Wirklich so schnell?

Performance von Native Images auf der GraalVM™

Startup, Laufzeit, Speicherverbrauch
von GraalVM Native Images

MARTIN LEHMANN

JAVA FORUM STUTTGART, SEPTEMBER 2021

@mrtnlhmnn

v.3.18
v.3

Martin Lehmann
Accso - Accelerated Solutions GmbH

Cheftechnologe

Martin Lehmann ist Diplom-Informatiker und als Cheftechnologe und Softwarearchitekt bei der Accso – Accelerated Solutions GmbH tätig. Seit Ende der 90er-Jahre arbeitet er als Softwareentwickler und -architekt in der Softwareentwicklung in diversen Projekten der Individualentwicklung für Kunden verschiedener Branchen. Seit den Zeiten von Java 1.0 beschäfigt er sich mit Java als Programmiersprache und als Ökosystem.

martin.lehmann@accso.de @mrtnlhmnn xing.to/mle

GraalVM™

- **Einführung: Graal? Native-Image!**
- Performance: Wie schnell ist das wirklich?
- Vorsicht: Laufzeitfehler bei Reflection & Co
- (Wie) testet man Native-Images?

GraalVM™ im Überblick

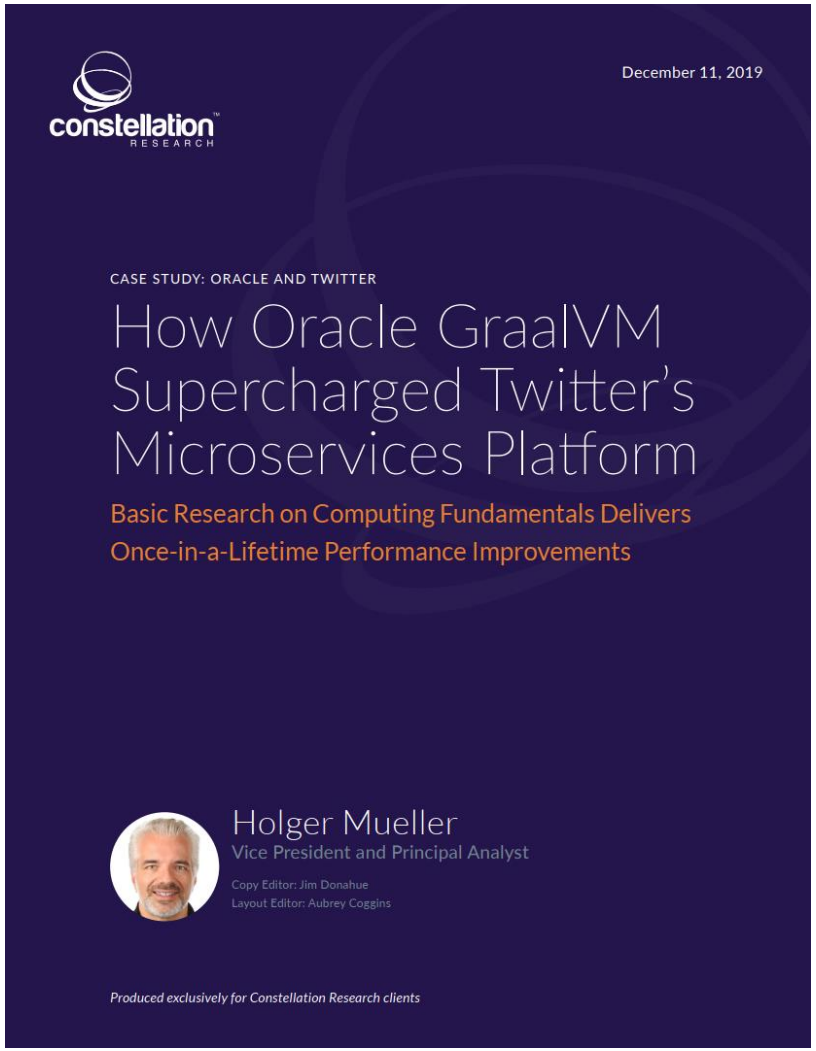
Graal und GraalVM

- General Recursive Applicative and Algorithmic Language
- **GraalVM** = Universal Virtual Machine für Anwendungen
  - von JVM-basierten Sprachen wie Java, Scala, Clojure, Kotlin, …
  - von JavaScript, Python, Ruby, R
  - von LLVM-basierte Sprachen wie C und C++
- Anwendungen entweder im **JVM-Modus** oder als **Native-Images**

Versionen

- Aktuell ist Version 21.2.0
- Community (CE) und Enterprise Edition (EE) für Java8, 11, 16
- http://graalvm.org
  https://github.com/oracle/graal
- Lizenzen siehe Übersicht auf Github
- https://www.graalvm.org/reference-manual/native-image/
  https://www.graalvm.org/examples/native-image-examples/

Twitter nutzt Graal schon produktiv



Not surprisingly, the owners were concerned about changing compilers, but after some prodding, a few gave it a try. As they showed positive results, more and more microservice owners have switched over to Oracle GraalVM, to the point that most of Twitter runs on Oracle GraalVM today.

Of course, Twitter started with smaller microservices to test the performance and stability of Oracle GraalVM but quickly gained the confidence to move the quintessential Tweet microservices to Oracle GraalVM. As of summer 2019, between 30 and 40 major services of Twitter—for instance, Tweet, Social and News—have been running in production powered by the Oracle GraalVM compiler.

"Being able to see 8–11 percent performance improvements on the same hardware without having to change the underlying code is a once-in-a-lifetime event, leading to substantial cost savings and future flexibility for Twitter."

—Chris Thalinger,
Staff Engineer, Twitter

https://www.constellationr.com/research/how-oracle-graal-supercharged-twitter-s-microservices-platform

GraalVM™ baut Native-Images, nutzt SubstrateVM

Ahead-of-Time Compile AOT

Gebaut wird ein **Standalone Executable**.

Alternativ kann man auch eine **Shared Library** erstellen.

Closed-world assumption:

- → Alle Klassen, alles an Bytecode muss zur Build-Zeit bekannt sein.
- → Optimierungen nur zur Build-Zeit, nicht mehr zur Laufzeit.
- → Cross-Compile ist z.Zt. nicht möglich.

Was wird mit verpackt?

- → Alle Anwendungsklassen über alle transitiven Abhängigkeiten, ausgehend von einer Main-Class

- → Native-Image benötigt zur Laufzeit keine (separate) JVM, stattdessen:
- → **SubstrateVM** = die Runtime, in der AOT-kompilierter Code ohne (separate) JVM ausgeführt wird
  - → Memory Management
  - → Thread Scheduling
  - → Garbage Collector (Default: Serial GC)

Alle Beispiele sind auf Github

Alle Beispiele auf Github: https://github.com/accso/graalvm-native

In diesen Folien (Stand: Juli 2021) benutzen und vermessen wir:

- Windows 10 mit WSL2 (Linux/Ubuntu)
- GraalVM 21.1.0 Community Edition (CE) für Java 11
- GraalVM 21.1.0 Enterprise Edition (EE) für Java 11
- Hotspot AdoptOpenJDK 11.0.9+11 und 16.0.1+9

20.3 **21.1** **21.2**	**CE** **EE**	**Java 8** **Java 11** **Java 16**	**JVM-Modus** **Native-Image**	**Build- und Laufzeit-Optimierungen**

Nicht alle Beispiele funktionieren in allen Varianten

Beispiel „mandelbrot“

- Linker-Fehler mit 21.1.0 bei JDK 16 statt JDK 11

```
Fatal error:java.lang.RuntimeException: java.lang.RuntimeException: There was an
error linking the native image: Linker command exited with 1

/usr/bin/ld: /opt/graalvm-ee-java16-21.1.0/lib/static/linux-
amd64/glibc/libjavajpeg.a(jpegdecoder.o):(.bss.jvm+0x0): multiple definition of `jvm'; /opt/graalvm-
ee-java16-21.1.0/lib/static/linux-amd64/glibc/libawt.a(awt_LoadLibrary.o):(.bss.jvm+0x0): first
defined here
collect2: error: ld returned 1 exit status
```

Beispiel „quarkus-timeserver“

- PGO-Optimierung führt zu Segfault unter 21.1.0/11/EE

```
[ [ SubstrateSegfaultHandler caught a segfault. ] ]
```

Kurzer Umgebungscheck

```
lehmann@edwards03:~/graal/graalvm-native $
$ echo $GRAALVM_HOME
/opt/graalvm-ce-java11-21.1.0

lehmann@edwards03:~/graal/graalvm-native $
$ $GRAALVM_HOME/bin/java -version
openjdk version "11.0.11" 2021-04-20
OpenJDK Runtime Environment GraalVM CE 21.1.0 (build 11.0.11+8-jvmci-21.1-b05)
OpenJDK 64-Bit Server VM GraalVM CE 21.1.0 (build 11.0.11+8-jvmci-21.1-b05, mixed
mode, sharing)

lehmann@edwards03:~/graal/graalvm-native $
$ $GRAALVM_HOME/bin/gu list
…
native-image          21.1.0          Native Image          Early adopter
```

