

GraphIt to CUDA compiler in  
2021 LOC: A case for high-  
performance DSL implementation  
via staging with **BuildSL**

**Ajay Brahmakshatriya**

Saman Amarasinghe

CSAIL, MIT

4th April 2022

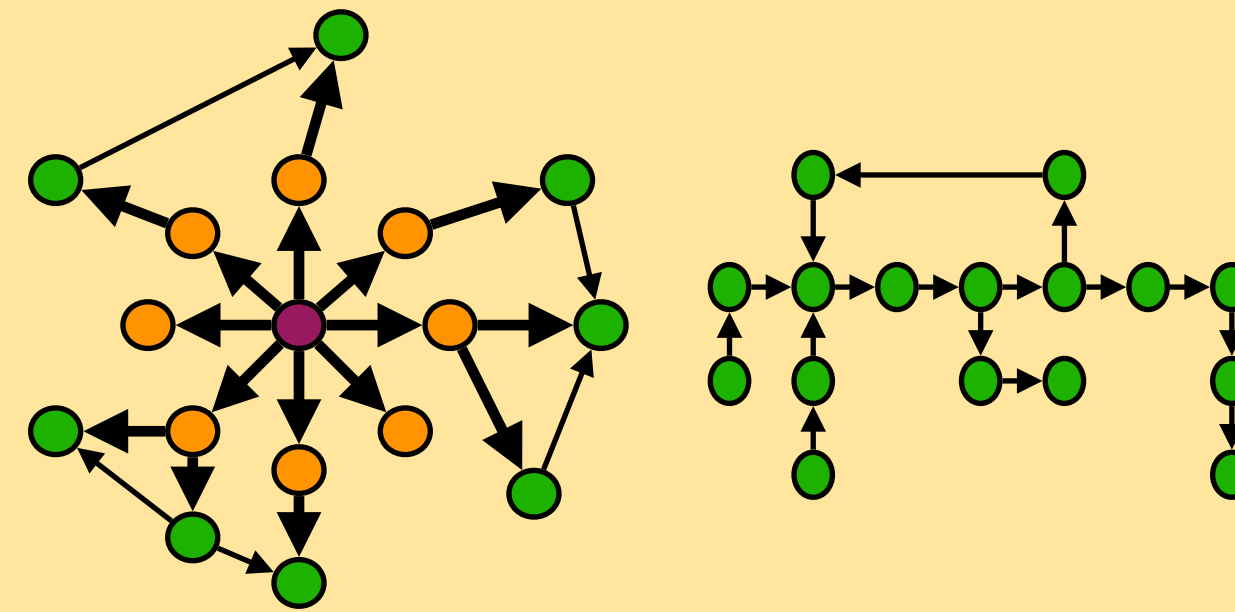
# DSLs enable high-performance

- Performance critical domains require *domain-specific* optimizations

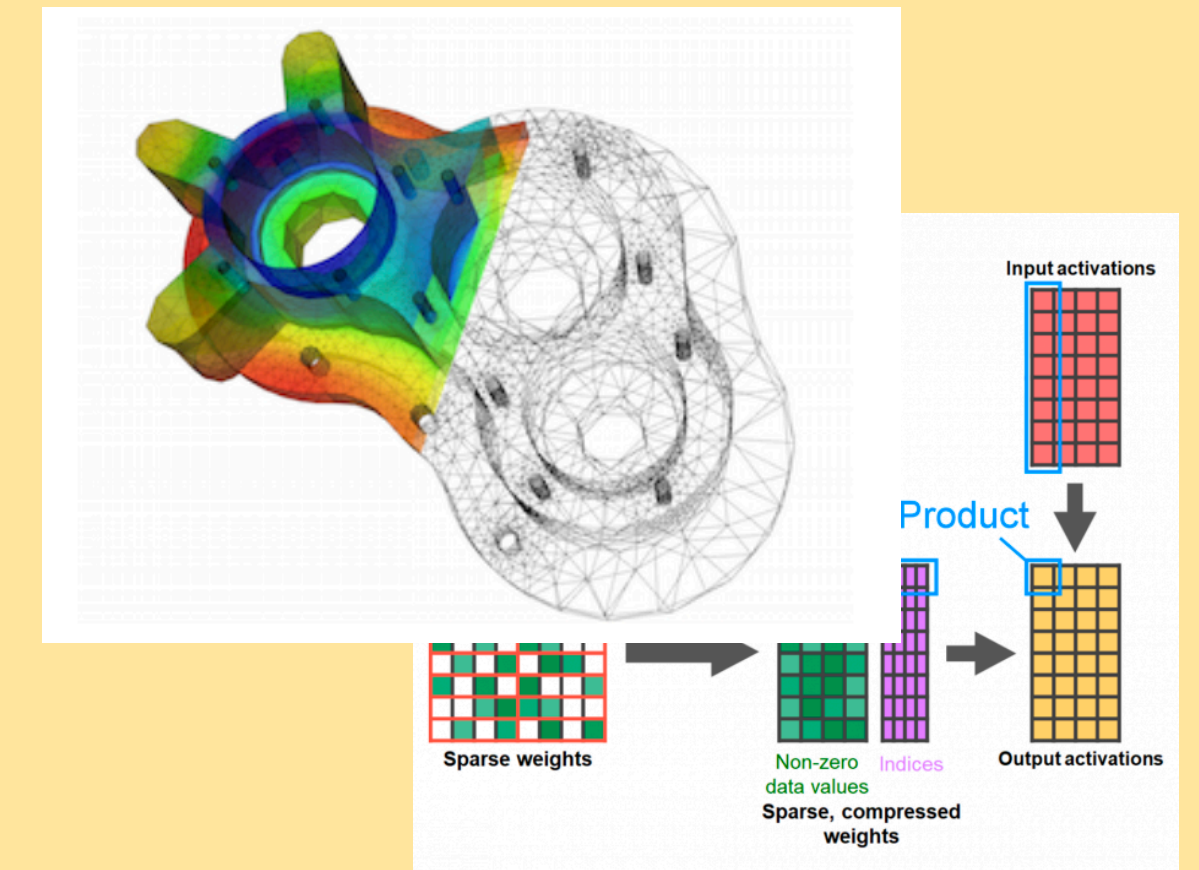
# DSLs enable high-performance



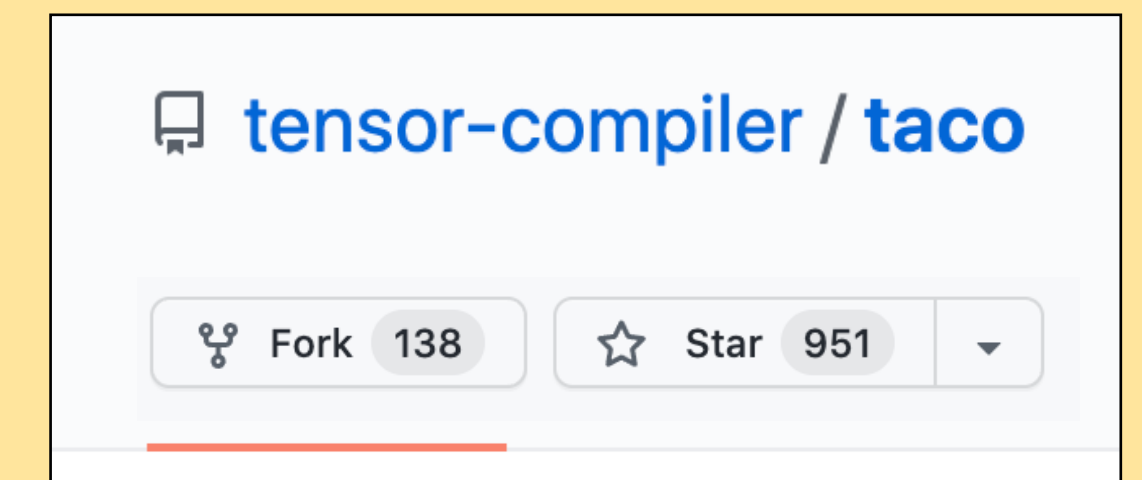
