AMD's Prototype HSAIL-enabled JDK8 for the OpenJDK Sumatra Project

JVM LANGUAGE SUMMIT

ERIC CASPOLE

JULY 2013

AGENDA

- Sumatra OpenJDK project
- GPU workload fundamentals
- AMD APU and Heterogeneous System Architecture (HSA)
- AMD HSAIL-enabled offload demo JDK
- Summary

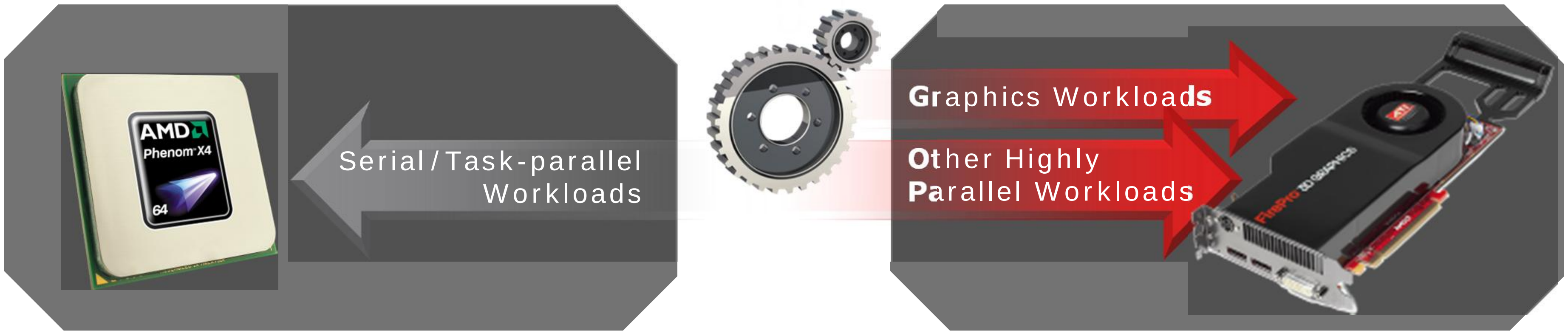
SUMATRA OPENJDK PROJECT

- Intending to enable Java applications to take advantage of GPU/APU
  - More or less transparently to the application
  - No application native code required
- Project started by Oracle and AMD shortly before JavaOne 2012
- GPU/APUs offer a lot of processing power
  - 2000 ASCI RED, Sandia National Laboratories
    - World's #1 supercomputer
    - http://www.top500.org/system/ranking/4428
    - ~3,200 GFLOPS
  - 2010 AMD Radeon™ HD 5970
    - ~4,700 GFLOPS
    - Already obsolete in my desk drawer
- HSA/OpenCL/CUDA standardize how to express both the GPU compute and host programming requirements
  - But not easy to use from Java without a lot of native code and expertise
  - Existing APIs include Aparapi, JOCL, OpenCL4Java, and others

IDEALLY, WE CAN TARGET COMPUTE AT THE MOST SUITABLE DEVICE

CPU excels at sequential, branchy code, I/O interaction, system programming.
Most Java applications have these characteristics and excel on the CPU.

GPU excels at data-parallel tasks, image processing, and data analysis.
Java is used in these areas/domains, but does not exploit the capabilities of the GPU as a compute device.



IDEAL DATA PARALLEL ALGORITHMS/WORKLOADS

- GPU SIMDs are optimized for data-parallel operations
  - Performing the same sequence of operations on different data at the same time
  - Each GPU core gets a unique work item id, often used as an array index
- The body of loops are a good place to look for data-parallel opportunities

```
// Each loop iteration is independent
for (int i=0; i< 100; i++)
    out[i] = in[i]*in[i];
```

- As a JDK 8 Stream operation:
  - This is a thread-safe calculation and could be a parallel stream

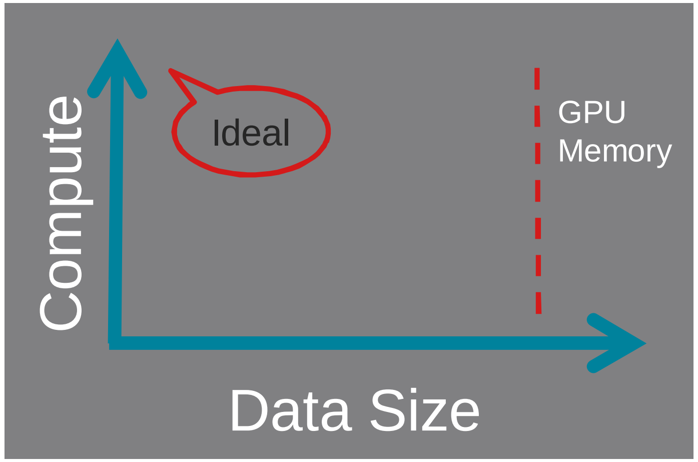
```
IntStream.range(0, in.length).forEach( p -> {
    out[p] = in[p] * in[p];
});
```

