Accelerating Inclusion-based Points-to Analysis on Heterogeneous CPU-GPU Systems

Yu Su   Ding Ye   Jingling Xue

University of New South Wales, Australia
{ysu,dye,jingling}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201314
June 2013

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

This paper describes the first implementation of Andersen's inclusion-based pointer analysis for C programs on a heterogeneous CPU-GPU system, where both its CPU and GPU cores are used. As an important graph algorithm, Andersen's analysis is difficult to parallelise because it makes extensive modifications to the structure of the underlying graph, in a way that is highly input-dependent and statically hard to analyse. Existing parallel solutions run on either the CPU or GPU but not both, rendering the underlying computational resources underutilised and the ratios of CPU-only over GPU-only speedups for certain programs (i.e., graphs) unpredictable. We observe that a naive parallel solution of Andersen's analysis on a CPU-GPU system suffers from poor performance due to workload imbalance. We introduce a solution that is centered around a new dynamic workload distribution scheme. The novelty lies in prioritising the distribution of different types of workloads, i.e., graph-rewriting rules in Andersen's analysis to CPU or GPU according to the degrees of the processing unit's suitability for processing them. This scheme is effective when combined with synchronisation-free execution of tasks (i.e., graph-rewriting rules) and difference propagation of points-to information between the CPU and GPU. For a set of seven C benchmarks evaluated, our CPU-GPU solution outperforms (on average) (1) the CPU-only solution by 50.6%, (2) the GPU-only solution by 78.5%, and (3) an oracle solution that behaves as the faster of (1) and (2) on every benchmark by 34.6%.

1 Introduction

Pointer analysis, a technique that statically determines the possible runtime values of a pointer, is crucial for debugging, security analysis, verification and compiler optimisation. The challenge of pointer analysis is to make judicious tradeoffs between precision and efficiency across several dimensions [18, 28, 30, 37, 35], including flow-sensitivity (by considering control flow) and context-sensitivity (by distinguishing calling contexts). Due to its scalability, Andersen's analysis [2], an inclusion-based flow- and context-insensitive pointer analysis, has been adopted by production compilers such as Open64, LLVM and GCC. Over the years, this analysis has undergone many improvements [12, 14, 23, 25, 29, 31, 32, 34].

Andersen's analysis represents a graph algorithm that is difficult to parallelise on both CPUs and GPUs. The analysis makes extensive modifications to the structure of the underlying graph in such a way that the modifications are highly input-dependent and statically hard to predict. As a result, existing techniques for parallelising graph algorithms such as breadth-first search (BFS) and single-source shortest paths on CPUs [1, 16] and on GPUs [3, 13, 16] cannot be directly used.

Recently, there have been limited attempts to parallelise Andersen's analysis [20, 21, 27]. These parallel solutions, which run on either a CPU [21, 27] or a GPU [20], suffer from two drawbacks. First, heterogeneous CPU-GPU systems have grown in popularity within the commercial platform and application developer communities. If a CPU-only (GPU-only) solution runs on a CPU-GPU system, the GPU (CPU) will be mostly idle, causing its computational resources to be underutilised. Second, CPUs and GPUs are fairly close in performance (with the latter being faster by 2.5X on average) [17]. CPUs behave better for some applications while GPUs prevail for others, indicating that the relative suitability of CPUs or GPUs for applications varies depending on their workload characteristics. In the case of Andersen's analysis, different types of graph rewriting rules are repeatedly applied. CPUs are more suitable for some rules while GPUs can handle others better. By using exclusively either a CPU or a GPU [20], the ratios of CPU-only over GPU-only speedups for Andersen's analysis fluctuate wildly across different programs, i.e., input graphs being analysed. The heterogeneous computing power of a CPU-GPU is therefore not fully exploited.

As mentioned earlier, Andersen's analysis is hard to parallelise even just for a CPU [21, 27] or a GPU [20]. We must address two new challenges when parallelising it on a heterogeneous CPU-GPU system. First, the workload distribution between the CPU and GPU must be balanced with negligible runtime overhead. This is nontrivial since different programs give rise to different graphs to be analysed and the structure of a graph changes unpredictably during the analysis. Second, the CPU-GPU communication must be minimised in terms of the amount of data exchanged and the degree of overlap with computation on CPU and GPU. This is also nontrivial because the graphs being analysed are not only dynamically changing but also sparse, making it hard to extract the "right" amount of information to communicate between the CPU and GPU.

In this paper, we describe the first implementation of Andersen's analysis on a heterogeneous CPU-GPU system. We take advantage of a previous formulation of this analysis in terms of graph-rewriting rules [20] to ensure that all rule

applications on CPU and GPU are synchronisation-free. To minimise workload imbalance, we prioritise the distribution of different types of graph-rewriting rules (i.e., workloads) to CPU or GPU according to the degrees of the processing unit's suitability for processing them. To minimise CPU-GPU communication, we adopt difference propagation to transfer new points-to information between CPU and GPU and overlap this process with some computations on CPU and GPU.

While this paper focuses on Andersen's analysis, the proposed techniques for minimising workload imbalance and communication on a CPU-GPU system are expected to be useful for parallelising other graph algorithms that make modifications to the structure of the underlying graph.

In summary, the contributions of this paper are:

- the first parallel solution of Andersen's analysis for C programs on a CPU-GPU system;
- a dynamic workload distribution scheme that dispatches a particular type of workload to the processor, CPU or GPU, that is better suited for the workload;
- a difference propagation scheme for transferring new points-to information discovered between CPU and GPU to reduce communication cost; and
- an evaluation using seven C benchmarks, showing that our CPU-GPU solution outperforms (on average) (1) the CPU-only solution by 50.6%, (2) the GPU-only solution by 78.5%, and (3) an oracle that behaves as the faster of (1) and (2) on every benchmark by 34.6%, where (1) and (2) are improved state-of-the-art implementations introduced in [20, 21].

The rest of this paper is organised as follows. Section 2 introduces Andersen's analysis and highlights some architectural differences between CPU and GPU. Section 3 describes our CPU-GPU solution of Andersen's analysis. Section 4 discusses several optimisations for improving its performance. Section 5 evaluates and analyses our solution. Section 6 discusses the related work. Section 7 concludes.

2 Background

We introduce Andersen's pointer analysis formulated earlier in terms of graph-rewriting rules [20]. We then highlight some architectural differences between CPU and GPU.

2.1 Andersen's Inclusion-Based Pointer Analysis

Andersen's analysis for a program is formulated as a set-constraint problem over a directed graph, $G = (V, E)$, called a *constraint graph*. After being initialised, G will be iteratively modified until a fixed point is reached.

2.1.1 Initialisation

Given a C program, its constraint graph is created with its node set V being the variables in the program and its edge set E being populated with the edges

representing five different types of statements in the program, as shown in Table 2.1. There is one edge per statement in the program. For example, $x \xrightarrow{\mathcal{P}} y$ represents a points-to, i.e., $\mathcal{P}$ edge directing out of node x into node y. Note that our notations for some edge labels are different from those used in [20].

Name	Statement	Edge
Points-to	$x = \&y$	$x \xrightarrow{\mathcal{P}} y$
Copy	$x = y$	$x \xrightarrow{\mathcal{C}} y$
Load	$x = *y$	$x \xrightarrow{\mathcal{L}} y$
Store	$*x = y$	$x \xrightarrow{\mathcal{S}} y$
Offset	$x = y + o$	$x \xrightarrow{\mathcal{F},\, o} y$

Table 2.1: Mapping statements to constraint edges (initialisation).

2.1.2 Constraint Resolution

Once the constraint graph for a program is initialised, new points-to and copy edges are introduced by applying the four types of graph-rewriting rules given in Table 2.2 until the constraint graph reaches a fixed point. If there is a copy edge from x to y and a points-to edge from y to z, then applying $\mathsf{COPY}(x)$ to x causes a new points-to edge from x to z to be added (if it does not exist yet). Rules $\mathsf{LOAD}(x)$ and $\mathsf{STORE}(x)$ are applied to discover new copy edges directing out of x. Finally, $\mathsf{OFFSET}(x)$ is applied to introduce new points-to edges directing out of x field-sensitively.

