HSA AND OPENCL 2.0
STATUS UPDATES
HAIBO XIE, PH.D.
DEC 19, 2014
HSA, where we are now
HSA FOUNDATION

HSA Foundation

- Not-for-profit industry standards body focused on making it dramatically easier to program heterogeneous computing devices
An *intelligent computing architecture* that enables CPU, GPU and other processors to work in *harmony* on a single piece of silicon by *seamlessly* moving the right tasks to the best suited processing element.
DISCRETE CPU AND GRAPHICS WITH GPU COMPUTE

- Compute acceleration works well for large offload
- Slow data transfer between CPU and GPU
- Expert programming necessary to take advantage of the GPU compute
FIRST AND SECOND GENERATION APUs

- First integration of CPU and GPU on-chip
- Common physical memory but not to programmer
- Faster transfer of data between CPU and GPU to enable more code to run on the GPU
Unified **Coherent** Memory enables all compute elements to have access to the same data

- CPU and GPUs architected to operate interchangeably
- Enables the application to run on the best compute element (CPU or GPU)
THE COMPILATION STACK
WILL SUPPORT HSAIL JIT’ED & NATIVE COMPILATION

- Front End
  - CLANG (C++, OpenCL)
  - LLVM
- Offline Compiler Process
- Low Level IR
- Brig (HSAIL and/or Native ISA)
- Finalize to Native ISA
- Hardware

- Native GPU ISA
  - ELF (Native ISA)
  - ELF Native Loader
  - Application Cache
  - Driver
  - Shader Compiler (First Compile)
  - Brig (HSAIL)
HSA DEVELOPMENT CAN START TODAY

<table>
<thead>
<tr>
<th>HSA Hardware Building Blocks</th>
<th>HSA Software Building Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Shared Virtual Memory</td>
<td>▶ HSAIL</td>
</tr>
<tr>
<td>- Single address space</td>
<td>- Portable, parallel, compiler IR</td>
</tr>
<tr>
<td>- Coherent</td>
<td>- Assembler</td>
</tr>
<tr>
<td>- Pageable</td>
<td>▶ HSA Runtime</td>
</tr>
<tr>
<td>- Fast access from all components</td>
<td>- Create queues</td>
</tr>
<tr>
<td>- Can share pointers</td>
<td>- Allocate memory</td>
</tr>
<tr>
<td>▶ Architected User-Level Queues</td>
<td>- Device discovery</td>
</tr>
<tr>
<td>▶ Signals</td>
<td>▶ LLVM Reference compiler</td>
</tr>
<tr>
<td></td>
<td>- CLANG/LLVM Reference Compiler</td>
</tr>
<tr>
<td></td>
<td>- HSAIL Code generation</td>
</tr>
<tr>
<td></td>
<td>- OpenCL, C++ AMP via CLANG</td>
</tr>
</tbody>
</table>

Provide industry-standard, architected requirements for how devices share memory and communicate with each other

Provide industry-standard compiler IR and runtime to enable existing programming languages to target the GPU
HSA FEATURES OF KAVERI

UNLOCKING ALL OF KAVERI’S GFLOPS

AN Unlocking all of Kaveri’s compute power

APU GFLOPS

GPU GFLOPS

CPU GFLOPS

ALL-PROCESSORS-EQUAL

GPU and CPU have equal flexibility to be used to create and dispatch work items

EQUAL ACCESS TO ENTIRE MEMORY

GPU and CPU have uniform visibility into entire memory space

Access to full potential of Kaveri’s APU compute power
HSA DEVELOPMENT CAN START TODAY
KAVERI AND FUTURE – HSA COMPLIANT APU

**Physical Integration**
- Integrate CPU & GPU in silicon
- Unified Memory Controller
- Common Manufacturing Technology

**Optimized Platforms**
- GPU Compute C++ support
- User mode scheduling
- Bi-Directional Power Mgmt between CPU and GPU

**Architectural Integration**
- Unified Address Space for CPU and GPU
- GPU uses pageable system memory via CPU pointers
- Fully coherent memory between CPU & GPU

**System Integration**
- GPU compute context switch
- GPU graphics pre-emption
- Quality of Service
AMD IS DELIVERY HSA SOFTWARE VIA OPEN SOURCE

