International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2018 IJSRCSEIT | Volume 3 | Issue 3 | ISSN : 2456-3307

Heterogeneous System Architecture (HSA)

Agbaje Michael .O,Bammeke Adekunle, Ohwo Onome Blaise
Babcock University, Nigeria

ABSTRACT

The measure of computerized information being created and put away is expanding at a disturbing rate. This information is classified and handled to distil and convey data to clients crossing various businesses for example, finance, online networking, gaming and so forth. This class of workloads is alluded to as throughput computing applications. Multi-core CPUs have been viewed as reasonable for handling information in such workloads. Be that as it may, energized by high computational throughput and energy proficiency, there has been a fast reception of Graphics Processing Units (GPUs) as computing engines lately. GPU computing has risen lately as a reasonable execution stage for throughput situated applications or regions of code. GPUs began as free units for program execution however there are clear patterns towards tight-sew CPU-GPU integration. In this paper, we look to comprehend cutting edge Heterogeneous System Architecture and inspect a few key segments that influences it to emerge from other architecture designs by analyzing existing inquiries about, articles and reports bearing and future open doors for HSA systems.

Keywords: heterogeneous system architecture, latency compute unit, throughput compute unit, CPU, GPU.

I. INTRODUCTION

Throughout the years, computer system architecture following the Moore's law has advanced from the single-core era to the multi-core era. Presently with the rise of Heterogeneous System Architecture (HSA), computer system architecture has brought together perspective of fundamental computing components enabling developers to compose applications that flawlessly coordinate Central Processing Units (CPUs) (called latency compute units) with Graphical Processing Units (GPUs) (called throughput compute units), while profiting from the best characteristics of each. As of late, GPUs have changed from the conventional unadulterated graphics accelerators to more broadly useful parallel processors that supports standard Application Programming Interface(API) and tools, for example, C++AMP, OpenCL™ and DirectCompute. (George Kyriazis, 2012)

These APIs however encouraging are as yet looked with many obstacles for the making of a domain that permits the GPU to be utilized as smoothly as the CPU for regular programming tasks: diverse memory spaces amongst CPU and GPU, non-virtualized hardware, et cetera. These obstacles are evacuated by HSA, in this way enabling software engineers to exploit the parallel processor in the GPU as a co-processor to the conventional multithreaded CPU.

The objective of the HSA system is to make a solitary brought together programming platform giving a solid establishment to the advancement of languages, structures, and applications that adventure parallelism. All the more particularly, HSA's objectives include:

- ✓ Removing the CPU/GPU programmability hindrance.
- ✓ Reducing CPU/GPU communication latency.
- ✓ Opening the programming platform to a more extensive scope of utilizations by empowering existing programming models.

- ✓ Creating a reason for the incorporation of extra processing components past the CPU and GPU.

HSA enables exploitation of the abundant information parallelism in the computational workloads of today and without bounds in a power-proficient way. It likewise gives continued support to conventional programming models and computer architectures. (George Kyriazis, 2012)

We are at the point where we are unable to power all of the transistors we have on chip. Ways around this have generally involved heterogeneous architectures, where you have multiple distinct computation units that are optimized in terms of performance and power for specific tasks. Then, computation can be directed to those units.

The main objective of this paper is to do an overview and descriptive study of Heterogeneous System Architecture by reviewing articles and journals about HSA paradigm and also related works done in the improvement of HSA.

II. LITERATURE REVIEW

2.1History of HSA

Computers and other innovation initially started with single-core processors; in the early 2000s, the historical backdrop of computing was perpetually changed by pushing in multi-core processors. The single-core processors were attaining limits, and they could not physically enhance these present designs without revising the whole production process. This lead to designing the processors on a multi-core format. After some time, the multi-core processor advanced from dual core to tri, quad, hex and octa core designs. A few processors now hold many several cores.

The development additionally proceeds regarding innovative design refinement. The present processors with various cores are currently designed with multi-threading features, achievement improvements, for example, memory-on-chip, and heterogeneous core design intended for special purposes. We can ascribe these developments to a few rising needs patterns: on one hand, contemporary technologies must turn out to be increasingly productive in networking, multimedia processing, and device recognition. On the other hand, energy effectiveness must increase also.

Since the earliest reference point of integrated circuits advancement, processors were designed with consistently expanding clock frequencies and sophisticated in-build optimization strategies. As individual CPU gates grow smaller through manufacture breakthroughs, semiconductor microelectronics likewise turn out to be increasingly enhanced regarding physical properties.

