A GPU Implementation of Inclusion-based Points-to Analysis *

Mario Méndez-Lojo[1] Martin Burtscher[2] Keshav Pingali[1,3]

[1]Institute for Computational Engineering and Sciences, University of Texas, Austin, USA
[2] Dept. of Computer Science, Texas State University, San Marcos, USA
[3] Dept. of Computer Science, University of Texas, Austin, USA
marioml@ices.utexas.edu, burtscher@txstate.edu, pingali@cs.utexas.edu

Abstract

Graphics Processing Units (GPUs) have emerged as powerful accelerators for many *regular* algorithms that operate on dense arrays and matrices. In contrast, we know relatively little about using GPUs to accelerate highly *irregular* algorithms that operate on pointer-based data structures such as graphs. For the most part, research has focused on GPU implementations of graph analysis algorithms that do not modify the structure of the graph, such as algorithms for breadth-first search and strongly-connected components.

In this paper, we describe a high-performance GPU implementation of an important graph algorithm used in compilers such as gcc and LLVM: Andersen-style inclusion-based points-to analysis. This algorithm is challenging to parallelize effectively on GPUs because it makes extensive modifications to the structure of the underlying graph and performs relatively little computation. In spite of this, our program, when executed on a 14 Streaming Multiprocessor GPU, achieves an average speedup of 7x compared to a sequential CPU implementation and outperforms a parallel implementation of the same algorithm running on 16 CPU cores.

Our implementation provides general insights into how to produce high-performance GPU implementations of graph algorithms, and it highlights key differences between optimizing parallel programs for multicore CPUs and for GPUs.

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

General Terms Algorithms, Languages, Performance

Keywords Inclusion-based Points-to Analysis, Irregular Programs, Graph Algorithms, GPU, CUDA

1. Introduction

GPU hardware is designed to process blocks of pixels at high speed and with wide parallelism, so it is well suited for executing regular algorithms that operate on dense vectors and matrices. We understand much less about how to use GPUs efficiently to execute *irregular* algorithms that use dynamic data structures like graphs and trees. Harish *et al.* [14] pioneered this field with their CUDA implementations of algorithms such as breadth-first search and single-source shortest paths. BFS has recently received much attention in the GPU community [19, 24, 26]. Barnat *et al.* [5] implemented a GPU algorithm for finding strongly-connected components in directed graphs and showed that it achieves significant speedup with respect to Tarjan's sequential algorithm. Other irregular algorithms that have been successfully parallelized using GPUs are n-body simulations and some dataflow analyses [9, 30].

An important characteristic of most of the irregular algorithms that have been implemented to date on GPUs is that they are graph analysis algorithms that do not modify the structure of the underlying graph [5, 14, 19, 24, 26]; when they do modify the graph structure, the modifications can be predicted statically and appropriate data structures can be pre-allocated for the program [9, 30]. However, there are many important graph algorithms in which edges or nodes are dynamically added to (or removed from) the graph at runtime in an unpredictable fashion, such as mesh refinements [11], compiler optimizations [3], and social network maintenance [7]. In TAO analysis [29], which is an algorithmic classification for irregular codes, these are called *morph* algorithms. Implementation of a morph algorithm on a GPU is challenging because it is unclear how to support dynamically changing graphs on a GPU; in particular, static graph representations such as compressed row storage (CRS), which work well on GPUs, cannot be used.

In this paper, we describe the first high-performance GPU implementation of a very important morph algorithm: Andersen's inclusion-based points-to analysis [3], which is a compiler analysis algorithm that takes a program as input and infers an over-approximation of the set of variables pointed to by each pointer in the program. Inclusion-based points-to analysis provides a good trade-off between precision of results and speed of analysis, and it has been incorporated into several production compilers including gcc and LLVM. A multi-CPU, shared memory implementation of this algorithm is presented by Méndez-Lojo *et al.* [25]; it achieves an average speedup of 6x on sixteen cores relative to a highly-tuned sequential implementation by Hardekopf [13], when analyzing a suite of 14 benchmark programs.

Although our paper focuses mainly on inclusion-based points-to analysis, many of the ideas presented here (especially the graph representation) are applicable to the implementation of other morph codes on the GPU. In addition, our work adds another data point to the ongoing debate regarding the performance of CPU and GPU architectures and their associated programming models [10, 20, 22, 34], and it confirms that modern GPUs can be used to accelerate a wide range of applications.

We summarize the contributions of this paper below.

- A GPU implementation of Andersen's analysis requires fundamental modifications with respect to the CPU code. The modifications include adapting the data structures to the GPU memory model (Section 4), distributing work to threads using novel scheduling policies and avoiding explicit worklists (Section 5), adding new algorithmic features based on primitives (sorts, pre-

* This work was supported in part by NSF grants 111176, 0923907, 0833162,0719966, and 0702353 and by grants from Qualcomm, NEC and Intel. This work was also supported in part by equipment and grants from NVIDIA Corporation.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

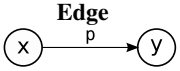
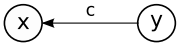
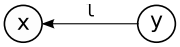
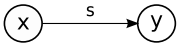
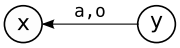
Code	Name	Edge
x=&y	points	
x=y	copy	
x=*y	load	
*x=y	store	
x=y+o	addPtr	

Figure 1. Basic edge types

fix sums) that can be executed very efficiently on the GPU (Sections 5 and 6), and minimizing the overhead penalty derived from exchanging data between the CPU and the GPU (Section 6). This paper can be useful for understanding some of the differences between optimizing codes for multicores and GPUs.

- We describe a graph representation suited for the implementation of morph algorithms on GPUs (Section 4). Our graph data structure is based on 'wide' sparse bit vectors and does not impose any constraint on the structure of the graph, allowing the algorithm to add and remove edges dynamically. This is the first GPU implementation of arbitrary graphs that takes into account three relevant performance factors: global address alignment, shared memory bank conflicts and thread divergence.
- Previous work [25, 31] shows how to formulate Andersen's points-to analysis as a graph rewriting problem. In Section 5, we introduce a modified set of rewrite rules such that any number of rules can be executed simultaneously without synchronization. These new rewrite rules are useful independent of the architecture chosen for the parallelization of the algorithm.
- Our GPU code, written in CUDA, outperforms an existing multi-CPU version of the same algorithm [25], achieving an average speedup of 7x with respect to the state of the art sequential CPU implementation [13].

The rest of this paper is organized as follows. Section 2 introduces Andersen's points-to analysis, formulating it as a graph rewriting system. A brief overview of the GPU hardware and software model is given in Section 3. Sections 4 and 5 describe how to compactly represent sparse graphs for morph algorithms on the GPU and how to implement the graph rewrite rules without resorting to synchronization. Important optimizations tailored to the GPU hardware are discussed in Section 6. Section 7 presents experimental results comparing the performance of the GPU, multi-core CPU, and sequential CPU implementations. Related work is discussed in Section 8. Section 9 summarizes our findings and concludes.