- Delivering the HSA first via Linux
  - Driver and Runtime will support Fedora and Ubuntu

- HSA Runtime, Driver and Compiler with delivered as binary and as open source
  - Binaries: Fedora RPM and Ubuntu as Debian Packages
  - Compiler and Runtime Source will be delivered via UIUC license
  - Linux Kernel Driver components under GPL2 License

- Delivering LLVM Based Native and HSAIL Code Reference Compiler for customer compiler development
  - Got to https://github.com/HSAFoundation/HSA-Docs-AMD/wiki to get access
BUILDING OUT KEY TOOLS FOR HSA

▲ Available as Beta Oct 16th
– LibHSAIL Assembler and Dissembler tools
– HSAIL Simulator - HSAIL Instruction Set Simulator
– CLOC : CL Offline Compiler. Compile CL kernels to HSAIL.
– SNACK
– C++ AMP Compiler in CLANG/LLVM based
– Java / Aparapi : Allows programs written in Java to be accelerated on the GPU. Aparapi is a Java library that converts Java bytecode to HSAIL. Aparapi enablement on HSA.

▲ In Development
– OpenMP/OpenACC compiler from Pathscale with C/C++ & Fortran support
– Java / Sumatra : Allows programs written in Java to be accelerated on the GPU. Sumatra is a modified JVM which generates HSAIL. Sumatra enablement on HSA is on-going.
Java 8 adds Stream, Lambda APIs
- CPU Multicore Parallelism
- APARAPI on HSA accelerates Lambdas
  - Parallel acceleration on HSA APU

Java 9 — HSA enabled Java (SUMATRA)
- Adds native APU acceleration to Java Virtual Machine (JVM)
- Developer uses Lambda, Stream API
- JVM generates HSAIL automatically
OpenMP 4.0 & OpenACC 1.0/2.0 with C, C++, Fortran. support coming for all
- Hawaii and Tahiti-based FirePro cards
- S10000, S9150/S9100, S9050/S9000, W9100, W9000, W8100, W8000
- AMD Kaveri via HSA Runtime

Exposes GPU compute capability for new markets

Supported via partnership with Pathscale

“Pathscale’s ENZO Compiler with OpenMP 4.0 and support for C, C++, and Fortran is used by customers in the Oil and Gas, Computational Science, Computer Aided Engineering, and other HPC segments. Our support for the AMD FirePro S9150 will enable these customers to benefit from the tremendous compute performance of the S9150 while leveraging their existing investment in OpenMP software.”

Christopher Bergstrom, CTO, Pathscale
OPENACC SUPPORT VIA PGI COMPILERS

△ Supports OpenACC 2.0 features. Full support for the OpenACC 1.0 specification via C, C++, Fortran
– PGI Server with Acceleration option
– PGI Workstation with Acceleration option
– VIA OpenCL SPIR 1.2 Runtime
WHAT IS SNACK? Simple No Api Compiled Kernels

- Accelerator programming model where accelerated functions are directly called from main program without API such as CUDA or OpenCL.

- Possible with HSA and new compilation process that generates user-function wrapper to launch compiled GPU kernel.

- Application program can be built with ANY compiler that uses standard linkage conventions.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hw.h"

int main() {
    char* input = "Gdkkn\x1FGR@\x1FVnqkc";
    size_t strlength = strlen(input);
    char *output=(char*)malloc(sizeof(char)*strlength+1);
    Scope_t scope = {.ndim=1,
                     .gdims={strlength},.ldims={1} };

    hw(input,output,scope);
    output[strlength] = '\0';
    printf("%s\n",output);
    free(output);
    return 0;
}

