GraphIt to CUDA Compiler in 2021 LOC: A Case for High-Performance DSL Implementation via Staging with BuilDSL

Ajay Brahmakshatriya
CSAIL, MIT
Cambridge, USA
ajaybr@mit.edu

Saman Amarasinghe
CSAIL, MIT
Cambridge, USA
saman@csail.mit.edu

***Abstract*—Domain-Specific Languages (DSLs) provide the optimum balance between generalization and specialization that is crucial to getting the best performance for a particular domain. DSLs like Halide and GraphIt and their rich scheduling languages allow users to generate an implementation best suited for the algorithm and input. DSLs also provide the right abstraction for generating code for diverse architectures like GPUs, CPUs, and hardware accelerators. DSL compilers are massive, typically spanning tens of thousands of lines of code and need a frontend, some analysis and transformation passes, and target-specific code generation. These implementations usually require a great deal of compiler knowledge and domain experts cannot prototype DSLs without getting compiler experts involved.**

Using multi-stage programming in a high-level language like Scala, OCaml, or C++, is a great solution because it provides easy-to-use frontend and automatic code generation abilities. The DSL writers typically implement their abstraction as a library in the multi-stage programming language and use it to generate specialized code by providing partial inputs. This solves the problem only partially because DSLs like GraphIt have shown that several domain-specific analyses and transformations need to be performed to get the best performance. Special care has to be taken when targeting massively parallel architectures like GPUs where factors like load balancing, warp divergence, coalesced memory accesses play a critical role.

In this paper, we demonstrate how to build an end-to-end DSL compiler framework and a graph DSL using multi-stage programming in C++. We show how the staged types can be extended to perform domain-specific data flow and control flow analyses and transformations. We also show how our generated CUDA code matches the performance of the code generated from the state-of-the-art graph DSL, GraphIt. We achieve all this in a very small fraction (8.4%) of the code size required to implement the traditional DSL compiler.

***Index Terms*—domain-specific-languages code-generation multi-stage programming data-flow analysis**

I. Introduction

The space of problems from scientific domains has seen a huge growth in recent years. With rapid advances in fields like Machine Learning and Data Science, domain experts are creating new abstractions to make these domains more accessible. These abstractions include libraries like TensorFlow [1], PyTorch [2], Keras [3] or compiler techniques like TVM [4], Tiramisu [5] among others. Although library techniques are usually easy to implement and adopt, using Domain-Specific Languages (DSLs) provides the right balance of programmability and performance. DSLs from domains like TACO [6] for tensor algebra, GraphIt [7]–[10] for high-performance graph applications, Halide [11] for image processing also separate the algorithm specification from the scheduling decisions. A side benefit of using a DSL to program the applications is that the user can now generate implementations for a variety of hardware like CPUs, GPUs, and hardware accelerators.

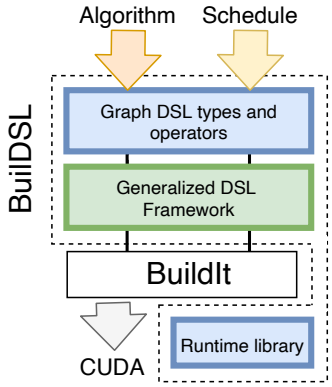

Fig. 1. A complete overview of the BuilDSL framework built on top of BuildIt. BuilDSL takes algorithm and schedule written in an embedded DSL and generates high-peformance CUDA code. The generalized DSL framework (green) is independent from the graph domain types and operators and includes implementation for GPU code generation and Kernel Fusion

DSLs are great at generating high-performance code because of several factors. First, they apply a series of domain specific analysis and transformations that general purpose languages like C++ or Python cannot reason about. Second, they expose a set of scheduling decisions to the user typically in the form of scheduling language that further lets the programmer tune the performance to varying input. Third, they apply various target specific transformations when generating code for diverse architectures. Unfortunately, because the DSL compilers combine so many optimizations and code generation techniques, their codebase usually spans tens to hundreds of thousands of lines of code which makes rapid prototyping of high-performance DSLs tricky. For example, the GraphIt and

978-1-6654-0584-3/22 © 2022 IEEE

TACO DSL compilers currently have more than 80,000 lines of C++ code each. Developers have to spend multiple months, if not years designing the programming interface, analysis, and transformations, and target-specific code generation.

A widely used approach is to embed the DSL in another language, preferably one that supports meta-programming or multi-stage programming like Scala, Python, OCaml among others. The benefit of such an approach is that the developer gets the frontend and code generation for free and can focus on only the implementation of the domain-specific operators just like in a library. This approach has been explored in the past by DSLs and DSL frameworks such as Forge [12], Jet [13], StagedSAC [14], and Delite [15] all of which aim for high-performance while keeping the implementation complexity low. Most frameworks that have exploited this technique are built in high-level functional programming languages. Although being easy to argue about correctness, high-level languages are not a natural fit for high-performance applications. Domain experts who are used to writing libraries and applications in low-level languages like C, C++, or CUDA have to now think in two different languages and programming paradigms when implementing the DSLs.

In this paper, we introduce BuilDSL, a DSL framework built on top of BuildIt. BuildIt [16] brings multi-stage programming to C++ through a lightweight library and helps us solve the multi-paradigm and multi-language problem. BuildIt lets the programmer write programs in multiple stages while maintaining the same C++ syntax and programming model across all stages. As shown in Figure 1, BuilDSL contains a generalized DSL framework that contains routines and utilities commonly required for implementing a DSL. Besides getting a frontend and some code generation from BuildIt for free, BuilDSL provides the framework support for analyzing and transforming the programs in a domain specific way, specializing it for certain inputs and algorithmic patterns and generate code for very different architectures like CPUs and GPUs. For example, BuilDSL has a CUDA extraction pass that automatically extracts CUDA kernels from generated annotated loop nests to be run on GPUs. BuilDSL also has features that make it extremely easy to implement optimizations like Kernel Fusion that combine various operations into a single CUDA kernel to avoid the launch overhead. The generalized framework also has support for extending the dynamic types to create domain-specific analyses for enforcing correctness and improving performance.

To demonstrate the usefulness of our framework, we implement a real graph DSL inspired by the state-of-the-art DSL, GraphIt. Just like GraphIt, our DSL takes algorithm and schedule inputs from the user and generates high-performance CUDA code to be run on GPUs. We show that our DSL can match the performance of GraphIt while keeping the implementation down to a very small fraction of the GraphIt compiler (less than 8.4% lines of code).

This paper makes the following contributions:

- We demonstrate that using multi-stage programming is an easy way to generate sophisticated code while keeping the complexity to minimum. We do this by implementing a compiler for regular expressions using BuildIt that fits in a mere 103 lines of C++ code
- We present BuilDSL, a DSL framework that combines routines and utilities that are essential for implementing high-performance DSLs
- We implement a CUDA generating graph DSL on top of our framework that matches the programming interface of the real-world graph DSL, GraphIt
- We show that the code generated from our graph DSL is on-par with that of the code generated from GraphIt on 5 diverse applications and 9 datasets while keeping our entire implementation in only 2021 lines of C++ code which is less than 8.4% of GraphIt's codebase

II. Motivating Example

Before we explain what it takes to implement an end-to-end compiler with analysis, transformations, and complicated GPU code generation, we want to show that using staging is an easy approach for implementing compilers for small languages. In this section, we will show how one can write a very naive interpreter for a simple language and convert it into a compiler with staging. For this example, we take a small subset of Regular Expressions (RegEx) with just `a-z`, `A-Z`, `0-9` and $\star$. This subset is simple enough to understand the implementation completely and at the same time is general enough that it has some complex control flow generated by the $\star$ character. We start with Algorithm 1 that shows how a naive interpreter could match a given RegEx pattern with an input string.

