Title
Accelerating Dynamically-Typed Language on Heterogeneous Platforms

Permalink
https://escholarship.org/uc/item/6jw316ct

Author
Qunaibit, Mohaned

Publication Date
2019

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA,  
IRVINE

Accelerating Dynamically-Typed Language on Heterogeneous Platforms

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Mohaned Qunaibit

Dissertation Committee:
Professor Michael Franz, Chair
Professor Alex Nicolau
Professor Ian G. Harris

2019
DEDICATION

I dedicate this dissertation to my dearest parents, Yousef and Mona. Thank you so much for all the love, endless support, wisdom, and encouragement. I, also, dedicate this dissertation to my lovely wife, Maha, and all of my wonderful brother and sisters, Hanadi, Sahar, Rasha, Abdulrahman, Alanoud and Joud.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>x</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>xii</td>
</tr>
</tbody>
</table>

1 Introduction
   1.1 Motivation .......................................................... 1
   1.2 Contributions ...................................................... 4

2 Background
   2.1 Graphics Processing Units (GPUs) ................................. 6
   2.2 Heterogeneous Programming Frameworks ............................ 7
   2.3 Interpreters and Virtual Machines ............................... 7

3 The MegaGuards System
   3.1 Overview ............................................................. 10
   3.2 Lightweight Pre-assessment ...................................... 12
   3.3 Guards Optimization .............................................. 13
      3.3.1 Type Stability Analysis .................................... 15
      3.3.2 Interprocedural Analysis Support .......................... 18
      3.3.3 MEGAGUARDS-Specialized AST ............................. 19
      3.3.4 Bounds Check Optimization ................................ 20
      3.3.5 Mega Guards Insertion ..................................... 22
   3.4 Parallel Analysis and Execution ................................ 23
      3.4.1 Dependence Analysis ......................................... 23
      3.4.2 Kernel Code Generation ..................................... 25
      3.4.3 Thread Mapping .............................................. 25
      3.4.4 Kernel Data Management .................................... 27
      3.4.5 Kernel Execution and Device Selection .................... 28
      3.4.6 Execution of A Cached Kernel .............................. 29
### 4 Redizer: A Transparent Parallel Reductions Transformation

4.1 Background and Motivation ................................................. 49
4.2 Overview ........................................................................... 52
4.3 Reduction Identification ....................................................... 53
4.4 Loop Flattening .................................................................. 56
4.5 Reduction Transformation and Thread Mapping ......................... 58
  4.5.1 Nestedness-Aware Reduction Transformation .......................... 59
  4.5.2 Two-Stage Reduction Transformation .................................... 62
4.6 Code Generation and Thread Scheduling .................................... 65
  4.6.1 Nestedness-Aware Reduction Code Generation ......................... 65
  4.6.2 Two-Stage Reduction Code Generation ................................. 68
  4.6.3 Loop Unrolling and Shuffling Instruction ............................... 71
4.7 Shared Storage Initialization Strategy ....................................... 71
4.8 Evaluation ........................................................................... 72
  4.8.1 System Setup .................................................................. 72
4.9 Performance vs Manual Native Implementation ......................... 72
  4.9.1 Benchmark Selection ....................................................... 72
  4.9.2 Experimental Setup ....................................................... 73
  4.9.3 Discussion .................................................................... 73
4.10 Performance vs Native Reduction APIs ..................................... 77
  4.10.1 Benchmark Selection ...................................................... 77
  4.10.2 Experimental Setup ...................................................... 77
  4.10.3 Discussion .................................................................. 78

### 5 Related Work

5.1 Type Inference and Guards Optimization .................................. 81
5.2 Heterogeneous Programming in Dynamic Languages .................. 82
5.3 Auto-Parallelization for Dynamic Languages ............................. 83
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Node specialization in Truffle.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Handling mis-speculation using type guards.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>GraalPython type specialization.</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>GraalPython+MEGAGUARDS System Overview.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>MEGAGUARDS unboxing process.</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>MEGAGUARDS specialized AST with inter-procedural invocations.</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>MEGAGUARDS specialized AST build process.</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>MEGAGUARDS bounds check optimization process.</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>MEGAGUARDS detailed internal analysis and offloading.</td>
<td>23</td>
</tr>
<tr>
<td>3.7</td>
<td>MEGAGUARDS polyhedral dependence analysis of a MEGAGUARDS-specialized AST.</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>MEGAGUARDS thread mapping and code generation.</td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>Sequential execution speedup of MEGAGUARDS-Truffle compared to CPython, PyPy, GraalPython normalized to GraalPython.</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Parallel execution speedup of MEGAGUARDS compared to OpenCL C/C++ (CPU and GPU) normalized to GraalPython on a ( \log_{10} ) scale.</td>
<td>40</td>
</tr>
<tr>
<td>3.11</td>
<td>Breakdown of MEGAGUARDS passes for parallel execution.</td>
<td>42</td>
</tr>
<tr>
<td>3.12</td>
<td>Peak performance of MEGAGUARDS with different data sizes for parallel execution.</td>
<td>44</td>
</tr>
<tr>
<td>3.13</td>
<td>Mean speedup of warming up steps of MEGAGUARDS normalized runs for GraalPython.</td>
<td>45</td>
</tr>
<tr>
<td>3.14</td>
<td>Mean speedup of warming up steps of MEGAGUARDS-CPU with various CU counts normalized runs for GraalPython.</td>
<td>46</td>
</tr>
<tr>
<td>3.15</td>
<td>Effect of KDM optimization on MEGAGUARDS.</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Python matrix multiplication implementation using simply written loop versus using reduction APIs.</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>REDIZER transformation overview.</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Example reduction identification process.</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Loop flattening transformation process for the imperfectly nested loops.</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Nestedness-Aware REDIZER slicing each thread to perform a tree-based reduction.</td>
<td>60</td>
</tr>
<tr>
<td>4.6</td>
<td>Nestedness-Aware AST reduction transformation.</td>
<td>61</td>
</tr>
<tr>
<td>4.7</td>
<td>Two-Stage reduction performing a tree-based reduction.</td>
<td>62</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.8</td>
<td>Two-Stage AST transformation process.</td>
<td>64</td>
</tr>
<tr>
<td>4.9</td>
<td>Nestedness-Aware Reduction Code Generation.</td>
<td>66</td>
</tr>
<tr>
<td>4.10</td>
<td>Two-Stage Reduction Code Generation.</td>
<td>69</td>
</tr>
<tr>
<td>4.11</td>
<td>Speedup of MEGAGUARDS with REDIZER (CUDA and OpenCL using GPU) compared to OpenCL C/C++ (GPU) and GraalPython normalized to MEGAGUARDS without REDIZER (OpenCL using GPU).</td>
<td>73</td>
</tr>
<tr>
<td>4.12</td>
<td>Mean speedup of running steps for $Set_R$ benchmarks normalized to MEGAGUARDS without REDIZER, all using GPU</td>
<td>76</td>
</tr>
<tr>
<td>4.13</td>
<td>Scalability of MEGAGUARDS with REDIZER (CUDA and OpenCL) with different $Set_R$ benchmarks’ input sizes compared to OpenCL C/C++ normalized to MEGAGUARDS without REDIZER, all using GPU</td>
<td>79</td>
</tr>
<tr>
<td>4.14</td>
<td>Peak bandwidth throughput on NVIDIA GPU GTX 1080 Ti for MEGAGUARDS with REDIZER, NVIDIA CUB, Thrust and PGI OpenACC, all using CUDA as backend framework.</td>
<td>80</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Sequential execution time (in seconds) for MEGAGUARDS-Truffle, CPython, PyPy and GraalPython. ......................................................... 37
3.2 Benchmark characteristics for parallel execution. .......................... 38
3.3 Time (in milliseconds) for each pass of the parallel execution. ............ 38
3.4 Parallel execution time (in seconds) for MEGAGUARDS, OpenCL C/C++ (CPU and GPU), and sequential GraalPython. ................................. 41

4.1 Execution time, in seconds, for benchmarks in $set_R$ for MEGAGUARDS without REDIZER, MEGAGUARDS with REDIZER, OpenCL C/C++ (GPU) and sequential GraalPython .......................................................... 74
4.2 Transformation cost, in milliseconds, of REDIZER and number of transformed loops using the Nestedness-Aware and Two-Stage transformations. ... 75
ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Michael Franz. Over the past six years, he has been an astonishing mentor who has provided guidance and encouragement throughout my studies at UCI. Professor Michael Franz welcomed me into his Secure Systems and Software lab. I had the opportunity to collaborate, work and meet with an outstanding group of researchers and friends. It has been one of my most gratifying period of my life, and I will be eternally grateful for Professor Michael Franz.

I would like to thank Professor Alex Nicolau and Professor Ian Harris who accepted to serve on my committee. I would like to express my gratefulness to my postdoc Dr. Yeoul Na, Professor Stefan Brunthaler, Professor Stijn Volckaert, for their invaluable insights, contributions and advice throughout my research journey. I am especially grateful to lab alumni, Andrei Homescu, Stephen Crane, Julian Lettner, Brian Belleville, Gulsem Savrun Yeniceri, Wei Zhang, Codrut Stancu, Mark Murphy, Eric Hennigan and Per Larsen, for their invaluable support and willingness to share their expertise. In addition, I would like to thank my friends in the lab for being great colleagues and for making graduate school much more enjoyable. I would like to thank to my great lab mates for making graduate school a wonderful experience, especially: Prabhu Rajasekaran, Alexios Voulimeneas, Joseph Nash, Dokyung Song, Taemin Park, Anil Altinay, Paul Kirth, Mitchel Dickerson, Matt Dees, Min-Yih Hsu and Fabian Parzefall.

This research is based upon work partially supported by Ministry of Higher Education (MOHE), in Saudi Arabia, Graduate Studies Scholarship program, the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the National Science Foundation under awards CNS-1513837 and CNS-1619211, and by the Office of Naval Research under award N00014-17-1-2782. The competence center SBA Research (SBA-K1) is funded within the framework of COMET Competence Centers for Excellent Technologies by BMWIT, BMDW, and the federal state of Vienna, managed by the FFG.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, the Office for Naval Research, or any other agency of the U.S. Government.
CURRICULUM VITAE

Mohaned Qunaibit

EDUCATION

Doctor of Philosophy in Computer Science 2019
University California, Irvine

Master of Science in Computer Science 2014
University California, Irvine

Bachelor of Science in Computer Science 2008
King Saud University

RESEARCH EXPERIENCE

Graduate Student Researcher 2013–2019
University of California, Irvine

WORK EXPERIENCE

Application Specialist 2009–2011
Saudi Basic Industries Corporation

TEACHING EXPERIENCE

Teaching Assistant Fall 2018
COMPSCI 141: Concepts in Programming Languages
University of California, Irvine

Teaching Assistant Winter 2019
COMPSCI 142A: Compiler and Interpreter
University of California, Irvine
REFEREED PUBLICATIONS

Automatic Parallel Reductions on GPUs: Just-In-Time Performance for Portable Python Code
Qunaibit, Mohaned and Na, Yeoul and Blaser, Luc and Franz, Michael

Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization
Qunaibit, Mohaned and Brunthaler, Stefan and Na, Yeoul and Volckaert, Stijn and Franz, Michael
32nd European Conference on Object-Oriented Programming

Losing control: On the effectiveness of control-flow integrity under stack attacks
Conti, Mauro and Crane, Stephen and Davi, Lucas and Franz, Michael and Larsen, Per and Negro, Marco and Liebchen, Christopher and Qunaibit, Mohaned and Sadeghi, Ahmad-Reza
22nd ACM SIGSAC Conference on Computer and Communications Security

SOFTWARE

MegaGuards https://github.com/securesystemslab/megaguards/
Parallel framework for Truffle languages.

GraalPython https://github.com/securesystemslab/graalpython-megaguards/
GraalPython with our Parallel framework (MEGA GUARDS) support.
Scientific applications are ideal candidates for the “heterogeneous computing” paradigm, in which parts of a computation are “offloaded” to available accelerator hardware such as GPUs. However, when such applications are written in dynamic languages such as Python or R, as they increasingly are, things become less straightforward. The same flexibility that makes these languages so appealing to programmers also significantly complicates the problem of automatically and transparently partitioning a program’s execution between a CPU and available accelerator hardware without having to familiarize themselves with a variety of annotations, libraries, and idiosyncrasies superimposed by existing frameworks.

A common way of handling the features of dynamic languages is by introducing speculation in conjunction with guards to ascertain the validity of assumptions made in the speculative computation. Unfortunately, a single guard violation during the execution of offloaded code may result in a huge-performance penalty and necessitate the complete re-execution of the offloaded computation. In the case of dynamic languages, this problem is compounded by the fact that a full compiler analysis is not always possible ahead of time.

We present MEGAGUARDS, a new approach for speculatively executing dynamic languages on heterogeneous platforms in an automatic and transparent fashion. Our method translates each target function or loop into a single static region devoid of any dynamic type features.
The dynamic parts are instead handled by a construct that we call a *mega guard* which checks all the speculative assumptions ahead of its corresponding static region. Furthermore, as part of improving the performance of massively parallel architectures, we introduced a loop transformation for arbitrary reduction operations. Notably, the advantage of MEGA GUARDS is not limited to heterogeneous computing. Since it removes guards from compute-intensive function calls and loops, the approach also improves sequential performance.

Our experiments indicate that MEGA GUARDS is approaching the performance-level of hand-optimized OpenCL C/C++ code, while simultaneously retaining economical Python implementations. Thus, MEGA GUARDS unites the efficiency and productivity of Python with the cutting-edge performance of heterogeneous computing.
Chapter 1

Introduction

1.1 Motivation

Heterogeneous computing, in which the execution of a program is shared between a CPU and other hardware such as a GPU or dedicated accelerator chips, is gaining in importance. By judiciously “offloading” some of the computations to available acceleration hardware, performance can in many cases be raised far beyond single threaded CPU capabilities.

Unfortunately, writing programs for heterogeneous computing is difficult. Some researchers are focusing on “transparent” approaches, in which computations are distributed to hardware accelerators fully automatically, while others are concentrating on a more manual approach in which programmers guide this process explicitly through specific programming language constructs or compiler-directed annotations. As Hager et al. noted in 2015 [41], for the long term it is still an open question

[...] whether accelerators will be automatically invoked by compilers and runtime systems, [...] or be explicitly managed by application programmers.
Looking back on decades of successful research on automated compiler optimizations, we can state that “transparent” approaches that perform optimizations without programmer intervention and that can automatically adapt to changes in available accelerator hardware are clearly preferable to manual approaches that might require program re-writing each time that the hardware is changed. Accordingly, much of the overall research on heterogeneous computing has focused on this automation aspect. A closer look at this prior work, however, reveals that most research on automating heterogeneous computing has centered on statically-typed programming languages.

