Inter-procedural data-flow analyses are slow. We parallelize these predicate propagation fixpoint algorithms efficiently on a GPU.

Our approach is (mostly) synchronization free even though the processed graphs in general are cyclic and have nodes with fan-in and fan-out degrees above 1. We detect and fix any data races that occur while propagating predicates in a SIMD fashion. Additionally, we solve the parallel termination problem by means of heuristics.

The GPU-codes of five data-flow analyses are up to 89.8 times faster than their sequential LLVM variants. Offloading the analyses to the GPU saves up to 26.5% of the total compilation time.

Inter-procedural rules make predicates flow through the entire program and make the analysis expensive.

Conceptually, predicate propagation is sequential from one statement to the next along the control-flow graph (CFG) until a fixpoint is reached. However, in principle, all predicates could be propagated at the same time. For instance, if a function is called from a number of call-sites with different argument predicates, all those predicates can be copied in parallel and attached to their formal parameters.

To best use a GPU\footnote{For reading the paper, basic GPU knowhow is necessary. At least it is necessary to understand that the CPU invokes kernels that then run parallel threads on the GPU. Applications may require several transitions between CPU and GPU. Smaller kernel can have more parallelism. Since CPU and GPU do not share memory, data has to be copied before and after a kernel executes. Groups (warps) of typically 32 GPU threads always execute the same instruction in a lock-step fashion (Single Instruction Multiple Threads, SMIT). No thread can make progress independently from the others. If control flow separates threads of a warp, they are divided into groups depending on the branch they took. As only one instruction can be dispatched only one group is active, while the other is idle and waits for a convergence point. Such thread divergence wastes runtime and should be avoided.} for such a fixpoint based predicate propagation, massive parallelism is needed, i.e., conceptually one GPU-thread per analyzed instruction. But as statements directly or indirectly depend on others there are two problems that limit performance: (1) Data race problem. A GPU-thread that reads a predicate from a predecessor in the CFG while the thread of the predecessor is concurrently updating it, may see a corrupted predicate. It is necessary to deal with such data races (but it suffices to do so on either the fan-in side or the fan-out side). (2) Termination problem. As there is no central work-list, a GPU-thread may falsely assume that there is nothing left to propagate if its predecessor has not offered anything new to it. Exiting too early would leave parts of the program un-analyzed.

Solving both problems with locks, semaphores, or barriers is expensive \cite{21} and prone to dead- and/or livelocks \cite{13, 24}.

To solve problem (1) we allow races to occur and detect plus fix any corrupted predicates (Sec. 3.1). In this case, we restart the parallel propagation from there. This technique does not need any locks around concurrent updates. To solve (2) each thread uses local predicates to statements for which the predicates are guaranteed to hold. For example, to obj, .field = x, the compiler can associate the predicate hasEscaped(x). Propagation rules for all types of statements spread the knowledge along the code. If hasEscaped(b) holds for $a = b$, hasEscaped(a) holds as well. Formally:

\[
\text{if (c) } \begin{array}{l}
\text{s1; pred(a)} \\
\text{s2; pred(b)}
\end{array}
\]

Another example is the rule for an if-statement:

\[
\text{if (c) } \begin{array}{l}
\text{s1; pred(a)} \\
\text{s2; pred(b)}
\end{array}
\]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

\textcopyright 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6277-1/19/02...

\$15.00

https://doi.org/10.1145/3302516.3307352
heuristics to speculate on global termination (Sec. 3.2). Heuristics decide how long control remains on the GPU by predicting whether a fixpoint has been reached yet. Once the heuristics decide that a fixpoint has been reached, control returns to the CPU. As the heuristics may have been wrong, the CPU starts a kernel on the GPU to check whether a fixpoint has actually been reached. If not, we retry and restart the predicate propagation on the GPU. The heuristics do not impede correctness, they only affect the total runtime.

Our contributions therefore are:

- For highest performance on the fast path, we allow races to occur and fix any corrupted data later (Sec. 3.1).
- Each thread locally speculates on global termination to avoid costly synchronization (Sec. 3.2).
- In a pre-processing step, we deslag the control flow graph to purge instructions that are irrelevant for a specific analysis (and only waste runtime).
- We present some optimizations to better utilize the GPU (higher work load per GPU-thread, reduction of kernel divergence).

The rest of the paper is organized as follows. Sec. 2 describes the data structures for the GPU. Sec. 3 discusses the two solutions. Sec. 4 covers GPU-specific optimizations. Sec. 5 shows how to implement five different data-flow analyses. Sec. 6 evaluates. Sec. 7 evaluates related work.

## 2 DATA STRUCTURES

Our Parallel Code Analysis framework (ParCan) use LLVM’s bitcode representation [6, 26] and can thus deal with any input language compilable to this format, e.g., C/C++. We use the CUDA C dialect for programming the GPU [7]. Communication between host and GPU is via explicit data transfer.

We maintain two data structures on the GPU: a read-only control-flow graph and per-instruction lists of predicates that are updated when propagating. If a specific data-flow analysis requires additional data structures, they can be declared outside of ParCan and included verbatim.

### 2.1 Encoding of Predicates

For predicates we use the predicate struct shown in Fig. 1. Predicates have a generic part and an analysis-specific payload, examples of which can be found in Sec. 5.

The fields `originating_method` and `originating_insn` encode which method and which of its instructions first instantiated the predicate. This information links analysis results or error messages back to the source code. With `return_edge_stack` and `_top` we encode to which caller a predicate must be sent upon a return statement (K is the maximal depth of the stack). For example, if two functions A and B both may call a function C, a predicate propagated to C from the call-site in A should only be propagated back to A. Hence we push A’s ID onto the `return_edge_stack` when processing the call.

Predicates are kept in lists of `predicate_chunks`. A portion of the GPU’s memory is used to pre-allocate `predicate_chunk` instances. Like Hoard [12], we maintain a number of free-lists of `predicate_chunk` instances to avoid lock contention. Free memory is stolen from other free-lists once the local free-list is empty.

