Toward Parallel CFA with Datalog, MPI, and CUDA

THOMAS GILRAY, University of Maryland
SIDHARTH KUMAR, University of Utah

We present our recent experience working to design parallel functional control-flow analysis (CFA) using an encoding in Datalog and underlying relational algebra implemented for SIMD coprocessors and supercomputers. Control-flow analysis statically models the possible propagations of data and control through a target program, finitely obtaining a bound on reachable expressions and environments and on possible return and argument values. We used Soufflé, a parallel CPU-based Datalog implementation from Oracle labs, and worked toward a new MPI-based distributed hash join implementation and an extension of the GPU-based relational algebra library RedFox.

In this paper, we provide introductions to functional flow analysis, Datalog, MPI, and CUDA, explaining the total process we are working on to bring these components together in an analysis pipeline toward the end of scaling functional program analyses by extracting their intrinsic parallelism in a principled manner.

Additional Key Words and Phrases: Control-flow analysis; CFA; Datalog; CUDA; MPI; Parallelism; Static analysis

ACM Reference format:
Thomas Gilray and Sidharth Kumar. 2016. Toward Parallel CFA with Datalog, MPI, and CUDA. 1, 1, Article 1 (January 2016), 12 pages.
DOI: 0000001.0000001

1 INTRODUCTION

A *control-flow analysis* (CFA) of a functional programming language models the propagation of data flows and control flows through a target program, across all possible executions. In static analyses of functional languages generally, a model of where functional values (i.e., lambdas, closures) can flow is required to model where control can flow from a call site, and vice versa. At a call site in Scheme, `(f x)`, the possible values for `f` determine which lambda bodies can be reached. Likewise, the possible callers for the lambda that binds `f`, `(lambda (... f ...) ...)`, influence the lambdas that can flow into `f`. This mutual dependence of data flow and control flow can be handled using an abstract interpretation, simultaneously modeling all interdependent language features. There are systematic approaches to designing analyses such as these, however the traditional worklist algorithms used to implement them in practice are inefficient and have difficulty scaling. Even with optimizations such as global-store widening and flat environments, the analysis is in $O(n^3)$ in the flow-insensitive case or in $O(n^4)$ in the flow-sensitive case. Using more advanced forms of polyvariance or context-sensitivity, the analysis becomes significantly more expensive.

In this paper, we describe our ongoing work to encode these analyses as declarative datalog programs and implement them as highly parallel relational algebra (RA), on the GPU and across many CPUs. Relational algebra operations, derived from Datalog analysis specifications, are computationally intensive and memory-bound in nature. GPUs provide massive fine-grained parallelism and great memory bandwidth, potentially making them the ideal target for solving these operations. We use Redfox (Wu et al. 2014), an open source tool which executes queries expressed in a specialized query language on GPUs. We also modify Redfox, adding capabilities to perform fixed-point iterations essential for solving RA operations derived from Datalog. We are also pursuing a Message Passing Interface (MPI)-based backend for solving RA operations across many compute nodes

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 0000001.0000001

on a network. This approach is also particularly promising, given that HPC is increasingly mainstream and supercomputers are getting faster cores and lower latency interconnects.

2 CONTROL-FLOW ANALYSIS

This section introduces control-flow analysis by instantiating it for the continuation-passing-style (CPS) λ-calculus. We follow the abstracting abstract machines (AAM) methodology, a systematic process for developing a static analysis (an approximating semantics) from a precise operational semantics of an abstract machine.

Static analysis by abstract interpretation proves properties of a program by running code through an interpreter powered by an *abstract semantics* that approximates the behavior of an exact *concrete semantics*. This process is a general method for analyzing programs and serves applications such as program verification, malware/vulnerability detection, and compiler optimization, among others (Cousot and Cousot 1976, 1977, 1979; Midtgaard 2012). Van Horn and Might's approach of *abstracting abstract machines* (AAM) uses abstract interpretation of abstract machines for *control-flow analysis* (CFA) of functional (higher-order) programming languages (Johnson et al. 2013; Might 2010; Van Horn and Might 2010). The AAM methodology is flexible and allows a high degree of control over how program states are represented. AAM provides a general method for automatically abstracting an arbitrary small-step abstract-machine semantics to obtain an approximation in a variety of styles. Importantly, one such style aims to focus all unboundedness in a semantics on the machine's address-space. This makes the strategy used for the allocation of addresses crucial to the tradeoff struck between precision and complexity (Gilray et al. 2016a), and results in a highly flexible and tunable analysis infrastructure. More broadly, the approach has been used to instantiate both traditional finite flow analyses and heavy-weight program verification (Nguyen et al. 2014).