When looking at existing research on heterogeneous computing for dynamic programming languages such as Python, we find that the emphases are reversed: most of the work in the dynamic languages domain focuses on explicit manual management of accelerators through programmer-directed addition of source code annotations and/or the use of idiosyncratic libraries. Getting to know these annotations and libraries is a time-consuming obstacle that may prevent programmers from re-writing their code to benefit from heterogeneous programming. In addition, customizing code to adhere to one specific library vs another naturally inhibits a program’s portability. Moreover, these libraries and annotations often force programmers to abandon the flexibility afforded by dynamic typing.

Automating heterogeneous computing is challenging for dynamic languages because the dynamic types of objects may change at any time during program execution. Consider, for example, an operation that changes from an integer addition to a string concatenation as a result of a type change in an underlying operand. In a dynamic compilation environment, such code will probably first be optimized to an integer addition. When then the type change is captured by a runtime type check, i.e., via a guard, the existing optimization is invalidated and the execution falls back to the interpreter or a less optimized version of the code. Eventually, it may then be optimized again for the new type.

Now consider what happens when such mis-speculation happens during the execution of a
piece of code that has been offloaded to a hardware acceleration device; we will call such pieces of code “kernels” in the remainder of this thesis. In case of a mis-speculation, the existing kernel may become invalid. But because the kernel was executing on a device external to the CPU, the performance penalties may be much higher than merely dropping back into an interpreter or a lower level of optimization. In the worst case, it may not even be possible to salvage the results computed so far, so that re-execution of the whole kernel will be required.

In general, transparent offloading may require complex static analyses such as points-to analysis to check dependencies across loop iterations. The additional code required to handle mis-speculations complicates the program to analyze further, making adoption of such static analysis techniques to dynamic compilation very difficult.

To overcome these challenges, we propose MEGAGUARDS, which removes the obstacles in dynamic languages that prevent compute-intensive loops to be transparently offloaded to GPUs or other acceleration devices. MEGAGUARDS translates a loop as a static region in which type changes or type mis-speculations do not exist. The key insight and novelty of MEGAGUARDS is how it guarantees that the offloaded code does not encounter type changes or type mis-speculations. To this end, MEGAGUARDS conducts a type stability analysis for loops to see if all the guards can be safely moved outside of the loops. If so, MEGAGUARDS removes all the guards from the loop and constructs a single guard, i.e., a “mega guard,” which checks all the speculative assumptions ahead of the loop. This way the loop itself can be seen as the static region. MEGAGUARDS then offloads among the stabilized loops if it can prove that the loop does not have any cross-iterational dependencies. The advantage of MEGAGUARDS, however, is not limited to enabling offloading. Since it removes guards from the loop, MEGAGUARDS also improves performance on a single threaded CPU.

We implemented MEGAGUARDS in GraalPython [109], a modern Python 3 implementation targeting the Java Virtual Machine (JVM). GraalPython relies on the Truffle framework.
to optimize interpreted programs on the CPU [103]. To determine parallelizable loops, we perform a bounds check optimization and conduct dependence analysis by leveraging the polyhedral model [36]. MEGAGUARDS dynamically translates parallel loops into OpenCL or CUDA code, which it then executes on the fastest acceleration device available on the target system. MEGAGUARDS significantly improves the performance of data-parallel applications over sequential execution of Python. Even if we cannot parallelize a loop, we still translate it to a guard-less AST, which also improves the sequential performance.

### 1.2 Contributions

This dissertation makes the following contributions, organized by chapter:

#### Chapter 3: The MegaGuards System

- We introduce a novel technique, MEGAGUARDS, that eliminates type speculation inside of loops to efficiently offload speculative code to kernels (Section 3.3.1). Eliminating speculation inside of loops also improves sequential performance (Section 3.5).

- We describe the design and implementation of MEGAGUARDS, a Python-based system that transparently offloads data-parallel loops to an acceleration device, such as a GPU, without requiring code rewriting or annotations from the programmer.

- We introduce a memory management design for acceleration devices that lift the burden of manually allocate, share and free data for all the generated acceleration code within MEGAGUARDS.

#### Chapter 4: Redizer Transformation

...
• We extend MEGAGUARDS with an effective optimization method for parallelizing complex reduction-style loops on GPUs, such as nested loops and loop sequences with one or multiple intra- or inter-loop reduction variables, supporting different aggregator operators (including even minus and divide).

• A practical Python engine extension using this reduction loop optimization that in our experiments, achieves 64% of the optimal hand-crafted native GPU performance automatically, i.e. without any programmer help or use of a specific library or GPU programming construct.

• We report an experimental evaluation showing that there is a substantial acceleration potential by including reduction dependencies in loop parallelization, leading to a 3x faster runtime in our benchmark of compute-intense Python code.
Chapter 2

Background

2.1 Graphics Processing Units (GPUs)

GPUs were specifically designed to perform graphics computations, thousands or millions of pixels, in parallel instead of the CPU. However in recent years, GPUs found their way into general-purpose programming with appearance of the programming models, i.e. CUDA [70] and OpenCL [90].

GPU architecture is based upon the execution model of single instruction, multiple thread (SIMT), a variant model of single instruction multiple data (SIMD) [44]. Commonly, a GPU consists of multiple compute units (CUs) where each CU has multiple streaming processors (SMs), i.e. computing cores. A group of SMs within a CU is referred to as warp or wavefront, where SMs within a warp will be fed by a single instructions stream, i.e. each thread will execute the same instruction in lock-step manner. Thus, instructions stream control-flow divergence require the GPU to execute each path sequentially and masking-out or turning-off SMs that do not take a path, which causes performance degradation and under-utilization of SMs [2, 24, 42]. Distributing and scheduling threads to SMs within a GPU are
by the GPU vendor’s driver and/or hardware based scheduler, e.g. Giga Thread scheduler.

2.2 Heterogeneous Programming Frameworks

Programming in a heterogeneous computing environment is highly challenging because heterogeneous programming frameworks (e.g., CUDA and OpenCL) have steep learning curves and requiring knowledge of the inner workings of the GPU.

To alleviate this problem for statically-typed languages, researchers have proposed transformations that map existing parallel paradigms for the CPU to run on the GPU [72, 37, 102]. Others proposed libraries and lambda expressions [33, 48, 78, 87, 61, 11] to automatically generate GPU code. Some techniques automatically parallelize sequential loops and run them on GPUs [59, 5]. New languages such as Lime [27, 3] implicitly perform parallel computations on GPUs.

Dynamically-typed languages have fewer options to simplify GPU programming and must typically resort to external APIs for generating OpenCL or CUDA code. Python programmers, for example, can use libraries such as Numba to design kernel code targeting CUDA. In-depth knowledge of the GPU’s architecture and manual data management remains necessary to use these libraries.

2.3 Interpreters and Virtual Machines

The fact that variable types can change at any moment in dynamically-typed languages hinders ahead-of-time optimization. The rate at which variable types change in practice is, however, usually minimal [23, 106]. This observation has inspired various specialization approaches that minimize the interpreter’s type-checking overhead [12, 13, 105, 100, 1, 111].
In our work, we leverage specialized types to eliminate all type-checking in the generated OpenCL and CUDA code.

Truffle [103], the self-optimizing runtime system we use in MEGAGUARDS, performs specialization via automatic node rewriting on an abstract syntax tree (AST). Truffle speculatively replaces generic AST nodes, which are capable of operating on variables of any data type, with nodes that are specialized for a specific data type (Figure 2.1). This speculation approach facilitates just-in-time compilation of Truffle’s hosted languages, which include GraalPython, FastR, and TruffleRuby. When a Truffle AST reaches a stable state, the Truffle framework invokes the Graal just-in-time compiler [28, 104] to further optimize the Truffle AST through partial evaluation and to compile the AST into highly optimized machine code.

To preserve the correctness of the program execution, Truffle must be able to handle mis-speculation. Figure 2.2 shows how Truffle embeds type guards into the specialized AST. Guards verify that the specialized input types for a node match the expected types, and trigger deoptimization if they detect a mismatch. When a node’s return type mismatches the specialized data type, an exception is thrown and Truffle also proceeds to deoptimize the node. During deoptimization, Truffle discards the specialized node and replaces it by a generic node.

GraalPython	extsuperscript{1}, the Python 3 VM we use in MEGAGUARDS, is built on top of Truffle.

\textsuperscript{1}ZipPy’s predecessor
boolean guard(Object left, Object right) {
    if (!((left instanceof Double)))
        return false;
    if (!((right instanceof Double)))
        return false;
    return true;
}

double doubleMul(VirtualFrame frame) {
    Object left = leftNode.execute(frame);
    Object right = rightNode.execute(frame);
    if (guard(left, right))
        return (double) left * (double) right;
    else {
        transferToInterpreterAndInvalidate();
        return this.replace(GenericMulNode()).execute(left, right);
    }
}

Figure 2.2: Handling mis-speculation using type guards.

# Python 3
1  a = [ i*2. for i in range(10)]
...  
2  a[1] = 'text'
...  
3  b = [[ 0. for j in range(10)] for i in range(20)]

Figure 2.3: GraalPython type specialization.

GraalPython’s type system specializes objects based on their content. In Figure 2.3, we see several examples of type specialization in GraalPython. At 1, the program creates a list a containing items of the same type. GraalPython internally specializes this list to be of type DoubleList. At 2, one of the list items is replaced by a value of a different type. Here, GraalPython generalizes the list to be of type ObjectList. At 3, the program creates a multi-dimensional list b. In this case, GraalPython specializes the nested lists to be of type DoubleList. GraalPython stores variables values in a virtual frame corresponding to the context that variables have been created in. This virtual frame is usually referred to as the context frame. Each variable in the context frame maintains its specialized type. This object layout design assists Truffle specialization process and minimizes node type generalization (i.e., deoptimization).
Chapter 3

The MegaGuards System

3.1 Overview

Figure 3.1 shows how MEGAGUARDS fits into the GraalPython ecosystem. Conceptually, MEGAGUARDS works as follows. First, whenever the interpreter executes a loop with an identifiable index expression, such as the \texttt{for i in range(n)} statement in Figure 2.1, MEGAGUARDS determines if the loop is a potential candidate for offloading to an accelerator device (Section 3.2). If the loop is a suitable candidate, MEGAGUARDS analyzes if the loop can be stabilized using our type stability analysis. If so, MEGAGUARDS eliminates all type checks from the loop and creates a \textit{mega guard} which checks all the speculative assumptions outside the loop (Section 3.3).
MegaGuards then performs a bounds check optimization analysis and marks operations that require run-time checks (Section 3.3.4). After that, MegaGuards performs a depen-
dence analysis to see if the loop iterations are independent of each other and thus can be safely offloaded (Section 3.4.1). MEGAGUARDS then optimizes the AST of the parallelizable loop and translates it into OpenCL or CUDA kernel code (Section 3.4.2). Finally, MEGAGUARDS compiles the OpenCL or CUDA kernel and adaptively selects the best acceleration device to offload (Section 3.4.5). If MEGAGUARDS finds that a loop is not a candidate for offloading, MEGAGUARDS will force GraalPython to execute that loop on top of Graal, a dynamic compiler. If the loop is proven to be type stable, however, MEGAGUARDS will still perform the mega guard optimization.

3.2 Lightweight Pre-assessment

MEGAGUARDS begins its analysis when the interpreter reaches a loop with an identifiable index expression that has an explicit number of iterations. MEGAGUARDS considers the loop a suitable candidate for offloading if its step sizes are constant.

For suitable candidate loops, MEGAGUARDS traverses the AST sub-tree constituting the loop to ensure that all the instructions in the loop are supported by the OpenCL and CUDA frameworks.
3.3 Guards Optimization

Truffle uses type guards and exceptions to handle mis-speculations, as shown in Section 2.3. MegaGuards hoists type, bounds, and overflow checks out of a loop to translate the loop into a *static region*. This way these checks are performed *before* that loop is executed. To this end, MegaGuards performs type stability analysis for each AST node, identifies all the input data to be type-guarded, and generates specialized, strongly-typed ASTs. Moreover, MegaGuards analyzes array subscripts and arithmetic operations in affine expressions to optimize bounds and overflow checks. The nodes in a specialized AST do not contain type checks but may contain bounds and arithmetic overflow checks that MegaGuards is unable to optimize (see Section 3.3.4).
ALGORITHM 1: Type Stability Analysis Algorithm.

Function DominantType(left, right)
Result: Return the strongest data type (e.g., (double, long) → double)
if left == None then return right;
else if right == None then return left;
else if left > right then return left;
else return right;
end

Function NodeVisitor(node) /* Depth-First tree traversal */
Result: Return data type of the tree
op ← node.getOp()
if op == AssignmentNode then
    leftDataType ← NodeVisitor(node.getLeftChild())
    rightDataType ← NodeVisitor(node.getRightChild())
    if leftDataType == rightDataType then
        return leftDataType
    else /* Possible data type change */
    | Exit MegaGuards and transfer to interpreter
    end
else if op == IfElseAssignmentNode then
    /* e.g \( \alpha = (1 \text{ if } \beta > 0 \text{ else } 2) \) */
    leftDataType ← NodeVisitor(node.getLeftChild())
    thenDataType ← NodeVisitor(node.getThenChild())
    elseDataType ← NodeVisitor(node.getElseChild())
    if thenDataType == elseDataType and leftDataType == thenDataType then
        return leftDataType
    else /* Possible data type change */
    | Exit MegaGuards and transfer to interpreter
    end
else
    if node is User-Defined Function Call then
        returnDataType ← None
        Enter new Scope
        foreach argument in node.getArguments() do
            /* assign argument data types to the function parameter */
            parameter ← NodeVisitor(argument)
        end
        returnDataType ← NodeVisitor(node.getFunctionRoot())
        /* assert all return sites have the same data type */
        Exit Scope
        return returnDataType
    else /* Other nodes, e.g., binary arithmetic, return, etc. */
        currentDataType ← None
        foreach child in node.getChildren() do
            childDataType ← NodeVisitor(child)
            currentDataType ← DominantType(childDataType, currentDataType)
        end
        return currentDataType
    end
end
MegaGuards now assesses the type stability of the loop. We say a loop is type-stable if we can deduce a single data type for each node and can guarantee all potential type changes can only result from outside the loop, not from the inside. MegaGuards performs this type stability analysis before executing or profiling the loop but it leverages type feedback information of live-in variables available in the context frame maintained by GraalPython (see Section 2.3). In figure 3.2, MegaGuards runs an unboxing pass on variables of generic boxed types (e.g., object lists) to augment the context frame with more precise information. If MegaGuards finds multiple types within in the same boxed data structure (e.g., a list...
# Python 3

```python
a = [ i for i in range(5)]
b = [ i*1. for i in range(5)]

def qux(x):
    return x*2
def baz(a, b, n):
    for i in range(n):
        b[i] = qux(b[i]) * qux(a[i])
baz(a, b, 5)
```

Figure 3.3: MEGAGUARDS specialized AST with inter-procedural invocations.

that stores both strings and integers), it will mark that structure as type-unstable in the context frame.
Guards Optimization:

Type Stability Algorithm

Build Specialized AST

Bounds Check Optimization

Generate specialized AST

Generate mega guard

boolean megaguard(Object n, Object m, Object alpha, Object a, Object b) {
    if (!(n instanceof Integer))
        return false;
    if (!(m instanceof Integer))
        return false;
    if (!(alpha instanceof Double))
        return false;
    if (!(Unbox(a) instanceof Double[][]))
        return false;
    if (!(Unbox(b) instanceof Double[][]))
        return false;
    if (!(this.validateBoundsAndOverflowAssumptions())
        return false;
    return true;
}

Figure 3.4: MEGA GUARDS specialized AST build process.
After unboxing, MEGAGUARDS runs Algorithm 1 on each AST node in a loop body to infer the type of each node, and to verify the type stability of each statement. The main method in the algorithm, NodeVisitor, traverses the loop’s AST statements in depth-first order, propagating the data types from the augmented context frame through each operation. For assignment operations, represented by AssignmentNode nodes in the AST, our algorithm consults the context frame to check if the source (rightDataType) and target (leftDataType) data types are the same. If so, the assignment operation itself is given that data type. If not, the assignment node is considered type-unstable, and MEGAGUARDS will force the entire loop to be executed by the interpreter. Similarly, MEGAGUARDS tags IfElseAssignmentNode nodes with the data type of its child nodes unless any of its child nodes have different data types, or if any child node is marked as type-unstable. In both of these cases, MEGAGUARDS forces the interpreter to execute the loop instead. MEGAGUARDS does, however, re-evaluate loops it fails to offload should they ever be executed again. For operations such as binary arithmetic operations and function calls, the algorithm tags the operation with the dominant type of the operation’s child nodes using the DominantType method. If our algorithm determines that all operations in the AST are type-stable, it will return the inferred data types for each node.

