Challenges for Graphics and Heterogeneous Architectures: Applications and Technology

Timour Paltashev, Graphics IP Engineering, Advanced Micro Devices, Sunnyvale, California, U.S.A.
{timour.paltashev}@amd.com; {timpal}@mail.npu.edu
Ilya Perminov, National Research University ITMO, St. Petersburg, Russian Federation
{Ilya.Perminov}@amd.com; {studentikispam}@gmail.com

Abstract

The STAR on first part presents comprehensive overview of popular applications demands and their mapping to graphics and computing architectures of recent Graphics Processing Units (GPU) and Accelerated Processing Units (APU). Heterogeneous System Architecture (HSA)progress and merge of multicore CPUs with GPU cores in AMD product line is analyzed from potential user point of view. Semiconductor technology progress and power reduction challenges with their influence to graphics and compute architecture evolution have been considered as well.

On the second part we present HSA architecture principles comprehensive overview and demonstrate the example of ASTC texture compression algorithm mapping to modern GPU/APU architecture and OpenCL-HSA software stack. The performance measurements show significant improvement with applied algorithms adjustments and modifications.

Keywords: *GPU, CPU, APU, graphics architecture, heterogeneous architecture, compute architecture, texture compression, ASTC.*

1. KEY INDUSTRY CHALLENGES FOR GRAPHICS ARCHITECTURE IN 2013-2017

Graphics and computing architecture progress has several inflection points based on different industry branches, media content creation and massive entertainment industry merging with communication and computing domains. We may briefly define programming platforms and application programming interfaces (API) with direct influence to graphics and compute capabilities of modern hardware. Implementation of certain requested functionality hardly depends on semiconductor industry technology progress as well as on general computation technology advances. Power budget reduction for the same application execution is considered to be one of critical features of all new designs in all range from handheld mobile computing to supercomputing in data centers. Extremely high cost of semiconductor manufacturing in small size nodes 14nm and below requires new approaches on system architecture using multichip configurations.

Very important influence also comes from independent software vendors (ISV) developing game engines and visual computing applications. Game and movie content creators always enquire for new visual effects of processing capabilities implemented in both software and hardware levels. Below is listed brief overview of platforms and technology development.

Platforms and APIs:

- OpenGL ES 3.0, OpenGL 4.4, Mantle (AMD), OpenGL (common)
- Windows 8.1 with DirectX 11.2, Windows 2015 “Threshold” with DirectX 12 and SVM lite support, Windows 2017 with DirectX 13 and full SVM support,
- OpenCL 2.0 (2014-15), OpenCL 2.1 (2015-16) and OpenCL 3.0 (2016-17)

Technology major trends:

- Interposer technology including advanced semiconductor and systems packaging with interposing on silicon (organic and glass as well)
- Using HBM or High Bandwidth Memory in GPU, CPU and APU
- Sequential transitions to 20 nm (2014), 14 nm (2016) and 10 nm (2017+) semiconductor manufacturing processes on foundries
- Virtual page migration (2014-15), Low power HSA-based DSP (2015)
- Chiplets (tiny chips) combined on multichip module (MCM)

Major CPU/GPU and SoC vendors develop their product roadmap responding to challenges in platforms and technology:

AMD response on product line (public info limited by 2015):

- Discrete GPUs: Bonaire and Hainan (28nm/2013), Hawaii (28nm/2013-2014), Tonga and Iceland(28nm/2014), Bermuda and Fiji (28nm/2015)
- APUs: Kabini (28nm/2013), Kaveri and Mullins(28nm/2014), Carrizo (28nm/2015) and Amur/Nolan (20nm/2015).

Intel’s response on product line (public info limited by 2016):

- CPU-GPU SoC: Haswell and Silvermont(22nm/2013), Broadwell and Airmont (14nm/2014), Braswell and Goldmont (14nm/2015), Cannonlake (10nm/2016).

Nvidia’s response on product line:

- DGPU Kepler II(GK11x) (28nm/2013), Maxwell (28nm/2014) and project Denver with custom ARM 64-bit core (28nm/2014)
- Mobile SoC and application processors: Logan (28nm/2013), Tegra K1 (28nm/2014) Tegra M1 (20nm/2015).

