HSA foundation

Politecnico di Milano
Seminar Room A. Alario
23 November, 2015

Antonio R. Miele
Marco D. Santambrogio
Politecnico di Milano
This presentation is based on the material and slides published on the HSA foundation website:

Heterogeneous processors have proliferated – make them better

- Heterogeneous SOCs have arrived and are a tremendous advance over previous platforms
- SOCs combine CPU cores, GPU cores and other accelerators, with high bandwidth access to memory
- How do we make them even better?
  - Easier to program
  - Easier to optimize
  - Easier to load balance
  - Higher performance
  - Lower power
- HSA unites accelerators architecturally
- Early focus on the GPU compute accelerator, but HSA will go well beyond the GPU
HSA foundation

- Founded in June 2012
- Developing a new platform for heterogeneous systems
- [www.hsafoundation.com](http://www.hsafoundation.com)
- Specifications under development in working groups to define the platform
- Membership consists of 43 companies and 16 universities
- Adding 1-2 new members each month
HSA consortium

Founders
AMD
ARM
Imagination
MediaTek
Qualcomm
Samsung
Texas Instruments

Promoters
LG Electronics

Supporters
Arteris
FabricEngine
Kadim
Lawrence Livermore National Laboratory
Allinea
Argonne
StreamComputing

Contributors
Analog Devices
Apical
Broadcom
Canonical
CEVA
ETRI
\text{\textregistered} Synopsys
tensilica
Toshiba
Ubuntu
YiK

Academic
NTHU Programming Language Lab
NTHU System Software Lab
University of Bristol
The University of Edinburgh
University of Illinois
University of Mississippi

HSA goals

- To enable power-efficient performance
- To improve programmability of heterogeneous processors
- To increase the portability of code across processors and platforms
- To increase the pervasiveness of heterogeneous solutions throughout the industry
Paradigm shift

• Inflection in processor design and programming
Key features of HSA

• hUMA – Heterogeneous Unified Memory Architecture
• hQ – Heterogeneous Queuing
• HSAIL – HSA Intermediate Language
Key features of HSA

- **hUMA** – Heterogeneous Unified Memory Architecture
- **hQ** – Heterogeneous Queuing
- **HSAIL** – HSA Intermediate Language
Legacy GPU compute

- Multiple memory pools
- Multiple address spaces
  - No pointer-based data structures
- Explicit data copying across PCIe
  - High latency
  - Low bandwidth
- High overhead dispatch

- Need lots of compute on GPU to amortize copy overhead
- Very limited GPU memory capacity
- Dual source development
- Proprietary environments
- Expert programmers only
Existing APUs and SoCs

- Physical integration of GPUs and CPUs
- Data copies on an internal bus
- Two memory pools remain
- Still queue through the OS
- Still requires expert programmers

APU = Accelerated Processing Unit (i.e. a SoC containing also a GPU)
Existing APUs and SoCs

- CPU and GPU still have separate memories for the programmer (different virtual memory spaces)
  1. CPU explicitly copies data to GPU memory
  2. GPU executes computation
  3. CPU explicitly copies results back to its own memory
An HSA enabled SoC

- Unified Coherent Memory enables data sharing across all processors
  - Enabling the usage of pointers
  - Not explicit data transfer -> values move on demand
  - Pageable virtual addresses for GPUs -> no GPU capacity contraints
- Processors architected to operate cooperatively
- Designed to enable the application to run on different processors at different times
Unified coherent memory

- Coherent Memory: Ensures CPU and GPU caches both see an up-to-date view of data.
- Pageable Memory: The GPU can seamlessly access virtual memory addresses that are not (yet) present in physical memory.
- Entire memory space: Both CPU and GPU can access and allocate any location in the system's virtual memory space.
Unified coherent memory

- CPU and GPU have a unified virtual memory spaces
  1. CPU simply passes a pointer to GPU
  2. GPU executes computation
  3. CPU can read the results directly – no copy need!
Unified coherent memory
Unified coherent memory

