Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization

Mohaned Qunaibit
University of California, Irvine
m.qunaibit@uci.edu
https://orcid.org/0000-0001-6759-7890

Stefan Brunthaler
National Cyber Defense Research Institute CODE, Munich, and SBA Research
brunthaler@unibw.de

Yeoul Na
University of California, Irvine
yeouln@uci.edu

Stijn Volckaert
University of California, Irvine
stijnv@uci.edu

Michael Franz
University of California, Irvine
franz@uci.edu

Abstract

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 rely on programmer annotations.

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.

This paper presents MegaGuards, a new approach for speculatively executing dynamic languages on heterogeneous platforms in a fully automatic and transparent manner. Our method translates each target 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. Notably, the advantage of MegaGuards is not limited to heterogeneous computing; because it removes guards from compute-intensive loops, the approach also improves sequential performance.

We have implemented MegaGuards along with an automatic loop parallelization backend in ZipPy, a Python Virtual Machine. The results of a careful and detailed evaluation reveal very significant speedups of an order of magnitude on average with a maximum speedup of up to two orders of magnitudes when compared to the original ZipPy performance as a baseline. These results demonstrate the potential for applying heterogeneous computing to dynamic languages.

2012 ACM Subject Classification Software and its engineering → Interpreters, Software and its engineering → Just-in-time compilers, Software and its engineering → Dynamic compilers

Keywords and phrases Type Specialization, Guards Optimization, Automatic Heterogeneous Computing, Automatic Parallelism

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.16

Funding This material is based upon work partially supported by 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 BMVIT, BMDW, and the federal state of Vienna, managed by the FFG. We also gratefully acknowledge a gift from Oracle Corporation. Mohaned Qunaibit was supported by MOHE's Graduate Studies Scholarship. 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.

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 [26], 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 paper. 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 [1], 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 ZipPy [61], a modern Python 3 implementation targeting the Java Virtual Machine (JVM). ZipPy relies on the Truffle framework to optimize interpreted programs on the CPU [58]. To determine parallelizable loops, we perform a bounds check optimization and conduct dependence analysis by leveraging the polyhedral model [25]. MegaGuards dynamically translates parallel loops into OpenCL 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] Our software will be publicly available at https://github.com/securesystemslab/zippy-megaguards

In summary, the contribution of this paper are as follows:

- 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 (Section 3).
- We report results of a careful and detailed evaluation (Section 4). Specifically, our experiments indicate that MegaGuards offers:
  - **Performance**: Our measurements show that MegaGuards (i) performs within 2.82× of the average performance of handwritten, native OpenCL C/C++ implementations on the GPU, and (ii) yields substantial speedups when compared with existing Python implementations (with average speedups exceeding 84×).
  - **Implementation Efficiency**: By way of optimizing pure Python code in an automatic and transparent manner, MegaGuards removes the necessity for the labor-intensive task of manually rewriting code. To quantify these gains in implementation efficiency, we measured reductions in (i) lines of code, and (ii) McCabe's cyclomatic complexity. Our results indicate average reductions by about three quarters in both dimensions.

2 Background

2.1 Heterogeneous Programming Frameworks

Programming in a heterogeneous computing environment is highly challenging because heterogeneous programming frameworks (e.g., CUDA [42] and OpenCL [48]) have steep learning curves and requiring knowledge of the inner workings of the GPU.

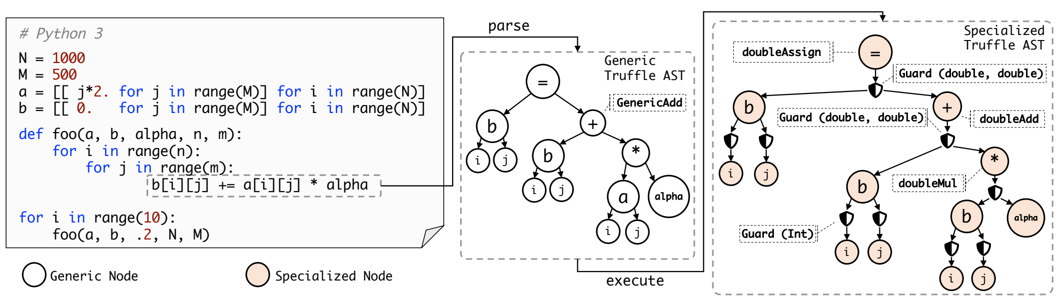
To alleviate this problem for **statically-typed languages**, researchers have proposed transformations that map existing parallel paradigms for the CPU to run on the GPU [44, 24, 56]. Others proposed libraries and lambda expressions [22, 28, 46, 47, 36, 7] to automatically generate GPU code. Some techniques automatically parallelize sequential loops and run them on GPUs [35, 3]. New languages such as Lime [18, 2] 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.

2.2 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 [16, 57]. This observation has inspired various specialization approaches that minimize the interpreter's type-checking overhead [9, 8, 59, 55, 1, 63]. In our work, we leverage specialized types to eliminate all type-checking in the generated OpenCL code.

Truffle [58], 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

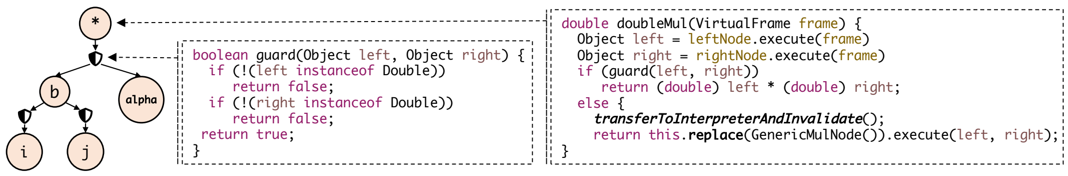


```
# Python 3
N = 1000
M = 500
a = [[ j*2. for j in range(M)] for i in range(N)]
b = [[ 0.   for j in range(M)] for i in range(N)]

def foo(a, b, alpha, n, m):
    for i in range(n):
        for j in range(m):
            b[i][j] += a[i][j] * alpha

for i in range(10):
    foo(a, b, .2, N, M)
```

