FGPU: An SIMT-Architecture for FPGAs

Muhammed Al Kadi, Benedikt Janssen, Michael Huebner

Chair for Embedded Systems for Information Technology

Ruhr University of Bochum

{muhammed.alkadi, benedikt.janssen, michael.huebner}@rub.de

ABSTRACT

Driven by its high flexibility, good performance and energy efficiency, GPGPU has taken on an increasingly important role in embedded systems. In this paper, we present the basic core of FGPU: a GPU-like, scalable and portable integer soft SIMT-processor implemented in RTL and optimized for FPGA synthesis with a single-level cache system. Compared to a performance-optimized MicroBlaze implementation on the same FPGA, the biggest implemented core of FGPU achieves average wall clock speedups of 49x and a measured power saving of 3.7x with an area overhead of 17.7x. Compared to an ARM CPU with a NEON vector processor, we measured an average speedup of 3.5x over the used benchmark. FGPU is highly parametrizable and it does not contain any manufacturer-specific IP-cores or primitives.

Keywords

GPGPU; SIMT; soft GPU; FPGA

1. INTRODUCTION

The official birth of General Purpose computing on Graphical Processing Units (GPGPU) was announced by Nvidia in 2007 as CUDA was introduced [14]. One year later, the Khronos group published the OpenCL specifications [10] as the manufacturer-independent programming language in this domain. Since that time, the application areas of GPUs expanded remarkably: not only on top of graphic cards in personal computers, but also in high performance computing platforms and embedded systems. GPUs have been used as accelerators in many supercomputers all over the world [1]. Nowadays, they can be programmed efficiently using high level languages, such as MATLAB [13].

Industry quickly recognized the advantages of GPGPU in embedded applications. Embedded GPUs, programmable with CUDA [15] or OpenCL [6], have been available since 2008. Current embedded Systems-On-Chips (SoCs) tend to integrate a hard GPU-core next to an ARM-CPU [6] or the

FPGA Programmable Logic (PL) [23]. Researchers in contrast, have put less focus on developing GPU architectures. Instead, many projects have focused on synthesizing tasks for FPGAs from high level GPU languages [16][17][22]. Others have tried to schedule kernels on a pre-synthesized overlay on the FPGA fabric [18][8]. Multi-processor platforms, programmable with GPU languages, have been realized by replicating modified soft microprocessors (like MIPS [11], LEON3 [2] or MicroBlaze [12]) and balancing the workload over them at runtime.

The main advantages of soft- over hard-GPUs do not differ from those in the microprocessors domain. Soft solutions enable application-specific adaptations and accelerate design space exploration [25]. They are easier to realize, multiply or integrate with other system parts. Because the physical characteristics of FPGAs have been steadily improving, system architects are now more willing to trade extra area for more integration flexibility and a shorter development cycle. Hence, as long as the design goals are met, portable software implementations on soft GPUs are more attractive than programming HDL in many applications.

Directly synthesizing kernels from GPU languages to FPGA logic with High Level Synthesis (HLS) tools may be more efficient than using soft GPUs [3]. But when multiple tasks have to be performed or the task size changes, the HLS-approach does not scale well: adding new tasks implies more occupied area and extra consumed power. Even when combined with partial reconfiguration, the time and the power needed to reconfigure the FPGA may be significant. On the other hand, performing software tasks on soft GPUs is a much more flexible solution.

FGPU (FPGA general purpose **G**raphical **P**rocessing **U**nit) is a portable, scalable and flexible soft Single-Instruction Multiple-Thread (SIMT) processor developed in VHDL-2002. Although it has been optimized and tested for the 7series FPGA architecture from Xilinx, it does not include any IP-cores or FPGA-primitives. It depends on the capabilities of the synthesis tool to infer the right FPGA logic like DSP or BRAM blocks from the VHDL code. It does not replicate, even partially, any other GPU architecture. The platform and execution models of FGPU follows those of the OpenCL standard [10]. FGPU has its own ISA, which is a extended subset of the MIPS assembly. The extra instructions are inspired from the execution model of OpenCL. The timing characteristics degrade very little as the size of the implemented core increases. FGPU has its own configurable 1-Level cache system, which can be connected to one or more 32- or 64bit AXI4 interfaces.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPGA'16, February 21-23, 2016, Monterey, CA, USA

© 2016 ACM. ISBN 978-1-4503-3856-1/16/02...\$15.00

DOI: <http://dx.doi.org/10.1145/2847263.2847273>

Figure 1: FGPU architecture

This paper is organized as follows: Section 2 presents the FGPU architecture, along with its execution model. The details about optimizing and synthesizing FGPU for FPGAs are discussed in Section 3. Testing and evaluating the proposed architecture and comparing it to other solutions is described in Section 4. Section 5 gives an overview of related work and a comparison to similar projects. Section 6 concludes the paper and future work is briefly described in Section 7.