Algorithm 1 A simple algorithm for a RegEx interpreter implementation main loop. The algorithm takes as input the RegEx pattern and the string to match both as strings

```
1: Input: RegEx pattern R, String to match S
2: Output: Boolean that says whether S exactly matches R.
3: rLen ← len(R)
4: sLen ← len(S)
5: current ← [false] × rLen + 1
6: next ← [false] × rLen + 1
7: progress(R, current, −1)
8: for s ← 0 to sLen − 1 do
9:     for r ← 0 to rLen − 1 do
10:        if current[r] then
11:            m ← R[r]
12:            if isnormal(m) then
13:                if S[s] == m then
14:                    progress(R, next, r)
15:            else if m ==′ .′ then
16:                progress(R, next, r)
17:            else
18:                error
19:    if count(next) == 0 then
20:        return false
21:    current ← next
22:    next ← [false] × rLen + 1
23: return current[rLen] == false
```

This algorithm takes in the RegEx pattern and the string to match and calculates their lengths. It then allocates two boolean arrays of the size of the RegEx pattern. The `current` array tracks which characters in the RegEx can currently be matched and the `next` arrays tracks which characters in the RegEx can be matched next. Since regular expressions have non-determinism in the matching process, more than one boolean in the `current`

```
#include <iostream>
#include <cstring>
#include "builder/dyn_var.h"
#include "builder/static_var.h"
#include "blocks/c_code_generator.h"
template <typename T>
using dyn = builder::dyn_var<T>;
template <typename T>
using static = builder::static_var<T>;
```

```
dyn<int (char*)> d_strlen("strlen");
dyn<int> match_regex(const char* re,
    dyn<char*> str);
bool is_normal(char m) {
  return m >= 'a' && m <= 'z' || m >= 'A'
    && m <= 'Z' || m >= '0' && m <= '9';
}
void progress(const char *re, static<char> *next, int p) {
  int ns = p + 1;
  if (strlen(re) == ns) {
    next[ns] = true;
  } else if (is_normal(re[ns])
      || '.' == re[ns]) {
    next[ns] = true;
    if ('*' == re[ns+1]) {
      // We are allowed to skip this
      // so just progress again
      progress(re, next, ns+1);
    }
  } else if ('*' == re[ns]) {
    next[p] = true;
    progress(re, next, ns);
  }
}
```

```
int main(int argc, char* argv[]) {
  builder::builder_context context(builder::UNSTRUCTURED);
  auto ast = context.extract_function_ast(match_regex,
    "match_re", argv[1]);
  std::cout << "#include <string.h>" << std::endl;
  block::c_code_generator::generate_code(ast, std::cout);
  return 0;
}
```

```
dyn<int> match_regex(const char* re, dyn<char*> str) {
  // allocate two state vectors
  const int re_len = strlen(re);
  static<char> *current = new static<char>[re_len + 1];
  static<char> *next = new static<char>[re_len + 1];
  for (static<int> i = 0; i < re_len + 1; i++)
    current[i] = next[i] = 0;
  progress(re, current, -1);
  dyn<int> str_len = d_strlen(str);
  dyn<int> to_match = 0;
  while (to_match < str_len) {
    // Don't do anything for $.
    static<int> early_break = -1;
    for (static<int> state = 0; state < re_len; ++state) {
      if (current[state]) {
        static<char> m = re[state];
        if (is_normal(m)) {
          if (-1 == early_break) {
            // Normal character
            if (str[to_match] == m) {
              progress(re, next, state);
              // If a match happens, it
              // cannot match anything else
              // Setting early break
              // avoids unnecessary checks
              early_break = m;
            }
          } else if (early_break == m) {
            // The comparison has been done
            // already, let us not repeat
            progress(re, next, state);
          }
        } else if ('.' == m) {
          progress(re, next, state);
        } else {
          printf("Invalid Character(%c)\n", (char)m);
          return false;
        }
      }

      // All the states have been checked
      // Now swap the states and clear next
      static<int> count = 0;
      for (static<int> i = 0; i < re_len + 1; i++) {
        current[i] = next[i];
        next[i] = false;
        if (current[i])
          count++;
      }
      if (count == 0)
        return false;
      to_match = to_match + 1;
    }
    // Now that the string is done,
    // we should have $ in the state
    static<int> is_match = (char)current[re_len];
    for (static<int> i = 0; i < re_len + 1; i++) {
      next[i] = 0;
      current[i] = 0;
    }
    return is_match;
}
```

Fig. 2. The complete implementation of the the RegEx interpreter written with BuildIt staged types to create a RegEx compiler. The 4 sections here show the i) BuildIt type includes ii) Helper functions for state transitions iii) Main function for staging and code generation and iv) The main implementation of the interpreter being staged. The compiler comes to a total of only 103 lines of C++ code.

array could be set to `true` at a time. The algorithm relies on the `progress` function that given a particular match decides which characters in the RegEx can be matched next by setting the `next` array. This function essentially encapsulates all the state transition logic. If we want to add more characters like `+`, `?`, or `()`, we would change the implementation of the `progress` function. Finally, the main part of the algorithm iterates over every character in the input string and updates the `next` array with each match in the `current` array. Before moving on to the next character, we swap `next` and `current` and clear `next`. When all the characters are done matching, the algorithm returns a successful match if and only if the last boolean in `next` (which corresponds to the end of the RegEx) is set.

This naive algorithm potentially matches every character in the RegEx with every character in the string. Further for each match, the `progress` function potentially scans through the entire RegEx to find the next locations to match further, blowing up the complexity. It is easy to see why such an interpreter would be very inefficient. Now we will use the C++ multi-stage programming framework BuildIt to convert this naive interpreter into a compiler using Futamura projections [17]. We choose BuildIt over other multi-stage frameworks because unlike other frameworks it allows writing and generating imperative code with rich control flow. BuildIt also allows updates to first stage variables and expressions under conditions based on expressions that are evaluated in the second stage. We will see why this is required next. Figure 2 shows the **entire** working source code for the compiler. The first section shows all the BuildIt headers and types being included. The second section shows some helper functions including the `progress` function.

The third section shows the `main` function that calls BuildIt to stage the interpreter and generate the code. Finally, the last section shows the actual matching function to be staged.

To write the compiler with BuildIt, we have to decide the types for all the variables because BuildIt uses the declared types of variables and expressions to decide what stage they will be executed in. Any expression of type `static<T>` (or any non BuildIt type) is evaluated in the first stage and expressions with type `dyn<T>` are converted into code to be executed in the second stage. Since we want to generate code for any given regular expression, we declare the RegEx input to be of type `const char*` and the string to match of type `dyn<char*>` as shown Line 42. The two boolean arrays `current` and `next` are declared to be of type `static<char>` because we want to completely evaluate them away in the first stage (Line 45-46). Line 52-55 show the two loops that iterate over each character in the input string and each boolean in the `current` array respectively. Line 61 shows a condition on `dyn<T>` variables. Based on this condition we make updates to the `next` array in the `progress` function. BuildIt's unique ability to support such patterns gives rise to complex control flow in the generated code. Let us look at a very simple RegEx input - `ab*c`. Figure 3 shows the code generated for this input when it is passed through the compiler above. The first thing we can notice is that all traces of the inner loop that iterates over characters of the RegEx have disappeared. The generated code instead has state transitions implemented as `if-and-goto`. Further, all traces of the complex logic in the `progress` function have disappeared too. This means that the developer could implement the `progress` function in an easy to understand but not so efficient way and it wouldn't affect the runtime at all. This example demonstrates that starting from a naive interpreter logic, we are able to generate quite efficient code which looks very different from the original code. Note that, this code is still not the best implementation of RegEx because we did not apply vectorization, parallelization, and other advanced compiler code generation techniques. We will show how this can be achieved with our graph DSL (§IV).

```
1  int match_re (char* arg1) {
2    char* var0 = arg1;
3    int var1 = strlen(var0);
4    int var2 = 0;
5    if (var2 < var1)
6      if (var0[var2] == 97) {
7        var2 = var2 + 1;
8        label0:
9        if (var2 < var1) {
10         if (var0[var2] == 98) {
11           var2 = var2 + 1;
12           goto label0;
13         }
14         if (var0[var2] == 99) {
15           var2 = var2 + 1;
16           if (var2 < var1)
17             return 0;
18           else
19             return 1;
20         } else
21           return 0;
22       } else
23         return 0;
24     } else
25       return 0;
26   else
27     return 0;
28 }
```

Fig. 3. Code generated from the RegEx compiler for the input `ab*c`. Notice that this generated code effectively only has one loop

III. Background and Challenges

In this section we provide some background into requirements for real world DSLs and the technical challenges in implementing their compilers.

