GPU Accelerated Dataflow Analysis

Akram Aziz
American University in Cairo
Cairo, Egypt
akramaziz@aucegypt.edu

Andrew Zaki
American University in Cairo
Cairo, Egypt
andrewhany@aucegypt.edu

Basant Abdelaal
American University in Cairo
Cairo, Egypt
basantelhussein@aucegypt.edu

Mohammed Zaieda
American University in Cairo
Cairo, Egypt
mzaieda@aucegypt.edu

Seifeldin Ashraf
American University in Cairo
Cairo, Egypt
seifeldinashraf11@aucegypt.edu

Youssef Hussein
American University in Cairo
Cairo, Egypt
youssefhussien@aucegypt.edu

Cherif Salama
American University in Cairo
Cairo, Egypt
cherif.salama@aucegypt.edu

Karim Ali
University of Alberta
Edmonton, Canada
karim.ali@ualberta.ca

ABSTRACT

Dataflow analysis is a critical technique for detecting software bugs during development. However, existing tools often suffer from scalability and speed issues. In this work, we explore the use of GPUs to accelerate dataflow analysis by targeting dataflow analysis problems. We propose a GPU-compatible implementation based on the matrix representation approach. By representing instructions as matrices, we achieve a significant speedup of up to 250X compared to CPU implementations. Our research aims to accelerate dataflow analysis using GPUs, providing a more efficient solution for real-time development environments. This approach overcomes the speed limitations of existing dataflow analysis tools, enabling faster bug detection and more precise analysis. By leveraging the parallel processing capabilities of GPUs, we handle large-scale programs and complex dataflow effectively. Our work bridges the gap between the increasing complexity of software systems and the need for scalable and efficient bug detection mechanisms. By accelerating dataflow analysis using GPUs, we enhance the precision and speed of static analysis, enabling faster bug detection and reporting in software development.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Graphics processors**; • **Theory of computation** → *Design and analysis of algorithms.*

KEYWORDS

Dataflow analysis, IFDS, GPU, LLVM, Matrix Multiplication.

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Akram Aziz, Andrew Zaki, Basant Abdelaal, Mohammed Zaieda, Seifeldin Ashraf, Youssef Hussein, Cherif Salama, and Karim Ali. 2023. GPU Accelerated Dataflow Analysis. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Heathrow's Terminal 5 opened in 2008 with a new luggage handling system that performed admirably in tests but failed badly in practice. Massive interruptions resulted from this, including broken luggage belts and the loss or misdelivery of luggage. As a result, over 500 flights were canceled, and about 42,000 baggage went missing, costing the airport more than £16 million [1]. The Ariane 5, a space exploration mission carried by the European Space Agency (ESA), had a similar line of action as Heathrow's terminal 5. As a result of the engineers recycling incompatible code from Ariane 4 and a conversion issue from 64-bit to 16-bit data the rocket engines failed just 36 seconds after their first flight [6]. The failure cost the ESA $370 million. Moreover, many nuclear reactors depend on software programs for the safety of the workers. A slight mistake in the software upgrade process might cost many people's lives [17]. In general, all of these issues and vulnerabilities can be detected, and even better, fixed during development through static analysis during development.

Static analysis is the process of scanning source code without executing it. Hence, providing developers with an understanding of their code base and helping to ensure that the code base is compliant, safe, and secure. Static program analysis can, with proper approximations, provide possible executions of the programs and provide guarantees about their properties. The challenge is providing high precision and efficient analyses.

As a result, it is critical to decide if it is more important to have a sound and precise static analysis than having an efficient analysis process. In a study carried out by Google and North Carolina state university [10], practitioners expressed their desire to have a tool that detects bugs early on during development and prevents the occurrence of false positives. Moreover, they wanted a real-time

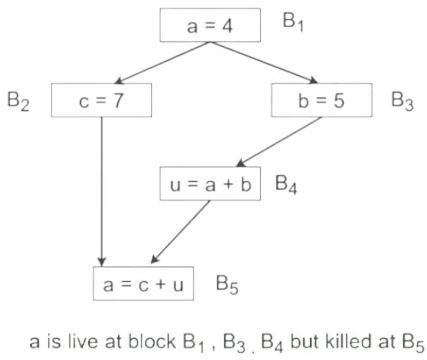

Figure 1: Live Variable Analysis Example, replicated from [7]. The figure illustrates a control flow graph with nodes and edges representing program statements and their dependencies.

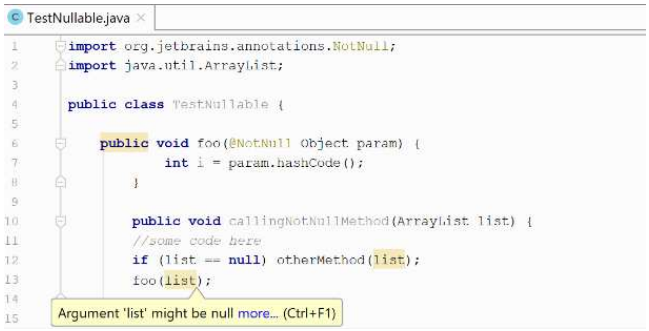

Figure 2: Nullness Analysis Example, replicated from [8].The example demonstrates the use of Nullness Analysis to identify the nullness property of variables at each program point.

analysis before compilation in which any warnings would be referenced while coding. Some were frustrated with the performance mainly when it ran on a large code base. Additionally, in another survey carried out by Microsoft [5], it was clear that more than 200 practitioners were frustrated with many false positives. Even more concerning, practitioners felt the tool would be unusable if the tool were slow. Therefore having a static analysis framework that is sound and precise with an acceptable speed is a major characteristic of any desired static analysis framework.

The matrix approach in dataflow analysis provides a systematic and efficient method for capturing and summarizing information flows within a program. It involves representing instructions as matrices, where each matrix captures the taint state of the instruction and its influence on subsequent instructions.