Note that these rules are formulated this way so that they can be easily applied in parallel to avoid synchronisation on both CPU and GPU, as will be explained in Section 3.

Rule	Semantics
$\mathsf{COPY}(x)$	$x \xrightarrow{\mathcal{C}} y \;\wedge\; y \xrightarrow{\mathcal{P}} z \;\Rightarrow\; x \xrightarrow{\mathcal{P}} z$
$\mathsf{LOAD}(x)$	$x \xrightarrow{\mathcal{L}} y \;\wedge\; y \xrightarrow{\mathcal{P}} z \;\Rightarrow\; x \xrightarrow{\mathcal{C}} z$
$\mathsf{STORE}(x)$	$x \xrightarrow{\mathcal{P}^{-1}} y \;\wedge\; y \xrightarrow{\mathcal{S}} z \;\Rightarrow\; x \xrightarrow{\mathcal{C}} z$
$\mathsf{OFFSET}(x)$	$x \xrightarrow{\mathcal{F},\, o} y \;\wedge\; y \xrightarrow{\mathcal{P}} z \;\Rightarrow\; x \xrightarrow{\mathcal{P}} z + o$

Table 2.2: Graph-rewriting rules (constraint resolution).

2.1.3 Sequential Algorithm

Andersen's analysis is formulated in terms of graph-rewriting rules as shown in Algorithm 1. Given a program, CreateGraph is called to initialise its constraint graph $G = (V, E)$ as per Section 2.1.1 and Apply is called to perform constraint resolution on all the nodes until a fixed point is reached as per Section 2.1.2.

In Apply, $E_t(x)$ denotes the set of edges of type t associated with the node x being processed. For each variable x in W (initialised with V), Apply loops over its edge set of type $t1$, i.e., $E_{t1}(x)$. For each y in this set, Apply loops

over its edge set of type $t2$, i.e., $E_{t2}(y)$. For each z in this set, two cases are distinguished. If $t1 = \mathcal{F}$, $z + o$ is added to $E_{t3}(x)$, i.e., x's edge set of type $t3$. Otherwise, z is added.

To represent the four types of rules given in Table 2.2 uniformly, we have modified STORE(x) conceptually. Instead of a points-to edge $y \xrightarrow{\mathcal{P}} x$, its inverse, known as a *pointed-by edge*, denoted by $x \xrightarrow{\mathcal{P}^{-1}} y$, is used. As will be explained later, only points-to edges are stored to save space.

Algorithm 1 Andersen's Analysis

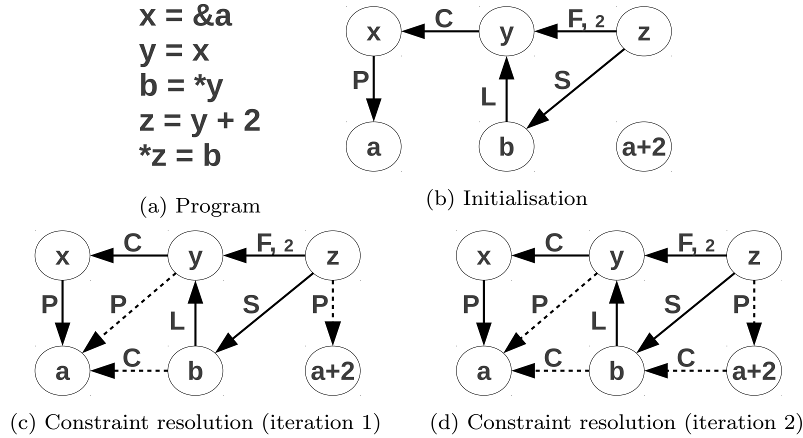
```
   Procedure ANDERSEN()
   begin
1  |  G = (V, E) ← CREATEGRAPH();
2  |  repeat
3  |  |  APPLY(C, P, P, V);
4  |  |  APPLY(L, P, C, V);
5  |  |  APPLY(P^-1, S, C, V);
6  |  |  APPLY(F, P, P, V);
   |  until fixed-point;

   Procedure APPLY(t1, t2, t3, W)
   begin
7  |  foreach x ∈ W do
8  |  |  foreach y ∈ E_t1(x) do
9  |  |  |  foreach z ∈ E_t2(y) do
10 |  |  |  |  if t1 = F then
11 |  |  |  |  |  Let the offset edge (x, y) be x --F,o--> y
12 |  |  |  |  |  E_t3(x) ← E_t3(x) ∪ {z + o}
13 |  |  |  |  else E_t3(x) ← E_t3(x) ∪ {z}
```

2.1.4 Example

For the program given in Figure 2.1(a), Figure 2.1(b) depicts the constraint graph initialised by CreateGraph. This graph is then modified iteratively by Apply. The modified graphs in the first two iterations are shown in Figures 2.1(c) and 2.1(d). In the first iteration, COPY is applied to y, resulting in $y \xrightarrow{\mathcal{P}} a$ to be added. Then applying LOAD to b causes $b \xrightarrow{\mathcal{C}} a$ to be added. At this stage, STORE is not applicable. Finally, OFFSET is applied to z, with $z \xrightarrow{\mathcal{P}} a + 2$ being discovered. In the second iteration, STORE can be applied to $a + 2$, giving rise to $a + 2 \xrightarrow{\mathcal{C}} b$. A fixed-point is then reached. The points-to information for each variable can be directly read-off from the final constraint graph obtained.

2.2 Architectural Differences between CPU and GPU

We recall the key differences [17] with respect to the CPU-GPU system used in this paper. The host is equipped with two eight-core Inter Xeon CPUs and the



(a) Program (b) Initialisation

(c) Constraint resolution (iteration 1) (d) Constraint resolution (iteration 2)

Figure 2.1: An example illustrating Andersen's analysis. For each points-to edge, its pointed-by edge is not shown. The new copy and points-to edges added during constraint resolution are depicted in dashed arrows. The fixed point is reached in iteration 2.

accelerator is an NVIDIA GPU, TESLA K20c, based on the Kepler architecture. The GPU consists of 13 streaming multiprocessors (SMs), each containing 192 cores, giving rise to thousands of GPU cores (two orders of magnitude more than the host). In addition, the GPU has a peak memory bandwidth of 208GB/s, about 10 times of that for the host. This suggests that the GPU is well suited for regular, balanced workloads with abundant data parallelism when its massive number of cores and high memory bandwidth are fully utilised. However, the GPU, which clocks at 0.71 GHz, is less powerful than a CPU, which clocks at 2.0 GHz. In addition, the GPU has memory access latency of $400 - 800$ cycles, making it less competitive than CPU if its cores and memory bandwidth are underutilised, which is hard to avoid for irregular, imbalanced workloads.

In the case of imbalanced workloads, the GPU's computational resources may be underutilised in two scenarios. First, each SM schedules threads in groups of 32 parallel threads, called *warps*. Threads in the same warp are able to execute in parallel the same instruction, but in serial if different instructions are encountered. In this situation, known as *warp divergence*, the GPU cores are not fully utilised. Second, warps are executed concurrently on each SM. When one warp is paused or stalled, other warps are executed in order to keep the cores busy and hide memory latency. If the workloads mapped to different warps are imbalanced, many warps (up to 350 across all SMs) may end up waiting for a few warps to finish. Due to the lack of enough active warps to hide memory access latency, we have a so-called *inter-warp imbalance* problem. For CPU, however, usually only dozens of threads can be launched at the same time. The inter-thread imbalance is not as severe, especially when the CPU's memory access latency, which is lower than the GPU, can be hidden by large caches.

When parallelising Andersen's analysis on a CPU-GPU system, we will exploit the respective architectural advantages of CPU and GPU to accelerate its performance.