3.3.2 Interprocedural Analysis Support

To support interprocedural type stability analysis, MEGAGUARDS does function cloning: it creates new variants of functions called within loops and specializes each variant based on its argument types. Figure 3.3 shows an example of a loop with two function calls. MEGAGUARDS runs Algorithm 1 on the loop in function baz. While traversing the loop’s AST, MEGAGUARDS identifies a user-defined function call to qux with one argument, b[i]. MEGAGUARDS creates a specialized version of this function using Algorithm 1. Since b[i] is of type double, the algorithm can determine that this specialized version of function qux
returns a value of type double. MEGAGUARDS reports this return type back to the call site in the loop and continues the loop traversal. MEGAGUARDS then identifies another call to function qux with argument a[i] of type int. Since MEGAGUARDS has only created a variant of qux specialized for arguments of type double, MEGAGUARDS creates another variant here specialized for argument type int.

### 3.3.3 MEGAGUARDS-Specialized AST

MEGAGUARDS translates the original ASTs for type-stable loops into specialized, strongly-typed ASTs based on the type information inferred during our type stability analysis. Figure 3.4 shows an example of such a translation. The figure shows how MEGAGUARDS converts the generic For node in the GraalPython AST into a specialized For node, which has a LoopInfo child node. The LoopInfo node stores the loop expression, loop bounds, and step size. The information in the LoopInfo node is later used for the bounds check optimization (Section 3.3.4), dependence analysis (Section 3.4.1) and kernel code generation (Section 3.4.2).

Each node in an MEGAGUARDS-specialized AST operates on a specific data type. doubleAssign, for example, can only assign a double floating-point value to a variable. The nodes in the GraalPython AST, on the other hand, are generic and can handle any data type. These GraalPython AST nodes contain type checks and conditional branches. The MEGAGUARDS-specialized nodes do not.

MEGAGUARDS supports translation of AST operations that operate on generic boxed data types. The assign operation in the GraalPython AST, for example, writes to b[i][j]. Variable b is of generic type ListList according to the original context frame generated by GraalPython, but during type stability analysis, MEGAGUARDS augments the context frame with more precise type information by unboxing b into a primitive data structure of
type `double[][]`. Based on feedback from this unboxing pass, MEGAGUARDS establishes that the `=` operation in question must be translated into a write operation of type `double` (i.e., a `doubleAssign` node).

The `GenericAdd` operation has two input types, representing the left and right sides of the add operation. Our type stability analysis recursively finds the dominant type for this operation. Since both sides are of type `double`, MEGAGUARDS can translate this node into a `doubleAdd`.

The MEGAGUARDS-specialized AST is considered to be type-stable and, thus, does not contain any traditional type guards. Once it is translated to OpenCL or CUDA code or a guard-less specialized Truffle AST, the loop will only need to handle bounds checks and arithmetic overflows, resulting in code with significantly fewer conditional branches.

### 3.3.4 Bounds Check Optimization

Dynamically-typed languages must perform a bounds check for every array access. MEGAGUARDS optimizes this bounds check for arrays whose subscripts are affine expressions. The form of an affine expression is $\alpha x + \beta$ where $x$ is a loop induction variable, and $\alpha$ and $\beta$ are loop-invariant values. Array subscripts in this form allow us to safely determine the upper and lower bounds for all array accesses before executing the loop. As with guards, MEGAGUARDS removes bounds checks from the loop body and inserts only checks for the upper and lower bounds ahead of the loop.

For array subscripts that are non-affine expressions, MEGAGUARDS cannot validate the bounds ahead of time. Instead, we convert the existing run-time bounds check into a simple check that sets a flag whenever an out-of-bounds violation occurs.
Figure 3.5: MEGAGUARDS bounds check optimization process.

Figure 3.5 shows a program containing both kinds of arrays. The array subscripts for lists \( b \) and \( c \) are affine expressions, so MEGAGUARDS hoists the bounds checks out of the loop. The array subscript for list \( a \), however, is a non-affine expression, so MEGAGUARDS still requires a run-time bounds check inside the loop. The run-time bounds check compares the value of the evaluated non-affine expression with the size of the data structure that we collected in the unboxing pass (Section 3.3). The check sets a \texttt{boundsViolated} flag if it detects a violation. MEGAGUARDS reads the value of this when the execution of the offloaded loop finishes. Access to the \texttt{boundsViolated} flag does not have to be thread-safe, as threads will only write to the flag if they detect a violation, and any thread that detects a violation will write...
the same value. If the flag has been set when the loop execution finishes, MEGAGUARDS discards the results of the loop execution and re-executes the loop in the interpreter instead. If the flag has not been set, MEGAGUARDS transfers the results of the loop execution to the host memory.

The size information we generate during the unboxing pass (see Section 3.3.1) also allows us to optimize and perform the bounds checks for multi-dimensional arrays. MEGAGUARDS uses this size information to ensure that all dimensions in a multi-dimensional structure contain the same number of elements. If we detect a multi-dimensional array with different-sized dimensions (e.g., if the first inner list of a two-dimensional array contains 5 elements and the second inner list contains 6 elements), our bounds check pass will not be able to guarantee the safety of all array accesses and it will not optimize the loop.

Similarly, MEGAGUARDS optimizes overflow checks for arithmetic operations. Based on the data collected during the unboxing pass, MEGAGUARDS can determine the upper and lower bounds of the arithmetic operations in the loop if the operations are in the form of affine expressions. If that is the case, MEGAGUARDS hoists overflow checks out of the loop. If MEGAGUARDS cannot verify the safety of the operations before executing the loop, it performs run-time overflow checks that set an overflowOccurred flag if an overflow occurs.

3.3.5 Mega Guards Insertion

Finally, MEGAGUARDS creates a mega guard for the MEGAGUARDS-specialized AST. The megaguard() in Figure 3.4 shows an example mega guard inserted above the specialized loop. This guard verifies that the effective run-time type of each variable matches the types in the MEGAGUARDS-specialized AST. If MEGAGUARDS detects a mismatch, it invalidates the specialized AST and rebuilds it based on the effective types. The mega guard also performs bounds and overflow checks hoisted out of the loop.
3.4 Parallel Analysis and Execution

After creating a specialized AST, MEGAGUARDS tests if the loop is eligible to be an OpenCL or CUDA kernel as shown in Figure 3.6. To guarantee the independence of loop iterations, MEGAGUARDS performs cross-iteration dependence analysis by leveraging a polyhedral model (Section 3.4.1).

3.4.1 Dependence Analysis

MEGAGUARDS runs a dependence analysis to verify that no flow (i.e., read after write), anti (i.e., write after read), or output (i.e., write after write) dependencies exist between the
different iterations of the loop. MEGAGUARDS does not offload any loops having such cross-iteration dependencies. MEGAGUARDS performs a lightweight dynamic tracking analysis that tracks scalars write and read locations to detect simple loop-carried data dependencies. Moreover, it incorporates a polyhedral dependence analysis using the Integer Set Library (ISL) and the Polyhedral Extraction Tool (PET) [98, 99], which provides a polyhedral compilation API.

On top of the dependence analysis, we also perform alias analysis to ensure that references between different data structures are completely separate. This alias analysis pass is necessary since the polyhedral analysis incorrectly treats aliases as references to separate memory locations, and might, consequently, fail to identify certain loop dependencies. Our alias analysis scans through data structure references to verify that each data structure does indeed point to a separate memory location. If we do detect aliases within the same loop, then we refrain from offloading that loop.

Figure 3.7 shows how we feed the MEGAGUARDS-specialized AST, generated from the code in Figure 2.1, to the polyhedral dependence analysis. MEGAGUARDS is able to verify that no cross-iteration dependencies exist in either of the loops and can therefore safely optimize
both loops.

MEGAGUARDS supports scalar privatization for temporary scalar variables that are not referenced outside the loop [14]. This eliminates loop-carried output dependencies resulting from the temporary scalar variables and, as a result, increases the number of offloading candidates.

3.4.2 Kernel Code Generation

Once it has fully analyzed a loop, MEGAGUARDS adds the necessary run-time bounds checks to the loop’s AST (see Section 3.3.4) and then translates the AST into OpenCL or CUDA code. MEGAGUARDS then compiles the code into a binary kernel and stores this kernel in a cache. Keeping this cache allows us to skip analysis and code generation and compilation.

3.4.3 Thread Mapping

MEGAGUARDS leverages OpenCL’s and CUDA’s multi-dimensional thread range capability, called NDRange and grid, respectively, to maximize the thread-level parallelism (TLP) for kernels with nested loops. Where possible, MEGAGUARDS attempts to parallelize entire nested loops. Our thread mapping scheme is compatible with existing concurrency schemes [52, 53].
Figure 3.8: MEGAGUARDS thread mapping and code generation.

MEGAGUARDS’s thread mapping follows an outer-loop-first policy to maximize the parallelized region and, at the same time, minimize the number of kernel invocations. MEGAGUARDS currently only supports thread mapping of perfectly nested loops. We leave support for imperfectly nested loops as future work.
NDRange allows us to specify the number of threads we want to create on each computing device. We map each thread to an N-dimensional index space. As the latest version of OpenCL and CUDA support up to three dimensions, MEGAGUARDS can map nested loops with up to three nesting levels to SIMT threads.

Figure 3.8 illustrates MEGAGUARDS’s thread mapping pass. MEGAGUARDS takes the list of independent loops produced by our dependence analysis as input (see Section 3.4.1), and searches for a perfectly nested form of loops starting from the outer-most independent loop. We repeat this process until we get a maximum of 3-D ranges. MEGAGUARDS’s thread mapping follows the outer-loop-first policy in order to maximize the parallelized region and, at the same time, minimize the number of kernel invocations.

MEGAGUARDS converts for loops into an OpenCL or CUDA kernel based on the SIMT programming model by rewriting the specialized AST into a kernel AST, as shown in Figure 3.8. In this step, an iteration vector of nested for loops is mapped to a unique thread ID given to each SIMT thread. For example, an iteration vector of a 2-level nested loop, \( (i, j) \), is mapped to a unique thread ID represented as a 2-D array value which can be accessed by the \texttt{getGlobalId(dim)} node. Then, the AST of a loop body is mapped to a kernel body and the For nodes are removed. In this example, \( n \times m \) threads are created according to the iteration space range of the nested for loops, \( (n, m) \). Instead of iterating loops with induction variables, the kernel body will be concurrently executed by the SIMT threads with their unique IDs.

### 3.4.4 Kernel Data Management

Before the execution of an offloaded loop can start, we need to make sure that all the data the loop accesses is present on the OpenCL or CUDA device. This means that MEGAGUARDS might have to copy data structures from the main memory to the OpenCL or CUDA device.
To avoid redundant copy operations, MEGAGUARDS manages a cache of data that is present on each OpenCL or CUDA device. MEGAGUARDS does not copy any data that is already present on the device, unless the data is marked as invalid in the cache. This kernel data management (KDM) optimization allows kernels to share common data. MEGAGUARDS automatically inserts the code that marks cache entries as invalid during the unboxing pass, when the associated data is modified.

MEGAGUARDS also optimizes map operations that write their results to a list they never read from. Instead of copying an empty result list before we offload a map operation, MEGAGUARDS simply allocates that list on the device but does not initialize it. MEGAGUARDS only copies the list from the OpenCL or CUDA device to the main memory when the offloaded kernel finishes its execution.

### 3.4.5 Kernel Execution and Device Selection

MEGAGUARDS proceeds to the kernel execution stage as soon as the interpreter reports the loop offset. For non-zero loop offsets, we only offload the remaining iterations of the loop.

MEGAGUARDS can execute kernels on a specific acceleration device or select the best device for each kernel adaptively. With adaptive device selection enabled, we compile kernels for each available acceleration device and cache the compiled kernels, one for each device. Then, we pick an accelerator to execute the kernel on and we store the total run-time of the loop. We configured MEGAGUARDS to always try a CPU device when a loop executes for the first time. After multiple kernel invocations, sufficient performance data will be available to select the fastest device for that kernel. A device is selected if it is faster than the others, and if the kernel has executed at least once on every accelerator. This strategy can cause the program to miss out on performance benefits for a few runs, but it quickly pays off when the selection converges. In case of a tie, MEGAGUARDS selects the GPU as the best execution
3.4.6 Execution of A Cached Kernel

Future executions of a kernel can use the cached kernel code if the following conditions are met:

- **the mega guard check passes**: The mega guard check reads the loop’s context frame, unboxes all variables in the frame, and compares their types with the cached copy of the kernel’s augmented context frame (see Section 3.3).

- **the loop body does not contain aliases**: We conservatively perform alias analysis to make sure that the program has not introduced new aliases since we performed the original translation of the loop.