In this approach, the taint state of instruction is represented as a column vector, where each element in the vector corresponds to a specific taint bit. The matrix itself represents the propagation rule of the instruction, specifying how the taint bits of the input influence the taint bits of the output.

By applying matrix multiplication operations, the matrix approach enables the summarization of information flows across multiple instructions. The resulting matrix represents a summary of the dataflow that has occurred in the program, capturing the dependencies and influences between different taint bits.

2 BACKGROUND AND RELATED WORK

2.1 What is IFDS?

Interprocedural Finite Distributive Subset (IFDS) is a widely used framework for dataflow analysis in software engineering. IFDS problems involve flow functions over a finite domain D that must be distributive over the merge operator $\cup$, meaning that $f(a) \cup f(b) = f(a \cup b)$ for any flow function f and any $a, b \in D$ [3]. IFDS enables the static analysis of programs in a template-driven manner, where the static analysis engineer defines flow functions for an analysis problem without needing to worry about solving the problem. IFDS is particularly useful for relating and plotting the relationships between different function calls with a distributive characteristic.

IFDS supports different types of static analyses, including Live Variable and Nullness analyses. Live Variable analysis determines whether a variable is still used at a certain point in the program, as shown in Figure 1, where the variable a is alive at blocks $B1$, $B3$, and $B4$, but not at $B5$ where it is assigned a new value. Nullness analysis, on the other hand, answers whether a pointer will become null in any execution path of the program, as demonstrated in Figure 2, where a static analyzer displays a message indicating that the *list* pointer may be a null pointer.

The goal of this project is to investigate GPU-acceleration of dataflow analyses within the IFDS framework. Our literature review is organized into three subsections. In Section 2.2, we examine current IFDS implementations, while Section 2.3 focuses on algorithmic acceleration techniques for static analysis. Section 2.4 discusses hardware acceleration methods, including the use of GPUs and their limitations. As such, our literature review covers both the IFDS framework and its previous implementations.

2.2 Current IFDS Implementations

In his work, Bodden [3] described an implementation of a generic IFDS/IDE solver on top of Soot. He compared his implementation with an IFDS implementation in the Watson Libraries for Analysis (WALA), both from a user's perspective and in terms of implementation. Bodden's implementation was directed towards extensibility and ease of use, with efficiency as a secondary goal. He also provided possible extensions to his IFDS/IDE implementation, including support for branched analyses, exceptional control flow, persisting IDE-based summary information mechanism, return flow functions, and additional extensions. While Bodden's work was prominent and directly related to the intended work, it lacked some key features such as acceleration of data flow analyses and GPU acceleration. Nonetheless, his work served as a viable starting point for this project, as it provided another IFDS/IDE implementation focused on goals other than acceleration.

One major limitation of Bodden's literature [3] was the lack of data flow analysis acceleration. Specifically, Bodden did not focus on using GPUs or any means of acceleration to speed up his IFDS/IDE implementation. This limitation highlights the need for further research and exploration into acceleration techniques for data flow analysis, which is the focus of this project. Despite this limitation,

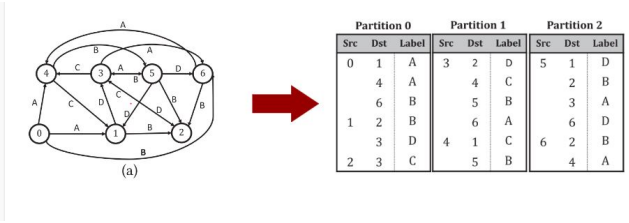

Figure 3: A vertex interval table defining the relations between program graph's partitions, adapted from [19].

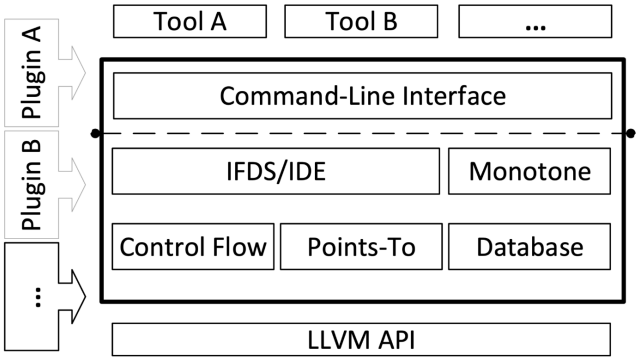

Figure 4: PhASAR's Layered Architecture, adapted from [14].

Bodden's work offered valuable insights into the implementation of IFDS/IDE solvers, and his approach to extensibility and ease of use will be considered in the development of our accelerated IFDS/IDE solver.

PhASAR is another static analysis framework optimized for C/C++ that was developed by a research team at Paderborn University [14]. The framework allows for solving arbitrary (decidable) data-flow problems on the LLVM intermediate representation (IR). PhASAR offers various data-flow solver implementations as well as all required helper analyses. It provides, among others, several algorithms for call-graph construction, computation of points-to information, class hierarchy, and data-flow information. PhASAR implements IFDS as a specialization of IDE using a binary lattice only using a top and a bottom element much alike the Hero's JAVA implementation [13]. In subsection 3.1, we explore our choice of PhASAR as a potential baseline implementation for GPU acceleration due to its compatibility with C/C++ and support for IFDS problems.

In addition to existing IFDS implementations such as Bodden's work and PhASAR, there are other static dataflow analysis frameworks that use LLVM as their foundation. The LLVM Dataflow Analysis framework, developed by Dr. Nick Sumner from Simon Fraser University [16], is one of these frameworks and includes three types of static analyses: Constant Propagation, File Policy, and Future Functions. While the implementation [16] is not scalable, the framework demonstrates the potential of using LLVM for static analysis on C/C++ programs due to its support for various data structures and methods to create call and control flow graphs. This framework could offer insights into the implementation of IFDS/IDE solvers and will be considered in the development of our accelerated static analysis solver. The limitations of existing IFDS implementations, including PhASAR [14] and Bodden's work [3], emphasize the need for further research and exploration into acceleration techniques for data flow analysis.

