1.1 "Architecting the Future through Heterogeneous Computing"

Lisa T. Su, Senior Vice President and General Manager,
Advanced Micro Devices, Austin, TX

1. Introduction

1.1 Current Computing Challenges

Anyone wishing to drive advances in computing technology must carefully negotiate key trade-offs. First, reducing power consumption is increasingly critical. Consumers want improved battery life, size, and weight for their laptops, tablets, and smartphones. Likewise, data-center power demands and cooling costs continue to rise. Concurrent is the demand for improved performance that enables compelling new user experiences. Users want to access devices through more natural interfaces (speech and gesture); they also want devices to manage ever-expanding volumes of data (home movies, pictures, and a world of content available in the cloud). An essential part of making these new user experiences available is programmer productivity; software developers must easily be able to tap into new capabilities by using familiar, powerful programming models. Finally, it is increasingly important that software be supported across a broad spectrum of devices; developers cannot sustain today's trend of re-writing code for an ever expanding number of different platforms.

To navigate this complex set of requirements, the computer industry requires a different approach – a more efficient approach to computer architecture. We need an approach that promises to deliver improvement across all four of these vectors: power, performance, programmability, and portability.

1.2 The Dawn of Heterogeneous Computing

[het·er·o·ge·ne·ous: *Composed of parts of different kinds; having widely dissimilar elements or constituents.]*

When applied to the world computing, heterogeneous means there are various programmable computing elements on a single platform, each with specific characteristics that excel at certain kinds of tasks. There are many examples of platforms with heterogeneous components, including network and IO processing, signal processing, graphics processing, and so on. So, what is changing now?

Historically, these computing elements have been treated as separate programming environments, each with its own programming language and tools, private memory, and separate address space. Accessing these different elements required explicitly copying data between address spaces and sending commands through driver interfaces to software written by a different team of programmers. This approach works well only when there is a clear delineation of work to be done. Today, however, the lines of delineation are blurring. In the drive to improve efficiency on power-constrained platforms, programmers are seeking more direct access to, and control of, the different programmable elements on a system in order to execute tasks with greater power efficiency. The leading example of this trend is the Graphics-Processing Unit (GPU), originally designed to execute specialized graphics computations in parallel. Over time, GPUs have become more powerful and more generalized, allowing them to be applied to general-purpose parallel computing tasks with excellent power efficiency.

Today, a growing number of mainstream applications are starting to leverage the high performance and power efficiency achievable only through such highly parallel computation. But current CPUs and GPUs are still designed as separate processing elements that do not work together efficiently – and they are cumbersome to program. Each has a separate memory space, requiring an application to explicitly copy data from CPU to GPU and back again. Also, a program running on a CPU queue works with the GPU. using system calls through a device-driver stack managed by a completely separate scheduler. This introduces significant dispatch latency, with overhead that makes the process worthwhile only when the application requires a very large amount of parallel computation. Further, it is impossible today for a program running on the GPU to directly generate work-items – either for itself or for the CPU. As a result of these inefficiencies, applications often resort to running compute loads on elements not ideally suited for the task at hand.

1.3 Heterogeneous System Architecture (HSA)

To fully exploit the capabilities of heterogeneous computing elements, it is essential for computer system designers to think differently. They must re-architect computer systems to tightly integrate the disparate compute elements on a platform into an evolved central processor, while providing a programming path that does not require fundamental changes for software developers. This is the primary goal of HSA.

HSA represents a new era in computer architecture. It delivers a high level of efficiency by creating an improved processor design that exposes the benefits and capabilities of mainstream programmable compute elements, working together seamlessly (see Figure 1.1.1*)*. With HSA, applications can create data structures in a single, unified address space; and they can initiate work-items on the hardware most appropriate for a given task. Sharing data between compute elements is as simple as sending a pointer. Multiple compute tasks can work on a single coherent memory region, using barriers and atomic memory operations, as needed, to maintain data synchronization (just as multi-core CPUs do now).

1.4 Who Benefits from HSA?

We are witnessing an explosion of data across every segment of computing, from client to server to embedded markets. Without more efficient means of handling this data, we may see a crisis in cost, power consumption, and quality of service. This trend drives the need for an evolved computing architecture that can process data more efficiently.
In the client space, cameras are one of the biggest factors driving this trend. Consumers are capturing a growing quantity of video and images with increasing resolution – all of which they will want to search, edit, and compress, for easy transfer and storage. Cameras also are becoming the basis of new human-computer interfaces, such as gesture recognition, biometric authentication, and augmented reality. Here, video data must be analyzed in real time, without consuming a device's processing power or draining the battery.