3 A CPU-GPU Solution of Andersen's Analysis

A naive solution would be to dynamically assign portions of the constraint graph of a program to the CPU and GPU and let them apply all graph-rewriting rules applicable to their own portions. As evaluated later, this simplistic solution suffers from poor performance, due to workload imbalance and communication overhead incurred, because the modifications to the underlying graph can be unpredictable.

The basic idea behind our solution is sketched in Algorithm 2. Initially, CreateGraph used in Algorithm 1 is called to initialise the constraint graph identically on both the CPU and GPU. Then Andersen's analysis is performed iteratively on both the CPU and GPU until a fixed-point is reached. The final points-to information will be available on both the CPU and GPU. The key novelty lies in proritising, i.e., sorting different types of graph-rewriting rules in a shared worklist $\mathcal{W}$, so that each side of a CPU-GPU system can always obtain the work from $\mathcal{W}$ that it is the most suitable to process.

Algorithm 2 A CPU-GPU solution of Andersen's analysis.

```
   begin
1      G = (V, E) ← CreateGraph();
2      repeat
3          Reset W;
           CPU side                          GPU side
4          FetchAndApply(W);                 FetchAndApply(W);
5          Transfer E_Δ_CPU to GPU;          Transfer E_Δ_GPU to CPU;
6          E_Δ ← E_Δ_CPU ∪ E_Δ_GPU;          E_Δ ← E_Δ_GPU ∪ E_Δ_CPU;
7          E ← E ∪ E_Δ;                      E ← E ∪ E_Δ;
       until fixed-point;
```

At each iteration, both the CPU and GPU calls FetchAndApply($\mathcal{W}$) to fetch the work from $\mathcal{W}$ and then apply appropriate rules to the work obtained until $\mathcal{W}$ is empty. Then both sides exchange only the new points-to and copy edges, $E_{\Delta_{\text{GPU}}}$ and $E_{\Delta_{\text{CPU}}}$, discovered, based on difference propagation. After the constraint graph at each side has been updated, the next iteration begins, if needed.

Section 3.1 describes our graph data representation. Section 3.2 focuses on CPU-GPU communication. Section 3.3 explains how to perform parallel rule applications on CPU and GPU, assisted by our dynamic workload distribution scheme.

3.1 Graph Data Representation

Constraint graphs are sparse and their structures change dynamically during Andersen's analysis. Therefore, selecting an appropriate data structure to store such graphs can have a profound impact on the amount of computations performed on both the CPU and GPU and the amount of data exchanged.

To represent edge sets compactly and support operations on them efficiently, sparse bit vectors and BDDs (Binary Decision Diagrams) are popular. BDDs

are complex and ill-suited for GPU [20]. Sparse bit vectors are 2X faster than BDDs on CPU [12]. To minimise CPU-GPU communication, we have opted to use sparse bit vectors uniformly on both CPU and GPU.

Consider a constraint graph $G = (V, E)$. The variables (or nodes) in V are mapped to consecutive integers, starting from 0. For each variable $x \in V$, its outgoing edges are distinguished according to the five basic types in Table 2.1. There are four separate sparse bit vectors storing its $\mathcal{P}$, $\mathcal{C}$, $\mathcal{L}$ and $\mathcal{S}$ edges, respectively. We do not store the $\mathcal{P}^{-1}$ edges explicitly to save space; we discuss how to handle the STORE rule in Section 3.3. As for the $\mathcal{F}$ edges, a simple normalisation ensures that each node has at most one offset edge, which is then stored conventionally. For example, if $x = y + 2$ and $x = z + 4$, then $t1 = y + 2; x = t1$ and $t2 = z + 4; x = t2$.

As an example, suppose node x has two outgoing points-to edges, $E_{\mathcal{P}}(x) = \{y, z\}$, where y and z are identified by integers 958 and 1920, respectively. The sparse representation for these two points-to edges are illustrated in Figure 3.1.

A sparse bit vector is a linked list that represents a set of integers. Each element consists of three fields: *bits* (several words) represents whether the corresponding integer belongs to the set, *base* (1 word) indicates the range of integers that is represented in this element, and *next* (1 word) points to the next element. For warp efficiency on GPU, 32 words for an element is suggested [20], where the *bits* spans 30 words (960 bits). As a result, the 32 threads in a warp perform set operations, e.g., union, on the 32 words concurrently.

Let us see how $E_{\mathcal{P}}(x) = \{y, z\}$ is represented in Figure 3.1. The first element's *base* is 0 and the 958-th bit in *bits* is set to 1. So it contains the integer 958, i.e., y. The second element's *base* is 2 and the 0-th bit in *bits* is 1. As a result, it contains z, identified by integer $960 \times 2 + 0 = 1920$.

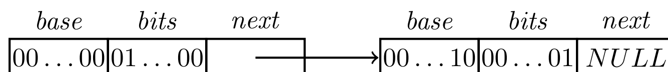


Figure 3.1: Sparse bit vector representing $\{958, 1920\}$.

3.2 Managing Communication between CPU and GPU

In heterogeneous CPU-GPU computing, the communication between the two sides can be a major cost. As constraint graphs are sparse and changing during the analysis, it is challenging but important to reduce the communication cost.

In the sequential setting, difference propagation [9, 24, 29] is used to reduce the work of propagating points-to edges in a constraint graph. As shown in Table 2.2, Andersen's analysis may modify a constraint graph by adding new points-to and copy edges, i.e., new $\mathcal{P}$ and $\mathcal{C}$ edges. We make use of difference propagation (for the first time) to ensure that at the end of each iteration, the CPU and GPU only need to exchange the new $\mathcal{P}$ and $\mathcal{C}$ edges introduced in that iteration.

Table 3.1 gives the graph-rewriting rules modified from Table 2.2 to support difference propagation. To facilitate concurrent applications of graph-rewriting rules, double buffering is used. In each rule, $\Delta\mathcal{P}$ ($\Delta\mathcal{C}$) in the promise signifies a new points-to (copy) edge produced in the previous iteration and $\delta\mathcal{P}$ ($\delta\mathcal{C}$) in the conclusion signifies a new points-to (copy) edge produced in the current

Rule	Semantics
$\mathsf{COPY}(x)$	$x \xrightarrow{\mathcal{C}} y \ \wedge\ y \xrightarrow{\Delta\mathcal{P}} z \ \Rightarrow\ x \xrightarrow{\delta\mathcal{P}} z$
$\mathsf{LOAD}(x)$	$x \xrightarrow{\mathcal{L}} y \ \wedge\ y \xrightarrow{\Delta\mathcal{P}} z \ \Rightarrow\ x \xrightarrow{\delta\mathcal{C}} z$
$\mathsf{STORE}(x)$	$x \xrightarrow{\Delta\mathcal{P}^{-1}} y \ \wedge\ y \xrightarrow{\mathcal{S}} z \ \Rightarrow\ x \xrightarrow{\delta\mathcal{C}} z$
$\mathsf{OFFSET}(x)$	$x \xrightarrow{\mathcal{F}, o} y \ \wedge\ y \xrightarrow{\Delta\mathcal{P}} z \ \Rightarrow\ x \xrightarrow{\delta\mathcal{P}} z + o$
$\Delta\mathsf{COPY}(x)$	$x \xrightarrow{\Delta\mathcal{C}} y \ \wedge\ y \xrightarrow{\mathcal{P}} z \ \Rightarrow\ x \xrightarrow{\delta\mathcal{P}} z$

Table 3.1: Graph-rewriting rules via difference propagation.

iteration. As before, $\mathcal{P}$, $\mathcal{C}$, $\mathcal{L}$, $\mathcal{S}$ or $\mathcal{F}$, identifies an edge of that type available at the end of the previous iteration. The ΔCOPY rule is new. Once again, the $\Delta\mathcal{P}^{-1}$ edges, the reversed $\Delta\mathcal{P}$ edges, are not actually stored.

In Algorithm 2, $E_{\Delta_{\text{CPU}}}$ ($E_{\Delta_{\text{GPU}}}$) denotes the set of $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges produced in the current iteration on CPU (GPU).