Because of physical constraints, this speedup has arrived at an end. Physical imperatives set up various hindrances in accomplishing high performance computing within power budgets. Computer architectures are endeavoring to cross these performance restricting walls utilizing number of innovative processor architectures employing area concurrency (Paul, 2014).

The single chip graphics processor by around year 2000 incorporated practically everything about the conventional top of the line workstation graphics pipeline and thusly, merited another name past VGA controller. The term GPU was instituted to indicate that the graphics device had turned into a processor. After some time, GPUs turned out to be more programmable, as programmable processors supplanted fixed function dedicated logic while keeping up the fundamental 3D graphics pipeline organization. What's more, computations turned out to be more exact after some time, progressing from indexed arithmetic, to integer and fixed point, to single precision floating-point, and recently to double precision floating-point. GPUs have progressed toward becoming hugely parallel programmable processors with hundreds of cores and thousands of threads.

As of late, processor instructions and memory hardware were added to help general-purpose programming languages, and a programming environment was made to enable GPUs to be modified utilizing well-known languages, including C and C++. This advancement makes a GPU a fully general purpose, programmable, many core processor, though still with some special advantages and limitations (Nickolls & Kirk, 2012).

Multicore processors have just been prominent to enhance the total performance. Considering the power and temperature constraints, they may be the sole practical solution. A considerable measure of studies to decide the best multicore setup is conducted and it is believed that the heterogeneous multicore is the best in power and performance trade-off (Sato, Mori, Yano, & Hayashida, 2012).

2.2 Features of HSA

The HSA engineering manages two sorts of compute units:

- ✓ A latency compute unit (LCU) is a generalization of a CPU. A LCU bolsters both its local CPU instruction set and the HSA intermediate language (HSAIL) instruction set.
- ✓ A throughput compute unit (TCU) is a generalization of a GPU. A TCU underpins just the HSAIL instruction set. TCUs perform extremely proficient parallel execution.

A HSA application can keep running on an extensive variety of platforms comprising of both LCUs and TCUs. The HSA structure enables the application to execute at the most ideal performance and power point on a given platform, without yielding adaptability. In the meantime, HSA enhances programmability, portability and compatibility.

George Kyriazis (2012), featured some noticeable compositional highlights of HSA and it incorporates:

- **Shared page table support**: To disentangle operating system and user software, HSA permits a single set of page table entries to be shared amongst LCUs and TCUs. This enables units of the two types to get to memory through the same virtual address. In simplified terms, the operating system just needs to oversee one set of page tables; along these lines empowering Shared Virtual Memory (SVM) semantics amongst LCU and TCU.
- **Page faulting**: operating systems permit user processes to access more memory than is physically addressable by paging memory to and from disk. What's more, the operating system and driver needed to create and manage a different virtual address space for the TCU to utilize. HSA expels the burdens of pinned memory and separate virtual address management, by permitting compute units to page fault and to utilize the same large address space as the CPU.
- **User-level command queuing**: Time spent waiting for the operating system kernel service was often a major performance bottleneck in earlier throughput compute systems. HSA radically lessens the time to dispatch work to the TCU by enabling a dispatch queue for every application and by permitting user mode process to dispatch directly into those queues, requiring no operating system kernel service transition or services. This makes the full performance of the platform accessible to the developer, limiting software driver overheads.
- **Hardware scheduling**: HSA gives a mechanism whereby the TCU engine hardware can switch between application dispatch queues automatically, without requiring operating system intervention on each switch. The operating system scheduler can characterize each part of the switching sequence and still looks after control. Hardware scheduling is quicker and devours less power.
- **Coherent memory region**: HSA grasps a completely coherent shared memory model, with unified addressing. This provides developers with the same coherent memory model that they

appreciate on SMP CPU systems. This enables developers to compose applications that closely couple LCU and TCU codes in popular design patterns like producer-consumer.

HSA Support Libraries

The HSA platform is intended to support high-level parallel programming languages and models, including C++ AMP, C++, C#, CUDA, OpenCL, OpenMP, Java and Python, to give some examples. HSA-aware tools create program binaries that can execute on HSA-enabled systems supporting multiple instruction sets (commonly, one for the LCU and one for the TCU) and furthermore can run on existing architectures without HSA support (George Kyriazis, 2012).