DATA POINTERS

SYSTEM MEMORY

Transmission of input data

GPU

GPU MEMORY

FLAT TREE
RESULT BUFFER
Unified coherent memory

DATA POINTERS

SYSTEM MEMORY

TREE

RESULT BUFFER

GPU

KERNEL

GPU MEMORY

FLAT TREE
RESULT BUFFER
Unified coherent memory

DATA POINTERS

Legacy

SYSTEM MEMORY

TREE

RESULT BUFFER

GPU

KERNEL

GPU MEMORY

FLAT TREE

RESULT BUFFER
Unified coherent memory
Unified coherent memory

DATA POINTERS

SYSTEM MEMORY

GPU

GPU MEMORY

FLAT TREE

RESULT BUFFER

KERNEL

RESULT BUFFER
Unified coherent memory
Unified coherent memory

DATA POINTERS
HSA and full OpenCL 2.0

SYSTEM MEMORY

RESULT BUFFER

TREE

GPU
KERNEL
Unified coherent memory
Unified coherent memory

DATA POINTERS

HSA

SYSTEM MEMORY

L

R

L

R

TREE

RESULT BUFFER

GPU

KERNEL
Unified coherent memory

DATA POINTERS

HSA

SYSTEM MEMORY

GPU

KERNEL

TREE

RESULT BUFFER
Unified coherent memory

DATA POINTERS

HSA

SYSTEM MEMORY

L

R

L

R

L

R

TREE

RESULT BUFFER

GPU

KERNEL
Unified coherent memory

DATA POINTERS - CODE COMPLEXITY

static void run_hsa_path()
{
    /* Allocation and initialization */
    tree = (node *) clSMAlloc(context, CL_MEM_READ_ONLY,
                              num_nodes * sizeof(node), 0);
    initialize_nodes(tree, num_nodes);
    root = construct_BST(num_nodes, tree);

    search_keys = (int *) clSMAlloc(context, CL_MEM_READ_ONLY,
                                   num_search_keys * sizeof(int), 0);
    initialize_search_keys(search_keys, num_search_keys, sort_input);

    found_key_nodes = (node **) clSMAlloc(context, CL_MEM_WRITE_ONLY,
                                            num_search_keys * sizeof(node *), 0);
    memset(found_key_nodes, 0, num_search_keys * sizeof(node *));

    /* GPU work enqueue */
    clSetKernelArgVMPointer(search_kernel, 0, root);
    clSetKernelArgVMPointer(search_kernel, 1, search_keys);
    clSetKernelArgVMPointer(search_kernel, 2, &num_search_keys);
    clSetKernelArgVMPointer(search_kernel, 3, found_key_nodes);

    clEnqueueNDRangeKernel(queue, search_kernel, 1, NULL,
                            &num_search_keys, &preferredLocalSize, 0, NULL, &kernel_event);

    /* Cleanup */
    clSMFree(context, tree);
    clSMFree(context, found_key_nodes);
    clSMFree(context, search_keys);
}
Key features of HSA

- **hUMA** – Heterogeneous Unified Memory Architecture
- **hQ** – Heterogeneous Queuing
- **HSAIL** – HSA Intermediate Language
hQ: heterogeneous queuing

• Task queuing runtimes
  – Popular pattern for task and data parallel programming on Symmetric Multiprocessor (SMP) systems
  – Characterized by:
    • A work queue per core
    • Runtime library that divides large loops into tasks and distributes to queues
    • A work stealing scheduler that keeps system balanced
• HSA is designed to extend this pattern to run on heterogeneous systems
hQ: heterogeneous queuing

• How compute dispatch operates today in the driver model
hQ: heterogeneous queuing