Below we introduce a *reference* CPU-GPU solution of Andersen's analysis, which has been useful in guiding the development and evaluation of our CPU-GPU solution. Consider the CPU-only and GPU-only implementations of Andersen's analysis detailed in Section 5.1. At this stage, it suffices to know that both proceed essentially by applying the double-buffering-based rules given in Table 3.1 based on exactly the same algorithm. Let w_i be the workload at the i-th iteration. Let t^i_{CPU} and t^i_{GPU} be the analysis times elapsed at the i-th iteration on CPU and GPU, respectively. By assuming constant work rates for CPU and GPU and zero communication and synchronisation overhead, a reference CPU-GPU solution spends the following analysis time at the i-th iteration:

$$t^i_{\text{REF}} = \frac{w_i}{\frac{w_i}{t^i_{\text{CPU}}} + \frac{w_i}{t^i_{\text{GPU}}}} = \frac{t^i_{\text{CPU}} \times t^i_{\text{GPU}}}{t^i_{\text{CPU}} + t^i_{\text{GPU}}} \tag{3.1}$$

The benefit at the i-th iteration from CPU-GPU computing is:

$$benefit(i) = \min(t^i_{\text{CPU}}, t^i_{\text{GPU}}) - t^i_{\text{REF}} \tag{3.2}$$

Let us analyse the potential performance gains achieved by performing CPU-GPU communication via difference propagation. Consider a CPU-GPU implementation that always produces the same amount of new points-to and copy edges at both sides at each iteration. Let $d^i_{\Delta\mathcal{P}\Delta\mathcal{C}}$ ($d^i_{\mathcal{P}\mathcal{C}}$) be the set of new (all) points-to and copy edges produced at the i-th iteration, which is half of the same points-to information produced by the CPU- or GPU-only implementation at the i-th iteration. If Host-to-Device and Device-to-Host transfers are concurrent, then the costs, $cost^i_{\Delta\mathcal{P}\Delta\mathcal{C}}$ and $cost^i_{\mathcal{P}\mathcal{C}}$, for exchanging $d^i_{\Delta\mathcal{P}\Delta\mathcal{C}}$ and $d^i_{\mathcal{P}\mathcal{C}}$ between the CPU and GPU at the i-th iteration are:

$$\begin{aligned} cost_{\Delta\mathcal{P}\Delta\mathcal{C}}(i) &= \frac{d^i_{\Delta\mathcal{P}\Delta\mathcal{C}}}{B} + S \\ cost_{\mathcal{P}\mathcal{C}}(i) &= \frac{d^i_{\mathcal{P}\mathcal{C}}}{B} + S \end{aligned} \tag{3.3}$$

where B and S are the host-to-device bandwidth and the startup cost, respectively, on the CPU-GPU system considered.

Figure 3.2 plots the functions $benefit$, $cost_{\Delta\mathcal{P}\Delta\mathcal{C}}$ and $cost_{\mathcal{P}\mathcal{C}}$ for `svn`, a program in our benchmark suite. In (3.3), B = 6GB/s and S = 10μs for the

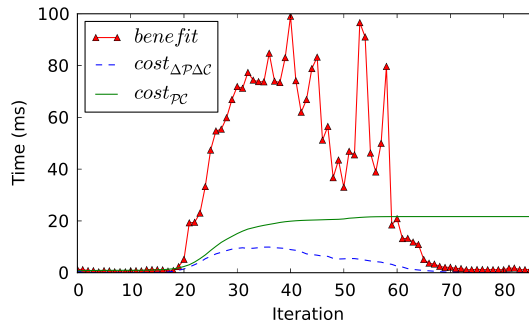


Figure 3.2: A cost-benefit analysis for the `svn` benchmark.

CPU-GPU system used in this paper. The startup cost, taken from [19], is negligible relative to the data transfer times that are between two to three orders of magnitude longer. During the iterations from 20 to 65, the cost of transferring $\mathcal{P}$ and $\mathcal{C}$ edges can be about 1/3 of the benefit while the cost of transferring $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges is moderate. From iteration 65 onwards, the cost of transferring $\mathcal{P}$ and $\mathcal{C}$ edges is overwhelming. Fortunately, the cost of transferring $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges is still no larger than the benefit (even it is small). Therefore, transferring $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ between the CPU and GPU is important to mitigate the negative impact of communication cost on performance.

3.3 Partitioning Computation for CPU and GPU

The objective here is to maximise parallel rule applications on both CPU and GPU at negligible synchronisation overhead. We first describe how to orchestrate the concurrent execution of graph-rewriting rules on CPU and GPU. We then describe how to distribute rule applications dynamically to CPU and GPU to ensure that workload balance is maintained.

3.3.1 Parallel Rule Applications

In our sparse representation of a constraint graph, different types of outgoing edges of a node are stored in different sparse bit vectors. Due to double buffering used in the rules given in Table 3.1, different applications of the same rule can be executed concurrently without synchronisation. In addition, applications of different rules can also be concurrent as long as these rules do not write into the same sparse bit vector storing $\delta\mathcal{P}$ or $\delta\mathcal{C}$.

At the GPU side, every rule application at a node x is executed by a warp as in [20] in a warp-centric manner [15]. Lines 12 − 13 in Algorithm 1 are implemented as:

$T \leftarrow$ Add o (if any) to each $t2$-neighbour of y
Union T with $t3$-neighbours of x

The first is a highly data-parallel operation due to the use of sparse bit vectors. When computing the union of two 32-word elements with the same base, 32 threads in a warp will work concurrently, one word per thread, with intra-warp divergence occurring at the first word (for testing the base) and last word.

At the host side, every rule application is executed sequentially inside a CPU thread. Instead of performing union operations on words, long words are used for efficiency.

The STORE rule requires the $\Delta\mathcal{P}^{-1}$ edges, i.e., new pointed-by edges to be stored, which can be space-consuming. We avoid this by adopting the same two-phase strategy used in [20]. In the first phase, a worklist is created that contains all pairs of (x, y) such that y has outgoing store edges and $y \xrightarrow{\mathcal{P}} x$. In the second phase, all pairs with the same first component are assigned to the same GPU warp or CPU thread. As a result, all applications of the STORE rule can be executed in parallel without synchronisation.

3.3.2 Dynamic Workload Distribution

To accelerate Andersen's analysis on a CPU-GPU system, it is critically important to minimise workload imbalance between the CPU and GPU. As shown in Algorithm 2, we use a work sharing scheme so that both the CPU and GPU fetch the work to do from a mutex-protected shared worklist, $\mathcal{W}$, at each iteration.

A simple-minded scheme, referred to as Naive, for implementing FetchAndApply in Algorithm 2 is given in Algorithm 3. The worklist $\mathcal{W}$ consists of all the N nodes in the constraint graph, as illustrated in Figure 3.3. The CPU and GPU repeatedly fetch a set $\mathcal{M}$ of nodes from the beginning of $\mathcal{W}$ and then apply all the rules to the nodes in $\mathcal{M}$.

Algorithm 3 A naive workload distribution scheme.

Procedure FetchAndApply($\mathcal{W}$)
begin
1 **while** $\mathcal{W} \neq \emptyset$ **do**
2 $\mathcal{M} \leftarrow$ get work from $\mathcal{W}$;
3 Apply($\mathcal{C}$, $\Delta\mathcal{P}$, $\delta\mathcal{P}$, $\mathcal{M}$);
4 Apply($\mathcal{L}$, $\Delta\mathcal{P}$, $\delta\mathcal{C}$, $\mathcal{M}$);
5 Apply($\Delta\mathcal{P}^{-1}$, $\mathcal{S}$, $\delta\mathcal{C}$, $\mathcal{M}$);
6 Apply($\mathcal{F}$, $\Delta\mathcal{P}$, $\delta\mathcal{P}$, $\mathcal{M}$);
7 Apply($\Delta\mathcal{C}$, $\mathcal{P}$, $\delta\mathcal{P}$, $\mathcal{M}$);

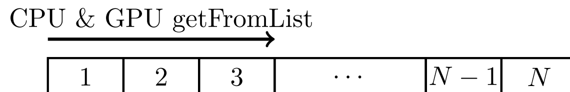


Figure 3.3: The shared worklist $\mathcal{W}$ used in Algorithm 3.

