GPU-Accelerated Flow-Sensitive Pointer Analysis for C/C++ Programs

JIAQI HE, University of Alberta, Canada
KARIM ALI, NYU Abu Dhabi, UAE

Flow-sensitive pointer analysis offers highly precise results that are essential for various security analyses, bug detection tools, and compiler optimizations. However, its high computational cost often leads to prohibitively long analysis times, especially for large, real-world programs. Despite decades of research, state-of-the-art algorithms still struggle to achieve acceptable performance at industrial scale, forcing developers to choose less precise alternatives.

To overcome these limitations, we present GPA, a GPU-accelerated flow-sensitive pointer analysis for C/C++ programs. To maximize hardware utilization, GPA dynamically balances computation by combining the massive parallelism of GPUs with a graph neural network that predicts per-variable workloads. Compared to state-of-the-art CPU-based analyses, GPA improves runtime performance by a factor of 1.3× to 14× on large programs (i.e., ≥275 KLOC LLVM IR) without sacrificing precision. However, on most small programs (i.e., <100 KLOC LLVM IR) and some medium ones (i.e., 100–275 KLOC LLVM IR), traditional CPU implementations run faster due to the memory management overhead on GPUs that GPA incurs. By making the computation of highly precise pointer information more tractable, GPA enables running analyses and developer tools that were previously infeasible on large codebases.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → **Program analysis**; **Massively parallel algorithms**.

Additional Key Words and Phrases: GPGPU, Flow-sensitive Pointer Analysis

ACM Reference Format:
Jiaqi He and Karim Ali. 2026. GPU-Accelerated Flow-Sensitive Pointer Analysis for C/C++ Programs. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE138 (July 2026), 22 pages. https://doi.org/10.1145/3808145

1 Introduction

Understanding patterns of memory accesses in a program is essential to various data-flow analyses that are used in compiler optimizations (e.g., dead code elimination [34]), reasoning about semantic properties of programs (e.g., data race detection [10]), and detecting security vulnerabilities (e.g., memory safety checking [38]). Pointer analysis achieves this understanding by computing memory locations that program variables may point to during execution. The more precise the pointer analysis, the more accurate downstream analyses or optimizations. For example, high-level synthesis [26] requires precise pointer information to reduce unnecessary memory sharing between instructions. Compilers such as LLVM [15] also rely on precise pointer information to perform safe, behavior-preserving optimizations.

Since pointer analysis is an undecidable problem [25], practical analyses must trade off between precision and scalability [32]. A precise analysis is ideally *flow-sensitive* (i.e., distinguishes results by program locations), *context-sensitive* (i.e., distinguishes results by calling contexts), and *field-sensitive*

Authors' Contact Information: Jiaqi He, University of Alberta, Edmonton, Alberta, Canada, jhe15@ualberta.ca; Karim Ali, NYU Abu Dhabi, Abu Dhabi, UAE, karim.ali@nyu.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE138
https://doi.org/10.1145/3808145

(i.e., distinguishes different fields of the same object). Among these configurations, flow-sensitive pointer analysis provides highly precise results but suffers from severe scalability issues. In fact, our empirical evaluation later shows that state-of-the-art CPU-based implementations [28] may require more than an hour to analyze programs with only a few hundred thousand lines of code. Such a long running time renders the analysis impractical in developer workflows where it is a prerequisite for other client analyses. Therefore, highly precise pointer analyses remain rarely adopted in industry-grade static analyzers and compilers [16].

Several parallel CPU-based approaches [21, 36] have improved performance on small (i.e., <100 KLOC) and medium (i.e., 100–275 KLOC) programs by up to 4.4× compared to prior work that is more than a decade old. More recent work [2] reports an average speedup of 5.3× compared to the same outdated baselines. Nevertheless, these approaches remain too slow for large, real-world codebases. This is because flow-sensitive pointer analysis has traditionally been formulated as a fixed-point computation problem over a program's Control-Flow Graph (CFG) or Def-Use Graph (DUG). This model is limited in its ability to handle large programs efficiently, particularly those with more than 200K lines of code, due to the massive size of the analyzed graph. To improve scalability through parallel execution on CPUs, another line of work models flow-sensitive pointer analysis and strong updates as a Context-Free Language (CFL) reachability problem that may be parallelized either locally [40] or distributively [30]. However, these approaches rely on dynamically deleting points-to relations to support strong updates. Unfortunately, such deletion of points-to relations is difficult to extend to more advanced hardware platforms (e.g., Graphics Processing Unit (GPU)) for higher degrees of parallelism.

Compared to CPUs, GPUs provide massive parallelism, with thousands of threads available to execute computations concurrently, making them an attractive target for accelerating pointer analyses. In fact, prior work has proposed flow-insensitive pointer analyses with context-sensitivity [18, 27, 39] on GPUs. However, to the best of our knowledge, no existing work has developed a flow-sensitive pointer analysis on GPUs. There are two main reasons for this gap. First, supporting flow-sensitivity, especially strong updates, does not naturally fit the CFL reachability model that underlies most prior GPU-based approaches [39]. Reasoning about strong updates with CFL reachability typically requires dynamically removing points-to relationships, which is inherently difficult to express and parallelize on GPUs. Therefore, an effective GPU-based flow-sensitive pointer analysis must carefully reformulate strong updates to avoid removing points-to relationships. Second, flow-sensitive pointer analysis exhibits highly irregular workloads, which means accelerating flow-sensitive pointer analysis using GPUs requires a more sophisticated workload balancing strategy to fully utilize the GPU cores. By design, all threads in a warp [23] must execute the same instruction or idle. Consequently, an ineffective workload balancing strategy may cause several threads to remain idle, leading to GPU under-utilization, which is one of the most critical factors affecting the overall GPU performance.

To overcome these challenges, we present GPA, a GPU-accelerated flow-sensitive pointer analysis for C/C++ programs. To avoid any points-to relation deletions during the analysis, GPA incorporates the graph rewriting system designed by Nagaraj and Govindarajan [21]. To address workload imbalance, GPA employs a Graph Neural Network (GNN)-based scheduling strategy that predicts the computational cost associated with each program variable. This prediction enables more balanced thread assignments by grouping nodes with similar workloads together when launching GPU kernels, reducing idle time and improving GPU utilization.

To evaluate the performance of GPA, we compare it to two baselines: a single-threaded CPU-based analysis (VSFS [2]) and a multi-threaded CPU-based analysis that we have implemented using Soufflé [13], which we refer to as Parallel Flow-Sensitive Pointer Analysis (PFS). Our empirical evaluation across 26 benchmark programs highlights distinct performance trends on different

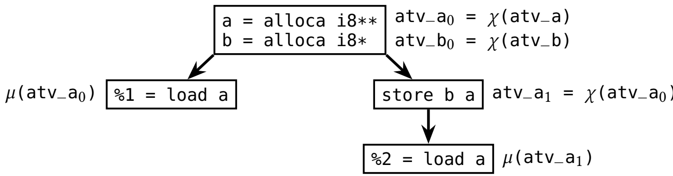

Fig. 1. A simple program written in LLVM Intermediate Representation (IR) with two branches. Each block contains pointer-related instructions, and χ (implicit write of memory) and μ (implicit read of memory) labels are marked to the side of them. Prefix atv_ stands for address-taken variable, which is a variable that may be accessed indirectly via pointers.

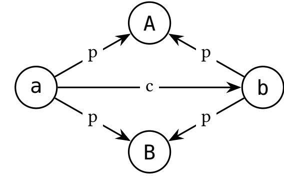

Fig. 2. A constraint graph after fixed-point computation. Each edge contains a label that specifies its type: p is a points-to edge and c is a copy edge.

program sizes. On small programs (i.e., <100 KLOC), VSFS outperforms PFS and GPA because the overhead introduced by both tools outweighs the benefits. Compared to VSFS, for medium programs (i.e., 100–275 KLOC), PFS achieves the highest speedup of 1.6× (min: 0.391×, max: 10.319×, geomean: 1.631×), while GPA achieves a moderate speedup of 1.1× (min: 0.345×, max: 11.733×, geomean: 1.148×). For large programs (i.e., ≥275 KLOC), GPA stands out with the highest speedup of 3.8× (min: 1.328×, max: 13.971×, geomean: 3.847×) compared to VSFS. Thanks to its higher scalability, GPA enables analyzing large programs that both VSFS and PFS cannot analyze even with a 24-hour time budget. By unlocking tractable performance at this scale, GPA enables the deployment of highly precise pointer analyses in security tools, bug detectors, and compilers on large software systems, which was previously infeasible.

To summarize, this paper makes the following contributions:

- A GPU-accelerated flow-sensitive pointer analysis that uses a GNN for workload balancing.
- A thorough evaluation and comparison of a state-of-the-art single-threaded CPU-based analysis, a multi-threaded CPU-based analysis, and our GPU-based flow-sensitive pointer analysis on a comprehensive benchmark suite.
- A detailed discussion on the best use cases for the studied analyses for various program sizes.

