GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting

Xiaodong Yu*§¶, Fengguo Wei†‖, Xinming Ou†, Michela Becchi‡, Tekin Bicer§, Danfeng (Daphne) Yao*
*Dept. of Computer Science, Virginia Tech, Blacksburg, VA
†Dept. of Computer Science and Engineering, University of South Florida, Tampa, FL
‡Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC
§Data Science and Learning Division, Argonne National Laboratory, Lemont, IL
Email: xyu@anl.gov, fwei@mail.usf.edu, xou@usf.edu, mbecchi@ncsu.edu, tbicer@anl.gov, danfeng@vt.edu

***Abstract*—Many popular vetting tools for Android applications use static code analysis techniques. In particular, Inter-procedural Data-Flow Graph (IDFG) construction is the computation at the core of Android static data-flow analysis and consumes most of the analysis time. Many analysis tools use a worklist algorithm, an iterative fixed-point approach, to construct the IDFG. In this paper, we observe that a straightforward GPU parallelization of the worklist algorithm leads to significant underutilization of the GPU resources. We identify four performance bottlenecks, namely, *frequent dynamic memory allocations*, *high branch divergence*, *workload imbalance*, and *irregular memory access patterns*. Accordingly, we propose *GDroid*, a GPU-based worklist algorithm implementation with multiple fine-grained optimizations tailored to common characteristics of Android applications. The optimizations considered are: *matrix-based data structure*, *memory access-based node grouping*, and *worklist merging*. Our experimental evaluation, performed on 1000 Android applications, shows that the proposed optimizations are beneficial to performance, and *GDroid* can achieve up to 128X speedups against a plain GPU implementation.**

***Index Terms*—**

I. Introduction

Android platforms nowadays hold 86% of the mobile device OS market share [1]. Consequently, Android phones are a top target of smartphone malware [2], [3]. Therefore, an efficient Android Apps security vetting system is desirable to keep the Google play store clean and safe. However, examining all new and updated Apps in a timely manner is extremely challenging. The Google Play store has currently more than 3.5M[1] Apps, most popular Apps update weekly or even daily, and around 7K[2] new Apps are released everyday. On the other hand, state-of-the-art vetting tools, such as DialDroid, can take 6K hours to analyze 110K real-world Apps [4], which is not scalable.

Many existing popular Android vetting tools, including FlowDroid [5], IccTA [6], DialDroid [4], and AmanDroid [7], use static analysis at their core. Static analysis can provide a comprehensive picture of the App's possible behaviors, whereas dynamic analysis can only screen the execution during a dry run. However, static analysis suffers from an inherent undecidability and has unbounded worst-case complexity. Practically, any static analysis-based tool has to trade precision for execution time. In order to preserve data-flow sensitivity, static analysis tools typically take 30 minutes to analyze an average-size (≈10MB) App [8]. As an example, we used one of the most popular tools – Amandroid [7] – to analyze 1000 random Android Apps. The blue line in Fig. 1 shows the analysis time: as can be seen, Amandroid can take up to 38 minutes to analyze a single App. A faster and more scalable implementation of static analysis for Android Apps is obviously key to achieving efficient security vetting.

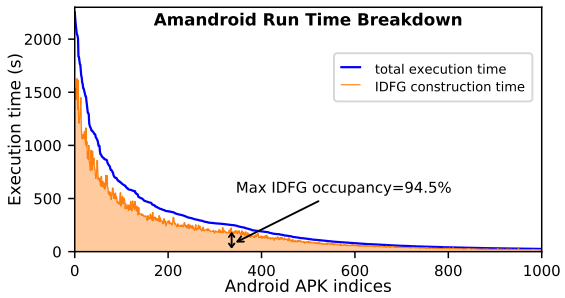

Fig. 1: Execution time of Amandroid. We analyze 1000 random Android Apps. The x-axis represents App indices sorted according to the descending order of Amandroid run time, while the y-axis shows the execution time. The blue line indicates the overall run time while the orange line indicates the IDFG construction time.

Over the last decade, Graphic Processing Units (GPU) have gained popularity due to their massive parallelism and computational power. GPUs have been successfully used to accelerate applications from a variety of domains, including bioinformatics [9], [10], biomedicine [11]–[13], and network intrusion detection [14], [15], to name a few. However, only a handful of previous works have accelerated program analysis on GPU [16]–[18], and all of them have addressed only conventional languages (e.g., C) using analysis approaches that are quite different from Android's. Moreover, they are insensitive to the data-flows of the application. We are not aware of any Android-specific static data-flow analysis implementation on GPU.

¶Work performed during the PhD studies at Virginia Tech.
‖Currently affiliate with Android Security and Privacy Group, Google.
[1]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
[2]https://www.statista.com/statistics/276703/android-app-releases-worldwide/

Accelerating Android static data-flow analysis on GPU is very challenging due to its irregular computational patterns. The worklist algorithm, the computational core of Android static data-flow analysis, has unstructured data accesses and unbalanced workload that conflict with GPU's execution model. It has been shown that straightforward implementations of irregular computational patterns [19] often lead to poor GPU utilization. Indeed, we have verified that a naïve GPU implementation of the worklist algorithm (performed by simply porting a CPU implementation of this algorithm to GPU) leads to a significant underutilization of the GPU's computational resources (we provide the details in Section III). In some cases, the naïve GPU version of the code even underperforms the CPU counterpart.

In this paper we propose *GDroid*, a highly optimized GPU-based worklist algorithm for static data-flow analysis of Android applications. To our best knowledge, this is the first work accelerating Android program analysis on HPC platforms. We first determine four performance bottlenecks in our naïve GPU implementation: *frequent dynamic memory allocations*, *high branch divergence*, *workload imbalance*, and *irregular memory access patterns*. We then propose three fine-grained optimizations to address these bottlenecks. These optimizations leverage Android-specific characteristics to refactor the algorithm to make it more suited to the GPU architecture and execution model. Specifically, we introduce the following optimizations: (i) **Matrix-based data structure**: we store data-facts using a fixed-size matrix-based data structure (rather than the set-based data structure of the original CPU code). This optimization can avoid dynamic memory allocations and reduce memory consumption. (ii) **Memory access-based node grouping**: we group ICFG nodes based on their memory access patterns rather than their statement types. By significantly reducing the number of ICFG node partitions (3 vs. 25), this optimization limits branch divergence. In addition, it allows a better use of the memory bandwidth. (iii) **Worklist merging**: to mitigate the workload imbalance problem, we postpone the processing of subsets of the worklist. By merging nodes appearing multiple times in the worklist, this optimization can also reduce the amount of operation performed.