- **the hoisted bounds and overflow checks are still valid**: We re-run part of the bounds and integer overflow check optimization pass to ensure that no new bounds checks or overflow checks are required.

If all three conditions hold, we offload the cached copy of the kernel code to the acceleration device. If not, we re-run the complete analysis and generate a new, specialized kernel.

3.5 Guards-optimized Sequential Execution

**MEGAGUARDS** translates the MEGAGUARDS-specialized AST to a guards-optimized Truffle AST if any of the parallel loop analysis fails (see Section 3.4). The resulting Truffle AST will not have any type checks but may still have bounds and overflow checks that MEGAGUARDS
is unable to optimize (see Section 3.3.4). In order to preserve the integrity of data, MEGAGUARDS backs up the modifiable data structures and restores them if a bounds violation or an overflow occurs. MEGAGUARDS then executes this guards-optimized Truffle AST directly on top of the Truffle/Graal stack.

If the sequential Truffle AST contains any nested loop that can be parallelized, MEGAGUARDS offloads the nested loop(s) and optimizes the data transfers within the sequential execution scope.

3.6 Implementation Capabilities and Limitations

3.6.1 Recursion

MEGAGUARDS constructs a call graph of function calls in the loops to verify that no recursion exists in any of the loops that can potentially be offloaded. If MEGAGUARDS does detect recursion in an offloading candidate, then it will execute the loop using the guards-optimized sequential execution instead.

3.6.2 Built-in Functions

MEGAGUARDS supports reduction throughout Python’s built-in reduce function and the embarrassingly parallel map operator. MEGAGUARDS specializes map’s and reduce’s apply functions based on the lists that are passed iteratively to the function.

MEGAGUARDS also supports many of Python’s built-in math functions (e.g., max, sqrt, cos, ...). MEGAGUARDS translates calls to such functions into calls to their respective counterparts in the OpenCL and CUDA frameworks, and then specializes the translated
calls based on the call arguments.

### 3.6.3 Non-local Control Constructs

Currently, MEGAGUARDS does not support language features that cause a non-local control flow, such as exceptions and generator expressions, i.e., suspend/resume. Our lightweight pre-assessment (see Section 3.2) checks if such a non-local control construct exists in the loop and if so, it falls back before proceeding to optimize guards.

### 3.6.4 Loop Transformations

MEGAGUARDS supports scalar variable loop privatization to increase the number of parallelizable loops. Other loop transformation techniques such as array variable loop privatization [95], loop splitting and loop peeling [30] could further enhance the parallelism if applied to MEGAGUARDS. Loop peeling, for example, splits any first or last few problematic iterations from the loop such that the remaining iterations are no longer dependent on each other. As a future work, MEGAGUARDS can incorporate such loop transformations and parallelize the transformed loops that become free of a loop-carried dependence.

### 3.7 Evaluation

### 3.8 Experimental Setup

We ran our benchmarks on the following system:

**CPU**: Intel Core i7-6700K @ 4 GHz Quad-Core CPU with Hyper-Threading representing 8
compute units (CU). 64GB of RAM. Turbo Boost disabled.

**GPU**: NVIDIA GeForce GTX 1080 Ti with 11GB of RAM and 3584 Stream Processors.

**OS**: Ubuntu x86 64-Bit 16.04.2 LTS using Linux kernel 4.4.0-140. GNU GCC 5.5.0, Oracle labsjdk1.8.0u192-jvmci-0.54 and GraalVM (github revision 08cd504).

We compared the performance of MEGAGUARDS with:

Python Systems:

- **CPython** version 3.5.2: The standard Python 3 interpreter.
- **PyPy 3** version 6.0.0 [10]: Python 3 implementation, uses a meta-tracing JIT compiler to compile Python code into machine code for CPU.
- **GraalPython** (github revision c90a23b) [56]: Python 3 implementation targeting Graal, uses the Truffle framework to JIT-compile specialized AST nodes into x86 machine code.

Heterogeneous Computing Frameworks:

- **OpenCL C/C++ (CPU)** Intel driver version 1.2.0.25
- **OpenCL C/C++ (GPU)** NVIDIA driver version 410.79

We ran each benchmark three times on each system and calculated the geometric mean of the execution times. We measured the execution times including data transfers from the CPU memory to the accelerator device memory and vise versa.

We ran the pure Python implementations of each benchmark to measure the MEGAGUARDS, GraalPython, PyPy and CPython performance. To properly measure peak performance, we warmed up the benchmarks to allow GraalPython and PyPy to just-in-time compile the Python code.
We compared five different backend/device selection configurations for MEGAGUARDS:

- **MegaGuards-Truffle**: running GraalPython sequentially on the CPU with our guards optimization enabled.
- **MegaGuards-CPU**: offloading to CPU OpenCL devices only.
- **MegaGuards-GPU**: offloading to GPU OpenCL devices only.
- **MegaGuards-CUDA**: offloading to GPU CUDA devices only.
- **MegaGuards-Adaptive**: using our adaptive device selection we discussed in Section 3.4.5.

We carefully chose program inputs that are representative of large, real-world data sets and simulations.

**Benchmark Selection**

We ported a set of benchmarks from the Rodinia benchmark suite [19, 20] to pure Python, using only Python built-in data types. We complemented this extensive set of benchmarks with the ones from the Numba Benchmark Suite [21] and the NVIDIA OpenCL SDK [71].

The selected benchmark programs (listed in Table 3.2) have two implementations: (i) pure Python, and (ii) a native hand-optimized version for OpenCL C/C++. The following benchmark programs have two implementations: (i) pure Python, and (ii) a native hand optimized version for OpenCL C/C++.

- **backprop**: is a machine-learning algorithm implementation [65] that trains the weights of connecting nodes on a multi-layered neural network...
• **bfs**: is a Breadth-First Search (BFS) algorithm implementation [86] to traverse all the connected nodes in a graph.

• **black-scholes**: This program calculates financial option prices using the Black-Scholes partial differential equations.

• **euler3d**: Euler3D is a CFD solver for the 3-D Euler equations [22].

• **hotspot**: HotSpot a 2-D temperature and power simulator for architectural floor plans [47].

• **hotspot3D**: HotSpot3D is a 3-D version of the hotspot benchmark.

• **lavaMD**: LavaMD is an implementation for classical molecular dynamics (MD) that calculates the particle potential and relocation within a large 3-D space [91, 93].

• **lud**: is an LU Decomposition algorithm implementation to calculate the solutions of a set of linear equations.

• **mandelbrot**: computes memberships for a Mandelbrot Set, a set of complex numbers, given a fixed number of iterations.

• **mm**: is a naive matrix multiplication implementation.

• **nbody**: N-body simulation widely used in Physics. We only compute the particle accelerations part for the purpose of this work.

• **nn**: is a Nearest Neighbor (NN) algorithm implementation which calculates the Euclidean distance from the destination longitude and latitude among $k$ nearest neighbors...

• **particlefilter**: Particle Filter is a statistical estimator used in tracking systems [35].

• **pathfinder**: is dynamic programming implementation for find a path on a 2-D grid on bottom to top row formation based on the smallest accumulated weights...
- **srad**: Speckle Reducing Anisotropic Diffusion (SRAD) is a derived partial differential equation (PDE) for diffusion method that is part of ultrasonic and radar imaging applications [92, 108].

We excluded **bfs** benchmarks from the table as the existing polyhedral analysis could not disprove dependence (see Section 3.4.1) and thus the benchmark only ran using MEGA-GUARDS-Truffle backend.
3.9 Effect of Guards Optimization

Figure 3.9: Sequential execution speedup of MEGAGUARDS-Truffle compared to CPython, PyPy, GraalPython normalized to GraalPython.
In Figure 3.9, we show the performance impact of our guards optimization by measuring the sequential performance of GraalPython with the guards optimization enabled (MegaGuards-­‐Truffle). The performance is normalized to the baseline GraalPython on a logarithmic scale. The last set of bars represents the geometric mean performance of each system. Standard errors are also marked in the figure. Table 3.1 shows the execution time for each benchmark (in seconds). We measured the performance with the largest data sizes the Rodinia benchmark suite provides.

Our guards optimization improves the sequential Python performance by up to 94.67×, 4.56×, and 16.40× compared to CPython, PyPy and GraalPython. On average, we achieve a performance improvement of 37.25×, 1.63× and 1.54× over CPython, PyPy and GraalPython. particlefilter shows the most substantial performance improvement (16.40× over GraalPython) because our guards optimization removes most of its overflow checks (see Section 3.3.4).

### Table 3.1: Sequential execution time (in seconds) for MegaGuards-­‐Truffle, CPython, PyPy and GraalPython.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GraalPython</th>
<th>MegaGuards-­‐Truffle</th>
<th>PyPy</th>
<th>CPython</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>0.395</td>
<td>0.401</td>
<td>0.388</td>
<td>9.749</td>
</tr>
<tr>
<td>bfs</td>
<td>2.243</td>
<td>2.231</td>
<td>3.145</td>
<td>25.212</td>
</tr>
<tr>
<td>blackscholes</td>
<td>163.911</td>
<td>131.224</td>
<td>110.574</td>
<td>1417.254</td>
</tr>
<tr>
<td>euler3d</td>
<td>4.676</td>
<td>2.496</td>
<td>3.436</td>
<td>236.314</td>
</tr>
<tr>
<td>hotspot</td>
<td>7.469</td>
<td>5.052</td>
<td>12.885</td>
<td>331.775</td>
</tr>
<tr>
<td>hotspot3D</td>
<td>2.011</td>
<td>2.066</td>
<td>4.809</td>
<td>168.776</td>
</tr>
<tr>
<td>lavaMD</td>
<td>15.048</td>
<td>12.142</td>
<td>11.464</td>
<td>477.417</td>
</tr>
<tr>
<td>lud</td>
<td>277.947</td>
<td>165.18</td>
<td>214.239</td>
<td>4860.023</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>10.667</td>
<td>8.646</td>
<td>21.948</td>
<td>173.757</td>
</tr>
<tr>
<td>mm</td>
<td>99.831</td>
<td>67.408</td>
<td>104.128</td>
<td>2641.379</td>
</tr>
<tr>
<td>nbody</td>
<td>286.941</td>
<td>284.992</td>
<td>284.383</td>
<td>2542.633</td>
</tr>
<tr>
<td>nn</td>
<td>0.312</td>
<td>0.142</td>
<td>0.338</td>
<td>12.151</td>
</tr>
<tr>
<td>particlefilter</td>
<td>11.506</td>
<td>0.702</td>
<td>3.202</td>
<td>51.301</td>
</tr>
<tr>
<td>pathfinder</td>
<td>0.507</td>
<td>0.3</td>
<td>0.542</td>
<td>21.773</td>
</tr>
<tr>
<td>srad</td>
<td>10.023</td>
<td>11.947</td>
<td>20.723</td>
<td>793.706</td>
</tr>
</tbody>
</table>
3.10 Parallel Execution Performance and Complexity

### 3.10.1 Characteristics of Kernels

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Loops</th>
<th>Kernels</th>
<th>Thread Count</th>
<th>MegaGuards-Adaptive</th>
<th>LOC</th>
<th>McCabe Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>8.4 x 10^8</td>
<td>0</td>
<td>1</td>
<td>51</td>
<td>203</td>
<td>13</td>
</tr>
<tr>
<td>hotspot</td>
<td>1.0 x 10^9</td>
<td>0</td>
<td>1</td>
<td>146</td>
<td>288</td>
<td>32</td>
</tr>
<tr>
<td>lud</td>
<td>1.0 x 10^7</td>
<td>0</td>
<td>2</td>
<td>55</td>
<td>391</td>
<td>44</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>6.7 x 10^7</td>
<td>0</td>
<td>2</td>
<td>82</td>
<td>456</td>
<td>24</td>
</tr>
<tr>
<td>mm</td>
<td>8.4 x 10^6</td>
<td>0</td>
<td>2</td>
<td>29</td>
<td>230</td>
<td>15</td>
</tr>
<tr>
<td>nbody</td>
<td>6.6 x 10^4</td>
<td>0</td>
<td>1</td>
<td>34</td>
<td>192</td>
<td>10</td>
</tr>
<tr>
<td>nn</td>
<td>1.6 x 10^7</td>
<td>0</td>
<td>1</td>
<td>82</td>
<td>456</td>
<td>24</td>
</tr>
<tr>
<td>pathfinder</td>
<td>3.0 x 10^7</td>
<td>0</td>
<td>1</td>
<td>63</td>
<td>258</td>
<td>22</td>
</tr>
<tr>
<td>particlefilter</td>
<td>1.2 x 10^6</td>
<td>0</td>
<td>10</td>
<td>255</td>
<td>719</td>
<td>101</td>
</tr>
<tr>
<td>srad</td>
<td>2.1 x 10^8</td>
<td>0</td>
<td>2</td>
<td>89</td>
<td>477</td>
<td>27</td>
</tr>
<tr>
<td>Median</td>
<td>3.0 x 10^7</td>
<td>0</td>
<td>1</td>
<td>59</td>
<td>273</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 3.2: Benchmark characteristics for parallel execution.

Table 3.3: Time (in milliseconds) for each pass of the parallel execution.

Table 3.2 shows the following characteristics for each benchmark:

- **Offloaded**: Number of offloaded loops.
- **Gen.**: Number of generated kernels.
- **Exec.**: Number of kernels’ executions.
• **Loops**: the number of executed and the number of offloaded loops. Nested loops are counted separately.

• **Kernels**: the number of generated kernels and the number of kernel invocations for a single run.

• **Thread Count**: the total number of parallel executions of the kernel(s) body for a single run.

• **MegaGuards-Adaptive**: the final acceleration device selection on the generated kernels using our adaptive selection technique (see Section 3.4.5).

• **LOC**: the lines-of-code counts for the Python and OpenCL C/C++ implementations of the benchmark’s source code.

• **McCabe Cyclomatic Complexity**: the Cyclomatic Complexity [63] of the Python and OpenCL C/C++ implementations of the benchmark’s source code.

Our analyses of the benchmarks’ source code, i.e., LOC and McCabe Cyclomatic Complexity, show that the plain Python implementations of the benchmarks are significantly less complex than the OpenCL implementations.
3.10.2 Parallel Execution Performance

Figure 3.10: Parallel execution speedup of MEGAGUARDS compared to OpenCL C/C++ (CPU and GPU) normalized to GraalPython on a log_{10} scale.
Table 3.4: Parallel execution time (in seconds) for MEGAGUARDS, OpenCL C/C++ (CPU and GPU), and sequential GraalPython.

Figure 3.10 shows MEGAGUARDS’s speedups normalized to GraalPython on a logarithmic scale. The last set of bars represents the geometric mean performance of each system. We measured the performance with the largest data sizes the Rodinia benchmark suite provides. We also marked standard errors in the graph but the errors are too small to be seen except in the MEGAGUARDS-CPU run of particlefilter. Table 3.4 shows the execution time for each benchmark complemented with the sequential execution time of GraalPython.