2 Background

2.1 Flow-Sensitive Pointer Analysis by Graph Rewriting

Traditional flow-sensitive pointer analysis is modeled as a fixed-point computation problem that propagates points-to sets along a graph representation of an input program. For example, given the program in Figure 1, an analyzer builds a DUG to capture how points-to sets flow from one program location to another. The analysis uses χ functions to define new points-to sets and μ functions to represent uses of existing sets. The analysis then repeatedly propagates these sets until it reaches a fixed point. This model creates major challenges for parallel execution. Propagating sets requires frequent merging operations, which in turn force threads to use atomic updates. If several threads try to merge into the same destination set, only one can proceed at a time while others stall. On large DUG graphs, such conflicts happen often and severely limit multi-threaded scalability.

To avoid these bottlenecks, researchers have reformulated pointer analysis as a graph rewriting problem [18, 19]. In particular, Nagaraj and Govindarajan [21] introduce a flow-sensitive pointer analysis algorithm that replaces set propagation with iterative graph rewriting. The algorithm starts with an initial constraint graph and applies rewriting rules to add new edges until the constraint graph reaches a fixed point. Once complete, the analyzer reads the points-to set of a variable p by collecting all nodes reachable through its points-to edges. For example, Figure 2 shows a constraint

graph with four nodes and five edges. The points-to sets of `a` and `b` are both {`A`, `B`} because they connect to those nodes via points-to edges. In general, the algorithm proceeds in four steps:

(1) Run an Andersen-style pointer analysis [28] on an input program `prog` to obtain preliminary points-to information for constructing memory Static Single Assignment (SSA) form.
(2) Build memory SSA form [8, 29] for `prog` using the results from (1).
(3) Construct an initial constraint graph from the memory SSA form.
(4) Apply graph rewriting rules until reaching a fixed point.

2.2 GPU Architecture and Workload Balancing

Modern GPUs exploit massive fine-grained parallelism. For example, an RTX 4090 [22] contains thousands of cores grouped into Streaming Multiprocessors (SMs) that executes *warps* in a single-instruction multiple-thread (SIMT) fashion. Threads in a warp must follow the same instruction path: if they diverge on control flow, some sit idle while others execute, degrading throughput.

The memory hierarchy of the machine compounds the problem. GPU cores run at lower frequencies than CPUs (e.g., 2.1 GHz vs. 5+ GHz), and global memory access may take 400–800 cycles. A GPU only achieves high utilization when workloads balance well across threads and memory accesses align with its bandwidth strengths. However, pointer analysis rarely exhibits these characteristics, due to the dynamic nature of its constraint graph.

To mitigate this challenge, researchers have explored several workload-balancing techniques. For example, Méndez-Lojo et al. [18] propose a warp-centric strategy. Their approach assigns each graph node to a thread, and each rewriting rule executes at the warp level. The authors also introduce heuristics to cut down synchronization among rules. However, their approach still suffers when threads in the same warp receive nodes with highly imbalanced workloads. In such cases, lightweight threads remain idle while heavier ones finish, wasting GPU resources.

To summarize, traditional CPU-based models and existing GPU-based approaches highlight two core challenges for a GPU-based flow-sensitive pointer analysis:

- **C1**: representing flow-sensitive pointer analysis without excessive synchronization overhead
- **C2**: balancing highly irregular workloads across thousands of GPU threads

In the next section, we present the design of GPA, a GPU-accelerated flow-sensitive pointer analysis that addresses **C1** using graph rewriting and GPU-friendly data structures, and addresses **C2** using a learned workload prediction model.

3 Designing Flow-Sensitive Pointer Analysis for GPU Execution (C1)

The goal of GPA is to accelerate flow-sensitive pointer analysis by parallelizing its fixed-point computation on GPUs. To achieve that, we build on the graph rewriting formulation of Nagaraj and Govindarajan [21] by designing GPU-friendly data structures and execution strategies that efficiently represent, update, and query the dynamically evolving constraint graph. Algorithm 1 outlines the overall algorithm, and the following subsections explain its main components.

3.1 Program Representation and Initial Graph Construction

GPA is a staged analysis, because it requires flow-insensitive pointer analysis results to bootstrap its flow-sensitive pointer analysis. To achieve that, GPA first constructs a memory SSA form of the input program, which makes the definitions and uses of memory accessed via pointers explicit at distinct LLVM IR instructions, allowing the analysis to distinguish the corresponding points-to information. Specifically, memory SSA annotates the input LLVM IR with auxiliary χ and μ labels. Intuitively, χ marks points where a pointer variable is updated and, therefore, creates a new version of its points-to information, while μ marks points where existing points-to information is read.

Algorithm 1: GPA Main Algorithm

Data: A program *prog* in LLVM IR
Result: A CFG annotated with points-to information
1 Perform Andersen-style pointer analysis;
2 Annotate *prog* with χ and μ functions;
3 Initialize points-to graph (with rules from Figure 3);
4 Create Compressed Sparse Row (CSR) for querying metadata of pointers;
5 Transfer all related data to GPU memory;
6 **repeat**
7 | Apply all graph rewriting rules (from Figure 6);
8 **until** *no new edges being added*;
9 Transfer points-to graph from GPU to CPU;
10 **return**;

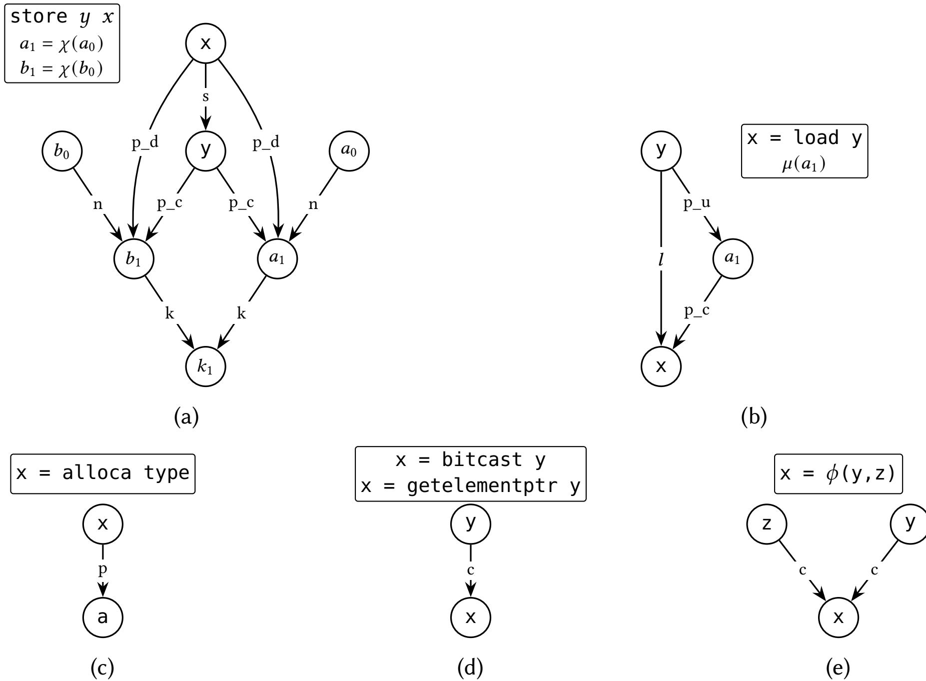

Fig. 3. The initial graph construction rules as defined by Nagaraj and Govindarajan [21], albeit updated to take LLVM IR instructions as input for use within GPA. The figure includes 8 main edge types: store (*s*), load (*l*), non-kill copy (*n*), copy (*c*), define (*d*), use (*u*), klique (*k*), and points-to (*p*). An edge label that starts with p_ denotes a potential edge introduced to conservatively approximate points-to flows.

These annotations allow the analysis to distinguish different versions of a pointer variable across program locations without explicitly tracking program states. In our implementation, GPA uses the Andersen-style pointer analysis from SVF [28] as the flow-insensitive pre-analysis and applies a standard SSA construction algorithm [8, 29] for memory SSA construction.

Using these annotations, GPA constructs an initial constraint graph following the rules in Figure 3. In this graph, nodes represent program variables and abstract memory locations, while edges encode pointer-related constraints that the analysis resolves via graph rewriting. The common edge types `p` (points-to), `c` (copy), `s` (store), and `l` (load) follow directly from LLVM IR semantics.

For a `store` instruction (Figure 3a) that may update points-to information associated with memory locations reachable from `x`, the initial construction rule inserts a set of auxiliary edges to conservatively model its effect without performing immediate strong updates. First, GPA adds an `s`-edge to represent the presence of a store operation. It then introduces `p_d` (potential definition) and `p_c` (potential copy) edges to capture the possibility that a pointer a_1 is defined and that points-to information from `y` may flow into a_1. Because GPA derives def-use relationships from a flow-insensitive pointer analysis, these edges are created conservatively to over-approximate all possible points-to flows. To handle both weak and strong updates, GPA further introduces `n` (non-kill copy) edges and `k` (klique) edges, which allow the analysis to distinguish cases where previous points-to information must be preserved from cases where it can be safely overwritten.