__kernel void hw(__global char* in, __global char* out) {
    int num = get_global_id(0);
    out[num] = in[num] + 1;
}
```

CLOC: CL offline compiler creates object code and header files for main program.
WHAT IS CLOC?

- cloc = CL offline compiler.
- Support OpenCL 2.0 Kernels with support HSA runtime Libraries
- cloc is the command line interface to create hsail, brig, or object from CL with the HSAIL LLVM backend compiler.
  - Supports LDS (Local Data Share)
  - Supports atomics between GPU thread memory references.
  - Supports GPU thread synchronization
  - Supports atomic operations as long as host language supports it. Example c++11 and c11 compatibility.
  - Sequential Consistency to implement release consistency
  - All thread scope operations
  - Supports calling SNACK functions from c, c++, and FORTRAN applications GCC, CLANG/LLVM.
  - Object linking to main application code

See https://github.com/HSAFoundation/CLOC
New features give GPUs more freedom to do the work they are designed to do.

- **Shared Virtual Memory**: Enables host and device kernels to directly share complex pointer-based data structures, eliminating explicit transfers between the host and devices while increasing programming flexibility.

- **Nested Parallelism**: Updated for improved programmability and increased application efficiency.

- **Generic Address Space**: Enables functions to be written without named address spaces which increases flexibility and saves time by eliminating the need for multiple functions to be written.
ACCESS HSA

- Get HSA stack from https://github.com/HSAFoundation/
- Get HAS specifications from http://www.hsafoundation.com
- These specifications are at the "1.0 Provisional" Level and are available now
  - HSA Platform System Architecture Specification: Defines the requirements for shared virtual memory, platform coherency, signaling, queuing mechanics and packet formats, context switching, and the HSA memory model.
  - HSA Runtime Programmer’s Reference Manual: Defines the APIs in the HSA Runtime used for tasks such as initialization and device discovery, queue creation, and memory management.
OpenCL 2.0 support from AMD
OPENCL 2.0 IS HERE
FEEL OPENCL 2.0 FROM AMD PRODUCT

△ HW: Basically all GCN-based GPU is ready for OpenCL 2.0
  – AMD FirePro W5100, AMD FirePro W9100, AMD FirePro S9150
  – AMD Radeon HD 7790 and above, AMD Radeon R5 M240 and above
  – A-Series AMD Radeon R4 Graphics and above

△ OS
  – Windows 7 (32/64bit) and above
  – Linux (OpenSUSE 13.1, Ubuntu 14.04 LTE, Red Hat Enterprise 7.0, 6.4, 6.5)

△ Driver
  – Catalyst Omega driver (14.501.1003)

△ Tools
  – CodeXL 1.6 is available now for OpenCL 2.0

△ Resources
  – http://developer.amd.com/
WHAT YOU CAN GET FROM OPENCL 2.0

- Shared virtual memory (SVM)
- Generic address space
- Device-side enqueue
- Atomics and synchronization
- Pipes
- Sub-groups
- Program-scope global variables
- Image enhancements
- Non-uniform work group size
SHARED VIRTUAL MEMORY

OCL 1.2 era
- CPU and GPU can’t share same virtual memory
- Introduce explicit data copy then performance and power penalty

OCL 2.0 era
- CPU and GPU can share the same virtual memory
- May or may not share actual physical memory, depending on the hardware architecture
- Three levels of SVM
  - Coarse-grained buffer SVM
  - Fine-grained buffer SVM
  - Fine-grained system SVM

Usage scenarios
- For those Kernel instance need process pointer-based data structure, like linked-list or tree
# SHARED VIRTUAL MEMORY

<table>
<thead>
<tr>
<th></th>
<th>Granularity of sharing</th>
<th>Memory Allocation</th>
<th>Mechanisms to enforce Consistency</th>
<th>Explicit updates between host and device?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse-Grained buffer SVM</td>
<td>OpenCL Memory objects (buffer)</td>
<td>clSVMAlloc</td>
<td>Synchronization points</td>
<td>yes, through Map and Unmap commands.</td>
</tr>
<tr>
<td>Fine Grained buffer SVM</td>
<td>Bytes within OpenCL Memory objects (buffer)</td>
<td>clSVMAlloc</td>
<td>Synchronization points plus atomics (if supported)</td>
<td>No</td>
</tr>
<tr>
<td>Fine-Grained system SVM</td>
<td>Bytes within Host memory (system)</td>
<td>Host memory allocation mechanisms (e.g. malloc)</td>
