FlexGrip: A Soft GPGPU for FPGAs

Kevin Andryc, Murtaza Merchant, and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA

***Abstract*—Over the past decade, soft microprocessors and vector processors have been extensively used in FPGAs for a wide variety of applications. However, it is difficult to straightforwardly extend their functionality to support conditional and thread-based execution characteristic of general-purpose graphics processing units (GPGPUs) without recompiling FPGA hardware for each application. In this paper, we describe the implementation of FlexGrip, a soft GPGPU architecture which has been optimized for FPGA implementation. This architecture supports direct CUDA compilation to a binary which is executable on the FPGA-based GPGPU without hardware recompilation. Our architecture is customizable, thus providing the FPGA designer with a selection of GPGPU cores which display performance versus area tradeoffs. The benefits of our architecture are evaluated for a collection of five standard CUDA benchmarks which are compiled using standard GPGPU compilation tools. Speedups of up to 30× versus a MicroBlaze microprocessor are achieved for designs which take advantage of the conditional execution capabilities offered by FlexGrip.**

I. Introduction

Over the past ten years, soft microprocessors have become ubiquitous in FPGA design [1]. Most FPGA designs use soft processors for sequential tasks, such as I/O interfacing and control that do not demand high performance. The benefits of soft processor usage include the ability of software designers to specify functionality in a familiar high-level language (e.g. C) and the flexibility to modify this functionality for the FPGA device without the need to recompile FPGA logic, a time-consuming process that can range from minutes to days. The success of soft microprocessors has led to alternative compute models which follow a similar simple program-compile design flow. Recently, soft vector processors [2][3], which provide performance benefits for applications exhibiting significant data parallelism have appeared. Although soft vector processors address a portion of the data parallel spectrum, they are limited in their support for significant multithreaded and conditional program execution. Multithreaded soft processors have been reported [4][5], although they have generally been constrained to executing a small number of threads.

Graphics processing units for general purpose computing (GPGPUs) have exploded onto the computing scene over the past five years as languages and compilers to program them have become more programmer-friendly to use. Today, GPUs are widely used to evaluate highly multithreaded data parallel applications expressed in high-level languages such as CUDA and OpenCL. A critical benefit of these devices is their ability to automatically manage the execution of highly multithreaded applications in hardware, freeing the programmer to focus on achieving maximum parallelization by writing efficient CUDA code. Although a number of previous projects have explored mapping GPU languages directly to FPGA hardware [6][7], "GPU-like" soft FPGA architectures [8][9], and soft multi-cores [10], a soft GPGPU architecture which allows for *direct* execution of CUDA binary code following compilation with the CUDA compile-time environment has not been reported. Previous architectures also primarily consider hardware synthesis for each application, which is a lengthy and potentially infeasible option for designers which desire to execute a number of GPGPU applications on the same FPGA substrate.

This paper focuses on the implementation of FlexGrip (FLEXible GRaphIcs Processor for general-purpose computing), a fully CUDA binary-compatible integer GPGPU which has been optimized for FPGA implementation. The amount of parallelism is customizable at multiple levels including the number of parallel operations per instruction (processors) per multiprocessor. The interaction between FlexGrip and an on-chip MicroBlaze soft processor is coordinated allowing for the seamless execution of sequential and parallel portions of a CUDA program. The hardware can be used for numerous CUDA programs without hardware resynthesis.

FlexGrip has been designed based on the NVIDIA G80 architecture [11] with compute capability, version 1.0. The architecture has been implemented in VHDL for a variety of parameters and evaluated in hardware using an ML605 Virtex-6 FPGA platform which includes DRAM. The RTL code supporting FlexGrip has been written to allow the design to be quickly customized for a variety of FPGA devices. A total of five CUDA benchmarks have been directly compiled to the architecture using standard NVIDIA compiler products. The effects of customizing the architecture to the numerous small memories, optimized external memory interfaces, and on-chip digital signal processing (DSP) units commonly found in FPGAs strongly influenced the architecture and its FPGA implementation.

The remainder of this paper is structured as follows. Section II provides background on similar work and an overview of relevant features of GPUs. Section III describes the architecture of FlexGrip and provides an overview of the entire FlexGrip system including the soft GPGPU, MicroBlaze, and the DRAM interface. Section IV describes our experimental approach and results are detailed in Section V. Section VI concludes the paper and offers directions for future work.