Figure 1 Node specialization in Truffle.



```
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 Handling mis-speculation using type guards.

data type, with nodes that are specialized for a specific data type (Figure 1). This speculation approach facilitates just-in-time compilation of Truffle's hosted languages, which include ZipPy, FastR, and TruffleRuby. When a Truffle AST reaches a stable state, the Truffle framework invokes the Graal just-in-time compiler [19, 60] 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 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.

ZipPy, the Python 3 VM we use in MegaGuards, is built on top of Truffle. ZipPy's type system specializes objects based on their content. In Figure 3, we see several examples of type specialization in ZipPy. At ①, the program creates a list `a` containing items of the same type. ZipPy 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, ZipPy generalizes the list to be of type `ObjectList`. At ③, the program creates a multi-dimensional list `b`. In this case, ZipPy specializes the nested lists to be of type `DoubleList`. ZipPy 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).

3 The MegaGuards System

3.1 Overview

Figure 4 shows how MegaGuards fits into the ZipPy ecosystem. Conceptually, MegaGuards works as follows. First, whenever the interpreter executes a loop with an identifiable index expression, such as the `for i in range(n)` statement in Figure 1, MegaGuards

```
# 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)]
```

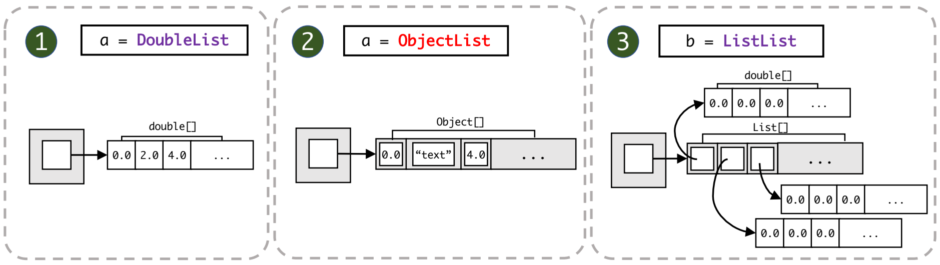


Figure 3 ZipPy type specialization.

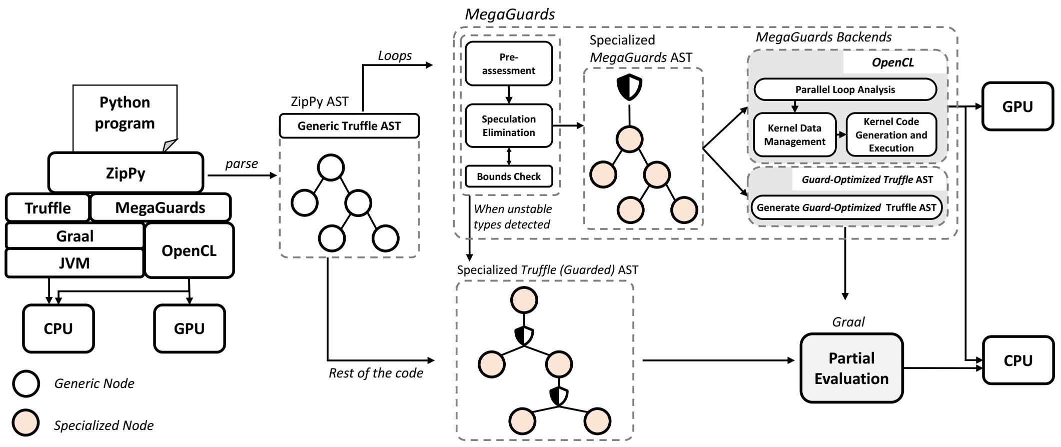


Figure 4 ZipPy+MegaGuards System Overview.

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 *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 3.3.3). After that, MegaGuards performs a dependence 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 kernel code (Section 3.4.2). Finally, MegaGuards compiles the OpenCL 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 ZipPy 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.

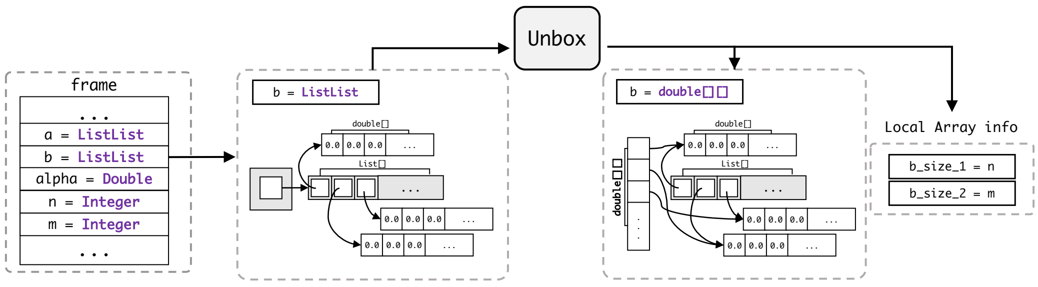
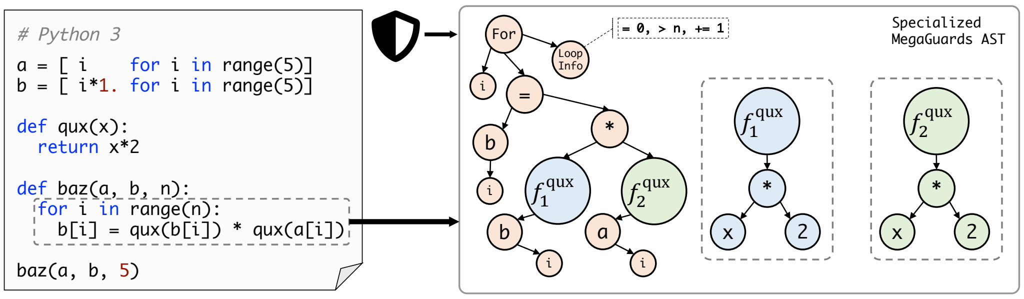


Figure 5 MegaGuards unboxing process.



```
# Python 3
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 6 MegaGuards specialized AST with inter-procedural invocations.

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 framework.