Beispiel „HelloWorld“ und Build-Skript

HelloWorld.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

build.sh

```
SRC=./src
TARGET=./target

mkdir -p ${TARGET}
${GRAALVM_HOME}/bin/javac -d ${TARGET} ${SRC}/HelloWorld.java
…

${GRAALVM_HOME}/bin/native-image --no-fallback \
   -H:+PrintAnalysisCallTree \
   -H:+ReportExceptionStackTraces \
   HelloWorld helloworld
```

`native-image` baut aus .class/.jar-Datei das Native-Image als Binary Executable

Native-Images kann man auch mit Maven erzeugen

```
<!-- Native Image -->
<plugin>
    <groupId>org.graalvm.nativeimage</groupId>
    <artifactId>native-image-maven-plugin</artifactId>
    <version>${graalvm.version}</version>
    <executions>
        <execution>
            <goals>
                <goal>native-image</goal>
            </goals>
            <phase>package</phase>
        </execution>
    </executions>
    <configuration>
        <!-- Set this to true if you need to switch this off -->
        <skip>false</skip>
        <!-- The output name for the executable -->
        <imageName>${exe.inMemory.file.name}</imageName>
        <!-- Set any parameters we need to pass to the native-image tool.
            no-fallback : create a native image that doesn't fall back to the JVM
            no-server : don't start a build server, which you then just need to shut down,
                        in order to build the image
        -->
        <buildArgs>
            --no-server --no-fallback -H:+PrintAnalysisCallTree -H:+ReportExceptionStackTraces
        </buildArgs>
    </configuration>
</plugin>
```

Beispiel „HelloWorld“: Builds sind sehr langsam

```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ bash ./build.sh
[helloworld:24685]    classlist:   2,843.83 ms,  0.96 GB
[helloworld:24685]        (cap):   1,030.44 ms,  0.96 GB
[helloworld:24685]        setup:   3,506.13 ms,  0.96 GB
[helloworld:24685]     (clinit):     280.51 ms,  1.72 GB
[helloworld:24685]   (typeflow):   9,981.97 ms,  1.72 GB
[helloworld:24685]    (objects):  12,530.03 ms,  1.72 GB
[helloworld:24685]   (features):     498.57 ms,  1.72 GB
[helloworld:24685]     analysis:  23,580.20 ms,  1.72 GB
…
[helloworld:24685]     universe:     529.21 ms,  1.72 GB
[helloworld:24685]      (parse):   4,522.41 ms,  2.29 GB
[helloworld:24685]     (inline):   4,935.98 ms,  2.29 GB
[helloworld:24685]    (compile):  25,480.25 ms,  2.35 GB
[helloworld:24685]      compile:  35,802.62 ms,  2.35 GB
[helloworld:24685]        image:   2,167.79 ms,  2.35 GB
[helloworld:24685]        write:     334.64 ms,  2.35 GB
…
[helloworld:24685]      [total]:  71,017.16 ms,  2.35 GB
```

build-log (gekürzt)

Beim Bauen können ausführliche Reports erzeugt werden, u.a. für

- Call Tree
- Used Classes
- Uses Packages
- Used Methods

native-image hat > 1000 Experten-Optionen

```
lehmann@edwards03:~/graal/graalvm-native $
$ $GRAALVM_HOME/bin/native-image --expert-options-all | grep -i print
…
... 88 verschiedene Argumente ...
```

```
lehmann@edwards03:~/graal/graalvm-native $
$ $GRAALVM_HOME/bin/native-image --expert-options-all | grep -i optimiz
…
... 40 verschiedene Argumente ...,
z.B.

-H:Optimize=2          Control native-image code optimizations:
                       0 - no optimizations,
                       1 - basic optimizations,
                       2 - aggressive optimizations
```

Vermutlich ist „aggressive“ der Default.

Native-Image: Start, Größe, File-Type

```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ ./target/helloworld
Hello, World!

lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ ls -laF target/*
-rw-r--r-- 1 lehmann lehmann  427 Jun 26 14:44 target/HelloWorld.class
-rwxr-xr-x 1 lehmann lehmann 9.4M Jun 26 14:45 target/helloworld*

lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ file ./target/helloworld
target/helloworld: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=2c3d598b5e35dce07d07750581f730ad192c8fbf, for GNU/Linux 3.2.0,
with debug_info, not stripped
```

Native-Image: Welche Java-Version?

```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ strings ./target/helloworld | grep com.oracle.svm.core.VM

com.oracle.svm.core.VM.Target.StaticLibraries=liblibchelper.a|libnet.a|libnio.a
|libjava.a|libfdlibm.a|libzip.a|libjvm.a
com.oracle.svm.core.VM.Target.Libraries=pthread|dl|z|rt
com.oracle.svm.core.VM.Target.LibC=com.oracle.svm.core.posix.linux.libc.GLibC
com.oracle.svm.core.VM.Target.CCompiler=gcc|linux|x86_64|9.3.0
com.oracle.svm.core.VM=GraalVM 21.1.0 Java 11 CE
com.oracle.svm.core.VM.Target.Platform=org.graalvm.nativeimage.Platform$LINUX_A
MD64
```

GraalVM™

- Einführung: Graal? Native-Image!
- **Performance: Wie schnell ist das wirklich?**
- Vorsicht: Laufzeitfehler bei Reflection & Co
- (Wie) testet man Native-Images?

„Performance“ von Native-Images?

Der übliche Disclaimer:
„Wer viel misst, misst viel Mist.“

Startup-Zeit

Memory

Laufzeit

Startup-Zeiten von „helloworld“

```
time java -cp target HelloWorld                                  run.sh
time ./target/helloworld                          (run-loop*.sh nicht gezeigt)
```

```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ . ./run-loop_graal-jvm.sh 100
...
real    0m10.123s
user    0m8.037s
sys     0m1.719s

lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ . ./run-loop_graal-native.sh 100
...
real    0m0.122s
user    0m0.103s
sys     0m0.026s
```

Startup des Native-Images ist signifikant schneller.

Der übliche Disclaimer:
„Wer viel misst, misst viel Mist.“

Startup-Zeit

- → Native-Image ist signifikant schneller im Startup.
- → In HelloWorld-Beispiel mit Faktor >> 80!
- → Da alles in das Native-Image verpackt ist, muss insbesondere keine JVM hochfahren.

Memory

Laufzeit

Native-Images haben ein optimiertes Memory-Footprint.

Der übliche Disclaimer:
„Wer viel misst, misst viel Mist.“

Startup-Zeit

- → Native-Image ist signifikant schneller im Startup.
- → In HelloWorld-Beispiel mit Faktor >> 70!
- → Da alles in das Native-Image verpackt ist, muss insbesondere keine JVM hochfahren.

Memory

- → Memory-Footprint am Beispiel 1) „helloworld“
- → Memory-Footprint am Beispiel 2) „mandelbrot“

Laufzeit

Memory von Beispiel 1 „helloworld“

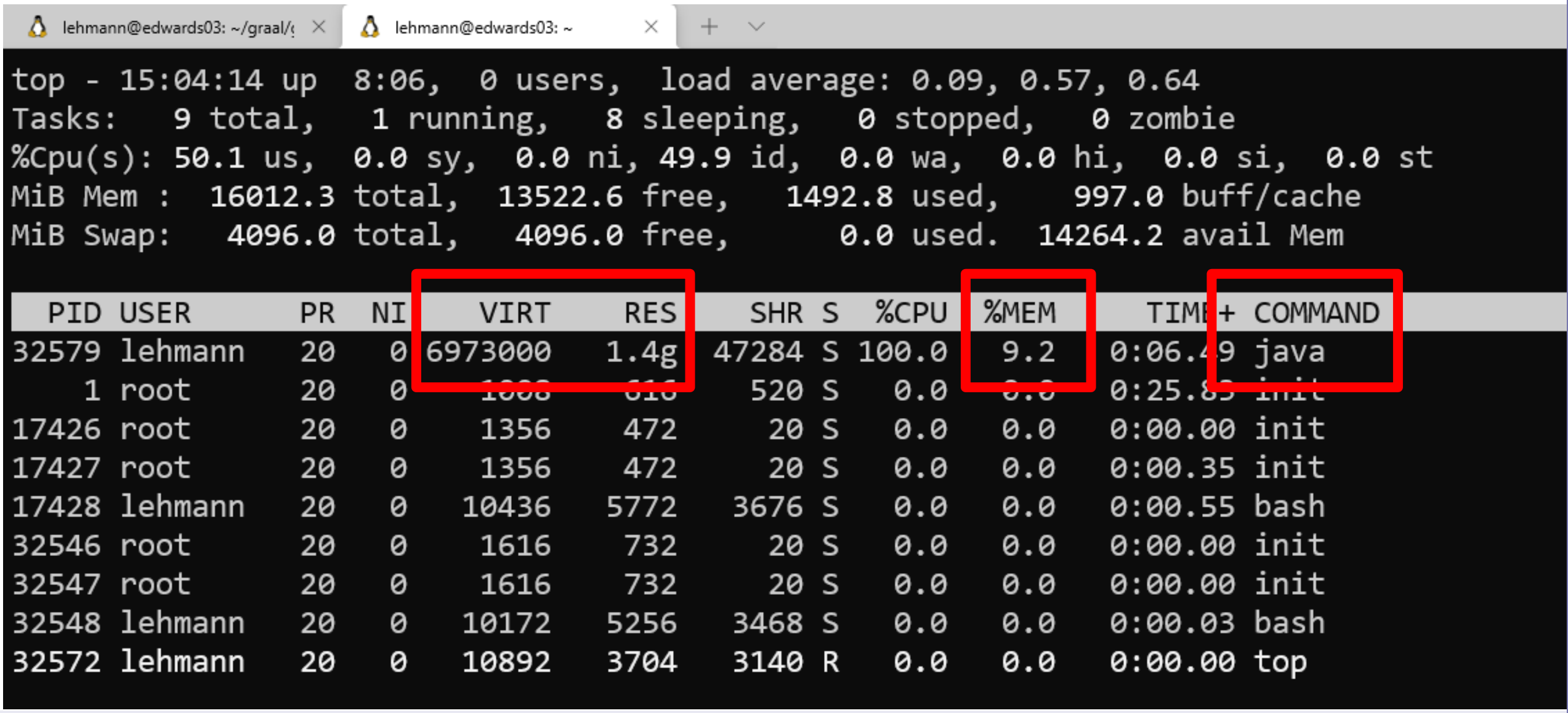
```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ /usr/bin/time --format %M java -cp target/ HelloWorld
66596
```


```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ /usr/bin/time --format %M ./target/helloworld
6852
```


```
lehmann@edwards03:~/graal/graalvm-native/helloworld $
$ man time
...
M      Maximum resident set size of the process during its lifetime,
       in Kilobytes.
```

