EigenCFA: Accelerating Flow Analysis with GPUs

Tarun Prabhu, Shreyas Ramalingam, Matthew Might, Mary Hall

University of Utah, Salt Lake City, Utah, USA

{tarunp,sramalin,might,mhall}@cs.utah.edu

Abstract

We describe, implement and benchmark EigenCFA, an algorithm for accelerating higher-order control-flow analysis (specifically, 0CFA) with a GPU. Ultimately, our program transformations, reductions and optimizations achieve a factor of 72 speedup over an optimized CPU implementation.

We began our investigation with the view that GPUs accelerate high-arithmetic, data-parallel computations with a poor tolerance for branching. Taking that perspective to its limit, we reduced Shivers's abstract-interpretive 0CFA to an algorithm synthesized from linear-algebra operations. Central to this reduction were "abstract" Church encodings, and encodings of the syntax tree and abstract domains as vectors and matrices.

A straightforward (dense-matrix) implementation of EigenCFA performed slower than a fast CPU implementation. Ultimately, sparse-matrix data structures and operations turned out to be the critical accelerants. Because control-flow graphs are sparse in practice (up to 96% empty), our control-flow matrices are also sparse, giving the sparse matrix operations an overwhelming space and speed advantage.

We also achieved speedups by carefully permitting data races. The monotonicity of 0CFA makes it sound to perform analysis operations in parallel, possibly using stale or even partially-updated data.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors and Optimization

General Terms Languages

Keywords abstract interpretation, EigenCFA, program analysis, flow analysis, lambda calculus, GPU, CPS, matrix

1. Introduction

GPUs excel at obtaining speedups for algorithms over continuous domains with low-control, high-arithmetic kernels. Flow analyses [22, 24, 25], on the other hand, tend to be fixed-point algorithms over discrete domains with high-control, low-arithmetic kernels, such as abstract interpretations [7, 8]. At first glance, GPUs seem ill-suited to accelerating flow analyses. Yet, with a shift in algorithmic perspective and the right data structures, GPUs make the bedrock flow analysis for higher-order programs—0CFA—nearly two orders of magnitude faster.

1.1 Motivation

After nearly a quarter century, higher-order control-flow analysis [11, 25] remains an important analysis for highly-optimizing, whole-program compilers of functional languages. Yet, the analysis also remains stubbornly expensive. Even one of its simplest formulations, 0CFA, is still "nearly" cubic in complexity: $O(n^3/\log n)$ [6, 19].

With CPU clock cycles no longer growing, this complexity places a *de facto* upper bound on the size of programs that can be analyzed in a reasonable amount of time. To analyze large programs, higher-order control-flow analysis must exploit the ever-increasing parallelism available on modern systems. Toward that end, we develop a GPU-accelerated algorithm for 0CFA—EigenCFA—that achieves a factor of 72 speedup over existing CPU techniques.

1.2 High-level methodology

To develop EigenCFA, we imagined the GPU as a platform for accelerating non-branching, data-parallel algorithms composed entirely of linear-algebra operations. So, we reduced Shivers' abstract-interpretive 0CFA to a data-parallel, non-branching algorithm composed entirely of matrix operations: matrix multiplication, matrix addition and matrix transposition. There are five key insights to this reduction:

1. **Canonicalization to binary continuation-passing style.**

   To achieve good data-parallelism on a GPU, we need control-flow uniformity among the GPU threads, *i.e.*, to avoid branching operations. In flow analysis, the key step to parallelize is the propagation of flow information at each call site. To eliminate branching from this propagation routine, we transform our programs into a canonical form: binary continuation-passing style (binary CPS). In binary CPS, every call site provides two arguments, and every function accepts two arguments. By eliminating the need to discriminate on the "instruction type" (there is only one: function call) and on the number of arguments, binary CPS eliminates branching from the propagation sub-routine.

2. **Abstract Church encodings.**

   One could use Church encodings to reduce every program construct (*e.g.*, `if`, `letrec`, `set!`) to binary continuation-passing style. But, these desugarings obscure control-flow. For instance, desugaring `set!` requires a global store-passing transform, and the Y combinator fogs up recursion. An abstract Church encoding exploits the approximation in 0CFA so that the encoded program has the same abstract control-flow as the original program, if no longer the same concrete behavior. For example:

   ```
   (set! x 10)
   ```

   is equivalent (as far as 0CFA is concerned) to:

   ```
   (let ((x 10)) #void)
   ```

POPL'11, January 26–28, 2011, Austin, Texas, USA.

3. **Matrix-vector encoding of the syntax tree.**

   We use binary continuation-passing style to provide uniformity to program syntax, but we still need a GPU-friendly way to encode the syntax tree of a program. We encode the syntax tree of a program as a collection of selector functions, which are themselves represented as matrices. Individual program terms are then encoded as vectors. (We'll write $\langle\!\langle t \rangle\!\rangle$ to mean the vector that encodes term t.) For instance, every call site has three components: the procedure expression, its first argument expression and its second argument expression. So, there are three selector matrices that operate on call sites: $\mathbf{Fun}$, $\mathbf{Arg}_1$ and $\mathbf{Arg}_2$. For example, for a call site $\texttt{(}f\ e_1\ e_2\texttt{)}$:

$$\langle\!\langle \texttt{(}f\ e_1\ e_2\texttt{)} \rangle\!\rangle \times \mathbf{Fun} = \langle\!\langle f \rangle\!\rangle$$
$$\langle\!\langle \texttt{(}f\ e_1\ e_2\texttt{)} \rangle\!\rangle \times \mathbf{Arg}_1 = \langle\!\langle e_1 \rangle\!\rangle$$
$$\langle\!\langle \texttt{(}f\ e_1\ e_2\texttt{)} \rangle\!\rangle \times \mathbf{Arg}_2 = \langle\!\langle e_2 \rangle\!\rangle.$$

4. **Matrix encoding of the abstract store.**

   In 0CFA, the abstract store (also known as the abstract heap) maps variable names to sets of values. It is the primary data structure used during the execution of 0CFA, so it must be encoded in a GPU-friendly way. Fortunately, it is straightforward to represent this data structure as a matrix. One axis of the store matrix represents variables; the other axis represents lambda terms. If the entry for variable i, lambda j is non-zero, this indicates that (a closure over) lambda j may get bound to variable i. Thus, if the matrix $\boldsymbol{\sigma}$ is an abstract store in matrix form, and v is a variable, then the vector $\langle\!\langle v \rangle\!\rangle \times \boldsymbol{\sigma}$ describes the possible values of the variable v.

5. **Linear-algebraic encoding of the transfer function.**

   Once the syntax tree of the program is described in terms of selection matrices, the next step is to describe the action of the small-step transition relation for 0CFA in terms of linear-algebraic operations. Examination of the small-step transition relation (Section 2.4) finds only three operations beyond syntactic selection: function lookup, join over functions and functional extension. We reduce function lookup to matrix-vector multiplication, join to matrix addition and functional extension to a combination of matrix addition, matrix multiplication and matrix transposition.

1.3 Key insights for acceleration: Sparseness and races

For the implementation, there are two insights that lead to acceleration: a sparse-matrix representation of the abstract store, and a tolerance of benign (monotonic) races that allows the analysis of call sites in parallel.

1. **Exploiting sparseness.**

   In practice, most control-flow graphs are sparse. In our matrix encoding of the store, the sparseness of this matrix is linked to the sparseness of the control-flow graph. Thus, sparse-matrix algorithms have a performance advantage over dense-matrix algorithms for all but the most pathological programs: programs which tend toward completeness in their control-flow graphs—programs in which any point may jump to any other point.

2. **Exploiting monotonicity to permit benign race conditions.**

   When analyzing call sites in parallel, they may both attempt to read from and/or write to the same location in the abstract store. Fortunately, the monotonic growth of the abstract store during an abstract transition guarantees us that these races are benign: if properly engineered, once an entry is set to a non-zero value in the abstract store, no other thread will set it back to zero. Monotonicity *also* guarantees soundness when a thread works with a stale or partially updated store.

1.4 Contributions

- Our *primary* contribution is the formulation of GPU-accelerated flow analysis. To the best of our knowledge, EigenCFA is the first such formulation.
- Our secondary claims are the reductions that made this formulation possible: the encoding of the syntax tree as selection matrices, abstract Church encodings and the reduction to linear algebra of the abstract semantics for 0CFA.

1.5 Outline

Theory and implementation cleave this work in halves:

- **Theory.**

  Section 2 is a brief review of the small-step formulation of Shivers' 0CFA for continuation-passing style. Section 3 defines EigenCFA, a linear-algebraic formulation of 0CFA. Section 4 describes abstract Church encodings—program transformations that are meaning-preserving only for the *abstract semantics* of our analysis, which more precisely and efficiently handle language constructs such as mutation, recursion, termination, basic values and conditionals.