For a `load` instruction (Figure 3b) that reads points-to information from pointer a_1, GPA adds `p_c` and `p_u` (potential use) edges to represent that `y` may use the points-to information associated with a_1 at that instruction, and that this information may subsequently flow into `x`. Finally, for an `alloca` instruction (Figure 3c), GPA introduces a new abstract memory location `a` and connects it to the corresponding pointer variable `x` with a `p`-edge, indicating that `x` initially points to `a`. The remaining cases correspond directly to their LLVM IR semantics and are shown in Figure 3d and Figure 3e.

GPA first constructs the initial constraint graph and stores it in a sparse bit vector on the host CPU. This preprocessing step ensures that the graph is fully available in a compact, GPU-friendly format. GPA then copies the graph to the GPU, where analysis executes in parallel to propagate its constraints through the graph.

3.2 Graph Encoding with Sparse Bit Vectors

After constructing the initial constraint graph, GPA needs an efficient representation that supports frequent updates during its fixed-point computation on the GPU. To meet this requirement, GPA encodes the graph using sparse bit vectors instead of traditional dynamic structures (e.g., adjacency lists with heap allocation). Sparse bit vectors have been used by prior work [18, 39] in static analysis to encode graph data structures due to their ability to efficiently support sparse edge insertions. In particular, a sparse bit vector is a special singly linked list that is designed to hold a set of integers. Figure 4 shows a typical sparse bit vector that contains three fields: *base*, *bits*, and *next*. The base field specifies the range of integers that can be represented in the current sparse bit vector block. A block with base value b and n bits in the bits field can hold integers from $2^b + 1$ to $2^b + n$. The bits field holds a fixed length of bits. Each set bit at i-th position means integer $2^b + i$ is stored in the current block. In this encoding, GPA maintains an array of sparse bit vectors for every edge type, where each sparse bit vector compactly encodes the outgoing edges of a single node. In other words, each node maintains a list of sparse bit vectors, one for every edge type e. For edge type e, the sparse bit vector associated with node i stores all destinations j such that an e-edge exists from node i to j.

Figure 5 illustrates this idea with a small constraint graph and its corresponding encoding. The example contains four pointer variables (IDs 1-4) and four edges of types `p` and `c`. Since there are two edge types (`p` and `c`) in this graph, GPA needs two arrays to represent it. To compactly store the outgoing p-edge from `x` (ID 1), GPA sets the corresponding bit in the first sparse bit vector of

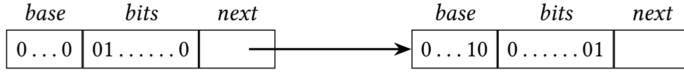

Fig. 4. An example illustrating sparse bit vectors that GPA uses to encode its constraint graph.

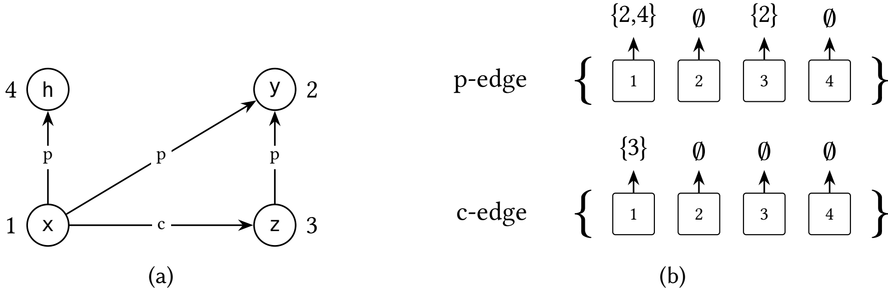

Fig. 5. A simple constraint graph (left) and its corresponding sparse bit-vector in GPA (right). A p-edge stands for a points-to edge. A c-edge stands for a copy edge. Indices in the sparse bit vector correspond to node indices in the original graph. For readability, we show sparse bit vectors as sets of destination node IDs. In the implementation, each entry is stored as one or more sparse bit-vector blocks.

the p-edges array. This setup reflects the fact that there are two p-edges starting from x and ending in y and h.

Since GPUs do not support dynamic allocation with new during execution, GPA prepares a pre-allocated memory pool on the host side. This pool contains all blocks of sparse bit vectors that the main analysis may need at runtime. To work with this pool efficiently, GPA stores the next field of each sparse bit vector block as an unsigned integer rather than a raw pointer. As a result, the next field of a sparse bit vector block specifies the offset for the address of the next block in the memory pool. In our implementation, we allocate space for 70,000,000 sparse bit vector blocks on the GPU to ensure that GPA does not exhaust available blocks on our benchmarks. However, users may configure the number of allocated blocks, provided that the analysis does not exhaust the pool. With each block sized at 128 bytes, this pool consumes roughly 8 GB of memory, which fits most GPUs. While more advanced GPUs may provide more memory, increasing the number of pre-allocated blocks also increases implementation overhead, primarily due to higher memory management and host-device coordination costs. In addition, GPA maintains a sparse bit vector on the CPU that uses a pointer as the *next* field. GPA uses that sparse bit vector to store the initial constraint graph, which it later transfers to the GPU.

3.3 Parallel Graph Rewriting on GPUs

Once GPA transfers the initial constraint graph encoded with sparse bit vectors to the GPU, it begins the fixed-point computation phase. Figure 6 shows the graph rewriting rules that Nagaraj and Govindarajan [21] have originally proposed. Each graph-rewriting rule contains a set of solid edges and one dashed edge, which reads as adding the dashed edge to the graph if all solid edges are present in the graph. In particular, there are three rules (i.e., Figure 6e, Figure 6f, and Figure 6g) that require metadata querying, where an algorithm needs to check if a pointer a and a_1 refer to the same address-taken variable. Since these rules are originally designed for single-threaded CPU execution, using them in a parallel execution environment requires a large number of atomic operations to synchronize the graph. For example, if an analysis launches a thread for each node in

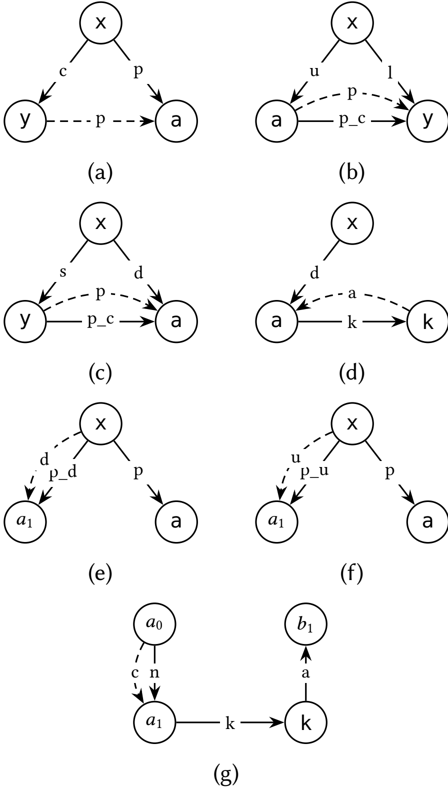

Fig. 6. The graph rewriting rules for flow-sensitive pointer analysis as defined by Nagaraj and Govindarajan [21]. Solid edges are existing relations. Dashed edges are newly added by applying the rules. Subscripts denote variable versions that χ labels introduce.

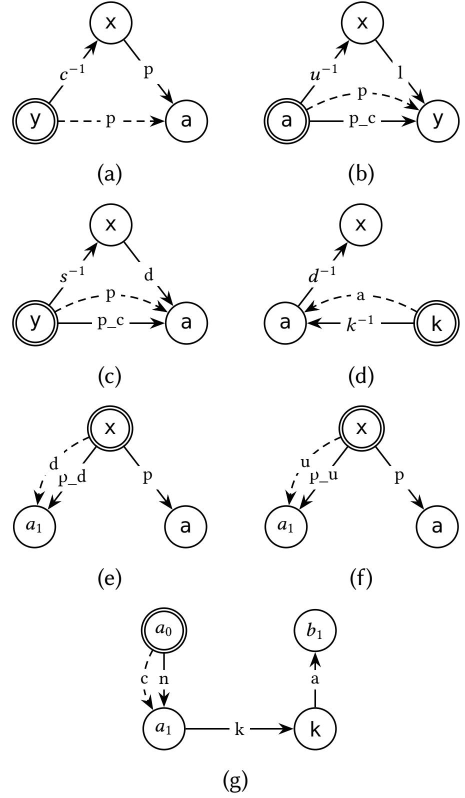

Fig. 7. GPA graph rewriting rules that use reverse edges denoted by the superscript −1. Solid edges are existing relations. Dashed edges are newly added by applying the rules. Subscripts denote variable versions that χ labels introduce. Starting nodes for applying graph rewriting rules are double lined.

the graph to apply the graph rewriting rule in Figure 6a, the operation of adding a new p-edge has to be guarded to avoid race conditions. This is because the rule adds an outgoing edge from a node (e.g., y) that is different from the initial node (i.e., x). Therefore, assigning x to a different thread is not enough to guarantee that only one thread accesses y.