• How compute dispatch improves under HSA
  – Application codes to the hardware
  – User mode queuing
  – Hardware scheduling
  – Low dispatch times
  – No Soft Queues
  – No User Mode Drivers
  – No Kernel Mode Transitions
  – No Overhead!
hQ: heterogeneous queuing

- AQL (Architected Queueing Language) enables any agent to enqueue tasks
hQ: heterogeneous queuing

- A work stealing scheduler that keeps system balanced
Advantages of the queuing model

- Today’s picture:
Advantages of the queuing model

- The unified shared memory allows to share pointers among different processing elements thus avoiding explicit memory transfer requests.
Advantages of the queuing model

- Coherent caches remove the necessity to perform explicit synchronization operation
Advantages of the queuing model

- The supported signaling mechanism enables asynchronous events between agents without involving the OS kernel.
Advantages of the queuing model

- Tasks are directly enqueued by the applications without using OS mechanisms
Advantages of the queuing model

• HSA picture:
Summary on the queuing model

• User mode queuing for low latency dispatch
  – Application dispatches directly
  – No OS or driver required in the dispatch path
• Architected Queuing Layer
  – Single compute dispatch path for all hardware
  – No driver translation, direct to hardware
• Allows for dispatch to queue from any agent
  – CPU or GPU
• GPU self-enqueue enables lots of solutions
  – Recursion
  – Tree traversal
  – Wavefront reforming
Other necessary HW mechanisms

- Task preemption and context switching have to be supported by all computing resources (also GPUs)
Key features of HSA

- **hUMA** – Heterogeneous Unified Memory Architecture
- **hQ** – Heterogeneous Queuing
- **HSAIL** – HSA Intermediate Language
HSA intermediate layer (HSAIL)

- A portable “virtual ISA” for vendor-independent compilation and distribution
  - Like Java bytecodes for GPUs
- Low-level IR, close to machine ISA level
  - Most optimizations (including register allocation) performed before HSAIL
- Generated by a high-level compiler (LLVM, gcc, Java VM, etc.)
  - Application binaries may ship with embedded HSAIL
- Compiled down to target ISA by a vendor-specific “finalizer”
  - Finalizer may execute at run time, install time, or build time
HSA intermediate layer (HSAIL)

- HSA compilation stack
  - Compiler Front Ends (OpenCL/C+/AMP/other)
  - LLVM IR
  - Compiler Back End
  - HSAIL

- HSA runtime stack
  - HSAIL
  - Language Runtime
  - HSA Runtime
  - HSA GPU Device
  - HSA CPU Device
  - Finalizer
  - User Space
  - Kernel Space
  - GPU Driver
  - OS System Services
  - GPU
  - CPU
HSA intermediate layer (HSAIL)

- Explicitly parallel
  - Designed for data parallel programming
- Support for exceptions, virtual functions, and other high level language features
- Syscall methods
  - GPU code can call directly to system services, IO, printf, etc
HSA intermediate layer (HSAIL)

- Lower level than OpenCL SPIR
  - Fits naturally in the OpenCL compilation stack
- Suitable to support additional high level languages and programming models:
  - Java, C++, OpenMP, C++, Python, etc…
HSA software stack

- HSA supports many languages
HSA and OpenCL

• HSA is an optimized platform architecture for OpenCL
  – Not an alternative to OpenCL
• OpenCL on HSA will benefit from
  – Avoidance of wasteful copies
  – Low latency dispatch
  – Improved memory model
  – Pointers shared between CPU and GPU
• OpenCL 2.0 leverages HSA Features
  – Shared Virtual Memory
  – Platform Atomics
HSA and Java

- Targeted at Java 9 (2015 release)
- Allows developers to efficiently represent data parallel algorithms in Java
- Sumatra “repurposes” Java 8’s multi-core Stream/Lambda API’s to enable both CPU or GPU computing
- At runtime, Sumatra enabled Java Virtual Machine (JVM) will dispatch selected constructs to available HSA enabled devices
HSA and Java

