Maximizing Applications Performance with GraalVM

Alina Yurenko
Developer Advocate for GraalVM
@alina_yurenko
March 03, 2020
Safe harbor statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image technology (including Substrate VM) is Early Adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.
Run programs faster anywhere

1. High performance for abstractions of any language
2. Low-footprint AOT mode for JVM-based languages
3. Convenient language interoperability and polyglot tooling
4. Simple embeddability in native and managed programs
GraalVM @graalvm · Oct 22
Which of these GraalVM supported languages interests you the most? If your answer is missing, comment it below.
GraalVM @graalvm · Oct 22
Which of these GraalVM supported languages interests you the most? If your answer is missing, comment it below.

- JavaScript: 39%
- Ruby: 12%
- Python: 43%
- R: 6%

1,009 votes · Final results
Production-ready!🎉

Pinned Tweet

GraalVM @graalvm · May 9

First production release - we are stoked to introduce GraalVM 19.0! 🚀🏆

Here's the announcement: medium.com/graalvm/announ....

Check out the release notes: graalvm.org/docs/release-n... and get the binaries:
Community Edition

GraalVM Community is available for free for evaluation, development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

DOWNLOAD FROM GITHUB

Enterprise Edition

GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

DOWNLOAD FROM OTN

get both: graalvm.org
Open Source LOC actively maintained for GraalVM

languages: 1700K

ruby: 700K  R: 460K

python: 290K  javascript: 250K

core: 1420K

compiler: 500K  truffle: 300K

substrate: 270K  sulong: 200K  other: 150K  visualvm: 520K

Total: 3,640,000 lines of code
Performance Metrics
Startup Speed

Peak Throughput

Low Memory Footprint

Reduced Max Latency

Small Packaging
Optimizing Performance with GraalVM
GraalVM

JIT
java MyMainClass

AOT
native-image MyMainClass
./mymaxclass
GraalVM Native Images

- Java program, compiled into a standalone native executable;
- Instant startup;
- Low memory footprint;
- AOT-compiled using the GraalVM compiler.
Java Dynamic Execution
Native Image Build Process

Input:
All classes from application, libraries, and VM

- Application
- Libraries
- JDK
- Substrate VM

Points-to Analysis
Run Initializations
Heap Snapshotting

Ahead-of-Time Compilation
Image Heap Writing

Output:
Native executable

Code in Text Section
Image Heap in Data Section

Iterative analysis until fixed point is reached
Startup Performance
AOT vs JIT: Startup Time

JIT
- Load JVM executable
- Load classes from file system
- Verify bytecodes
- Start interpreting
- Run static initializers
- First tier compilation (C1)
- Gather profiling feedback
- Second tier compilation (GraalVM or C2)
- Finally run with best machine code

AOT
- Load executable with prepared heap
- Immediately start with best machine code
AOT vs JIT: Memory Footprint

**JIT**
- Loaded JVM executable
- Application data
- Loaded bytecodes
- Reflection meta-data
- Code cache
- Profiling data
- JIT compiler data structures

**AOT**
- Loaded application executable
- Application data
Startup Speed
Peak Throughput
Low Memory Footprint
Reduced Max Latency
Small Packaging

AOT
JIT
Demo: startup and memory footprint
Demo: startup and memory footprint
Microservice Frameworks: Startup Time

<table>
<thead>
<tr>
<th>Framework</th>
<th>Native Image</th>
<th>JDK 12</th>
<th>JDK 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helidon</td>
<td>35 ms</td>
<td>1030 ms</td>
<td>988 ms</td>
</tr>
<tr>
<td>Micronaut</td>
<td>37 ms</td>
<td>2087 ms</td>
<td>2101 ms</td>
</tr>
<tr>
<td>Quarkus</td>
<td>16 ms</td>
<td>952 ms</td>
<td>940 ms</td>
</tr>
</tbody>
</table>
Microservice Frameworks: Memory Usage

- **Helidon**: 31 MB Native Image, 106 MB JDK 12, 116 MB JDK 8
- **Micronaut**: 41 MB Native Image, 172 MB JDK 12, 180 MB JDK 8
- **Quarkus**: 17 MB Native Image, 125 MB JDK 12, 121 MB JDK 8
Peak Performance
AOT vs JIT: Peak Throughput

JIT
- Profiling at startup enables better optimizations
- Can make optimistic assumptions about the profile and deoptimize

AOT
- Needs to handle all cases in machine code
- Predictable performance
- Profile-guided optimizations help
Profile-Guided Optimizations (PGO)

native-image --pgo-instrument → Instrumented Binary → Profiles (.iprof)

