Evolution of Gpus, Moving Towards GPGPUS: A Survey

Shivam Bharadwaj
GLBITM, Gr. Noida, India

Tejas Upmanyu
GLBITM, Gr. Noida, India

Sandeep Saxena
GLBITM, Gr. Noida, India

ABSTRACT
Graphics Processing Units (GPUs) broke out by the end of 1990s, devoted to the goal of providing ubiquitous interactive 3D graphics which, a few years back then, was a far-fetched dream. By the end of decade, the technology grew exponentially, with nearly every computer containing a GPU, providing a high performance, visually rich, brilliant 3D computer graphics. This unprecedented growth was a cumulative outcome of rising demand for high-quality games, manufacturing processes advancements, and employment of inherent parallelism for computation.

Today, the raw computational power of a GPU dwarfs that of the most powerful CPU, and the gap is continuously widening. World's most powerful supercomputers (e.g., Tianhe-2(China) and Titan(USA)) use GPUs at their core [1]. This paper provides a review of GPU technology, overview of general purpose application of GPUs, architectural highlights, Enhancement of GPGPUs. In the end, we take a look at future research directions and challenges to parallel computing chips.

General Terms
Parallel computing, Compute Unified Device Architecture (CUDA), OpenCL,

Keywords
Graphics processing unit (GPU), central processing unit (CPU), graphics pipeline, parallel-computing

1. INTRODUCTION
Gordon Moore [2], predicted way back in 1970 that, the processing power of chips would double every two years and it still holds almost correct. Semiconductor scaling challenges, power efficiency and thermal points along with higher levels of intricacies involved in exploiting the power of instruction level parallelism (ILP) has motivated researchers, to go beyond single-core processors. This performance limitation has led microprocessor manufacturers to turn towards multi-core chips for a promising future. Benefits of multi-core organization of chips are numerous, but to tap on full power, it demands sophisticated implementation of parallelization via programming.

In this situation, Graphics Processing Units (GPUs), have attracted a lot of attention as they are very much resourceful not only for graphics workload, but also for mainstream computing tasks due to inherent parallelism, high memory bandwidth and inbuilt support for both single-precision and double-precision IEEE floating point arithmetic calculations. Even with all that muscle packed in, GPUs had been quite less hungry, when it comes to power. Worth every dollar spent on, GPU chips have grown on quite a large scale from being a peripheral commodity, to the most powerful and programmable processing devices available till date. Of course, GPUs are "*Multi-Core*" but not some small multi-core, modern GPUs could have over thousands of cores (NVIDIA TESLA K80 GPU accelerator having a massive 4992 CUDA cores [3]).

The exponential rise in graphics hardware performance coupled with advanced programming platforms have made GPUs a promising direction, for resource-intensive tasks in multifarious computing domains. Since the early days of computers, CPUs had been conventionally used to process multiple domain-based applications, generally called *workloads*. With enormous computing power driven by high-degree of parallelism and memory bandwidth packed right-in along and appreciable energy-efficiency coupled with advances in general-purpose computing using them, GPUs tend to outrun the CPUs as preferable computing engines in the recent years.

The use of GPUs for general-purpose computing started out by use of corner-case of graphics APIs. Here, programmers used graphics-pipeline for data transfer and carefully used buffer memory for mapping program data. The architecture didn't support this sort of operations, however for proper workloads great speedups were observed. This triggered hardware companies to add explicit hardware functionality for general purpose computing tasks, subsequently software functionality also improved for the same. The first commercial outcomes were NVIDIA's CUDA [4] and AMD's CTM [5] which opened a new field of GPU programming using high-level programming interface based on C and added hardware functionality.

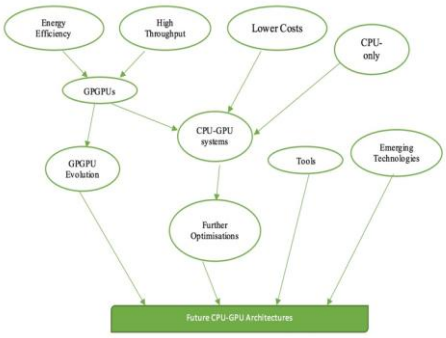

Figure 1 Evolution of CPU-GPU Architecture

Since 2011, we've seen chip-level integration of GPUs forming heterogeneous chips, where CPU and GPU exist on the same die. Early examples include AMD Fusion APUs [6], INTEL Sandy Bridge [7] and ARM MALI [8]. These models present sharing of memory and address space between CPU and GPU and are readily programmable by using different platforms such as Opens [9]. It is predicted that the combination of CPU-GPU will result in advent of throughput optimized cores, the future of general-purpose GPU computing. Based on present literature survey, the future work and research on GPUs can be represented by fig. 1.

