  TriangleProblem() : GBTLXProblem() {}
  TriangleProblem(Signature &sig)
    : GBTLXProblem(sig) {}

  void randomProblemInstance() {
    uint64_t *val = new uint64_t;
    *val = 0;

    // E.g., call external graph generator
    const unsigned int N(10);
    auto *L = new grb::Matrix<uint64_t>(N, N);
    generateGraph(L, N, N);

    Signature s;
    s.in.push_back(L);
    s.out.push_back(val);
    this->sig = s;
  }
};

class TriangleCounter: public GBTLXSolver {
  public:
  void semantics(GBTLXProblem &p) {
    typedef grb::Matrix<uint64_t> MatrixT ;

    MatrixT *inp =
      any_cast<MatrixT *>(p.sig.in[0]);

    MatrixT B(inp->nrows(), inp->ncols());

    //MatMul with mask
    // B = L .* (L +.* L)
    mxm(B, *inp , grb::NoAccumulate(),
        grb::ArithmeticSemiring<uint64_t>(),
        *inp, *inp);

    //Perform reduction
    uint64_t *out =
      any_cast<uint64_t *>(p.sig.out[0]);
    reduce(*out,
           grb::NoAccumulate(),
           grb::PlusMonoid<uint64_t>(), B);

    }

#ifdef HIGHPERFORMANCE
    void solve(GBTLXProblem &p);
#endif
};
```

Fig. 2. Structure of the Triangle Counting Problem Specification.

operations to count the number of triangles in the given adjacency matrix, defined in the `semantics` method. This method uses the input and output defined in `TriangleProblem` as parameters to GBTL operations. In this case, the operations are based on the mathematical formulation described above. Finally, the `HIGHPERFORMANCE` macro allows the make system to link and run the GBTLX generated algorithm.

The second file is the driver application, utilizing the derived objects. The user declares and instantiates the initial input and final output variables, in the `Signature`, as well as `TriangleProblem` and `TriangleCounter`. `TriangleProblem` takes the `Signature` object, binding it internally. `TriangleCounter` then applies itself on `TriangleProblem`, using those bound member objects as parameters to the operations in the `semantics` function, thereby executing the application. The method `generateAndFill`, is responsible for instantiating the associated input matrix along the lower triangle.

```
#include <graphblas/graphblas.hpp>
grb::Matrix<uint64_t> generateAndFill(
   std::string const &pathname) {
   /*assign input data to input objects*/
}

int main(int argc, char **argv) {
    //create GBTL initial objects
    //load matrix from file argv[1]
    grb::Matrix<uint64_t> L(
      generateAndFill(argv[1]));

    uint64_t val = 0;

    //Pass I/O for the Problem
    Signature sig;
    sig.in.push_back(&L);
    sig.out.push_back(&val);

    //create a Problem
    TriangleProblem td(sig);

    //create a Solver
    TriangleCounter t;

    //run the Solver on the Problem
    t.solve(td);

    std::cout << "Number of triangles "
              << val << std::endl;
}
```

Fig. 3. Structure of the Triangle Counting Application.

Interface. The system header file `gbtlx.hpp`, wraps all the GBTL operations and defines the base GBTLX objects and abstract member functions. When the `solve` function is called a computational trace file is generated from the wrapped GBTL functions in `gbtlx.hpp`. This trace file contains a list of input/output data structures, and operations performed by the application. In this case, it includes the input matrix and the result, as well as the set of operations performed on that input matrix to calculate the number of triangles. By the definition, the operations are a matrix multiplication followed by a reduction. It is important to note that during the matrix multiplication, there is a mask of the input matrix L. This allows encapsulation of both the point-wise multiply and the matrix multiplication in a single step. This trace file, seen in Figure 4, will be read as input into our SPIRAL backend for analysis before producing the final binary.

```
spiral_session := [
    rec(op := "triangle_count"), //function name
    rec(op := "MatrixCreation",row:= 90,col:= 90,
    ptr := 0x7fffff45bb30, mat = 0x7fffff45bb30),
    rec(op := "Matrix Multiplication",
        output = IntHexString("0x7fffff45bb60"),
        inputA = IntHexString("0x7fffff45bb30"),
        inputB = IntHexString("0x7fffff45bb30"),
        mask = IntHexString("0x7fffff45ba30")),
    rec(op := "reduce(matrix->scalar)",
        /*many more arguments*/),
];
```

Fig. 4. Generated Trace file for Triangle Counting.