As evaluated later, Naive suffers from poor performance due to workload imbalance, because it does not consider the suitability of CPU and GPU for processing different types of rules. As discussed in Section 2.2, the GPU is more powerful than the CPU for regular, balanced workloads, but performs more poorly on irregular, imbalanced workloads. In Andersen's analysis, severe inter-warp workload imbalance can occur when the warps are processing nodes with varying out-degrees of theirs edges. Figure 3.4 plots the distributions of

the sizes of the $\mathcal{P}$ edges and $\mathcal{C}$ edges for `svn`, a program in our benchmark suite, in the *lg-sqrt* format, after Andersen's analysis is finished. These sizes vary greatly, starting from 0 and approaching 10000. However, the CPU is capable of handling such imbalanced workloads more efficiently.

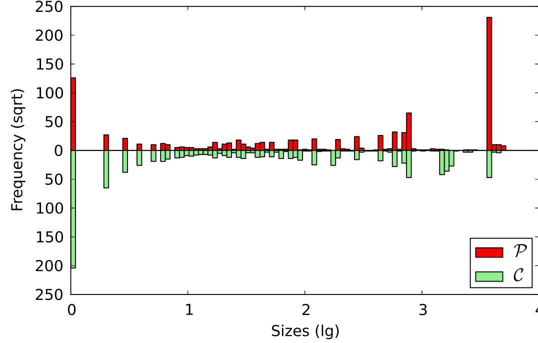


Figure 3.4: Sizes of points-to and copy edges for `svn`.

Different applications of the same rule at the same node may induce different workloads at different iterations. However, it is difficult to decide precisely where to execute a rule (on the CPU or GPU) and when, since these workloads are changing in an unpredictable manner during the analysis.

Our key observation is that different types of rules in Andersen's analysis tend to exhibit different workload characteristics. In general, some rules are more amenable to CPU execution while the others fare better on the GPU. It is thus possible to prioritise different types of rules according to the suitability of CPU or GPU for processing them.

Consider the four rules, COPY, LOAD, OFFSET and ΔCOPY, given in Table 3.1. We will deal with the STORE rule differently later. For each of these four rules applied at a node x, three types of outgoing edges, indicated as $t1$, $t2$ and $t3$ in lines $7-13$ in Algorithm 1, are accessed. In Table 3.1, $t1$ and $t2$ appear in the premise of a rule and $t3$ in its conclusion.

For each rule R applied at node x in lines $7-13$ in Algorithm 1, the amount of work performed is dictated by $|E_{t1}(x)| \times |E_{t2}(y)|$, where $t1$ and $t2$ indicate the types of edges processed in the premise of the rule. We write DI_R to represent the *degree of imbalance* for the workloads performed by applying rule R at all the possible nodes in the constraint graph, measured by how $|E_{t1}(x)| \times |E_{t2}(y)|$ varies across these nodes (using, for example, its standard derivation).

We propose to use the DI_R of rule R to determine the suitability of the CPU or GPU for applying the rule. The higher DI_R is, the more (less) suitable the CPU (GPU) is for applying rule R. According to the workload characteristics of the four rules, COPY, LOAD, OFFSET and ΔCOPY, we have:

$$DI_{\Delta\mathsf{COPY}} > DI_{\mathsf{COPY}} > DI_{\mathsf{LOAD}} > DI_{\mathsf{OFFSET}} \tag{3.4}$$

We observe that the larger $\max_{v \in V} |E_T(v)|$ is for a particular type of edges in a constraint graph, the greater $|E_T(v)|$ varies across different nodes in V. In general, $|E_{\mathcal{F}}(v)|$ is much smaller than $\min(|E_{\Delta\mathcal{P}}(v)|, |E_{\Delta\mathcal{C}}(v)|)$ and $|E_{\mathcal{C}}(v)|$, $|E_{\mathcal{L}}(v)|$ and $|E_{\mathcal{P}}(v)|$ are approximately an order of magnitude larger than $\max(|E_{\Delta\mathcal{P}}(v)|,$

Algorithm 4 A DI-based dynamic workload distribution.

Procedure FetchAndApply($\mathcal{W}$)
begin
1 Apply($\Delta\mathcal{P}^{-1}$, $\mathcal{S}$, $\delta\mathcal{C}$, $\mathcal{W}_{\mathsf{STORE}}$); // STORE
2 **while** $\mathcal{W} \neq \emptyset$ **do**
3 ($\mathcal{M}, r$) $\leftarrow$ get work from $\mathcal{W}$;
4 **if** r = COPY **then** Apply($\mathcal{C}$, $\Delta\mathcal{P}$, $\delta\mathcal{P}$, $\mathcal{M}$);
5 **if** r = LOAD **then** Apply($\mathcal{L}$, $\Delta\mathcal{P}$, $\delta\mathcal{C}$, $\mathcal{M}$);
6 **if** r = OFFSET **then** Apply($\mathcal{F}$, $\Delta\mathcal{P}$, $\delta\mathcal{P}$, $\mathcal{M}$);
7 **if** r = ΔCOPY **then** Apply($\Delta\mathcal{C}$, $\mathcal{P}$, $\delta\mathcal{P}$, $\mathcal{M}$);

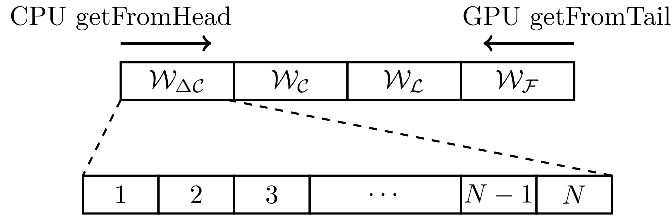


Figure 3.5: The shared worklist $\mathcal{W}$ used in Algorithm 4.

$|E_{\Delta\mathcal{C}}(v)|$). Therefore, DI_{OFFSET} is the smallest. While being close to $|E_{\mathcal{L}}(v)|$ when the constraint graph is initialised, $|E_{\mathcal{P}}(v)|$ and $|E_{\mathcal{C}}(v)|$ can increase dramatically during the analysis. So DI_{LOAD} is the second smallest. In general, $|E_{\mathcal{P}}(v)|$ is much larger than $|E_{\mathcal{C}}(v)|$, as indicated in Figure 3.4. Therefore, we have $DI_{\Delta\mathsf{COPY}} > DI_{\mathsf{COPY}}$.

As the $\Delta\mathcal{P}^{-1}$ edges are not stored, we deal with the STORE rule differently. As discussed in Section 3.3.1, a separate worklist, $\mathcal{W}_{\mathsf{STORE}}$, is maintained from which the set V_{STORE} of nodes that require the STORE rule to be applied are obtained. Let $DI_{\mathsf{STORE}} = \max\{|E_{\Delta\mathcal{P}^{-1}}(x)| \mid x \in V_{\mathsf{STORE}}\}$ be used to approximate the degree of imbalance for this rule. Of course, the $\Delta\mathcal{P}^{-1}$ edges, which are not stored, are deduced from the $\Delta\mathcal{P}$ edges. At each iteration, we decide dynamically whether the CPU or GPU is more suitable to apply the STORE rule to the nodes in V_{STORE}. Our simple heuristic is to select the CPU if and only if $DI_{\mathsf{STORE}} \geqslant \tau$, which is set as 20 empirically.

