SUMATRA NEW FEATURES
COORDINATING BETWEEN THE GPU AND THE JVM

TOM DENEAU

JVM LANGUAGE SUMMIT 2014

AGENDA

- Brief Review of HSA and Sumatra Project Organization

- Look at some features added to Sumatra over the last year that involve cooperation with the JVM
  - Deoptimization
  - Allocation
  - Safepoints

- Along the way, point out challenges presented by HSA targets

- This code is available in the Sumatra JDK and Graal repository
  - http://hg.openjdk.java.net/sumatra/sumatra-dev/jdk/
  - http://hg.openjdk.java.net/graal/graal/

HSA ARCHITECTURE, QUICK OVERVIEW

- Heterogeneous System Architecture standardizes CPU/GPU functionality
- Unified address space for GPU and CPU
  - A pointer is a pointer
- GPU can access pageable system memory using CPU pointers
- Fully cache coherent memory between CPU and GPU
- HSAIL is a virtual ISA for parallel programs
  - Finalized to native GPU ISA at runtime
- Explicitly parallel model
- HSA Foundation Home page: http://hsafoundation.com/

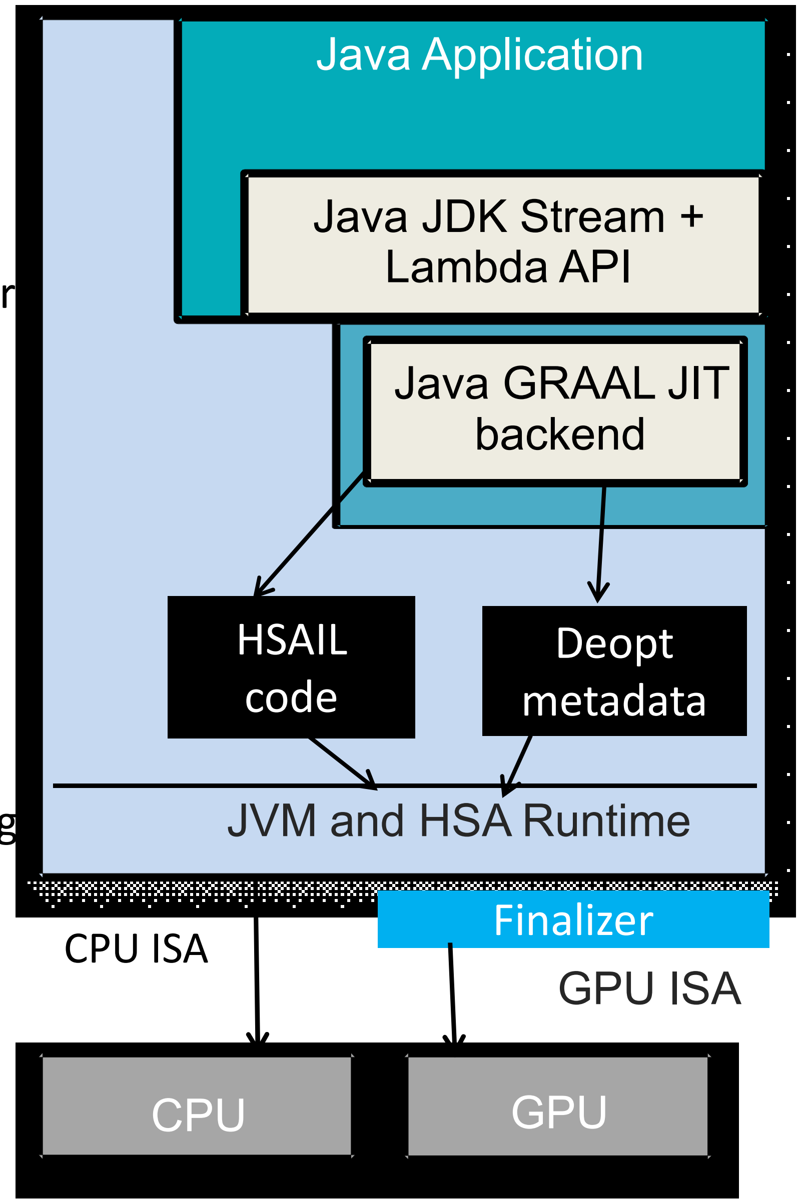
HSA ARCHITECTURE, QUICK OVERVIEW

- Kernel code is dispatched across range of workitems
  - Think of it as hundreds of threads, each running the same function per invocation
  - HSAIL instruction to get workitemabsid, e.g. can be used as array index
  - Example: out[n] = in[n] * in[n];

```
kernel &run(kernarg_u64 %_out,  kernarg_u64 %_in) {
   ld_kernarg_u64 $d0, [%_out];
   ld_kernarg_u64 $d1, [%_in];
   workitemabsid_u32 $s2, 0;  // id for this workitem
   cvt_s64_s32 $d2, $s2;
   mad_u64 $d3, $d2, 4, $d1;
   ld_global_f32 $s3, [$d3];
   mul_f32 $s5, $s3, $s3;
   mad_u64 $d4, $d2, 4, $d0;
   st_global_f32 $s5, [$d4];
   ret;
}
```

