<table>
<thead>
<tr>
<th>Time (24h)</th>
<th>Topics (no exact times)</th>
</tr>
</thead>
</table>
| 8:50-9:30     | Gathering up  
Intro: The rationale for HSA: GPU's, DSPs and so much more  
The architecture pillars of HSA, Shared memory (SVM), The memory model of HSA,  
Signaling & Atomics, etc. |
| 9:30-10:00    | Dispatch/execution model, Queues and AQL, The system software: kernel driver and runtime,  
HSAIL, Finalizer, BRIG                                                     |
| 10:00-10:30   | **Coffee Break**                                                                                                                                         |
| 10:30-11:00   | Integration of HSA platform features, Interoperation with other devices in the system                                                                  |
| 11:00-11:30   | System virtualization, Platforms and benchmarks: the power of open platforms and open software                                                             |
| 11:30-12:00   | A comparison with other data parallel system architectures, System Architecture research opportunities                                                        |
| 12:00-12:15   | Q & A                                                                                                                                                   |
| 12:15-13:50   | Lunch                                                                                                                                                   |
# The Agenda – Software & Toolchains (Afternoon)

<table>
<thead>
<tr>
<th>Time (24h)</th>
<th>Topics (no exact times)</th>
</tr>
</thead>
</table>
| 13:50-14:30  | Gathering up  
Intro: The rationale for HSA: GPU’s, DSPs and so much more,                                                                                                                                                       |
| 14:30-15:00  | HSA software toolchains:  
LLVM, GCC, HCC, Python  
Integrating HSAIL into a new toolchain, experiences and gotcha’s using                                                                                                                                            |
| 15:00-15:35  | BRIG, HSAIL code generation, debugging metadata  
Debugging and profiling an HSA enabled application using these toolchains with CodeXL or gdb, HSA tools extensions for ROCm and CodeXL                                                                 |
| 15:35-15:50  | Coffee Break                                                                                                                                                                                                            |
| 15:50-16:30  | Application frameworks using  
HSA/ROCR: CAFFE, TORCH, SPARK, …  
Application containers (Docker) Software models,                                                                                                                                                                     |
| 16:30-17:00  | Software research opportunities  
Q & A                                                                                                                                                                                                                       |
The Platform Architecture
End users require applications with accelerators running faster at lower power.

Always on, visually aware devices will offer greater capability in a lower power budget, scaling with every advance in app processing.

Mobile and tablet devices will use the CPU, GPU and DSP working seamlessly together for content creation, gaming and more.

Intelligent cloud video analytics are more efficient, and make best use of every server upgrade.

Sophisticated ADAS real-time analytics will be easier to develop, adapt to any platform, and be more robust.

HSA architecturally integrates the accelerators in today’s complex SoCs to be easily and efficiently utilized by application developers.
Driving New Innovation

HBM
High Bandwidth Memory

CoWoS

The first GPU devices to use HBM and CoWoS
Radeon R9 Fury X, June 2015

Extreme
Performance/Watt

Radeon R9 Nano
46 GFLOP/Watt
Sept 2015

Open GPU Computing Platform

ROCM 1.0
April 2016

14 NM

Radeon RX480
June 2016
The vision

MAKE HETEROGENEOUS PROGRAMMING MUCH EASIER

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single source programming in common high-level languages</td>
</tr>
<tr>
<td>2</td>
<td>Enable the programming language(s) of the developer</td>
</tr>
<tr>
<td>3</td>
<td>Eliminate data copies</td>
</tr>
<tr>
<td>4</td>
<td>Common address space</td>
</tr>
<tr>
<td>5</td>
<td>Standardized command submission to the processor (GPU, DSP, FPGA, …)</td>
</tr>
<tr>
<td>6</td>
<td>Eliminate software layers between application and hardware</td>
</tr>
<tr>
<td>7</td>
<td>ISA agnostic for CPU, GPU and other accelerators</td>
</tr>
<tr>
<td>8</td>
<td>Open source software stack</td>
</tr>
</tbody>
</table>

High performance

Low power
HSA – An open Platform

Open Architecture, membership open to all

Delivered via royalty free standards
- Royalty Free IP, Specifications and APIs

ISA agnostic for both CPU and GPU
- Vendors for x86, MIPS, ARM and many GPU architectures

Membership from all areas of computing
- Hardware companies
- Operating Systems
- Tools and Middleware
- Applications
- Universities
The HSA Foundation Membership

<table>
<thead>
<tr>
<th>Founders</th>
<th>AMD</th>
<th>ARM</th>
<th>Imagination</th>
<th>MediaTek</th>
<th>Qualcomm</th>
<th>Samsung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promoters</td>
<td>LG Electronics</td>
<td>GPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supporters</td>
<td>allinea</td>
<td>Argonne</td>
<td>UChicago</td>
<td>Argonne</td>
<td>Arteris</td>
<td>codeplay</td>
</tr>
<tr>
<td></td>
<td>Lawrence Livermore National Laboratory</td>
<td>Linaro</td>
<td>Multicore</td>
<td>AVWare</td>
<td>Oak Ridge</td>
<td>National Laboratories</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sandia National Laboratories</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contributors</td>
<td>ANALOG DEVICES</td>
<td>CANONICAL</td>
<td>CEVA</td>
<td>OMP</td>
<td>ETRI</td>
<td>ITRI</td>
</tr>
<tr>
<td></td>
<td>LUXOFT</td>
<td>MARVEL</td>
<td></td>
<td></td>
<td></td>
<td>Industrial Technology Research Institute</td>
</tr>
<tr>
<td></td>
<td>tensilica</td>
<td>Texas Instruments</td>
<td>TOSHIBA</td>
<td></td>
<td>Synopsys</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Academic</td>
<td>icsa</td>
<td>NTHU Programming Language Lab</td>
<td>NTHU System Software Lab</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tampere University of Technology</td>
<td>University of Mississippi</td>
<td>UNIVERSITY OF NORTH TEXAS</td>
<td>University of Bristol</td>
<td>Rice University</td>
<td>Northeastern</td>
</tr>
</tbody>
</table>
OpenCL™ with HSA, not OpenCL™ vs HSA!

