Time- and Space-Efficient Flow-Sensitive Points-to Analysis

RUPESH NASRE, The University of Texas at Austin

Compilation of real-world programs often requires hours. The term *nightly build* known to industrial researchers is an artifact of long compilation times. Our goal is to reduce the absolute analysis times for large C codes (of the order of millions of lines). Pointer analysis is one of the key analyses performed during compilation. Its scalability is paramount to achieve the efficiency of the overall compilation process and its precision directly affects that of the client analyses. In this work, we design a time- and space-efficient flow-sensitive pointer analysis and parallelize it on graphics processing units. Our analysis proposes to use an extended bloom filter, called multibloom, to store points-to information in an approximate manner and develops an analysis in terms of the operations over the multibloom. Since bloom filter is a probabilistic data structure, we develop ways to gain back the analysis precision. We achieve effective parallelization by achieving memory coalescing, reducing thread divergence, and improving load balance across GPU warps. Compared to a state-of-the-art sequential solution, our parallel version achieves a 7.8× speedup with less than 5% precision loss on a suite of six large programs. Using two client transformations, we show that this loss in precision only minimally affects a client's precision.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Points-to analysis, flow sensitivity, program analysis, compilers, bloom filters, approximation, GPGPU

ACM Reference Format:
Nasre, R. 2013. Time- and space-efficient flow-sensitive points-to analysis. ACM Trans. Architec. Code Optim. 10, 4, Article 39 (December 2013), 27 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555296

1. INTRODUCTION

Continued acceleration of open-source efforts and expansion of code bases challenges the scalability of current compilation processes. Compilation of large source codes often requires hours, hampering programmer productivity, eating important system resources, and reducing, ultimately, a company's revenue. Optimizing analyses for better running times has been a long-standing research agenda and several novel mechanisms have been developed toward this goal.

Points-to analysis is a key static analysis technique during compilation. It enables analyses and transformations of several, otherwise pessimistically analyzed code fragments, such as loop-invariant code motion, common subexpression elimination, dead-code elimination, program slicing, and so forth. Precision of underlying points-to

New Article, Not an Extension of a Conference Paper.
Author's addresses: R. Nasre, Institute for Computational Engineering and Sciences, The University of Texas at Austin, USA.

DOI: http://dx.doi.org/10.1145/2555289.2555296

analysis directly affects the client analyses and transformations [Hind and Pioli 2000]. However, industry-strength compilers need to use flow-insensitive pointer analysis because of the high analysis time and memory cost of a flow-sensitive analysis.

The benefit of a flow-sensitive pointer analysis over that of a flow-insensitive analysis has not been clear [Hind and Pioli 1998]. However, it has been shown that a precise pointer analysis is helpful to several clients, such as typestate verification [Fink et al. 2008], security analysis [Chang et al. 2008], bug detection [Guyer and Lin 2005], and the analysis of multithreaded programs [Salcianu and Rinard 2001]. As a result, there is a renewed interest in the area of flow-sensitive pointer analysis, and the scalability of such analyses has been greatly improved [Hardekopf and Lin 2011; Li et al. 2011; Yu et al. 2010; Lhoták and Chung 2011; Hardekopf and Lin 2009; Kahlon 2008].

However, despite these efforts, industrial response to the adoption of these analyses has been lukewarm. For instance, widely used compilers like GCC [GCC 2013] and LLVM [Lattner and Adve 2004] rely on flow-insensitive pointer analysis, despite the known advantages of a flow-sensitive analysis. One of the main reasons behind this pessimistic reaction is high absolute running times of several analyses over large-sized codes. As an example, a state-of-the-art flow-sensitive pointer analysis [Hardekopf and Lin 2011] over `gs`, an open-source postscript viewer, totaling 0.4 million lines of C code, requires more than half an hour to complete! Considering that pointer analysis consumes only a part of the total compilation time, the overall compilation is painfully slow.

Analysis parallelization provides an attractive avenue under such circumstances. The compilation process can take advantage of the multiple cores present on the build system to analyze programs. In the past, researchers have proposed several interesting approaches to parallelize general dataflow analyses [Lee et al. 1990; Kramer et al. 1992; Prabhu et al. 2011].

Former research on parallelizing pointer analysis has focused mainly on flow-insensitive analyses. However, these approaches are insufficient for flow-sensitive points-to analysis, which poses the following challenges. First, absolute runtimes of sequential flow-sensitive analyses are quite high; for industrial applications with millions of lines of code, these are prohibitively high. Current parallelization techniques can reduce the runtime only by a small factor. Therefore, we need sophisticated mechanisms to reduce the runtime of the single-threaded execution as well as to gain nontrivial speedups with parallel execution. Second, the memory requirement of a flow-sensitive analysis is often several times higher compared to that of its flow-insensitive counterpart, due to maintaining per-program-state information vis-a-vis whole-program information. Third, any efforts in reducing analysis time or memory usually results in reduced precision. In general, improving on one dimension results in trading off on another dimension. This is best illustrated using the tradeoff chart shown in Figure 1. The chart shows three axes: one each for analysis time, memory requirement, and precision loss. An analysis can then be denoted as a triangle on this chart depending on its placement on each axis. The desired place for an analysis is the center of the triangle where it obtains the minimum time, minimum memory usage, and lowest loss in precision.

In practice, however, the desired place is not achieved. For illustration purposes, the chart shows relative placements of sequential flow-insensitive (`seqFI`) and flow-sensitive (`seqFS`) points-to analyses. A `seqFI` has low time and memory requirements but suffers from high precision loss. In contrast, a `seqFS` has good precision (low precision loss) but high time and memory requirements. Simple parallelization of `seqFS` achieves better runtime, keeping the memory requirement and precision intact. This parallel analysis is shown as plot `parFS` in Figure 1. When the build process runs over millions of lines of code, the absolute runtime of `parFS` quickly becomes the bottleneck.

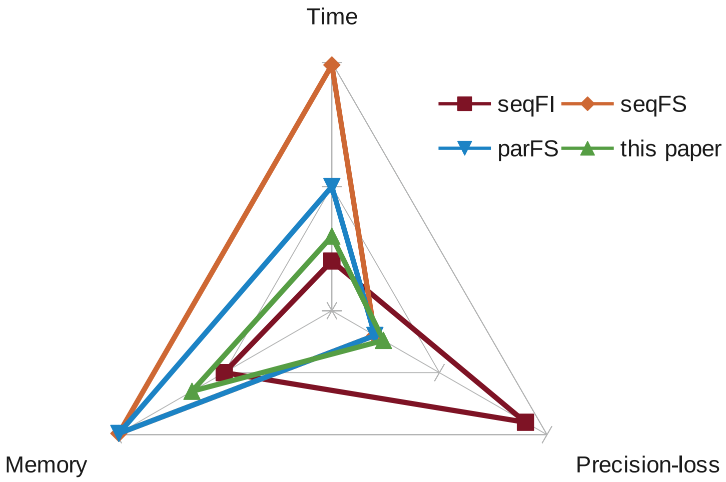

Fig. 1. Tradeoff between analysis time, memory requirement, and precision (chart not to scale).

In this work, we target such large codes (for smaller codes, existing techniques can be applied). To improve the time and space efficiency, we propose to parallelize and approximate the analysis by storing points-to information in bloom filters. Bloom filters are probabilistic data structures to store sets. Although time and space efficient, bloom filters pose several challenges in our case.

—Bloom filters are best suited for membership queries of elements. However, they are not suited for supporting enumeration queries—required for propagating points-to information across pointers.
—A bloom filter and its extensions do not support an arbitrary number of element deletions. This poses a challenge for a flow-sensitive analysis wherein a previously computed points-to information may get killed and removal of a points-to fact from the bloom filter is necessary to ensure precision.
—Storing points-to sets in bloom filters in an approximate manner introduces false-positives while querying. This reduces precision, (partially) taking away the benefits of flow sensitivity.

Further, it is unclear how to exploit parallelism out of a flow-sensitive points-to analysis built using bloom filters. The rest of this article addresses the aforementioned challenges and illustrates how to regain analysis precision and how to effectively parallelize the analysis on a Graphics Processing Unit (GPU). Specifically, this work makes the following contributions.

—We design the first points-to analysis on GPUs that uses approximate representation. We use multibloom, an extension of the basic bloom filter, to store all the points-to information in a succinct manner [Nasre et al. 2009]. This greatly reduces the memory requirement of the analysis and enables it to scale to large programs. We use several techniques such as bootstrapping the analysis and multiple hash functions to keep the imprecision due to approximation under control.
—We present the first parallel flow-sensitive points-to analysis on GPUs. We show how bloom filters help us achieve better spatial locality, enabling improved parallelism. We also exploit the monotonicity property of the analysis to reduce synchronization bottlenecks. We also dynamically distribute tasks to GPU threads to reduce load imbalance and thread divergence across threads. Further, we alter the memory layout

of the bloom filter to achieve memory coalescing. Our analysis is field sensitive (different fields of an aggregate are treated as separate variables) and interprocedural but context insensitive (calling context of a function is ignored).
—Using a suite of six large open-source programs, we illustrate the effect of our analysis and show that for several configurations, our analysis results in significant improvements over a sequential CPU version (average 7.8× speedup and 5.9× memory reduction) with small precision loss.
—Using two clients—loop invariant code motion and aggressive dead-code elimination—which are part of the LLVM framework [Lattner and Adve 2004], we show that the precision loss due to approximate points-to analysis only marginally affects the precision of the clients. This means a client can continue using our improved analysis without incurring much precision penalty.

The rest of this article is organized as follows: We explain preliminary background in Section 2, which covers sequential flow-sensitive points-to analysis and the use of bloom filters. Section 3 explains how to use bloom filters for flow-sensitive points-to analysis. Section 4 details GPU parallelization of the analysis. Section 5 evaluates the effectiveness of our approach. Section 6 discusses the relevant related work. We conclude in Section 7.

2. BACKGROUND

In this section, we briefly introduce flow-sensitive points-to analysis and summarize bloom filters.

2.1. Flow-Sensitive Points-to Analysis

In a flow-sensitive analysis, the points-to information is computed by respecting the control flow in the program. Specifically, points-to information at each basic block is computed using an iterative dataflow analysis. The dataflow equations are as shown in the following:

$$\begin{aligned} \mathrm{IN(B)} &= \cup\, \mathrm{OUT(P)}, \forall \mathrm{P} \in \mathrm{predecessors(B)} \\ \mathrm{OUT(B)} &= \mathrm{GEN(B)} \cup (\mathrm{IN(B)} - \mathrm{KILL(B)}). \end{aligned} \tag{1}$$

GEN(B) indicates all the dataflow facts generated in B, that is, all the variables defined in B. KILL(B) indicates all the dataflow facts in the program that get killed (or overwritten) by a definition in B. Thus, the set of incoming dataflow facts denoted by IN(B) for basic-block B is computed by the set-union of the outgoing dataflow facts of all the predecessors of B. The set of outgoing dataflow facts denoted by OUT(B) for basic-block B is the union of the new dataflow facts generated in B and all its incoming dataflow facts that are not killed by the definitions in B. Since we deal with *may* alias analysis, the confluence (or meet) operator is union (∪). Also, since the IN sets are defined in terms of the OUT sets of predecessors, this becomes a forward-flow problem.