In our CPU-GPU solution of Andersen's analysis, FetchAndApply($\mathcal{W}$) in Algorithm 2 is given in Algorithm 4. A so-called DI-based dynamic workload distribution scheme, referred to as IDD, is therefore adopted. At an iteration, either the CPU or GPU applies the STORE rule to all the nodes in V_{STORE} depending on whether $DI_{\mathsf{STORE}} \geqslant \tau$ holds or not. As for the other four rules, COPY, LOAD, OFFSET and ΔCOPY, the shared worklist $\mathcal{W}$, which is described below, maintains all rule applications to be executed at any iteration. The CPU and GPU repeatedly fetch a set $\mathcal{M}$ of nodes from the worklist $\mathcal{W}$ and apply appropriate rules to the nodes in $\mathcal{M}$.

Based on (3.4), our shared worklist, $\mathcal{W}$, illustrated in Figure 3.5, consists of four sub-worklists for rules ΔCOPY, COPY, LOAD and OFFSET, sorted in decreasing order of their degrees of imbalance. Each sub-worklist consists of all the nodes in

the constraint graph as in Naive, shown in Figure 3.3. The CPU and GPU will fetch the work from the two opposite sides. This ensures that each side of the system will always apply rules in decreasing order of its suitability for processing these rules. As a result, IDD achieves better load balance than Naive.

While concurrent applications of the same rule in Table 3.1 can be synchronisation free, concurrent applications of different rules may require synchronisation when they may have write-write races (e.g., between COPY and ΔCOPY). To avoid such synchronisation altogether on CPU or GPU, we have introduced two barriers, one between $\mathcal{W}_{\Delta\mathcal{C}}$ and $\mathcal{W}_{\mathcal{C}}$ and one between $\mathcal{W}_{\mathcal{L}}$ and $\mathcal{W}_{\mathcal{F}}$. For this reason, a call to FetchAndApply($\mathcal{W}$) in Algorithm 4 always returns a set $\mathcal{M}$ of nodes associated with the same rule type, r.

4 Optimisations

We describe several optimisations to further improve the performance of our CPU-GPU solution of Andersen's analysis.

4.1 Optimisation I: On Hiding Communication Overhead

In Algorithm 2, $E_{\Delta_{\text{CPU}}}$ is the union of $E^{\Delta\mathcal{P}}_{\Delta_{\text{CPU}}}$ and $E^{\Delta\mathcal{C}}_{\Delta_{\text{CPU}}}$, which are the sets of $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges produced on CPU, respectively. $E_{\Delta_{\text{GPU}}}$ is similarly decomposed into $E^{\Delta\mathcal{P}}_{\Delta_{\text{GPU}}}$ and $E^{\Delta\mathcal{C}}_{\Delta_{\text{GPU}}}$ on GPU. Concurrent Host-to-Device and Device-to-Host transfers are used to maximise their overlap. To further reduce communication cost, computation-communication overlap is employed at each iteration. As $|E^{\Delta\mathcal{P}}_{\Delta_{\text{CPU}}}| > |E^{\Delta\mathcal{C}}_{\Delta_{\text{CPU}}}|$ and $|E^{\Delta\mathcal{P}}_{\Delta_{\text{GPU}}}| > |E^{\Delta\mathcal{C}}_{\Delta_{\text{GPU}}}|$ usually, the $\Delta\mathcal{C}$ edges are exchanged before the $\Delta\mathcal{P}$ edges between the CPU and GPU. As soon as one side has received the $\Delta\mathcal{C}$ edges from the other, the operations indicated in lines 6 – 7 are performed on $\Delta\mathcal{C}$, by overlapping with the transfer of the $\Delta\mathcal{P}$ edges.

4.2 Optimisation II: On ΔP-Equivalence and STORE

$\Delta\mathcal{P}$-equivalent variables have the same set of outgoing $\Delta\mathcal{P}$ edges in the current iteration [20]. For example, if $E_{\Delta\mathcal{P}}(x) = E_{\Delta\mathcal{P}}(y)$, then applying COPY($z$), where $E_C(z) = \{x, y\}$, yields the same result for z if either $E_{\Delta\mathcal{P}}(x)$ or $E_{\Delta\mathcal{P}}(y)$ is used.

Work on identifying (1) $\Delta\mathcal{P}$-equivalent variables [20] and (2) the set V_{STORE} of variables where the STORE rule is applied (the first phase of this rule application as discussed in Section 3.3.2) is expensive on CPU, costing over 3X more than on GPU. Therefore, such computations are performed on the GPU, with the results transferred to the CPU.

4.3 Optimisation III: On Adaptive Heterogeneity

As illustrated in Figure 3.2, the performance benefit of a CPU-GPU solution of Andersen's analysis is more than offset by the CPU-GPU communication cost incurred during the first and last few iterations. Therefore, an adaptive scheme is used. When the cost exceeds the benefit, the faster of the two, CPU and GPU, will perform the iteration alone. For the first and last few iterations, the workloads of rule applications are small with negligible imbalance. The GPU

is more suitable and thus preferred. Therefore, our CPU-GPU solution begins in the GPU-alone mode, switches to the heterogeneous mode when $t_{tran} \leqslant t_{comp} * \alpha$, and returns to the GPU-alone mode again when $t_{tran} > t_{comp} * \alpha$. Empirically, $\alpha = 0.2$ is used.

5 Evaluation

We show that our parallel solution of Andersen's analysis on a CPU-GPU system achieves better average speedups than CPU-only and GPU-only solutions for a set of seven C benchmarks considered. Even if the better of the speedups from CPU-only and GPU-only solutions is selected for each benchmark, our CPU-GPU solution remains faster on average.

For the CPU-GPU system used in our experiments, the host (running 64-bit Ubuntu 12.04) is equipped with two eight-core 2.00GHz Intel Xeon E5-2650 CPUs with 62GB of RAM. Each core has a 64KB L1 cache and a 256KB L2 cache. Each CPU has a 20MB L3 unified cache shared by its eight cores. The code for the host is written in C++ using POSIX threads and compiled under "GCC -O3". The GPU used is a 0.71GHz NVIDIA Tesla K20c GPU with 13 SMs, each containing 192 cores. Each SM has a 64KB of on-chip memory configured as 48 KB of shared memory and 16 KB of L1 cache. All SMs share a 1280KB L2 cache. The CUDA code is compiled under "nvcc -arch=sm_30" (v5.0).

Table 5.1 lists the seven C benchmarks used, with the number of variables ranging from 53K to 559K and the number of statements ranging from 55K to 560K. Note that every strongly-connected component (SCC) formed by copy edges is collapsed as its variables have the same points-to edges.

Benchmark	#Variables	#Statements
perl	53,362	55,977
python	92,599	92,827
svn	107,708	122,558
gcc	120,870	127,171
gdb	232,814	198,933
vim	246,944	89,226
gimp	558,867	565,655

Table 5.1: Benchmark suite: sizes of initial constraint graphs.

5.1 Implementations

Our CPU-GPU solution is implemented as follows:

- For GPU kernels, we use 13 thread blocks (for 13 SMs), 864 threads per block for COPY, LOAD, STORE and ΔCOPY, and 1024 threads for OFFSET. This represents the same configuration as in [20], limited by available shared memory (48KB per block) and maximum number of threads per block (1024).
- As for the host, there are 16 compute threads, one per core, and two control threads. The GPU-control thread is responsible for fetching the

work from a shared worklist for the GPU to do, launching kernel execution and exchanging $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ between the CPU and GPU. The CPU-control thread oversees and coordinates the execution of the 16 compute threads.

The 16 CPU compute threads and the GPU-control thread will fetch the work from a mutex-protected shared worklist, illustrated in Figure 3.5. The granularity of each fetch is determined empirically. For a CPU thread, a chunk of 128 nodes appears to be a good choice. As for the GPU, any value in the range 1K – 8K is adequate, even when the GPU happens to get the last chunk from the shared worklist. So 8K is used.