- An optimized platform architecture, which runs OpenCL™ very well
  - Complementary standard, not a competitor to OpenCL™
  - Focused on the hardware and system platform runtime definition, not a driver API
  - Supports many more languages than C/C++, including managed code languages, along with OpenCL™ or other runtimes

- OpenCL™ benefits from a rich and consistent platform infrastructure
Some non-HSA platforms support a few of these platform features. In combination they form a well-rounded base for application programmability.
What defines HSA Platforms and HSA agents?

There are two different machine models ("small" and "large")
- target different functionality levels
- It takes into account different feature requirements for different platform environments
- In all cases, the same HSA application programming model is used to target HSA agents and provides the same power–efficient and low-latency dispatch mechanisms, synchronization primitives and SW programming model

Applications written to target HSA small model machines will generally work on large model machines
- If the large model platform and host Operating System provides a 32bit process environment

<table>
<thead>
<tr>
<th>Properties</th>
<th>Small Machine Model</th>
<th>Large Machine Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform targets</td>
<td>embedded or personal device space (controllers, smartphones, etc.)</td>
<td>PC, workstation, cloud server, etc running more demanding workloads</td>
</tr>
<tr>
<td>Native pointer size</td>
<td>32bit</td>
<td>64bit (+ 32bit ptr if 32bit processes are supported)</td>
</tr>
<tr>
<td>Floating point size</td>
<td>Half (FP16*), Single (FP32) precision</td>
<td>Half (FP16*), Single (FP32), Double (FP64) precision</td>
</tr>
<tr>
<td>Atomic ops size</td>
<td>32bit</td>
<td>32bit, 64bit</td>
</tr>
</tbody>
</table>
Why Shared Virtual Memory (SVM)?

OpenCL 1.x & others: copy data to dedicated memory

- Host CPU must walk all data structures
  - Data is referenced by “handle+offset” only for accelerator code
- Host CPU must copy data to buffer
- Needs to copy more data than needed if accelerator processing selects next step
- Only when data buffer is ready accelerator can process it

High software overhead invalidates benefits for a lot of workloads
Why Shared Virtual Memory (SVM)?

Buffered SVM (OpenCL 2.x): Applications must lock data before accelerator use

- Host CPU must walk all data structures, process some of the data
- Host CPU locks selected data to buffer, accelerator can process it after
- If host CPU finds ptr outside of range, needs to switch locked ranges
- data is referenced by ptr to locked memory only, no handles needed

Better, still a lot of host CPU overhead
Why Shared Virtual Memory (SVM)?

Full SVM (OpenCL 2.x, HSA): Accelerator references & processes data directly
- Accelerator code can parse application data structures in memory, including ptrs
- Accelerator touches only the data needed, greatly reducing bus traffic
- Read, write and execute protections enforced by (IO-)MMU page tables
  - shared system page tables (SVM-IOMMU)
  - or device (shadow) page tables (device SVM)

HSA: Demand paging, SW traps (page/access/privilege violation faults)
Atomic memory updates fundamental to efficient thread synchronization

- Atomic RMW operations needed to implement primitives like mutexes, Semaphores, histograms, ..., previously only implemented on CPU
- HSA agents support platform atomics semantics as defined in memory model

HSA agent supports 32bit or 64bit* values for atomic ops

- CAS, SWAP, add, increment, sub, decrement, ...
- and other common arithmetic and logic atomic ops

On PCI-Express system platform atomics map to PCI-E atomics

*machine model dependent
The HSA Signals Infrastructure

Hardware-assisted signaling and synchronization primitives

- Memory semantics, equivalent to platform atomics
  - e.g. 32bit or 64bit value, content updated atomically
  - wait on value by HSA agents and AQL packets
- Power-efficient & low-latency
- synchronization between threads on HSA agents and host CPU

Allows one-to-one, many to one and one-to-many signaling

- System Software, runtime & application SW use infrastructure to build higher-level synchronization primitives like mutexes, semaphores, ...
HSA Signals (1)

HSA agents communicate with each other by using cache coherent global memory, or by using signals.

A signal is represented by an opaque signal handle

```c
typedef uint64_t hsa_signal_handle_t
```

A signal carries a value, which can be updated or conditionally waited upon via an API call or HSAIL instruction.

- The value occupies four or eight bytes depending on the machine model in use.

```c
typedef uintptr_t hsa_signal_value_t
```
HSA Signals (2)

Updating the value of a signal is equivalent to sending the signal.

In addition to the update (store) of signals, the API for sending signal must support other atomic operations with specific memory order semantics:

- Atomic operations: AND, OR, XOR, Add, Subtract, Exchange, and CAS
- Memory order semantics: Release and Relaxed
The application may wait on a signal, with a condition specifying the terms of wait.

```cpp
enum hsa_signal_condition_t {
    HSA_EQ: The two operands are equal.
    HSA_NE: The two operands are not equal.
    HSA_LT: The first operand is less than the second operand.
    HSA_GTE: The first operand is greater than or equal to the second operand.
};
```
The wait can be done either in the HSA component via an HSAIL wait instruction or via a runtime API defined here.

- Waiting on a signal returns the current value at the opaque signal object;
- The wait has a runtime defined timeout which indicates the maximum amount of time that an implementation can spend waiting;
- The signal infrastructure allows for multiple senders/waiters on a single signal.

The wait may return before

- The condition is satisfied
- The specified timeout has elapsed
- It is the users burden to check the return status of the wait API before consuming the returned value.