Note that the IN and the OUT sets are defined with a circular interdependence. A solution to the earlier dataflow equations is computed iteratively until a fixed point is reached.

Efficient flow-sensitive analyses depend on the partial static single assignment (SSA) form of the program. In partial SSA form, each top-level (or non-address-taken) variable is converted to an SSA form; that is, each of its definitions is modeled as an assignment to a new variable. The remaining (address-taken) variables are not converted to SSA form, because their precise definitions cannot be modeled without performing a precise points-to analysis. However, since pointer analysis and the dataflow analysis are cyclically dependent, the program in partial SSA form is incrementally refined to include more and more definitions in SSA form as more points-to facts are generated.

2.2. Bloom Filters

Bloom filters [Bloom 1970] are probabilistic data structures that support fast insertions and membership queries; both the operations are performed in $O(1)$ time using hash functions. A basic bloom filter has a fixed-sized bucket of bits. Insertion of an element is done by hashing the element and setting an appropriate bit in the filter. Membership of an element is then simply rehashing the element and checking if the corresponding bit in the filter is set. Since two elements may hash to the same bit, conflicts can occur. If two elements conflict and only one of them is inserted in the bloom filter, then querying for the other element can generate a *false positive*. That is, although the second element was never added to the filter, the corresponding conflicting bit is set due to the first element's insertion and the query for the second element affirms the membership for the second element.

To reduce the false-positive rate, multiple hash functions are used. Thus, an element is inserted by setting multiple bits in the filter according to different hash functions. An element is in the set (represented by the bloom filter) if all the bits at the positions computed by different hash functions are set. Even if one of the bits is reset, it indicates that the element was not inserted. The false-positive rate usually reduces exponentially with the number of hash functions. Note, however, that there is no possibility of a false negative in this basic bloom filter, because no bit is ever reset.

To support deletion, counting bloom filters [Fan et al. 2000] have been introduced, which maintain a fixed-sized counter per element and increment and decrement the counter on insertion and deletion, respectively. As can be easily guessed, counting bloom filters may suffer from false negatives when the fixed-sized counter overflows. In a flow-sensitive analysis, since dataflow facts may get generated and killed arbitrarily, and we cannot a priori estimate a practical bound on the number of their insertions and deletions, a counting bloom filter cannot be used for our purpose. Maintaining a large number of bits per counter is a possibility, but it takes away the benefits of the reduced storage.

Bloom filters have been extended in various application domains [Mackert and Lohman 1986; Manber and Wu 1994]. For our purpose, we use *multibloom*, a multidimensional bloom filter designed for flow-insensitive analysis [Nasre et al. 2009]. The basic idea of a multibloom is to divide the filter space into multiple orthogonal dimensions, for pointers, contexts, hash functions, and pointees (objects that are pointed by pointers). A points-to fact is then represented by setting a bit characterized in all these dimensions: For a pointer, in a calling context, using a hash function, for a pointee. Since a multibloom was designed for a flow-insensitive analysis, it does not support element deletion and the question of its applicability for a flow-sensitive analysis was left open.

3. APPROXIMATE ANALYSIS USING MULTIBLOOM

Given a C/C++ program in an appropriate intermediate form (e.g., partial SSA form in LLVM), the goal of our analysis is to compute approximate flow-sensitive points-to information for the program with theoretically bounded imprecision and practically affordable precision. Initially, only the top-level variables (i.e., the variables whose addresses are never taken) are assumed to be in SSA form. As the analysis progresses and finds more and more points-to information, the partial SSA form is strengthened to add definitions of other variables (for stores). This process is repeated until no more points-to information can be computed, indicating a fixed point.

As mentioned earlier, an issue with using a basic bloom filter is that it is suitable only for membership queries of the form whether p points to q. But it is ill-suited for queries of the form "what is the points-to information of p?" or "copy points-to information of p

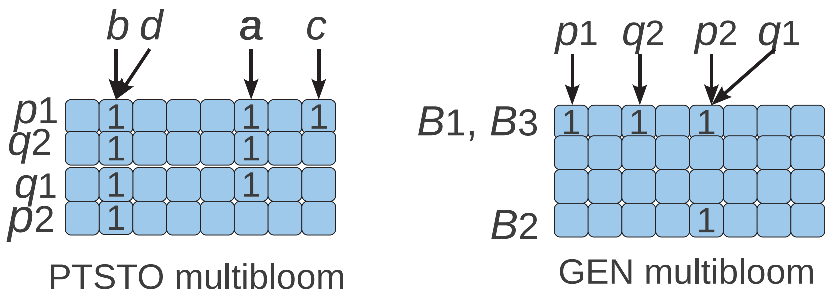

Fig. 2. PTSTO and GEN multiblooms for Example 3.1.

to q." This issue is addressed using a *multibloom* [Nasre et al. 2009], which bucketizes the bloom filter bits assigned to each pointer and supports processing of the pointer statements. For instance, Figure 2 shows two multiblooms. Each multibloom is a two-dimensional matrix of bits (instead of a traditional one- dimensional bloom filter)—each row, called a bucket, corresponds to pointers and columns correspond to pointees. Due to weak typing of C, the set of pointers and the set of pointees are identical. Note that the number of rows can be different from the number of pointers (this is true in general but in our implementation the two numbers coincide, as discussed later) and the number of columns can be different from the number of pointees. A points-to fact $p_1 \rightarrow b$ is stored in a multibloom by hashing p_1 and b to their row and column, respectively, and setting the corresponding bit. For instance, $hash\_ptr(p_1) = 0$ and $hash\_object(b) = 1$, so the points-to fact $p_1 \rightarrow b$ is stored by setting the bit PTSTO[0][1]. A copy operation $p = q$ can then be implemented by performing a bitwise-OR of q's bucket onto p.

A flow-sensitive points-to analysis demands *killing* a dataflow fact (refer to Equation (1)), which is not supported by a multibloom. We overcome this difficulty (partially) by exploiting a crucial property of a flow-sensitive points-to analysis.

PROPERTY 1. *In SSA form, flow-sensitive points-to information at each basic block of a CFG increases monotonically; that is, if a pointer points to a variable at a basic block, it continues to point to it at that basic block.*

Thus, in a flow-sensitive points-to analysis, a KILL operation does not affect the monotonicity of points-to updates. This property has been observed before [Hardekopf and Lin 2011] and has been used to prove correctness. In our case, we exploit the property to improve synchronization (see Section 4) and to reuse the existing framework of flow-insensitive analysis, namely, multibloom. We explain how a KILL operation is modeled using multibloom in Section 3.2.

Example 3.1. Let us assume that at any point during the analysis, the points-to information obtained is $\texttt{PTSTO}(p_1) = \{a, c, d\}$, $\texttt{PTSTO}(p_2) = \{d\}$, $\texttt{PTSTO}(q_1) = \{a, b\}$, and $\texttt{PTSTO}(q_2) = \{a, d\}$. In addition, let $\texttt{GEN}(B_1) = \{p_1\}$, $\texttt{GEN}(B_2) = \{p_2, q_1\}$, and $\texttt{GEN}(B_3) = \{q_1, q_2\}$. This information is stored in multiblooms as shown in Figure 2. Note that variables b and d hash to the same offset in PTSTO. Similarly, basic blocks B_1 and B_3 hash to the same bucket in GEN, and pointers p_2 and q_1 hash to the same offset. This constitutes false positives. For instance, $q_1 \rightarrow d$ is a spurious points-to fact stored in multibloom due to hash conflicts. Typically, only a few conflicts lead to real false positives; the rest are not queried by clients. We illustrate later in this section how to improve precision lost due to false positives.

A flow-sensitive analysis needs to maintain two kinds of data: def-use information and points-to information. In our experience, the amount of def-use information is often comparable to that of the points-to information. Therefore, we store both kinds of information in (different) multiblooms.

3.1. Using Multiblooms

Our implementation maintains five multiblooms, one each for GEN, KILL, IN, and OUT, and one for storing the points-to information. The def-use-related four multiblooms are specified as a 5-tuple $<N_B, N_H, N_A, H_B, H>$ where N_B is the number of buckets for basic blocks (for the four dataflow multiblooms) or pointers (for PTSTO multibloom) hashed using the family of hash functions H_B, N_H is the number of arrays assigned to each basic block that are hashed using the same number of hash functions from family H, and N_A is the size of each array in bits. For instance, a multibloom configuration $<10K, 2, 40, \{f\}, \{f_1, f_2\}>$ uses a function f to map a basic block or a pointer to an entry, has two hash functions f_1 and f_2 to hash the dataflow variables to appropriate columns (bits), and stores 40 bits in each array. The total size of one multibloom is $N_B \times N_H \times N_A = 10K \times 2 \times 40 = 80K$ bits or 10K bytes. The total size thus allotted for the dataflow information is 40K bytes. An additional space is required for another multibloom to store points-to information, specified similarly, but with a possibly different configuration. In GEN and the other three multiblooms, the rows correspond to basic blocks and the columns indicate dataflow variables. For points-to multibloom, the rows correspond to pointers and the columns indicate address-taken variables. In our implementation, we treat pointers and objects in a uniform fashion, unlike in strongly typed languages such as Java wherein pointers (called references) and objects are clearly demarcated [De and De'Souza 2012]. Thus, in Java, an object cannot act as a reference.

In the sequel, we describe how these flow-sensitive multiblooms are used to update the dataflow facts and the points-to information.

3.2. Updating Dataflow Information Using Multibloom

Recipes for updating the dataflow information are presented in Algorithms 1 through 4. These procedures are invoked repeatedly by the underlying pointer analysis (explained in Section 3.3).

Algorithm 1 is invoked to update the GEN information for a basic block B when a variable x is generated. Both B and x are hashed using appropriate hash functions ($hash_f \in H_B$ and $f_h \in H$, respectively). For each hash function in set H, the corresponding bit in the bucket for x is set.