- **Implementation.**

  Section 5 describes how we map the algorithm for EigenCFA down to a (CUDA-enabled) GPU. In fact, this section describes three different approaches for performing the mapping: two of these end up slower than or only as fast as the CPU version, but the final implementation, which uses sparse matrices, achieves the desired speedup. Section 6 gives the results of our empirical evaluation. We tested two (GPU) implementations of EigenCFA against two (CPU) implementations of 0CFA, and found a factor of 72 speedup.

2. Background: Binary CPS and 0CFA

We are going to accelerate Shivers's original 0CFA for continuation-passing style (CPS). To enable high performance on the GPU, we are going to specialize it for a canonicalized form: binary continuation-passing style. In this section, we'll define binary CPS and briefly review 0CFA. Readers familiar with 0CFA may wish to skip this section. Our definition of 0CFA is taken from Might and Shivers's recent small-step reformulations. (We refer readers to [20] for details such as abstraction maps and proofs of soundness.)

Binary CPS is a canonicalized variant of the continuation-passing style λ-calculus in which every procedure accepts exactly two arguments. We use binary, as opposed to unary, CPS because the CPS transform on lambda terms introduces an additional argument—the continuation argument—and CPS cannot use Currying to handle multiple arguments, since it violates the "no procedure may return" principle. We are using binary, instead of variadic, CPS because we want to eliminate branching from the small-step transition relation we're about to define; branches interfere with data-parallel single-instruction, multiple-thread (SIMT) operations in the GPU.

The uniformity in binary CPS also has the benefit of making our forthcoming matrix encodings of syntax trees predictably structured. This predictable structure is amenable to the arithmetic optimizations and compressions presented in section 5.

2.1 Binary CPS

The grammar for binary CPS contains calls and expressions, and expressions are either lambda terms or variables:

$$\begin{aligned} call \in \mathsf{Call} &::= (f\ e_1\ e_2) \\ f, e \in \mathsf{Exp} &= \mathsf{Var} + \mathsf{Lam} \\ v \in \mathsf{Var} &\text{ is a set of identifiers} \\ lam \in \mathsf{Lam} &::= (\lambda\ (v_1\ v_2)\ call). \end{aligned}$$

2.2 Concrete semantics

The simplest small-step concrete semantics for binary CPS uses just three domains in its state-space, Σ:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Clo} \\ clo \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env}, \end{aligned}$$

and one transition rule, $(\Rightarrow) \subseteq \Sigma \times \Sigma$:

$$\begin{aligned} (\llbracket (f\ e_1\ e_2) \rrbracket, \rho) &\Rightarrow (call, \rho''), \text{ where} \\ (\llbracket (\lambda\ (v_1\ v_2)\ call) \rrbracket, \rho') &= \mathcal{E}(f, \rho) \\ clo_1 &= \mathcal{E}(e_1, \rho) \\ clo_2 &= \mathcal{E}(e_2, \rho) \\ \rho'' &= \rho'[v_1 \mapsto clo_1, v_2 \mapsto clo_2], \end{aligned}$$

where the function $\mathcal{E} : \mathsf{Exp} \times \mathit{Env} \rightharpoonup \mathit{Clo}$ evaluates expressions:

$$\begin{aligned} \mathcal{E}(v, \rho) &= \rho(v) \\ \mathcal{E}(lam, \rho) &= (lam, \rho). \end{aligned}$$

2.3 Abstract state-space for 0CFA

Might and Shivers reformulated 0CFA as an abstract interpretation of the concrete semantics just given [20]. The core of the abstraction is the elimination of environments from the semantics, and the use of an abstract store to represent all environments. So, abstract states ($\hat{\Sigma}$) pair the current call site with the abstract store, and closures lose their environments:

$$\begin{aligned} \hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{\mathit{Store}} \\ \hat{\sigma} \in \widehat{\mathit{Store}} &= \mathsf{Var} \to \widehat{\mathit{Lams}} \\ \hat{L} \in \widehat{\mathit{Lams}} &= \mathcal{P}(\mathsf{Lam}). \end{aligned}$$

We assume the natural partial orders on these sets: lambda sets are ordered by inclusion, abstract stores are ordered point-wise by inclusion, and abstract states use a product ordering. For instance, $(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(v) = \hat{\sigma}_1(v) \cup \hat{\sigma}_2(v)$.

2.4 0CFA Abstract Semantics

The transition relation for the abstract semantics, $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, mirrors that of the concrete semantics:

$$\begin{aligned} (\llbracket (f\ e_1\ e_2) \rrbracket, \hat{\sigma}) &\leadsto (call, \hat{\sigma}'), \text{ where} \\ \llbracket (\lambda\ (v_1\ v_2)\ call) \rrbracket &\in \hat{\mathcal{E}}(f, \hat{\sigma}) \\ \hat{L}_1 &= \hat{\mathcal{E}}(e_1, \hat{\sigma}) \\ \hat{L}_2 &= \hat{\mathcal{E}}(e_2, \hat{\sigma}) \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [v_1 \mapsto \hat{L}_1, v_2 \mapsto \hat{L}_2], \end{aligned}$$

as does the argument evaluator, $\hat{\mathcal{E}} : \mathsf{Var} \times \widehat{\mathit{Store}} \to \mathcal{P}(\mathsf{Lam})$:

$$\begin{aligned} \hat{\mathcal{E}}(v, \hat{\sigma}) &= \hat{\sigma}(v) \\ \hat{\mathcal{E}}(lam, \hat{\sigma}) &= \{lam\}. \end{aligned}$$

A notable change in the abstract semantics is the nondeterminism that results from branching to the set of possible lambda terms for the procedure argument (note the appearance of $\in$).

2.5 Computing 0CFA

To compute classical, flow-insensitive 0CFA with the small-step transition relation, we need a family of functions that computes the output store with respect to each call site $call$, $\hat{f}_{call} : \widehat{\mathit{Store}} \to \widehat{\mathit{Store}}$:

$$\hat{f}_{call}(\hat{\sigma}) = \bigsqcup \left\{ \hat{\sigma}' : (call, \hat{\sigma}) \leadsto (\_, \hat{\sigma}') \right\}.$$

Then, we can construct the "pass" function, $\hat{F} : \widehat{\mathit{Store}} \to \widehat{\mathit{Store}}$, which performs one full pass of the analysis by considering the effect on the store of every call site in the program, $call_1, \ldots, call_n$:

$$\hat{F}(\hat{\sigma}) = (\hat{f}_{call_1} \circ \cdots \circ \hat{f}_{call_n})(\hat{\sigma}).$$

Because the function $\hat{F}$ is continuous and monotonic, and the height of the abstract store lattice is finite, the result of 0CFA is least fixed point of the function $\hat{F}$:

$$\mathrm{lfp}(\hat{F}) = \hat{F}^n(\bot_{\widehat{\mathit{Store}}}) \text{ for some finite } n,$$

where the bottom store maps everything to the empty set:

$$\bot_{\widehat{\mathit{Store}}} = \lambda v.\emptyset.$$

Complexity If the function $\hat{F}$ adds one entry to the abstract store per application, there can be at most $|\mathsf{Var}| \times |\mathsf{Lam}|$ applications before it saturates the abstract store and must terminate. Since the cost of each application is proportional to $|\mathsf{Call}|$, the complexity of this 0CFA is cubic, as expected.

3. EigenCFA: A linear encoding of 0CFA

In this section, we discuss our linear-algebra encoding of both binary CPS and 0CFA. In brief, individual program terms will be represented as vectors. The structure of the syntax tree will be compiled into static selection matrices that operate on these term vectors. Finally, the pass function ($\hat{F}$) from 0CFA will be encoded as a function operating on stores represented as matrices.

Running example. Throughout the remainder of this paper, all of the examples will be with respect to this program (the call-sites $c_1, c_2, \ldots$ are explicitly labelled for future reference):

$$\begin{aligned} ((\lambda_1\ (v_1\ v_1')\ &(v_1\ \ v_1\ \ v_1')_{c_1}) \\ &(\lambda_2\ (v_2\ v_2')\ (v_2'\ \ v_2\ \ v_2')_{c_2}) \\ &(\lambda_3\ (v_3\ v_3')\ (v_3\ \ v_3\ \ v_3)_{c_3}))_{c_4} \end{aligned}$$

and the same abstract store, $\hat{\sigma}$:

$$\begin{aligned} \hat{\sigma}(v_1) &= \{\lambda_2\} \\ \hat{\sigma}(v_2) &= \{\lambda_2\} \\ \hat{\sigma}(v_3) &= \{\lambda_2\} \\ \hat{\sigma}(v_1') &= \{\lambda_3\} \\ \hat{\sigma}(v_2') &= \{\lambda_3\} \\ \hat{\sigma}(v_3') &= \{\lambda_3\}. \end{aligned}$$

We do this to improve presentation and to emphasize the differences between each data representation strategy in Section 5.

3.1 Encoding terms as vectors

We encode each program term (a lambda term, a variable or a call site) as a vector over the set $\{0, 1\}$. Every term in the program will have a unique vector representation.[1] For convenience, we write the vector encoding of term t as $\langle\!\langle t \rangle\!\rangle$.

