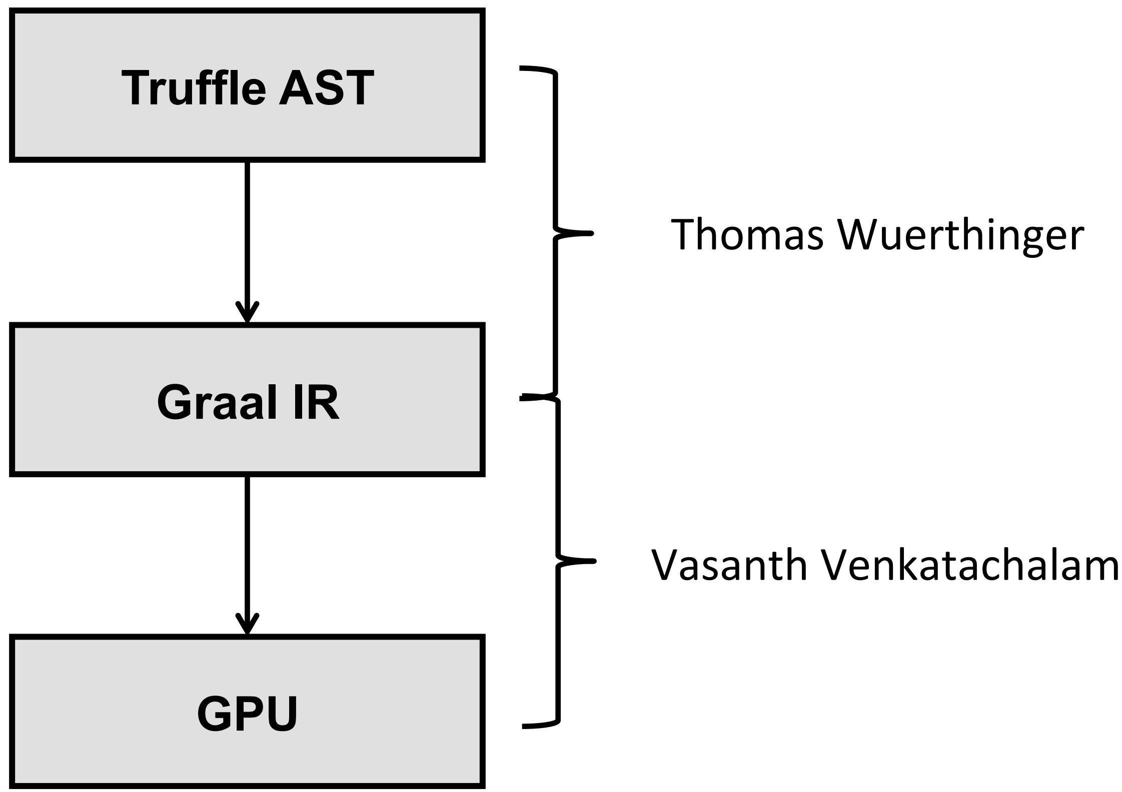
Talk Agenda


Truffle AST
Graal IR
GPU
Thomas Wuerthinger
Vasanth Venkatachalam

ORACLE®

Graal Status

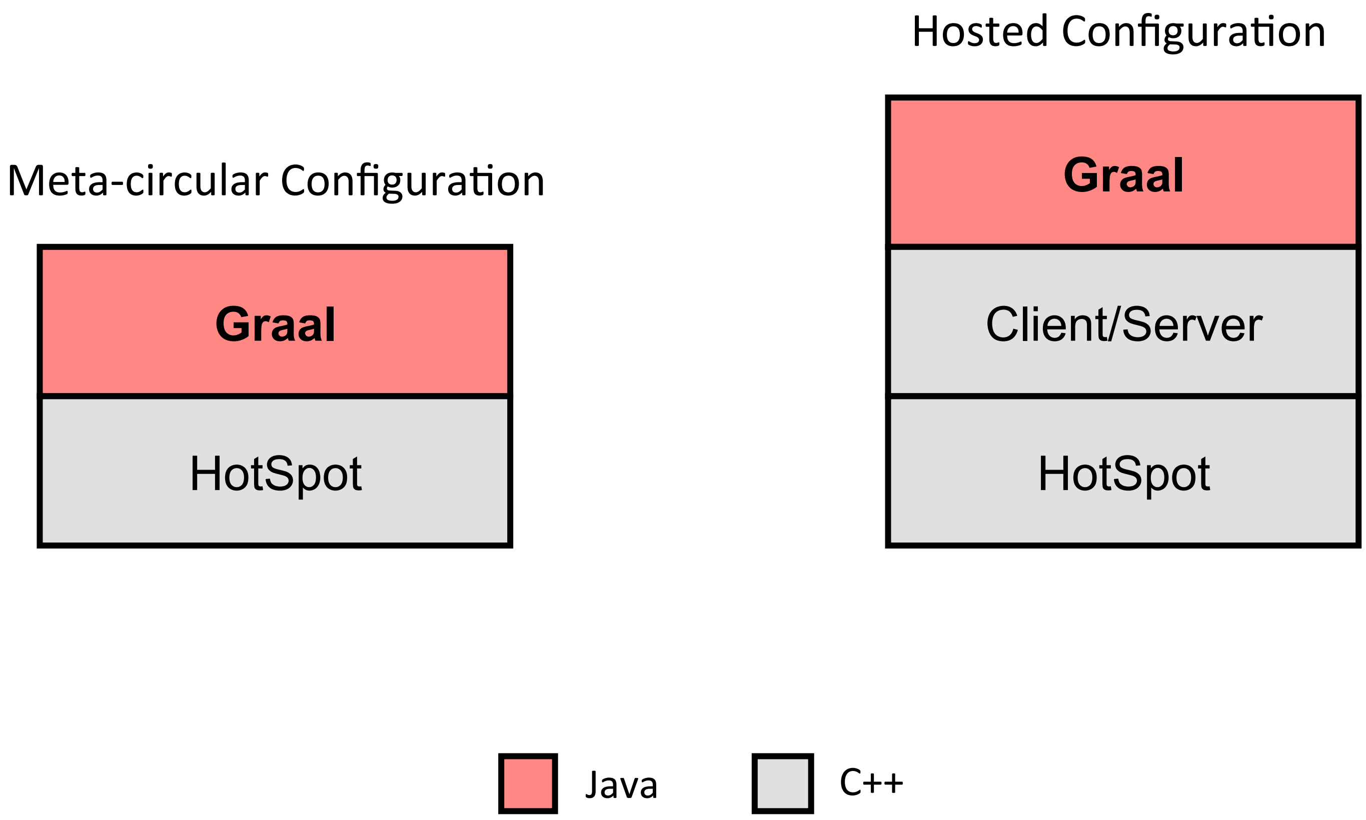
Thomas Wuerthinger

JVM Language Summit, July 30, 2013

