Stream Processors and GPUs: Architectures for High Performance Computing

Christos Kyrkou, *Student Member, IEEE*

***Abstract*—Architectures for parallel computing are becoming all the more essential with the increasing demands of multimedia, scientific, and engineering applications. These applications require architectures which can scale to meet their real-time constraints, with all current technology limitations in mind such as the memory gap, and power wall. While current many-core CPU architectures show an increase in performance, they still cannot achieve the necessary real-time performance required by today`s applications as they fail to efficiently utilize a large number of ALUs. New programming models, computer architecture innovations, coupled with advancements in process technology have set the foundations for the development of the next generation of supercomputers for high-performance computing (HPC). At the center of these emerging architectures are Stream Processors and Graphics Processing Units (GPUs). Over the years GPUs exhibited increased programmability that has made it possible to harvest their computational power for non graphics applications, while stream processors because of their programming model and novel design have managed to utilize a large number of ALUs to provide increased performance. The objective of this survey paper is to provide an overview of the architecture, organization, and fundamental concepts of stream processors and Graphics Processing Units.**

***Index Terms*— High Performance Computing (HPC), Stream Programming model, Stream Processors. Graphics Processing Units (GPUs), General Purpose computation on a Graphics Processing Unit (GPGPU),**

I. Introduction

Advancements in modern technology and computer architecture allows today`s processors to incorporate enormous computational resources into their latest chips, such as multiple cores on a single chip. The challenge is to translate the increase in computational capability with an increase in performance. Hence, new architectures that are optimized for parallel processing rather that single thread execution have emerged [1]. Parallel computing architectures emerged as a result of increasing computation demands of various applications ranging from multimedia, to scientific and engineering fields. All these applications are *compute intensive* and require up to 10s of GOPS (Giga Operations Per Second) or even more. Only a small fraction of time spent for memory operations and the majority of operations using computational resources. Also most applications come with some type of real time constraint. These characteristics require parallel and scalable architectures that can efficiently utilize processing resources to achieve high performance. Such parallel architectures must consider the following [2]: They need an efficient management of communication in order to hide long latencies with useful processing work. Additionally, to keep their processing resources utilized they need a memory hierarchy that provides high bandwidth and throughput. Finally, the increasing number of computational resources will require more power and it will be challenging to manage it efficiently. Two such parallel architectures are Stream Processors and Graphic Processing Units (GPUs).

Relying on parallel architectures alone is not sufficient to gain high performance. Such architectures require a programming model that can expose the inherit application parallelism and data flow, so that the hardware can be efficiently utilized. Developing such a programming model requires a dramatic shift from the existing sequential model used in today's CPUs, to a data driven model that suits the parallel nature of the underlying hardware. The stream programming model was developed with these considerations in mind [3]. In this model data are grouped together into streams, and computations can be performed concurrently on each stream element. This exposes both the parallelism and locality of the application yielding higher performance. The introduction of the stream programming model had as a result the development of specialized stream processors [4], optimized for the execution of stream programs, thus, combining both high performance and programmability.

Driven by the billion-dollar market of game development with ever increasing performance demands, GPUs have evolved into a massively parallel compute engine. Because of the high performance demands the architecture of a GPU is drastically different from that of a CPU, transistors are used for computational units instead of caches and branch prediction and their architecture is optimized for high throughput instead of low latency [2]. Moreover, GPU performance doubles every six months [5] in contrast to the CPU performance which doubles every 18 months. Consequently GPUs offer order(s) of magnitude greater performance and are widely considered as the computational engine for the future. Since early 2002-2003 there has been a massive interest in utilizing GPUs for general purpose computing applications under the term General Purpose computing on GPU (GPGPU) [1]. This shift was primarily motivated by the evolution of the GPU from just a hardwired implementation for 3D graphics rendering, into a flexible and programmable computing engine.

C. Kyrkou is with the Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus (e-mail: kyrkou.christos@ucy.ac.cy).

The purpose of this survey paper is to provide an in depth overview of these emerging parallel architectures. The outline of this paper is as follows: The fundamentals of the stream programming model and a general architecture of a stream processor are discussed in Section II. Section III provides details on four stream processor architectures. An introduction to GPUs and details about their evolution and architectural trends are given in Section IV, while Section V provides a discussion on GPGPUs, and the next generation of GPUs that are enhanced for general purpose computing. Finally Section VI concludes the paper.

II. Stream Processors Fundamentals

A. Stream Programming Model

The stream programming model arranges applications into a set of computation *kernels* that operate on data *streams* [3]. Expressing an application in terms of the stream programming model exposes the inherent locality and parallelism of that application, which can be efficiently handled by appropriate hardware to speed-up parallel applications. By using the stream programming model to expose parallelism, producer-consumer localities are revealed between kernels, as well as true data localities in applications. These localities can be exploited by keeping data movement locally between kernels that communicate which are more efficient rather than using global communication paths. Fig. 1 shows how kernels are chained together with streams.

- *Streams:* A collection of data records of the same type, ranging from single numbers to complex elements.
- *Kernels:* Operations that are applied on the input stream elements. Kernels can perform simple to complex computations and can have one or more input and output streams.

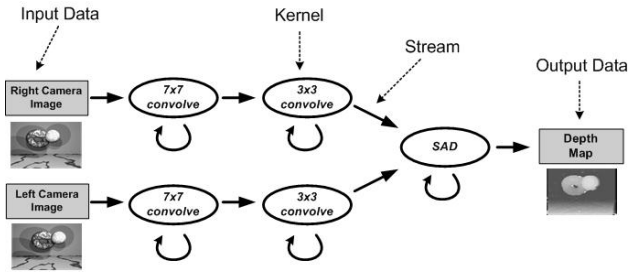

Fig. 1. Stereo Depth Extraction in the stream model[6].

The advantage of expressing the application in the form of a *stream program* is that it exposes two types of localities. The first is *kernel locality*. During a kernel execution, all references are to variables local to the kernel, except the values read from the input stream or written to the output stream. The second is *producer-consumer locality*, which involves streams moving between kernels. One kernel produces a stream which another kernel in the immediately consumes. By efficiently sequencing such kernels, the stream values are kept local and are consumed soon after they are produced. The stream model ensures that kernel programs will never access the main memory directly. The stream programming model defines communication and concurrency between *streams* and *kernels* at three distinct different levels [4]. In this way take the locality and parallelism of the application are exposed. These restrictions in communication help in the most efficient use of bandwidth.

Communication:

- *Local:* Used for temporary results produced by scalar operations within a kernel.
- *Stream:* For data movement between kernels. All data are expressed in the form of streams.
- *Global:* Necessary for global data movement either between to and from the I/O devices, or for data that remain constant throughout the application lifespan.

Concurrency:

- *Instruction Level Parallelism (ILP):* Parallelism exploited between the scalar operations within the kernel.
- *Data Parallelism:* Applying the same computation pattern on different stream elements in parallel.
- *Task Parallelism:* As long as no dependencies are present, multiple computation and communication tasks can be executed in parallel.

B. Stream Operations

Typical operations that can be performed on streams are [5]:

- *Map-Apply:* This operation is used to process all elements of a stream by a function.
- *Gather and Scatter*: Addressing mode often used when addressing vectors. Gather is a read operation with an indirect memory reference, while scatter is a write operation with an indirect memory reference. Both types of memory referencing are shown in Table 1.

TABLE 1: Gather and Scatter memory addressing

Scatter	Gather
for (i=0; i<N; ++i) *++A[B[i]];*	*for (i=0; i<N; ++i)* *A[i] = B[i] + C[D[i]];*

- *Reduce:* The operation of computing a smaller stream from a larger input stream.
- *Filtering:* The operation of selecting a subset of stream element from the whole stream and discarding the rest. The location and number of filtered elements are variable and are not known beforehand.
- *Sort:* Transforms a stream into an ordered set of data.
- *Search:* This operation finds a specific element within the stream, or the set of nearest neighbors to a specified element.

C. Generic Stream Processor Architecture

Stream processors belong to a new class of architectures that are compute intensive, and achieve high performance by reducing dynamic control, and providing a large number of programmable functional units. The main architectural components of a stream processor are shown in Fig. 2. Stream processors are specifically designed for the stream execution

model, in which applications have large amounts of explicit parallel computation, structured and predictable control, and memory accesses that can be performed at a coarse granularity. Hence, their architecture also takes into account the concurrency and communication levels defined by the stream programming model in order to maintain high locality and parallelism levels. Specifically, their memory hierarchy is partitioned into three distinct levels of storage. Each of the memory levels provides an order of magnitude more bandwidth as it gets closer to the functional units. This maintains temporary data close to the functional units while only true global data are stored in the external memory. Because of the efficient bandwidth use, most of the work of a stream processor is done on-chip with only 1% of global data references requiring external memory access. The three levels of memory hierarchy are given below [4]:

- *Local Register File (LRF):* Used for local data communication and fast access of temporary data by the functional units.
- *Stream Register File (SRF):* Used to store streams and transfer data between the LRFs of major components.
- *Off-Chip Memory:* Stores global data and is used only when necessary.

Better performance can be achieved by partitioning the SRF and LRF as well, which will also reduce the number of ports into the register files.