The situation is similar in the server and cloud markets, where there is also an unprecedented growth of data: In 2010, 245 Exabytes crossed the Internet. This number is expected to grow to a Zetabyte (1000 Exabytes) in 2015. Unfortunately, the networking bandwidth required to handle this is not growing at the same pace. Moreover, before any of this data can be made useful to any consumer, it must be compressed for storage, uncovered via search, and analyzed to extract context and meaning. All of this is driving the need for better, more efficient computing architectures that can process data in parallel to deliver higher-performance and lower-power use. This is what HSA is designed to deliver.

2. HSA Key Concepts

2.1 HSA Goals

The HSA strategy creates a single unified programming platform that provides a strong foundation for the development of languages, frameworks, and applications, all exploiting parallelism. HSA goals include:

- Removing the CPU - GPU programmability barrier.
- Reducing CPU - GPU communication latency.
- Opening the programming platform to a wider range of applications by enabling existing programming models.
- Creating a basis for the inclusion of additional processing elements beyond the CPU (called *Latency Compute Units*, or LCUs), and GPU (called *Throughput Compute Units*, or TCUs).

HSA exploits the abundant data parallelism in computational workloads, today and in the future, in a power-efficient manner. It also provides continued support for traditional programming models and computer architectures.

While HSA requires certain functionality to be available in hardware, it also allows room for innovation. It enables a wide range of solutions that span both functionality (of small and complex systems), and time (with backward and forward compatibility). By standardizing the interface between the software stack and the hardware, HSA enables two dimensions of simultaneous innovation:

- Software developers can target a large and future-proof installed hardware base.
- Hardware vendors can differentiate core IP while maintaining compatibility with the existing and future software ecosystems.

2.2 Unified Programming Model

Today, GPUs support general-purpose programming APIs such as DirectCompute and OpenCL™. While these APIs are a step forward, the programmer still must explicitly copy data across address spaces, effectively treating the GPU as a remote processor.

Task programming APIs, such as 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, NIVIDIA's Thrust provides a 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 LCUs and TCUs.

HSA provides a programming interface containing queue and notification functions. This interface lets devices access load-balancing and device-scaling provided by the higher-level task queuing library. The goal is to allow developers to leverage both LCU and TCU devices by writing in task-parallel languages, such as the ones used today for multicore CPU systems. The goal of HSA- is to enable existing - and data-parallel languages and APIs, as well as enable their natural evolution, without requiring the programmer to learn a new HSA-specific programming language. Thus, the programmer is not tied to a single language, but rather has available many possibilities that can be leveraged from the ecosystem.

2.3 Unified Address Space

Achieving the goal of a unified programming model requires a number of hardware and software innovations: First, HSA defines a unified address space across LCU and TCU devices. All HSA devices must support hardware virtual-address translation, so that a pointer (that is, a virtual address) can be freely passed between devices, and shared page tables to ensure that identical pointers access the same physical address. Thus, an HSA-specific Memory Management Unit (HMMU) supports the unified address space. The HMMU allows the TCUs to share page-table mappings with the CPU. HSA supports unaligned accesses for loads and stores; however, atomic accesses must be aligned.

Today, many compute problems require much larger memory spaces than can be provided by traditional GPUs, whether this is the local memory of a discrete GPU, or the pinned-system memory used by an APU. But, partitioning a program to repeatedly use a small memory pool can require a huge programming effort; for that reason, large workloads often are not ported onto the GPU. If the HSA-throughput engine uses the same pageable virtual-address space as the CPU, problems can be ported easily to an HSA system, without extra coding effort. This significantly increases computational performance of programs requiring very large data sets.

A unified address space also allows data structures containing pointers (such as linked lists, and various forms of tree and graph structures) to be freely used by both LCUs and TCUs. Today, such data structures require special handling by the programmer, and often they are the main reason why certain algorithms cannot be ported to a GPU. With HSA, this is handled transparently.

HSA platforms also provide hardware support for a fully-coherent shared-memory model. This provides programmers with the same coherent memory model that they enjoy on SMP CPU systems; Thus, it enables developers to write applications that closely couple LCU and TCU codes in popular design patterns like producer-consumer. The coherent memory heap is the default heap on HSA, and is always present. Implementations can also provide a non-coherent heap, which advanced programmers can request when they know that there is no sharing between processor types (see Figure 1.1.2).