<td>Synchronization points plus atomics (if supported)</td>
<td>No</td>
</tr>
</tbody>
</table>
Memory layout resign in OCL 1.2 introduce penalty, even worse than CPU did.
Memory layout resigned in OCL 1.2 introduce penalty, even worse than CPU did.

| Tree (size in M) | CPU – time (ms) | GPU (OpenCL 2.0) | GPU (OpenCL 1.2) *
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23.46</td>
<td>5.17</td>
<td>3.22 (+1.92) (+49.58 +8.50)</td>
</tr>
<tr>
<td>5</td>
<td>86.11</td>
<td>24.87</td>
<td>13.95 (+4.15) (+259.12 +34.30)</td>
</tr>
<tr>
<td>10</td>
<td>180.73</td>
<td>51.58</td>
<td>N/A</td>
</tr>
<tr>
<td>25</td>
<td>381.77</td>
<td>129.58</td>
<td>N/A</td>
</tr>
</tbody>
</table>
OCL 1.2 era
- The parallelism granularity is fixed before the Kernel instance launch
- CPU has to interact for different Kernel with different numbers of thread. Then penalty

OCL 2.0 era
- Device kernels can enqueue kernels to the same device with no host interaction, enabling flexible work scheduling paradigms and avoiding the need to transfer execution control and data between the device and host, often significantly offloading host processor bottlenecks

Usage scenarios
- Nested parallel work load
- Make it easier to implement recursive algorithms, nested loop structures, etc..
  - Ex: Spawning a new kernel to explorer a sub-graph in a graph algorithm.
GENERIC ADDRESS SPACE

⚠ OCL 1.2 era
– Each pointer parameter in a function definition must have an associated address space
– You must create an explicit version of the function for each desired address space
  void fooL (local int *p) { ... }
  void fooP (private int *p) { ... }
  void fooG (global int *p) { ... }

⚠ OCL 2.0 era
– Data location is spotted by the compiler at compiling time
– So you only need
  void foo (int *p) { ... }

⚠ Usage scenarios
– Easier programming
– Easier data structure definition
PROGRAM-SCOPE GLOBAL VARIABLES OVERVIEW

▲ OCL 1.2 era
– OpenCL 1.2 permits declaration of only constant-address-space variables at program scope

▲ OCL 2.0 era
– OpenCL 2.0, allows you to declare variables in the global address space at program scope (i.e., outside function scope)

▲ Usage scenarios
– By using program-scope variables, you can potentially eliminate the need to create buffers on the host and pass them into each kernel for processing
NONUNIFORM WORKGROUP SIZE

✈️ OCL 1.2 era
- Each workgroup size had to divide evenly into the corresponding global size

✈️ OCL 2.0 era
- The final workgroup in each dimension can be smaller than all of the other workgroups in the “uniform” part of the NDRange

✈️ Usage scenarios
- Reduce the effort you’ll require to map problems onto NDRanges
- Easier programming
OPENCL 2.0 TAKE AWAYS

- OpenCL 2.0 is here now
  - Check your hardware and driver to enjoy
- OpenCL 2.0 improved programmability
- OpenCL 2.0 covered more usage scenarios to make GPGPU more pervasive

### Shared Virtual Memory
Enables host and device kernels to directly share complex pointer-based data structures, eliminating explicit transfers between the host and devices while increasing programming flexibility.

### Nested Parallelism
Updated for improved programmability and increased application efficiency.

### Generic Address Space
Enables functions to be written without named address spaces which increases flexibility and saves time by eliminating the need for multiple functions to be written.
THANK YOU
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

© 2013 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.
HSAIL CODE EXAMPLE

01: version 0:95: $full : $large;
02: // static method HotSpotMethod<Main.lambda$2(Player)>
03: kernel &run (  
04:     kernarg_u64 `%_arg0` // Kernel signature for lambda method  
05: ) {  
06:     ld_kernarg_u64 $d6, `%_arg0`; // Move arg to an HSAIL register  
07:     workitemabsid_u32 $s2, 0; // Read the work-item global “X” coord  
08:   
09:     cvt_u64_s32 $d2, $s2; // Convert X gid to long  
10:     mul_u64 $d2, $d2, 8; // Adjust index for sizeof ref  
11:     add_u64 $d2, $d2, 24; // Adjust for actual elements start  
12:     add_u64 $d2, $d2, $d6; // Add to array ref ptr  
13:     ld_global_u64 $d6, [$d2]; // Load from array element into reg  