MEGAGUARDS shows substantial speedups compared to other systems using pure Python benchmarks. For this set of benchmarks, our system performed up to 286.9× faster than GraalPython and 31.79× on average. MEGAGUARDS approaches the performance of native hand-optimized OpenCL C/C++ code (CPU and GPU), being only 2.2× slower on average, without requiring extensive knowledge on heterogeneous computing frameworks. Note that dynamic languages are typically one or two orders of magnitudes slower than C/C++.
Figure 3.11: Breakdown of MEGAGUARDS passes for parallel execution.

Figure 3.11 shows the cost of each analysis pass in MEGAGUARDS during a cold run (*Cold Run*), and when utilizing pre-evaluated (i.e., cached) kernels (*Peak*). Table 3.3 shows the execution time of each analysis pass for each benchmark (in milliseconds). Noticeably in Figure 3.11, our guards optimization and bounds checking stages account for limited overhead due to their inexpensive computations.

In the black-scholes and nbody benchmarks, MEGAGUARDS approaches the performance of the OpenCL C/C++ implementations and is able reduce the number of bounds and overflow checks significantly (see Section 3.3.4). MEGAGUARDS outperformed GraalPython by 286.9× and 114×.

The mm and mandelbrot benchmarks had minimal data transfer rates, and have 2-level nested loops that MEGAGUARDS assigned to a 2-dimensional thread range (see Section 3.4.3). This resulted in large speedups, especially when executing on GPU acceleration devices. In the mm benchmark, MEGAGUARDS created two specialized kernels for the same loop, one for double
floating point-typed variables and one for long integer-typed variables. NVIDIA’s optimized OpenCL implementation of \texttt{mm} outperformed \textsf{MEGAGUARDS} by 10.19×. The reason is that this hand-optimized OpenCL implementation aggressively exploits data locality between local threads. Plus, unlike Python, the OpenCL code does not include safety checks for detecting out-of-bounds array accesses and arithmetic overflows because in a static language like OpenCL writing a safe code is user’s responsibility. \textsf{MEGAGUARDS} bounds and overflow checks enforcement value the safety of the kernel operations and guarantees integrity of the result. Nevertheless, the high performance of the \texttt{mm} benchmark on \textsf{MEGAGUARDS} demonstrates the flexibility of our system to adapt to type changes at run time without degrading performance.

The situation in \texttt{particlefilter}, \texttt{srad}, \texttt{hotspot} and \texttt{lud} is similar. \textsf{MEGAGUARDS} generated ten, two, one and two specialized kernels for these benchmarks respectively. Most kernels were assigned to a 1-dimensional thread range in \texttt{particlefilter} and \texttt{lud}, and to 2-dimensional thread ranges in \texttt{srad} and \texttt{hotspot}. In the OpenCL-GPU implementation, the core computation of \texttt{particlefilter}, \texttt{srad}, \texttt{hotspot} and \texttt{lud} features cooperative local threads that share data through a local cache. As a result, OpenCL-GPU outperformed \textsf{MEGAGUARDS} using a GPU by 2.35×, 2.58×, 1.48× and 8.28×, respectively.

Overall, we observed that acceleration-compatible loops experienced speedups by up to an order of magnitude under \textsf{MEGAGUARDS}. 

3.10.3 Peak Performance with Various Input Sizes

Figure 3.12: Peak performance of MEGAGUARDS with different data sizes for parallel execution.
To show the scalability of MEGAGUARDS, we measured the peak performance with different data sizes. The results are normalized to the sequential GraalPython implementation. As shown in Figure 3.12, MEGAGUARDS yields performance gains relative to the size of the inputs. MEGAGUARDS-Adaptive follows the best device performance curves with the varying input sizes and relieves the user from manually setting a specific accelerator device.

### 3.10.4 Performance of MegaGuards on Each Run Step

So far, we only measured the peak performance of GraalPython. We gave GraalPython’s underlying Truffle/Graal stack and PyPy a couple of warm up runs to specialize, optimize and compile every hot path in GraalPython’s AST into x86 machine code. MEGAGUARDS, by contrast, specializes, optimizes and compiles the program’s hottest paths (i.e., loops) immediately. MEGAGUARDS therefore brings even greater performance benefits to end users who do not warm up the program interpreter.

![Graph showing the performance of MegaGuards](image)

Figure 3.13: Mean speedup of warming up steps of MEGAGUARDS normalized runs for GraalPython.
These performance benefits are illustrated in Figure 3.13. In this figure, we see the mean performance of each run of our benchmarks. In the first run, GraalPython is executing the benchmarks in the interpreter. Through the second to the sixth runs, the JIT compiler specializes and optimizes the code. After that, GraalPython reaches a steady state. By contrast, MEGAGUARDS specializes the loop from the first run, thanks to our type stability analysis before executing a loop. This leads to large performance benefits instantly. During the run steps, our MEGAGUARDS-Adaptive execution mode explores acceleration devices and benchmarks their performance at run time until it settles on the best device.

3.10.5 Scalability

![Figure 3.14: Mean speedup of warming up steps of MEGAGUARDS-CPU with various CU counts normalized runs for GraalPython.](image)

Figure 3.14 shows how MEGAGUARDS scales with the number of CUs. We scaled the number of available CUs for the CPU OpenCL device and measured the mean performance of each run of our benchmarks. MEGAGUARDS shows performance gains as we increase the number
of CUs.

We see the down curve in the middle of the performance graph. This is because the CPU is shared among other processes such as the Graal compiler’s background analysis. This also explains MEGAGUARDS-CPU underperforms compared to OpenCL-CPU C/C++ in Figure 3.10.

### 3.10.6 Effect of Kernel Data Management

![Figure 3.15: Effect of KDM optimization on MEGAGUARDS.](image)

In Figure 3.15, we show the effect of our kernel data management optimization on peak performance. This optimization yields an average performance gain of $4.46 \times$. The KDM optimization provides the larger performance gains for the benchmarks with the higher data transfer rates.
Chapter 4

Redizer: A Transparent Parallel Reductions Transformation

Python is one of the most popular languages in data science and machine learning because of its high productivity and lean syntax. Many algorithms in this domain contain heavy computational loops with reduction operations that pose a real performance bottleneck. Contrary to simple data-independent loops, it is a much harder challenge to accelerate such code: Programmers must decompose and redesign the algorithms to fit to specific library calls or constructs that enable GPU accelerations. For many algorithms, this is a truly complicated work in contradiction to the initial idea of Python’s productivity and convenience. We can therefore assume that many programmers spare the effort and rather leave the acceleration potential unexploited.

In the native GPU programming languages such as OpenCL and CUDA C/C++, on the other hand, the issue is performance portability. To achieve the best performance, the programmers often hand optimize their OpenCL/CUDA code tailored to a specific GPU hardware. Unfortunately, such platform-specific optimizations often do not work for different
platforms and thus in the worst case the programmers must rewrite their optimizations for every single platform.

In this Chapter, we present a new method that achieves both high programmer productivity through the use of lean syntax Python and complete portability across different hardware platforms by dynamically optimizing parallel reduction loops specialized for the underlying GPU hardware. Programmers can write plain sequential Python loops even with complex nesting and reduction dependencies, without the need for any GPU libraries or constructs. Our JIT-optimization engine will transparently generate the code optimized for the underlying GPU hardware with an efficiency close to the native GPU implementation. Our evaluation also confirms that reduction acceleration indeed constitutes a significant hidden potential beyond loop-independent parallelization for many compute-intense Python applications that our method can unleash without extra costs for the programmer.

## 4.1 Background and Motivation

Python is one of the most popular languages in data science and machine learning. The language’s flexibility and lean syntax allow the scientists to solely focus on high-level algorithms, rather than investing time to learn some complicated language features. There is a conflict, however, since many scientific applications and machine learning algorithms often include heavy computational loops which need acceleration on GPUs; and in order to enable this acceleration, the scientists need to learn how to use GPU-enabled libraries or language constructs. Figure 4.1 shows how a simple Python loop should be rewritten to exploit general GPU-enabled reduction APIs.  

\[1\]

In general, even if such a manual GPU parallelization would yield a significant acceleration,

\footnote{Note that for this simple illustrative case, a single matrix multiplication API call could be used, but there is no such perfect API function match for many other algorithmic pieces in the real world.}
Figure 4.1: Python matrix multiplication implementation using simply written loop versus using reduction APIs.

programmers often do not want to take the extra effort, leaving a lot of acceleration potential unexploited.

One viable option to resolve this conflict and retain the level of productivity provided by the language is to use a tool that automatically accelerates such computational loops. This option, however, is often considered less practical, limited, and/or much less performant than using libraries and language constructs. There have been previous efforts to automatically accelerate loops or language idioms for data science languages like Python and R [80, 62, 73, 60, 96, 31]. In these systems, there is still a significant performance gap compared to using GPU-enabled libraries and compiler directives, and directly using hand-optimized OpenCL/CUDA code.

One might think that this performance gap is something fundamental, e.g., due to the imprecision of dependency analysis and the use of dynamic languages. Historically, dynamic languages are often considered to be less efficient at the cost of its high productivity. On the contrary, however, we observed that a primary reason for this performance gap comes from the lack of generality of the analysis, mainly because it does not cover reduction patterns. Many algorithmic loops are not fully independent but engage a certain though limited dependency that essentially is a reduction. Therefore, including reduction dependencies in loop parallelization is what we expect to be an substantial step towards a convergence of automatic and manual parallelization.
This dissertation takes this path of advanced automatic loop parallelization: We present a new method that automatically and dynamically parallelizes loops with possibly complex reduction dependencies on GPUs. It accelerates plain Python nested loops and loop sequences with intra- and inter-loop reduction dependencies of various aggregator operators, without annotation or help from the programmer.

Although several previous work has addressed reduction dependency analysis in the context of static compilers [50, 51, 75, 77, 81, 83, 97, 107, 29, 84, 85, 25], there is no such solution yet for dynamic languages, mainly due to the dynamic nature and time constraint of a just-in-time (JIT) compiler. We therefore carefully designed our just-in-time analysis to minimize the analysis costs by actively leveraging the information available from the language’s runtime and the previous analysis steps, and we refine loop parallelization techniques to a blend that is both fast and effective in practice. While JIT-optimization imposes a stricter time limit on the analysis compared to ahead-of-time compilers, it offers at the same time much more aggressive optimization possibilities, since the runtime engine knows the dynamic constellation and can selectively optimize for specific runtime cases that profit most from GPU-acceleration.

The JIT-engine of this system infers detailed runtime information for types, loop dependencies and ranges. Moreover, MEGAGUARDS already features automatic loop parallelization, however, only for data-independent loops. We extended this system to incorporate reduction-aware loop analysis and transformation to corresponding CUDA and OpenCL parallel code. Enabling reduction awareness for loop parallelization indeed offers a significant performance improvement. REDIZER offers an extra speedup of $3 \times$ to MEGAGUARDS that already provides very effective data-independent loop parallelization. Together with MEGAGUARDS, REDIZER achieves 67% of the hand-written OpenCL performance on average. Moreover, the results show that our method achieves the throughput on par with manual approaches and outperforms native API solutions in some cases.
4.2 Overview

We present an automatic reduction parallelization method for Python, which provides flexible and comprehensive support for parallel reductions. For example, our method can parallelize irregular reduction patterns like the following:

```python
for i in range(n):
    likelihood, countOnes = 0.0, 0
    for x in range(diameter):
        for y in range(diameter):
            if (distance[x][y] < radius):
                countOnes += 1
                likelihood += v1[x][y] - v2[x][y]
    weights[i] *= exp(likelihood / countOnes)
```

The example code is a simplified version of a reduction pattern found in the particlefilter’s algorithm. The algorithm has imperfectly nested loops which contain multiple reduction variables under the if statement. To express the same reduction algorithm using reduction libraries that enable GPU accelerations (in the similar way as shown in Figure 4.1), the programmer must manually rewrite the code to have separate loops for each reduction operation, and adjust the loop ranges according to the if statement. Then, the programmer should convert each reduction operation into a separate reduction library call. The resultant program will have multiple separate kernel invocations per reduction operation enclosed by a sequential outer loop (i.e., the loop at line 1). With our reduction identification and transformations, these imperfectly nested loops with multiple reduction operations can be automatically transformed into a single kernel invocation.

Figure 4.2 shows an overview of our design, REDIZER and its interaction with MEGAGUARDS. We take a loop as type-specialized AST, identifies reduction operations in it, and determines if the loop can be transformed into a parallel reduction (Section 4.3). If so, we first normalize and flatten the loop AST to simplify the subsequent transformation process (Section 4.4), and then it transforms the AST into our parallel reduction AST. Our method has different AST transformation and thread mapping strategies for loops that are nested within parallel loops.
(Section 4.5.1) and for loops that are not (Section 4.5.2), to optimize reduction workload distribution for each of these cases. After the AST reduction transformation, we add hint nodes to guide the code generation backend (Section 4.6). The hint nodes specify which data should be allocated in the shared or the global memory and where to put thread synchronization points.

### 4.3 Reduction Identification

In the reduction identification process, we (i) decide whether a loop with cross-iteration dependency can be transformed to a parallel reduction loop. Once we identify reductionizable loops, (ii) we identify the depth of the reduction loop, and (iii) detect reduction variables and their associated reduction operators. This information is later used in the reduction transformation stage.

We apply our analysis on non-parallelizable loops once the independent loop analyzer has determined parallelizable and non-parallelizable loops through its loop dependency analysis.

Since the analysis happens at runtime, our goal is to minimize the analysis time. In order to avoid unnecessary redundant computation, our reduction identification process leverages

---

Figure 4.2: REDIZER transformation overview
information produced during the type specialization and loop dependency analysis. This information includes:

- array alias information.
- whether the loop has an explicit range (a fixed number of iterations with a fixed step size).
- whether the loop carries dependency and the type of the dependency.
- a set of variables that causes loop-carried output dependency
- whether it includes any recursive call,
- or has any other incompatible operation with the accelerator device framework (OpenCL, CUDA, etc.).

We consult this information to determine if the dependency of the loop is caused by output dependency. This means the loop is potentially in a reduction form. If the loop is marked non-parallelizable because of any other reason, the reduction identification process decides early not to perform parallel reduction on this loop without further analysis.

We add all the variables that cause loop-carried output dependency to a list of must reduce variables. The must reduce list will be used to assist the verification process to assert that all output dependencies of the loop are identified as reduction variables.