1.1 Overview of Graphics Processing Unit (GPU)
"We've all heard 'desktop supercomputer' claims in the past, but this time it's for real: NVIDIA and its partners will be delivering outstanding performance and broad applicability to

the mainstream marketplace. Heterogeneous computing is what makes such a breakthrough possible [13]." Modern GPUs could be well seen as first class parallel-processing units, with outstanding computational capabilities and an exploding growth rate, far above that of CPUs and is becoming increasingly preferred in domain-specific parallel-data jobs. A GPU is a commodity computer chip optimized for graphics (2D/3D), being highly- parallel and multithreaded, it provides a brilliant visual output for games, high-resolution videos etc. Developments have led GPUs to be a versatile piece of hardware serving as a programmable visual engine and a highly scalable computing engine. Relatively simpler-core architectures are better direction for multicore processors as it turns out that the simpler architecture of cores gives numerous advantages over traditional architectures. They are readily scalable and have a wide range on scope. Additionally, programming platforms are pervasive and easy to use giving performance/cost ratio times better than traditional CPUs. Use of CPUs and GPUs on the same die for achieving accelerated performance has been a recent approach, also known as GPU-accelerated computing or heterogeneous computing. Continuous betterments of GPUs for mainstream non-graphics workloads is a hot topic of research presently.

1.2 Why GPUs? GPUs vs CPUs

CPUs and GPUs are both, classes of computing engines, before we understand why GPU is a better option in near future; we need to understand the major differences in both.

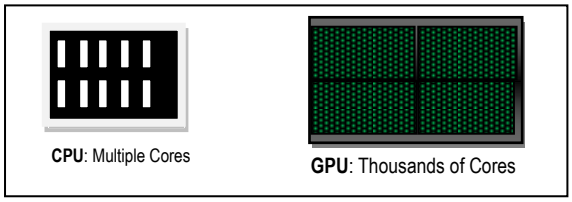

Figure 2 CPU vs GPU: Number of cores

The basic difference between a CPU and a GPU starts from their architecture and how they process tasks. The Former contains fewer optimized cores for implementing sequential processing of tasks. Parallelism is present, but at the task-level. On the other hand, a GPU comprises of a few thousands of smaller cores, built in a massively parallel architecture designed for efficient handling of simultaneous tasks. Thus, GPU implements data-level parallelism [11]. *Throughput* is defined as the amount of work done in a given amount of time [12]. CPUs are designed as to be low on latency and throughput, whereas the GPUs are designed to perform, with high throughput and latency. To minimize latency in CPUs, a large number of caches are required. The design allows to fetch maximum running applications from the caches. Due to this, GPUs are able to dedicate more of their transistor area to derive greater horsepower and hence can process over tens of thousands of threads concurrently. Additionally, GPUs have advanced, faster memory interfaces as they have to load/transfer higher amounts of data compared to CPUs.

GPUs are employed in almost all kinds of problems that require data parallelism (e.g., Graphics, Image and video processing, physics, scientific computing, gaming). GPUs seem to be perfect for such problems, in fact as the data increases, GPUs become more efficient. GPUs employ *Stream Processing* to achieve high throughput. Additionally, threads are hardware-managed hence, one is not required to write code for each of them individually.

GPUs are better in terms of computing power, memory bandwidth and high-energy efficiency. To harness full power of a GPU, sophisticated programming is required to exploit embedded parallelism, which may be difficult and requires greater amount of time.

Figure 3 CPU vs GPU: Transistor Allocation

If we consider the performance of latest and the best CPUs and GPUs out there. We see the results in favor of the GPUs as shown in fig. 4. In High Performance Applications(HPC), GPUs tend to beat out the CPUs brutally. For various scientific workloads as shown in fig.4, GPUs put a speedup of approximately 2.5-7 times the CPU performance.

1.3 Need of Heterogeneous Architectures

When both CPU and GPU are integrated on the same die, the resultant architecture is known as Heterogeneous Architecture. These type of architectures offer some great advantages over traditional ones. Firstly, it saves a lot of cost overheads due to use of shared resources by both CPU and GPU. Secondly, this allows better performance as there is no need of explicit data transfers between both the chips. Apart from performance improvements, it also gives rise to system developments.

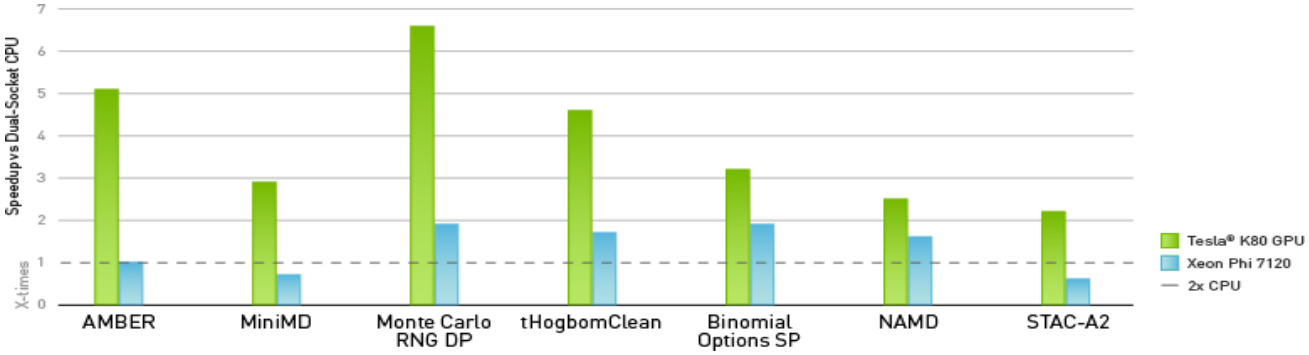

Figure 4 Intel's Xeon Phi CPU vs Nvidia Tesla K80 GPU on various HPC[15].