The CPU-only (GPU-only) solution is derived from our CPU-GPU solution, by ignoring the GPU-control (CPU-control) thread, so that the entire analysis is now performed on the CPU (GPU) alone. The GPU-only solution is the same as the state-of-the-art GPU implementation introduced in [20]. The CPU-only solution is faster than the CPU implementation introduced in [21], based on the experimental results given in [20], since our CPU-only solution is able to apply graph-rewriting rules in parallel without synchronisation.

5.2 Speedups

Figure 5.1 compares the speedups of our CPU-GPU solution of Andersen's analysis against the CPU-only and GPU-only solutions (normalised to GPU-only). For each benchmark, the left bar is for CPU-only, the middle bar for CPU-only and the right bar for our solution. Each of our speedup bars is shown as a breakdown of five components, contributed by (1) NAIVE (the naive workload distribution given in Algorithm 3), (2) IDD (our dynamic workload distribution scheme given in Algorithm 4), (3) Opt I, (4) Opts I + II, and (5) Opts I + II + III, where the three optimisations are described in Section 4.

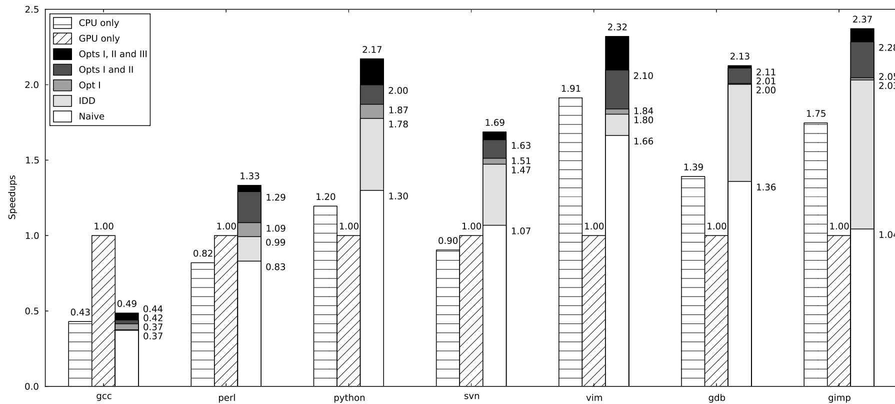


Figure 5.1: Speedups of our parallel Andersen's analysis and their CPU-only and GPU-only versions (normalised to GPU-only).

We observe that the performance ratios of CPU-only over GPU-only vary wildly across these benchmarks. It is therefore not easy to decide which of the two analyses to use for a given program. However, our CPU-GPU solution outperforms (1) CPU-only by 50.6%, (2) GPU-only by 78.5%, and (3) an oracle

that behaves as the faster of (1) and (2) for each benchmark by 34.6% on average. In addition, our solution is faster than the oracle for six benchmarks. The only exception is `gcc`, for which our solution is slightly better than CPU-only but a lot worse than GPU-only. There are two main reasons behind. First, `gcc` induces fewer new points-to edges than the other benchmarks during the analysis. The overhead incurred in CPU-GPU communication and workload distribution is relatively high. Second, the performance gap between GPU-only and CPU-only, 2.3X, is the highest among all the benchmarks (with `vim` coming as the second highest). Thus, workload balance is relatively hard to achieve.

5.3 Dynamic Workload Balancing

Our dynamic workload distribution scheme, IDD, has succeeded in accelerating Andersen's analysis further on top of NAIVE for all benchmarks. The performance improvements are substantial in `python`, `svn`, `gdb` and `gimp`.

To understand why IDD is effective, some statistics are given in Table 5.2. R_x, where x is one of the five rules given in Table 3.1, represents the ratio of the time spent by GPU-only over CPU-only on applying Rule x (accumulated in all iterations). $R_{\mathsf{OFFSET}} < R_{\mathsf{LOAD}} < R_{\mathsf{COPY}} < R_{\Delta\mathsf{COPY}}$ holds across the seven benchmarks. This justifies the priorities assigned to the four rules in (3.4) on CPU and GPU.

Benchmark	IDD workload distribution					
	R_{OFFSET}	R_{LOAD}	R_{COPY}	$R_{\Delta\mathsf{COPY}}$	S_{GPU} (%)	R_{STORE}
perl	0.07	0.17	0.53	1.62	31.0	7.18
python	0.04	0.15	0.70	3.63	37.1	4.52
svn	0.04	0.13	0.61	2.41	34.2	7.29
gcc	0.09	0.40	0.57	0.61	81.6	1.21
gdb	0.04	0.28	0.90	3.31	13.7	5.35
vim	0.08	0.50	0.64	5.59	100.0	0.19
gimp	0.07	0.56	0.64	3.58	77.6	0.87

Table 5.2: Analysis of our CPU-GPU solution (workload distribution).

Let us analyse `gcc` and `vim` to see why performing Andersen's analysis on both CPU and GPU is the least beneficial among the seven benchmarks. For `gcc`, R_{OFFSET}, R_{LOAD}, R_{COPY} and $R_{\Delta\mathsf{COPY}}$ are smaller than 1, indicating that the GPU is more suitable than the CPU for applying these rules. For `vim`, speedup is limited for a different reason. The ΔCOPY rule is particularly expensive to apply, consuming 30.3% (83.0%) of the analysis time of CPU-only (GPU-only). As a result, the GPU is rather inefficient for the entire analysis. As discussed in Section 5.1, the granularities of workloads fetched from the shared worklist by a CPU thread and a GPU kernel are 128 and and 8K nodes, respectively. The GPU stalls for 8.5% of its analysis time, waiting for the CPU to finish. The ratio can be lowered if the granularities are reduced. However, the overall performance even worsens due to less efficient GPU kernel execution and more synchronisation overhead incurred.

Let us look at the effectiveness of the heuristic $DI_{\mathsf{STORE}} \geqslant \tau$ used in determining where to execute the STORE rule, on CPU or GPU, in line 1 of Algorithm 4. In Table 5.2, S_{GPU} stands for the percentage of iterations that the STORE rule

is executed on the GPU. The larger R_{STORE} is (i.e., the slower the GPU is than the CPU in applying the rule), the smaller S_{GPU} is (the fewer iterations that the GPU will be asked to execute the rule). For `vim`, where $S_{GPU} = 100\%$, the `STORE` rule is always applied on the GPU, which is much more efficient than the CPU (with $R_{\mathsf{STORE}} = 0.19$). As the number of stores in a program is small, the benefit of adaptively determining where to execute the `STORE` rule is small. Nevertheless, setting τ to 20 still delivers a speedup of 3.3% (2.9%) compared to when the rule is applied on CPU (GPU) exclusively.

5.4 Optimisations

For the three optimisations described in Section 4, their effects on performance are shown in Figure 5.1. We analyse them using the statistics given in Table 5.3.

Benchmark	Opt I		Opt II	Opt III	S_{REF}
	$O_{\Delta\mathcal{P}}$ (%)	$O_{\Delta\mathcal{C}}$ (%)	O_H (%)	O_I (%)	
perl	8.1	10.6	9.3	21.1	1.31
python	13.2	9.4	3.9	36.2	1.40
svn	10.5	7.4	5.5	13.6	1.20
gcc	15.2	16.5	14.6	21.8	0.48
gdb	12.3	5.0	5.2	15.0	1.10
vim	15.5	9.7	6.7	34.9	1.18
gimp	11.3	5.3	8.8	27.6	1.03

Table 5.3: Analysis of our CPU-GPU solution (optimisations).