```
//load SPIRAL graph package
Load(graph);
Import(graph);

//parse trace for operations
//perform constraint analysis
t := parse("spiral_session");

//If all constraints met generate code
//load Triangle Counting Options
opts := TCDefaults;
//t is now TriangleCount(param(TInt, "n"));
/*www.spiral.net for RuleTree Overview*/
rt := RandomRuleTree(t, opts);
srt := SumsRuleTree(rt, opts);
cs := CodeSums(srt, opts);

//create files
PrintTo("solve.hpp",
PrintCode("solve", cs, opts));
```

Fig. 5. SPIRAL script for High-Performance Code Generation.

Code Generation. The code generation backend, SPIRAL, utilizes a script file, seen in Figure 5, to generate the high performance equivalent of the operations in `semantics`. The script reads from the trace file and does constraint analysis on the set of operations performed. For this example, the system needs to determine that the set of operations include a matrix multiplication masked by the input matrix L, followed by a reduction. It also has to know whether or not the input graph is undirected (i.e. the matrix is symmetric) in order to generate the correct triangle counting algorithm. Finally, the system has to know that the output is a scalar integer. After these constraints have been checked a high-performance algorithm is generated, and the build system will link `solve.hpp` during compilation of the final high performance binary, effectively replacing the GBTL operations. Figure 6 shows an example of the generated triangle counting algorithm, with casting back to `uint64_t` on completion. This generated algorithm takes advantage of an insight where matrix multiplication is not needed, reducing computation time [9].

IV. System Walkthrough

GBTLX is designed as a user-triggered inspector/code generator, in which user input is given via Makefile targets. In this system, the user specifically decides what type of output they desire. This could be reference, high performance, or debug output, with the final binary being created off this decision. In addition, all GBTLX applications conform to a delayed execution model. In this model, the set of operations that comprise an application is captured and executed such that after the first input is given only the final output is received. There is no inspection of temporaries in between operations. This model allows the SPIRAL backend to accurately generate high-performance code. Figure 1 illustrates the system overview of GBTLX from user code to generated code.

User Application. The user written application has the same general format. First, the user defines two derived classes, referred to generally as the problem specification. These classes, embody the graph problem being written as seen in the previous example through the `TriangleProblem` and `TriangleCounter` objects. These classes are derived