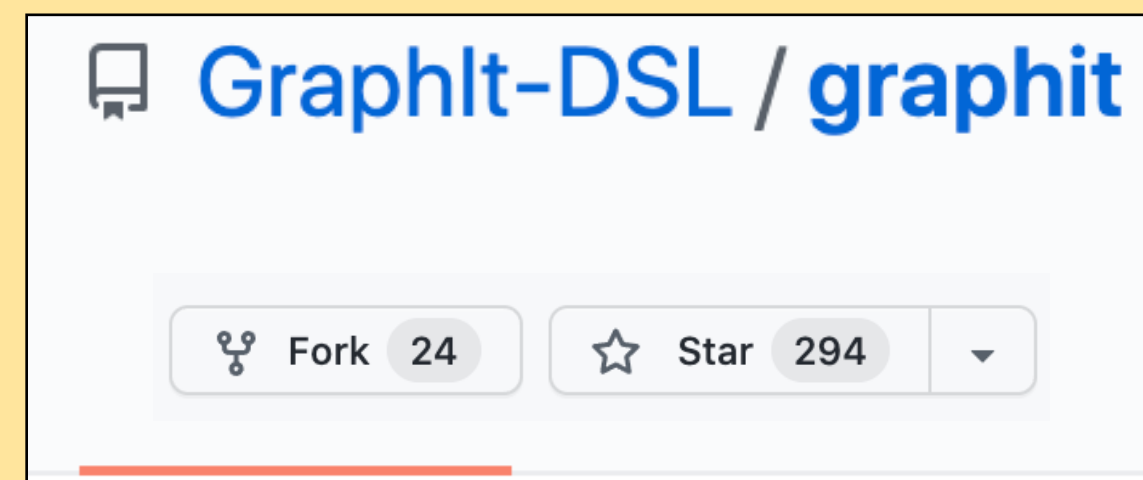
Image Processing



Graph Processing



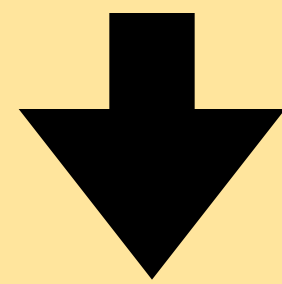
Sparse Array Processing



- Built **for** domain-experts *with* domain-experts!

# DSLs enable high-performance

- Built **for** domain-experts *with* domain-experts!

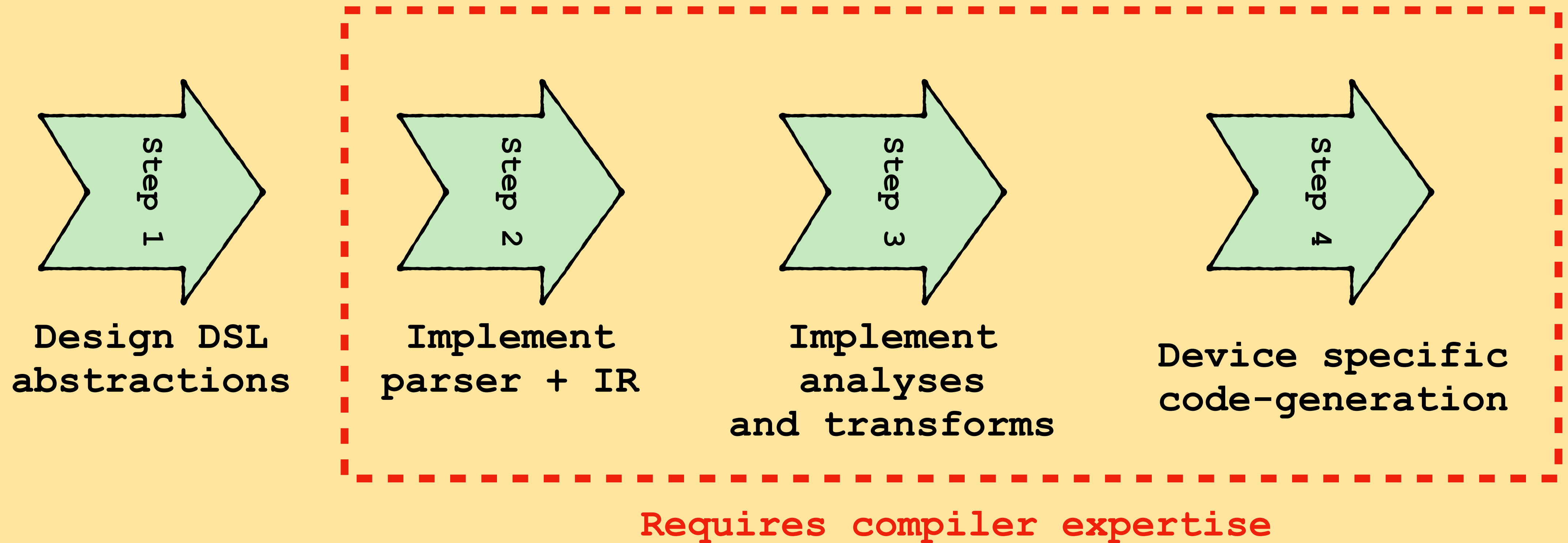


- Built **for** domain-experts **by** domain-experts!

