Vortex: OpenCL Compatible RISC-V GPGPU

***Abstract*—**

The current challenges in technology scaling are pushing the semiconductor industry towards hardware specialization, creating a proliferation of heterogeneous systems-on-chip, delivering orders of magnitude performance and power benefits compared to traditional general-purpose architectures. This transition is getting a significant boost with the advent of RISC-V with its unique modular and extensible ISA, allowing a wide range of low-cost processor designs for various target applications. In addition, OpenCL is currently the most widely adopted programming framework for heterogeneous platforms available on mainstream CPUs, GPUs, as well as FPGAs and custom DSP.

In this work, we present Vortex, a RISC-V General-Purpose GPU that supports OpenCL. Vortex implements a SIMT architecture with a minimal ISA extension to RISC-V that enables the execution of OpenCL programs. We also extended OpenCL runtime framework to use the new ISA. We evaluate this design using 15nm technology. We also show the performance and energy numbers of running them with a subset of benchmarks from the Rodinia Benchmark suite.

***Index Terms*—GPGPU, OpenCL, Vector processors**

I. Introduction

The emergence of data parallel architectures and general purpose graphics processing units (GPGPUs) have enabled new opportunities to address the power limitations and scalability of multi-core processors, allowing new ways to exploit the abundant data parallelism present in emerging big-data parallel applications such as machine learning and graph analytics. GPGPUs in particular, with their Single Instruction Multiple-Thread (SIMT) execution model, heavily leverage data-parallel multi-threading to maximize throughput at relatively low energy cost, leading the current race for energy efficiency (Green500 [12]) and applications support with their accelerator-centric parallel programming model (CUDA [19] and OpenCL [17]).

The advent of RISC-V [2], [20], [21], open-source and free instruction set architecture (ISA), provides a new level of freedom in designing hardware architectures at lower cost, leveraging its rich eco-system of open-source software and tools. With RISC-V, computer architects have designed several innovative processors and cores such as BOOM v1 and BOOM v2 [4] out-of-order cores, as well as system-on-chip (SoC) platforms for a wide range of applications. For instance, Gautschi et al. [9] have extended RISC-V to digital signal processing (DSP) for scalable Internet-of-things (IoT) devices. Moreover, vector processors [22] [15] [3] and processors integrated with vector accelerators [16] [10] have been designed and fabricated based on RISC-V. In spite of the advantages of the preceding works, not enough attention has been devoted to building an open-source general-purpose GPU (GPGPU) system based on RISC-V.

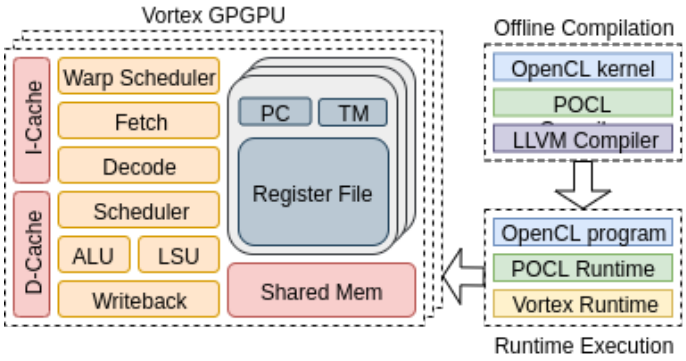


Fig. 1: Vortex System Overview

Although a couple of recent work have been proposed for massively parallel computations on FPGA using RISC-V, (GRVI Phalanx) [11], (Simty) [6], none of them have implemented the full-stack by extending the RISC-V ISA, synthesizing the microarchitecture, and implementing the software stack to execute OpenCL programs. We believe that such an implementation is in fact necessary to achieve the level of usability and customizability in massively parallel platforms.

In this paper, we propose an ISA RISC-V extension for GPGPU programs and microarchitecture. We also extend a software stack to support OpenCL.

This paper makes the following key contributions:

- We propose a highly configurable SIMT-based General Purpose GPU architecture targeting the RISC-V ISA and synthesized the design using a Synopsys library with our RTL design.
- We show that the minimal set of five instructions on top of RV32IM (RISC-V 32 bit integer and multiply extensions) enables SIMT execution.
- We describe the necessary changes in the software stack that enable the execution of OpenCL programs on Vortex. We demonstrate the portability by running a subset of Rodinia benchmarks [5].

