Heterogeneous System Architecture (HSA): Software Ecosystem for CPU/GPU/DSP and other accelerators

Timour Paltashev
Graphics IP Engineering Division
Advanced Micro Devices, Sunnyvale, California, U.S.A.
{timour.paltashev}@amd.com

Abstract

This STAR report describes the essentials of Heterogeneous System Architecture (HSA) with introduction and motivation for HSA, architecture definition and configuration examples. HSA performance advantages are illustrated on few sample workloads. Kaveri APU - first AMD HSA-based product is briefly described.

Keywords: *GPU, CPU, DSP, APU, heterogeneous architecture.*

1. INTRODUCTION

HSA is a new hardware architecture that integrates heterogeneous processing elements into a coherent processing environment. Coherent processing as a technique ensures that multiple processors see a consistent view of memory, even when values in memory may be updated independently by any of those processors. Memory coherency has been taken for granted in homogeneous multiprocessor and multi-core systems for decades, but allowing heterogeneous processors (CPUs, GPUs and DSPs) to maintain coherency in a shared memory environment is a revolutionary concept. Ensuring this coherency poses difficult architectural and implementation challenges, but delivers huge payoffs in terms of software development, performance and power. The ability for CPUs, DSPs and GPUs to work on data in coherent shared memory eliminates copy operations and saves both time and energy. The programs running on a CPU can hand work off to a GPU or DSP as easily as to other programs on the same CPU; they just provide pointers to the data in the memory shared by all three processors and update a few queues. Without HSA, CPU-resident programs must bundle up data to be processed and make input-output (I/O) requests to transfer that data via device drivers that coordinate with the GPU or DSP hardware. HSA allows developers to write software without paying much attention to the processor hardware available on the target system configuration with or without GPU, DSP, video hardware and other types of specialized compute accelerators.

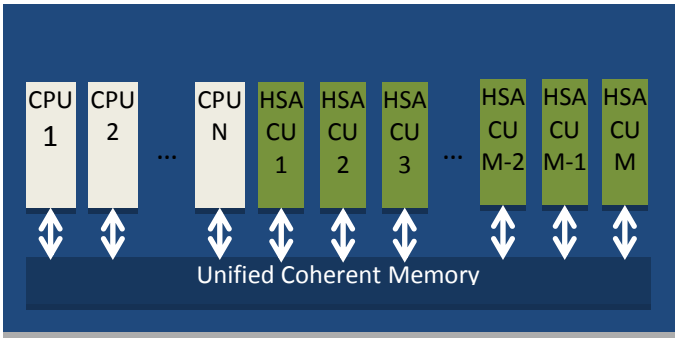

Figure 1: **Generic HSA Accelerated Processing Unit (APU)**

Fig.1 depicts generic HSA APU with multiple CPU cores and accelerated compute units (CU) which may include any type.

2. HSA ESSENTIAL FEATURES FOR USERS

Essential HSA features include:

- Full programming language support
- User Mode Queueing
- Heterogeneous Unified Memory Access (hUMA)
- Pageable memory
- Bidirectional coherency
- Compute context switch and preemption

Shared page table support. To simplify OS and user software, HSA allows a single set of page table entries to be shared between CPUs and CUs. This allows units of both types to access memory through the same virtual address. The system is further simplified in that the operating system only needs to manage one set of page tables. This enables Shared Virtual Memory (SVM) semantics between CPU and CU.

Page faulting. Operating systems allow user processes to access more memory than is physically addressable by paging memory to and from disk. Early CU hardware only allowed access to pinned memory, meaning that the driver invoked an OS call to prevent the memory from being paged out. In addition, the OS and driver had to create and manage a separate virtual address space for the CU to use. HSA removes the burdens of pinned memory and separate virtual address management, by allowing compute units to page fault and to use the same large address space as the CPU.

User-level command queuing. Time spent waiting for OS kernel services was often a major performance bottleneck in prior throughput computing systems. HSA drastically reduces the time to dispatch work to the CU by enabling a dispatch queue per application and by allowing user mode process to dispatch directly into those queues, requiring no OS kernel transitions or services. This makes the full performance of the platform available to the programmer, minimizing software driver overheads.

Hardware scheduling. HSA provides a mechanism whereby the CU engine hardware can switch between application dispatch queues automatically, without requiring OS intervention on each switch. The OS scheduler is able to define every aspect of the switching sequence and still maintains control. Hardware scheduling is faster and consumes less power.

Coherent memory regions. In traditional GPU devices, even when the CPU and GPU are using the same system memory region, the GPU uses a separate address space from the CPU, and the graphics driver must flush and invalidate GPU caches at required intervals in order for the CPU and GPU to share results. HSA embraces a fully coherent shared memory model, with unified addressing. This provides programmers with the same coherent memory model that they enjoy on SMP CPU systems.

This enables developers to write applications that closely couple CPU and CU codes in popular design patterns like producer-consumer. The coherent memory heap is the default heap on HSA and is always present. Implementations may also provide a non-coherent heap for advance programmers to request when they know there is no sharing between processor types.