Faster communication and reduced costs along with bandwidth improvements allows a great deal of optimizations. Chip-level integrations not only reduce the latency significantly but also opens new communication pathways between CPU-GPU. Under "Further optimizations" in fig.1, The CPU-GPU system is designed to increase its efficiency and performance as a whole. For this to happen, rather than being designed for all workloads, the CPU architecture must be designed to perform tasks which GPGPU performs poorly. As we see in present, combination of CPU-GPU contains a lot of redundant components, which could be reduced in future architectures. For instance, previously there was a slow external memory interface for communication between the both, now being on the same die, they share common last level cache. This opens new direction in usage optimizations. Formally, CPUs and GPUs were designed as separate chips and hence it is observed that such separate organization doesn't seem to work efficiently.

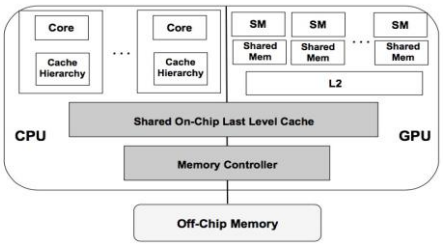

Figure 5 Heterogeneous Computing Architecture [14]

Based on current study, major research directions have been presented in fig.1. The apex shows the requirements that led to development of general purpose computing on GPUs (GPGPUs). We discuss the need for Unified Shader Architecture given by NVIDIA in section 2 with GPGPU architectural highlights. In section 3, we discuss various programming models for GPGPUs. We then discuss recently explored mechanisms for overcoming the major setbacks of GPU computing - Performance depletion under control flow divergence and poor process scheduling in section 4. In section 5, we present a study of latest contemporary GPU architectures, additional to this we will analyze the various issues and challenges before parallel computing chips in section 6. In the end, we sum up our study with conclusions in section 7.

2. OVERVIEW OF GENERAL PURPOSE GPUs

In this Section, we shall take a look at various GPGPU architectural highlights.

The modern GPU chip is a successor of a fixed function graphics pipeline, consisting of vertex processors and pixel processors. It was "Fixed" since it couldn't be programmed to carry out different operations except graphics through it. Initially, there was an imbalance of vertex and pixel shaders as earlier processes required more of pixel shaders than the vertex shaders. However, soon it was realized that non-unified architecture was unbalanced and was inefficient at resource utilization. This inspired the development of unified shader architecture, first found on Nvidia Tesla. The vision behind Tesla was to build a unified architecture with better resource utilization using scalar cores and all the shader operations being performed on same processors. Fig.6 shows contemporary Nvidia GPGPU architecture [16] [17]. It packs 15 processing elements known as *streaming multiprocessors* or SM units and 6 64-bit memory controllers. Every GPU SM unit includes 192 single-precision CUDA cores with each one of them having fully pipelined floating-point units along with ALUs. It also features full IEEE 754-2008 compliant single and double precision arithmetic processing along with the fused multiply-add (FMA) operation. Fig. 7 shows a diagram of a SM processing core. An SM includes 32 *single instructions multiple thread* (SIMT) lanes that collaboratively issue 1 instruction each cycle, it also includes special function units (SFUs) for rapid approximate transcendental operations. Threads are maintained in groups of 32 called *Warps*, 4 warp schedulers are also present aside 8 instruction dispatch units, which allows 4 warps to be processed simultaneously. Each thread is provided access to over 255 registers. The cores run on the primary GPU clock instead of the 2x shader clock. A unified memory request path for loads and store is supported, with an L1 cache per SM. The memory hierarchy is depicted in fig.8, with each SM containing 64KB of memory, configurable to as 48KB of shared memory along with an L1 cache of 16KB, or reverse i.e., 48KB of L1 cache and 16KB of shared memory. The GPU features a dedicated L2 cache of size 1536KB. In fact, the register files, shared memories, L1 cache, L2 cache and DRAM memory are Single-Error Correct Double-Error Detect (SECDED) ECC code protected.

GPUs use multithreading at an enormous scale to fetch maximum resource utilization. Keeping this in mind, threads are maintained in a large pool called warps. For instance, Nvidia Maxwell architecture launched in 2014 [19], has over 5.2 billion transistors and 2048 CUDA cores supporting 16 SMs. It enables 64 active warps per SM (2048 threads per SM). To avoid memory spills, up to 255 registers are made accessible per thread (64K 32-bit registers). Figure 8 depicts an example of warp scheduling. In each cycle, the scheduler chooses a ready to execute warp. Subsequently next instruction is assigned to threads of that selected warp. Warp selection considers parameters such as instruction type and fairness during selection. To achieve maximum efficiency, all lanes must be occupied.