### 2.2 Encoding of Control-Flow Graphs

The CFG is then encoded in an array of `instruction` structures on the GPU, see Fig. 1. Each structure holds the parent method’s ID (`func`), the instruction type, and the operand values (if any). The succ and pred arrays hold the indices of the successors and predecessors in the CFG. To simplify the decoding of instruction operands, for each combination of operand types we use a specific instruction type. For example, the assignment `var1 = var2` uses opcode `ASG_L2L` (assign local-to-local), `var1.field = var2` is `ASG_L2F` (assign local to object field), ARG for passing an argument value to a formal parameter of a callee (implicit parameters, such as `this` are made explicit), etc. The sample code in Fig. 2 results in Table 1. For brevity, it only lists the predecessors.

Fig. 3 shows the conceptual kernel that propagates predicates with one GPU-thread per line of the table and one case per instruction type. As real CFGs have more instructions than a GPU has threads there is a virtualization.

### 2.3 Blurring of Method Entries

With the encoding described so far, each instruction within a method (each node of the CFG) has only one or only a few predecessors. In
A naive concurrent implementation of predicate propagation on a GPU would require many slow barriers or locks to maintain data structure consistency and to implement termination detection.

### 3.1 Managed Race Conditions

The initialization of a data-flow analysis is always free of races as it works on individual instructions. Initial predicates are never corrupt. In contrast, the propagation can face synchronization issues. To illustrate this, Fig. 5 shows the 2nd and 3rd statement of Fig. 2. Remember that conceptually there is a GPU-thread per instruction, for example, thread 1 for 'b = c' and thread 2 for 'a = b'. Without synchronization it can happen that while thread 1 is filling in the data of predicate P(b), thread 2 is already reading it to propagate
P(b) further. Similarly for P(c). This can even cause a broken P(b) to be copied away from instruction 2 and onward by other threads along the CFG.

Instead of preventing such inconsistencies by means of costly locks, we let them happen. It helps that this only happen rarely on GPUs. Due to the GPU’s SIMT (single-instruction, multiple-thread) nature all threads in a warp are implicitly synchronized. The lock-step execution model prevents threads of a warp from executing different instructions at the same time, i.e., they execute the same instruction at the same time so that either all read a predicate’s field or write a predicate’s field at a time. As predicates in one analysis pass are all of the same type and size, no divergence can occur so that the threads of a warp do not cause corrupted predicates. But there are no such guarantees between warps as they are scheduled independently, so corruption does happen, but only rarely.

Instead of using costly locks or semaphores we never synchronize but both detect and deal with corrupted predicates.

**Detecting corrupted predicates.** We add a checksum to each predicate and propagate it as well. A corrupted predicate can be detected if the propagated checksum does not match a freshly computed one. The GPU-threads can perform the preparatory work that is later needed to detect corrupted predicates in parallel and without any divergence.

**Dealing with corrupted predicates.** With the checksum every thread could locally check if one of the predicates associated to its instruction(s) is corrupted. If so it could instantly repair the affected predicate by pulling it again from its predecessor(s). However, this would not only add divergence. It would also increase the size of the kernel and the number of registers it uses for fast local variable access. This in turn would reduce the amount of parallelism as the total number of threads is limited by the number of GPU registers divided by the number of registers a thread uses.

We thus separate the parallel propagation work from the detection/repair. In Fig. 6 the outer loop on the CPU alternates between a (parallel) predicate propagation that includes the preparatory work, and a (parallel) verification phase in which the threads check all their predicates for correctness and repair them if needed. Using a repair path in the verifier also guarantees progress, i.e., it guarantees that a fixpoint will be reached eventually. The repair phase cannot cause data races as it only uses concurrent reads.

As corruptions only occur rarely, we can let the propagation run for many cycles/until termination (Sec. 3.2) and only then verify and restart. This strips the slow and divergence-prone verification and repair from the propagation cycle. Only if corruptions happened a lot, the propagation would waste time working on corrupted predicates – work that needs to be re-done after the repair.

Checksums are a lightweight way to detect and resolve races. Other solutions involve slow transitions between CPU ↔ GPU to implement global synchronizations. For instance, another way to avoid races is to split the computation and propagation into two kernels, causing an extra transition. In Sec. 6 we show that transitions should be avoided especially as races occur rarely.

### 3.2 Fixpoint Determination using Heuristics

Consider the following scenario when propagating predicates in Table 1. Threads 0...13 are performing propagation work for instructions 0...13. In the first cycle thread 1 can pull a predicate from instruction 0. However, the other threads 2...13 have nothing to do yet as none of their predecessors have any work/new predicates to offer. With only thread-local information they cannot decide if a fixpoint has been reached. Normally, synchronization barriers are needed to ensure that no thread exits early but waits for propagation work that could be handed to it later on.

There are two naive solutions to avoid synchronization barriers. First, we could return to the CPU after each (parallel) propagation step. This is too slow as it needs repeated switching between CPU and GPU and back. Second, we could maintain some global state to keep track of the propagation. This suffers because synchronization is needed to keep the global state consistent and additionally there is the need to wait for some threads to finish (using some form of global barrier). Unfortunately, GPUs only support barriers for blocks (multiple warps are grouped to blocks). Building a global barrier on top of these local barriers is prone to dead- and livelocks as the scheduling for warps and blocks is unknown [15, 24].

To get around this, we let the GPU-threads attempt to perform propagation work as long as purely thread-local heuristics suggest (without synchronization) that a fixpoint has probably not been reached yet. If the heuristics over-estimate the time needed to reach a fixpoint (so that GPU-threads stay too long on the device) performance degrades as we cannot return the results of the dataflow analysis. If the heuristics under-estimate, we return to the CPU too early while there is propagation work still left to be done. The heuristics do not affect the accuracy of an analysis in any way. They only optimize the runtime by avoiding transitions. The computed fixpoint is always correct. The verification kernel that also tests for data structure consistency (see Fig. 6) detects this so that in response the outer loop launches the propagation kernel again. Instead of synchronization overheads in each propagation cycle, we only have the overhead from the potentially superfluous GPU → CPU → GPU transition(s) if the heuristics misfire early.

