Can we run in parallel?
Automating Loop Parallelization for TornadoVM

Shreyansh Kulshreshtha
IIT Mandi, India
shreyanshkuls@outlook.com

Rishi Sharma
IIT Mandi, India
rishi-sharma@outlook.com

Manas Thakur
IIT Mandi, India
manas@iitmandi.ac.in

Abstract

In recent times, the performance of single-core processors has almost stagnated, attributed primarily to the technical difficulties imposed by working on micro scales and the power wall problem. Nevertheless, the overall computational performance is still improving every year, thanks to the advent of multi-processor systems, which range from multi-core CPUs to heterogeneous GPUs to highly customizable FPGAs. To get the best out of these hardware units, however, we need to keep all the units busy. If some units are idle, while others are computing, we are wasting computing potential and time. To prevent this wastage and increase the utilization of the available computing power, we run chunks of the program across different processing units simultaneously, resulting in *parallelization*.

Parallelization enables programs to run much faster than their unparallelized versions. However, owing to the overhead of parallelization, it is usually important to parallelize those portions of code that take the most time (i.e., loops). In general, we divide the iterations of loops into chunks and run them simultaneously. But a fair amount of analysis is required to make sure that the parallel program will run correctly and consistently. Furthermore, actually writing parallel code needs quite a bit of expertise in parallel programming and sometimes in architecture-specific constructs, something which most scientific programmers do not possess. This problem is tackled by TornadoVM [2], a Java accelerator plugin for OpenJDK, that with the programmers' aid can parallelize a piece of a Java program and automatically run it on heterogeneous hardware. TornadoVM currently supports parallelization of `for` loops, which can be enabled programmatically by inserting `@Parallel` annotations at appropriate places, as can be seen in Figure 1.

```
1 for (@Parallel int i = 0; i < n; ++i) {
2     // Do something
3 }
```

Figure 1. Placement of `@Parallel` annotation.

Parallelizing loops, although being a potentially powerful technique to execute a program much faster, in general cannot be used for every loop present in the program. If a loop contains loop-level dependences, parallelizing it could lead to incorrect results. As an example, Figure 2 shows a code where each iteration of the loop swaps adjacent array elements by using a local temporary variable `temp`. The write to `ar[i]` from iteration `i` and the read of `ar[i-1]` from iteration `i+1` on line 5 represent the *read-after-write* dependence. Similarly, the writes to `ar[i]` on line 5 from iteration `i` and `ar[i-1]` on line 6 from iteration `i+1` represent the *write-after-write* dependence. Since the memory location is mutated in each iteration, these can lead to incorrect results if the iterations are run in parallel.

```
1 public void swap(int[] ar) {
2     int n = ar.length;
3     for(int i = 0; i < n; i++) {
4         int temp = ar[i];
5         ar[i] = ar[i-1];
6         ar[i-1] = temp;
7     }
8 }
```

Figure 2. Code to swap adjacent array elements.

When an annotated program is passed to TornadoVM, it does not perform any checks to ensure that the marked loops are parallelizable and assumes that the programmer has inserted the annotations correctly. Since it is not an easy task for a general programmer to understand loop-level dependence and side-effects in large codebases, we present to you a tool called `AutoTornado`. As the name suggests, `AutoTornado` automatically identifies the loops that can be parallelized and passes this information to TornadoVM, which in turn parallelizes the appropriate loops. Analyses in `AutoTornado` are written using Soot [3], a Java optimizing framework, the best-in-class analysis tool for Java.

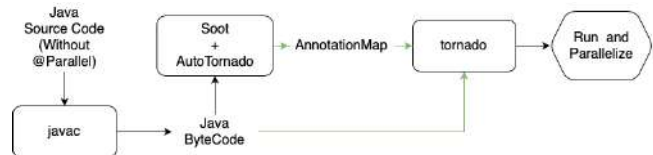

Figure 3. Workflow of AutoTornado.

An overview of the workflow can be seen in Figure 3. The Java bytecode of the unannotated program is first read by our tool `AutoTornado`, which identifies `for` loops and performs dependence analysis. For each function, it adds the

annotations of the parallelizable loops inside it in an object file `AnnotationMap`. We have modified TornadoVM to read annotations from `AnnotationMap` in addition to the ones present in the original Java bytecode. So after the analyses, our tool automatically invokes TornadoVM which runs the program with parallelized loops.