2.3 Algorithmic Acceleration

Algorithmic optimizations have been proposed to improve the efficiency of dataflow analysis in static analysis tools. In this subsection, we introduce two relevant papers that cover most of the implementations in research. One approach proposed by Møller and Shwartzbach [4] is to use numerical methods to accelerate the convergence of numerical sequences while maintaining precision and guaranteeing termination in a finite number of iterations. Their method is designed to remain close to the Kleene iteration, making it easy to implement in existing static analyzers. They describe a general framework and its application to two numerical abstract domains, with experimental results showing a significant reduction in the number of iterations and computation time needed to compute the fixpoint compared to the Kleene iteration.

Another approach to accelerating dataflow analysis algorithmically is to implement predicate propagation, as shown by BlaB and Philippsen [2]. Their framework propagates predicates along the Control Flow Graph (CFG) until a fixpoint is reached, allowing corrupted predicates to happen to avoid costly synchronization and fixing the problem later. This approach has been used to detect and fix common data races in a Single Instruction Multiple Data (SIMD) fashion. By detecting and fixing data races in a parallel fashion, this approach can significantly reduce the amount of time needed for dataflow analysis, making it particularly useful for large-scale software projects. Together, these algorithmic optimizations offer promising solutions for improving the efficiency and scalability of dataflow analysis in static analysis tools.

However, a current limitation of the algorithmic acceleration solutions is that they focus on accelerating a limited number of analyses, and the currently adapted optimizations are not generic and are specific to certain types of analyses. The lack of a generic framework covering many types of analyses while maintaining high performance using algorithmic optimizations is a checkpoint that this literature highlights, and it is where our research will begin. Our aim is to develop a generic accelerated IFDS/IDE solver that incorporates algorithmic optimizations for dataflow analysis, including the techniques introduced in this subsection, and can be applied to a broad range of analyses without sacrificing performance or precision.

2.4 GPU Acceleration

In recent years, there has been growing interest in using GPUs to accelerate static analysis. Three related works in this area are the work of Yu et al. [18], Zhang et al. [19], and Ji et al. [9].

The work by Yu et al. proposes a GPU acceleration approach for static dataflow analysis that optimizes the worklist algorithm for Inter-procedural Data-Flow Graph (IDFG) construction [18]. The worklist algorithm is used to construct IDFGs in static analysis by propagating datafacts between nodes in a graph until a fixpoint

```
1 int id(int i) { return i; }
2
3 int inc(int i) { return ++i; }
4
5 int add(int i, int j) { return i + j; }
6
7 int main() {
8   int a = 0;
9   int b = 1;
10  a = inc(a);
11  b = id(b);
12  int c = add(a, b);
13  return 0;
14 }
```

```
1 Call Graph:
2 __psrCRuntimeGlobalCtorsModel --> main
  __psrCRuntimeGlobalDtorsModel
3 main --> _Z3inci _Z2idi _Z3addii
4 __psrCRuntimeGlobalDtorsModel -->
5 _Z3inci -->
6 _Z2idi -->
7 _Z3addii -->
```

Figure 5: On the right, a sample program analysed by PhaSAR. On the left, PhASAR's call graph construction from the program.

is reached. Yu et al. applied a straightforward parallelization approach by splitting the graph into subgraphs and processing them using a GPU kernel, but this only resulted in a 2X speedup due to performance bottlenecks like dynamic memory allocation, load imbalance, and irregular memory accesses. To address this, Yu et al. [18] proposed Android-specific optimizations, resulting in a 128X speedup.

Another related work was Graspan-G, which is a GPU-based implementation of the IFDS algorithm that reformulates the problem as a Context Free Language (CFL) reachability problem. It was proposed by Zhang et al. [19], who sought to improve the performance of IFDS by leveraging the parallel processing power of GPUs. Graspan-G was designed to handle large-scale program analysis tasks, such as call graph construction and data flow analysis, more efficiently than traditional CPU-based methods. Graspan-G [19] requires two inputs: a Context Free Grammar (CFG) and a program graph annotated with symbols from the grammar. The annotated program graph is partitioned to fit into memory, and a vertex interval table is created to define the relation between partitions, as shown in Figure 3. Graspan-G uses a Destination Distribution Map (DDM) to choose two partitions for processing based on the highest outgoing neighbor count.

The most recent work to our knowledge in this area is the work of Ji et al. [9]. Ji et al. present FlowMatrix, a matrix-based approach for Dynamic Information Flow Tracking (DIFT) that addresses the challenge of performance and scalability. DIFT involves tracking dynamic data flows in programs to dynamically analyze the flow of data among program states. However, the large number of states in a program can make the number of data flows massively large, making it challenging to efficiently perform dynamic data analysis queries. FlowMatrix is a novel matrix-based representation for DIFT operations that enables efficient and versatile DIFT analysis using GPUs. The authors show that DIFT under dependency-based information flow rules can be cast as linear transformations over taint states, which enables concise representation of DIFT operations using matrices. Overall, FlowMatrix provides an efficient and practical approach for improving the scalability and performance of DIFT techniques.

While these works have shown promising results, there is still a need for further research to develop generic accelerated worklist algorithms that can be applied to a broad range of analyses without sacrificing performance or precision in contrast to the work of Yu et al. [18]. Our work will build on the promises made by the literature in developing a generic accelerated static analysis implementation that does not rely on domain-specific optimizations.

$$
\begin{aligned}
&\textit{Object flow}: && OF ::= M\ VF \\
&\textit{Value flow}: && VF ::= (A\ MA?)^* \\
&\textit{Memory alias}: && MA ::= \overline{D}\ VA\ D \\
&\textit{Value alias}: && VA ::= \overline{VF}\ MA?\ VF
\end{aligned}
$$

Figure 6: An example of a Context Free Grammar of a program fed as an input to Graspan-G, adapted from [19].

While FlowMatrix [9] employs a matrix-based representation for DIFT operations, our goal is to adapt this approach for static data flow analysis. Specifically, we aim to use matrix-based representation for static analysis operations, which differs from FlowMatrix in its focus and target application.

3 PARALLELIZATION ATTEMPTS OF CURRENT STATIC ANALYSIS IMPLEMENTATIONS

To avoid reinventing the wheel, we began our research by exploring existing static analysis frameworks to determine their feasibility for GPU acceleration. Our goal was to parallelize one of these baselines and compare its performance on the GPU to its current CPU performance. There was three main metrics we used to asses the different baselines we explored:

(1) To have a simple architecture.
(2) To be able to reproduce the framework results.
(3) To have parallelization potential.

We explored four main baselines: PhASAR [14], Graspan-G [19], LLVM Data-flow Analysis framework [16], and Flow-Matrix [9]. In this section, we will discuss each of these baselines in detail, including their design and implementation, and explain why they were ultimately not parallizable in our scope. Through this exploration, we gained valuable insights into the strengths and limitations of existing approaches to static analysis and how they can inform us when developing our own implementation.

3.1 PhASAR

Since we will be using CUDA to parallelize the data-flow analysis, we began by exploring existing static analysis frameworks to determine their feasibility for GPU acceleration. We chose PhASAR as a baseline implementation because it is written in C/C++ and therefore potentially GPU-compatible. Additionally, PhASAR is one of the few C/C++ frameworks that supports IFDS problems, which is desirable for our purposes.

One of the advantages of PhASAR is that it allows the user to specify which approach to use (either IFDS or monotone) and which target analysis to run from the command line interface. The output of these analyses can then be processed and presented to the user directly. PhASAR can also be extended with custom analyses provided as compiled plugins of a shared object file, which enables analysis of programs written in different languages due to the IR

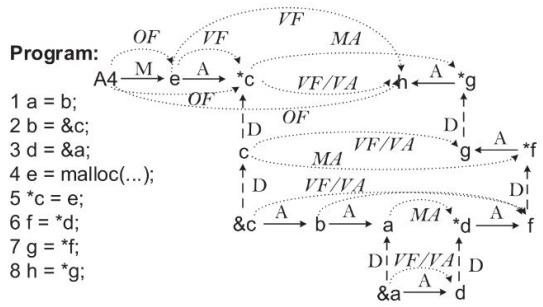

Figure 7: An example of Graspan-G adding a new transitive edge with a label equal to the left-hand side of the grammar, adapted from [19].

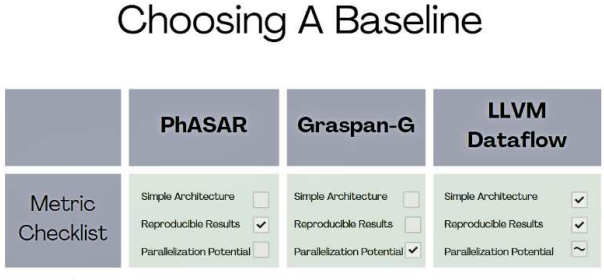

Figure 8: A summary of the evaluation results of the baselines against our metrics: architecture simplicity, results reproducibility, and parallelization potential."

nature of the program. PhASAR can also be used as a library in other tools.

In Figure 5 on the left, we have a simple program that we analyzed using PhASAR. This program has three function calls, and the call graph constructed by the framework, shown in Figure 5 on the right, illustrates the relationships between the main function in the program and its function calls. However, after spending time exploring PhASAR and attempting to parallelize it on the GPU, we found that its layered architecture (as shown in Figure 4) made it difficult to make it GPU-compatible. Specifically, IFDS in PhASAR is dependent on out-of-scope data-flow analyses in the code base, which makes it difficult to use as a baseline implementation for our purposes. Instead, we decided to use PhASAR as a way to write our own code base. Therefore, we concluded that we needed to find a framework that is first GPU-compatible.

3.2 Graspan-G

Another baseline that we explored throughout the project was Graspan-G. It is a GPU-based implementation of the IFDS algorithm that reformulates the problem as a CFL reachability problem. It was proposed by Zhang et al. [19] as a way to improve the performance of IFDS by leveraging the parallel processing power of GPUs. We spent time exploring this baseline to see its effectiveness and if it is worth it to convert out problem from being based on matrices to a CFL reachibilty problem.

Graspan-G requires two inputs: a CFG and a program graph annotated with symbols from the grammar. as shown in Figure 6 and Figure 7, respectively. The static engineer defines the relation between every two nodes and associates each edge with a symbol from the grammar defined. Graspan-G has three phases: preprocessing, processing, and post-processing. In the pre-processing phase, Graspan-G converts the annotated program graph into different partitions to fit into memory. A vertex interval table is created to define the relation between partitions, and a DDM is introduced to choose two partitions for processing based on the highest outgoing neighbor count, as shown in Figure 3. During the processing phase, Graspan-G searches for the right-hand side of the grammar in every pair of partitions and adds a new transitive edge with a label equal to the left-hand side of the grammar, as shown in Figure 7. The algorithm uses a bit vector data structure optimized for GPU to search through all edges to find any identical grammar to the defined one, enabling parallel processing of all edge pairs. Finally, the post-processing phase should result in a new program graph with transitive edges, and there are APIs to traverse the output graph to get useful insights about the analysis.

However, after running Graspan-G, there was no saved output, and the APIs could not be accessed. Additionally, we faced several challenges with Graspan-G, including the need for manually creating the required inputs and the significant amount of memory required for the vertex interval table. As a result of these challenges, we decided to shift gears towards other baselines with more stable, understandable, and simpler code bases, such as LLVM Dataflow Analysis framework [16]and our own custom GPU-based implementation.

3.3 LLVM Dataflow Analysis Framework

Following our difficulties with PhaSAR and Graspan-G, we decided to shift our focus to a simpler static analysis framework. We explored the LLVM Dataflow Analysis framework [16], developed by Nick Sumner from Simon Fraser University. This framework can apply static analysis over C and C++ programs and comes with three static analyses supported.

Constant Propagation is the first static analysis supported in this framework. It identifies exact constant values that can be defined at compile time. Additionally, the framework prints all computable constant arguments of the module's function calls. The second static analysis is File Policy, which identifies straightforward mistakes in the usage of fread, fwrite, and fclose when invoked on previously closed files. Finally, Future Functions is another static analysis supported in the LLVM Dataflow Analysis Framework. It determines the functions that might be called in the future at all call locations in a program. The future functions analysis offered by the framework uses backward dataflow analysis. The LLVM Dataflow Analysis framework [16] employs well-known techniques in dataflow analysis using the worklist algorithm. The algorithm goes over the input code, which is c code, which is then converted be in an LLVM bytecode and forms blocks. It then adds the blocks in the worklist

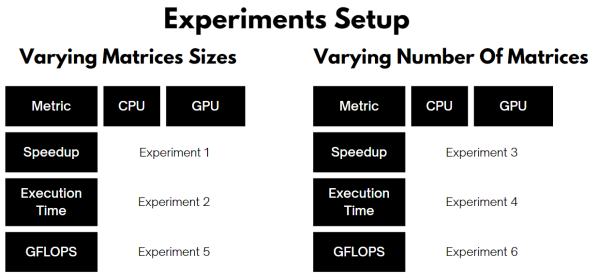

Figure 9: Figure shows summary of the experiments done to test validity of the GPU architecture vs the CPU on our Matrix-based approach.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   58C    P8    10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Figure 10: Figure shows specifications of the GPU used in the experiments.

to loop over them and perform the analysis (either constant propagation or file policy). Finally, it merges the results of all the blocks as if they are lattices. Sumner's implementation [16] includes summaries and uses the SummaryKey to indicate if a method has been visited or not per context, allowing for result merging if necessary. However, the final methods are revisited, and summaries are recalculated each time. Therefore, a summary for each function should be implemented using a better approach, especially for composite functions.

Despite the simplicity of the LLVM Dataflow Analysis framework's architecture [16], and its reliable results, our attempts to parallelize the framework were not promising. One of the major blockers was the heavy reliance on LLVM, which supports many dynamic data structures and methods to create call and control flow graphs. Dynamic data structures are not GPU-friendly because they require frequent memory allocations and deallocations, which can cause memory fragmentation and overhead on the GPU [11]. GPUs excel at executing large numbers of parallel computations simultaneously, but they are optimized for regular, predictable access patterns to memory. Dynamic data structures, on the other hand, have unpredictable access patterns and require complex memory management, which can limit the efficiency of parallelization on the GPU [11]. As a result, parallelizing dynamic data structures on the GPU would require either changing the framework's dependency on LLVM or changing LLVM's dynamic data structures. Unfortunately, both approaches proved unfeasible.

We then shifted to try the same logic of paralyzing LLVM Dataflow Analysis on a GPU but on a CPU. We created a pool of threads and assigned each thread to be responsible for analyzing one function at a time after trying to implement locking and unlocking techniques using Mutex. We have encountered run-time errors. This is because of the fact that LLVM is not thread-safe. This means that the functions and the methods implemented inside LLVM are not protected, as the same threads may use the same data at the same time. We tried to go around this fact by applying more data protection methods. However, it was very challenging because of the dependency on LLVM. While the implementation of the LLVM Dataflow Analysis framework may not be parallelizable or capable of being accelerated, it demonstrates the potential of using LLVM for static analysis on C/C++ programs. The framework offers valuable insights into the implementation of IFDS/IDE solvers and may be considered in the development of our accelerated matrix-based static analysis solver.

Section 3 Summary: Our exploration of the current data-flow analysis implementations, including PhaSAR [14], Graspan-G [19], and LLVM Dataflow Analysis framework [16], led to several findings, which are summarized in Figure 8. PhaSAR produced reliable and reproducible results, but its layered architecture made its parallelization unfeasible in the context of this research project. Graspan-G was a GPU-based framework but had a different formulation of the problem that required CFL-reachability input rather than a matrix-based input. Finally, our exploration of the LLVM Dataflow Analysis framework spanned the longest period of our project. However, due to the framework's heavy reliance on dynamic LLVM data structures, which are not coherent with GPUs, parallelizing this framework was not feasible. The limitations of existing IFDS implementations emphasize the need for further research and exploration into acceleration techniques for data flow analysis. As a result, we decided to start our own simple proof of concept implementation of a matrix-based accelerated static analysis tool.

4 PROPOSAL FOR MATRIX-BASED GPU-ACCELERATED DATAFLOW ANALYSIS IMPLEMENTATION

4.1 Matrix Representation for Efficient Dataflow Analysis

We have adapted the novel approach of FlowMatrix [9] for dataflow analysis in our research. The FlowMatrix approach provides a matrix-based representation for efficient dataflow analysis. The matrix representation captures the taint state of instructions and enables the summarization of information flows. By representing each instruction as a matrix, we can encode the influence of input taint bits on output taint bits using matrix coefficients. This allows us to express the taint rule of an instruction as an n×n matrix, where each element represents the influence of a specific input taint bit on an output taint bit. By performing matrix multiplication operations, we can efficiently compute the summarized data flow matrix, which serves as a concise summary of the information flows that occurred during the program execution. This approach enables us to extract valuable insights into the propagation of taint states across instructions, enhancing our understanding of dataflow dynamics in software systems. Building upon the core principles

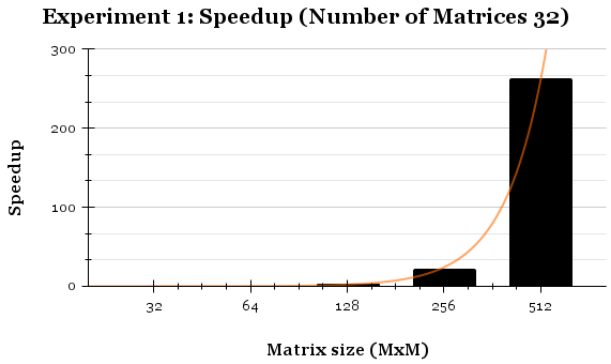

Figure 11: Figure shows Experiment 1 measuring speedup while fixing number of matrices to 32.

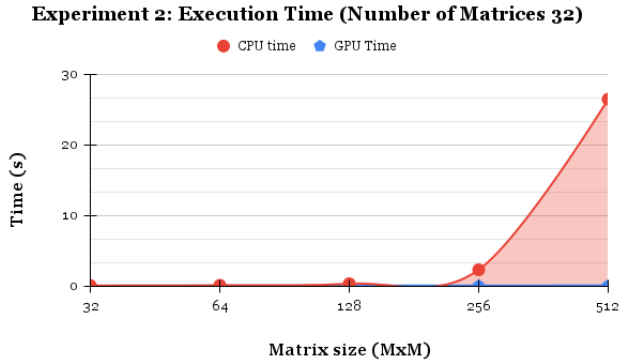

Figure 12: Figure shows Experiment 2 measuring execution time while fixing number of matrices to 32.

and techniques of FlowMatrix, we have tailored this matrix-based representation to suit our specific research objectives. Leveraging this approach, we can pave the way for our own algorithm for multiple matrix multiplication, utilizing the computational power of GPUs to accelerate the analysis process [9].

4.2 Proposal

For our main proposal, we are aiming to preform multiple matrix multiplication. Each matrix multiplication corresponds to the propagation of the taint state from one instruction to another. In our implementation, we created a vector of matrices in order to store each matrix information one after the other, and we will be preforming the multiplication on that vector.

4.2.1 ***Direct Approach:*** A direct approach to preform the matrix multiplication on the whole vector, is to copy the first and the second matrices to the GPU to be multiplied together; the result of their multiplication will then be copied back and stored in the vector of matrices. For the second iteration, the result of the last multiplication will be copied back to the GPU along with the third matrix to be multiplied together. Similarly, the result of the second multiplication will be copied back to the main vector. This process of copying two matrices to the GPU, multiplying them, and storing their result back will be repeated till we manage to multiply all of the matrices in the main vector. However, such process is very time consuming. The time complexity of completing such process will be O(N), where N is the number of matrices in the main vector. The accumulated overhead from coping the matrices to and from the GPU will eventually slow down the performance instead of speeding it up. Therefore, we needed to find and explore a more optimized approach.

4.2.2 ***Optimized Approach:*** For a more optimized solution, we will be copying the whole vector of matrices to the GPU, where each two adjacent matrices will be multiplied together. In the second iteration, each two adjacent result matrices will be multiplied together as well to accumulate the results of the matrices. Such approach will be efficient to accumulate the final result within time complexity equals to $\lceil Log(N) \rceil$, where N is the number of matrices in the main vector. Despite this approach being more efficient than the direct approach, it will lead to thread divergence while preforming the matrix multiplication on the GPU. Thread divergence occurs as the execution paths of the threads that belongs to the same block differ from one thread to another. Such problem might lead to a performance degradation.

To solve such problem, and get the minimum number of thread divergence in the program, we will be assigning the threads responsible for the first half of the matrix multiplication to preform the multiplication. For instance, In the case where we have eight matrices in the vector of matrices, threads responsible for the first matrix will get the result of the first matrix by the fourth matrix, and the threads responsible for the second matrix will get the result of the second matrix by the fifth matrix. This re-arrangement will successfully guarantees the minimum thread divergence occurrence in the program. However, since the matrix multiplication is not commutative, the results obtained from such re-arrangement will lead to a skewed results at the end.

4.2.3 ***Our Optimized Approach:*** In order to eliminate the thread divergence while maintaining the correct results, we needed to preform some preparation on the vector of matrices before preforming any multiplications. The preparation that we did is mainly re-arranging the arrangement of the matrices in the vector. Since we need to multiply the first by the second matrix, we will need to reallocate the position of the second matrix and place it in the beginning of the second half of the vector so that we are able to preform the same algorithm done in the previously mentioned approach. Therefore, we created a function that takes the vector of the matrices and then re-arrange its elements before sending the vector to the GPU. For the rest of the iterations, we needed to preform the same re-arrangement algorithm on the intermediate results that we obtained as the output matrices of the first iteration will be the input for the second one. With that approach, we managed to get the results within time complexity equals to $\lceil Log(N) \rceil$, where N is the number of matrices in the main vector, and maintained the minimum thread divergence.

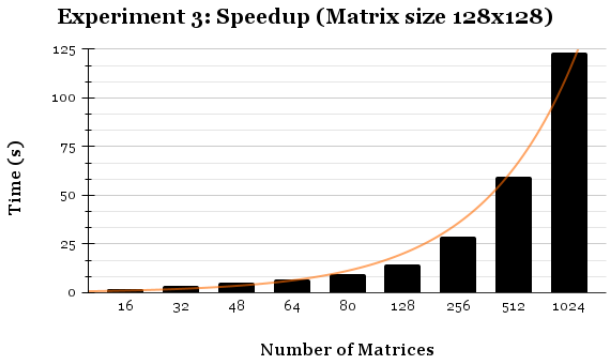

Figure 13: Figure shows Experiment 3 measuring speedup while fixing sizes of matrices to 128×128.

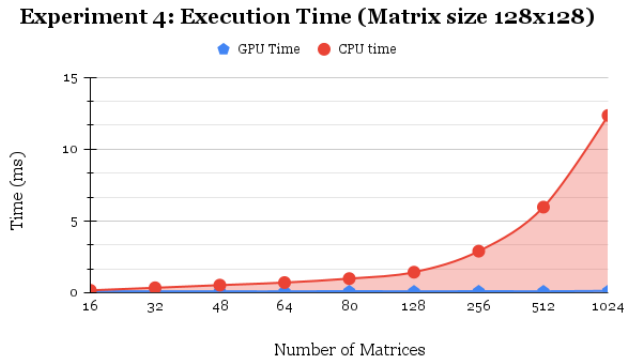

Figure 14: Figure shows Experiment 4 measuring execution time while fixing sizes of matrices to 128×128.

5 EXPERIMENTATION AND EVALUATION

To evaluate the effectiveness of our matrix-based approach for dataflow analysis, we conducted two sets of experiments, as depicted in Figure 9. The experiments were designed to investigate the impact of two main factors: the size of the matrices multiplied, and the number of matrices multiplied. For each experiment, we measured the execution time, the speedup (i.e., the ratio of the execution time on the CPU to the execution time on the GPU), and the number of Giga Floating-Point Operations Per Second (GFLOPS) achieved by the GPU architecture compared to the CPU, in order to observe the effects of these factors. GFLOPS is a measure of the computing power of a processor or a computer system in terms of the number of floating-point operations it can perform in one second. Our experiments were conducted using a GPU Tesla T4 with CUDA Version 12 and specifications listed in Figure 10.

In Experiment 1 (Figure 11), we measured the speedup achieved by the GPU compared to the CPU when multiplying vector matrices of size 32×32, gradually increasing up to 512×512. The number of matrices multiplied was fixed at 32 for each run, and the performance of the CPU was compared to that of the GPU, with the speedup computed accordingly. Due to the time taken by the CPU for larger matrices, we did not plot the results for sizes above 512×512. In Experiment 5 (Figure 15), we aimed to measure the GFLOPS to assess the processing power of the GPU compared to the CPU under the same conditions as Experiment 1. As shown in Figure 11, the speedup increased as the size of the matrices increased, indicating the superior performance of the GPU compared to the CPU. The GFLOPS obtained by the GPU, as shown in Figure 15, also increased with the size of the matrices, demonstrating the computing power of the GPU.

Similar results were obtained in Experiment 3 (Figure 13) and Experiment 6 (Figure 16), where the speedup and GFLOPS were measured while varying the number of matrices multiplied, with their sizes fixed at 128×128. The number of matrices multiplied ranged from 16 to 1024 matrices. In Experiment 1 and Experiment 3, the GPU demonstrated a significant speedup and GFLOPS compared to the CPU, reaching up to $265X$ and $125X$, respectively. Additionally, in Experiment 5 and Experiment 6, the GPU had a larger number of GFLOPS compared to the CPU, indicating the increase in processing power of the GPU. These results suggest that the GPU has the potential to achieve faster computation times and better performance for tasks that involve significant floating-point operations.

In Experiments 2 and 4, we measured the execution time taken by the GPU compared to the CPU to execute the vector of matrices multiplications. In Experiment 2 (Figure 12), we gradually increased the sizes of matrices multiplied from 32×32 up to 512×512, while fixing the number of matrices to 32. In Experiment 4 (Figure 14), we varied the number of matrices multiplied from 16 to 1024, while fixing their sizes to be 128×128. As shown in Figures 12 and 14, as the size of the matrices or the number of matrices increased, the execution time taken by the CPU increased exponentially, while the GPU's execution time increased at a much lower rate. These findings further confirm the results of Experiments 1 and 3, indicating the superior performance of the GPU compared to the CPU for tasks involving large-scale matrix multiplications.

In conclusion, our experiments showcase the effectiveness of our matrix-based approach for dataflow analysis, which utilizes the FlowMatrix approach to provide an efficient matrix-based representation for capturing information flows in software systems [9]. By representing each instruction as a matrix, we can efficiently compute the summarized data flow matrix, which serves as a concise summary of the information flows that occurred during program execution. Our experiments demonstrate the potential of leveraging the computational power of GPUs to accelerate the analysis process, enabling us to extract valuable insights into the propagation of taint states across instructions and enhancing our understanding of dataflow dynamics in software systems. These findings have important implications for the development of more efficient and effective techniques for dataflow analysis, paving the way for the development of high-performance computing systems that can handle large-scale software systems.

6 LIMITATIONS AND FUTURE WORK

Although our approach achieved a speedup of up to 250 times compared to CPU execution time, there are still opportunities for

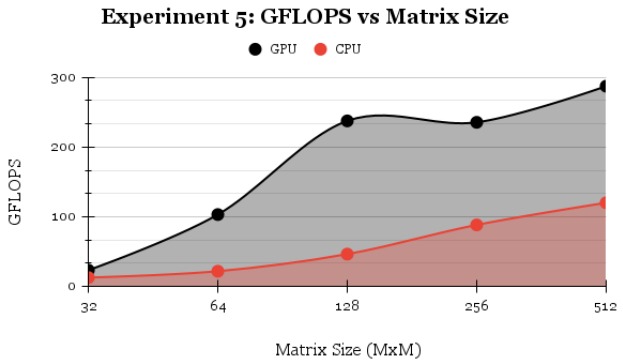

Figure 15: Figure shows Experiment 5 measuring GFLOPS while changing matrices sizes.

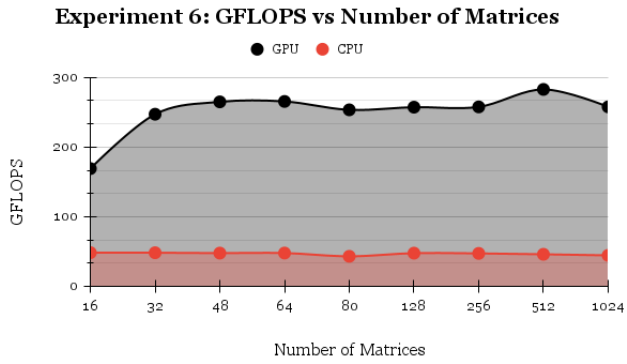

Figure 16: Figure shows Experiment 6 measuring GFLOPS while changing number of matrices.

further optimization. Our approach currently cannot handle sparse matrices efficiently, which is a significant limitation since sparse matrices tend to have irregular memory access patterns due to their sparsity. Additionally, our current binary matrix representation could be handled in a special way on the GPU. For example, we could consider using a compressed sparse matrix format such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) to reduce memory requirements and improve cache utilization. By addressing these limitations, we could potentially achieve a higher speedup than $250X$.