```
void generatedFunction(int *res, int *IJ, int n) {
  int t1;
  t1 = 0;
  for (int i1 = 1; i1 < n; i1++) {
    int t2;
    int  *j1, *jm1;
    t2 = 0;
    j1 = (1 + IJ + n + IJ[i1]);
    jm1 = (1 + IJ + n + IJ[(i1 + 1)]);
    while ((((j1 < jm1))) && (((*(j1) < 0))))) {
      j1 = (j1 + 1);
    }
    while ((((j1 < jm1))) && (((*(j1) < i1))))) {
      int i2, t3;
      i2 = *(j1);
      int  *j11, *j1m1, *j21, *j2m1;
      t3 = 0;
      j11 = (1 + IJ + n + IJ[i2]);
      j1m1 = (1 + IJ + n + IJ[(i2 + 1)]);
      j21 = (1 + IJ + n + IJ[i1]);
      j2m1 = (1 + IJ + n + IJ[(i1 + 1)]);
      while ((((j11 < j1m1))) &&
             (((*(j11) < 0))))) {
        j11 = (j11 + 1);
      }
      while ((((j21 < j2m1))) &&
             (((*(j21) < 0))))) {
        j21 = (j21 + 1);
      }
      while ((((((j11 < j1m1))) &&
               (((j21 < j2m1)))))
             && (((((*(j11) < i1))) &&
                  (((*(j21) < i1))))))) {
        if (((*(j11) < *(j21)))) {
          j11 = (j11 + 1);
        } else if (((*(j21) < *(j11)))) {
          j21 = (j21 + 1);
        } else {
          t3 = (t3 + 1);
          j11 = (j11 + 1);
          j21 = (j21 + 1);
        }
      }
      t2 = (t2 + t3);
      j1 = (j1 + 1);
    }
    t1 = (t1 + t2);
  }
  *(res) = t1;
}
```

Fig. 6. Triangle Counting Algorithm generated by GBTLX. The algorithm is based off this paper [9].

from the base objects `GBTLXProblem` and `GBTLXSolver`, described in a later section.

The user then creates a separate main application file. In the main application, the user declares the `GBTLXProblem` and `GBTLXSolver` objects. Additionally, the user creates a `Signature` object to encapsulate the initial input and final output data structures. This is necessary because the system creates mirrored data structures for use in any backend generated functions. As an example, the system would convert an adjacency matrix into a flattened one-dimensional array using compressed sparse row format. The user then places these data structures in a class called `Signature`, which is passed into the constructor of `GBTLXProblem`. Finally, the user applies the `GBTLXSolver` to the `GBTLXProblem`, using the member function `solve`. The main application is written separately from the problem specification because of the trace file discussed in the next section.

GBTLX Interface. The interface, `gbtlx.hpp`, acts as the translator between GBTL and the SPIRAL backend. All GBTL functions are blocking or synchronous functions; they must return before the application can continue. In order to get GBTL to work within the delayed execution paradigm, the system wraps all of the user facing operations using C macros as seen in Figure 8 through `OBSERVE`. These macros allow the system to intercept GBTL functions without modifying the GBTL library keeping usage the same. The system utilizes these macros to trigger additional functionality depending on the given compile-time flag. As a result, the system has transformed each of the GBTL operations into either blocking or non-blocking functions, depending on the compile-time flag.

GBTLXProblem/Solver. In addition to the wrapped functions, the GBTLX base objects, `GBTLXProblem` and `GBTLXSolver`, are implemented in the interface. The `GBLTXProblem` object is the specific instance of a problem the user is trying to solve. Its abstract member function, `randomProblemInstance` is responsible for creating a smaller representative problem used during trace generation. This function's written representation should match characteristics of the original input dataset, like types and shape, and can be an external call to a graph generator. In addition, `GBTLXProblem's` implicit `Signature` captures the initial input and final output for the user application. `Signature` is responsible for holding not only the input and output of the application but also any additional data structures unique to the problem.

`GBTLXProblem's` complement, `GBTLXSolver`, contains the set of operations needed to solve a problem generally. This is captured through `GBTLXSolver's` abstract member function, `semantics`. In `semantics`, the user uses library-defined functions from a framework like GBTL, placing in data structures from `GBTLXProblem` as necessary. `GBTLXSolver`'s `solve` function either executes `semantics`, or is overridden, executing the SPIRAL generated function. Passing `GBTLXProblem` into `solve` allows for reuse of the `GBTLXSolver` on a variety of `GBTLXProblem` objects with different properties. Figure 8 shows some code associated with the interface, specifically the masked functions and the GBTLX base objects.

High-Performance. There are a few different targets that are available through GBTLX's build system. The most meaningful target is is the high performance target.

The high-performance target leverages the SPIRAL backend to generate a high-performance equivalent of the `GBTLXSolvers`' operations, replacing those operations. To do this the build system first links the problem specification file together with an internal driver application, used for computational trace generation. The internal driver uses user-modified targets in the Makefile in order to replace the derived GBTLX object names with generalized object names `USER_PROBLEM` and `USER_SOLVER`. Then the internal driver creates a `randomProblemInstance`, and executes the `GBTLXSolver's semantics` function on that instance. `Semantics` calls the wrapped GBTL operations, which not only execute, but also print out information about that function to a trace file. This trace file would contain the operations, and the operations' inputs and outputs, including operators and masks. These pieces are important for the SPIRAL backend to accurately determine if optimization is applicable.

The internal driver, seen in Figure 7, is called in place of the user written driver because of potential complexity in user applications. These applications could use large datasets, causing extended computation times or have user unknown exceptions. The internal driver instead creates a representative `GBTLXProblem` instance via the user implemented `randomProblemInstance`, to save on execution time. Trace generation does not complete if there are application exceptions at run-time. Once the trace file is generated, the SPIRAL backend is launched, generating the high-performance algorithm and linking it to the final binary, by overriding `solve` with the function defined in `solve.hpp`. All of this is done without the need of the user to delineate which region of their application could be optimized.