14:   @LO:  
15:     ld_global_u64 $d0, [$d6 + 120]; // p.getTeam()  
16:     mov_b64 $d3, $d0;  
17:     ld_global_s32 $s3, [$d6 + 40]; // p.getScores ()  
18:     cvt_f32_s32 $s16, $s3;  
19:     ld_global_s32 $s0, [$d0 + 24]; // Team getScores()  
20:     cvt_f32_s32 $s17, $s0;  
21:     div_f32 $s16, $s16, $s17; // p.getScores()/teamScores  
22:     st_global_f32 $s16, [$d6 + 100]; // p.setPctOfTeamScores()  
23:     ret;  
24:   };

36 | HSA and OpenCL 2.0 – status updates | DECEMBER 22, 2014 |
HSA IS SUPPORTING RICHER PROGRAMING LANGUAGE'S

- Bring power&perf benefits of accelerated computing to the programmer

- Broader Set of Languages are support
  - C++AMP “parallel_for_each” + functor
  - Java:8 streams + lambda API
  - OpenMP : #pragma omp parallel for via OpenMP 4.0
  - OpenACC 2.0: Directives
  - Bolt: C++ template library calls
  - OpenCL 2.0 Richer foundation with SVM and Platform Atomics
  - Python Fast Iterative Development

- Support via via key open source compiler technologies
  - CLANG/LLVM for to allow you to develop your own compiler to support Domain Specific Languages and other languages
  - GCC via GIMPLE to HSAIIL – first compiler will be OpenMP 4.0 based
DECLARATION

Part of the figures and statements contributed by HSA Foundation members:

*Phil Rogers, AMD*

*Ben Sander, AMD*

*Yeh-Ching Chung, National Tsing Hua University*

*Benedict Gaster, Qualcomm*

*Hakan Persson, ARM*

*Wen Mei Hwu, Universities of Illinois*

Take a look at:

OPENCL 2.0 – SHARED VIRTUAL MEMORY (SVM)

Summary

- Allows the GPU to share data pointer-based data structures with the CPU
  - Example: construct a b-tree on CPU, update the b-tree on the GPU and then hand it back to the CPU
- Without SVM, GPUs don’t understand CPU pointers
  - Programmers have to translate pointer based data structure into a format consumable by the GPU
  - Overhead in translation, redundant copies, poor developer experience

3 levels of SVM

- Coarse-Grained buffer (Core feature of OpenCL 2.0)
  - Allocated with OpenCL API (clSVMAlloc)
  - Memory consistency enforced at synchronization points (ex: end of kernel) with map/unmap

- Fine-Grained buffer (Optional feature)
  - Allocated with clSVMAlloc
  - Memory consistency guaranteed at synchronization points
  - C1X atomics to synchronize between CPU and GPU (optional feature, HSA device)

- Fine-Grained system (Optional feature)
  - Same functionality as Fine-Grained buffer
  - Extend to the entire CPU memory region (ex: malloc)
OPENCL 2.0 – C++11 ATOMICS

Summary
- A subset of C11 atomics and synchronization operations to enable assignments in one work-item to be visible to other work-items in a work-group, across work-groups executing on a device or for sharing data between the OpenCL device and host

OpenCL implements a relaxed memory consistency model
- Workitems/threads may observe memory ops in different orders
  - Thread 1 writes to memory location A, then writes to B
  - Thread 2 may see to B before seeing the update to A.
- No guarantee when memory writes become visible to other workitems or devices
- New atomic functions allow developers to enforce orders of certain memory operations

New OpenCL atomics are based on C11 atomics
- Memory order (relaxed, acquire, release, acquire&release, sequential consistency)
- Memory scope (work item, work group, device, all svm devices)
- Memory space (local, global)

Allow fine grain synchronization between different work items, work groups, kernel instances, devices
- Ex: concurrently update a linked-list or b-tree by different workitems or workgroup or kernels
- Synchronization between CPU and GPU is an optional feature only available for HSA device.
Summary

- Functions can be written without specifying a named address space for arguments. This is especially useful for those arguments that are declared to be a pointer to a type, eliminating the need for multiple functions to be written for each named address space used in an application.

OpenCL device memory consists of different disjoint memory regions (global, local, private).

Programmers often have to duplicate codes to handle pointers from different memory regions.