2. Inclusion-based points-to analysis

Points-to analysis is a static analysis technique that determines what a pointer variable may point to during the execution of a program. The results of this analysis are useful for program optimization, program verification, debugging and whole program comprehension [17]. The literature contains many variations of points-to analysis: context-sensitive versus context-insensitive, flow-sensitive versus flow-insensitive, etc. [3, 6, 12, 13, 15, 33, 35]. These variations make different trade-offs between precision and running time, but production compilers like gcc and LLVM seem to have settled on context-insensitive, flow-insensitive points-to analysis because the more precise alternatives are currently intractable for very large programs.

2.1 Andersen-style points-to analysis

The most popular algorithm for context-insensitive, flow-insensitive points-to analysis is known as inclusion-based or Andersen-style analysis [3]. The asymptotic worst-case complexity of the algorithm is $O(n^3)$, where n is the number of variables in the program, although this worst-case behavior is rarely observed in practice: there is a plethora of heuristics (e.g., [12, 13, 32]) that dramatically speed up the analysis.

Traditionally, inclusion-based points-to analysis is formulated as a set-constraint problem. Each statement in the input program adds a new constraint to the system, which is iteratively solved until a fixpoint is reached. However, many constraint problems can also be formulated in terms of graph rewriting rules [16, 31]. We now describe a graph-based formulation [25] of Andersen's analysis.

1. *Initialization.* The input program is read, discarding any statement not related to pointer manipulations. Since we assume that we are analyzing C programs, there are five statements of interest: $x = \&y$ (points), $x = y$ (copy), $x = *y$ (load), $*x = y$ (store), and $x = y+o$ (pointer arithmetic, abridged as 'addPtr').
2. *Constraint graph creation.* For each pointer variable in the input program, we add a new node to a *constraint* graph, which is the only data structure required by this particular formulation of the analysis. For each pointer-related statement, we add an edge as indicated in Figure 1. Note that the resulting graph might contain multiple edges of different types between two given nodes. An example is shown in Figure 3. The program contains five variables and four statements, so the initial constraint graph in Figure 3(a) has five nodes and four edges.
3. *Solving constraints*. Most of the analysis time is spent in this phase, in which we repeatedly apply a set of four rewrite rules in any order. The rules are listed in Figure 2. Intuitively, each rewrite rule updates the graph locally to satisfy some constraint. For brevity, we will only cover the intuition behind the copy rule; a more formal explanation of each rule can be found elsewhere [25]. The copy rule states that if variable y has an outgoing points edge to z and an outgoing copy edge to x, then an edge of type points must exist between x and z. In other words, the rule augments the points-to set of x by adding one variable that is already present in the points-to set of y. Newly added edges are shown using dashed lines. The formula in the last row indicates the postcondition that will hold once we have applied all the copy rules involving x and y: the points-to set of y is a subset of that of x.

   Notice that each rewrite rule is triggered if there is a node with two outgoing edges at which the relevant invariant is not satisfied because of a missing edge between two variables in the constraint graph. Such a node is called an *active* node. In Figure 2, the active node for each rule is shaded. When an active node is processed and a new edge is added to the graph, it may cause other nodes to become active. There may be many active nodes in a given constraint graph, a fact that we exploit in the parallel algorithm described in Section 2.2.

When no more graph rewrite rules can be applied, the process terminates. Termination is ensured because the process only adds new edges to the constraint graph, and there is only a finite number of edges that can be added. The solution to the points-to problem can be read off the points-to subgraph. It can be proven that the resulting solution is equivalent to the one obtained by solving a system of constraints.

An example of this graph-based analysis is illustrated in Figure 3. In the initial state (a), there are two active nodes, x and z. We choose to apply the copy rule for z first, adding a new points edge $y \xrightarrow{p} w$. Now x is the only active node, firing a store rule that

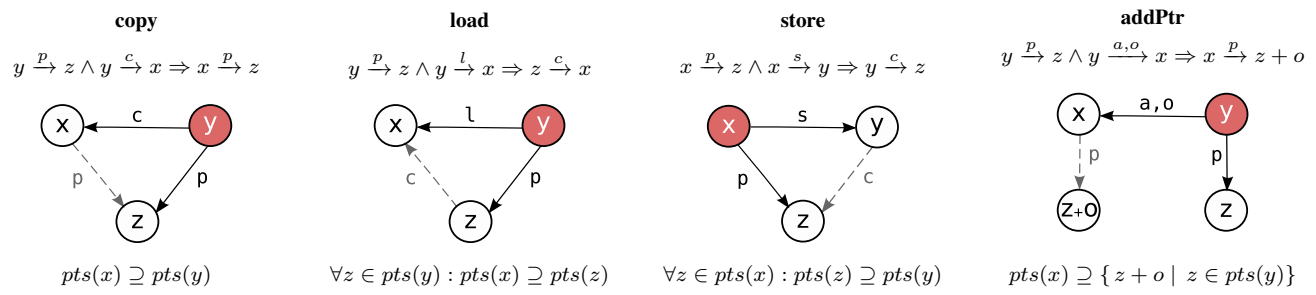


Figure 2. Constraint graph rewriting rules

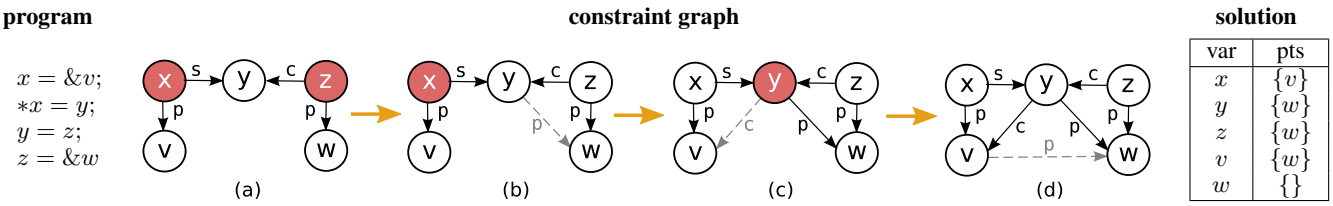


Figure 3. Graph rewrite example

adds $y \xrightarrow{c} v$. After applying another copy rule (transition from (c) to (d)), the preconditions of all the rewrite rules are satisfied and we reach a fixpoint. The points-to solution is shown on the right hand side of Figure 3.

2.2 Parallelism in Andersen's points-to analysis

Graph rewrite rules can be applied concurrently, provided that the graph data structure is properly synchronized such that edges can be added to it in a concurrent fashion. To understand this simple parallelization scheme, consider the possible scenarios that can happen when a rule R is adding an edge from x to y:

- Another rule reads an edge that starts at node x. This edge will not be removed by R because no edges are removed.
- Another rule adds an edge $x \rightarrow z$ such that $y \neq z$. This new edge cannot affect the update rule executed by R because it does not depend on that edge. On the other hand, the concrete representation of the edge set needs to be synchronized so it supports concurrent additions. For example, in Figure 3(a), there are two active nodes x and z, and both are trying to add a new outgoing edge to y. In a parallel setting, we allow the two rules to perform the addition concurrently.
- Another rule tries to add the *same* edge. The two rules can be interleaved in any fashion and the final state will be the same. The work performed by one of the activities is redundant, but it is irrelevant which one actually performs the edge addition.