If a loop has not been parallelized and the reason is solely due to output dependency, our method checks whether the loop forms a reduction pattern. To do so, it traverses the AST in depth-first order. If a node has multiple child nodes, it traverses the child nodes in the logical execution order. We specially handle assignment nodes; Since every assignment node is potentially a reduction operation, we check the following conditions to determine whether the assignment node may be a reduction:
1. The assignment is in this form: \( \alpha = \beta \ OP \ f \)

2. The assignment is inside a loop body.

3. Both \( \alpha \) and \( \beta \) are the same variable.

4. \( OP \) is a supported binary operator (i.e., \( \min() \), \( \max() \), \( \text{add} \), \( \text{mul} \), \( \text{sub} \), and \( \text{div} \)).

5. \( \alpha \) has not been identified as a reduction variable yet.

6. \( \alpha \) has output dependency across iterations, or has not been privatized yet.

Generally, we do not allow the reduction variable \( \alpha \) to be an array element, since it entails extra array dependency analysis. This is our design choice to keep the cost of reduction analysis as low as possible, while fully reusing information already available from previous steps, i.e., type specialization and parallel loop dependency analysis. The reduction variable \( \alpha \) can be an array access, however, if the target loop is nested within one or more parallel loops and the array subscript does not depend on the induction variable of the target loop. We can do this because this array access should already be privatized during the course of parallelizing the outer loops unless the array subscript depends on the induction variable of the current target loop.

The supported binary operations include all the built-in commutative and associative operations such as add, multiply, \( \min() \) and \( \max() \). Notably, we also permit reductions with \( \text{subtract} \) and \( \text{divide} \): Although both operators are neither commutative nor associative, their partial results are, and we can safely reduce them in parallel (see Section 4.5.1).

When all the conditions are met, we add the variable \( \alpha \) to the reduction variables list, and associate this variable with the operator \( OP \). We then traverse \( f \), the right node of the binary operation (1): This node can be in any form, e.g., simple arithmetic operations, function call, etc. However, reading from any other reduction variable within the same reduction
loop is not allowed as it will induce incorrect calculation. As soon as any of these conditions is violated, we traverse the assignment normally, and privatize the left operand, $\alpha$. If $\alpha$ is in the must reduce list or has already been added in the reduction variable list for another statement, we exit our analysis and deem the loop to be non-parallelizable. Additionally, our reduction identifier will not accept reductions in different levels of nested loops.

Consider a loop AST shown in Figure 4.3 as an example. The reduction identifier recognizes a form of reduction for the assignment $c += a[i] - b[i]$, as variable $c$ is accumulating the subtract result of $a[i] - b[i]$ on each iteration. In this case, the identifier determines that the loop is transformable, and $c$ is a reduction variable with addition as the reduction operator.

**4.4 Loop Flattening**

General accelerator frameworks such as CUDA and OpenCL mandate a total number of threads to be specified before launching parallel computations. Ideally, a single parallelizable loop runs each loop iteration on a single thread such that the number of threads is equal to the number of iterations. In order to partition a loop into threads, we need to know the specific range of the loop. Loop ranges in Python are specified using the range operator,
range(), which takes a variable number of arguments representing start, stop and the step size of the loop index. To calculate the total number of iterations, we use the following range size equation:

\[
\text{range size} = \left\lceil \frac{\text{stop} - \text{start}}{\text{step}} \right\rceil
\]  

(4.1)

Before transforming a reduction loop into a parallel reduction loop, we perform loop normalization and loop flattening. If the target loop is imperfectly nested, dealing with it for the rest of the transformation steps could be complicated. Thus, we first transform imperfectly nested loops into flat loops to simplify and uniform the rest of the transformation process. During the course of loop flattening, we also normalize loops, such that the loop variables start at zero and get incremented one at every iteration.

As shown in Figure 4.4, loop flattening collapses both nested loops \(L_1\) and \(L_2\) into a single loop \(L\). Consequently, the outer loop body \(B_1\) inlines the nested loop body \(B_2\), adopting the entire AST of \(B_2\) under \(B_1\)'s AST. After that, we insert two new blocks: the total range size block \(S\) and the induction variable recovery block \(I\). The total range size block \(S\) will be executed by the host device before the kernel execution, to provide loop range information as a parameter for the kernel execution. This block calculates the total range sizes for both loops \(L_1\) and \(L_2\) using the equation (4.1). To simplify this loop partitioning process, we normalize the target loop whenever possible, i.e., when the loop variable does not start at zero or has a different increment than one. To calculate the full range of the flattened nested loops \(L_2\) and \(L_1\), we multiply both ranges to get the total number of iterations for the loop \(L\), denoted as \(L_{\text{size}}\).

The induction variable recovery block \(I\) is inserted as the first block of the loop body \(B_1\). The block \(I\) recovers the original induction variables of the loops \(L_1\) and \(L_2\). These recovered induction variables are necessary for the statements that reference the original
After the loop flattening, the loop $L_1$’s child nodes, $E_1$, $P_1$, $P_2$, $E_2$ and $E_3$ constitute the loop body $B$, as shown in 2.

### 4.5 Reduction Transformation and Thread Mapping

Our reduction transformation pass differently handles (i) the loops nested within parallel loops (Section 4.5.1) and (ii) the ones not nested within any parallel loop (Section 4.5.2). We handle them differently because the optimal thread mappings for these loops could
be different. For the loops that are not nested within parallel loops, Redizer schedules
threads in a way that maximizes the parallelism. To this end, we translate such loops into
two separate kernels, one for producing sub-reduction results with multiple work groups and
one for merging and reducing the sub results into the final results, as shown in Figure 4.7.
This thread mapping strategy, however, would not suit for the loops that are already nested
within parallel loops because generating multiple kernels inside nested loops will lead to extra
kernel invocations and synchronization overheads. Instead, we schedule such loops within a
single work group, to be able to synchronize data for a reduction loop without invoking an
extra kernel. We do this by extending the parallel loops with an extra dimension for this
reduction loop, as shown in Figure 4.5.

Our AST-based transformations naturally fuse multiple reductions in a single transformed
loop (see Figure 4.10). This way, we can avoid having separate routines for each reduction
operation.

Our transformations are optimized using a well studied tree-based kernel decomposition
algorithm with a sequential addressing scheme as described by Harris [43]. We opt in for
using the sequential addressing over the interleaved addressing to avoid shared memory bank
conflicts.

4.5.1 Nestedness-Aware Reduction Transformation

Nestedness-aware transformation is a reduction loop transformation for loops that are nested
within parallel loops. As shown in Figure 4.5, an additional range of threads (i.e., helper
threads) will be scheduled as a result of this transformation. How to determine the number
of threads for this inner loop reduction is discussed in Section 4.6.1. Together with the main
threads, the helper threads will produce partial computing results and these results will be
synchronized and merged according to the tree-based reduction. Then, the final results of
the reduction will be merged by the main threads.

After normalizing and flattening a loop, we perform our AST reduction transformation as shown in Figure 4.6. In the first step, we allocate a buffer to store partial results of each reduction variable in the shared memory. Each thread will have a dedicated space within this buffer. This shared buffer allows synchronization of partial calculations between threads.

We then traverse the pre-loop statements to find an initialization statement for each reduction variable. This initialization statement is used in the reduction variable initialization block to initialize the shared buffer spaces dedicated to the main threads. The shared buffer spaces for helper threads will be initialized with zero or another value, depending on the type of the reduction. If a reduction variable has an addition operator, for example, the initialization value will be zero, which is the additive identity for numbers. The initialization strategy for each reduction operator is described in Section 4.7.

If we encounter any global memory writes within the pre-loop statements, we privatize this global memory access by allocating the corresponding buffer in the shared memory. We ensure that only the main thread can update the global memory and can write to this privatized buffer. Consequently, any subsequent reads from the global array will be replaced by the reads from this shared buffer. This way, we ensure that helper threads will not
Figure 4.6: Nestedness-Aware AST reduction transformation.

overwrite the global memory nor read any unsynchronized value from the global memory.

The block \( RR \) is responsible for writing back the final result of the reductions. This block will copy the final result from the shared storage of the main thread to the corresponding reduction variable. We place the block \( RR \) after the loop \( L \) and before the post-loop statements \( E \). Both blocks \( RR \) and \( E \) will be executed only once by the main thread.

After adding the blocks \( RR \) and \( E \), we move on to the next step 2. In this step, we slice the loop \( L \) with the number of threads, i.e., main and helper threads; hence, we modify the step size of the loop \( L \) to be the number of threads. We offset the loop’s starting index by the thread’s id; thread id is assumed to be a unique number in the range \([0, \text{NUMBER OF THREADS})\). Then, we traverse the loop \( L \), and replace each reduction variable with the corresponding shared storage. As a result, each helper thread will do a partial calculation using the same operations in the original loop body with a reduced number of iterations. \( LPC \) and \( BPC \) denote this partitioned version of \( L \) and \( B \), after this transformation. The partial calculation loop implicitly applies the global load optimization [43] to hide memory
Finally, we create a new loop (BR) with the loop body (BR) to perform a tree-based reduction with the sequential addressing scheme. In the loop body (BR), we synchronously merge the results of each reduction variable for each interleaving thread using the associated operators. We specially handle subtract and division since these are neither associative nor commutative. First, each thread normally performs partial reductions with proper initial values as discussed in Section 4.7. We then use inverted operators, i.e., addition for subtract and multiply for division, to merge partial results of the reduction and produce the final result. To see how this inverted operation for partial results produce correct results, consider the following property: \( a_0 - a_1 - a_2 = (a_0 - a_1) + (0 - a_2) \).

### 4.5.2 Two-Stage Reduction Transformation

Figure 4.7 shows how tree-based reduction is performed in two stages. The two-stage tree-based reduction scheme distributes partial reduction operations to multiple work groups. This mechanism allows reduction operations to utilize more parallel threads, compared to distributing the workload within a single work group. However, since there is no way to synchronize data across different work groups, this reduction method requires another kernel...
invocation to merge all the partial results into the final results after all the partial reductions are produced. Because of this additional kernel invocation, we apply our two-stage reduction transformation only for reduction loops that are not nested within any parallel loop. Having multiple kernel invocations for loops nested within parallel loops can cause a substantial overhead.

In the first stage of this reduction scheme, the threads scheduled in the same work group share and synchronize data to produce a partial reduction result. At the end of the tree-based reduction, the main thread will hold the partial reduction result of the group. The main thread will write its partial result in a global shared memory, which all groups share, but without synchronization. In the second stage, threads are scheduled for a single work group to repeat the tree-based reduction process with the partial results stored in the global memory. After finishing the second stage, the final reduction results will be forwarded to the host device. The number of threads in each stage is discussed in Section 4.6.2

**AST Transformation**

Figure 4.8 describes the AST transformation process for the two-stage tree-based reduction. Unlike loops nested within parallel threads, any statements before and after the target loop (e.g., P and E in Figure 4.4) are executed outside the scope of this transformation. Since the loop is not nested within parallel loops, these statements will be executed sequentially on the host device.

In the step 1 in Figure 4.8, we allocate buffers in the shared and the global memory for each reduction variable. Each thread will have its own space within the shared buffer, and each work group will have its own space in the global buffer to store the partial reduction results of the group. The shared buffer allows synchronization of partial calculations among threads within each work group, while the global buffer will be shared across groups without
synchronization. Each group has a main thread which will collect the final results of each reduction variable and store the result into the global buffer.

We retrieve the initial value of each reduction variable from the JIT engine’s context frame. Using this initial value, we initialize the shared buffer area allotted for the main thread in the first work group. All other threads will be initialized based on the strategy described in Section 4.7. We do not initialize the global buffers because in this step, this area will only be overwritten by per-group partial reduction results. We insert the initialization statements for the shared buffers in the initialization block (RI), which we insert before the loop (L). We also insert a new reduction result block (RR) after the loop (L), which only the main thread of each group will execute. This block (RR) is responsible for writing back the main thread’s reduction results to the global buffer.

In the step 2, an additional AST is generated to represent two-stage reductions. For the first stage AST, the partitioned loop (LPC) and the tree-based reduction loop (LR) are generated in the similar way as in the Nestedness-Aware Reduction Transformation (see Section 4.5.1).
In the second stage AST, (R) initializes per-thread shared storage of each reduction variable using the default value based on our strategy in Section 4.7. In the loop (LM), each thread reads partial reduction values from the global buffer and generates intermediate results for each reduction variable. Then, the loop (LR) performs the final tree-based reduction using the intermediate results produced by (LM). The reduction result block (RR) writes the results of the main threads to the global buffer. Finally, the final result for each reduction variable is delivered to the host device.

4.6 Code Generation and Thread Scheduling

After the reduction transformations, the transformed AST is in a form that can be directly translated into a GPU kernel, since the workload distribution and the data placement have been determined. For this translation, we use MEGAGUARDS’ GPU code generation backend which can directly translate a specialized AST into either an OpenCL or CUDA kernel. To communicate with this GPU translation backend, we insert hint nodes. These hint nodes indicate whether data should be allocated in the global or in the shared memory and where to put thread synchronization points. We insert a synchronization point within the reduction loop (LR), and for any global memory write privatized to a shared buffer as discussed in Section 4.5.1.

4.6.1 Nestedness-Aware Reduction Code Generation

In Figure 4.9, we have a Python implementation of the LU decomposition algorithm. The polyhedral dependence analysis has proved that the loops with the induction variable j do not carry data dependencies, while the polyhedral analysis has marked that the loops with the induction variables i and k carry output dependencies. The loop with the induction variable
i will run sequentially as our reduction identifier concluded that the loop is not transformable due to the nested data independent loops with the induction variable j. However, our reduction identifier found that the two loops with the induction variable k are transformable. Since both loops are nested within parallel loops, we perform our nestedness-aware reduction transformation on these loops (see Section 4.5.1). Therefore, the transformed loops will become a two dimensional range of threads rather than one dimensional; thus, the transformed loops are sliced into additional multiple threads. The LU decomposition algorithm features

Figure 4.9: Nestedness-Aware Reduction Code Generation.
reductions with subtract operations. Our transformation replaces subtract operations with additions for the intermediate results merging process. The code generation backend then translates the transformed AST into the target framework, i.e., CUDA in this example.

To provide performance portability, we partition workload depending on the underlying acceleration device specification. We perform a tree-based reduction so the number of threads must converge in base-2 logarithm steps. Accordingly, we calculate the number of local threads for the reduction dimension with the following equation:

\[ 2^\left\lfloor \min\left(\log_2(\text{iterations})\right, \frac{\log_2(\text{MaxWorkGroupSize})}{\text{dimensions}} \right\rfloor \]

We add another hint node for the code generation backend to specify a reduction loop, so the backend can create the adequate number of threads that converges in the base-2 log during its workgroup formation.