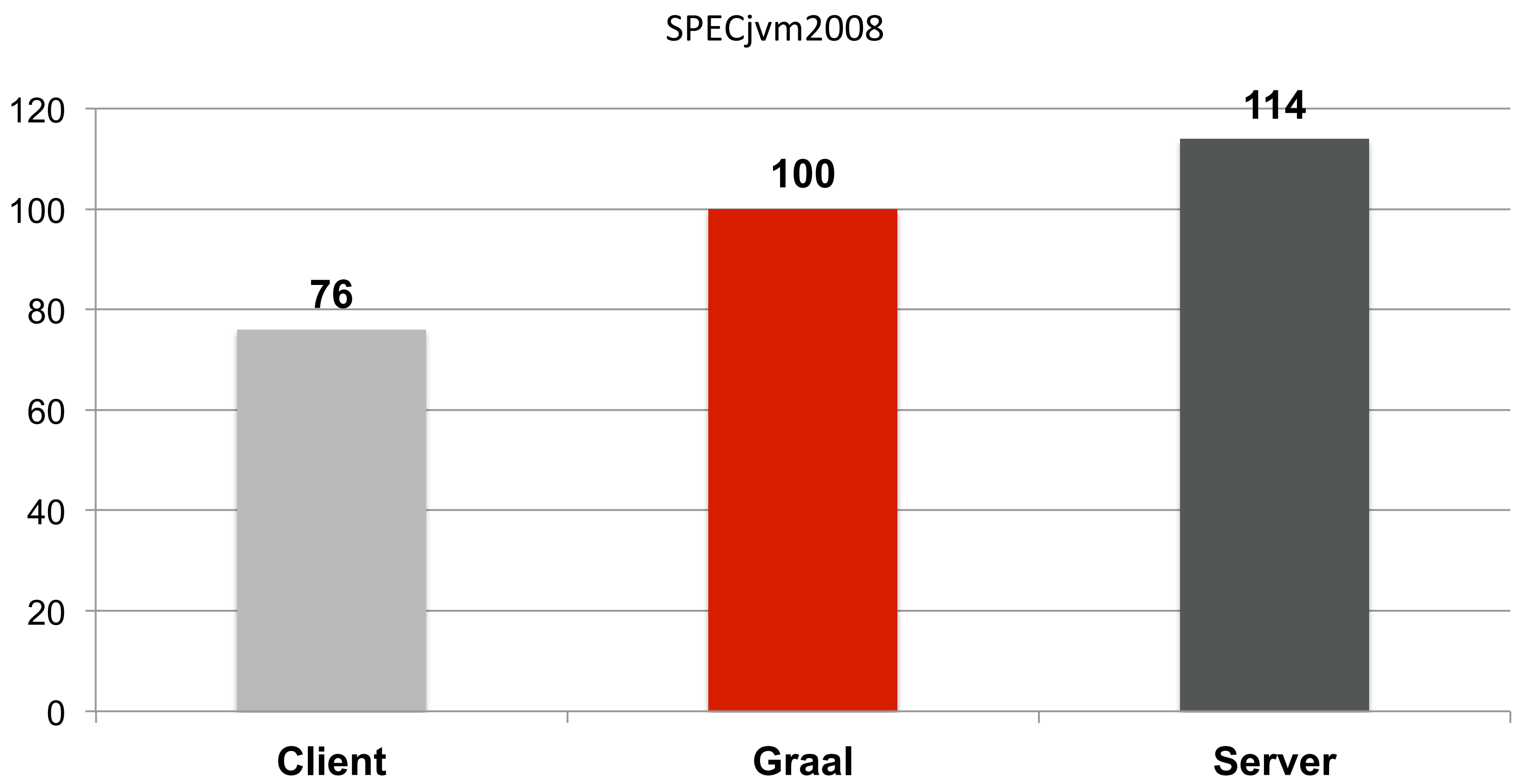
Disclaimer

The following is intended to provide some insight into a line of research in Oracle Labs. 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, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Graal Architecture