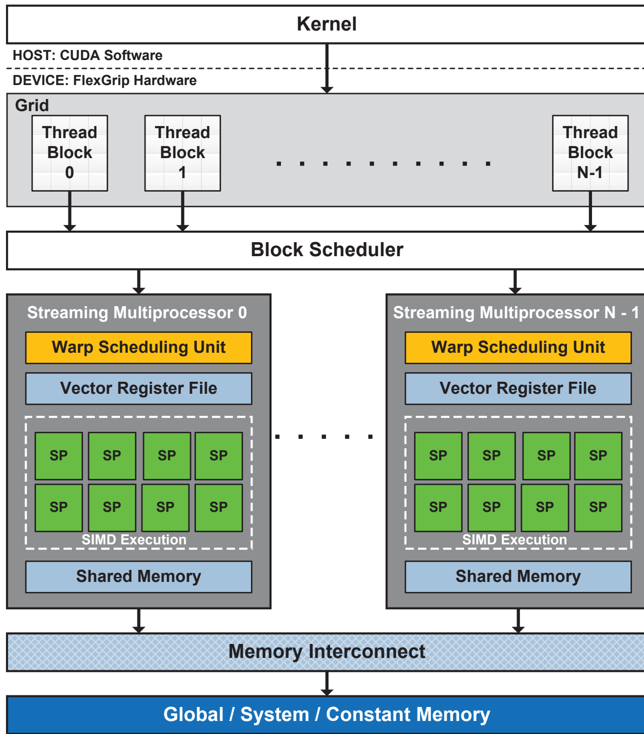

Fig. 1. Overview of a GPGPU architecture. The architecture can support multiple streaming multiprocessors

II. Background

A. GPGPUs

GPGPUs have a many-core device architecture and possess substantial parallel processing capabilities. As shown in Fig. 1, a typical GPGPU consists of an array of multiprocessors (each with two or more processors) enabling the device to execute numerous threads in parallel. In a GPGPU, a majority of the silicon area is dedicated to data processing units with only a small portion assigned to data caching and flow control circuitry. Such a design architecture makes a GPGPU suitable for solving streaming compute-intensive problems.

Although several different companies manufacture GPGPUs, in describing the devices we will use terminology commonly used with NVIDIA devices. A GPGPU is primarily made up of an array of streaming multiprocessors (SMs), with each multiprocessor consisting of multiple *scalar processor* (SP) cores that generally use 32-bit operands. The term *streaming multiprocessor* implies that scalar processors in an SM perform the same operation, SIMD style. The vector register file contains a pool of registers that is strictly partitioned across scalar processors. This way, every processor uses its own set of registers to store operands and intermediate results, steering them clear of any data dependent hazards. A shared memory serves as a communication medium between different cores residing in the same SM. In addition, there is a read-only constant memory accessible by all the threads. The constant memory space is a cache for each SM, thus allowing fast data access as long as all threads read the same memory address.

In the CUDA programming model, the host program launches a series of kernels organized as a *grid* of *thread blocks*. A thread block represents a collection of operations which can be performed in parallel. The NVIDIA device architecture partitions thread blocks and groups them into *warps*, where a warp is a smaller set of simultaneous operations, some of which may be performed conditionally. Multiple warps may be assigned to a single SM and scheduled over time. To manage fine-grained scheduling, each SM is architected as a single instruction, multiple-thread (SIMT) processor. A single instruction is mapped to the scalar processors in the SM and each processor thread maintains its own program counter (PC). Every thread performs the same operation on a different set of data, but is free to independently execute data-dependent branches. Branching threads diverge from the normal execution flow and scalar processors which do not execute the branch must be marked (deactivated) during this execution. The instructions pointed to by the branching threads are executed serially, while the non-branching threads are *masked*.

B. Differences between GPGPUs and Vector Processors

In general, GPGPUs and vector processors have many similarities and a few differences [12]. Both architectures support wide data parallel, SIMD-style computation using multiple parallel compute lanes, provide support for conditional operations, and require optimized interfaces to on-chip and off-chip memory. However, soft vector processors contain a number of limitations regarding implementation and compiler support that are addressed by GPU architectures. The following GPGPU-specific issues are explored as part of this work:

1) GPGPUs provide support for significant amounts of compute threads both within an SM and across SMs. Vector processors are generally limited to a single thread per SIMD processor (similar to an SM). Our architecture supports the implementation of numerous threads.
2) The memory system for GPGPUs is architected to take advantage of the presence of numerous threads which can be switched with low overhead by a thread scheduler. Vector processors generally rely on deep pipelining to overcome memory latency.
3) The conditional branch mechanism for GPGPUs is typically implemented in *hardware* to simplify both the user programming model and the associated compiler. The burden for handling conditional operations in vector processors generally falls on both the programmer and the compiler, often leading to inefficiencies.

FlexGrip has been designed to address all three points, although for this initial implementation, more emphasis has been placed on the first and third points, support for multithreading and conditional branch implementation.

C. Related Work

A number of previous projects have examined the implementation of data parallel applications on FPGAs. The VEGAS [3] and VENICE projects [13] examine the implementation of soft vector processors on a range of FPGAs. These architectures support a customizable number of operations performed in parallel, a optimized memory interface, and a compiler. The VESPA project [2] explored a similar approach and also considered the customization of the soft vector processor instruction set and data bit widths. As mentioned in the previous section, although similar, vector processors have a more constrained operating model compared to GPGPUs. Specifically, vector processors require a compiler to perform strip mining of vector accesses and explicitly manage the implementation of multiple threads.