We'll have vectors of two lengths: expression vectors (of length $|\mathsf{Exp}|$) and call vectors (of length $|\mathsf{Call}|$). Formally:

$$\begin{aligned} \vec{e} &\in \overrightarrow{\mathsf{Exp}} = \{0,1\}^{|\mathsf{Exp}|} \\ \vec{call} &\in \overrightarrow{\mathsf{Call}} = \{0,1\}^{|\mathsf{Call}|} \\ \vec{v} &\in \overrightarrow{\mathsf{Var}} \subset \overrightarrow{\mathsf{Exp}} \\ \vec{lam} &\in \overrightarrow{\mathsf{Lam}} \subset \overrightarrow{\mathsf{Exp}}. \end{aligned}$$

We can assign each expression a number from 1 to $|\mathsf{Exp}|$, so that the expression vector with 1 at slot i is the expression vector for expression i. We can do likewise for call sites.

More specifically, variables are represented as vectors of size $|\mathsf{Exp}|$ (notably not of size $|\mathsf{Var}|$). λ-terms are also represented as vectors of size $|\mathsf{Exp}|$ (also notably not of size $|\mathsf{Lam}|$). The first $|\mathsf{Var}|$ entries of an expression vector represent variables, and the last $|\mathsf{Lam}|$ entries represent lambda terms.

Running example. Here is the expression vector representing lambda term λ_2:

$$\begin{array}{c} \begin{array}{ccccccccc} v_3' & v_2' & v_1' & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

And, here is the expression vector representing variable v_2:

$$\begin{array}{c} \begin{array}{ccccccccc} v_3' & v_2' & v_1' & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

It's worth asking why we don't split expressions into two vector types—one for variables, and one for lambda terms. We took this approach because it allows us to eliminate branching from argument evaluation later on. As it is formulated in the abstract semantics, the argument evaluator $\hat{\mathcal{E}}$ must look up its expression argument if it is a variable and return its argument if it is a lambda term. We'll be able to construct a single linear operator that has the effect of the argument evaluator $\hat{\mathcal{E}}$—no branching necessary.

3.2 Encoding the syntax tree as matrices

To encode the syntax tree, we'll use selectors encoded as static matrices to destructure syntax terms. That is, given a term vector $\langle\!\langle t \rangle\!\rangle$, to figure out the syntactic children of term t, we'll have a matrix for each type of child (e.g. first formal parameter, second argument) that maps $\langle\!\langle t \rangle\!\rangle$ to that child. For instance, the matrix $\mathbf{Call}$ maps lambda terms to their call site, so that $\langle\!\langle \texttt{(λ (}v_1\ v_2\texttt{)}\ call\texttt{)} \rangle\!\rangle \times \mathbf{Call} = \langle\!\langle call \rangle\!\rangle$.

There are six static syntax matrices:

$$\begin{aligned} \mathbf{Fun} :&\quad \overrightarrow{\mathsf{Call}} \to \overrightarrow{\mathsf{Exp}} \quad (\textit{function applied in a call site}) \\ \mathbf{Arg}_1 :&\quad \overrightarrow{\mathsf{Call}} \to \overrightarrow{\mathsf{Exp}} \quad (\textit{the first argument in a call site}) \\ \mathbf{Arg}_2 :&\quad \overrightarrow{\mathsf{Call}} \to \overrightarrow{\mathsf{Exp}} \quad (\textit{the second argument in a call site}) \\ \mathbf{Call} :&\quad \overrightarrow{\mathsf{Lam}} \to \overrightarrow{\mathsf{Call}} \quad (\textit{the call site of a } \lambda\textit{-term}) \\ \mathbf{Var}_1 :&\quad \overrightarrow{\mathsf{Lam}} \to \overrightarrow{\mathsf{Exp}} \quad (\textit{the first formal of a } \lambda\textit{-term}) \\ \mathbf{Var}_2 :&\quad \overrightarrow{\mathsf{Lam}} \to \overrightarrow{\mathsf{Exp}} \quad (\textit{the second formal of a } \lambda\textit{-term}) \end{aligned}$$

These matrices are constructed once per program and remain constant through the duration of the analysis.

Running example. The **Fun** matrix determines the function being applied at a call site for our running example:

$$\begin{array}{cc} & \begin{array}{ccccccccc} v_3' & v_2' & v_1' & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \begin{array}{c} c_1 \\ c_2 \\ c_3 \\ c_4 \end{array} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Running example. In order to look up the function being applied at call site labelled c_2, we multiply the vector representing c_2 with the **Fun** matrix. The result is the vector representing the variable v_2':

$$\begin{array}{cc} & \begin{array}{c} \begin{array}{cccc} c_1 & c_2 & c_3 & c_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \\ & \begin{array}{ccccccccc} v_3' & v_2' & v_1' & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \times & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\ & \begin{array}{ccccccccc} v_3' & v_2' & v_1' & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ = & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

3.3 Encoding flow sets as vectors

0CFA manipulates flow sets; flow sets are sets of lambda terms ($\widehat{Lams}$). We also need to represent these as linear-algebraic values. A natural encoding of flow sets is to use bit vectors of length $|\mathsf{Lam}|$:

$$\vec{L} \in \overrightarrow{\mathsf{Lam}}^\star = \{0,1\}^{|\mathsf{Lam}|}.$$

In some sense, we appear to be breaking with the uniform representation of expressions by creating a special encoding of lambda terms. However, it is more accurate to think of the set $\overrightarrow{\mathsf{Lam}}^\star$ as representing sets of abstract closures than sets of lambda terms. Moreover, we can (and do) efficiently interconvert values between the set $\overrightarrow{\mathsf{Lam}}$ and the set $\overrightarrow{\mathsf{Lam}}^\star$ simply by adding or ignoring 0-padding on the front of the vector.

For a set of lambda terms, $\hat{L}$, we write their vector encoding as $\langle\!\langle \hat{L} \rangle\!\rangle$, and we expect the following property to hold:

$$\langle\!\langle \{lam_1, \ldots, lam_n\} \rangle\!\rangle = \langle\!\langle lam_1 \rangle\!\rangle + \cdots + \langle\!\langle lam_n \rangle\!\rangle,$$

where the operator $+$ is element-wise Boolean-OR.

3.4 Encoding abstract stores as matrices

Now that we have a representation for syntactic domains and for flow sets, we need a matrix representation for the abstract store. Fortunately, the store is a map from variables to flow sets, and a linear operator (encoded as a matrix) is a natural way of representing such a map.

We do have a design choice here, and the right answer is not immediately obvious. We *could* use a $|\mathsf{Var}|$-by-$|\mathsf{Lam}|$ matrix to represent stores; but we'll be able to eliminate a branch during argument evaluation if we make all stores into slightly larger $|\mathsf{Exp}|$-by-$|\mathsf{Lam}|$ matrices. So, formally:

$$\boldsymbol{\sigma} \in \mathbf{Store} = \overrightarrow{\mathsf{Exp}} \to \overrightarrow{\mathsf{Lam}}^\star,$$

[1] We could formulate the analysis more generally in terms of any orthogonal basis vectors for terms, but there doesn't seem to be any performance advantage to doing so.

We write the matrix-encoding of an abstract store $\hat{\sigma}$ as $\langle\langle\hat{\sigma}\rangle\rangle$. And, we define the encoding by relating abstract and matrix stores:

$$\langle\langle lam\rangle\rangle \times \boldsymbol{\sigma} = \langle\langle lam\rangle\rangle$$
$$\langle\langle v\rangle\rangle \times \langle\langle\hat{\sigma}\rangle\rangle = \langle\langle\hat{\sigma}(v)\rangle\rangle,$$

where the operator $\times$ is actually Boolean matrix multiplication.

According to these rules, the lower portion of all stores (the part that corresponds the lambda portion of expression vectors) must be the identity matrix.

Running example. The matrix representation of the store is:

$$\begin{array}{c|ccc|}
 & \lambda_3 & \lambda_2 & \lambda_1 \\
v'_3 & 1 & 0 & 0 \\
v'_2 & 1 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
v_1 & 0 & 1 & 0 \\
\cline{2-4}
\lambda_3 & 1 & 0 & 0 \\
\lambda_2 & 0 & 1 & 0 \\
\lambda_1 & 0 & 0 & 1
\end{array}$$

Under this encoding of stores, the evaluation of an expression takes a single matrix multiplication:

Theorem 3.1. $\langle\langle\hat{\mathcal{E}}(e,\hat{\sigma})\rangle\rangle = \langle\langle e\rangle\rangle \times \langle\langle\hat{\sigma}\rangle\rangle$.