3.3 Guards Optimization

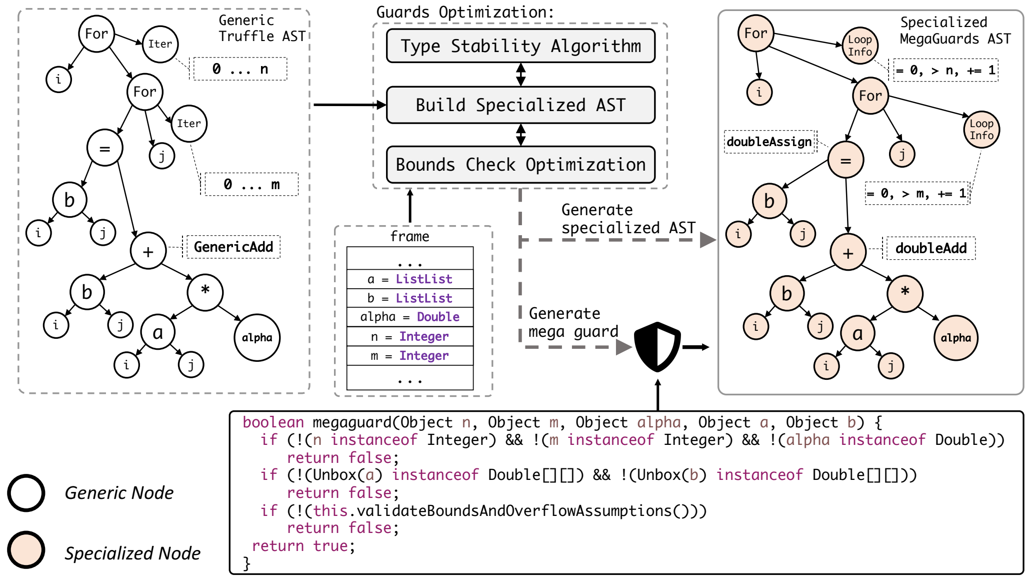
Truffle uses type guards and exceptions to handle mis-speculations, as shown in Section 2.2. MegaGuards hoists type, bounds, and overflow checks out of a loop to translate the loop into a *static region*. This way these checks are performed *before* that loop is executed. To this end, MegaGuards performs 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 3.3.3).

3.3.1 Type Stability Analysis

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 ZipPy (see

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 == AssigmentNode 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 α = (1 if β > 0 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
end
```



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

Figure 7 MegaGuards specialized AST build process.

Section 2.2). In figure 5, 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 that stores both strings and integers), it will mark that structure as type-unstable in the context frame.

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.

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 6 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`.

3.3.2 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 7 shows an example of such a translation. The figure shows how MegaGuards converts the generic `For` node in the ZipPy 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 3.3.3), 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 ZipPy AST, on the other hand, are generic and can handle any data type. These ZipPy AST nodes contain type checks and conditional branches. The MegaGuards-specialized nodes do not.

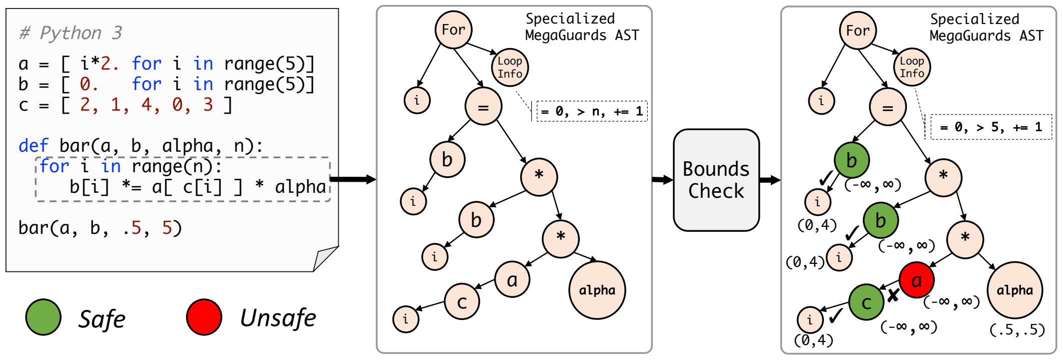
MegaGuards supports translation of AST operations that operate on generic boxed data types. The assign operation in the ZipPy AST, for example, writes to `b[i][j]`. Variable `b` is of generic type `ListList` according to the original context frame generated by ZipPy, but during type stability analysis, 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 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.

3.3.3 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 α and β 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.



```
# Python 3
a = [ i*2. for i in range(5)]
b = [ 0.   for i in range(5)]
c = [ 2, 1, 4, 0, 3 ]

def bar(a, b, alpha, n):
  for i in range(n):
    b[i] *= a[ c[i] ] * alpha

bar(a, b, .5, 5)
```

Figure 8 MegaGuards bounds check optimization process.

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 8 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 `boundsViolated` flag if it detects a violation. MegaGuards reads the value of this when the execution of the offloaded loop finishes. Access to the `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.

3.3.4 Mega Guards Insertion

Finally, MegaGuards creates a *mega guard* for the MegaGuards-specialized AST. `megaguard()` in Figure 7 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

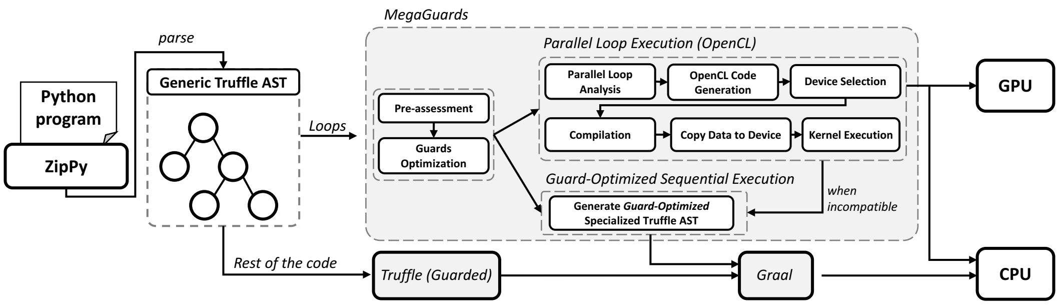


Figure 9 MegaGuards detailed internal analysis and offloading.

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 a OpenCL kernel as shown in Figure 9. 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 polyhedral dependence analysis using the Integer Set Library (`ISL`) and the Polyhedral Extraction Tool (`PET`) [53, 52], 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 10 shows how we feed the MegaGuards-specialized AST, generated from the code in Figure 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 [10]. 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 3.3.3) and then translates the AST into OpenCL 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.

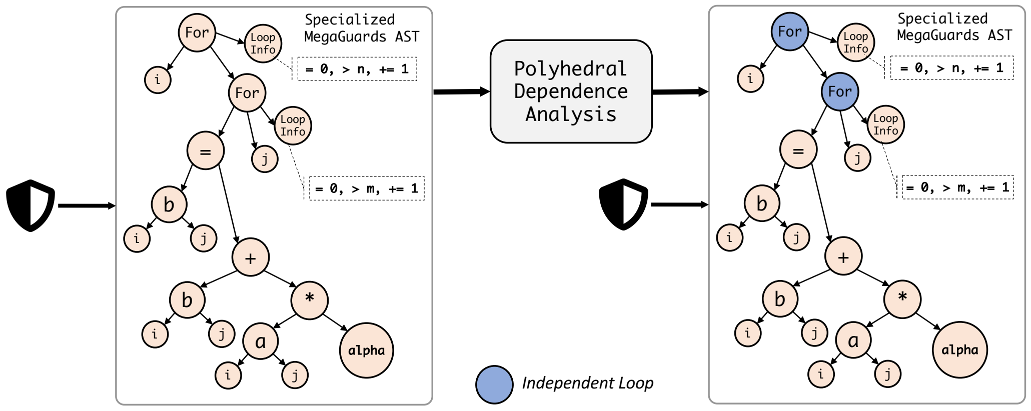
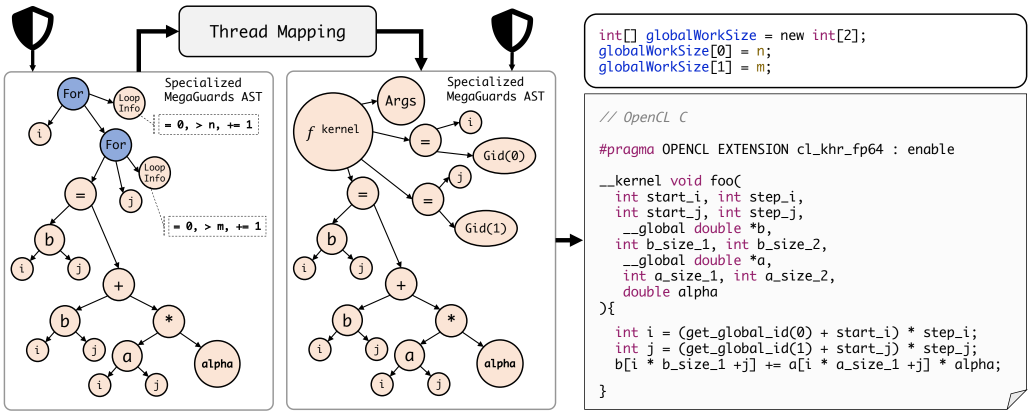


Figure 10 MEGAGUARDS polyhedral dependence analysis of a MEGAGUARDS-specialized AST.