We evaluate the three proposed optimizations using 1000 randomly selected Android APKs. We find that the first and third optimizations can significantly improve performance compared to the naïve GPU implementation, while the second optimization brings only slight performance benefits. The combined use of the three optimizations, however, leads to the optimal performance: *GDroid* achieves up to 128X speedups over a plain GPU implementation.

Our contributions can be summarized as follows:

- We propose *GDroid*, a highly optimized GPU implementation of static data-flow analysis tailored to Android applications. GDroid is the first implementation of Android program analysis on HPC platforms and serves as the core of fast Android App security vetting.
- We perform a straightforward GPU parallelization of Android's static data-flow analysis, and identify four performance bottlenecks. Accordingly, by leveraging characteristics of Android applications, we propose three fine-grained optimizations to address these performance bottlenecks: matrix-based data structure, memory access based node grouping, and worklist merging.
- We evaluate the efficacy of our proposed optimizations using 1000 Android Apps. Our results show that *GDroid* achieves up to 128X speedups over a plain GPU implementation.

The rest of this paper is organized as the follows. In section 2, we provide the backgrounds regarding static data-flow analysis and the worklist algorithm. In section 3, we introduce our straightforward GPU implementation and analyze its performance bottlenecks. In section 4, we propose our *GDroid* and elaborate the Android-specific optimizations. In section 5, we provide the relate works. In section 6 and 7, we discuss and conclude the paper.

II. Background

In this section, we provide some background information on static program analysis for Android applications. We first introduce Android static data-flow analysis and then present the Worklist algorithm for Inter-procedural Data-Flow Graph (IDFG) construction.

A. Static Data-Flow Analysis for Android Apps

The ultimate goal of static analysis for Android applications is to achieve a minimum false positive rate while capturing all potentially dangerous App behaviors. The creators of Amandroid [7] pointed out that abnormal data flow behavior is a common phenomenon of many security problems. Accordingly, Amandroid conducts static data-flow analysis at its core to build the application's Inter-procedural Data-Flow Graph (IDFG), and then it performs specific analyses by adding plugins on top of the IDFG. This approach computes all objects' points-to information in a context- and flow-sensitive way. Thanks to the IDFG reuse, it shows better versatility and efficiency than other tools focusing on specific analysis tasks. However, due to the time consuming IDFG construction, Amandroid is more computationally expensive. Fig 1 shows a breakdown of Amandroid running time. The orange line indicates the IDFG construction time. As can be seen, this computation takes at least 58% and up to 96% of the total Amandroid execution time. This fact urges us to accelerate IDFG construction on GPU.

B. The Worklist Algorithm for IDFG Construction

An IDFG consists of an Inter-procedural Control-Flow Graph (ICFG) and the node-wise data-fact sets. The data-facts indicate the objects' points-to information. Formally, let C be an Android component, the IDFG can be defined as follows:

$$IDFG(E_C) \equiv ((N, E), \{fact(n) | n \in N\}) \quad (1)$$

where E_C is the environment method of C, N and E are the nodes and edges of the ICFG starting from E_C, and $fact(n)$ is the data-fact set of the statement associated with node n.

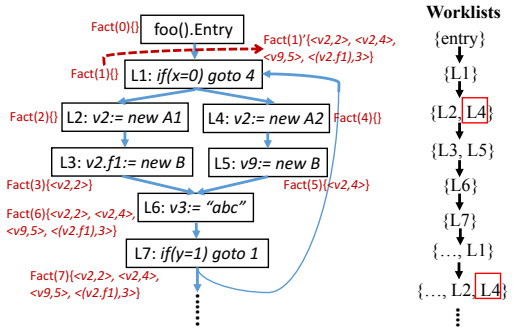

Fig. 2: A sample Inter-procedural Data-Flow Graph (IDFG). Each box is an ICFG node, and each node has a data-fact set colored in red. The blue arrow-lines indicate the ICFG paths.

Fig. 2 shows a sample IDFG. Each ICFG node is associated with a initially empty data-fact set $fact(i)\{\}$. The worklist algorithm constructs IDFG by generating the data-facts and flowing them along the ICFG paths into the corresponding sets. Alg. 1 presents the worklist algorithm. The while loop (ln. 10-ln. 14) is the computational core that iteratively processes the nodes to generate and propagate the data-facts. In each iteration, the algorithm pops out the nodes from the current *worklist* in turn and processes them through the analyzer **ProcessNode**() (ln. 11-ln. 13). **ProcessNode**() analyzes each node to generate the facts and propagates them to the node's successors. Any updated successors then are collected into a set *nodes* (ln. 13) to form the next *worklist* (ln. 14). The algorithm keeps iterating until all nodes are visited and all data-fact sets reach the fixed-point (i.e., the next *worklist* is empty).

Algorithm 1: the Worklist Algorithm for IDFG Construction

```
1  Require: entry point procedure EP;
2  Ensure: IDFG                                   //Inter-procedural Data-Flow Graph
3  Procedure IDFGBuilding(EP)
4      icfg ≡ (N,E);          //load the Inter-procedural Control-Flow Graph
       /* give one data-fact set to each ICFG node */
5      for n_i ∈ N do
6          new empty Set fact(i){};
7          n_i ← fact(i){};
8      new empty List worklist;
9      worklist ← EntryNode_EP;                   //initialize worklist
10     while worklist ≠ ∅ do
11         n ← worklist.front();                  //get the source node
12         worklist.pop_front();
13         nodes ← ProcessNode(icfg,n);  //generate and propogate
             the data-facts
14         worklist ← nodes;                      //form next worklist
15     return (icfg, fact{})
```

The worklist Algorithm shares some similarities with the BFS and DFS algorithms, but has two major differences: it propagates the data-facts and revisits the nodes. For example, in Fig. 2, after visiting node L7, the algorithm updates L1's data-fact set $fact(i)$ to $fact(i)'$ and re-insert L1 into the worklist for revisiting. These two differences prevent the direct reuse of existing optimized GPU implementations of BFS and DFS.