A. The GraphIt DSL

GraphIt is a DSL for graph computations that generates high-performance C++ and CUDA to be run on CPUs and GPUs among other hardware. GraphIt separates what is computed (specified in the algorithm language) from how it is computed (specified in a scheduling language). The GraphIt algorithm language is an imperative language that uses abstract data structures and operators. This means that objects like vertex sets and edge sets can have different representations like bitmaps, boolmaps or sparse queues and CSR, COO, or blocked CSR based on what is suitable for the optimizations applied. Similarly, the implementation of the operators combines a variety of parallelization techniques, iteration direction, deduplication strategies, etc. The scheduling language enables programmers to easily search through this complicated tradeoff space by composing together a large set of edge traversal and vertex data layout optimizations. These choices allow the developer to fine-tune the performance for their algorithm and graph input. The GraphIt GPU backend can also generate CUDA code to be run on the NVIDIA GPUs and is the current state-of-the-art in terms of performance [9]. The whole of the GraphIt compiler framework is about 82,000 lines of C++ code out of which about 26,000 lines are required for the GPU part of the compiler.

B. Staging with BuildIt

BuildIt [16] brings the idea of staging or multi-stage programming to C++ through a light-weight library. BuildIt uses a type-based approach, meaning the declared types (and only the types) of the variables and expressions are used to decide the stage in which they would be evaluated. As seen in RegEx example in Figure 2, BuildIt also uses the same syntax for all operations and control flow constructs across all stages. This provides a very seamless interface to the user and makes moving code between stages much easier. Another benefit of BuildIt is that it uses an imperative programming model and hence is easier for domain experts to use.

Let us look at a few parts of the GraphIt compiler, specifically the GPU backend and the challenges one would face while implementing it with staging. Although these are being explained in the context of a graph DSL, parallelization, data-structure choices, and fusing kernels is essential for performance in any DSL and these would similarly apply to other domains -

C. UDF Analysis and Transformations

GraphIt's programming model requires the programmer to write serial code for the algorithm and the compiler automati-

cally parallelizes it based on the graph domain knowledge. One of the challenges in doing this is to insert atomics and other synchronization primitives at the right place to ensure correctness. The GraphIt compiler could conservatively insert atomics at every access to shared data, but that would compromise the performance. To solve this problem, the compiler performs a data-flow analysis to identify which variables are shared and independent between different threads. It only inserts atomics when a variable is potentially modified by more than one thread at the same time. This data-flow analysis is seeded by the choice of load balance and direction of iteration and goes through all the operations that are performed on edges and vertices. This analysis is implemented as a mid-end pass on the compiler IR. When implementing the compiler with staging, we do not have an IR or a pass infrastructure. We will show in Section IV how BuilDSL extends the types to perform the data-flow analysis to provide the same guarantees. The same idea can be used to implement a variety of analysis and transformation passes.

D. Scheduling Language and Specialization

The GraphIt DSL first applied the idea of separating the algorithm from the schedule for the applications from the graph domain. Other DSLs like Halide [11], TACO [6], Tiramisu [5] also apply similar techniques to applications from other domain. Such a separation allows the developer to write the application once and fine-tune the performance of the generated code to better suit the inputs by simply tweaking the schedule. Having a clear separation between inputs for correctness and inputs for performance also lets the programmer apply techniques like auto-tuning to choose the best schedule and hence generate the best performing code. The GraphIt GPU backend, G2 identified 7 independent scheduling dimensions that are critical for the performance of graph applications on GPUs and exposed them to the user through the scheduling language. The GraphIt DSL compiler implements scheduling as transformation and code generation passes in the compiler with each schedule transforming the IR in different ways. Once again, BuilDSL is able to support specializing the generated code without having explicit IR and transformation passes.

E. GPU Code Generation

The GraphIt DSL compiler can generate code for massively parallel GPUs with thousands of threads organized in complex thread hierarchies. Besides assigning work to threads in a load-balanced way, the GPU backend also has to take care of moving data between host and GPU, shared memory allocation, warp, thread, and grid synchronization, avoid warp divergence, and accessing global memory in a coalesced way among other things. The GraphIt DSL GPU backend achieves all this with a combination of code generation passes and runtime libraries. This significantly increases the complexity of the code generation passes.

F. Kernel Fusion

An optimization that the GraphIt GPU backend relies on heavily is Kernel Fusion. When enabled, this moves outer loops into the CUDA kernel to avoid the cost of launching a new kernel for every step in every iteration. This is particularly useful when the outer loop runs for a large number of iterations and performs very little work per iteration. Kernel Fusion can improve performance by up to 1000x for some applications and inputs [9]. Implementing Kernel Fusion requires identifying local variables that are being used inside the loop, hoist and transfer them between the host and GPU, insert appropriate synchronization between individual steps to ensure semantic equivalence and using a fixed number of threads to implement each step in the loop (which potentially require a different number of threads). Because Kernel Fusion is a non-local optimization, its implementation adds signifant complexity to the code generation passes. Kernel Fusion is a critical optimization for performance and the generalized DSL framework in BuilDSL provides support for implementing Kernel Fusion with ease for a variety of operators.

IV. Implementation

In this section, we will explain how BuilDSL handles all the challenges explained in Section III and how we use the framework to implement a GPU graph DSL that matches the performance of the state-of-the-art graph DSL, GraphIt.

A. Graph DSL Programming Model

Before we get into how each component of BuilDSL is implemented, we will introduce BuilDSL's programming interface with an example application. BuilDSL is implemented as an embedded DSL in C++ using the BuildIt multi-staging library. The design of BuilDSL's programming API including data-types and operators is similar to GraphIt's carefully designed API. Just like GraphIt's types, all data-types are abstract meaning they could have varying implementations based on scheduling parameters. These data types are implemented as extensions to (or wrappers around) BuildIt's `dyn<T>` types. As a result, instead of getting completely evaluated in the first stage, they generate code to be run in the second stage. BuilDSL packs all the implementation inside the operators that it exposes to the users like the `vertexset_apply` and `edgeset_apply`. These operators use a combination of `dyn<T>` (arguments) and `static<T>` (schedule) inputs to generate specialized high-performance code to be run on GPUs. We will explain the key operators that make use of BuilDSL's specialization and the scheduling objects associated with them next.

vertexset_apply: The `eg::vertexset_apply` applies a user-defined function (UDF) to a set of vertices in a `VertexSubset` in parallel. The operator takes two arguments, a `VertexSubset` and a function that accepts a `Vertex`. The first parameter can also be an `EdgeSet` (the graph data structure), in which case the function is applied to all the vertices in the graph. This operator is typically used to initialize values associated with the vertices or for updating the values at a per vertex level (without looking at the neighbors).

edgeset_apply: The `eg::edgeset_apply` is one the main operators in our DSL and applies a UDF to each edge in a set. The operator can be invoked by the `apply` function that takes

as arguments an `EdgeSet` (graph data structure) and a function that takes as arguments a pair of vertices corresponding to the source and the destination of the edge. The user can add clauses like `from` and `to` that filter out edges that originate from and are incident to a subset of vertices respectively. These clauses can either take a `VertexSubset` or a function that takes as argument a `Vertex` and returns a boolean. There is another version of this operator `apply_modified` that also tracks vertices that have a certain property updated and return it as a `VertexSubset`. This is used to implement active frontiers in data-driven algorithms like BFS and CC.