Program binaries that can run on both LCUs and TCUs contain CPU ISA (Instruction Set Architecture) for the LCU and HSA Intermediate Language (HSAIL) for the TCU. A finalizer converts HSAIL to TCU ISA. The finalizer is regularly lightweight and might be run at install time, compiler time, or program execution time, contingent upon decisions made by the platform implementation.

HSA API Level

This provides insight into the current tools and APIs used in heterogeneous software development.

2.3 HSA INTERMEDIATE LANGUAGE (HSAIL)

HSAIL is a low level instruction set intended for parallel compute in a shared virtual memory environment. HSAIL is SIMT (Single-Instruction Multiple-Thread) in form and does not dictate hardware microarchitecture. It's intended for fast compile time, moving most optimization to HL compiler. And furthermore at an indistinguishable level as PTX: an intermediate assembly or Virtual Machine Target. It's represented as bit-code in a Brig file format which help late binding of libraries (Hedge, 2013).

HSAIL is the intermediate language for parallel compute in HSA

- Generated by a high level compiler (LLVM, gcc, Java VM, and so on)
- Compiled down to GPU ISA or other parallel processor ISA by an IHV Finalizer
- Finalizer may execute at run time, install time or build time, contingent upon platform type.

COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

At a high level, CUDA is a proprietary tool for execution of general purpose programs on NVIDIA graphics cards. To utilize it, you should have NVIDIA hardware and NVIDIA's compiler. CUDA is an adaptable parallel programming model and software platform for the GPU and other parallel processors that enables the software engineer to sidestep the graphics API and graphics interfaces of the GPU and basically program in C or C++. The CUDA programming model has a SPMD (single-program multiple data) software style, in which a software engineer composes a program for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU. Actually, CUDA additionally gives a facility to programming multiple CPU cores too, so CUDA is an environment for composing parallel programs for the whole heterogeneous computer system (Nickolls & Kirk, 2012).

OPEN COMPUTE LANGUAGE (OpenCL)

OpenCL is a recent and open heterogeneous programming standard bolstered by the Khronos Compute Working Group. OpenCL is an industry standard programming language for parallel computing, that gives a bound together programming model to CPUs, GPUs, Smart Phones, Tablets, Servers (Cloud). Enabling programming engineers to compose programs once and runs cross-platform. It is likewise upheld by all major hardware & software vendors (Nickolls & Kirk, 2012).

HSA additionally uncovered a few advantages for picking lower level programming interface for those that need a definitive control and performance. A portion of the advantages of utilizing OpenCL on HSA incorporates

- ✓ Avoidance of inefficient duplicates
- ✓ Low latency dispatch
- ✓ Improved memory model
- ✓ Pointers shared amongst CPU and GPU

CUDA and OpenCL are compared by studying their platform models, memory models, and execution models. (Grossman, 2013)

Platform Model

A platform model indicates how the hardware accessible to a software engineer on a specific system is exhibited, both conceptual and through the API. Both OpenCL and CUDA platform models depict discrete devices which are overseen through an API from a host program. These devices have separate address spaces from the host program and utilize explicit transfer to receive input and return output to the host program. OpenCL and CUDA give techniques to accessing metadata on every device in a platform, (for example, memory size, computational units accessible, and so forth).

At the highest granularity, an OpenCL installation can contain at least one or more platforms, each of which contains at least one or more devices. Inside each OpenCL device there are multiple compute units. Access to the platforms and devices in an OpenCL program is more express and verbose than in CUDA and requires the formation of contexts and command queues. An OpenCL context is a gathering of at least one or more OpenCL devices. OpenCL command queues are utilized to issue commands to devices, and each commands queue is unequivocally connected with a solitary OpenCL device. CUDA is by default less explicit than OpenCL, however regardless it bolsters a considerable amount of similar operations on a CUDA platform. Consistently, CUDA has a selected device which CUDA operations are verifiably issued to, however CUDA enables you to unequivocally set a currently active device. While there is a model of "streams" of work in CUDA like OpenCL's command queues, CUDA streams are not required to be expressly given by the developer to each device operation.