Opt I, which overlaps communication with computation, is the most beneficial for `gcc` and `perl` but the least for `gdb` and `gimp`. Its effectiveness depends on the degree of overlap between (1) the process of exchanging their respective $\Delta\mathcal{P}$ sets between the CPU and GPU and (2) the computations performed on the local and remote $\Delta\mathcal{C}$ sets on both the CPU and GPU. In Table 5.3, $O_{\Delta\mathcal{P}}$ and $O_{\Delta\mathcal{C}}$ represent the times elapsed on performing (1) and (2) in percentage, respectively, over the total analysis time for a benchmark. The transfer times for $\Delta\mathcal{P}$ are completely hidden for `gcc` and `perl` since $O_{\Delta\mathcal{P}} < O_{\Delta\mathcal{C}}$ for each benchmark, but only hidden by less than 50% for `gdb` and `gimp` since $O_{\Delta\mathcal{P}} > 2 \times O_{\Delta\mathcal{C}}$ for each benchmark. The (unhidden) communication cost is 8.4% on average, with `gcc` reaching 16.9%, since it induces much fewer edges than the other benchmarks during Andersen's analysis.q

Opt II, which relies on the GPU to identify $\Delta\mathcal{P}$-equivalent variables and the variables for which the `STORE` rule should be applied, is generally more effective than Opt I. In Table 5.3, O_H represents the time spent (in percentage) on these computations over the total analysis time for a benchmark. Opt II is the least effective for `python`, `svn` and `gdb` because the values of O_H for these benchmarks, 3.9%, 5.5% and 5.2%, are small.

Opt III, which decides adaptively whether to perform an iteration of Andersen's analysis on CPU or GPU or both, is profitable for all the benchmarks except `gdb`. In Table 5.3, O_I represents the percentage of iterations executed on the GPU alone. For `gdb`, $O_I = 15\%$. By offloading this much of the total analysis to the GPU, the potential performance benefit obtained may not outweigh the cost incurred. A similar problem exists for `svn`, where $O_I = 13.6\%$.

5.5 Overall Effectiveness

We discuss the effectiveness of our CPU-GPU solution with respect to the reference CPU-GPU solution with its analysis time (3.1) derivable from those of CPU-only and GPU-only solutions. In Table 5.3, S_{REF} represents the speedup of our solution over this reference. Our solution outperforms the reference in six benchmarks with an average speedup of 1.1X. The exception is `gcc` again, for the reasons discussed above.

These results demonstrate the effectiveness of our solution. By dispatching graph-rewriting rules to the "better side" of a CPU-GPU system to apply, our solution performs better than the reference, for which even zero communication and synchronisation overhead has been assumed.

6 Related Work

There is no shortage of optimisations on Andersen's pointer analysis in the sequential setting [12, 14, 23, 25, 29, 32, 34]. Andersen's analysis is $O(n^3)$, where n is the number of variables, when difference propagation, $\Delta\mathcal{P}$, is employed to reduce the work of propagating points-to edges [9]. Some later improvements can be found in [24] [29]. In the GPU implementation of Andersen's analysis introduced in [20], $\Delta\mathcal{P}$ is used to overlap the analysis with the transfer of the new points-to information back from the GPU to the host. This paper represents the first to take advantage of difference propagation for not only $\Delta\mathcal{P}$ but also $\Delta\mathcal{C}$ to reduce the amount of data exchanged between CPU and GPU.

There is a lot of work on parallelising graph algorithms such as breadth-first search (BFS) and single-source shortest paths on CPUs [1, 16] and on GPUs [3, 13, 16]. Unlike Andersen's analysis, these graph algorithms do not modify the structure of the underlying graph. In the case when some modifications are made [26], the modifications can be predicted statically. Therefore, these existing techniques cannot be directly used to parallelise Andersen's analysis.

Recently, there are a few attempts on parallelising Andersen's analysis on multi-core CPUs [21, 27] or multi-core GPUs [20]. This paper presents the first parallel solution on a CPU-GPU system, where both its CPU and GPU cores are used, based on the state-of-the-art GPU implementation reported in [20]. By taking advantage of the performance characteristics of two different architectures, Andersen's analysis can be accelerated further if both CPU and GPU are used.

There are also a lot of efforts on parallelising graph algorithms on CPU-GPU systems, where the structure of the underlying graph is not modified. In [16], the best from a few implementations of BFS is selected dynamically for each level of the BFS algorithm. Later, workload-aware and fixed-partitioned-space strategies [22] are considered for BFS. In [10], a graph programming model, Totem, is presented and applied to two applications, BFS and PageRank. In [11], workload distribution is studied in the Totem framework, by distinguishing workloads in terms of node degrees, so that the suitability of CPU or GPU for the workloads can be estimated. However, this scheme cannot be directly applied to Andersen's analysis since the degree of a node changes dynamically in an unpredictable manner during the analysis. In this paper, we have introduced a new dynamic workload distribution to minimise workload imbalance

for Andersen's analysis.

A large number of techniques have been proposed for accelerating loop-oriented programs on GPUs [4, 5, 7, 6, 8, 33, 36]. However, these techniques cannot be directly applied to parallelise Andersen's analysis.

7 Conclusion

This paper describes the first parallel implementation of Andersen's analysis on a CPU-GPU system. The presence of dynamic and unpredictable modifications to a constraint graph makes it difficult to balance workloads between CPU and GPU. The sparsity of a constraint graph posts obstacles in engineering efficient CPU-GPU communication. To overcome these two challenges, we distribute graph-rewriting rules to the CPU or GPU that is better suited for processing the rules and adopt difference propagation of points-to information between the CPU and GPU to reduce the communication cost. On a set of seven C programs evaluated, our CPU-GPU solution outperforms on average the state-of-the-art CPU-only and GPU-only implementations.

Bibliography

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.

[2] Lars Ole Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen*, 1994.

[3] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on CUDA. In *IPDPS*, pages 544–555, 2011.

[4] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS*, pages 225–234, 2008.

[5] Peng Di, Qing Wan, Xuemeng Zhang, Hui Wu, and Jingling Xue. Toward harnessing doacross parallelism for multi-gpgpus. In *ICPP*, pages 40–50, 2010.

[6] Peng Di, Hui Wu, Jingling Xue, Feng Wang, and Canqun Yang. Parallelizing sor for GPGPUs using alternate loop tiling. *Parallel Computing*, 38(6-7):310–328, 2012.

[7] Peng Di and Jingling Xue. Model-driven tile size selection for DOACROSS loops on GPUs. In *Euro-Par*, pages 401–412, 2011.

[8] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs. In *ICPP*, pages 350–359, 2012.

[9] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *ESOP*, pages 90–104. Springer-Verlag, 1998.

[10] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*, pages 345–354, 2012.

[11] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. On graphs, GPUs, and blind dating - a workload to processor matchmaking quest. In *IPDPS*, 2013.

[12] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, volume 42, pages 290–299, 2007.

[13] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208, 2007.

[14] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of C code in a seconds. In *CGO*, pages 254–263, 2001.

[15] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, pages 267–276, 2011.

[16] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, pages 78–88, 2011.

[17] 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 *ISCA*, pages 451–460, 2010.

[18] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *SIGSOFT FSE*, pages 343–353, 2011.

[19] Daniel Lustig and Margaret Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *HPCA*, 2013.

[20] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, pages 107–116, 2012.

[21] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443, 2010.

[22] Lluis-Miquel Munguia, David A Bader, and Eduard Ayguade. Task-based parallel breadth-first search in heterogeneous environments. In *HiPC*, pages 1–10, 2012.

[23] Rupesh Nasre and R.Govindarajan. Prioritizing constraint evaluation for efficient points-to analysis. In *CGO*, 2011.

[24] David J Pearce, Paul HJ Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *SCAM*, pages 3–12, 2003.

[25] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126–135, 2009.

[26] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. Eigencfa: accelerating flow analysis with GPUs. In *POPL*, pages 511–522, 2011.

[27] Sandeep Putta and Rupesh Nasre. Parallel replication-based points-to analysis. In *CC*, pages 61–80, 2012.

[28] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

[29] Manu Sridharan and StephenJ Fink. The complexity of Andersens analysis in practice. In *SAS*, volume 5673, page 205, 2009.

[30] Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO*, pages 1–11, 2013.

[31] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA*, pages 254–264, 2012.

[32] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw., Pract. Exper.* To appear.

[33] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *ICS*, pages 244–255, 2009.

[34] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS*, pages 180–195, 2002.

[35] X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA*, pages 188–198, 2011.

[36] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.

[37] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO*, pages 218–229, 2010.