We experimented with the following three heuristics:

**Global:** A single global boolean on the GPU is set to false if some thread changes a predicate. If the variable is true at the end of a

```cpp
bool verified() {
    parfor k = 0 ... #threads // = # of insns // on GPU
    foreach predicate P in instruction(k):
        if corrupted(P):
            repair(P);
        return false;
    // heuristic barrier support:
    if could propagate P to succ(k)
        return false;
    return true;
}

do { // outer loop on CPU
    parfor k = 0 ... #threads // = # of insns
    predicate_prop_kernel(k); // on GPU
} while not verified();

Figure 6: Managed racy propagation (pseudo code).
cycle, a fixpoint is assumed and control returns to the CPU to call the verification kernel.

**Toggle**: A fixed number of propagation cycles is done before returning to the CPU’s outer loop.

**Groups**: Use an array of global counters. Every block is responsible for an index of the array. Depending on the size of the array, two or more warps may share a counter. Each time a thread changes an instruction’s predicate, it increments its block’s counter. A fixpoint is assumed if a block’s counter remains unchanged between propagation cycles.

These heuristics neither use atomic variables nor synchronization. A race in heuristics *Global or Groups* would let control more quickly return to the CPU (were the verification kernel determines that a fixpoint has not been reached yet and restarts the propagation kernel in response). *Global* is sub-optimal (which the performance section shows) since without synchronization a GPU may cache a single boolean variable for longer periods so that many warps exit too early. For *Toggle* a manual selection of the fixed number of cycles is sub-optimal in most cases. *Groups* is best, see Sec. 6.1.

## 4 PERFORMANCE OPTIMIZATIONS

### 4.1 Deslagging of the CFG

Often there are instructions in a CFG that do not affect the predicates but that only propagate them to the next instruction. Any such node in the CFG adds useless cycles before the fixpoint is reached. From a program’s CFG we can therefore build its analysis-specific deslagged CFG that leaves out uninteresting nodes. Doing so sequentially is in general more costly than leaving them in the CFG. But on a GPU the deslagging can be done in parallel in a pre-processing step. It marks all instructions in parallel if one of the propagation rules affects them. It marks a call instruction if any of the instructions in the called method is marked. It marks a control flow statement (e.g. if) if there is a marked statement in at least one of its branches. Once a fixpoint is reached, unmarked instructions are deslagged from the CFG.

### 4.2 Sorting the Instructions of a Program

GPUs can run many threads concurrently (1000+). They are organized into individually managed warps (typically 32 threads). Divergence on a GPU happens when some GPU-threads in a single warp have to follow a control-flow path that is different from what other threads in that warp do. A GPU then executes the threads that follow one path to completion, while halting the others. Once they have finished, the halted GPU-threads are resumed. This reduces the amount of parallelism. Thus, for best performance, all threads in a warp should perform the same actions.

We see divergence whenever the threads in a warp have to handle various types of instructions. This is even worse if there is virtualization and the threads have to process many instructions. However, as it is irrelevant in which order the instructions are processed to perform predicate propagation, a pre-processing step sorts the instruction array with respect to the instruction type (respecting the predecessor/successor relations) when we encode the CFG. Sorting makes it more likely that all threads in a warp process the same instruction type. This reduces the amount of divergence.

### 4.3 Intra-Warp Load Balancing

The per-thread execution time depends on the number of predicates a thread has to process. If one thread has more work to do than others in the same warp, up to 31 cores idle until the bottleneck thread is done. Similar imbalances exist at the level of blocks.

Coarse-grain load balancing on the GPU in general requires atoms, locks or barriers to synchronize the concurrent access to shared data structures ([14, 19, 25, 31, 37]). To avoid synchronization as far as possible as they are prone to dead-/livelocks and sequentialize the execution of threads, we perform a fine-grained load-balancing within a warp, exploiting that threads of a warp are already implicitly synchronized (SIMT execution model) and can directly communicate with each other by reading the other threads’ registers (SHFL instructions). So, instead of idling, once done with its own predicates, a thread checks the number of unfinished predicates of the other threads in the same warp and takes over some of the predicates of the thread with the most unfinished ones. (Processed predicates are marked with an atomic instruction to avoid processing them multiple times.)

## 5 USE CASES

The purpose of this section is two-fold. First, we demonstrate how a broad spectrum of different data-flow analyses can be implemented in ParCA. Table 2 shows the variation w.r.t. direction, scope, payload, and data structures for storing the analysis results.

For each data-flow analysis the implementer needs to specify the payload of the predicate class, how predicates should be instantiated in parallel (initialization rules), how they should be modified or copied in parallel for each type of instruction (propagation rules), and which queries retrieve – also in parallel – the results of the analysis. ParCA generates the GPU kernels from these rules.

Second, this section also introduces the five analyses that Sec. 6 quantitatively compares to (existing) implementations in LLVM/COIN that perform all steps sequentially, including the queries.

### 5.1 Escape Analysis (EA)

The Escape Analysis [17] in our set of use cases is based on and compared to a sequential LLVM version [5]. We sketched its idea in Sec. 2.4. Let us now demonstrate that ParCA is expressive enough to implement it. As the syntax to formulate rules and queries is not the contribution of this paper, we only illustrate it in examples.
**class** isNew extends predicate {
    **bool** escaped; // *if escaped or not**