## BuildIt!

# Writing DSLs is hard

Typical DSL development cycle

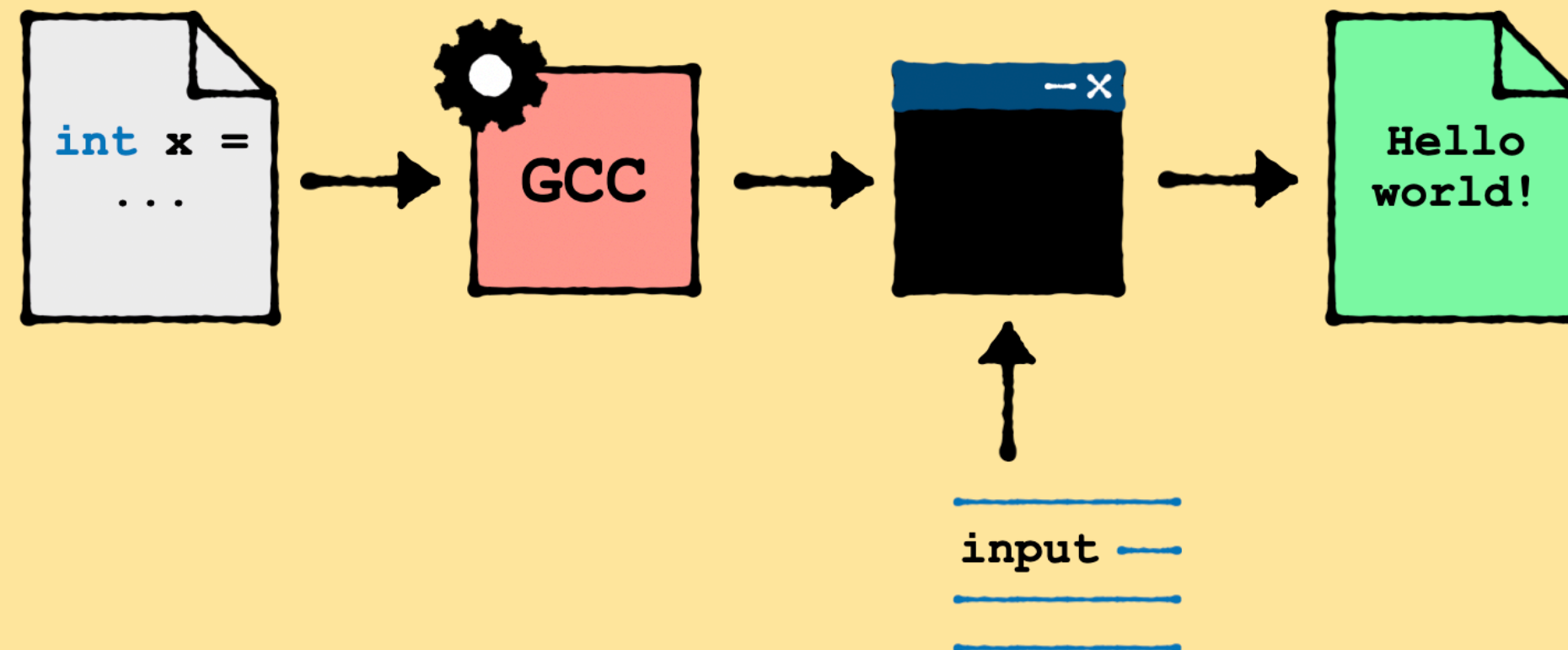


- Domain experts can and do write high-performance libraries

# BuildIt

## Multi-stage programming in C++

- BuildIt brings multi-stage programming to C++ in a lightweight way

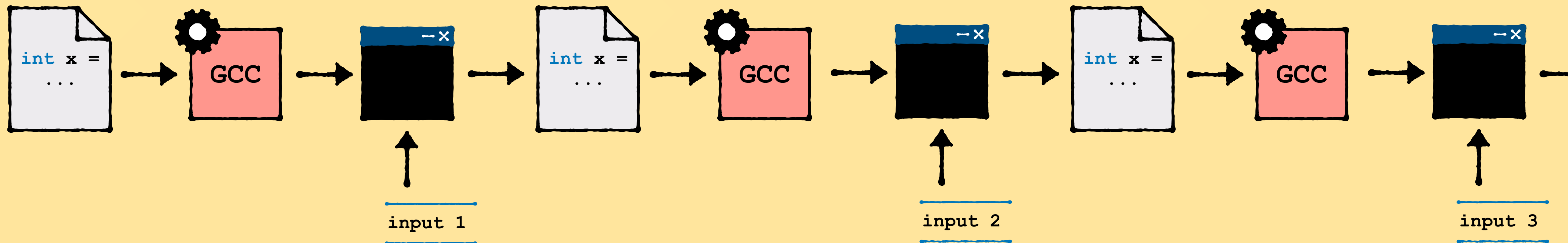
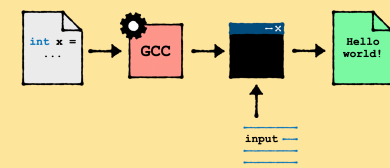




# BuildIt

## Multi-stage programming in C++

- BuildIt brings multi-stage programming to C++ in a lightweight way



# BuildIt

## Multi-stage programming in C++

- BuildIt uses types to differentiate stages/binding times

`dyn_var<T>`      `static_var<T>`



# BuildIt

## Multi-stage programming in C++

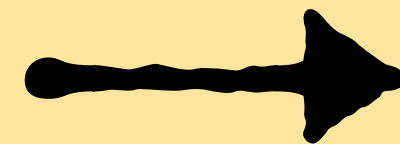
- BuildIt uses types to differentiate stages/binding times