2.2 Queuing

A queue is a physical memory area where a producer places a request for a consumer. HSA devices communicate with one another using queues. Latency processors already send compute requests to each other in queues in popular task queuing run times like ConcRT and Threading Building Blocks. With HSA, latency processors and throughput processors can queue tasks for each other and 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.

Hardware-managed queues have a significant performance advantage because an application running on an LCU can queue work directly to a TCU, without the need for a system call. This allows for very-low-latency communication between devices, opening up many new possibilities. Now, the TCU device can be viewed as a peer device, or a co-processor.

LCUs also can have queues. This allows any device queue to work for any other device. Specifically:

- **An LCU can queue to a TCU**. This is typical of OpenCL™-style queuing.
- **A TCU can queue to another TCU (including itself)**. This lets a workload running on a TCU queue additional work without a round-trip to the CPU, which would add considerable (and often unacceptable) latency.
- **A TCU can queue to an LCU**. This allows a workload running on a TCU to request system operations, such as memory allocation or I/O.

This concept is shown in Figure 1.1.3.

2.3 Pre-Emption and Context Switching

TCUs provide excellent opportunities for offloading computation, but the current generation of TCU hardware does not support pre-emptive context switching; therefore, it is difficult to manage in a multi-process environment. This has presented several problems:

- A rogue process might occupy the hardware for an arbitrary amount of time, because processes cannot be pre-empted.
- A faulted process might not allow other jobs to execute on the unit until the fault has been handled; again, because the faulted process cannot be pre-empted.

HSA hardware supports job pre-emption, flexible job scheduling, and fault-handling mechanisms to overcome the above drawbacks. These concepts allow an HSA system (a combination of HSA hardware and HSA system software) to maintain high throughput in a multi-process environment. (A traditional multi-user OS exposes the underlying hardware.)

To do this, HSA-compliant hardware provides mechanisms to guarantee that no TCU process (graphics or compute) can prevent other TCU processes from making progress.

2.4 HSA Intermediate Language (HSAIL)

HSA exposes the parallel nature of TCUs through the HSA Intermediate Language (HSAIL). HSAIL is translated onto the underlying hardware's ISA (instruction set architecture). While HSA TCUs are often embedded in powerful graphics engines, the HSAIL language is focused purely on compute, and does not expose graphics-specific instructions. The underlying hardware executes the translated ISA without awareness of HSAIL.

The smallest unit of execution in HSAIL is called a *work-item*. A work-item has its own set of registers, can access assorted system-generated values, and can access private (work-item local) memory. Work-items use regular loads and stores to access private memory, which resides in a special, private data-memory segment.

Work-items are organized into *work-groups*. Work-groups can share data through group memory, again using normal loads and stores. Memory shared across a work-group is identified by address. Each work-item in a work-group has a unique identifier called its local ID. Each work-group executes on a single compute unit, and HSA provides special synchronization primitives for use within a work-group.

A work-group is part of a larger group called an *N-Dimensional Range* (NDRange). Each work-group in an NDRange has a unique work-group identifier called its global ID, available to any work-item within the NDRange. Work-items within an NDRange can communicate through memory, because the address space (excluding group and private data) is shared across all work-items and is coherent with the LCU. Figure 1.1.3 shows an NDRange, a work-group, and a work-item.

A *finalizer* translates HSAIL into the underlying hardware ISA at runtime. The finalizer also enforces HSA Virtual Machine semantics as part of the translation to ISA. Thus, the underlying hardware architecture does not have to adhere strictly to the HSA Virtual Machine. The HSA Virtual Machine can also be implemented on an LCU by having the finalizer convert HSAIL to LCU ISA.

HSAIL has a unified memory view: The virtual address, rather than a special instruction encoding, determines whether a load or store address is private, shared among work-items in a work-group, or globally visible. This relieves the programmer of much of the burden of memory management. Memory between the LCU and TCU cores is coherent. The HSA Memory Model is based on a relaxed consistency model, but is consistent with the memory models defined for C++11, .net, and Java. Because the entire address space of the LCU is available to the TCU, the programmer can handle large data sets without special code in order to stream data into, and out of, the TCU.

3. HSA at Work

The HSA team at AMD analyzed the performance of Haar Face Detect, a commonly used multi-stage video analysis algorithm used to identify faces in a video stream. The team compared a CPU/GPU implementation in OpenCL™ against an HSA implementation. The HSA version seamlessly shares data between CPU and GPU, without memory copies or cache flushes, because it assigns each part of the workload to the most appropriate processor with minimal dispatch overhead. The net result was a 2.3× relative performance gain at a 2.4× reduced power level*. This level of performance is not possible using only a multicore CPU, only a GPU, or even combined CPU and GPU with today's driver model. More importantly, this is accomplished using just simple extensions to C++, not a totally different programming model.