2.1 A Concrete Operational Semantics

This section reviews the process of producing a formal operational semantics for a simple language (Plotkin 1981), specifically, the untyped λ-calculus in *continuation-passing style* (CPS). CPS constrains call sites to tail position so that functions may never return; instead, callers explicitly pass a continuation forward to be invoked on the return value (Plotkin 1975). This makes our semantics tail recursive (small-step) and easier to abstract while entirely eliding the challenges of manually managing a stack and its abstraction, a process previously discussed in the context of AAM (Johnson and Van Horn 2014; Van Horn and Might 2010). Using an AAM that explicitly models the stack in a precise manner, while allowing for adjustable allocation, has also been recently addressed (Gilray et al. 2016b).

The grammar structurally distinguishes between call-sites *call* and atomic-expressions *ae*:

$$\begin{aligned}
call \in \mathsf{Call} &::= (ae\ ae\ \ldots) \mid (\mathtt{halt}) \\
lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ call) \\
ae \in \mathsf{AE} &::= lam \mid x \\
x \in \mathsf{Var}\ &\textit{is a set of program variables}
\end{aligned}$$

Instead of specifically affixing each expression with a unique label, we assume two identical expressions occurring separately in a program are not equal. While a direct-style language with a variety of continuations (e.g., argument continuations, `let`-continuations, etc.), or extensions such as recursive-binding forms, conditionals, mutation, or primitive operations, would add complexity to any semantics, they do not affect the concepts we are exploring and so are left out.

We define the evaluation of programs in this language using a relation $(\to_\Sigma)$, over states of an abstract-machine, which determines how the machine transitions from one state to another. States (ς) range over control expression (a call site), binding environment, and value store components:

$$\begin{aligned}
\varsigma \in \Sigma &\triangleq \mathsf{Call} \times \mathit{Env} \times \mathit{Store}\\
\rho \in \mathit{Env} &\triangleq \mathsf{Var} \rightharpoonup \mathit{Addr}\\
\sigma \in \mathit{Store} &\triangleq \mathit{Addr} \rightharpoonup \mathit{Value}\\
a \in \mathit{Addr} &\triangleq \mathsf{Var} \times \mathbb{N}\\
v \in \mathit{Value} &\triangleq \mathit{Clo}\\
\mathit{clo} \in \mathit{Clo} &\triangleq \mathsf{Lam} \times \mathit{Env}
\end{aligned}$$

Environments (ρ) map variables in scope to an address for the visible binding. Value stores (σ) map these addresses to values (in this case, closures); these may be thought of as a model of the heap. Both these functions are partial and accumulate points as execution progresses.

Evaluation of atomic expressions is handled by an auxiliary function $(\mathcal{A})$ which produces a value (clo) for an atomic expression in the context of a state (ς). This is done by a lookup in the environment and store for variable references (x), and by closure creation for λ-abstractions (lam). In a language containing syntactic literals, these would be translated into equivalent semantic values here.

$$\begin{aligned}
\mathcal{A} &: \mathsf{AE} \times \Sigma \rightharpoonup \mathit{Value}\\
\mathcal{A}(x,\ (\mathit{call},\ \rho,\ \sigma)) &\triangleq \sigma(\rho(x))\\
\mathcal{A}(\mathit{lam},\ (\mathit{call},\ \rho,\ \sigma)) &\triangleq (\mathit{lam},\ \rho)
\end{aligned}$$

The transition relation $(\to_\Sigma) : \Sigma \rightharpoonup \Sigma$ yields at most one successor for a given predecessor in the state-space Σ. This is defined:

$$\overbrace{((\mathit{ae}_f\ \mathit{ae}_1\ \ldots\ \mathit{ae}_j), \rho, \sigma)}^{\varsigma} \to_\Sigma (\mathit{call}', \rho', \sigma')$$

$$\begin{aligned}
\text{where}\quad ((\lambda\ (x_0 \ldots x_j)\ \ \mathit{call}'), \rho_\lambda) &= \mathcal{A}(\mathit{ae}_f,\ \varsigma)\\
v_i &= \mathcal{A}(\mathit{ae}_i,\ \varsigma)\\
\rho' &= \rho_\lambda[x_i \mapsto a_i]\\
\sigma' &= \sigma[a_i \mapsto v_i]\\
a_i &= (x_i, |\mathit{dom}(\sigma)|)
\end{aligned}$$

Execution steps to the call-site body of the lambda invoked (as given by the atomic-evaluation of ae_f). This closure's environment (ρ_λ) is extended with a binding for each variable x_i to a fresh address a_i. A particular strategy for allocating a fresh address is to pair the variable being allocated for with the current number of points in the store (a value that increases after each set of new allocations). The store is extended with the atomic evaluation of ae_i for each of these addresses a_i. A state becomes stuck if `(halt)` is reached or if the program is malformed (*e.g.*, a free variable is encountered).