In the nestedness-aware reduction, the reduction loop may get a fewer number of threads than the loop iteration count, since we schedule the reduction loop within a single work group and the maximum number of threads is bounded by \textit{MaxWorkGroupSize} of the acceleration device and divided by the number of range \textit{dimensions} to balance ranges’ cardinality between dimensions. More than one iterations are assigned to each thread in this case.

**Adaptive Kernel Selection**

We observed that parallelizing reduction loops with a very small iteration count, i.e., lower than the physical warp/wavefront size of the GPU, can introduce more overhead than the performance gain. To solve this issue, we generate two versions of code when the reduction loop count is not statically known: (1) a kernel with our parallel reduction enabled and (2) a kernel that sequentially executes the nested reduction loop(s). \textit{Redizer} checks the
reduction loop count at run time by consulting the JIT-engine’s context frame and calculates the current value of the reduction loop count. Based on this value, we can decide which version to invoke at run time.

4.6.2 Two-Stage Reduction Code Generation

Since our Two-Stage transformation is two-stage process, we generate two kernels. The first stage kernel will be executed by larger number of threads compare to the subsequent stage kernel. As the first stage will have multiple groups of threads each group will perform partial calculations and merge results. Then, those merge results of each group will stored in a global storage that will be the input of the subsequent stage, which will only do merge results process.
In Figure 4.10, we have a Python implementation of the Srand algorithm. MEGAGUARDS detects output dependencies for both loops with induction variables \(i\), denoted as \(\text{Loop}_i\), and \(j\), denoted as \(\text{Loop}_j\). Since both loops are not within a parallel thread, we perform a two-stage reduction transformation (see Section 4.5.2). Our transformation will flatten...
both loops, $\text{Loop}_i$ and $\text{Loop}_j$, and pass the total number of iterations as a parameter to the generated kernel for the first stage. The first stage kernel will perform the partial reduction calculations using the same logic of $\text{Loop}_i$‘s body. This transformation features two reduction variables $\text{Sum}$ and $\text{Sum2}$. Unlike the commonly used `reduce()` functions in parallel libraries which require multiple reduction function calls to handle this kind of loops, our transformation will fuse these two reduction operations into a single reduction routine.

For the Two-Stage Reduction, our method schedules multiple work groups in the first stage. Consequently, the number of threads assigned for the first stage is not restricted by the maximum workgroup size. Therefore, the number of global threads is calculated using this equation:

$$2^{\lceil \log_2(\text{iterations}) \rceil}$$

In the first stage, some threads might need to do more partial calculations to make up for this floor adjustment. In the second stage, the number of threads is determined based on the first stage’s number of groups. Consequently, the number of threads in the second stage is calculated by this equation:

$$\min(2^{\lceil \log_2(\text{numberOfFirstStageGroups}) \rceil}, \text{MaxWorkGroupSize})$$

This normally fits into one group of threads but it depends on the specification of the acceleration device. If the number of groups is larger than $\text{MaxWorkGroupSize}$, the number of threads for the subsequent stage will be the $\text{MaxWorkGroupSize}$ and each thread will process reductions for multiple iterations. We incorporated this methodology within the code generation backend so it awares our reduction algorithms. We add a hint node to the backend indicating each stage of the reduction. Using this hint, the backend arranges work groups and threads based on the target acceleration device. This will ensure performance portability of REDIZER’s optimizations, while hiding the complexity of manually scheduling workload from the programmers.
4.6.3 Loop Unrolling and Shuffling Instruction

The number of iterations of loop \( L_R \) can be determined ahead-of-execution since the number of iterations is bounded by the maximum number of local threads, i.e., MaxWorkGroupSize. Thus as part of the code generation process, we instruct MEGAGUARDS’ code generator to completely unroll loop \( L_R \) based on the calculated workload upon the execution. However, the workload of kernel executions might differ between runs, so we generate and cache multiple specialized and loop unrolled kernels which will then be automatically selected before executing the kernel.

CUDA framework supports special shuffling instructions \([74]\) that allow threads in the same warp to share local variables without the need for a shared storage. However, a kernel’s workgroup/block size might exceed the warp size, limiting the shuffling opportunity. To overcome this limitation, we introduce a hybrid approach that utilizes both shared storage and shuffling instructions. As we completely unroll \( L_R \), the last five iterations will be executed within a warp. Therefore, the last five iterations will be translated to utilize CUDA’s shuffling instructions.

4.7 Shared Storage Initialization Strategy

Each reduction operation must have one thread that initializes the shared storage using the starting value of the reduction operation, i.e., the current value of the reduction variable. A default initialization value for rest of the threads depends on the type of the reduction operator associated with each reduction variable. The basic principle is to initialize it with the identity element. For addition and subtract operations, the default initialization is 0. For multiply and divide operations, the default initialization is 1. For min() and max() operations, the default initialization is MAXIMUM and MINIMUM value of the variable data type,
respectively.

4.8 Evaluation

4.8.1 System Setup

We ran our benchmarks on the following system:

CPU: Intel Core i7-6700K\(^2\) with 64GB of RAM.

GPU: NVIDIA GeForce GTX 1080 Ti with 11GB of VRAM with peak memory bandwidth of 484 GB/s.

OS: Ubuntu x86 64-Bit 16.04.2 LTS, Linux kernel 4.4.0-140.

Software: GNU GCC 5.5.0, CUDA [70] 10.0.130, NVIDIA driver ver. 410.79, Oracle labsjdk1.8.0u192-jvmci-0.54, GraalVM [57] (github revision 08cd504), OpenACC PGI [38] CE ver. 18.10, CUB [68] ver. 1.8.0 and Thrust [69] ver. 1.9.3.

4.9 Performance vs Manual Native Implementation

4.9.1 Benchmark Selection

We used the existing set of pure Python benchmarks from Rodinia, Numba, and NVIDIA benchmark suites, denoted as \( Set_{all} \), that have been implemented with simply written loops without external libraries. Six of those benchmarks, which are backprop, lud, mm, particlefilter and srand, denoted as \( Set_{R} \), contain reduction loops that REDIZER can parallelize. That said, our experiment includes all benchmarks \( Set_{all} \) to demonstrate the overall impact of our transformations.

\(^2\)Turbo Boost disabled
Figure 4.11: Speedup of MEGAGUARDS with REDIZER (CUDA and OpenCL using GPU) compared to OpenCL C/C++ (GPU) and GraalPython normalized to MEGAGUARDS without REDIZER (OpenCL using GPU).

4.9.2 Experimental Setup

We ran the Python implementations of each benchmark to measure the performance of REDIZER. To properly measure the peak performance, we warmed up the benchmarks first to allow GraalPython to just-in-time compile the Python code. We ran the hand optimized OpenCL C/C++ implementation of each benchmark on the GPU to measure the differences between JIT-ed Python code against statically compiled C/C++ code. We also warmed up the OpenCL C/C++ implementation to hide device configuration and kernel compilation, and measured the peak performance.

We ran each benchmark five times on each system and calculated the geometric mean of the execution times. We measured the execution times including data transfers from and to the GPU. We carefully chose program inputs that represent real-world large datasets and simulations.

4.9.3 Discussion

Figure 4.11 shows speedups normalized to MEGAGUARDS using OpenCL as a backend targeting the GPU without REDIZER. We ran benchmarks with REDIZER on both backends
Table 4.1: Execution time, in seconds, for benchmarks in $Set_R$ for MegaGuards without Redizer, MegaGuards with Redizer, OpenCL C/C++ (GPU) and sequential GraalPython of MegaGuards, OpenCL and CUDA, denoted as MegaGuards (OpenCL) + Redizer and MegaGuards (CUDA) + Redizer. The second from the last set of bars, GeoMean, represents the geometric mean performance over all benchmarks $Set_{all}$ on each system. The last set of bars, GeoMean2, represents the geometric mean performance over the six benchmarks $Set_R$. We measured the performance with the largest data sizes that each benchmark suite provides.

MegaGuards+Redizer outperformed the baseline MegaGuards on average by $1.63 \times$ for $Set_{all}$ and by $2.98 \times$ for $Set_R$, and up to $7.25 \times$ using the same backend, OpenCL. Overall, Redizer on top of MegaGuards achieves 67% and 72% of the hand optimized C/C++ OpenCL and CUDA performance on $Set_{all}$, and 87% and 92% of the OpenCL and CUDA backends on $Set_R$.

For backprop, a back propagation neural network program, the pure Python implementation running with MegaGuards+Redizer outperformed Rodinia’s hand optimized C/C++ OpenCL version by $1.84 \times$. Redizer identified and transformed 8 reduction loops, 4 of them were transformed with Nestedness-Aware Reduction and 4 of them with Two-Stage

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GraalPython</th>
<th>MEGAuards OpenCL</th>
<th>MEGAuards + Redizer OpenCL</th>
<th>MEGAuards + Redizer CUDA</th>
<th>C/C++ OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>0.395</td>
<td>0.134</td>
<td>0.037</td>
<td>0.037</td>
<td>0.068</td>
</tr>
<tr>
<td>lud</td>
<td>277.947</td>
<td>4.961</td>
<td>1.446</td>
<td>1.384</td>
<td>0.571</td>
</tr>
<tr>
<td>mm</td>
<td>99.831</td>
<td>0.628</td>
<td>0.087</td>
<td>0.086</td>
<td>0.062</td>
</tr>
<tr>
<td>nbody</td>
<td>286.941</td>
<td>3.215</td>
<td>2.48</td>
<td>1.948</td>
<td>2.685</td>
</tr>
<tr>
<td>particlefilter</td>
<td>11.506</td>
<td>0.212</td>
<td>0.093</td>
<td>0.093</td>
<td>0.075</td>
</tr>
<tr>
<td>srad</td>
<td>10.023</td>
<td>0.939</td>
<td>0.353</td>
<td>0.344</td>
<td>0.348</td>
</tr>
<tr>
<td>GeoMean</td>
<td>26.705</td>
<td>0.803</td>
<td>0.269</td>
<td>0.255</td>
<td>0.235</td>
</tr>
</tbody>
</table>
Reduction, while the hand optimized C/C++ OpenCL version missed two viable reduction loops. Redizer’s reduction transformations created a synergy with MegaGuards’ KDM optimization which allows kernels to share common data, avoiding redundant data transfers to/from the host. Using our method, more loops can run on the GPU and as a result, more data can stay and be reused within the GPU.

One of the Two-Stage reduction loops has the number of iterations lower than the GPU’s warp size, and thus our code generation backend decided to execute it sequentially. MegaGuards+Redizer outperformed Numba’s C/C++ OpenCL version of nbody by 1.37× after performing a single nestedness-aware reduction transformation. Redizer provides a significant performance boost over the baseline MegaGuards for the other four benchmarks in Set\(_R\) with speedups ranging between 2.28× and 7.33×. None of the four benchmarks outperformed the hand optimized C/C++ OpenCL version because in the hand optimized code loops are aggressively restructured to share data between threads, especially for lud and mm. Also, the hand optimized C/C++ OpenCL versions of particlefilter and srand manually avoid unnecessary data transfer for scratch data that are only used for intermediate results.

Table 4.1 shows the execution time (in seconds) for each system. Notably, applying our method has no negative performance impact over the rest of Set\(_\text{all}\) benchmarks on MegaGuards. Table 4.2 shows the cost of performing our transformation (in milliseconds) starting from identifying a reduction loop to the loop flattening and the process of transforming the sequential loop AST into a parallel one. The cost also includes failed attempts to identify a reduction for loops that are not parallelizable.
Figure 4.12: Mean speedup of running steps for Set\textsubscript{R} benchmarks normalized to MEGAGUARDS without REDIZER, all using GPU.

Figure 4.12 shows speedups during the warm up steps of MEGAGUARDS+REDIZER normalized to the baseline MEGAGUARDS. Our transformation is done during the first step, and as we can see our analysis and transformation pass has no negative impact. In fact, our transformation outperformed the baseline even in early stages of warmup runs.

Figure 4.13 shows the performance of our method on a variety of input sizes for Set\textsubscript{R} benchmarks. Our transformation reached a steady state when the GPU reached full utilization. Notably, MEGAGUARDS+REDIZER accelerated GraalPython’s performance for Python programs by 48.8× on average.
4.10 Performance vs Native Reduction APIs

4.10.1 Benchmark Selection

To the best of our knowledge, there is no available implementation of $Set_R$ benchmarks to measure performance of REDIZER against existing optimized parallel reduction APIs libraries. Porting such benchmarks to a reduction API is tedious work and out of the scope of this dissertation. However, we included two micro benchmarks, \texttt{Sum} and \texttt{Srand}, denoted as $Set_M$. This benchmark selection is aligned with previous work [82]. \texttt{Sum} benchmark has a single sum reduction on an input array, and \texttt{srand} has a reduction loop that consists of two reductions as shown in Figure 4.10, but without the diffusion part. Both benchmarks ran with an input array of size $2^{26}$ with three different data types for each: \texttt{SumInt}, \texttt{SumLong}, \texttt{SumDouble}, \texttt{SrandInt}, \texttt{SrandLong} and \texttt{SrandDouble}. Python implementations of $Set_M$ benchmarks consist of simply written loops without annotations, while OpenACC PGI C/C++ implementations consist of simply written loops with acceleration directives. Whereas, Thrust and CUB are implemented using reduction APIs.

4.10.2 Experimental Setup

We ran each of $Set_M$ benchmarks 500 times and averaged the results. We measured the reduction performance by the effective bandwidth utilization using the following equation [82]:

$$\text{Bandwidth} = \frac{\text{InputArraySize} \times \text{SizeOf(DataType)}}{\text{ReductionTime}}$$ (4.2)

For consistency, all systems were measured on the CUDA framework, and we performed warmups and dry runs to hide kernel compilation time and to make sure that input data has been transferred to the target GPU.
4.10.3 Discussion

Figure 4.14, shows the effective bandwidth utilization for each system with \( S_{eM} \) benchmarks.

Sum

Both CUB and Thrust perform summation using a single \texttt{reduce()} method library call. CUB outperformed all other systems especially for the input array of type \texttt{int}. \textsc{MegaGuards} enforces a runtime overflow checks for the \texttt{int} and \texttt{long} binary arithmetic operation, which contribute to this anticipated overhead.