2. FGPU ARCHITECTURE

2.1 Execution Model

The FGPU execution model can be considered as a simplified version of the one described in the OpenCL standard [10]. For each task to be executed, an *index space* has to be defined: its size and the number of dimensions are determined at runtime. For each point in this space, a work-item will be launched¹. The programmer may parallelize the execution by linking the coordinates of each work-item to a part of the data to be processed. For example, in order to perform a vector addition, the programmer may define a 1-dimensional index space of the size L , where L is the length of any of the input vectors. The work-item launched for index $0 \leq i < L$ may add the elements located at the i^{th} entries in each input vector and write the result into the i^{th} location in the result vector.

The index space has up to 3 dimensions and it is decomposed in equally-sized work-groups. The depth of the index space along any dimension may be up to 2^{32} and must be a multiple of the work-group size along the same dimension. A work-group can include up to 512 work-items. A work-item is uniquely identified by its 3D-coordinates tuple in the index space, named *global ID*. The *Global Offset* of a work-group is the smallest global ID of any included work-item. The *local ID* of a work-item reveals its relative coordinates inside the work-group and it is calculated by subtracting its work-group global offset from its own global ID.

2.2 Memory Model

In the current version of FGPU, each work-item can use two types of memories to perform its computations:

- *Private Memory*: Each work-item has 32 registers of 32bits. The registers are not accessible from other

work-items. The first register (R0) is read-only and contain the value 0. Private memory is the closest memory to the ALUs as well as the fastest one.

- *Global Memory*: It is external to FGPU and can be read or written from any work-item. Its address is limited to 32bit and hence its size to 4GB.

2.3 Platform Model

FGPU accommodates several *Compute Units (CUs)*, each holds a single array of *Processing Elements (PEs)* (see Figure 1). All member PEs in an array share the same program counter. The whole platform is controlled over a single 32bit AXI4 *Control Interface* while data can be transferred through multiple other AXI4 *Data Interfaces*.

To run a task on FGPU, its binary code has to be stored in the *Code RAM (CRAM)*. Other information that can not be determined at compile time, e.g. the number of work-items to be launched, linkage information to assign the parameter values, the address of the first instruction of the task in CRAM, all have to be stored in the *Link RAM (LRAM)*.

When the execution is started, the *Work-Group (WG) Dispatcher* begins to assign groups of up to 512 work-items to free CUs. A *Wavefront (WF) Scheduler* divides a WG further into multiple wavefronts, each of 64 work-items, and schedule them on the PE array. The *RunTime Memory (RTM)* is written whenever a WG or a WF is scheduled with all runtime-relevant information that may be accessed by the work-items during execution, e.g. the coordinates of each scheduled work-item in the index space and the parameter values. Reads and writes to global memory are handled by a local memory controller. CUs have no local memories or caches.

Memory operations issued by CUs are forwarded to a central memory controller (see Section 2.5). It includes an internal multi-bank direct-mapped cache. Accesses to the global memory are shaped in bursts and sent over a configurable number of AXI4 interfaces.

2.4 Compute Unit Architecture

2.4.1 Compute Vector

The 8 processing elements within a compute unit are placed into a compute vector module (see Figure 2). They are designed with a very deep pipeline of 18 stages for better timing performance. To overcome the high pipeline latency, a 2x faster clock is used for the compute vector with respect to the clock of other CU components. Each PE has 2048 registers to be used by 64 work-items, where each work-item

¹The term *work-item* is defined in the OpenCL execution model and it is equivalent to *thread* in CUDA terminology.

Figure 2: Compute unit overview

can access exclusively 32 registers. Physically, two dual-ported BRAMs are multiplexed in time to hold the register files. Any ALU operation may have up to three operands, which enables performing a multiply-and-accumulate (macc) within a single instruction. Executing any instruction is repeated over 8 clock cycles on all PEs, which corresponds to executing the same instruction from 64 work-items. Selecting the corresponding register file for the work-item under execution is achieved without extra latency through setting the address inputs of the BRAMs accordingly. Despite of the deep pipeline and even if an instruction depends on the results computed in a previous one, it is possible to execute the instructions within a wavefront after each other without inserting delays in between as long as no memory access is required.

2.4.2 Wavefront Scheduler

When a CU gets a work-group assigned by the WG dispatcher, the corresponding number of wavefronts have to be consequently scheduled and managed internally in the CU. On a single CU, it is possible to concurrently execute up to 8 WFs from different work-groups.

A WF has a single program counter and its instructions are executed in-order. After requesting a memory operation, the WF is placed on standby until its requests have been served. The WF scheduler is responsible of updating the program counters, fetching next instructions and waking up a WF when its memory requests have been served. It is also responsible of initializing the RTM before the execution begins. In order not to overrun the CU memory controller with 64 memory requests at once, a WF gets broken into quarter WFs when the execution reaches a memory operation. A quarter WF gets scheduled only when the CU memory controller is capable of handling new requests.