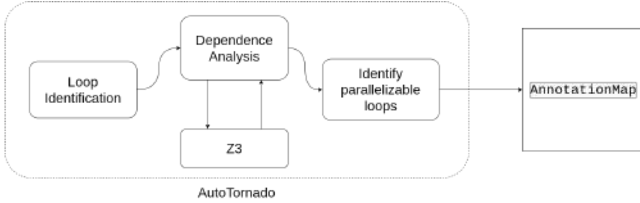

Figure 4. Architecture of AutoTornado.

`AutoTornado` comprises of three parts (see Figure 4): (i) identifying supported loops, (ii) dependence analysis, and (iii) writing information to `AnnotationMap`, all written in Soot. The first component identifies canonical `for` loops in the program that are supported by TornadoVM. The second component then performs dependence analysis on the identified loops, using the Z3 Theorem Prover [1], which we have integrated with Soot, to solve dependences across array indexes. Using this information, the third component classifies loops as parallelizable or non-parallelizable and writes this information to an object file `AnnotationMap`. This file is then used by our modified TornadoVM to parallelize the marked loops during runtime.

Dependence analysis works over the Loop-level dependence, which can occur over different kinds of memory accesses including scalar variables, object field references, and array references. The first step in this analysis is to identify the variables which are local to the loop as these can be allocated on the stack of the thread in which the iteration is running, whereas, a variable declared outside the loop may be shared among threads. As opposed to Java source code, Java bytecode does not have scopes which makes it harder to differentiate between variables declared inside and outside the loop. So to discriminate between the two, we work with liveness analysis and graph-reachability on the control-flow graph using def-use chains.

If the scalar and field variables that are not local to the loop are written to in any iteration, the loop has a dependence. Array references are more interesting as the run-time values of the array indices can sometimes be not available during static analysis. To avoid any dependences, for each writes to a location in an array, we want to make sure that the same location in the array is not accessed in a different iteration. For this, we use Z3 Theorem Prover to find a solution to the array dependence problem modeled as a Satisfiability Problem, which can be formulated as follows: "Is there a satisfying assignment for variables under the given constraints, such that the index can be the same for two different iterations?"

```
1 public void foo(int ar[]) {
2     for(int i=0; i<10000; i++) {
3         int k1 = f1(i);
4         int k2 = f2(i, k1);
5         int k3 = f3(i, k2);
6         ar[k3] = k2;
7     }
8 }
```

Figure 5. Code to demonstrate logic constraints.

As an example, Figure 5 shows a simple loop given to the program for dependence analysis. The scalar analysis marks `k1`, `k2` and `k3` as locally-scoped. Here, `f1`, `f2` and `f3` can be one of the supported operators, as mentioned above. The constraints in equation (1) are generated for the array reference at line 6 for two separate iterations represented by the superscript. If the solver returns `unsatisfiable` then there are no dependences in the loop.

$$\begin{aligned}(k3^u == f3(i^u, k2^u)) \wedge (k2^u == f2(i^u, k1^u)) \wedge \\ (k1^u == f1(i^u)) \wedge (i^u \geq 0) \wedge (i^u < 10000) \wedge \\ (k3^v == f3(i^v, k2^v)) \wedge (k2^v == f2(i^v, k1^v)) \wedge \\ (k1^v == f1(i^v)) \wedge (i^v \geq 0) \wedge (i^v < 10000) \wedge \\ (i^u \neq i^v) \wedge (k3^u == k3^v) \quad (1)\end{aligned}$$

We tested *AutoTornado* over comprehensive synthetic tests that examine its correctness in varying cases of dependence contexts. The results of the tests can be seen in Table 1.

	Parallelizable	Not Parallelizable
Total Tests	25	7
Precise & Correct	14 (56%)	7 (100%)
Unsound	0	0

Table 1. Precision and soundness of different test cases.

Our program analysis is *sound*, but *incomplete*. The same is evident from the tests. Most of the conservative behavior is from the scalar analysis, due to the problem of scope.

To conclude, our tool to identify `AutoTornado` loop-carried dependences, including dependences over stack variables, objects, as well as over array references. Most of the loops that are marked not parallelizable conservatively come from the scalar analysis. In future, we plan to work on improving the precision of the scalar analysis, which would directly improve the precision of the dependence analysis. We also plan to handle function calls inside loops.

Keywords: TornadoVM, Dependence Analysis, Static Analysis, Program Analysis, Soot, Z3, Parallelism

References

[1] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[2] Juan Fumero. 2020. TornadoVM: Accelerating Java with GPUs and FPGAs. (2020). https://www.infoq.com/articles/tornadovm-java-gpu-fpga/

[3] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press.