`static_var<T>`

`dyn_var<T>`

Evaluation of expressions of type `dyn_var<T>` is delayed to next stage

```
dyn<int> x = 0;  
dyn<float> y = 3.14f;  
  
x = y + 2.71f;  
  
if (x > 8.0f)  
    y = 0;  
else  
    y = 1;
```



```
int x = 0;  
float y = 3.14f;  
  
x = y + 2.71f;  
  
if (x > 8.0f)  
    y = 0;  
else  
    y = 1;
```

# BuildIt

## Multi-stage programming in C++

- BuildIt uses types to differentiate stages/binding times

`dyn_var<T>`

`static_var<T>`

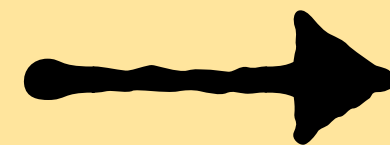
`static_var<T>` is completely evaluated in the current stage

```
static<int> x = 0;

dyn<float> y = 0.0f;

y = y + x;

if (x > 1)
    y = 0;
else
    y = 1;
```



```
float = 3.14f;

y = y + 0; // x = 0

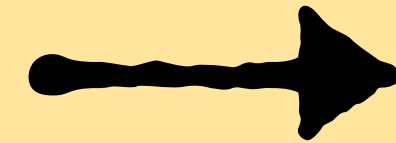
// no condition on x

y = 1;
```

# BuildIt

## Specialization with `static_var(s)`

```
dyn_var<int> power_f(dyn_var<int> base,  
    static_var<int> exponent) {  
  
    dyn_var<int> res = 1, x = base;  
  
    while (exponent > 1) {  
        if (exponent % 2 == 1)  
            res = res * x;  
        x = x * x;  
        exponent = exponent / 2;  
    }  
    return res * x;  
}
```



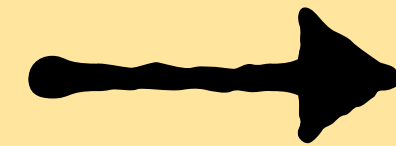
```
int power_5 (int arg0) {  
    int var0 = arg0;  
    int var1 = 1;  
    int var2 = var0;  
    var1 = var1 * var2;  
    var2 = var2 * var2;  
    var2 = var2 * var2;  
    int var3 = var1 * var2;  
    return var3;  
}
```

```
exponent = 5
```

# BuildIt

## Specialization with `static_var(s)`

```
dyn_var<int> power_f(dyn_var<int> base,  
    static_var<int> exponent) {  
  
    dyn_var<int> res = 1, x = base;  
  
    while (exponent > 1) {  
        if (exponent % 2 == 1)  
            res = res * x;  
        x = x * x;  
        exponent = exponent / 2;  
    }  
    return res * x;  
}
```



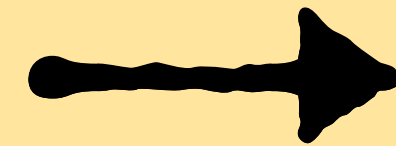
```
int power_10 (int arg0) {  
    int var0 = arg0;  
    int var1 = 1;  
    int var2 = var0;  
    var2 = var2 * var2;  
    var1 = var1 * var2;  
    var2 = var2 * var2;  
    var2 = var2 * var2;  
    int var3 = var1 * var2;  
    return var3;  
}
```

`exponent = 10`

# BuildIt

## Specialization with `static_var(s)`

```
dyn_var<int> power_f(dyn_var<int> base,  
    static_var<int> exponent) {  
  
    dyn_var<int> res = 1, x = base;  
  
    while (exponent > 1) {  
        if (exponent % 2 == 1)  
            res = res * x;  
        x = x * x;  
        exponent = exponent / 2;  
    }  
    return res * x;  
}
```

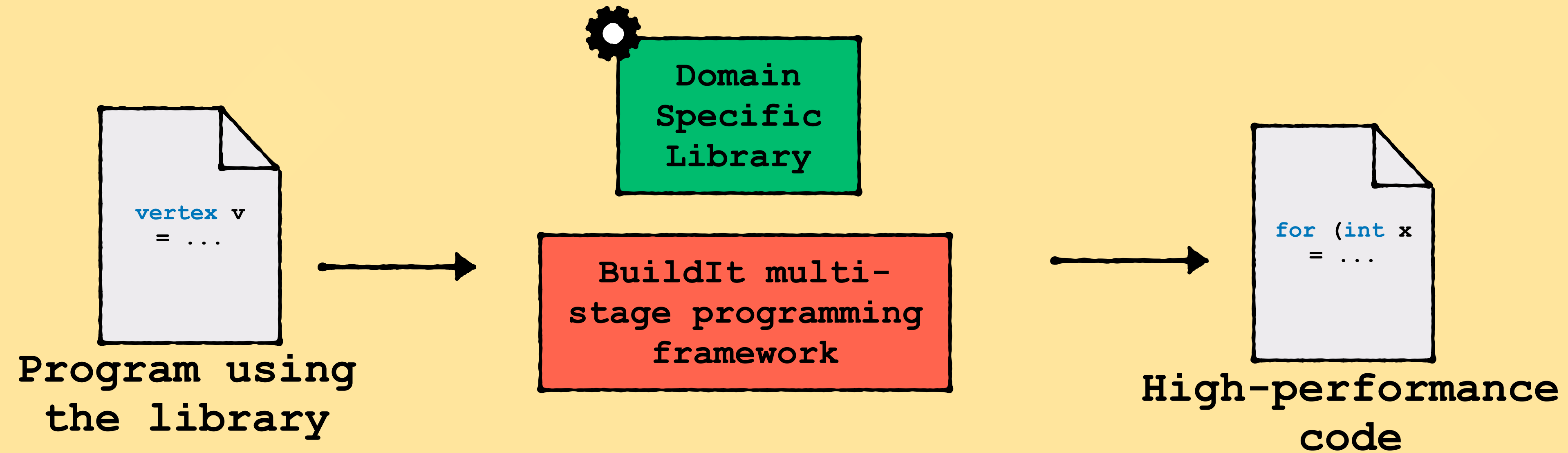


```
int power_15 (int arg0) {  
    int var0 = arg0;  
    int var1 = 1;  
    int var2 = var0;  
    var1 = var1 * var2;  
    var2 = var2 * var2;  
    var1 = var1 * var2;  
    var2 = var2 * var2;  
    var1 = var1 * var2;  
    var2 = var2 * var2;  
    int var3 = var1 * var2;  
    return var3;  
}
```

