Recent Trends in Software and Hardware for GPGPU Computing: A Comprehensive Survey

B. Neelima
Ph. D Scholar
Dept. of Information Technology
NITK-Surathkal, Karnataka, INDIA
reddy_neelima@yahoo.com

Dr. Prakash S Raghavendra
Asst. Professor
Dept. of Information Technology
NITK-Surathkal, Karnataka, INDIA
srp@nitk.ac.in

***Abstract*--With the growth of Graphics Processor (GPU) programmability and processing power, graphics hardware has become a compelling platform for computationally demanding tasks in a wide variety of application domains. This state of art paper gives the technical motivations that underlie GPU computing and describe the hardware and software developments that have led to the recent interest in this field.**

Keywords: Graphics Processor (GPU), GPU Computing

I INTRODUCTION

The GPU's rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. The highly parallel graphics processing unit (GPU) is rapidly gaining maturity as a powerful engine for computationally demanding applications.

The GPU has always been a processor with ample computational resources. Over the past few years, the GPU has evolved from a fixed-function special-purpose processor into a full-fledged parallel programmable processor. This survey paper gives an overview of structure of graphics pipeline to programmable GPU pipeline. A survey paper on GPU Computing at this time is much useful as this field has captured interest of many academic and industry researchers. In this paper, we tried to give a complete development picture of hardware and software environment for GPU computing till date, giving a glimpse of our future work in this area.

II GPU ARCHITECTURE

This section gives an overview of graphics pipeline and its evolution as programmable pipeline.

A Graphics Pipeline

The input to the GPU is a list of geometric primitives, typically triangles, in a 3-D coordinate system. Through many steps, those primitives are shaded and mapped onto the screen, where they are assembled to create a final picture. Initially the input primitives are formed from individual vertices. Each vertex must be transformed into screen space and shaded. The vertices are assembled into triangles and rasterized. Rasterization is a process of determining which screen-space pixel locations are covered by each triangle. Using color information and texture form of information from global memory, each fragment is shaded to determine its final color. Finally the fragments are assembled into a final image with one color per pixel [1, 2].

B Programmable GPU Pipeline

More complicated shading and lighting operations were not efficiently expressed by fixed function pipeline. The key development was user specified programmable vertex and fragment processors have replaced the fixed function per-vertex and per-fragment operations. Later Shader programming was developed and the current GPUs support the unified Shader Model on both vertex and fragment shaders. GPU pipeline stages are divided in space, not in time. The part of the processor working on one stage feeds its output directly into a different part that works on the next stage [1, 2].

In the early days of programmable stages, the instruction set of the vertex and fragment programs were quite different, so these stages were separate. However, as both the vertex and fragment programs became more fully featured, and as the instruction sets converged, GPU architects reconsidered a strict task-parallel pipeline in favor of a unified shader architecture, in which all programmable units in the pipeline share a single programmable hardware unit. While much of the pipeline is still task-parallel, the programmable units now divide their time among vertex work, fragment work, and geometry work. These units can exploit both task and data parallelism. As the programmable parts of the pipeline are responsible for more and more computation within the graphics pipeline, the architecture of the GPU is migrating from a strict pipelined task-parallel architecture to one that is increasingly built around a single unified data-parallel programmable unit.

978-1-4244-6653-5/10/$26.00 ©2010 IEEE

III GPU COMPUTING

This section describes the GPU programming model and mapping of general purpose computations onto the GPU.

A The GPU Programming Model

The programmable units of the GPU follow a single program multiple-data (SPMD) programming model. As shader programs have become more complex, programmers prefer to allow different elements to take different paths through the same program, leading to the more general SPMD model. One of the benefits of the GPU is its large fraction of resources devoted to computation. Allowing a different execution path for each element requires a substantial amount of control hardware. Instead, today's GPUs support arbitrary control flow per thread but impose a penalty for incoherent branching. Elements are grouped together into blocks, and blocks are processed in parallel.

B General-Purpose Computing on the GPU

The programmer specifies geometric primitives for the computation for which rasterizer generates a fragment at each pixel. Each fragment is then shaded by an SPMD general purpose fragment program, which computes the value with the math operation by accessing the global memory. The result from global memory is then used as input for future computations. But now the non graphics interface to hardware, specifically programmable units are developed. Initially the programmer defines a structured grid of threads for the computation domain and then an SPMD general-purpose program computes the value of each thread. The value of each thread is a combination of math operation and global memory access. The resulting buffer in global memory can then be used as an input for computations.