- Evolution of the Java acceleration before the Sumatra project
HSA software stack
HSA runtime

• A thin, user-mode API that provides the interface necessary for the host to launch compute kernels to the available HSA components
• The overall goal is to provide a high-performance dispatch mechanism that is portable across multiple HSA vendor architectures
• The dispatch mechanism differentiates the HSA runtime from other language runtimes by architected argument setting and kernel launching at the hardware and specification level
HSA runtime

• The HSA core runtime API is standard across all HSA vendors, such that languages which use the HSA runtime can run on different vendor’s platforms that support the API

• The implementation of the HSA runtime may include kernel-level components (required for some hardware components, ex: AMD Kaveri) or may be entirely user-space (for example, simulators or CPU implementations)
HSA runtime

Agent

- Start Program
- Exit Program

OpenCL Runtime

- Platform, Device, and Context Initialization
- SVM Allocation and Kernel Arguments Setting
- Build Kernel
- Command Queue
- Resource Deallocation

HSA Runtime

- HSA Runtime Initialization and Topology Discovery
- HSA Memory Allocation
- HSAIL Finalization and Linking
- Enqueue Dispatch Packet
- HSA Runtime Close
HSA taking platform to programmers

- Balance between CPU and GPU for performance and power efficiency
- Make GPUs accessible to wider audience of programmers
  - Programming models close to today’s CPU programming models
  - Enabling more advanced language features on GPU
  - Shared virtual memory enables complex pointer-containing data structures (lists, trees, etc) and hence more applications on GPU
  - Kernel can enqueue work to any other device in the system (e.g. GPU->GPU, GPU->CPU)
    - Enabling task-graph style algorithms, Ray-Tracing, etc.
HSA taking platform to programmers

• Complete tool-chain for programming, debugging and profiling
• HSA provides a compatible architecture across a wide range of programming models and HW implementations
HSA programming model

• Single source
  – Host and device code side-by-side in same source file
  – Written in same programming language

• Single unified coherent address space
  – Freely share pointers between host and device
  – Similar memory model as multi-core CPU

• Parallel regions identified with existing language syntax
  – Typically same syntax used for multi-core CPU

• HSAIL is the compiler IR that supports these programming models
Specifications and software

**HSA BUILDING BLOCKS**

http://hsafoundation.com
http://github.com/HSAFoundation

**HSA Hardware Building Blocks**

- **hUMA Shared Virtual Memory**
  - Single address space
  - Coherent
  - Pageable
  - Fast access from all components
  - Can share pointers

- **hQ Architected User-Level Queues**

- **Signals**

- **Context Switching**

- **Platform Atomics**

*Industry standard, architected requirements for how devices share memory and communicate with each other*

**HSA Software Building Blocks**

- **HSAIL**
  - Portable, parallel, compiler IR
  - Defined Memory Model

  *HSA Platform System Arch Specification*

- **HSA Runtime**
  - Create queues
  - Allocate memory
  - Device discovery

  *HSA System Runtime Specification*

- **Multiple high level compilers**
  - CLANG/LLVM/HSAIL
  - C++, OpenMP, OpenACC, Python, OpenCL™, etc

  *Industry standard compiler IR and runtime to enable existing programming languages to target the GPU*
HSA architecture V1

- GPU compute C++ support
- User Mode Scheduling
- Fully coherent memory between CPU & GPU
- GPU uses pageable system memory via CPU pointers
- GPU graphics pre-emption
- GPU compute context switch
Partners roadmaps

HETEROGENEOUS SYSTEM ARCHITECTURE ROADMAP

2011
- Physical Integration
  - Integrate CPU and GPU in Silicon
  - Unified Memory Controller
  - Common Manufacturing Technology

2012
- Optimized Platforms
  - GPU Compute C++ Support
  - User Mode Scheduling
  - Bi-Directional Power Mgmt Between CPU and GPU