Proof. By case-wise analysis and the rules for the encoding. □

Running example. Here's the look-up of variable v'_2:

$$\begin{array}{cccccc}
v'_3 & v'_2 & \cdots & v_1 & \lambda_3 & \cdots \\
0 & 1 & \cdots & 0 & 0 & \cdots
\end{array}
\times
\begin{array}{c|ccc|}
 & \lambda_3 & \lambda_2 & \lambda_1 \\
v'_3 & 1 & 0 & 0 \\
v'_2 & 1 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
v_1 & 0 & 0 & 1 \\
\cline{2-4}
\lambda_3 & 1 & 0 & 0 \\
\lambda_2 & 0 & 1 & 0 \\
\lambda_1 & 0 & 0 & 1
\end{array}$$

$$= \begin{array}{ccc}
\lambda_3 & \lambda_2 & \lambda_1 \\
1 & 0 & 0
\end{array}$$

3.4.1 Updates on the store

During the abstract transition, we make updates to stores of the form:

$$\hat{\sigma}' = \hat{\sigma} \sqcup [v_1 \mapsto \hat{L}_1, v_2 \mapsto \hat{L}_2].$$

We need matrix operations that correspond to this update operation.

First, we have to construct a store representing a single mapping: $[v \mapsto \hat{L}]$. Fortunately, matrix transposition and matrix multiplication make this straightforward:

Lemma 3.1. $\langle\langle[v \mapsto \hat{L}]\rangle\rangle = \langle\langle v\rangle\rangle^{\top} \times \langle\langle\hat{L}\rangle\rangle + \langle\langle\bot_{Store}\rangle\rangle$.

It's also the case that Boolean-OR acts as join on stores:

Lemma 3.2. $\langle\langle\hat{\sigma}_1 \sqcup \hat{\sigma}_2\rangle\rangle = \langle\langle\hat{\sigma}_1\rangle\rangle + \langle\langle\hat{\sigma}_2\rangle\rangle$.

And, these two lemmas give us store update:

Theorem 3.2 (Store Update Theorem)**.**

$$\langle\langle\hat{\sigma} \sqcup [v_1 \mapsto \hat{L}_1, v_2 \mapsto \hat{L}_2]\rangle\rangle$$
$$= \langle\langle\hat{\sigma}\rangle\rangle + \langle\langle v_1\rangle\rangle^{\top} \times \langle\langle\hat{L}_1\rangle\rangle + \langle\langle v_2\rangle\rangle^{\top} \times \langle\langle\hat{L}_2\rangle\rangle.$$

Proof. By the prior two lemmas. □

3.4.2 Running example: Store update

This example shows all the moving parts for store update.

Running example. To update with λ_2 flowing to variable v'_2, we multiply the vectors representing λ_2 and v'_2 and add the result, $\boldsymbol{\sigma}_{\text{frag}}$, to the current store:

$$\begin{array}{c|c|}
v'_3 & 0 \\
v'_2 & 1 \\
\vdots & \vdots \\
v_1 & 0 \\
\cline{2-2}
\lambda_3 & 0 \\
\lambda_2 & 0 \\
\lambda_1 & 0
\end{array}
\times
\begin{array}{ccc}
\lambda_3 & \lambda_2 & \lambda_1 \\
0 & 1 & 0
\end{array}
=
\begin{array}{c|ccc|}
 & \lambda_3 & \lambda_2 & \lambda_1 \\
v'_3 & 0 & 0 & 0 \\
v'_2 & 0 & 1 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
v_1 & 0 & 0 & 0 \\
\cline{2-4}
\lambda_3 & 0 & 0 & 0 \\
\lambda_2 & 0 & 0 & 0 \\
\lambda_1 & 0 & 0 & 0
\end{array}$$

$$\begin{array}{c|ccc|}
 & & \boldsymbol{\sigma}_{\text{frag}} & \\
v'_3 & 0 & 0 & 0 \\
v'_2 & 0 & 1 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
v_1 & 0 & 0 & 0 \\
\cline{2-4}
\lambda_3 & 0 & 0 & 0 \\
\lambda_2 & 0 & 0 & 0 \\
\lambda_1 & 0 & 0 & 0
\end{array}
+
\begin{array}{|ccc|}
 & \boldsymbol{\sigma}_{\text{curr}} & \\
1 & 0 & 0 \\
1 & 0 & 0 \\
\vdots & \vdots & \vdots \\
0 & 1 & 0 \\
\hline
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{array}
=
\begin{array}{|ccc|}
 & \boldsymbol{\sigma}_{\text{new}} & \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\vdots & \vdots & \vdots \\
0 & 1 & 0 \\
\hline
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{array}$$

3.5 Linear encoding of the transfer function

Earlier, we constructed a family of transfer functions, $\hat{f}_{call}$, which produced the effect of the transition relation on the store for a given call site. We can construct an equivalent linearized function, $\mathbf{f}_{call} : \mathbf{Store} \to \mathbf{Store}$,

$$\begin{aligned}
\mathbf{f}_{call}(\boldsymbol{\sigma}) &= \boldsymbol{\sigma}' \\
\vec{L} &= \langle\langle call\rangle\rangle \times \mathbf{Fun} \times \boldsymbol{\sigma} \\
\vec{L}_1 &= \langle\langle call\rangle\rangle \times \mathbf{Arg}_1 \times \boldsymbol{\sigma} \\
\vec{L}_2 &= \langle\langle call\rangle\rangle \times \mathbf{Arg}_2 \times \boldsymbol{\sigma} \\
\vec{v}_1 &= \vec{L} \times \mathbf{Var}_1 \\
\vec{v}_2 &= \vec{L} \times \mathbf{Var}_2 \\
\boldsymbol{\sigma}' &= \boldsymbol{\sigma} + \left(\vec{v}_1^{\top} \times \vec{L}_1\right) + \left(\vec{v}_2^{\top} \times \vec{L}_2\right).
\end{aligned}$$

3.6 Soundness

We can prove that the linearized transfer function is equivalent to the traditional one, under our matrix-vector encoding:

Theorem 3.3 (Soundness)**.** $\langle\langle\hat{f}_{call}(\hat{\sigma})\rangle\rangle = \mathbf{f}_{call}\langle\langle\hat{\sigma}\rangle\rangle$.

Proof. The proof proceeds bidirectionally. First, show that every entry in $\langle\langle\hat{f}_{call}(\hat{\sigma})\rangle\rangle$ must be in $\mathbf{f}_{call}\langle\langle\hat{\sigma}\rangle\rangle$, and then *vice versa*. In both directions, it's easiest to split into cases: the one in which the entry exists in $\hat{\sigma}$, and the one in which it is fresh. □

3.7 EigenCFA: The Algorithm

Algorithm 1 gives the high-level algorithm for EigenCFA. The algorithm assumes that the store $\boldsymbol{\sigma}$ is empty at the start.

```
while $\boldsymbol{\sigma}$ changes do
    foreach call do
        // Lookup function and arguments in call
        $\vec{L} := (\langle\!\langle call \rangle\!\rangle \times \mathbf{Fun}) \times \boldsymbol{\sigma}$
        $\vec{L}_1 := (\langle\!\langle call \rangle\!\rangle \times \mathbf{Arg}_1) \times \boldsymbol{\sigma}$
        $\vec{L}_2 := (\langle\!\langle call \rangle\!\rangle \times \mathbf{Arg}_2) \times \boldsymbol{\sigma}$

        // Formal arguments of function, $\vec{L}$
        $\vec{v}_1 := (\vec{L} \times \mathbf{Var}_1) \times \boldsymbol{\sigma}$
        $\vec{v}_2 := (\vec{L} \times \mathbf{Var}_2) \times \boldsymbol{\sigma}$

        // Update store
        $\boldsymbol{\sigma} := \boldsymbol{\sigma} + \underbrace{\vec{v}_1^\top \times \vec{L}_1}_{Bind\ L_1\ to\ v} + \underbrace{\vec{v}_2^\top \times \vec{L}_2}_{Bind\ L_2\ to\ v'}$
    end
end
```

Algorithm 1: EigenCFA

4. Abstract Church encodings

It's possible to transform any program into binary CPS using mechanisms like Church encodings and the Y combinator. Some encodings, like transforming the term `(let ((`v e`))` *body*`)` into the term `((λ (`v`)` *body*`)` e`)` are benign. But, because some encodings transform data-flow into control-flow, they obscure the control-flow of the original program. Even Church-encoded integers raise thorny control-flow issues. Yet, handling forms like `set!` or `letrec` as special cases in the transfer function is not practical: this would force conditional tests and branching, disrupting the data-parallelism that we have carefully engineered and protected.

To avoid branching but preserve precision, we turn to "abstract Church encodings." An *abstract Church encoding* is a program transformation that is meaning-preserving for an *abstract* semantics, but not for the concrete semantics.

Caution The abstract Church encodings in this section work for the abstract semantics of 0CFA and the simple linear model of EigenCFA. Some of the forthcoming GPU optimizations require alphatization of the program, a constraint that these encodings violate. Where violations occur, we will note how to adapt.

4.1 Abstracting termination as non-termination

We actually *need* abstract Church encodings to handle program termination: notice that our CPS language has no `halt` form. So to encode `halt`, we'll use *non*-termination; that is, we use the non-terminating program Omega as the abstract encoding for termination. In other words, we apply the following rewrite rule after conversion to a binary CPS that contains a `halt` primitive:

```
halt ⟹
    (λ (a b) ((λ (f g) (f f f))
      (λ (f g) (f f f))
      (λ (f g) (f f f))))
```

This works because once an abstract interpreter hits Omega, that branch won't contribute any more changes to the abstract store, so the abstract interpretation can reach a fixed point.