As with the memory hierarchy the functional units of a stream processor are distributed in ALU Clusters. All clusters consist of the same type and amount of functional units. The functional units within a cluster may differ between them, ranging from multipliers to floating point units, to ALUs. The ALU clusters execute kernel instructions that they receive from the μController in SIMD fashion. Each of the clusters operates on an element of the input stream, thus, providing data parallelism. The SIMD organization helps provide the necessary data bandwidth to feed all ALU clusters.

Because the stream programming model exposes the concurrency and communication in an application it is desirable that the instruction scheduling be handled by the compiler. Hence, it is possible to exploit the instruction level parallelism of a kernel via VLIW instructions. Each VLIW instruction consists of several ALU operations as well as control signals for every LRF in a cluster. Moreover, VLIW control is efficient since additional hardware for register renaming, dependency detection, or operation reordering is not necessary. The free transistors can be used for additional computational units.

Stream processors usually work as coprocessors for a general purpose host processor. The host processor is responsible for the execution of scalar code. Any kernel instructions and stream load/stores are dispatched from the host processor to the stream controller. From there on they are assembled and sent to the appropriate unit. Additionally, the host processor manages the system resources. The stream controller manages the communication with the host processor. It keeps track of the stream instructions that it receives from the host processor and sequences the VLIW instructions to the μController. The external memory controller handles global data references from the external DRAM. Various stream processors differ in the number of ALU clusters, the types of ALUs, the number of ALUs in a cluster, the capacity of the SRF and LRF, and their bandwidth.

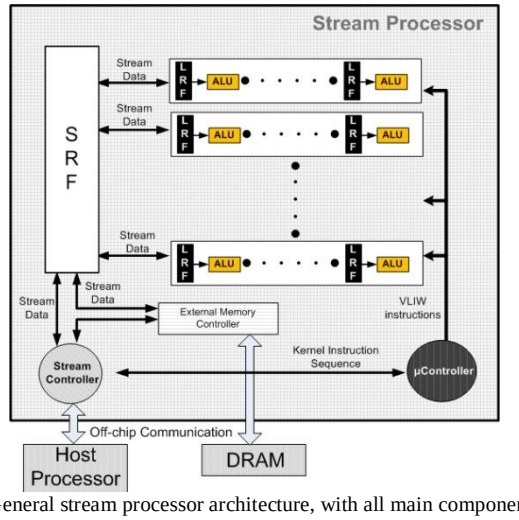

Fig. 2. General stream processor architecture, with all main components.

III. Stream Processors Architectures

A. Imagine

Imagine [6, 7, 8] was a project overtaken at the Stanford University and was sponsored by Intel and Texas Instruments. It involved the implementation of a stream processor for media applications, and it was the first implementation of a stream processor chip. This project had many contributions such as the stream processor architecture which was later employed into other stream processors, and development tools for stream applications including a stream programming language (StreamC and KernelC) and compiler. A prototype Imagine processor was design and fabricated in conjunction with Texas Instrument. Imagine contains 21 million transistors and has a die size of 16mm x 16mm in a 0.15 micron standard cell technology [6]. The architecture of Imagine is illustrated in Fig. 3.

1) Architecture

Image operates as a coprocessor to a general purpose processor host, and is geared towards multimedia applications such as encryption, 3D graphics, and video processing. The host processor is responsible for the execution of scalar code while all stream instructions are sequenced and issued to Imagine via a stream controller. All external kernel inputs and outputs are communicated by data streams which reside in the SRF. Incoming instructions operate on those streams. The input and output streams generated during kernel executions are also stored in the SRF. The Imagine architecture takes advantage of the stream model properties such as the data-parallel organization of records within a stream, the sequential ordering of accesses to streams, locality of kernel data, and simple control flow within kernels.

There are 8 ALU clusters in Imagine and each is comprised of six 32-bit floating point units (FPUs) (3 adders, two multipliers, and a divide/square-root unit). The clusters operate in an eight wide SIMD manner. That is, the same instruction is executed on all 8 clusters on different elements

of the input data stream. The six FPUs execute VLIW instructions issued by the cluster microcontroller which also keeps track of the program counter. Local Register Files (LRF) in each cluster are used to provide the ALUs with data. Mechanisms are present to provide communication between the ALUs and the SRF as well as inter cluster communication for data transfer between clusters.

Upon the arrival of a kernel instruction from the host processor the micro-controller starts issuing VLIW instructions. Each cluster reads a stream element and executes the instruction it has received from the micro-controller. The clusters can read elements from the same or different streams. The clusters proceed to write the output stream element to one or more streams back to the SRF.

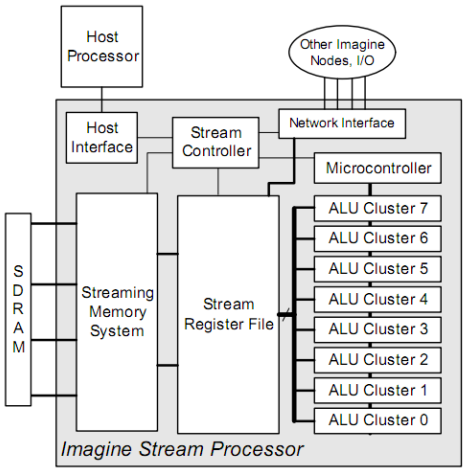

Fig. 3. The Imagine stream processor architecture [7]

Imagine exploits data-level parallelism (DLP) through SIMD execution of a kernel on all eight arithmetic clusters, one stream element at each one. Instruction level parallelism (ILP) is exploited by using VLIW instructions, and task-level parallelism (TLP) is achieved by overlapping stream data memory accesses and I/O operations with kernel execution.

The bandwidth hierarchy employed in Imagine manages to capture the locality in the stream programming model. The first hierarchy level captures the locality within kernels by passing and storing intermediate results in the LRFs. The producer-consumer relations between kernels are captured by the second level of hierarchy which is the SRF. Streams generated during consecutive kernels are stored in the SRF eliminating the need to access the external memory. Also the SRF captures the locality of temporary streams. Requests to off-chip DRAM are left for the truly global data.

The SRF allows for 22 read/write access to all 22 stream clients simultaneously, and has a total capacity of 128 KB. Within each cluster there are 15 LRFs. These LRFs contain kernel constants, parameters and local variables. On each cluster there are four 32-entry and thirteen 16-entry LRFs for a total of 272 words per cluster and 2176 words across all 8 clusters. Running at 500 MHz, the SDRAM can transfer data to the processor's streaming memory unit at a rate of about 2.67 GB/s. The SRF can communicate with the ALU clusters at a rate of 32 GB/s, nearly an order of magnitude faster than transfer rates with SDRAM. The ALUs can directly forward their results amongst each other using the Local Register Files, which provide a bandwidth of about 544 GB/s; another order of magnitude faster than the bandwidth of the previous tier.

The stream controller is the final component of Imagine. It stores each instruction in a scoreboard and issues them to the proper module as soon as its dependencies are resolved. Hardware source dependencies are resolved at runtime. Data dependencies are encoded statically.

2) Imagine stream ISA

All memory references in the Imagine are made using 'stream load' and 'stream store' instructions. This is done to simplify the architecture by optimizing for stream accesses instead of individual accesses. This increases throughput and simplifies programming of the Imagine. Communication is performed with 'send' and 'receive' instructions. Imagine's stream ISA consists of the following stream instructions [8]:

- *Load*: Used for external memory communication. It transfers streams from the external DRAM to the SRF.
- *Store*: Used for external memory communication. It transfers streams from the SRF to the external DRAM.
- *Receive*: Used for communication with other I/O devices. Transfers streams from I/O to the SRF.
- *Send*: Used for communication with other I/O devices. Transfers streams from the SRF to an I/O.
- *Cluster Op*: Used for kernel execution. The input streams are first read from the SRF, then there elements are processed and the resulting output streams are stored back to the SRF.
- *Load microcode*: Loads streams consisting of kernel microcode (VLIW instructions) from the SRF into the micro-controller.

3) Performance

To evaluate the performance of the Imagine stream processor three benchmark sets were used [9]. These benchmarks include applications such as MPEG-2 encoding, polygon rendering, depth extraction, and Fast Fourier Transform, a representative workload of today's popular media applications. The first set was made out of synthetic benchmarks aimed at finding the actual peak performance for each Imagine subsystem and validating the theoretical peak performance. The second a set of media processing kernels are used to explore how the performance is affected with various kernel characteristics. The final benchmark set tracks how the Imagine performance is affected by the stream length (# of elements in stream). The results indicate that Imagine sustains about 43% of its theoretical peak performance on the first set and between 16%-60% on the second set. The performance of Imagine is affected by time spent to kernel prologues/epilogues, unused slots in the VLIW instructions, and load imbalance amongst the functional units. Performance metrics on selected kernels and applications are shown in Table 2. The benchmark results reveal that the Imagine processor and the stream paradigm are well suited for multimedia applications because of the inherent parallelism of such applications.