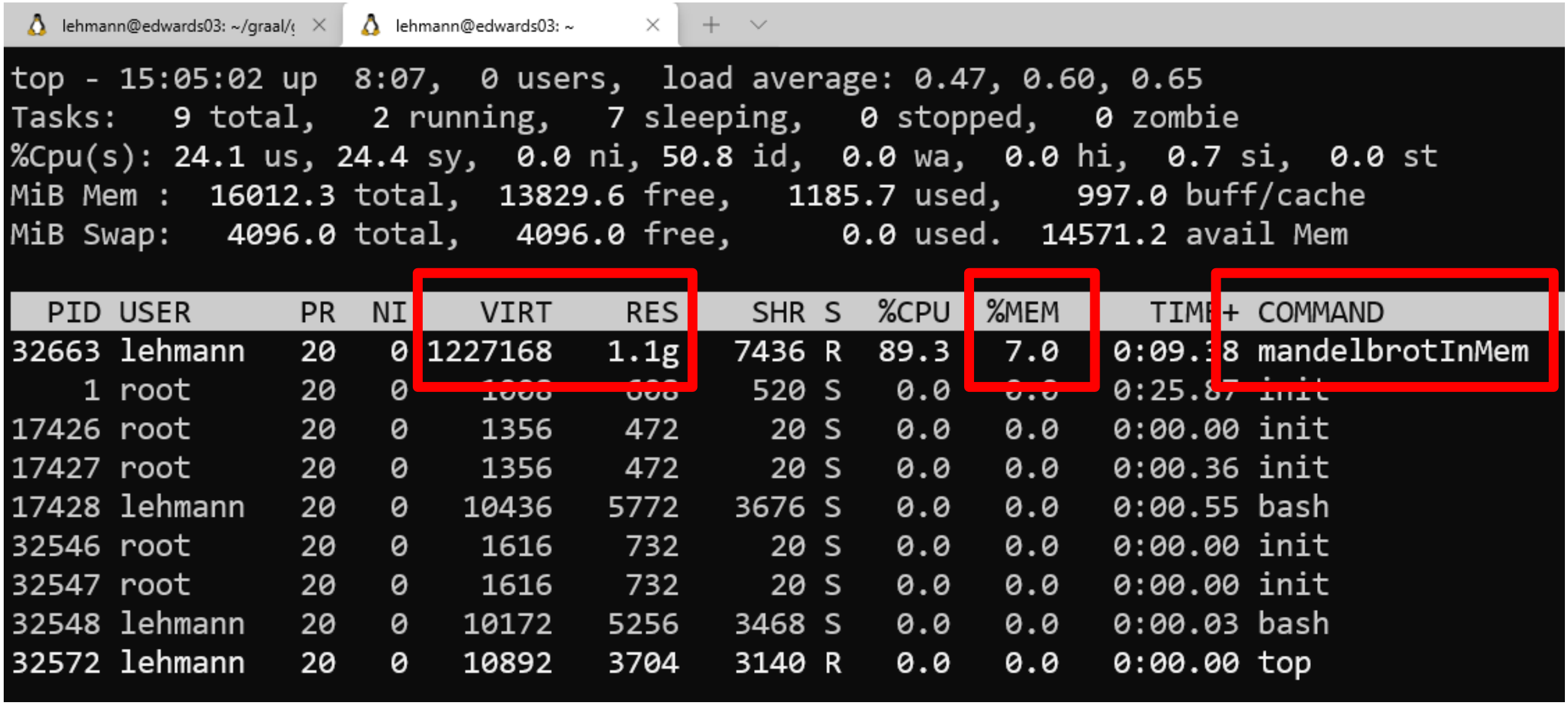
Memory von Beispiel 2 „mandelbrot“

JVM



Memory von Beispiel 2 „mandelbrot“

native



Auf lange Sicht ist die JVM durch Just-in-time-Optimierungen schneller als das Native-Images.

Der übliche Disclaimer:
„Wer viel misst, misst viel Mist.“

Startup-Zeit

- → Native-Image ist signifikant schneller im Startup.
- → In HelloWorld-Beispiel mit Faktor >> 70!
- → Da alles in das Native-Image verpackt ist, muss insbesondere keine JVM hochfahren.

Memory

- → Memory-Footprint am Beispiel 1) „helloworld“
- → Memory-Footprint am Beispiel 2) „mandelbrot“

Laufzeit

- → Performance am Beispiel 1) „filefinder“
- → Performance am Beispiel 2) „mandelbrot“
- → Durchsatz am Beispiel 3) „quarkus-timeserver“

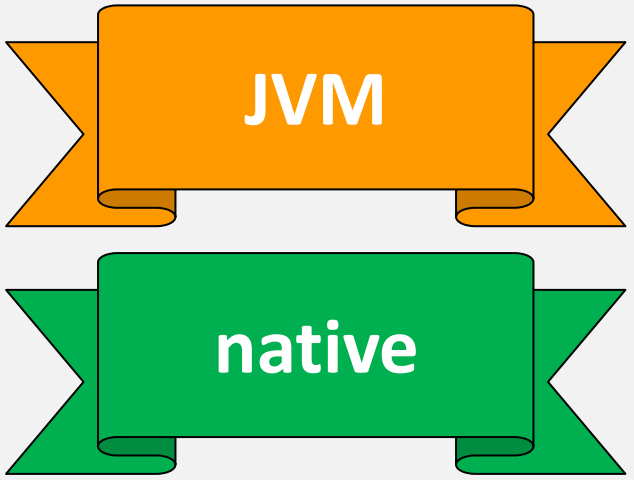
Laufzeit für Beispiel 1) „filefinder“

```
time java -cp target FileFinder ${DIR} -name ${PATTERN}
time ./target/filefinder        ${DIR} -name ${PATTERN}
time ./target/filefinderWithJS  ${DIR} -name ${PATTERN}
```
run.sh

```
lehmann@edwards03:~/graal/graalvm-native/filefinder $
$ bash ./run.sh /opt "*.py" 2>&1 | grep real

real    0m0.824s          // Graal/JVM-Modus


real    0m0.275s          // Graal/Native-Image
real    0m0.325s          // Graal/Native-Image
                          //   mit JavaScript
```