To fully evaluate a program call_0 using these transition rules, we *inject* it into our state space using a helper $\mathcal{I} : \mathsf{Call} \to \Sigma$:

$$\mathcal{I}(\mathit{call}) \triangleq (\mathit{call}, \varnothing, \varnothing)$$

We may now perform the standard lifting of $(\to_\Sigma)$ to a collecting semantics defined over sets of states:

$$s \in S \triangleq \mathcal{P}(\Sigma)$$

Our collecting relation $(\rightarrow_s)$ is a monotonic, total function that gives a set including the trivially reachable state $\mathcal{I}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$s \rightarrow_s \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightarrow_\Sigma \varsigma'\} \cup \{\mathcal{I}(call_0)\}$$

If the program $call_0$ terminates, iteration of $(\rightarrow_s)$ from $\bot$ (i.e., the empty set $\varnothing$) does as well. That is, $(\rightarrow_s)^n(\bot)$ is a fixed point containing $call_0$'s full program trace for some $n \in \mathbb{N}$ whenever $call_0$ is a terminating program. No such n is guaranteed to exist in the general case (when $call_0$ is a non-terminating program) as our language (the untyped CPS λ-calculus) is Turing-equivalent, our semantics is fully precise, and the state-space we defined is infinite.

2.2 An Abstract Operational Semantics

Now that we have formalized program evaluation using our concrete semantics as iteration to a (possibly infinite) fixed point, we are ready to design a computable approximation of this fixed point (the exact program trace) using abstract interpretation. Previous work has explored a wide variety of approaches to systematically abstracting a semantics like these (Johnson et al. 2013; Might 2010; Van Horn and Might 2010). Broadly construed, the nature of these changes is to simultaneously finitize the domains of our machine while introducing non-determinism both into the transition relation (multiple successor states may immediately follow a predecessor state) and the store (multiple values may become conflated at a single address). We use a finite address space to cut the otherwise mutually recursive structure of values (closures) and environments. (Without addresses and a value store, environments map variables directly to closures and closures contain environments). A finite address space yields a finite state space overall and ensures the computability of our analysis. Typographically, components unique to this *abstract* abstract machine wear hats so we can tell them apart without confusing essential underlying roles:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &\triangleq \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &\triangleq Var \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &\triangleq \widehat{Addr} \rightarrow \widehat{Value} \\
\hat{a} \in \widehat{Addr} &\triangleq \mathsf{Var} \\
\hat{v} \in \widehat{Value} &\triangleq \mathcal{P}(\widehat{Clo}) \\
\widehat{clo} \in \widehat{Clo} &\triangleq \mathsf{Lam} \times \widehat{Env}
\end{aligned}$$

Value stores are now total functions mapping abstract addresses to a *flow set* $(\hat{v})$ of zero or more abstract closures. This allows a range of values to merge and inhabit a single abstract address, introducing imprecision into our abstract semantics, but also allowing for a finite state space and a guarantee of computability. To begin, we use a monovariant address set $\widehat{Addr}$ with a single address for each syntactic variable. This choice (and its alternatives) is at the heart of our present topic and will be returned to shortly.

Evaluation of atomic expressions is handled by an auxiliary function $(\hat{\mathcal{A}})$ which produces a flow set $(\hat{v})$ for an atomic expression in the context of an abstract state $(\hat{\varsigma})$. In the case of closure creation, a singleton flow set is produced.

$$\begin{aligned}
\hat{\mathcal{A}} &: \mathsf{AE} \times \hat{\Sigma} \rightharpoonup \widehat{Value} \\
\hat{\mathcal{A}}(x,\ (call,\ \hat{\rho},\ \hat{\sigma})) &\triangleq \hat{\sigma}(\hat{\rho}(x)) \\
\hat{\mathcal{A}}(lam,\ (call,\ \hat{\rho},\ \hat{\sigma})) &\triangleq \{(lam,\ \hat{\rho})\}
\end{aligned}$$

The abstract transition relation $(\leadsto_{\hat{\Sigma}}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ yields any number of successors for a given predecessor in the state-space $\hat{\Sigma}$. As mentioned when introducing AAM, there are two fundamental changes required using this approach. Because abstract addresses can become bound to multiple closures in the store and atomic evaluation produces a flow set containing zero or more closures, one successor state results for each closure bound to the address for ae_f. Also, due to the relationality of abstract stores, we can no longer use strong update when extending the store $\hat{\sigma}'$.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \leadsto_{\hat{\Sigma}} (call', \hat{\rho}', \hat{\sigma}')$$

$$\begin{aligned}
\text{where}\quad & ((\lambda\ (x_0 \ldots x_j)\ call'), \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\
& \hat{v}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\
& \hat{\rho}' = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{v}_i] \\
& \hat{a}_i = x_i
\end{aligned}$$