II. Background

A. Open-Source OpenCL Implementations

POCL [13] implements a flexible compilation backend based on LLVM, allowing it to support a wider range of device targets including general purpose processors (e.g. x86, ARM, Mips), General Purpose GPU (e.g. Nvidia), and TCE-based processors [14] and custom accelerators. The custom accelerator support provides an efficient solution for enabling OpenCL applications to use hardware devices with specialized fixed-function hardware (e.g. SPMV, GEMM). POCL is comprised

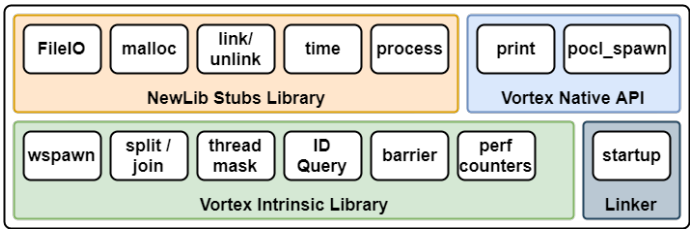


Fig. 2: Vortex Runtime Library

of two main components: A back-end compilation engine, and a front-end OpenCL runtime API.

The POCL runtime implements the device-independent common interface where the target implementation of each device plugs into POCL to specialize their operations. At runtime, POCL invokes the back-end compiler with the provided OpenCL kernel source. POCL supports target-specific execution models including SIMT, MIMD, SIMD, and VLIW. On platforms supporting MIMD and SIMD execution models such as CPUs, the POCL compiler attempts to pack as many OpenCL work-items to the same vector instruction, then the POCL runtime will distribute the remaining work-items among the active hardware threads on the device with provided synchronization. On platforms supporting SIMT execution model such as GPUs, the POCL compiler delegates the distribution of the work-items to the hardware to spread the execution among its hardware threads, relying on the device to also handle the necessary synchronization. On platforms supporting VLIW execution models such as TCE-based accelerators, the POCL compiler attempts to "unroll" the parallel regions in the kernel code such that the operations of several independent work-items can be statically scheduled to the multiple function units of the target device.

III. The OpenCL Software Stack

A. Vortex Native Runtime

The Vortex software stack implements a native runtime library for developing applications that will run on Vortex and take advantage of the new RISC-V ISA extension. Figure 2 illustrates the Vortex runtime layer, which is comprised of three main components: 1) Low-level intrinsic library exposing the new ISA interface, 2) A support library that implements NewLib stub functions [7], 3) a native runtime API for launching POCL kernels.

1) Instrinsic Library: To enable Vortex runtime kernel to utilize the new instructions without modifying the existing compilers, we implemented an intrinsic layer that implements the new ISA. Figure 2 shows the functions and ISA supported by the intrinsic library. We leverage RISC-V's ABI which guarantees function arguments being passed through the argument registers and return values begin passed through *a0* register. Thus, these intrinsic functions have only two assembly instructions: 1) The encoded 32-bit hex representation of the instruction that uses the argument registers as source registers, and 2) a return instruction that returns back to the C++

```
1  vx_intrinsic.s
2  vx_split:
3    .word 0x0005206b     # split a0
4    ret
5  vx_join:
6    .word 0x0000306b     #join
7    ret
8
9  kernel.cl
10 #define __if(cond) split(cond); \
11           if(cond)
12
13 #define __endif join();
14 void opencl_kernel(){
15     int id = vx_getTid();
16     __if(id<4) {
17         // Path A
18     } else {
19         // Path B
20     } __endif
21 }
```

Fig. 3: This figure shows the control divergent __if __endif macro definitions and how they could be used to enable control divergence in OpenCL kernels. Currently, this process is done manually for each kernel.

program. An example of these intrinsic functions is illustrated in Figure 3. In addition, to handle control divergence, which is frequent in OpenCL kernels, we implement __if and __endif macros shown in Figure 3 to handle the insertion of these intrinsic functions with minimal changes to the code. These changes are currently done manually for the OpenCL kernels. This approach achieves the required functionality without restricting the platform or requiring any modifications to the RISC-V compilers.

2) Newlib Stubs Library: Vortex software stack uses the NewLib [7] library to enable programs to use the C/C++ standard library without the need to support an operating system. NewLib defines a minimal set of stub functions that client applications need to implement to handle necessary system calls such as file I/O, allocation, time, process, etc..

