Heterogeneous System Architecture: A Technical Review

George Kyriazis, AMD

Rev. 1.0
8/30/2012

Table of Contents

1.Introduction

1.1. Overview

The Heterogeneous System Architecture (HSA) provides a unified view of fundamental computing elements. HSA allows a programmer to write applications that seamlessly integrate CPUs (called *latency compute units*) with GPUs (called *throughput compute units*), while benefiting from the best attributes of each.

GPUs have transitioned in recent years from pure graphics accelerators to more general-purpose parallel processors, supported by standard APIs and tools such as OpenCL™ and DirectCompute. Those APIs are a promising start, but many hurdles remain for the creation of an environment that allows the GPU to be used as fluidly as the CPU for common programming tasks: different memory spaces between CPU and GPU, non-virtualized hardware, and so on. HSA removes those hurdles, and allows the programmer to take advantage of the parallel processor in the GPU as a peer or co-processor to the traditional multi-threaded CPU.

1.2. Goals

The essence of the HSA strategy is to create a single unified programming platform providing a strong foundation for the development of languages, frameworks, and applications that exploit parallelism. More specifically, HSA's 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 and GPU.

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

1.3. Features

This section defines some commonly used HSA terms and features.

The HSA architecture deals with two kinds of compute units:

- A **latency compute unit (LCU)** is a generalization of a CPU. An LCU supports both its native CPU instruction set and the HSA intermediate language (HSAIL) instruction set (described in section 2.5 below).

- A **throughput compute unit** (TCU) is a generalization of a GPU. A TCU supports only the HSAIL instruction set. TCUs perform very efficient parallel execution.

An HSA application can run on a wide range of platforms consisting of both LCUs and TCUs. The HSA framework allows the application to execute at the best possible performance and power points on a given platform, without sacrificing flexibility. At the same time, HSA improves programmability, portability and compatibility.

Prominent architectural features of HSA include:

- **Shared page table support**. To simplify OS and user software, HSA allows a single set of page table entries to be shared between LCUs and TCUs. 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 LCU and TCU.

- **Page faulting**. Operating systems allow user processes to access more memory than is physically addressable by paging memory to and from disk. Early TCU 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 TCU 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 TCU 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 TCU 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 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 may also provide a non-coherent heap for advance programmers to request when they know there is no sharing between processor types.

Figure 1.1 shows a simple HSA platform. The HSA APU (Accelerated Processing Unit) contains a multi-core CPU, a GPU with multiple HSA compute units (H-CUs), and the HSA memory management unit (HMMU). These components communicate with coherent and non-coherent system memory.

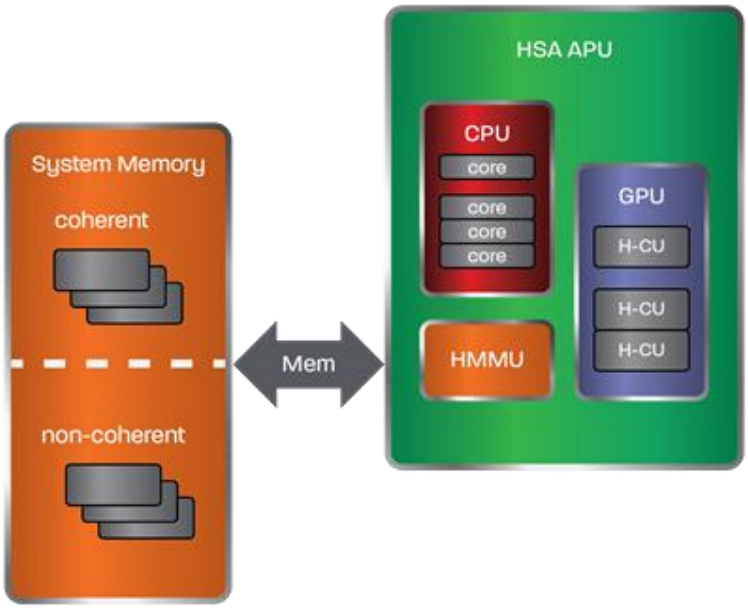

Figure 1.1 HSA Platform

1.4. Programming Interfaces

The HSA platform is designed to support high-level parallel programming languages and models, including C++ AMP, C++, C#, OpenCL, OpenMP, Java and Python, to name a few.