The parallelism in inclusion-based points-to analysis is a particular example of *amorphous data-parallelism*, a generalization of data-parallelism that is ubiquitous in irregular programs [29]. Note that this parallelism is independent of the programming model or the underlying parallel hardware: it is a property of the algorithm.

3. GPU architecture and programming model

We briefly describe the micro-architecture of modern graphics processors and the CUDA programming model for using them. Although we focus on NVIDIA GPUs, the concepts discussed here also apply to other similar architectures.

The Fermi architecture [1] on which our work is based consist of up to 16 identical streaming multiprocessors (SMs), each of which contains 32 tightly coupled processing elements (PEs) that are sometimes called CUDA cores. Whereas each PE is able to run an independent thread of instructions, all 32 PEs in an SM must either execute the same instruction in the same cycle or wait. This Single Instruction Multiple Thread (SIMT) execution model is tantamount to running instructions that conditionally operate on 32 individual data items. A set of 32 threads that run together in this manner is called a *warp*.

Warps are automatically subdivided by the hardware into sets of threads that want to execute the same instruction. The sets are then serially executed until they re-converge, which degrades performance. Therefore, it is very important to avoid *thread divergence*, i.e., situations where not all threads follow the same control flow, as occur in certain if-then-else and looping statements.

Up to 48 warps can simultaneously be resident in an SM. The PEs execute the warps in multithreading style to hide latencies, that is, the PEs in an SM are time-shared among the warps. Because only one warp is actively executed in any one cycle, threads belonging to different warps can execute different instructions. The PEs do not support out-of-order execution within threads but are able to arbitrarily interleave warps. Hence, it is important to have a large number of warps running concurrently to extract the full performance of the GPU.

The memory subsystem is also optimized for warp-based execution. If the threads in a warp simultaneously access words in main memory that lie in the same aligned 128-byte segment, the hardware merges the 32 reads or writes into one *coalesced* memory transaction that is as fast as accessing a single word. But if a warp requests 32 scattered words, the hardware has to perform 32 separate memory transactions. Thus, coalesced memory accesses are crucial to achieve a high memory bandwidth.

The PEs within an SM share a pool of parallel threads called *thread block*, synchronization hardware, an L1 data cache, and a software-controlled cache called *shared memory*. The shared memory is as fast as the L1 data cache and allows threads in a thread block to quickly exchange data. A warp can simultaneously access 32 words in shared memory as long as the words reside in different banks or all accesses within a bank request the same word.

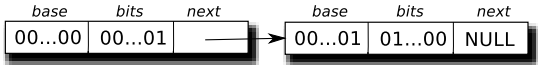


Figure 4. Sparse bit vector representing $\{0, 62\}$ (*bits* is assumed to be 32 bits wide)

Otherwise, bank conflicts occur that result in (partial) serialization of the 32 accesses.

The SMs operate largely independently. They can only communicate through global memory (DRAM). Thus, synchronization between SMs must be accomplished using atomic operations on global memory locations.

NVIDIA's CUDA programming model extends the C/C++ programming language with several parallel programming primitives to exploit the architectural capabilities of GPUs. A CUDA program consists of host code running on the CPU and device code running on the GPU. The device code includes one or more functions, called *kernels*, that can be invoked by the CPU.

4. Graph representation on the GPU

Creating an efficient data structure to represent the constraint graph under the GPU memory model is a challenging problem. The data structure has to compactly represent millions of edges (the analysis of the linux kernel results in a constraint graph with 1.498 billion edges) and allow dynamic modifications. At the same time, the memory layout of the graph has to be specifically designed for the GPU architecture to minimize memory transactions, maximize coalescing, and avoid divergence within the threads of a warp.

One feasible representation for the constraint graph is an adjacency matrix. For instance, the points-to graph can be represented by an $n \times n$ dense matrix (where n is the number of variables in the program); call this matrix P. If we assign a unique id to each variable in the program, then $P(i, j) = 1$ if the variable with id i points to the variable with id j. In a similar fashion, matrices C, S, L and A would represent the other types of edges in our problem.

The matrix representation has one major advantage: the graph rewrite rules can be expressed in terms of matrix-matrix multiplications, which can be performed quickly on a GPU (NVIDIA provides the CUBLAS library for this purpose). For example, we can apply all the available copy rules at once and update the points-to matrix P by computing $P = P + C^t * P$.

The disadvantage of this approach is that it wastes a lot of space since graphs in this application are very sparse. We computed the initial and final density of the P and C matrices using the points-to algorithm of Hardekopf [13] to analyze three inputs: the gcc compiler, the vim editor and the linux kernel. Densities are calculated as the number of non-zero entries in the corresponding matrix (i.e., the number of edges in the graph) divided by the total number of entries, which is n^2. It can be seen that both the initial and final matrices are very sparse. (The matrices for load, store and pointer arithmetic edges are also extremely sparse.)

input	P_i	P_f	C_i	C_f
gcc	$5 * 10^{-7}$	$6 * 10^{-4}$	$6 * 10^{-6}$	$4 * 10^{-5}$
vim	$2 * 10^{-7}$	$8 * 10^{-4}$	$10 * 10^{-7}$	$2 * 10^{-5}$
linux	$1 * 10^{-7}$	$2 * 10^{-3}$	$2 * 10^{-7}$	$2 * 10^{-4}$

An alternative representation tailored to sparse graphs is the Compressed Sparse Row representation. The limitation in this case is that Andersen's analysis, like other morph codes, dynamically adds new edges to the graph. Since the final number of neighbors for each node in the constraint graph cannot be statically predicted and can vary dramatically from variable to variable, adjacency list-based representations like CSR are not adequate.

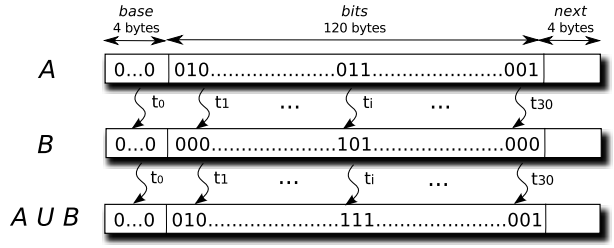


Figure 5. Union of two sparse bit vector elements on the GPU using a warp

One representation of sparse graphs that allows the addition and removal of new edges is based on sparse bit vectors. A sparse bit vector is a data structure that compactly represents sets of integers. Internally, it is a linked list in which each element contains three fields: the *base* of that element, the *bits*, and a pointer to the next element. The base indicates the range of integers possibly contained in the current element; the bits indicate whether a particular integer belongs to the set or not.