A weak update is performed on the store instead which results in the least upper bound of the existing store and each new binding. Join on abstract stores distributes point-wise:

$$\hat{\sigma} \sqcup \hat{\sigma}' \triangleq \lambda \hat{a}.\ \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a})$$

Unless it is desirable, and provably safe to do so (Might and Shivers 2006), we never remove closures already seen. Instead, we strictly accumulate every closure bound to each $\hat{a}$ (i.e., abstract closures which simulate closures bound to addresses which $\hat{a}$ simulates) over the lifetime of the program. A flow set for an address $\hat{a}$ indicates a range of values which over-approximates all possible concrete values that can flow to any concrete address approximated by $\hat{a}$. For example, if a concrete machine binds $(y, 345) \mapsto clo_1$ and $(y, 903) \mapsto clo_2$, its monovariant approximation might bind $y \mapsto \{\widehat{clo}_1, \widehat{clo}_2\}$. Precision is lost for $(y, 345)$ both because its value has been merged with $\widehat{clo}_2$, and because the environments for $\widehat{clo}_1$ and $\widehat{clo}_2$ in-turn generalize over many possible addresses for their free variables (the environment in $\widehat{clo}_1$ is less precise than that in clo_1).

To approximately evaluate a program according to these abstract semantics, we first define an abstract injection function, $\hat{\mathcal{I}}$, where the store begins as a function, $\bot$, that maps every abstract address to the empty set.

$$\hat{\mathcal{I}} : \mathsf{Call} \to \hat{\Sigma}$$

$$\hat{\mathcal{I}}(call) \triangleq (call, \varnothing, \bot)$$

We again lift $(\leadsto_{\hat{\Sigma}})$ to obtain a collecting semantics $(\leadsto_{\hat{s}})$ defined over sets of states:

$$\hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$$

Our collecting relation $(\leadsto_{\hat{s}})$ is a monotonic, total function that gives a set including the trivially reachable finite-state $\hat{\mathcal{I}}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$\hat{s} \leadsto_{\hat{s}} \hat{s}', \text{where}$$

$$\hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \land \hat{\varsigma} \leadsto_{\hat{\Sigma}} \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(call_0)\}$$

Because $\widehat{Addr}$ (and thus $\hat{\Sigma}$) is now finite, we know the approximate evaluation of even a non-terminating $call_0$ will terminate. That is, for some $n \in \mathbb{N}$, the value $(\leadsto_{\hat{s}})^n(\bot)$ is guaranteed to be a fixed point containing an approximation of $call_0$'s full concrete program trace (Tarski 1955).

2.2.1 Widening and Extension to Larger Languages. Various forms of widening and further approximations may be layered on top of the naïve analysis ($\leadsto_{\hat{s}}$). One such approximation is store widening, which is necessary for our analysis to be polynomial-time in the size of the program. This structurally approximates the analysis above, where each state contains a whole store, by pairing a set of states without stores, with a single, global store that overapproximates all possible bindings. This global store is maintained as the least-upper-bound of all bindings that are encountered in the course of analysis.

Setting up a semantics for real language features such as conditionals, primitive operations, direct-style recursion, or exceptions, is no more difficult, if more verbose. Supporting direct-style recursion, for example, requires an explicit stack as continuations are no longer baked into the source text by CPS conversion. Handling other forms is often as straightforward as including an additional transition rule for each.

3 DATALOG-BASED FLOW ANALYSIS

In this section, we will give an overview of Datalog as a declarative logic-programming language, show how it may be executed using a bottom-up fixed-point algorithm, and discuss our encoding of abstract semantics from the previous section.

3.1 Datalog

A datalog program consists of a set of relations, along with the *rules* pertaining to them. A relation encodes a set of tuples known as *facts*. For example, if we have a relation *Parent*, the fact $Parent(p, c)$ may assert that p is a parent of c. A rule then takes the form $a_0 \ :- \ a_1, \ldots, a_j$ where a comma denotes conjunction and each atom a_i is of the form $r(x, \ldots)$, where r is a relation and each x is a variable. In Datalog the turnstile means "is implied by"; this is because the rules are formally an implication form of a Horn clause. Horn clauses are disjunctions where all but one of the atoms are negated: $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_j$. This is the same as $a_0 \vee \neg(a_1 \wedge \ldots \wedge a_j)$ by De Morgan's laws, which is the same as an implication: $a_0 \iff a_1 \wedge \ldots \wedge a_j$. For example, a rule for computing grandparents can be specified:

$$GrandParent(gp, c) \ :- \ Parent(gp, p), Parent(p, c).$$

3.2 Datalog solvers

A number of strategies exist for executing a datalog program—that is, finding a sound least-fixed-point that contains all the input facts and obeys every rule. Unlike solvers for Prolog and other logic programming systems, bottom-up approaches have tended to prevail (Ullman 1989) although this can depend on extensions to the language and its use. Bottom-up solving begins with the set of facts provided as input, and iterates a monotonic function encoding the set of rules until a least-fixed-point is reached. The data structure for relations and the operation of these increasing iterations varies by approach.