ALGORITHM 1: Updating GEN multibloom

```
Require: Basic block $B$ wherein variable $x$ is generated.
1: $entry_B = hash_f(B)$
2: for $h = 0$ to $N_H - 1$ do
3:     $bit_x = f_h(x)$
4:     $old = GEN[entry_B][h][bit_x]$
5:     if $old == 0$ then
6:         $GEN[entry_B][h][bit_x] = 1$
7:         $retval = $ true
8: return $retval$
```

Updating KILL multibloom requires resetting the corresponding bits. Due to false positives, resetting a bit may lead to unsound analysis. For instance, if two points-to facts $a \rightarrow x$ and $b \rightarrow y$ point to the same bit in a bloom filter, then killing one fact also kills another, which is incorrect. Using more hash functions reduces this possibility, but does not guarantee soundness. To solve this problem, we exploit the structure of multibloom and the behavior of pointer analysis. First, we assign a separate entry per pointer in the multibloom. This ensures that the points-to information of two different pointers never conflict with each other. Thus, killing a points-to fact $a \rightarrow x$ cannot remove any pointee of another pointer b. The only possibility is that it may kill another points-to fact $a \rightarrow y$ of the same pointer a. This is where we exploit the behavior of pointer analysis to solve this crucial problem. We make the following key observation.

PROPERTY 2. *A kill operation in a flow-sensitive points-to analysis kills all the points-to facts of a pointer or none at all.*

Property 2 implies that pointer analysis does not involve killing of a points-to fact in isolation; it involves killing *all* the facts of a pointer. Thus, one does not encounter a situation when $a \rightarrow x$ is killed but $a \rightarrow y$ is retained; either both are killed (strong update) or both are retained (weak update). Therefore, the bits corresponding to the pointees of a pointer must all be reset or left untouched during a KILL. With this observation, updating the KILL multibloom for a strong update is easy: we simply clear all the bits from all the hash buckets for the corresponding pointer (Algorithm 2). For a weak update, no bits are reset. Strong updates are performed on the destination pointer for copy and load statements, as well as for store statements $*p = q$ when *FIPTSTO*(p) is a singleton (*FIPTSTO* is bootstrapped flow-insensitive information as discussed in Section 3.3).

ALGORITHM 2: Updating KILL multibloom for a strong update

```
Require: Basic block $B$ wherein variable $x$ is generated.
1: $entry_B = hash_f(B)$
2: for $h = 0$ to $N_H - 1$ do
3:   for $bit = 0$ to $N_A - 1$ do
4:     $KILL[entry_B][h][bit] = 0$
```

The procedure for updating OUT multibloom is slightly complicated and is shown in Algorithm 3. The procedure takes two parameters: the basic block B whose OUT information is to be updated and a flag indicating if it is a strong or a weak update. If the update is weak, then no information is killed (Line 6). But if the update is strong, then the information from the KILL multibloom is used to update OUT (Line 8). This process is repeated for each hash function (`for` loop at Line 2) and for each bit (`for` loop at Line 3). Note that the dataflow equation (Equation (1)) is being implemented using fast bitwise functions, which is critical for high performance of the multibloom. Here, | is a bitwise-OR operator, & is a bitwise-AND operator, and $a \mid= b$ means $a = a|b$.

ALGORITHM 3: Updating OUT multibloom

```
Require: Basic block $B$ for which $OUT_B$ is to be updated, flag $\in$ {strong, weak}.
1: $entry_B = hash_f(B)$
2: for $h = 0$ to $N_H - 1$ do
3:   for $bit = 0$ to $N_A - 1$ do
4:     $OUT[entry_B][h][bit] \mid= GEN[entry_B][h][bit]$
5:     if flag == weak then
6:       $OUT[entry_B][h][bit] \mid= IN[entry_B][h][bit]$
7:     else
8:       $OUT[entry_B][h][bit] \mid= (IN[entry_B][h][bit]$ & $!KILL[entry_B][h][bit])$
```

Algorithm 4 shows the pseudocode for updating the IN multibloom. It computes a set-union of the incoming OUT information for a basic block using the bitwise-OR operator.

ALGORITHM 4: Updating IN multibloom

```
Require: Basic block $B$ for which $IN_B$ is to be updated.
1: $entry_B = hash_f(B)$
2: for all $P \in predecessors(B)$ do
3:   $entry_P = hash_f(P)$
4:   for $h = 0$ to $N_H - 1$ do
5:     for $bit = 0$ to $N_A - 1$ do
6:       $IN[entry_B][h][bit] \mid= OUT[entry_P][h][bit]$
```

3.3. Flow-Sensitive Points-to Analysis Using Multibloom

We now explain how the flow-sensitive multibloom is used to compute the points-to information. The analysis supports the following forms of statements: address-of (`p = &q`), copy (`p = q`), load (`p = *q`), store (`*p = q`), and malloc (`p = malloc(...)`). A call-site-based malloc statement is converted to an address-of statement by introducing a new variable. A statement with a different form (e.g., multiple indirections `**p`) is normalized to one of the five forms by introducing suitable temporaries. In addition, SSA-ϕ functions are modeled using set-union, which is implemented using bitwise-OR of the corresponding bloom filter buckets.

Algorithm 5 shows how an address-of statement is analyzed. For a statement $p = \&q$, the variable q needs to be inserted into the points-to information of p. Since our flow-sensitive multibloom does not explicitly represent sets, we set the bit corresponding to the hashed value of q (Lines 3–4). This procedure is repeated for all the hash buckets of p (Line 2). Since `malloc` is modeled as an address-of statement, memory allocation is analyzed using the same algorithm.

ALGORITHM 5: analyze_addressof / analyze_malloc

```
Require: Statement $p = \&q$
1: $entry_p = hash_f(p)$
2: for $h = 0$ to $N_H - 1$ do
3:   $bit_q = f_h(q)$
4:   $PTSTO[entry_p][h][bit_q] = 1$
```

Algorithm 6 analyzes the copy statement. When dealing with explicit sets, a copy statement $p = q$ can be modeled using a wrapper over Algorithm 5 while enumerating over q's points-to set. However, since this set is only implicitly maintained in the multibloom, we cannot use this method. In Algorithm 6, the analysis performs a union of the buckets of the two pointers. This is done by iterating over all the bits of q and computing a bitwise-OR with the corresponding bits of p.

ALGORITHM 6: analyze_copy

```
Require: Statement $p = q$, multibloom $MB$   // MB could be PTSTO or IN
1: $entry_p = hash_f(p)$
2: $entry_q = hash_f(q)$
3: for $h = 0$ to $N_H - 1$ do
4:   for $bit = 0$ to $N_A - 1$ do
5:     $MB[entry_p][h][bit] \mid= MB[entry_q][h][bit]$
```

Algorithm 7 analyzes the load statement $p = *q$. Similar to the copy statement, the points-to information of q cannot be enumerated. The original multibloom implementation [Nasre et al. 2009] addresses this issue by keeping multilevel pointers (*ptr, *ptr,*

ALGORITHM 7: analyze_load

```
Require: Statement $p = *q$
1: $entry_p = hash_f(p)$
2: $entry_q = hash_f(q)$
3: for all $r \in FIPTSTO(q)$ do
4:   if $q$ points to $r$  // Algorithm 10 then
5:     analyze_copy($p = r, PTSTO$)  // Algorithm 6
```

***ptr, ...*) as separate dimensions—one for each dereferencing level. In contrast, we solve this issue by taking advantage of the information precomputed by a bootstrapped flow-insensitive analysis. We bootstrap our flow-sensitive analysis by Andersen's flow-insensitive context-insensitive analysis [Andersen 1994] and use this precomputed information (denoted as *FIPTSTO*) to generate points-to queries for copying the points-to information of the dereferenced pointer (i.e., each pointee in $*q$ in this case). For each such pointee, `analyze_copy` is invoked (Line 5). Note that *FIPTSTO* is *exact*; that is, no approximations are added to its representation, since its storage requirement is comparatively very small. Bootstrapping allows us to avoid the imprecision added by the additional dimensions [Nasre et al. 2009], although using a flow-insensitive analysis also adds some imprecision. For instance, if a hypothetical, precise points-to analysis computes $q \rightarrow \{a, b\}$ and *FIPTSTO* computes $q \rightarrow \{a, b, c, d\}$, then while analyzing the load statement $p = *q$, additional points-to facts $q \rightarrow \{c, d\}$ would be checked against the flow-sensitive multibloom. If the multibloom stores $q \rightarrow \{a, b, c, e\}$, then the spurious fact $q \rightarrow c$ is used to copy c's points-to information to p. In contrast, if we use some other bootstrapping analysis with a better precision, it may compute $q \rightarrow \{a, b, d\}$ and the spurious fact $q \rightarrow c$ would not be used. In general, in an exact analysis (e.g., [Hardekopf and Lin 2011]), using a particular less precise bootstrapped analysis is a matter of efficiency and does not alter the precision of the flow-sensitive analysis. But in an approximate analysis with bloom filter, using a more precise bootstrapped analysis may improve the overall precision. We use *FIPTSTO* with reliance on the fact that, in practice, higher-level pointers are less often address-taken variables and hence add less imprecision due to dereferencing compared to that added by the variables of primitive types. Since loads and stores dereference a higher-level pointer, using flow-insensitive information does not adversely affect precision. Although we focus on weakly typed C programs, most programs in practice adhere to the strict typing rules imposed by the programming language, reducing the possibility of type-casts and points-to relationships across widely separated dereferencing levels (e.g., in most programs, a pointer of type *int *** usually points to variables of type *int ** and not to those of type *int* or *struct node*). Note, however, that our algorithm does not expect the program to be strongly typed.

We note that bootstrapping is not essential for the correctness of our flow-sensitive bloom-filter- based analysis. In the absence of the auxiliary information, it is possible to iterate through all the program objects to generate points-to queries. Of course, such a processing is inefficient. Hence, we utilize bootstrapping for improved performance as well as precision. It is not mandatory to bootstrap with only a flow-insensitive analysis. The only requirement, for soundness, is that the bootstrapping analysis computes a superset of the points-to information computed by the flow-sensitive analysis.