Relevant Workloads

native-image --pgo → Optimized Binary
AOT vs JIT: Peak Throughput

Handled requests per second

Cumulative number of requests

- Native Image (EE with PGO)
- Native Image (CE)
- JDK 8, Java HotSpot VM
GraalVM native image for real-world projects
Simplify Native Image Configuration

Introducing the Tracing Agent: Simplifying GraalVM Native Image Configuration

Christian Wimmer  Follow
Jun 5, 2019 · 6 min read

tl;dr: The tracing agent records behavior of a Java application running, for example, on GraalVM or any other compatible JVM, to provide the GraalVM Native Image Generator with configuration files for reflection, JNI, resource, and proxy usage. Enable it using java -agentlib:native-image-agent=...
Simplifying native-image generation with Maven plugin and embeddable configuration

In this blog post we show two features recently added to GraalVM to simplify the generation of native images. One is a Maven plugin so you can include the native image generation in your build without calling the command line utilities manually. Then we look at `native-image.properties` as a way to include the configuration for your library in the jar file to avoid manual configuration.
Micronaut

Create your first Micronaut GraalVM application:

Helidon

Helidon and GraalVM:

https://helidon.io/docs/latest/#/guides/36_graalnative
Quarkus

https://quarkus.io/guides/building-native-image
Spring Boot Applications as GraalVM Native Images

https://www.youtube.com/watch?v=3eoAxphAUIg
Spring Boot Applications as GraalVM Native Images

```
Alinas-MacBook-Pro:~/spring-graal-native/spring-graal-native-samples$ ls
commandlinerunner  spring-petclinic-jpa  vanilla-orm2
commandlinerunner-maven springmvc-tomcat  vanilla-rabbit
kotlin-webmvc       vanilla-grpc        vanilla-thymeleaf
logger              vanilla-jpa         vanilla-.tx
messages            vanilla-orm         webflux-netty
```

“Spring Graal Native” project: https://github.com/spring-projects-experimental/spring-graal-native
GraalVM Native Image vs GraalVM JIT

- Use GraalVM Native Image when
  - Startup time matters
  - Memory footprint matters
    - Small to medium-sized heaps (100 MByte – a few GByte)
  - All code is known ahead of time

- Use GraalVM JIT when
  - Heaps size is large
    - Multiple GByte – TByte heap size
  - Classes are only known at run time
GraalVM
Language Ecosystem
### Multiplicative Value-Add of GraalVM Ecosystem

<table>
<thead>
<tr>
<th>Languages</th>
<th>*</th>
<th>GraalVM</th>
<th>*</th>
<th>Embeddings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
<td>Optimizations</td>
<td></td>
<td>HotSpot JVM</td>
</tr>
<tr>
<td>JavaScript</td>
<td></td>
<td>Tooling</td>
<td></td>
<td>Oracle RDBMS</td>
</tr>
<tr>
<td>Ruby</td>
<td></td>
<td>Interoperability</td>
<td></td>
<td>Node.js</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>Security</td>
<td></td>
<td>Standalone</td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td>C/C++, FORTRAN, ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add your own language or embedding or language-agnostic tools!
JavaScript & Node.js

- ECMAScript 2019 compliant JavaScript engine;
- Access to GraalVM language interoperability and common tooling;
- Constantly tested against 90,000+ npm modules, including express, react, async, request
Compatibility Tool

Quickly check if an NPM module, Ruby gem, or R package is compatible with GraalVM.

```
express
```

**Graal.js**

<table>
<thead>
<tr>
<th>NAME</th>
<th>VERSION</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>express</td>
<td>~&gt; 5.0</td>
<td>100.00% tests pass</td>
</tr>
<tr>
<td>express</td>
<td>~&gt; 4.16</td>
<td>100.00% tests pass</td>
</tr>
<tr>
<td>express</td>
<td>~&gt; 4.15</td>
<td>100.00% tests pass</td>
</tr>
<tr>
<td>express</td>
<td>~&gt; 4.14</td>
<td>100.00% tests pass</td>
</tr>
</tbody>
</table>

React.js Server Side Rendering

- Example app: Talkyard.io
- Server-side part written in Scala, client side: React.js;
- Nashorn: ~800 renders per second;
- GraalVM: ~2000 renders per second.

Nashorn Migration Guide

Migration guide from Nashorn to GraalVM JavaScript

This document serves as migration guide for code previously targeted to the Nashorn engine. See the javainterop.md for an overview of supported Java interoperability features.