- Particularly if we can loop in any order and get same result

```
for (int i=99; i<= 0; i--)
    out[i] = in[i]*in[i];
```

CHARACTERISTICS OF AN IDEAL DISCRETE GPU WORKLOAD

- Each iteration contains sequential code
  - Modern GPU may have more than 1,500 stream cores
  - All GPU cores should be in “lock step” to get the best performance
  - Few divergent branches per wavefront
- Good balance between data size (low) and compute (high)
  - Data used in discrete GPU computation must be copied to/from card memory
- Transfer of data to/from the GPU can be costly
  - Trivial compute often not worth the transfer cost
  - May still benefit, by freeing up CPU for other work
- HSA will remove some of these limitations



WATCH OUT FOR DEPENDENCIES AND BOTTLENECKS

- Data dependencies can violate the "in any order" guideline

```
// for loop style
for (int i=1; i<100; i++) {
    out[i] = out[i-1] + in[i];
}
```

```
// stream style
IntStream.range(0, in.length).forEach( p -> {
    out[p] = out[p-1] * in[p];
});
```

- Mutating shared data can force use of atomic constructs
  - Note lambdas do not allow modifying captured values

```
for (int i=0; i< 100; i++)
    sum += in[i];
```

- Sometimes we can refactor to expose some parallelism

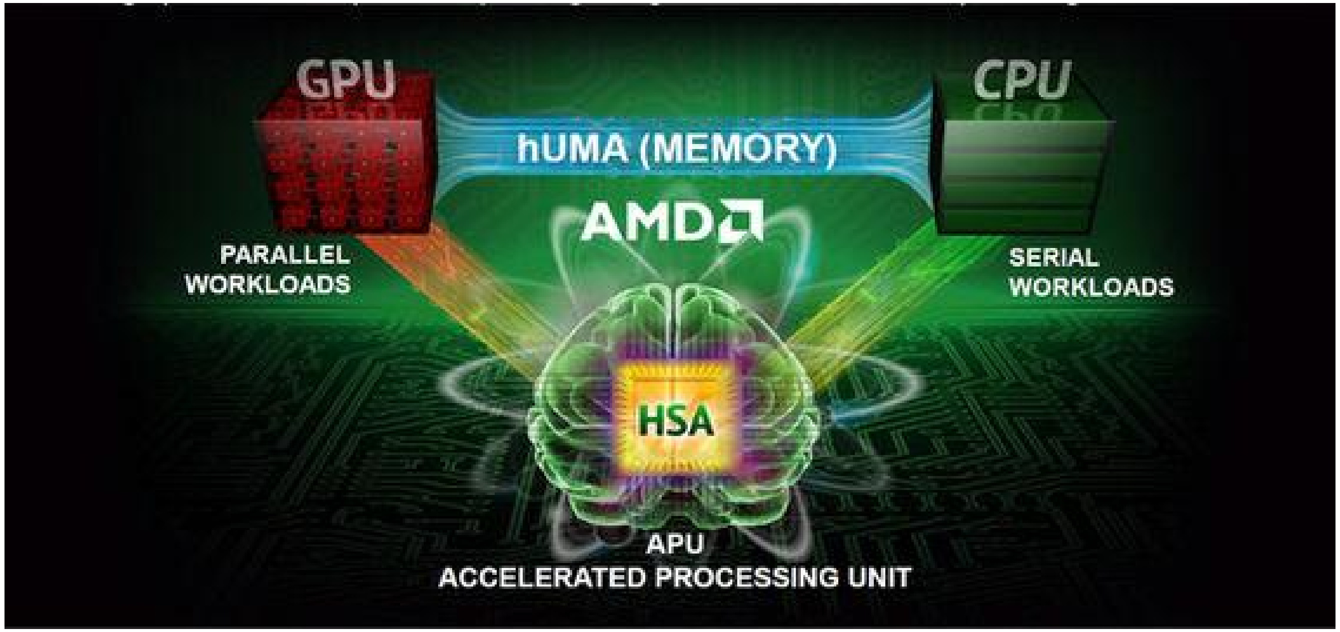
```
for (int n=0; n<10; n++)
   for (int i=0; i<10; i++)
     partial[n] += data[n*10+i];

for (int i=0; i< 10; i++)
     sum+=partial[i];
```