To efficiently execute a graph-rewriting based algorithm on a GPU, GPA must avoid atomic operations as much as possible. To achieve that, our key idea is to introduce reversed edges for certain edge types. Figure 7 shows our modified graph-rewriting rules. In those rules, GPA denotes reverse edges by the superscript -1 for edge types c, u, s, d, and k. The graph-rewriting rules do not require atomic operations when adding new edges, because all new edges are added to a node are assigned to the same thread executing the rule. This design guarantees that no two threads add edges to the same node at the same time. For example, if GPA launches a thread to apply the graph-rewriting rule in Figure 7a on two nodes A and B with two threads t_1 and t_2, there will be no race condition because t_1 only writes to the p-edges of A and t_2 only writes to the p-edges of B.

To avoid race conditions, our graph rewriting rules must satisfy two requirements. First, for each added edge $a \longrightarrow b$, GPA must apply these graph rewriting rules starting from node a. Second, for each edge type e, all edges must be stored consistently as either forward edges or reverse edges. While it is easy to satisfy the first requirement, the second requirement poses a challenge for GPA. In particular, Figure 7 shows both forward k-edges and reverse k-edges (i.e., k^{-1}), because it is not possible to orient k-edges exclusively in one direction such that either k-edges or reverse k-edges (i.e., k^{-1}) appear without violating the first requirement. In fact, we have tried to encode both requirements using the original graph rewriting rules (Figure 6) and then solve them on Z3 [9]. In particular, we treat the existence of each forward edge (e.g., p) and its corresponding reverse edge (e.g., p^{-1}) as boolean decision variables. We then derive constraints from Figure 6 to ensure that newly added edges are always written by the thread responsible for the source node. However, Z3 returns *UNSAT*, which means that at least one edge type must be represented using both forward and reverse edges. Storing both forward and reverse edges for an edge type introduces another synchronization problem. For example, if GPA stores both forward p-edges and reverse p^{-1}-edges and then adds a new p-edge $x \longrightarrow y$, the reverse p^{-1}-edge $y \longrightarrow x$ must also be added, and vice versa. However, k-edges are a special case, because GPA only adds them during graph initialization. No graph rewriting rule in Figure 7 adds a k-edge or k^{-1}-edge, which means that there is no need to synchronize k-edges and k^{-1}-edges, because they remain fixed once constructed. Therefore, limiting this case to only k-edges, GPA efficiently eliminates the need for atomic operations during fixed-point computation on a GPU.

In essence, GPA implements each rewriting rule as a GPU kernel function. For each graph rewriting rule, it launches one thread per node. This design allows thousands of threads to apply the rules in parallel and quickly expand the graph. During each iteration, every kernel reports whether it has added a new edge. If no rule introduces new edges, GPA concludes that it has reached a fixed point and terminates the computation.

3.4 Efficient Relation Checks and Metadata Mapping

Graph-rewriting-based flow-sensitive pointer analysis requires more intricate checks than its flow-insensitive counterpart. In the setting of flow-sensitive pointer analysis, a new pointer is introduced into the computation at each program location where an instruction defines a new points-to set (e.g., `store x y`). The analysis introduces that intermediate pointer (e.g., a_1) as a version of an address-taken variable (e.g., a), creating a new node in the graph for that pointer. Since these intermediate variables are constructed based on the result of an Andersen-style pointer analysis, which is flow-insensitive, the graph rewriting rules should check if the points-to relationship still holds in a flow-sensitive setting. In addition, the graph rewriting rules should check if a strong update is required at a `store` instruction. These checks are handled by the rules in Figure 7e, Figure 7f, and Figure 7g, where GPA must determine whether a program variable originates from an address-taken variable introduced by SSA transformation and whether two pointers refer to the same address-taken variable.

On a CPU, those checks are straightforward using a map m from address-taken variables to the intermediate variables that originate from them. On a GPU, however, GPA relies on lightweight metadata structures for efficiency. To achieve that, GPA precomputes a mapping from each address-taken pointer to the set of intermediate variables that originate from it (e.g., χ functions or address-taken copies). GPA then stores this mapping in a CSR-like layout. Three arrays (`derived_ids`, `derived_offsets`, and `derived_counts`) collectively capture the intermediate variables associated with each address-taken variable. In particular, `derived_ids` stores the flattened IDs of all intermediate variables in m, while the i-th element of `derived_offsets` and `derived_counts`

Fig. 8. A points-to graph with nodes that have different workloads.

store the offset and the number of intermediate variables that originate from the address-taken variable *i*.

At runtime, GPA exposes the device function, which is a function executed directly on the GPU by each thread, called `isVersionOf()`. Given a node, the function first queries its canonical address-taken variable using the node-to-variable map. It then performs a bounded scan over its derived memory region IDs. This design avoids costly string-based pointer name comparisons and supports fast version queries, which are critical for modeling strong updates and metadata-sensitive rewriting directly on the GPU.

4 Workload Balancing of Flow-Sensitive Pointer Analysis (C2)

Although our optimized rewriting rules (Figure 7) reduce most of the GPU overhead, GPA must still balance work across GPU threads. Flow-sensitive pointer analysis generates workloads that vary dramatically across program variables. While some nodes trigger only a handful of rewrites, others may expand into hundreds of edges. Therefore, if GPA assigns these tasks to GPU threads disregarding their imbalance, lightweight threads will sit idle while heavy ones finish, wasting valuable parallel resources. To address this issue, we propose an approach that uses GNN prediction to compute a warp-friendly scheduling.

4.1 Workload Prediction with GNNs

A naive scheduler would assign nodes to threads uniformly at random. This naive scheduling wastes hardware resources on graphs similar to the example in Figure 8. In the figure, `x` has 100 outgoing *p*-edges plus one *c*-edge, which makes `h` add 100 *p*-edges (Figure 6a), while `c` adds only 1 *p*-edge to `b`. If both land in the same warp, the `c` thread sits idle.

To predict these workloads, one could hand-craft rules from the initial points-to graph. However, these heuristics would perform poorly, because rewrites evolve the graph in ways that are hard to approximate. Instead, GPA trains a GNN to predict per-variable workload, grouping variables of similar expected cost into the same warp. Unlike traditional neural networks that require fixed-size vector inputs, GNNs naturally operate on graphs by propagating information along edges so that each node aggregates context from its neighborhood. To predict the workload generated by each variable, we have trained a GNN on the graph representation of the programs in our training dataset. Using a GNN introduces additional runtime overhead for workload prediction. However, this cost is amortized by the substantial reduction in warp-level imbalance.

4.2 GNN Training

We train the workload predictor on a large corpus of LLVM programs. Specifically, we use the DeepDataFlow [7], POJ-104 [20] datasets, and SVF test suite [31], totaling 514K LLVM IR files. For each program, we run GPA without any workload balancing (GPANaive) and record, per node and per iteration, the number of newly added edges by edge type. Rather than predicting an exact

workload value for each node, we formulate workload prediction as a classification problem over a fixed set of workload ranges. This design choice aligns with the needs of the GPA scheduler, which only requires grouping nodes with similar computational cost to achieve warp-level load balance, rather than a total ordering of nodes by precise cost. Concretely, we discretize observed workloads into a fixed set of bins and use the bin index as the training label for each node.

To ensure sufficient context propagation (i.e., allowing each node to aggregate information from multi-hop neighborhoods in the constraint graph that may influence its rewriting workload), we use a Graph Attention Network (GAT) with $T = 32$ iterations of message passing. At each step, node i updates its representation using features from its neighbors:

$$\boldsymbol{h}_i' = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \, \mathbf{W}\boldsymbol{h}_j\right), \tag{1}$$

where $\boldsymbol{h}_i \in \mathbb{R}^F$ is the feature vector of node i, $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is a learnable weight matrix, $\mathcal{N}(i)$ is the set of neighbors nodes of i in the constraint graph, and α_{ij} is the attention coefficient, which is computed as:

$$\alpha_{ij} = \frac{\exp\big(\text{LeakyReLU}\big(\boldsymbol{a}^\top \left[\mathbf{W}\boldsymbol{h}_i \,\|\, \mathbf{W}\boldsymbol{h}_j\right]\big)\big)}{\sum_{k \in \mathcal{N}(i)} \exp(\text{LeakyReLU}(\boldsymbol{a}^\top \left[\mathbf{W}\boldsymbol{h}_i \,\|\, \mathbf{W}\boldsymbol{h}_k\right]))}, \tag{2}$$

where $\boldsymbol{a} \in \mathbb{R}^{2F'}$ is a learnable weight vector and $\|$ denotes concatenation. Intuitively, this mechanism allows each node to focus on the most relevant neighbors when predicting its workload. Rather than extracting explicit features from source code (e.g., node degrees and instruction opcode), we build a heterogeneous dependency graph and initialize its nodes with trainable embeddings, allowing the model to automatically learn predictive representations from graph structure and connectivity patterns.