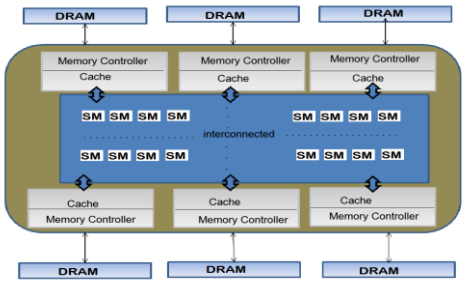

Figure 6 Contemporary GPU Architecture [14]

In such case, all 32 threads in a warp must take the same execution path, but in case if the threads diverge due to flow of control and different paths may get executed serially. The threads on a non-execution path are disabled while the others are re-converged to original path upon completion. SMs employ stacks to manage this divergence and re-convergence.

GPUs have been designed keeping in mind the scaling they must offer. Lack of global structures has facilitated it. GPUs implement better and efficient warp scheduling,

Table 1 Comparison of CPU and GPU

Parameter	CPU	GPU
Latency	Intolerant to Latency	Tolerant to latency
Thread Count	10s of Threads sequentially processed	10000s of threads processed in parallel
Cores	Fewer (Multi-threaded)	Thousands of SIMT Cores
Core Complexity	High	Low
Parallelism	Task Parallelism	Data Parallelism
Caching, Prefetching	Yes(to minimize latency)	No
Throughput	Low	High

multithreading is employed to hide large latencies. Moreover, there is no synchronization of threads globally across the GPU. GPUs also seem to be deficient of data lines for feeding purposes, instead of these there is a rich memory hierarchy composed of registers, caches and shared memory for persistence of data locality. Such an organization increases energy efficiency of GPUs and provide better scaling. Figure 9 shows the GPU advancements since 2007.

As observed the GPUs are advancing at an unprecedented rate nearly outpacing the Moore's law, with the latest architecture by Nvidia boasting a humongous performance of about 6.14 TFLOPS on the other side the rival AMD with a huge performance power of about 11.5 TFLOPS [21]. Single precision performance seems to grow over 6 times since 2006 with Double precision performance alleviated by a teraflop since its advent in 2008. Albeit, the increment in size of memory structures is relatively slow as the shared and memory registers have shown increase of about 4 times and 8 times respectively whereas memory bandwidth growth has been quite slow, witnessing an increment of about 2.2 times.

As discussed above, bandwidth seems to be a field of concern for GPU performance scaling. This limitation has been partially indemnified by manipulating the nature of workloads. Typical nature of workloads shifted to GPU are arithmetically intense and thus, show great performance due to GPU FLOP performance, but this doesn't seem to work well with all types of processes, specifically the ones with high bandwidth requirements. There have been numerous proposals for *spatial multitasking* [22] arithmetically intense and bandwidth demanding workloads on the same GPU chip.

Lee et al. [24] studied CPU scaling for throughput applications using GTX 280 vs Core i7-960 and revealed that the performance gap is of about two and a half times. If we take a look at raw facts and figures, comparing latest Nvidia state of the architecture named "Maxwell" to Intel's brand new architecture "Skylake" the gap between CPU and GPU performance in HPC is already wide enough. GPUs have shown a massive leap to over 10 times in performance while the CPU has only been able to double its GFlop performance.

Figure 7 A view of Nvidia Kepler SMX processing unit [18]

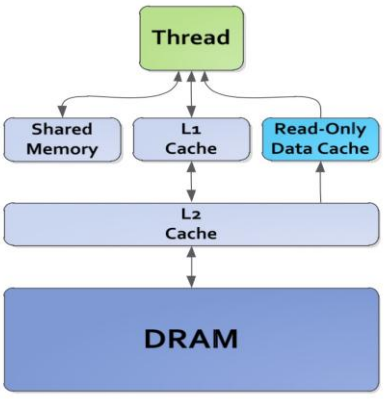

Figure 8 Kepler Memory hierarchy [18]

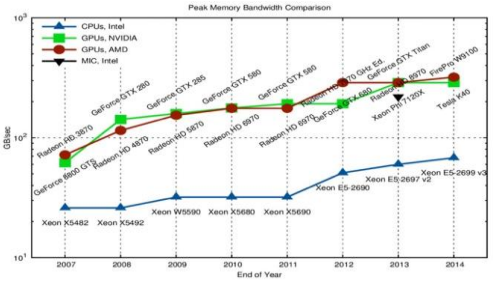

Figure 9 Peak Memory Bandwidth Comparison [20]

