Maximizing Performance with GraalVM

Thomas Wuerthinger (@thomaswue)
Senior Research Director
Oracle Labs
June 25, 2019
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™
Community Edition (CE)

GraalVM CE is available for free for development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for GraalVM CE for Linux on x86 64-bit systems.

DOWNLOAD FROM GITHUB

Enterprise Edition (EE)

GraalVM EE provides additional performance, security, and scalability relevant for running critical applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for GraalVM EE for Linux or Mac OS X on x86 64-bit systems.

DOWNLOAD FROM OTN
GraalVM™

JIT

copyright ©

AOT

native-image MyMainClass

./mymainclass

java MyMainClass

OpenJDK™

Scala

Groovy

Kotlin

Java™
GraalVM AOT for Native Images

Input: All classes from application, libraries, and VM

- Application
- Libraries
- JDK
- Substrate VM

Points-to Analysis
Run Initializations
Heap Snapshotting
Iterative analysis until fixed point is reached

Ahead-of-Time Compilation
Code in Text Section
Image Heap Writing
Image Heap in Data Section

Output: Native executable

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.
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: Startup Time

- **Helidon**: GraalVM AOT: 35 ms, GraalVM JIT: 988 ms
- **Micronaut**: GraalVM AOT: 37 ms, GraalVM JIT: 2101 ms
- **Quarkus**: GraalVM AOT: 16 ms, GraalVM JIT: 940 ms
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
AOT vs JIT: Memory Footprint

- **Helidon**: GraalVM AOT - 106 MB, GraalVM JIT - 31 MB
- **Micronaut**: GraalVM AOT - 180 MB, GraalVM JIT - 41 MB
- **Quarkus**: GraalVM AOT - 121 MB, GraalVM JIT - 17 MB
Web Server Startup and Memory Footprint

Starting up and serving 2 requests in the first 10s

- **JIT 800ms / 350Mb**

- **AOT 8ms / 13Mb**
Which is fastest?

```java
int negate1(int a) {
    return -a;
}

int negate2(int a) {
    int b = a + 0;
    return -b * 1;
}

int negate3(int a) {
    Object[] array = new Object[] {Integer.valueOf(a)};
    return -(Integer)array[0];
}

static Object[] cachedArray = new Object[1];
int negate4(int a) {
    cachedArray[0] = Integer.valueOf(a);
    return -(Integer)cachedArray[0];
}
```
Performance is hard to measure
AOT vs JIT: Throughput

Handled requests per second

Cumulative number of requests sent by ApacheBench

GraalVM AOT
JDK 8, Java HotSpot VM
Profile-Guided Optimizations (PGO)

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

Relevant Workloads → native-image --pgo
AOT vs JIT: Throughput

Handled requests per second

Cumulative number of requests sent by ApacheBench

- Red: GraalVM AOT with PGO
- Orange: GraalVM AOT
- Blue: JDK 8, Java HotSpot VM
- Black: Native Image (EE with PGO)
- Gray: Native Image (CE)
AOT vs JIT: Peak Performance

• JIT
  – Profiling at startup enabled better optimizations
  – Can make optimistic assumptions about the profile and deoptimize

• AOT
  – Needs to handle all cases in machine code
  – Profile-guided optimizations help
  – Predictable performance
More Benchmarks...

• Optimizing a compiler for too few benchmarks results in typical overfitting problems

• Therefore we started together with academic collaborators https://renaissance.dev

• All benchmark data can be interesting; careful with conclusions though.
AOT vs JIT: Max Latency

• JIT
  – Many low latency GC options available
    • G1
    • CMS
    • ZGC
    • Shenandoah

• AOT
  – Only regular stop&copy collector
  – Assumes small heap configuration
  – Can quickly restart; could use load balancer instead of GC

• Peak vs max latency trade-offs:
  – Loop safepoints
  – Parallel stop-the-world GC
AOT vs JIT: Packaging Size

• JIT
  – Use jlink for smaller package
  – Lightweight docker image (e.g., alpine linux)

• AOT
  – Everything in single binary
  – Can run on bare metal docker
  – Substantially smaller constant overhead

• Peak vs packaging trade-offs:
  – Inlining
  – Code duplication
GraalVM JIT

Peak Throughput
Max Latency
No Configuration

GraalVM AOT

Startup Time
Memory Footprint
Packaging Size

Can AOT get better?

– Collecting profiles up-front
– Low-latency GC option
– Tracing agent for configuration
GraalVM can do much more...

```javascript
const express = require('express');
const app = express();
app.listen(3000);
app.get('/', function(req, res) {
    var text = 'Hello World!';
    const BigInteger = Java.type('java.math.BigInteger');
    text += BigInteger.valueOf(2).pow(100).toString(16);
    text += Polyglot.eval('R', 'runif(100)')[0];
    res.send(text);
});
```
## Multiplicative Value-Add of GraalVM Ecosystem

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

* Add your own language or embedding or language-agnostic tools!
GraalVM Community

- [https://www.graalvm.org](https://www.graalvm.org)
- Open source on GitHub at [https://github.com/oracle/graal](https://github.com/oracle/graal)
Q/A

@graalvm

@thomaswue
Integrated Cloud
Applications & Platform Services