```
int[] globalWorkSize = new int[2];
globalWorkSize[0] = n;
globalWorkSize[1] = m;
```

```
// OpenCL C

#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void foo(
  int start_i, int step_i,
  int start_j, int step_j,
   __global double *b,
  int b_size_1, int b_size_2,
   __global double *a,
   int a_size_1, int a_size_2,
   double alpha
){
  int i = (get_global_id(0) + start_i) * step_i;
  int j = (get_global_id(1) + start_j) * step_j;
  b[i * b_size_1 +j] += a[i * a_size_1 +j] * alpha;

}
```

Figure 11 MEGAGUARDS thread mapping and code generation.

3.4.3 Thread Mapping

MEGAGUARDS leverages OpenCL's multi-dimensional thread range capability, called NDRange, 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 [30, 31].

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 supports up to three dimensions, MEGAGUARDS can map nested loops with up to three nesting levels to SIMT threads.

Figure 11 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 kernel based on the SIMT programming model by rewriting the specialized AST into a kernel AST, as shown in Figure 11. 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 `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` × `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 device. This means that MegaGuards might have to copy data structures from the main memory to the OpenCL device.

To avoid redundant copy operations, MegaGuards manages a cache of data that is present on each OpenCL 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 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 device.

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 3.3.3). 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

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.

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 framework, and then specializes the translated calls based on the call arguments.

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.

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 [51], loop splitting and loop peeling [20] 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.

4 Evaluation

4.1 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 16.04.2 LTS using Linux kernel 4.4.0-122. GNU GCC 5.5.0, Oracle labsjdk1.8.0_151-jvmci-0.39, GraalVM v0.30 and AMD APP SDK v3.0.136.

We compared the performance of MegaGuards with:
Python Systems:

- **CPython** version 3.5.2: The standard Python 3 interpreter.
- **PyPy 3** version 5.10.0 [6]: Python 3 implementation, uses a meta-tracing JIT compiler to compile Python code into machine code for CPU.
- **ZipPy** (github revision `ff6d067`) [49]: 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 ver. 1.2.0.25
- **OpenCL C/C++ (GPU)** NVIDIA driver ver. 390.59

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, ZipPy, PyPy and CPython performance. To properly measure peak performance, we warmed up the benchmarks to allow ZipPy and PyPy to just-in-time compile the Python code.

We compared four different backend/device selection configurations for MegaGuards:

- **MegaGuards-Truffle**: running ZipPy 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-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.

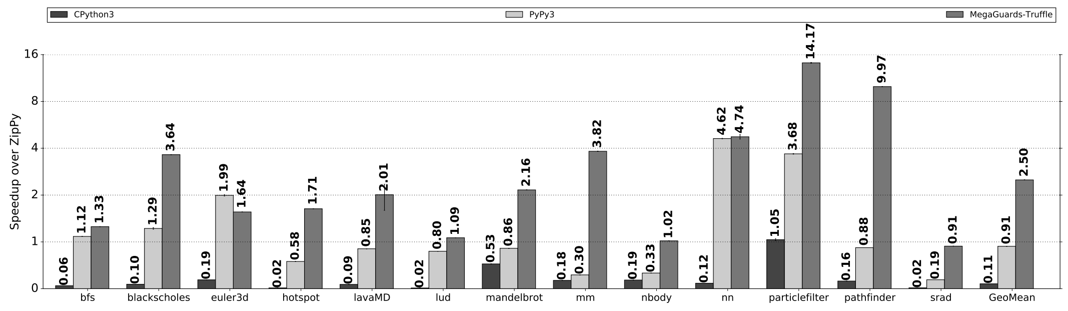


Figure 12 Sequential execution speedup of MegaGuards-Truffle compared to CPython, PyPy, ZipPy normalized to ZipPy on a $\log_2$ scale.

Table 1 Sequential execution time (in seconds) for MegaGuards-Truffle, CPython, PyPy and ZipPy.

	bfs	blackscholes	euler3d	hotspot	lavaMD	lud	mandelbrot	mm	nbody	nn	particlefilter	pathfinder	srad
ZipPy	16.913	139.482	40.366	7.205	50.329	148.981	98.069	1018.838	498.037	16.914	61.751	4.78	**4.668**
MG-Truffle	**12.759**	**38.326**	24.617	**4.219**	**25.037**	**136.91**	**45.418**	**266.377**	**487.023**	**3.567**	**4.359**	**0.48**	5.135
PyPy3	15.154	108.412	**20.236**	12.389	59.004	186.196	113.609	3452.785	1489.154	3.662	16.769	5.454	24.412
CPython	264.321	1462.925	212.529	418.922	546.66	9616.348	185.193	5734.737	2688.715	144.596	58.968	29.221	244.785

Benchmark Selection

We ported a set of benchmarks from the Rodinia benchmark suite [13, 14] to pure Python, using only Python built-in data types [2]. We complemented this extensive set of benchmarks with the ones from the Numba Benchmark Suite [15] and the NVIDIA OpenCL SDK [43].

The selected benchmark programs (listed in Table 4) have two implementations: (i) pure Python, and (ii) a native hand-optimized version for OpenCL C/C++. We excluded bfs, euler3d and lavaMD benchmarks from the table as the existing polyhedral analysis could not disprove dependence (see Section 3.4.1) and thus the benchmarks only ran using MegaGuards-Truffle backend.