4. An Industry Approach

To reach beyond mere niche adoption, it is essential to provide a deployment path beyond the domain of any single hardware vendor. The ultimate goal for many software developers is, understandably, "write once, run everywhere". But, this requires a unified install-base across all platforms and devices- this is the HSA vision. Consequently, the HSA Foundation (HSAF) was formed as an open-industry-standards body to unify the computing industry around a common approach. The founding members of HSAF are: AMD, ARM, Imagination Technologies, MediaTek, Texas Instruments, Samsung Electronics, and Qualcomm.

The HSA Foundation aims to help system designers integrate diverse computing elements (such as CPUs and GPUs) in a way that eliminates the inefficiencies of sharing data and sending work-items between them. HSA provides a single target for low-level software and tools via HSAIL (See the HAS solution stack im Figure 1.1.4)). HSAIL is sufficiently flexible, and yet low-level enough, to allow each hardware vendor to efficiently map to its particular underlying hardware design. Most importantly, HSAIL frees the programmer from the burden of tailoring a program to a specific hardware platform – the same code runs on target systems with different CPU/GPU configurations.

5. Getting Involved with HSA

HSA is about delivering new improved user experiences through advances in computing architectures that deliver enhanced benefits across all four key vectors: improved power efficiency; improved performance; improved programmability; and improved portability across computing devices. To achieve this vision, the HSA Foundation is open to contributions from like-minded professionals across the computing industry - IHVs, OEMs/ODMs, OSVs, language and tools vendors, library and middleware vendors, as well as application vendors – who want to help realize the next era in computer system architecture and innovation. For information on HSA, HSAF, foundation membership, and contact information, please visit www.HSAFoundation.com.

6. References

"Heterogeneous computing" See
http://en.wikipedia.org/wiki/Heterogeneous_computing

A.L. Varbenescu, P. Hijma, R. v. Nieuwpoort, H. Bal,
"Towards an Effective Unified Programming Model for Many-Cores"
(SC12, November 10-16, 2012, Salt Lake City, Utah, USA).
See http://www.cs.vu.nl/~rob/papers/APDCM-2011.pdf

G. Kyriazis, "Heterogeneous System Architecture: A Technical Review"
(Sunnyvale, CA: AMD, 2012). See
http://developer.amd.com.php53-23.ord1-1.websitetestlink.com/wordpress/media/2012/10/hsa10.pdf

J. E. Stone, D. Gohara, G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems" in: Computing in Science & Engineering vol.12, issue 3, pp. 66-73 (May-June 2010).
See http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2964860/

N. Rubin, "HSAIL. An Introduction to the HSA Intermediate Language"
(AMD: Fusion12 Developer Summit).
See http://www.slideshare.net/hsafoundation/hsail-final-11junepptx#btnNext

For further information,
see http://developer.amd.com/resources/amd-fds-videos/

*HW Configuration:
4GB RAM; Windows 7 (64-bit); OpenCL™ 1.1
APU: AMD A10 4600M with Radeon™ HD Graphics
CPU: 4 cores @ 2.3MHz (turbo 3.2GHz)
GPU: AMD Radeon HD 7660G, 6 compute units, 685MHz

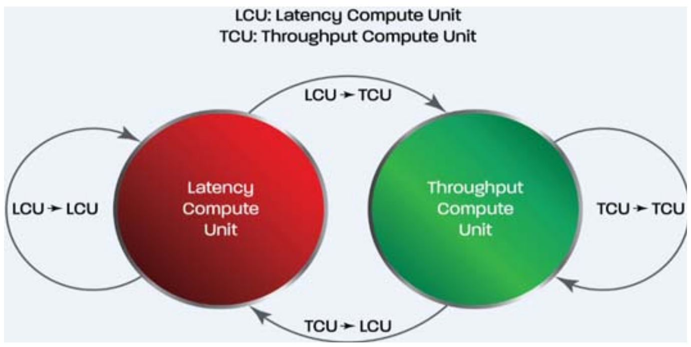

Figure 2.1: Compute Unit Queuing

Figure: 1.1.1: Compute Unit Queuing.