Wait reads the value, hence acquire synchronizations may be applied.
Example – HSA Signal wait (1)

```c
hsa_signal_handle_t wait_handle;
hsa_signal_value_t wait_value = 2;

thread_2()
{
    hsa_signal_create(0, &wait_handle);
    hsa_signal_add_relaxed(wait_handle, 3);
    init_data(ptr);
    hsa_signal_sub_relaxed(wait_handle, 1);
}

thread_1()
{
    hsa_signal_value_t ret;
    hsa_signal_wait_timeout_acquire(wait_handle, WAIT_TIME, timeout_flag, HSA_EQ, wait_value, &ret);
    // assert(ret == wait_value);
    update_data(ptr);
}
```

Timeline

**thread_1**

- hsa_signal_wait_timeout_acquire (value == 2)
- thread_1 is blocked
- value = 2
- Return signal value
- Condition satisfied, the execution of thread_1 continues

Timeline

**thread_2**

- value = 0
- hsa_signal_add_relaxed (value = value + 3)
- value = 3
- hsa_signal_substract_relaxed (value = value - 1)
Example – HSA Signal wait (2)

```c
hsa_status_t
hsa_signal_wait_timeout_acquire(hsa_signal_handle_t signal_handle,
                                uint64_t timeout,
                                bool long_wait,
                                hsa_signal_condition_t cond,
                                hsa_signal_value_t compare_value,
                                hsa_signal_value_t * return_value)
{
    /* Signal wait pseudo code */
    hsa_status_t status = HSA_SUCCESS;
    /* Initialize and do some error checking */

    struct _signal_handle* tmp = signal_map(signal_handle);
    if(tmp == NULL)
        return HSA_STATUS_ERROR_INVALID_SIGNAL;

    bool (*signal_compare)(hsa_signal_value_t,
                           hsa_signal_value_t, hsa_signal_condition_t);

    while(!signal_compare(tmp->value, compare_value, cond)){
        timeout--;
        if(timeout = 0){
            status = HSA_STATUS_INFO_SIGNAL_TIMEOUT;
            break;
        }
    }

    if(return_value){
        (*return_value) = tmp->value;
    }
    return status;
}
```

If `signal_handle` is invalid, then return signal invalid status

Signal wait condition function

Compare `tmp->value` with `compare_value` to see if the condition is satisfied?
If `timeout = 0` then return signal time out status

If the condition is satisfied, then return signal and status
User mode queuing
- Low latency dispatch
- Application dispatches directly to queue
- No OS or driver call required in performance path
- “Unlimited” # of queues per process

Architected Queuing Layer (AQL)
- Single compute dispatch path for all hardware
- No driver translation, direct hardware processing
- Standard across vendors!
- Guaranteed backward compatibility

Allows for dispatch to queue from any agent
- CPU or GPU or DSP or FPGA, etc.

Agent self enqueue enables
- Recursion, Tree traversal, Wavefront reforming

Benefits from shared virtual memory, platform atomics and coherency
HSA Command and Dispatch Flow

SW view:
- User-mode dispatches to HW
- No Kernel Driver overhead
- Low dispatch times
- CPU & GPU dispatch APIs

HW view:
- HW / microcode controlled
- HW scheduling
- Architected Queuing Language (AQL)
- HW-managed protection

**HW view:**
- HW view:
  - HW / microcode controlled
  - HW scheduling
  - Architected Queuing Language (AQL)
  - HW-managed protection
The AQL Queue Definition

AQL queue structure

typedef struct hsa_queue_s {
    hsa_queue_type_t type;
    uint32_t features;
}

#define HSA_LARGE_MODEL
void * base_address;
#endif defined HSA_LITTLE_ENDIAN
void * base_address;
#else
    uint32_t reserved0;
#endif

hsa_signal_t doorbell_signal;
uint32_t size;
uint32_t reserved1;
uint64_t id ;
} hsa_queue_t

format=packet format
B=Barrier (1 bit)
a=acquire fence scope (2 bit)
r=release fence scope (2 bit)
Rsv=reserved

single-/multi-producer
AGENT_DISPATCH | KERNEL_DISPATCH

Kernel Dispatch
Agent Dispatch
Barrier AND
Barrier OR

Header

AQL packet payload
(AQL packet size 64 Byte)

AQL packets
The AQL Queue Definition

In addition to the data held in the queue structure, the queue also defines two properties (readIndex and writeIndex) that define the location of “head” and “tail” of the queue.

- **readIndex**: The read index is a 64-bit unsigned integer that specifies the packetID of the next AQL packet to be consumed by the packet.

- **writeIndex**: The write index is a 64-bit unsigned integer that specifies the packetID of the next AQL packet slot to be allocated.

Both indices are not directly exposed to the user, who can only access them atomically by using dedicated HSA core runtime APIs.

- The available index functions differ on the index of interest (read or write).
- action to be performed (addition, compare and swap, etc.)
- and memory consistency model (relaxed, release, etc.)
The AQL Queue Definition

The read index is automatically advanced when a packet is read by the packet processor

When the packet processor observes that

- The read index matches the write index, the queue can be considered empty;
- The write index is greater or equal than the sum of the read index and the size of the queue, then the queue is full.

The *doorbell_signal* field of a queue contains a signal that is used by the agent to inform the packet processor to process the packets it writes.

- The value that the doorbell signaled is equal to the ID of the packet that is ready to be launched.
The AQL Queue Definition

An HSA agent submits a task to a queue by performing the following steps:

- 1.) Allocate a packet slot (by atomically incrementing the writeIndex)
- 2.) Initialize and copy packet data into AQL queue packet slot
- 3.) As last step, mark packet as valid (by setting format entry of packet header)
- 4.) Notify the Packet Processor of the packet (With doorbell signal)
The AQL Queue Definition

The new task might be consumed by the packet processor even before the doorbell signal has been signaled by the agent.