2.4.3 Runtime Memory

The RTM is a 2 ported-RAM with one write and one read 72bits ports. It is written either by the WG dispatcher when a WG gets assigned on the CU or by the WF scheduler when a wavefront is scheduled internally in the CU. It holds miscellaneous data which can only be determined at runtime, e.g. the global offset of scheduled work-groups or the local indices of work-items in their corresponding WGs. RTM can be read by work-items during execution by calling special assembly instructions. RTM enables realising the *Work-Item Built-In Function* defined in the OpenCL standard [10]. These functions can be linked to macros of assembly instructions that copy the corresponding content from RTM into the register files.

2.4.4 CU Memory Controller

Whenever a memory operation is executed on the compute vector, its outcomes get latched in a buffer. Then, the individual memory requests are handled by a vector of controllers or *stations*.

a) In case of a *write* operation: the requested address and the written data are forwarded into a FIFO (see Figure 2). The station can handle new requests directly after writing the FIFO.

b) In case of a *read* operation: after pushing the request into the FIFO, stations listen to the data read out from the cache. The cache has multiple banks and it serves several data words at once. If the address of the served data matches, the corresponding word is selected and latched. The *Write Back* module collects the read data from stations and writes them back into the register files.

Before pushing a new read request into the FIFO, a check is done whether the last waiting read request will serve the new one. In this case, the new request gets ignored. It is very probable that the work-items will access sequential addresses when reading or writing the global memory. Because the register files consists physically of two ram blocks and runs at double the frequency of the memory controller, the read data can be written back while the register files are being updated by other instructions without causing any congestion.

2.5 Memory Controller

FGPU has a central memory controller that can be connected to a single or multiple memory blocks over a configurable number of 32- or 64bit AXI4 interfaces (see Figure 3). To overcome the latency when reading the global memory, multiple read transactions with different IDs can be initiated and managed on the same AXI4 read channel. The controller can serve multiple CUs and it includes a direct-mapped data cache with one read and one write ports. Both ports spans over N data words. All read and write transaction to global memory are performed as bursts, where the burst size is the same of a cache block. To minimize data traffic to global memory, a write-back strategy is used. The controller includes a *Byte Dirty* memory to mark dirty bytes within a cache block.

A incoming memory request from a CU is caught by a free *Station*. Then, the station checks if the corresponding cache block has been already mapped to a physical location in the global memory (*Tag Valid Bit*), to which location it has been mapped (*Tag Address*) and if the cache block has been populated by data from global memory (*Block Valid Bit*):

Figure 3: Global Memory Controller

- In case of a *Cache Hit*:
 - For Write Operations: the write command enters the *Write Pipeline* and the dirty bit gets set for the updated block as well as the corresponding bits in the *Byte Dirty* memory.
 - For Read Operations: If the *Block Valid* bit is set, the cache can be read safely. Otherwise, a *Tag Manager* gets signaled to populate the cache block with the corresponding content from global memory.
- In case of a *Cache Miss*: the station asks a tag manager to allocate the corresponding tag.
 - For Write Operations: As soon as the tag memory is updated accordingly, the write command enters the *Write Pipeline*. The tag manager does not read the global memory to fill the cache block. This is done only for read operations.
 - For Read Operations: the tag manager checks if the cache block is dirty. Only in this case, cache content will be written back into the global memory and the corresponding *Dirty Bit* gets cleared. The content of the *Byte Dirty* memory enables to selectively write the dirty parts of the cache block. After that, the global memory is read and the cache block gets populated with the requested data.

Then, reads and writes can be executed on cache. Each outstanding request is assigned an increased priority as more time is spent on waiting. Reading stations compare the read address port of the cache with their own ones and retire when a match is detected. Since the cache read port spans over multiple data words, many stations can be served at once. Write requests have to go through the *Write Pipeline* before they get marked as served. The pipeline gives the necessary time to announce the address that is going to be written to other stations, check if any one is waiting to write the same cache entry, gather data and perform the write for as many stations as possible at once. When the tag value of a cache block is changed, all outstanding stations that

are waiting to read from the block will restart the whole procedure and go to the first step in serving their requests. To avoid race conditions when modifying tag values, every new tag is protected for a minimum time from being deallocated. The protection time should enable the corresponding outstanding stations to finish before they get restarted. After all work-items finish, the *Cache Cleaner* module checks the whole cache and writes back any dirty data it finds into global memory.

When a cache location is read and written by multiple CUs, the requests may be served in any order. Overwriting data that has been already written by some CUs when validating the content of a cache block is even possible. Anyway, the OpenCL standard does not enforce constraints on synchronising the memory accesses outside the borders of a CU². Serving a read request may take at least 8 clock cycles. When a cache block has to be cleaned and new data has to be read from global memory, a latency of over 50 clock cycles is typical in realistic situations³.