All the datastructures passed to these arguments are runtime values and hence are declared as `dyn<T>`. There are various ways this operator can map the execution of the UDF on each edge to the threads, warps and thread blocks of the GPU. At the same time, there are various choices with respect to the actual representation of the data structures involved and how they are updated. These choices are specified using a `Schedule` object that is passed to the constructor of the `edgeset_apply` operator. Since these choices are supplied at the compile time in the DSL and are used for specialization, the argument is declared as `static<T>`.

fuse_kernel: The `eg::fuse_kernel` operator allows combining calls to various operators into a single GPU kernel launch as opposed to separate kernel launches for each of them. The operator takes as argument a C++ lambda that wraps around the calls to the operators to be fused. The operator also takes a boolen (`static<T>`) to determine if the operators are to be actually fused. The operators are fused only if this boolean evaluates to true. This allows enabling/disabling fusion as a scheduling option. `fuse_kernel` can also be wrapped around control flow structures like `if-then-else` and loops to fuse all the operators in every iteration or branch.

Scheduling objects: The scheduling objects declared as `static<T>` are used by BuilDSL to specialize the code generated by the operators. The type `Schedule` is an abstract class that has two derived types, the `SimpleGPUSchedule` and `HybridGPUSchedule`. The `SimpleGPUSchedule` has members and functions to configure choices for load balancing, vertex set representation, vertex set deduplication, iteration direction and edge blocking. The `HybridGPUSchedule` type allows combining two `Schedule` objects based on some runtime condition like size of the active vertex set. We will explain the details of how these scheduling objects affect the generated code further in this section.

Figure 4 shows an example of the BFS application written in BuilDSL making use of all the operators. The example also shows the construction and use of a `Schedule` object that is passed to the `edgeset_apply` operator for specialization.

Now that the programming model is clear, we will explain how BuilDSL addresses each of the challenges mentioned in Section III.

B. GPU Code Generation

The biggest challenge when implementing high-performance DSLs is providing an abstraction for parallelization. Especially

```
1  VertexData<int> parent("parent");
2  GraphT edges("edges");
3
4  static void updateEdge(Vertex src, Vertex dst) {
5    parent[dst] = src;
6  }
7  static dyn<int> toFilter(Vertex v) {
8    return parent[v] == -1;
9  }
10 static void reset(Vertex v) {
11   parent[v] = -1;
12 }
13 static void BFS(dyn<char*> graph_name, dyn<int> src,
14   dyn<float> t, eg::Schedule &s1, bool to_fuse) {
15   ...
16   edges = eg::runtime::load_graph(graph_name);
17   parent.allocate(edges.num_vertices);
18   VertexSubset frontier =
19     eg::runtime::new_vertex_subset(edges.num_vertices);
20
21   eg::vertexset_apply(edges, reset);
22
23   parent[src] = src;
24   frontier.addVertex(src);
25   eg::fusee_kernel(to_fuse, [&]() {
26     while(frontier.size() != 0) {
27       eg::edgeset_apply(s1).from(frontier).to(toFilter)
28         .apply_modified(edges, frontier, parent,
29         updateEdge);
30     }
31   });
32   ...
33 }
34 ...
35 int main(int argc, char* argv[]) {
36   // Define a schedule for specialization
37   SimpleGPUSchedule s1.
38   s1.configLoadBalance(VERTEX_BASED);
39   s1.configDeduplication(DISABLED);
40   // Extract the specialized implementation
41   // and generate code
42   auto ast = builder::extract_function(BFS, "BFS",
43     s1, true);
44   eg::pipeline::run_eg_pipeline(ast, std::cout);
45 }
```

Fig. 4. Implementation of the BFS algorithm in BuilDSL. Notice the call to the `vertexset_apply` and `edgeset_apply` operators

for backends like NVIDIA GPUs that have a clear separation between code that runs of the host and the code that runs on the GPU, the code generation can become complex with `__global__` kernel generation, data transfer, kernel launch parameters, synchronization primitives, etc. Although the CUDA programming model is great for performance, it is not the most programmer-friendly for domain experts. OpenMP style pragma annotations is another popular abstraction for expressing parallelism and is used in many state-of-the-art graph libraries like Ligra [18] and GAPBS [19]. BuilDSL extends the BuildIt framework to be able to generate CUDA kernels from annotated loop nests using BuildIt's annotation system and adding a CUDA kernel extraction pass. Figure 5 shows a doubly nested loop written with BuildIt annotated as `"CUDA_KERNEL"`. The CUDA extraction pass in BuilDSL, identifies such annotated loops and converts them into CUDA kernel while mapping the outer loop to the blocks in a grid and the inner loop to the threads in a block. The pass replaces all accesses to the outer loop index with `blockIdx.x` and the inner loop index with `threadIdx.x`. This pass also identifies all the local variables that are used inside the doubly nested loop that are declared globally or in the enclosing functions.

```
GraphT g = ...;
builder::annotate("CUDA_KERNEL");
for (dyn<int> cta = 0; cta < 60; cta = cta + 1) {
  for (dyn<int> t = 0; t < 512; t = t + 1) {
    dyn<int> tid = cta * 512 + t;
    dyn<int> edge_dst = g.edges[tid];
    ...
  }
}
```

Fig. 5. Example of an annotated loop nest to be converted into a CUDA kernel

```
void __global__ cuda_kernel_0(GraphT arg0) {
    int var0 = blockIdx.x * 512 + threadIdx.x;
    int var1 = arg0.edges[var0];
    ...
}
...
GraphT g = ...;
cuda_kernel_0<<<60, 512>>>(g);
cudaDeviceSynchronize();
```

Fig. 6. Generated CUDA and host code for an annotated loop nest

These variables are passed to the kernel as arguments and are copied back after the kernel is finished executing. The loop bounds are used to pass the kernel launch parameters at launch. Figure 6 shows the host and GPU generated code for this loop nest. This basic primitive for launching CUDA kernels by annotating loop nests enables the DSL developer to implement various load-balancing techniques by tuning the grid and block dimensions and is not limited to the graph domain. The BuilDSL runtime library also provides warp, thread, and grid-level synchronization primitives that can be called from within the loop nests. The implementation of the operators like **`vertexset_apply`** and **`edgeset_apply`** in BuilDSL use such annotated loop nests inside them to map the execution of the UDFs on each vertex or edge to GPU threads automatically.