2. SOFTWARE VENDORS VISIBLE CHALLENGES

Graphics ISVs traditionally have their own set of requests and challenges which may enable new applications. We may consider following list which can be complemented any time by new ideas:

1. Virtual Reality Holographic Rendering for head-mounted displays (HMD).
2. Global illumination rendering in real time.
3. Decoupled shading to process highly detailed scenes.
4. Object+texture space combined memory hierarchy.

VR for HMD generates a lot of attention, like product of Occulus Rift and Valve startup companies. They have significantly higher requirements for processing speed due to zero latency tolerance problem. Head movement demands smooth andsoftimage update, visible to eye and not causing movement artifacts. It requires high refresh rates and high resolution stereo image generation comparing to existing game consoles. In addition it requires image warp and post rendering to account for simulated lens optics. Next few years will be spent to find optimized solutions for HMD VR image generation. It may require significant computational power growth for GPU considering high resolution frame rate doubled or tripled versus latest game consoles.

Global illumination is one of favorite applications applied by researchers to GPU since they become more programmable in mid-2000s. Traditional local illumination shading and texturing algorithms implementation in popular game titles have been used as architecture optimization anchor while leaving serious capabilities for general computing which support global illumination models. We may group target applications and with their influence to architecture specifications.

First group with dense stream compute pattern (more suitable for GPU cores):

- 3D graphics in games and engineering
- High performance libraries for compute problems suitable for GPU acceleration

Second group with sparse compute pattern (more suitable for CPU/ latency optimizing system)

- Compiled OpenCL/C++ code for sparse problems
- Ray tracers for global illumination
- Other *Khronos*group platforms and based applications

Third groupwith specialsignal and image processing pattern (more suitable for DSP cores + fixed function blocks)

- Media processing

Such application groups create different vectors on architecture trends and are challenging designers to create computing machines which may fulfill several requirements. Some of them may look quite opposite and generate several issues on architecture optimization. Modern complex Systems-on-Chip (SoC) with different types of processing cores could be potential platforms. But simply putting together on the piece of silicon multiple cores does not solve the problem of programmability; even it makes the problem worse. New architecture concept of Heterogeneous System Architecture (HSA) could solve potential problem of creating multipurpose and power/cost efficient computing machines.

3. INTRODUCTION TO HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)

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.

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

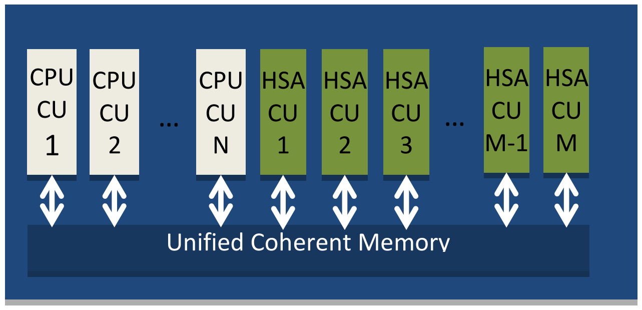

Figure 1: **Generic HSA Accelerated Processing Unit (APU)**

4. HSA OVERVIEW

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 GPU 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 CPU-type CU and one for the GPU/DSP type CU) 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 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.

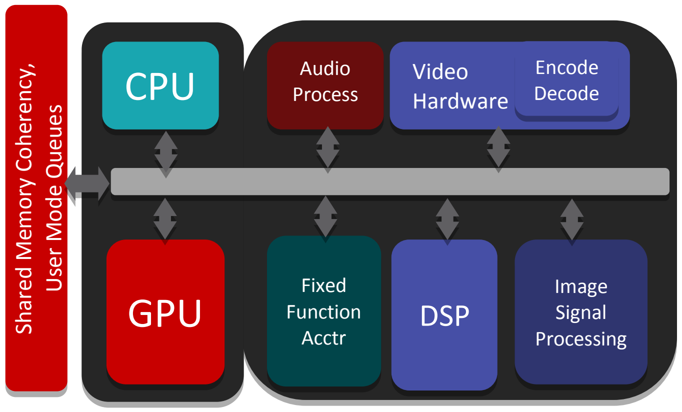

Figure 2: **HSA architecture example platform.**

5. 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.