Algorithm 2: Kernel of Plain GPU Implementation

```
1  int *h_icfg, *h_stmt, *h_fact_set;          //ICFG nodes, statement
     information, and data-fact sets
   /* copy data from host to device using dual-buffering */
2  d_icfg←h_icfg; d_stmt←h_stmt; d_fact_set←h_fact_set;
3  Kernel P_WORKLIST(d_icfg, d_stmt, d_fact_set)
4      local int current_worklist, next_worklist;   //in shared memory
       /* two-level parallelization */
5      int nid ⟵ threadIdx.x;          //one thread processes one node
       /* SBDA enables different blocks to process different methods */
6      int mid ⟵ blockIdx.x ∗ blockDim.x;
7      current_worklist ⟵ init_nodes[mid];
8      while !current_worklist.empty() do
9          for i ∈ current_worklist.size()%32 do
10             if nid+i∗32 < current_worklist.size() then
                   /* different threads handle different ICFG nodes */
11                 src ⟵ current_worklist.pop(nid);      //get the
                     source node
12                 d_fact_set(mid, src) ⟵
                     GEN_KILL(d_stmt(mid, src));        //generate the
                     data-facts
13                 dest ⟵ SEARCH(d_icfg(mid, src)); //collect the
                     destination nodes
                   /* propagate the data-facts and form next worklist */
14                 for n ∈ dest do
15                     d_fact_set(mid, n) ∪ d_fact_set(mid, src);
16                     if d_fact_set(mid, n).update() then
17                         next_worklist.insert(n);
18         __syncthread;
19         current_worklist ⟵ next_worklist;
20     h_fact_set ← d_fact_set;          //copy the results back to host
```

III. Our Plain GPU Implementation

In this section, we present our plain GPU implementation, as described in Alg. 2. It is termed plain implementation since it uses only generic approaches without refactoring the algorithm by leveraging any Android-specific characteristics. We first introduce the basic implementation designs, and then analyze the performance bottlenecks.

A. Plain Implementation Design

The plain implementation employs two fine-grained generic GPU implementation techniques, including dual-buffering and two-level parallelization.

1) Dual-Buffering Data Transfer: The worklist algorithm can consume tens of GB memory during a single Android App analysis, which could easily exceed the memory capacity of the commodity GPU. Once the excess happens, we have to divide the ICFG to sub-graphs and process them in turn on GPU. This approach renders multiple data copy ins and outs between CPU and GPU. Although recent advance technologies like NVLink and unified memory can speed up the transfer, CPU-GPU data communication is still one of the major bottlenecks of the GPU performance.

To hide the data transfer overhead, we employ the dual buffering approach by leveraging the asynchronous execution of CUDA kernel and data-transfer engine (Alg. 2 line 2). We allocate two buffers in the GPU memory and create two CUDA execution streams. As the starting point, Stream 1 copies the first sub-graph to Buffer 1. It then launches the kernel to process the first sub-graph, and simultaneously, Stream 2 copies the second sub-graph to Buffer 2. The kernel

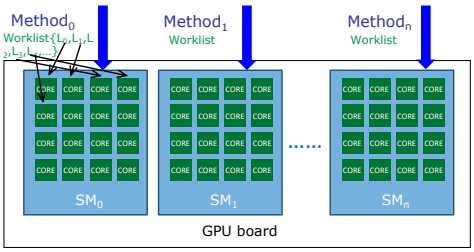

Fig. 3: The two-level parallelization. Different methods are processed in different SM. Each core handles one ICFG node in the current corresponding worklist.

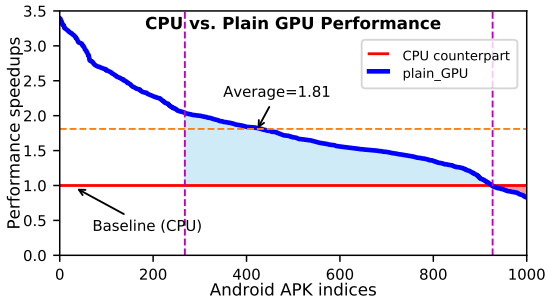

Fig. 4: The performance comparison between the plain GPU implementation and the CPU counterpart. The x-axis represents the App indices sorted according to the descending order of the improvements. The y-axis indicates the speedups compared to the CPU version.

engine keeps processing sub-graphs alternating between the two buffers until the worklist algorithm converges. With dual-buffering, the *(i+1)*th data communication overhead is hidden by overlapping the *i*th kernel execution.

2) Two-level Parallelization: GPU architecture supports two-level parallelism, as depicted in Fig. 3. It is intuitive to map each ICFG node in the worklist onto a CUDA thread (Alg. 2 line 5 and line 10-17). In order to leverage the thread-block level parallelism, we employ the Summary-based Bottom-up Data-flow Analysis (SBDA) [20] to make the Android method analyses parallelizable. The Android methods have dependencies due to the call statements. SBDA generates an unified heap manipulation summary for each method, and lets the IDFG construction utilize the summaries to avoid method revisiting and interleaving. Hence the methods at the same layer are independent of other methods and can be processed by different thread-blocks simultaneously (Alg. 2 line 6). Though the SBDA is conservative, it can still preserve the flow and context-sensitive [21].

B. Performance Analysis

We evaluate the efficiency of our plain GPU implementation using a dataset containing 1K Android Apps. We first compare its execution time to the CPU counterpart and then discover the performance bottlenecks.

1) Comparison with CPU Counterpart: We run our plain implementation on an NVIDIA TESLA P40 GPU. To make the CPU counterpart fairly comparable to the GPU version, we re-implement the worklist algorithm in Amandroid [3] (written in Scala) using multithreading C. Fig. 4 shows the performance comparison between the GPU and CPU implementations. The plain GPU implementation can only achieve 1.81X speedups against the CPU on average. The ultimate achievement is 3.39X speedups, and for the majority (65.9%) of the Apps, GPU only achieves less than 2X speedups (the sky-blue area). For 7.3% of the Apps, GPU even runs slower than the CPU (the red area). Apparently, the plain implementation largely underutilizes the GPU's computation capacity.

2) Performance Bottlenecks: As described in Section II-B, the worklist algorithm is an irregular application to the GPU. We reveal four performance bottlenecks existing in the plain GPU implementation caused by the incompatibility between the original worklist algorithm's computation patterns and the GPU's architecture.

Frequent Dynamic Memory Allocations Although researchers try to mitigate the overhead of GPU dynamic memory allocation [22] recently, it is still a major performance bottleneck due to the hardware limitation. The original worklist algorithm use the *set* data structure to store the data-facts. The exact size of each set is unable to be foreknown; hence we should pre-allocate a fixed-size GPU memory space for each set. The worklist algorithm keeps updating the sets by inserting new data-facts. In the case that the data-fact's volume exceeds the pre-allocated set size, GPU has to dynamically re-allocate the memory space for it. Pre-allocating large space for each set might decrease the frequency of dynamic allocations, but can lead to memory waste.