```
int main(int argc, char **argv) {
   //create a Problem, randomInstance, and Solver
   USER_PROBLEM p;
   p.randomProblemInstance();
   USER_SOLVER s;
   //run Solver semantics
   s.solve(p);
}
```

Fig. 7. Structure of GBTLX Internal Driver.

Reference/Debug. The other targets are the reference target and the debug target. In reference, instead of creating an output file and launching the SPIRAL backend, the interface will call into GBTL directly to execution the original functions. This path is used for correctness verification and is the default build option. Furthermore, the debug target will circumvent delayed execution by allowing inspection of temporary objects used during computation. This is useful for developers to understand how exactly their application is getting the final output.

SPIRAL. The open source SPIRAL backend has been extensively applied to the area of FFTs, and it's scope has been broadened to include new application domains. Within the SPIRAL system is a mathematical descriptor language, Operator Langauge (OL). OL describes the set of mathematical operations being performed for a computation [14]. This set of operations is then placed in a rewrite system that works in a similar fashion to an optimization problem, resulting in generated code. We add on to this system cursory mathematical formulations for graph algorithms utilizing the existing infrastructure available in SPIRAL. This specifically includes OL objects for triangle counting and ktruss enumeration, that can target different hardware platforms.

V. Results

We tested GBTLX using two popular graph algorithms, triangle counting and k-truss enumeration. Triangle counting

```
//GBTLX objects for capture and execution
struct Signature {
    vector<any> in;
    vector<any> out;
    vector<any> in_out;
};

class GBTLXProblem {
public:
    GBTLXProblem() {}
    GBTLXProblem(Signature &Sig) : sig(Sig) {}
    virtual void randomProblemInstance() = 0;
    Signature sig;
};

class GBTLXSolver {
public:
    virtual void semantics(GBTLXProblem &p)=0;
    void solve(GBTLXProblem &p){
        semantics(p);
    }
};

// mxm operation wrapper
template</*many more arguments*/>
void wrapped_mxm(CMatrixT       &C,
                 MaskT const    &Mask,
                 AccumT          accum,
                 SemiringT       op,
                 AMatrixT const &A,
                 BMatrixT const &B) {
#ifdef OBSERVE
    fprintf(stderr,
        "rec(op := \"Matrix Multiplication...",
        &C, &A, &B, &Mask);
    mxm(C,Mask,accum,op,A,B);
#endif
#ifdef REFERENCE
    mxm(C,Mask,accum,op,A,B);
#endif
}

// Do something similar for reduce operation
template</*many more arguments*/>
void wrapped_reduce(/*many more arguments*/){
    ...
}