    **int** local_var_id; // traced variable
}

Figure 7: EA predicate.

rules.add(new InitializationRule() {
    .trigger("localvar = new")
    .after("pr.local_var_id = localvar ;" +
        "pr.escaped = false ;") ;
}

Figure 8: EA instantiation rule.

void assign_L2L_prop(instruction ins) {
    foreach p in predecessor(ins):
        foreach x in predicate(p):
            // the condition as specified in the rule:
            if (x.local_var_id == localvar2)
                // after section of the rule:
                ins.add_pred(x);
}

Figure 10: GPU code generated from Fig. 9.

**Payload.** A single boolean value and a value to store a variable’s ID suffices (see Fig. 7). A per-method table keeps local variables. For x = new X(), where x is the third local variable of the method, the value stored in a predicate’s local_var_id would be 3.

**Initialization rules.** Whenever the trigger localvar = new is seen, the newly created object starts as unescaped. From the initialization rule in Fig. 8 ParCAn creates a kernel similar to the one in Fig. 3 to initialize the predicates in parallel on the GPU.

**Propagation rules.** If no other rule applies, predicates are copied verbatim. Fig. 9 holds an example of a developer-supplied assignment rule. It clones every predicate pr that is associated with the right-hand-side’s local variable (rhs) of the assignment, associates it with the lhs, and adds it to the current instruction ins. The ParCAn code generator processes such rules. It maps the trigger to the opcode and it places the rule’s condition and the after-section into the generated *prop routines (Fig. 10).

As mentioned in Sec. 2.2, stack-allocation is also illegal (conservatively) if an object’s reference is assigned to another object’s field. Given localvar1.field = localvar2 or static_variable = localvar2, we assume that the localvar2 object escapes. Fig. 11 shows the necessary propagation rules. The after-section of rule 1 is placed into assign_L2F_prop (the trigger matches the opcode of the assign-local-to-field). With this rule the analysis can now also compute that the second object created in Fig. 2 cannot be stack-allocated. Rule 2 is similar for static fields. Its after-section is placed into assign_L2S_prop (assign-local-to-static-variable). **Query.** The escape analysis needs to check the predicates at the return statements (Fig. 12). They are all added to the result set if they fulfill the given condition, namely that their escape flag is set or that the predicate originally came from the same method.

---

**5.2 Flow-sensitive Points-to Analysis (FsP)**

As a use case example with more complex propagation rules and queries, we picked a flow-sensitive points-to-analysis that determines for each local variable (or parameter) to which objects it may point [23]. The result is a set of predicates that store the points-to-information for each local variable. For example if an instruction has in its predicate list PointsTo(p, 1), PointsTo(p, 2), and PointsTo(q, 1), this encodes that at this point of the CFG (a) the local variable p may point to an object allocated by the NEW at instruction 1 or 2, and (b) that q may also point to the object from instruction 1. As sequential implementations of this analysis are too slow to be useful, this is not part of LLVM. For the evaluation we use a home-grown serial implementation to propagate the points-to-information.

**Payload.** The analysis-specific payload is straightforward. There are two fields to store the ID of the local variable (int ptr_var_id) and of the NEW instruction (int new_id).

**Initialization rules.** Similar to Sec. 5.1 the instruction ptr_var_id = NEW triggers the creation of a new predicate that stores the ID of the variable and the NEW instruction.

**Propagation rules.** Fig. 13 shows the propagation rules for a copy operation like x = y. Rule 1 clones a predicate, e.g., pr(y, 1) and replaces its ptr_var_id with x (x points to the same location as y). Rule 2 processes incoming predicates. ParCAn allows to
5.3 Dead Code Elimination (DCE)

To also include a backward data-flow analysis to our set of use cases, we picked a Dead Code Elimination that uses liveness analysis to detect dead variables and hence dead code [18]. When activated (-dce in the LLVM framework) DEC runs multiple times since dead life, not all critical variables that appear on the lhs of an assignment.

5.4 Global Value Numbering (GVN)

As another use case we picked Global Value Numbering [38], as it needs expressions instead of variables in the predicate payload.

As shown in Fig. 15 the operator and the operands of the expression are encoded as integers. Additionally, there is a pointer that refers to the instruction that makes the expression redundant. The initialization and propagation rules as well as the query are straightforward [38]. The latter uses a condition to NULL-check the rep_id.

We implemented GVN based on the static single assignment form (SSA) and question propagation [38]. We compare this to the sequential implementation in the COINS compiler framework [4].

5.5 Andersen’s Points-to Analysis (AA)

Finally, we picked Andersen’s context- and flow-insensitive Points-to Analysis [10] because it does not store its results in predicates but builds an external graph data structure while traversing the CFG. We show how to map AA to ParCan. The generated code results in a graph traversal on the GPU that other AA implementations also use [29, 43]. We compare our implementation to a multi-core CPU [30] solution, to an alternative GPU-parallel version [29], and to a sequential implementation for LLVM [2].

Payload. A predicate only contains a reference to an edge in the external points-to graph.

Initialization rules. For pointer-related instructions the rules in Table 3 initialize the external points-to graph. Each application of a rule adds a constraint as an edge to that graph which later has to be resolved. Whenever a thread works on a predicate it resolves the constraints of its edge in the external graph.