2013
- Architectural Integration
  - Unified Address Space for CPU and GPU
  - GPU Uses Pageable System Memory via CPU Pointers
  - Fully Coherent Memory Between CPU & GPU

2014
- System Integration
  - GPU Compute Context Switch
  - GPU Graphics Preemption
  - Quality of Service
Partners roadmaps

AMD “CARRIZO” NOTEBOOK AND AIO PLATFORM

“Carrizo”

- "Excavator" (XV) CPU with ~30% perf increase at 15W
  - 4 XV cores, 2 MB total L2
- AMD Radeon 3rd Generation Graphics Core Next (GCN)
  - 8 GFX CUs, 2 RBs, Higher memory efficiency, Delta Color Compression
  - Full HSA: Hi Perf Bus for Gfx & DRAM, Fine-grain Preemption for Context Switches
  - DirectX 12
- Multimedia
  - Universal Video Decoder (UVD6); 9-18x 1080p 30fps H.264 decode
  - Video Compression Engine (VCE3.1): 9x 1080p 30fps H.264 encode
  - Audio Co-Processor (ACP2)
- Integrated Platform Security Processor (Trust Zone)
  - Dedicated, Trustzone compatible security subsystem
  - TPM2.0, crypto acceleration, secure boot
- Memory Technology
  - Up to 2-channels DDR3-2133
  - Dual SoDIMM per channel
- Display and I/O
  - DCE11 – Display Controller Engine
  - Up to 3 Display interfaces/heads, HDMI 2.0
  - PCIe Gen3 x8 for dGPU expansion, PCIe Gen3 x4 for GPP
  - AMD wireless display support (Miracast)
- Power Management
  - Connected Standby, STAPM, PPT/TDC/EDC tracking, BBB
- Integrated FCH
  - 4x USB3.0/2.0, 4x USB2.0, 2x SATA3, SD, GPIO, SPI, I2S, I2C, UART
- Targeted notebook / convertible form factors
  - BGA (FP4), ~12W-35W TDPs

2015
Partners roadmaps

ARM AND HSA

ARM, as a founder member, has been committed to the HSAF since launch

- Actively contributes to the HSA specifications and working groups
- Is committed to the continued development of this important standard

ARM customer base is showing increasing interest in HSA features for their next generation SoCs

ARM customers can already build real heterogeneous systems based on, for example:

- ARM Cortex-A72 high performance application processor
- ARM Mali-T880 compute enabled GPU
- ARM CoreLink CCI-500 cache coherent interconnect
- ARM CoreLink CCN cache coherent network family

ARM is actively developing next generation processor and interconnect IP to extend the system capabilities aligned with HSA standards including:

- Full memory coherency
- Shared Virtual Memory

© Copyright 2015 HSA Foundation. All Rights Reserved.
Partners roadmaps

HSA ROADMAP AT MEDIATEK

SMP
Symmetric Multi-Processing

HMP – 2013
Heterogeneous Multi-Processing

HC – 2015
Heterogeneous Computing

HSA
Heterogeneous System Architecture

Copyright © MediaTek Inc. All rights reserved.

10/5/2015
Partners roadmaps

EFFICIENT EXECUTION OF WORKLOADS IN AN SOC ENVIRONMENT

Heterogeneous System Architecture
- Efficient execution on CPU, GPU and beyond
- Consistent programming model

Efficiency building blocks
- Full cache coherency
  - Reduced bandwidth and improved efficiency
- User mode queues
  - Significant reduction in work item latency
- Signals
  - Efficient control over scheduling and synchronization

Benefits for Compute and also Graphics

Staged roll-out from 2016 onwards
Partners roadmaps

IMAGINATION HSA COMPLIANT IP COMING SOON

Imagination Smart Vision IP Platform

We will be rolling out:

- HSA across all MIPS I-class and P-class CPUs
- HSA across all PowerVR GPUs
- HSA compliant fabric solutions