MEET HSA AND HSAIL

- Heterogeneous System Architecture standardizes CPU/GPU functionality
  - Be ISA-agnostic for both CPUs and accelerators
  - Support high-level programming languages
  - Provide the ability to access pageable system memory from the GPU
  - Maintain cache coherency for system memory between CPU and GPU
- Specifications and simulator from HSA Foundation
  - HSAIL portable ISA is "finalized" to particular hardware ISA at runtime
  - runtime specification for job launch and control
  - HSAIL simulator for development and testing before hardware availability

AMD ACCELERATED PROCESSING UNIT

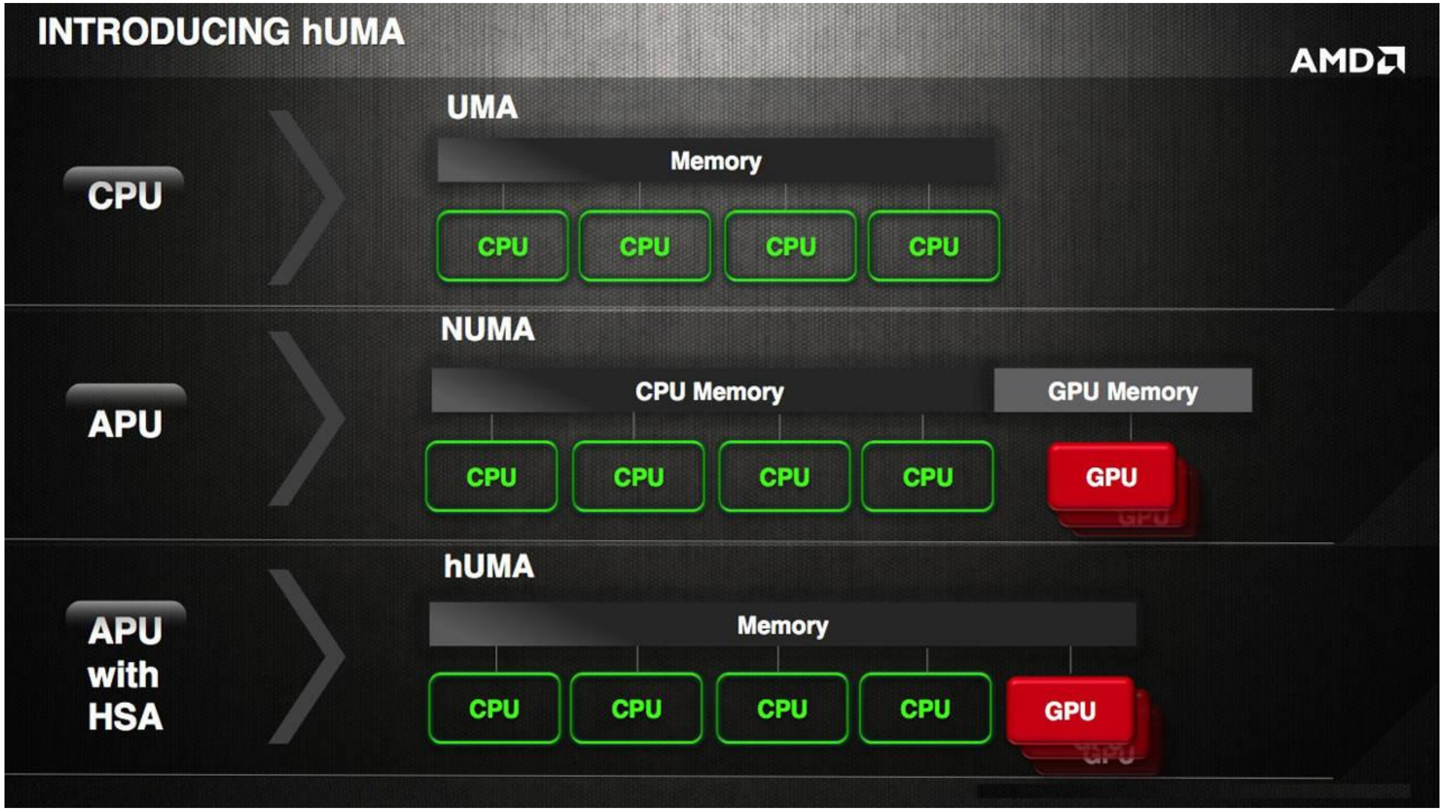
- AMD APU
  - CPU/GPU on one integrated chip
  - Various APU models shipping since June 2011
  - The upcoming Berlin APU will be the first to support HSA software stack
- HSA makes a great platform for Java offload
  - Direct access to Java heap objects in main memory from GPU cores
  - No extra copying over bus to discrete card
  - Pointer is a pointer from CPU or GPU application code