Figure 4 shows the representation of a set of integers P using a sparse bit vector. The *bits* field is 32 bits wide, so each element can store up to 32 integers. The first element of P has base 0, so it can only contain numbers between 0 and 31. Since the rightmost bit is set, 0 is in P. Since the 30^{th} bit of the element with base 1 is also set and 32*1+30=62, we have $P = \{0, 62\}$. By assigning unique integer identifiers to each variable in the program, a sparse bit vector can be used to represent the set of neighbors of a given variable. In our example, if P represents the points-to set of variable x, then $pts(x) = \{0, 62\}$. Sparse bit vectors have been used in some CPU implementations of points-to analysis ([13, 15, 23], among others).

The sparse bit vectors used in our implementation occupy 128 bytes per element. The base and the next pointer use one word each; the rest of the space is dedicated to the *bits* field. Therefore, each element can hold up to 960 integers. The 32-word width matches the GPU memory bus. Assuming that the elements are 128-byte aligned, bringing one element from global into shared memory requires exactly one transaction: thread i ($i \in \{0..31\}$) brings in the i-th word. Once the element is in shared memory, each thread of the warp can manipulate its own word without causing any bank conflicts. Finally, many set operations can be performed concurrently by all the threads within a warp with little divergence.

Consider, for example, the union of two sparse bit vector elements with the same base, defined as the bitwise OR of their respective *bits* fields: only the thread that corresponds to the *next* word will diverge, since performing an OR of two identical bases does not change the base. Set intersection can be implemented in a similar fashion using the logical AND operation. A visual representation of the union operation is depicted in Figure 5. It takes one memory transaction (400-800 cycles [2]) to transfer each element from global to shared memory, one cycle to do the bitwise OR of the two elements, and another transaction to write the result back to global memory.

The 128-byte element representation we propose can waste large amounts of memory. For instance, if we want to represent a singleton set using a standard bit vector element (in many CPU implementations [13, 23], the *bits* field is 4 bytes-wide) we need only 12 bytes, ten times less space than what the wide representation requires. However, sets containing large sequences of contiguous integers benefit from wider elements: the set $\{0, .., 959\}$ occupies 128 bytes, while the standard representation requires 360 bytes (thirty elements) of storage.

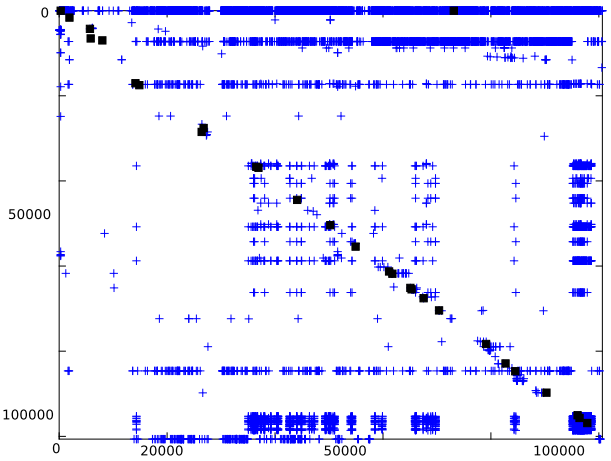


Figure 6. Adjacency matrix of the gcc points-to graph

Using wide elements can also result on a performance penalty when most of the words in the *bits* field are zero, since the amount of useful work performed by the threads within a warp diminishes. Consider the example in Figure 5: if a thread is performing bitwise operations such as union or intersection on two empty words, then it is basically working on data that is not useful to the algorithm.

We compared the storage needed by the wide and standard bit vector representations when storing the final constraint graph. We used Hardekopf's points-to analysis [13] and the same inputs as in the experiments in Section 7. On average, the GPU-tailored sparse bit vectors use 2.3x more space. The reason why using wider elements is fairly efficient is that the analyzed programs exhibit spatial locality. If a variable points at some other variable i, then there is a high probability that it also points to variables with identifiers close to i. This holds because identifiers are sequentially assigned to variables as they appear in the program, so variables in the same block/function receive similar ids.

This distribution can be observed in Figure 6, in which we plotted the adjacency matrix for the points-to edges at the beginning (solid squares) and at the end (shaded crosses) of the analysis. Since Andersen's algorithm does not removes edges, the initial edges are also part of the final graph. The input is gcc, which has 120K variables. As we can see in Figure 6, the initial points-to subgraph corresponds almost perfectly to a diagonal adjacency matrix: variables point to others that appear close together in the program. After repeatedly applying the rewrite rules, there is a clustering effect in the points-to sets: once a variable is determined to possibly point to another variable in some other block, then it probably might point to its aliases, too. In Figure 6, we also observe that some nodes have a large number of incoming points-to edges; they correspond to global variables and allocation sites.

In summary, the proposed graph representation is a good fit for the implementation of morph algorithms on a GPU. Furthermore, it is the first implementation of arbitrary graphs we are aware of that takes into account global address alignment, shared memory bank conflicts and thread divergence. Since we expect that the graph being manipulated by other algorithms (in particular, flow analyses) will also be sparse and share the same locality characteristics as in our application, we believe that irregular codes such as [30] could benefit from our representation.

5. Parallel rule application on the GPU

A parallel CPU implementation of Andersen's algorithm depends on multiple features that have been studied in depth in the context

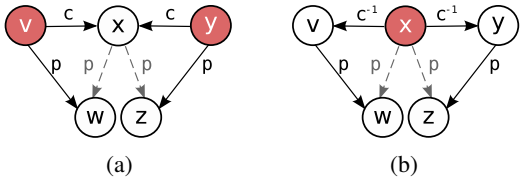


Figure 8. Simultaneous application of copy rules requires synchronization, which can be avoided by reversing the copy edges

of that particular hardware model: concurrent data structures, work schedulers, dynamic memory allocators, etc. However, there is no standard GPU counterpart for many of these basic building blocks. In this section, we present solutions for some of these problems.

Parallel execution of the rewrite rules in Figure 2 requires synchronization in the graph data structure. An example is shown in Figure 8(a): both copy rules simultaneously try to add an outgoing edge to node x, so synchronization is needed. Modern GPUs support atomic compare and swap operations, but overusing them may result in a substantial performance penalty. We devised a novel algorithmic solution to dramatically reduce the amount of synchronization needed to implement Andersen's analysis.

Figure 8(b) shows the intuitive idea: instead of storing an outgoing copy edge in the source variable, we store the *reversed* edge, which we denote by c^{-1}, in the destination node. The copy rule is adapted for this new type of edge, and now we add $x \xrightarrow{p} z$ if there exists a path $x \xrightarrow{c^{-1}} y \xrightarrow{p} z$. The modified rewrite rule is called a reversed copy rule, or copy^{-1} rule. Note that the only active node in Figure 8(b) is x. The benefit of the new formulation is that, as long as there are no two concurrent rules working on the same active node, we do not depend on synchronization: active nodes only add outgoing edges to themselves.

The new set of rewrite rules is shown in Figure 7. They require flipping the copy, load, and pointer arithmetic edges. The store^{-1} rule also depends on storing the incoming points-to edges. Modifications made by a 'reversed' rewrite rule are now local: the edge is added to the active node of that rule.