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 LCUs and TCUs contain CPU ISA for the latency unit and HSA Intermediate Language (HSAIL) for the TCU. A *finalizer* converts HSAIL to TCU 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.

1.5. Implementation Components

An HSA implementation is a system that passes the HSA Compliance Test Suite. It consists of:

- A heterogeneous hardware platform that integrates both LCUs and TCUs, which operate coherently in shared memory.
- A software compilation stack consisting of a compiler, linker and loader.
- A user-space runtime system, which also includes debugging and profiling capabilities.
- Kernel-space system components.

This paper first discusses the main concepts behind HSA. Then it examines each of the above components in more detail.

2. Concepts

HSA is a system architecture encompassing both software and hardware concepts. Hardware that supports HSA does not stand on its own, and similarly the HSA software stack requires HSA-compliant hardware to deliver the system's capabilities.

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 (small vs. complex systems) and time (backwards and forwards 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 hardware install base.
- Hardware vendors can differentiate core IP while maintaining compatibility with the existing and future software ecosystems.

This section describes important HSA concepts.

2.1. 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 LCUs and TCUs.

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 LCU and TCU 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.

2.2. Unified Address Space

HSA defines a unified address space across LCU and TCU devices. All HSA devices support virtual address translation: a pointer (that is, a virtual address) can be freely passed between devices, and shared page tables ensure that identical pointers resolve to the same physical address.

Internally, HSA implementations provide several special memory types (some on chip, some in caches, and some in system memory), but there is no need for special loads or stores. A TCU memory operation (including atomic operations) produces the same effects as a LCU operation using the same address.

All memory types are managed in hardware. 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 have to be aligned.

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

In addition, a unified address space 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 are the main reason why certain algorithms cannot be ported to a GPU. With HSA, this is handled transparently.

2.3. Queuing

HSA devices communicate with one another using queues. Queues are an integral part of the HSA architecture. 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 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 LCU can queue work to a TCU 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 TCU device can be viewed as a peer device, or a co-processor.

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

- **A LCU can queue to a TCU**. This is the typical scenario of OpenCL™-style queuing.
- **A TCU can queue to another TCU (including itself)**. This allows a workload running on a TCU to 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 2.1.

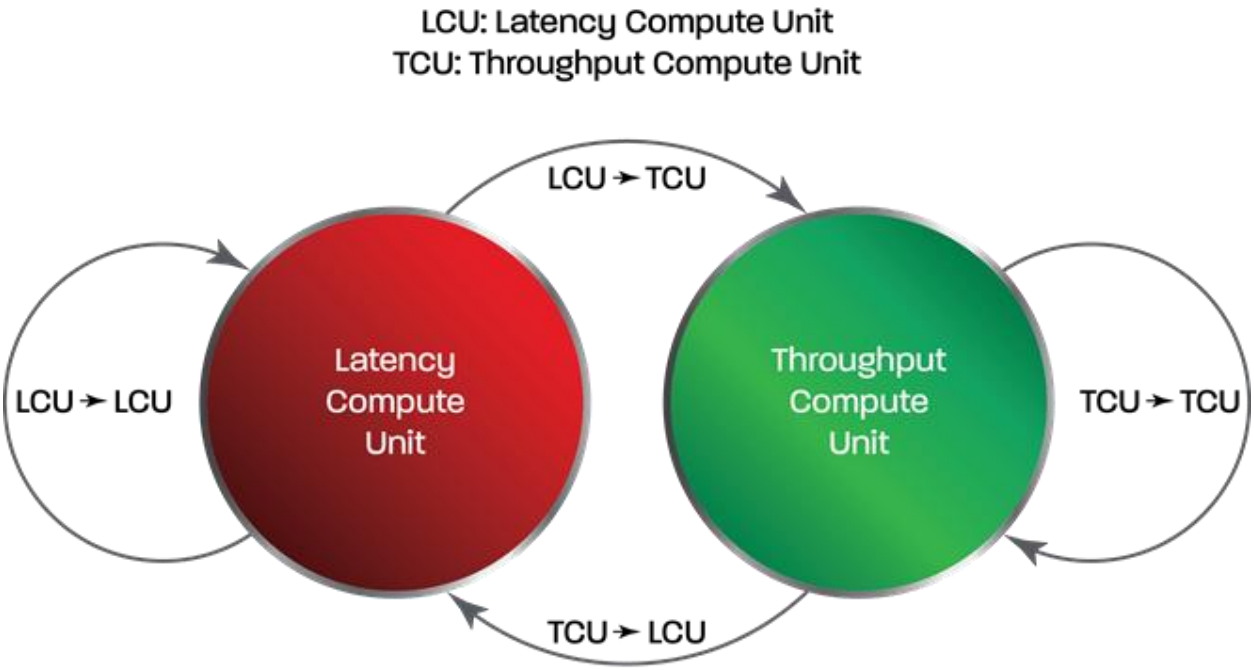

Figure 2.1 Compute Unit Queuing

2.4. Preemption and Context Switching

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

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

HSA supports job preemption, 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, as a traditional multi-user OS exposes the underlying hardware to the user.