Propagation rules. This phase resolves the constraint set. A thread tries to apply the rewriting rules of Table 3 to the constraint that is reachable through it predicate. If a pattern can be applied a new constraint (edge) is added (column rewriting rule) to the external graph which also results in a new predicate. To implement a load-balancing this new predicate is assigned to an instruction that does not yet have a predicate. Hence, as the points-to graph grows, more and more threads are involved in constraint resolution. Fig. 16 shows in pseudo-code how this can be expressed in ParCan. The method lookup_pattern returns the newly added edge. If e is NULL no pattern can be applied. The fixpoint is reached if no predicates are created anymore.
The **Query** calls a graph method to check if two nodes have a connecting edge in the points-to graph.

**Synchronization Issues.** While ParCAn can deal with corrupted predicates there is no built-in mechanism to handle any concurrently modified external data structures. To avoid the complexity of reordering or fixing an external data structure in the verification phase, ParCAn provides an uncorrupted-test, see Fig. 16. This guard ensures that side-effects (e.g., lookup_pattern) only materialize if a predicate is sound. Other than that the external functions themselves need to efficiently implement the necessary mutual exclusion. Our graph implementation uses pre-allocated arrays of nodes and edges. To safely add edges (constraints) to a node’s edge-list we use the GPU’s atomic compare-and-exchange instructions. We allow duplicate edges per node to avoid thread synchronization and filter out the duplicates afterwards to reduce divergence. As this is graph-specific code, the details are outside of the scope of this paper.

## 6 PERFORMANCE

Table 4 gives the characteristics of the benchmark programs (which other publications [29, 30, 43] also use to evaluate code analyses) and that vary w.r.t. the number of instructions and the size of the CFGs. Table 2 mentioned the structural diversity of the data-flow analyses. Table 4 shows that their dynamic behavior is also different. In the ParCAn codes the number of steps is different before the threads reach a fixpoint. In a step all threads propagate their predicates once. The \(pr\) numbers show how many predicates the GPU has to deal with in total. The \(corr\) numbers show that corrupted predicates do occur, albeit rarely. DCE and GVN are close together as their traversal is similar.

For each of the five analyses (EA – AA) we have a sequential implementation (in LLVM, in COINS, or home-grown) and a ParCAn implementation. For AA there are two more parallel versions. We verified that all versions always produce the same analysis results.

For all runtime measurements we used a Linux workstation (Debian 9 "Stretch") equipped with a 3.40 GHz quad-core Intel Core i7-3770 CPU. Our CUDA code targets the NVIDIA CUDA Toolkit 9.1. We used the CUDA Driver API to manually instantiate the context of a kernel and to avoid runtime artefacts of the first call of a kernel. The kernels run on a TITAN XP with 3,840 CUDA Cores @ 1.582 GHz organized into 30 SMs. The GPU has 12 GB of memory and 64/4096 KB of programable L1/L2 caches. ParCAn was compiled using LLVM 7.0.

All the analyses are implemented in Java, including the codes for the compiler framework COINS and the multi-core implementation of Andersen’s points-to analysis. We used the Java HotSpot 64-bit Server VM 18.3 to execute them. To minimize the influence of the garbage collector to zero (profiler-verified) we maximized the size of the JVM heap.

We performed all runtime measurements 100 times and only report averages. Each Java benchmark is executed 105 times, with the first 5 measurements dropped to exclude JIT warmup effects.

Sec. 6.1 works with the ParCAn codes on the GPU only since we evaluate the impact of the optimizations. Sec. 6.2 and 6.3 compare ParCAn performance (with all optimizations engaged) to the sequential analyses and to other parallel versions of AA.

### 6.1 Optimizations

Let us first select ParCAn’s heuristics to detect termination (Sec. 3.1). Recall that the heuristics rule how many transitions CPU ↔ GPU there are. Fig. 17 shows the speedup of the heuristics compared to the worst case that uses a transition GPU ↔ CPU after every step, i.e., whenever all threads have propagated all their predicates once.

The **Toggle** heuristics return to the CPU after \(t\) propagation steps. For **Toggle** (gray boxes) we measured the runtime for all fixed \(t \in [1; \text{steps}]\). The performance critically depends on \(t\). It varies between benchmarks and between analyses. **Toggle** of course performs best for \(t = \text{steps}\), i.e., when there are no extra transitions. Only with careful hand-tuning, \(t\) can be set to the ideal value.

**Global** is the slowest heuristics (black boxes). The lack of a global synchronization mechanism, the weak memory consistency model, and caching effects cause many CPU ↔ GPU transitions because warps have a different view on the global synchronization variable. **Atomic** operations and **memfence** can help to bypass at least the memory consistency model issues but they lower the performance even more as they sequentialize the threads.

The **Groups** heuristics is best (white boxes). It achieves a performance close to **Toggle** but without the need to manually tune \(t\). Recall that in **Groups** the threads track the progress of other thread blocks by reading a shared counter array. Hence, if predicates are
GPU-Accelerated Fixpoint Algorithms for Faster Compiler Analyses

CC ’19, February 16–17, 2019, Washington, DC, USA

Table 4: Benchmark statistics. Instruction count insns according to cloc, number of steps (all threads propagate their predicates once) before reaching the fixpoint, pr = total number of predicates (1000), corr = number of corrupted predicates.

<table>
<thead>
<tr>
<th>benchmark programs</th>
<th>insns ( \times 1000 )</th>
<th>CFG size (MB)</th>
<th>EA steps</th>
<th>EA pr</th>
<th>EA corr</th>
<th>EAE steps</th>
<th>EAE pr</th>
<th>EAE corr</th>
<th>DCE steps</th>
<th>DCE pr</th>
<th>DCE corr</th>
<th>AA steps</th>
<th>AA pr</th>
<th>AA corr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>19.6</td>
<td>8.6</td>
<td>23</td>
<td>1.7</td>
<td>1</td>
<td>95</td>
<td>830</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>213</td>
<td>19</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>emacs-26.1</td>
<td>236</td>
<td>7.2</td>
<td>25</td>
<td>1.7</td>
<td>1</td>
<td>95</td>
<td>830</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>213</td>
<td>19</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>pine-4.64</td>