To train the GNN, we divide the dataset into three disjoint subsets: a training set containing 80% of the data for model learning, a validation set with 10% of the data for early stopping, and a testing set with the remaining 10% reserved for evaluating the final performance of the trained model. We train our the GNN using the Adam optimizer [14] with an initial learning rate of 0.001, employing categorical cross-entropy for classification tasks. We carry out the training for up to 200 epochs, stopping early if the validation loss does not improve for 20 consecutive epochs. To accelerate computation on the GPU, we scale the gradients to FP16 precision. The final model classifies nodes into workload bins with 97% accuracy compared to the collected ground truth.

4.3 Integration into GPA

We integrate workload prediction into GPA as a lightweight preprocessing step that preserves the semantics of the flow-sensitive pointer analysis. GPA uses this workload prediction solely to determine the execution order and grouping of nodes, without changing the rewriting rules or the conditions under which edges are added.

Before each kernel launch, GPA predicts a workload bin for every node and groups nodes by their predicted bins. GPA then launches kernels per bin so that warps process nodes of similar expected cost. Because all graph rewriting rules remain unchanged and every node is eventually processed, this reordering does not affect the fixed-point result. This bin-based scheduling reduces intra-warp divergence, increases GPU utilization, and improves throughput, especially on graphs with highly skewed workloads. We evaluate the effectiveness of this workload prediction strategy in Section 5.7.

5 Evaluation

We have implemented GPA as a standalone analysis that takes the LLVM IR of a program as input, and executes a flow-sensitive pointer analysis on a GPU. As a front-end, we use SVF [28] to parse input programs.

To evaluate GPA, we compare its runtime performance against state-of-the-art CPU-based flow-sensitive pointer analyses. In particular, we evaluate VSFS [2], which is an enhancement of SFS [12] by sharing points-to sets at different program locations. For multi-threaded flow-sensitive pointer analysis, we collect necessary facts to build a constraint graph from SVF and solve the fixed-point of the constraint graph using Soufflé [13], an open-source parallel Datalog engine [17] used by various static analyzers [1, 4]. Soufflé expresses the analysis as a set of rules and executes them in parallel on CPU cores. We do not apply Soufflé autotuning to avoid program-specific optimizations, but design the analysis rules to avoid large intermediate relations. In the remainder of this section, we refer to this implementation as PFS. To ensure fairness in our comparison, we verify that all analyses under study produce identical points-to sets for every benchmark program (i.e., no difference in precision).

Our evaluation aims at answering the following research questions:

(1) **RQ1**: How does GPA compare to single-threaded CPU-based flow-sensitive pointer analyses in terms of runtime performance?
(2) **RQ2**: How does GPA compare to multi-threaded CPU-based flow-sensitive pointer analyses in terms of runtime performance?
(3) **RQ3**: Do program size and structure affect the relative advantages of GPUs versus CPUs?
(4) **RQ4**: Does GNN-based workload prediction help scale the analysis to large programs?

5.1 Benchmark

We evaluate GPA on 29 benchmark programs ranging from 1 KLOC to over 1 MLOC. To avoid overlap with the GNN training data, our selection includes 16 programs from the SPEC CPU 2017 suite [5] and 13 benchmarks used in evaluating VSFS [2], which together represent the standard workloads for pointer analysis research. We omit mutt, used for evaluating VSFS, from our benchmark, due to running into compilation errors when transforming it to LLVM IR. For all other programs, we compile them with Clang 16.0.1 [6], because the last LLVM version that SVF supports is LLVM 16. We chose optimization level `-O0` for all compilations to preserve the program's original control flow and pointer behavior, which is altered at other optimization levels. While higher optimization levels may change IR size and shape, we do not expect them to affect the qualitative performance trends observed in our evaluation. Table 1 summarizes our benchmark statistics. We categorize our benchmark programs as small, medium, or large based on their LLVM IR instruction counts, using cutoffs of 100 KLOC and 275 KLOC according to the empirical distribution of program sizes in our benchmark suite. We use this categorization to facilitate result presentation, and it does not affect the underlying analysis.

For each analysis, the input is a single LLVM bitcode file. For the SPEC CPU 2017 benchmark, we replace the default linker with `llvm-link` to ensure a whole-program bitcode output. For the VSFS benchmark, we use `wllvm`, which also relies on `llvm-link`, to produce a single bitcode file.

5.2 Experimental Setup

We run all experiments on a server with an AMD Ryzen 9 7950X 16-core processor, 125 GB RAM, running Ubuntu 22.04. We have implemented GPA in CUDA 12.5 on an NVIDIA RTX 4090 GPU (23.55 GB memory, warp size 32, 128 SMs). To minimize the impact of the environment, we run each analysis five times per program, and report the average runtime across these runs. We observe

Table 1. Statistics for each benchmark program, including number of lines of the original C/C++ source code in KLOC, number of compiled LLVM IR instructions in KLOC, and number of pointer-related instructions. Programs are listed in an increasing order of the number of LLVM IR instructions. Programs are categorized as small, medium, or large based on their LLVM IR instruction counts.

	Program	# C/C++ (KLOC)	# LLVM IR (KLOC)	Pointer-Related LLVM IR Instructions					
				`alloca`	`load`	`store`	`copy`	`call`	`phi`
Small	LBM	1	8	139	2,771	419	889	71	1
	MCF	3	8	348	2,578	924	1,171	180	30
	DEEPSJENG	10	31	905	9,007	2,803	3,240	943	23
	DPKG	40	58	2,912	14,402	5,081	5,920	5,813	132
	NAB	24	60	2,723	18,802	6,060	6,792	3,016	112
	DU	28	60	2,756	14,819	5,763	6,304	2,008	329
	XZ	33	60	2,695	16,225	6,365	8,985	1,843	122
	LEELA	21	80	8,626	12,993	9,853	8,641	9,665	88
	NANO	59	87	3,289	22,317	7,936	8,389	5,337	429
	PSQL	25	89	3,283	19,321	8,364	5,452	7,135	212
	JANET	23	94	6,018	21,775	8,920	15,134	6,428	352
Medium	I3	27	109	4,352	27,111	8,868	15,313	8,427	887
	BAKE	25	117	3,853	13,670	6,512	4,553	5,547	127
	TMUX	48	128	789	15,897	6,509	19,384	11,910	5,893
	ASTYLE	14	155	3,580	17,216	14,508	32,396	20,502	2,700
	x264	96	194	6,723	52,967	15,177	38,656	5,036	573
	NINJA	17	198	655	5,108	3,326	10,145	4,797	1,745
	MRUBY	48	218	9,558	51,578	21,731	45,816	10,259	809
	NAMD	8	271	18,726	95,651	36,746	30,637	4,404	702
Large	POVRAY	170	275	12,324	72,972	28,356	41,519	15,245	214
	BASH	115	304	11,475	70,325	29,350	19,111	16,832	2,118
	LYNX	134	348	9,131	80,267	25,791	27,046	22,383	3,460
	IMAGICK	259	555	17,538	159,026	57,071	85,586	34,629	804
	OMNETPP	134	556	45,585	94,257	60,440	62,441	57,479	1,396
	PERLBENCH	362	850	21,741	220,260	58,486	132,286	27,615	8,200
	XALANCBMK	520	1,341	122,545	234,904	158,646	143,937	131,551	1,718
	PAREST	427	2,781	267,742	444,110	311,547	300,330	322,417	23,501
	GCC	1,304	3,302	113,305	785,686	248,311	491,234	183,620	23,137
	BLENDER	1,577	3,444	220,616	894,017	369,324	532,919	164,520	12,460

negligible runtime variance across the runs per program. For VSFS, we use the `total time` reported in its analysis statistics. For PFS and GPA, we use the `time` command to measure the wall-clock runtime. When evaluating GPA, we account for prediction overhead, while training cost is excluded as a one-time offline expense. For all runs, we set a timeout of 24 hours and a memory cap of 125 GB. We partition workloads into 10 equal-width bins covering 1–1,000, plus one bin for workloads exceeding 1,000. This fixed scheme avoids program-specific tuning and promotes generalization across programs of varying sizes and structures.

The three largest programs (PAREST, GCC, and BLENDER) exceed 2.7 MLOC in LLVM IR each. Neither the analyses under study nor the flow-insensitive pre-analysis used by GPA could analyze these programs due to memory exhaustion. Even on a machine with 488 GB RAM, Andersen-style analysis fails for these programs. We therefore exclude them from evaluation.

Table 2. The running time (in seconds) for VSFS, PFS, and GPA for each benchmark program, as well as the speedups relative to the baseline VSFS. The fastest running time per program is in bold. We also show the CPU side memory usage (CPU Mem) and GPU side memory usage (GPU Mem), both in MBs, for GPA.