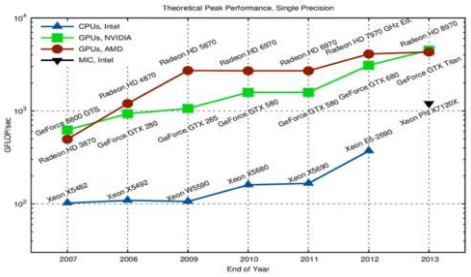

Figure 10 Thermal Peak Performance Single Precision Comparison [20]

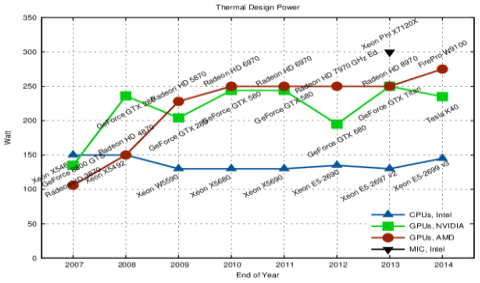

Figure 11 Thermal Design Power Comparison [20]

3. FORWARDING GPGPU ARCHITECTURE

To achieve better CPU-GPU systems in future, we need to take GPGPU design forward by mitigating all the challenges and deficiencies before the present GPU architectures. One of the great challenges before contemporary GPU architectures is handling of control-flow, which causes divergence of threads leading to loss in performance. Further, we also look at another obstacle which is warp scheduling, and discuss a better scheduling policy.

Branching is quite tedious to implement in case of GPUs as current architectures fetch one instruction per warp. GPUs at present use *single instruction multiple data* (SIMD) execution model enabling dynamic control. Grouping of threads is done to make a convoy of 32 logical threads called a warp, subsequently executed on a SIMD based hardware. Each thread in a warp can follow an independent control-flow path. This is a hybrid of SIMD technology and is known as *Single instruction multiple thread* (SIMT). In case, a control flow statement causes threads to diverge in a warp and take different execution pathways. It is up to the hardware (SM) to ensure proper execution. Figure 8 illustrates a sample problem where the warp size is assumed to be 4 and in those 4 threads, 2 of them follow the 'if' path while the other 2 take the 'else' path.The streaming multiprocessor has to visit each path in a sequential manner. An active mask field is present to mark the number of threads on a particular path in a control-flow graph.

```
if (thread.Idx < 2) {
// do work1
}
else{
// do work 2
}
```

Figure 8 Sample branching code

This is done via a reconvergence predicate stack used to partially serialize execution and proves to be a naïve approach, as it serializes execution making the inherent parallelism in GPU unutilized leading to performance losses. We first discuss stack based reconvergence then, we build up enhance schemes over it.

3.1 Stack Based Reconvergence

GPUs consists of large number of processing elements (SMs), each of them includes a big number of parallel execution lanes called CUDA cores in the Nvidia context. All of these parallel lames operate in SIMD fashion. As each of the thread in a CUDA core is independent to follow its own execution path, resulting in a hybrid known as SIMT. A great obstacle with the SIMT execution model is to ensure high utilization of resources in case of divergence of threads in a warp. The reason backing this concern for thread divergence are that masked operations keep on utilizing the available resources irrespective of need. Secondly, serialized execution of branches with every divergence denies parallelism. From this it is clear that care is necessary at the point of reconvergence of threads. In contemporary GPUs, all the diverged threads reach a specific point known as control-flow merge point or reconvergence point, also said that each diverged threads reconverge at the *immediate post-dominator* instruction of that branch [25]. The post-dominator (PDOM) instruction is the first instruction in the control flow that is present in both the diverged ways [26]. Stack based reconvergence is implemented treating execution of control-flow as a serial stack. In this, each time a divergent branch is encountered information about both the possible paths for execution are *pushed* into the stack, which may be hardware or software implemented. From related works, it is known that NVIDIA GPUs use the hardware based model [27]. The path at the top of stack (TOS) is executed sequentially. When the control

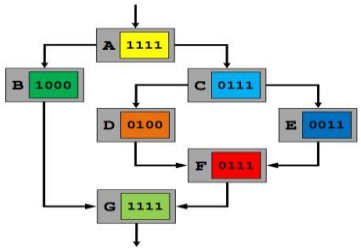

Figure 12 Sample Control-Flow Graph [28]

flow detects reconvergence point the entries are *popped*, turning towards alternate branch for execution. This can be considered as a depth-first search (DFS) traversal of the control-flow graph. The key feature being each path being executed serially and one by one. A Program Counter (PC) is associated with each node in the control-flow graph, track of which is maintained by reconvergence stack. The PC points to the first instruction in the node A and the reconvergence PC (RPC) is set to end of node A. When a warp undergoes a branching, the threads diverge and the stack stores the information about both executed and to-be executed path. In the beginning, the TOS is changed to the PC of reconvergence point The information about RPC is communicated explicitly via compiler analysis. Then, the PC of right path (block C) is *pushed* into the reconvergence stack along with the RPC (block G), after that the information of the non-taken path, i.e., block B is *pushed* similarly. In the end, left path now being the TOS is executed. It is clear that at a time, only a single path, the one at the TOS is executed. Thus, this execution model is also known as *single-path execution* (SPE) model [28].