TABLE 2: PERFORMANCE METRICS FOR IMAGINE FOR MEDIA APPLICATIONS AND KERNELS

Applications	Performance (GOPS)	
	400 MHz	200 MHz
Depth Extraction	*11,92*	*4,91*
MPEG-2 Encoding	*15,35*	*7,36*
Polygon Rendering	*5,91*	*N/A*
QR Decomposition	*10,46*	*4,81*
7x7 Convolution (Kernel)	*52,6*	*9,76*
2D Discrete Cosine Transform (Kernel)	*22,6*	*6,92*

B. Merrimac

Merrimac [10] is a stream inspired supercomputer that uses stream architecture and advanced interconnection networks to provide up to an order of magnitude more performance than traditional cluster-based supercomputers. Merrimac aims at exploiting the advantages of the stream programming model and stream processor architectures to boost the performance of scientific applications. The main components that comprise Merrimac are illustrated in Fig. 4.

1) Architecture

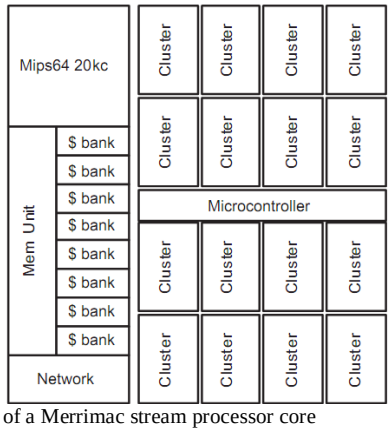

Fig. 4. Floorplan of a Merrimac stream processor core

The core element of Merrimac is a stream processor core. The stream processor can act as a standalone coprocessor to a general purpose processor, or can be interconnected with other identical stream cores via a Network-on-Chip to form a streaming supercomputer. A single stream processing core contains a total of 16 arithmetic clusters. Within each cluster there are 4 FPUs a SRF bank and a LRF used by the FPUs.

As with Imagine a host processor is responsible for fetching all instructions and dispatching any stream instructions to the microcontroller, which will eventually dispatch them to the clusters, and stream memory references to the stream memory hierarchy.

The Merrimac board system has a total of 16 stream processing cores interconnected via high radix NoC. Each core has a network interface which directs off-chip memory references to the routers. Each processor communicates with a 2GByte DRAM and four router chips. The routers interconnect the 16 processors on the board, providing flat memory bandwidth of 20GBytes/s per node. Routers also act as gateways for external board references when more than one board are used, each router can sustain a bandwidth of 5GBytes/s per node.

2) Performance

Merrimac's performance was evaluated with three typical scientific applications were converted to stream programs and executed on a cycle-accurate Merrimac simulation model. The performance over the three applications ranged from 11.4 GFLOPS to 33.5 GFLOPS. The FPUs utilization ranges from 7 to 50 floating point operations per global memory access. The majority of memory references are satisfied by the first level of hierarchy which is the LRF. Only a small portion of the total memory references, about 1%, require access to the external memory. The performance measures suggest that stream processing architectures are well suited for scientific applications.

3) Fault Tolerance on Merrimac

Fault detection techniques for soft-error fault tolerance were applied on Merrimac as an example of a compute intensive architecture [11]. A hybrid fault tolerant model is proposed where software or hardware fault tolerant techniques can be used to detect soft-errors. This model uses hardware techniques to conserve off-chip bandwidth and on-chip storage, and a combination of software-hardware (Instruction Replication, Kernel Re-execution, Mirrored Clusters) techniques for redundancy of computational resources. The high amount pf logic found in compute intensive architectures such as Merrimac makes hardware replication very costly.

C. FT64 Stream Processor

FT64 (Fei Teng 64) is another stream processor that is primarily targeting scientific applications [12]. FT64's instruction set architecture is optimized and fine tuned to provide best performance for scientific applications. It acts as a coprocessor to a host general purpose processor. A novel stream programming language, SF95 (Stream FORTRAN95), and its compiler, SF95Compiler (Stream FORTRAN95 Compiler), were also developed to facilitate the development of scientific applications for FT64.

1) Architecture

FT64 is mainly composed of a stream controller (SC), a stream register file (SRF), a micro controller (UC), four ALU clusters, a stream memory controller (SMC), a DDR memory controller (DDRMC), a host interface (HI) and a network interface (NI). The architecture of FT64 is shown in Fig. 5.

The stream controller is responsible for issuing and sequencing of the stream instructions received from the host processor. Its main components are an interface with the host, a scoreboard and an instruction issuing unit. The issuing unit can issue an instruction every 9 cycles.

The microcontroller is responsible for the instruction fetch and decode stages of the processor, and controls the kernel execution on the 4 ALU clusters in a SIMD fashion. The microcontroller stores VLIW instructions which then issues to the clusters to proceed with their execution.

The four ALU clusters can be viewed as the read operands and execute stage of the FT64 stream processor. The clusters communicate with each other through an inter-cluster network. Each cluster is composed of eight functional units an inter-cluster network and eight local register files. Each of the

eight functional units consists of four multiply-accumulate units, a divide/square (DSQ) unit, a communication unit, and a scratch pad memory.

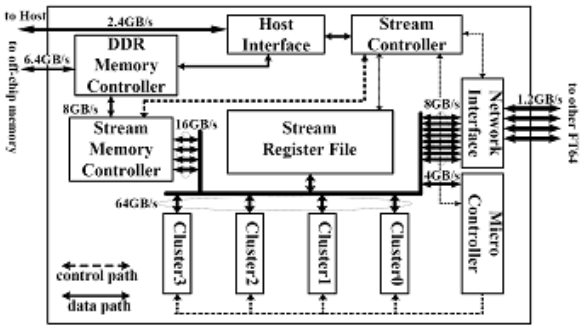

Fig. 5. FT64 top level block diagram

The memory hierarchy of FT64 follows that of other stream processors. It consists of the three levels the SRF, LRF and an external memory. The LRF are distributed amongst the ALU clusters, having a total capacity of 19KBs and a bandwidth of 544GB/s. The SRF buffers derived streams and kernel-level programs and is distributed into 2068 blocks having a total capacity of 256KB. The total bandwidth provided by the SRF reached 64GB/s. The external memory is used to store and load streams. It is controlled by both the stream memory controller and the DDR memory controller which cooperate to handle all transfer requests. The stream memory controller receives access instructions from the stream controller and sends read/write requests to the DDR memory controller which handles the memory access. The SMC has an 8GB/s bandwidth with the DDR memory controller while the DDR memory controller communicates with the external DDR with a bandwidth of 6.4GB/s.

The network interface provides a means for combining two or more FT64s into a powerful stream supercomputer. The communication between the stream processors is done by partitioning a stream into 32-bit packets. To facilitate the stream transfer between processors the ISA is enriched with network stream instructions for sending and receiving streams. The total bandwidth capable by the network interface is 1.2GB/s.

2) Instruction Set Architecture

FT64's ISA is comprised of 11 stream level instructions and 140 kernel-level instructions. The design of the kernel level instructions focuses on scientific computing with support for double-support floating-point calculations. The ISA stream instruction types are illustrated in Table 3.

TABLE 3: STREAM LEVEL INSTRUCTIONS OF FT64

Stream Instructions	Operation
MOVE	Move the data in the source register to the destination register
WRITE_IMM	Write a immediate value to the destination register
BARRIER	Insure the execution order of the instructions
RESET	Reset FT64
MEMOP	Transfer a stream between DRAM and SRF, addressing modes including sequential, strided, indexed, and bit-reversed
LOAD_UCODE	Load the micro code from SRF to UC storage
CLUSTOP	Start the execution of the kernel on the clusters
CLUSTER_RESTART	Restart the kernel with the input or output stream
SYNCH_UC	Synchronize the execution of the clusters
NETOP	Transfer a stream between the local SRF and remote SRF
NET_RESTART	Restart a stream to continue the network transfer

3) Performance

Nine typical scientific application kernels are tested on the FT64 and the performance for all the applications is compared to the Itanium 2, a very popular VLIW general purpose processor. The speedup achieved by using FT64 is illustrated in Table 4. In seven out of nine applications the stream processor outperforms Itanium 2, and even for the other two applications the difference between them is not large.

TABLE 4: COMPARISON OF PERFORMANCE BETWEEN FT64 AND ITANIUM 2

Test Applications	Performance Speedup FT-64/ Itanium 2
Swim	1.04
EP	2.55
MG	1.35
CG	0.10
FFT	8.01
Laplace	2.41
Jacobi	1.04
GEMM	1.97
NLAG-5	0.97

D. Storm-1 Stream Processor

A spinoff company named Stream Processors Inc. was started from the Imagine project. The company`s current flagship product is the Storm-1 stream processor. The storm-1 chips are aimed at replacing more expensive and hard to design ASIC and FPGA DSP solutions. Targeted applications for Storm-1 include video surveillance, video conferencing and multi-function printers.