4.2 Abstracting mutation as binding

To Church encode a construct like `set!`, we'd have to (1) perform cell boxing on all mutable variables, and then (2) eliminate all cells with a store-passing-style transformation. These kinds of global transforms alter the control-flow and data-flow behavior of the program. Yet, 0CFA has a single abstract store that represents *all* program environments. As a result, `let` and `set!` have *exactly the same effect on the abstract store*; so we can apply a rewriting rule:

```
(set! v e) ⟹ (let ((v e)) #void)
```

From the abstract store's perspective, these terms are equivalent.

4.3 Encoding recursion as mutation

Now that we can handle mutation, we can avoid using the Y combinator (or a grisly polvariadic, mutually recursive variant thereof) to handle recursion. `Letrec` normally desugars into "`let`s and `set!`s." But, since `let` has the same effect as `set!`, we can actually turn `letrec` into `let` with an abstract rewrite rule:

```
    (letrec ((v₁ lam₁) ... (vₙ lamₙ)) e)
⟹ (let ((v₁ lam₁) ... (vₙ lamₙ)) e)
```

0CFA does not distinguish their effects on the abstract store.

4.4 Abstracting basic values as non-termination

Most CFAs encode all numbers as a single abstract value. In fact, most even convert *all* basic values into a single abstract value. We can do the same, using the `halt` function from before as the single abstract value. (Any attempt to apply a basic value shouldn't allow the program to continue normal execution.) All primitive operations that operate on basic values ignore their arguments and return this basic value.

4.5 Abstractly encoding conditionals

Most CFAs do not attempt to evaluate conditionals; their behavior is to always branch in both directions at an `if`. We could Church encode Booleans and conditionals, but this introduces a level of indirection, as conditionals appear to interprocedurally flow through their Church-encoded condition. Or, we could exploit the fact that flow sets merge in 0CFA to simulate the non-determinism of the conditional with the non-determinism of procedure call. So, after we CPS transform `if` forms, we can abstractly encode them with a `let`-based rewrite:

```
   (if e call₁ call₂)
⟹ (let ((next (λ () call₁)))
     (let ((next (λ () call₂)))
       (next)))
```

It's safe to drop the conditional expression e, because after the CPS transform, this expression is atomic, and it makes no changes to the store. By the time the analysis reaches the call site `(next)`, both continuations will have been bound to `next` in the abstract store.

5. Our GPU implementation narrative

In this section, we discuss the details of mapping our high-level algorithm for EigenCFA down to the nuts and bolts of a GPU implementation. It took effort to discover which optimizations and data structures were the right ones, so we will discuss both the right turns and the wrong ones on the road to EigenCFA. We present three iterations of our implementation: (1) naïve, (2) dense and (3) sparse.

We wrote a pre-processor in Racket which transformed the input program and generated the static syntax matrices. These matrices were passed to a C program which copied them into the GPU memory and launched the CUDA (GPU) kernels that performed the analysis. (For a brief summary of CUDA and the high-level architecture of the GPU we targeted, see the appendix.) All three of our implementations picked up at this point.

5.1 Attempt 1: Naïve implementation with CUBLAS

Our first implementation of EigenCFA used NVidia's CUBLAS library for linear algebra to perform the matrix operations. This implementation was an almost verbatim transliteration of Algorithm 1, and it turned out to be *slower* than our CPU implementation. We identified several problems:

- **Dense matrix computations**.

  Although most of the matrices in our analysis were sparse, the CUBLAS library is written for accelerating dense-matrix operations. Since the matrices are all quadratic in the size of the program, they consumed all of the parallel processing facilities of the GPU, even for small programs.

- **Memory requirements.**

  The matrices and vectors in EigenCFA only contain boolean values. Use of 4-byte floats as required by the CUBLAS libraries for each element was wasteful. As we describe in Section 5.2, using one bit per entry yielded large constant-factor speedups and substantial memory savings.

- **Redundant operations.**

  The static syntax matrix lookups (each a large multiplication) were repeated in each iteration. Eventually, we were able to compress these matrices and simplify their application.

5.2 Attempt 2: Bit-packed matrix implementation

To overcome some of the limitations of the naïve approach, we optimized the matrix and vector representations.

5.2.1 Bit-packing

Since the vectors and matrices contain Boolean entries, one obvious way to reduce the sizes of the matrices was to use bit-vectors for terms and values, and bit-matrices for the abstract store and the syntactic matrices. Once encoded as bit-packed data structures, we also gain word-level parallelism by shifting from Boolean to bitwise-logical operations.

Running example. Here is the same store lookup from earlier, with bit-packed matrices:

$$\begin{bmatrix} 0 & 128 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 196 & 58 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

Each column of the store has been compressed. To make this example small but interesting, we assume that the size of each word is eight bits. The $\otimes$ operator is bitwise matrix-multiplication that works on our bit-packing scheme: the multiplication and addition in conventional matrix multiplication algorithm become bitwise-AND and bitwise-OR.

In our implementation, the word size is 32 bits—the word size on the GPU on which we ran the analysis. This means that all vectors are padded to bring their size to the nearest multiple of 32. This bit-packing reduced the size of the matrices by almost a factor of 32. (The word-alignment padding is why the reduction in size isn't always *exactly* a factor of 32.)

Running example. Here is bit-packed store update:

$$\begin{bmatrix} 0 \\ 128 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 128 & 0 \end{bmatrix}$$

$$\oplus \begin{bmatrix} 1 & 0 & 0 \\ 196 & 58 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 196 & 186 & 1 \end{bmatrix}$$

(The $\oplus$ operator performs a bitwise-OR.)

5.2.2 Compressing static syntax matrices

Each static syntax matrix is quadratic in the size of the program. Transferring them to and manipulating them on the GPU imposes significant time and memory costs. Fortunately, we can exploit the uniformity of binary CPS to compress them and optimize their application at the same time.

By choosing the numbering scheme for variables and lambda terms carefully, we compressed the operators $\mathbf{Var}_1$ and $\mathbf{Var}_2$ into small, explicit formulae. Specifically, if the lambda term λ_n is the nth lambda term, then in an expression vector, $(n + |\mathsf{Lam}|)$ is the expression number of its first formal, and $(n + 2|\mathsf{Lam}|)$ is the expression number of it second formal.

Now, each expression vector has three distinct segments: the $\mathbf{Lam}$ segment for the lambda terms; the $\mathbf{Var}_1$ segment for formals in the first position; and the $\mathbf{Var}_2$ segment for formal in the second position. With this scheme, we can determine $\langle\!\langle v_k \rangle\!\rangle$ and $\langle\!\langle v'_k \rangle\!\rangle$ from the representation of λ_k without any matrix multiplication.

Running example. If v_k and v'_k are, respectively, the first and second formal arguments for λ_k, these are the vectors representing λ_2, v_2 and v'_2:

$$\langle\!\langle \lambda_2 \rangle\!\rangle = \begin{array}{c} \begin{array}{ccc} \lambda_2 & \lambda_2 & \lambda_1 \end{array} \\ \left[\begin{array}{ccc} 0 & 1 & 0 \end{array} \right] \end{array}$$

$$\langle\!\langle v_2 \rangle\!\rangle = \begin{array}{c} \begin{array}{ccc|ccc|ccc} v'_3 & v'_2 & v'_1 & v_3 & v_2 & v_1 & \lambda_3 & \lambda_2 & \lambda_1 \end{array} \\ \left[\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array} \right] \end{array}$$

$$\langle\!\langle v'_2 \rangle\!\rangle = \left[\begin{array}{ccc|ccc|ccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

The $\mathbf{Call}$ matrix disappears after making the observation that there is exactly one call site for every λ-term. That is, the *same vector* (suitably truncated) can represent both the λ-term and its corresponding call site. (When we shift to a sparse-matrix implementation, we will be able to compress the remaining static syntax matrices.)

Caution: Abstract Church encodings The compression strategy we have chosen implicitly alphatizes the program, so that each variable has a unique lambda term that binds it. Some of the abstract Church encodings carefully exploit the behavior of 0CFA on non-alphatized code. There are two ways to resolve this problem: (1) more sophisticated abstract Church encodings or (2) a look-up table in the GPU implementation:

1. The encoding approach adds a procedure for every variable v:

   $$\texttt{write-}v = (\lambda\ (v\ q)\ (q))$$

   All mutation of the variable v must pass through this procedure.

2. The lookup-table approach creates a $2|\mathsf{Lam}|$-entry table in which entry n contains the expression number of the first formal and the entry $n + |\mathsf{Lam}|$ contains the expression number of the second formal for the nth lambda term.