6. ASTC OVERVIEW

As an example, we have modified ASTC compression algorithm to utilize HSA features. ASTC is a modern texture compression format developed by ARM and AMD. As the other texture compression formats, it aims on reducing requirements to both memory size and bandwidth.In such case,textures are stored in memory and transferred to GPU in a compressed form. Unpacking only occurs inside GPU, usually between L1$ and L2$ caches. Such approach also reduces power consumption, because overall GPU↔VRAM traffic could be directly converted to power consumption. This is especially important for mobile devices such as notebooks, tablet PCs and smartphones.

Texture access pattern is highly random and texture access time is a critical factor affecting the overall performance of the graphics subsystem. Because of it, most of the texture compression schemes provide fixed rate compression and decoders are implemented in hardware. This, in turn, obviously means lossy compression. Therefore, almost all known texture compressors are block based: an image is divided into blocks of a small size and each block gets compressed and accessed independently.

The ASTC format offers an unusual degree of flexibility and supports both 2D and 3D textures, at both standard (LDR) and high dynamic range (HDR), while providing better image quality than most formats in common use today. It also provides a rich set of compression bit rates from eight bits per pixel down to less than one bit per pixel in very fine steps. ASTC has fixed block size of 128 bits and supports 2D tiles from 4x4 to 12x12 pixels.

The very basic idea is as following: interpolation weights and up to four color endpoint pairs are stored in a compressed block, the decoder picks one color pair for every pixel and blends colors using interpolation weight to produce output color.Color and weight data could be encoded using various modes. Moreover, block layout and special bounded integer sequence encoding (BISE) allows flexible allocation of bits between different types of information. Still, decoding is rather efficient and fast.

However,that is not true for encoding. Achieving decent quality at a reasonable speed is a non-trivial task, because of format complexity. Currently there is only one ASTC encoder – ASTC Evaluation Codec which is a part of Mali Texture Compression Tool written by ARM. Our implementation is based on this codec.

Base algorithm tries to find the best colors and weights encoding for every possible block mode. It stops searching once a compression error for current block gets lower than the error limit. For the performance reasons, heuristics and early exit conditions are heavily used. There are predefined speed/quality settings named veryfast, fast, medium, thorough and exhaustive, which limits search space. Compression times for the test texture[1] are shown in Table 1.

Quality Settings	Peak signal-to-noise ratio (PSNR), dB	Compression time, seconds
Veryfast	41.931500	0.8
Fast	44.712035	1.9
Medium	45.716011	12.2
Thorough	46.072663	47.1
Exhaustive	46.203190	109.3

Table 1: **Compression times for different speed/quality settings (Mali TCT 4.2, APU – AMD A10-7850K[2])**

The thorough and exhaustive settings are of a particular interest, because high quality modes are most demanded in 3D content development.

7. MAPPING ASTC TO HSA

As well as other block compression schemes;the ASTC could be easily parallelized on a block level (which in fact is done in ASTC Evaluation Codec). However, moving entire algorithm to a GPU would be very inefficient, because of high threads divergence caused by early exit condition heuristics and branches. Nevertheless,some computation steps can be performed in parallel.

So compression of a single block is consists of a sequence of stages, where some stages could be effectively implemented on GPU cores. Schematically it is shown on Fig. 3, where boxes represent parallel steps and circles represent sequential steps of compression of a single block.

[1] Sample texture turret_diffuse_map.png (512x512 pixels) from the Mali TCT 4.2 was [illegible] all tests.

[2] AMD APU A[illegible]7850K -- 4 [illegible]ores @3.7[illegible]z, 8 GPU cores @720[illegible]

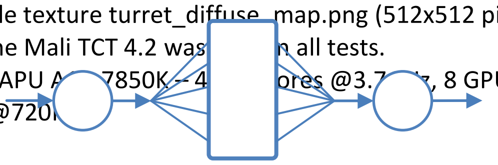

Figure 3: **Compression stages for a single image block**

By observing source code and profiling original codec, we have chosen three candidates for GPU offloading, which was implemented in OpenCL kernels:

- realign_weights()
- find_best_partitionings()
- compute_angular_endpoints()

Still, parallel parts of a single block compression process cannot create reasonable load level for a single GPU core. Therefore, image blocks could be compressed in batches: CPU thread executes sequential stages for a batch of blocks and prepares data for GPU. Schematically this approach is depicted on Fig. 4.