The datalog solver *bddbddb* uses binary decision diagrams (BDDs) to encode relations and supports efficient relational algebra (RA) over these structures (Whaley et al. 2005). BDDs are decision trees that encode arbitrary relations by viewing each bit in the input as a binary decision. They have the potential to be a compact and efficient representation of a relation, but have a worst-case exponential space complexity and are highly sensitive to variable ordering (in what order are bits of input branched on).

The datalog solver Soufflé uses the semi-naïve bottom up algorithm with partial-evaluation-based optimization (Scholz et al. 2016). A relational algebra machine (RAM) executes the program in bottom-up fashion by using efficient underlying data structures for relations (such as prefix trees) and directly selects and propagates tuples in a series of nested loops. This RAM then has a partial-evaluation applied to it that optimizes the interpretation for the specific set of datalog rules. The tool accepts a datalog program as input, performs this partial evaluation and writes a C++ program that parallelizes the outside loops of each RA operation using pthreads.

3.3 Encoding CFA

Various program analysis have been implemented using Datalog. Both bddbddb and Soufflé presented program analyses in their experiments. Another prominent example is the DOOP framework for Java (Smaragdakis et al. 2011), used to demonstrate a generalization of object-sensitive flow analysis.

To encode a control-flow analysis as a Datalog problem, we first represent the abstract syntax tree (AST) as a set of relations where each expression and variable in the program has been enumerated. In the soufflé syntax, the encoding for Lambda expressions, conditional expressions, and Variable references is as follows:

```
.decl SyntaxLambdaExp(e:Exp, x:Var, ebody:Exp) input
.decl SyntaxIfExp(e:Exp, x:Var, e0:Exp, e1:Exp) input
.decl SyntaxVarExp(e:Exp, x:Var) input
```

A tuple (e_0, x, e_1) in the SyntaxLambdaExp relation indicates that the expression e_0 is a lambda with the formal parameter x and the expression body e_1. A tuple (e_0, x, e_1, e_2) in the SyntaxIfExp relation indicates that the expression e_0 is a conditional branching on the variable x with the true branch e_1 and false branch e_2. A tuple (e, x) in the SyntaxVarExp relation indicates that the expression e is a variable reference returning the value of variable x.

We then encode the constraints of our analysis directly as rules in datalog, such as:

```
StoreEdge(x,y) :-
  SyntaxVarExp(e,x),
  ReachableExp(e,kaddr),
  KStore(kaddr,y,ebody,kaddr0).
```

This rule adds a flow from x to y when a reachable expression e is returning the value at x and its continuation binds the variable y.

4 MANY-NODE INNER-JOINS AND FUTURE WORK

Modern supercomputers' use of very low latency interconnect networks and highly optimized compute cores opens up the possibility of implementing highly parallel relational algebra using Message-Passing Interface (MPI). MPI is a portable standard library interface for writing parallel programs in a HPC setting and has been highly optimized on a variety of computing infrastructures, from small clusters to high-end supercomputers.

4.1 Parallel Join

Radix-hash join and merge-sort join are two of the most popularly used parallel implementations of the inner join operation. Both these algorithms involve partitioning the input data so that they can be efficiently distributed to the participating processes. For example, in the radix-hash approach a tuple is assigned to a process based on the hash output of the column-value on which the join operation is keyed. With this approach, tuples on both relations that share the same hash value are always assigned to the same process. For every tuple in the left-hand side of the join relation is matched against all the tuples of the right-hand side of the join relation. Fast lookup data-structures like hash tables, or radix-trees (TRIE) can be used to organize the tuples within every process. The initial distribution of data using hashing reduces the overall computation overhead by a factor of the number of processes (n).