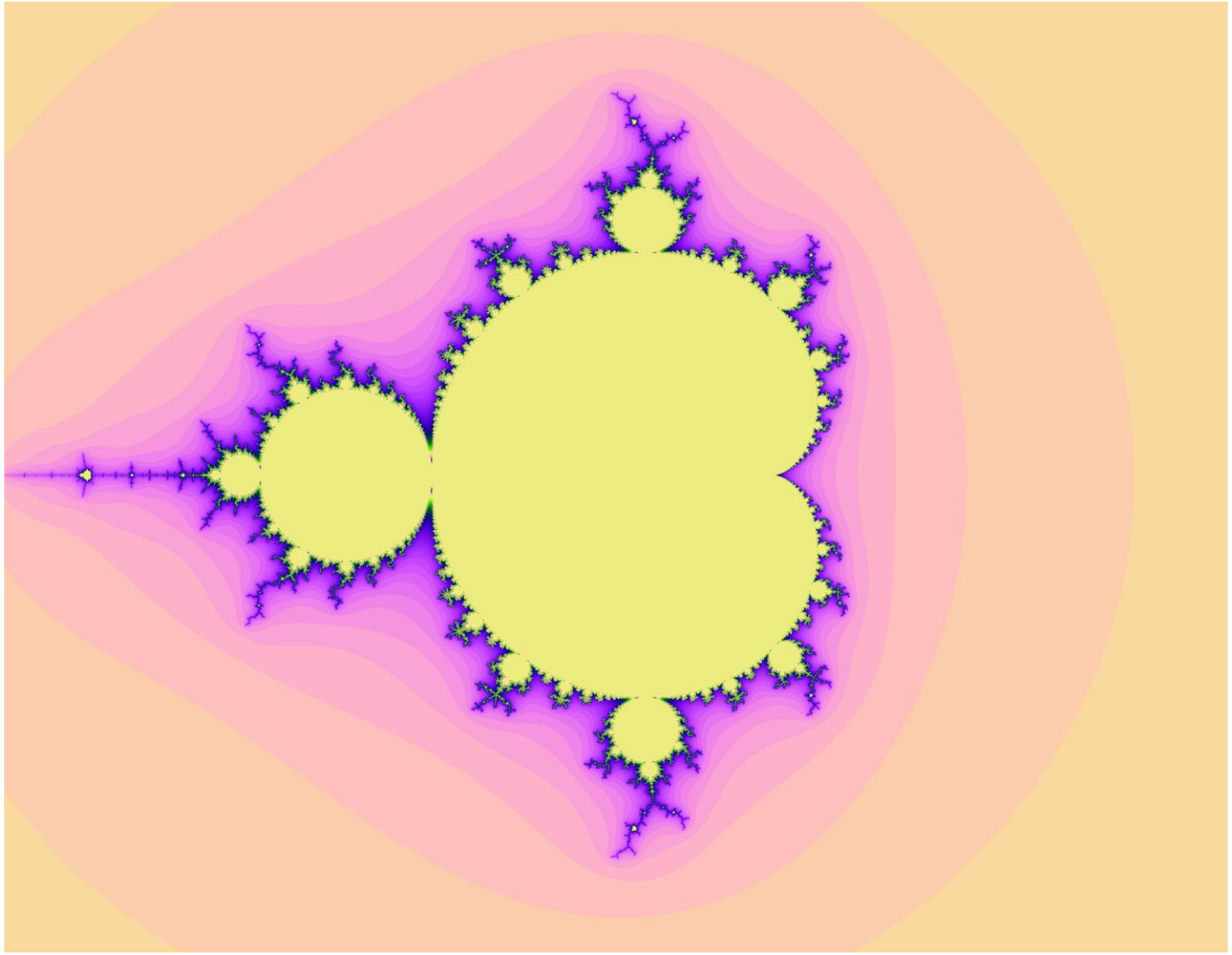


Laufzeit für das Native-Image im Vergleich zum JVM-Modus sehr gut – aber Startup dominiert.

Laufzeit für Beispiel 2) „mandelbrot“

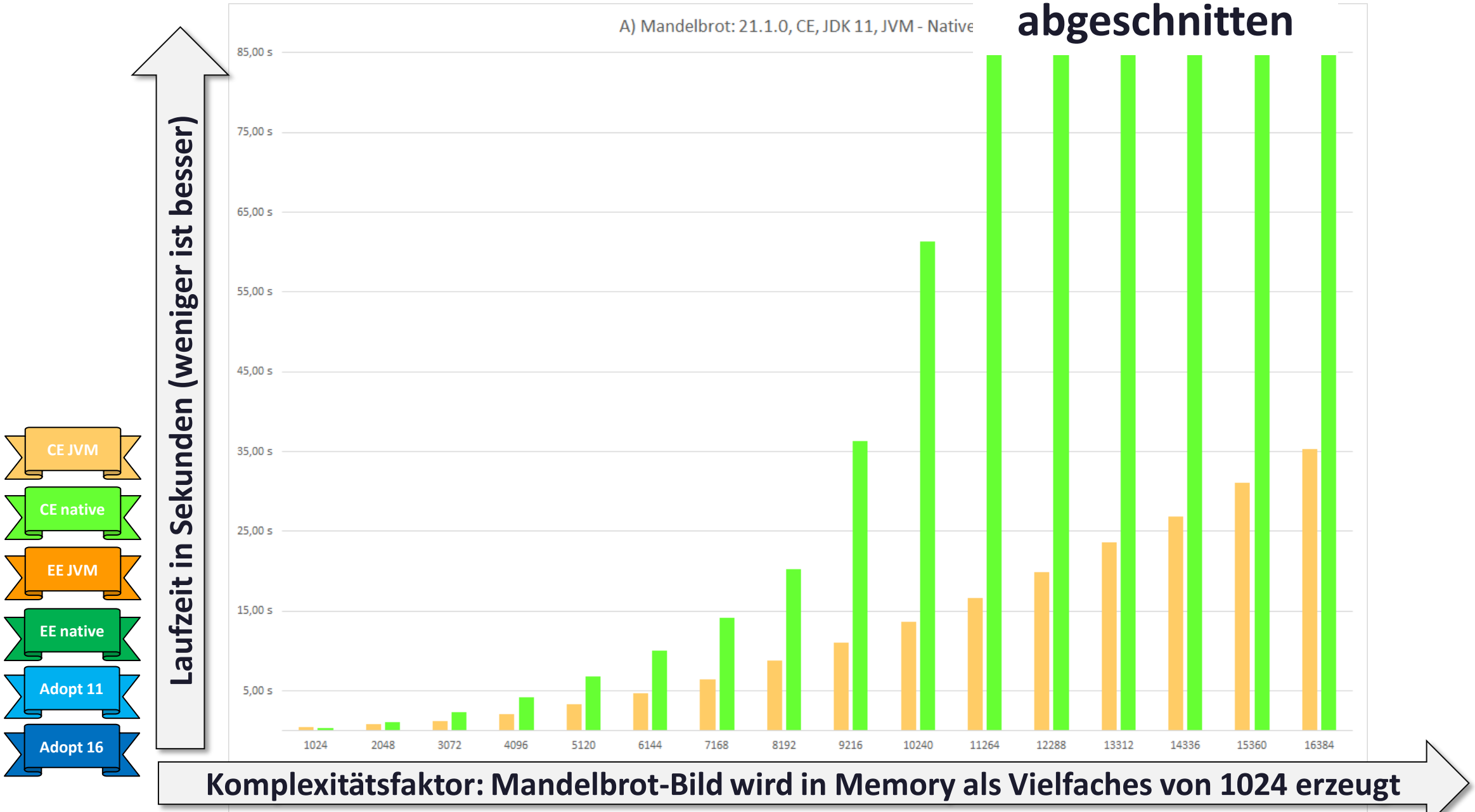
- Apfelmännchen wird in PNG-Datei erzeugt
- nutzt ImageIO und AWT