2.6 Instruction Set Architecture

FGPU has its own ISA. It can be divided into two parts: i) a *CPU-like* part whose instructions can be considered as a subset of MIPS ISA, and ii) an *OpenCL-relevant* part whose instructions are built to support the execution of unmodified OpenCL-kernels in future. These instructions are designed to replace OpenCL built-in functions. Figure 4 shows a simple FIR filter implemented in FGPU ISA next to an equivalent OpenCL implementation. In this example, the *get_global_id(0)* OpenCL function is replaced by the first three assembly instructions. To run the benchmarks presented in Section 4, it was enough to implement a total of 18 assembly instructions. All instructions have the same 32bit format.

²Global atomics enable avoiding race conditions when multiple CUs read and modify the same memory location. They will be supported and implemented as an extension to the current architecture in future work.

³This number does not take into account the response latency of the global memory slave AXI4 interface.

<pre>__kernel void fir(*input_array, *filter, int *res_array, int filter_len) { int index = get_global_id(0); int i = 0; int acc = 0; do { acc += input_array[indx+i] * filter[i]; i++; } while(i < filter_len); res[index] = acc; }</pre>	<pre># FIR filter using 1D index space. It has 4 Parameters: # 0 : address of the first element in input array # 1 : address of the first element in coefficients array # 2 : address of the first element in results array # 3 : filter length (L) LID r1, d0 # local ID: load the local work-item index in its work-group into r1 WGOFF r2, d0 # Work-Group Offset: load the work-group global offset into r2 ADD r1, r1, r2 # ADD integers: r1 has now the global id of the work-item LP r2, 3 # Load Parameter: r2 has filter length LP r3, 0 # Load Parameter: r3 is a pointer to the input array LP r4, # # Load Parameter: r4 is a pointer to the coefficients array ADDI r5, r0, 0 # ADD Immediate: r5 will be the loop index (initialized with 0) ADDI r6, r0, 0 # ADD Immediate: r6 will contain the result (initialized with 0) begin: LW r10, r4[r5] # Load Word: load a coefficient into r10 ADD r11, r5, r1 # ADD integers: calculate the index of an element in input array LW r11, r3[r11] # Load Word: load the input element into r11 MACC r6, r10, r11 # Multiply and ACCumulate: update the result ADDI r5, r5, 1 # ADD Immediate: update loop index BNE r5, r2, begin # Branch if Not Equal: repeat the iteration if necessary LP r20, 2 # Load Parameter: r20 is a pointer to the result array SW r6, r20[r1] # Store Word: store the result r6 into the index r1 in result array RET #RETurn: end of task</pre>
(a) FIR filter as OpenCL kernel	(b) Equivalent implementation in FGPU ISA

Figure 4: FIR filter in OpenCL and FGPU assembly

3. FGPU IMPLEMENTATION

3.1 Scalability

To synthesize a scalable GPU-like architecture on FPGAs, the following issues must be considered:

3.1.1 Implementing Register Files

Running more work-items simultaneously on the FPGA requires more register files to be synthesized. For example, 4K work-items with 32 registers of 32bits require 4M bits of RAM, or equivalently 64K 6-input LUTs. This corresponds to about a third of the available LUTs on the Xilinx z7045 FPGA, which has been used during development. It is possible to realize the same storage just with 64 modules or about 12% of the available BRAMs on the same FPGA. The main challenge when using BRAMs is getting the instruction operands out of a smaller number of access ports. Three read- and one write-operations must be performed per clock cycle on any register file. Therefore, we have doubled the clock frequency of the BRAMs that hold the register files and used two BRAMs in every PE. Hence, a single BRAM has to deliver three operands for the next ALU instruction every four “fast” clock cycles. Considering the two BRAMs, the needed number of operands for a single instruction can be read out per single “slow” clock cycle with a latency of another “slow” clock cycle.

3.1.2 Using DSP Blocks

Using DSP blocks that are available on most modern FPGAs would spare a significant amount of LUTs and improve the operating frequency as well. These blocks are capable of performing integer multiplication, addition or even logical shift. To extend these operations on 32bit words, multiple DSP blocks have to be chained and signals that go through the DSPs will form the most critical paths. This problem is avoided in FGPU by using the additional pipeline registers inside and at the borders of the DSP blocks. Five pipeline stages are used when performing 32bit multiplication. To

reduce the additional pipeline latency, DSP blocks operate in the same double-frequency domain as the register files.

3.1.3 Preserving the Operating Frequency

To mitigate the degradation in the operating frequency as the design gets bigger, many FPGA-specific techniques have been applied, e.g. minimizing the use of reset signals and activating the optional output registers of BRAM blocks. Since two different clocks have been used, the most critical paths after place and route were the ones that cross over the borders of any clock domain. Therefore, additional pipeline stages have been added to remove any logic located on the crossing paths. Sometimes empty pipeline stages have been inserted in the same clock domain just before crossing to the other one. High-fanout signals have been determined and limited through multiple manual place and route iterations. Slice registers have been placed on some signals, e.g. the read data bus driven by the central memory controller which has to reach all CUs that are expected to be placed apart from each other over the whole FPGA chip.