AMD hUMA ARCHITECTURE

AMD

- Upcoming AMD APUs feature heterogeneous Uniform Memory Access
  - Designed to work with HSA
  - Pointer is a pointer from CPU or GPU application code -- no copying over a bus



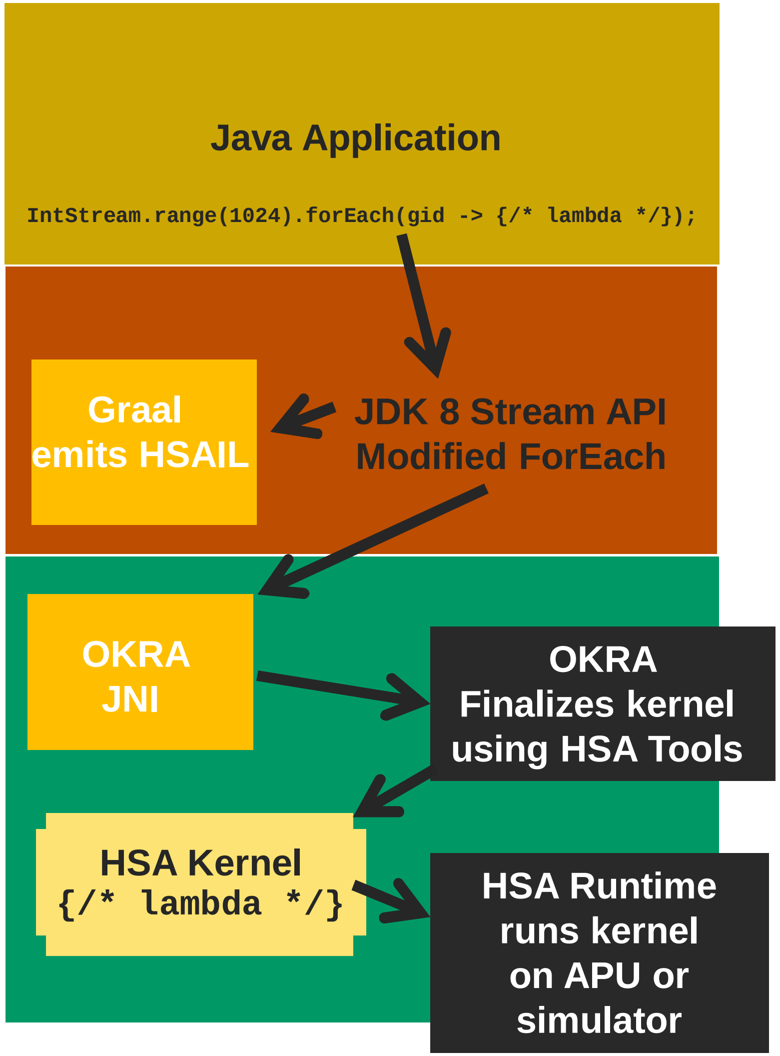
AMD SUMATRA PROTOTYPE: APU MEETS STREAM API

AMD

- Enables HSA APU offload of some JDK 8 parallel stream lambdas
  - Use of `parallel()` means developer thinks it's thread-safe
  - Immediate offload via flag or C2 compiler intrinsic to offload later
  - No special API or coding requirements for application developer
- We are adding HSAIL support to Graal
  - Basic HSAIL functionality already committed into Graal project
- We hook into `j.u.stream.ForEachOp` to redirect to our HSA offload code
  - `ForEach` "side effect" operation fits well with GPU data-parallel model
  - Do math, set field values, but no allocation or synchronization yet
  - Direct access to Java objects in the heap from GPU cores
- Seamless fallback to regular JDK code if code gen or offload fails
- This code available in Graal and a JDK webrev to be built together
- Can be easily run in open-source HSA simulator on regular systems