			PFS		GPA			
	Program	**VSFS**	**Time**	**Speedup**	**Time**	**Speedup**	**CPU Mem**	**GPU Mem**
Small (<100 KLOC)	LBM	**0.012**	0.026	0.462	3.27	0.004	7.99	13.93
	MCF	**0.044**	0.053	0.830	2.838	0.016	9.70	18.98
	DEEPSJENG	**0.087**	0.098	0.888	3.066	0.028	32.47	54.42
	DPKG	**2.950**	3.051	0.785	4.544	0.649	85.56	201.21
	NAB	**0.486**	0.573	0.848	3.578	0.136	65.39	131.31
	DU	1.301	**1.101**	1.182	3.87	0.336	81.75	128.61
	XZ	1.252	**0.910**	1.376	3.909	0.320	98.25	126.62
	LEELA	**2.614**	7.690	0.340	5.004	0.522	155.91	252.07
	NANO	5.128	**3.131**	1.638	5.783	0.887	166.97	233.55
	PSQL	2.205	**2.080**	1.060	4.647	0.474	120.41	274.98
	JANET	**4.769**	5.698	0.842	5.837	0.817	146.43	306.52
Medium (100–275 KLOC)	I3	4.314	**4.134**	1.044	6.152	0.701	177.89	329.91
	BAKE	**3.282**	4.022	0.816	5.171	0.635	111.00	206.69
	TMUX	43.862	**12.424**	3.530	15.546	2.821	438.95	362.54
	ASTYLE	365.014	35.374	10.319	**31.109**	11.733	935.37	6395.09
	X264	23.606	**12.903**	1.829	16.333	1.445	1364.78	418.20
	NINJA	1.647	**0.858**	1.920	4.776	0.345	90.89	177.27
	MRUBY	10.47	**8.910**	1.175	9.235	1.134	314.32	495.37
	NAMD	**2.662**	6.807	0.391	7.361	0.362	287.48	539.11
Large (≥275 KLOC)	POVRAY	70.917	44.924	1.579	**26.084**	2.719	1283.65	714.66
	BASH	117.628	181.900	0.647	**40.569**	2.899	1451.49	930.78
	LYNX	502.615	259.183	1.939	**146.003**	3.442	1615.86	1176.59
	IMAGICK	51.255	**27.982**	1.832	38.59	1.328	945.98	1507.68
	OMNETPP	OOT	OOT	OOT	**6,148.381**	≥13.971	23031.05	1887.04
	PERLBENCH	1434.723	687.315	2.087	**516.019**	2.780	10944.14	2522.88
	XALANCBMK	OOT	OOT	OOT	**9692.712**	≥8.914	22317.38	4760.44

5.3 Comparison with Single-Threaded CPU-Based Analysis (RQ1)

Table 2 reports the running times (in seconds) for VSFS, PFS, and GPA, as well as the speedups relative to VSFS as the single-threaded CPU-based baseline implementation. For each benchmark program, we highlight the fastest running time in bold.

Small programs. For these programs, VSFS is consistently faster than GPA. This result is expected, because the GPU overhead (e.g., memory allocation and transfers) takes at least 2–3 seconds, which dominates the total running time when the workload is small. Across all 11 small programs, GPA has an average slowdown of approximately 6× (min: 1.127×, max: 250×, geomean: 5.952×). We calculate these slowdown statistics using the inverse of the speedup numbers that Table 2 reports.

Medium programs. For these programs, GPA begins to see some gains as it outperforms VSFS on 4/8 benchmark programs. Overall, GPA achieves an average speedup of 1.1× (min: 0.345×, max: 11.733×, geomean: 1.148×). The workloads of these programs are large enough to start benefiting from CPU parallelism, but remain insufficient to fully amortize GPU overhead or expose enough parallelism to make GPU execution consistently advantageous.

Large programs. For these programs, the results show the benefits of using GPA. Across the 7 programs, GPA outperforms VSFS with an average speedup of 3.8× (min: 1.328×, max: 13.971×, geomean: 3.847×). At this scale, the GPU overhead is negligible compared to the hundreds of seconds that VSFS requires to compute its results. GPA leverages GPU bandwidth to process millions of new

edges concurrently, delivering substantial acceleration. In particular, GPA successfully analyzes OMNETPP and XALANCBMK, both of which VSFS fails to analyze within the 24-hour time limit.

> For large programs, GPA has an average speedup of 3.8×, enabling analysis of programs that VSFS cannot handle. For medium programs, GPA has a moderate average speedup of 1.1×. However, GPA is not suitable for small programs due to the incurred GPU overhead.

5.4 Memory Consumption of GPA

In addition to runtime, Table 2 reports both CPU-side and GPU-side memory usage for GPA. Across all benchmark programs, CPU memory consumption ranges from 7.99 MB to 22.5 GB, while GPU memory ranges from 13.9 MB to 6.2 GB. For small benchmarks (<100 KLOC), GPU memory usage is modest, typically below 310 MB, with CPU memory under 170 MB. For medium programs, GPU memory ranges from 177 MB to 6.3 GB, while CPU memory is 90 MB–1.33 GB. For large programs, GPU memory ranges from 714 MB to 4.6 GB, whereas CPU memory may exceed 20 GB.

Overall, both CPU and GPU memory grow with program size. GPU memory is comparable to or lower than CPU memory on most large programs, except in cases where poor points-to locality increases sparse bit-vector allocation. A notable exception is ASTYLE, where GPA consumes approximately 6.2 GB of GPU memory but only 0.91 GB of CPU memory. This behavior is caused by irregular points-to locality, which forces GPA to allocate 52,388,555 sparse bit-vector blocks to encode the constraint graph.

> While memory usage generally scales with program size and remains within the capacity of modern GPUs for most workloads, irregular points-to locality can significantly amplify GPU memory consumption.

5.5 Comparison with Multi-Threaded CPU-Based Analysis (RQ2)

Small programs. For these programs, Table 2 shows that PFS outperforms VSFS on 4/11 programs. The overhead of fact generation is smaller than the GPU overhead of GPA. Therefore, for these small programs, PFS also consistently beats GPA. Compared to VSFS, PFS achieves an average speedup of 0.9× speedup (min: 0.34×, max: 1.638×, geomean: 0.857×).

Medium programs. For these programs, Table 2 shows that PFS outperforms VSFS on 6/7 programs with an average speedup of 1.6× (min: 0.391×, max: 10.319×, geomean: 1.631×). Similarly PFS outperforms GPA on 6/7 programs. This is because CPUs can parallelize effectively at this scale without incurring the heavy GPU overhead that GPA incurs.

Large programs. For large programs, PFS still outperforms VSFS but only moderately, with an average speedup of 1.5× (min: 0.647×, max: 2.087×, geomean: 1.5×). Unlike GPA, PFS cannot analyze OMNETPP and XALANCBMK within the time limit (i.e., 24 hours). This is because the CPU cores cannot fully parallelize the millions of edge insertions that are required at the scale of these programs. On the other hand, GPA scales to these programs by utilizing the massive parallelism of GPUs coupled with its GNN workload prediction strategy. Overall, GPA outperforms PFS on 6/7 large programs.

For large programs, GPA outperforms PFS, achieving an average speedup of 3.8× compared to only 1.5× by PFS. The better performance of GPA enables it to analyze large programs that PFS cannot analyze due to heavy fixed-point computation. PFS incurs less overhead than GPA for small programs, achieving the best performance on medium programs with an average speedup of 1.6×.

5.6 Effects of Program Size and Structure on Analysis Time (RQ3)

While program size correlates with runtime, it does not reliably predict the actual analysis time of flow-sensitive pointer analysis. In practice, a large program may be analyzed quickly, whereas a smaller program incurs substantial analysis time. Such cases indicate that program size alone does not determine the pointer-related complexity of a program. To better understand what drives analysis cost, we examine four factors: (1) the number of pointers introduced by SSA, (2) the complexity of def–use relations (i.e., average χs per `store`, average μs per `load` , and measured as #stores$\cdot\chi$ + #loads$\cdot\mu$, which approximates the interaction volume among store and load constraints), (3) the average points-to set size, and (4) the total number of new edges generated during analysis. Table 3 reports these per-program statistics gathered by GPA.

For VSFS, all four metrics exhibit strong positive correlation with analysis time. In particular, the number of pointers and DUG complexity correlate strongly with runtime with Pearson correlation coefficient [24] of 0.78 and 0.87, indicating that larger pointer populations and denser def-use interactions lead to higher analysis cost. The average points-to set size also shows strong correlation with VSFS runtime with the Pearson correlation of 0.97, reflecting the set-based execution model of VSFS, in which the dominant cost arises from repeatedly propagating and merging points-to sets. These results suggest that VSFS is sensitive to both the scale of pointer variables and the volume of points-to relations.

For GPA, the cost model differs fundamentally. Although the number of pointers and DUG complexity correlate positively with runtime (Pearson r = 0.82 and 0.66, respectively), the average points-to set size shows virtually no correlation with analysis time (Pearson $r \approx 0.03$). In other words, large points-to sets do not inherently increase GPU cost. This is because GPA implements flow-sensitive pointer analysis through graph rewriting, where rules match local constraint patterns and generate new edges. Consequently, the dominant cost arises from the volume of rewrite activity rather than from iterating over points-to sets. To confirm this intuition, we measure the total number of edges generated during analysis as a proxy for rewrite work. This metric exhibits a strong positive correlation with runtime (Pearson $r \approx 0.85$), indicating that total rewrite volume drives GPU execution cost.