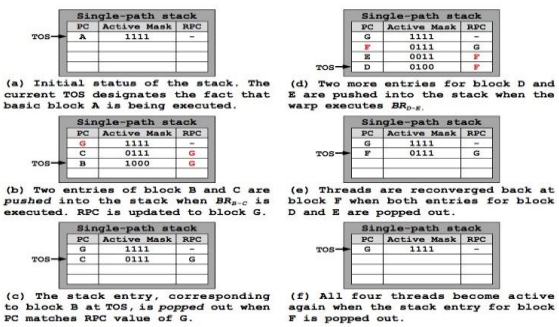

Figure 13 Execution flow of SPE model [28]

The example execution flow also brings to light two drawbacks of the SPE model. Firstly, the execution is serialized executing one path at a time, which works for almost all kind of workloads but results in some serious degradation of performance as it doesn't utilize the inherent parallelism efficiently. Second, the SIMD utilization decreases each time a control flow divergence is encountered. SIMD utilization has been a focus of research, recently.

In [26], Fung et al., put forward *Dynamic Warp Formation* to mitigate the deficiency of SIMD utilization during divergence.

3.2 Dynamic Warp Formation (DWF)

The performance loss due to thread divergence is inevitable, if the number of warps is unity. Contemporary GPUs support about 48 - 64 active warps, in case if a number of warps diverge at a single point in control-flow, then the diverged threads at that point from different warps following the same execution path can be clubbed to form new warp or warps. Since the newly formed warps of diverged threads are on the same execution path, they won't diverge and exhibit better hardware utilization. The creation of new warps from a pool of diverged threads on the same execution path is done by thread scheduler. It combines threads with same PC values. Figure 14 illustrates the concept of Dynamic Warp Formation, in which there are two paths A and B in a divergent branch, 2 warps 0 and 1 diverge at the same point. Here, threads of both 0 and 1 diverge into paths A and B. threads on the same path are combined to form new dynamically formed warps as shown having no divergence.

Dynamic Warp Formation can minimize area overheads by providing register-file configuration used in contemporary GPUs. This variety of DWF is known as "Lane-Aware" DWF. To reduce the number of ports in a register file, the number of SIMT lanes is kept statically fixed for thread execution, thus such a mechanism is needed. For instance, 32 SIMT lanes can be simultaneously fed using 32 register bank files. In case of DWF, the scheduler's job is to ensure that different SIMT lanes are used for threads of newly formed warps, which removes the requirement of having cross-bar connection among ALUs and register file banks resulting in profoundly simpler design.

DWF utilizes SIMT hardware resources to a greater length, but the basic condition for usage being existence of more than one warp. Also, if the progression of warps is different, there might be difficulties in regrouping of threads. The authors demonstrate a performance benefit of 20.7% for the mechanism.

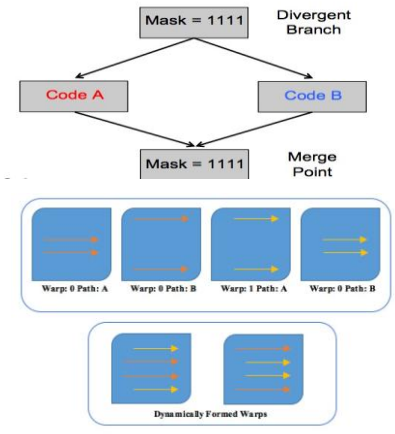

Figure 14 Illustration of DWF

3.3 Dynamic Warp Subdivision

Dynamic Warp Subdivision (DWS), proposed by Meng et al. [29] selectively deviates from stack-based reconvergence to diminish serialization. The core concept backing DWS is to consider both the left and the right paths as independently schedulable units or *warp-splits*, allowing serialization to be reduced. With DWS, *intra-warp* tolerance can be achieved. In DWS, a divergent path may employ the SPE model stack or instead an additional hardware data structure known as *Warp-Split Table* or WST which finds its use in tracking of independently schedulable warp-splits. Split-warps prove to be better than stack as they are concurrently schedule and dynamically formed and do not converge as early as the PDOM.

To overcome slow SIMD utilization, DWS employs three techniques. First, Warp-Split table, just like stack in SPE model to store information about PC. However, the RPC is not the PDOM of the diverging branch, rather it pertains to the last entry in the stack. Second, to mitigate the impact of delayed reconvergence and recursive subdivision, the warp-splits with identical PC values are attempted to be combined dynamically as well as opportunistically. Different from PDOM reconvergence in stack, the opportunistic combination may never occur. The decision of splitting a warp in the first place: a warp is only subdivided if the divergent branch's immediate PDOM is succeeded by a basic block of number of instructions being less than N. According to authors, this value of N (subdivision threshold) must be 50 instructions.