4.2 Effect of Guards Optimization

In Figure 12, we show the performance impact of our guards optimization by measuring the sequential performance of ZipPy with the guards optimization enabled (MegaGuards-Truffle). The performance is normalized to the baseline ZipPy 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 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 62.3×, 12.7×, and 14.1× compared to CPython, PyPy and ZipPy. On average, we achieve a performance improvement of 22.72×, 2.74× and 2.50× over CPython, PyPy and ZipPy. particlefilter shows the most substantial performance improvement (14.17× over ZipPy) because our guards optimization removes most of its overflow checks (see Section 3.3.3).

[2] Will be publicly available at https://github.com/securesystemslab/megaguards-benchmarks

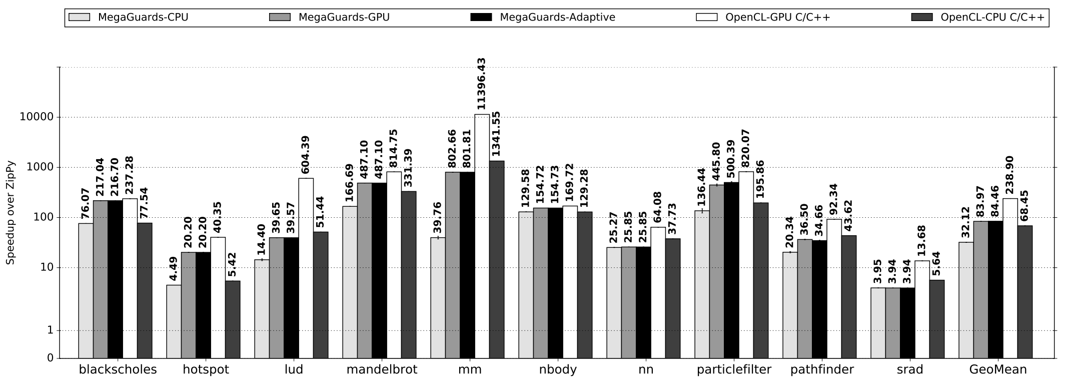


Figure 13 Parallel execution speedup of MegaGuards compared to OpenCL C/C++ (CPU and GPU) normalized to ZipPy on a $\log_{10}$ scale.

Table 2 Parallel execution time (in seconds) for MegaGuards, OpenCL C/C++ (CPU and GPU), and sequential ZipPy.

	blackscholes	hotspot	lud	mandelbrot	mm	nbody	nn	particlefilter	pathfinder	srad
ZipPy	139.482	7.205	148.981	98.069	1018.838	498.037	16.914	61.751	4.78	4.668
MG	0.644	0.357	3.765	0.201	1.271	3.219	0.654	0.123	0.138	1.186
MG-GPU	0.643	0.357	3.757	0.201	1.269	3.219	0.654	0.139	0.131	1.185
MG-CPU	1.834	1.606	10.346	0.588	25.626	3.843	0.669	0.453	0.235	1.181
OpenCL-GPU	**0.588**	**0.179**	**0.246**	**0.12**	**0.089**	**2.934**	**0.264**	**0.075**	**0.052**	**0.341**
OpenCL-CPU	1.799	1.329	2.896	0.296	0.759	3.852	0.448	0.315	0.11	0.828

4.3 Parallel Execution Performance and Complexity Analysis

4.3.1 Characteristics of Kernels

Table 4 shows the following characteristics for each benchmark:

- **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 [38] 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.

4.3.2 Parallel Execution Performance

Figure 13 shows MegaGuards's speedups normalized to ZipPy 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

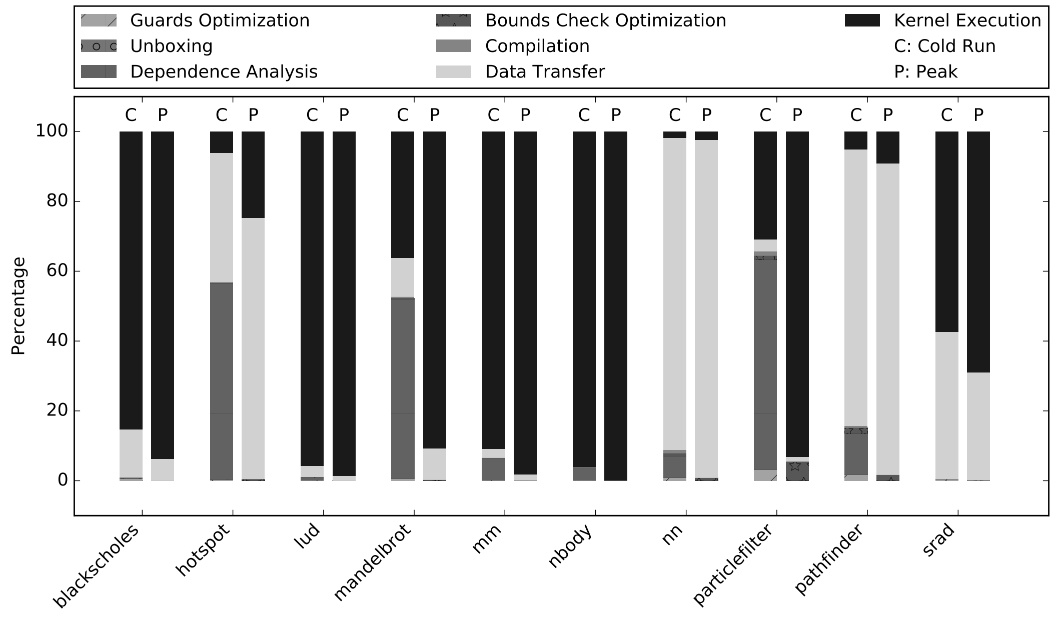


Figure 14 Breakdown of MegaGuards passes for parallel execution.

Table 3 Time (in milliseconds) for each pass of the parallel execution.

	blackscholes		hotspot		lud		mandelbrot		mm		nbody		nn		particlefilter		pathfinder		srad	
	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak
Guards Optimization	4	0	4	0	2	0	3	0	2	0	4	0	2	0	9	0	3	0	5	0
Unboxing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dependence Analysis	0	0	873	0	21	0	266	0	89	0	125	0	14	0	174	0	21	0	0	0
Bounds Check Optimization	1	0	2	2	2	1	2	1	2	0	2	2	2	1	3	4	3	2	1	1
Compilation	2	0	2	0	2	0	2	0	3	0	2	0	2	0	4	0	1	0	1	0
Data Transfer	95	38	573	305	80	31	58	34	36	23	2	1	204	122	10	1	141	108	591	362
Kernel Execution	592	563	95	101	2422	2353	187	342	1321	1221	3238	3247	4	3	89	68	9	11	803	808

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 2 shows the execution time for each benchmark complemented with the sequential execution time of ZipPy.

MegaGuards shows substantial speedups compared to other systems using pure Python benchmarks. For this set of benchmarks, our system performed up to 802× faster than ZipPy and 84× on average. MegaGuards approaches the performance of native hand-optimized OpenCL C/C++ code (CPU and GPU), being only 2.82× 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 14 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 shows the execution time of each analysis pass for each benchmark (in milliseconds). Noticeably in Figure 14, 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

Table 4 Benchmark characteristics for parallel execution.

benchmark	Loops		Kernels		Thread Count	MegaGuards-Adaptive		LOC		McCabe Cyclomatic Complexity	
	Total	Offloaded	Gen.	Exec.		CPU	GPU	Python	OpenCL	Python	OpenCL
blackscholes	2	1	1	100	8.4×10^8	0	1	51	203	13	41
hotspot	12	2	1	10	1.0×10^9	0	1	146	288	32	99
lud	3	2	2	1024	1.0×10^7	0	2	55	391	44	104
mandelbrot	3	3	1	1	6.7×10^7	0	1	42	183	16	51
mm	3	3	2	2	8.4×10^6	0	2	29	230	15	49
nbody	2	2	1	1	6.6×10^4	0	1	34	192	10	44
nn	3	1	1	1	1.6×10^7	0	1	82	456	24	67
pathfinder	2	1	1	101	3.0×10^7	0	1	63	258	22	92
particlefilter	36	16	10	94	1.2×10^6	0	10	255	719	101	205
srad	9	4	2	4	2.1×10^8	0	2	89	477	27	136
Median	3	2	1	7	3.0×10^7	0	1	59	273	23	79.5

Offloaded: Number of offloaded loops. **Gen.**: Number of generated kernels. **Exec.**: Number of kernels' executions.

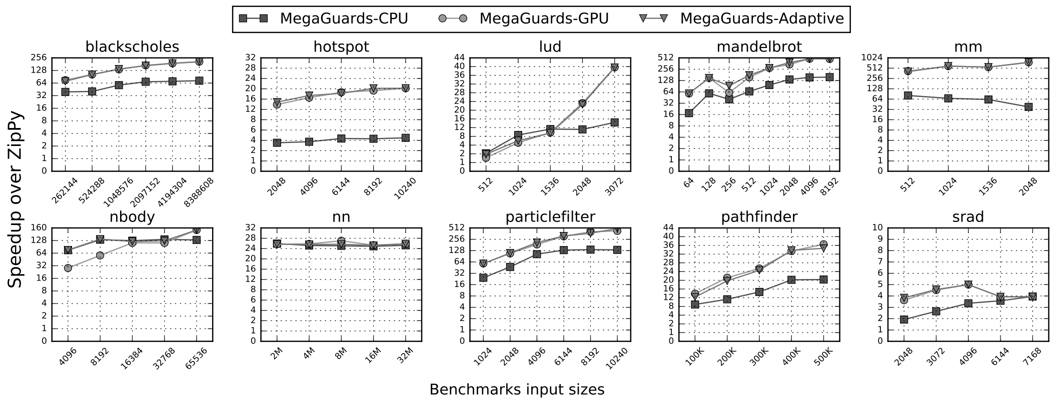


Figure 15 Peak performance of MegaGuards with different data sizes for parallel execution.