- Because the packet processor might be already processing some other packets
- and observes new valid work packets available when prefetching to writeIndex
- So it may process the new valid packets right away

In any case, the agent must ring the doorbell for every batch of packets it writes.
Example - Enqueue AQL packet (2)

Allocate an AQL packet slot

```c
hsa_aql_dispatch_packet_t dispatch_packet;
uint64_t write_index = 0;
hsa_queue_create(component, 1024, 0, NULL, NULL, &queue);
write_index = hsa_queue_cas_write_index_release(queue, write_index, write_index+1);
```
Example - Packet processor

```c
/* The following pseudo code is simulates
 * the behavior of packet processor
 */

uint64_t read_index = 0;
uint64_t current_write_index = hsa_queue_load_write_index_relaxed(queue);
while(1)
{
    /* wait doorbell here */
    hsa_signal_wait_acquire(queue->doorbell, HSA_NE,
    current_write_index, &current_write_index);

    // Receive doorbell
    PACKET_PROCESS:
    hsa_aqi_dispatch_packet_t *AQL_packet = queue->base_address + read_index*AQL_SIZE;
    // Check if barrier packet
    if(AQL_packet->header.format == HSA_AQL_PACKET_FORMAT_BARRIER)  
        hsa_signal_wait_acquire(barrier, HSA_EQ, signal_barrier, &signal_barrier);

    /* do the jobs contain in packet */
    hsa_queue_store_read_index_relaxed(queue, (read_index + 1)%queue->size);
    read_index = hsa_queue_load_read_index_acquire(queue);
    AQL_packet.header.format = HSA_AQL_PACKET_FORMAT_INVALID;
    hsa_signal_store_release(AQL_packet->completion_signal, read_index-1);
    current_write_index = hsa_queue_load_write_index_relaxed(queue);
    if(read_index != current_write_index)
        goto PACKET_PROCESS;

    // Check if barrier packet
    if(AQL_packet->header.format == HSA_AQL_PACKET_FORMAT_BARRIER)  
        hsa_signal_wait_acquire(barrier, HSA_EQ, signal_barrier, &signal_barrier);

    // Update readIndex, change packet state to invalid, and send completion signal.
    if there is any packet in queue, process the packet.
```
HSA Memory Model

Defines data visibility, ordering between all threads in the HSA System

Designed to be compatible with C++11, Java, OpenCL and .NET Memory Models

Relaxed consistency memory model for parallel compute performance

- HRF based definition, scopes, relaxed atomics
- Plan: formal definition, automated verification

Visibility controlled by:

- Load.Acquire
- Store.Release
- Fences
HSA Security & QoS Model

User- and privileged memory

Hardware enforced process isolation

Security

System page table attributes for accelerator

Limited execution privilege for AQL

Context Switch

Quality of Service

Hardware assisted scheduling

Terminate and Reset

Preemption*

Hardware assisted scheduling

Terminate and Reset
Securing HSA agent resources in the system

An HSA agent supports full MMU

- Allows for user protected execution, protected memory and memory curtaining
- Read, write and execute protections by page table entry
- Brings Inter-process protection
- Improved Fault handling (page fault, access fault, privilege violation fault)
- User mode and supervisor mode execution modes

Pre-emption with the ability to kill errant tasks on the GPU

- Preemption and context switch capabilities
- Per-process or per-work item kill available to the OS
To lower the software overhead, the system software infrastructure does look a little different…
HSA Intermediate Layer – HSAIL

- HSAIL is a virtual ISA for parallel programs
  - Finalized to ISA by a JIT compiler or “Finalizer”
  - ISA independent by design for CPU & GPU
- Explicitly parallel
  - Designed for data parallel programming
- Support for exceptions, C++ virtual functions, and other high level language features
- Agent Dispatch to call OS and system runtime
  - GPU/accelerator code can call directly to OS and other system runtime services, I/O, printf, etc.
- Debugging, Profiling support requirements
How is HSAIL and BRIG used?

Intermediate language for parallel compute in HSA

- Generated by a “High Level Compiler” (GCC, LLVM, Java VM, etc)
- Expresses parallel regions of code
- Binary format of HSAIL is called “BRIG”
- Brings parallel acceleration to mainstream programming languages

```c
main() {
    ...
    #pragma omp parallel for
    for (int i=0; i<N; i++) {
    }
    ...
}
```

![Diagram](image_url)
The Multivendor Specification requirements

Products today are Systems on a Chip (SoC) using different IP blocks

- Often from different vendors, synthesized and integrated onto a die
- Data fabric is highly architecture dependent (AXI, PCI-Express, HT, ...)
  - Each IP core and software needs to be adapted to support fabric protocols
- HSA Multivendor specification requirements
  - Goal: simpler integration into the common platform/software model
  - Define protocols and interfaces that each HSA agent supports and converts to
  - Define HSA runtime and driver sections into vendor specific and vendor neutral blocks
- Currently focuses on software defined interfaces
- In the future, common external HSA compliant hardware protocols may be referenced
Being there, Virtualizing the HSA system

HSA/ROCm Docker™ Application containers

- Utilizing SVM, processes & user queues of HSA to isolate container execution
- In addition to general virtualization features of the hardware

KVM, Xen™

- Several projects & implementations, used both in academia and industry
- Building a KVM-based Hypervisor for a Heterogeneous System Architecture Compliant System (ACM)
The Architecture Benefits in Benchmarks

Traditional Patterns Result

Benchmark: NUCAR HeteroMark
The Benefits of Collaborative Execution

Taking advantage of fine-grain data access & execution in device dispatch in HSA™ vs OpenCL™

CPU Producer GPU Consumer (Background Extraction)

GPU Producer CPU Consumer (Gene Alignment)

Benchmark: NUCAR HeteroMark
A comparison of platforms for parallel accelerator compute and their software design
A current platform for GPU Compute

The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces
- High overhead dispatch, data copies across PCIe
- New languages and APIs for programming necessary (OpenCL, CUDA, etc.)
  - Memory is accessed via handles (OpenCL1.x)
  - or “pseudo-SVM” (“transparent” copy with pointer aliasing)
- Some but not all HSA platform features may be still available

➡️ Dual source development

- Expert programmers only
- Strict limits to High-Level Language use
  - C++AMP 1.0: special directives with high overhead
  - Because it locks down and copies data in the background
The First Gen. of APUs and SOCs: Physical Integration

Benefits

- Some memory copies are gone, because the same memory is accessed. Two memory pools remain (cache coherent + non-coherent memory regions)
- Jobs are still queued through the OS driver chain and suffer from overhead
  - But the memory is not accessible concurrently, because of cache policies
- Still requires experts to get good performance
- Supports (coarse-grain) "buffer SVM"
- This is only an intermediate step in the journey
AN HSA Enabled SOC

Benefits

◆ **Unified Coherent Memory, collaborative data sharing across all processors**

◆ Processors architected to operate cooperatively
  - Can exchange data “on the fly” with atomics & cache coherency, like CPU threads do
  - The lower job dispatch overhead allows tasks to be handled by the GPU that previously were “too costly” to transfer over and remained on the less optimal CPU implementation

◆ Well suited to High Level Programming Languages

◆ **HLL SW logic can remain largely unchanged**
  - Data can be passed by pointer reference
  - Instead of copying it into different regions
  - Compiler/Runtime target different processors same way
  - “Transparency” if necessary
HSA architected Platform Profiling and Debugging: because nothing is perfect from the start

Profiling

• Common timeline across HSA accelerators & system
• Common HSA hardware events (+ HW specific)
• Common HSA profiling counter definitions (+ HW specific counters)
• Consistent profiling methodology for all HSA accelerators

Debugging

• Breakpoints
• Exception handling
• Single-step
• Tracing
• HSAIL Disassembly
• Emulation support
• Libraries
• Plugins
The things to come...