Algorithm 8 analyzes the store statement $*p = q$. It also makes use of the precomputed flow-insensitive points-to information for iterating over the points-to set of p, invoking `analyze_copy` for each dereferenced pointee (Line 9). In addition, the analysis needs to take care of weak and strong updates. If p points to a single pointee, then it models a strong update; otherwise, it is a weak update (Lines 3–6). Unfortunately,

ALGORITHM 8: analyze_store

```
Require: Statement $*p = q$ in block $B$
 1: $entry_p = hash_f(p)$
 2: $entry_q = hash_f(q)$
 3: if $FIPTSTO(p)$ is singleton then
 4:     update_OUT($B$, strong)  // Algorithm 3
 5: else
 6:     update_OUT($B$, weak)  // Algorithm 3
 7: for all $r \in FIPTSTO(p)$ do
 8:     if $p$ points to $r$  // Algorithm 10 then
 9:         analyze_copy($r = q, PTSTO$)  // Algorithm 6
10: update IN multibloom  // Algorithm 4
```

checking if a points-to set is singleton cannot be done in a naïve multibloom, since even if a single bit in a bucket is set, it could be an artifact of hash collision. One way to circumvent this issue is by maintaining a counter with each pointer p as an indicator of p's points-to set size. However, this counter quickly becomes useless as points-to information from other pointers is copied to p's buckets. For instance, at an analysis point, let the points-to set sizes of p and q be 1 and 1, respectively. Let the analysis now process a copy statement $p = q$. This would involve copying the bit buckets of q to those of p. What should be the new counter of p? Since a multibloom does not maintain names of each pointee (due to which it gains large memory savings), we cannot find out the *common set of pointees* between p and q. In particular, the analysis fails to find out if both the pointers point to the same variable, in which case, p's points-to set would have continued to remain singleton. As a consequence of this inability, the approximate analysis is forced to increase p's counter pessimistically to $1 + 1$. Over the analysis, this pessimism quickly gets compounded and the counters of most pointers would be >1. Therefore, to address this issue, we rely again on the exact flow-insensitive points-to information computed by the bootstrapped analysis. If the points-to set of p is a singleton in *FIPTSTO*, the analysis performs a strong update.

Finally, Algorithm 9 analyzes the SSA-ϕ statement. It simply merges the IN information from all the incoming basic blocks and uses the procedure for *analyze_copy* statement as a subroutine.

ALGORITHM 9: analyze_ϕ

```
Require: Statement $\phi$ in block $B$
 1: for all $P \in predecessors(B)$ do
 2:     analyze_copy($B = P, IN$)  // Algorithm 6
```

3.4. Querying Points-to Information Using Multibloom

Clients query the computed points-to information in two ways: to check if two pointers alias and to check if a pointer points to an object. We explain how these queries are answered using our flow-sensitive multibloom, despite not storing the points-to information explicitly.

Algorithm 10 answers a points-to query "does pointer p point to pointee x?" This is checked by retrieving the bit corresponding to the hashed value of x. If the bit is set (due to x or due to a false positive), then p is considered to point to x, otherwise not. This is where the power of multiple hash functions can be seen. With multiple hash functions, multiple bits of the multibloom, one corresponding to each hash function, are checked. If *any* of these bits is reset, then it is guaranteed that x was not inserted (or inserted but killed) in the multibloom. This enables the multibloom to answer the points-to query in negative (Line 5). Otherwise, the query is answered in positive.

ALGORITHM 10: Querying for points-to relationship

```
Require: pointer $p$, pointee $x$
1: $entry_p = hash_f(p)$
2: for $h = 0$ to $N_H - 1$ do
3:     $bit_x = f_h(x)$
4:     if $PTSTO[entry_p][h][bit_x] == 0$ then
5:         return false
6: return true
```

Example 3.2. Consider a multibloom with four-bit buckets and two hash functions h_1 and h_2 for pointees and one for the pointers called *hash*. Let $h_1(x) = h_1(y) = 1$ while $h_2(x) = 0$ and $h_2(y) = 3$. Also, let $hash(p) = 0$. Now, a points-to fact $p \to x$ is inserted by setting $\mathrm{MB}[hash(p)][h_1(x)] = \mathrm{MB}[0][1]$ and $\mathrm{MB}[hash(p)][h_2(x)] = \mathrm{MB}[0][0]$ in the multibloom. To answer a query "does p point to y?", Algorithm 10 would check bits $\mathrm{MB}[hash(p)][h_1(y)] = \mathrm{MB}[0][1]$ and $\mathrm{MB}[hash(p)][h_2(y)] = \mathrm{MB}[0][3]$. However, MB[0][3] is not set, indicating that p does not point to y. This is possible due to multiple hash functions; using only h_1 would result in a false positive.

Algorithm 11 answers an alias query "does pointer p alias with pointer q?" Two pointers alias if they have a common pointee. In multibloom terms, two pointers alias if they have the same bit in corresponding buckets set. Here, too, multiple hash functions reduce the probability of false positives. If N_H is the number of hash functions, then each bucket corresponding to a hash function must have at least one common bit set for the two pointers. Thus, the number of common bits must be at least N_H (Line 8); the number of common bits can be greater than N_H due to hash conflicts.

ALGORITHM 11: Querying for alias relationship

```
Require: pointer $p$, pointer $q$
1: $entry_p = hash_f(p)$
2: $entry_q = hash_f(q)$
3: $count = 0$
4: for $h = 0$ to $N_H - 1$ do
5:     for $bit = 0$ to $N_A$ do
6:         if $PTSTO[entry_p][h][bit]$ & $PTSTO[entry_q][h][bit]$ then
7:             $++count$
8: return $count >= N_H$
```

3.5. Improving Precision

A bloom filter is a probabilistic data structure and is bound to suffer from false positives. A false positive occurs when a points-to fact not added to the bloom filter is answered to be present in the filter. The presence of false positives compels a client to make conservative decisions during code optimization. For a bloom filter of size N bits and with N_h hash functions, the false-positive rate P, after n elements are inserted in the bloom filter, is given by the following equation [Bloom 1970]:

$$P = \frac{(1/2)^{N_h}}{\left(1 - \frac{n*N_h}{N}\right)}. \tag{2}$$

We use several techniques to minimize precision loss.

—We use multiple hash functions. Note that for fixed values of n and N, P reduces exponentially with increasing the number of hash functions N_h. Therefore, each additional hash function significantly reduces the false-positive rate. For instance, for $N = 2{,}048$ and $n = 256$, the false- positive rate P is 0.57 for a single hash function, which reduces to 0.33 for two hash functions and to 0.12 for four hash functions. Note that each hash function increases the space requirements linearly.

—The analysis assigns a separate entry per pointer or basic block; that is, the hash function H_B is an identity mapping from a pointer to its bucket. As discussed in Section 3.2, this is necessary to ensure sound processing of KILL statements. However, this technique also improves precision by completely eliminating conflicts due to hashing pointers and by avoiding polluting the points-to information by pointers with a large points-to set (most bucket bits for such pointers are set to 1).

—We bootstrap our analysis with a flow-insensitive points-to analysis. This helps us avoid imprecision in case of load and store statements due to multiple dereference dimensions proposed in flow-insensitive multibloom [Nasre et al. 2009], as discussed in Section 3.3.

—We intersect FS points-to information with FI points-to information. It is well known that the points-to information computed for each pointer by a flow-sensitive analysis is a subset of that computed by a flow-insensitive analysis. However, due to approximation, our analysis may compute a superset of the flow-sensitive information but not strictly a subset of flow-insensitive information. We take advantage of bootstrapping to improve the analysis precision by *intersecting* the computed approximate points-to information with the flow-insensitive points-to information. This is done by storing *FIPTSTO* in a multibloom and then performing a bitwise-AND of each pair of corresponding buckets. Therefore, the final points-to sets are guaranteed to be subsets of flow-insensitive points-to sets. We have observed that this technique proves quite valuable in practice, especially in situations when a pointer involves many manipulations.

4. POINTS-TO ANALYSIS ON GPU

The approximate flow-sensitive points-to analysis on the GPU is outlined in Figure 3. The host runs the `main` function, which starts with reading the input program, creating a Control-Flow Graph (CFG) out of it, and transferring the CFG to the GPU's main memory (Lines 2–4). The host code also calls the initialization kernel to zero out all the bits of the multiblooms (Line 5). It then calls another kernel, which computes the flow-insensitive inclusion-based points-to information *FIPTSTO* (Line 6), which is used for bootstrapping and approximating during the flow-sensitive analysis later. The flow-insensitive version is exact (i.e., it does not add any approximations to the points-to information representation) and uses sparse bit vectors to store points-to information.

The host code then repeatedly calls the main analysis kernel, which uses the control-flow graph and the points-to constraints to compute flow-sensitive points-to information, as described in Section 3.3. Each thread works on a statement to update its dataflow and points-to information, as shown in the processing kernel `solvekernel`. The kernel calls the corresponding algorithm (Algorithms 5–8) based on the statement type (Lines 19–26). In each iteration, a flag *changed* is transferred to the host to indicate if another iteration is necessary (Line 11). At the fixed point, the computed *PTSTO* multibloom is transferred to the host from the GPU device (Line 14). The computed points-to information can be queried for aliasing as discussed in Section 3.4.

4.1. Parallelization

The analysis is implemented in CUDA in a data-parallel manner by assigning threads to various program statements to be analyzed. We exploit massive multithreading on the GPUs to invoke the kernel `solvekernel` in a topology-driven manner; that is, each statement is considered to be *active* in each kernel invocation. On a CPU [Hardekopf and Lin 2011], a data-driven approach is preferred, which maintains an explicit worklist of active statements (or nodes in the constraint graph). A data-driven

```
main(P) {
  read input program P
  create CFG
  transfer CFG                                  // cpu → gpu
  initialize multiblooms                        // gpu
  compute FI exact points-to                    // gpu, Andersen's analysis

  do {
    reset changed                               // gpu
    solvekernel                                 // gpu
    transfer changed                            // cpu ← gpu
  } while changed

  transfer PTSTO                                // cpu ← gpu
}

solvekernel {
  stmt = getStatement(thread-id)                // over each statement
  case stmt of
    $p = malloc(...)$: analyze_malloc(stmt)     // Algorithm 5
    $p = \&q$:         analyze_addressof(stmt)  // Algorithm 5
    $p = q$:           analyze_copy(stmt)       // Algorithm 6
    $p = *q$:          analyze_load(stmt)       // Algorithm 7
    $*p = q$:          analyze_store(stmt)      // Algorithm 8
    $\phi$:            analyze_phi(stmt)        // Algorithm 9
}
```

Fig. 3. Points-to analysis on GPU.

approach is relatively more work-efficient. However, by employing a topology-driven GPU implementation, the analysis avoids the overheads of maintaining a centralized worklist.

Synchronization. Flow-sensitive multibloom is stored as an array of words in the GPU's global memory. Each word stores 32 bits of information. Write operations on multibloom are implemented in a lock-free manner by using atomic instructions, whereas read operations are synchronization free. In general, absence of synchronization may lead to data races; but in our case, since the multibloom information monotonically increases, the data race does not affect correctness. At the worst, a read operation may see stale points-to information and propagate it across pointers during the analysis. However, since the algorithm continues until a fixed point (determined using the *changed* flag in Algorithm 3), a read is guaranteed to see the updated information *sometime* in future.