<td>315</td>
<td>9.6</td>
<td>21</td>
<td>3.0</td>
<td>1</td>
<td>99</td>
<td>950</td>
<td>0</td>
<td>20</td>
<td>370</td>
<td>1</td>
<td>21</td>
<td>296</td>
<td>1</td>
</tr>
<tr>
<td>mplayer-1.3.0</td>
<td>319</td>
<td>9.7</td>
<td>29</td>
<td>4.0</td>
<td>1</td>
<td>88</td>
<td>1,570</td>
<td>1</td>
<td>17</td>
<td>483</td>
<td>1</td>
<td>20</td>
<td>392</td>
<td>1</td>
</tr>
<tr>
<td>vml-8.1</td>
<td>320</td>
<td>9.7</td>
<td>21</td>
<td>7.7</td>
<td>3</td>
<td>111</td>
<td>2,990</td>
<td>1</td>
<td>18</td>
<td>931</td>
<td>1</td>
<td>15</td>
<td>745</td>
<td>3</td>
</tr>
<tr>
<td>svn-11.0.0</td>
<td>415</td>
<td>12.6</td>
<td>27</td>
<td>8.2</td>
<td>3</td>
<td>97</td>
<td>7,960</td>
<td>3</td>
<td>20</td>
<td>897</td>
<td>1</td>
<td>21</td>
<td>782</td>
<td>2</td>
</tr>
<tr>
<td>perl-5.28</td>
<td>452</td>
<td>13.7</td>
<td>22</td>
<td>9.8</td>
<td>2</td>
<td>89</td>
<td>7,120</td>
<td>3</td>
<td>15</td>
<td>1,176</td>
<td>1</td>
<td>13</td>
<td>942</td>
<td>4</td>
</tr>
<tr>
<td>python-3.7.1</td>
<td>521</td>
<td>15.9</td>
<td>26</td>
<td>13.3</td>
<td>4</td>
<td>108</td>
<td>7,860</td>
<td>3</td>
<td>21</td>
<td>1,358</td>
<td>3</td>
<td>23</td>
<td>1,228</td>
<td>3</td>
</tr>
<tr>
<td>gimp-2.10</td>
<td>712</td>
<td>21.7</td>
<td>23</td>
<td>16.5</td>
<td>5</td>
<td>139</td>
<td>11,300</td>
<td>5</td>
<td>16</td>
<td>1,980</td>
<td>4</td>
<td>19</td>
<td>1,584</td>
<td>5</td>
</tr>
<tr>
<td>php-7.1.23</td>
<td>966</td>
<td>29.4</td>
<td>29</td>
<td>17.9</td>
<td>6</td>
<td>134</td>
<td>12,970</td>
<td>5</td>
<td>24</td>
<td>2,149</td>
<td>7</td>
<td>26</td>
<td>1,740</td>
<td>4</td>
</tr>
<tr>
<td>gdb-8.2</td>
<td>1,904</td>
<td>58.1</td>
<td>22</td>
<td>18.6</td>
<td>6</td>
<td>155</td>
<td>18,390</td>
<td>7</td>
<td>18</td>
<td>2,235</td>
<td>5</td>
<td>19</td>
<td>1,788</td>
<td>6</td>
</tr>
<tr>
<td>gcc-8.2</td>
<td>2,902</td>
<td>88.5</td>
<td>28</td>
<td>20.4</td>
<td>7</td>
<td>161</td>
<td>22,680</td>
<td>9</td>
<td>12</td>
<td>2,451</td>
<td>6</td>
<td>15</td>
<td>1,936</td>
<td>7</td>
</tr>
<tr>
<td>tkhark-2.64</td>
<td>3,696</td>
<td>112.8</td>
<td>23</td>
<td>30.6</td>
<td>10</td>
<td>208</td>
<td>24,390</td>
<td>11</td>
<td>21</td>
<td>3,676</td>
<td>9</td>
<td>24</td>
<td>2,794</td>
<td>10</td>
</tr>
<tr>
<td>linux-4.14.5</td>
<td>6,273</td>
<td>190.2</td>
<td>27</td>
<td>51.8</td>
<td>17</td>
<td>212</td>
<td>45,460</td>
<td>18</td>
<td>10</td>
<td>6,223</td>
<td>16</td>
<td>13</td>
<td>5,103</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 5: Reasons for the impact of the optimizations.

<table>
<thead>
<tr>
<th>benchmark programs</th>
<th>% dropped instructions</th>
<th>% divergence-free warps (before → after sorting)</th>
<th>% warps with load-balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EA</td>
<td>DCE</td>
<td>GNU</td>
</tr>
<tr>
<td></td>
<td>% dropped instructions</td>
<td>% divergence-free warps (before → after sorting)</td>
<td>% warps with load-balancing</td>
</tr>
<tr>
<td></td>
<td>EA</td>
<td>DCE</td>
<td>GNU</td>
</tr>
<tr>
<td>ex</td>
<td>43</td>
<td>31</td>
<td>17</td>
</tr>
<tr>
<td>emacs-26.1</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>pine-4.64</td>
<td>47</td>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>mplayer-1.3.0</td>
<td>52</td>
<td>49</td>
<td>15</td>
</tr>
<tr>
<td>vml-8.1</td>
<td>52</td>
<td>49</td>
<td>15</td>
</tr>
<tr>
<td>svn-11.0.0</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>perl-5.28</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>python-3.7.1</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>gimp-2.10</td>
<td>52</td>
<td>49</td>
<td>15</td>
</tr>
<tr>
<td>php-7.1.23</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>gdb-8.2</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>gcc-8.2</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>tkhark-2.64</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>linux-4.14.5</td>
<td>47</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>average</td>
<td>48.1</td>
<td>39.7</td>
<td>17</td>
</tr>
</tbody>
</table>

Sorting the Instructions. As mentioned before, thread divergence causes their serialization until they reach a convergence point ~ before they run in parallel again. Fig. 18(b) shows that sorting the instructions by their type always speeds up the runtime. Sorting increases the fraction of the warps that are free of divergence, see the second segment of Table 5. For FsP the deslapping removed most types of instructions. Hence, sorting the remaining ones yields large stretches of the same type in the instruction table, i.e., many warps without divergence. The same holds for the few instruction types that remain after the deslapping for EA. But even for AA where deslapping did not remove any instructions, sorting achieves a few more divergence-free warps.

By means of **Intra-Warp Load Balancing** the idle threads of a warp help the busy ones. As long as the number of predicates per thread is more or less evenly distributed this optimization has little effect. In its third segment, Table 5 shows the fraction of the warps that face a load imbalance.

Fig. 18(c) shows the speedup that a load-balancing achieves. FsP benefits the most. The reason is that the propagated predicates pile up at join nodes of the CFG, while other nodes/instructions see far fewer predicates. In contrast, DCE and GVN propagate predicates on the coarser granularity of functions, causing smaller piles and less imbalance. Interprocedural analyses do not benefit a lot from intra-warp load balancing as their propagation paths are too short.

evenly distributed over the CFG all blocks have work to do and know that a fixpoint has not been reached yet. On the other hand, underutilized blocks of threads seem to be active long enough to notice the updates that happen in other blocks.

Let us now evaluate the individual impact of the three crucial optimizations from Sec. 4. To do so, we configured ParCAn with the Group heuristics and engaged all optimizations, except for the one whose effectiveness we discuss.