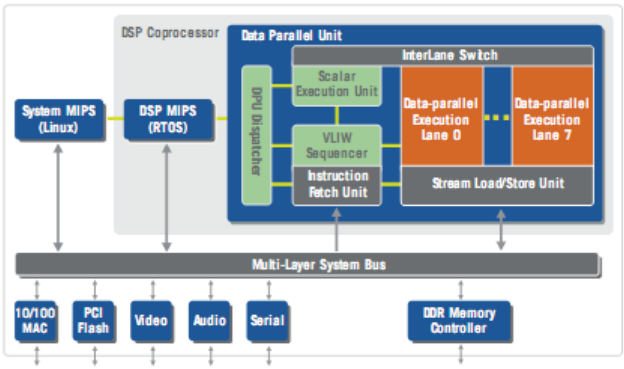

Fig. 6. Storm-1 Processor block diagram

The Storm-1 SP8LP-G220 [13] processor has 16 clusters and 5 individual ALU units per cluster. A detailed block diagram is illustrated in Fig. 6. The ALUs in each cluster are capable of 4x8/2x16/1x32-bit operations per cycle, including MACs. The stream processors performance can reach up to 30 16-bit GOPS. The stream registry file's capacity is 128 KB (16 KB per cluster) and the local registry file is 9.5 KB (304

words per cluster).A dedicated 96 KB of memory are reserved for VLIW instructions.

E. Stream Processors and Other Architectures

1) Stream Processors Vs Vector Processors

Stream processors share a lot of similarities with vector processors such as their ability to expose data parallelism by operating on large amount of data, and hiding memory latencies with overlapping instruction executions [4]. However, stream processors extend the capabilities of vector processors in the following two ways. First, stream processors have an additional layer of memory hierarchy by splitting the vector register file into the local register file and stream register file. This allows for the LRFs provide high aggregate bandwidth to support a large number of ALUs, and the SRF to be made large enough to provide coarse grain locality. Second, a stream processor executes all kernel operations on one stream element before moving to the next, thus, reducing the intermediate number of variables that need to be stored in the LRFs. These two enhancements can reduce memory bandwidth demands and thus can support more ALUs.

2) Stream Processors Vs DSPs

Although stream processors relate and may be considered as a DSP, they have significant differences with traditional DSP architectures. Stream processors are highly parallel and keep 1,000's of operations in flight every cycle. This happens primarily because of the differences in the memory hierarchy of the two processor architectures. Traditional DSPs rely on large caches to reduce memory latencies; additionally DSPs need the programmer to manage direct memory access. In contrast stream processors use compiler-managed distributed memory hierarchy with background data movement and execution scheduling. Furthermore, the bandwidth hierarchy of stream processors hides long memory latencies and minimizes requests to the external memory.

F. Scalability and Future of Stream Processors

Stream Processors are emerging as a popular solution for high performance parallel computing. They are considered more area and power efficient than traditional architecture processors for applications that exhibit high degree of parallelism. Stream processors are a promising architecture for parallel applications, thus, it is worth considering how they will scale with future technologies. This was the focus of the work done in [14] where the scalability of stream processors was explored by looking into intra-cluster and inter-cluster scaling. The former technique was based on increasing the number of ALUs per cluster, while the latter on increasing the number of ALU clusters. The performance for these scaling techniques was also studied on a set of multimedia applications.

Intra-cluster was found to be effective in terms of cost and performance for up to 10 ALUs per cluster, and in terms of area and energy for up to 5 ALUs per cluster. Inter-cluster scaling was found to be effective for up to 128 clusters with only a slight decrease in terms of energy and area. By combining the two techniques a 640 ALU stream processor with 5 ALUs in each of the 128 cluster is feasible in the 45nm technology. Such a processor would be able to sustain about 300 GOPS with only 2% degradation in area and 7% degradation in energy per ALU, compared to a conservative 40 ALU stream processor.

Stream processors close the gap between special and general-purpose processors without sacrificing programmability [14]. With competitive energy efficiency, lower recurring costs compared to ASIC solutions, and with advantages in flexibility compared to FPGAs, stream processors are expected to become an attractive solution for signal processing applications. The four dedicated stream processors presented in section III are summarized in Table 5.

TABLE 5: Summary of stream processors specifications and performance
N/A = Not available data

	Imagine [7]	**Merrimac [10]**	**Storm-1 [13]**	**FT-64 [12]**
Local Register File				
- Bandwidth	544 GB/s	2,560 GB/s	*N/A*	544 GB/s
- Capacity	17,4 KB	6,144 KB per cluster	19 KB	19 KB
Stream Register File				
- Bandwidth	32 GB/s	512 GB/s	*N/A*	64GB/s
- Capacity	128 KB	1 MB, 128 KB per cluster	256 KB, 16 KB per cluster	256 KB
ALU clusters				
# ALUs per cluster	6 32-bit	4 64-bit	5 32-bit	8 64-bit
# Clusters	8	16	8	4
Functional Unit Types	3 ALUs, 2 Multipliers, 1 divide/square-root unit	4 Floating Point MADD units	5 MAC units	4 floating point MACs, 1 DSQ unit, & other communication modules
Performance				
Average Performance	6-15 GOPS	11,4 – 32,2 GFLOPS	224 8x8 bit GMACS	1-9 GFLOPS
Frequency	400MHz	N/A	700 MHz	500 MHz

G. Stream Processing Inspired Architectures – Cell BE

A similar architecture to that of stream processors can be found in the Cell Broadband Engine [15]. The Cell Broadband Engine was developed by IBM, Sony and Toshiba. The objective was to design a processor with reconfigurable elements to adapt to, and excel, at a variety of multimedia applications. Cell can perform stream processing to provide improved performance, with the necessary software support. Cell's main components are 8 SPEs (Synergistic Processing Elements) which are controlled by one PPE (Power Processing Element). Each of these SPEs is assigned an operation (Kernel) by the PPE, and are connected on a bidirectional ring bus to allow data vectors (streams) to be run through multiple SPEs connected in series.

The Cell BE exhibits many similarities to the general architecture of a stream processor. Cell's SPEs are similar to the ALU clusters of a stream processor. Each of the SPEs has its own register file just as a stream processor has its own LRF. Furthermore, each SPE contains a 256 KB local storage unit that allows for quick local accesses, similar to the local

scratch pads in the clusters of most stream processors. The SPEs also have a memory flow controller that behaves much like the inter-cluster communication unit that is found in stream processors.

However, there are differences between Cell and the architecture of a stream processor. The SPEs are each full processors, so they may each execute a different kernel, and kernel operations can be chained together by connecting the SPEs together, unlike stream processors, where the same kernel (VLIW instructions) is executed across all clusters.

IV. Graphic Processing Units

A graphics processing unit (GPU) is a specialized processor that offloads 3D graphics rendering workload from the microprocessor. It is equipped with a large number of special hardware for mathematical operations and fast floating point operations. A GPU is optimized for 3D graphics processing and more recently for video processing. The primary goal of GPUs is to provide high-performance for rich 3D experience. Consumer demands for videogames, advances in technology, and the potential for increased performance of the inherit parallelism in the feed-forward graphics pipeline paved the way for the development of this specialized processor.

A. The Graphics Pipeline

The task of any 3D graphics processing system is to synthesize an image from a description of a scene, under some time constraints in order to achieve real time performance [16]. The process of synthesizing the image is traditionally implemented as a pipeline of specialized stages called the *graphics pipeline*. This is an efficient way of implementing the image synthesis procedure primarily because of the producer-consumer relationships that are present between stages.

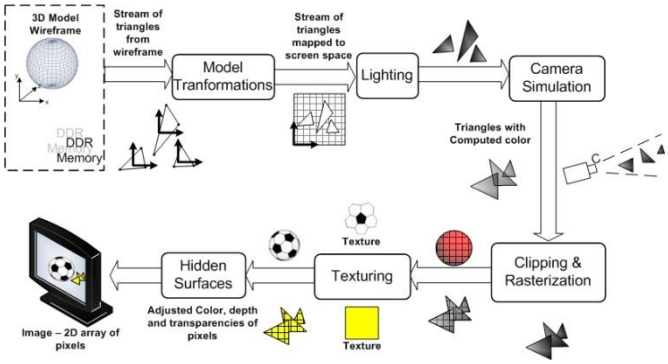

Fig. 7. GPU graphics pipeline abstraction: 1) Transformations: 2) Lighting: 3) Camera View: 4) Rasterization: 5) Texturing: 6) Pixel Operations:

The input to the graphics pipeline is typically a representation of a 3D model or object in the form of a wireframe consisting of a set of triangles. The more triangles that are used to represent the 3D model, the higher the quality, however, the processing time required by the GPU increases. Each triangle is assembled by vertices, and each vertex has information such as its coordinates (*x*, *y*, and *z*) and color. A 3D model scene is forwarded to the graphics pipeline one vertex at a time. Different graphics pipeline implementations may have different stages and operations depending on the targeted quality and performance. However, some fundamental stages are implemented by all graphics pipelines. These steps are shown in the graphics pipeline in Fig. 7 and can be summarized as follows [16]:

1) Model Transformations

Before the main computation can begin the GPU must transform all objects in a common coordinate system using operations such as translations, rotations or scaling. Usually this common coordinate system is the screen space, and is done on per vertex manner. The entire hierarchy of transformations can be done with a single matrix-vector multiply. The output of this stage is a stream of vertices expressed in the same 2D coordinate system.

2) Lighting

The GPU computes each vertex`s lighting properties based on the scene lighting conditions. This is done by considering both the camera position and the position of the light source. Multiple lights are handled by summing the contribution of each individual light.

3) Camera Simulation – Viewing Transformation

In this stage the vertices are set up back into triangles. Each color 3D triangle is projected onto a virtual camera`s film plane. The output of this procedure is stream of triangles ready to be converted into pixels.

4) Clipping & Rasterization

The *clipping* procedure involves discarding vertices and triangles that are not visible to the camera space because they are occluded by pixels with smaller depth values. The procedure of checking the depth of an object is called *Z-testing*. Rasterization is the process in which the 2D scene is converted into a raster format and *fragments* are created from triangles. Fragments can be considered as candidate pixel values and there may be multiple fragments per pixel location. In the case where multiple triangles overlap a fragment, then its final color is the interpolated value of the overlapping triangle's color. Since each fragment can be treated independently this stage of the pipeline has been made extremely parallel over the years.

5) Texturing

To add more realism to the scenes 1D, 2D or 3D images called textures [17] are mapped over the geometry of an object. Textures are stored in high speed memories which must be accessed by each pixel on order to determine or modify its color. More than one access to a texture may be required to adjust the visual properties of textures when they appear either smaller or larger than their native size. In order to hide latencies of accessing the texture memories specialized cache designs are utilized.

6) Hidden Surfaces & Pixel Operations

In this final pipeline stage the contributions of each pixel is calculated according to its screen position. Also in this stage the visibility of an object is determined. This is done by examining the depth (Z-test) of each object with respect to the viewer. In the case of overlapping pixels the pixel which has the smaller distance from the viewer is the one that is written on the display.

Some additional graphics pipeline operations include:

- *Culling*: The Process of removing part of an object (a group of triangles) based on a set of criteria (depth, transparency).
- *Alpha Blending*: The process of combining the primitives of two objects in order to create the appearance of partial transparency.

B. Graphics APIs

Graphics APIs assist in the development of Graphics applications. There is a close relationship between the Graphics API and the underlying hardware implementation, as the hardware is an implementation of the API specified pipeline. Hence, the API can be viewed as a hardware abstraction level (HAL), which provides an efficient way for developers to take advantage of the hardware of any specific graphics card. However, as general purpose computing on GPUs becomes more prominent, the next generation of APIs starts to evolve beyond the 3D graphics pipeline into a much more general programming interface [1]. There are two widely adopted APIs for writing graphics rendering applications for the GPU, OpenGL and DirextX. OpenGL is a cross-language, cross-platform open standard API. DirectX is developed my Microsoft and is part of its Windows operating systems.

C. The evolution of GPUs

Through the history of the GPUs the basic graphics pipeline stages have evolved from purely hardwired implementations to processor-like programmable unit implementations that have revolutionized the way GPUs are used. One such evolution was the introduction of programmable processors given below [16]:

- *Vertex processor:* Their purpose is to transform each input vertices individual coordinates to a global coordinate system. They also determine the depth value of each vertex.
- *Geometry processor:* They add/remove vertices from a 3D mesh and are responsible for geometric transformations on the input vertices/triangles.
- *Pixel/Fragment processor:* They are responsible for calculating the color of each individual pixel as well as to add effect such as lighting, shading and toning.

These programmable processors replaced most of the fixed-function stages in the graphics pipeline. These processors could be programmed with small programs called *shader* programs. Hence we have vertex, geometry, and pixel shaders, a type of program for each processor. The introduction of this *programmable shader model* allowed some GPU parts to be programmed according to a specified instruction set. The three programmable processors were responsible for implementing a specific part of the graphics pipeline, thus, each one had its own instruction set and each required its own shader program. Additionally, each programmer had to express the problem in terms what the specific shader could process (vertices, texture, etc.). Managing three different programmable engines, however, presented a great obstacle towards the efficient programming of GPUs. Consequently, GPU developers realized that more emphasis had to be given on increasing not only the GPU programmability but the ease with which the programmer can program a GPU. As a result the three different shader types were merged into one *unified shader model*. The *unified shader model* introduced a consistent instruction set across all three processor types, so each of the programmable processors could perform most of the graphics pipeline related tasks depending on the shader program it is running. Hence, the programmer had a large pool of programmable shader to work with, and could adjust each processor's "job" according to the application demands. Additionally dynamic load balancing/scheduling algorithms could be used to dynamically adjust the work amongst the processors and the distribution of each shader program, to achieve a much better utilization of the available resources. Vertex, geometry, and pixel shaders became threads, running different programs on the programmable cores. These cores are massively parallel programmable units called *stream processing cores*. The ATI Xenos chip[16] for Xbox360 was the first to introduce this concept. This new architecture provided one large pool of programmable floating-point processors where each could be programmed to perform a certain task from the graphics pipeline. As a result it was now possible to have a much better utilization of the processing resources and load balancing with the application being able to dynamically change the role of each processor, thus, adjusting to the demands of the current task. It is the flexibility offered by these programmable processors and their increased computational power, and floating point support that has turned attention to GPUs for general purpose computing.

D. GPU Architectural Features

1) Fixed Function Processing

Fixed-Function hardware is used to accelerate common 3D pipeline tasks that have an abundance of parallelism. These tasks are texturing and rasterization. For high quality images texturing is expensive and thus handled by fixed function hardware. Interpolation and sampling are the main operations of the rasterization phase which are very demanding even with optimization methods and thus benefit from fixed-point implementation [17].

2) Memory System

GPUs are optimized for high throughput rather than latency contrary to CPUs. Thus, a different approach is needed when implementing the GPU memory system. GPU memory systems must deliver high bandwidth through wide memory busses and specialized Graphics DDR. GPUs employ small read-only caches that help in reducing the bandwidth requirements on the main memory. Also GPUs benefit from small caches that capture spatial locality. A the number of programmable processors increases significant amounts of on-chip storage are used to hold execution context, stream data, and temporary data [17].

3) Pipeline Schedule and control

GPUs use hardware logic for mapping and scheduling of computation onto chip processing resources. Hardware scheduling logic handles assignment of computational resources to threads, keep track of ordered processing, and discard unnecessary pipeline work. Additionally thread management is performed entirely in hardware [17].

E. GPU Examples

Following is a description of two GPU families the ATI Radeon X800 and the NVIDIA GeForce 6, both of the same generation. Also Tables 6 and 7 provide an historical timetable for the two main GPU families from NVIDIA and ATI respectively, along with the main features of each generation. [18, 19].

TABLE 6: NVIDIA`s GeForce GPU Series

GPU Series	Important Remarks
GeForce 256 (August 1999)	- First PC graphics chip with hardware implementations of transform, lighting, and shading operations - 220 nm fabrication process
GeForce2 (April 2000)	- Twin texture processor per pipeline design - Doubling texture fillrate per clock compared to GeForce 256 - 180 nm fabrication process
GeForce3 (February 2001)	- First programmable Pixel and Vertex Shaders - 180 nm fabrication process
GeForce4 (February 2002)	- Enhancements to anti-aliasing capabilities - Improved memory controller - 150 nm fabrication process
GeForce FX (2003)	- New Shader Model 2 specification - Optimized version of the GeForce 3 - 130 nm fabrication process
GeForce 6 (April 2004)	- Shader Model 3.0 - High dynamic range imaging - Scalable Link Interface - PureVideo capability (Hardware support for video processing) - 130,110 and 90 nm fabrication process
GeForce 7 (June 2005)	- Last graphics card designed for AGP bus - Widened pipeline and an increase in clock speed - Mostly 90 nm fabrication process
GeForce 8 (November 2006)	- First ever GPU to fully support Direct3D 10 - 80 nm fabrication process - First GPU with fully unified shader architecture - Introduction of CUDA - Introduce single instruction multiple thread (SIMT) execution model
GeForce 9 (February 2008)	- Revisions to existing late 8-series products - Two G92 GPU cores - Two separate 256-bit memory busses one for each GPU - 65 nm fabrication process - Support for CUDA
GeForce 100 (March 2009)	- Revisions to 9 series parts
GeForce 200 (16 June 2008)	- 1.4 billion transistors - New GT200 core - 65 nm fabrication process - Largest commercial GPU ever constructed - 576mm^2 die surface area - Largest CMOS-logic chip that has been fabricated at the TSMC foundry - Support for CUDA - Added Double Precision Floating Support for Scientific Applications
GeForce 300 – Fermi (End of 2009)	- First NVIDAI chip with DirectX 11 support - Shader Model 5.0 - GDDR5 memory support - 45 nm fabrication process

TABLE 7: ATI`s Radeon GPU Series

GPU Series	Important Remarks
R100 (2000)	- New Hyper-Z technology - Limited pixel shader programmability - 180 nm low-k fabrication process
R200 (2001)	- ATI's first programmable pixel and vertex shader architecture - More advanced pixel shader 1.4. - 180 nm low-k fabrication process
R300 (2002)	- First fully Direct3D 9 capable consumer graphics chip - Doubling of transistor count, clock rate improvements - Mostly 150 nm fabrication process
R420 (2004)	- Extensions to the Shader Model 2 feature-set. - Increased shader programmability - 130 nm low-k fabrication process - Used GDDR3
R520 (2005)	- Complete Shader Model 3 support - Enhancements including the floating point render target with anti-aliasing - Ultra Threaded Dispatch Processor - 90 nm fabrication process
R600 (2006)	- Direct3D 10.0 support - Based on a VLIW unified shader architecture - Programmable tessellation units - Shader Model 4.0 (Unified shader model) - 80,65 and 55 nm fabrication process
R700 (2008)	- First GPU to support GDDR5 - 40 and 55 nm fabrication process - Increase in Stream Processing cores
Evergreen (2009)	- 40 nm fabrication process - 1400 stream processors

1) ATI Radeon X800series

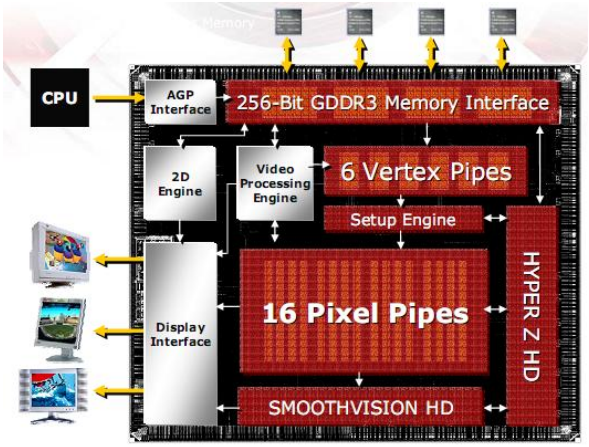

Fig. 8. Radeon X800 Block Diagram [20]

The Radeon X800 [20] series is the high-end graphics card solutions from the R420 family of GPUs developed by ATI in 2004. The Radeon X800 (Fig. 8) series chips were fabricated using 130nm low-k fabrication process. The vertex processing engine employed in the X800 series consists of six vertex programmable processors each of which has 2 ALUs (a 128-bit floating point vector ALU and a 32-bit scalar ALU), and control flow logic. The floating point ALU in each vertex processor is designed to perform transformation operations on the input vertices position data (coordinates and perspective). The scalar ALU is needed after the vertex shader program has finished, for removing the triangles, or a part of them, that are outside of the viewing window (or by any other criterion). Once all the vertex operations are done the Setup Engine

reassembles the vertices into triangles, lines or points (the 3 primitive forms). It also assigns parameters to the newly formed triangles such as color, depth, or texture coordinates.

The next step is to convert the triangles in to pixels. This is done through the Pixel Processing Engine. There are 16 independent pixel processing pipelines in the pixel processing engine each with its own pixel and texture processing unit and are separated into 4 groups call quad pipelines (each with 4 pixel & texture processors). Each pixel processor is programmable and controls the operations performed on each pixel. Each of the quad pipelines receives triangle data from the setup engine and performs an early hierarchical Z-test to discard the non-visible pixels of the triangle. Pixels that were not discarded by this procedure are captured later during Z-testing in the Hyper-Z HD module.

Rasterization takes place next, where for each pixel Z-tests, anti-aliasing and color value calculations take place. When a frame is complete it is sent to the frame buffer where they are either used in subsequent rendering procedures or are sent to the display engine for output.

The Radeon X800 memory system incorporates a 256-bit GDDR2 memory interface capable of providing up to 32 GB/s of bandwidth.

2) NVIDIA GeForce 6 Series

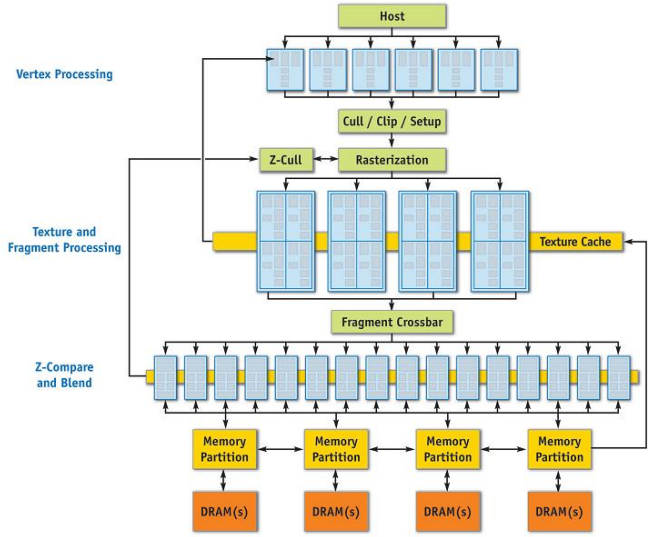

Fig. 9. Block Diagram of the GeForce 6 Series Architecture [21]

The GeForce 6 architecture [21] was developed at the same time as X800, around 2004. Fig 9. illustrates a high level block diagram for the NVIDIA GeForce 6 series GPUs. The GeForce 6 receives commands, texture and vertex data from the CPU via the host interface. A vertex fetch unit is used to read the vertices referenced by the CPU commands. The vertex programmable processors allow for a shader program to be applied to each of the vertices in the object performing transformations, skinning and other operations. Vertex shader programs can fetch texture data via a texture cache which is shared with the pixel processors. Additionally, there is also a vertex cache that stores vertex data found for both pre and post vertex processing stages. All vertices are then grouped into primitives (lines, triangles, points). The Cull/Clip/Setup unit then performs operations on these primitives to prepare them for the rasterization unit. The rasterization block that follows uses the z-cull block to discard pixels that are occluded by objects closer to the viewpoint. The fragment processor is used for pixel and texture processing. A shader program is applied to each of the pixels in order to determine specific pixel properties such as color and depth. The fragment processor operates on squares of four pixels called quads. Moreover, it executes the same instruction for hundreds of pixels in an SIMD fashion. A texture unit is used to fetch texture data from memory with an option of filtering the data before it returns it to the pixel processor.

Pixels leave the pixel processor and move on to the Z-compare and blend unit. There according to their depth they are either assigned a final color and are passed to the frame buffer (contains the output frame), or are discarded if they are hidden by closer pixels.

The memory system is partitioned into four banks each with its own DRAM. All rendered surfaces are stored in the DRAM, while textures and input data can be stored either in the DRAMs or system memory. With the four independent DRAMs the GPU is afforded a wide 256 bit bus, allowing for bandwidth up to 35 GB/s.

GeForce 6 series GPUs support operations in 16-bit and 32-bit floating point format. High-end GeForce 6 GPUs have up to six vertex processors whereas lower-end GPUs have two.

V. Stream Processing and General Purpose Computation on Graphics Processing Units

The raw computational power offered by modern GPUs and the need to use it in computationally demanding applications lead to the evolution of a GPU to a general purpose computational engine. The evolution of the graphics pipeline from fixed-hardware implementation into a grid of programmable processors enabled many researchers to port their algorithms and applications on the GPU.

A. The GPU as a Stream Processor

Focusing on the demands for general purpose computing modern GPUs employ grids of massively parallel programmable stream processing cores [1] that are efficient for high performance computing, while using additional fixed function hardware for graphics processing. The incorporation of stream processing in GPUs has expanded their application span and also made it easier to program them.

Expressing the graphics pipeline as a stream program is rather a natural progression because of its inherit localities and parallelism that make it a very good match for the stream processing model (Fig. 10) [2]. Traditionally the graphics pipeline is implemented as a series of stages with data flowing between them in a specific manner. This is pretty similar to how kernels communicate and exchange information using streams. Furthermore, the producer-consumer locality found in the stream processing model is also present in the graphics pipeline with the output of one stage being the input to following stages. Finally, the computations performed by each pipeline stage are similar for a across a large amount of data (let that be vertices, triangles or pixels); hence, each stage can be mapped to an individual kernel.

However, there are some differences between a stream program executed on the graphics pipeline and one executed on a general purpose stream processor. First, in the graphics pipeline every element of a stream is processed by the same units whereas in a stream processor elements of the same stream are executed on different clusters. Second, a stream program on the graphics card has access to much smaller memory than on a general stream processor. A processing unit in a graphics card has access to only a few fast registers even though graphic cards have a lot of memory.

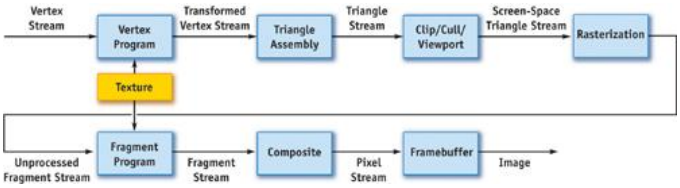

Fig. 10. Mapping the Graphics Pipeline to the Stream Programming Model [2].

Some examples of general stream computations performed on GPUs include: solving partial differential equations, motion physics simulation and general distance field calculations. None of these applications, however, make use of the extensive features offered by the vertex and pixels shaders and only use basic operations to perform their calculations.

A dedicated GPU stream processor example is the AMD FireStream [22] developed by ATI which targets the stream processing/GPGPU domain of applications. The latest generation of these specialized stream processors provides a computational power of up to 1.2 TFLOPs with a 2 GB of GDDR5 memory.

B. Graphics Processing Unit Architectures for General Purpose Computing

The graphics processing unit has become an integral part of today's mainstream computing systems. Over the past six years, there has been an increase in the performance and capabilities of GPUs. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outperforms its CPU counterpart. 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. This effort in general-purpose computing on the GPU (GPGPU) [1], has positioned the GPU as a compelling alternative to traditional microprocessors in future high-performance computer (HPC) systems.

The main reason for this shift is the increased programmability, computational power and precision of modern GPUs. There is an expectation amongst the research community that the use of GPUs can potentially offer speedups of 5 up to 20 times, compared to the latest generation of CPUs [5]. It must be noted, however, that not all applications benefit from parallel hardware. Even though GPU & CPU architectures are both converging towards a highly parallel multicore platform, there is always a need for a processor that is optimized for fast single thread execution. Applications with intensive control flow, and unpredictable memory access patterns are not at all suited for parallel architectures. As a result the architectures and concepts found in high performance single threaded CPUs will continue to be present in future architectures.

The latest generation of GPUs from all major Graphics Processor manufacturers namely Intel, NVIDIA, and AMD reflects the emphasis on enhanced GPU programmability and enhanced computational power for general purpose computing in order to allow for general purpose applications to harvest the computational power of GPUs. Current GPU architectures are highly optimized for HPC applications with features that may not necessarily needed for 3D graphics processing such as 64-bit floating point arithmetic and flow control units. Nonetheless, these GPUs still have specialized hardware that is only used for 3D graphics processing. The rest of this section provides a description about the latest efforts from Intel, NVIDIA and AMD on the realization of GPUs with enhanced features for HPC applications.

1) Larrabee

Larrabee [23] is Intel's codename for a future graphics processing architecture which is based on the x86 architecture. It is a multicore architecture that aims to be a high-end GPU on general purpose platform. Due to its x86. This makes it better suited for HPC applications rather than a more traditional based GPU architecture. Fig. 11 (a) shows the architecture of Larrabee illustrating the ring network and CPU cores.

Larrabee significantly differs from current GPU architectures because instead of using graphics optimized hardware it uses enhanced general purpose architecture for most of the graphics processing. This gives Larrabee an advantage as it is more suitable for general purpose computing. Larrabee includes fixed function logic for texture filtering only. Rasterization, interpolation, and alpha blending that are traditionally implemented in hardware in GPUs designs, are implemented in software in Larrabee. The software implementation of these parts allows for optimizations and additions according to the applications requirements. The software rendering pipeline in Larrabee consists of the following steps. First each primitive set is assigned to a core, which performs operations such as vertex shading, geometry shading, and culling. After rasterization sets of pixels are formed which are assigned to a single core for the latter stages of the rendering process, which includes depth, stencil and blending operations.

The CPU core (Fig. 11 (b)) of Larrabee has two main components, a scalar unit and a vector processing unit each with its own register file. The scalar unit is an in-order, dual-issue Pentium processor with enhanced features such as multi-threading, 64-bit extensions, and prefetching, with task and instruction scheduling arranged by the compiler. Additionally, it supports additional features such as page faults, subroutines, and cache coherence mechanisms. The 512-bit vector processing unit is responsible for handling integer and floating point instructions. It operates in SIMD fashion and is fully programmable. The vector unit supports a variety of instructions from standard to complex mathematical operations such as fused multiply-add, and logical operations.

There is also support for gather and scatter memory addressing and predication for all vector instructions. Larrabee uses two level cache hierarchy. A dedicated L1 cache per core and a shared coherent L2 cache with each core having its own L2 cache subset. The instruction set of Larrabee includes Instructions for explicit cache control and explicit prefetching.

The communication between the cores, L2 caches and any other component is facilitated by a 512-bit wide (per direction) bi-directional ring network. For more than 16 cores multiple short linked rings are used. The ring network provides a platform for the cores to communicate with their L2 caches, and also assists in data coherency. Finally it allows for communication between fixed function logic, the cores, and L2 caches and memory.

Larrabee supports four threads of execution with separate register files. Thread switching occurs in the presents of stalls and long memory latencies.

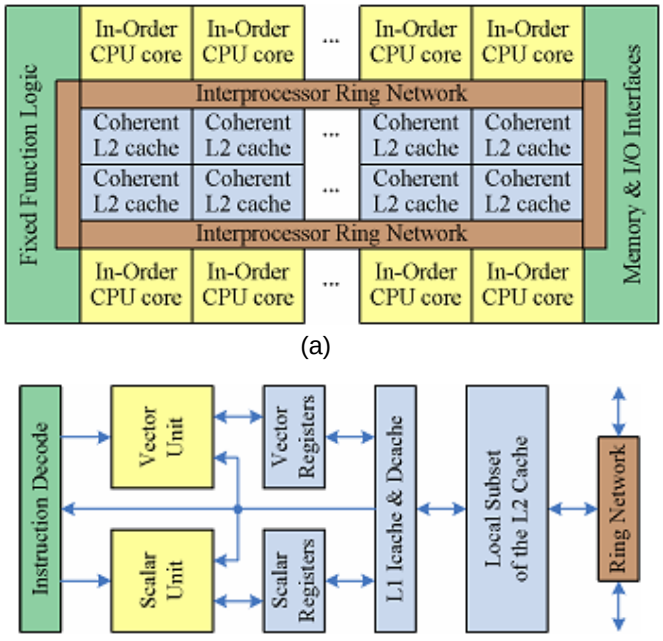

Fig. 11. (a) Larrabee manycore architecture. The CPU cores are connected by a bidirectional ring network.[23] (b) Block diagram of a Larrabee core, consisting of L1 and L2 caches and the two main processing cores the scalar and vector unit [23].

Performance studies on well known games have shown that the speed performance increases nearly linearly with increasing number of cores. Dynamic load balancing is very important to achieving high average performance, high bandwidth. Also in combination with the increased programmability of Larrabee`s cores provides a great deal of flexibility. Theoretically Larrabe can achieve up to 2 TFLOPS of computing power with 32 cores. Many applications in C/C++ that run on existing x86 PC software can be recompiled and executed on Larrabee with a few modifications. This offers high application portability and will assist in boosting the performance of compute intensive applications.

Larrabee provides opportunities in a wide range of applications both in high performance computing and graphics rendering. Its x86-based architecture along with its multicore implementation provides both high performance and flexibility.

2) NVIDIA CUDA

CUDA (Compute Unified Device Architecture) is a hardware/software architecture from NVIDIA that enables its GPUs to be programmed by high level programming languages such as C or C++. It was first introduced in the GeForce 8800 and has been applied in many scientific applications [1] including medical imaging, fluid dynamics, and numerical linear algebra.

From a software point of view CUDA organizes programs into a host program consisting of sequential threads running on the host CPU and parallel kernels that execute in parallel on the GPU. These kernel threads are organized in thread blocks and grids of parallel thread blocks. A thread block is a set of concurrently executing threads that can cooperate through synchronization mechanisms and shared memory. Each thread in a thread block has its own context (program counter, registers, etc.), and executes an instance of the Kernel. A grid is an array of thread blocks that execute the same kernel and read/write results to the same memory [24].

From a hardware point of view the hierarchy of threads just described maps to a hierarchy of stream processors on the GPU. The hardware arrangement of CUDA architecture is show in Fig. 12. The GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores, and other specialized cores within each streaming multiprocessors execute threads. Each streaming multiprocessor executes a group of 32 threads called a warp. Warps are scheduled by special units in the streaming multiprocessors the warp schedulers, without any overhead, several warps execute concurrently by interleaving their instructions.

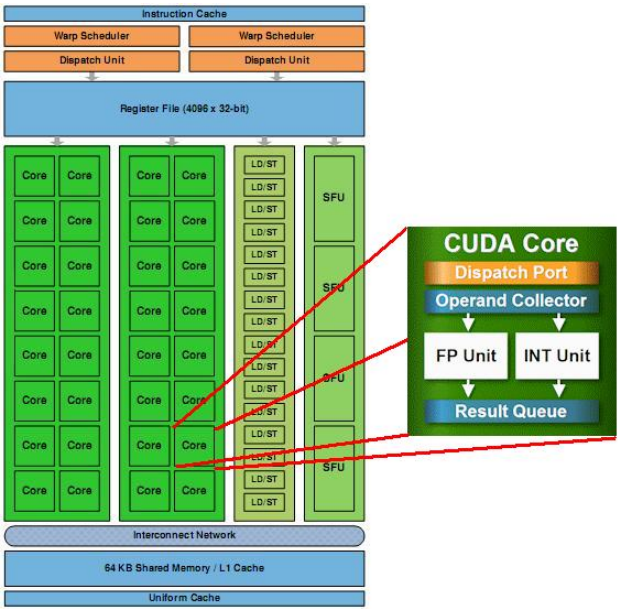

Fig. 12. Illustration of a streaming multiprocessor consisting of 32 CUDA cores and other special purpose hardware. A CUDA Core consists of a floating point and integer units along with supporting hardware. This streaming multiprocessor architecture is used in the Fermi GPU [26].

a) NVIDIA Tesla

Tesla [25] is a unified graphics and general purpose computing family of GPUs from NVIDIA. It is based on the CUDA architecture and its primary target is to provide high performance computing for applications that require large scale simulations and calculations. The processing capabilities of Tesla GPUs derive from many sources. Some of them, like texture and rasterization units, serve mainly the purposes of graphics processing. For general processing, the main units are streaming multiprocessor cores (SM) and special function units (SFU). Each core can perform two floating point operations per cycle, each SFU is equipped with four multipliers; hence can perform 4 ultiplication instructions per cycle. This gives 16 operations per cycle for 8 SPs and 8 operations per cycle for two SFUs in a single streaming multiprocessor. If an application can make both types of units work simultaneously it can achieve 24 operations per cycle per SM. For a 1.5 GHz GPU with 16 SMs this gives 576 GFLOPS. The performance obtainable in real applications is, however, much smaller. One of reasons for smaller performance can be limitations caused by slow memory accesses. The memory system of Tesla architecture consists of even more layers than that of a CPU. Apart from the main memory of the host system, there is a DRAM card memory and two pools of cache memory, one within each SM, called shared memory, and the second within each texture unit. Both caches can be utilized by scientific applications. An important feature of the architecture is that each thread issues its own memory requests. However, the performance is best when requests can be blocked by memory management unit to access contiguous memory areas.

b) NVIDIA FERMI

Fermi [26, 27] is the latest generation of graphic processing units from NVIDIA that is based on CUDA, and is the first GPU architecture designed specifically for supercomputing. Fermi boasts a great set of improvements compared to the previous generations of GPUs. It is implemented with 3 billion transistors; it has 2 times more cores, with 8 times the peak double precision floating point performance, and 20 times faster thread context switching. Fermi marks a major departure from past GPU trends towards general purpose programmability. The GPU gets support for advanced control flow mechanisms such as indirect branches and fine-grained exception handling.

In the Fermi architecture (Fig. 13) there are 16 streaming multiprocessors. Each streaming multiprocessor includes 32 CUDA cores, 16 load/store units, four special function units, and a 32K-word register file along with thread control logic. In total Fermi features up to 512 CUDA cores. Each CUDA core has a fully pipelined integer ALU and a floating point unit. A CUDA core executes a floating point or integer instruction per clock for a thread The Fermi architecture implements the new IEEE 754-2008 floating point standard, providing the fused multiply-add capabilities for both single and double precision.

Each Stream Multiprocessor has a total of 64KB on-chip memory. This memory can be either configured as 48KB of shared memory with 16KB of L1 cache or 16KB of shared memory and 48KB of L1 cache. An additional 768KB are used as a unified L2 cache that service all load, store and texture requests. The reconfigurable scheme for the shared and L1 cache of each stream multiprocessor benefits both types of programs, those that make extensive use of shared memory and those that make extensive use of L1 cache.

Fermi is the first ever GPU to support ECC protection of data in registers, shared memories, L1 and L2 caches, and DRAM; combining high performance with reliability. This is an extremely desired feature for high performance computing environments for which Fermi is intended.

The Fermi architecture could possibly be the first complete GPU computing architecture which delivers such a high level of double-precision floating point performance from a single with a flexible error protected memory and support for high level languages.

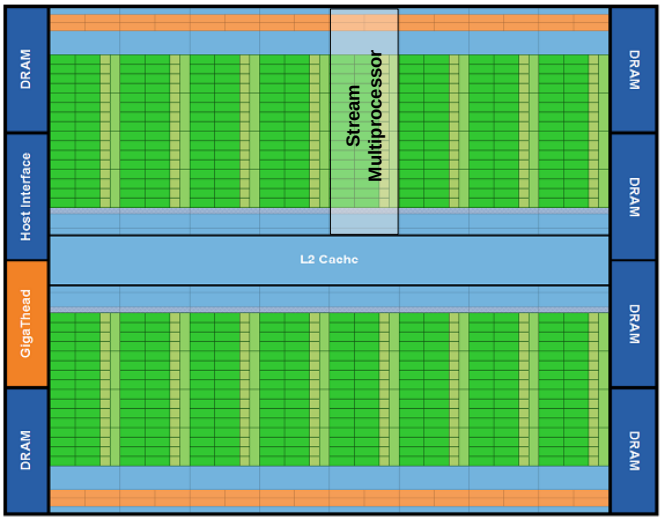

Fig. 13. Fermi's architecture [26] consists of 16 Streaming Multiprocessors a shared L2 Cache, and DRAM Modules.

3) AMD Fusion

Fusion a future generation of microprocessors developed by AMD that merges general purpose execution with 3D rendering into a single architecture. As such AMD Fusion is a heterogeneous multicore architecture that combines general purpose processors and graphics cores. The Fusion series will introduce a new modular design methodology that will allow the design of future heterogeneous multi-core systems to support a wider range of combinations. It will also feature the implementation of a universal video decoder in hardware. AMD Fusion is expected to be released somewhere in 2011 [28].

VI. Conclusions: Looking Forward

Stream Processors and GPUs are very promising parallel architectures that have been deployed in media and scientific applications with success. In favoring throughput instead of latency these processors manage to provide GFLOPS and even TFLOPS of performance and thus, have speedup many parallel applications. As their programming models and development tools mature more performance benefits will

come from these two processor architectures. However, there many challenges that lie ahead for computer architects in designing the next generation of stream processors and GPUs. How can more resources be used to increase their performance and functionality? How can architectural combined with circuit techniques reduce the power consumption of future parallel processors? Can increased performance be accompanied with increased programmability? These are challenges that must be dealt with in order to continue increasing the performance of processors in each generation. At a higher level, programming models, languages, programmers, and evaluation tools need to also adapt and evolve on the parallel architectures in order to make efficient utilization of the underlying hardware.

REFERENCES

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, 2008.

[2] M. Pharr , R. Fernando, "GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)," Addison-Wesley Professional, 2005.

[3] W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, A. Das, "Stream Processors: Progammability and Efficiency," ACM Queue, 2004, pp. 52-62.

[4] Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P., and Owens, J. D., "Programmable Stream Processors," IEEE Computer, August 2003, pp. 54-62.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51.

[6] http://cva.stanford.edu/projects/imagine/, Imagine Stanford, retrieved 8 Oct 2009.

[7] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," in Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on, 2002, pp. 282-288.

[8] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., and Chang, A., "Imagine: Media Processing with Streams," IEEE Micro, March/April 2001, pp. 35-46.

[9] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, A. Das, "Evaluating the Imagine Stream Architecture," ISCA 2004, 2004, pp. 14-25.

[10] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, U. J. Kapasi, "Merrimac: Supercomputing with Streams," SC 2003, November 2003.

[11] M Erez, N. Jayasena, T. J. Knight, W. J. Dally, "Fault Tolerance Techniques for the Merrimac Streaming Supercomputer," SC 2005.

[12] X. Yang, X. Yan, Z. Xing, Y. Deng, J. Jiang, Y. Zhang, "A 64-bit stream processor architecture for scientific applications," Proceedings of the 34th annual international symposium on Computer architecture, 2007, pp. 210-219.

[13] SPI (June 2008), "Stream Processing: Enabling the new generation of easy to use, high-performance DSPs," White Paper.

[14] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owen, and B. Towles, "Exploring the VLSI Scalability of Stream Processors," 9th Symposium on High Performance Computer Architecture, Anaheim, California, USA, February 2003, pp. 153–164.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, Introduction to the Cell multiprocessor, IBM Journal of Research and Development, Vol 49, Num 4/5, 2005.

[16] D. Luebke, G. Humphreys, "How GPUs Work," Computer, vol. 40, no. 2, Feb. 2007, pp. 96-100.

[17] K. Fatahalian and M. Houston, "Gpus: a closer look," *Queue*, vol. 6, no. 2, pp. 18-28, 2008.

[18] http://www.nvidia.com/object/geforce_family.html, NVIDIA, retrieved 15 Nov 2009.

[19] http://www.amd.com/uk/Pages/AMDHomePage.aspx, ATI, retrieved 15 Nov 2009.

[20] ATI, "ATI Radeon X800 Architecture," White Paper.

[21] E. Kilgariff, R. Fernando, "GPU Gems 2: The GeForce 6 SeriesGPU Architecture (Gpu Gems)," Addison-Wesley Professional, 2005.

[22] http://ati.amd.com/products/streamprocessor/specs.html, AMD, retrieved 13 Nov 2009.

[23] L. Seiler et al, "Larrabee: A Many-Core x86 Architecture for Visual Computing," ACM Transactions on Graphics, 2008.

[24] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," Micro, IEEE, vol. 28, no. 4, pp. 13-27, 2008.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," Micro, IEEE, vol. 28, no. 2, pp. 39-55, 2008.

[26] NVIDIA, "NVIDIA's next generation CUDA Computer Architecture: Fermi," White Paper.

[27] P.N. Glaskowsky, NVIDIA, NVIDIA's Fermi: The First Complete GPU Computing Architecture," White Paper, 2009.

[28] http://sites.amd.com/us/fusion/Pages/index.aspx, AMD, retrieved 16 Nov 2009.