5.2.3 Store update optimization

Under the encoding described in the previous section, the store is also logically divided into the same three equal-sized segments: the $\mathbf{Lam}$ region, the $\mathbf{Var}_1$ region and the $\mathbf{Var}_2$ region. As a result, we determined that store update, which adds $\vec{v}^{\top} \times \vec{L}$, can only modify a (known) third of the store: the region in which the variables in the vector $\vec{v}$ live. (Recall that, in the abstract semantics, updates for first-argument formals are carried out separately from updates for second-argument formals.) Exploiting this knowledge cuts the number of operations on update by two-thirds.

5.2.4 GPU-only optimizations

A few optimizations in our second implementation apply only to the GPU:

- **Reducing the number of kernel calls.**

  The CPU initiates parallel operations on the GPU through kernel invocations. Each kernel call has non-trivial overhead, so kernels should be combined whenever synchronization between them is unnecessary. We performed the calculations for $\vec{L}, \vec{L}_1$ and $\vec{L}_2$ (see Algorithm 1) within the same kernel. Combined with the static matrix compression and the store update optimization, the total number of kernel calls per iteration dropped from 14 to 3.

- **Constant memory.**

  Since we were iterating over all call sites in the program, there was strong locality in the syntax matrix lookup, and this data was read-only on the GPU. Therefore, it was a good candidate to use the GPU's constant memory, which unlike the global memory, is cached for low latency memory accesses. Although the constant memory on the GPU was not large enough to store the entire static matrix, we could leverage this locality and store only that part of the matrix which we knew would be used at any point of time. This increased speed by 40% compared to the scheme without constant memory.

- **Aligning thirds of the vectors to word boundaries.**

  In order to efficiently implement static syntax matrix compression (Section 5.2.2), we aligned each segment of the expression vectors—$\mathbf{Lam}$, $\mathbf{Var}_1$ and $\mathbf{Var}_2$—along word boundaries. We also padded the store to take into account this change in the size of the vectors.

Running example. The figure below shows 0-padded vectors where $|\mathsf{Lam}| = 3$ and the word size is 4.

$$\overbrace{\begin{array}{cccc} \lambda_d & \lambda_3 & \lambda_2 & \lambda_1 \\ \mathbf{0} & 0 & 1 & 0 \end{array}}^{\langle\langle \lambda_2 \rangle\rangle}$$

$$\overbrace{\left[\begin{array}{ccc|cccc|ccc} \cdots & v' & \cdots & v_d & v_3 & v_2 & v_1 & \cdots & \lambda & \cdots \\ \cdots & 0 & \cdots & \mathbf{0} & 0 & 1 & 0 & \cdots & 0 & \cdots \end{array}\right]}^{\langle\langle v_2 \rangle\rangle}$$

$$\overbrace{\left[\begin{array}{cccc|ccc|ccc} v'_d & v'_3 & v'_2 & v'_1 & \cdots & v & \cdots & \cdots & \lambda & \cdots \\ \mathbf{0} & 0 & 1 & 0 & \cdots & 0 & \cdots & \cdots & 0 & \cdots \end{array}\right]}^{\langle\langle v'_2 \rangle\rangle}$$

Running example. The corresponding 0-padded store is below. When creating the store, we need to add blank rows corresponding to the pads:

$$\begin{array}{c|cccc|} & \lambda_d & \lambda_3 & \lambda_2 & \lambda_1 \\ v'_d & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ v'_1 & \mathbf{0} & 1 & 0 & 0 \\ \hline v_d & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ v_1 & \mathbf{0} & 0 & 1 & 0 \\ \hline \lambda_d & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \lambda_3 & \mathbf{0} & 1 & 0 & 0 \\ \lambda_2 & \mathbf{0} & 0 & 1 & 0 \\ \lambda_1 & \mathbf{0} & 0 & 0 & 1 \end{array}$$

- **Shared memory.**

  In our implementation of matrix multiplication, each thread calculates the value of exactly one cell in the result. We arranged the threads so that all threads within a block write to adjacent cells within the same row of the result. This means that all of the threads will read from exactly the same part of the vectors being multiplied. This yields significant data reuse. So, we copied the relevant segments of the vectors into shared memory and observed a 25% reduction in the running-time.

5.2.5 Limitations of this approach

To our surprise, these optimizations only barely beat the performance of our CPU implementation of 0CFA. Through debugging, we narrowed the cause of the poor performance down to a few principal factors:

- **Unnecessary operations.**

  The first phase of the store update operation generates a new store which contains new bindings; the second phase then adds this new store to the old store. Not only does this waste memory on a second store, the update only impacts a small number of locations, yet we pay a cost proportional to the size of the store—*twice*.

- **Failure to exploit "superposition."**

  EigenCFA can do things 0CFA can't: EigenCFA can represent multiple terms, *e.g.*, call sites, in the same vector, simply by setting additional bits. This gives EigenCFA control over a different kind of parallelism—the ability to act on "superimposed" terms. Yet, our implementation ignores this ability.

- **Unutilized parallelism.**

  Since the analysis is flow-insensitive, all of the call sites could be evaluated in parallel and their effects then merged. However, since we were using dense-matrix operations, performing large matrix multiplications for each call site consumed the storage capacity of the GPU. To process each call site required a return to the CPU to retrieve the partial solution from the GPU, and copy the input data for the next call site, sequentializing the computation.

5.3 Success: Sparse matrix implementation

Before constructing our third implementation, we observed that the abstract store tended to be sparse. In fact, 96% of all entries in the final store matrix contained zeros. And, on average, each variable was bound to about two lambda terms. Using a dense matrix to encode the store is inefficient. So, we switched to using a sparse matrix representation of the store, and this optimization turned out to be the critical accelerant for EigenCFA.

5.3.1 A sparse representation

The sparse matrix representation we use is essentially ELL form, in which the sparse matrix has a fixed maximum number of columns. Each row has a header indicating how many cells in that row are active, and an auxiliary array indicates the column for each nonzero element.[5] ELL is best suited to sparse matrices that have roughly comparable numbers of non-zero elements, and lends itself to computationally efficient access to the sparse matrix. This scheme is not as memory-efficient as the more commonly used CSR (Compressed Sparse Row) representation [3], but it has a smaller memory footprint than the dense representation and it retains, for our purposes, the performance advantages.

We allocate our initial store matrix by allocating a fixed number of columns to each row in the store. Initially, we assume no flow set will hold more than 4% of all lambda terms in the program. If, during the course of the analysis, this number proves to be insufficient, we save the store, terminate the analysis and restart with a twice as many columns.

The header cell of each row is the index of the first free slot available in that row. Each entry in a flow set row is the assigned number of the lambda term in the flow set, *e.g.*, if row i contains n, then λ_n may flow to variable v_i.

Running example. We sketch the sparse store representation for our running example below:

$$\begin{array}{c} v'_3 \\ v'_2 \\ \vdots \\ v_1 \\ \\ \lambda_3 \\ \lambda_2 \\ \lambda_1 \end{array} \left[\begin{array}{c|ccc} 2 & 3 & \_ & \_ \\ 3 & 3 & 2 & \_ \\ \vdots & \vdots & \vdots & \vdots \\ 2 & 2 & \_ & \_ \\ \hline 2 & 3 & \_ & \_ \\ 2 & 2 & \_ & \_ \\ 2 & 1 & \_ & \_ \end{array}\right]$$

In order to add λ_2 to the flow set for v_1, we append the number corresponding to λ_2 to the row vector representing v_1, now a linear-time operation. (It's a linear-time operation because we check for membership before appending.)

5.3.2 Optimizing the static matrices

In addition to sparsifying the store, we also compressed the remaining static syntax matrices. Since every variable and λ-term has an assigned row in the store, when evaluating the function and argument expressions of a given call site, all EigenCFA needs is the corresponding row number in the store. With this insight, we reduced the $\mathbf{Fun}$, $\mathbf{Arg}_1$ and $\mathbf{Arg}_2$ matrices to look-up vectors of size $|\mathsf{Call}|$. This eliminates the need to perform a true matrix multiplication in the argument-lookup phase of Algorithm 1.

Running example. The $\mathbf{Fun}$ matrix becomes the vector:

$$\begin{array}{c} \begin{array}{cccc} c_1 & c_2 & c_3 & c_4 \end{array} \\ \left[\begin{array}{cccc} 5 & 1 & 3 & 8 \end{array}\right] \end{array}$$

5.3.3 Parallelizing call-site evaluation

Since the store-update operation was simplified, we freed up enough GPU resources to parallelize each iteration across call sites.

5.3.4 Tolerating race conditions

Since global synchronization on a GPU is expensive, it's important to engineer store update to behave correctly in the presence of races once call sites are analyzed in parallel. It will likely be the case that one GPU thread sees a store has been only partially updated by another GPU thread. Fortunately, abstract transfer functions in 0CFA are monotonic, which means that flow sets only get larger. More importantly, it means that 0CFA is information-preserving during each iteration: working with stale data does not affect the soundness.

We ensure that the store grows monotonically by writing to it only when a GPU thread has new information to add. And, for overall efficiency, we tolerate some rare inefficiencies: a row may have more than one copy of the same lambda term if they race on trying to add the same lambda at the same time. This prevents threads with stale data from overwriting any changes to the store, without the cost of global synchronization.