Fig. 18(a) shows that **CFG deslapping** causes a speedup for all analyses (except for AA). Recall that this optimization drops instructions from the CFG that do not affect the predicates. The number of dropped instructions obviously depends on the analysis. Table 5 shows in its first segment the percentage of dropped instructions for the 14 benchmark programs. For EA most of the instruction types are dropped, except for NEW, assignments (e.g., \( x = y \)), method calls, and return. DCE and GVN have a lower drop rate (about 17%) as all those instructions need to remain in the CFG that touch a variable or expression. For AA no instruction type can be purged at all as they are all used for load balancing of predicates. Enabling the deslapping leaves the AA runtimes unchanged. For FsP the deslapping achieves the highest drop rate (almost 70%) as only a few pointer-related instructions survive. Hence, for FsP there are the best speedups. (Note, that before the FsP-query can run, a post-processing step is needed that spreads the points-to information to the formerly dropped instructions.)
This also applies to EA since the propagation stops in the function that instantiates the object.

6.2 Parallelization Gain
Table 6 shows how much speedup the parallel fixpoint computation in ParCAn achieves compared to the sequential implementations in LLVM or COIN, as well as to other parallel versions of AA. For example, the ParCAn EA is 1.9 times faster than the EA in LLVM, even though ParCAn must ship the CFG to the GPU. Table 6 reports the runtimes of just the analyses, including shipping but without the rest of the compilation. In all but the few grey cells the ParCAn analyses are faster, even compared to known-to-be-good sequential implementations in LLVM or COIN (first three columns).

Only for small programs like ex the speedup is moderate since there are too few predicates (see \(pr\) in Table 4) to fully utilize the GPU. The speedup grows with an increasing number of instructions. For the largest input program Linux we achieve speedups of 82.3 (EA), 89.8 (DCE), and 85.3 (GVN).

FSP benefits most from the parallelization. ParCAn reaches a speedup of up to 1551.3 compared to our home-grown sequential flow-sensitive points-to analysis. Given that flow-sensitive points-to analyses are known to be too slow for practical use (LLVM does not even provide an interface to add such an analysis pass [1]), this is a promising result, as runtime in the range of 485 ms to 83 s (for the benchmark programs) may be acceptable.

AA is not a classic data-flow fixpoint problem. Instead it is a constraint-set problem that is interpreted as a graph traversal problem and mapped onto the fixpoint engine. Recall that it is not a natural fit to ParCAn and uses an external data structure to store its results. Nevertheless the ParCAn version outperforms the hand-tuned sequential LLVM implementation and achieves a speedup of 7.7 on average. While ParCAn is on par with a special-purpose GPU version of AA [28], it achieves a speedup of up to 5.6 compared to a multi-CPU solution [30] (that does not suffer from CPU ↔ GPU data transfers).

### Table 6: Speedup of ParCAn analyses over other implementations. Values are normalized to the runtime of ParCAn.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>EA LLVM</th>
<th>DCE LLVM</th>
<th>GVN COIN</th>
<th>FSP LLVM</th>
<th>AA CPU-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>1.9</td>
<td>1.4</td>
<td>2.2</td>
<td>33.0</td>
<td>1.0</td>
</tr>
<tr>
<td>emacs</td>
<td>11.7</td>
<td>14.2</td>
<td>9.2</td>
<td>94.5</td>
<td>1.1</td>
</tr>
<tr>
<td>pine</td>
<td>14.5</td>
<td>13.8</td>
<td>12.9</td>
<td>113.2</td>
<td>1.1</td>
</tr>
<tr>
<td>mplayer</td>
<td>12.9</td>
<td>13.9</td>
<td>13.5</td>
<td>121.8</td>
<td>1.2</td>
</tr>
<tr>
<td>vim</td>
<td>14.7</td>
<td>15.5</td>
<td>14.9</td>
<td>111.1</td>
<td>1.3</td>
</tr>
<tr>
<td>svn</td>
<td>14.7</td>
<td>16.7</td>
<td>15.2</td>
<td>122.3</td>
<td>1.3</td>
</tr>
<tr>
<td>perl</td>
<td>17.7</td>
<td>18.1</td>
<td>17.9</td>
<td>153.7</td>
<td>1.3</td>
</tr>
<tr>
<td>python</td>
<td>19.3</td>
<td>20.3</td>
<td>21.7</td>
<td>193.3</td>
<td>1.4</td>
</tr>
<tr>
<td>gamp</td>
<td>18.3</td>
<td>24.2</td>
<td>23.2</td>
<td>237.3</td>
<td>1.5</td>
</tr>
<tr>
<td>php</td>
<td>24.8</td>
<td>28.2</td>
<td>23.2</td>
<td>250.4</td>
<td>1.5</td>
</tr>
<tr>
<td>gdb</td>
<td>27.3</td>
<td>31.3</td>
<td>29.1</td>
<td>528.8</td>
<td>1.5</td>
</tr>
<tr>
<td>gcc</td>
<td>29.2</td>
<td>35.2</td>
<td>31.3</td>
<td>533.2</td>
<td>1.6</td>
</tr>
<tr>
<td>tshark</td>
<td>42.2</td>
<td>49.3</td>
<td>51.2</td>
<td>793.3</td>
<td>2.0</td>
</tr>
<tr>
<td>linux</td>
<td>82.3</td>
<td>89.8</td>
<td>85.3</td>
<td>1551.3</td>
<td>2.0</td>
</tr>
<tr>
<td>min</td>
<td>1.9</td>
<td>3.1</td>
<td>2.2</td>
<td>33.0</td>
<td>1.5</td>
</tr>
<tr>
<td>average</td>
<td>23.7</td>
<td>26.7</td>
<td>25.1</td>
<td>345.5</td>
<td>2.0</td>
</tr>
<tr>
<td>max</td>
<td>82.3</td>
<td>89.8</td>
<td>85.3</td>
<td>1551.3</td>
<td>2.0</td>
</tr>
</tbody>
</table>


6.3 End-to-End compile-time Improvement
We also evaluated the total compilation time. As the baseline for Fig. 19 we first compiled the benchmark programs with CLANG [3] with -O3. Then we replaced three sequential analyses (AA, EA, and DCE) with their ParCAn versions. Fig. 19 holds the runtime decomposition for the benchmark programs. For the LLVM baseline and the ParCAn measurements only the analysis times differ, all other segments of the bars remain unchanged. Only for the smallest benchmark program ex ParCAn causes a slowdown as the overheads of creating the GPU data structure, shipping the data to the GPU, deslaging, and converting the results back to LLVM’s data structures cannot be amortized. For all other benchmark programs ParCAn speeds up the compilation (bars above 1). As discussed in Sec. 6.2 ParCAn gets more effective with larger inputs that results in a better utilization of the GPU. Hence, for the largest benchmark...
7 RELATED WORK

We limit the discussion of related work to the parallelization of code analyses and skip attempts to parallelize other phases of the compiler or to parallelize the compilation of many input files (think of parallel ‘make’). This section is structured as follows: We first look at frameworks to express and run code analyses. Only a few of them employ parallelism. We then cover the literature on parallel fixpoint algorithms in general. ParCAn is the first attempt we know of that applies parallel fixpoint computation on a GPU to various code analyses. The third part of this section looks into approaches to parallelize specific points-to analyses. We close with a brief discussion of work that can make use of ParCAn.