`exponent = 15`

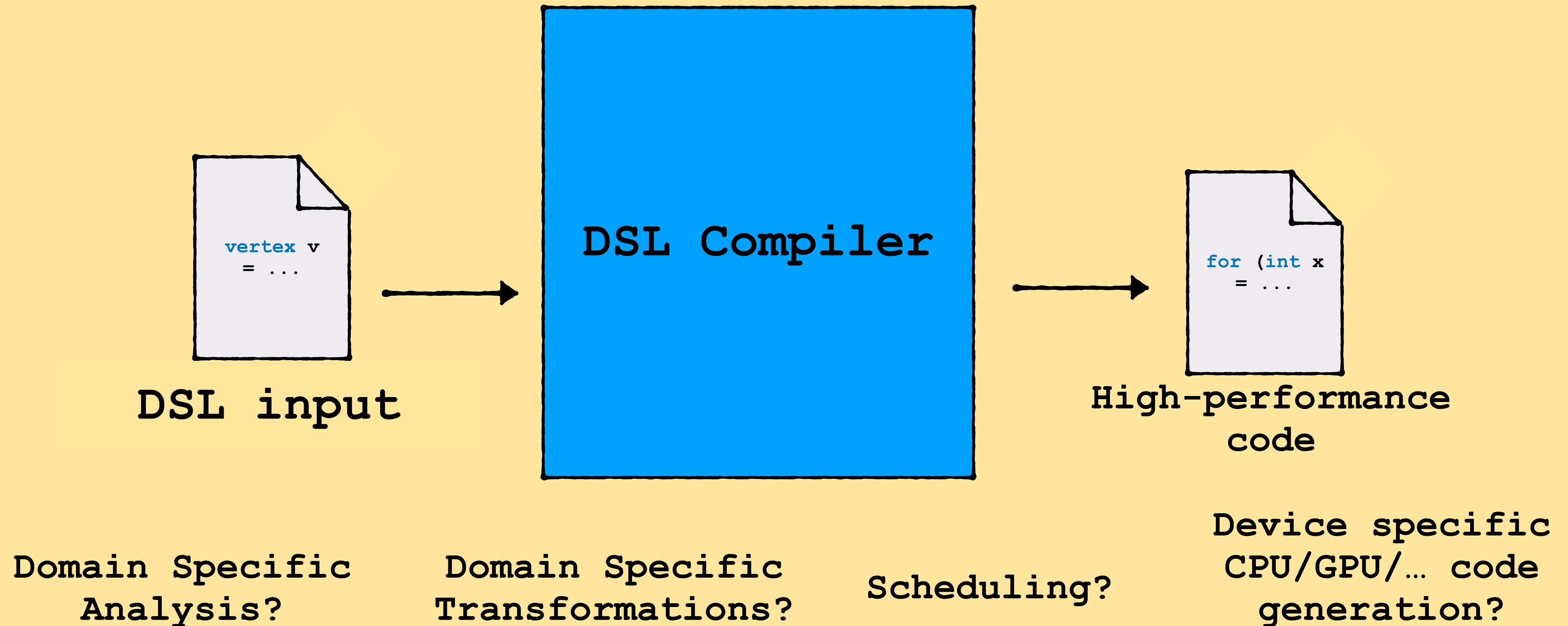
# Staging libraries with BuildIt

A simple intuition



# Staging libraries with BuildIt

A simple intuition





# Building a DSL demo

## Language for Einsum expressions

- Implement a DSL for Einsum expressions

$$\mathbf{A}[j] = \mathbf{B}[j][k] * \mathbf{C}[k]$$

- Scheduling and parallelize for CPUs and GPUs
- Implement constant propagation analysis + transformation

# Building a DSL demo

Language for Einsum expressions

$$A[j] = B[j][k] * C[k]$$

shorthand for

$$\forall j, A[j] = \sum_{k=0}^N B[j][k] * C[k]$$

- Implement loop nests for LHS and RHS
- Implement sum reduction

# Building a DSL demo

## Language for Einsum expressions

$$\forall j, A[j] = \sum_{k=0}^N B[j][k] * C[k]$$

```
struct Tensor {
    float * data; std::vector<int> dims;
    operator = (T rhs_expr) {
        if (this->rank == 1) {
            for (int i = 0; i < this->dims[0]; i++) {
                data[i] = evaluate_rhs(rhs_expr, i);
            }
        } else if (this->rank == 2) {
            for (int i = 0; i < this->dims[0]; i++) {
                for (int j = 0; j < this->this[1]; j++) {
                    data[I * dim[0] + j] = evaluate_rhs(rhs_expr, i, j);
                }
            }
        } else if
            ...
    }
};
```

# Building a DSL demo

## Language for Einsum expressions

$$\forall j, A[j] = \sum_{k=0}^N B[j][k] * C[k]$$

```
struct Tensor {
    float * data; std::vector<int> dims;

    void recursive_loop(int idx, int offset, T rhs) {
        if (idx < dims.size()) {
            for (int i = 0; i < dims[idx]; i++) {
                recursive_loop(idx + 1, offset + ..., rhs(i));
            }
        } else {
            data[offset] = rhs;
        }
    }

    operator = (T rhs_expr) {
        recursive_loop(0, rhs_expr);
    }
};
```

# Building a DSL demo

## Language for Einsum expressions

$$\forall j, A[j] = \sum_{k=0}^N B[j][k] * C[k]$$

```
struct Tensor {
    float * data; std::vector<int> dims;

    void recursive_loop(int idx, int offset, T rhs) {
        if (idx < dims.size()) {
            for (int i = 0; i < dims[idx]; i++) {
                Same approach for recursively iterating over RHS free variables!
            } else {
                data[offset] = rhs;
            }
        }
    }

    operator = (T rhs_expr) {
        recursive_loop(0, rhs_expr);
    }
};
```