of the OpenCL C/C++ implementations and is able reduce the number of bounds and overflow checks significantly (see Section 3.3.3). MegaGuards outperformed ZipPy by 217× and 154×.

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 mm outperformed MegaGuards by 14.2×. 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. 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 mm benchmark on MegaGuards demonstrates the flexibility of our system to adapt to type changes at run time without degrading performance.

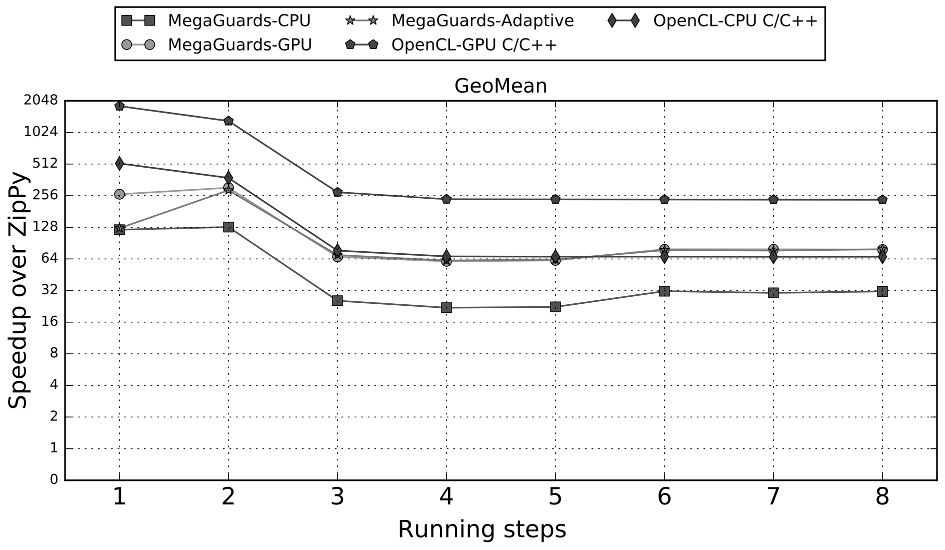


Figure 16 Mean speedup of warming up steps of MegaGuards normalized runs for ZipPy.

The situation in particlefilter, srad, hotspot and lud is similar. MegaGuards generated ten, two, one and two specialized kernels for these benchmarks respectively. Most kernels were assigned to a 1-dimensional thread range in particlefilter and lud, and to 2-dimensional thread ranges in srad and hotspot. In the OpenCL-GPU implementation, the core computation of particlefilter, srad, hotspot and lud features cooperative local threads that share data through a local cache. As a result, OpenCL-GPU outperformed MegaGuards using a GPU by 1.6×, 3.4×, 2× and 15.2×, respectively.

Overall, we observed that acceleration-compatible loops experienced speedups by up to an order of magnitude under MegaGuards.

4.3.3 Peak Performance with Various Input Sizes

To show the scalability of MegaGuards, we measured the peak performance with different data sizes. The results are normalized to the sequential ZipPy implementation. As shown in Figure 15, 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.

4.3.4 Performance of MegaGuards on Each Run Step

So far, we only measured the peak performance of ZipPy. We gave ZipPy's underlying Truffle/Graal stack and PyPy a couple of warm up runs to specialize, optimize and compile every hot path in ZipPy'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.

These performance benefits are illustrated in Figure 16. In this figure, we see the mean performance of each run of our benchmarks. In the first run, ZipPy is executing the benchmarks in the interpreter. Through the second to the sixth runs, the JIT compiler

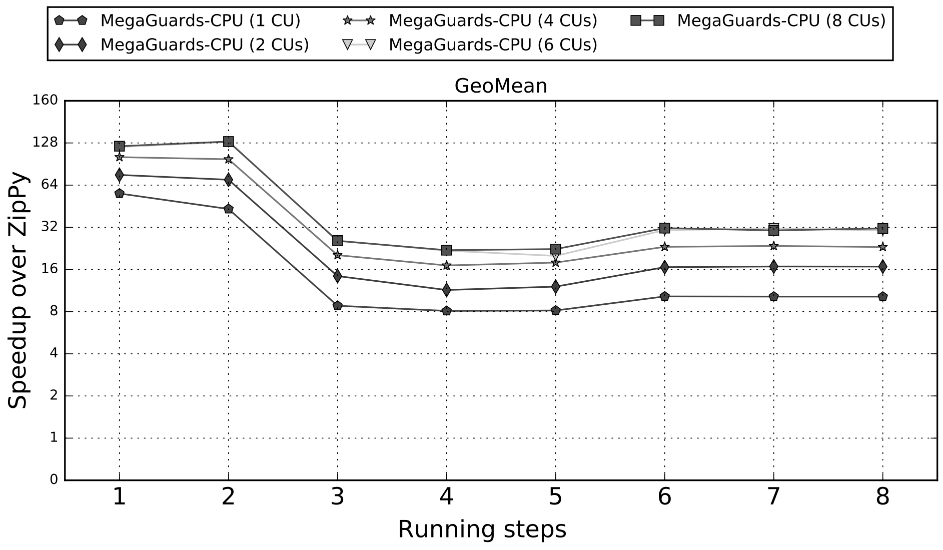


Figure 17 Mean speedup of warming up steps of MegaGuards-CPU with various CU counts normalized runs for ZipPy.

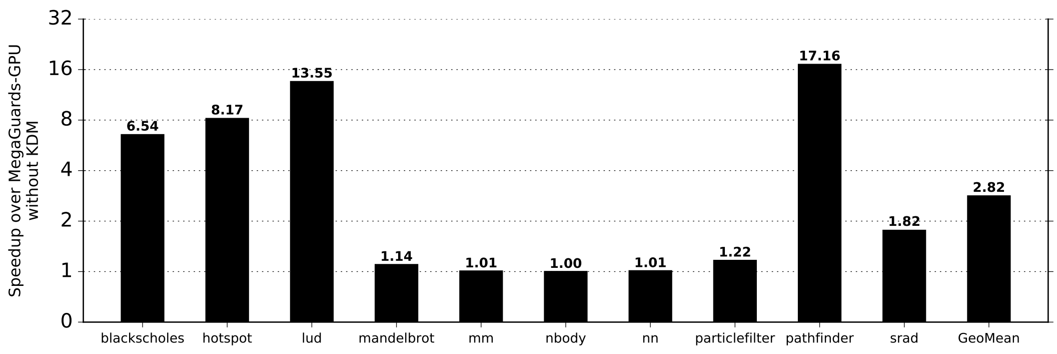


Figure 18 Effect of KDM optimization on MegaGuards.

specializes and optimizes the code. After that, ZipPy 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.

4.3.5 Scalability

Figure 17 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 13.

4.3.6 Effect of Kernel Data Management

In Figure 18, we show the effect of our kernel data management optimization on peak performance. This optimization yields an average performance gain of 2.82×. The KDM optimization provides the larger performance gains for the benchmarks with the higher data transfer rates.

5 Related Work

5.1 Type Inference and Guards Optimization

Several guards optimizations [4, 32, 17] 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 [4]. Dot et al. proposed a HW/SW-based profiling mechanism to reduce the number of guards [17]. 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 [32]. 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 [33]. 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 [34] and unPython [23] 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 [5]. Copperhead is a Python programming model that performs GPU parallelization using aggregate operations, called skeletons, such as `map` and `reduce`, that are implicitly parallel [11]. Chakravarty et al. also proposed a similar skeleton-based aggregate operations to dynamically generate GPU code for Haskell programs [12]. Jibaja et al. proposed a language extension for JavaScript to support SIMD vector instructions [29]. This requires users to explicitly specify data types of program operations. River Trail [27] 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 paper 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 [50, 41, 54, 45]. Plangger and Krall introduced a vectorization technique that uses SIMD vector instructions on the CPU [45]. 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 [50] and pqR [41] 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 [54].