C. Kernel Fusion

All parallel operators in BuilDSL are implemented using the annotated loop nests. As a result, each call to an operator generates separate CUDA kernels that are launched from the host code. As explained in Section III, this can lead to a lot of overhead when each operator does very little work and the kernel launch overhead starts to dominate. The user might want to fuse a series of calls to an operator or even entire loop nests into a single GPU kernel. We want to achieve this without having to write a fused version for each combination like in some library approaches. BuilDSL uses the **`static`**`<T>` variables to implement Kernel Fusion in a generic way. BuilDSL defines an enum `context` that can have values `HOST` or `DEVICE`. As shown in Figure 7 Line 1 BuilDSL also has a **`static`**`<`**`enum`**` context>` variable `current_context` that is initialized to `HOST`. BuilDSL now exposes an operator **`fuse_kernel`** (Line 4) to the user to wrap around a series of calls or entire loop nests. The implementation of this operator starts one annotated loop nest and sets the `current_context` to `DEVICE`. Now each operator implementation checks the `current_context`. If it is set to `HOST`, it creates its own loop nest. Otherwise, it just uses the loop indices from the loop nest created by **`fuse_kernel`**. If the operator needs more threads than what is spawned by the **`fuse_kernel`**, it adds another loop

```
static<enum current_context> = HOST;
dyn<int> *cta_ptr, *t_ptr = nullptr;

void fuse_kernel(std::function<void(void)> body) {
  current_context = DEVICE;
  builder::annotate("CUDA_KERNEL");
  for (dyn<int> ct = 0; ct < MAX_CTA; ct = ct + 1) {
    for (dyn<int> t = 0; t < MAX_T; t = t + 1) {
      // Save references to indices
      // for use inside operator
      cta_ptr = ct.addr();
      t_ptr = t.addr();
      body();
    }
  }
  current_context = HOST;
}

void vertexset_apply(VertexSubset &s, udf_t f) {
  // Specialization based on current context
  if (current_context == HOST) {
    current_context = DEVICE;
    builder::annotate("CUDA_KERNEL");
    for (dyn<int> ct = 0; ct < MAX_CTA; ct = ct + 1) {
      for (dyn<int> t = 0; t < MAX_T; t = t + 1) {
        dyn<int> tid = ct * MAX_T + t;
        ...
      }
    }
    current_context = HOST;
  } else {
    dyn<int> ct = *cta_ptr;
    dyn<int> t = *t_ptr;
    dyn<int> tid = ct * MAX_T + t;
    ...
    // Synchronize the grid to ensure correctness
    grid_sync();
  }
}
```

Fig. 7. Implementation of the **`fuse_kernel`** operator in BuilDSL and the specialization of the **`vertexset_apply`** operator to implement Kernel Fusion

to simulate more threads with the fixed threads while inserting appropriate block and grid synchronization primitives. Since the `current_context` variable is declared as **`static`**`<T>` all conditions based on it are completely evaluated during compile time and do not introduce any runtime overheads.

Unlike the GraphIt DSL compiler, the implementation of the Kernel Fusion optimization does not require any additional passes that generate separate kernels, hoist variable declarations as parameters or introduce specialized loop nests. BuilDSL implements it as specialization based on **`static`**`<T>` variable reducing the complexity and the lines of code required for implementing the optimization to the minimum. We will compare the exact lines of code in BuilDSL and GraphIt required to implement Kernel Fusion in Section V.

D. UDF Analysis and Transformations

As explained in Section III, one of the major challenges in implementing a high-performance DSL is performing analysis on the code written by the user and using that analysis to transform the generated code for both correctness and performance. Since BuilDSL lacks a pass infrastructure to implement domain-specific passes, the analysis has to be packed in the extended types. The types are a dual of passes when moving from compilers to staging. To implement a generic data-flow analysis in BuilDSL, we track the analysis bits as **`static`**`<T>` with the values to be analyzed. These bits can then

```
1  // Type to hold a Vertex
2  struct Vertex {
3    // The actual dynamic vertex this value holds
4    dyn<int> vid;
5    // Dataflow analsis bits for this value
6    enum access_type {INDEPENDENT, SHARED, CONSTANT};
7    access_type current_access;
8    ...
9    // Operator overloads for Vertices
10   Vertex operator + (const Vertex& rhs) {
11     // Call the internal + operator on dyn<int>
12     Vertex out = vid + rhs.vid;
13     // Update analsis bits
14     if (current_access == INDEPENDENT
15       && rhs.current_access == CONSTANT
16       || current_access == CONSTANT
17       && rhs.current_access == INDEPENDENT)
18       out.current_access = INDEPENDENT;
19     else
20       out.current_access = SHARED;
21     ...
22     return out;
23   }
24 };
25 // Type to hold expressions like parent[dst]
26 template <typename T>
27 struct VertexDataIndex {
28   // The actual variable and index being tracked
29   // v[index]
30   VertexData<T> v;
31   Vertex index;
32   // Specialization for the += operator
33   // with atomics when index is shared across threads
34   dyn<T> operator += (const dyn<T>& rhs) {
35     if (index.current_access == INDEPENDENT)
36       return v[index] += rhs;
37     else
38       return atomicSum(&v[index], rhs);
39   }
40 };
41 // Edgeset apply seeding the analysis
42 void vertexbased_loadbalance(..) {
43   ...
44   Vertex src = ...;
45   Vertex dst = ...;
46   // In vertex based load balance,
47   // one source vertex is assigned to one thread,
48   // but the same destination can be visited by
49   // multiple threads
50   src.current_access = INDEPENDENT;
51   dst.current_access = SHARED;
52   udf_body(src, dst);
53   ...
54 }
```

Fig. 8. Implementation of the data-flow analysis to track SHARED and INDEPENDENT vertex data indices for inserting atomics in appropriate cases

be updated or propagated with each operation by overloading the calls to the operators. Finally, these **static<T>** bits can be queried at the time of operator implementation to specialize the behavior, which is the dual of transformations.

Figure 8 shows the implementation of the data flow analysis to track shared and independent values for inserting atomics at appropriate places. This analysis tracks whether indices used to index into **VertexData** can be shared across different threads or is guaranteed to be unique across threads. If it is guaranteed that no two threads would update the same index, we don't have to insert any atomics. This analysis is seeded by the load balance implementation where a load balance technique like VERTEX_BASED sets the current_access analysis bit of the source vertex to INDEPENDENT while the bit for the destination vertex is set to be SHARED. The VERTEX_BASED load balance assings each source vertex to a different thread, while the same destination

```
1 SimpleGPUSchedule s1;
2 s1.configDirection(direction_type::PULL,
3   frontier_rep::BITMAP);
4 s1.configLoadBalance(load_balance::VERTEX_BASED);
5 s1.configFrontierCreation(frontier_rep::BITMAP);
6 ...
7 edgeset_apply(s1).from(frontier).apply_modified(...);
```

Fig. 9. Example of creating, configuring and applying a **SimpleGPUSchedule**

can be potentially visited by two threads when two vertices in the graph have a common neighbor. This bit is queried to specialize the implementation of the += reduction operator as shown on Figure 8 Line 34.

The beauty of implementing data-flow analysis with **static<T>** variables is that the developer only has to implement how the values are propagated at each operation as shown in Line 14. The developer doesn't have to implement any convergence analysis or worry about back edges. BuildIt treats **static<T>** variables in a special way, such that it unrolls loops till the **static<T>** values reach a previously visited value. Thus, if there is a back-edge that further updates the analysis bits, the loop will be unrolled with differing implementation till the analysis converges. This way of implementing analysis not only requires very few lines of code but most importantly doesn't require any knowledge of compiler analysis to implement. This technique also generalizes well to other analyses and BuilDSL uses the exact same technique to track which **VertexData** is being tracked in the **edgeset_apply.apply_modified** operator to produce the output frontier.