Locality. One of the compelling advantages of our multibloom is its improved spatial locality. Since a multibloom has a static user-configurable size (unlike dynamically growing lists or graphs), it is allocated prior to the analysis in a contiguous chunk of memory. Further, buckets for a pointer corresponding to multiple hash functions are stored in contiguous memory since all our algorithms access each of the hash buckets for a pointer. We take advantage of the shared memory (on-chip cache) at the basic blocks to buffer parts of the multibloom. Thus, each thread processing a copy statement $p = q$ copies each hash bucket of p into shared memory and then performs a bitwise-OR with q's corresponding hash bucket, which is stored in global memory. After all the bucket unions, the buckets in shared memory are written through to p's buckets in global memory. This greatly improves access latency for p's buckets. We target the copy statement (*analyze_copy* routine) for this purpose since it is used by most other routines (like load, store, and phi).

Due to limited shared memory size available at the thread block (48KB on NVIDIA GPUs), exploiting locality in this manner puts some limits on the bucket size and the number of hash functions. For instance, if the number of threads per block is 1,024, then 12 words of shared memory are available per thread. If the bucket size is set to 64 bits (two words), then the analysis can use a maximum $12 / 2 = 6$ hash functions (which suffices for moderately sized programs). The number of hash functions can be increased by either reducing the bucket size or by reducing the number of threads per block. For instance, by reducing the number of threads to 256, 12 hash functions with bucket size of four words can be configured. However, reducing the number of threads reduces the amount of parallelism, hampering performance. Therefore, we split the bucket copying to process hash buckets one by one (instead of all of them at once). Thus, *analyze_copy* copies the first hash bucket of p in shared memory, performs a bitwise-OR with q's first bucket (from global memory), writes through p's bucket back to global memory, and then brings p's next bucket into shared memory. This process splitting is valid because individual hash functions of a multibloom are independent. It makes the number of hash functions N_H independent of the shared memory size. In our example, with 1,024 threads per block, all 12 words of per-thread shared memory are available for a bucket and the number of hash functions can be set to an arbitrarily large value.

Reducing Thread Divergence. GPU threads are divided into SIMD lanes called warps or wavefronts. All threads in a warp must execute in lock-step. If warp threads evaluate branches differently and execute different instructions, then the hardware divides threads into chunks, with each chunk executing the same instruction. These chunks are executed sequentially until they all converge again. This phenomenon is called thread divergence. It reduces parallelism and, thus, performance. For the best execution time, thread divergence must be avoided. If points-to statements are assigned to threads, then different warp threads may execute a different number of instructions, leading to thread divergence. For instance, a load statement is more expensive to analyze than a copy statement, which, in turn, is more expensive than an address-of statement. To reduce thread divergence, we divide the statements into groups based on the statement type: address-of/malloc, copy, load/store/phi. These groups are then laid out contiguously to threads. This assigns less expensive copy statements to consecutive warp threads and also more expensive load/store/phi statements to another set of consecutive warp threads. Thus, except at the boundaries of these groups, warp threads are almost always synchronized, reducing thread divergence. Address-of and malloc statements need to be executed only once prior to the analysis, since they do not affect points-to information further.

Load Balance. Despite reducing thread divergence, our analysis suffers from load imbalance. To understand why this happens, consider two warps, one analyzing copy statements and another analyzing load statements. Due to different amounts of information processed, there is a huge imbalance among the two tasks. Load imbalance may lead to poor work efficiency and reduced resource utilization, ultimately affecting performance. We address this issue with the observation that the more expensive operations (load/store/phi statements) are wrappers over the less expensive operation (copy). Therefore, it is possible to perform task assignment using the primitive copy operation. To achieve this, we maintain a dynamically growing task array, which keeps track of all the copy operations to be performed in the next iteration of the analysis.

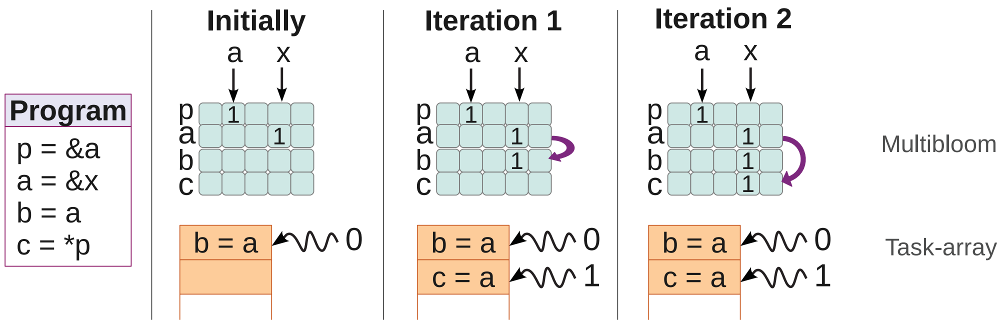

Fig. 4. Task array allocation (see Example 4.1).

A load/store/phi operation adds multiple tasks to this task array. Threads are then assigned to these tasks instead of individual statements as we assumed so far. The task assignment is done based on the thread id; that is, a thread with id i handles the copy operation at entry i in the task array. Thus, the analysis now involves two synchronous steps. In the first step, threads are assigned to load/store/phi statements, which push copy operations to the task array. In the second step, threads are assigned to each copy task in the task array. At the beginning of the analysis, all copy statements are added to the task array. The task array is implemented as two fixed-sized associative arrays: one for the source and another for the target pointer of the copy operation. The task array supports concurrent task pushing by atomic increments of a single integer maintained, representing the array size. Use of a task array achieves perfect load balancing in the second step of execution, since each copy operation involves executing congruent sets of instructions by different threads (on different buckets).

Example 4.1. Consider a program with the following statements: $p = \&a; a = \&x; b = a; c = *p$, as shown in Figure 4. Here, the address-of statements are analyzed only once and the points-to information is updated as $p \rightarrow a, a \rightarrow x$. The task array is initialized with all the copy statements, in this case $b = a$. This statement is analyzed in the first iteration (by a single thread in this example), updating b's points-to information. Prior to the second iteration, the load statement $c = *p$ is processed to add the copy statement $c = a$ into the task array. Thus, the task array now contains two copy statements, $b = a, c = a$. Both these statements are analyzed in parallel in the second iteration. Note that the former copy statements (added in previous iterations) need to be analyzed again in the future iterations because the points-to information of the source pointer may have changed.

Memory Coalescing. The GPU's memory subsystem is optimized for warp-based processing. If the 32 warp threads access 32 contiguous words, then the hardware combines those memory accesses and issues a single memory request to the memory controller. This phenomenon is called *memory coalescing* and is critical for high performance. If warp threads perform uncoalesced memory accesses, then the hardware has to make multiple memory requests to fetch the required data. While it is difficult to perform coalesced memory accesses when points-to information is stored as lists or sparse bit vectors, flow-sensitive multiblooms can be accessed in an efficient manner on a GPU due to their regular structure. We store hash buckets of pointers in a column-major order instead of the row-major order. Thus, if two warp threads are assigned to consecutive pointers, their copy operations are fully coalesced. Further, corresponding hash buckets of consecutive pointers are interleaved for coalesced accesses. Thus, the first hash bucket of pointer p_1 is stored, followed by the first hash bucket of pointer p_2 (in column-major order), and so on. However, a column-major order alone does not guarantee coalesced accesses; consecutive pointers should also be assigned to consecutive warp threads. This is difficult because each thread operates on a copy task and the tasks are arbitrarily assigned to threads, without any exploitable ordering between them.

We solve this issue by sorting the task array before assigning it to threads. The pair of identifiers corresponding to the target and source pointers in the copy operations acts as the composite sorting key. Thus, all copy operations are sorted on the target pointer's identifier, and for the same target pointer, operations are sorted on the source pointer's identifier. Sorting ensures that copy operations on pointers with consecutive identifiers appear consecutively in the task array and hence get assigned to consecutive warp threads. This, combined with the column-major storage of the multibloom, helps in improving memory coalescing, resulting in improved throughput.

Sorting can be efficiently implemented on GPUs using a barrier-based divide-and-conquer way [Satish et al. 2009, 2010]. Still, one may argue that repeated sorting (once per iteration) has a high cost—and this is indeed the case, as observed in our experiments. However, sorting also enables the following key optimizations, whose combined effect improves the overall performance. First, copy operations with the same target appear together and are (almost always) assigned to multiple threads in the same warp. While executing the copy operations, warp threads use CUDA-supported thread-voting functions and share the on-chip shared memory for efficiently copying buckets from different source pointers. Second, sorting helps ensure unique tasks in the task array. Thus, while adding a new task $p = q$, a simple binary search in the array suffices to check if the task is already present. Note that this is critical because multiblooms do not track pointees explicitly and hence have no way of checking if a particular pointee was already added. By avoiding duplicate tasks, we improve work efficiency and the storage requirement considerably, in turn improving performance. For instance, without a sorting phase (and, therefore, without removing any duplicates), *tshark* can be analyzed in 112.7 seconds, whereas with sorting of the task array, the analysis completes in 90.4 seconds despite a sorting overhead of 4.6 seconds (across all the analysis iterations).

5. EXPERIMENTAL EVALUATION

We perform our experiments on a 1.15GHz NVIDIA Tesla C2070 GPU with 6GB of main memory and 448 cores distributed over 14 Streaming Multiprocessors (SMs). This Fermi-based GPU has 64kB of fast memory per SM that is split between the L1 data cache and the shared memory. All SMs share an L2 cache of 768kB. We compiled the CUDA programs (see later) with *nvcc* version 4.1 and the *-O3 -arch=sm_20* flags.

To run the CPU code, we used a Nehalem-based machine running Ubuntu 10 with 12 quad-core 2GHz Xeon processors. The 48 CPU cores share 128GB of main memory. Each core has a 64kB L1 cache and a 256kB L2 cache. Each processor has a 16MB L3 cache that is shared among its four cores. Since the analysis is serial, we ran the code as a single-threaded application.

In the evaluation, we used six large open-source programs. Their characteristics (lines of source code, number of statements, and number of pointers) are given in Figure 5. The largest program `tshark` is ~2 million lines and has ~1.8 million points-to constraints. Most of these programs have been used by other researchers before [Hardekopf and Lin 2011, 2012].