Fumero et al. offload the `Apply` to GPU using the collected type information profile on Graal's partial evaluation [21]. 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 [37]. 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 exection 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 [39, 40]. Mehrara et al. proposes a system that dynamically parallelizes data-parallel loops in JavaScript by handling runtime type changes based on TLS [39]. 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 [40], 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 paper, 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 [37, 39] or rely on language-specific primitives [54, 21]. MegaGuards, by contrast, parallelizes *for loops*, which are a nearuniversal 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 languages, such as R, JavaScript, and Ruby.

6 Conclusions

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. The key component in our system is an *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.

Because our system is built on top of the ZipPy Python 3 virtual machine, which itself builds on 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.

Although MegaGuards is just a first step, our results indicate that this research direction holds tremendous potential for further investigation. Presently, we are interested in extending MegaGuards in multiple ways. First, we plan on improving the adaptive device selection to be able to select the best device for generated kernels ahead-of-time. Second, we plan to apply our *a priori* type stability analysis on a complete program to produce a precompiled and optimized executable form. Moreover, our plan is to add support for heterogeneous computing to generators [62]. Finally, support for collaborative CPU and GPU parallelism is also an interesting research direction.

References

1 Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA ’14, pages 777–790, New York, NY, USA, 2014. ACM.

2 Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC ’12, pages 271–276. ACM, 2012.

3 Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction*, CC’10, pages 244–263. Springer-Verlag, 2010.

4 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: A trace-based jit compiler for cil. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’10, pages 708–725, New York, NY, USA, 2010. ACM.

5 James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

6 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Pypy, 2017. URL: `http://pypy.org/`.

7 Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 194–205. ACM, 2016.

8 Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 1–14. ACM, 2010.

9 Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 429–451. Springer-Verlag, 2010.

10 Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. Automatic generation of nested, fork-join parallelism. *The Journal of Supercomputing*, 3(2):71–88, 1989.

11 Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56. ACM, 2011.

12 Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14. ACM, 2011.

13 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54. IEEE Computer Society, 2009.

14 Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '10, pages 1–11. IEEE Computer Society, 2010.

15 Continuum Analytics. Numba benchmark suite, 2017. URL: `https://github.com/numba/numba-benchmark`.

16 L Peter Deutsch and Allan M Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM, 1984.

17 G. Dot, A. Martinez, and A. Gonzalez. Removing checks in dynamically typed languages through efficient profiling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 257–268, Feb 2017.

18 Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12. ACM, 2012.

19 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. `doi:10.1145/2542142.2542143`.

20 Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

21 Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time gpu compilation for interpreted languages with partial evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 60–73, New York, NY, USA, 2017. ACM.

22 Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime code generation and data management for heterogeneous computing in java. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 16–26. ACM, 2015.

23 Rahul Garg and José Nelson Amaral. Compiling python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 19–30. ACM, 2010.

24 Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75. ACM, 2014.

25 Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

26 Gregory D. Hager, Mark D. Hill, and Katherine Yelick. Opportunities and challenges for next generation computing, 2015.

27 Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River trail: A path to parallelism in javascript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 729–744, New York, NY, USA, 2013. ACM.

28 Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, PACT '15, pages 419–431. IEEE Computer Society, 2015.