SUMATRA PROJECT OVERVIEW

- Using slightly modified JDK (sumatra-dev), 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
- Native code in the JVM
  - Part of Graal's hotspot
  - Passes arguments to HSA Runtime and runs kernel
  - Handles any post-kernel-execution processing



DEOPTIMIZATION

DEOPTIMIZATION, MOTIVATION

- Handle cases where compile time assumptions are violated
  - For example, class hierarchy assumptions
- Throw exceptions back to CPU
  - For instance, null reference in an input object stream
- Handle cases where GPU needs CPU to take some action before proceeding
  - Example: allocation when no space left on heap, GC required
  - Similarly for safepoints

DEOPTIMIZATION OVERVIEW

- If kernel cannot complete on the GPU, revert to the interpreter on the CPU
- As with CPU deoptimization, keep track of reasons for deoptimization
  - Decide whether kernel should be recompiled
  - Or not offloaded in the future.
  - The logic for deciding to recompile or stop offloading is not implemented yet

HSA FEATURES AFFECTING DEOPTIMIZATION DESIGN

- No HSAIL “stack”, registers saved as stores to global memory
- Registers do not live across function calls
  - Register saving must be inlined into each function
- The total number of HSAIL registers used in a kernel affects the code quality when code is finalized
  - Even if registers are used on a "cold" path
  - Thus should not just “save all registers”
- We currently have one deopt exit point per kernel
  - Save the union of the actual registers and stack slots that are live at any of the deopt points

DEOPTIMIZATION, COMPILATION PHASE

- We fully utilize Graal features to provide HSA deoptimization
- Compiler keeps track of the deoptimization state at each deopt point
  - Indicates which HSAIL registers or stackslots need to be saved
  - Indicates which saved values contain oops, etc.

DEOPT, EXECUTION PHASE

AMD

- Kernel dispatch to GPU is done when all workitems have returned
  - It might have set a flag indicating it is deoptimizing and returned early
- Workitems that do not deoptimize have finished as they normally would
- Each deoptimizing workitems must save its state so the interpreter can finish
- Can be very large number of workitems
  - Each of which has its own HSAIL state
  - Because of divergence, different workitems can have different deopt points

```
if (condition) {
   output = myObjArray[k].getValue();    // possible NPE or IndexOutOfBounds
} else {
   output = new MyObj().getValue();      // possible deopt from out of memory.
}
```

DEOPT EXECUTION PHASE, SAVING STATE

- Goal: avoid allocating state-saving space for the entire possibly very large range of workitems
- Compiler tells us size of the total union of saved registers
- Once we allow a workitem to start, we must have room to save its state
  - Because it might deoptimize
- Any HSA target has a maximum number of concurrent workitems
  - Only allocate state-saving space for this maximum number of possible concurrent workitems
- Deopting workitems set a "deopt happened" flag
  - And atomically bump an index of where to save in the save area
- Before beginning execution, workitems check this "deopt happened" flag.
  - If true, exit early without running at all.
  - And just set a flag that they never ran
  - Note: no saved state

DEOPT EXAMPLE

- Workitems 1 and 3 have deopted and saved their state. 2000 and up saw flag set and are never-rans.

id	action	Deopt PC	Saved State		
			$s0	$d2	$d3
0	Finished				
1	Deopted	1111	123	456	789
2	Finished				
3	Deopted	2222	234	567	890
4-1999..	Finished				
2000	NeverRan				
2001	NeverRan				
Etc.	NeverRan				

DEOPT PATH, POST-KERNEL EXECUTION PHASE

- When the GPU dispatch completes (still in thread in VM mode), each workitem will have either:
  - Finished normally
  - Deopted
  - Or exited early (never ran)
- If there was at least one deopt, then:
  - For the workitems that finished normally, there is nothing additional to do
  - For each deopting workitem, run through the interpreter starting from the byte code index of the deoptimization point
  - For each never-ran workitem, run on the CPU as a javaCall of the kernel method
  - Deopts and never-rans handled sequentially, although other policies are possible

SAVED HSAIL STATE TO THE INTERPRETER

- Workitems that deoptimize must run in the interpreter from the deopt bytecode
  - Using the per workitem saved state to fill in locals, bytecode stack, etc.
- We want to have a CPU stack with frames that the interpreter can handle
- Gilles Duboscq of the Graal team had a clever idea
  - Compile a deopt trampoline code graph, use it to generate host code
  - The trampoline host code takes the hsail deoptId and a pointer to the saved hsail frame as input
  - Generate code for each possible deopt id in this kernel to:
    - Move from each saved HSAIL register representing a local into a host local
    - Immediately deoptimize
  - It is this trampoline code that we invoke for each deoptimizing workitem

DEOPTIMIZATION AND GC

- While kernel is running, we are in "thread in VM" mode
  - So GC cannot happen
- However, post-kernel we run each deopting workitem through the interpreter
  - Transition back to Java mode which can cause GCs
- Oops in save area are properly handled as part of GC oops_do workflow