- Alle nachfolgenden Performance-Messungen aber nur „in Memory“ mit Bild-Puffer



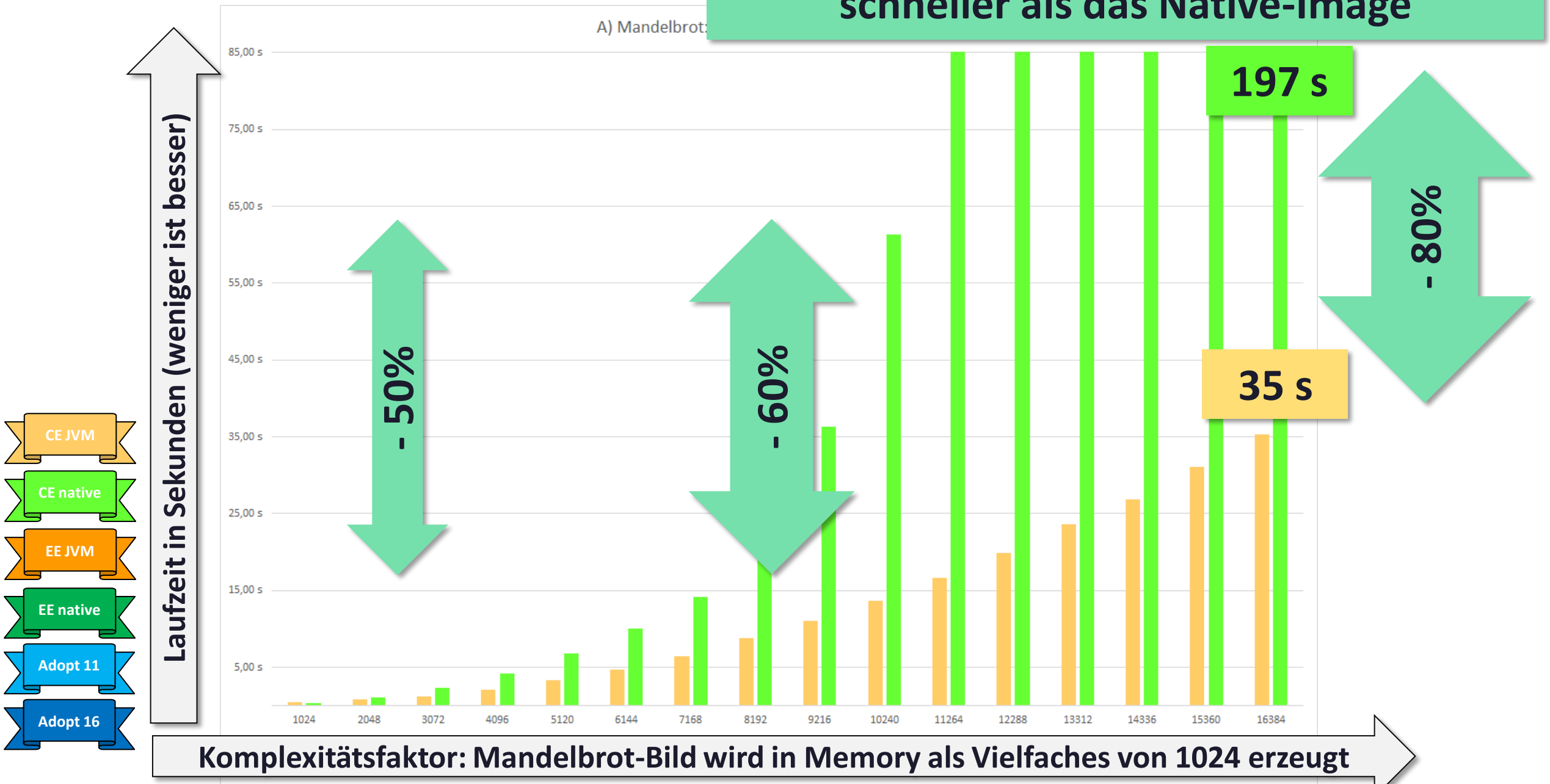
Laufzeit für Beispiel 2) „mandelbrot“

Laufzeit für Beispiel 2) „mandelbrot“



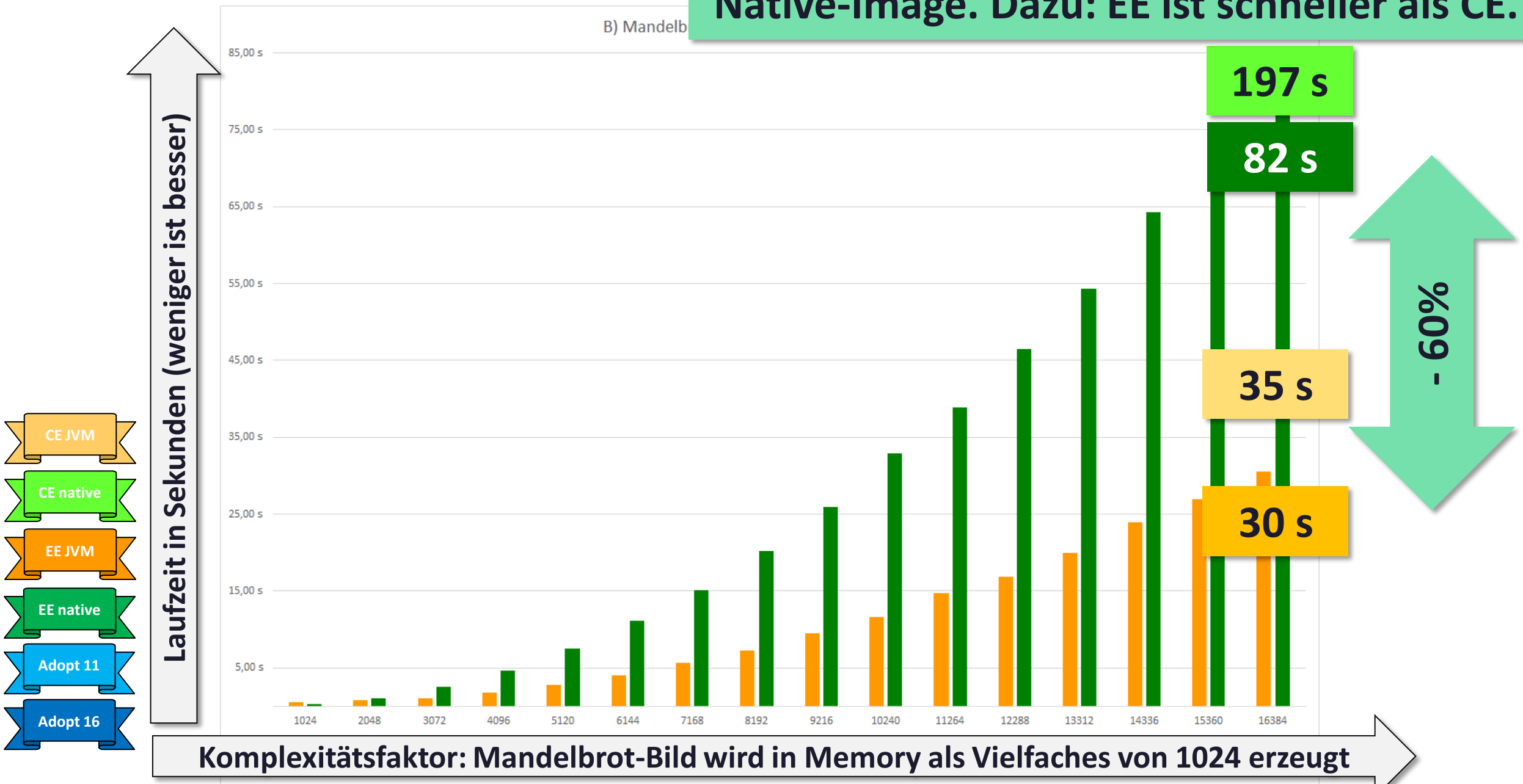
Laufzeit „mandelbrot“

Graal 21.1.0 CE: JVM-Modus ist signifikant schneller als das Native-Image



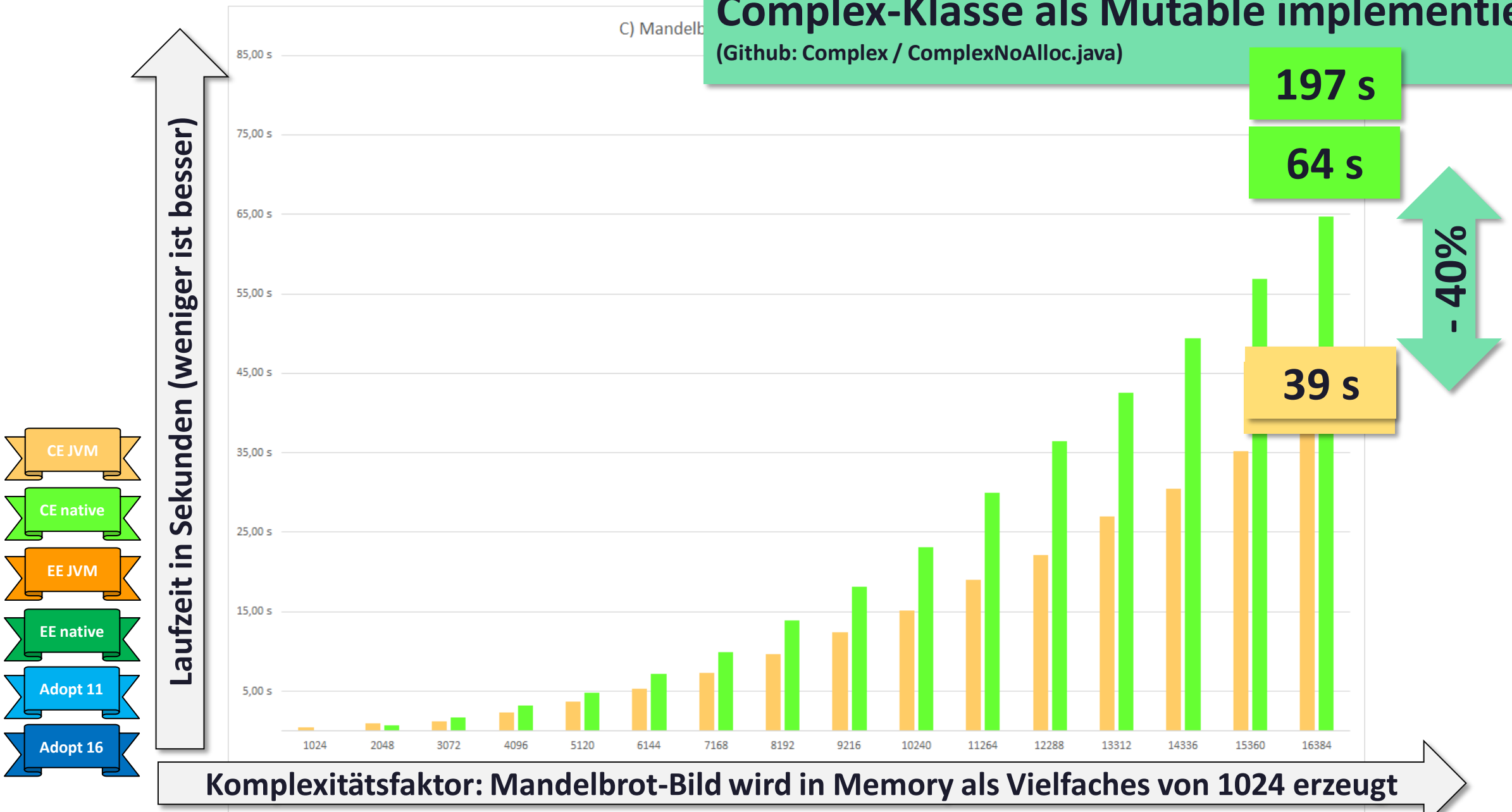
Laufzeit „mandelbrot“

Ähnliches Ergebnis bei EE, JVM schneller als Native-Image. Dazu: EE ist schneller als CE.