3.4 Large Warp Microarchitecture and Two-Level Scheduling

Narasiman et al. [30] proposed Large Warp Microarchitecture is a similar mechanism to dynamically form warps at runtime. However, the proposal differs in the method employed for warp formation. It all starts out with an initial warp having size slightly greater than the width of SIMT lane. At runtime, the technique generates smaller-warps lesser than the SIMT width from the initial larger one, this is depicted in figure 15. Here, a larger warp, consisting of total 16 threads arranged in a 2-D manner of 4 threads per warp and 4 such warps. We assume that the SIMT lane width is 4 threads. At each time interval, the scheduler picks any of the threads and maps to different lanes. The scheme assumes a similar register-file organization as in Fung et al's. [26] dynamic warp formation model.

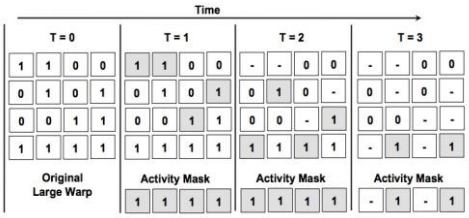

Figure 15 Sub-Warp Formation at Runtime [14]

In addition to Large Warp Microarchitecture, authors also propose a better scheduling algorithm called "Two-level scheduling". Previously, GPUs have been relying on Round-Robin warp scheduling which gives equal priority to all the concurrently executing warps. In such a scenario, memory requests of one warp might generate row-buffer hits for other warps requesting memory. As a consequence, simultaneously all warps get stuck at a single high-latency memory task. The authors proposed a fine solution to this problem by grouping large warp sets into smaller ones. Warps of a set are scheduled

together and execute individually. Howbeit, in case of high latency tasks the scheduler switches to another smaller set of warps. According to evaluation by authors, the schemes provide a performance speedup of 19.1% and an area overhead of 224 bytes.

4. FUTURE DIRECTIONS AND CHALLENGES

GPUs featuring enormous computing power and memory bandwidth has enabled their deployment in a multitude of High-performance applications. In Recent times, CPU-GPU systems have gained a lot of momentum as the heterogeneous processing elements have complementary attributes, which enables applications to perform even better on such systems. This section discusses challenges and future directions of research in CPU-GPU systems and parallel computing chips. We divide the Future research and challenges to parallel computing chips in 3 parts, namely – Challenges to parallel computing chips, Emerging technologies and Research tools.

4.1 Challenges to parallel computing chips.

4.1.1 Power Efficiency

Following Moore's Law, the number of devices packed on a single chip are increasing exponentially, with diminishing feature size. The result of this is that for all class of computers, whether mobile devices or the greatest supercomputers, power and energy will be an issue of grave concern. Because packing more devices on a chip is easy compared to designing powering circuits and cooling mechanisms. A chip's utility is determined by its performance at a specific power level, generally 3W for mobile devices and 150W for servers or desktops. For example, the power consumption for future supercomputers is expected to be 20MW and the supercomputers would perform exascale (10^{18} operations/second) calculations. An efficiency of approximately 20 picojoules (pJ) per FLOP is s must. Achieving this level of energy efficiency is a major challenge and will require a lot of research and innovation.

4.1.2 Memory Bandwidth

The memory bandwidth growth for GPUs has been quite slow, and renders an obstacle in high performance computing. Several techniques can improve utilization of bandwidth such as improving data locality and removal of overfetching of data from DRAM which is not of use for processor. Utilization can also be improved by increasing density of data transmitted via DRAM, using techniques like data compression. Power is another obstacle for off-chip memory. Assuming efficiency of 20pJ/bit for today's GPU chips, for fully tapping Maxwell's bandwidth of 224.3 GB/sec, approximately 35.88 W is required. Coupled with significant energy required for DRAM access and signaling results in much greater power consumption relative to on-chip memory (over 250 times less). For GPUs to grow in performance, increasing off-die memory bandwidth is a serious challenge before the research community.

Table 2. Projected DRAM bandwidth and energy [31]

DRAM technology	2010; 45nm	2017; 16nm
DRAM interface pin bandwidth	4 Gbps	50 Gbps
DRAM interface energy	20 to 30 pJ / bit	2 pJ / bit
DRAM access energy	8 to 15 pJ / bit	2.5 pJ / bit

4.2 Emerging Technologies

Emerging Technologies could be one of the hot areas for future research in GPGPUs as there has been no considerable work till date. Emerging technologies such as non-volatile memory(NVM), 3D stacking to GPUs. NVMs offer lower power consumption being a lower leakage and low power component, but still needs fine tuning to mitigate some disadvantages. 3D stacking can provide much needed potential to the memory bandwidth in GPUs. Nvidia announced its latest "Volta" architecture with *stacked DRAM* capabilities to be launched sometime in 2017.