AMD SUMATRA PROTOTYPE: DIAGRAM

- Our JDK uses open-source HSAIL tools
- OKRA is a layer allowing easy use of the HSAIL tools from Java
- HSAIL tools assemble and finalize the HSAIL source emitted by Graal
- OKRA passes arguments to HSA Runtime and runs kernel



HOW IT WORKS

- APUs have dozens to hundreds of GPU cores
  - HSA workitem id is used as array index for each GPU core
  - Each core does one workitem per wavefront
  - Think of it as hundreds of threads, each running one function per invocation

- This JDK allows `IntStream` or `Object Array/Vector/ArrayList` stream offload
  - We added an extra class into `j.u.stream` to handle our extra stream processing
  - Stream source object array passed as hidden parameter to HSA
    - Object Stream kernel receives array ref and uses work item id as array index
    - Regular CPU lambda code receives Object as its parameter
  - `IntStream` range comes from HSA workitem id itself

- Collect the lambda target method at `ForEachOp` diversion point
  - Send lambda method to Graal HSAIL compiler
  - Graal emits HSAIL text then sent to HSA Finalizer for kernel creation
  - Kernel is cached for subsequent executions

MORE DETAILS

- Lambda arguments collected from consumer object created by stream API
  - Captured args passed as parameters to HSA kernel same as CPU code

- Referenced fields are accessed through memory ops like CPU-compiled methods
  - Offsets into objects computed by Graal same as CPU codegen

- Static fields accessed through JNI indirect reference
  - No finalized code patching at this time, so no GC-changeable embedded constants

- OKRA is a temporary interface to interact with HSA Runtime
  - Java thread calls our OKRA JNI code and blocks while kernel runs
  - OKRA is designed to work well with the HSA simulator

`IntStream` EXAMPLE

- Offload baseball statistics using `IntStream`
  - Player objects have accessors for various stat categories
  - Calculate the batting average for each player
  - `IntStream.forEach` lambda code in red is converted to HSA kernel

```
Player[] players = Arrays.stream(allHitters).map(objMapper).
        parallel().sorted().toArray(Player[]::new);

IntStream.range(0, players.length).parallel().forEach(n -> {
    Player p = players[n];
    if (p.getAb() > 0) {
        p.setBa((float)p.getHits() / (float)p.getAb());
    } else {
        p.setBa((float) 0.0);
    }
});
```

HSAIL FOR `IntStream` LAMBDA FROM GRAAL

```
version 0:95: $full : $large;
// static method HotSpotMethod<Main.lambda$7(Player[], int)>
kernel &run (
          kernarg_u64 %_arg0
          ) {
          ld_kernarg_u64  $d6, [%_arg0];          // Captured array ref
          workitemabsid_u32 $s2, 0;               // work item id is a gpu idiom

@L4:      ld_global_s32 $s0, [$d6 + 16];          // load array length
          cmp_ge_b1_u32 $c0, $s2, $s0;            // compare length to workitemid
          cbr $c0, @L5;                           // return if greater
@L6:      cvt_s64_s32 $d0, $s2;
          mul_s64 $d0, $d0, 8;                    // convert work item into array index
          add_u64 $d3, $d6, $d0;
          ld_global_u64 $d0, [$d3 + 24];          // load player object
          mov_b64 $d3, $d0;
          ld_global_s32 $s3, [$d0 + 20];          // this is inlined getAb()
          cmp_lt_b1_s32 $c0, 0, $s3;              // if (p.getAb() > 0)
          cbr $c0, @L7;
@L8:      mov_b32 $s16, 0.0f;
          st_global_f32 $s16, [$d0 + 76];          // p.setBa((float) 0.0);
@L9:      ret;
@L7:      ld_global_s32 $s1, [$d0 + 28];          // inlined getHits()
          cvt_f32_s32 $s16, $s1;                   // cast (float)p.getHits()
          cvt_f32_s32 $s17, $s3;                   // cast (float)p.getAb()
          div_f32 $s16, $s16, $s17;               // hits / ab
          st_global_f32 $s16, [$d0 + 76];         // inlined setBa()
          brn @L9;
@L5:      ret;
};
```