//macro to intercept GBTL operations
#define mxm wrapped_mxm
#define reduce wrapped_reduce
```

Fig. 8. Abbreviated GBTLX Interface between GBTL and SPIRAL.

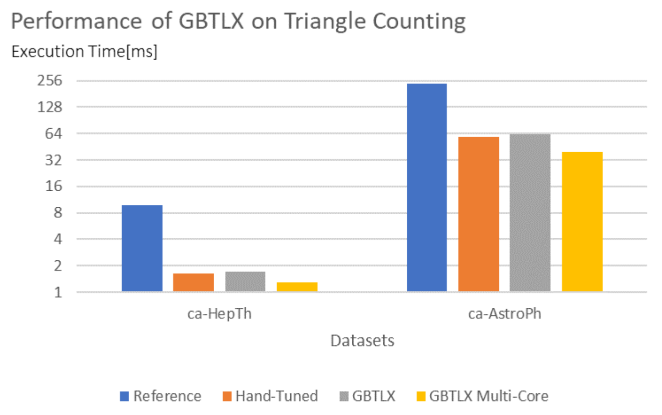

Fig. 9. Performance of GBTLX generated triangle counting algorithm compared against reference and hand-tuned [9]. Y-axis scale is log.

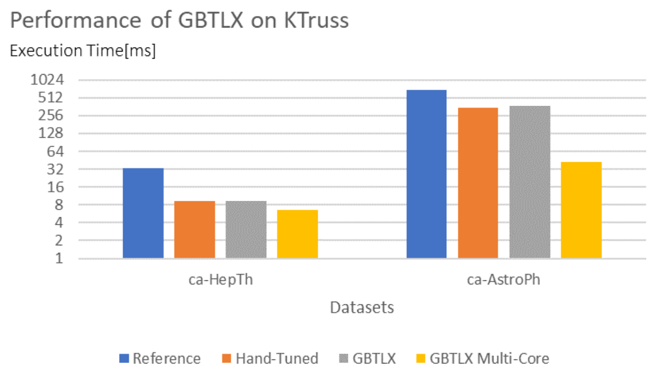

Fig. 10. Performance of GBTLX generated ktruss algorithm compared against reference and hand-tuned [10]. Y-axis scale is $\log_2$.

gives the exact number of triangles present in a given graph, while k-truss enumeration gives the subset of a graph in which each edge in the subset is supported by at least $k-2$ other edges [16]. Both algorithms were run using the reference library, a hand optimized version, and our SPIRAL generated algorithm on two data sets, a smaller dataset, ca-HepTh and a larger dataset, ca-AstroPh. Ca-HepTh, has 9877 nodes, and 25998 edges, while ca-AstroPh has 18772 nodes and 198110 edges. Neither dataset was pre-sorted.

All experiments were run on an Intel Skylake architecture, with g++ 9.2.1. We used GBTL version 3.0 with the `optimized_sequential` backend to generate those results. In addition, we ran generated multi-threaded versions of the two algorithms using four threads. The aggregated results are shown in Figures 9 and 10. The results show that GBTLX code's performance is on par with the hand-tuned code, and that GBTLX is able to leverage multiple cores via OpenMP. These results are obtained while maintaining the clean code structure of original GBTL code.

VI. Conclusion

This paper presents a first look at GBTLX, a GraphBLAS/GBTL implementation that interprets GBTL as an embedded DSL and leverages code generation and automatic performance tuning to overcome the problem of combinatorial explosion in the GraphBLAS API and to avoid materialization of huge non-essential temporaries. The first look at GBTLX uses triangle counting and k-truss enumeration as examples that have concise GBTL implementations and for which previous work has shown how to implement them in low level C to obtain high performance. This paper demonstrates how GBTLX translates the high-level C++/GBTL code into the low level C code without loss of performance, using an inspector/code generator paradigm. GBTLX leverages the SPIRAL code generation system as a backend in the same vein as the FFTX system. Ultimately, the hope is to leverage this technology across libraries and application domains.

VII. Acknowledgement

This work was supported in part by Defense Advanced Research Projects Agency (DARPA) contract HR0011-20-9-0018. This material is also based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation. [DM20-0655]

References

[1] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[2] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[3] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the graphblas api for c," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 643–652, IEEE, 2017.

[4] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, IEEE, 2016.

[5] "GraphBLAS Template Library (GBTL), Version 3.0." Available at https://github.com/cmu-sei/gbtl, June 2020.

[6] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[7] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*, ch. Spiral. Springer, 2011.

[8] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. ryan Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005.

[9] T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan, "First look: Linear algebra-based triangle counting without matrix multiplication," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2017.

[10] M. Blanco, T. M. Low, and K. Kim, "Exploration of fine-grained parallelism for load balancing eager k-truss on gpu and cpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2019.

[11] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[12] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 456–471, 2013.

[13] "SPIRAL Project, Version 8.1.2." Available at https://www.spiral.net.

[14] F. Franchetti, D. G. Spampinato, A. Kulkarni, D. T. Popovici, T. M. Low, M. Franusich, A. Canning, P. McCorquodale, B. Van Straalen, and P. Colella, "Fftx and spectralpack: A first look," in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pp. 18–27, IEEE, 2018.

[15] S. Parimalarangan, G. M. Slota, and K. Madduri, "Fast parallel graph triad census and triangle counting on shared-memory platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1500–1509, IEEE, 2017.

[16] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis,"