Several FPGA-targeted projects consider the mapping of GPGPU applications represented in OpenCL to multithreaded implementations. The OpenRCL project [10] focused on a compiler for a multi-core architecture. The results for a single application mapped to a 30-core architecture using this LLVM-based compiler showed a 5× power improvement versus a commercial GPU for similar performance. Kingyens and Steffan [8] described a GPU-like architecture which includes substantial multithreading. This architecture was described in the context of a graphics application although it was not fully implemented in RTL or in hardware. Al-Dujaili, *et al.* [9] implemented a simple GPU-like processor which requires hand-compilation of GPU programs. The memory interface limits operational speed. Although these projects examined a similar goal to ours, the ability to target CUDA or OpenCL code to FPGAs without hardware recompile, the architectures and compilers do not take advantage of the dynamic thread scheduling, memory access, and coordinated parallel branch mechanism found in GPGPUs and expected by GPGPU compilers. Our implementation is fully compatible with CUDA integer binaries and typical GPGPU operation.

A number of recent projects, including one commercial offering, have examined *synthesizing* designs specified in CUDA and OpenCL to application-specific circuits implemented in FPGAs. The MARC architecture [14], a multi-core with custom datapaths, was optimized on a per-application basis to achieve competitive performance versus full-custom FPGA implementation. The FCUDA project [6] developed a tool which converts CUDA programs to a synthesizable version of C. A high-level synthesis tool and FPGA compiler then converts this code to hardware circuits. Owaida, *et al.* [7] presented an approach which converts OpenCL code to a synthesizable RTL template. This approach is appropriate for applications and programmer coding styles which match well with the template. Finally, Altera has developed an OpenCL compiler [15] which converts OpenCL programs to a series of custom parallel compute cores. Although all of these approaches generate circuits which are optimized for a specific application and reap the associated area, performance, and energy benefits, they all require the substantial compile time associated with FPGA synthesis, place, and route. The migration of a new application to the FPGA requires substantially more time than the few seconds normally found when targeting CUDA programs to GPGPUs.

Our approach attempts to effectively support the CUDA programming and compile environments available to GPU programmers on FPGAs without the need for costly hardware compilation or remapping to parallel RISC-style integer cores. We envision such a system as being particularly useful for environments such as computing-in-the-cloud or embedded processing where compute nodes contain reconfigurable logic that may be used for many different purposes at different times. In these cases, the extra cost, complexity, or power consumption of an off-the-shelf GPGPU in the nodes may be unwanted or unnecessary. The soft GPGPU can be swapped into the FPGA as needed and used to execute recently-compiled (perhaps on-the-fly compiled) CUDA code. Our approach provides a fast way to target CUDA programs to these environments.

D. GPGPU Optimizations for the FPGA and Contributions

In developing the soft GPGPU, a series of optimizations for FPGA implementation were considered. These optimizations, which include the effective use of block RAMs and DSP blocks, are critical to FlexGrip performance. Specific contributions of our work include:

- We provide a detailed analysis of the operation and resources consumed by the FlexGrip design along with energy and power consumption.
- Our approach to implementing the hardware-based conditional branch control circuitry that is central to GPU architectures is analyzed.
- We consider FPGA performance tradeoffs as the number of scalar processors in the soft GPGPU are varied. The variation in compute density also affects the energy consumption of the device. We evaluate the performance and energy consumption of the architecture versus a competing soft processor approach using a Xilinx MicroBlaze.

The numerical results of these contributions are quantified in Section V.

III. FlexGrip System Overview

A. FlexGrip System Overview

Our FlexGrip soft GPGPU is used in concert with a Xilinx MicroBlaze to execute parallel operations. The FlexGrip soft GPGPU is attached to the Xilinx MicroBlaze soft core microprocessor via the AXI bus as shown in Fig. 2. During execution of a program, the MicroBlaze processor loads a driver that communicates control, status, and data to the AXI bus interface logic. The control logic acts as an interface between the AXI bus and the FlexGrip GPGPU. It executes functions depending on the values written to the control register. Once the driver is loaded, it dispatches CUDA instructions and data which in turn are loaded into system and global memory, respectively, by the control logic. In addition, the driver loads parameters associated with the CUDA kernel