- Improved Data Interop
  - Fixed function accelerators (e.g. FPGA)
  - Local device memory

- Architecture
  - Improved Architected Profiling and Debug
  - BRIG, new linking formats

- Programming Models
  - Fully formalized memory model
  - HSAIL Parallel loops
  - Flexible API and access semantics
The major goals of the HSA vision have been reached

- The first HSA conformant platform products and SOC IP are now available
  - E.g. AMD Carrizo (AMD A10-8xxx) and Bristol Ridge APUs, and HSA IP cores from other vendors
  - AMD’s ROCm drivers and runtimes extend the HSA model even to discrete AMD GPUs!
- All common OS have now native support for Shared Virtual Memory (SVM)
- Optimized drivers and runtimes for AMD products and other vendors are available
- Compilers, libraries, frameworks have been adapted and optimized
- HSA 1.0 targets GPU compute, HSA 1.1 adds DSP, multi-vendor integration and more efficient non-HSA device integration

But we’re still at the beginning fully utilizing the benefits...

- Optimizing data fabrics, IP block microarchitecture to support HSA macroarchitecture
- Using HSA accelerators in more workloads efficiently
- Integrating more domain specific processors into the HSA environment efficiently
Software & Toolchains
HSA – An open platform

Open Architecture, membership open to all

Delivered via royalty free standards
- Royalty Free IP, Specifications and APIs

ISA agnostic for both CPU and GPU
- Vendors for x86, MIPS, ARM and many GPU architectures

Membership from all areas of computing
- Hardware companies
- Operating Systems
- Tools and Middleware
- Applications
- Universities
Some non-HSA platforms support a few of these platform features. In combination they form a well-rounded base for application programmability.
What defines HSA Platforms and HSA agents?

<table>
<thead>
<tr>
<th>Properties</th>
<th>Small Machine Model</th>
<th>Large Machine Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform targets</td>
<td>embedded or personal device space (controllers, smartphones, etc.)</td>
<td>PC, workstation, cloud server, etc running more demanding workloads</td>
</tr>
<tr>
<td>Native pointer size</td>
<td>32bit</td>
<td>64bit (+ 32bit ptr if 32bit processes are supported)</td>
</tr>
<tr>
<td>Floating point size</td>
<td>Half (FP16*), Single (FP32) precision</td>
<td>Half (FP16*), Single (FP32), Double (FP64) precision</td>
</tr>
<tr>
<td>Atomic ops size</td>
<td>32bit</td>
<td>32bit, 64bit</td>
</tr>
</tbody>
</table>

- There are two different machine models ("small" and "large")
  - target different functionality levels
  - It takes into account different feature requirements for different platform environments
  - In all cases, the same HSA application programming model is used to target HSA agents and provides the same power–efficient and low-latency dispatch mechanisms, synchronization primitives and SW programming model

- Applications written to target HSA small model machines will generally work on large model machines
  - If the large model platform and host Operating System provides a 32bit process environment

*min. Load and store on memory
OpenCL™ with HSA, not OpenCL™ vs HSA!

An optimized platform architecture, which runs OpenCL™ very well

- Complementary standard, not a competitor to OpenCL™
- Focused on the hardware and system platform runtime definition, not a driver API
- Supports many more languages than C/C++, including managed code languages, along with OpenCL™ or other runtimes

OpenCL™ benefits from a rich and consistent platform infrastructure
HSA: An open software platform

- HSA specifications are publicly available (at http://www.hsafoundation.com)
  - HSA Conformance suite ensures platform product is fully compatible with software

- HSA platforms feature a full open source Linux stack: OS drivers, compilers, tools, libraries, frameworks and applications
  - AMD: Includes Linux kernel drivers user space drivers and HSA runtime
  - HSA Foundation reference implementation for kernel and user drivers by AMD
  - HSA Foundation sources and documentations at: https://github.com/HSAFoundation
  - Optimized AMD system software is available as Radeon Open Compute (ROCm)

- Allows a single shared implementation for many components
- Enables university research and industry collaboration in all areas
  - Released AMD sources at: https://github.com/RadeonOpenCompute
Radeon Open Compute (ROCM)

Modern Heterogeneous HPC and Ultra-scale Platform for Large Scale Systems

Performance
Rich Foundation built for Latency Reduction and Throughput Optimization

Open
First Fully Open Source Professional GPU Computing Solution

Freedom
Foundation to Explore the Boundaries of GPU Computing
AMD ROCm 1.2: Linux open source drivers

AMD open source HSA compatible drivers for AMD APUs discrete AMD GPUs with HSA device features (Fiji = Radeon Fury/X/Nano)

- ROCm allows use of HSA software features on AMD discrete GPU
  - Also on Intel CPU platforms, requires PCI-E atomics (Intel: Haswell, Skylake, …)!
  - AMD: Carrizo, Bristol Ridge A10-8xxx, A12-8xxx)

- ROCK: HSA kernel mode driver (amdkfd), graphics kernel driver (amdgpu)
  - in upstream to main Linux kernel source, github currently based on 4.1 onwards

- ROCR, ROCT: HSA user runtime, Kernel driver “Thunk Layer” Interface for Runtime
  - Provides the HSA interfaces for compilers and other application runtimes