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

2.5. 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 cooperating teams called *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 2.2 shows an NDRange, a work-group, and a work-item.

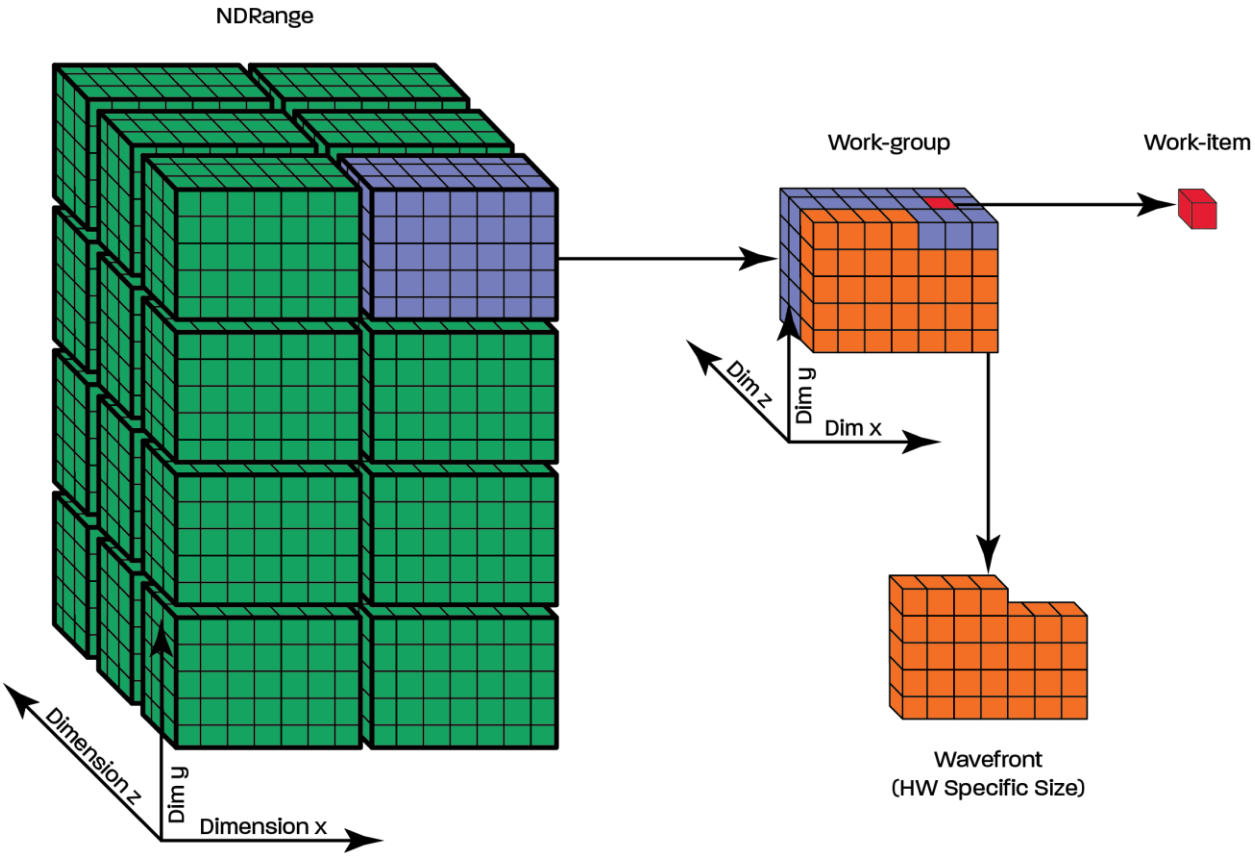

Figure 2.2 NDRange, work-group and work-item

The 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. Because of that, the underlying hardware architecture does not have to strictly adhere to the HSA Virtual Machine. The HSA Virtual Machine can also be implemented on an LCU, by having the finalizer convert HSAIL to the LCU's ISA.