E. Scheduling language and specialization

In this section, we will explain how BuilDSL implements a scheduling language and specialized code generation. The GraphIt DSL compiler implements scheduling by progressively transforming the IR before code generation. Because BuilDSL lacks a domain-specific pass framework, once again we will rely on extending the types to affect the code we generate based on the scheduling input. The GraphIt GPU extensions, G2 identified the 7 independent dimensions of scheduling that are critical for the performance of graph applications on GPUs namely - load balance, vertex subset representation, the direction of traversal, deduplication, vertex ordering, edge blocking, and kernel fusion. The programmer can choose one of many options for each of these dimensions giving rise to a total of about 576 combinations, the highest supported by any graph GPU framework [9]. We support all these dimensions with BuilDSL. The programmer encodes these options by the means of a scheduling object like shown in Figure 9 configuring one or more of the scheduling options. Line 7 shows how this schedule can be applied to the operators when implementing the algorithm. The implementation of the operators can then specialize the code based on these scheduling objects by introducing branches on these flags. Since the scheduling objects are not of **dyn<T>** type, these branches are completely evaluated avoiding any kind of runtime overhead.

One performance-critical schedule that GraphIt supports is direction optimization, where the direction of iteration

```
...
// Check if the applied schedule is Hybrid
if (s_isa<HybridGPUSchedule>(s)) {
  HybridGPUSchedule *h = s_to<HybridGPUSchedule>(s);

  // Branch based on dyn<T> expressions and recursively
  // call edgeset_apply
  if (in_set.size() <= h->threshold * graph.num_vertices)
    edgeset_apply(h->s1).from(in_set).apply(...);
  else
    edgeset_apply(h->s2).from(in_set).apply(...);
  return;
}

SimpleGPUSchedule s1, s2;
s1.config...
s2.config...
// Create and apply a Hybrid Schedule combining two
// simple schedules
HybridGPUSchedule h1(s1, s2, 0.05);
```

Fig. 10. Implementation of `edgeset_apply` operator with a `HybridGPUSchedule` and an example of how the user can create a hybrid schedule combining two simple schedules

is changed at runtime based on the size of the frontier. This optimization is used by applications like BFS and BC especially when dealing with power-law degree distribution graphs where the frontier size varies a lot. Just like RegEx example, we leverage BuildIt's unique ability to have `static<T>` code inside conditions based on `dyn<T>` expressions. To apply direction optimization, the programmer starts by creating a `HybridGPUSchedule` object that combines two separate `Schedule` objects (top-level abstract class of all scheduling classes) with a threshold. The first schedule is applied if the fraction of the active vertices in the input frontier is less than the threshold and the second schedule otherwise. While implementing the `edgeset_apply` operator, the implementation checks if the schedule being applied is a hybrid schedule. If yes, it branches on the size of the input frontier and recursively calls the `edgeset_apply` operator in the `then` and `else` branch with the two schedules respectively. Because BuildIt uses the same syntax for all stages, this implementation is very similar to how one would branch in a graph library keeping the code generation complexity to the minimum. On the other hand the GraphIt DSL compiler has to rely on duplicating, creating `if-then-else` IR nodes and doing specialized code generation when it detects a hybrid schedule further increasing the compiler complexity. Section V shows the number of lines of code required to implement code generation in GraphIt, while Figure 10 shows the complete implementation of hybrid scheduling in BuilDSL. `HybridGPUSchedule` can also be arbitrarily nested to create very complex runtime conditions. The dynamic condition can also be generalized to be based on other runtime parameters instead of just the input frontier size.

F. Runtime Library

Although most of the code generated from BuilDSL is specialized for best performance, some of the code doesn't change a lot across different applications and inputs like vertex set allocation and management, priority queue implementation, I/O, graph transformations among others. To keep the staging complexity to a minimum, we implement this in a runtime library that is linked with the generated code. BuilDSL can insert calls to the routines in the runtime library because BuildIt supports calls to external `dyn<T>` functions.

TABLE I. LINES OF CODE REQUIRED TO IMPLEMENT VARIOUS PARTS OF BUILDSL AND THE GRAPHIT DSL COMPILER. FOR FAIRNESS WE HAVE ONLY COUNTED THE GPU BACKEND AND THE GPU RUNTIME LIBRARY FROM GRAPHIT ALTHOUGH IT ALSO HAS A CPU BACKEND. THE LINES OF CODE FOR BUILDIT HAS BEEN SHOWN SEPARATELY.

Component	BuilDSL	GraphIt
BuildIt	6,808	-
Frontend	-	9,593
Scheduling Language	151	2,401
Midend Analysis and Transformations	-	9,601
Types and Operator Implementation	1,320	-
Code Generation	550	2,188
Runtime Library	1,570	2,470
Total	3,591	26,253
Total (compiler only)	2,021	23,783

Finally, although the implementation techniques explained in this section all talk about how they apply to the graph domain, it is very easy to see how these can be applied to DSLs from other domains. The generalized DSL framework part of BuilDSL provides the basic blocks like extracting CUDA kernels, support for Kernel Fusion, etc. The scheduling, analysis, and transformations can be made specific to the domain by extending the types.

Our companion paper [20] shows the implementation and schedules for five graph applications implemented with BuilDSL and discusses in detail the generated CUDA code for each of them.

V. EVALUATIONS

In this section, we demonstrate the performance of the code generated from BuilDSL and how it compares against the code generated from the state-of-the-art GPU graph compiler GraphIt. We will also compare the lines of code required to implement BuilDSL with that of the GraphIt DSL compiler.

A. Implementation Complexity

The primary metric for the quality of a DSL is the performance of the code it generates and how it stacks up against other library and compiler frameworks. However, with the increasing number of DSLs coming out tailored to very specific domains, it is also important to see how easy or difficult it is to implement the compiler for the particular DSL. In this paper, we make the case that using staging and BuilDSL makes it significantly easier to rapidly prototype and test a DSL without having to compromise on performance. We will start with comparing the number of lines of code required to implement various parts of BuilDSL explained in Section IV with the lines of code required to implement the corresponding features in the GraphIt DSL compiler as passes. Table I shows the various components and the lines of C++ code required to implement them.

Looking at the total lines of code we can see right away that BuilDSL takes only about 15% of the lines of code required to implement a full DSL compiler. This difference becomes

even more prominent when we ignore the number of lines of code that go into the runtime library (8.4%). BuildIt a library that BuilDSL primarily depends upon currently stands at 6,808 lines of C++ code and has been shown separately in the table. Although this a large component of BuilDSL, BuildIt is a generalized C++ library for multi-stage execution and is completely independent of any DSL implementation. For fairness reasons, we have counted only the lines of code of GraphIt that are required for the GPU backend and the GPU runtime library although it also implements a CPU backend. The bulk of the implementation complexity of GraphIt lies in the front-end and the mid-end. This is because the frontend has a tokenizer, a parser, front-end IR definitions which BuilDSL doesn't have to implement because the DSL is implemented embedded in C++ on top of BuildIt and its types. The mid-end of GraphIt has the IR definitions and the bulk of the analysis and transformation passes which implement visitors over the IR and manipulate it. This mode of transformation takes up more lines of code compared to the staging-based specialization in BuilDSL as part of the types and operator implementation. Finally, GraphIt's GPU backend implementation is also huge because it has to manually generate code as strings by visiting every IR node for both GPUs and the host. The backend also has to do specialized code generation for transferring data between hosts and GPUs and implement Kernel Fusion. The code generation in BuilDSL on the other hand is completely handled by BuildIt and hence is for free. The only small part that we have to implement is the CUDA extraction passes and the Kernel Fusion infrastructure as part of the generalized DSL framework. Once again, this implementation is very generic and can be reused across DSLs. Finally, the runtime library in BuilDSL is slightly smaller than that of GraphIt's because GraphIt implements all load balancing as a series of runtime library routines whereas BuilDSL takes care of load balancing in the operator implementation itself.