IV SOFTWARE ENVIRONMENT

In the past, the majority of GPGPU programming was done directly through graphics APIs. With DirectX 9, higher level shader programming languages are developed like Cg (C for Graphics), HLSL (DirectX High Level Shader Language), GLSL (OpenGL Shading Language) etc, but they are inherently shading languages only where computation must be expressed in graphics terms only. Then there was a development of higher level languages designed for computations by abstracting the GPU as streaming processor. The stream programming model structures programs to express parallelism and allows for efficient communication and data transfer, matching the parallel processing resources and memory system available on GPUs. In this section we give a brief overview of the evolution of GPU computing languages starting from shading languages, streaming languages to the latest industry consortium product named OpenCL (Open Specification for Computation Language) along with vendor specific languages.

A Shading Languages

Shading languages used in offline rendering are geared towards maximum image quality. RenderMan Shading Language [3] is the first shading languages and defines six shader types.

Cg [4] was a portable shading language to ease 3D programming. The Cg compiler outputs DirectX or OpenGL shader programs. Cg provides abstractions that are very close to the hardware, with instruction sets that expand as the underlying hardware capabilities expand. The GLSL is the standard shading language for OpenGL for the language features (e.g. integers) that do not directly map to hardware. The language unifies vertex and fragment processing in a single instruction set, allowing conditional loops and branches [5]. HLSL is a C-style shader language for DirectX based proprietary shading language developed by Microsoft for use with the Microsoft Direct3D API [3].

B Streaming Languages

This section gives an overview of different evolutions done on non commercial (Academic/research) Projects and general purpose language development that can run on multiple platforms and also vendor specific language developments.

1 Non-commercial stream programming languages

BrookGPU was an early academic research projects with the goal of abstracting the GPU as a streaming processor. Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar and efficient language [6, 7]. Sh [8] is a Meta programming developed at the University of Waterloo. It is built on top of C++.

The ACOTES project gives partial solutions to the programmer, for streaming applications, to identify opportunities for parallelizing the algorithm [9]. Microsoft's Accelerator is very compute centric, relies on just-in-time compilation of data-parallel operators to fragment shaders. It is an array-based language based on C# [10]. Shallows is a C++ library aimed to reduce the time spent on writing and debugging OpenGL related C/C++ code [11]. Brahma is an open source shader meta-programming framework for the .NET platform that generates shader code from intermediate language at runtime, enabling developers to write GPU code in C# [12]. The StreamIt [13] programming model has been proposed to exploit parallelism in streaming applications on general purpose multi core architectures. Scout [14] is a GPU programming language designed for scientific visualization.

Finally, OpenCL [15] is a Multi-architectural Languages, a recent standard developed by an industry consortium called Khronos Group formed in 2008, for programming heterogeneous computers. OpenCL consists of a programming language for accelerator cores and a set of platform API calls to manage the OpenCL programs and devices. The programming language is based on C99 having CUDA (Compute Unified Device Architecture) like syntax and using SPMD kernels for the data-parallel programming model. Task-parallelism is also supported with single-threaded kernels, expressing parallelism using 2–16 wide vector data types and multiple tasks. Kernel execution is managed by command queues, which supports out-of-order execution, enabling relatively fine-grained task-parallelism. Out-of-order tasks are synchronized by barriers or by explicitly specifying dependencies. Synchronization between command queues is also explicit. The standard defines requirements for OpenCL compliance in a similar manner as OpenGL, and extensively defines floating-point requirements. Optional extensions are also defined, including double precision and atomic functions.

2 General purpose stream programming languages

HMPP Workbench [16] is a directive-based compiler for hybrid computing, based on C and FORTRAN directives. RapidMind [17] was started in 2004 based on the academic research related to the Sh project and was acquired by Intel in 2009. The RapidMind team and technology was integrated into the Intel Ct product. RapidMind lets the programmer specify any computation he wants to leverage multiple cores within existing C++ applications, without changing compiler or workflow.

3 Vendor specific stream programming languages

Ct or C/C++ for Throughput Computing Technology [18] from Intel is Scalable, Portable, and Deterministic Parallel Programming model. CTM (Close To the Metal) [19] provides a low-level hardware abstraction layer (HAL) for the R5XX and R6XX series of ATI GPUs. AMD also offers the compute abstraction layer (CAL), which adds higher level constructs. The AMD Stream SDK [19] is the successor of ATI's CTM initiative and provides the means to program AMD GPUs directly. The software stack consists of the AMD Compute Abstraction Layer (CAL), which supports the R6xx-series and newer AMD GPUs, and the high-level Brook+ language. Brook+ is AMD's extension of BrookGPU, and is a high-level programming language. The programmer defines kernels that operate on input and output data streams, and multiple kernels operating on the same streams create an implicit dependency graph.