Install HOWTO: http://gpuopen.com/compute-product/rocm/
Open Source Compilers

CLANG/LLVM compiler toolchain targeting HSAIL/BRIG

- Base for Heterogeneous Compute Compiler (HCC), a fully optimizing HSA C++ 11/14 single source offline compiler, with standard LibC++, supports several different accelerator modes:
  - HC++ API, inspired by C++14 and C++AMP APIs and directives as superset
  - Heterogeneous Interface for Portability (HIP), allows CUDA application compilation via common HIP infrastructure API (more details later)
  - C++AMP™ 1.2 standard compatible
  - C++ 17 Parallel STL
  - OpenMP™ 3.1 C & C++ Support Today for CPU

GNU compiler, optimized HSAIL/BRIG, patch set in up-stream for GCC7
"Vector ADD", hcc C++ vs OpenCL

int column = 128;
int row = 256;
// define the compute grid bounds<2> grid { column, row };

float* a = new float[grid.size()];
float* b = new float[grid.size()];
float* c = new float[grid.size()];

// Using standard C++17 launch syntax
parallel::for_each(par, begin(grid), end(grid), [&](index<2> idx) {
    int i = idx[1] * column + idx[0];
    c[i] = a[i] + b[i];
});
int column = 128;
int row = 256;
// define the compute grid
bounds<2> grid { column, row };

float* a = new float[grid.size()];
float* b = new float[grid.size()];
float* c = new float[grid.size()];

// Using standard C++17 launch syntax
parallel_for_each(par, begin(grid), end(grid),
    [&](index<2> idx) {
        int i = idx[1] * column + idx[0];
        c[i] = a[i] + b[i];
    })
Bring Python via Anaconda to ROCm

Embracing Python Developer Community with Heterogeneous Acceleration

Numba: speed up your applications with high performance functions written directly in Python

- Rich set of options to optimize for APU or discrete GPU
  - Async execution, specify group size, use shared memory
  - Data transfer can be performed implicitly based on kernel arguments

ANACONDA on ROCm a New Class of Performance for Python
Portable OpenCL™ (pocl)

“Portable OpenCL” API runtime is an OpenCL™ 1.2 and 2.0 API implementation

- Among other targets, HSA runtime architecture supported, creates HSAIL code from OpenCL™ kernels to finalize through HSA Runtime
- Global/local/private memory
- Barriers
- Most of the OpenCL 1.2 kernel builtins
- OpenCL™ 2.0 shared virtual memory (SVM)
- OpenCL™ 2.0 atomics
- Performance similar to vendor specific implementations

Portable HSA (phsa) is a portable HSA implementation for CPU & DSP
HIP – Heterogeneous Compute Interface for Portability

Path for CUDA Applications to AMD enable GPU’s

Port from CUDA to a common C++ programming model
  HIP code runs through either CUDA NVCC or HCC
  HiPify tools simplify porting from CUDA to HIP

Builds on HCC Compiler
  *Host and device code can use templates, lambdas, advanced C++ features*
  *C-based runtime APIs (hipMalloc, hipMemcpy, hipKernelLaunch and more)*
Growing the heterogeneous ecosystem

Vendor OpenCL Libraries

- OpenCL runtime using HSA
- Lower dispatch latency
- Common HSA backend
- HSAIL offline optimization
- Efficient synchronization

HSA Runtime Libraries in HSAIL

- Previous benefits plus:
  - HSAIL offline optimization (manual or automated)
  - Call by reference (SVM)
  - Using Multivendor accelerator IP with common SW interface
  - Readily usable across HSA platforms

High-Level Language to HSAIL

- C++ 11/14 (HCC) + PSTL, Python and other languages
- Target accelerator via common Language Runtime library on top of HSA Runtime & HSAIL
- Simplified programming without efficiency loss
## Comparison of ROCm Compute Languages

<table>
<thead>
<tr>
<th>Term</th>
<th>HIP</th>
<th>HCC</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Device</strong></td>
<td>int deviceld (0..n-1)</td>
<td>hc::accelerator</td>
<td>cl_device</td>
</tr>
<tr>
<td><strong>Queue</strong></td>
<td>hipStream_t</td>
<td>hc::accelerator_view</td>
<td>cl_command_queue</td>
</tr>
<tr>
<td><strong>Event</strong></td>
<td>hipEvent_t</td>
<td>hc::completion_future</td>
<td>cl_event</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>void *</td>
<td>void *; hc::array;</td>
<td>cl_mem</td>
</tr>
<tr>
<td></td>
<td>grid</td>
<td>extent</td>
<td>NDRange</td>
</tr>
<tr>
<td></td>
<td>block</td>
<td>tile</td>
<td>work-group</td>
</tr>
<tr>
<td></td>
<td>thread</td>
<td>thread</td>
<td>work-item</td>
</tr>
<tr>
<td></td>
<td>warp</td>
<td>wavefront</td>
<td>sub-group</td>
</tr>
<tr>
<td><strong>Device Kernel</strong></td>
<td><strong>global</strong></td>
<td>lambda inside</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hc::parallel_for_each or [hc]__kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel Launch</strong></td>
<td>hipLaunchKernel</td>
<td>hc::parallel_for_each</td>
<td>clEnqueueNDRangeKernel</td>
</tr>
<tr>
<td><strong>Atomic Builtins</strong></td>
<td>atomicAdd</td>
<td>hc::atomic_fetch_add</td>
<td>atomic_add</td>
</tr>
<tr>
<td><strong>Precise Math</strong></td>
<td>cos(f)</td>
<td>hc::precise_math::cos(f)</td>
<td>cos(f)</td>
</tr>
<tr>
<td><strong>Single-Source</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>C++ Kernel Language</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>C++ Runtime</strong></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>When to use it:</strong></td>
<td>Porting existing CUDA code.</td>
<td>Path to ISO C++ compatibility</td>
<td>Porting existing OpenCL code.</td>
</tr>
<tr>
<td></td>
<td>New project where portability to NVIDIA and AMD GPUs is required</td>
<td>New projects where C++ preferred</td>
<td>New project where Portability to GPU, FPGA, CPU required</td>
</tr>
</tbody>
</table>