29 Ivan Jibaja, Peter Jensen, Ningxin Hu, Mohammad R. Haghighat, John McCutchan, Dan Gohman, Stephen M. Blackburn, and Kathryn S. McKinley. Vector parallelism in javascript: Language and compiler support for simd. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, PACT '15, pages 407–418, Washington, DC, USA, 2015. IEEE Computer Society.

30 Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.

31 Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing gpu concurrency in heterogeneous architectures. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 114–126. IEEE, 2014.

32 Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 37–48, New York, NY, USA, 2013. ACM.

33 Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

34 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

35 Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 91–100. ACM, 2009.

36 Thibaut Lutz and Vinod Grover. Lambdajit: A dynamic compiler for heterogeneous optimizations of stl algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 99–108. ACM, 2014.

37 X. Ma, J. Li, and N. F. Samatova. Automatic parallelization of scripting languages: Toward transparent desktop parallel computing. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–6. IEEE, March 2007. doi:10.1109/IPDPS.2007.370488.

38 T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

39 Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 87–98. IEEE Computer Society, 2011.

40 Yeoul Na, Seon Wook Kim, and Youngsun Han. Javascript parallelizing compiler for exploiting parallelism from data-parallel html5 applications. *ACM Trans. Archit. Code Optim.*, 12(4):64:1–64:25, 2016.

41 Radford M. Neal. pqr, 2016. URL: http://www.pqr-project.org/.

42 NVIDIA Corporation. Cuda, 2017. URL: https://developer.nvidia.com/cuda-zone.

43 NVIDIA Corporation. Nvidia opencl sdk code samples, 2017. URL: https://developer.nvidia.com/opencl.

44 Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10. IEEE Computer Society, 2013.

45 Richard Plangger and Andreas Krall. Vectorization in pypy's tracing just-in-time compiler. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, pages 67–76, New York, NY, USA, 2016. ACM. doi:10.1145/2906363.2906384.

46 Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. Rootbeer: Seamlessly using gpus from java. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 375–380. IEEE Computer Society, 2012.

47 Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217. ACM, 2015.

48 John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, 2010.

49 Secure Systems and Software Laboratory. Zippy, 2015. URL: https://github.com/securesystemslab/zippy.

50 Justin Talbot, Zachary DeVito, and Pat Hanrahan. Riposte: A trace-driven compiler and parallel vm for vector code in r. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 43–52. ACM, 2012.

51 Peng Tu and David Padua. Automatic array privatization. In *Compiler optimizations for scalable parallel systems*, pages 247–281. Springer, 2001.

52 Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, pages 54:1–54:23, 2013.

53 Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *In Second International Workshop on Polyhedral Compilation Techniques (IMPACT '12)*, 2012.

54 Haichuan Wang, David Padua, and Peng Wu. Vectorization of apply to reduce interpretation overhead of r. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 400–415. ACM, 2015.

55 Haichuan Wang, Peng Wu, and David Padua. Optimizing r vm: Allocation removal and path length reduction via interpreter-level specialization. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 295:295–295:305. ACM, 2014.

56 Zheng Wang, Daniel Powell, Björn Franke, and Michael O'Boyle. *Exploitation of GPUs for the Parallelisation of Probably Parallel Legacy Code*, pages 154–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

57 Thomas Wurthinger, Christian Wimmer, Christian Humer, Andreas Woss, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676. ACM, 2017.

58 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204. ACM, 2013.

59 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages - DLS '12*, page 73. ACM Press, 2012.

60 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. `doi:10.1145/2384577.2384587`.

61 Wei Zhang. *Efficient Hosted Interpreter for Dynamic Languages*. PhD thesis, University of California Irvine, 2015.

62 Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating iterators in optimizing ast interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 727–743. ACM, 2014.

63 Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The hiphop compiler for php. *SIGPLAN Not.*, pages 575–586, 2012.