Each HSAIL program has its own set of registers. There are four kinds of registers: single-bit control registers, and 32-bit, 64-bit and 128-bit registers that can be treated as integer or as floating point.

HSAIL instructions have a simple 3-address RISC-like format. All arithmetic operations are either register-to-register or constant-to-register.

The finalizer converts this load-store type architecture to the underlying processor ISA. The HSA load-store model is not imposed on the underlying processor.

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, and 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 to stream data into and out of the TCU.

3. Memory Model

3.1. Overview

A key architectural feature of HSA is its unified memory model. In the HSA memory model, a combined latency/throughput application uses a single unified virtual address space. All HSA-accessible memory regions are mapped into a single virtual address space to achieve Shared Virtual Memory (SVM) semantics.

Memory regions shared between the LCU and TCU are coherent. This simplifies programming by eliminating the need for explicit cache coherency management primitives, and it also enables finer-grained offload and more efficient producer/consumer interaction. The major benefit from coherent memory comes from eliminating explicit data movement and eliminating the need for explicit heavyweight synchronization (flushing or cache invalidation). The support of existing programming models that already use flushing and cache invalidation can also be supported, if needed.

3.2. Virtual Address Space

Not all memory regions need to be accessible by all compute units. For example:

- TCU work-item or work-group private memory need not be accessible to the LCUs. In fact, each work-item or work-group has its own copy of private memory, all visible in the same virtual address space. Private memory accesses from different work-items through the same pointer result in accesses to different memory by each work-item; each work-item accesses its own copy of private memory. This is similar to thread-local storage in CPU multi-threaded applications. Access to work-item or work-group memory directly by address from another accessor is not supported in HSA.

- LCU OS kernel memory should not be accessible to the TCUs. The OS kernel must have ownership of its own private data (process control blocks, scheduling, memory management), so it is to be expected that TCUs should not have access to this memory. The OS kernel, however, may expose specific regions of memory to the TCUs, as needed.

When a compute unit dereferences an inaccessible memory location, HSA requires the compute unit to generate a protection fault. HSA supports full 64-bit virtual addresses, but currently physical addresses are limited to 48 bits, which is consistent with modern 64-bit CPU architectures.

3.2.1. Virtual Memory Regions

HSA abstracts memory into the following virtual memory regions. All regions support atomic and unaligned accesses.

- **Global**: accessible by all work-items and work-groups in all LCUs and TCUs. Global memory embodies the main advantage of the HSA unified memory model: it provides data sharing between LCUs and TCUs.
- **Group**: accessible to all work-items in a work-group.
- **Private**: accessible to a single work-item.
- **Kernarg**: read-only memory used to pass arguments into a compute kernel.
- **Readonly**: global read-only memory.
- **Spill**: used for load and store register spills. This segment provides hints to the finalizer to allow it to generate better code.
- **Arg**: read-write memory used to pass arguments into and out of functions.

3.3. Memory Consistency and Synchronization

3.3.1. Latency Compute Unit Consistency

LCU consistency is being dictated by the host processor architecture. Different processor architectures may have different memory consistency models, and it is not the scope of HSA to define these models. HSA needs to operate, however, within the constraints of those models.

3.3.2.Work-item Load/Store Consistency

Memory operations within a single work-item to the same address are fully consistent and ordered. As a consequence, a load executed after a store by the same work-item will never receive stale data, so no fence operations are needed for single work-item consistency. Memory operations (loads / stores) at different addresses, however, could be re-ordered by the implementation.

3.3.3. Memory Consistency across Multiple Work-Items

The consistency model across work-items in the same work-group, or work-items across work-groups, follows a “relaxed consistency model”: from the viewpoint of the threads running on different compute units, memory operations can be reordered.

- Loads can be reordered after loads.
- Loads can be reordered after stores.
- Stores can be reordered after stores.
- Stores can be reordered after loads.
- Atomics can be reordered with loads.
- Atomics can be reordered with stores.

This relaxed consistency model allows better performance. In cases where a stricter consistency model is required, explicit fence operations or the use of the special load acquire (ld_acq) and store release (st_rel) is needed.

4.System Components

As described in the introduction, the basic components of an HSA system are:

- Compliant heterogeneous computing hardware.