# Building a DSL demo

Source code

Demo source code

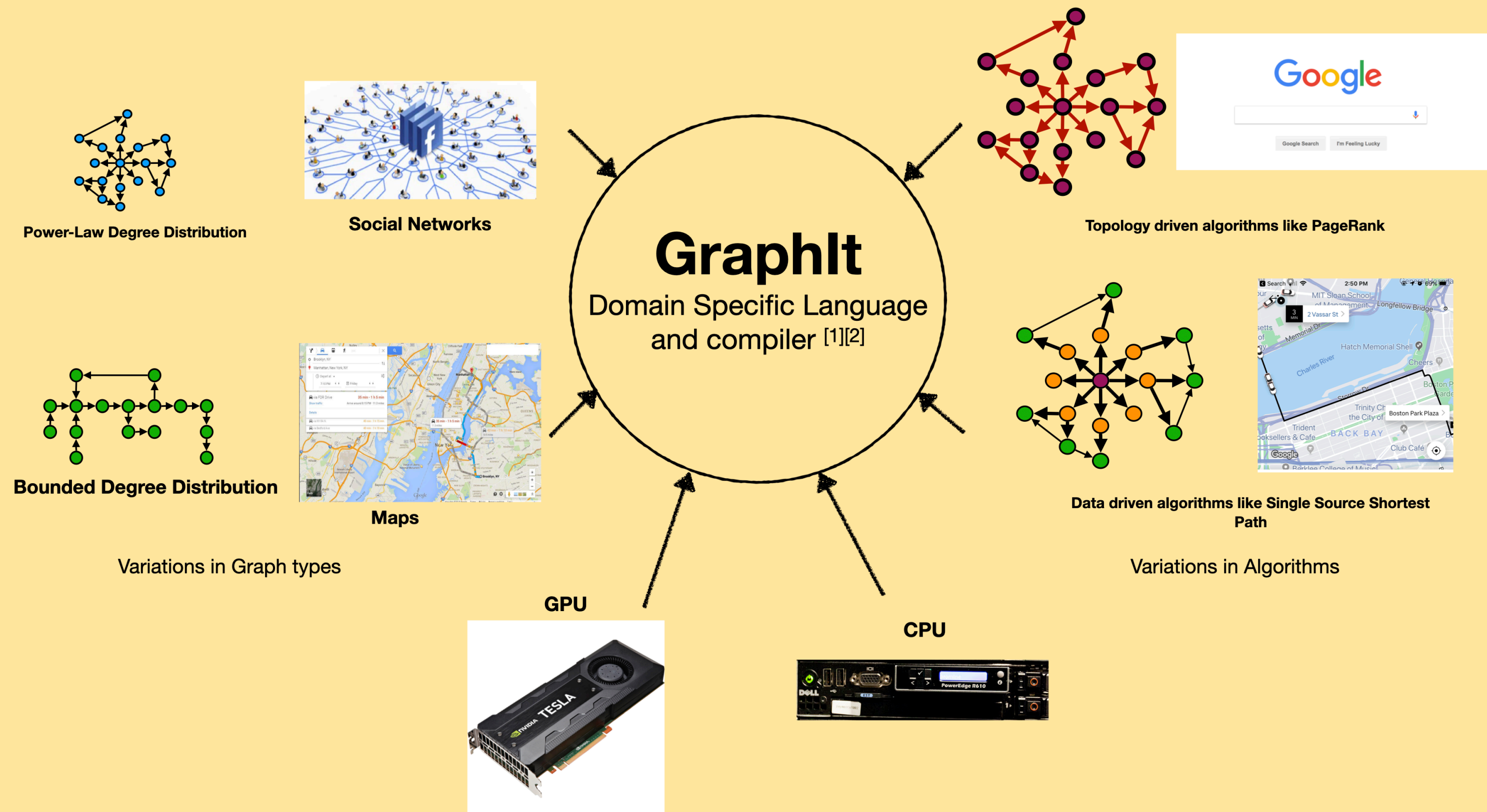
<https://github.com/BuildIt-lang/einsum-lang>

BuildIt

<https://buildit.so>

# GraphIt DSL

- A real world state of the art graph DSL



1. Zhang et.al. , GraphIt: a high-performance graph DSL. Proc. ACM Program. Lang. 2, OOPSLA, Article 121 (November 2018)

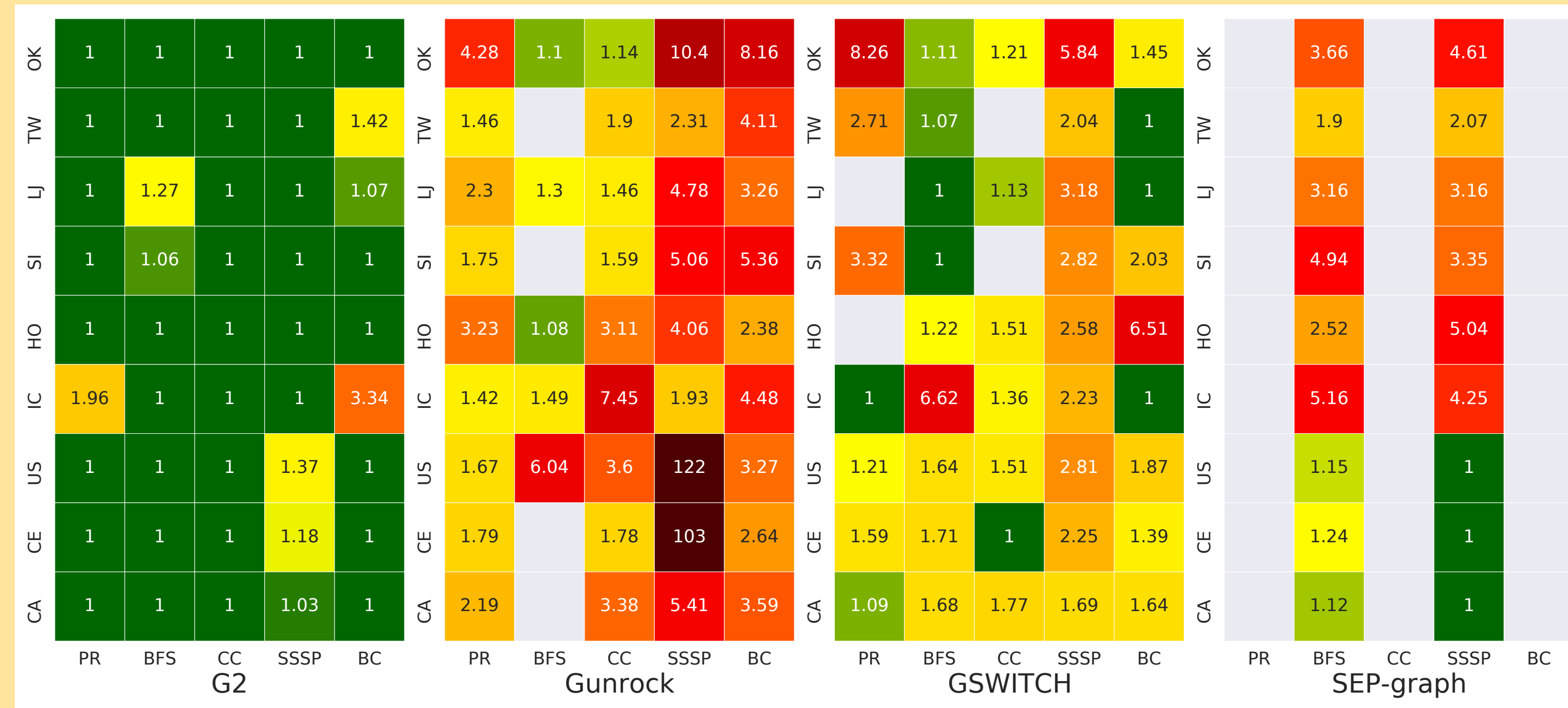
2. Brahmakshatriya et.al. , Compiling Graph Algorithms for GPUs with GraphIt. International Symposium on Code Generation and Optimization (CGO 2021)