For instance, let's say that two call sites c_1 and c_2 are being evaluated in parallel and c_2 reads from a row before c_1 updates it. If the evaluation of c_1 does not yield any new store bindings, nothing will be written to the store and the updates of c_2 will be preserved. In the next iteration, these updates will be visible during the evaluation of c_1 and the final result will still be correct.

Robustness against races from monotonicity We can argue more formally that 0CFA is robust in the presence of stale or partially completed stores and out-of-order analysis for call sites. Suppose that the current abstract store $\hat{\sigma}$ is reset at some point to an arbitrarily chosen weaker store, $\hat{\sigma}'$, so that $\hat{\sigma}' \sqsubseteq \hat{\sigma}$. This represents acting on stale or partially completed data. We can show that the analysis will still reach the most precise result:

Theorem 5.1. *If* $\hat{\sigma}' \sqsubseteq \mathrm{lfp}(\hat{F})$ *then* $\hat{F}^{\infty}(\hat{\sigma}') = \mathrm{lfp}(\hat{F})$.

Proof. By definition, $\hat{\sigma}' = \mathrm{lfp}(\hat{F}) \sqcap \hat{\sigma}'$. Applying $\hat{F}^{\infty}$ to both sides yields $\hat{F}^{\infty}(\hat{\sigma}') = \mathrm{lfp}(\hat{F}) \sqcap \hat{F}^{\infty}(\hat{\sigma}')$. By Kleene's fixed-point theorem, $\hat{F}^{\infty}(\hat{\sigma}')$ must also be a fixed point. Tarski-Knaster guarantees this fixed point will be equal to or greater than the least fixed point. Thus, $\mathrm{lfp}(\hat{F}) \sqcap \hat{F}^{\infty}(\hat{\sigma}') = \mathrm{lfp}(\hat{F})$. □

We can also analyze calls in any order—also on partially completed or stale data. We show this by proving that applying the transfer function for any call site to a point below the least fixed point remains consistent with the least fixed point:

Theorem 5.2. *If* $\hat{\sigma}' \sqsubseteq \mathrm{lfp}(\hat{F})$ *then* $\hat{f}_{call}(\hat{\sigma}') \sqsubseteq \mathrm{lfp}(\hat{F})$*, for any call site* $call$.

Proof. By contradiction, and cases: (1) $\hat{f}_{call}(\hat{\sigma}') \sqsupseteq \mathrm{lfp}(\hat{F})$ and (2) $\hat{f}_{call}(\hat{\sigma}') \not\sqsubseteq \mathrm{lfp}(\hat{F})$ and $\hat{f}_{call}(\hat{\sigma}') \not\sqsupseteq \mathrm{lfp}(\hat{F})$. □

5.4 GPU-specific optimization: Texture memory

A final GPU-specific tweak led to additional speedup. The selection matrices (which had by now been reduced to a linear array) were all stored in texture memory, a way of designating read-only data in global memory so that it is cached for low-latency memory access. We did this because texture data doesn't have the size restrictions of constant memory. The use of texture memory resulted in a roughly 20% speedup over the use of global memory.

6. Empirical evaluation

In total, we created four implementations to evaluate and compare. In case it is not clear, EigenCFA and 0CFA have exactly the same precision, so time is the only relevant metric for comparison. *Ultimately, our fastest GPU implementation was a factor of 72 faster than our fastest CPU implementation at scale.* We compared two CPU implementations against two GPU implementations:

- **CPU (S)**: A fast CPU implementation of 0CFA in Racket. We implemented a traditional small-step version of 0CFA according to best practices.
- **CPU (Sp)**: A sparse-matrix CPU implementation of EigenCFA in C. Since many of our optimizations to the sparse-matrix implementation were not GPU-specific, and many good sparse matrix algorithms for the CPU exist, we had to make sure that merely representing 0CFA as a sparse matrix algorithm was not the sole cause of the speedup. We implemented this CPU version of the sparse-matrix algorithm in order to show that the speedups from the GPU mattered. In fact, this matrix CPU version beat our Racket CPU version in performance, so we report our relative speedup against this sparse-matrix CPU implementation instead.
- **GPU (D)**: A dense-matrix implementation of EigenCFA on the GPU. It performed terribly. We implemented this version as a cautionary point of comparison. (This implementation corresponds to Attempt 2.)
- **GPU (Sp)**: A sparse-matrix implementation of EigenCFA on the GPU. We implemented this version to measure the speedup over the fastest CPU version, which ended up being a factor of 72.

6.1 Platform

We evaluated our implementations on an NVidia GTX-480 "Fermi" GPU with 1.5 GB of memory. The host machine (on which we also ran the CPU implementations) was equipped with an Intel i7 CPU running at 2.79 GHz.

6.2 Benchmarks

To benchmark our implementations across a range of program sizes, we exploited Van Horn and Mairson's recent work on the complexity of control-flow analyses [26, 27]. Because Van Horn and Mairson offer *constructive* proofs of complexity, we can extract a "benchmark generator" from these proofs that emits a program of the requested size that will be worst-case to analyze for k-CFA when $k \geq 1$. For 0CFA, the programs it generates are difficult but not worst-case. Might, Smaragdakis and Van Horn's recent work on the complexity of k-CFA also used this generator, and their empirical results provide a sense of its verisimilitude to handwritten benchmarks [21]. In short, these benchmarks are about an order of magnitude harder to analyze than code written by humans.

6.3 Measurements

Table 1 presents the running time of the analysis for each implementation. The first column is the number of terms in the program. A time of "∞" indicates that the analysis took longer than 6 hours to complete. Figure 1 is a plot of the same data. Note the logarithmic scale on the time-axis. As programs grow larger, the sparse-matrix GPU implementation of EigenCFA has a significant advantage over the other implementations.

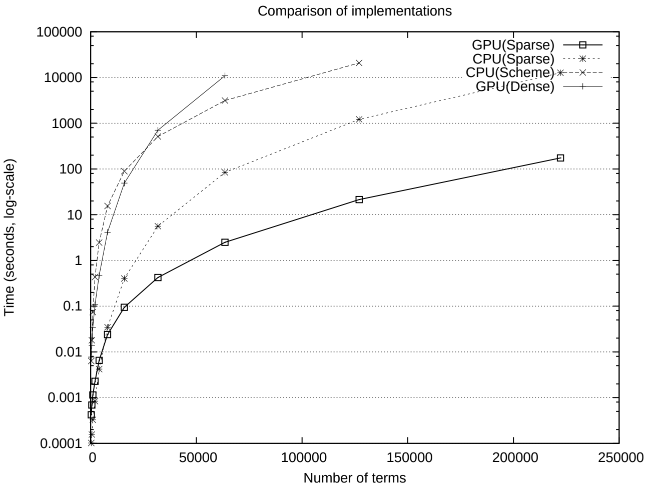

Figure 1. Comparison of implementations: Running times versus number of terms. Note that the time axis is **log-scale**. The sparse-matrix GPU implementation of EigenCFA clearly dominates.

Terms	GPU (Sp)	CPU (Sp)	CPU (S)	GPU (D)
297	0.4 ms	0.1 ms	6.3 ms	7.7 ms
545	0.7 ms	0.16 ms	18 ms	13.9 ms
1,041	1.15 ms	0.33 ms	74.3 ms	34.2 ms
2,033	2.27 ms	0.84 ms	433 ms	0.11 s
4,017	6.51 ms	4.2 ms	2.46 s	0.47 s
7,985	24.01 ms	34.7 ms	15.53 s	4.13 s
15,921	94.48 ms	0.4 s	1m 30s	49.16 s
31,793	0.4 s	5.6 s	8m 30s	11m 43s
63,537	2.49 s	1m 24s	52 min	3hr 2m
127,025	21.3 s	20 min	5hr 46m	∞
222,257	2m 53s	3hr 30 m	∞	∞

Table 1. Analysis running times versus number of terms. (∞ means greater than 6 hours.)

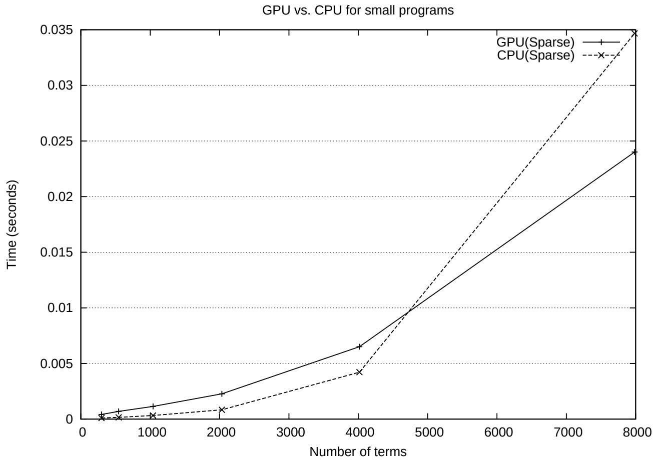

Figure 2. Comparison of GPU and CPU at smaller program sizes. Up to 4500 program terms, the CPU implementation beats the GPU implementation.