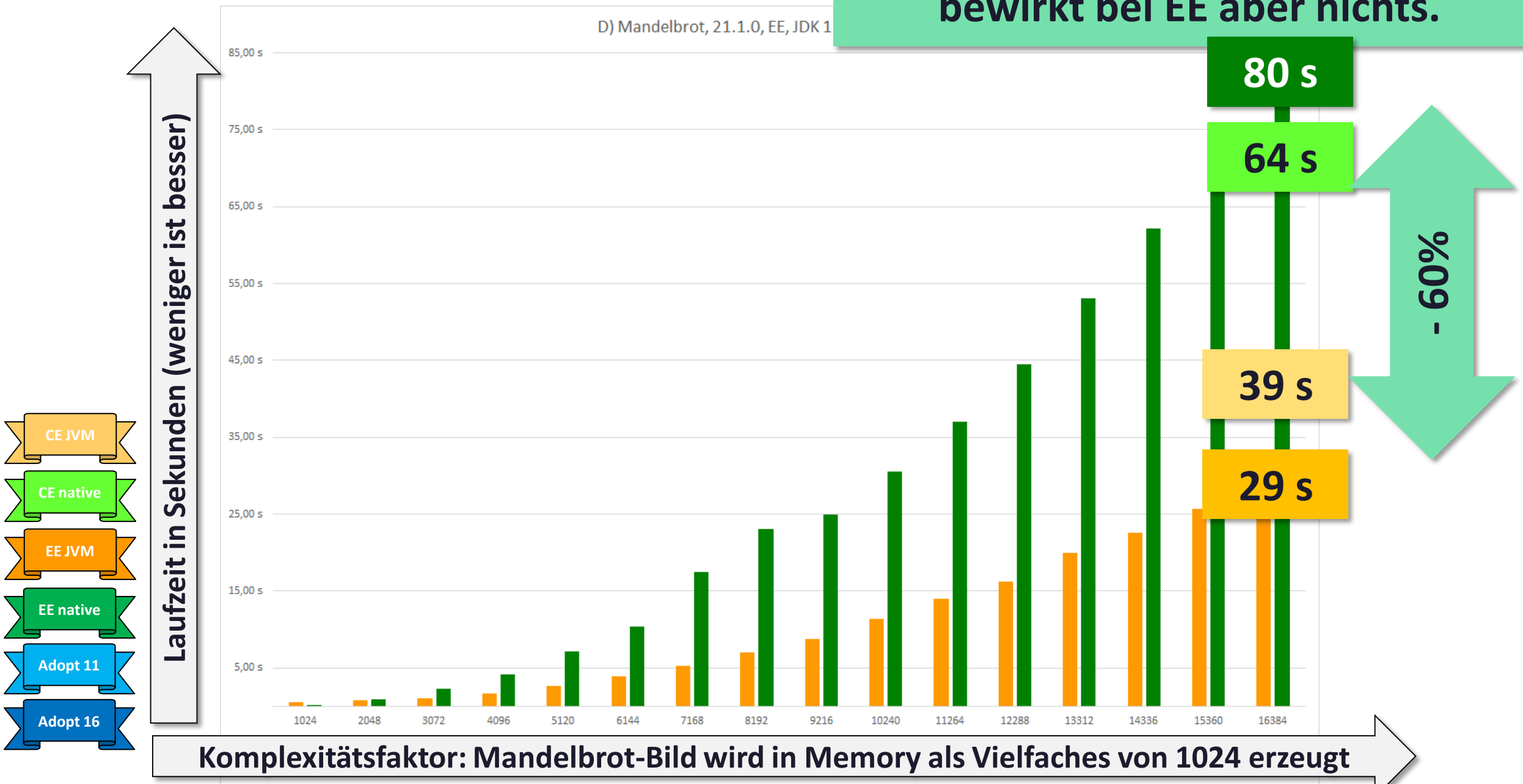
Laufzeit „mandelbrot“

CE Native wird 3x schneller, wenn man die Complex-Klasse als Mutable implementiert.
(Github: Complex / ComplexNoAlloc.java)



Laufzeit „mandelbrot“

Die gleiche Optimierung bewirkt bei EE aber nichts.

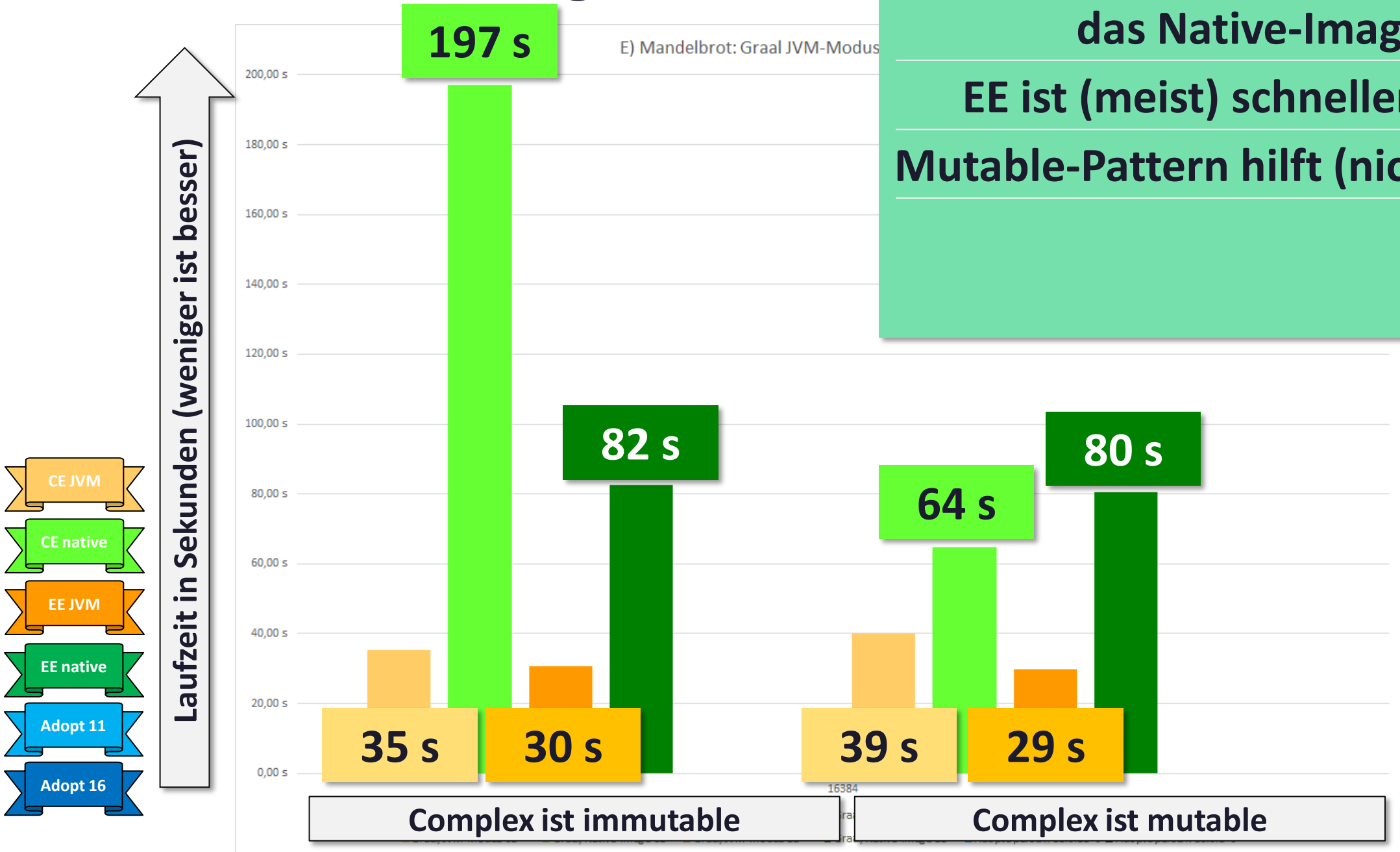


Zusammenfassung

Graals JVM-Modus ist schneller als das Native-Image.

EE ist (meist) schneller als CE.

Mutable-Pattern hilft (nicht immer).