CUDA [20] is a general-purpose multi-threaded SIMD model for GPGPU programming. Sequential code executed on the CPU as host code. Parallel code is implemented as a set of kernel functions to be executed in an SIMD fashion, by a number of threads. Threads are grouped as a grid of thread blocks, and more than one thread block can be assigned to a multiprocessor. The CUDA memory model has an off-chip global memory space, which is accessible by all threads, an off-chip local memory space, which is private to each thread, a fast on-chip shared memory space, which is shared only by threads in the same thread block, and registers, which are private to each thread. The foundation of the CUDA software stack is CUDA PTX, which defines a virtual machine for parallel thread execution. CUDA has plug-ins to support the developer. PyCUDA is a Python wrapper for functions that run on GPU. It also has MATLAB plug-in called Jacket which is developed on Acceleryes and a FORTRAN library by name Flagon.

V HARDWARE ENVIRONMENT

AMD, Intel, and NVIDIA are the three major GPU vendors, where AMD and NVIDIA dominate the high performance gaming market. Intel, however, has disclosed plans to release a high-performance gaming GPU called Larrabee. In this section we will give a review of NVIDIA GT200, NVIDIA's new architecture by code name Fermi, AMD RV770 and Intel's Larrabee.

GT200 Architecture: It is typically programmed using CUDA which exposes an SPMD programming model using a large number of threads organized into blocks. In addition to eight scalar processors, the streaming multiprocessor also contains a double precision unit, two special function units, 16 KB of shared memory, 8 KB constant memory cache, and 16384 32-bit registers.

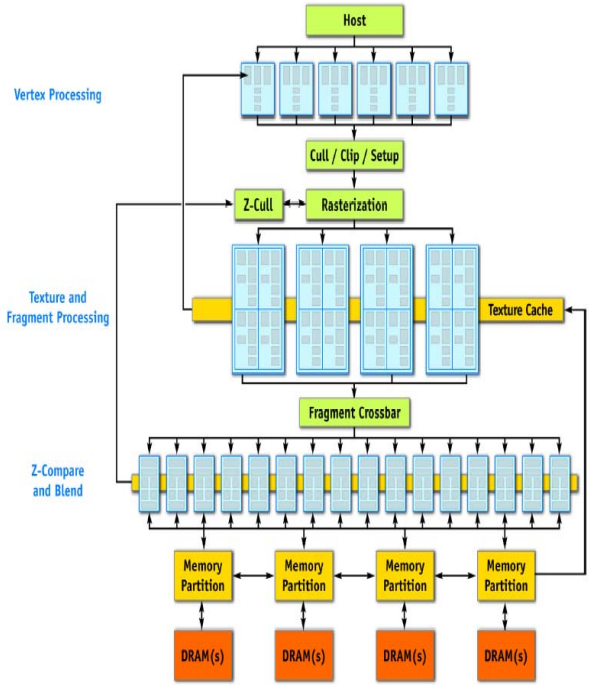

Figure 1: GT 200 Architecture (Courtesy: GPU-Gems2, Chap-30)

Each scalar processor is a fully pipelined arithmetic-logic unit capable of integer and single precision floating point operations, while the special function unit contains four arithmetic-logic units, mainly used for vertex attribute interpolation in graphics and in calculating transcendentals. The GT200 has eight 64-bit memory controllers, providing an aggregate 512-bit memory interface to main GPU memory. It can either be accessed from the streaming multiprocessors through the texture units that use the texture cache [4, 20].

GT300 Architecture: NVIDIA's next-generation CUDA architecture, code-named Fermi [4], adds powerful new features for general-purpose computing. The number of transistors on chip has increased to 3.0 billion. The number of CUDA cores has increased to 512. The number of wrap schedulers per Streaming processor is increased to 2 and special function units to 4. All vital parts of memory are also protected by ECC, and the new architecture has cache hierarchy with 16 or 32 KB L1 data cache per Streaming Multiprocessor, and a 768 KB L2 cache shared between all Streaming Multiprocessors. The memory space is also unified, so that shared memory and global memory use the same address space, thus enabling execution of C++ code directly on the GPU. The new chip also supports concurrent kernel execution, where up-to 16 independent kernels can execute simultaneously.