3.2 Portability

Although FGPU architecture targets specific FPGA resources like DSP or byte-enabled true dual-port BRAM blocks, no IP-cores or primitives have been used. But a successful and efficient implementation on any FPGA depends primarily on the ability of the manufacturer’s synthesis tools to infer the targeted blocks from the VHDL code. Although FGPU has been synthesized and tested only for the 7series from Xilinx, it is absolutely possible to implement it on other FPGA families without modifying the RTL Code. But a closer look to the synthesis outcomes is recommended when porting the design to other types of FPGAs for the first time. Fanout limitations are included in the RTL code as signal attributes and they are manufacturer-dependent⁴. They may need to be adjusted when targeting other FPGA chips

⁴Synthesis tools ignore VHDL attributes that could not be interpreted. This enables having multiple manufacturer-dependent ones in the same design.

Figure 5: Development Platform

from the same manufacturer to achieve better performance. Anyway, they are optional and they do not affect the portability at all.

3.3 Flexibility

FGPU is a highly customizable architecture and offers a large space for design exploration. The user can customize the design to fit to the available FPGA resources. All parameters have to be determined before synthesis in a VHDL-package file. To generate the results presented in this paper, we focused on studying the parameters of direct influence on scalability. Other non scalability-critical ones have been fixed on realistic and near-optimal values found during behavioral simulations, e.g. the size of a cache blocks or the number of tag managers in the central memory controller.

4. RESULTS

4.1 Development Platform

The ZC706 FPGA board with z7045 Zynq has been used for development (see Figure 5). The on-chip ARM Cortex-A9 processor includes two cores: one is used to control FGPU and the other for power measurements. The on-board 1G DDR3-SDRAM PS memory is considered as a global memory. FGPU accesses the DDR through the four AXI HP (High Performance) ports of the ARM.

All applications are programmed in FGPU ISA and the binaries are automatically generated with a special tool developed in C++. To optimize FGPU architecture and discover bottlenecks and bugs, we depended on cycle accurate simulations using Questa Sim v10.4. A realistic and highly customizable model for the global memory with a adjustable number of AXI4 interfaces is developed and integrated. The simulation platform offers many statistical measurements

Table 1: Area requirements for different configurations

	LUTs	FFs	BRAM	DSPs
Available	219K	437K	545	900
8 CUs	124K (57%)	159K (36%)	167 (31%)	192 (21%)
4 CUs	76K (35%)	88K (20%)	99 (18%)	96 (11%)
2 CUs	57K (26%)	60K (14%)	65 (12%)	48 (5.3%)
2 CUs (min)	21K (9.6%)	35K (8.1%)	47 (8.5%)	48 (5.3%)
MicroBlaze	7K (3.2%)	5.6K (1.3%)	27 (5.0%)	6 (0.7%)

Table 2: Adjustable parameters with direct influence on design scalability

Module	Parameter	Range
CU	# CUs	2, 4, 8
	# Outstanding mem. requests	16, 24, 32
Memory Controller	Cache Size	1 to 8KB
	# Cache Read Banks	2, 4, 8
	# Outstanding mem. requests	32, 64
	# AXI4 interfaces	1, 2, 4
	# Tag Managers	2, 4, 8, 16

that we have used to make architectural decisions. In addition, it enables testing any application intensively with different settings and offers automated check for the correctness and the completeness of the data that must be written back in the global memory. This check is also done after a task is executed on the hardware platform.

The CRAM has 16KB of storage and its content is included in the bitstream. The LRAM size is set at 4KB which is enough to hold the settings of 16 kernels. Its content has been defined in an XML file and integrated in the bitstream but it was modified at runtime to change the task size. The conversion from the XML representation to binary has been automated with awk script. FGPU has two 16bit control registers, namely *Start* and *Finish*. By setting the i^{th} bit in the start register, the kernel at index i in LRAM will be launched. When it finishes, the corresponding bit in the finish register gets set. The execution time on FGPU is measured between setting the start register and reading the corresponding value from the finish register.

4.2 Area Requirements

Table 1 gives an overview of the needed resources when FGPU is synthesized with 8, 4 and 2 CUs. All other parameters listed in Table 2 are configured for maximum performance. The entry “2 CUs (Min)” indicates the case where all parameters are configured for minimum resource usage. All implementations could be successfully placed and routed without frequency degradation at 200 and 400MHz for the normal and double clock domains, respectively. This

Figure 6: Wall clock time speedup for 8 CUs over ARM+NEON implementation for variable task size

Figure 7: Wall clock time speedup for FGPU with different number of CUs over MicroBlaze implementation for a task size of 16K

is very close to the practical maximum operating frequency of BRAMs on the used FPGA which is 454MHz [24]. The most critical resources when implementing FGPU are the LUTs. Anyway, this applies only to the 7series FPGAs from Xilinx, which have 2 FFs per LUT. When implementing FGPU on older FPGA families where this ratio becomes 1 FF per LUT, the most critical resources will be the FFs.