Large Branch Divergences The GPU executes in the single instruction multiple threads (SIMT) fashion. Too many branches can degrade the performance to nearly serial execution. Unfortunately, the original worklist algorithm classifies the ICFG nodes based on their statement or expression types, and can render 25 different node groups. Specifically, there are nine categories of statements in Android apps: *AssignmentStatement*, *EmptyStatement*, *MonitorStatement*, *ThrowStatement*, *CallStatement*, *GoToStatement*, *Ifstatement*, *ReturnStatement*, and *SwitchStatement*. Furthermore, *AssignmentStatement* consists of 17 different types of expression: *AccessExpr*, *BinaryExpr*, *CallRhs*, *CastExpr*, *CmpExpr*, *ConstClassExpr*, *ExceptionExpr*, *IndexingExpr*, *InstanceOfExpr*, *LengthExpr*, *LiteralExpr*, *VariableNameExpr*, *StaticFieldAccessExpr*, *NewExpr*, *NullExpr*, *TupleExpr*, and *UnaryExpr*. This grouping scheme renders dozens of branches, which is a disaster to the GPU execution.

Load Imbalances In the worklist algorithm context, an ideally balanced workload is a worklist with the node number that is a multiple of the CUDA warp size. It can ultimately utilize the GPU resources since no CUDA cores will be idle during execution. However, between each iteration, the worklist size can sharply differ from one another. Each worklist size is decided by the ICFG structure and unpredictable, makes the

[3]https://github.com/arguslab/Argus-SAF.git

Algorithm 3: Kernel of Our *GDroid*

```
   /* use matrix-based data structure to substitute set-baed one */
   /* group the ICFG nodes according to the memory-access pattern */
1  d_icfg_grp←h_icfg; d_stmt_grp←h_stmt; d_fact_mat←h_fact;
2  Kernel GDROID (d_icfg_grp, d_stmt_grp, d_fact_mat)
3      local int worklist;
4      int nid ⟵ threadIdx.x;
5      int mid ⟵ blockIdx.x ∗ blockDim.x;
6      while !worklist.empty() do
           /* partially sort the worklist for grouping optimization */
7          worklist.sort();
           /* process only head list due to worklist merging optimization */
8          if nid < worklist.size() && nid < 32 then
9              src ⟵ worklist.pop(nid);
               /* use matrix entry lookup to substitute set update */
10             d_fact_mat(mid,
                 src).lookup(GEN_KILL(d_stmt_grp(mid, src)));
11             dest ⟵ SEARCH(d_icfg_grp(mid, src));
12             for n ∈ dest do
13                 d_fact_mat(mid, n).lookup(d_fact_mat(mid,
                     src));
                   /* merge the dest nodes into the current worklist */
14                 if d_fact_mat(mid, n).update() then
15                     worklist.merge(n);
16         __syncthread;
17     h_fact ← d_fact_mat;
```

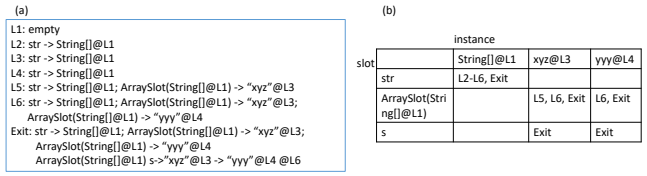

Fig. 5: (a) The data-facts stored in the original set-based data structure. (b) The corresponding data-fact matrix using *MAT* optimization.

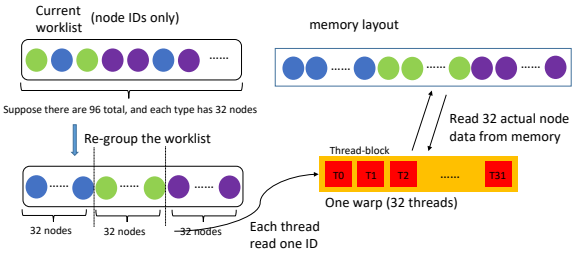

Fig. 6: A worklist processing example using our *GRP* optimization. Three colors indicate the node types: one-time fact-generation, single-layer, and double-layer.

GPU implementation very hard to achieve load balance.

Irregular Memory Access Patterns Achieving coalesced memory accesses is essential for exploiting GPU's massive memory bandwidth capacity. Each GPU memory access reads or writes a 128B memory block. An ideal regular access pattern achieves coalesced memory access by serving all 32 threads in a CUDA warp with the 128B block. Otherwise, multiple memory accesses should be performed and leads to bandwidth wastes. Similar to graph traversal problems, the worklist algorithm's accesses to the ICFG nodes are unstructured, making it extremely difficult to guarantee efficient bandwidth utilization. Moreover, different Android statements and expressions have different memory access patterns, and the data-fact sets are dynamically changing. These make the memory access pattern of the worklist algorithm even more irregular.

IV. The Optimization Designs

In the section, we propose our *GDroid* consisting of three optimizations aiming to break through the performance bottlenecks described above. These optimizations refactor the worklist algorithm by leveraging fine-grained Android-specific characteristics to make it compatible with the GPU's architecture and execution model. Alg. 3 describes the *GDroid* with all three optimizations applied. The statements implementing the optimizations are highlighted in red. We elaborate the three optimizations in the rest of the section.

A. MAT: Matrix-based Data Structure for Storing the Data-Facts

The original worklist algorithm uses the *set* data structure to store the data-facts since it can efficiently insert newly-generated facts. However, the *set* data structure can significantly degrade the GPU-based implementation's performance due to the frequent dynamic memory allocations. Moreover, we find that many repetitive data-facts exist among different data-fact sets. Fig. 5(a) shows some sample data-fact sets of different ICFG nodes stored using the *set* data structure. We can see that data-fact sets of nodes L2, L3, and L4 have exactly the same facts. These repetitions are caused by data-fact propagation and waste a large amount of memory.

A data-fact is a pair consisting of a *slot* and an *instance*. We observe that the pools of *slot* and *instance* can be predetermined prior to the worklist algorithm. The algorithm generates a data-fact by combining a specific *slot* and an *instance* from each pool then propagates it along the ICFG paths. Accordingly, in our first optimization, we propose the *matrix*-based data structure (*MAT*) (Alg. 3 line 1) to substitute the original *set*-based data structure for the data-facts. Fig. 5(b) shows a sample data-fact matrix. The rows of the matrix represent the *slot* pool, and the columns represent the *instance* pool. Once the algorithm generates and propagates a data-fact to a node, it marks the corresponding cell of the matrix. With the *MAT* optimization, we replace the re-allocating and updating of the dynamic-sized sets by the entry looking-up of the fixed-sized matrices (Alg. 3 line 10). The matrix operations are classical regular computation patterns for GPU. Moreover, this optimization can reduce memory consumption due to removing repetitive data-facts.

We implement this optimization by using the fixed-size bit-masks to store matrix cells. Specifically, for a method having n statements, we use an n-bit bit-mask to implement one cell, each bit representing a statement. When a data-fact is propagated to a statement, the corresponding bit is set to 1. This implementation can further reduce the memory usage.