The distribution of work to threads is done in a warp-centric manner, in a very similar way to [18]. Each active node is assigned to a warp, which executes all the possible rules of a specific type. The selected level of granularity seems to be adequate: a) using an entire block to process and active node will result in many idle threads since there is very little work to be done for some variables, b) using one thread will result in high intra-warp divergence (poor performance) since the number of rewrite rules that need to be applied is not uniform across active nodes.

The pseudo-code of Andersen's algorithm on the GPU is shown in Figure 9. The comments indicate whether the code is being executed on the CPU, GPU or is a data transfer between the two devices. The input is read on the CPU and then transferred to the GPU, where we create the initial constraint graph (*initialize* kernel). Then, we repeatedly apply each reversed graph rewrite rule (*rule* kernel) on the GPU until the constraint graph reaches a fixpoint. The termination condition is verified on the CPU by first transferring a Boolean variable from the global memory of the GPU. When the process terminates, we copy the solution (i.e., the points-to edges) to the CPU. Note that the rest of the constraint graph is necessary for the solving phase but is not part of the output of this algorithm.

The *rule* kernel is executed entirely on the GPU. Each warp is assigned a variable x and then applies the transitive closure to the edges of the specified types. Multiple warps will never work on the same variable because variables are assigned by atomically

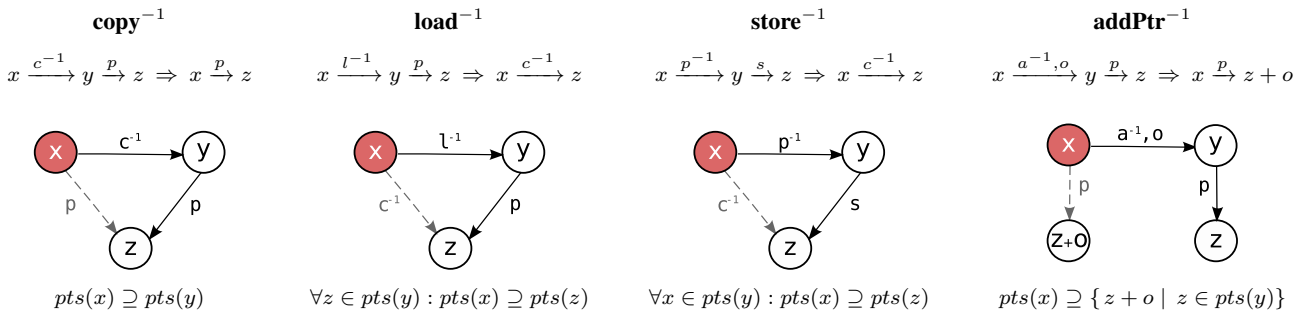


Figure 7. Constraint graph rewriting rules, modified to avoid synchronization

```
andersen():
  read input                          // CPU
  transfer initial constraints        // CPU→GPU
  initialize_kernel()                 // GPU
  do
    rule_kernel(C^-1, P, P)           // GPU
    rule_kernel(L^-1, P, C^-1)        // GPU
    rule_kernel(P^-1, S, C^-1)        // GPU
    rule_kernel(A^-1, P, P)           // GPU
    transfer changed                  // GPU→CPU
  while changed                       // CPU
  transfer P                          // GPU→CPU

rule_kernel(R, S, T):                 // GPU
  foreach x in variables
    if R ≠ A^-1
      foreach x --R--> y
        union S–neighbors of y to T–neighbors of x
    else
      foreach x --a^-1,o--> y
        N ← add o to each S–neighbor of y
        union N to T–neighbors of x

    if T–neighbors of x changed
      changed ← true
```

Figure 9. Pseudo-code of Andersen's algorithm on the GPU

incrementing a global integer. Thus, our algorithm does not need an explicit worklist for the active elements (unlike most of the CPU implementations we know of); instead, we simply check if it is possible to apply a given rule to *all* variables. Since there are many warps (see Section 7) executing concurrently, the overhead of processing non-active nodes is almost negligible.

The transitive closure is implemented as follows. Given a variable x, we traverse the set of neighbors for the first relation (variable R in the pseudo code). The traversal implies decoding the sparse bit vector representing the outgoing edges of that type. For each neighbor y, we union the sparse bit vector containing all its S-neighbors with the sparse bit vector that contains the T-neighbors of x. The pointer arithmetic rule requires an extra step, since we need to add the offset to each points-to neighbor of y before performing the union. The computation of the union of two adjacency lists is explained in Section 4.

The union of two adjacency list might result in the addition of new elements to the sparse bit vector representing the adjacency list of a particular combination of variable and edge type. Although recent CUDA implementations offer dynamic memory allocation [2], we created a custom allocator. We divided a region of global memory into two *element pools*. One is dedicated to the points-to edges and the other is used for all other types of edges. Each time a warp has to allocate a new element, it simply atomically increments the pointer to the next free element in the corresponding pool. The division of the heap into two regions has two advantages: a) the more regions we have, the lower the contention on the free list pointers is and, more importantly, b) all elements containing points-to edges are stored together. When the analysis terminates, we can minimize the amount of data transferred from the GPU to the CPU by copying only the points-to region.

6. Optimizations

We now describe several GPU-based optimizations that dramatically improve the performance of our implementation.

6.1 Minimize memory consumption

The store^{-1} rewrite rule in Figure 7 introduces a performance problem since it depends on also storing the reversed (incoming) points-to edges, which can be prohibitive in terms of memory usage. In order to avoid this, we use a different, two-phase strategy to implement the store rule. In the first phase, we create a worklist containing all pairs of variables (x, y) such that y has outgoing store edges, and $y \xrightarrow{p} x$ is in the constraint graph. In the second phase, we assign all pairs with an identical first component to the same warp. Since an active node is processed by only one warp, there is no synchronization required except for removing elements from the worklist. Creating an explicit worklist to handle store edges may seem expensive, but it performs well in practice because the number of store edges is very small for most input programs (about 5% of the edges in the final graph).

6.2 Avoid redundant rule application

The graph rewrite rules in Figure 7 need to be applied only once. For instance, the copy^{-1} rule does not need to be fired in a particular iteration if the points-to sets of all the variables have not changed during the last iteration of the main loop in the pseudo-code in Figure 9.

A possible solution to avoid repeated work is to distinguish between two types of points-to edges: the ones that have been added to the constraint graph *before* the last iteration (P) and the ones added during the last and current iterations (ΔP). The distinction results in two major modifications of the algorithm: a) the graph rewrite rules are now defined in terms of edges in ΔP, not P, and b) we need an additional kernel that performs the updates $\Delta P = \Delta P - P$ and $P = P \cup \Delta P$.

The idea of working exclusively on the newly added edges is not novel [15]. However, the GPU implementation benefits from it in two distinct ways:

- At the beginning of each iteration, we transfer ΔP from the GPU to the CPU in parallel with the execution of the rewrite kernels using streams [2]. This approach completely hides the transfer latency and greatly reduces the overall runtime.