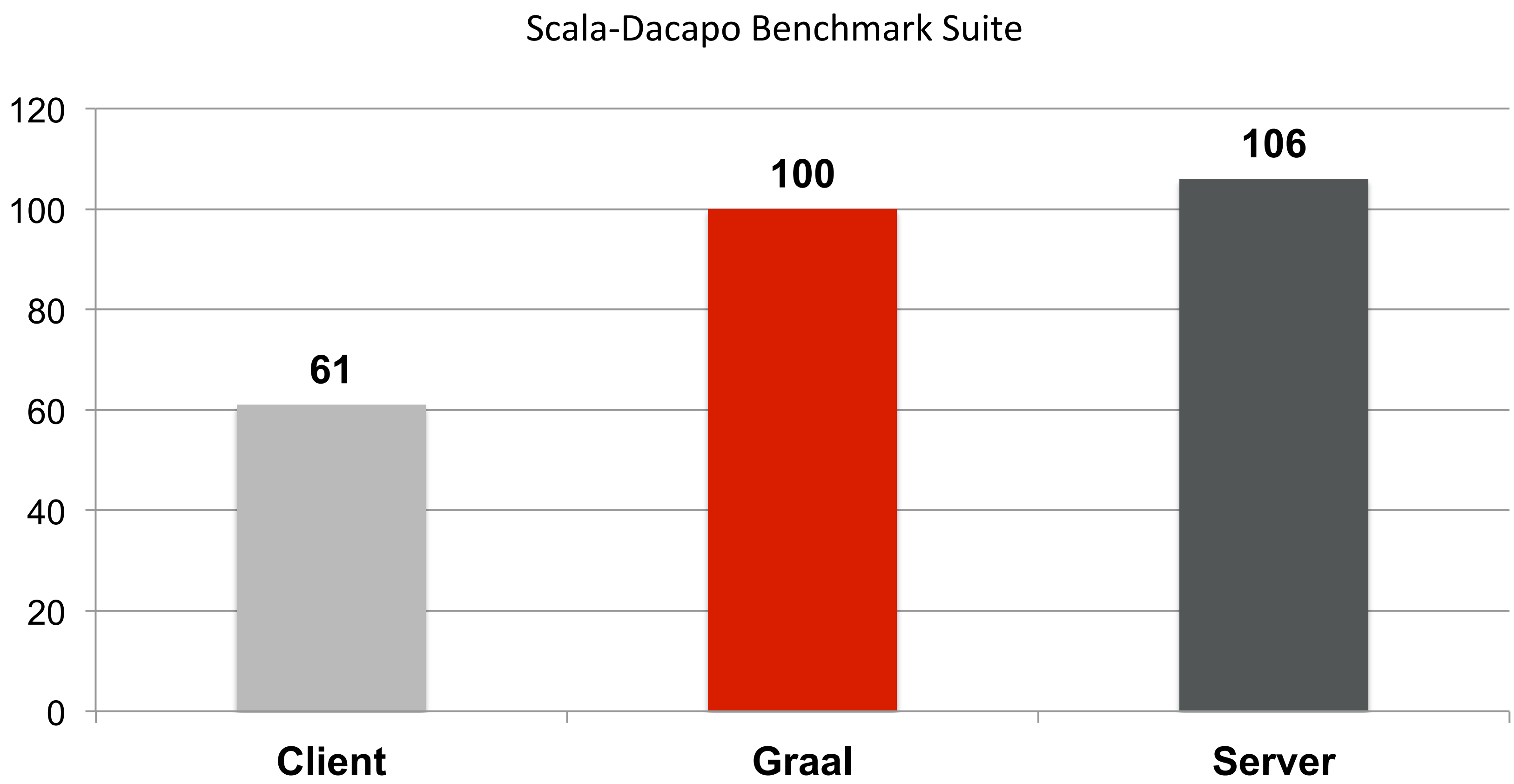


Java Peak Performance



Configuration: Intel Core i7-3770 @ 3,4 Ghz, 4 Cores 8 Threads, 16 GB RAM
Comparison against HotSpot changeset tag hs25-b37 from June 13, 2013

Scala Peak Performance

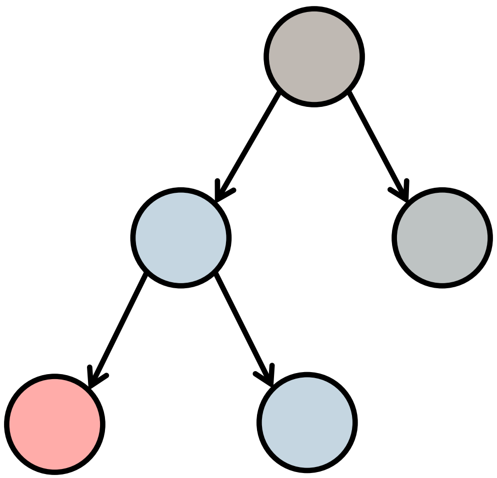


Configuration: Intel Core i7-3770 @ 3,4 Ghz, 4 Cores 8 Threads, 16 GB RAM
Comparison against HotSpot changeset tag hs25-b37 from June 13, 2013