- A software compilation stack.
- A user-space runtime system.
- Kernel-space system components.

Hardware components have already been discussed.

The compilation stack performs the work required to build an executable. HSA takes a liberal view of "executable", in the sense that an executable may reside as an intermediate form in memory, and need not be stored to disk. OpenCL™ drivers follow that model, as they support compilation of compute kernels during execution.

The runtime system contains the user mode management software required to execute a compiled HSA program. It is responsible for submitting commands to queues, abstracting device functionality, The runtime system also exposes HSA features to developers of applications, libraries and programming tools.

Software system components are responsible for resource allocation and management, job execution and scheduling, and in general any operations that need a more global system view. Time-critical operations, such as job submission, do not require a kernel call, and therefore are not the responsibility of the kernel. System components are described in the next section.

4.1. Compilation Stack

The HSA compilation stack converts a high-level language into HSAIL. The compilation stack is based on the LLVM compiler infrastructure (see llvm.org), which uses a plug-and-play environment. Figure 4.1 shows an outline of the stack.

The compiler stack consists of two main components, a front end and a back end, linked together through the LLVM intermediate representation (LLVM IR). The front end converts high-level languages into LLVM IR, then the back end converts LLVM IR into HSAIL. At the LLVM IR level, the program structure is data parallel. Suitable optimizations are also performed at that level. The compiler front end is responsible for extracting the data parallel section of the program, as needed.

For OpenCL, the compiler stack is part of the OpenCL Runtime, and is called during the execution of an OpenCL program. Both the input and the output of the compiler are already in data-parallel form.

For C++AMP (C++ Accelerated Massive Parallelism), the compiler stack is called during program compilation. The C++AMP compiler extracts the data parallel sections of the source program and passes them through the HSA compiler stack, while passing non-data-parallel sections through the traditional path.

Additional front-ends might utilize either of the above approaches, or alternatively might use a novel approach as dictated by the input language.

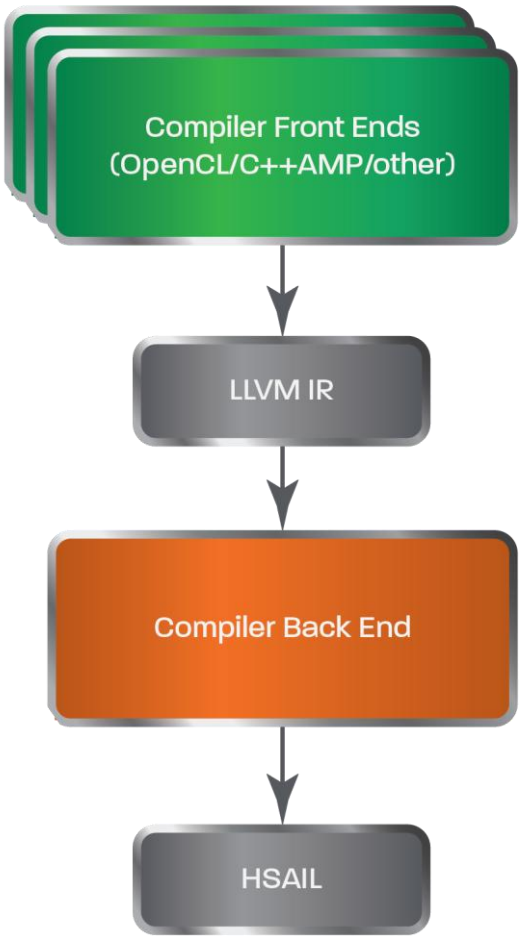

Figure 4.1 HSA Compilation Stack

4.2. Runtime Stack

The HSA runtime executes an HSA program. The runtime does not exist by itself, but rather works in tandem with the environment in which it runs, as shown in Figure 4.2.

For example, in OpenCL, the HSA runtime is called from the OpenCL runtime. In C++AMP, the HSA runtime is called from the C++AMP runtime. For any additional language bindings, the appropriate language runtime makes the HSA runtime calls. The HSA runtime API is public, so it is also possible for the application itself to call the runtime directly, if needed.

The HSA runtime dispatches work to the HSA device hardware. During dispatch, the finalizer converts HSAIL into the underlying ISA of the device. It may not be necessary to call the finalizer for a CPU device; in other words, the LLVM back-end for a CPU device may generate CPU-specific code directly, without intermediate HSAIL.