	K			V		
	$\{a, c\}$	$\{b\}$	$\{a, c\}$	x	y	z
hash(K)	38	12	38	x	y	z
sort(K,V)	12	38	38	y	x	z
diff(K)	0	1	0	y	x	z
prefix(K, max)	0	1	1	y	x	z

Figure 10. Example of detection of ΔP-equivalent variables

- Computing differences between sets of edges (i.e., differences between sparse bit vectors) can be efficiently implemented using a warp-based approach that is similar to the union operation.

6.3 Detect pointer-equivalent variables

ΔP-equivalent variables have the same outgoing ΔP edges in the current iteration. It is desirable to identify ΔP-equivalent variables, since much redundant work can be avoided. For example, assume that the $\Delta P(x) = \{a, c\}$, $\Delta P(y) = \{b\}$ and $\Delta P(z) = \{a, c\}$. If the three variables are copy^{-1} neighbors of some other node in the constraint graph, then applying the copy^{-1} rule to x (or z) and then to y produces the same result as applying it to all three variables because the new points-to sets of x and z are identical.

Detection of ΔP-equivalent variables is extremely efficient on the GPU. We illustrate the mechanism with an example. In Figure 10, we have a map containing keys (ΔP) and values (variables). The key at column i corresponds to the value at the same column (e.g., $\Delta P(x) = \{a, c\}$). We first compute a hash value for the keys and then sort both keys and values according to the hash. Then we apply a difference function between keys such that $K'(i) = i$ if $K(i-1) \neq K(i)$ or $i = 0$. The final step computes a prefix sum of the keys, using the maximum operator: $K'(i) = max(K_0, ..., K_i)$. The final map verifies that the variable at column i is ΔP-equivalent to the one at column $K(i)$. For example, variable z has the same ΔP as variable $V(K(2)) = V(1) = x$.

Detection of ΔP-equivalent variables is implemented using the Thrust library [28], which supports fast data-parallel operations such as sorting, prefix sums, and reductions. It is interesting to note that the multi-CPU implementation of [25] does not try to detect ΔP-equivalent variables, and it is not clear whether the described mechanism will perform well on a CPU.

6.4 Collapse cycles

When two or more variables belong to a cycle of copy^{-1} edges, they are pointer-equivalent (i.e., their points-to sets will be identical by the end of the analysis) and the corresponding strongly connected component can be collapsed. For example, a pair of statements of the form *a = b; b = a;* produces a cycle of constraints that imply that *pts(a) = pts(b)*. In the literature, cycle detection comes in two flavors: *offline* methods [32] look for cycles during a preprocessing phase whereas *online* methods [12] look for cycles during the solving process.

Some intermediate techniques, such as Hybrid Cycle Detection [13] (HCD), combine the two: potential cycles are identified in the offline phase, and these are collapsed during analysis. Potential cycles arise from statements of the form **a = b; b = *a;*. Without knowing *pts(a)*, we do not know the nodes that participate in cycles with *b*, so these cycles cannot be eliminated during preprocessing, but we can remove them during the constraint solving process whenever we add nodes to *pts(a)*.

Our implementation of Andersen's analysis uses only Hybrid Cycle Detection. Offline techniques greatly improve the performance of the sequential version, but they often introduce a bottleneck to scalability in the parallel codes. The offline phase of HCD is executed on the CPU. The online phase is implemented as a GPU kernel, and it implies merging variables that are pointer equivalent (i.e., variables that belong to the same strongly connected component), by selecting one *representative* node and adding to it all the outgoing edges of the non-representative variables.

Cycle collapsing seems to require implementing two extra graph operations: node and edge deletion. An alternative approach that performs well in practice is to ignore non-representative variables and their edges. A warp only processes variables identified as representative in a *representative table* in global memory.

program	vars	stmts	program	vars	stmts
ex	11	13	vim	246	108
perl	54	68	php	339	325
python	92	111	mplayer	537	377
nh	97	114	gimp	558	649
svn	107	139	pine	612	315
gcc	120	156	linux	1,503	420
gdb	232	241	tshark	1,555	1,789

Figure 11. Benchmark suite: number of variables and statements (in thousands)

7. Experimental evaluation

This section compares the performance of Andersen's analysis on the GPU with two previous CPU implementations: a sequential version by Hardekopf [13] and our multi-CPU version [25]. The source code of the multi-CPU and GPU analyses is available at `http://clip.dia.fi.upm.es/~mario/`. In the rest of this section, we refer to the multi-CPU version as the *reference* implementation.

The reference implementation is very similar to the sequential analysis, except for the necessary synchronization on the data structures and minor algorithmic modifications. However, the GPU implementation introduces major algorithmic changes, as described in the previous sections. Another important difference is the use of a Binary Decision Diagram [8] data structure. The benefits of BDDs in the context of points-to analysis have been touted by many researchers [6, 35]. The reference implementation uses a BDD to compactly represent the points-to edges, while all the other types of edges are internally represented using sparse bit vectors. In contrast, our implementation only uses 'wide' sparse bit vectors because BDDs are extremely complex and ill-suited for GPUs.

Figure 11 shows the benchmark suite used in our experiments. It consists of fourteen C programs ranging from 11K to 1555K variables (nodes in the constraint graph) and 13K to 1789K statements (initial edges). Most of the programs in our benchmark suite have been used by other researchers in this area [13, 25].

We evaluated the performance of the CUDA implementations on a 1.15 GHz NVIDIA Tesla C2070 GPU with 14 streaming multiprocessors (448 processing elements, i.e., CUDA cores) and 6 GB of main memory. This Fermi GPU has a 64 KB L1 cache per SM. We dedicate 48 KB to shared memory (user-managed cache) and 16 KB to the hardware-managed cache. All the streaming multiprocessors share an L2 cache of 768 KB. We compiled the CUDA code with nvcc v4.1 RC2 and the -arch=sm_20 flag.

To execute the CPU codes, we used a machine running Ubuntu 10 with four 4-core 2.7 GHz AMD Opteron processors. The 16 CPU cores share 24 GB of main memory. Each core has a 64 KB L1 cache and a 512 KB L2 cache. Each processor has a 6 MB L3 cache that is shared among its four cores. The sequential implementation is written in C++ and compiled with gcc and the -O3 flag. The reference implementation is written in Java on top of the Galois framework [21]. The Java Virtual Machine used is the 64-bit Sun HotSpot server version 1.6.0_24.

input	CPU-s	CPU-1	CPU-16	GPU
ex	400	3.17	1.54	**5.00**
gcc	1,000	1.20	**4.63**	3.57
nh	1,280	1.22	5.54	**6.74**
perl	1,990	1.12	6.18	**6.22**
vim	10,110	1.30	**9.39**	1.28
tshark	12,110	0.89	3.53	**5.13**
svn	14,630	0.96	5.70	**10.09**
python	17,890	0.85	3.99	**14.54**
gimp	20,500	0.92	**7.83**	3.45
gdb	31,300	0.90	6.95	**9.40**
pine	38,950	0.92	4.93	**5.21**
php	44,670	0.86	5.97	**6.54**
mplayer	66,260	0.83	6.07	**7.97**
linux	120,340	1.05	7.67	**10.39**