SMALL OBJECT STREAM EXAMPLE

- Same example as Object Stream
  - The `Stream.forEach` lambda is converted to an HSA kernel
  - Stream source array is passed as a hidden parameter to kernel

```
Stream<Player> s = Arrays.stream(allHitters).map(objMapper).parallel().sorted();

s.forEach(p -> {
    if (p.getAb() > 0) {
        p.setBa((float)p.getHits() / (float)p.getAb());
    } else {
        p.setBa((float)0.0);
    }
});
```

HSAIL FOR OBJECT STREAM LAMBDA

```
version 0:95: $full : $large;
// static method HotSpotMethod<Main.lambda$3(Player)>
kernel &run (
          kernarg_u64 %_arg0
          ) {
          ld_kernarg_u64  $d6, [%_arg0];          // Hidden stream source array ref
          workitemabsid_u32 $s2, 0;

          cvt_u64_s32 $d2, $s2;         // Convert work item id to long
          mul_u64 $d2, $d2, 8;          // Adjust index for sizeof ref
          add_u64 $d2, $d2, 24;         // Adjust for actual elements data start
          add_u64 $d2, $d2, $d6;        // Add to array ref ptr
          ld_global_u64 $d6, [$d2];     // Load from array element into parameter reg
@L0:
          ld_global_s32 $s0, [$d6 + 20];          // inlined getAb()
          cmp_lt_b1_s32 $c0, 0, $s0;              // if (p.getAb() > 0)
          cbr $c0, @L1;
@L2:
          mov_b32 $s16, 0.0f;
          st_global_f32 $s16, [$d6 + 76];        // p.setBa((float)0.0);
@L3:
          ret;
@L1:
          ld_global_s32 $s3, [$d6 + 28];         // load p.getHits()
          cvt_f32_s32 $s16, $s3;                 // (float) p.getHits()
          cvt_f32_s32 $s17, $s0;                 // (float) p.getAb()
          div_f32 $s16, $s16, $s17;
          st_global_f32 $s16, [$d6 + 76];        // inlined setBa()
          brn @L3;
};
```

CURRENT LIMITATIONS OF HSAIL OFFLOAD DEMO JDK

- Currently not allowed in an offloaded kernel
  - No heap allocation
  - No exception handling or try/catch inside a kernel
  - No calling methods that would be a JNI or runtime call
  - No synchronization in kernels
  - No method handles in target lambda methods

- Kernels are called by JNI code using JNI Critical
  - So no GC during kernel execution
  - No debug info or oop maps for kernels
  - No way to have embedded oops in finalized code like with `nmethods`

FEATURES HOPEFULLY STANDARDIZED IN SUMATRA

- What is the heuristic or coding model for offloading?
  - We chose parallel streams based on our experience with Aparapi and GPUs
  - This model does not require developers to learn new API, etc.
- GC interaction?
  - Possible or worthwhile to have safepoints during kernel execution?
- What runtime calls or allocation from a kernel can be supported?
  - Runtime calls imply pausing the GPU kernel and resuming on the CPU
- Exception handling?
  - Throw inside kernel with its own try-catch block handling it
  - Throw causing kernel abort and handled in runtime on CPU
- What synchronization can be supported in kernels?
  - Between GPU cores
  - Between CPU and GPU

FEATURES HOPEFULLY STANDARDIZED IN SUMATRA

- Details of HSA versus discrete card offload?
  - Copying/replacing buffers to card vs. direct heap access in HSA
  - Any difference in interaction with JVM runtime?
- How to detect and configure various offload runtime systems from Java?
  - HSAIL/BRIG, PTX, etc.
  - Select offload GPU(s) if more than one available

SUMMARY

- We can offload simple JDK 8 Stream API `forEach` lambdas to HSA systems
  – Seamlessly offload normal JDK 8 code
  – No special coding or API required

- Basic HSAIL code generation now in Graal repository

- HSAIL simulator is available and our HSAIL demo JDK supports it
  – Detailed check-out and build instructions on the Sumatra wiki: https://wiki.openjdk.java.net/display/Sumatra/Main

- GPU offload for Java is here
  – GPUs offer unprecedented performance for the appropriate workload
  – Don't assume everything can/should execute on the GPU
  – Look for "islands of parallel in a sea of sequential"

- Lots of work remains!

LINKS AND REFERENCES

- 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
  - Home page: http://hsafoundation.com/
  - Specifications at http://hsafoundation.com/standards/
- AMD Kaveri APU Overview
  - http://www.theregister.co.uk/2013/05/01/amd_huma/

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