# GraphIt DSL on GPUs

State of the art performance

Optimization	Gunrock	GSWITCH	SEP-Graph	G2
Load Balancing	VERTEX BASED, EDGE BASED, TWC	CM, WM, TWC, STRICT	VERTEX BASED	ETWC, TWC, STRICT, CM, WM, VERTEX BASED, EDGE BASED
Edge Blocking	Not supported	Not supported	Not supported	Supported
Vertex Set Creation	Fused/Unfused	Fused/Unfused	Fused	Sparse Queue/Bitmap/Boomlet
Direction Optimization	Push/Pull/Hybrid	Push/Pull/Hybrid	Push/Pull/Hybrid	Push/Pull/Hybrid
Deduplication	Supported	Not supported	Supported	Supported
Vertex Ordering	Supported	Supported	Supported	Supported
Kernel Fusion	Supported	Not supported	Supported	Supported
Total combinations	48	32	16	<b>576</b>



Largest Scheduling Space

State of the art performance

23,783 lines of compiler code!!!

# GraphIt DSL with BuildIt

```
GraphIt 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;
    ...
  }
}
```

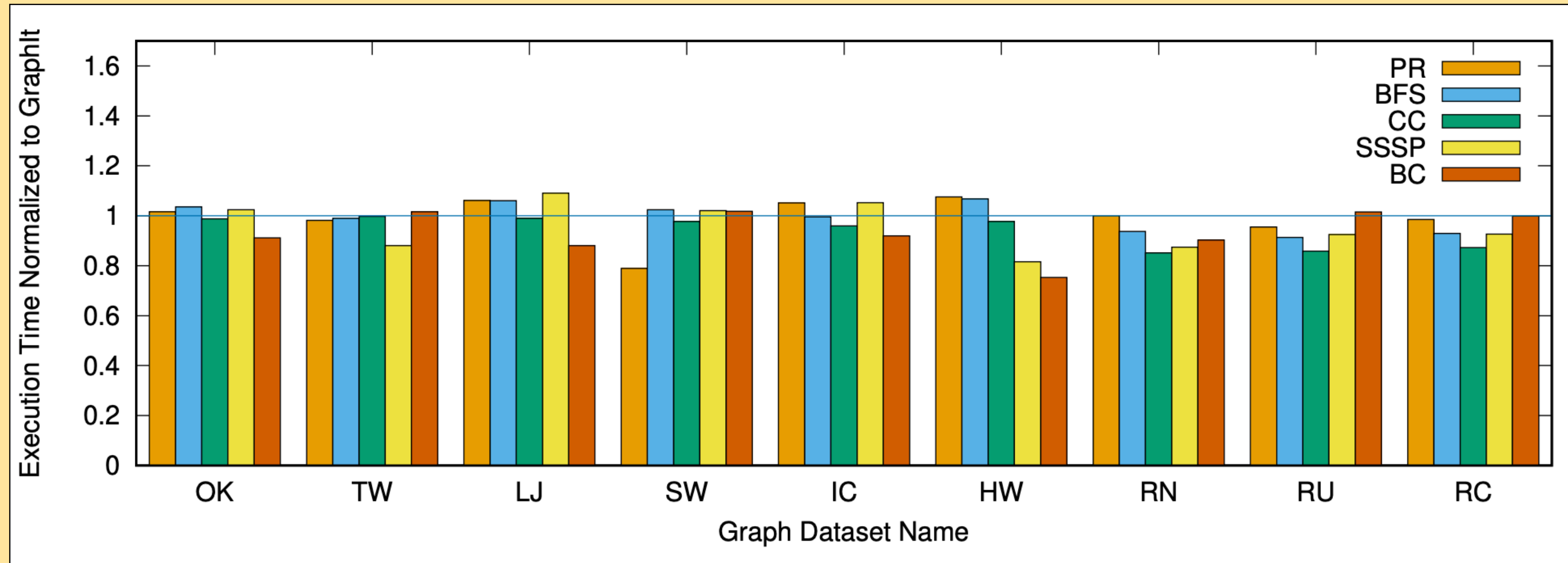
BuildIt annotations for GPU code

```
struct Vertex {
  dyn_var<int> vid;

  // Data flow analysis with static values
  static_var<int> shared;

  operator + (...) {
  }
};
```

Dataflow analysis to insert atomics



Obtain comparable performance with GraphIt compiler for 45 experiments

In just 2021 lines of code on top of BuildIt!

```
SimpleGPUSchedule s1;
s1.configDirection(direction_type::PULL, frontier_rep::BITMAP);
s1.configLoadBalance(load_balance::VERTEX_BASED);
s1.configFrontierCreation(frontier_rep::BITMAP);
edgeset_apply(s1).from(frontier).apply_modified(...);
```