Figure 12. Runtimes (in ms) for the sequential online phase (CPU-s column), and speedups achieved by CPU-x and GPU

Each GPU kernel can be configured with respect to the number of blocks and the number of threads per block it uses. Having many threads per block seems to be a good choice, since communication among threads within the same block is cheaper (they can communicate through the local memory of the streaming multiprocessor). However, the hardware imposes limits on the number of threads per block (1024 in Fermi GPUs). Other factors that impose constraints on the number of threads per block are the register and the shared memory usage. The number of persistent blocks and threads per block used by the most relevant kernels in our implementation is shown in the following table.

kernel	blocks	threads
update P, ΔP	14	1024
cycle collapsing (HCD)	14	512
copy^{-1} / load^{-1} / store^{-1}	14	864
addPtr^{-1}	14	1024

Since the GPU used in our experiments has fourteen SMs, we use that many blocks. Within each block, the thread count is not always maximized since operations on sparse bit vectors heavily rely on caching of data in the fast shared memory. In our warp-based approach, a kernel with 1024 threads (32 warps) restricts the shared memory usage of each warp to just 1,536 bytes.

The first set of results, shown in Figure 12, contains the online analysis times (in milliseconds) for each program in the benchmark suite using the sequential analysis, denoted by $CPU\text{-}s$. We sorted the inputs according to the sequential analysis runtimes, which are not always proportional to the number of variables or statements. The other columns list the speedups achieved by the reference implementation using x threads ($CPU\text{-}x$) and the implementation discussed in this paper (GPU). In the case of $CPU\text{-}x$, we only show the results for one and sixteen threads. The best speedups are marked in bold.

In the case of $CPU\text{-}s$ and GPU, each benchmark was run three times (there is very little variability) and the median runtime (or its speedup) is reported. In the case of $CPU\text{-}x$, in order to minimize the effects of JIT compilation, each benchmark was run five times, and the speedup achieved by the median runtime is reported. We also minimized the influence of garbage collection in the $CPU\text{-}x$ results by maximizing the size of the JVM heap to the point where the measured time spent in GC is always zero. Finally, we verified that the three outputs (points-to of every variable in the original program) are identical.

The first observation about Figure 12 is that the reference implementation has better scalability than what was previously reported [25]: the average scalability when using 16 threads is 6.07x

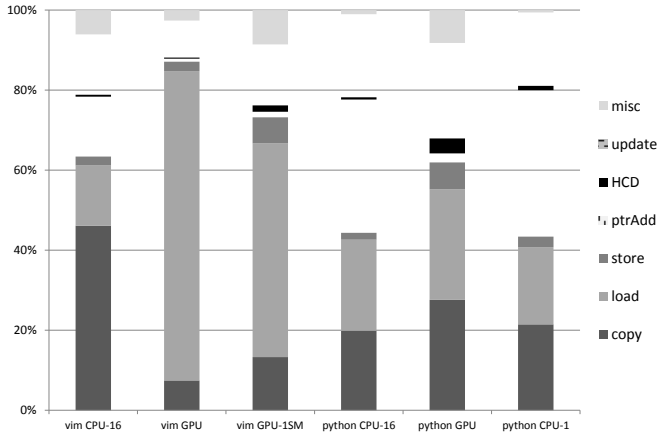


Figure 13. Breakdown of the online analysis times for the vim and python benchmarks

if we exclude the ex benchmark, which is an obvious outlier because of its small size. Another observation is that the multi-CPU implementation using one thread is faster than the sequential version for some of the inputs, which is counter-intuitive. This is due to minor algorithmic changes in the multi-core version, which reduce the memory usage and simplify the scheduling policy [27] (i.e., how elements are removed from the worklist).

The GPU implementation performs remarkably well. For all the inputs, it is significantly faster than the sequential analysis. For 11 out of 14 benchmarks, it is also faster than the best reference runtime, which always happens at the highest thread count, except for ex. Although the results indicate that GPU outperforms $CPU\text{-}x$, we acknowledge that establishing a completely fair comparison between the two parallel codes is difficult: there are remarkable differences at the algorithmic, language and hardware levels. For instance, the reference implementation is written in Java whereas the GPU code is written in low-level CUDA. From the hardware perspective, the two machines used in our experiments have a similar number of processing units (fourteen streaming multiprocessors and sixteen cores, respectively), but the CPU cores are significantly faster, exploit instruction-level parallelism, and contain large caches. In any case, it might come as a surprise that CUDA codes can be competitive with highly-tuned multi-core implementations for algorithms as irregular as Andersen's points-to analysis.

Our experimental data reveal the different behavior of the two parallel analyses. For instance, the CUDA code performs significantly worse for the vim input (8x slowdown with respect to CPU-16) but much better for python (3.5x speedup). We broke down the online analysis times of these two benchmarks to understand the cause of the performance differences. The results are shown in Figure 13. We divided the algorithm into seven components: the rewrite rules, updating P and ΔP, cycle collapsing (HCD), and a miscellaneous category that includes worklist accesses ($CPU\text{-}x$) and detection of pointer-equivalent variables (GPU).

The left hand-side of Figure 13 shows the breakdowns of the time spent analyzing vim by CPU-16, GPU and GPU-1SM, which is identical to GPU except for the fact that only one streaming multiprocessor is active. This last bit of information is useful for finding scalability issues; for instance, the breakdown of GPU looks similar to that of GPU-1SM, and both reveal that the GPU implementation spends a lot of time computing load rules. In this particular case, we found that the data representation was responsible for the slowdown. Operations involving BDDs can be cached because the operands have canonical representations. The memo-

	CPU-s		CPU-16			GPU		
input	offline	online	offline	online	speedup	offline	online	speedup
ex	20	400	73	259	1.27	73	80	**2.75**
gcc	340	1,000	210	216	**3.15**	210	280	2.73
nh	270	1,280	156	231	4.01	156	190	**4.48**
perl	160	1,990	121	322	4.85	121	320	**4.88**
vim	250	10,110	153	1,077	**8.42**	153	7,870	1.29
tshark	3,090	12,110	1,567	3,432	3.04	1,567	2,360	**3.87**
svn	210	14,630	188	2,568	5.38	188	1,450	**9.06**
python	220	17,890	167	4,488	3.89	167	1,230	**12.96**
gimp	1,110	20,500	634	2,618	**6.65**	634	5,950	3.28
gdb	490	31,300	265	4,502	6.67	265	3,330	**8.84**
pine	670	38,950	333	7,900	4.81	333	7,470	**5.08**
php	620	44,670	352	7,486	5.78	352	6,830	**6.31**
mplayer	750	66,260	375	10,921	5.93	375	8,310	**7.72**
linux	1,210	120,340	543	15,685	7.49	543	11,580	**10.03**

Figure 14. Comparison of runtimes (in ms) for the whole analysis: CPU (sequential), CPU (parallel, 16 threads), and GPU

ization is very useful when the same operation is applied over and over during analysis, as it happens in vim. In fact, disabling this cache in the reference implementation results in an average slowdown of almost two orders of magnitude.