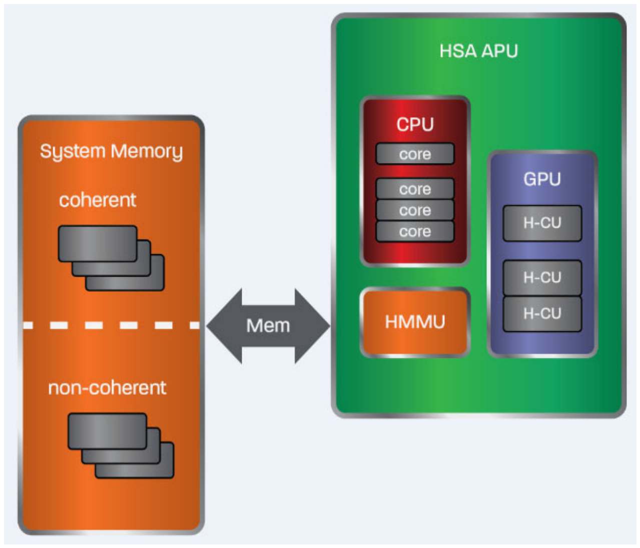

Figure: 1.1.2: An HSA Platform

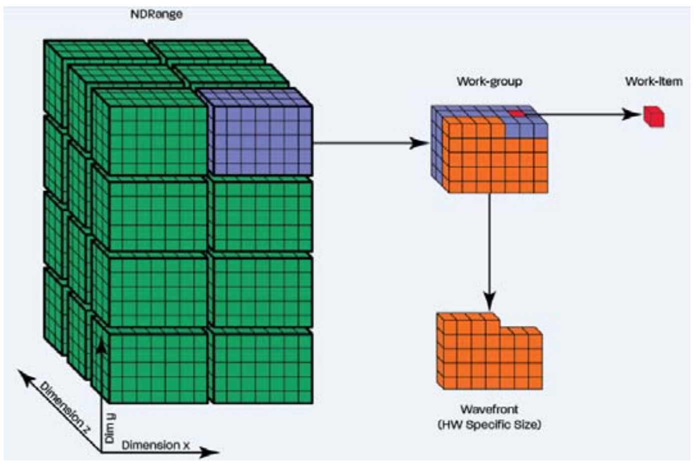

Figure: 1.1.3: The NDRange Heirarchy: NDRange, Work-Group, Work-Item.

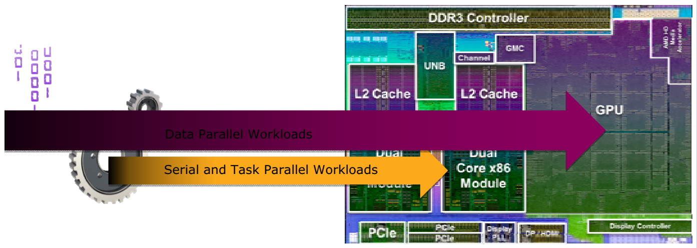

Figure 1.1.1: An HSA Accelerated Processing Unit

Figure: 1.1.4: An HSA Accelerated Processing Unit.

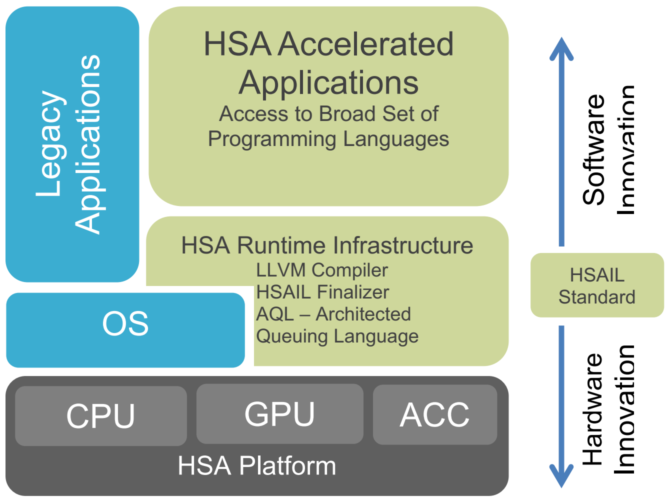
Legacy Applications
HSA Accelerated Applications
Access to Broad Set of Programming Languages
HSA Runtime Infrastructure
LLVM Compiler
HSAIL Finalizer
AQL – Architected Queuing Language
OS
CPU
GPU
ACC
HSA Platform
Software Innovation
HSAIL Standard
Hardware Innovation

Figure: 1.1.5: The HSA Solution Stack .