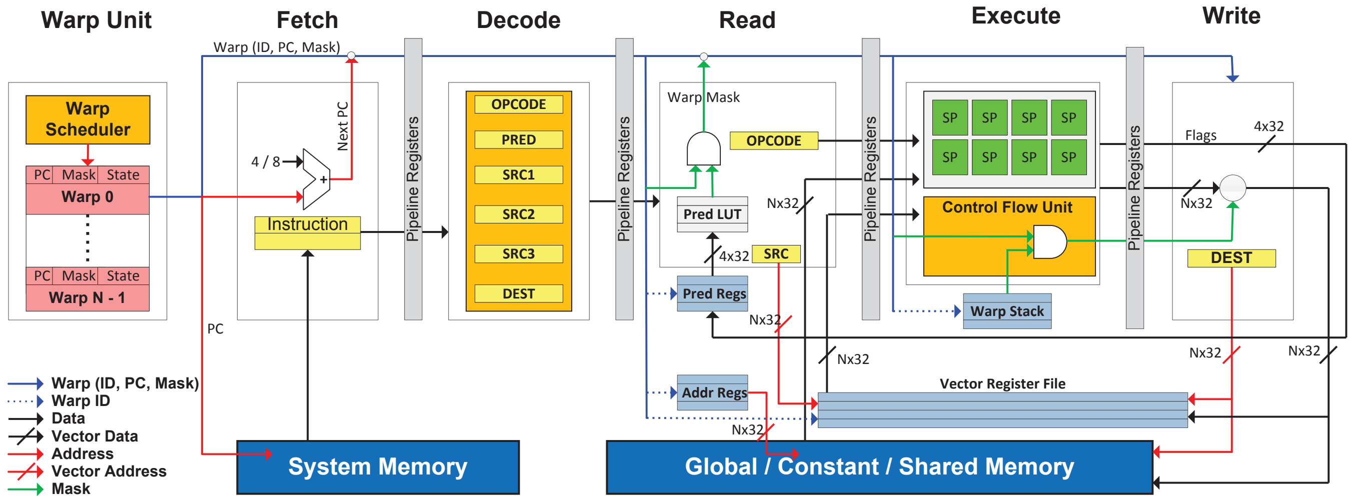

Fig. 3. Block diagram depicting the details of the FlexGrip Streaming Multiprocessor

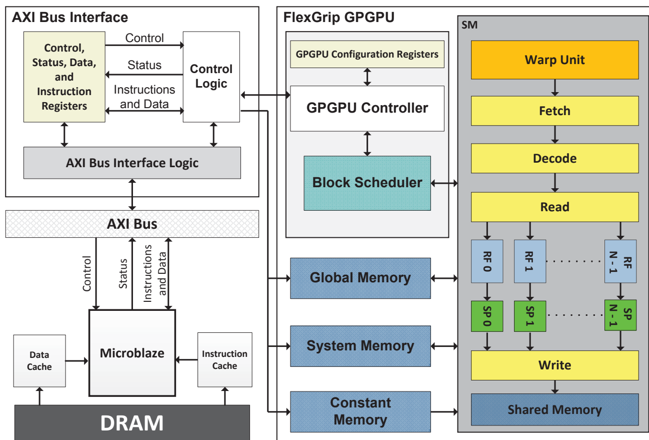

Fig. 2. Overview of the system architecture showing the FlexGrip GPGPU connected to the MicroBlaze processor via the AXI bus.

program such as thread block and grid dimensions, number of thread blocks per SM, the number of registers used per thread, and the shared memory size. These parameters are stored in the GPGPU configuration registers. After initialization, control flow is passed to the GPGPU to execute the CUDA kernel. During this period, the MicroBlaze processor can continue execution concurrently with the GPGPU.

FlexGrip follows a SIMT model in which an instruction is fetched and mapped onto multiple scalar processors simultaneously. The block scheduler is responsible for scheduling thread blocks in a round-robin fashion. The number of thread blocks scheduled at the same time is determined by the number of scalar processors in an SM and the number of SMs. After scheduling the thread blocks, the block scheduler signals the warp unit to initiate scheduling the warps, which are contained within the respective thread blocks. The maximum number of thread blocks that can be scheduled to a SM is restricted by the available shared memory and SM registers.

B. FlexGrip Streaming Multiprocessor

For this custom FPGA implementation we have developed a five-stage pipelined SM architecture, shown in Fig. 3. The SM includes Fetch, Decode, Read, Execute and Write stages. The *warp unit* at the front of the pipeline coordinates the execution of instructions through the pipeline. The following sections elaborate on the different blocks used in this architecture. Once the block scheduler assigns thread blocks to a specific SM, the warp unit assigns threads to specific scalar processors (SP). This unit schedules warps in a round-robin fashion. Each warp includes a program counter (PC), a *thread mask*, and state. Each warp maintains its own PC and can follow its own conditional path. The mask is used to prevent thread execution within a warp for threads which do not meet specific conditions. The warp state indicates the status of the warp: Ready, Active, Waiting or Finished. The Ready state indicates that the warp is idle and is ready to be scheduled, while the Active state indicates that the warp is currently active in the pipeline.

Within a warp, threads are arranged in rows depending on the number of scalar processors (SP) instantiated within an SM. For example, for an 8-SP configuration, a warp with 32 threads would be arranged in four rows with each row containing 8 threads. Similarly, for a 16-SP configuration, a warp would be arranged in two rows with 16 threads each. The maximum parallelism is achieved with 32 SPs and one row.

The Fetch stage is the initial stage of the execution pipeline and is responsible for fetching four or eight-byte CUDA binary instructions from system memory. After fetching the instruction, the PC value is incremented (by 4/8 bytes) to point to the next instruction. The Decode stage decodes the binary instruction to generate several output tokens such as the operation code, predicate data, source and destination

operands.

In the Read stage, source operands are read from the vector register file or shared/global memory blocks depending on the decoded inputs. The vector register file is partitioned, with each thread assigned a set of general-purpose registers. The address register file stores memory addresses for load and store instructions. All instructions can include an optional predicate flag that controls conditional execution of the instruction (predicate instructions). The predicate register file is used to store these predicate flags, each of which is then used as an index into a predicate look-up table which obtains the predicated instruction (i.e.: less than, greater than, etc). The warp mask is updated by combining the current mask with the predicated instruction. The constant memory is a read-only memory which is initialized by the host.