The breakdown of python (right hand-side of Figure 13) shows that, in the reference implementation, there is no phase with major scalability issues since the CPU-16 plot is almost identical to that of CPU-1. Instead, the measured performance slowdown (with respect to GPU) is caused by the slower execution of the pointer arithmetic rules. Adding an integer ($offset$) to every element in a set is a highly data-parallel operation when its internal representation uses sparse bit vectors: a left shift of the $bits$ field by $offset$ positions. The CPU implementations cannot use the same approach because points-to sets are represented with a BDD.

Figure 14 compares the *total* analysis runtimes, which do not include the time spent in reading the inputs from disk. We show the runtimes of the sequential, the reference (using sixteen threads), and the GPU implementation. For each version, we show the time spent in the offline and online phase. As explained in Section 6, we only use the Hybrid Cycle Detection technique. Since its offline component is always executed on the CPU, the runtimes in the *offline* column are identical for GPU and CPU-16. An important observation is that the offline phase is significantly faster in the parallel implementations because it has been partially parallelized. Finally, the *speedup* column is the total runtime of the corresponding parallel implementation divided by the sequential total runtime. The best speedups are marked in bold.

The GPU total runtimes do include the time involved in exchanging information between the CPU and the GPU. Although in many GPU algorithms this transfer represents a major bottleneck for the overall performance, our implementation overlaps the transfer of the points-to subgraph with the execution of the rewrite rules, cf. Section 6. The time to transfer the initial constraint graph from the CPU to the GPU is negligible (never more than 10 ms).

The average speedup of the reference implementation is 6x; the GPU code achieves a 7x average speedup. These results are remarkable given that some phases of the offline optimizations such as the detection of Strongly Connected Components (SCC) are still sequential. It is future work to implement the offline phase in CUDA and evaluate the benefit of using GPU versions of the SCC algorithm [5].

8. Related Work

There have been numerous implementations of parallel graph algorithms using various computer architectures, including distributed memory supercomputers [36], shared memory supercomputers [4], and multi-core SMP machines [21]. In the context of points-to analyses, the only parallel implementation we know of [25] has been discussed in depth in previous sections.

Graphics Processing Units have only recently been used for the parallelization of irregular programs. Harish [14] describes CUDA implementations of important graph algorithms such as BFS, Single Source Shortest Paths, Minimum Spanning Tree, etc. In all these algorithms, the structure of the graph being manipulated remains unchanged, which significantly simplifies the GPU implementation. Hong [18] proposes a warp-centric approach for the parallelization of BFS, which is similar to the solution adopted in this paper for distributing work among threads.

Burtscher [9] describes a GPU implementation of an n-body simulation (Barnes Hut algorithm) that is based on unbalanced octrees, which is twice as fast as a multicore implementation running on 128 CPU cores. In this algorithm, the octree used to record the spatial decomposition of the bodies is populated in the initialization phase, so synchronization is required to correctly grow the tree. However, over 80% of the execution time is spent in the force calculation phase, which does not modify the octree's structure.

The closest work to this paper is the GPU implementation of a 0-CFA analysis by Prabhu *et al* [30]. As in Andersen's algorithm, the graph containing the solution for the dataflow analysis only grows over time until a fixpoint is reached. Our work improves on their solution in several ways:

- Our sparse bit vector representation allows arbitrary, dynamic addition of edges to the graph. In contrast, the size of the adjacency list used by Prabhu is statically determined: if too many outgoing edges are added to a node, the execution is aborted. Also, a substantial amount of memory is wasted for variables with few outgoing edges.
- We modified the graph rewrite rules to avoid synchronization, which is required in the implementation by Prabhu.
- The transfer of data between the GPU and the CPU does not impose any performance penalty in our implementation.

9. Conclusions and future work

Our work presents solutions for many of the challenges involved in implementing highly efficient codes on the GPU. In particular, we have shown that porting code from the CPU to the GPU demands fundamental changes in the data structures and algorithmic components being used: classical CPU solutions would result in very poor performance on the GPU. Other important aspects, such as the data transfer between devices, are absent from CPU implementations and need to be carefully included in the design of any GPU code. We expect other researchers in the area to benefit from the techniques presented in this paper, thus reducing the effort required to implement complex algorithms on GPUs.

This paper also confirms that it is possible to efficiently implement highly irregular codes such as Andersen's analysis on the GPU. Although the graphics card utilized in our experiments is cheaper than the multi-CPU machine we are comparing against, we achieve better performance on the GPU. It is interesting to note that although the programming effort involved in the GPU implementation was significantly larger (35% more person-hours) than for the SMP implementation [25], the CUDA version is quite compact in terms of source code size, requiring only 3,000 lines of code (compared to 9,000 in the CPU version). This is primarily due to the smaller set of data structures used by the GPU implementation.

In the future, we intend to implement several other irregular algorithms for which there exist highly competitive, parallel CPU implementations [29]. We expect the programming effort to become less as the GPU architecture and programming model evolve toward supporting more general-purpose features.

References

[1] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.

[2] *CUDA C Programming Guide 4.0*. NVIDIA, 2011.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[4] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[5] J. Barnat, P. Bauch, L. Brim, and M. Češka. Computing Strongly Connected Components in Parallel on CUDA. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, pages 541–552. IEEE Computer Society, 2011.

[6] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 103–114, New York, NY, USA, 2003. ACM.

[7] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations*. Springer-Verlag, 2005.

[8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[9] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Morgan Kaufmann, 2011.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68:1370–1380, October 2008.

[11] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. Symp. on Computational Geometry (SCG)*, 1993.

[12] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 85–96, New York, NY, USA, 1998. ACM.

[13] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[14] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC'07: Proceedings of the 14th international conference on High performance computing*, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. *SIGPLAN Not.*, 36(5):254–263, 2001.

[16] Fritz Henglein. Type inference and semi-unification. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, LFP '88, pages 184–197, New York, NY, USA, 1988. ACM.

[17] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM.

[18] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[19] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *20th International Conference on Parallel Architectures and Compilation Techniques*, PACT'11, 2011.

[20] Song Huang, Shucai Xiao, and Wu chun Feng. On the energy efficiency of graphics processing units for scientific computing. In *IPDPS*, pages 1–8, 2009.

[21] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI)*, 42(6):211–222, 2007.

[22] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[23] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[24] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[25] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, October 2010.

[26] Duane G. Merrill, Michael Garland, and Andrew S. Grimshaw. Scalable gpu graph traversal. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, 2012.

[27] Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS '11: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[28] NVIDIA. Thrust library version 1.4.0. http://code.google.com/p/thrust/.

[29] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[30] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. Eigencfa: accelerating flow analysis with gpus. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.

[31] Thomas W. Reps. Program analysis via graph reachability. Technical Report Technical Report Number 1386, University of Wisconsin, 1998.

[32] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 47–56, New York, NY, USA, 2000. ACM.

[33] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[34] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

[35] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 131–144, New York, NY, USA, 2004. ACM.

[36] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.