Execution Model

The execution model of a heterogeneous programming model portrays the conceptual model for the execution of user-written computation. Both OpenCL and CUDA are SIMD (Single-Instruction Multiple-Data) programming models. The software engineer composes a kernel for the device and expressly shows it is for device execution utilizing language keywords. Threads executing these kernels utilize a special thread ID to choose their data sources. Both OpenCL and CUDA utilize a batched kernel invocation model where large numbers of kernel instances or threads are launched in a single API call. Both CUDA and OpenCL group individual threads into little accumulations. Kernel invocation dispatch numerous thread collections without a moment's delay. One region in which OpenCL's and CUDA's execution model veer is setting up a user-written kernel for execution on a device. CUDA compiles kernels for execution at compile-time utilizing NVIDIA's compiler. OpenCL program and OpenCL kernel are compiled or loaded at runtime. An OpenCL program represents a collection of executable functions. An OpenCL kernel object is associated with a program object and determines a single entry point to that program. This makes OpenCL both a more adaptable and unequivocal programming model than CUDA with regards to executing computation on various sorts of devices. Then again, every OpenCL program must set up executable objects before executing them on a device whereas CUDA prepares them for the user implicitly.

Memory Model

Both CUDA and OpenCL utilize discrete address spaces to represent the memory available from a device, even in the situations where an OpenCL host

application is executing utilizing an indistinguishable memory as an OpenCL device (as is regularly the situation when multi-core CPUs are gotten to through OpenCL). All together for computation executing on a device to have access to input values from the host program, those values must have been previously and explicitly copied to global buffers related with that device. For the host application to get output values from device computation, those values must be duplicated out of global device buffers and into the host program's address space. Both CUDA and OpenCL give API calls to duplicate in and duplicate out of global memory, and also approaches to utilize special purpose memory, (for example, texture memory) which may enhance performance for the correct access patterns on certain hardware. The CUDA and OpenCL kernel languages additionally incorporate special keywords for determining local, scratchpad memory available from a kernel. This content of scratchpad memory has the same lifetime from a thread group on a compute unit and exhibits lower latency.

Limitations of HSA

The limitations/bottlenecks of HAS are considered temporary as developments are still ongoing. These limitations includes but not limited to (Hedge, 2013):

- ✓ Programmability
- ✓ Communication overhead

Related Works

Hsu, Chen, & Chen(2015) introduced a virtual platform conforming to the HSA programing model and the HSA Intermediate Language (HSAIL) specification. This platform has an advanced simulator demonstrating the cutting edge GPU microarchitecture intended for Single Instruction Multiple Data (SIMD) processing. The platform additionally gives a simulation framework, including OpenCL and OpenGL API, the driver for simulator, and compilation flow from OpenCL kernel and OpenGL shader program to HSAIL and lastly to a custom instruction set. This platform was considered with several benchmarks. OpenCL benchmarks are for the most part the AMD sample programs. OpenGL benchmarks are programs of classic shading algorithms. On this platform, the performance issue in various GPU microarchitecture, ISA configuration, task scheduling algorithms and SIMD control divergence handling mechanisms were broken down.

Arora, (2012) explored the engineering and advancement of general purpose CPU-GPU systems; which was begun by portraying cutting edge designs in GPGPU (General-Purpose Graphics Processing Unit). Considered answers for key GPGPU issues – performance loss due to control-flow divergence and poor scheduling. As an initial step, chip integration offers better performance. In any case, lessened latencies and increased bandwidth are enabling optimizations previously not possible. Comprehensive CPU-GPU system enhancement methods, for example, CPU core optimizations, redundancy elimination and the optimized design of shared components was depicted. Furthermore, opportunistic enhancements of the CPU-GPU system by means of collaborative execution was considered. Finally, recommended future work open doors for CPU-GPU systems.

Grossman (2013) exhibited four heterogeneous programming frameworks, each with the high-level objective of enhancing programmability of heterogeneous platforms while either keeping up or enhancing performance. This objective was accomplished by balancing architectural transparency with programming abstractions. Each of the programming models or runtime systems introduced, positions itself at an alternate point between accentuating programmability and performance. Apparently every one of them lie somewhere close to the low-level heterogeneous programming models (like CUDA and OpenCL) and high-level models (like OpenACC or CUDA libraries). By striking a superior harmony amongst abstraction and transparency, these programming models

empower software engineers to be constructive and create elite applications on heterogeneous platform. In any case, in spite of research effort in heterogeneous programming models, it is anything but difficult to contend that the issue of proficient advancement of reasonably complex applications on real-world, distributed, heterogeneous systems is to a great extent unsolved.