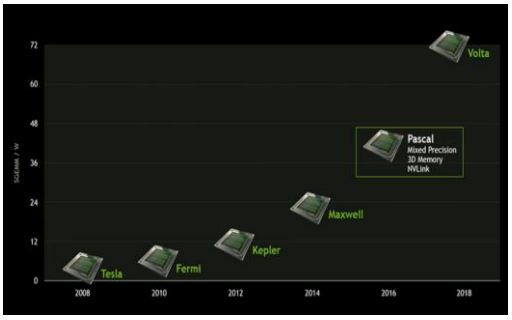

Figure 16 GPU emerging technologies [32]

4.3 Research Tools

The lack of proper research tools is one of the major areas of research in GPUs. While there are performance models available, but there is a lack of power models for GPGPUs. Once developed integration of such GPGPU models with CPU models requires further work. Similarly, there is an absence of temperature models for GPUs as well. This offers short term research opportunities in the field.

5. CONCLUSIONS: THINKING FORWARD

The sphere of GPGPU computing is expanding at an unprecedented rate. With approximately 400 million CUDA-capable GPUs sold till date. Single-threaded processors have become history, efficient use of parallelism is required for better architectures. The GPU has come up as a highly capable computing engine, designed to deliver high performance exceeding that of low latency CPUs. Early efforts in GPUs for general purpose computing has been remarkable, giving birth to GPGPUs as a payoff. The blend of CPUs and GPUs resulting in development of heterogeneous architectures proves to be better alternative than using either alternative. Due to their "unparalleled" efficiency on a range of applications, the CPU-GPU systems are rather becoming pervasive, becoming the only choice for HPC. Along with accelerating previous applications. We expect GPUs to enable a whole plethora of new applications possible. As GPU programming models evolve, making it easier to program, a greater adoption rate would be seen.

In this paper, we discussed GPUs as a better computing engine and the need of heterogeneous architectures. Following that, we discussed general purpose computing on GPUs and architectural highlights in section 2. In section 3, we looked at the various schemes and technologies used to create better GPGPU architectures. Section 4 discussed future research directions and challenges. Finally concluding the study in section 5.

6. REFERENCES

[1] URL: http://www.top500.org/lists/2015/06/.

[2] Gordon E. Moore, the co-founder and chairman emeritus of Intel and Fairchild Semiconductor.

[3] URL: http://www.nvidia.com/object/tesla-servers.html.

[4] NVIDIA Corporation. CUDA Toolkit 4.0. http://developer.nvidia.com/ category/zone/cuda- zone.

[5] AMD Close to the Metal (CTM). http://www.amd.com/.

[6] The AMD Fusion Family of APUs. http://sites.amd.com/us/fusion/apu/ Pages/fusion.aspx.

[7] URL: https://en.wikipedia.org/wiki/Sandy_Bridge.

[8] URL: http://www.androidauthority.com/arm-mali-closer-look-605021/.

[9] Khronos Group. OpenCL - the open standard for parallel programming on heterogeneous systems. http://www.khronos.org/opencl/

[10] URL: http://allegroviva.com/gpu-computing/difference-between-gpu-and-cpu/#prettyPhoto

[11] URL:http://www.nvidia.com/object/what-is-gpu-computing.html

[12] 'An Introduction to Modern GPU architecture' by Ashu Rege, Director of Developer Technology, NVIDIA

[13] Preliminary views on GPU technology by Burton Smith, Technical Fellow, Microsoft Formerly, Chief Scientist at Cray

[14] "The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing," Manish Arora, University of California, San Diego.

[15] URL:http://www.nvidia.in/object/gpu-computing-in.html

[16] E. Lindholm et al. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 2008.

[17] C. M. Wittenbrink et al. Fermi GF100 gpu architecture. IEEE Micro, 2011.

[18] NVIDIA's next generation cuda compute architecture: Kepler GK110. Technical report, 2012.

[19] URL: http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/

[20] URL:http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

[21] URL:http://www.pcgamer.com/hardware-report-card-nvidia-vs-amd/

[22] J. T. Adriaens et al. The case for gpgpu spatial multitasking. *High Performance Computer Architecture*, 2012.

[23] J. Nickolls et al. The GPU computing era. IEEE Micro, 2010.

[24] V. W. Lee et al. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In International Symposium on Computer Architecture, 2010

[25] Steven Muchnick. Advanced Compiler Design and Imple- mentation. Morgan Kaufmann, 1997

[26] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In 40th International Symposium on Microarchitec- ture (MICRO-40), December 2007.

[27] Collange, Sylvain. Stack-less SIMT Reconvergence at Low Cost, 2011.

[28] Minsoo Rhu, Mattan Eraz, The Dual-Path Execution Model for Efficient GPU Control Flow, 2013.

[29] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In 37th International Symposium on Computer Architecture (ISCA-37), 2010.

[30] V. Narasiman et al. Improving GPU performance via large warps and two-level warp scheduling. In International Symposium on Microarchitecture, 2011.

[31] S.W Keckler et al. GPUs and The Future of Parallel Computing. IEEE Micro,2011.

[32] [32]URL: http://www.extremetech.com/gaming/201417-nvidias-2016-roadmap-shows-huge-performance-gains-from-upcoming-pascal-architecture.