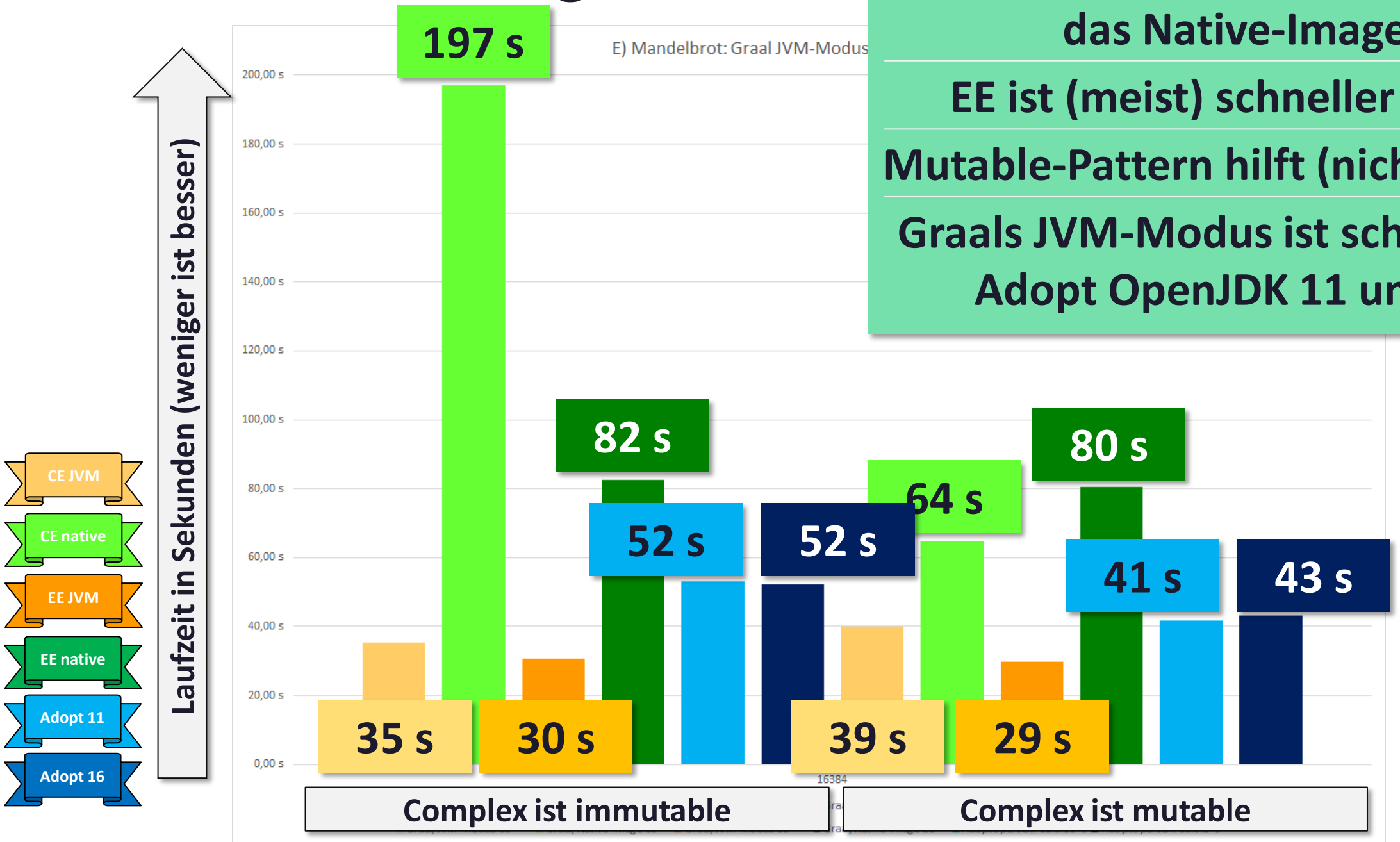
Zusammenfassung

Graals JVM-Modus ist schneller als das Native-Image.

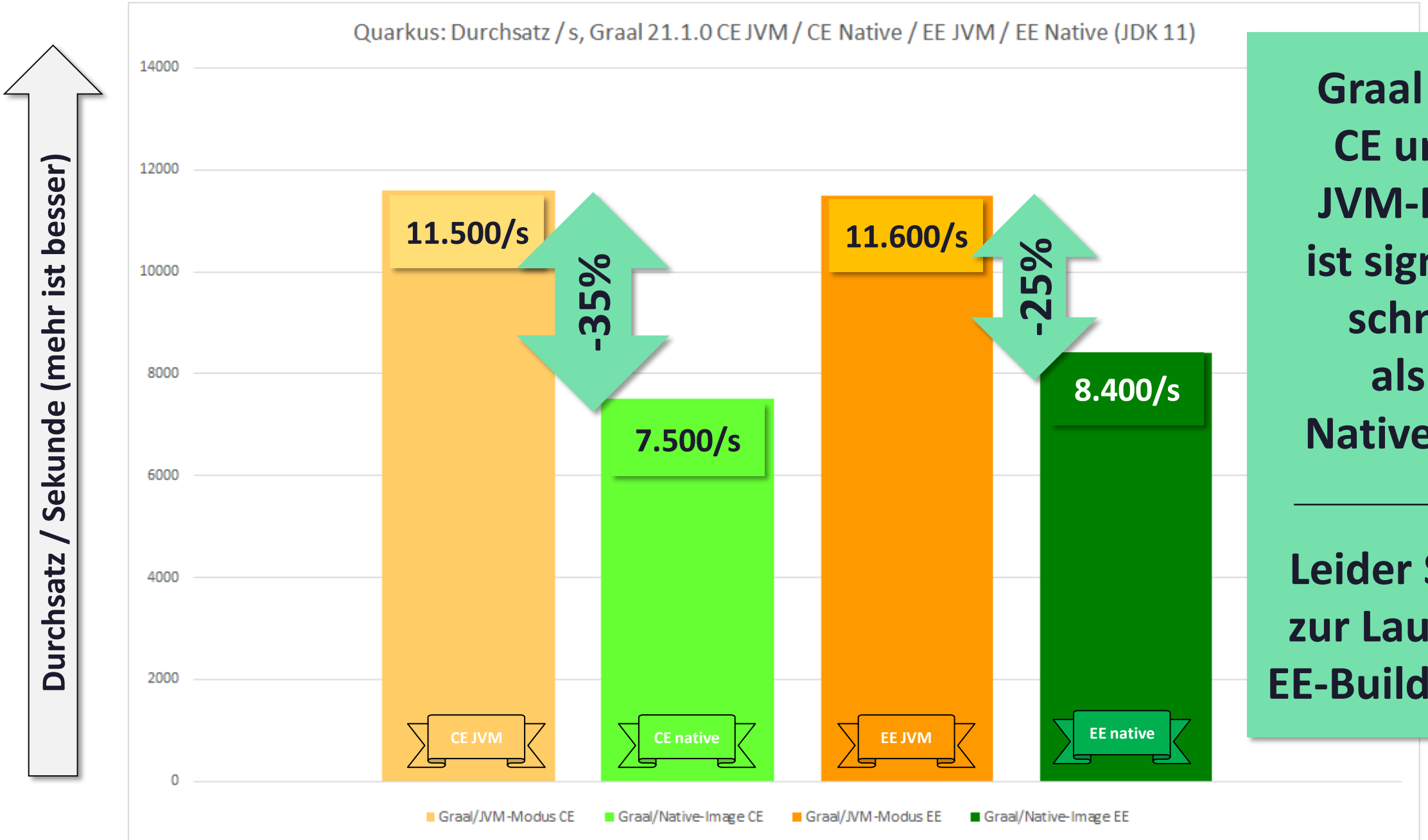
EE ist (meist) schneller als CE.

Mutable-Pattern hilft (nicht immer).

Graals JVM-Modus ist schneller als Adopt OpenJDK 11 und 16.



Durchsatz von Beispiel 3) „quarkus-timeserver“



Graal 21.1.0 CE und EE: JVM-Modus ist signifikant schneller als das Native-Image

---

Leider Segfault zur Laufzeit bei EE-Build mit PGO

GraalVM™

- Einführung: Graal? Native-Image!
- Performance: Wie schnell ist das wirklich?
- **Vorsicht: Laufzeitfehler bei Reflection & Co**
- (Wie) testet man Native-Images?

Dynamische Aspekte müssen AOT bekannt sein

Dynamische Aspekte, die erst zur Laufzeit zum Tragen kommen und daher AOT nicht / unzureichend bekannt bzw. optimiert werden können:

- ✓ Reflection
- ✓ Property-Handling
- ✓ Resource-Handling
- ✓ Dynamic Proxy
- ✓ JNI
- ✓ Logging
- X Byte-Code-Enhancement

Siehe Github-Beispiele „reflection“, „properties“, „resources“ „dynamicproxy“, „mandelbrot“

AOT muss alles Nötige bekannt sein, damit das Native-Image vollständig ist.

Dabei hilft ein **Java-Agent**, der – beim normalen JVM-Start – die nötigen Infos (z.B. für Reflection) mitschneidet und als JSON-Dateien rausschreibt.
Vollständigkeit hängt natürlich von Coverage ab!

Laufzeit-Fehler am Beispiel „mandelbrot“

```
Exception in thread "main" java.lang.NoSuchFieldError: java.awt.image.ColorModel.pData
at com.oracle.svm.jni.functions.JNIFunctions$Support.getFieldID(JNIFunctions.java:1125)
at com.oracle.svm.jni.functions.JNIFunctions.GetFieldID(JNIFunctions.java:429)
at java.awt.image.ColorModel.initIDs(ColorModel.java)
at java.awt.image.ColorModel.<clinit>(ColorModel.java:220)
…
at com.oracle.svm.core.classinitialization.ClassInitializationInfo.initialize(
   ClassInitializationInfo.java:295)
at java.awt.image.BufferedImage.<clinit>(BufferedImage.java:287)
…
at MandelbrotToPNGFile.createMandelbrotImage(MandelbrotToPNGFile.java:28)
at MandelbrotToPNGFile.main(MandelbrotToPNGFile.java:24)
```

Erfordert, dass alle JNI-Konfigurationen beim Build des Native-Images verfügbar sind.