The HSA platform is designed to support high-level parallel programming languages and models, including C++ AMP, C++, C#, OpenCL, OpenMP, Java and Python, as well as few others. HSA-aware tools generate program binaries that can execute on HSA-enabled systems supporting multiple instruction sets (typically, one for the LCU and one for the TCU) and also can run on existing architectures without HSA support.

Program binaries that can run on both CPUs and CUs contain CPU ISA (Instruction Set Architecture) for CPU unit and HSA Intermediate Language (HSAIL) for the CU. A *finalizer* converts HSAIL to CU ISA. The finalizer is typically lightweight and may be run at install time, compile time, or program execution time, depending on choices made by the platform implementation.

HSA architecture example platform is depicted on Figure 2.

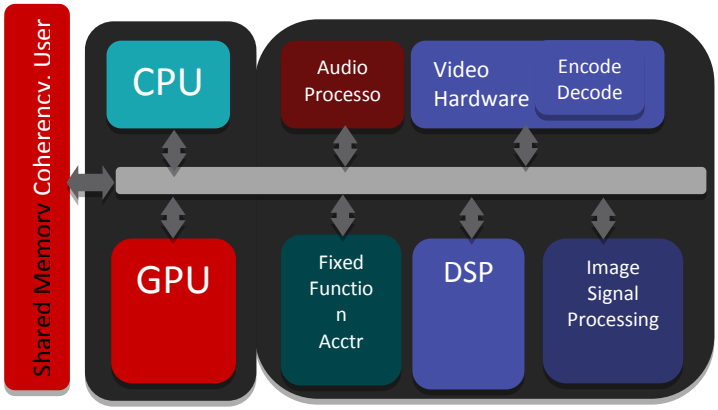

Figure 2: **HSA architecture example platform.**

3. HSA IMPLEMENTATION AND CONCEPTS

Unified Programming Model. General computing on GPUs has progressed in recent years from graphics shader-based programming to more modern APIs like DirectCompute and OpenCL™. While this progression is definitely a step forward, the programmer still must explicitly copy data across address spaces, effectively treating the GPU as a remote processor.

Task programming APIs like Microsoft's ConcRT, Intel's Thread Building Blocks, and Apple's Grand Central Dispatch are recent innovations in parallel programming. They provide an easy to use task-based programming interface, but only on the CPU. Similarly, Thrust from NVIDIA provides a similar solution on the GPU.

HSA moves the programming bar further, enabling solutions for task parallel and data parallel workloads as well as for sequential workloads. Programs are implemented in a single programming environment and executed on systems containing both CPUs and CUs.

HSA provides a programming interface containing queue and notification functions. This interface allows devices to access load-balancing and device-scaling facilities provided by the higher-level task queuing library. The overall goal is to allow developers to leverage both CPU and CU devices by writing in task-parallel languages, like the ones they use today for multicore CPU systems. HSA's goal is to enable existing task and data-parallel languages and APIs and enable their natural evolution without requiring the programmer to learn a new HSA-specific programming language. The programmer is not tied to a single language, but rather has available a world of possibilities that can be leveraged from the ecosystem.

Queuing. HSA devices communicate with one another using queues. Queues are an integral part of the HSA architecture. CPUs already send compute requests to each other in queues in popular task queuing run times like ConcRT and Threading Building Blocks. With HSA, both CPUs and CUs can queue tasks to each other and to themselves.

The HSA runtime performs all queue allocation and destruction. Once an HSA queue is created, the programmer is free to dispatch tasks into the queue. If the programmer chooses to manage the queue directly, then they must pay attention to space available and other issues. Alternatively, the programmer can choose to use a library function to submit task dispatches.

A queue is a physical memory area where a producer places a request for a consumer. Depending on the complexity of the HSA hardware, queues might be managed by any combination of software or hardware. Queue implementation internals are not exposed to the programmer.

Hardware-managed queues have a significant performance advantage in the sense that an application running on a CPU can queue work to a CU directly, without the need for a system call. This allows for very low-latency communication between devices, opening up a new world of possibilities. With this, the CU device can be viewed as a peer device, or a co-processor.

CPUs can also have queues. This allows any device to queue work for any other device.

4. CONCLUSION

The current state of the art of GPU/DSP and other high-performance computing is not flexible enough for many of today's computational problems.

HSA is a unified computing framework. It provides a single address space accessible to both CPU and GPU (to avoid data copying), user-space queuing (to minimize communication overhead), and preemptive context switching (for better quality of service) across all computing elements in the system. HSA unifies CPUs and GPU/DSPs into a single system with common computing concepts, allowing the developer to solve a greater variety of complex problems more easily.

5. REFERENCES

[1] Heterogeneous System Architecture: A Technical Review, Advanced Micro Devices, Rev. 1.0.

[2] http://developer.amd.com/resources/heterogeneous-computing/

About the author

Timour Paltashev is a professor at Northwestern Polytechnic University, College of Engineering and Senior Manager in Advanced Micro Devices. His contact email is timpal@mail.npu.edu.