**HSA Foundation AMD**
What defines HSA Platforms and HSA agents?

An HSA compatible platform consists of one or more “HSA agents”
- Each follows Architected Methods to enqueue or dispatch & execute work items
- Each follows the HSA memory model for HSA related operations
- Some HSA agents can submit AQL packets for execution
- Some HSA agents can be a “target” of AQL packets
  - Most HSA agents can operate in both roles or even operate “outside of HSA”
- Architected infrastructure allows exchanging data with non-HSA compliant components in a platform
- Fundamental data types are naturally aligned

Definitions can be found in the “System Architecture” specification
HSAIL – Solving the Portability Challenge

CPU ISAs
- ISA innovations added incrementally (ie NEON, AVX, etc)
  - ISA retains backwards-compatibility with previous generation
- Two dominant instruction-set architectures: ARM and x86

Device Parallel Process ISAs (GPUs)
- Massive diversity of architectures in the market
  - Each vendor has own ISA - and often several in market at same time
- No commitment (or attempt!) to provide any backwards compatibility
  - Traditionally graphics APIs (OpenGL, DirectX) provide necessary abstraction
HSA Parallel Execution Model

**Basic Idea:**

Programmer supplies an HSAIL “kernel” that is run on each work-item. Kernel is written as a single thread of execution.

Programmer specifies grid dimensions (scope of problem) when launching the kernel.

Each work-item has a unique coordinate in the grid.

Programmer optionally specifies work-group dimensions (for optimized communication).
What is HSAIL?

HSAIL is the intermediate language for parallel compute in HSA

- HSAIL is a low level instruction set designed for parallel compute in a shared virtual memory environment.
- Generated by a high level compiler (LLVM, gcc, Java VM, etc)
- Compiled down to Parallel Processor ISA by an IHV Finalizer
- Finalizer may execute at run time, install time or build time, depending on platform type

HSAIL is SIMT in form and does not dictate hardware microarchitecture

- HSAIL is designed for fast compile time, moving most optimizations to HL compiler
- Limited register set avoids full register allocation in Finalizer
- HSAIL is at the same level as PTX: an intermediate assembly or Virtual Machine Target
- Represented as bit-code in in a Brig file format with support late binding of libraries.
How is HSAIL and BRIG used?

Intermediate language for parallel compute in HSA

- Generated by a “High Level Compiler” (GCC, LLVM, Java VM, etc)
- Expresses parallel regions of code
- Binary format of HSAIL is called “BRIG”
- Brings parallel acceleration to mainstream programming languages
HSAIL INSTRUCTION SET - OVERVIEW

Similar to assembly language for a RISC CPU

- Load-store architecture
- Destination register first, then source registers

```
ld_global_u64  $d0, [d6 + 120] ; $d0= load(d6+120)
add_u64        $d1, $d0, 24  ; $d1= $d2+24
```

140 opcodes (Java™ bytecode has 200)

- Floating point (single, double, half (f16))
- Integer (32-bit, 64-bit)
- Some packed operations
- Branches
- Function calls
- Platform Atomic Operations: and, or, xor, exch, add, sub, inc, dec, max, min, cas
  - Synchronize host CPU and HSA Component!

Text and Binary formats (“BRIG”)
Segments and Memory (1/2)

7 segments of memory
- global, readonly, group, spill, private, arg, kernarg
- Memory instructions can (optionally) specify a segment
- Control data sharing properties and communicate intent

Global Segment
- Visible to all HSA agents (including host CPU)

Group Segment
- Provides high-performance memory shared in the work-group.
- Group memory can be read and written by any work-item in the work-group
- HSAIL provides sync operations to control visibility of group memory

```
ld_global_u64   $d0,[$d6]
ld_group_u64    $d0,[$d6+24]
st_spill_f32    $s1,[$d6+4]
```
Segments and Memory (2/2)

Spill, Private, Arg Segments
- Represent different regions of a per-work-item stack
- Typically generated by compiler, not specified by programmer
- Compiler can use these to convey intent – i.e. spills

Kernarg Segment
- Programmer writes kernarg segment to pass arguments to a kernel

Read-Only Segment
- Remains constant during execution of kernel
FLAT ADDRESSING

Each segment mapped into virtual address space
- Flat addresses can map to segments based on virtual address

Instructions with no explicit segment use flat addressing

Very useful for high-level language support (i.e., classes, libraries)

Aligns well with OpenCL 2.0 “generic” addressing feature

```
ld_global_u64 $d6, [%_arg0] ; global
ld_u64 $d0,[$d6+24] ; flat
```
Registers

Four classes of registers:
- **S**: 32-bit, Single-precision FP or Int
- **D**: 64-bit, Double-precision FP or Long Int
- **Q**: 128-bit, Packed data.
- **C**: 1-bit, Control Registers (Compares)

Fixed number of registers
- S, D, Q share a single pool of resources
- S + 2*D + 4*Q <= 128
- Up to 128 S or 64 D or 32 Q (or a blend)

Register allocation done in high-level compiler
- Finalizer doesn’t perform expensive register allocation
SIMT Execution Model

HSAIL Presents a “SIMT” execution model to the programmer

- “Single Instruction, Multiple Thread”
- Programmer writes program for a single thread of execution
- Each work-item appears to have its own program counter
- Branch instructions look natural

Hardware Implementation

- Most hardware uses SIMD (Single-Instruction Multiple Data) vectors for efficiency
- Actually one program counter for the entire SIMD instruction
- Branches implemented with predication

SIMT Advantages

- Easier to program (branch code in particular)
- Natural path for mainstream programming models and existing compilers
- Scales across a wide variety of hardware (programmer doesn’t see vector width)
- Cross-lane operations available for those who want peak performance
Wavefronts

Hardware SIMD vector, composed of 1, 2, 4, 8, 16, 32, 64, 128, or 256 “lanes”

Lanes in wavefront can be “active” or “inactive”

```
if (cond) {
    operationA; // cond=True lanes active here
} else {
    operationB; // cond=False lanes active here
}
```