Summary. These results reveal that the same structural metrics influence VSFS and GPA in fundamentally different ways. While VSFS runtime is strongly affected by points-to set cardinality due to its set-based propagation model, GPA runtime is largely insensitive to the average points-to set size and is instead governed by the amount of graph rewriting performed. This highlights that GPU-based flow-sensitive pointer analysis follows a distinct cost model in which rewrite volume, rather than points-to size, is the primary driver of performance. However, neither the average points-to set size nor the eventual rewrite volume can be known prior to running the analysis. Since these quantities determine runtime but are unavailable beforehand, we employ a GNN to predict analysis cost and guide execution decisions.

Table 3. Statistics for each benchmark program, including the number of pointers, the average number of χ functions at a `store` instruction, the average number of μ functions at a `load` instruction, the average size of points-to sets, and the total number of added edges. Programs are listed in an increasing order of the number of LLVM IR instructions.

	Program	**# Pointers**	**Avg. # χ**	**Avg. # μ**	**Avg. `pts()` size**	**# Edges**
Small (<100 KLOC)	LBM	8,026	1	1	0.86	4,297
	MCF	7,884	1	1.03	1.82	13,415
	DEEPSJENG	30,204	1.09	1.01	1.26	21,163
	DPKG	56,713	1.08	1.02	8.19	145,150
	NAB	58,743	1.02	1.06	1.17	78,317
	DU	54,979	1.15	1.13	25.88	82,326
	XZ	58,353	1.25	1.09	34.67	72,165
	LEELA	79,804	1	1.56	10.51	214,297
	NANO	80,987	1.21	1.22	64.64	159,194
	PSQL	87,872	1.06	1.08	9.19	201,269
	JANET	93,110	1.41	1.15	16.54	214,729
Medium (100–275 KLOC)	I3	113,310	1.39	1.32	14.39	263,379
	BAKE	63,998	1.08	1.1	22.4	151,347
	TMUX	127,027	1.85	2.03	124.76	255,923
	ASTYLE	149,367	4.36	3.26	337.63	552,937
	X264	197,726	1.76	1.21	126.46	302,213
	NINJA	85,602	1.08	1.31	15.88	66,649
	MRUBY	212,826	1.08	1.12	17.73	361,838
	NAMD	261,833	1	1	0.85	249,029
Large (≥275 KLOC)	POVRAY	266,017	1.14	1.14	193.82	535,735
	BASH	280,398	1.17	1.17	103.65	591,859
	LYNX	331,328	1.12	1.29	253.95	789,014
	IMAGICK	551,008	1.03	1.05	1.93	1,186,323
	OMNETPP	712,651	1.02	1.52	52.36	1,595,126
	PERLBENCH	782,375	1.46	1.69	836.63	2,160,470
	XALANCBMK	1,272,898	1.04	1.32	98.36	3,931,560

Program size alone does not determine pointer analysis cost. Instead, structural properties drive runtime, with VSFS influenced by points-to set growth and GPA influenced by the volume of rewrite activity.

5.7 GNN-based Workload Prediction for Large Programs (RQ4)

Workload balance is crucial for GPU effectiveness. We compare a naive version of GPA (i.e., no prediction) to a GNN-guided version of GPA and PFS on 7 large programs. Figure 9 shows that, using GPANaive as baseline, GNN-guided GPA improves performance on 6/7 benchmarks and matches on IMAGICK. The small average points-to set size (1.93) in IMAGICK implies limited workload skew. Therefore, GNN prediction brings little benefit. We see the largest gain on PERLBENCH, which has the largest average points-to set size (836.63), and thus heavy skew. For this setting, grouping nodes by predicted cost reduces intra-warp idling. On average, GNN-guided GPA achieves an average speedup of 1.2× compared to its naive counterpart (min: 0.99×, max: 1.41×, geomean: 1.19×).

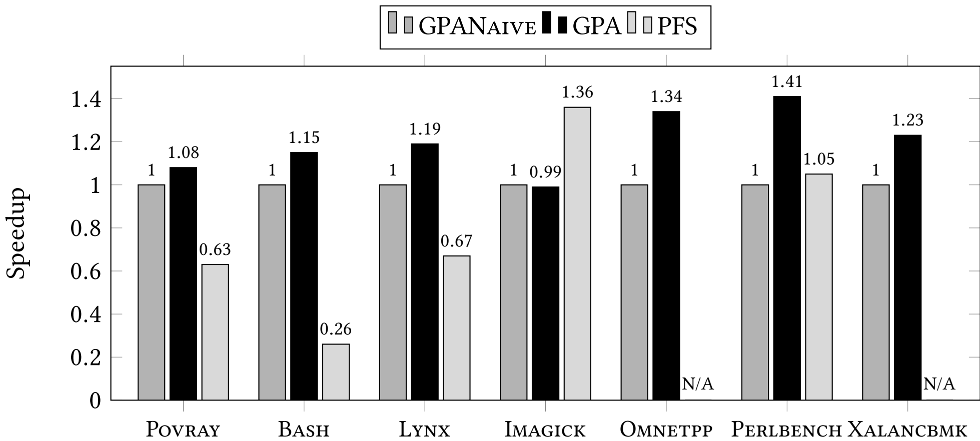

Fig. 9. Speedup of GPA and PFS compared to GPANaive on our large benchmark programs. We use N/A to denote results that we could not record due to timeouts.

> Grouping nodes by GNN-predicted workload reduces warp divergence and improves throughput, with an average speedup of 1.2× compared to no prediction on large programs.

5.8 Summary

We have observed that each approach tends to dominate in a different range of program sizes. For small programs, VSFS handles them best, because neither GPU or multi-threaded CPU parallelism pays off. For medium programs, Soufflé stands out due to its relatively low overhead and moderate parallelism capabilities. Finally, GPA scales best to large programs, where GPUs excel at massive workloads that CPUs cannot handle efficiently.

6 Related Work

Flow-sensitive pointer analysis has been extensively studies for the past decade. While great improvement has been achieved on the algorithm side, more recent research has been focusing on leveraging massive parallelism that modern hardware architectures provide.

VSFS [2] is an enhancement of SFS [12], which remained the prevailing state-of-the-art flow-sensitive pointer analysis for years. VSFS removes redundant propagation of unchanged points-to sets along the DUG of an input program. The main observation is that most of the points-to sets are not modified when being propagated from one program location to another. Therefore, keeping a separate copy for these points-to sets at different program locations is unnecessary, and degrades the algorithm performance. To remove this unnecessary propagation and storage of points-to sets, VSFS assigns version numbers to points-to sets such that the same points-to sets across the program share the same version number. VSFS achieves an average speed-up rate of 5.31× against SFS across 15 benchmark programs. In our evaluation, we use VSFS as our baseline for a single-threaded CPU-based analysis. While VSFS is more suitable for small programs, GPA outperforms it on medium and large programs.

Méndez-Lojo et al. [18] implements the first flow-insensitive pointer analysis that runs on a GPU. The authors translate the flow-insensitive pointer analysis into a fixed-point computation over edges of a points-to graph. For each graph rewriting rule, the analysis starts a kernel and assigns

each node in the points-to graph to a thread. The analysis achieves an average speedup of 7× against a sequential CPU graph-based implementation. Su et al. [27] later improve upon that idea by adding an imbalance-aware workload partitioning scheme. Their implementation first assigns each node in the points-to graph to a thread until it detects a workload imbalance. The algorithm then switches to a task-pool-based model, which is managed by a worklist. Their implementation improves runtime performance by an average of 46% compared to the analysis by Méndez-Lojo et al. [18]. However, the pointer analysis is flow-insensitive, which exhibits less complex graph rewriting rules and requires no metadata querying. On the other hand, GPA is a flow-sensitive pointer analysis that runs on a GPU, requiring more careful workload management to parallelize complex graph rewriting rules.

Zuo et al. [39] model context-sensitive but flow-insensitive pointer analysis as a CFL reachability problem. Their system represents pointer dereference, assignment, and allocation using three edge types and determines aliasing by checking whether paths between nodes satisfy a given grammar. To efficiently encode graph edges, they use sparse bit vectors and provide both CPU and GPU backends, reporting speedups ranging from 3× to 100×. Their formulation adopts the graph representation of [37], which is originally proposed for demand-driven pointer analysis. However, their formulation lacks flow-sensitivity and strong updates, which limits its applicability. In contrast, GPA is flow-sensitive and supports strong updates, and we have designed it specifically for whole-program analysis on GPUs.