The GraphIt DSL compiler is also a large project created over more than 3 years and maintained by at least 4 developers. BuilDSL on the other hand was developed entirely by just one developer over a few months. Although BuilDSL borrows a lot of design decisions and optimization tricks from GraphIt it still shows that BuilDSL can significantly reduce the time for prototyping and testing the DSL especially for tricky backends like GPUs.

B. Performance Evaluations

Now that we have shown how BuilDSL makes it easier to implement DSLs over traditional compiler techniques, we want to demonstrate that the code generated from BuilDSL is on-par with the code from other frameworks in terms of performance. Since GraphIt already shows that its performance beats or very closely matches the performance from other state-of-the-art frameworks like Gunrock [21], GSwitch [22] and SEP-Graph [23], we only compare the performance of the code generated from BuilDSL with that of GraphIt. The relative performance to other frameworks can be calculated from the evaluations in the GraphIt paper [9].

TABLE II. GRAPH INPUTS USED FOR EVALUATION. THE EDGE COUNT SHOWS THE NUMBER OF UNDIRECTED EDGES.

Graph Input	Vertex count	Edge count
soc-orkut [24] (OK)	2,997,166	212,698,418
soc-twitter-2010 [24] (TW)	21,297,772	530,051,090
soc-LiveJournal [25] (LJ)	4,847,571	85,702,474
soc-sinaweibo [24] (SW)	58,655,849	522,642,066
hollywood-2009 [25] (HW)	1,139,905	112,751,422
indochina-2004 [25] (IC)	7,414,865	301,969,638
road_usa [26] (RU)	23,947,347	57,708,624
road_central [25] (RC)	14,081,816	33,866,826
roadNet-CA [25] (RN)	1,971,281	5,533,214

We benchmark 5 applications - PageRank (PR), Breadth First Search (BFS), Single Source Shortest Path with Delta Stepping (SSSP), Betweenness Centrality (BC) and Connected Components (CC). These applications include a mix of topology-driven, data-driven, and priority-driven algorithms that fully stress the various aspects of BuilDSL. Similarly, we use 9 different graph datasets with these applications (shown in Table II). These datasets include a mix of power-law degree and bounded degree high-diameter graphs which have different characteristics and sparsity patterns. We run our evaluations on an NVIDIA Volta V100 (32 GB memory, 4MB L2 cache, and 80 SMs) GPU. Both for GraphIt and BuilDSL we tune the best schedule for each algorithm and graph input for a fair comparison (since the programming interface and the scheduling language for both BuilDSL and GraphIt is similar, these schedules are very close to each other).

Figure 11 shows the execution time of the code generated from BuilDSL normalized to the execution time of the code generated from GraphIt across the 5 applications and 9 graph inputs. In both cases the best schedule is chosen for each algorithm and graph input. Across all applications graphs, BuilDSL has a 1.03× geometrical mean speedup with an at most 8.38% slowdown and up to 25.15% speedup. BuilDSL can match the performance of GraphIt because we support the exact same space of optimization choices as that of GraphIt.

For topology-driven applications like PR and CC, where all the edges are processed each round, BuilDSL switches the layout of the graph data structure to COO which is more suitable and easier to load balance on GPUs. PR also benefits from Edge Blocking which BuilDSL supports the same way as GraphIt. Applications like BFS and BC whose performance is very susceptible to variations in sizes of the active vertex set across rounds, greatly benefit from the hybrid scheduling in BuilDSL which as explained in Section IV is implemented as specialization based on `dyn<T>` expressions as opposed to just `static<T>` expression. We apply direction optimization and changing data structures for the vertex sets to improve the performance of BFS and BC. Both these algorithms also apply Kernel Fusion for bounded-degree large diameter graphs to reduce the kernel launch overhead which is made possible by the infrastructure support from the generalized DSL framework in BuilDSL. SSSP uses the priority queue abstraction that is mostly implemented by calls to a runtime library, but the actual vertex set processing is done by the same operators in BuilDSL.

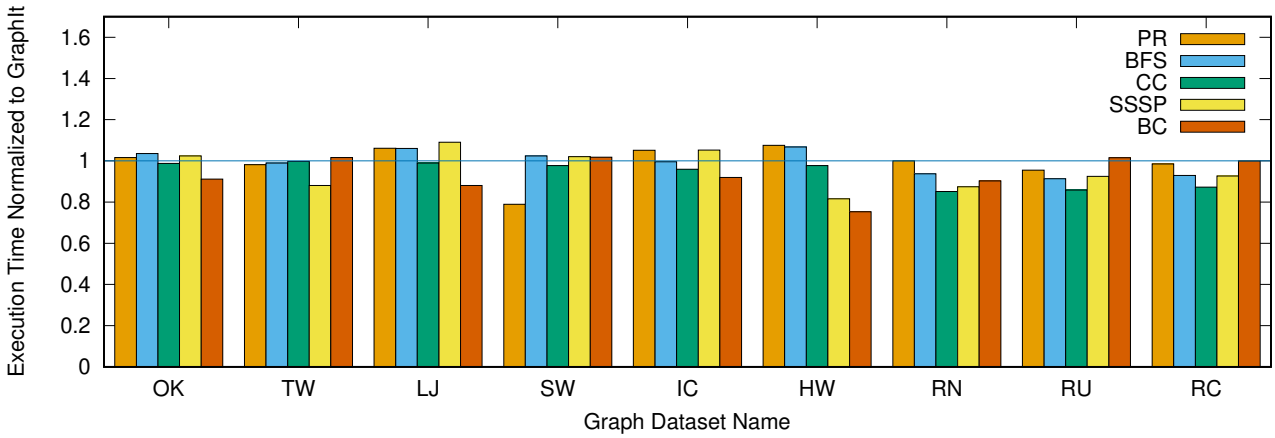

Fig. 11. Execution time of code generated from BuilDSL normalized to the execution time of the code generated from GraphIt across 5 applications and 9 graph inputs (lower the better). The best schedule for each algorithm and input is chosen in both the cases

SSSP also uses kernel fusion for road graphs like BC and BFS.

Our evaluation shows that all the optimizations that we have implemented in BuilDSL are critical for performance across commonly used applications. The GraphIt GPU paper has shown that the lack of some of these optimizations can lead to up to 1000× slowdown in some cases.

VI. Related Works

High-performance DSLs have played a critical role in optimizing applications from several domains. Halide [11] a DSL for image processing applications was the first that introduced the idea of separating algorithms from schedules. Other DSLs like GraphIt [7]–[10], TACO [27], Tiramisu [5], Taichi [28] have applied the same ideas to other high-performance application domains. Lightweight Modular Staging [29] and other works [30]–[38] have contributed to the idea of staging and its applications to other domains. Works like Forge [39], Jet [13], StagedSAC [14] and others [1], [40], [41] either apply the idea of Futamura Projections [17] to interpreters to create DSL compilers or implement end-to-end DSL frameworks with staging. A lot of these DSLs also target high-performance domains and generate code for different architectures like GPUs and CPUs. BuildIt [16] extends ideas from the BUILDER [42] framework and brings the idea of multi-stage programming with a library approach to C++. This is critical because most high-performance DSLs want to generate C or C++ and BuildIt brings the two stages together. Delite [43] and other related works [44]–[46] have created building blocks for components of high-performance DSLs that make it easy to rapidly prototype a DSL for a new domain. MLIR [47] is a DSL framework that makes it easy to reuse compiler analysis and transformation passes across domains with its extensible IR. But MLIR still requires a lot of compiler knowledge and is not necessarily easy to use for domain experts. Many high-performance graph libraries and frameworks have been built to target different platforms like Ligra [18], [48], Gunrock [21], GSwitch [22], SEP-Graph [23], IrGL [49] among others [50]–[68]. GraphIt [7]–[10] is a high-performance graph DSL that can generate code for CPUs, GPUs, multi-cores and other hardware accelerators with its unique GraphIR intermediate representation and scheduling language. The GraphIt GPU backend combines 7 orthogonal dimensions of scheduling that enables it to generate the code most suitable for each algorithm and input. The GraphIR and GraphVMs allow for code reuse when developing different backends but the framework is very large and not as accessible for domain experts.