Inactive lanes consume hardware resources but don’t do useful work

Tradeoffs

- “Wavefront-aware” programming can be useful for peak performance
- But results in less portable code (since wavefront width is encoded in algorithm)
Cross-Lane Operations

Example HSAIL cross-lane operation: “activelaneid”

- Dest set to count of earlier work-items that are active for this instruction
- Useful for compaction algorithms

```
activelaneid_u32 $s0
```

Example HSAIL cross-lane operation: “activelanesshuffle”

- Each workitem reads value from another lane in the wavefront
- Supports selection of “identity” element for inactive lanes
- Useful for wavefront-level reductions

```
activelanesshuffle_b32 $s0, $s1, $s2, 0, 0
// s0 = dest, s1= source, s2=lane select, no identity
```
Working group strived to limit optional modes and features in HSAIL

- Minimize differences between HSA target machines
- Better for compiler vendors and application developers
- Two modes survived

Machine Models, Base + Full Profiles

- Small: 32-bit pointers, 32-bit data
- Large: 64-bit pointers, 32-bit or 64-bit data
- Vendors can support one or both models
Convolution / Sobel Edge Filter

\[ G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \]

\[ G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \]

\[ G = \sqrt{G_x^2 + G_y^2} \]
Convolution / Sobel Edge Filter

\[ G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \]

\[ G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \]

\[ G = \sqrt{G_x^2 + G_y^2} \]
Convolution / Sobel Edge Filter

$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$

$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$

$G = \sqrt{G_x^2 + G_y^2}$
AMD released CodeXL™ as Open Source Software

- Supports Windows™ and Linux™
- Supports combined host and accelerator (GPU) profiling and debugging
- Supports OpenCL™, HSA Runtime/ROCm, Vulkan™
- A basis for common, system-wide HSA profiling and debugging toolkit for HSA platforms
- Extensible and open
  - https://github.com/GPUOpen-Tools/CodeXL
  - http://gpuopen.com/compute-product/codexl/
Drive Deeper Visibility into your Code

- **ROCM Profiler** The Command line tool for profiling application on headless server

- **CodeXL** for Visualizing and Analyzing your Application Performance

- **ROCM Debugger**
  - Debugging solution for inspect and run-control
  - GDB based Solution
Increased Investment in Key Solutions for ROCm

Build a Rich Foundation of Libraries and Applications

**Core math libraries**
- BLAS (Skinny, Non-Square, Square) SGEMM & DGEMM
- FFT
- SPARSE
- RAND
- OpenVX

**Third-party Frameworks**
- Kokkos C++ Framework
- CHARM++ Frameworks
- HPX Framework*
- QUDA A library for QCD on GPU*
- HIP CUB*
- HIP Modern GPU Library*
- NCCL*

**Benchmarks**
- SHOC
- Parboil
- Lullesh
- MiniFE
- Cloverleaf
- HPL
- CoMD
- MiniAMR
- HPGMG
- XSBench

**Applications**
- Chroma Dynamics
  - MILC*
  - Chroma*
- Molecular Dynamics
  - AMBER*
  - NAMD*
  - Gromacs
  - LeanMD
  - HoodMD-BLUE*
- Bioinformatics
  - PEANUT
  - BEAGLE
  - GPU-Blast

*In Development
Bringing Over The Key Deep Learning Frameworks

Instinctive Computing foundation for Machine Learning and Neural Networks

Supporting Key Neural Network Frameworks

Torch 7 - HCC
Caffe – OpenCL and HCC
Tensorflow is being ported VIA HIP
Addressing Key issue in Machine Learning

Instinctive Computing foundation for Machine Learning and Neural Networks Core Solvers

- mlOpen
  - Optimized Solver for Convolution based Neural Network
    - Direct Convolution Forward and Backward Propagation
    - Pooling (Max, Random, Ave, Total), Normalization, Dropout, Softmax, Entropy, Log Regression
    - Learning types: Fixed, Linear, Exp_step, Exp_Inv, Total
    - Neuron Types: Passthru, Logistic, Tanh, RELU, BRELU, SOFTRELU, ABS, Square, Square Root, Linear, Power
  - Control over Nodes in Fully Connected Multilayer Nets

- Caffe for CFAIR 10 now 9x faster on forward propagation than current public OpenCL Caffe port
### The things to come...

<table>
<thead>
<tr>
<th>Coarse grain memory v2</th>
<th>Improved Data Interop</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fixed function accelerators (e.g. FPGA)</td>
<td>Architected Debug and profiling</td>
</tr>
<tr>
<td></td>
<td>Local device memory</td>
<td>BRIG, new linking formats</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Programming Models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fully formalized memory model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSAIL Parallel loops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flexible API and access semantics</td>
</tr>
</tbody>
</table>
Research Opportunities
Looking beyond the traditional GPU languages

Dynamic Languages are one of the most interesting areas for exploration of heterogeneous parallel runtimes and have many applications

Dynamic Languages have different compilation foundations targeting HSA / HSAIL

- Java/Scala, JavaScript, Dart, etc

HSA allows efficient support for standards based language environments

- like OpenMP, Java, Fortran, GO, Haskell, Python,
- And domain specific languages like Halide, Julia, and many others

Making the HSA platform performant
We are looking to build out a worldwide stage

How to Join

- Get started today developing with ROCm - Getting Started
  https://radeonopencompute.github.io/install.html

- Access to all source code of ROCm @ GitHub RadeonOpenCompute
  https://github.com/RadeonOpenCompute

- Show case your applications, libraries and tools leveraging ROCm on GPUOpen
  www.gpuopen.com

“The power of one, if fearless and focused, is formidable, but the power of many working together is better.”

– Gloria Macapagal Arroyo
Thank you!

HTTP://WWW.HSAFOUNDATION.COM
HTTP://WWW.GPUOPEN.COM
HTTP://DEVELOPER.AMD.COM

Trademark Attribution
HSA Foundation, the HSA Foundation logo and combinations thereof are trademarks of HSA Foundation, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

With thanks to Dr. John Glossner, CEO GPT & President HSA Foundation, Greg Stoner, Sen. Director AMD & Director HSA Foundation, Dr. David Kaeli, Northeastern U. and his research team And others for some material and feedback