Leveraging RISC-V Vectorization: Accelerating Java Programs with TornadoVM and OCK

Juan Fumero[1]*, Athanasios Stratikopoulos[1], Colin Davidson[2], Harald van Dijk[2], Uwe Dolinsky[2], Michail Papadimitriou[1], Maria Xekalaki[1] and Christos Kotselidis[1]

[1]Department of Computer Science, The University of Manchester
[2]Codeplay Software Ltd., UK†

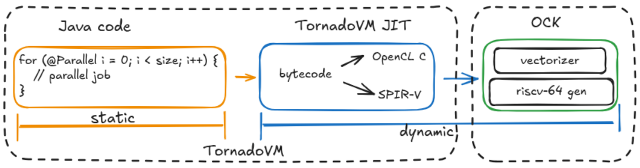
Abstract

This paper presents an approach to accelerate Java applications on RISC-V processors equipped with vector extensions. Our approach utilizes a two-stage compilation chain composed of two open-source compilation frameworks. The first compilation is performed by TornadoVM, a Java Framework that includes a Just-In-Time (JIT) compiler and a runtime system that translate Java Bytecode into OpenCL and SPIR-V. The second compilation is operated by the oneAPI Construction Kit (OCK), a programming framework that translates OpenCL and SPIR-V code into an efficient binary augmented with vector instructions for RISC-V CPUs. We also present a preliminary performance evaluation using matrix multiplication. Results demonstrate a substantial performance improvement in the code generated when compared against functionally equivalent single-threaded and multi-threaded Java implementations, achieving speedups up to 33x and 4.6x respectively.

Introduction

Java is widely used due to its extensive ecosystem for a variety of application domains. However, in some cases the performance of Java applications can be constrained due to the inability of the Java Virtual Machine (JVM) to efficiently execute computationally intensive workloads that offer high data parallelism. The JVM prioritizes stability and backward compatibility, leading, in some cases, to a slower adoption of new features, especially those related to hardware acceleration. The strict adherence to a predictable development process makes it challenging to integrate cutting-edge advancements rapidly, potentially leaving innovative use cases underserved.

This paper introduces a two-stage compilation chain for Java targeting RISC-V accelerators to enable efficient execution of Java applications on RISC-V architectures by exploiting vector instructions. Our approach leverages TornadoVM [1], an existing high-performance computing framework for Java, to target RISC-V vector units through a new integration with OCK [2], a recently developed framework for implementation of open standards[1]. We extended TornadoVM to be able to target OCK as a driver for OpenCL and SPIR-V, allowing to execute Java code on RVV 1.0 vector instructions. We demonstrate our approach



```
for (@Parallel i = 0; i < size; i++) {
   // parallel job
}
```

Figure 1: *TornadoVM/OCK JIT compiler workflow.*

through a performance evaluation centered on matrix multiplication, a fundamental operation in AI, ML and scientific computing. Our experimental results using a RISC-V SBC show that our system achieves significant speedups compared to optimized, multi-core Java implementations running on the same platform.

2-stage Compilation for RISC-V

Figure 1 illustrates the execution and compilation workflow of our enhanced system, combining TornadoVM with OCK. The workflow begins with a Java application that utilizes the TornadoVM framework, depicted on the left-hand side of the figure. During run-time, the TornadoVM JIT compiler optimizes the Java methods tagged for acceleration using Java annotations (e.g., `@Parallel`), transforming the bytecodes into either OpenCL C or SPIR-V code. Subsequently, the TornadoVM runtime dispatches the generated code to the appropriate driver implementation. In this case, it is the OCK framework, which is responsible for handling both OpenCL and SPIR-V inputs. Crucially, OCK performs automatic vectorization on the received code, ultimately producing highly optimized RISC-V vector code corresponding to the parallel regions originally expressed in Java.

*Corresponding author: `juan.fumero@manchester.ac.uk`

†No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation. ©Codeplay Software Ltd.., Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