There has also been several attempts to implement flow-sensitive pointer analysis on multi-core CPUs. For example, Nagaraj and Govindarajan [21] present the first graph-rewriting-based flow-sensitive pointer analysis on multi-core CPUs. Their algorithm starts by running an Andersen-style pointer analysis. It then builds the full SSA from that result. Then, it translates the full SSA into a points-to graph for each instruction and performs a fixed-point computation on the graph. In their implementation, the authors use Intel Threading Building Blocks [3] to manage the parallel work. Their evaluation shows that their approach achieves an average speedup of 3.2× across 10 benchmark programs. In addition, Zhao et al. [36] introduce pointer SSA form that captures the def-use relationships of heap variables. The authors implement their algorithm in LLVM as an analysis pass that uses Habanero-C [11] to manage task parallelism. Across 12 benchmark programs, their implementation achieves an average speedup of 4.5×. Unlike their algorithm, GPA runs on a GPU to fully exploit the benefit of parallelism offered by modern hardware architectures.

More recently, Zuo et al. [40] design a graph-based engine for flow-sensitive alias analysis by modeling it as a reachability problem, exploiting parallelism through graph partitioning. Sun et al. [30] extend this approach to a distributed setting, enabling execution on cloud infrastructures. However, both systems run on CPUs and rely on dynamically deleting alias relationships during analysis, which makes their formulations difficult to extend to GPU architectures. In contrast, GPA implements flow-sensitive analysis with strong updates on GPUs using a graph-rewriting model that never deletes points-to relations and employs a GNN-based scheduler to balance workload across GPU threads.

Wang et al. [35] address pointer analysis in an incremental setting, reducing analysis time by leveraging code commit histories after an initial analysis. Unlike their approach, GPA performs whole-program pointer analysis on GPUs. Furthermore, Wang et al. [35] focus on flow-insensitive incremental analysis, whereas GPA targets flow-sensitive pointer analysis with strong updates. Finally, Tekle and Liu [33] demonstrate that a wide range of pointer analyses can be expressed declaratively in Datalog while preserving precise complexity guarantees. Their work shows that analyses such as Andersen pointer analysis and context-sensitive variants can be encoded using carefully designed Datalog formulations, motivating subsequent systems such as Soufflé. However, their work does not address flow-sensitive pointer analysis, which is the primary focus of our work.

7 Conclusion

Flow-sensitive pointer analysis underpins a wide range of data-flow analyses. However, traditional CPU-based implementations struggle to scale due to prohibitive runtimes on large programs. In this paper, we present GPA, the first GPU-accelerated flow-sensitive pointer analysis for C/C++ programs. GPA models the analysis as a graph rewriting problem, with GPU-friendly data structures and execution strategies. To address workload imbalance across warps, we have trained a GNN to predict the per-variable workload and assign variables with similar costs to the same warp.

We have evaluated GPA on 26 benchmark programs that are commonly used in pointer analysis research. On large programs, GPA delivers its strongest results, achieving an average speedup of 3.8× (up to 14×) compared to state-of-the-art CPU-based analyses. Incorporating GNN-based workload prediction further improves scalability, boosting performance by an additional 20% on these workloads. For medium programs, GPA provides a moderate average speedup of 1.1×. However, for small programs, the overhead of GPU execution outweighs the benefits, making traditional CPU-based analyses more efficient than GPA.

Overall, GPA makes flow-sensitive pointer analysis practical at scales where existing approaches remain infeasible, enabling precise program analyses on large codebases.

References

[1] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 84–104. doi:10.1007/978-3-662-53413-7_5

[2] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 222–235. doi:10.1109/CGO51591.2021.9370334

[3] Intel Threading Building Block. 2025. https://github.com/uxlfoundation/oneTBB

[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. doi:10.1145/1640089.1640108

[5] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar (Eds.). ACM, 41–42. doi:10.1145/3185768.3185771

[6] Clang. 2025. https://github.com/llvm/llvm-project/releases/tag/llvmorg-16.0.1

[7] Chris Cummins, Hugh Leather, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Michael F. P. O'Boyle. 2020. Deep Data Flow Analysis. *CoRR* abs/2012.01470 (2020). arXiv:2012.01470 https://arxiv.org/abs/2012.01470

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. doi:10.1145/115372.115320

[9] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[10] Bernd Fischer, Giulio Garbi, Salvatore La Torre, Gennaro Parlato, and Peter Schrammel. 2024. Static Data Race Detection via Lazy Sequentialization. In *Networked Systems - 12th International Conference, NETYS 2024, Rabat, Morocco, May 29-31, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14783)*. Springer, 124–141. doi:10.1007/978-3-031-67321-4_8

[11] Habanero-C. 2025. https://github.com/habanero-rice/hclib

[12] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011.* IEEE Computer Society, 289–298. doi:10.1109/CGO.2011.5764696

[13] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part*

II (Lecture Notes in Computer Science, Vol. 9780), Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. doi:10.1007/978-3-319-41540-6_23

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[15] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. doi:10.1109/CGO.2004.1281665

[16] LLVM. 2025. https://reviews.llvm.org/D139703

[17] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. 2018. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM Books, Vol. 20. ACM / Morgan & Claypool, 3–100. doi:10.1145/3191315.3191317

[18] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan (Eds.). ACM, 107–116. doi:10.1145/2145816.2145831

[19] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 428–443. doi:10.1145/1869459.1869495

[20] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. doi:10.1609/AAAI.V30I1.10139

[21] Vaivaswatha Nagaraj and R. Govindarajan. 2013. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, Christian Fensch, Michael F. P. O'Boyle, André Seznec, and François Bodin (Eds.). IEEE Computer Society, 19–28. doi:10.1109/PACT.2013.6618800

[22] Nvidia. 2025. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[23] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Version 13.0.

[24] Karl Pearson and Francis Galton. 1895. VII. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London* 58, 347-352 (1895), 240–242. arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rspl.1895.0041 doi:10.1098/rspl.1895.0041

[25] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1467–1471. doi:10.1145/186025.186041

[26] Nadesh Ramanathan, George A. Constantinides, and John Wickerson. 2020. Precise Pointer Analysis in High-Level Synthesis. In *30th International Conference on Field-Programmable Logic and Applications, FPL 2020, Gothenburg, Sweden, August 31 - September 4, 2020*, Nele Mentens, Leonel Sousa, Pedro Trancoso, Miquel Pericàs, and Ioannis Sourdis (Eds.). IEEE, 220–224. doi:10.1109/FPL50879.2020.00044

[27] Yu Su, Ding Ye, Jingling Xue, and Xiangke Liao. 2016. An Efficient GPU Implementation of Inclusion-Based Pointer Analysis. *IEEE Trans. Parallel Distributed Syst.* 27, 2 (2016), 353–366. doi:10.1109/TPDS.2015.2397933

[28] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. doi:10.1145/2892208.2892235

[29] Yulei Sui, Hua Yan, Zheng Zheng, Yunpeng Zhang, and Jingling Xue. 2018. Parallel construction of interprocedural memory SSA form. *J. Syst. Softw.* 146 (2018), 186–195. doi:10.1016/J.JSS.2018.09.038

[30] Zewen Sun, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Zhiqiang Zuo, Zhaokang Wang, Yue Li, Xuandong Li, Qingda Lu, Wenwen Peng, and Shengjian Guo. 2023. BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1431–1443. doi:10.1145/3611643.3616348

[31] SVF. 2025. https://github.com/SVF-tools/Test-Suite

[32] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485524

[33] K. Tuncay Tekle and Yanhong A. Liu. 2016. Precise complexity guarantees for pointer analysis via Datalog with extensions. *Theory Pract. Log. Program.* 16, 5-6 (2016), 916–932. doi:10.1017/S1471068416000405

[34] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 697–709. doi:10.1145/3503222.3507764

[35] Jiayi Wang, Yu Wang, Ke Wang, and Linzhang Wang. 2025. SILVA: A Scalable Incremental Layered Sparse Value-Flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 34, 8 (2025), 229:1–229:40. doi:10.1145/3725214

[36] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, Christophe Dubach and Jingling Xue (Eds.). ACM, 59–70. doi:10.1145/3178372.3179517

[37] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 197–208. doi:10.1145/1328438.1328464

[38] Mohamed Tarek Ibn Ziad, Sana Damani, Mark Stephenson, Stephen W. Keckler, and Aamer Jaleel. 2025. GPUArmor: A Hardware-Software Co-design for Efficient and Scalable Memory Safety on GPUs. *CoRR* abs/2502.17780 (2025). arXiv:2502.17780 doi:10.48550/ARXIV.2502.17780

[39] Zhiqiang Zuo, Kai Wang, Aftab Hussain, Ardalan Amiri Sani, Yiyu Zhang, Shenming Lu, Wensheng Dou, Linzhang Wang, Xuandong Li, Chenxi Wang, and Guoqing Harry Xu. 2020. Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan. *ACM Trans. Comput. Syst.* 38, 1-2 (2020), 4:1–4:39. doi:10.1145/3466820

[40] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: an evolving graph system for flow- and context-sensitive analyses of million lines of C code. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 914–929. doi:10.1145/3453483.3454085

Received 2025-09-11; accepted 2026-03-25