Srand

Since \texttt{srand} performs two reductions, both CUB and Thrust required two separate library calls. The first reduction of \texttt{srand} is a normal summation while the second reduction performs a summation of squares for the input array. CUB required an additional step to compute the squares while Thrust library provides a \texttt{transform_reduce()} method that allows to simultaneously compute the square and the sum. Nevertheless, both Thrust and CUB required two separate reduction routines for the input array, which resulted in poor overall bandwidth utilization. On the other hand, both OpenACC and \textsc{Redizer} were able to fuse both reductions to be performed in a single reduction routine. OpenACC outperformed all other systems including \textsc{Redizer} for the input arrays of \texttt{int} and \texttt{long} data types. The main reason is extra overflow checks for Python: Python enforces overflow checks for addition and multiply operations, while C/C++ does not. However, for \texttt{double} data type, \textsc{MegaGuards+Redizer} performed on par with OpenACC, and outperformed CUB, and Thrust as no overflow check is required for \texttt{double}. 
Figure 4.13: Scalability of MegaGuards with Redizer (CUDA and OpenCL) with different $Set_R$ benchmarks’ input sizes compared to OpenCL C/C++ normalized to MegaGuards without Redizer, all using GPU.
Figure 4.14: Peak bandwidth throughput on NVIDIA GPU GTX 1080 Ti for MegaGuards with Redizer, NVIDIA CUB, Thrust and PGI OpenACC, all using CUDA as backend framework.
Chapter 5

Related Work

5.1 Type Inference and Guards Optimization

Several guards optimizations [6, 54, 26] have been proposed to reduce the number of type checks during executions of a dynamically-typed languages. Bebenita et al. proposed a profile-based guards optimization that significantly reduces the number of type checks on hot execution paths [6]. Dot et al. proposed a HW/SW-based profiling mechanism to reduce the number of guards [26]. The disadvantage of this approach is that the profiling itself incurs some overhead, and that guards optimization only applies to code paths that execute during profiling. MEGAGUARDS, by contrast, would also optimize those code paths. Kedlaya et al. proposed to optimize guards using type inference and type feedback [54]. The type inference phase has similarities to our approach, but does enforce guards around global variables and function calls. MEGAGUARDS, on the other hand, propagates global variable types to the generated mega guard and creates internal specialized variants of functions based on their argument types. Thus, the MEGAGUARDS-specialized AST contains guards-less regions with static typing properties and optimized bounds and overflow checks.
5.2 Heterogeneous Programming in Dynamic Languages

Although their popularity resulted from ease of use and high productivity, dynamically-typed languages are less attractive in terms of performance given their traditional lack of support for parallelism. PyCUDA and PyOpenCL facilitate native GPU programming in Python to enable accelerations on parallel hardware for dynamically-typed languages [55]. Harnessing the full potential of GPU with these platforms, however, requires low-level understanding of heterogeneous programming models and faces a steep learning curve.

Several programming models have been proposed to ease the exploitation of parallel hardware for dynamically-typed languages. Numba [58] and unPython [34] proposed an annotation-based solution to perform vectorized operations on both CPU and GPU. Theano provides a set of pre-compiled vector operations for GPUs [7]. Copperhead is a Python programming model that performs GPU parallelization using aggregate operations, called skeletons, such as map and reduce, that are implicitly parallel [15]. Chakravarty et al. also proposed a similar skeleton-based aggregate operations to dynamically generate GPU code for Haskell programs [16]. Jibaja et al. proposed a language extension for JavaScript to support SIMD vector instructions [49]. This requires users to explicitly specify data types of program operations. River Trail [46] proposed a programming model for the implicitly parallel operations targeting OpenCL acceleration devices. To get the full performance benefits, however, these approaches require code annotations and the use of parallel libraries and special type definitions such as parallel arrays and NumPy’s type system. The overarching goal of this dissertation is, however, to automatically and transparently exploit heterogeneous parallel hardware without code rewriting and knowledge of underlying system architecture.
5.3 Auto-Parallelization for Dynamic Languages

There are previous approaches to automatically vectorize or parallelize vector computations in dynamically-typed languages [94, 67, 101, 76]. Plangger and Krall introduced a vectorization technique that uses SIMD vector instructions on the CPU [76]. The technique is employed in PyPy. Plangger and Krall also applied loop unrolling and array bounds check optimizations to enhance the vectorization. This solution relies on NumPy’s type system, however, and only works for loops whose structure matches one of the patterns supported by the tool. Similarly, Riposte [94] and pqR [67] exploit parallelism on the CPU vector operations for the R language. Wang et al. vectorize the `Apply` class of operations targeting multi-core CPUs and GPUs [101].

Fumero et al. offload the `Apply` to GPU using the collected type information profile on Graal’s partial evaluation [31]. Their approach bears some similarities with ours in how they check input data types before offloading and how they handle mis-speculations occurred while executing kernels. However, Fumero et al. rely on a language’s parallel semantic, i.e., `Apply` and still requires that developers deal with the effects of arithmetic overflows themselves. On the other hand, MEGAGUARDS is not confined to certain types of operations, such as vector computations, or specific forms of operations, such as operations in the `Apply` class, which is what prior work does. Moreover, MEGAGUARDS ensures the integrity of the computed results with its in place bounds and overflow checks.

Ma et al. proposed pR that automatically parallelizes loops and independent tasks from unmodified R code [62]. pR identifies all parallelizable elements including loop iterations and methods and dispatches them to multiple CPU cores. pR does not incorporate a JIT compiler but it conducts dependence analysis and parallelization at the interpretation level. With this feature, pR does not require complicated pointer and type analysis for parallelization because the language itself does not have pointers and types. However, this approach hardly
benefits from SIMD execution on the GPU because the interpreter has complex control flows and frequently accesses shared VM state, which must generally be avoided during GPU execution. MEGAGUARDS’s parallelization is part of the JIT compilation process, and the generated code does not contain complex control flow instructions or accesses to shared VM state.

Thread-level speculation (TLS) based approaches facilitate automatic parallelization for dynamically-typed languages at the JIT compilation level [64, 66]. Mehrara et al. propose a system that dynamically parallelizes data-parallel loops in JavaScript by handling runtime type changes based on TLS [64]. In this mechanism, all live-in data is saved before speculative execution and when there is a speculation failure (e.g., a type change), the program restarts from the checkpoint with the saved data. Na et al. leverages the property of idempotence to recompute the mis-speculated loops without side effect [66], instead of checkpoint-and-recovery which may require handling of large amount of data. However, both of these TLS-based approaches do not remove speculations in multi-threaded execution. This makes it hard to move to GPUs because the mis-speculation penalty of the kernel execution is significant either with checkpointing or recomputing, due to the excessive data transfer and kernel invocation overhead. Furthermore, the mis-speculation handling code may result in complex control-flows, which should be avoided in the kernel execution. In this dissertation, we separate speculations from the offloaded kernel code based on our type feedback and analysis. This feature makes MEGAGUARDS effectively accelerate dynamically-typed languages on GPUs.

Generally, none of these techniques apply to multiple languages since the previous approaches are either based on single language platforms [62, 64] or rely on language-specific primitives [101, 31]. MEGAGUARDS, by contrast, parallelizes for loops, which are a near-universal language construct, and preserves portability of the programs. Since MEGAGUARDS is based on Truffle, a multi-language platform, our approach therefore generalizes to other Truffle lan-
5.4 Reduction Transformations

Pugh and Wonacott [79] introduced a concept of reduction dependences. The following research has been conducted on a precise reduction detection and an implicit modeling to ignore the post step of reduction loop dependencies [50, 51, 75, 77, 81, 83, 97, 107].

Redon and Feautrier [29, 84, 85] modeled reductions with a Systems of Affine Recurrence Equations (SAREs) to facilitate the recognition of induction patterns by performing array expansion and renaming. Gupta et al. [39, 40] extended Redon and Feautrier to simplify and eliminate anticipated reduction computation interferences with other statements. The practicality of this method in the domain of dynamically-typed languages and their just-in-time compilers is unclear.

Doerfert et al. [25] extended the SARE approach and proposed techniques to automatically detect and transform reductions on the LLVM IR targeting multi-core CPUs at compile time. This technique bears similarity with our approach in identifying reductions without any special preprocessing of the code. However, as a static approach, their reduction modeling does not consider GPU architecture constraints when scheduling a reduction, and does not address the overhead of parallelizing a small reduction loops that cannot be known at compile time.

Stock et al. [89] present a reorder stencil computation technique by leveraging associative and commutative reduction properties on the polyhedral model at compile time. However, they do not discuss how to identify reductions or whether their method is applicable for nested reduction within parallel loops on the GPU.
Fumero et al. [32] presented TornadoVM, a task-based parallel API targeting OpenCL enabled acceleration devices for Java. Their work allows users to specify reduction loops using their annotations and task-based API. However, their work can only support a single reduction variable at a time and does not support nested reductions within parallel loops.

Reddy et al. [82] incorporated the language constructs for reductions into PENCIL [4], an intermediate DSL for accelerators. Their work utilizes the polyhedral model to perform loop nest, including imperfectly nested loops, transformations at compile-time. This approach provides a great flexibility with the easy syntax in expressing parallel reduction routines. However, the programmers still need to learn how to use these constructs to better exploit its full capability.

High-level functional data parallel languages, such as Lift [88], Tangram [18, 17], Julia [9, 8] and Futhark [45], provide interfaces with high-level GPU programming functional patterns such as \texttt{map()} and \texttt{reduce()}. These languages help abstract data-parallel computations and dynamically adapt to the underlying GPU capabilities. However, utilizing functional patterns require loop restructuring to work around the inherent limitations of reduction APIs.
Chapter 6

Conclusion and Future Work

We have presented the design and implementation of MEGAGUARDS, a new system that automatically and transparently optimizes Python programs by offloading compute-intensive kernels to accelerator devices using both CUDA and OpenCL. The key component in our system is a priori type stability analysis step that overcomes the speculative limitations of type feedback by analyzing potential, future type changes. Only after this analysis ensures that no unexpected type changes can occur, we continue to optimize the code for execution on acceleration devices; if a dependence is detected, we mark the code for sequential execution.

Moreover, we have presented the design and implementation of REDIZER, an automatic loop parallel transformation with general reduction dependencies. REDIZER demonstrates how MEGAGUARDS system facilitates a comprehensive loop analysis framework at the JIT-compiler level to perform REDIZER’s transformations. Moreover, our JIT-approach provides a new benefit of extracting rich run-time information, e.g., loop ranges and the data size, with which we further improve our performance portability.

In our evaluation, we could see on the one hand, that this approach offers a significant extra speedup over a runtime already featuring independent loop parallelization, and on
the other hand, that the efficiency is comparable to corresponding hand-optimized native GPU code. Therefore, we believe that our approach allows Python programmers to focus on their algorithmic problem, while their plain Python reduction loops can still compete with manually written massively parallel machine code.

6.1 Future Work

Although MegaGuards is just a first step, our results indicate that this research direction holds tremendous potential for further investigation. As MegaGuards currently support builtin Python’s data structures, i.e. lists and tuples, library based data structures, e.g., NumPy’s arrays and data-frames, can be supported and incorporated with a thin layer interface that handle the boxing and unboxing process. Moreover, Supporting Python’s generators [110] holds a great potential for parallelism and heterogeneous computing.

Because our system is built on top of the Truffle framework, MegaGuards generalizes to all other Truffle languages, such as TruffleRuby and FastR. All the major Truffle languages—JavaScript, R, and Ruby—can, therefore, directly benefit from the presented techniques and their implementations and enjoy “free” and significant speedups. Extending support beyond Python can be achieved by a direct tree translation to MegaGuards’ intermediate AST. Along with the AST translation, MegaGuards requires an implementation process on how to unbox and manage boxed data structures.

The MegaGuards system can benefit from applying our priori type stability analysis on a complete program to produce a precompiled and optimized executable form. This approach can also determine the liveliness of large data structures and improve device data locality led by the KDM. Moreover, improving the adaptive device selector to select the best device for generated kernels ahead-of-time can reduce the warm up process to find the most suitable
acceleration device for each generated kernel.
Bibliography


95


Simon Urbanek. multicore: Parallel processing of r code on machines with multiple cores or cpus. R package (v: 0.1-7), 2011.


Appendix A

MegaGuards + Redizer Performance
Portability On A Different Machine

System Setup

- **CPU**: Intel Core i7-8750H\(^1\) with 32GB of RAM.

- **GPU**: NVIDIA GeForce GTX 1060 with 6GB of VRAM with peak memory bandwidth of 192 GB/s.

- **OS**: Ubuntu x86 64-Bit 18.04.1 LTS, Linux kernel 4.18.0-18.

- **Software**: GNU GCC 7.4.0, CUDA [70] 10.0.130, NVIDIA driver ver. 418.56, Oracle labsjdk1.8.0u192-jvmci-0.54, GraalVM [57] (github revision 08cd504), OpenACC PGI [38] CE ver. 19.4, CUB [68] ver. 1.8.0 and Thrust [69] ver. 1.9.3.

\(^1\)Turbo Boost disabled
Benchmark Results

![Figure A.1: Speedup of MegaGuards with Redizer (CUDA and OpenCL using GPU) compared to OpenCL C/C++ (GPU) and GraalPython normalized to MegaGuards without Redizer (OpenCL using GPU).](image)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GraalPython</th>
<th>MegaGuards (OpenCL)</th>
<th>MegaGuards (OpenCL) + Redizer</th>
<th>OpenCL C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>0.508</td>
<td>0.135</td>
<td>0.043</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.073</td>
</tr>
<tr>
<td>lud</td>
<td>329.485</td>
<td>6.308</td>
<td>3.401</td>
<td>3.276</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.402</td>
</tr>
<tr>
<td>mm</td>
<td>131.69</td>
<td>1.47</td>
<td>0.259</td>
<td>0.259</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.175</td>
</tr>
<tr>
<td>nbody</td>
<td>548.184</td>
<td>7.888</td>
<td>7.251</td>
<td>5.727</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.338</td>
</tr>
<tr>
<td>particlefilter</td>
<td>21.892</td>
<td>0.255</td>
<td>0.164</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.11</td>
</tr>
<tr>
<td>srad</td>
<td>19.12</td>
<td>1.301</td>
<td>0.789</td>
<td>0.772</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.955</td>
</tr>
<tr>
<td>GeoMean</td>
<td>41.431</td>
<td>1.219</td>
<td>0.573</td>
<td>0.530</td>
</tr>
<tr>
<td>GeoMean+</td>
<td></td>
<td></td>
<td></td>
<td>0.490</td>
</tr>
</tbody>
</table>

Table A.1: Execution time, in seconds, for benchmarks in SetR for MegaGuards without Redizer, MegaGuards with Redizer, OpenCL C/C++ (GPU) and sequential GraalPython.
Figure A.2: Mean speedup of running steps for $Set_R$ benchmarks normalized to MEGA-GUARDS without Redizer, all using GPU.
Figure A.3: Scalability of MegaGuards with Redizer (CUDA and OpenCL) with different SetR benchmarks’ input sizes compared to OpenCL C/C++ normalized to MegaGuards without Redizer, all using GPU.
Figure A.4: Breakdown of MEGAGUARDS passes for parallel execution.

Figure A.5: Peak bandwidth throughput on NVIDIA GPU GTX 1060 for MEGAGUARDS with REDIZER, NVIDIA CUB, Thrust and PGI OpenACC, all using CUDA as backend framework.