We implemented our multibloom-based flow-sensitive points-to analysis in CUDA and LLVM [Lattner and Adve 2004] using the NVPTX backend, which transforms an LLVM bitcode to PTX assembly [NVPTX 2012]. The complete analysis takes place in two phases. In the first phase, the LLVM front-end parses the input C program and

	Characteristics			Time(s)	Speedup	Mem(MB)	Reduction(×)	Precision(%)	
B/M	KLOC	Stmt(K)	Vars(K)	sfs	bloomfs	sfs	bloomfs	sfs	bloomfs
tshark	1,946	1,789	1,952	624	6.9	6,284	3.7	100	98.6
linux	1,503	420	2,071	1,740	11.5	7,529	4.3	100	95.3
hphp	877	682	853	1,429	10.4	6,472	8.8	100	93.7
gimp	877	649	752	962	4.7	4,310	6.7	100	98.5
gdb	475	362	529	529	6.4	2,430	5.4	100	97.6
gs	438	489	639	1,583	9.1	4,494	8.2	100	96.4

Fig. 5. Benchmark characteristics and performance of the configuration Vars-8-160.

transforms it into LLVM intermediate representation, and the NVPTX backend transforms the IR into its PTX assembly. In the second phase, a CUDA compiler transforms the assembly into executable code and executes the analysis on the GPU. Initially, the LLVM front-end transforms top-level variables into SSA form. Our pointer analysis pass collects points-to constraints and allocates storage on the GPU. The storage is required for the constraints, the CFG, and the multiblooms. CFG is represented in a compressed sparse-row storage (CSR) format, with all edges stored in an array and each node maintaining an offset into the edges array at the start of its edge list [Nasre et al. 2013]. The communication phase transfers the CFG and the constraints to the GPU memory as discussed in Algorithm 3 and, in the end, transfers the computed points-to solution back to the CPU. The compute phase runs the kernel repeatedly on the GPU, which computes points-to information until a fixed point. We time both the compute phase and the communication phase and report all the global memory allocated on the GPU.

For implementing multiblooms, we used several additive and rotative string hashing algorithms [Partow 2013]. We experimented with several values for various constants (prime numbers used in hashing) and chose the ones that provided the best precision.

We compare our bloom-filter-based flow-sensitive points-to analysis, named `bloomfs`, with Hardekopf and Lin's sparse flow-sensitive exact points-to analysis, named `sfs` [Hardekopf and Lin 2011], downloaded from Hardekopf's website. Both analyses use the same LLVM framework. We ran each program thrice and report the minimum time for both `bloomfs` and `sfs`; the memory requirement and the precision remain fixed across runs for the same program. As an auxiliary bootstrapped analysis (which computes *FIPTSTO*), both the variants use Andersen's inclusion-based points-to analysis [Andersen 1994] but with different implementations: For `sfs` it is implemented in C++ (as part of the software downloaded from Hardekopf's website) and runs on the CPU, while for `bloomfs` it is implemented in CUDA and runs on the GPU [Nasre et al. 2013]. Both implementations compute the same flow-insensitive points-to information.

5.1. Analysis Performance

Figure 5 shows the effect on analysis time, memory requirement, and precision for the two flow-sensitive variants. The analysis time and memory for `sfs` are those reported by the executable, and we consider its precision as the base since it is an exact analysis; hence, its precision is set to 100%. `sfs` times allocation and initialization of data structures, optimizing constraints and the main solve routine, and reports all the memory used by the process by the end of the analysis. The analysis time for `bloomfs` covers steps 4–14 from Algorithm 3, which includes transfer of points-to constraints, multibloom initializations, computing initial flow-insensitive points-to sets, computing flow-sensitive points-to solution, and transforming the results back to the CPU. The reported memory includes that for the CFG, *FIPTSTO*, the points-to constraints, and the multiblooms. We used the `gettimeofday` function to measure time and noted the CPU memory usage from the `/proc` file system as done by `sfs`, while counting all the `cudaMalloc`ed memory for `bloomfs`.

With 100% precision, `bloomfs` must compute the same information as that computed by `sfs`. However, being an approximate analysis, `bloomfs` is less precise than an *exact* flow-sensitive analysis. Therefore, it is imperative to measure the precision loss due to approximations added by the multibloom-based analysis. Unfortunately, it is difficult to compute the precision for `bloomfs` relative to that of `sfs` as the former's points-to information is not stored explicitly. To address this issue, we rely on LLVM's alias analysis evaluator (*aa-eval*), which queries a points-to analysis pass with pairs of pointers from the program and calculates a percentage of queries that have been answered as "not alias." A higher value of the *NoAlias* percentage usually indicates better precision. In our case, since `bloomfs` computes a superset of the points-to information computed by `sfs`, *NoAlias* percentage is a faithful indicator of relative precision. We normalize the *NoAlias* percentage obtained with running `bloomfs` with respect to that obtained with running `sfs`.

Since the analysis runs in the second phase as a separate program, we faced the difficulty of how to use the analysis results in LLVM for alias queries (and for running other client analyses). We addressed this issue by creating a third phase in which the result of the second phase (the points-to multibloom) is read from a file on disk and is used to answer queries. Since the multibloom does not involve any pointer-based data structures, it is easy to dump to and reload from disk. The third phase invokes another alias analysis pass with the same input program (as in the first phase), which loads the PTSTO multibloom from the disk and answers queries from the alias analysis evaluation (*-aa-eval*) pass. We do not account for the time taken for the disk I/O in our timing measurements.

We experimented with several configurations for the multiblooms. The results in Figure 5 are reported for the configuration N_B-N_H-N_A = Vars-8-160. Thus, each pointer has a separate entry with 8 hash functions and 160 bits for each bucket (totaling Vars * 8 * (160/8) bytes per multibloom). From the analysis time results in Figure 5, we find that our GPU implementation can parallelize the analysis, well achieving up to 11.5× speedup (geomean 7.8×). This is possible because of several algorithmic and architectural optimizations discussed in Section 4 that are possible because of the regular structure of multibloom. Use of multibloom improves locality, reduces thread divergence and load imbalance, and enables coalesced memory accesses. Bloom filters have been primarily designed for space-time tradeoffs [Bloom 1970], and we observe the significant space savings from the memory results. We report the memory required by `sfs` and reduction factor achieved by `bloomfs`. `sfs` stores points-to information in binary decision diagrams (BDDs), while `bloomfs` uses multiblooms. Overall, approximating the points-to information using multiblooms reduces the storage requirement by a significant factor (geomean 5.9×, maximum 8.8× for `hphp`). This is not surprising since we store only a few bits per points-to pair (equal to the number of hash functions) compared to BDDs, which require bits logarithmic in the number of inserted elements. In addition, BDDs are irregular data structures, require frequent modifications to the underlying hierarchical structure, exhibit poor spatial locality, and pose challenges for parallelization. BDDs are well suited for storing large amounts of mostly static information (e.g., state diagrams or search spaces) but may not be the best choice for dynamically changing information such as points-to.

The last columns of Figure 5 show the imprecision due to false positives in the multibloom, in terms of *NoAlias* percentage, relative to that of `sfs`. We observe that it loses less than 5% precision compared to the exact analysis. The precision is an artifact of our design decisions about bootstrapping the analysis, multiple hash functions, and intersecting *FSPTSTO* and *FIPTSTO*, as discussed in Section 3.5. Precision can be improved further by changing the multibloom configuration, at the cost of additional analysis time and storage. For example, configuration Vars-16-160 improves precision

	Time(s)		Memory(MB)	
B/M	gpufi	bloomfs	gpufi	bloomfs
tshark	2.4	90.4	714	1,681
linux	11.6	151.3	760	1,737
hphp	15.7	137.4	250	732
gimp	5.9	204.7	225	646
gdb	3.3	82.7	175	452
gs	15.8	174.0	197	547

Fig. 6. Relative performance of flow-insensitive and flow-sensitive GPU versions (`bloomfs` configuration is Vars-8-160).

to within 2% of the exact analysis with a 4.8× speedup and a 3.1× memory reduction (see Section 5.3).

5.2. Comparison against Flow-Insensitive Analysis

To assess the relative performance of our parallel flow-sensitive analysis, we compare it with the state-of-the-art flow-insensitive GPU implementation of points-to analysis [Mendez-Lojo et al. 2012], referred to as `gpufi`. We emphasize that since the two versions execute different algorithms and compute different information, this comparison should only be used as a rough estimate of the performance difference. `gpufi` uses sparse bit vectors to represent the points-to information, whereas `bloomfs` uses multiblooms. `gpufi` is optimized for warp-based execution and the bit vector is customized for coalesced memory accesses. It assigns a single thread block to each SM, whereas our `bloomfs` implementation assigns multiple blocks to better hide the memory latency. `gpufi` is a standalone program and is not an LLVM pass. Therefore, we converted each benchmark's pointer-related statements into a format that `gpufi` expects. We executed both `gpufi` and `bloomfs` on the same GPU. The analysis times and the memory requirements for the two analyses are presented in Figure 6. From the results, we observe that there is an order of magnitude (15.4×) difference between the analysis times of the two versions. This is an artifact of the large number of dataflow and points-to information propagations required for a flow-sensitive analysis. We also report the memory requirement of `gpufi` versus `bloomfs`. `gpufi` uses a custom memory allocator, which preallocates 3,930MB of global memory on the GPU. Therefore, we modified the implementation to report only the used portion of this preallocated chunk. We observe that, on average, `gpufi` uses 2.5× less memory than `bloomfs` (maximum 760MB for *linux*).

While it is not the focus of this work, we also compared the precision of the flow-insensitive Andersen's analysis `fi` against that of the flow-sensitive analysis `sfs`. We found that the precision improvement due to flow sensitivity differs across benchmarks. On average, the *NoAlias* percentage for our benchmark suite is 82.6% for `fi`, whereas it is 91.4% for `sfs` (88.7% for `bloomfs`). The smallest benefit is obtained for `tshark` (84.9% for `fi` versus 85.1% for `sfs`), whereas flow sensitivity is most useful for `gs` (86.8% for `fi` versus 93.4% for `sfs`). Note that for both `fi` and `sfs`, the LLVM IR is in SSA form for top-level variables (non-address-taken variables).

5.3. Effect of Multibloom Configurations

Recall that a user can alter various configuration parameters of the multibloom: number of pointer entries N_B, number of hash functions N_H, and number of bits per bucket N_A. We experimented with several values for these tunable parameters: $N_B \in \{Vars\}$, $N_H \in \{4, 8, 12, 16\}$, and $N_A \in \{32, 64, 160, 320\}$. The overall effect for these 16 configurations is shown in Figure 7, sorted on average precision across all the benchmarks.