Both Nashorn and GraalVM JavaScript support a similar set of syntax and semantics for Java interoperability. The most important differences relevant for migration are listed here.

Nashorn features available by default:

- `Java.type`, `Java.typeName`
- `Java.from`, `Java.to`
- `Java.extend`, `Java.super`
- `Java package globals`: Packages, `java`, `javax`, `javafx`, `javax`, `com.org`, `edu`

Nashorn compatibility mode

GraalVM JavaScript provides a Nashorn compatibility mode. Some of the functionality necessary for Nashorn compatibility is only available when the `js.nashorn-compat` option is enabled. This is the case for Nashorn-specific extensions that GraalVM JavaScript does not want to expose by default. Note that you have to enable [experimental options](Options.md#Stable and Experimental options) to use this flag.

The `js.nashorn-compat` option can be set using a command line option:

```
$ js --experimental-options --js.nashorn-compat=true
```

[https://github.com/graalvm/graaljs/blob/master/docs/user/NashornMigrationGuide.md](https://github.com/graalvm/graaljs/blob/master/docs/user/NashornMigrationGuide.md)
JavaScript + Java + R
FastR

- GNU-R compatible R implementation
  - Including the C/Fortran interface
- Built on top of the GraalVM platform
  - Leverages GraalVM optimizing compiler
  - Integration with GraalVM dev tools
  - Zero overhead interop with other GraalVM languages

warm-up curves, i.e. lower is better, of rJava on GNU-R, FastR and the native Java interoperability in FastR
GraalVM Python

- Python 3 implementation;
- High performance;
- Focus on supporting SciPy and its constituent libraries;
- Easy interop with Java and the rest of GraalVM languages.

```
$ graalpython [options] [-c cmd | filename]
```
Do even more with GraalVM: Cross-Platform Development
Polyglot tools: GraalVM VisualVM
Using grCUDA to Access Nvidia GPUs

- Efficient exchange of data between host language and GPU without burdening the programmer
- Expose GPU resources in ways that are native in the host language, e.g., as arrays
- Allow programmers to invoke existing GPU code from their host language
- Allow programmers to define new GPU kernels on the fly
- Polyglot interface: uniform bindings across several programming languages

- Implemented as a “Truffle Language” (although “CUDA” is a platform, not a language)
- Developed by NVIDIA in collaboration with Oracle Labs
- BSD 3-clause license
Twitter uses GraalVM compiler in production to run their Scala microservices
GraalVM in practice at the Dutch National Police

https://www.youtube.com/watch?v=poNlwZjoYjs
• Peak performance: +10%
• Garbage collection time: -25%
• Seamless migration

Cloud Infrastructure
The rich ecosystem of CUDA-X libraries is now available for GraalVM applications.

GPU kernels can be directly launched from GraalVM languages such as R, JavaScript, Scala and other JVM-based languages.

Learn more: https://devblogs.nvidia.com/grcuda-a-polyglot-language-binding-for-cuda-in-graalvm/
Odnoklassniki use GraalVM in a production Java workload (70 mln users, ~600K req/min, ~7K servers) for React server-side rendering

Project Roadmap
Version Roadmap

• Latest release: GraalVM 20.0;
  • includes JDK8- and JDK11-based builds;
• Predictable release schedule;
• LTS releases: last major release of the year.

https://www.graalvm.org/docs/release-notes/version-roadmap
Recent Updates

- JDK-11 based builds;
- Extended Windows support;
- New GC option in native image: try it with `-H:+UseLowLatencyGC`;
- WebAssembly support;
- Support for JFR in GraalVM VisualVM;
- VS Code plugin preview;
- Class Initialization changes in native images.
Contributions are welcome!

• How to contribute:

• Report an issue: https://github.com/oracle/graal/issues

• Submit your PR: https://github.com/oracle/graal/pulls

• Extend libraries support: graalvm.org/docs/reference-manual/compatibility/

• Contribute to documentation: https://www.graalvm.org/docs/
When to consider GraalVM

1. High performance for abstractions of any language
2. Low footprint ahead-of-time mode for JVM-based languages
3. Convenient language interoperability and polyglot tooling
4. Simple embeddability in native and managed programs
Where to Get Started

- Download: graalvm.org/downloads
- Follow updates: @GraalVM / #GraalVM
- Get help:
  - graalvm.org/slack-invitation/
  - graalvm-users @oss.oracle.com
Thank you!

Alina Yurenko
@alina_yurenko