Power, et al., (2013) built up a Heterogeneous System Coherence (HSC) for CPU-GPU systems to relieve the coherence bandwidth impacts of GPU memory demands. HSC replaces a standard directory with a region directory and adds a region buffer to the L2 cache. These structures enable the system to move bandwidth from the coherence network to the high-bandwidth direct-access bus without sacrificing coherence. The outcomes were evaluated with a subset of Rodinia benchmarks and the AMD APP SDK and it demonstrated that HSC can enhance performance contrasted with a conventional directory protocol by an average of more than 2x and a maximum of more than 4.5x. Furthermore, HSC decreases the bandwidth to the directory by an average of 94% and over 99% for four of the broken down benchmarks

III. METHODOLOGY

Different works on HSA were inspected and different conceivable routes by which HSA can be enhanced were recognized and furthermore the current devices and procedures utilized. These related works evaluated, concentrated on improving on the bottlenecks of HSA.

Table 1

Bottlenecks	Procedures	Results
Programming model	Balancing architectural transparency with programming abstractions.	By striking a superior harmony, programmability of the heterogeneous system architecture can be achieved and maintained. Thus, enhancing performance.
	A virtual platform conforming to the HSA programing model and the HSA Intermediate Language (HSAIL) specification	Explore microarchitecture design and evaluate the performance issues for both the OpenCL and OpenGL applications.
Performance	Chip integration	Better performance
Communication overhead	Heterogeneous System Coherence replaces a standard directory with a region directory and adds a region buffer to the L2 cache.	Moves bandwidth from the coherence network to the high-bandwidth direct-access bus without sacrificing coherence.

These procedures can be utilized individual or combined in a number of ways to better mitigate the bottlenecks identified, improve and fast track its acceptance of Heterogeneous System Architecture in our day to day computations.

These systems however effective and in spite of research effort in heterogeneous system architecture, it is anything but difficult to contend that the issue of proficient advancement of realistically complex applications on real-world, distributed, heterogeneous systems is to a great extent unsolved.

IV. RESULT

The architectural path for the future is clear. An open design, with published specifications and an open source execution programming stack. Permitting programming designs set up on Symmetric Multi-Processor (SMP) systems relocate to the heterogeneous world. Furthermore, Heterogeneous cores cooperating consistently in coherent memory, permitting low latency dispatch and no software fault lines. This has brought about game-changing HSA, parallel compute adoption at tipping point, intense and developing programming ecosystem and winning the heart and psyches of developers.

V. CONCLUSION

Heterogeneity is an undeniably essential trend and the market is at last beginning to make and receive the important open benchmarks. HSA is a bound together computing framework. It gives a single address space available to both CPU and GPU (to avoid information replicating), user-space queuing (to limit communication overhead), and preemptive context switching (for better quality of service) over all computing elements in the system. HSA binds together CPUs and GPUs into a single system with common computing concepts, enabling the developer to solve a greater variety of complex issues all the more effortlessly. However, the current state of the art of GPU high-performance computing is not flexible enough for many of today's computational problems.

VI. REFERENCES

[1]. Arora, M. (2012). The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing. San Diego.

[2]. George Kyriazis, A. (2012). Heterogeneous System Architecture: A Technical Review.

[3]. Grossman, M. (2013). Programming Models and Runtimes for Heterogeneous Systems. Houston, Texas.

[4]. Hedge, M. (2013). Heterogeneous System Architecture and the Software Ecosystem.

[5]. Hsu, Y., Chen, H.-Y., & Chen, C.-H. (2015). A Heterogeneous System Architecture Conformed GPU platform supporting OpenCL and OpenGL.

[6]. Nickolls, J., & Kirk, D. (2009). Appendix A: Graphics and omputing GPUs.

[7]. Nickolls, J., & Kirk, D. (2012). Appendix A: Graphics and Computing GPUs.

[8]. Paul. (2014, August 24). The History of the Multi core processor. Retrieved from BurnWorld.com: www.burnworld.com/the-history-of-the-multi-core-processor/

[9]. Power, J., Basu, A., Gu, J., Puthoor, S., Beckmann, B. M., Hill, M. D.,.Wood, D. A. (2013). Heterogeneous System Coherence for Integrated CPU-GPU Systems.

[10]. Sato, T., Mori, H., Yano, R., & Hayashida, T. (2012). Importance of Single-Core Performance in the Multicore Era. Thirty-Fifth Australasian Computer Science Conference (ACSC 2012). Melbourne, Australia.