An HSA GPU device uses the kernel GPU driver (explained in detail below) to allocate and manages resources needed for the dispatch of compute kernels. For a CPU Device, OS system services take the role of the driver.

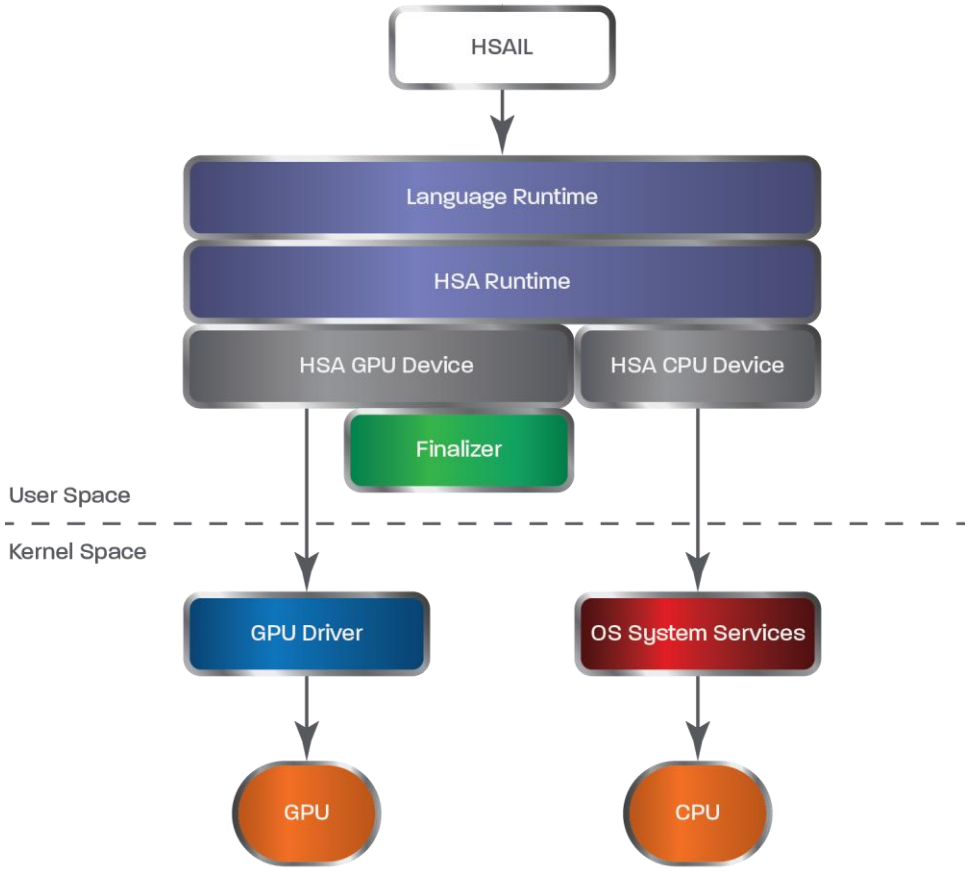

Figure 4.2 HSA Runtime Stack

4.2.1. Object File Format Extensions

The HSA application binary interface (ABI) is based on the ELF object format. HSA-specific ELF sections contain all HSA-related information.

HSA-aware compilers are required to generate native CPU code for an entire application, including compute kernels intended to run on TCUs. The generated native CPU code must be fully compliant with the existing native ABI and object file format on the respective OS platform. This ensures that an HSA executable can always run on any system, even on systems without HSA support.

An HSA-unaware toolchain will ignore the HSA-specific sections, and therefore will load and execute an HSA program using only native code on the LCUs. An HSA-aware toolchain will recognize the HSA-specific sections and dispatch compute kernels to TCUs.

4.2.2. Function Invocations and Calling Conventions

An HSA program can contain LCU functions and TCU functions. A function call therefore can fall into one of the following categories:

- **LCU function calls LCU function**. This function call uses the standard calling convention for the CPU ISA.

- **TCU function calls TCU function**. HSA defines a calling convention in HSAIL. The finalizer can implement this calling convention using whatever is most appropriate for the underlying hardware (for example, putting function parameters in registers vs. memory). The same finalizer is called for all TCU functions in an executable, which ensures calling convention consistency.

- **LCU function calls TCU function, or vice versa**. HSA uses queues to provide an asynchronous architectural model between heterogeneous caller/callee boundaries. A standard synchronous calling convention model, similar to remote procedure calls, can be implemented on top of the existing asynchronous model.