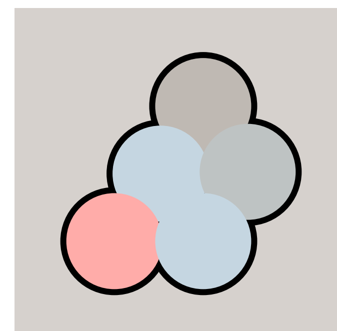
Truffle: Dynamic Language Frontend

AST Interpreter

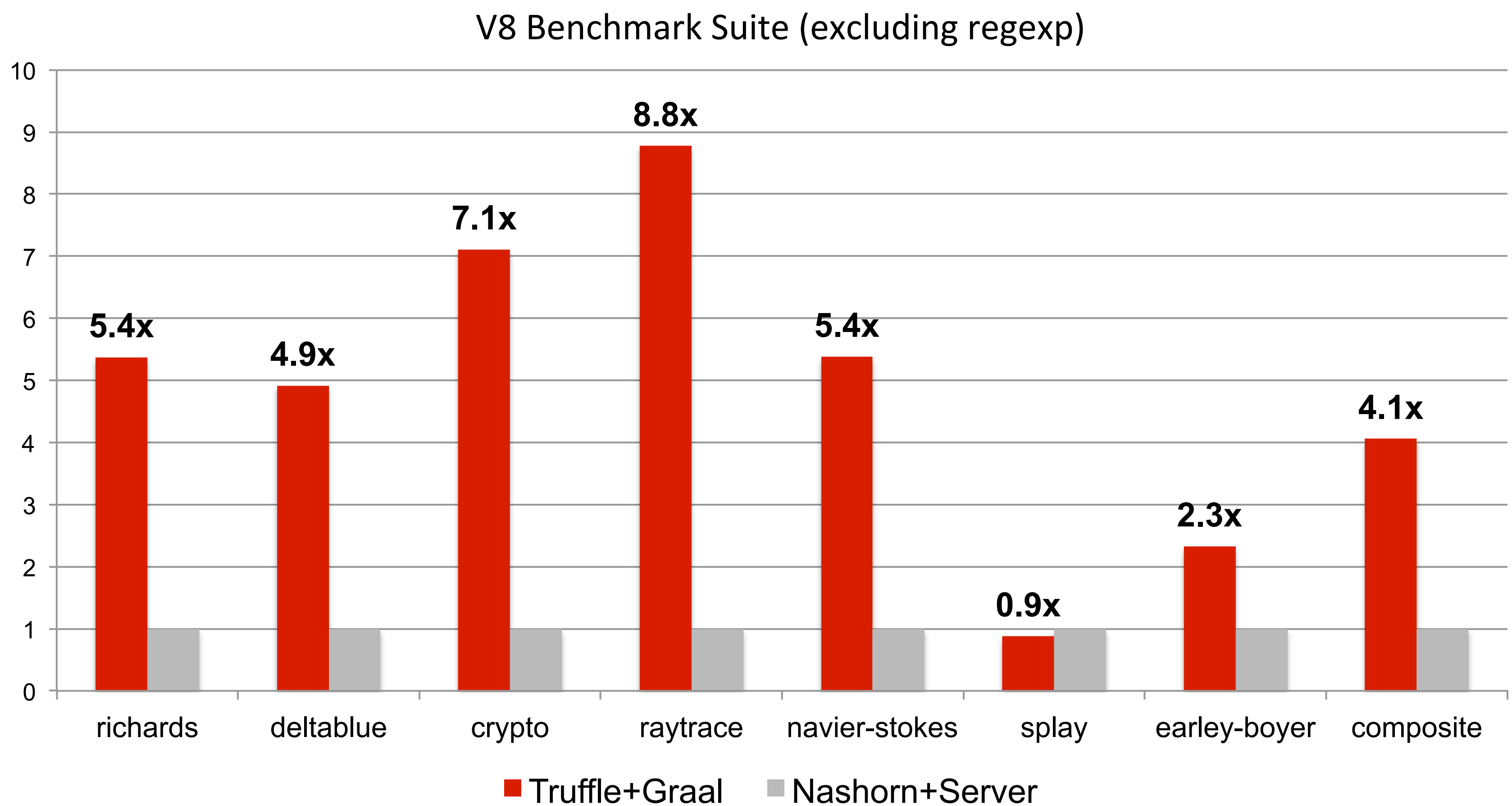
Compiled Code



automatic partial
evaluation

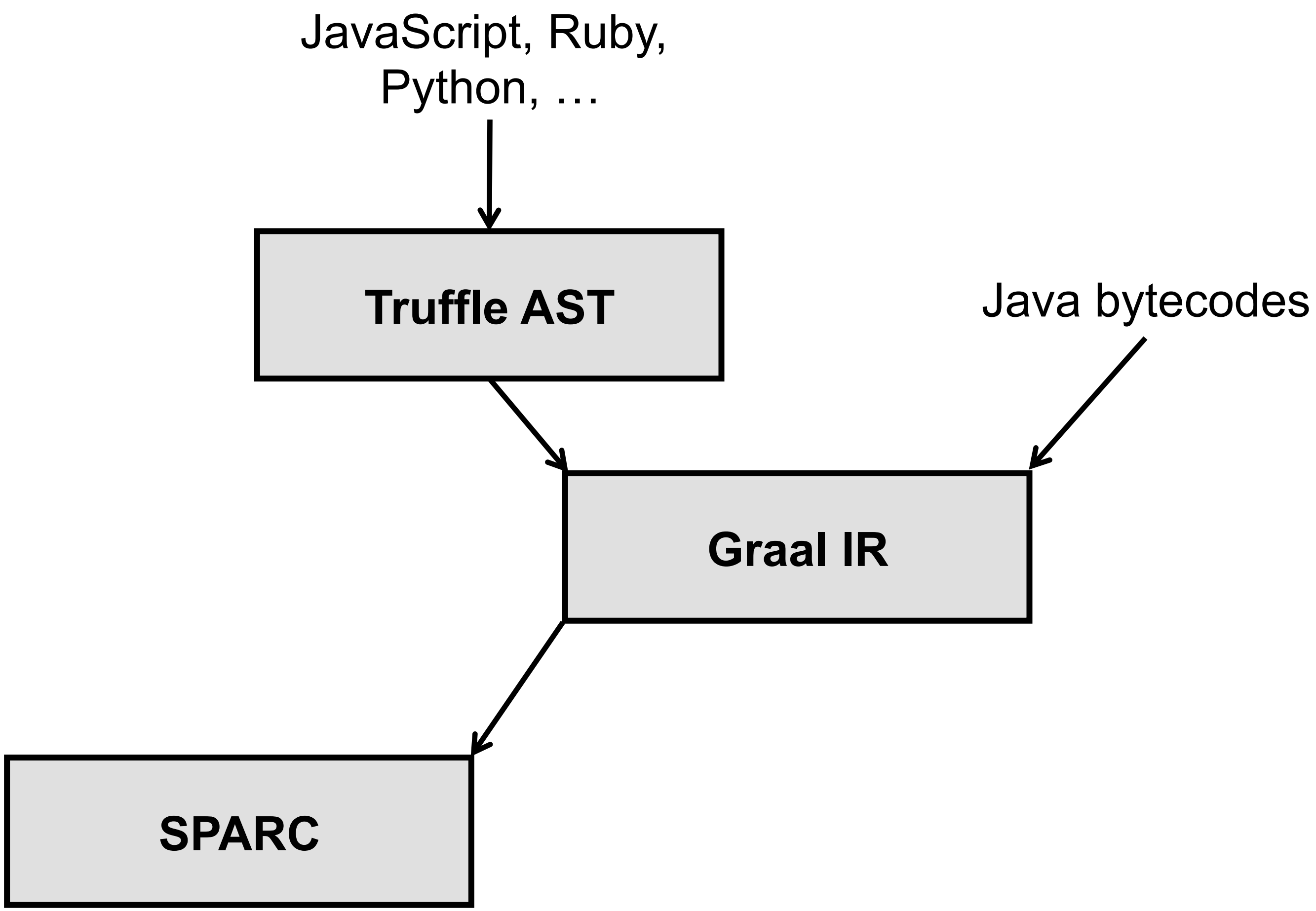