[1] `https://github.com/uxlfoundation/oneapi-construction-kit`

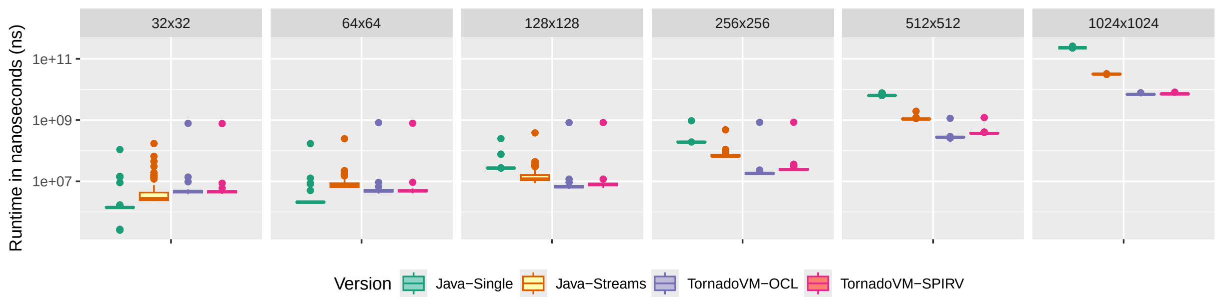


Figure 2: *Performance of MxM using TornadoVM OCK vs Java on RISC-V Spacemit K1. The lower, the better.*

Performance Evaluation

We benchmarked our approach using the canonical Matrix-Matrix Multiplication (MxM) algorithm. MxM is fundamental to a wide range of computationally intensive domains, including Artificial Intelligence (AI) and Deep Learning, making it a relevant benchmark for evaluating performance.

Hardware Our evaluation utilizes a RISC-V based Single Board Computer (SBC) [2]. This platform features an 8-core Spacemit K1 RISC-V processor clocked at 1.6 GHz, coupled with 4 GB of RAM. The system runs Bianbu OS 1.0.5, a RISC-V optimized distribution based on Ubuntu.

Software We employed TornadoVM version 1.0.10-dev (commit *ec*667*bd*65) in combination with OCK (commit 65036*b*8) as our software stack, using LLVM 19.1.5 and GCC 13.2. The underlying Java environment was OpenJDK 21.0.5.

Methodology We implemented MxM in Java using three distinct strategies: 1) a single-threaded CPU implementation, 2) a multi-threaded implementation utilizing all available CPU cores, and 3) a TornadoVM-accelerated version, leveraging OCK to exploit the RISC-V vector instructions. Each implementation was executed 100 times. The performance results are presented as box-plots, providing a comprehensive visualization of the execution time distributions for each configuration.

Analysis The x-axis in Figure 2 shows the performance of MxM on RISC-V with RVV 1.0, with the x-axis representing matrix size and the y-axis indicating execution time in nanoseconds (lower is better). The overall execution time includes the compilation time for all implementations.

For matrices up to 64x64, the sequential Java outperforms both Java parallel streams and TornadoVM. However, for larger matrices, Java streams achieve speedups of 2.3x to 7.2x over sequential Java. The TornadoVM's OpenCL C backend delivers speedups ranging from 4.1x to 33x compared to the sequential Java, and 2x to 4.6x compared to multi-core Java streams. The SPIR-V backend shows slightly lower performance than OpenCL C, with speedups of 3.4x to 32x over sequential Java and 1.4x to 4.4x over multi-core Java. Notably, for large matrices, both TornadoVM backends (OpenCL and SPIR-V) outperform all Java implementations even in this first run, which includes the JIT compilation.

Lessons learnt Although cross-compilation from an x86 to RISC-V is possible, we wanted to put this platform to the test and check its limits. The on-device compilation for the LLVM and OCK dependencies were the most critical ones. Due to the Banana Pi F3's 4GB RAM, single-threaded compilation was necessary, resulting in lengthy build times of LLVM/OCK (four days for LLVM, two for OCK). OCK compilation was further extended by iterative optimization of compiler flags for RISC-V. Besides, active cooling was also implemented to prevent overheating during compilation. Despite these challenges, successful on-device compilation demonstrates the feasibility of building the entire software stack directly on such devices.

Conclusions & Future Work

This work presented a compilation chain that can accelerate the performance of Java applications on RISC-V architectures that have vectorization support. Future work will focus on expanding the range of supported Java applications, and analyzing the performance gap between OpenCL and SPIR-V implementations.

Acknowledgement

This work is partially supported by the European Union's Horizon Europe programme under grant agreement No 101092850 (AERO), and UK Research and Innovation (UKRI) for grant numbers 10048318 and 10048915.

References

[1] Juan Fumero et al. "Dynamic application reconfiguration on heterogeneous hardware". In: VEE 2019. DOI: 10.1145/3313808.3313819. URL: https://doi.org/10.1145/3313808.3313819.

[2] Alastair Murray and Ewan Crawford. "Compute Aorta: A toolkit for implementing heterogeneous programming models". In: IWOCL '20. DOI: 10.1145/3388333.3388652. URL: https://doi.org/10.1145/3388333.3388652.

[2] https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3