Beispiel „reflection“: Java-Agent erzeugt JSON-Files

build.sh (gekuerzt)

```
# create META-INF/native-image/reflect-config.json (note: "output-dir")
${GRAALVM_HOME}/bin/java \
     -agentlib:native-image-agent=config-output-dir=./META-INF/native-image \
     ReflectionCaller StringManipulator reverse "hello"

# ... and then append to it in the second run (note: "merge-dir")
${GRAALVM_HOME}/bin/java \
     -agentlib:native-image-agent=config-merge-dir=./META-INF/native-image  \
     ReflectionCaller StringManipulator capitalize "world"
```

Beispiel „reflection“: Java-Agent erzeugt JSON-Files

build.sh (gekuerzt)

```
# create META-INF/native-image/reflect-config.json (note: "output-dir")
${GRAALVM_HOME}/bin/java \
     -agentlib:native-image-agent=config-output-dir=./META-INF/native-image \
     ReflectionCaller StringManipulator reverse "hello"

# ... and then append to it in the second run (note: "merge-dir")
${GRAALVM_HOME}/bin/java \
     -agentlib:native-image-agent=config-merge-dir=./META-INF/native-image  \
     ReflectionCaller StringManipulator capitalize "world"

lehmann@edwards03:~/graal/graalvm-native/reflection $
$ ls target/META-INF/native-image
jni-config.json  proxy-config.json  reflect-config.json  resource-config.json
serialization-config.json
```

Beispiel „reflection“: Java-Agent erzeugt JSON-Files

```
lehmann@edwards03:~/graal/graalvm-native/reflection $
$ cat target/META-INF/native-image/reflect-config.json
[ {
      "name":"StringManipulator",
      "methods":[
         {"name":"capitalize","parameterTypes":["java.lang.String"] },
         {"name":"reverse","parameterTypes":["java.lang.String"] } ]
} ]
```

Beispiel „reflection“: Java-Agent erzeugt JSON-Files

```
lehmann@edwards03:~/graal/graalvm-native/reflection $
$ cat target/META-INF/native-image/reflect-config.json
[ {
      "name":"StringManipulator",
      "methods":[
         {"name":"capitalize","parameterTypes":["java.lang.String"] },
         {"name":"reverse","parameterTypes":["java.lang.String"] } ]
} ]


# create native-image (again) with explicit reflection configuration
${GRAALVM_HOME}/bin/native-image \
    --no-fallback \
    -H:+PrintAnalysisCallTree -H:+ReportExceptionStackTraces \
    -H:ReflectionConfigurationResources= \
            ./META-INF/native-image/reflect-config.json \
    ReflectionCaller reflectionCallerWithExplicitConfiguration
```

Vorsicht Falle: Laufzeit-Fehler und -Fallback auf JVM

build.sh (gekuerzt)

```
# create native-image with explicit reflection configuration
${GRAALVM_HOME}/bin/native-image \
    --no-fallback \
    -H:+PrintAnalysisCallTree -H:+ReportExceptionStackTraces \
    -H:ReflectionConfigurationResources= \
            ./META-INF/native-image/reflect-config.json \
    ReflectionCaller reflectionCallerWithExplicitConfiguration
```

Ohne „--no-fallback“ fällt das Native-Image bei Problemen zur Laufzeit automatisch auf eine installierte JVM zurück (aus $PATH bzw. $JAVA_HOME).

→ Für Reproduzierbarkeit den Fallback deaktivieren (vor allem, wenn man die Zielumgebung nicht komplett unter Kontrolle hat).

Spring Native – Spring Apps auf Graal/Native

We're not going to talk about that in this video. Instead, we're going to look at a particular component inside Graal VM called the native image builder and SubstrateVM. SubstrateVM lets you build native images out of your Java application. Incidentally, I *also* did a podcast with Oracle Labs' Oleg Shelajev on this and other uses of GraalVM. The native image builder is an exercise in compromise. If you give GraalVM enough information about your application's runtime behavior - dynamically linked libraries, reflection, proxies, etc. - then it can turn your Java application into a statically linked binary, sort of like a C or Go-lang application. The process is, being honest here, sometimes... *painful*. BUT, once you do that, then the tool can generate native code for you that is *blazingly* fast. The resulting application takes *way* less RAM and starts up in below a second. *Waaay* below a second. Pretty tantalizing, eh? It sure is!

https://spring.io/blog/2020/04/16/spring-tips-the-graalvm-native-image-builder-feature

GraalVM™

- Einführung: Graal? Native-Image!
- Performance: Wie schnell ist das wirklich?
- Vorsicht: Laufzeitfehler bei Reflection & Co
- **(Wie) testet man Native-Images?**

Wie testet man ein Native-Image?

Auf Zielumgebung bzw. in Docker & Co

Funktionale Tests nur mit dem Java-Build

Erkennt funktionale Fehler (früh), wie bisher

Nutzt bestehende Pipeline und Test-Automatisierung

Erste Smoke-Tests mit dem N.I.

Erkennt Fehler beim Build des N.I.

Insbesondere dynamische Aspekte wie Reflection & Co

Integrations-tests mit dem N.I.

Erkennt, wenn Abhängigkeiten nicht richtig ins N.I. verpackt wurden.

Fallback-Falle!

Alternativ: Native-Image-Tests mit Junit. Und: Quarkus bietet @NativeImageTest. (siehe Github-Beispiele)

Fazit: GraalVM ist top, Native-Images noch nicht

Gut

- All-in-one-Sorglos-Binary-Paket mit „allem drin“.
- Startup-Zeiten sind sensationell schnell.
- Optimierter Speicherverbrauch
- Tool-Chain sehr stabil, z.B. auch Maven
- Selbst Polyglott integrierbar.
- Vieles funktioniert erwartungskonform (z.B. Exceptions, GC, …)

Schlecht

- Performance: Auf lange Sicht ist die JVM nicht zu schlagen.
- Build-Zeiten sind exorbitant.
- Nicht alles nutzbar, nicht alles stabil, manche Fehler sieht man erst zur Laufzeit.
- Sehr viele Varianten zur Optimierung

Ungewohnt

- Monitoring zur Laufzeit ist … anders. VisualVM & Co funktionieren nicht.
- Man muss (oder darf) alte/altbekannte Unix-Tools wie strings, file, strace, ltrace, ldd und gdb wieder auspacken.

GraalVM ist ein hochspannendes Projekt und die Zukunft der Java-Plattform.
Ob man schon heute Native-Images nutzen möchte, sollte man sich aber sehr gut überlegen.

Call for Discussion: New Project: Leyden

mark.reinhold at oracle.com mark.reinhold at oracle.com
Mon Apr 27 16:38:55 UTC 2020

- Previous message: Type-parameterized complement to Object.equals(Object)
- Next message: Call for Discussion: New Project: Leyden
- **Messages sorted by:** [date] [thread] [subject] [author]

I hereby invite discussion of a new Project, Leyden, whose primary goal will be to address the long-term pain points of Java's slow startup time, slow time to peak performance, and large footprint.

Leyden will address these pain points by introducing a concept of _static images_ to the Java Platform, and to the JDK.

- A static image is a standalone program, derived from an application, which runs that application -- and no other.

- A static image is a closed world: It cannot load classes from outside the image, nor can it spin new bytecodes at run time.

These two constraints enable build-time analyses that can remove unused classes and identify class initializers which can be run at build time, thereby reducing both the size of the image and its startup time. These constraints also enable aggressive ahead-of-time compilation, thereby reducing the image's time to peak performance.

Static images are not for everyone, due to the closed-world constraint, nor are they for every type of application. They often require manual configuration in order to achieve the best results. We do, however, expect the results to be worthwhile in important deployment scenarios such as small embedded devices and the cloud.

Project Leyden will take inspiration from past efforts to explore this space, including the GNU Compiler for Java [1] and the Native Image feature of GraalVM [2]. Leyden will add static images to the Java Platform Specification, and we expect that GraalVM will evolve to implement that Specification. Developers who use only the standard, specified static-image feature will then be able to switch with ease between Leyden (in the JDK), Native Image (in GraalVM), and whatever other conforming implementations may arise, choosing amongst tradeoffs of compile time, startup time, and image size.

We do not intend to implement Leyden by merging the Native Image code from GraalVM into the JDK. Leyden will, rather, be based upon existing components in the JDK such as the HotSpot JVM, the `jaotc` ahead-of-time compiler [3], application class-data sharing [4], and the `jlink` linking tool [5].

I propose to lead this Project with an initial set of Reviewers that includes, but is not limited to, Alex Buckley, Bob Vandette, Claes Redestad, Igor Veresov, Ioi Lam, Mandy Chung, and Vladimir Kozlov.

This Project will start with a clone of the current JDK main-line release, JDK 15, and track main-line releases going forward. We expect to deliver Leyden over time, in a series of JEPs that will likely span multiple feature releases.

Comments?

- Mark

@mrtnlhmnn @accso

https://speakerdeck.com/mrtnlhmnn

https://github.com/accso/graalvm-native

ACCSO

https://accso.de/

https://accso.de/publikationen/

ACCSO
ACCELERATED SOLUTIONS

Accso – Accelerated Solutions GmbH
T | +49 6151 13029-0
E | info@accso.de
@ | www.accso.de

Hilpertstraße 12 | 64295 Darmstadt
Moltkestraße 131a | 50674 Köln
Balanstraße 55 | 81541 München