JavaScript Peak Performance



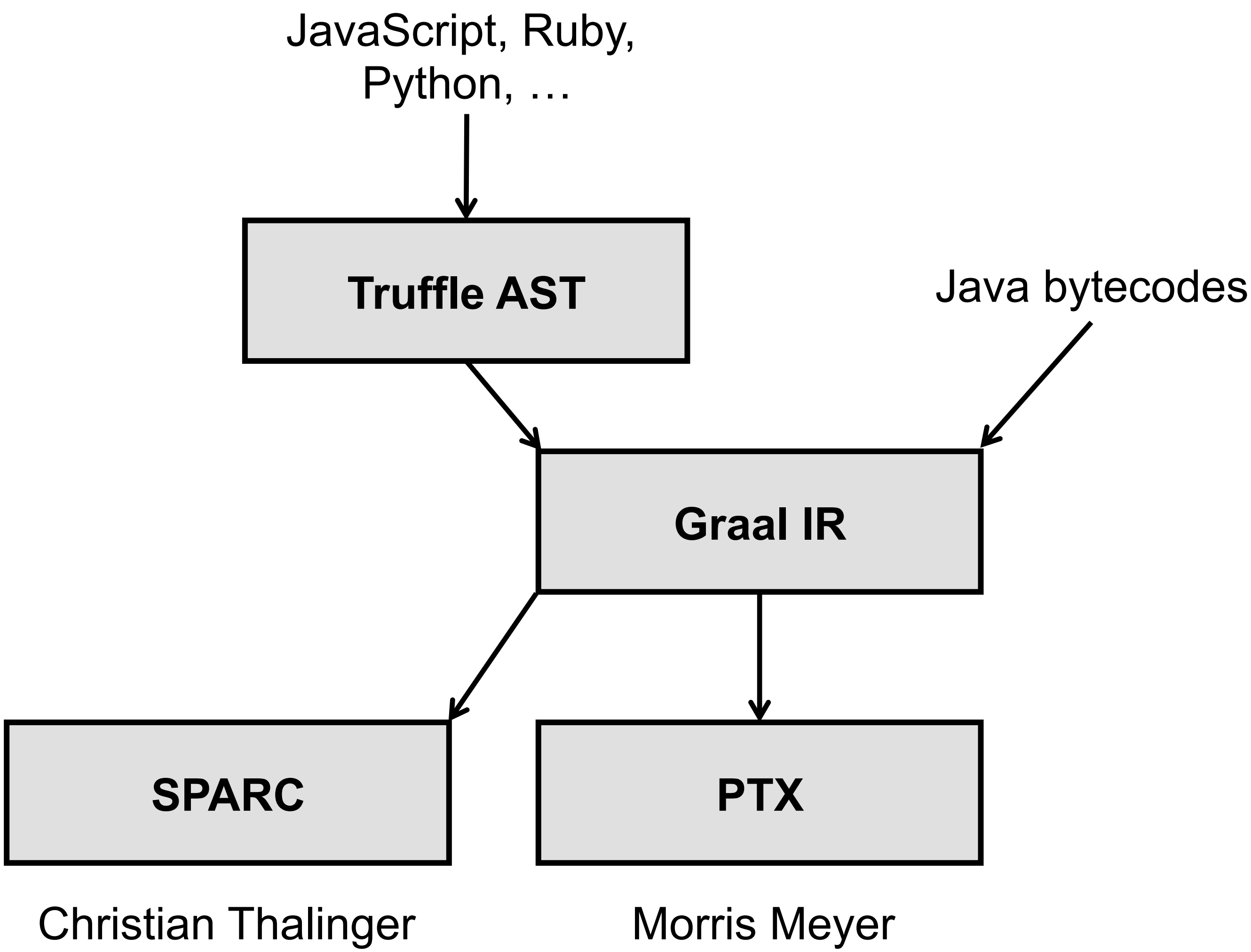
Configuration: Intel Core i7-3770 @ 3,4 Ghz, 4 Cores 8 Threads, 16 GB RAM
Comparison against JDK 8 Early Access Release, Build b99 from July 19, 2013

New Graal Backends (1)