ALLOCATION FROM THE GPU

ALLOCATION FROM THE GPU, MOTIVATION

- Enable offload to the GPU lambda expressions that use object allocation
- Since HSA Device has a coherent view of system memory including Java Heap, this should be possible
  - GPU can access objects on heap
  - Can access data structures that control the heap
- Note: Graal compiler will avoid the actual allocation if can prove that the allocated objects do not escape
- This section will deal with allocating objects that really do escape

ALLOCATION IMPLEMENTATION, SNIPPETS

- Graal has a feature called Snippets
- In a lowering phase, nodes are replaced with the graph of the snippet
  - Which is written in java
- For HSAIL object allocation, we made our own modified versions of the standard object allocation snippets
- Snippets allowed us to write our new allocation fastpath and slowpath logic in java

ALLOCATION, FAST PATH

- In Hotspot JVM, each Java thread uses its own use Thread Local Allocation Buffers (TLAB) to allocate memory on the heap
- Important TLAB fields:
  - HeapWord* _start; // start of TLAB
  - HeapWord* _top; // address after last allocation, bumped with each allocation
  - HeapWord* _end; // end of TLAB

SHARING OF TLABS

- A TLAB per workitem would have meant too many TLABs
- Thus, we allow multiple workitems to allocate from a single TLAB
- HSAIL Kernels use special TLABs that are not used for allocation by any JavaThreads
- In other respects, treated like normal TLABs

ALLOCATION FAST PATH

- Normal single owner TLAB fastpath (as on CPU)

```
If (top + size < end) {
    oldtop = top;
   top += size;
   return oldtop;
}
```

- HSAIL Multiple owner TLAB fastpath

```
oldtop = atomic_get_and_add(top, size);
If (oldtop + size < end) {
   return oldtop;   // fastpath success
} else {
   if (oldtop < end) {
      // overflowed but we were first overflower
      lastGoodTop = oldtop;
      //  continue slowpath, other workitems using this TLAB will also overflow
   }
}
```

ALLOCATION FAST PATH

- HSAIL-specific code here is really just the use of atomic_add on tlab.top
- The other logic in the fastpath allocation (formatting object, etc.) inherits from its superclass NewObjectSnippets

HSAIL ALLOCATION SLOW PATH

AMD

- What to do when the current TLAB overflows
- High level choices
  - Give up and deoptimize immediately
  - Do an “eden allocate” using CAS
    - Lots of CAS contention when many workitems overflow
  - Designate one workitem to allocate a new TLAB while still on the GPU
    - Other workitems overflowing on same TLAB will spin waiting for new TLAB
    - This is the current default

ALLOCATION SLOW PATH REFILL TLAB FROM GPU

- Several workitems overflow on TLAB, one needs to be the refiller

- TLAB refiller is the "first overflower" workitem as described above
  - Allocates TLAB from Eden using CAS

- Other workitems then wait for the first overflower to indicate that a new TLAB is ready

- If designated refiller cannot get a new TLAB
  - Workitems waiting on that TLAB will deoptimize
  - GC will presumably open up some space.

ALLOCATION EXAMPLE

AMD

- 1 TLAB shared by 8 workitems: top=1000, end=1450
- Each workitem needs 100 bytes

Fast Path

id	allocation	New top
1	1300	1400
2	1000	1100
3	X	1500
4	1100	1200
5	X	1600
6	1200	1300
7	X	1700
8	X	1800

Workitem 3, first overflower, gets new TLAB starting at 8000. 5,7,8 also overflow and wait for new TLAB then 3,5,7,8 try again

Slow Path

id	allocation
1	1300
2	1000
3	8100
4	1100
5	8000
6	1200
7	8200
8	8300

SAFEPOINTS

HSAIL SAFEPOINTS, MOTIVATION

- While kernel is running we are in "thread in VM" mode
  - So GC or other safepoints cannot happen
- Other Java threads (on CPU) that need to safepoint will be blocked waiting for GPU kernel thread
- Kernels can be long-running
  - Because they have a large grid size (lots of workitems)
  - Because each individual workitem is long running

HSAIL COMPILER SAFEPOINTS

- A global flag is set by the JVM indicating a safepoint is requested
- Checks of this flag are inserted in the HSAIL code by compiler at usual places, such as loop ends, etc.
  - The HSAIL backend implementation of Graal's SafePointNode
- When flag is seen set, workitem will deoptimize
- Post-kernel, during deoptimization handled in interpreter,
  - real safepoint takes place and GC or other activity can proceed
- Safepoint requested flag is also checked at workitem entry, producing a never-ran if detected

FUTURE WORK AND CONCLUSIONS

- Use CPU parallelism when handling deopters and never-rans
  - Or redispatch never-rans to GPU
- Investigate ways to spin on GPU rather than deopting for safepoints, GC
- Recompile or stop offloading for certain deoptimizations
- Investigate other shared TLAB allocation policies not requiring atomics

CONCLUSION

- With a coherent view of system memory, HSA targets can interact with the JVM in exciting ways that we have seen here.

LINKS AND REFERENCES

AMD

- 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/

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