4.3 Speedup

The benchmark used in this work includes 7 typical application for GPGPU computing that have been considered in related work. To measure the efficiency of the designed memory controller, the *memcpy* task has been added. An equivalent software implementation of the benchmark in C++ is compiled and executed as bare-metal application on two architectures: a single hard ARM core with the NEON vector engine and a soft MicroBlaze processor. The ARM CPU is clocked at 667MHz and its cache system is enabled. The

Figure 8: Wall clock time speedup for 8 CUs over MicroBlaze implementation for task size between 256 and 256K

NEON SIMD has been always used, where auto vectorization is performed by the compiler. The reference MicroBlaze is configured by the default settings for maximum performance and it runs at 185MHz. The ARM as well as the MicroBlaze compilers are configured for maximum optimization -O3 and no debug symbols are generated. The FGPU and the reference MicroBlaze are synthesized using Vivado 2015.2 with a performance-oriented strategy for synthesis, placement and routing. We varied the problem size from 64 to 256K integers. The size of work-groups was set on 64 work-items. Similar to any other GPU architecture, the work-group size should be a multiple of the wavefront size to get optimal performance on FGPU. All time measurements are repeated and averaged over 10 runs and they include flushing the content of the written cache region after the execution ends. FGPU performs cache flushing automatically when all work-items retire.

Figure 6 illustrates the relationship between the speedup and the task size for some considered applications. To execute a task on FGPU, a minimum execution time of 4 us is needed for the initialization of RTM memories at the beginning and flushing the dirty cache content at the end. When taking the ARM with the NEON engine as a reference, a minimum task size of 256 is needed to achieve any speedup with FGPU. The effect of the number of CUs on speedup is depicted in Figure 7. Even for applications that are less computationally intensive, remarkable improvements can be achieved by using more CUs. This is because of the improved throughput to the global memory. The memory operations generated by a single CU usually target memory addresses with minimum stride. As more CUs are involved, much more cache blocks have to be updated and hence the central memory controller has more requests to serve. The latency for populating a cache block or cleaning it will have less effect on the overall performance.

Figure 8 shows the wall clock time speedup of FGPU with 8 CUs over MicroBlaze on the whole benchmark. After averaging all speedups for task sizes from 256 to 256K and then taking the average over the whole benchmark, a speedup of

Figure 9: Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K

49x could be achieved with FGPU. The minimum accelerations are measured for applications like vector additions or FIR filtering. For the most computational intensive tasks, speedups up to 105x for cross correlation and 166x for matrix multiplication have been measured. The achieved speedup improves as the ratio of the number of performed arithmetic operations to the required number of memory accesses increases. Even when FGPU is used like a DMA, a bandwidth up to 7.7Gbps could be achieved, which is 66x faster than the MicroBlaze implementation. When compared to the ARM+NEON implementation, FGPU managed to provide remarkable improvements with similar relative changes over the different applications of the benchmark (see Figure 9). FGPU could provide an average speedup of 3.5x over the whole benchmark with a maximum of 35x when multiplying matrices of 512x512 integers.

4.4 Power Consumption

The power measurements have been done via the Texas Instruments UCD90120A power-supply sequencer and monitor on the Xilinx ZC706. The measurements were done on the second core of the ARM processor using the PMBus protocol, while the first core controlled task execution on FGPU. The voltage and current values of all supply rails have been periodically sampled. Because measuring one value pair takes about 1.7ms, we set the task size at maximum during power measurements and repeated the task execution many times when necessary to make at least 500 samples. The two ARM cores synchronized the measurement procedure with task execution over a flag located in the DDR memory. All reported power measurements are averaged over the whole amount of taken samples. Because the power consumed by the PL represents only a part of the measured one, the following trick was used: we measured the power consumed when PL is not programmed, the first ARM core is idle, and only the power measurement runs on the second core. Then, we subtracted the value we got from all future measurements. Figure 10 shows the ratio of the power consumed by different FGPU implementations to the

Figure 10: Measured and estimated ratios of the power consumption of different FGPs to the one consumed by MicroBlaze over the benchmark

one consumed by MicroBlaze over the whole benchmark. In average, the biggest and smallest FGPs consumed 13.0x and 3.3x more power than MicroBlaze, respectively. If the estimated power consumptions for the different designs after place & route with Vivado are considered, the previous ratios should have been be at 4.6x and 1.7x, respectively (see Figure 10). Absolute values for the measured consumed power through the UCD90120A chip were 5.19W for the biggest FGPU, 1.94W for the smallest one and 1.18W for the MicroBlaze. Despite of the high power consumption of FGPU, taking the speedups into account leads to the fact that FGPU is capable of providing a minimum power saving of 3.2x over MicroBlaze. The power measurements of the ARM+NEON implementation could not be made because the PS power supply is not included completely in the UCD90120A chip.