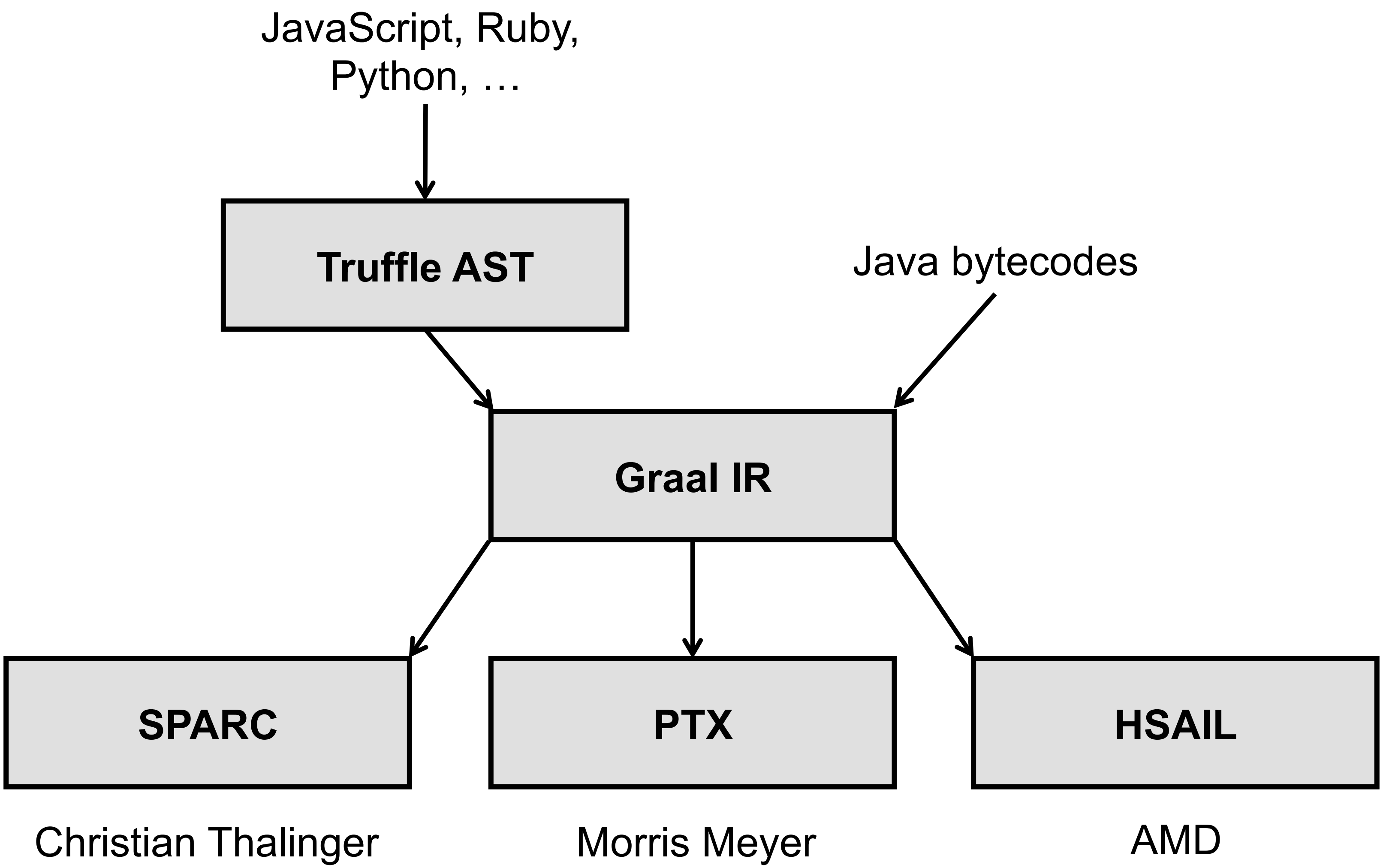


Christian Thalinger

New Graal Backends (2)



New Graal Backends (3)



Acknowledgements

Oracle Labs
Michael Haupt
Shams Imam (Intern)
Peter Kessler
Christos Kotselidis
Helena Kotthaus (Intern)
David Leibs
Roland Schatz
Chris Seaton (Intern)
Doug Simon
Michael Van De Vanter
Christian Wimmer
Christian Wirth
Mario Wolczko
Thomas Wuerthinger

JKU Linz
Gilles Duboscq
Matthias Grimmer
Christian Haeubl
Christian Humer
Christian Huber
Manuel Rigger
Lukas Stadler
Bernhard Urban
Andreas Woess

Hardware and Software

ORACLE

Engineered to Work Together

ORACLE®

Adding an HSAIL GPU back-end to Graal

JVM LANGUAGE SUMMIT

VASANTH VENKATACHALAM, JULY 2013

AGENDA

Why we are interested in GPU offload

Special considerations for Java GPU compilation

Why we chose Graal

– How we use Graal with the HSA runtime stack

Heterogeneous System Architecture Intermediate Language (HSAIL) code generation back-end for Graal

– Development and testing status

– Example HSAIL output for a Java program

Summary

WHY WE ARE INTERESTED IN GPU OFFLOAD

Typically, offloading the data-parallel parts of a program to a GPU would improve the performance per watt compared to running the entire program on the CPU.

- In a data-parallel computation in which the same computation is repeated over different data (and the results are not dependent on each other), the individual computations can be executed in parallel on multiple cores.
- For example, imagine squaring the elements of a large array. The individual square operations can be run in parallel on different cores because they don't depend on one another.
- A typical GPU offers more cores for the same density than a CPU (due to the smaller form factor). Because of this, we expect to get better performance.

SPECIAL CONSIDERATIONS FOR JAVA GPU COMPILATION

Java needs a programming model to express data-parallel workloads.

- We chose to use Java 8's lambda and stream API.

JVM needs to generate code for GPU while running on CPU in addition to generating code for the CPU.

- So the compilation framework and JVM will need to deal with targeting multiple ISAs.
- We refer to this as adding "multi-ISA support" to the JVM.

Ideally, the JVM can target a single, common intermediate format for HSA-enabled GPU devices instead of targeting each possible GPU ISA.