There are many frameworks for static code analysis. Most of them are purely sequential and focus on making the analyses scale to large programs [22, 33, 39, 46, 46]. There are some exceptions that employ parallelism, but they suffer from restricted applicability. Aiken et al. [8] use summaries at the granularity of methods to gather analysis information in a bottom-up fashion (only). As the summaries decouple the caller from the callee they can be computed in parallel. Bolt [9] uses map-reduce parallelism for top-down analyses but it does not scale very well. EigenCFG [36] hard-codes analyses on the GPU but it is restricted to simple analyses of type- and value-flow (so-called OCFAs that do not allow any calls). Nicarus [44] is restricted to analyzing functional programs. ParCAn is more general. It is not restricted to top-down- or bottom-up-analyses, it can be used for fine-grained analyses instead of working at the level of functions, and it also allows to use analysis-specific data structures (think of AA’s points-to graph).

Figure 19: End-to-end compile times with LLVM using ParCAn analyses (AA, EA, and DCE) instead of the official sequential ones. Runtimes normalized to the LLVM execution time. Bars above 1 indicate a speedup/slowdown.

<table>
<thead>
<tr>
<th>Language</th>
<th>Normalized end-to-end execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1</td>
</tr>
<tr>
<td>gdb</td>
<td>1</td>
</tr>
<tr>
<td>tshark</td>
<td>1</td>
</tr>
<tr>
<td>linux</td>
<td>1</td>
</tr>
</tbody>
</table>

Instead of the small black segment of the bar that indicates the cost of the ParCAn analyses for the *linux* benchmark, a decomposition of the sequential LLVM compilation would show a much larger black segment, as it is this segment where ParCAn’s offloading of analyses to the GPU helps.

Literature on general fixpoint problems focuses on reducing the complexity of the problem so that it can be solved faster. These approaches are orthogonal to ParCAn and could be added to ParCAn. Let us briefly look at three examples. (a) Ashley et al. [11] improve fixpoint computations by exploiting the presence of higher-order functions. (b) Instead of considering all predicates associated with an instruction, projections help to skip predicates that cannot be affected by the instruction or that are irrelevant for the query. This is the focus of work by Chen et al. [16]. (c) Nasre et al. [35] show that by sorting the predicates of an instruction according to their order of importance they reach a fixpoint faster.

As points-to analyses are among the most time-consuming ones, there have been specific attempts to parallelize them. We discuss three examples. (a) Mendez et al. [29] ported Anderson’s alias-analysis to GPUs. They optimize the performance by first extracting a smaller constraint-graph from the code, and then applying rewrite rules in parallel until a fixpoint is reached. Their work is restricted to AA, whereas ParCAn performs on par and can also be used for other analyses. (b) Nasre et al. [34] parallelized another points-to analysis for the GPU. However, they store points-to data in a probabilistic Bloom Filter and thus only achieve imprecise results. With ParCAn we can reach more exact points-to results within the same time budget. (c) Su et al. [42] presented a parallel points-to analysis that is incremental and demand-driven. A query for a points-to relationship initiates the work and recursively triggers more work, including predicate propagation. Parallelism is improved by caching graph queries (an orthogonal optimization to ours) and sorting the graph queries to avoid redundant queries (again an orthogonal optimization to ours). There is a higher degree of parallelism in ParCAn as we exploit the massive per-instruction parallelism in a mostly synchronization-free way.
Finally there is some work that can be extended to use ParCAn as a building block. (a) Mendez et al. [28] describe a generic sequential fixpoint algorithm for Java byte code into which, like in ParCAn, different analyses can be plugged in. Their interface is simpler and targets the analysis of Java code. Conceptually, their analysis-specific plugins could also be compiled for the ParCAn engine to find fixpoints in parallel. (b) Lerner et al. [27] describe a domain specific language for expressing propagation rules of predicates (propagation facts). Programs in this DSL can then be proven correct. Whereas they compile the rules and facts into sequential code, their DSL could also be compiled into ParCAn input.

8 CONCLUSION AND FUTURE WORK
With ParCAn it is possible to express data-flow analyses that require to find a fixpoint and to then run them in parallel on a GPU. Hence, with ParCAn a time-consuming phase of the sequential compilation process can be parallelized. ParCAn is general and can handle inter- and intra-procedural analyses, both with forward and backward orientation. ParCAn works with predicates at a fine-grained instruction level for better utilization of the parallel hardware. Compared to the sequential analyses, for large benchmark codes ParCAn achieves a speedup of up to 82.3 (Escape Analysis), 89.7 (Dead Code Elimination), 85.33 (Global Value Numbering), and 10.3 (Andersen’s Analysis). This saves up to 26.5% of the total compile time on benchmarks.

ParCAn accelerates data-flow analyses by propagating predicates (data-flow knowledge) along the CFG until a fixpoint is reached. Instead of guarding predicates from conflicting concurrent updates, we let corrupted predicates happen (to avoid costly synchronization) and fix the problem later. The crucial optimizations that make ParCAn run fast include (a) a blurred entry to methods, (b) a parallel deslaging of the CFG that removes instructions from the CFG that do not have any impact on the results of an analysis, (c) a sorting of instructions w.r.t. their type to reduce the divergence among the GPU threads, and (d) a synchronization-free intra-warp load-balancing.

Future Work. There are two main ways to further exploit GPU capabilities to make ParCAn run faster. First, so far ParCAn solely operates on global memory and does not exploit the GPU’s memory hierarchy. It does not yet use the fast shared memory, but considers operates on global memory and does not exploit the GPU's memory hierarchy. It does not yet use the fast shared memory, but considers

REFERENCES