Figure 11 illustrates a comparison of many FGPU implementations and MicroBlaze in different aspects. Since LUTs are the most critical resource for all architectures when implemented on the z7045 FPGA, their usage is considered as a metric to compare area requirements of all designs. Even in the worst cases, FGPU was in average at least as fast as the ARM+NEON solution.

5. RELATED WORK

5.1 Soft GPGPUs

Because of their complexity and in contrast to soft CPUs, there have been quite few attempts to implement GPUs as configurable or application-specific soft cores on FPGAs for general purpose computing. In [2], a soft GPU based on the LEON3 processor was presented and tested only for matrix multiplication. Speedups up to 3x were achieved over the original LEON3 implementation. Andryc, *et al.* [5] tried to replicate the Nvidia G80 architecture⁵ on a Virtex6 FPGA in the FlexGrip project. The reported implementation is capable of executing 27 integer CUDA instructions of compute

⁵G80 is the first generation architecture of CUDA-programmable GPUs released in 2006 [14].

Figure 11: Average wall clock time speedup, power saving and area overhead for different FGPU over MicroBlaze and ARM+NEON implementations

capability 1.0 and supports thread-level branching in hardware. On the VLX240T FPGA, it was possible to synthesize 8, 16, or 32 scalar processors (SPs) at 100 MHz organized in a single compute unit where all SPs execute the same instruction at once. Instructions and data were accessed by FlexGrip through a single 32bit AXI4-interface connected to a MicroBlaze processor. Taking the performance of the MicroBlaze processor as a reference, FlexGrip was capable of achieving an average speedup of 22x in simulation for 32 SPs over a benchmark of 5 applications. The best simulated speedup was 29x for matrix multiplication and the worst one was about 11x for autocorrelation. The best average power saving was %66 for the 32 SP implementation.

MIAOW is an open source GPGPU [7] which introduces a similar architecture to the southern Islands from AMD and uses its ISA [4]. It targets a hybrid implementation: register files, on-chip networks and memory controllers are provided as behavioral C/C++ modules while the rest is implemented in RTL. MIAOW is capable of running many unmodified OpenCL benchmarks. The developed compute unit has been synthesized with 32nm technology⁶. It occupies 15mm² and can run at 222MHz. The developers reported an FPGA-implementation of MIAOW, named *Neko*: It consists of a single compute unit next to a MicroBlaze on a Virtex7 FPGA. *Neko*'s CU has 16 floating-point ALUs with no memory controller. The MicroBlaze has to schedule wavefronts on the CU and perform memory accesses. *Neko* needs 195285 LUTs and 137 BRAMs for the whole CU. No timing informations or test results have been reported. To the best of our knowledge, there has been no successful attempt to synthesize a soft GPU with multiple compute units on a single FPGA.

⁶A compute unit in MIAOW accommodates 4 vectors of 16 floating-point ALUs with 1024 vector- and 512 scalar-registers.

5.2 Soft Vector Processors

In contrast to soft GPGPUs, soft vector processor architectures have been extensively studied in many projects. The VENICE architecture [20] implements up to 4 ALUs next to a NIOS soft processor. An average speedup of 21x and up to 72x have been reported @190MHz on a Stratix IV device for the biggest design. In VESPA [26], a MIPS with a soft vector processor has been implemented. An average speedup of 11.3x was achieved with 32 ALUs @ 96MHz on Stratix III. MXP [21] has a similar architecture to VENICE but it is more scalable and it can run at higher frequencies. The authors reported estimated average cycle count speedup of 116x with 64 ALUs and up to 918x for matrix multiplication while taking a NIOS II/f as reference. But wall clock time speedups drop significantly because the operating frequency degrades from 283 for the NIOS to 122MHz for the biggest MXP on a Stratix IV device. On a similar development platform that we have used during FGPU development [9], MXP could achieve a maximum speedup of 2.07x over the ARM+NEON implementation for similar 32bit applications that we have targeted in the FGPU benchmark.

To parallelize data handling on vector processors, intrinsics or architecture-dependent code have to be inserted into the application code [19] [21]. Automatic vectorization can be performed by the compiler but it usually delivers suboptimal results [26]. On the other side, FGPU as well as GPU programmers distribute the workload for different problem sizes on a configurable number of work-items. They do not need to think about the underlying hardware structure. Scheduling the work-items on the different CUs is done at runtime. Register files of all work-items are stored in hardware and context switching is possible on-the-fly. In addition, FGPU does put and workload on other system components and it manages all data transfers by itself. Vector processors usually share data and instruction caches with a scalar CPU. Many soft vectors use a multi-bank scratchpad for temporal storage of the data under processing. The programmer should manually initiate DMA transfers to get the data from the main memory into the scratchpad while processing other pieces of information [20] [21].