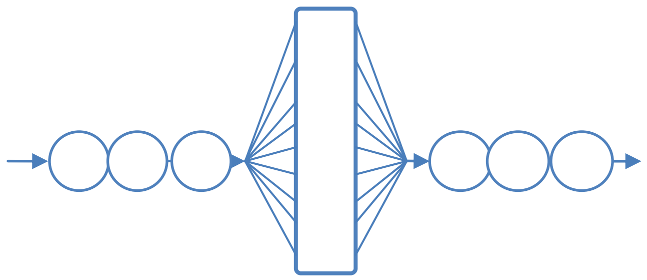

Figure 4: **Compression stages for a batch compression**

Our experiments show that a batch size of 512-1024 blocks provides reasonable tradeoff between memory consumption and better GPU utilization.

Offloading some work to GPU cores could increase overall performance in two ways:

- Executing parallel stages on GPU core is often more efficient in terms of time and power.
- CPU threads could process another batch while waiting results from GPU.

However, traditional GPGPU approach with discrete CPU and GPU devices faces following restrictions:

- Data should be transferred between CPU and GPU over PCIe, which has much lower bandwidth than RAM or video framebuffer. Coping time for small tasks is comparable with execution time. Sparse access to large buffers is also causes difficulties. It is possible to directly access such buffers from GPU over PCIe without copying data. However, all that host memory should be prepinned, even if many pages will never be used. It causes OS overhead and may lead to unnecessary paging activity.
- High number of kernel invocations results in high driver overhead.

In contrast, HSA platform lacks such restrictions by providing features such as hardware scheduling, user-level command queuing and coherent shared virtual memory. The last one is also greatly simplifies acceleration of existing applications.

Another approach we have used to accelerate ASTC encoding is JIT (just in time) compilation. Some compression parameters, such as tile size and searching limits, are constant for a chosen image and quality settings. As OpenCL kernels are naturally compiled at a runtime, we are able to replace such variables with macro definitions and pass actual values at runtime. This allows OpenCL compiler to make additional optimizations, reduce binary size and register pressure. It also helps increasing kernel occupancy level, which in turns allows better hiding memory latency. Results for one of the implemented kernels are shown in Table 2.

	Static compilation	JIT compilation
Binary size	24124 bytes	7628 bytes
Scalar GPRs	76	50
Vector GPRs	65	36

Table 2: **Comparing static and JIT compilation**

8. RESULTS AND FUTURE WORK

As a proof of concept we have implemented HSA accelerated ASTC encoding for LDR images without alpha channel. Currently, HSA software stack remains in development state, so some features are not yet available or optimized. Still the results (Table 3) are rather promising: HSA provides up to 5x speedup.

Quality Settings	Compression time in seconds		Speedup
	Original codec	HSA accelerated codec	
Medium	12.2	3.4	3.59x
Thorough	47.1	10.6	4.44x
Exhaustive	109.3	21.3	5.13x

Table 3: **Comparing compression times for original and modified codecs**

Note that original codec (Mali TCT 4.2) goes with 32bit binary. Our implementation was compiled for x64 target and also benefits from larger register file and SIMD instructions.

Currently CPU threads just wait while GPU executes kernels. It results in overall CPU utilization of 60-90%. Therefore, there is a lot of room for increasing performance even further by implementing dynamic load balancing between CPU and GPU cores. HSA profiling and instrumentation tool progress may give a chance to useheterogeneous cores more efficiently providing better load balance between GPU and CPU cores.

9. 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.

HSAis 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.There is a long way ahead on migration of classic sequential programming algorithms and tasks to HSA platforms to provide power/cost efficient computing in several domains of human activity.

10. REFERENCES

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

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

[3] http://malideveloper.arm.com/develop-for-mali/tools/astc-evaluation-codec/

[4] Paltashev T.T., Perminov I.V., Texture compression techniques, Scientific Visualization, National Research Nuclear University "MEPhI". - 2014. - Т. 6, вып. 2014-1. - С. 96-136. - ISSN 2079-3537

About the authors

TimourPaltashev is Senior Manager of Graphics IP Engineering at Advanced Micro Devices and Professor at Northwestern Polytechnic University, College of Engineering. His contact email istimpal(at)mail.npu.edu.

IlyaPerminov is a PhD candidate at St. Petersburg National Research University of Information Technologies, Mechanics and Optics. His contact email is studentikispam(at)gmail.com