B. GRP: Memory Access Pattern-Based Node Grouping

The original worklist algorithm classifies the ICFG nodes based on their statement or expression types. This classifica-

tion scheme works well on CPU but is significantly ineffi
on GPU due to a large number of divergences. We observ
this statement type-based grouping is not necessary if w
determine *slot* and *instance* pools. As indicated in sectior
generating and propagating the data-facts can be con
to looking-up and combining the entries from the two
Accordingly, we propose a memory access pattern-based
node grouping scheme for the GPU implementation. S
ically, we discover that there are only three memory a
patterns existing:

(i) The one-time fact-generation pattern, e.g., *ConstCla pression*, *NullExpression*, *LiteralExpression*. The with this pattern only generate new data-facts a first visiting. Any re-visiting of these nodes will propagate the data-facts.

(ii) The single-layer pattern, e.g., *VariableNameExpre StaticFieldAccessExpression*. The nodes with this p require single de-reference. It means every visiti them might produce new fact and should access the global memory once.

(iii) The double-layer pattern, e.g., *AccessExpression*, *IndexingExpression*. The nodes with this pattern require double de-reference. It means every visiting of them might produce new fact and should access the global memory twice.

Accordingly, our optimization classifies the ICFG nodes into three groups and stores the nodes in the same group consecutively at GPU memory (Alg. 3 line 1). After each worklist is formed, the optimization performs a partial sort to cluster the nodes in the worklist based on their groups (Alg. 3 line 7). Our optimized grouping scheme can significantly reduce the divergences, given that it has only three groups. Moreover, it can maximize the coalesced memory accesses. Fig. 6 shows an example of optimized node grouping usage. In this case, after the node grouping, each CUDA warp will always process the nodes with the same memory access pattern hence minimize the branch divergence. Furthermore, each memory access has a significant chance to serve multiple threads since the nodes in the same group are stored together, consequently can maximize memory bandwidth utilization.

C. MER: Worklist Merging

We observe that many worklists can lead to imbalanced workloads in the GPU. For instance, if a worklist has 36 ICFG nodes, GPU needs two warps to process it. The second warp has only four threads in effect, hence wastes the GPU's computational resources. Moreover, we find that the original worklist algorithm has many redundant node analyses. In the example shown in Fig. 2, node L4 is processed twice in both the third and eighth worklists due to data-fact updating. However, since the data-fact propagation in the worklist algorithm is monotone, *Fact'(4)* inevitably is the superset of *Fact(4)*. In other words, avoiding processing the L4 in the third worklist will not affect the final IDFG results.

Aiming to solve this load imbalance issue and remove the redundancies, we propose the *worklist merging* (*MER*) optimization. In this optimization, we divide each worklist into two sub-lists: the *head list* and the *tail list*. The *head list* contains the number of nodes that can fully occupy a CUDA warp, while the *tail list* contains the remaining nodes (the imbalance sub-workload). In each iteration, we first process the head list (Alg. 3 line 8) and collect their destination nodes. Instead of processing the tail list, we then merge the destination nodes with the tail list and remove the repetitive nodes to form the next worklist(Alg. 3 line 15). Fig. 7 shows an example of the worklist merging optimization. The head list has 32 nodes hence can fully occupy the warp. The algorithm collects the destination nodes of the head list and merges them with the tail list that has only four nodes. It then removes the repetitive node N_{33} and forms the next worklist. The algorithm keeps processing the worklists in this manner until reaching the fix-point. Postponing the tail list processing can ultimately mitigate the load imbalance and avoid redundant operations. And since the worklist algorithm is insensitive to the node processing order, the *MER* will not affect the final results.

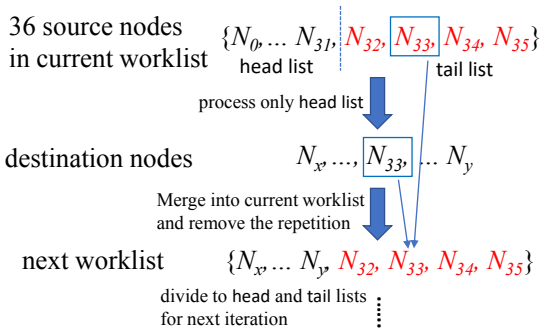

Fig. 7: A worklist merging (*MER*) example. It eliminates the tail list (imbalanced workload) processing and the redundancy (N_{33}).

V. Optimization Evaluations

In this section, we evaluate the efficacy of our *GDroid*. We run the experiments on the machines quipped with a 10-cores Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz, 64GB RAM system memory, and NVIDIA TESLA P40 GPU. This Pascal micro-architectural GPU has 24GB total global memory and 30 streaming-multiprocessors (SM). Each SM has 128 CUDA cores and a 48KB shared memory. The CUDA version is 10. We use the multithreading C-based Amandroid described in section III-B1 as the CPU counterpart and test the GDroid, the plain GPU implementation, and the CPU counterpart over the same 1000 App dataset. These Apps are randomly selected from different categories. Their characteristics are listed in table I.

We currently manually tune the parameters. Empirically 4-5 thread-blocks/Streaming-Multiprocessor (SM) achieves optimal GPU utilization. When the total number of methods is much larger than the number of SM, we assign multiple methods (usually 3-4) to one block, instead of the basic one-method-per-block approach. We leave the auto-tuning design as future work.

TABLE I: Dataset Characteristics

no. of CFG Nodes	no. of Methods
6217	268
no. of Variable	max Worklist length
116	74

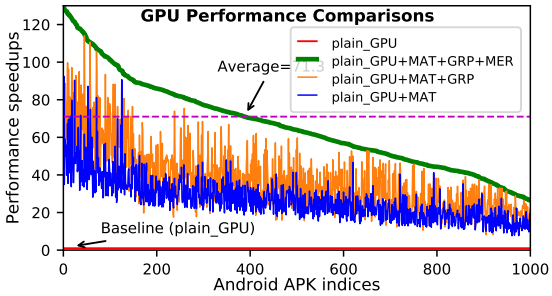

Fig. 8: The *GDroid*'s performance overview. The x-axis indicates the App indices while the y-axis indicates performance speedups against the plain GPU implementation.

We verify the output of the GPU implementations with the original IDFG. Neither the GDroid nor the plain GPU implementation affects the IDFG correctness. In this evaluation, we plan to answer two questions: (i) Can *GDroid* with combined optimizations achieve the optimal performance? (ii) To what extent can each optimization independently improve the performance?