Support generating all 576 combinations of schedules

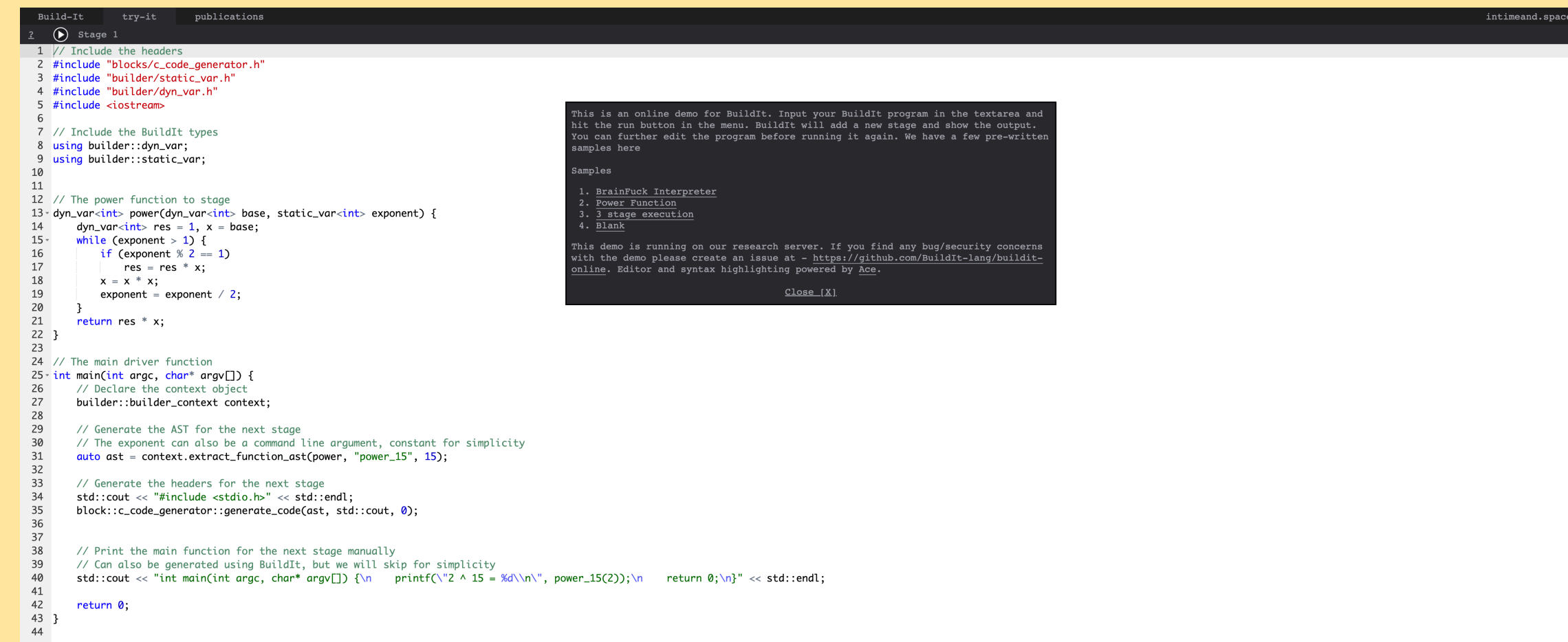
```
// Specialization based on current context
if (current_context == HOST) {
  current_context = DEVICE;
  builder::annotate("CUDA_KERNEL");
  for (dyn<int> ct = 0; ct < MAX_CTA; ct = ct + 1) {
    for (dyn<int> t = 0; t < MAX_T; t = t + 1) {
      dyn<int> tid = ct * MAX_T + t;
      ...
    }
  }
  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();
}
```

Fusing multiple operators into single kernel to avoid launch overheads

# BuildIt

Write high-performance DSLs with ease

BuildIt is available open source  
under the MIT license at  
<https://buildit.so>



The screenshot shows the BuildIt online demo interface. On the left, there is a code editor with C++ source code. The code includes headers, defines a power function using a DSL, and a main driver function. On the right, there is a terminal window showing the output of the program, which is a list of samples: 1. BrainFuck Interpreter, 2. Power Function, 3. 3 stage execution, 4. Blank. Below the terminal, there is a note about the demo running on a research server and a link to create an issue on GitHub.

```
1 // Include the headers
2 #include "blocks/c_code_generator.h"
3 #include "builder/static_var.h"
4 #include "builder/dyn_var.h"
5 #include <iostream>
6
7 // Include the BuildIt types
8 using builder::dyn_var;
9 using builder::static_var;
10
11
12 // The power function to stage
13 dyn_var<int> power(dyn_var<int> base, static_var<int> exponent) {
14     dyn_var<int> res = 1, x = base;
15     while (exponent > 1) {
16         if (exponent % 2 == 1)
17             res = res * x;
18         x = x * x;
19         exponent = exponent / 2;
20     }
21     return res * x;
22 }
23
24 // The main driver function
25 int main(int argc, char* argv[]) {
26     // Declare the context object
27     builder::builder_context context;
28
29     // Generate the AST for the next stage
30     // The exponent can also be a command line argument, constant for simplicity
31     auto ast = context.extract_function_ast(power, "power_15", 15);
32
33     // Generate the headers for the next stage
34     std::cout << "#include <stdio.h>" << std::endl;
35     block::c_code_generator::generate_code(ast, std::cout, 0);
36
37     // Print the main function for the next stage manually
38     // Can also be generated using BuildIt, but we will skip for simplicity
39     std::cout << "int main(int argc, char* argv[]) {\n    printf(\"2 ^ 15 = %d\\n\", power_15(2));\n    return 0;}\n" << std::endl;
40
41     return 0;
42 }
43
44
```

This is an online demo for BuildIt. Input your BuildIt program in the text area and hit the run button in the menu. BuildIt will add a new stage and show the output. You can further edit the program before running it again. We have a few pre-written samples here.

Samples

1. BrainFuck Interpreter
2. Power Function
3. 3 stage execution
4. Blank

This demo is running on our research server. If you find any bug/security concerns with the demo please create an issue at - <https://github.com/BuildIt-lang/buildit-online>. Editor and syntax highlighting powered by Ace.

[Close \[X\]](#)

Try Online: [buildit.so/tryit/](https://buildit.so/tryit/)

Ajay Brahmakshatriya ([ajaybr@mit.edu](mailto:ajaybr@mit.edu))