```c
int add_local(__local int* p, __local int* q) { return *p + *q; }
int add_global(__global int* p, __local int* q) { return *p + *q; }
int add_local_global(__local int* p, __global int* q) { return *p + *q; }
```

Generic address space provides a single address space covering global, local and private.

- With the example above, one version is needed.
- int add(int* p, int* q) { return *p + *q; }
- Avoid duplication of code, better developer experience.
- A generic address space pointer can be cast to a pointer to global, local or private.
- A pointer to global, local or private can be cast to a generic address space pointer.
- A pointer to global, local or private can be implicitly converted to a generic address space pointer, but not the converse.
OPENCL 2.0 – DEVICE SIDE KERNEL ENQUEUE

Summary

- Device kernels can enqueue kernels to the same device with no host interaction, enabling flexible work scheduling paradigms and avoiding the need to transfer execution control and data between the device and host, often significantly offloading host processor bottlenecks
  - Enable nested parallel work load
  - Avoid the transfer of control back to the CPU in order to create new workloads
  - Allow creating additional parallel workloads based on runtime decision and data local to a workitem
  - Make it easier to implement recursive algorithms, nested loop structures, etc..
    - Ex: Spawning a new kernel to explore a sub-graph in a graph algorithm.

Kernel being enqueued by another kernel is written in Block syntax

Flag to control when a child kernel starts execution (no_wait, wait_kernel, wait_work_group)

Device queue, event and event related built-in functions are available in the kernel for managing computation/kernel enqueueing flow.
OPENCL 2.0 – PIPES

Summary
- Pipes are memory objects that store data organized as a FIFO. OpenCL 2.0 provides built-in functions for kernels to read from or write to a pipe, providing straightforward programming of pipe data structures that can be highly optimized by OpenCL implementers
  - Only a kernel can read from or write to a pipe
  - Not accessible from the host/CPU

- Suitable for inter-kernel communication or for constructing a processing pipeline
  - Ex: data analytics pipeline using pipes to pass data from one filter/model to another

- A pipe packet is defined to be a data element in a pipe. They packet type could be:
  - Any OpenCL data type (scalar or vector)
  - A user defined type

- OpenCL provides a set of new builtins for pipe objects
  - Reading or writing
  - Reserving a number of packet entries for reading or writing
  - Querying for the number of packets available
OPENCL 2.0 – OTHER FEATURES

- Images – Support for 2D image from buffer
- Program scope variables in global address space
- New built-in functions
  - Address space qualifier functions
  - New work group functions (predicate, broadcast, reduce, scan)
- Support images with the read_write qualifier
<table>
<thead>
<tr>
<th>Programming Technique</th>
<th>USE CASE DESCRIPTION</th>
<th>HSA ADVANTAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Pointers</strong></td>
<td><strong>Binary tree searches</strong>&lt;br&gt;GPU performs searches in a CPU created binary tree</td>
<td>GPU can access existing data structures containing pointers&lt;br&gt;Higher performance through parallel operations</td>
</tr>
<tr>
<td><strong>Platform Atomics</strong></td>
<td><strong>Binary tree updates</strong>&lt;br&gt;CPU and GPU operating simultaneously on the tree, both doing modifications</td>
<td>CPU and GPU can synchronize using Platform Atomics&lt;br&gt;Higher performance through parallel operations</td>
</tr>
<tr>
<td><strong>Large Data Sets</strong></td>
<td><strong>Hierarchical data searches</strong>&lt;br&gt;Applications include object recognition, collision detection, global illumination, BVH</td>
<td>GPU can operate on huge models in place&lt;br&gt;Higher performance through parallel operations</td>
</tr>
<tr>
<td><strong>CPU Callbacks</strong></td>
<td><strong>Middleware user-callbacks</strong>&lt;br&gt;GPU processes work items, some of which require a call to a CPU function to fetch new data</td>
<td>GPU can invoke CPU functions from within a GPU kernel&lt;br&gt;Simpler programming does not require “split kernels”&lt;br&gt;Higher performance through parallel operations</td>
</tr>
</tbody>
</table>