VII. Conclusion

We present BuilDSL, a DSL framework built on top of the light-weight C++ staging library BuildIt. BuilDSL provides the framework support for GPU code generation, kernel fusion, analyses, and specialization based on schedules all of which are commonly required to implement high-performance DSL. We build a graph DSL on top of BuilDSL that generates high-performance GPU code that matches the performance of the code generated from the state-of-the-art graph DSL compiler GraphIt with a small fraction (8.4%) of the lines of the code.

VIII. Data Availability Statement

BuilDSL and all its components are available open-source under the MIT license [69]. The procedure to obtain the dataset and reproduce the results in this paper are also included. All of the source code for BuilDSL including the generalized DSL framework and the graph DSL implementation can be found under the `BuilDSL/include` and `BuilDSL/src` directories adding up to 2021 lines of C++ code. The implementation for the five applications with their schedules are under the `BuilDSL/apps` directory. Our companion paper [20] explains in detail the implementation of these applications and the generated code for each of them.

Acknowledgments

This research was supported by DARPA SDH Award #HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019.

[3] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI*, 2018.

[5] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proc. CGO*, 2019.

[6] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," in *Proc. OOPSLA*, 2017.

[7] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," in *Proc. OOPSLA*, 2018.

[8] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with graphit," in *Proc. CGO*, 2020.

[9] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe, "Compiling graph applications for gpus with graphit," in *Proc. CGO*, 2021.

[10] A. Brahmakshatriya, E. Furst, V. Ying, C. Hsu, C. Hong, M. Ruttenberg, Y. Zhang, D. C. Jung, D. Richmond, M. Taylor, J. Shun, M. Oskin, D. Sanchez, and S. Amarasinghe, "Taming the zoo: A unified graph compiler framework for novel architectures," in *Proc. ISCA*, 2021.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. PLDI*, 2013.

[12] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun, "Forge: Generating a high performance dsl implementation from a declarative specification," in *Prof. GPCE*, 2013.

[13] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky, "Jet: An embedded dsl for high performance big data processing," in *Proc. BigData*, 2012.

[14] V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky, "Stagedsac: A case study in performance-oriented dsl development," in *Proc. PEPM*, 2012.

[15] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proc. PPoPP*, 2011.

[16] A. Brahmakshatriya and S. Amarasinghe, "Buildit: A type based multistage programming framework for code generation in c++," in *Proc. CGO*, 2021.

[17] Y. Futamura, "Partial evaluation of computation process, revisited," *HOSC*, 1999.

[18] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.

[19] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, 2015.

[20] A. Brahmakshatriya and A. Saman, "Application suite for a graph dsl in buildsl," 2022.

[21] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *SIGPLAN Not.*, 2016.

[22] K. Meng, J. Li, G. Tan, and N. Sun, "A pattern based algorithmic autotuner for graph processing on GPUs," in *Proc. PPoPP*, 2019.

[23] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "SEP-Graph: Finding shortest execution paths for graph processing under a hybrid framework on GPU," in *Proc. PPoPP*, 2019.

[24] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. AAAI*, 2015.

[25] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, 2011.

[26] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS implementation challenge - shortest paths," http://www.dis.uniroma1.it/challenge9/.

[27] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. W. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe, "Simit: A language for physical simulation," *ACM Trans. Graph.*, 2016.

[28] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: A language for high-performance computation on spatially sparse data structures," *ACM Trans. Graph.*, 2019.

[29] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," *SIGPLAN Not.*, 2010.

[30] W. Taha, "A gentle introduction to multi-stage programming," in *Proc. Domain-Specific Program Generation*, 2004.

[31] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," in *Proc. PEPM*, 1997.

[32] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," *SIGPLAN Not.*, 1997.

[33] C. Calcagno, W. Taha, L. Huang, and X. Leroy, "Implementing multi-stage languages using asts, gensym, and reflection," in *Proc. GPCE*, 2003.

[34] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha, "Mint: Java multi-stage programming using weak separability," *SIGPLAN Not.*, 2010.

[35] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic, "A monadic approach for avoiding code duplication when staging memoized functions," in *Proc. PEPM*, 2006.

[36] Y. Kameyama, O. Kiselyov, and C.-c. Shan, "Closing the stage: From staged code to typed closures," in *Proc. PEPM*, 2008.

[37] Y. Kameyama, O. Kiselyov, and C.-c. Shan, "Shifting the stage: Staging with delimited control," in *Proc. PEPM*, 2009.

[38] G. Wei, Y. Chen, and T. Rompf, "Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming," in *Proc. OOPSLA*, 2019.

[39] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun, "Forge: Generating a high performance dsl implementation from a declarative specification," *SIGPLAN Not.*, 2013.

[40] T. Rompf and N. Amin, "Functional pearl: A sql to c compiler in 500 lines of code," *SIGPLAN Not.*, 2015.

[41] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun, "Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns," in *Proc. CGO*, 2016.

[42] Stanford Compiler Group, "The builder library, a tool to construct or modify suif code within the suif compiler," 1994. [Online]. Available: https://suif.stanford.edu/suif/suif1/docs/builder_toc.html

[43] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," *SIGPLAN Not.*, 2011.

[44] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun, "Building-blocks for performance oriented dsls," *EPTCS*, 2011.

[45] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proc. PACT*, 2011.

[46] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A generic parallel collection framework," in *Proc. Euro-Par*, 2011.

[47] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *Proc. CGO*, 2021.

[48] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. SPAA*, 2017.

[49] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," *SIGPLAN Not.*, 2016.

[50] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable GPU graph traversal," *ACM Trans. Parallel Comput.*, 2015.

[51] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *Proc. OOPSLA*, 2016.
[52] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proc. PPoPP*, 2017.
[53] H. Liu and H. H. Huang, "SIMD-x: Programming and processing of graph algorithms on GPUs," in *Proc. USENIX ATC*, 2019.
[54] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for GPU-friendly graph processing," in *Proc. ASPLOS-XXIII*, 2018.
[55] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proc. HPDC*, 2014.
[56] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proc. PACT*, 2015.
[57] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. HIPC*, 2007.
[58] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. PPoPP*, 2011.
[59] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," in *Proc. SC*, 2015.
[60] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin, "Frog: Asynchronous graph processing on GPU with hybrid coloring model," *TKDE*, 2017.
[61] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "Multigraph: Efficient graph processing on GPUs," in *Proc. PACT*, 2017.
[62] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. IPDPS*, 2014.
[63] J. Soman, K. Kishore, and P. Narayanan, "A fast GPU algorithm for graph connectivity," in *Proc. IPDPSW*, 2010.
[64] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *Prof. IPDPS*, 2013.
[65] S. Che, "GasCL: A vertex-centric graph model for GPUs," in *Proc. HPEC*, 2014.
[66] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to GPUs," in *Proc. SIGMOD*, 2016.
[67] A. Gaihre, Z. Wu, F. Yao, and H. Liu, "XBFS: eXploring runtime optimizations for breadth-first search on GPUs," in *Proc. HPDC*, 2019.
[68] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *Proc. PACT*, 2017.
[69] A. Brahmakshatriya and S. Amarasinghe, "Replication package for the paper: Graphit to cuda compiler in 2021 loc: A case for high-performance dsl implementation via staging with buildsl." [Online]. Available: https://n2t.net/10.5281/zenodo.5788581