The Execute stage consists of multiple scalar processors and a single control flow unit. This unit operates on control flow instructions such as branch and synchronization instructions which are described in more detail in the next section. Each thread is mapped to one scalar processor, enabling parallel execution of threads. The scalar processors support integer-type addition, subtraction, multiplication, multiply and add, data type convert operations, shifting operations and Boolean logic operations.

The Write stage stores intermediate data in the vector register file, memory addresses in the address register file, and predicate flags in the predicate register file. Final results are stored in the global memory. All pipeline stages output a stall signal that is fed to the preceding stage. The stall signal indicates that the stage is busy and not ready to accept new data.

TABLE I
FLEXGRIP-SUPPORTED CUDA INSTRUCTIONS

Opcode	Description
I2I	Copy integer value to integer with conversion
IMUL/ IMUL32/ IMUL32I	Integer multiply
SHL	Shift left
IADD	Integer addition between two registers
GLD	Load from global memory
R2A	Move register to address register
R2G	Store to shared memory
BAR	Barrier synchronization
SHR	Shift right
BRA	Conditional branch
ISET	Integer conditional set
MOV/ MOV32	Move register to register
RET	Conditional return form kernel
MOV R, S[]	Load from shared memory
IADD, S[], R	Integer addition between shared memory and register
GST	Store to global memory
AND C[], R	Logical AND
IMAD/ IMAD32	Integer multiply-add; all register operands
SSY	Set synchronization point; used before potentially divergent instructions
IADDI	Integer addition with an immediate operand
NOP	No operation
@P	Predicated execution
MVI	Move immediate to destination
XOR	Logical XOR
IMADI/ MAD32I	Integer multiply-add with an immediate operand
LLD	Load from local memory
LST	Store to local memory
A2R	Move address register to data register

C. Conditional Branch Circuitry

A key contribution of the soft FlexGrip GPGPU is its ability to support thread-level branching in hardware. A warp diverges if the branch outcome may not be the same for all threads in the warp. The set synchronization instruction is used to set the reconvergence point of the branch instruction that will be reached irrespective of whether or not the branch is taken. In case of divergence, execution for some SPs proceeds along one path (e.g., taken) until the reconvergence point is reached. When the point is reached, the execution switches back to the other path (not-taken) for the remaining SPs. When the reconvergence point is reached for these processors, thread execution of the same set of instruction operations is performed by all processors once again. A stack in the Execute stage circuitry is used to keep track of the PC for SPs which are stalled waiting for conditional execution in an alternate path to terminate.

In order to synchronize warps within a thread block, CUDA supports explicit *barrier synchronization*. Warps that reach the barrier instruction first have to wait for other warps to reach to the same checkpoint, and are marked as Waiting. When all the threads in a warp finish executing the kernel, the warp is declared Finished. The warp unit contains a warp state memory and a warp data memory to hold intermediate values (not shown in Fig. 3). The warp state memory holds the state of each warp and warp data memory holds the thread mask and the warp PC. The barrier synchronization instruction is used before potentially divergent branch instructions.

D. CUDA Instructions

The soft GPGPU supports the NVIDIA G80 instruction set with compute capability 1.0. Instructions were tested based on the requirements of the selected benchmarks. We tested 27 integer CUDA instructions as a part of this research. The list of all supported instructions is shown in Table I. All instructions needed by our benchmark circuits are supported.

E. FPGA-Specific Considerations

All circuitry described in this section has been implemented in a Virtex-6 FPGA and has been shown to operate correctly. While a strength of the FlexGrip architecture is its ability to execute numerous CUDA binaries without the need for FPGA design recompilation, a user may select to create a new FlexGrip implementation, if desired. The FlexGrip architecture is designed such that different counts of scalar processors per SM or SMs per GPGPU can be implemented by modifying parameters in a configuration file and rerunning Xilinx tools. Depending on the target FPGA platform, the user can customize FlexGrip to maximize performance or area.

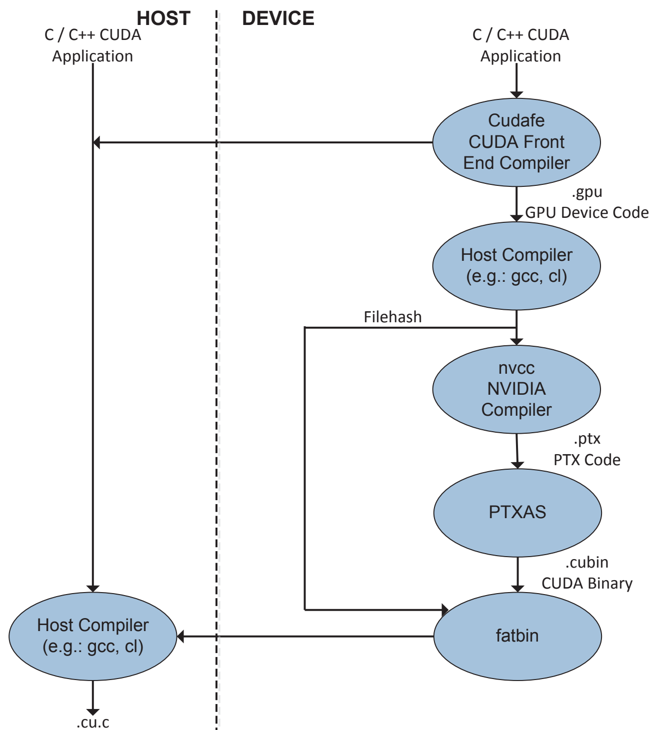

Fig. 4. Software Flow for the FlexGrip Soft GPU

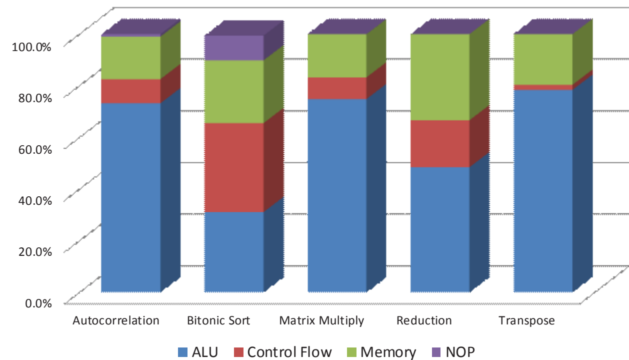

Fig. 5. Percent of instruction operations executed for each benchmark

For a specific FlexGrip hardware implementation, a small set of in-design registers are used to store application specific configuration information, such as the grid size and thread block count.

Most of FlexGrip source code was written in custom VHDL code to provide for fine-grained control, although MATLAB's Simulink was used for coarse-grained functions. Xilinx System Generator converts MATLAB Simulink blocks to RTL code for rapid development of FPGA designs. For example, Simulink was used to connect DSP, adder, and multiply blocks together to form SP functional units. To minimize data latency, we heavily utilize dual-ported block RAMs throughout the design. In the case of the warp unit scheduler, the state information and the data are stored in block RAM indexed by the warp ID. This allows warps to be scheduled every clock cycle after an initial one clock cycle of latency. Similarly, the vector, predicate, and address registers use dual-port block RAM providing simultaneous read and write access. To support the numerous integer arithmetic instructions, the scalar processors take advantage of Xilinx's DSP48E1 digital signal processing blocks. A single DSP slice can support add/subtract, multiply, multiply-add, shift, and bitwise logic instructions.

IV. Experimental Methodology

A. Software Flow

The complete CUDA binary code generation flow is illustrated in Fig. 4. At compile time, the input program is divided by the CUDA front-end ($cudafe$) into C/C++ host code and the GPU device code. The GPU code is fed to the host compiler (e.g.: gcc, cl) to generate a filehash containing device code descriptors. The device descriptors are evaluated by runtime libraries whenever device code is invoked by the system. The NVIDIA CUDA compiler ($nvcc$) converts this information to PTX assembly instruction code which is then converted to CUDA binary instructions (.cubin). This code, along with the device code descriptors, are merged (fatbin) and compiled together with the host compiler to produce a final executable. Microsoft Visual Studio 2008 and NVIDIA Toolkit v2.3 are used together to compile the CUDA code file. The NVIDIA toolkit is comprised of the NVIDIA CUDA compiler (nvcc), and the CUDA driver and runtime API libraries required for building the executable and the cubin file.

B. Design Environment and Benchmarks

Synthesis was performed using the Xilinx ISE 14.2 toolkit and Modelsim SE 10.1 was used for simulation and verification. A block-level simulation approach was adopted, where each block was individually verified using logic simulation in addition to a system level verification. Inputs were stimulated using CUDA binary instructions and data stored in block RAM. To rapidly evaluate a variety of benchmarks and data, we generated Memory Initialization Files (.mif) that were used to populate Xilinx Block RAM cores.

We have evaluated five CUDA applications, *bitonic sort, autocorrelation, matrix multiplication, parallel reduction* and *transpose* from the University of Wisconsin [16] and the NVIDIA Programmer's Guide [17], using FlexGrip. The mix of data-parallel (e.g. multiply, transpose) and control-flow intensive (e.g. bitonic sort) benchmarks helped us evaluate our platform. Fig. 5 provides a breakdown of the instruction operations by type for each of the benchmarks.

V. Experimental Results

The FlexGrip soft GPGPU design was implemented on a Xilinx ML605 development board which utilizes a Virtex-6 VLX240T device. The device area and design operating frequency for designs with a varying number of scalar processors are annotated in Table II. We performed experiments and

TABLE II
AREA COMPARISON OF FLEXGRIP IMPLEMENTATIONS

Parameters	Freq. (MHz.)	LUTs	Registers	BRAM	DSP48E
8 SP	100	71,323	103,776	120	156
16 SP	100	113,504	149,297	132	300
32 SP	100	231,436	240,230	156	588

compared performance and energy results against a Xilinx MicroBlaze soft-core processor with about 3,000 LUTs running at 100 MHz using C versions of the same benchmarks. For the purposes of this paper, a design with a single SM and 8 scalar processors was implemented and benchmarked on the ML605 board, while 16- and 32-SP designs were simulated. The FlexGrip design implemented in hardware could successfully run all five benchmarks using the same FPGA bitstream. The CUDA compile times for all benchmarks were less than one second.

A. Architecture Scalability

We ran experiments by varying the number of scalar processors within a single SM which effectively varies the number of threads that can be executed in parallel. Fig. 6 shows application speedups versus a MicroBlaze for a varying number of SPs per SM. Application speedups range from 7× to 29× with an average speedup close to 12× for 8 SPs, 18× for 16 SPs, and 22× for 32 SPs. Since they are highly data parallel, *matrix multiplication* and $reduction$ show the largest speedups. *Reduction* has a highly symmetric data flow graph consisting of multiple iterations. The number of array elements in the benchmark is halved with each iteration, progressively leading to smaller number of scheduled warps. Considering the array size to be a multiple of 32 (the warp size), all active threads remain tightly packed within a warp in every iteration, thus fully utilizing the warp at all times. In $bitonic$, the sorting network consists of a fixed number of swapping operations that are performed at every stage. Though the warp divergence increases with an increased number of parallel threads, the divergence cost is amortized by performing more swapping operations in parallel. $Transpose$ shows less speedup due to low arithmetic intensity and memory bandwidth limitations. *Matrix multiply* has better performance than *transpose*, as the former has higher arithmetic density which amortizes the number of required memory accesses.

One common limitation to cycle speedup for all benchmarks targeted to our architecture is memory access. Memory operations are most effective when the burst data is written and read in parallel. This action requires the memory to be split up into multiple banks and coalesced, such that consecutive memory addresses fall into consecutive banks. Most data parallel CUDA kernels include neighboring threads that access consecutive memory locations. However, for control flow intensive applications where data accesses are not sequential, memory mapping is more of a challenge, especially if multiple threads access the same memory location. For the sake of architectural simplicity, enhanced support for memory

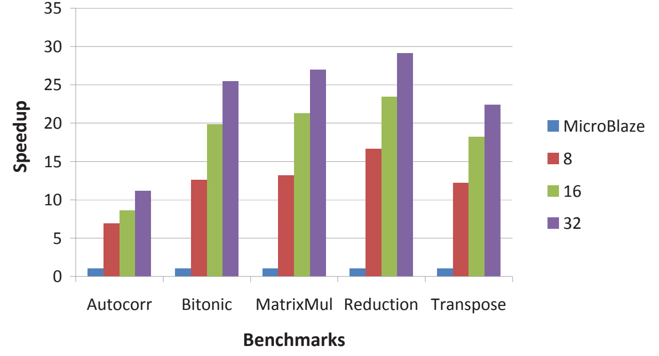

Fig. 6. Speedup vs. MicroBlaze for variable scalar processors

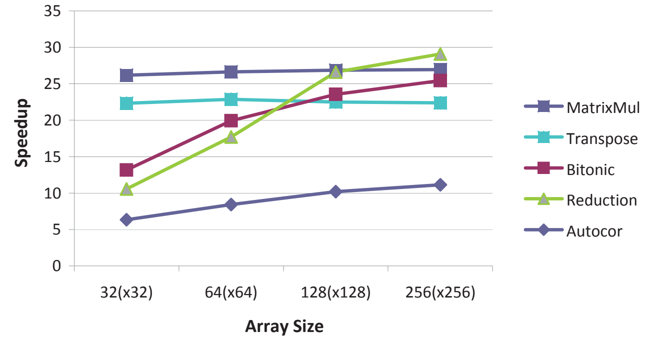

Fig. 7. Speedup of 32-SP GPGPU vs. MicroBlaze for varying problem size

coalescing was not included in our first soft GPGPU prototype and will be addressed in the future. The matrix benchmarks pay a slightly larger penalty for memory bandwidth limitations due to a larger number of scatter-gather memory operations.

B. Application Scalability

Experiments were conducted to observe the performance of the soft GPGPU in comparison to MicroBlaze for varying problem (input data array) sizes of each benchmark. The speedup results are shown in Fig. 7. Due to its regular kernel structure, $reduction$ reaps the steepest performance benefits of almost 30× as the size of the array becomes large. With increasing array size, performance increases gradually for both $autocorrelation$ and $bitonic$ up to a certain point and then begins to taper off. This result can be attributed to the accumulation of the warp divergence penalty over the execution time of larger arrays, amortizing the parallel processing benefits. *Matrix multiply* shows a speedup of about 27×, with $transpose$ showing an average speedup of 22×. The flat curve of both benchmarks are due to limitations of the memory bandwidth.

C. Energy Efficiency

We used Xilinx's XPower power estimator tool to determine static and dynamic power for the designs (Table III). Since

TABLE III
FPGA POWER ESTIMATES (W) AT 100 MHZ

	Dynamic	Static	Total
8 SP	1.59	2.03	3.62
16 SP	1.92	2.04	3.97
32 SP	2.32	2.05	4.37
MicroBlaze	0.37	2.00	2.37

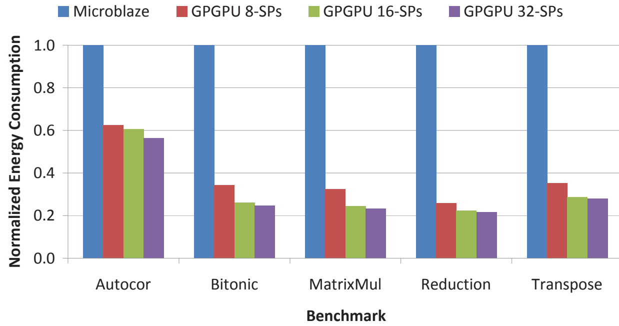

Fig. 8. Normalized dynamic energy consumption versus MicroBlaze for different SP counts

static power is largely a function of the device size, we evaluate the dynamic energy consumption of the implementations. This value is determined by multiplying dynamic power by application execution time. We performed this experiment for 8-, 16-, and 32-SP architectures and compared the results against the MicroBlaze processor. Fig. 8 depicts the normalized dynamic energy consumption for the various benchmarks. On average, the GPGPU requires 66% less energy than the MicroBlaze processor, with the largest energy decrease of 78% for the 32-SP $reduction$ implementation.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, a CUDA binary-compatible, soft GPGPU architecture is described. The scalable design has been fully-implemented and tested on a Xilinx ML605 development board. A novel design aspect of GPGPUs versus microprocessors and vector processors is the ability to handle thread divergence and barrier synchronization in hardware. The FlexGrip soft-core GPGPU provides control circuitry which can automatically handle complex conditional control operations in hardware, similar to the GPGPU programming model. Our design has been validated using five benchmarks which were compiled from CUDA to a binary representation. All five benchmarks were executed using the same FlexGrip design (no need to create a new bitstream). The binary was executed on the soft GPGPU without any per-application hardware modifications. Experimental results demonstrate application speedups of up to 30× versus a MicroBlaze soft processor for highly parallel benchmarks.

Future enhancements include optimizing the area and improving the memory infrastructure to take advantage of coalescing and DMA. Scalar processors will operate on a separate clock domain enabling them to be clocked at higher frequencies in addition to supporting floating point and transcendental functions. Finally, we plan to release the source code for FlexGrip to the reconfigurable computing research community to allow for in-FPGA hardware experimentation of GPGPUs for a wide range of researchers.

ACKNOWLEDGMENTS

We thank L-3 KEO for their support and contributions. We also thank Xilinx for the donation of the ISE 14.2 toolkit and Modelsim SE 10.1 software.

REFERENCES

[1] K. Pocek, R. Tessier, and A. DeHon, "Birth and adolescence of reconfigurable computing: A survey of the first 20 years of field-programmable custom computing machines," in *Highlights of the First Twenty Years of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, Apr. 2013, pp. 3–19.

[2] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, scalable, and extensible FPGA-based vector processors," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2008, pp. 61–70.

[3] C. Chou, A. Severance, A. Brandt, Z. Liu, S. Sant, and G. Lemieux, "VEGAS: Soft vector processor with scratchpad memory," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2011, pp. 15–24.

[4] B. Fort, D. Capalija, Z. Vranesic, and S. Brown, "A multithreaded soft processor for SoPC area reduction," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, Apr. 2011, pp. 131–142.

[5] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *International Conference on Field Programmable Logic and Applications*, Sep. 2007, pp. 210–215.

[6] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Symposium on Application Specific Processors*, Jul. 2011, pp. 35–42.

[7] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, May 2011, pp. 186–193.

[8] J. Kingyens and J. G. Steffan, "A GPU-inspired soft processor for high-throughput acceleration," in *IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2010, pp. 1–8.

[9] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and W.-F. Wong, "Guppy: A GPU-like soft-core processor," in *International Conference on Field Programmable Technology*, Dec. 2012, pp. 57–60.

[10] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: Low-power high-performance computing with reconfigurable devices," in *International Conference on Field Programmable Logic and Applications*, Sep. 2010, pp. 458–463.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," in *IEEE Micro*, vol. 28, no. 2, March-April 2008, pp. 39–55.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Boston, MA: Morgan Kaufmann, 2011.

[13] Z. Liu, A. Severance, S. Singh, and G. Lemieux, "Accelerator compiler for the VENICE vector processor," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2012, pp. 229–232.

[14] I. Lebedev, S. Chen, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, "MARC: A many-core approach to reconfigurable computing," in *International Conference on Reconfigurable Computing*, Dec. 2010, pp. 7–12.

[15] "Altera Corporation. Implementing FPGA design with the OpenCL standard," white paper WP-01173-1.0, November 2011.

[16] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte, and K. Compton, "ERCBench: An open-source benchmark suite for embedded and reconfigurable computing," in *International Conference on Field Programmable Logic and Applications*, Aug. 2010, pp. 408–413.

[17] "Nvidia CUDA programming guide," version 2.3.1, Aug. 2009.