4.2.3. C++ Compatibility

HSA provides access to both LCU and TCU compute resources. It can execute any application written in a high-level programming language for traditional LCUs.

In addition to the general computation capability in the LCU, the TCU in HSA also has strong support for C++ features. It supports the handling of thread divergence incurred by control flow transfers, including indirect and virtual function calls. When some or all divergent flow paths for work-items converge, execution may be re-converged to improve performance. HSA also handles C++ exceptions, recursion, and dynamic memory allocation.

The C++ programming language can be extended to specify tasks for different compute units. Alternatively, the compiler can automatically partition C++ programs to allow single-source programs with tasks that run concurrently on both LCUs and TCUs.

Function invocations across the LCU and TCU boundary are supported, but rather than providing a traditional call interface, HSA provides a method of dispatching calls across the boundary using queues. Due to the lack of continuous stack boundaries across the dispatch, exceptions require special handling.

4.3. System (Kernel) Software

The basic HSA system software components are:

- **Kernel mode driver**: manages HSA hardware resources and graphics interoperability.

- **HMMU device driver**: manages the unified addressing model.

- **Scheduler**: manages TCU thread execution.

- **Memory Manager**: manages TCU memory regions.

Each component is described below.

4.3.1. Kernel Mode Driver

The HSA kernel mode driver supports numerous HSA functions, including:

- Registration of HSA compute applications and runtimes.
- Management of HSA resources and memory regions.
- Creation and management of TCU process control blocks (PCBs).
- Scheduling and context switching of TCUs.
- Graphics interoperability.

4.3.2. HMMU Device Driver

The HMMU device driver supports the unified memory model and manages the HMMU hardware. The HMMU device driver provides a variety of memory management related functions:

- Initialization of HMMU hardware and OS memory manager interfaces.
- Processing of page validation requests from the OS memory manager for HSA-related pages.
- Handling TCU HSA-specific page faults.
- Providing support interface for the HSA kernel-mode driver.

The above functions, when used together, can be used for capability management and load balancing for power and performance.

4.3.3. Scheduler

The scheduler manages scheduling and context switching of TCU jobs. Scheduling in HSA can be performed in software, in hardware, or in a combination of both An HSA implementation can choose how to split scheduling work between software and hardware to best match system requirements.

4.3.4. Memory Manager

The OS memory manager is a critical system component that requires modifications to support HSA. HSA needs exposed interfaces to support HSA memory management-based events. The areas of support in the OS include:

- Propagating HSA-related page table updates to HMMU for synchronization.
- Servicing TCU page faults from the HMMU device driver.
- Performing page invalidations and associated TLB operations across memory domains.
- Propagating process termination for resource cleanup.

4.4. Workload Data Flow

The HSA approach to workload data flow reduces kernel mode transitions by allowing a direct connection (using user-space queues) between the TCUs and the user mode application. This is in contrast to the legacy GPU model, which relies on copying the workload, patching command buffers, and numerous transitions between user mode and kernel mode.

This enables a high performance implementation of common application data flows, such as producer/consumer operations, and others that can benefit from SVM semantics by not copying data between LCUs and TCUs.