- The intermediate format can be considered the "bytecodes" for a GPU target.
- This extra translation layer provides the advantage of portability.
  - GPU ISAs change more frequently than CPU ISAs.
  - The high-level language (C, Java, etc.) compilers don't need to change every time there's an ISA change. Only the final translation layer would need to be updated.

WHY WE CHOSE GRAAL

Graal is a highly extensible, open-source, just-in-time compiler for Java.

Graal is written in Java.

- Graal can be developed using Java IDEs (e.g., Eclipse, NetBeans).
- These existing tools make Graal straightforward to debug.
- Because Graal is written in Java, it can run on any platform and thus be treated as a cross-compiler.
- In particular, we can compile for the GPU while running on the CPU.
- This would allow us to create a multi-ISA framework.

We chose Graal based on the recommendation of the Hotspot team.

- We got the feedback that leveraging Graal would be the most efficient way to come up with a working prototype for JVM-driven GPU code generation.

HSAIL PRIMER

AMD

HSAIL is the code that the JVM will emit

Gets translated to the ISA of the GPU device by the “finalizer”

Generated code is ASCII text form, which aids in debugging

Example: signed 32-bit multiplication

```
mul_s32 $s3, $s0, $s1
```

Source1

Source2

Destination

Mnemonic (mul, add, sub, div, Etc.)

Type modifier (s, u, b, f)

Length modifier (1, 8., 16, 32, 64 etc)

Register model

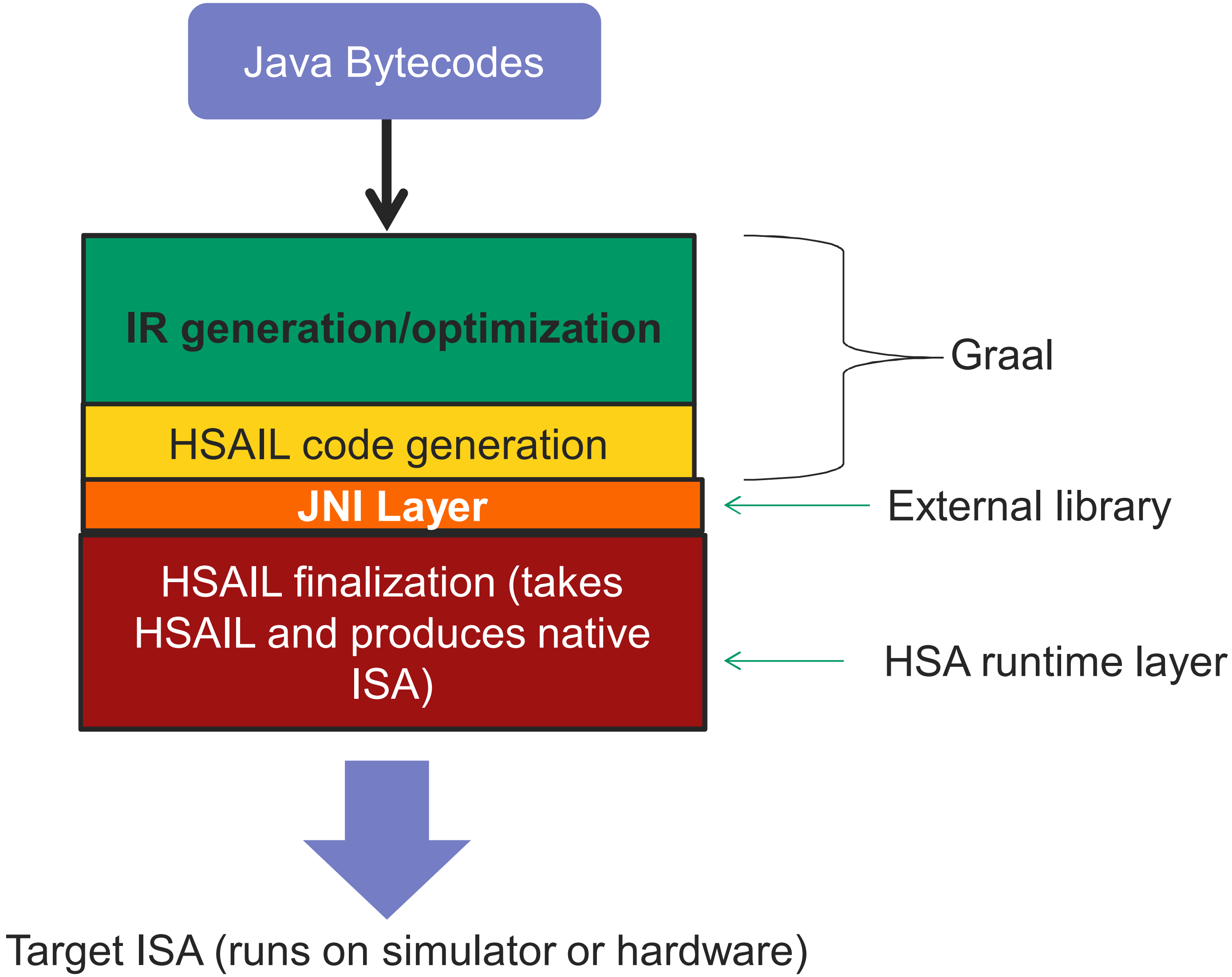
128 32-bit registers (s0-s127)

64 64-bit registers (d0-d63)

32 128-bit registers (q0-q31)

8 control registers (c0-c7)

HOW SUMATRA USES GRAAL AND THE HSAIL BACK-END



HSAIL BACK-END FOR GRAAL: DEVELOPMENT STATUS

Checked into the public branch

Features

- Supports basic arithmetic, control flow, convert instructions
- Mapping for common intrinsics (Math.sqrt -> sqrt(src, dest))
- Register spilling
- Loads and stores through compressed and non-compressed references
- Supports compilation of Java lambda/stream API constructs
  - Graal development environment (e.g., Eclipse™) does not support Java 8 yet