More recently (Barthels et al. 2015, 2017), there has been a concerted effort to implement JOIN operations on clusters using an MPI backend. The commonly used radix-hash join and merge-sort join have been re-designed for this purpose. Both these algorithms involve a hash-based partitioning of data so that they are be efficiently distributed to the participating processes and are designed such that inter-process communication is minimized. In both of these implementations one-sided communication is used for transferring data between process. With

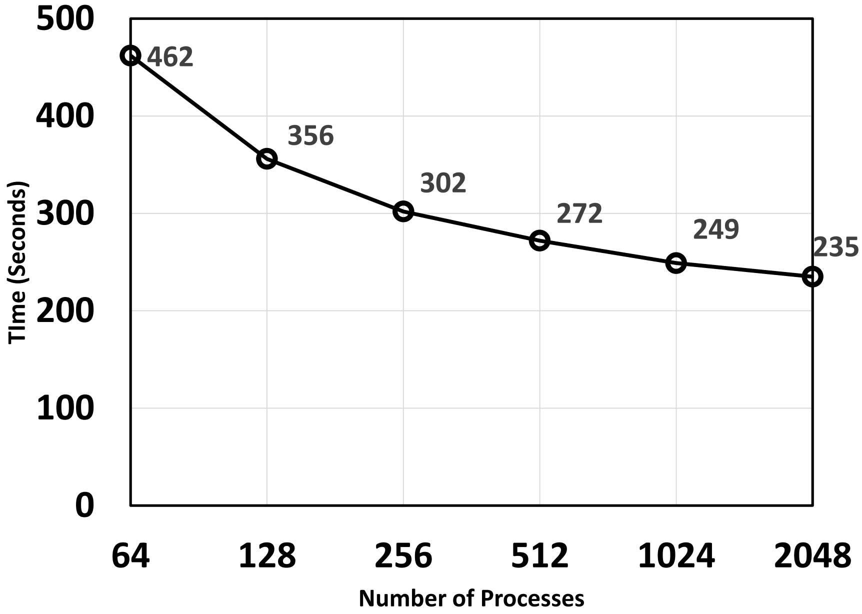
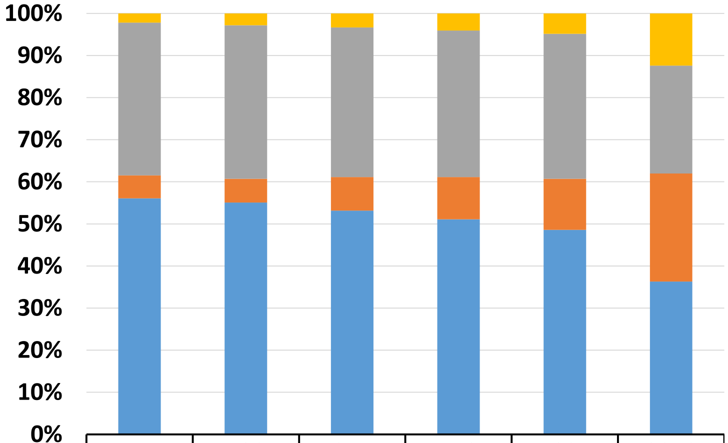


Fig. 1. Strong-scaling results.



Number of processes	64	128	256	512	1024	2048
🟨 Fixed Point Check	10	10	10	11	12	29
■ Insert	167	129	107	94	85	60
🟧 Network exchange	25	20	24	27	30	60
🟦 Join	258	195	160	138	120	85

Fig. 2. Strong-scaling timing breakdown.

one-sided communication the initiator of a data transfer request can directly access parts of the remote memory and has full control where the data will be placed. Read and write operations are executed without any involvement of the target machine. This approach of data transfer involves minimal synchronization between particiapting processes and have been shown to scale better that traditional two-sided communication. The implementation of parallel join has shown promising performance numbers; for example, the parallel join algorithm of (Barthels et al. 2017) ran successfully at 4,096 processor cores with up to 4.8 terabytes of input data.

4.2 Benchmarking: transitive closure

Computing the transitive closure of a graph involves repeated join operations until a fixed point is reached. We use the previously discussed radix-hash join algorithm to distribute the tuples across all processes. The algorithm can then be roughly divided into four phases: 1) Join 2) network communication 3) insertion 4) checking for a fixed point. In our join phase every process concurrently computes the join output of the local tuples. In the next phase every process sends the join output results to the relevant processes. This is a all-to-all communication phase, which we implemet using MPI's all_to_all routines. The next step involves inserting the join output result received from the network to the output graph's local partition. In the final step we check if the size of the output graph changed on any process, if it does then we have not yet reached a fixed point and we continue to another iteration of these 4 steps.

We performed a set of strong-scaling experiments to compute the transitive closure of graph with 412148 edges—the largest graph in the U. Florida Sparse Matrix set (Davis and Hu 2011). We used the Quartz supercomputer at the Lawrence Livermore National Laboratory (LLNL). For our runs, we varied the number of processes from 64 to 2048. A fixed point was attained after 2933 iterations, with the resulting graph containing 1676697415 edges. As can be seen in Figure 1, our approach takes 462 seconds at 64 cores and 235 seconds at 2048 cores, corresponds to an overall efficiency of 6.25%. We investigated these timings further by plotting the timing breakdown of by the four major components (join, network communication, join, fixed-point check) of the algorithm. We observe (see Figure 2) that for all our runs the total time is dominated by computation rather than communication; insert and join together tended to take up close to 90% of the total time. This is quite an encouraging result as it shows that we are not bound primarily by the network bandwidth (at these scales and likely moderately higher ones) and it gives us the opportunity to optimize the computation phase.

5 PARALLELIZING DATALOG ON THE GPU

Programmable GPUs provide massive fine-grained parallelism, higher raw computational throughput, and higher memory bandwidth compared with multi-core CPUs. As a result they are a favorable alternative over traditional CPUs when it comes to high throughput implementations of applications. GPU implementations can potentially provide several orders of magnitude in performance improvement over traditional CPUs. As a result, GPU technology is increasingly widespread and has been successfully adopted by significant number of data-intensive scientific applications such as molecular dynamics (Anderson et al. 2008), physical simulations (Mosegaard and Srensen 2005), and ray tracing in graphics (Parker et al. 2010).

5.1 GPU architecture

Threads provide the finest level of parallelism in a GPU. A GPU application is composed of a series of multi-threaded data-parallel kernels. Data-parallel kernels are composed of a grid of parallel work-units called Cooperative Thread Arrays (CTAs) which in turn consist of an array of threads. In such processors, threads within a CTA are grouped into logical units known as warps that are mapped to SIMD units called Stream Multiprocessors (SMs) (see Figure 3). The programmer divides work into threads, threads map to thread blocks (CTAs), and thread blocks map to a grid. The compute work distributor allocates thread blocks to SMs. Once a thread block is distributed to an SM the resources for the thread block are allocated (warps and shared memory) and threads are divided into groups of (typically) 32 threads called warps.

5.2 Redfox

We use Redfox (Wu et al. 2014) a GPU-based open-source tool to run the relational algebra (RA) kernels translated from Datalog. Redfox is used for compiling and executing queries expressed in a specialized query language on GPUs. Typically, the parallelism involved in solving relational-algebra operations on GPUs is challenging due to unstructured and irregular data access as opposed to other domain-specific operations, such as those common to dense linear algebra. Redfox tackles these issues and provides an ecosystem to accelerate relational computation including algorithm design, system implementation, and compiler optimizations. It bridges the semantic gap between relational queries and GPU execution models, allowing its clients to operate exclusively in terms of RA semantics, and maintains significant performance speedup relative to the baseline CPU implementation.

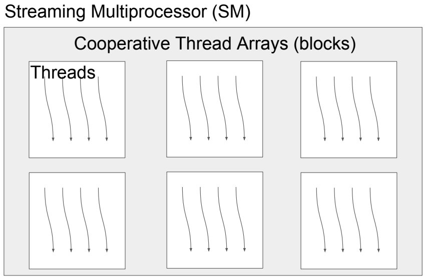

Streaming Multiprocessor (SM)
Cooperative Thread Arrays (blocks)
Threads

Fig. 3. High-level overview of GPU.

Redfox takes advantage of the fine-grained massive parallelism offered by GPUs. It is comprised of (a) a specialized language front-end that allows the specification of sequences of RA operations that combine to form a query, (b) an RA to GPU compiler, (c) an optimized GPU implementation of select RA operators, and (d) a supporting runtime. The relational data is stored as a key-value store to support a range of workloads corresponding to queries over data sets. We use our own system to transform datalog queries into RA kernels. Redfox provides a GPU implementation of the following set operations: union, intersection, difference, cross product, inner-join, project and select. Among all the RA primitive operators, inner-join is the most complex and is more compute intensive than the rest of the RA primitives. Another problem with joins is that their output size can vary, i.e. between zero to the product of the sizes of the two inputs. One of the salient contributions of redfox is an optimal implementation of the join operation (Haicheng Wu and Yalamanchili 2014).

5.3 Fixed-point iterations with Redfox

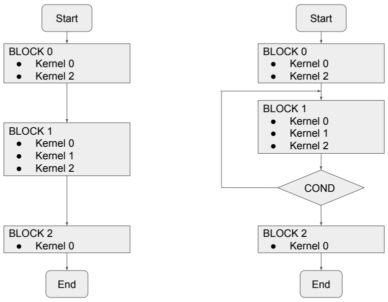


Fig. 4. Redfox execution with (right) and without (left) conditional branches.

One of the major challenges in adapting redfox to solve RA kernels derived from datalog queries was to perform fixed-point iterations. For fixed-point iterations redfox needed to process loops and, until now, Redfox was only used in a sequential mode, where a block unconditionally transitioned to the next block. In the original Redfox paper's experiments, the authors did use a fixed-point computation, but had manually unrolled the benchmark to the needed number of iterations. In our application, we need the ability to run basic blocks (each a straight-line sequence of RA kernels), in a loop, until the relation in contention does not change and the system reaches a fixed point—regardless of how many loops this requires. In order to facilitate execution of loops in redfox, we have added conditional branches, that allows execution to choose a target basic block based on the equality of two input relations. We used the COND kernel of GPU and use the outcome of the kernel to schedule the target block. Typically, in fixed-point iterations we check if the values stored in relation after execution of a certain kernel changed or not, if it remains unchanged then we have attained a fixed point and the execution can move forward, otherwise the set of kernel is executed again (see Figure 4).

5.4 Preliminary results

We evaluated the performance of Redfox in computing the transitive closure of large-sized graphs. For benchmarking we used the open source graphs available at (Davis and Hu 2011). Out of all relation operations used in computing the transitive closure, join is computationally the most complex. We found that the join operation manage to scale decently well with larger graphs. Time consumed performing join operation across 188 iterations for input graph of 25,674 edges (output size is 6,489,757 edges) took 3.6 seconds. Total time for other kernal operation (project, union, copy) along with I/O time was 3.3 seconds. This total time (3.6 + 3.3 seconds) is almost comparable to the time taken by the highly optimized code Souffle (5.6 seconds) to compute the transitive closure of the same graph. We surmise, that Souffle is able to extract parallelism sufficient enough to solve the problem for this graph. Our hypothesis is that the GPU performance may become significantly faster than Souffle for very large scale graphs.

6 FUTURE WORK AND CONCLUSION

We have outlined a possible pipeline for extracting parallelism from a control-flow analysis in a principled way and have implemented GPU-based and MPI-based transitive closure algorithms to experiment with parallizing this kind of problem. We are also interested in writing PGAS based backends for our RA kernels. Partitioned global address space (PGAS) is a commonly used parallel programming model, that follows the ideals of shared memory access but operates in a distributed setting—it assumes a global memory address space that is logically partitioned, portions of which are local to each process. The two main implementations of this programming model are chapel (Chamberlain et al. 2007) and UPC++.

REFERENCES

Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 227, 10 (2008), 5342 – 5359. DOI:http://dx.doi.org/10.1016/j.jcp.2008.01.047

Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1463–1475. DOI:http://dx.doi.org/10.1145/2723372.2750547

Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 517–528. DOI:http://dx.doi.org/10.14778/3055540.3055545

B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. DOI:http://dx.doi.org/10.1177/1094342007078442 arXiv:http://dx.doi.org/10.1177/1094342007078442

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Paris, France, 106–130.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, Los Angeles, CA, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, San Antonio, TX, 269–282.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI:http://dx.doi.org/10.1145/2049662.2049663

Thomas Gilray, Michael D. Adams, and Matthew Might. 2016a. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. *Proceedings of the International Conference on Functional Programming (ICFP)* (September 2016).

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016b. Pushdown Control-Flow Analysis For Free. *Proceedings of the Symposium on the Principles of Programming Languages (POPL)* (January 2016).

Molham Aref Haicheng Wu, Daniel Zinn and Sudhakar Yalamanchili. 2014. Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs. In *The 5th International Workshop on Accelerating Data Management Systems Using Modern Processor and*

Storage Architectures (ADMS).

J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *Proceedings of the International Conference on Functional Programming*.

J. Ian Johnson and David Van Horn. 2014. Abstracting Abstract Control. In *Proceedings of the ACM Symposium on Dynamic Languages*.

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *Comput. Surveys* 44, 3 (Jun 2012), 10:1–10:33.

Matthew Might. 2010. Abstract Interpreters for free. In *Static Analysis Symposium*. 407–421.

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 13–25.

J. Mosegaard and T. S. Srensen. 2005. Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU. In *Eurographics Symposium on Virtual Environments*, Erik Kjems and Roland Blach (Eds.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/EGVE/IPT_EGVE2005/105-111

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 139–152. DOI: http://dx.doi.org/10.1145/2628136.2628156

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH '10)*. ACM, New York, NY, USA, Article 66, 13 pages. DOI: http://dx.doi.org/10.1145/1833349.1778803

G. D. Plotkin. 1975. Call-by-name, call-by-value and the lambda-calculus. In *Theoretical Computer Science 1*. 125–159.

Gordon D Plotkin. 1981. A structural approach to operational semantics. (1981).

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 196–206.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhotak. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Symposium on Principles of Programming Languages*. 17–30.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

Jeffrey D Ullman. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 140–149.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *International Conference on Functional Programming*. 51.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118.

Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 44, 11 pages. DOI: http://dx.doi.org/10.1145/2544137.2544166