A. *Overview of* GDroid*'s Performance*

Fig. 8 shows the performance comparison of *GDroid* with various optimization combinations by using the performance of plain GPU implementation as the baseline. The x-axis shows the App indices sorted according to the descending order of the *GDroid*'s performance. The y-axis shows the performance speedups against the baseline. The figure indicates that applying all three optimizations to the GPU implementation can achieve the optimal performance: a 128X peak speedup and 71.3X average speedup against the plain GPU implementation. As also can be seen, the matrix-based data structure (*MAT*) and the worklist merging (*MER*) optimizations can largely improve the performance of GPU implementation. Though the memory access pattern based node grouping (*GRP*) optimization can also improve the performance, it is not significant. In the following sections, we will zoom in the performance and evaluate each optimization respectively.

B. *Evaluation of Matrix Data Structure Optimization*

We firstly apply the matrix-based data structure (*MAT*) optimization and compare the performance to the plain GPU implementation (the baseline), as shown in Fig. 9. The x-axis shows the App indices while the y-axis shows the performance speedups against the baseline. Notice that we sort the Apps based on the descending order of the improvements.

The comparison indicates that *MAT* optimization can significantly improve the performance of GPU implementation. It achieves at least 7.6X and up to 92.4X and 26.7X on average

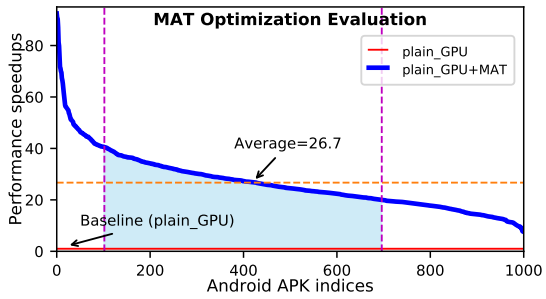

Fig. 9: The efficacy evaluation of matrix-based data structure optimization (*MAT*) by comparing to the performance of plain GPU implementation (the baseline). The x-axis indicates the App indices while the y-axis indicates the execution time speedups against the baseline.

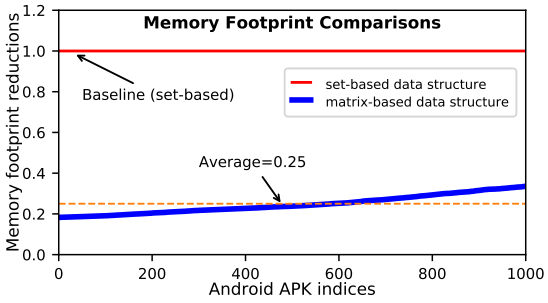

Fig. 10: The memory footprint comparison between the matrix-based and the original set-based data structures for storing the data-facts. The x-axis indicates the App indices while the y-axis indicates the memory footprint.

speedups against the plain GPU implementation. For 59.4% of the Apps, the *MAT* can improve 20X-40X performance (the sky-blue area in Fig. 9). The performance improvement is remarkable because *MAT* optimization can eliminate dynamic memory allocation, which is one of the major GPU performance bottlenecks.

Since *MAT* optimization is also designed to reduce memory consumption, we compare its memory footprint to the footprint of the original set-based data structure. Fig. 10 shows the comparison between these two data structures. As can be seen, the matrix-based data structure can reduce 75% memory consumption on average. It at most needs 34% memory space compared to the set-based data structure. The *MAT* data structure can significantly reduce the memory footprint thanks to the elimination of storing repetitive data-facts.

C. *Evaluation of Memory-Access Pattern-Based Node Grouping Optimization*

In this section, we apply the memory-access pattern-based node grouping (*GRP*) optimization in addition to the GPU implementation with *MAT* (the baseline). Fig. 11 shows the performance comparison between the GPU implementation with and without the *GRP* optimization. The y-axis shows the

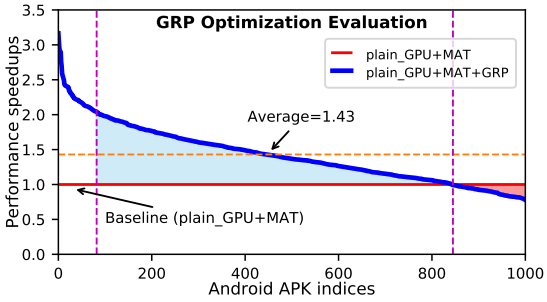

Fig. 11: The efficacy evaluation of memory-access pattern-based node grouping (*GRP*) optimization by comparing to the performance of GPU implementation with only *MAT* optimization (the baseline). The x-axis indicates the App indices while the y-axis indicates the execution time speedups against the baseline.

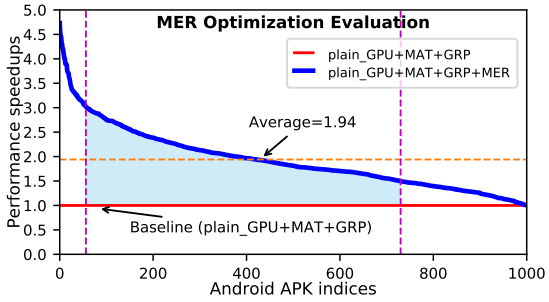

Fig. 12: The efficacy evaluation of worklist merging (*MER*) optimization by comparing to the performance of GPU implementation with *MAT+GRP* optimization (the baseline). The x-axis indicates the App indices while the y-axis indicates the execution time speedups against the baseline.

TABLE II: Worklist Profiling

	Worklist sizes		
	≤32	>32 & ≤64	>64
before MER	87.6%	4.3%	8.1%
after MER	74.4%	11.9%	13.7%
	no. of Worklist iteration		
	average	max	min
before MER	5.6K	6.8K	4.3K
after MER	4.5K	5.8K	3.6K

speedup of the *GRP* over the baseline. Notice that the baseline in this figure is the GPU implementation with *MAT* rather than the plain GPU implementation. Accordingly, the speedup is the improvement in addition to the *MAT* optimization. For example, 1.43X in Fig. 11 indicates the speedup against *MAT* and is equivalent to 38.2X (1.43*26.7) speedup against plain GPU implementation.

As can be seen, the *GRP* can only slightly improve performance. For 76.3% of the Apps, it only achieves less than 1.5X speedups (the sky-blue area in Fig. 11); for 15.5% of the Apps, it even degrades the performance (the red area in Fig. 11). This performance degradation is caused by the fact that most worklists have small sizes. Table II shows the profiling results of the worklists. The third line of the table indicates that, on average, 87.6% of the worklists in one IDFG construction instance have less than 32 ICFG nodes. It means that most of the worklists can fit into a single warp, hence they cannot take the divergence reduction merit of *GRP* but will suffer from its sorting overhead. However, thanks to the increasing of the coalesced memory access rate, in some cases, the *GRP* can still achieve a slight improvement in the overall performance.

D. Evaluation of Worklist Merging Optimization

We finally apply the worklist merging (*MER*) optimization in addition to the GPU implementation with both *MAT* and *GRP* optimizations. Fig. 12 shows the performance comparison between the GPU implementation with and without *MER*. The y-axis shows the performance speedups compared to GPU implementation without *MER* (the baseline). We emphasize that here the baseline is not the plain GPU implementation but rather the implementation with both *MAT* and *GRP*. Accordingly, the speedup indicates the improvement over *MAT+GRP*, namely, 1.94X represents 74X(1.94*1.43*26.7) speedup over plain GPU implementation.

As can be seen, the *MER* optimization can significantly improve performance. It can achieve up to 4.76X speedups over the baseline performance. For a majority of the Apps (67.4%), the *MER* can achieve 1.5X-3X speedups (the sky-blue area in Fig. 12). The average performance increase is 1.94X.

The improvement by using *MER* is significant since it can efficiently reduce the number of worklist algorithm iterations and enlarge the worklist sizes. The eighth line of Table II indicates that, on average, the *MER* optimization can decrease 1.1K iterations in one worklist algorithm instance. Besides the iteration reduction, it merges the *tail list* with the predecessor worklists, thus can enlarge the sizes of the worklist. The fourth line of Table II indicates that 25.6% worklists require more than one warp to process. Hence the *MER* can boost the divergence reduction benefits of the *GRP* optimization.

E. Evaluation Summary

- Our *GDroid* can achieve up to 128X speedups against the plain GPU implementation.
- The *MAT* and *MER* optimizations can significantly improve the GPU implementation's performance (20X-40X) while the *GRP* can slightly improve it (less than 1.5X).
- The *MAT* can also reduce 75% memory usage.

VI. Related Work

Many tools have been proposed for applying the static analysis of Android security problems. FlowDroid [5] first builds a call graph based on Spark/Soot [23] by performing a flow-insensitive points-to analysis, then conducts the taint and on-demand alias analysis based on the call graph. FlowDroid performs only partially text- and flow-sensitive analysis due to the computational cost concerns [24]. IccTA [6] extends FlowDroid by adopting IC3 [25] Intent resolution engine.

IccTA can track data flows through regular Intent calls and returns but can't through remote procedure call (RPC). On the contrary, DroidSafe [26] tracks both Intent and RPC calls but can't track data flows through "stateful ICC" nor inter-app analysis. DialDroid [4] is a scalable and accurate tool designed for inter-app Inter-Component Communication (ICC) analyses. It makes a balance of the data-flow analysis accuracy and runtime cost.

Each of the above tools is built to perform one or several specific kinds of static analysis and is not flexible to extend the capability. On the other hand, Amandroid [7] provides generic support to multiple Android static analyses. It builds the DFG and DDG, then adds low-cost plugins to realize various specific analyses. Amandroid calculates all objects' points-to information in a context- and flow-sensitive way. Although this boosts the precision, it drastically increases the computation burdens [8].

The DFG is constructed through Inter-procedural Data-flow Analysis. This data-flow analysis is conducted using iterative algorithms. The conventional iterative search algorithm visits each ICFG node once in one iteration, and keeps iterating until no further changes occur to the data-flow sets [27]. This approach has a regular computation pattern hence can leverage the well-studied depth-first and breadth-first search (BFS/DFS) [28]–[35] to efficiently visit the CFG in each iteration. However, it has large redundancy and slow convergence due to the fixed full workload in each iteration. The worklist algorithm is an alternative that dynamically updates the worklist after each node visiting. The inter-procedural, finite, distributive subset (IFDS) [36] and its extension, the inter-procedural distributed environment transformers (IDE) [37], are two well-known conceptual frameworks using the worklist algorithm as the core. Some optimizations [38]–[40] have been proposed to algorithmically improve the IFDS efficiency. Recently, two mature IFDS/IDE implementations gain massive popularity. IBM releases T.J. Watson Libraries for Analysis (WALA) implementing the IFDS [41]. Heros [42] provides IFDS/IDE solver on top of Soot [43]. Although the worklist algorithm algorithmically outperforms the conventional algorithm, it still potentially requires high computational cost due to the unpredictable number of iterations. Even worse, the dynamic update of the worklist makes the computation pattern pretty irregular and extremely hard to be parallelized.

All aforementioned work only target on executing the analyses on sequential platforms. The existing works on executing static analyses on modern parallel platforms are rare, and only a handful of them use GPUs. Prabhu et al. [16] accelerate the 0CFA, a higher-order control-flow analysis algorithm, with GPU. They leverage the sparse-matrix to optimize CFG data structures and achieves 72X speedups over the CPU version. Mendez-Lojo et al. [17] accelerate the Andersen-style inclusion-based point-to analysis on GPU. They treat the analysis as a graph modification and traversal problem and focus on providing general insights about implementing graph algorithms on GPU. Su et al. [18] also implement Andersen's inclusion-based pointer analysis. Their design has a similar idea to [17] and proposes several techniques to further optimize the implementation. They claim their work can achieve 46% speedup against [17]. Each of the above three work targets only on one specific type of analysis, and the design is not directly applicable to other static analyses. Moreover, none of them associates the algorithm with specific language or file type; significant application-specific tuning may be required when applying their algorithms to analyze real-world cases.

VII. Takeaways

Application-Specific Optimizations Many domain applications have complex computation patterns. For example, our Android program analysis has set updating and fixed-point iterations in addition to the node traversal. Our work, along with some other previous researches, has shown that straightforward implementations or even utilizing optimized generic libraries for these applications are insufficient to fully leverage the parallel devices' capacity. Moreover, some well-studied optimization approaches are not directly applicable. For example, in our case, the generic optimization – allocating fixed memory space, is not usable unless we pre-determine the data-fact sets' size by exploiting the data-flow analysis's characteristics. This paper provides a further illustration indicating that fine-grained application-specific optimizations are the key to broaden the applicability of high-performance platforms e.g., GPU.

High-Performance Implementations of Security Applications Typical cybersecurity solutions emphasize on achieving defense functionalities. Our statistical results [44] show that only around 5% of recent top-tier security conference papers consider the high-performance implementations, and less than 1% of recent top-tier HPC conference papers focus on security applications. We believe the irregular computation patterns, for example, the pattern in this work and the Finite-State Machine for the Network Intrusion Detection [45], [46], particularly deter the HPC-based implementations of security application. On the other hand, execution efficiency and scalability of the security applications are equally important, especially for real-world deployment. Our work provides an instance showcasing that security applications can also benefit from the power of HPC devices with fine-grained application-specific implementations and optimizations.