Work in progress

- Function call support
  - Thankfully, Graal can aggressively inline
- Create an HSAIL-aware register allocator instead of using the existing x86 solution
- Emitting useful annotations alongside the code generated

HSAIL BACK-END FOR GRAAL: TEST COVERAGE

- Expanding testing coverage
  - 130 unit test cases and demo applications
  - Java 8- and Java 7-based test cases, including lambda and stream API examples
  - Includes regression tests that check that the results returned by Java and HSAIL executions are identical
- Tests have been run on a simulator as well as prototype AMD hardware
- Open-source simulator available at HSA Foundation GitHub Repository
  - Supports HSAIL debugging features such as single stepping and viewing the HSAIL registers
  - OKRA is a Java interface to some of the features of the HSA runtime

For more details see

https://wiki.openjdk.java.net/display/Sumatra/The+HSAIL+Simulator

EXAMPLE HSAIL CODE GENERATED FOR A SAMPLE JAVA PROGRAM (SQUARES)

```
Intstream forEach (i-> {
    out[i] = in[i] * in[i];
});
```

What the compiler sees!

```
private static void lambda$67(int[], int[], int)  {
out[i] = in[i] * in[i]
}
```

Parameter passed to lambda

```
kernel &run (
    kernarg_u64 %_arg0,
    kernarg_u64 %_arg1
    ) {
    ld_kernarg_u64  $d6, [%_arg0];
    ld_kernarg_u64  $d2, [%_arg1];
    workitemabsid_u32 $s1, 0;
     cvt_s64_s32 $d0, $s1;
    mul_s64 $d0, $d0, 4;
    add_u64 $d2, $d2, $d0;
    ld_global_s32 $s0, [$d2 + 24];
    mul_s32 $s3, $s0, $s0;
    cvt_s64_s32 $d1, $s1;
    mul_s64 $d1, $d1, 4;
    add_u64 $d6, $d6, $d1;
    st_global_s32 $s3, [$d6 + 24];
    ret;
};
```

Parameter passing

Load workitem id of current workitem

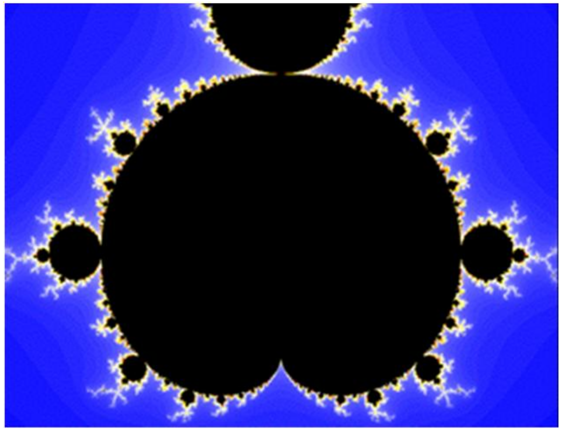
Load in[i]

in[i] * in[i]

Store to out[i]

- **Data-parallel execution model**
  - **Each workitem has a unique id**
  - **workitemabsid instruction returns the id of the current workitem**

HSAIL CODE FOR MANDELBROT LOOP BODY



```
count = 0;
maxIterations = 64;
while ((count < maxIterations) &&
       (zx * zx + zy * zy < 8)) {
          newzx = zx * zx - zy * zy + lx;
          zy = 2 * zx * zy + ly;
          zx = newzx;
          count++;
}
```

10x performance speed-up compared to Java parallel execution on prototype hardware

```
@L4:
    mul_f32 $s18, $s20, $s20;      //zx*zx
    mul_f32 $s21, $s19, $s19;      //zy*zy
    add_f32 $s22, $s21, $s18;      //zx*zx+zy*zy

    cmp_geu_b1_f32 $c0, $s22, 8.0f;  //zx*zx+zy*zy < 8 ?
    cbr $c0, @L5;                    //if not, then exit
@L6:
    sub_f32 $s18, $s18, $s21;      //zx*zx – zy*zy
    add_f32 $s18, $s18, $s16;      //+lx
    mul_f32 $s20, $s20, 2.0f;      //2*zx
    mul_f32 $s20, $s20, $s19;      //*zy
    add_f32 $s20, $s20, $s17;      //+ly
    add_s32 $s0, $s0, 1;           //count++
    mov_b32 $s19, $s20;            //$s19=zy
    mov_b32 $s20, $s18;            //zx = newzx
@L3:
    cmp_lt_b1_s32 $c0, $s0, 64;  //count < maxIterations?
    cbr $c0, @L4;                //if not then exit
```

SUMMARY

- GPU offload is beneficial for improved performance and power savings
- We have contributed an HSAIL back-end for Graal
- Prototype supports a variety of Java 8 and Java 7 test cases
  - Tested on simulator and hardware
- This work allows JVMs to compile for HSAIL-enabled GPU devices
- We encourage OpenJDK community feedback and contributions

REFERENCES

AMD DevCentral blog on HSAIL-based GPU Offload

- http://developer.amd.com/community/blog/hsail-based-gpu-offload-the-quest-for-java-performance-begins/

Sumatra OpenJDK GPU/APU offload project

- Project home page: http://openjdk.java.net/projects/sumatra/
- Wiki: https://wiki.openjdk.java.net/display/Sumatra/Main

Graal JIT compiler and runtime project

- Project home page: http://openjdk.java.net/projects/graal/

HSA Foundation:

- http://hsafoundation.com/
- http://hsafoundation.com/standards/

AMD Developer Summit 2013 (APU 2013)

- http://developer.amd.com/apu
- Explore latest developments in heterogenous computing, OpenCL™, C++ AMP and related technologies
- Keynotes from industry leaders, how-to sessions and technical planning, experience hub featuring first-ever technology demonstrations.

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION