# *FastLanes: An FPGA Accelerated GPU Microarchitecture Simulator*

Kuan Fang, Yufei Ni, Jiayuan He, Zonghui Li, Shuai Mu, Yangdong Deng
Institute of Microelectronics
Tsinghua University
Beijing, China
{fk10, niyf09, hejy10, lizonghui11, mus04}@mails.tsinghua.edu.cn, dengyd@tsinghua.edu.cn

***Abstract*—Graphic Processing Units (GPUs) have emerged as a new general purpose computing platform that attracts significant research efforts. Currently, GPU architecture research resorts to time-consuming software simulations to evaluate microarchitecture innovations. In this paper, we propose FastLanes, an FPGA based simulator for a generic GPU microarchitecture, to enable hardware-accelerated simulation. FastLanes consists of a function model and a timing model, both implemented on FPGA. The functional model implements the full functionality of a multiprocessor of GPU and emulates multiple multiprocessors via time-division multiplexing. We develop a hybrid implementation strategy in which certain GPU logic is directly mapped to FPGA, while the other logic is simulated by reusing the same FPGA logic. A corresponding context shifting mechanism is proposed to store execution states of threads from FPGA to external on-board memory, and vice versa. Such a mechanism makes it possible to simulate hundreds of GPU cores on a single FPGA evaluation board. Driven by the functional simulation results, the timing model considers the detailed configuration of GPU microarchitecture to derive the performance evaluation. A compiler tool-chain is also developed to allow the execution of NVIDIA GPU binary on FastLanes. Experimental results prove that FastLanes outperforms its software equivalent by up to 2 orders of magnitude.**

***Keywords—Graphics Processing Unit; GPU; FPGA; simulation; SIMD;***

## I. INTRODUCTION

Our era is witnessing a fundamental transition of the computing paradigm. Future computers will irreversibly take the parallel computing model, although the sequential computing had been dominating in the past. This strong momentum motivated the development of new computer architectures. Among these, Graphics Processing Unit (GPU) is perhaps the most successful new architecture. Especially, GPUs have received extremely wide applications with the introduction of general computing on GPU (GPGPU) model [1]. The popularity is reflected by the fact that 9 out of the top 20 supercomputer on the Green500 list adopt GPUs as the main computing resource [2,3].

Accordingly, significant research effort is being dedicated to the GPU microarchitecture. A large body of work has been proposed to improve various aspects of GPU microarchitecture. Such work includes reducing the overhead of branch divergence on SIMD hardware (e.g., [4]), tolerating variance in memory latency (e.g., [5]), supporting speculative execution (e.g., [6]), using prefetching to enhance performance (e.g. [7]), heterogeneous integration (e.g. [8]), and so on.

Current research on GPU microarchitecture universally depends on cycle-accurate software based simulators like GPGPU-Sim [9] to evaluate the performance implications of the proposed techniques. However, such software simulators suffer from a series of inherent disadvantages. First, the simulation throughput is slower than that of the GPU hardware by several orders of magnitude. A simulated run of complicated workload may take days to finish. Second, the lack of hardware details hinders computer architects to make accurate design decisions. The performance parameters of GPU components under complex interaction are challenging to estimate on a software basis.

The above problems have been noticed by the multi-core CPU community. For the first problem above, a series of FPGA based simulators have been proposed UT-FAST [10, 11], FabScalar [12], Transformer [19], HAsim [20, 21], RAMP GOLD [23]. Such hardware-accelerated architecture simulators significantly improve the productivity of computer architects. On the other hand, this line of techniques cannot be directly applied to GPU research yet because it does not address GPU microarchitecture features such as SIMD lanes and fine-grain multithreading. In addition, the running of a massively number of threads on GPU also poses significant challenges to FPGA acceleration. FabScalar is a recent development to address the second problem mentioned above. By offering a set of canonical building blocks of CPU, computer architects are able to composing many-core CPU designs with configurable parameters. The composed design is totally synthesizable so that the hardware overhead and performance implications can be accurately derived.

Following the spirit of the above works, we develop an FPGA based cycle-accurate GPU microarchitecture simulator, FastLanes. Consisting of both a function model and a timing model, it simulates the behavior of a generic many-core GPU microarchitecture and provide detailed timing statistics of program execution. The major contributions of our work are as follows.

- To the best of the authors' knowledge, this is the first work to introduce a comprehensive FPGA accelerated GPU

978-1-4799-2987-0/13/$31.00 ©2013 IEEE

simulation framework. Our FPGA based simulator significantly improves the throughput of GPU microarchitecture simulation. Experimental results prove that FastLanes outperforms its software equivalent by up to 2 orders of magnitude.

- We identified a set of unique problems for FPGA accelerated GPU microarchitecture simulation and proposed efficient solutions. Especially, our simulation context switching mechanism allows simulating a GPU consisting of an arbitrary number of cores on a FPGA with a limited capacity. We also proposed hardware techniques to superimpose GPU-specific timing information to simulated memory operations that are physically performed by memory resources on an FPGA evaluation board.
- The FPGA based simulation framework will be released as an open-source platform. Besides simulation, the functional model can also serve as a canonic GPU microarchitecture template. Researchers can use it as a foundation to test new microarchitecture features.

The rest of this paper is structured as follows. Section 2 reviews the preliminaries. We then elaborate our FPGA accelerated GPU microarchitecture simulation framework in Section 3. Section 4 introduces the software tool-chain for our simulation framework. Experimental results are reported in Section 5. We conclude the paper and outline future research directions in Section 6.

## II. PRELIMINARIES

In this section, we first review a generic GPU microarchitecture. Then the related work in FPGA accelerated architecture simulators are introduced. We also briefly explain the FPGA platform used in this research so that the readers can have an understanding on the computing and memory resources for microarchitecture simulation

### A. A generic GPU microachitecture

GPUs have evolved from a fixed-function graphics pipeline into a massively parallel computing platform supporting tens of thousands of concurrent threads. Fig. 1 is an illustration of a generic GPU microarchitecture that has been adopted by modern GPUs. The computing power of a GPU mainly comes from multiple multiprocessors (e.g. streaming multiprocessors in NVIDIA GPU and compute unit in AMD GPU). Each multiprocessor is equipped with a SIMD lane consisting of tens to hundreds of scalar processors. It also has a large register file, an on-chip L1 cache and a software-controlled scratchpad memory. The latest GPUs also have a unified L2 cache. The video memory (i.e., global memory in NVIDIA's terminology) follows the GDDR standard and is installed off the GPU chip.

GPU adopts a single instruction multiple threads (SIMT) execution model. A thread is the basic unit of parallel execution. Tens of thousands of threads can be launched concurrently. The threads are organized into a thread block in which threads can synchronize and share data through the on-chip shared memory. An invocation of a GPU function usually involve multiple threads blocks. During execution, a given number threads in a block constitute a warp (or a wavefront in AMD's terminology). Threads in each warp execute the same instruction on different data in lock steps. Many warps on the same multiprocessor execute in a multithreaded manner to hide the memory latency. Note that every active thread has its own registers so that the switching of threads is very fast on a GPU.

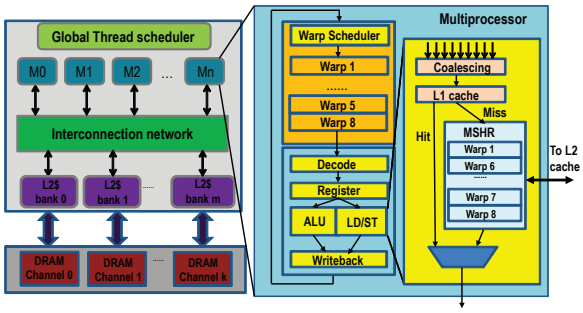


Fig. 1. A generic GPU Microachitecure

### B. Microarchitecture simulators

Computer architects rely on simulators to verify microarchitecture features. Software based simulators have become popular due to their low cost and high flexibility. Simple-Scalar [13] and its derivatives are one of the most well-known families of software based CPU microarchitecture simulator, which significantly boost the productivity of microarchitecture research. In the field of GPU research, Attila [14] is one of the earliest software based simulators for graphics APIs. Barra [15] and Macsim [16] are two functional simulators for GPGPU microarchitectures. They offer a higher simulation speed but cannot provide performance statistics. GPGPU-Sim [17] is the most-commonly used cycle-accurate GPU simulator. However, simulation with GPGPU-Sim can be time-consuming, especially for benchmarks with a heavy workload. The major reason leading to the inefficiency is twofold. First, the software simulation has to be slower because the instruction of the target microarchitecture has to be emulated by multiple instructions of the simulation host machine. Second, software simulation of many-core processors on a CPU platform is even more time-taking due to the lack of sufficient computing resource. In other words, a single CPU core is substantially underutilized when simulating a single GPU core, but there are not enough CPU cores on a single process chip to match the number of cores on a GPU chip.

To improve the simulation speed, a few FPGA accelerated microarchitecture simulators were proposed in the past years. UT-FAST [10, 11] is a single-core simulator. It only implement the timing model on a Xilinx Virtex 4 FPGA, while the functional execution is performed by a software simulator, QEMU [18]. Transformer [19] is a similar cross-platform simulator, which improves simulation throughput by loosely-coupled parallel simulation. HAsim [20, 21] is a 16-way time-division multiplexing simulator using a single physical core implemented on FPGA to simulate a multicore architecture. RAMP Gold [22, 23] simulates up to 64 CPU cores with a single functional model. A target multiprocessor's simulation contexts such as register file data and L1 caches are stored in on-chip RAMs. As a result, the upper bound of the number of CPU cores that can be simulated on a single FPGA is

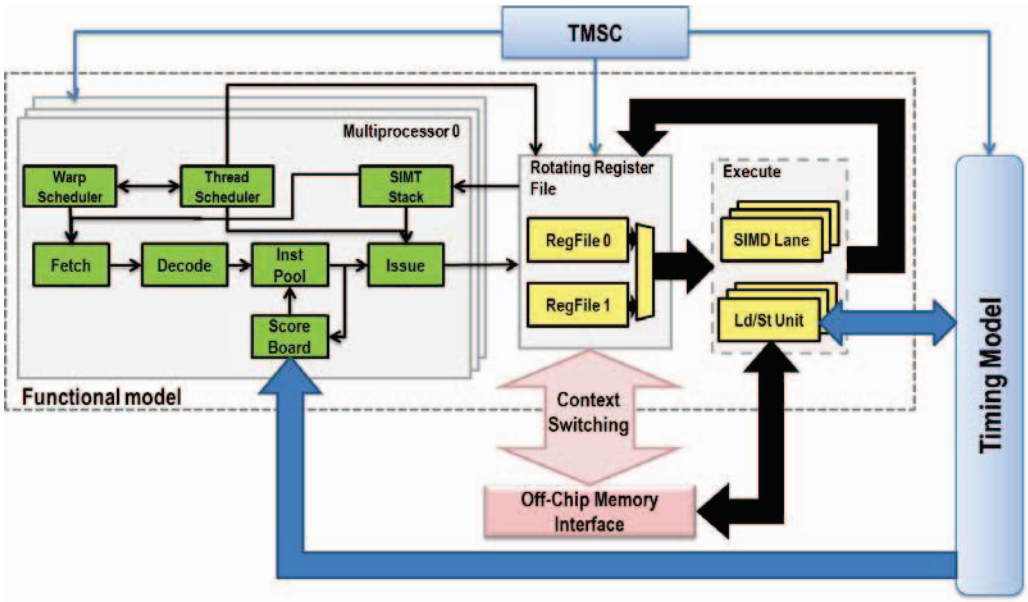


Fig. 2. Overall architecture of FastLanes

determined by the capacity of on-chip storage. When simulating 64 cores, 90% of the on-chip Block RAMs are used. FabScalar [12] is an open-source toolkit that generates register-transfer level (RTL) code on the basis of a superscalar template. The RTL code can be synthesized for physical implementation. The simulation of FabScalar are performed at either RTL or gate level to provide accurate performance statistics. All FPGA based simulators mentioned in this paragraph are designed for CPU and the techniques cannot be directly applied to GPU microarchitecture simulation. First, a GPU supports thousands of active threads, each having a dedicated set of registers. Such a large number of registers is far beyond the on-chip memories of any FPGAs available today. Second, GPUs have a more complicated memory system consisting of multiple types of memory with divergent latency/bandwidth characteristics. In addition, the GDDR memory used by GPUs are usually not available on FPGA platforms.

In addition, many of the abovementioned FPGA based simulators need to be re-synthesized if the microarchitecture configuration is alternated. Given a complex microarchitecture, the logic and physical synthesis of FPGA is time consuming. For example, implementing a 64-core CPU model on FPGA takes about 2 hours [23].

### C. Hardware platform

In this work, we use a Xilinx VC707 evaluation board as the simulation platform. It is built around a Virtex-7 FPGA with a capacity of 20M equivalent gates. The FPGA on-chip memory resources include registers, block RAMs and distributed RAMs. The block RAMs are dedicated SRAM modules, while the distributed RAMs are memory assembled from configurable FPGA logic. The Virtex-7 FPGA used in this work is equipped with 1,030 36Kb block RAM modules and supports up to 8,175 Kb of distributed RAMs. The evaluation board is equipped with 1GB DDR3 memory. The board also provides multiple I/O interfaces, among which an Ethernet link enables fast communication between FPGA and a PC. In this work, we use the Ethernet interface to download programs and simulation configurations to FPGA and collect program execution statistics.

## III. Design Considerations for Scalable GPU Microarchitecture Simulation

FastLanes implements both a functional model and a timing model on a single FPGA. The functional model offers the complete functionality of the generic microarchitecture illustrated in Fig. 2 It is able to execute GPU binaries and generate correct output. However, it ignores performance-limiting factors such as cache miss and memory latency. The timing model uses the results of the functional execution to drive the simulation of the memory hierarchy and other performance related hardware mechanisms. Such a paradigm has become popular in modern software-based and FPGA-based architecture simulators because the decoupling of the two major aspects of architecture modeling leads to a more elegant implementation with a lowered complexity.

It must be noted that a modern GPU is too complex to fit into even the largest single-chip FPGA. Therefore, the FPGA functional model only implements a smaller number of GPU multiprocessors (streaming multiprocessor in NVIDIA's terminology). The FPGA logic is then reused to simulate all GPU cores in a time-division multiplexing manner.

The GPU microarchitecture substantially vary from its multi-core CPU equivalent and thus pose a unique set of challenges to the design of FastLanes. First of all, modern GPUs exploit several levels of parallelism. They deploys tens of processing multiprocessor, with each consisting of a wide SIMD lane. Tens of groups of threads share the SIMD lane through multithreading. Suppose we want to directly map a target GPU with 32 SIMD lanes inside each multiprocessor and 1024 32-bit registers for each Lane. Even with today's Xilinx 7 Series FPGA device that offers an equivalent gate count of 20M as well as the corresponding memory resource, it is only feasible to completely implement 2 to 4 multiprocessors

of a modern GPU. Hence, the time-division multiplexing simulation is much more complicated than the case of multi-core CPUs. Second, GPUs uses a hardware multithreading technique to minimized the overhead of thread switching. As a result, thousands of active threads on a GPU all have their own registers. When using the time-division multiplexing based simulation, the thread context has to be swapped out of FPGA to off-chip memory. Such an overhead has to be carefully managed, Third, typically hundreds of threads are physically running on a GPU chip. A GPU simulator has to accurately capture the interaction. On the other hand, FastLanes can only execute the instructions of a small number of threads at a given moment. For a better accuracy, different groups of threads must be quickly switched to emulate the behavior of parallel execution, but certainly frequent rotations lead to excessive off-chip traffic. The duration of a time slice in the time-division multiplexing has to be carefully coordinated so as to balance the simulation accuracy and efficiency.

### *A. Hybrid Implementation of GPU Micorarchitecture*

An FPGA based microarchitecture simulation has to resolve a fundamental trade-off. On the one hand, FPGAs do not have sufficient capacity to implement a direct mapping of a typical GPU with tens of multiprocessors' pipeline. On the other hand, the simulation states have to be loaded and stored to off-chip memory if FPGA hardware are reused to simulate different multiprocessor in a GPU. Such a switching of simulation context, however, tends to incur significant performance overhead. To resolve the problem, FastLanes adopts a hybrid approach to utilize the characteristic of GPUs' two-level parallel architecture. The FPGA hardware implement the instruction frontend logics of all multiprocessor, but only SIMD lanes and the corresponding register files The SIMD lanes are then reused among multiple frontends for instruction execution. We choose to realize two SIMD lanes in hardware for faster switching. When a SIMD lane performs simulation, the other SIMD lane fetches its next simulation states and thus can starting execution immediately upon a context switching. After the switching, the previously active SIMD lane writes its simulation context to memory. Such an arrangement enables the overlap of simulated instruction execution and context switching. Out techniques hit a balance between hardware complexity and simulation speed.

A Time-Multiplexed Simulation Controller (TMSC) is implemented to take charge of the time-division multiplexing simulation of multiple multiprocessors. At a given simulation granularity (i.e., the number of simulated instruction cycles for a multiprocessor before switching to another), the TMSC checks the state of the pipeline and make sure the switching will not affect the simulation results. Since the functional frontend and timing model are switched simultaneously, it must be ensured that there are no on-the-fly branch operations or load/store operations before switching. When the switching conditions are satisfied, TMSC stalls the current multiprocessor and switches to simulating the next multiprocessor.

### *B. Context Switching*

The simulation context of a multiprocessors need to be preserved before the TMSC switches to another multiprocessors. Such a context includes the program counter, registers, scoreboards' data, SIMT stacks (for branch processing inside a warp), etc. Existing FPGA-based CPU simulators (e.g., [22, 23]) simply deploys multiple set of registers and RAMs on FPGA hardware with each set storing the context of a target core. For multi-core CPU simulation, this strategy is reasonable because the context of a CPU pipeline only consumes relatively small number of RAMs and registers. Nonetheless, the context of GPUs are much more complicated. A GPU is usually equipped with tens of multiprocessors, while each multiprocessors supports over one thousand threads. Here every thread has its own registers. In addition, the scratchpad memory (i.e., shared memory in NVIDIA's terminology) also need to be preserved. For instance, NVIDIA's Fermi GPU has 16 multiprocessors with each having a SIMD width of 32 and up to 48KB of shared memory. The overall context is over 8 Mbit, which is far beyond the on-chip distributed RAMs of a Xilinx Vertex 7 FPGA.

To address the above problem, one major contribution of this work is an *off-chip context switching* mechanism. Rather than storing all context in on-chip RAMs and registers at a switch, FastLanes stores such context as the register file in the external memory installed on the FPGA evaluation board. As illustrated in Fig. 2, two register files are physically implemented on FPGA and work in a rotating way. Between the two, one is working as the active register file of the current target multiprocessor, while the other is for context switching. After switching, the latter register file still keeps the data of a previously active multiprocessor. So it first saves the current data to a specific space in the off-chip memory, and then loads the stored data of the register file of the next multiprocessor to be simulated from the off-chip memory. We reserve a separate memory space for each multiprocessor. The two register files exchange their roles after each context switching. Such a rotation scheme guarantees efficient simulation at a minimized hardware overhead.

Note that the bandwidth of the external memory is limited and memory operations of the multiprocessor under simulation also consume the bandwidth. The resultant off-chip context switch traffic is the major bottleneck of simulation. Before the context is restored from the external memory, the simulation of the next multiprocessor cannot be started. In FastLanes, we choose a simulation granularity equal to the expected duration of a context switch so as to maximize the probability of hiding the switching overhead. During simulation, however, the actual duration of a switching varies. So FastLanes always wait until the context is completely exchanged before starting simulating the next multiprocessor.

### *C. Analytical Performance Model of FastLanes*

To better understand its simulation capability, we develop an analytical performance model for FastLanes. Consider a target GPU architecture equipped with $M$ multiprocessors. Each multiprocessor has a SIMD width of $L$ and supports $W$ active warps. Note that the parameter $M$ does not affect the

simulation speed unless the competition to the global memory is considered.

Assume the simulated cycles on the functional model before a switching are $cycle_{FM}$ and the context switching takes $cycle_{switch}$ cycles. Then the simulation efficiency, $\lambda$, is the ratio of simulating cycles to the host cycles.

$$\lambda = \frac{cycle_{FM}}{\max(cycle_{FM}, cycle_{switch})} \quad (1)$$

$cycle_{switch}$ is affected by the data size of switched contexts, the off-chip memory interface and the cooperation between the functional model and memory interface. In FastLanes, only the register file data need to be switched between FPGA and off-chip memory. The size of the register file is $D \times R \times W \times L$ bit, where $D$ is the width of the data, $R$ is the number of registers allocated to each thread. Because there is a two-way switching between FPGA and off-chip memory, the total amount of switching contexts is 2 times of the size of register file. Thus, the $cycle_{switch}$ can be derived in (2).

$$cycle_{switch} = 2 \times (D \times R \times W \times L) \times \frac{LT_{OMI}}{BW_{OMI} \times (1 - P_{LS}) \times \frac{freq_{OMI}}{freq_{FM}}} \quad (2)$$

where $freq_{host}$ is the clock frequency of the FPGA and $BW_{OMI}$ and $LT_{OMI}$ are the bandwidth and access latency of off-chip memory.

The same off-chip memory also simulate GPU's cache and global memory, whose operations have a higher priority than the context switching to be processed during simulation. So the memory interface can be occupied by load/store operations in the simulated GPU program with a possibility of $P_{LS}$, which is determined by the instruction flow. To better capture the characteristics of a benchmark, we define a branching factor, $\beta$, as the average number of divergent threads in a warp. We also define a factor $\eta$ as the ratio of the simulated target cycles over cycles of functional model. The factor $\eta$ not only depends on the functional model design, but also varies by benchmarks.

The clocks of the memory interface and functional model can be different, but here we assume they are identical for simplicity of analysis.

Then the simulation throughput (simulated operations per second), $V$, can be computed as (3).

$$\begin{aligned} V &= freq_{host} \times L \times \eta \times \beta \times \lambda \\ &= \left(\frac{1}{2 \cdot D \cdot R \cdot W}\right) \times (\beta \cdot (1 - P_{LS})) \times \left(freq_{host} \cdot \frac{BW_{OMI}}{LT_{OMI}} \cdot \eta\right) \times cycle_{FM} \\ &= \mu_{target} \times \mu_{benchmark} \times \mu_{host} \times cycle_{FM} \end{aligned} \quad (3)$$

The fundamental sources determining the performance of our FPGA accelerated GPU architecture simulator is demonstrated in (3). The overall simulation performance is determined by 4 factors. The first and second factors capture the impacts exerted by the target microarchitecture and simulation benchmark, respectively. Our simulation methodology is independent of the number of SIMD lanes and the size of register files of the target microarchitecture. In other words, Fastlanes is able to scale to a future GPU with a larger amount of computing and memory resurces. The third factor, $\mu_{host}$, reflects the performance of the simulation platform. Clearly, a larger off-chip memory bandwidth and shorter latency helps improve the simulation throughput. Also, we should maximize $\eta$, i.e., the percentage of effective simulation by reducing the simulation overhead. The fourth factor is $cycle_{FM}$, which reflects the basic trade-off of simulation speed and accuracy. A high $cycle_{FM}$ hide the context more effectively, but incurs a larger inaccuracy. In the work reported in this paper, we set c$ycle_{FM}$ to be $100 \times W$ for a proper tradeoff.

## IV. IMPLEMENTATION

Based on the design philosophy introduced in the previous section, we build an efficient FPGA based simulation framework for GPU microarchitecture. We explain the implementation details in this section.

### A. Functional Model

The functional model of FastLanes implements the full functionality of a multiprocessor in a typical GPU. It is able to execute multiple warps in a multithreaded manner. A warp scheduler module schedules ready warps with a round-robin algorithm. Synchronization and branch re-convergence [4] are realized by a thread controller and a SIMT-stack [5]. The thread controller manages the synchronization of multiple threads by recording the status of threads and stalling early-finished threads. The SIMT-stack is implemented with dual-port block RAMs and works with the tread controller to determine the issue mask. The predicate mask and re-convergence point of program counter (PC) are decoded from the instruction and pushed into the stack at a branch operation. Pushing a branch into stack needs to write the top 3 lines of the stack, and a pop of the re-convergence point needs an additional port to read the current next PC stored next to the top of stack. Thus a branch operation has to be performed in 2 cycles.

The memory model of FastLanes emulates the GPU memory behavior with both on-board DDR3 RAM and low-latency block RAM on FPGA. GPU poses significant challenges for memory simulation due to its intricate memory hierarchy consisting of registers, on-chip scratchpad memory, various types of cache, and global memory. The register file of a multiprocessor is simulated by FPGA's distributed RAM. Since the current FPGA on-chip RAM only has dual ports, the access to register files is in a time-divided manner. FPGA does not have a circuitry directly equivalent to cache. So the GPU cache is modeled by two FPGA modules. One module stores the data and is totally functional, while the other emulates the behavior of cache tag and is also associated with logic to reflect the cache latency as well as replacement policy. The global memory of GPU is implemented on the DDR3 memory installed on the FPGA evaluation board.

### B. Timing Model Design

The timing model of FastLanes extracts timing statistics of running a given benchmark by considering both the results of functional simulation and the configuration of target microarchitecture. The overall organization of the timing model is illustrated in Fig. 3. It is also responsible for the in-order commitment of memory requests. It receives requests

from Load Store Units (LSU), issues them to the simulated memory system on the FPGA evaluation board, sends the results back to the LSUs, and marks the scoreboard at the proper instruction cycle.

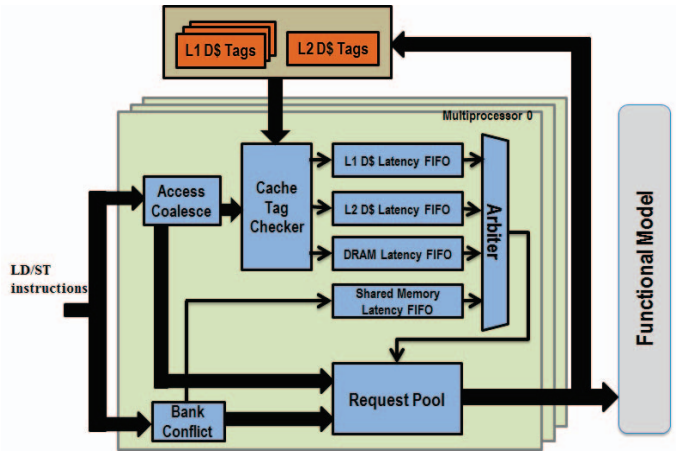


Fig. 3. Timing model of FastLanes

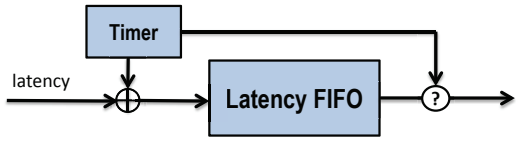


Fig. 4. Organization of latency FIFO

The timing model of FastLanes uses the on-board DDR3 memory to simulate the L1/L2 cache, the shared memory, and the global memory of a GPU. A load/store instruction issued by a multiprocessor incurs a memory request containing load/store data, address as well as other book-keeping information. When a request comes to the timing model, it is divided into segments in the memory coalescing module (global memory accesses) or the bank conflict module (shared memory accesses). These segments are processed in multiple cycles on the target memory system due to the coalescing and bank conflict processing. Note that the target GPU adopts a GDDR5 memory interface, which has a wider bus but a longer latency than the DDR3 memory. The time model first translates the GDDR5 memory requests into a set of equivalent operations for DDR3. Besides getting the data, FastLanes also needs to convert the latency values back to the timing of GDDR5.

A batch of SIMD memory requests in a single cycle may contain a great number of addresses and data, which may consume a lot of FPGA memory when they are directly feed into the pipeline of timing model. Accordingly, we do not directly emulate the memory behavior of a GPU. Instead, all on-the-fly memory requests are put into a request pool to be serviced by the DDR3 on-board memory. Each batch of memory requests will be assigned with a unique index. Since the timing of DDR3 is different from that of GPU's GDDR5, we propose a hardware based technique to model the timing of GPU memory read operations. We need 4 FIFOs, designated as latency FIFOs, to capture the different latencies of L1 cache, L2 cache, shared memory, and global memory. The hardware organization of the latency FIFO is depicted in Fig. 4 Given a memory read request, we first check the tags of L1 and L2 cache to derive the hit/miss status. Then the index of a memory request is put into the corresponding FIFO(s). For instance, a memory request corresponding to a L2 cache miss will be placed into the latency FIFO of global memory. In other words, the expiration cycle of a memory request is fixed as a predefined latency value according to the respective processing path. As long as the order of load/store operations and the just-in-time commitment to scoreboard can be maintained, we are able to deal with the memory requests at the proper time. As a result, we can emulate arbitrary latency with the memory system of the FPGA development board. For a given latency FIFO, the number of cycles is set as the typical latency value of the corresponding GPU memory resource. The processing for the shared memory is simpler because cache behavior is not involved.

### C. *A software tool-chain for FastLanes*

We develop a software tool-chain to translate an executable for a target GPU into FastLanes binary. Currently, FastLanes takes CUDA as the programming frontend, while other GPU program languages such as OpenCL [24] can also be used with proper extension to our software tool-chain. A CUDA program consists of both CPU and GPU code, while only the GPU code is processed by our tool-chain. NVIDIA defines a virtual instruction set, PTX [25], for its GPUs. CUDA programs are first compiled into PTX format, which is in turn dynamically translated into real GPU binary via a runtime. FastLanes adopts a reduced set of PTX instructions as machine code. To simulate an input CUDA program, it is first compiled into a PTX file with NVIDIA's CUDA compilers and then uses our tool-chain to translate the PTX file into a binary that can be interpreted by FastLanes.

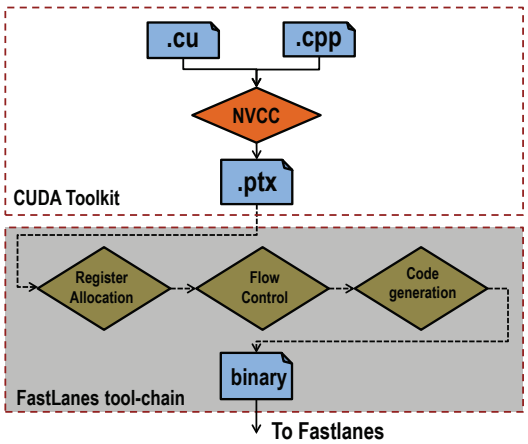


Fig. 5. Software tool-chain for FastLanes

Fig. 5 demonstrates the software tool-chain for FastLanes. A CUDA program to be simulated on FastLanes is first compiled into a PTX file by NVIDIA's NVCC compiler [25]. Next our software tool chain translated the PTX file into executable code on FastLanes. There are three main steps in the processing flow. The first step is to optimize register usage, because NVCC compiler does not perform register reusing when generating PTX file and the resultant number of registers used in PTX can be quite large. The problem for FPGA is even more challenging due to the limited resources. Accordingly, we

set the maximum number of registers as 32 and perform optimization accordingly. After register optimization, our tool-chain resort to local memory to lower the register occupancy if a PTX file uses more than 32 registers. The second step is for the control flow, especially branches, in CUDA programs. Since GPU follows a SIMD fashion when processing a warp of threads, both branch targets and re-convergence point have to be recorded during execution. Because the PTX file does not contain such information, our tool-chain calculates the re-convergence point by representing a program's control flow as a direct graph. In the third part, all the instruction in PTX file is converted into FastLanes machine code and stored in a CODE file for running on FPGA.

## V. EVALUATION

FastLanes is implemented on a Xilinx VC707 evaluation board with a clock rate of 200MHz. The functional module runs at 50 MHz since some operations need to finish in multiple cycles on FPGA. By default, it simulates a GPU configuration listed in Table I. It can be configured to simulate microarchitectures with less computing resources (e.g., narrower width of SIMD lane and/or a smaller number of multiprocessors) without re-programming the FPGA logic. It accepts GPU binaries from an on-board Ethernet interface and there is no need to re-program the FPGA when switching to simulating another benchmark.

TABLE I. CONFIGURATION OF A TEMPLATE GPU

| Attribute | Parameter |
|---|---|
| Multiprocessor | 16 |
| SIMD Lane Width | 32 |
| Warp | 32 |
| Registers per Warp | 1024 |
| L1 Cache | 16 KB |
| L2 Cache | 128 KB |
| DRAM | 512 MB |

### A. Hardware Cost

Table II lists the hardware cost of implementing various modules of FPGA based simulator for the template GPU. The frontend logics of 16 multiprocessors are directly mapped to FPGA, while only one set of 32-wide SIMD lane and 6 LD/ST units are implemented. We now deploy two register files working in a rotation manner to hide the latency of context switching. They consume around 25% of total distributed RAM. Clearly, it would significantly exceed the capacity to distributed RAM if we implemented all register files of the 16 multiprocessors on FPGA. The cache tags of L1 and L2 cache as well as the SIMD stack are implemented in block RAM. They consume around 49% of the total resource. So it is advisable to keep them on FPGA to save the memory traffic of context switching.

TABLE II. HARDWARE USAGE OF DIFFERENT MODULES OF FASTLANES

| Module | LUT | Slice Register | Distributed RAM (kb) | Block RAM |
|---|---|---|---|---|
| Function Model | 56,153 | 341,522 | 2,080 | 384 |
| Frontend(×16) | 31,772 | 328,016 | 32 | 384 |
| Instruction Buffer | 783 | 219 | 2 | 0 |
| Scoreboard | 4,016 | 1,024 | 0 | 0 |
| SIMT Stack | 2,224 | 4,54 | 0 | 24 |
| Other | 920 | 3,428 | 0 | 0 |
| Register File(×2) | 7,863 | 6,248 | 2,048 | 0 |
| Execute (SIMD lane and LD/ST) | 19,486 | 4,096 | 0 | 0 |
| Others | 32 | 3,162 | 0 | 0 |
| Timing Model | 35,037 | 24,880 | 0 | 116 |
| Overall | 91,190 | 366,402 | 2,080 | 500 |
| Occupancy | 30.04% | 60.34% | 25.44% | 48.54% |

### B. Timing Performance

We use FastLanes to simulate 10 kernels taken from NVIDIA's GPU computing SDK [26]. As described in Table II, these kernels cover a wide range of GPU computing patterns. The simulated Instructions per Cycle (IPC) of these benchmarks are shown in Fig. 6 We also ran the benchmarks on a real Kepler GPU and observed a similar trend of performance.

TABLE III. DESCREPTION OF BENCHMARK PROGRAMS

| Benchmark | Abbreviation | Description |
|---|---|---|
| blackSholes | BS | Option Pricing |
| scalarProd | SP | Scalar Products |
| fastWalshTransform1 | FW1 | Shared memory operations |
| fastWalshTransform2 | FW2 | Single in-global memory radix-4 Fast Walsh Transform pass |
| fastWalshTransform3 | FW3 | Modulate two arrays |
| quarsirandomGenerator kernel1 | QR1 | Quarsi-random generation kernel 1 |
| quarsirandomGenerator kernel2 | QR2 | Quarsi-random generation kernel 2 |
| Scankernel 1 | Scan1 | Parallel scan kernel 1 |
| Scankernel 2 | Scan2 | Parallel scan kernel 2 |
| Scankernel 3 | Scan3 | Parallel scan kernel 3 |

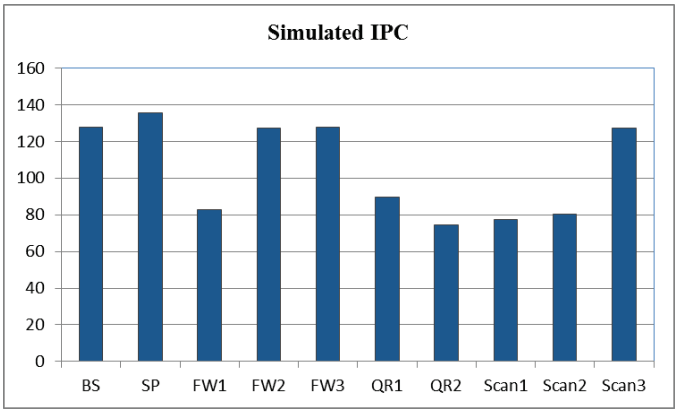


Fig. 6. Simulated IPC of benchmarks

To validate the simulation throughput of FastLanes, we simulate the same set of benchmarks with GPGPU-Sim [9]. The comparison of simulation throughput is drawn in Fig. 7. Clearly, we outperform GPGPU-Sim by almost two orders of magnitude in terms of Million Instructions Per Second (MIPS).

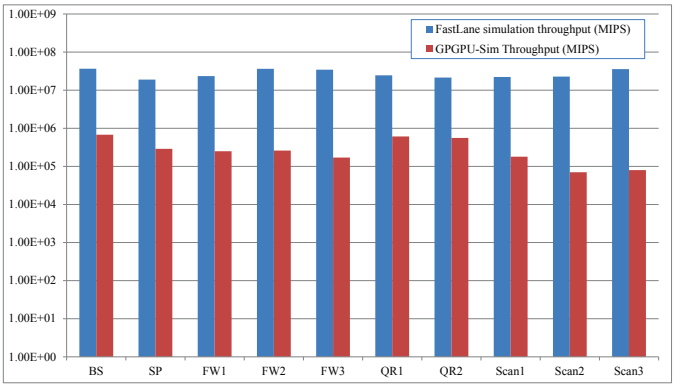


Fig. 7. Simulation throughput

## VI. Conclusion and Future Work

In this paper, we propose FastLanes, an FPGA accelerated cycle-accurate GPU microarchitecture simulator. After identifying a set of unique challenges posed by the highly complex GPU microarchitecture, we develop efficient FPGA based solutions. Especially, we propose a context switching mechanism with rotating register files to enable simulating GPUs consisting of an arbitrary number of cores on a FPGA with a limited capacity. Such a technique allows us to simulate a full GPU microarchitecture on FPGA. We also proposed hardware techniques to superimpose GPU-specific timing information to memory operations performed on a FPGA development board. Our FPGA based simulator significantly improves the throughput of microarchitecture simulation. Experimental results prove that FastLanes outperforms its software equivalent by up to 2 orders of magnitude.

This work will be extended in several directions. First, we will refine our FPGA implementation flow by adopting a more incremental design methodology to further reduce the turn-around time. Second, we are going to enhance FastLanes to support concurrent running kernels and dynamic kernel launch. In fact, we are able to study dynamic thread scheduling strategies with the help of FastLanes. Meanwhile, we will use FastLanes to evaluate microarchitecture improvements to lower the overhead of execution divergence among different threads caused by branching and memory behaviors. Finally, we are going to use FastLanes to perform a systematic study on how to take advantage of both Single-Instruction Multiple-Data and Multiple-Instruction Multiple-Data parallelism on a unified platform.

## Acknowledgment

Our thanks to the support of National Science Foundation under contract number 20121302065-61272085 and Tsinghua Self-Innovation Foundation under contract number 20121087905.

## References

[1] Blythe, D., "Rise of the graphics processor," in Proceeding of IEEE, 2008. vol. 96, no. 5, 761- 778.

[2] Mu, S. et al., "Evaluating the potential of graphics processors for high performance embedded computing," in Proc. Design, Automation & Test in Europe Conference, 2011.

[3] Green500, "The Green500 List - November 2012," http://www.green500.org/?q=lists/green201211.

[4] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in Proc. MICRO, 2007.

[5] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in Proc. ISCA, 2010.

[6] J. Menon, M. Kruijf, and K. Sankaralingam , "iGPU: exception support and speculative execution on GPUs," in Proc. ISCA, 2012.

[7] J. Lee, N. B. Lakshminarayana, H. Kim, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in Proc. MICRO, 2010.

[8] Y. Zhu, Y. Deng, and Y. Chen, "Hermes: an integrated CPU/GPU microarchitecture for IP routing," in Proc. DAC, pp. 1044-1049, 2011.

[9] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, "Analyzing CUDA workloads using a detailed GPU simulator," in Proc. ISPASS. 163-174, 2009.

[10] D. Chiou, H. Angepat, N. P. Patil, and D. Sunwoo, "Accurate Functional-First Multicore Simulators," in Computer Architecture Letters, 8(2), July 2009.

[11] D. Chiou et al., "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," In Proc. MICRO, 2007.

[12] N. Choudhary et al., "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template," in Proc. ISCA , 2011.

[13] "The SimpleScalar Tool Set," Version 2.0. http://www.simplescalar.com/docs.html.

[14] C. Gonzalez et al., "ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures," in Proceeding of IEEE Symposium on Performance Analysis of Systems and Software, 231-241, 2006.

[15] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," in Proceeding of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 351-360, 2010.

[16] "MacSim: A CPU-GPU Heterogeneous Simulation Framework," comparch.gatech.edu/hparch/macsim/macsim.pdf.

[17] G. Bakhoda et al., "Analyzing CUDA workloads using a detailed GPU simulator," in Proc. ISPASS, pp. 163-174, 2009.

[18] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in USENIX 2005 Annual Technical Conference, FREENIX Track, pp. 41–46, 2005.

[19] Z. Fang et al., "Transformer: a functional-driven cycle-accurate multicore simulator," In Proc. DAC, 2012.

[20] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs," in Proc. ISPASS, 2008.

[21] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in Proc. HPCA, 2011.

[22] Z. Tan et al., "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," In Proc. DAC, 2010.

[23] Z. Tan et al., "A Case for FAME: FPGA Architecture Model Execution," In Proc. ISCA , 2010.

[24] Khronos, "OpenCL - The Open Standard For Parallel Programming Of Heterogeneous Systems," http://www.khronos.org/opencl/.

[25] NVIDIA, "Parallel Thread Execution ISA Version 3.1", http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf.

[26] NVIDIA, "GPU Computing SDK," https://developer.nvidia.com/gpu-computing-sdk.