RV770 Architecture: The current generation of AMD FireStream cards [19] is based on the RV770 architecture. It employs an SPMD model over a grid of groups. The SIMD engine is the basic core of the RV770, containing 16 shader processing units (SPUs), 16 KB of local data share, an undisclosed number of 32-bit registers, 4 texture units, and an L1 texture cache. Groups are automatically split up into wave fronts of 64 threads, and the wave fronts are executed in SIMD fashion by four consecutive runs of the same instruction on the 16 shader processing units. Synchronization is done using memory barriers.

Thread grids are processed in either pixel shader or compute mode. The pixel shader mode is primarily for graphics, and restricts features to that of the R6xx architecture without local data share. Using the compute shader mode, output is required to be a global buffer.

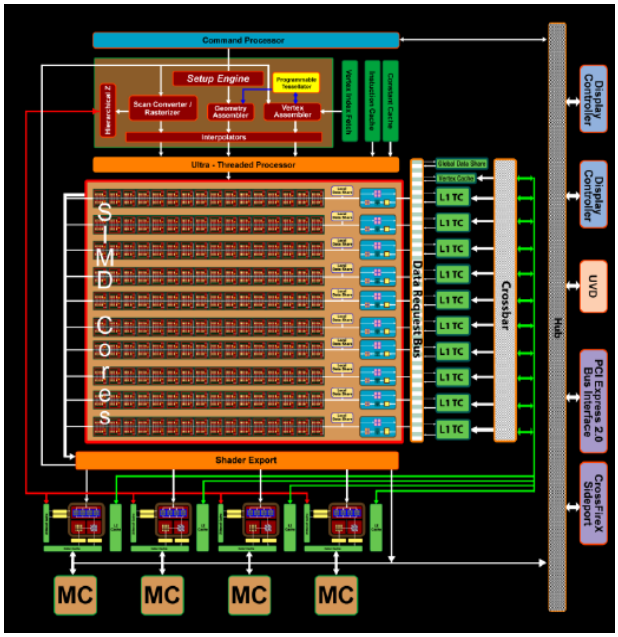

Figure 2: RV770 GPU Block Diagram

Larrabee: The upcoming Larrabee GPU from Intel [19] can be considered a hybrid between a multi-core CPU and a GPU. GPU that shares the same architecture with the CPU could ease software development and task sharing. Future PCs will be able to use their Larrabee GPUs as coprocessors for compute intensive tasks other than graphics. The Larrabee architecture is based on many simple in-order CPU cores that run an extended version of the x86 instruction set. Each core is based on the Pentium chip with an additional 16-wide SIMD unit which executes integer, single precision, or double-precision instructions. Larrabee has fully programmable graphics pipelines and much hardwired acceleration logic than other GPUs. A striking design choice is the lack of a hardware rasterizer, which forces the Larrabee to implement rasterization in software.

VI OBSERVATIONS

By looking at the software and hardware evolutions, it is clear that there has been lot of developments done both to the hardware and software to map the general purpose computations onto GPU and now most of the research is concentrated on making these processes much easier for the systems and programmer.

Development from shading languages to streaming programming has given a boost for GPU Computing. There are many academic research projects on stream languages, which gave birth to many vendor specific languages, for example Brook+ of AMD is an extension of BrookGPU developed by Ian Buck of Stanford University. NVIDIA's CUDA is a higher level interface than AMD's HAL and CAL. Similar to Brook, CUDA provides a C-like syntax for

executing on the GPU and compiles offline and exposes two levels of parallelism. CUDA is more flexible and complex, whereas RM (RapidMind, now acquired by Intel and trade named as Ct) is more portable.

However, all of this flexibility and potential performance gain comes with the cost of requiring the user to understand more of the low-level details of the hardware, notably register usage, thread and thread block scheduling, and behavior of access patterns through memory. Hence all the industries have formed a consortium and trying to develop an industry standard language. OpenCL is one of the efforts towards that.

Nvidia is a big, established company, and can introduce a new processor architecture and programming model to enhance the GPU Computing capabilities. One such effort made is Fermi, which can make many researchers to look into this field for high performance computing. It may have strong contention from Intel's Larrabee's.

However, the major challenge for application developers is to bridge gap between theoretical and experienced performance that is writing well balanced applications where there is no apparent single bottleneck. It is not only application developers who have demands to architectures but economic, power and space requirements in data centers impose hard limits to architecture specifications.