6. CONCLUSION

FGPU is a scalable, portable, flexible and highly customizable SIMT soft processor architecture designed specially for FPGAs. Its platform and execution models as well as its ISA architecture are inspired from the OpenCL standard and it can be programmed through a well defined interface. FGPU includes a multi-bank and direct-mapped cache system which has been optimized to serve many outstanding read and write requests at once. It can drive multiple 32- or 64bit AXI4 interfaces to fetch data from global memory at maximum throughput. By taking the average over the whole presented benchmark and over all FGPU configurations that have been presented in this paper, power savings between 3.2x and 4.5x simultaneously with speedups between 10.6x and 48.5x have been achieved over a MicroBlaze implementation on the same FPGA. The penalty in area overhead was between 3.0x and 17.7x. Compared to an equivalent ARM implementation with the NEON SIMD engine, the smallest FGPU core could achieve the same performance while the biggest one could provide a speedup of 3.5x in average.

7. FUTURE WORK

We intend to extend the ISA of FGPU to cover more benchmarks. Enabling branches at the work-item level would be necessary to implement algorithms like reduction and sorting. A very important milestone is providing a compiler for OpenCL kernels by developing a backend for the LLVM framework. Implementing local memories for the CUs as well as local and global atomics are planned for far future. Integrating the floating point DSPs that are available in some modern FPGAs within the proposed architecture would extend the targeted application domain significantly.

8. REFERENCES

- [1] Top500 Consortium: The Top 500 supercomputing sites. <http://www.top500.org> [Online; accessed 11-Sep-2015].
- [2] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and W.-F. Wong. Guppy: A GPU-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 57–60, Dec 2012.
- [3] Altera Corporation. Implementing FPGA Design with the OpenCL Standard. White Paper, 2013.
- [4] AMD, Inc. Southern Islands Series Instruction Set Architecture. Reference Guide, 2012.
- [5] K. Andryc, M. Merchant, and R. Tessier. FlexGrip: A soft GPGPU for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 230–237, Dec 2013.
- [6] ARM Limited. Take GPU Processing Power Beyond Graphics with Mali GPU Computing. White Paper, 2014.
- [7] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam. Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU. *ACM Trans. Archit. Code Optim.*, 12(2):21:21:1–21:21:25, June 2015.
- [8] D. Capalija and T. Abdelrahman. Tile-based bottom-up compilation of custom mesh of functional units FPGA overlays. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.
- [9] S. J. Jie and N. Kapre. Comparing soft and hard vector processing in FPGA-based embedded systems. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–7, Sept 2014.
- [10] Khronos Group. OpenCL 1.2 Specification, 2012.
- [11] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynnek. MARC: A Many-Core Approach to Reconfigurable Computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7–12, Dec 2010.
- [12] S. Ma, M. Huang, and D. Andrews. Developing application-specific multiprocessor platforms on FPGAs. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–6, Dec 2012.
- [13] Mathworks, Inc. GPU Programming in MATLAB. Available on mathworks.com/newsletters [Online; accessed 11-Sep-2015].
- [14] Nvidia Corporation. Nvidia’s Next Generation CUDA Compute Architecture: Fermi. White Paper, 2009.
- [15] Nvidia Corporation. Nvidia Jetson TK1 Development Kit: Bringing GPU-accelerated computing to Embedded Systems (Technical Brief, v1.0), 2014.
- [16] M. Owaid, N. Bellas, K. Daloukas, and C. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193, May 2011.
- [17] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Application Specific Processors, 2009. SASP ’09. IEEE 7th Symposium on*, pages 35–42, July 2009.
- [18] R. Rashid, J. Steffan, and V. Betz. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 20–27, Dec 2014.
- [19] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft Vector Processors with Streaming Pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA ’14, pages 117–126, New York, NY, USA, 2014. ACM.
- [20] A. Severance and G. Lemieux. VENICE: A Compact Vector Processor for FPGA Applications. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 245–245, April 2012.
- [21] A. Severance and G. Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.
- [22] K. Shagrirhaya, K. Kepa, and P. Athanas. Enabling development of opencl applications on fpga platforms. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 26–30, June 2013.
- [23] Xilinx, Inc. UltraScale Architecture and Product Overview (v2.2), DS890, 2014.
- [24] Xilinx, Inc. Block Memory Generator v8.2, LogiCORE IP Product Guide (PG058), 2015.
- [25] P. Yiannacouras, J. Steffan, and J. Rose. Exploration and Customization of FPGA-Based Soft Processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):266–277, Feb 2007.
- [26] P. Yiannacouras, J. Steffan, and J. Rose. Portable, Flexible, and Scalable Soft Vector Processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(8):1429–1442, Aug 2012.