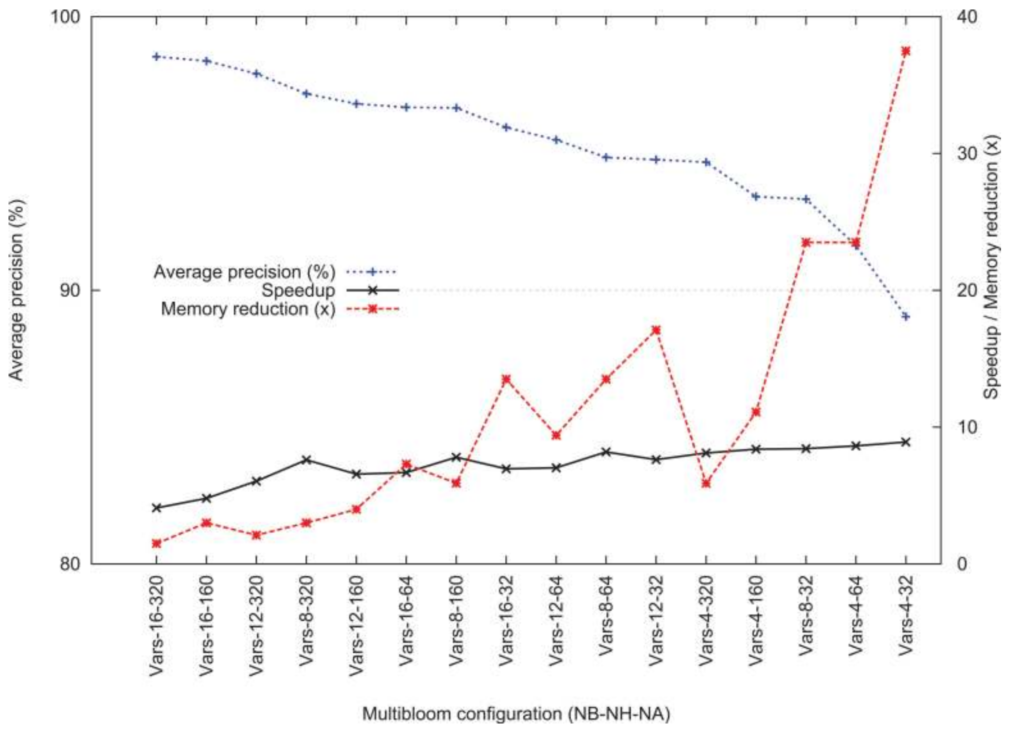

Fig. 7. Effect of various configurations on analysis time, precision, and memory requirement.

The left y-axis shows the average precision, while the right y-axis shows the average speedup achieved as well as the average memory reduction.

We observe that average precision improves considerably with increasing N_H and slowly with increasing N_B. Smaller multibloom configurations provide great memory savings (up to 37.5× reduction for $N_H = 4$) as well as good speedups (up to 8.9×). Such configurations are well suited in scenarios where precision is not the primary objective. For high-precision requirements, it is beneficial to increase N_H and N_B with moderate speedups (4–6×) and some memory savings (1–3×). However, the most interesting use case is to choose a configuration such that we obtain balanced benefits on all three fronts (see Figure 1). Several multibloom configurations achieve precision within 5% of that of an exact analysis with over 5× speedup and memory savings by a factor. For instance, configuration Vars-8-64 achieves 94.9% precision with 8.2× speedup and 13.5× reduced memory. Multibloom enables a client to choose an appropriate configuration to suit its needs.

Effect of Optimizations on Precision. We now discuss the effect of various optimizations to improve analysis precision (refer to Section 3.5). Depending on the program, one requires a different number of hash functions to achieve reasonable precision. For smaller programs (like 179.art, 183.equake from SPEC 2000), a few (two to eight) hash functions suffice [Nasre et al. 2009]. For larger programs, as in this work, the number of hash functions must be increased up to 16 to achieve good precision. For instance, increasing the number of hash functions from 4 to 16 improves the precision of the analysis from 89.0% to 96.0% (for $N_A = 32$). Intersecting *FIPTSTO* and *FSPTSTO* improves precision by another 3.4% on an average across all the benchmarks (maximum 5.7% for `hphp`).

	LICM		ADCE	
B/M	sfs	bloomfs	sfs	bloomfs
tshark	8,561	8,291 (97%)	40	38
linux	3,852	3,637 (94%)	11	11
hphp	6,490	6,046 (93%)	35	33
gimp	5,139	5,039 (98%)	33	28
gdb	4,072	3,903 (96%)	19	16
gs	6,392	6,080 (95%)	21	17

Fig. 8. Effect of precision on clients. For LICM, the results show the number of load instructions hoisted. For ADCE, the results show the number of basic blocks removed. The multibloom configuration evaluated is Vars-8-160.

5.4. Effect on Client Analyses

Although the *NoAlias* percentage provides some measure of precision, it is not a completely faithful indicator because the queries are simply $\binom{n}{2}$ combinations of pointers in the program. More importantly, precision of points-to analysis is better decided by the clients that use it. Therefore, in this section, we evaluate our multibloom-based flow-sensitive analysis using two clients: Loop-Invariant Code Motion (LICM) and Aggressive Dead Code Elimination (ADCE). Our goal is to check how the loss in precision due to multibloom's false positives affects real program transformations.

LICM uses alias information for promoting memory loads/stores to registers. The promotion is possible if there are no aliases to the loaded/stored memory location. ADCE is a transformation pass to remove redundant computations from the program. It uses the computed alias information to find out functions without any side effects and deletes calls to such functions.

We fed back the computed points-to information to our LLVM pass in the third phase as discussed in Section 5.1 and used that to answer queries from the clients. The results are shown in Figure 8 for our set of benchmarks for both the exact analysis on the CPU and the approximate analysis on the GPU for configuration Vars-8-160. For LICM, we show the number of load instructions promoted to registers, and for ADCE, we show the number of basic blocks identified as redundant. We observe that the precision obtained by our approximate analysis is close to that by the exact analysis. In particular, `bloomfs` promotes 95.6% loads compared to that by `sfs`. For the multibloom configuration Vars-16-160, `bloomfs` achieves a better precision of 96.8%.

`bloomfs` is relatively more imprecise for ADCE under configuration Vars-8-160; ADCE is able to remove only 89.6% basic blocks compared to the situation when it uses `sfs`. However, a salient feature of multibloom is its ability to trade off analysis time for better precision. With configurations Vars-16-160 and Vars-32-192, ADCE is able to remove 92.7% and 94.4% basic blocks, respectively.

These results show that our multibloom-based parallel implementation of flow-sensitive points-to analysis can be a practical solution for large C programs.

6. RELATED WORK

We now compare and contrast our work with earlier approaches. We divide the related work into three categories: parallel pointer analysis, sequential pointer analysis, and the use of bloom filters.

Parallel Pointer Analysis. Parallel points-to analysis is only moderately explored in literature. However, there is convincing evidence that pointer analysis can benefit by parallelization on the multicore and the many-core platforms.

Mendez-Lojo et al. [2010] propose the first parallel implementation of flow-insensitive inclusion-based points-to analysis by exploiting the constraint graph formulation. The work models points-to analysis using graph-rewrite rules and shows that graph formulation helps in easy exploitation of amorphous data parallelism available in irregular algorithms such as pointer analysis. This work, based on speculative execution on multicore systems, has been extended to GPUs by exploiting several architectural features [Mendez-Lojo et al. 2012]. In particular, the dynamic constraint graph is built using warp-size sparse bit vectors to reduce thread divergence and achieve memory coalescing. Further, propagation of points-to information is modeled using a *pull*-based mechanism (in contrast to a *push*-based mechanism) to ensure single-writer policy and, in turn, reduce synchronization. In comparison, we parallelize computationally more expensive flow-sensitive pointer analysis and model it using a multibloom instead of a graph. Our performance considerations on the GPU remain the same as those of Mendez-Lojo et al.; however, we reduce thread divergence by grouping the same kinds of points-to constraints, achieve memory coalescing using a sorted task array, and use lock-free computation to mitigate the synchronization cost.

Edvinsson et al. [2011] propose a parallel points-to analysis for object-oriented programs. The key parallelization insight used in that work is that different target methods of polymorphic function calls and independent control-flow branches can be analyzed in parallel. In comparison, our work targets C-like programs, although our techniques are applicable in a wider setting. C-like imperative programs usually have limited polymorphism vis-a-vis object-oriented codes. Further, the technique of concurrently analyzing independent control-flow branches exploits a limited amount of parallelism. In particular, it does not exploit the intra-basic-block parallelism that may appear across two instructions in the same block. For instance, two statements $p = q$ and $r = s$ can be analyzed in parallel even if they are part of the same control-flow branch. Speculative parallelism [Mendez-Lojo et al. 2010] can uncover this block-level parallelism. Our multibloom-based analysis works at the statement level and is also able to parallelize intrablock codes.

Putta and Nasre [2012] propose a parallel flow-insensitive analysis based on replication. The key idea is to exploit the monotonicity property of the points-to solution to duplicate selected points-to sets across threads and merge thread-local results periodically. By restricting threads to work on the local data, multiple threads can operate in parallel without any conflict with each other. While threads may see stale points-to information, monotonicity of points-to information (i.e., sizes of point-to sets never reduce) ensures correctness and periodic merging ensures progress. The main advantage of this analysis is that it can adapt to the number of available cores irrespective of the data dependence across various program statements being analyzed. In comparison, we deal with flow-sensitive points-to analysis on GPUs, exploit the monotonicity property to reuse a bloom-filter-based framework, and do not replicate the points-to information across threads.

Bootstrapping [Kahlon 2008] proposes to use the less precise results of an inexpensive analysis for speeding up the more precise and expensive analysis. In particular, the proposed approach focuses on computing Steensgaard's partitions imposed by a unification-based analysis [Steensgaard 1996] and then running an expensive inclusion-based analysis [Andersen 1994] on each partition. In the evaluation, the work uses partitions of aliases to simulate parallel processing—considering time required for the largest processing of partitions. However, parallelizing is not the main objective of the work, and the parallelism extracted is coarse grained. In comparison, our work uses fine-grained synchronization and focuses on parallelizing flow-sensitive analysis. We use bootstrapping to better analyze load and store statements (Section 3.3) and to improve precision (Section 3.5).

The work on program decomposition identifies various program components on which different analyses can be executed in parallel [Zhang et al. 1996; Ruf 1997]. Program decomposition is complementary to our approach in particular and parallel analyses in general.

Sequential Pointer Analysis. The area of sequential points-to analysis is rich in the literature. See the survey by Hind and Pioli [2000].

Most flow-insensitive analyses are based on Andersen's inclusion-based [Andersen 1994] and Steensgaard's unification-based [Steensgaard 1996] approaches. An inclusion-based analysis models unidirectional flow of points-to information in a pointer assignment statement, whereas a unification-based approach merges the points-to sets of the two pointer expressions in an assignment. This makes an inclusion-based pointer analysis more precise and more expensive to compute ($O(n^3)$) compared to a unification-based analysis ($O(n\alpha(n, n))$). Due to the high cost of an inclusion-based approach, until recently, production compilers like GCC relied on a primitive address-taken analysis or a unification-based analysis. However, due to significant advances in the last decade [Berndl et al. 2003; Whaley and Lam 2004; Zhu and Calman 2004; Hardekopf and Lin 2007, 2009], these compilers now use inclusion-based analysis. A few key techniques toward scaling inclusion-based analysis are online cycle detection [Fähndrich et al. 1998; Pearce et al. 2004; Hardekopf and Lin 2007], difference propagation [Pearce et al. 2004], and the use of Binary Decision Diagrams (BDDs) [Berndl et al. 2003; Whaley and Lam 2004; Zhu and Calman 2004].