In terms of future work, we could explore migrating the LLVM Dataflow Analysis framework [16] to become thread safe by designing immutable data structures where possible. Immutable objects can be safely accessed and shared across multiple threads without the need for explicit synchronization, reducing the chances of data corruption and simplifying thread safety considerations. Furthermore, for taint analysis, we could build a pipeline of analysis instead of just analyzing one program or one type of analysis in parallel. This could be achieved by performing a higher dimension of matrix multiplication, i.e. $M \times M$ matrix of matrices to be multiplied. Building 3-dimensional grids on the GPU could make this possible. These future directions could significantly improve the performance and scalability of our matrix-based GPU acceleration approach for dataflow analysis, and could be pursued in future research efforts.

We could also examine utilizing algorithmic changes like loop unwinding or vectorization to further improve our method. Loop unwinding, also known as loop unrolling, entails substituting a loop with a fixed number of repetitions with a sequence of separate instructions, decreasing the burden of handling the loop and possibly exposing more chances for parallelism [12]. Vectorization entails executing multiple instructions at once on the same data, exploiting the SIMD (Single Instruction, Multiple Data) abilities of modern processors [15]. By investigating these algorithmic transformations, we could potentially attain even higher accelerations and make our approach more applicable to a wider range of programs.

7 CONCLUSION

In this paper, we explored the potential of accelerating dataflow analysis frameworks on the GPU. Specifically, we focused on PhaSAR, Graspan-G, and the LLVM Dataflow Analysis framework. Our main contributions were designing a worklist algorithm suitable for GPU computation, which requires the LLVM compiler to be thread safe, and designing an n-vector matrix multiplication on the GPU as a proof of concept for speeding up taint analysis.

Our approach addresses a significant limitation of current dataflow analysis frameworks, which either do not produce analysis results or are not compatible with GPUs. We designed our own implementation inspired by FlowMatrix [9], achieving a speedup of up to 250x compared to the CPU implementation. We accomplished this by noticing that the main logic of FlowMatrix happened in multiplying a vector of matrices in sequence, which could be more efficient on the GPU. We designed a novel architecture for such multiplication on the GPU, which could be further optimized for even better results.

Overall, our research findings demonstrate the potential of using GPUs to accelerate dataflow analysis and highlight areas for future research. By addressing the limitations of our approach, such as handling sparse matrices efficiently and exploring additional algorithmic transformations, we could achieve even higher speedups and make our approach more applicable to a wider range of programs.

REFERENCES

[1] Michael Krigsman. [n. d.]. It failure at Heathrow T5: What really happened. https://www.zdnet.com/article/it-failure-at-heathrow-t5-what-really-happened/

[2] Thorsten Blaß and Michael Philippsen. 2019. GPU-accelerated fixpoint algorithms for faster compiler analyses. In *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*, José Nelson Amaral and Milind Kulkarni (Eds.). ACM, 122–134. https://doi.org/10.1145/3302516.3307352

[3] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, Eric Bodden, Laurie J. Hendren, Patrick Lam, and Elena Sherman (Eds.). ACM, 3–8. https://doi.org/10.1145/2259051.2259052

[4] Olivier Bouissou, Yassamine Seladji, and Alexandre Chapoutot. 2012. Acceleration of the abstract fixpoint computation in numerical program analysis. *J. Symb. Comput.* 47, 12 (2012), 1479–1511. https://doi.org/10.1016/j.jsc.2011.12.050

[5] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 332–343. https://doi.org/10.1145/2970276.2970347

[6] December 7 Francis CastanosMonday and archivist Francis Castanos is a librarian. 2020. The cloth of doom: The weird, doomed ride of Ariane Flight 36. https://www.thespacereview.com/article/4085/1

[7] geeksforgeeks. 2022. Data Flow Analysis in compiler. https://www.geeksforgeeks.org/data-flow-analysis-compiler/

[8] jetbrains. [n. d.]. @nullable and @notnull: Intellij idea. https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html

[9] Kaihang Ji, Jun Zeng, Yuancheng Jiang, Zhenkai Liang, Zheng Leong Chua, Prateek Saxena, and Abhik Roychoudhury. 2022. FlowMatrix: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association, USA, 417–434.

[10] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[11] David B Kirk and Wen-mei W Hwu. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, Chapter 5.

[12] W. Lee, D. Shin, and I. Hwang. 2011. Power-efficient loop unwinding for embedded VLIW processors. *ACM Trans. Embed. Comput. Syst.* 10, 4 (2011). https://doi.org/10.1145/2043910.2043914

[13] T. Reps, S. Horwitz, and M. Sagiv. 1995. Interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 49–61. https://doi.org/10.1145/199448.199462

[14] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

[15] B. Sinharoy, M. Silberstein, T. Koshy, and X. S. Liu. 2011. POWER7: IBM's Next Generation Processor. *IEEE Micro* 31, 2 (2011), 6–15. https://doi.org/10.1109/MM.2011.24

[16] Nick Sumner. 2018. LLVM Dataflow Analysis Framework. https://github.com/nsumner/llvm-dataflow-analysis. [Software].

[17] Bill K.H. Sun and Andrei N. Kossilov. [n. d.]. The computerization of Nuclear Power Plant Control Rooms. *Advances in Nuclear Science and Technology* ([n. d.]), 155–169. https://doi.org/10.1007/0-306-47812-9_5

[18] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Daphne Yao. 2020. GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 274–284. https://doi.org/10.1109/IPDPS47924.2020.00037

[19] 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. https://doi.org/10.1145/3466820