VIII. Conclusion and Future Work

In this paper, we propose *GDroid*, a highly optimized GPU-based worklist algorithm for Android static data-flow analysis. We first design the plain GPU implement using only general approaches, including dual-buffering data transfer and two-level parallelization. We experimentally show this plain implementation is largely inefficient and discover four major performance bottlenecks: frequent dynamic memory allocation, large branch divergences, load imbalance, and the irregular memory access pattern. Accordingly, we leverage Android-specific characteristics and propose three fine-grained optimizations, including matrix-based data structure for data-facts, memory access pattern based node grouping, and worklist merging. The

matrix-based data structure is also designed to reduce memory consumption. In our evaluation, the experiment results show that all three optimizations, especially the matrix-based data structure and worklist merging optimizations, can improve the performance. The optimal GPU implementation can achieve up to 128X speedups against plain GPU optimization. The matrix-based data structure can save 75% memory space on average compared to the set-based data structure.

In the future, given the amount of Android Apps is large, we consider to map the worklist algorithm onto multi-GPU platforms or even GPU clusters. This kind of implementation requires sophisticated designs regarding data partitions and communications between GPUs [47], [48]. Besides, we can implement the algorithm on other emerging parallel devices, e.g., Automata Processor (AP) [49]. AP is approved to be more efficient than GPU for some traversal-based algorithms [50], [51]. There is room for an optimized AP-based worklist algorithm.

Acknowledgements

This work has been supported by ONR under Grant ONR-N00014-17-1-2498 and NSF under the grants CNS-1565314/1838271, CNS-1717862, and CCF-1741683. The authors thank Drs. Matthew Hicks and Ali Butt for their feedback on the work.

References

[1] Gartner. Gartner Says Worldwide Sales of Smartphones Returned to Growth in First Quarter of 2018, 2018. https://www.gartner.com/newsroom/id/3876865.

[2] Karim O Elish, Xiaokui Shu, Danfeng Daphne Yao, Barbara G Ryder, and Xuxian Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.

[3] Ke Tian, Danfeng Yao, Barbara G Ryder, and Gang Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 262–271. IEEE, 2016.

[4] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[6] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291. IEEE Press, 2015.

[7] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[8] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do Android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341. ACM, 2018.

[9] Kaixi Hou, Hao Wang, and Wu-chun Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 780–789, May 2016.

[10] Jing Zhang, Hao Wang, and Wu-chun Feng. cublastp: Fine-grained parallelization of protein sequence search on CPU+ GPU. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 14(4):830–843, 2017.

[11] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[12] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. An enhanced image reconstruction tool for computed tomography on GPUs. In *Proceedings of the Computing Frontiers Conference*, CF'17. ACM, 2017.

[13] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. GPU-based iterative medical CT image reconstructions. *Journal of Signal Processing Systems*, 91(3-4):321–338, 2019.

[14] Xiaodong Yu and Michela Becchi. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 18:1–18:10, New York, NY, USA, 2013. ACM.

[15] Xiaodong Yu. *Deep packet inspection on large datasets: algorithmic and parallelization techniques for accelerating regular expression matching on many-core processors*. University of Missouri-Columbia, 2013.

[16] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating Flow Analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.

[17] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 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 '12, pages 107–116, New York, NY, USA, 2012. ACM.

[18] Yu Su, Ding Ye, Jingling Xue, and Xiang-Ke Liao. An efficient GPU implementation of inclusion-based pointer analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):353–366, 2016.

[19] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[20] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Notices*, volume 46, pages 567–577. ACM, 2011.

[21] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150. ACM, 2018.

[22] Isaac Gelado and Michael Garland. Throughput-oriented GPU memory allocation. In *Proceedings of the 24th Symposium on*

Principles and Practice of Parallel Programming (PPoPP '19), pages 27–37. ACM, 2019.

[23] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International conference on compiler construction*, pages 18–34. Springer, 2000.

[24] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Highly Precise Taint Analysis for Android Applications. Technical report, EC SPRIDE, 2013.

[25] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press, 2015.

[26] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[27] Darren C Atkinson and William G Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 52. IEEE Computer Society, 2001.

[28] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[29] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, 2011.

[30] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, 2012.

[31] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378–389. IEEE, 2012.

[32] Federico Busato and Nicola Bombieri. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1826–1838, 2014.

[33] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[34] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12, 2016.

[35] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 38–52. ACM, 2019.

[36] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[37] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[38] Derek Rayside and Kostas Kontogiannis. A generic worklist algorithm for graph reachability problems in program analysis. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*, pages 67–76. IEEE, 2002.

[39] Hakjoo Oh. Large spurious cycle in global static analyses and its algorithmic mitigation. In *Asian Symposium on Programming Languages and Systems*, pages 14–29. Springer, 2009.

[40] Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *International Conference on Compiler Construction*, pages 124–144. Springer, 2010.

[41] IBM. *T.J. Watson Libraries for Analysis (WALA)*. http://wala.sourceforge.net/wiki/index.php/Main˙Page.

[42] Eric Bodden. 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*, pages 3–8. ACM, 2012.

[43] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[44] Xiaodong Yu. *Algorithms and Frameworks for Accelerating Security Applications on HPC Platforms*. PhD thesis, Virginia Tech, 2019.

[45] Xiaodong Yu, Bill Lin, and Michela Becchi. Revisiting state blow-up: Automatically building augmented-FA while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications*, 32(10):1822–1833, Oct 2014.

[46] Xiaodong Yu, Wu-chun Feng, Danfeng (Daphne) Yao, and Michela Becchi. O3FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ANCS '16, pages 1–11, New York, NY, USA, 2016. ACM.

[47] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang, Sayantan Sur, and Dhabaleswar K Panda. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *2011 IEEE International Conference on Cluster Computing*, pages 308–316. IEEE, 2011.

[48] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K Panda. GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, 2013.

[49] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying Automata Processing: GPUs, FPGAs or Micron's AP? In *Proceedings of the International Conference on Supercomputing*, ICS '17. ACM, 2017.

[50] Indranil Roy, Nagakishore Jammula, and Srinivas Aluru. Algorithmic techniques for solving graph problems on the automata processor. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 283–292. IEEE, 2016.

[51] Xiaodong Yu, Kaixi Hou, Hao Wang, and Wu-chun Feng. Robotomata: A Framework for Approximate Pattern Matching of Big Data on an Automata Processor. In *2017 IEEE International Conference on Big Data (Big Data)*, 2017.