The earliest works on pointer analysis were flow sensitive [Landi and Ryder 1992; Choi et al. 1993; Landi et al. 1993; Emami et al. 1994; Wilson and Lam 1995]. However, gains due to the expensive flow sensitivity have been disputed [Hind and Pioli 1998]. A common argument against flow sensitivity is that running a flow-insensitive analysis on a program converted into SSA form may be precise enough for clients. Despite the pessimism, it is shown that a precise pointer analysis helps several clients, such as typestate verification [Fink et al. 2008], security analysis [Chang et al. 2008], bug detection [Guyer and Lin 2005], and the analysis of multithreaded programs [Salcianu and Rinard 2001]. As a result, there has been renewed interest in the area of flow-sensitive pointer analysis, and the scalability of such analyses, particularly for C programs, has been greatly improved [Hardekopf and Lin 2011; Li et al. 2011; Lhoták and Chung 2011; Hardekopf and Lin 2009; Kahlon 2008]. Our bit of contribution here is to improve it further with parallelization and controlled approximation.

Use of multibloom for flow-insensitive pointer analysis has been proposed before by us [Nasre et al. 2009], and we found that multibloom offers great benefits in terms of both time and space, affecting only a little precision. However, it was unclear how to extend the benefits to a flow-sensitive analysis. Further, multibloom's regular structure and locality have not been exploited for efficiency. We also do not know how to parallelize such an analysis on GPUs. In this work, we illustrate how to design an efficient flow-sensitive points-to analysis using multibloom under massive parallelism.

Bloom Filters in Other Applications. The Bloom filter was first introduced as a probabilistic data structure for trading off space for time [Bloom 1970], and former research found several application domains where it could be useful. It was used in distributed networks to implement compact caches at proxy servers [Fan et al. 2000]. Since the routing tables stored at the proxy servers tend to get bigger and need to be periodically sent to other proxy servers for synchronization, a bloom-filter-based representation of the routing tables offers benefits both in terms of the storage requirement and the network bandwidth. The Bloom filter was also used for distributed database joins [Mackert and Lohman 1986]. The idea is to send a compact bloom-filter-based approximate representation of a table to another node in the network and perform an initial join locally on that node. Due to the condition in the join command, the total size of this intermediate result would reduce, and this intermediate join can then be sent back to the originating node. The originating node can then perform a join operation with this intermediate result to filter out some more tuples. The overall benefit is achieved by reducing the amount of data sent across the network. Gremillion [1982] used loom filters to improve the performance of differential files in a database environment. He use loom filters to check if two files contain the same data. Thus, if the bloom filter representations of the two files differ, the two files definitely contain different data. Manber and Wu [1994] used bloom filters in checking validity of proposed passwords against previous passwords used and against dictionary words. This is done by maintaining a checksum and checking a new password's checksum against the old checksum. In a similar way, a dictionary of words can be stored in an approximate manner in a bloom filter for fast hash-based lookup, achieving improved performance.

Bloom filters also underwent several advances. Mitzenmacher [2001] and Fan et al. [2000] proposed compressed bloom filters to enable more efficient transmission of the bloom filter across servers on a network. The compression further reduces the storage requirement but incurs a small cost in compression-decompression. However, its main usage is in reducing the network traffic where the network latency dominates the compression-decompression time. For faster performance while using external storage, Manber and Wu [1994] imposed a locality restriction on the hash functions used in a bloom filter. To support multisets, that is, to allow (and distinguish between) multiple occurrences of the same element in the bloom filter, Cohen and Matias [2003] proposed spectral bloom filters. To support deletions, Fan et al. [2000] proposed counting bloom filters. Counting bloom filters maintain a counter per data element instead of a bit, as in the case of a simple bloom filter. This counter is increased when an element that hashes to this location is added and reduced when the element is deleted. However, since the number of bits per counter is fixed, the number of additions and deletions should not exceed beyond a limit. Unless the number of bits is sufficiently large, a counting bloom filter may suffer from false negatives. Hence, they are not suitable for sound flow-sensitive analysis.

7. CONCLUSIONS AND FUTURE WORK

We proposed the use of multibloom and GPU parallelization for efficient flow-sensitive points-to analysis. Multibloom not only offers great benefits in terms of time and storage but also is a promising data structure from a parallelization perspective. Multibloom also offers tunable parameters to trade off minimal precision for improved analysis time. We showed how to effectively parallelize the analysis for GPUs and achieve memory coalescing, better load balance, and reduced thread divergence. Using six large open-source programs and two client transformations, we illustrated that our GPU-based flow-sensitive points-to analysis offers significant performance benefits over the sequential analysis in terms of both time and space, without adversely affecting precision.

ACKNOWLEDGMENT

We thank the reviewers for helpful suggestions especially on the effect of multibloom configurations, precision comparison between flow-insensitive and flow-sensitive analyses, and comparison with parallel implementations.

REFERENCES

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen.

Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N. 2003. Points-to analysis using BDDs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, NY, 103–114.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13,* 7, 422–426.

Chang, W., Streiff, B., and Lin, C. 2008. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS'08)*. ACM, New York, NY, 39–50.

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93).* ACM, New York, NY, 232–245.

Cohen, S. and Matias, Y. 2003. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03).* ACM, New York, NY, 241–252.

De, A. and De'Souza, D. 2012. Scalable flow-sensitive pointer analysis for java with strong updates. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'12).* Springer-Verlag, Berlin, 281–305.

Edvinsson, M., Lundberg, J., and Lwe, W. 2011. Parallel points-to analysis for multi-core machines. In *HIPEAC*.

Emami, M., Ghiya, R., and Hendren, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '94).* 242–256.

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, NY, 85–96.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8,* 3, 281–293.

Fink, S. J., Yahav, E., Dor, N., Ramalingam, G., and Geay, E. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol. 17,* 2, 9:1–9:34.

GCC. 2013. GCC, the GNU Compiler Collection. Retrieved November 22, 2013 from http://gcc.gnu.org/.

Gremillion, L. L. 1982. Designing a bloom filter for differential file access. *Commun. ACM 25*, 600–604.

Guyer, S. Z. and Lin, C. 2005. Error checking with client-driven pointer analysis. *Sci. Comput. Program. 58,* 1–2, 83–114.

Hardekopf, B. and Lin, C. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'07)*. 290–299.

Hardekopf, B. and Lin, C. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedingso the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. 226–238.

Hardekopf, B. and Lin, C. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11).* IEEE Computer Society, Washington, DC, 289–298.

Hind, M. and Pioli, A. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Proceedings of the 5th International Symposium on Static Analysis (SAS'98).* Springer-Verlag, London, 57–81.

Hind, M. and Pioli, A. 2000. Which pointer analysis should I use? In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00)*. 113–123.

Kahlon, V. 2008. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08).* ACM, New York, NY, 249–259.

Kramer, R., Gupta, R., and Soffa, M. 1992. The combining DAG: A technique for parallel data flow analysis. In *Proceedings of the 6th International Parallel Processing Symposium*. 652–655.

Landi, W. and Ryder, B. G. 1992. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92).* ACM, New York, NY, 235–248.

Landi, W., Ryder, B. G., and Zhang, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'93)*. 56–67.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*.

Lee, Y.-F., Marlowe, T., and Ryder, B. 1990. Performing data flow analysis in parallel. In *Proceedings of Supercomputing*. 942–951.

Lhoták, O. and Chung, K.-C. A. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11).* ACM, New York, NY, 3–16.

LI, L., CIFUENTES, C., AND KEYNES, N. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 343–353.

MACKERT, L. F. AND LOHMAN, G. M. 1986. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 149–159.

MANBER, U. AND WU, S. 1994. An algorithm for approximate membership checking with application to password security. *Inf. Process. Lett. 50,* 4, 191–197.

MENDEZ-LOJO, M., BURTSCHER, M., AND PINGALI, K. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 107–116.

MENDEZ-LOJO, M., MATHEW, A., AND PINGALI, K. 2010. Parallel inclusion-based points-to analysis. In *OOPSLA*.

MITZENMACHER, M. 2001. Compressed Bloom Filters. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. ACM, New York, NY, 144–150.

NASRE, R., BURTSCHER, M., AND PINGALI, K. 2013. Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY.

NASRE, R., RAJAN, K., RAMASWAMY, G., AND KHEDKER, U. P. 2009. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS'09)*.

NVPTX. 2012. NVPTX backend for LLVM. Retreived May 20, 2013 from https://github.com/llvm-mirror/llvm/tree/master/lib/Target/NVPTX.

PARTOW, A. 2013. General Purpose Hash Function Algorithms. Retrieved November 22, 2013 from http://www.partow.net/programming/hashfunctions/.

PEARCE, D. J., KELLY, P. H. J., AND HANKIN, C. 2004. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Control 12,* 4, 311–337.

PRABHU, T., RAMALINGAM, S., MIGHT, M., AND HALL, M. 2011. Eigencfa: accelerating flow analysis with gpus. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 511–522.

PUTTA, S. AND NASRE, R. 2012. Parallel replication-based points-to analysis. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. Springer-Verlag, Berlin, 61–80.

RUF, E. 1997. Partitioning dataflow analyses using types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM, New York, NY, 15–26.

SALCIANU, A. AND RINARD, M. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'01)*. ACM, New York, NY, 12–23.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS'09)*. IEEE Computer Society, Washington, DC, 1–10.

SATISH, N., KIM, C., CHHUGANI, J., NGUYEN, A. D., LEE, V. W., KIM, D., AND DUBEY, P. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY, 351–362.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM, New York, NY, 32–41.

WHALEY, J. AND LAM, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 131–144.

WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings Conference on Programming Language Design and Implementation (PLDI'95)*. 1–12.

YU, H., XUE, J., HUO, W., FENG, X., AND ZHANG, Z. 2010. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, 218–229.

ZHANG, S., RYDER, B. G., AND LANDI, W. 1996. Program decomposition for pointer aliasing: A step toward practical analyses. *SIGSOFT Softw. Eng. Notes 21,* 6, 81–92.

ZHU, J. AND CALMAN, S. 2004. Symbolic pointer analysis revisited. In *Proceedings Conference on Programming Language Design and Implementation (PLDI'04)*. 145–157.

Received June 2013; revised August 2013; accepted September 2013