VII RELATED WORK

There have been similar survey papers given on hardware and software environments or techniques of GPU computing [1, 2]. A survey of general purpose computing on graphics hardware by John D Owens is one such paper, which appeared in Eurographics-2005. Most of the literature concentrates on vendor specific languages like CUDA, Brook+ only. Our paper covers all the evolutions starting from the rendering shading language. A survey paper on GPU Computing at this time is much useful as this field has captured interest of many academic and industry researchers. We hope this paper gives a complete development picture of hardware and software environment for GPU computing till date, giving a glimpse of our future work in this area. We hope it gives a good insight into the latest research happening in GPU computing for the readers of this paper.

VIII FUTURE WORK

GPUs are the best performing architectures per watt and uses cache miss as opportunity not as a problem. GPU performs best when the same operation is independently executed on a large set of data, but do not perform well on algorithms that require extensive synchronization and communication. All the software evolutions discussed have made it easy for the programmer to map general computations on to GPU than programming the fixed pipeline. However, they require programmers to learn proprietary C/C++ extensions and techniques. Programmers must rewrite at least some of their code to expose hidden data level parallelism.

Several Studies have been made to optimize GPGPU applications. A compile time transformation scheme has been developed for automatic optimization of CUDA programs by Baskaran et al [21]. CUDA-Lite [22] is a translator which generates codes for optimal tiling of global memory data. OpenMP-to-GPGPU translator [23] was developed by Lee et al to translate standard OpenMP shared memory programs into CUDA-based GPGPU programs. Narayanan et al [24] developed a framework for efficient and scalable execution of domain specific templates on GPUs. But all these models are specific to a programming model or vendor specific. We are proposing to develop a programming frame work where it gives a common platform for all heterogeneous architectures and also provides a library link for the legacy models developed already for those who want to use and see the performance difference. Based on this model, a compiler framework works to standardize the transformation of general purpose application to GPU computing application by providing a common platform or intermediate representation.

IX CONCLUSIONS

Currently GPU computing has focused only on GPU Cards. But future computing has to focus on using hundreds of accelerators along with CPU cores than symmetric processing. In future GPU architectures, If GPU is directly connected to the memory through the bus, which may alleviate PCI express bottleneck, the computation algorithm and application development will see a new phase altogether.

The best design strategy would be not to overlook soft wares that existed for decades, but to develop new applications and algorithms for existing and future architectures. High-performance computing on GPUs has attracted an enthusiastic following in the academic community as well as within the industry, so there is growing expertise among programmers and alternative approaches to software development. There is a quest for industry standard model and many companies are working towards that goal.

Developers needing high performance right now must choose a nonstandard approach. CUDA is a strong contender, but it's only one option among many. As with most things, one should try to select the best tool for the job as whether straightforward C++ code to be executed or a complex scientific application to be executed on GPU core.

REFERENCES

[1] Andre R Brodtkorb et al "State-of-the-art in Heterogeneous Computing", IOS Press, 2005.

[2] John D. Owens et al "A Survey of General-Purpose Computation on Graphics Hardware." In Euro graphics 2005, State of the Art Reports, August 2005, pp. 21-51.

[3] http://en.wikipedia.org/wiki/

[4] http://developer.nvidia.com/

[5] www.opengl.org/documentation/glsl/

[6] http://graphics.stanford.edu/proje cts/brookgpu/

[7] Ian Buck et al, "Brook for GPUs: Stream Computing on Graphics Hardware"

[8] http://libsh.org/

[9] http://www.hitech-projects.com/euprojects/ACOTES/

[10] http://research.microsoft.com/

[11] http://shallows.sourceforge.net/

[12] http://brahma.ananthonline.net/

[13] http://groups.csail.mit.edu/cag/strea mit/

[14] Patrick McCormick et al,"Scout: a data-parallel programming language for graphics processors". Parallel Computing Vol-33, Issues 10-11, November 2007, 648-662pp.

[15] www.khronos.org/opencl/

[16] www.caps-entreprise.com/hmpp.html

[17] rapidmind.net/pdfs/RapidMindGPU .pdf

[18] http://intel.com/research/

[19] developer.amd.com/

[20] http://www.digital- daily.com/

[21] M. M. Baskaran et al, "A compiler framework for optimization of affine loop nests for GPGPUs", ACM International Conference on Supercomputing (ICS), 08.

[22] S. Ueng et al, "CUDA-lite: Reducing GPU programming complexity", International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2008

[23] Seyong Lee et al," OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization" ACM-SIGPLAN-PPoPP'09.

[24] Narayanan Sundaram et al, "A framework for efficient and scalable execution of domain specific templates on GPUs", IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2009