In the interest of showing trade-offs, Figure 2 compares the performance of the sparse-matrix implementation of the analysis running on the CPU and the GPU for smaller program sizes. For small programs (less than 4500 terms), the CPU (barely) outperforms the GPU for the following reasons:

- **Kernel invocation cost.**

  There is a fixed cost associated with each invocation of a kernel on a GPU. At smaller program sizes, since the number of iterations and the time taken per iteration is small, the start-up cost becomes a significant percentage of the total running time.

- **Faster CPU clock.**

  The CPU clock runs twice as fast as the GPU (2.8 GHz against 1.4Ghz). In small programs, there isn't as much parallelism that can be exploited by the GPU. In this situation, the slower clock of the GPU limits speed.

- **Fewer CPU iterations.**

  On larger program sizes, the number of iterations is large enough that in the limit, both the serial and parallel algorithms converge in exactly the same number of iterations. For the smaller programs, this isn't true and the GPU often takes twice as many iterations to converge.

Figure 3 is a plot of the speedup obtained by the sparse-matrix GPU implementation of EigenCFA over the CPU. Negative speedup values (barely visible at the left side of the chart) indicate a slowdown. Figure 2 shows this territory of the chart in more detail.

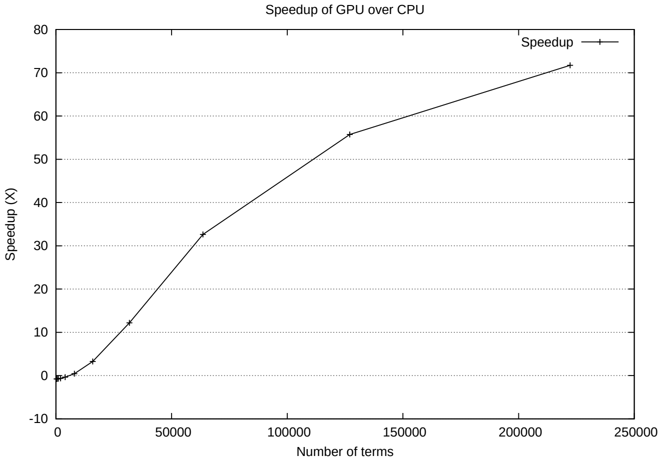

Figure 3. Speedup (in multiples) of GPU over CPU versus the number of program terms.

7. Related Work

Our work in flow analysis descends from a long line of research, beginning with the Cousots' foundational work on abstract interpretation [7, 8], continuing through Jones' work on control-flow analysis [11] and, most recently, through Shivers' work on k-CFA [20, 24, 25].

The literature on parallelizing static analyses is sparse. To the best of our knowledge, there haven't been any efforts to parallelize higher-order control-flow analyses. Notable very recent contributions include Mendez-Lojo *et al.*'s use of multicore architectures to accelerate classical inclusion-based points-to analysis [18] and Lopes *et al.*'s framework for distributed software model checking [16]. Given that 0CFA can also be phrased as an inclusion-based analysis, Mendez-Lojo's techniques are likely applicable.

Most of the prior work in parallelizing static analyses have been focused on data-flow analysis for first-order, imperative programs. Classically, data-flow analyses have been performed by iterative methods such as those originally proposed by Kildall [12] and Hecht [10], elimination methods [2, 23] or hybrid algorithms which combine both approaches [17]. Most efforts at parallelizing data flow analysis have involved parallelizing these algorithms. Zobel [28] and Gupta [9] parallelized Allen-Cocke's interval analysis. Ryder [15] improved on the graph partitioning scheme from [2] which led to a more effective parallel algorithm. Lee implemented the hybrid approach for MIMD architectures using message passing between the processors [14]. Gupta et al. moved away from parallelizing existing algorithms to developing specific techniques for parallel program analysis. In [13], they convert the control-flow graph of a program into a DAG and solve the data-flow problem for each node of the graph in parallel.

The GPU has been used to accelerate static analysis most notably by Banterle and Giacobazzi [4] who implemented the Octagon Abstract Domain (OAD) on a GPU. Since OAD computations are based on matrices, it was easily mapped to the GPU.

8. Future Work

There are at least two promising avenues for future work: adapting our techniques to pointer analysis and exploiting "superposition" within EigenCFA to achieve a genuinely new kind of flow analysis.

8.1 Accelerating pointer analyses

Given recent results unearthing the connection between control-flow and pointer analyses [21], we believe that the techniques presented here can be adapted to pointer analyses as well. Pointer analyses face additional hurdles, such as the fact that its small-step transition relation is much more complex, and a reduction to binary CPS seems out of the question. But these problem do not seem insurmountable.

8.2 Exploiting "superposition"

Using matrices to represent the store and vectors allows "superposition" to be used as another potential source of parallelism. This parallelism is implicit in the structure of the matrices themselves, so it could be exploited in addition to the explicit parallelism that was described in section 5.3.

8.3 Phased analysis

It took a lot of syntactic normalization and abstract Church encodings to impose control-flow uniformity on the abstract transfer function. We hypothesize that a better approach would be to allow different syntactic forms, and then process each kind of form in parallel—to process all function calls in parallel, then all `set!` statements in parallel, then all conditionals, and so on, in each pass of the analysis.

A. The GPU and CUDA

In this section, we provide a brief, high-level description of the architecture, memory hierarchy and programming model of the GPU. CUDA (Compute Unified Device Architecture) is a general purpose parallel computing architecture that leverages the parallel compute engine in NVidia GPUs. The parallel programming model provides three levels of abstraction—a hierarchy of thread groups, shared memories and barrier synchronization [1].

Threads on a GPU are organized into groups called blocks. Only threads within a block can be synchronized cheaply. Synchronization of threads across blocks would have to be done on the host which can be quite expensive. Threads are scheduled and executed in groups of parallel threads called warps. Divergence in execution paths of threads within a warp can have an adverse impact on performance. The GPU has different types of memories:

- **Global memory**: is large, global, read-write, uncached DRAM.
- **Shared Memory**: is small, private to each block, read-write memory whose access time is potentially as low as register access time
- **Constant Memory**: is small, global, read-only[3] and cached.
- **Texture Memory**: is read-only[3] and cached. It is optimized for 2D spatial locality which means that certain memory access patterns can be very efficient.

Data placement in memory and divergent execution in threads must be carefully controlled to maximize performance.

References

[1] *NVIDIA CUDA Programming Guide 2.3*, Aug. 2009.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137+, 1976. ISSN 0001-0782.

[3] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. Sparse Matrices. In *PETSc Users Manual*, chapter 3, pages 55–66. 3.0.0 edition, Dec. 2008.

[4] F. Banterle and R. Giacobazzi. A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware. In H. R. Nielson and G. Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, chapter 20, pages 315–332. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74060-5.

[5] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. ISBN 9781-605-5874-4-8.

[6] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 9781-595-9368-9-9.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[9] R. Gupta, L. Pollock, and M. L. Soffa. Parallelizing data flow analysis. 1990.

[10] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN 0-444-00216-2.

[11] N. D. Jones. Flow Analysis of Lambda Expressions (Preliminary Version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, London, UK, 1981. Springer-Verlag. ISBN 3-540-10843-2.

[12] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[13] R. Kramer, R. Gupta, and M. L. Soffa. The Combining DAG: A Technique for Parallel Data Flow Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):805–813, Aug. 1994. ISSN 1045-9219.

[14] Y. F. Lee, T. J. Marlowe, and B. G. Ryder. Performing data flow analysis in parallel. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 942–951, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-89791-412-0.

[15] Y. F. Lee, B. G. Ryder, and M. E. Fiuczynski. Region Analysis: A Parallel Elimination Method for Data Flow Analysis. *IEEE Trans. Softw. Eng.*, 21(11):913–926, 1995. ISSN 0098-5589.

[16] N. P. Lopes and A. Rybalchenko. Distributed and Predictable Software Model Checking. In *Proc. of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2011.

[17] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4.

[18] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems languages and Applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM. ISBN 9781-450-3020-3-6.

[19] J. Midtgaard and D. Van Horn. Subcubic control flow analysis algorithm. *Higher-Order and Symbolic Computation*, To appear.

[20] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 13–25, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3.

[21] M. Might, Y. Smaragdakis, and D. V. Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315, New York, NY, USA, 2010. ACM. ISBN 9781-450-3001-9-3.

[22] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, Jan. 1995. ISSN 0164-0925.

[23] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986. ISSN 0360-0300.

[24] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, volume 23, pages 164–174, New York, NY, USA, July 1988. ACM. ISBN 0-89791-269-1.

[25] O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[26] D. Van Horn and H. G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 85–96, New York, NY, USA, 2007. ACM. ISBN 9781-59593-815-2.

[27] D. Van Horn and H. G. Mairson. Deciding k-CFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 275–282, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

[28] A. Zobel. Parallel interval analysis of data flow equations. volume II. The Penn State University press, Aug. 1990.

[3] For code running on the GPU, this memory is read-only. Code running on the host, i.e. the CPU, can write to it.