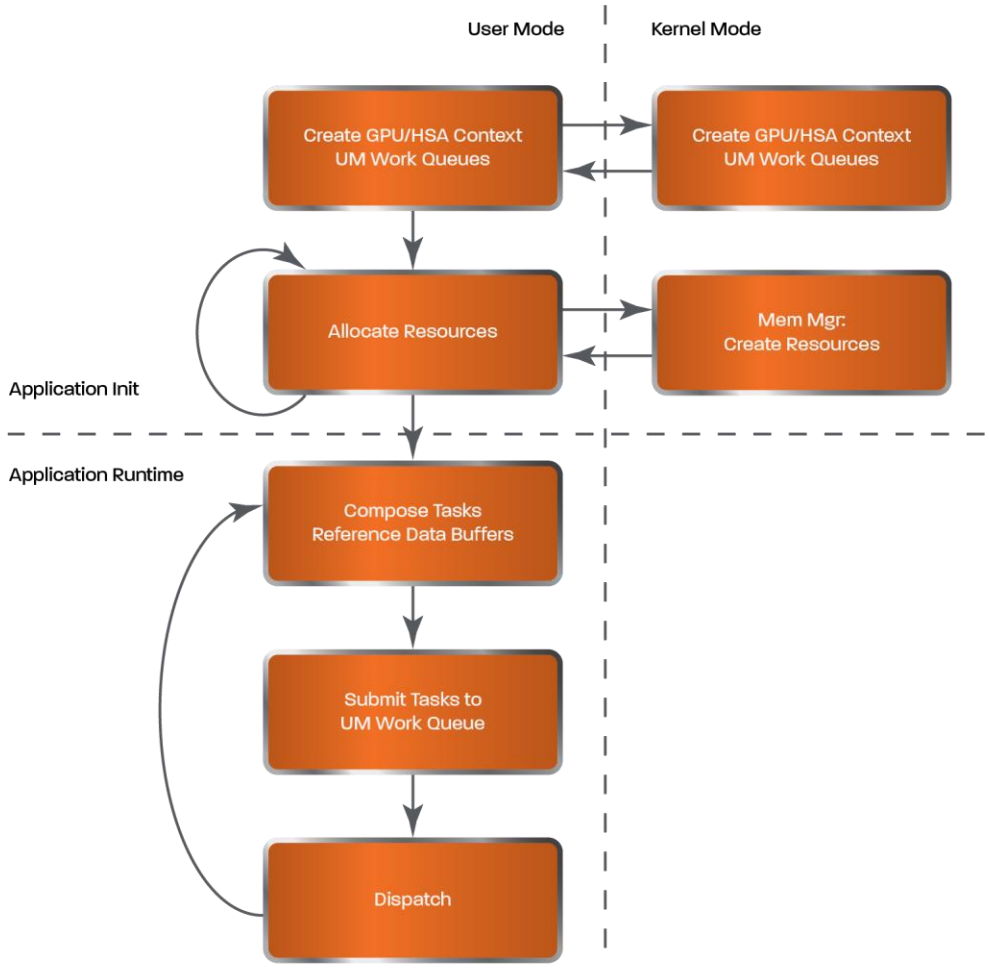

Figure 5.1 HSA Data Flow

4.5. Device Discovery and Topology Reporting

Device discovery is a necessary part of HSA. An HSA system uses various mechanisms to report HSA capabilities and features for both integrated and discrete HSA platform components. Different mechanisms can be used and extended for specific scenarios during initialization and runtime on an HSA-compliant platform.

In particular, to enumerate the platform topology, the HSA system software uses information provided by the OS and the platform firmware (using UEFI and/or ACPI). All this information is used to discover and characterize the complete system properties and report them in a consistent but flexible way to the application and language runtimes, allowing the software to scale and take advantage of all the HSA features of the platform.

Components requiring recognition during HSA discovery include:

- LCU cores and properties. This may include number of cores, number of cache levels, cache topology, TLB, FPU characteristics, power states, etc.

- TCU cores and properties. This includes compute unit size and arrangement, local data store properties, work queue properties, HMMU details, etc. An important characteristic is physical location (discrete vs. integrated).

- Support components. This includes PCI-E switches, memory bank properties, memory coherency properties, unified cache existence and properties, etc.

The configuration information gathered during device discovery affects decisions made during HSA system operation. Performance and power tuning based on this information helps HSA use available hardware to its maximum.

4.6. Memory Objects

Memory objects related to HSA design are defined within both user space and kernel space. These memory objects may be accessed by the HSA compute application, kernel mode driver compute component, and by the TCU. Memory objects can be:

- User-space buffers, like compute kernel program and data, and command queues.

- Kernel-space buffers, like job control blocks, user application contexts, scheduler run-list entries, and other system managed resources.

A user process requests memory objects through the HSA runtime. The requests get serviced by the kernel mode component, which also manages the kernel-space buffers. Allocated buffers can be handed directly to HSA hardware, without additional intervention from system software.

4.7. Interoperation with Graphics Stacks

An HSA application can access the same physical memory as existing graphics APIs without requiring a memory copy. This allows TCU devices to use graphics buffers directly, and conversely allows graphics to display results computed on TCUs directly, with no CPU intervention.

The HSA architecture does not explicitly specify how this is done. This allows HSA to interoperate with legacy graphics software and hardware, but also allows the flexibility for more streamlined future hardware designs.

5.Summary

The current state of the art of GPU 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 GPUs into a single system with common computing concepts, allowing the developer to solve a greater variety of complex problems more easily.