3) Vortex Native API: The Vortex native API implements some general purpose utility routines for applications to use. One of such routines is *pocl_spawn()* which allows programs to schedule POCL kernel execution on Vortex. *pocl_spawn()* is responsible for mapping work groups requested by POCL to the hardware: 1) It uses the intrinsic layer to find out the available hardware resources, 2) Uses the requested work group dimension and numbers to divide the work equally among the hardware resources, 3) For each OpenCL dimension, it assigns a range of IDs to each available warp in a global structure, 4) It uses the intrinsic layer to spawn the warps and activate threads, and finally 5) Each warp will loop through the assigned IDs, executing the kernel every time with a new OpenCL *global_id*. Figure 4 shows an example. In the original OpenCL code, the *kernel* is called once with global/local sizes as arguments. POCL wraps the kernel with three loops and sequentially calls with the logic that converts x,y,z to global ids. For a Vortex version, warps and threads are spawned, then each thread is assigned a different work-group to execute the

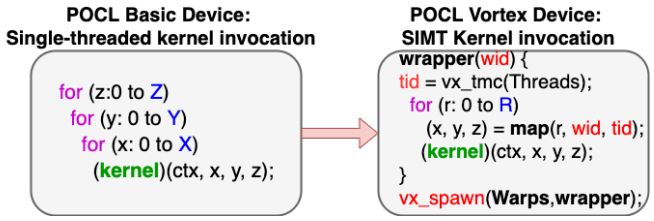


Fig. 4: SIMT Kernel invocation modification for Vortex in POCL

kernel. POCL provides the feature to map the correct *wid*, which was a part of the baseline POCL implementation to support various hardware such as vector architecture.

B. POCL Runtime

We modified the POCL runtime, adding a new device target to its common device interface to support Vortex. The new device target is essentially a variant of the POCL basic CPU target with the support for pthreads and other OS dependencies removed to target the NewLib interface. We also modified the single-threaded logic for executing work-items to use Vortex's *pocl_spawn* runtime API.

C. Barrier Support

Synchronizations within work-group in OpenCL is supported by barriers. POCL's back-end compiler splits the control-flow graph (CFG) of the kernel around the barrier and splits the kernel into two sections that will be executed by all local work-groups sequentially.

IV. Vortex Parallel Hardware Architecture

A. SIMT Hardware Primitives

SIMT, Single Instruction-Multiple Threads, execution model takes advantage of the fact that in most parallel applications, the same code is repeatedly executed but with different data. Thus, it provides the concept of Warps [19], which is a group of threads that share the same PC and follows the same execution path with minimal divergence. Each thread in a warp has a private set of general purpose registers, and the width of the ALU is matched with the number of threads. However, the fetching, decoding, and issuing of instructions is shared within the same warp which reduces execution cycles.

However, in some cases, the threads in the same warp will not agree on the direction of branches. In such cases, the hardware must provide a thread mask to predicate instructions for each thread, and an IPDOM stack, to ensure all threads execute correctly, which are explained in Section IV-C.

B. Warp Scheduler

The warp scheduler is in the fetch stage which decides what to fetch from I-cache as shown in Figure 5. It has two components: 1) A set of warp masks to choose the warp to schedule next, and 2) a warp table that includes private information for each warp.

There are 4 thread masks that the scheduler uses: 1) an active warps mask, one bit indicating whether a warp is active or not, 2) a stalled warp mask, which indicates which warps should not be scheduled temporarily (e.g., waiting for a memory request), 3) a barrier warps stalled mask, which indicates warps that have been stalled because of a barrier instruction, and 4) a visible warps mask to support hierarchical scheduling policy [18].

Each cycle, the scheduler selects one warp from the visible warp mask and invalidates that warp. When visible warp mask is zero, the active mask is refilled by checking which warps are currently active and not stalled.

An example of the warp scheduler is shown in Figure 6(a). This figure shows the normal execution; The first cycle warp zero executes an instruction, then in the second cycle warp zero is invalidated in the visible warp mask, and warp one is scheduled. In third cycle, because there are no more warps to be scheduled, the scheduler uses the active warps to refill the visible mask, and schedules warp zero for execution.

Figure 6(b) shows how the warp scheduler handles a stalled warp. In the first cycle the scheduler schedules warp zero. In the second cycle, the scheduler schedules warp one. At the same cycle, because the decode stage identified that warp zero's instruction requires a change of state, it stalls warp zero. In the third cycle, because warp zero is stalled, the scheduler only sets warp one to visible warp mask and schedules warp one again. When warp zero updates its thread mask, the bit in stalled mask will be set to 0 to allow scheduling.

Figure 6(c) shows an example of spawning warps. When warp zero executes a *wspawn* instruction (Table I) , which activates warps and manipulates the active warps mask by setting warps two and three to be active. When it's time to refill the visible mask, because it no longer has any warps to schedule, it includes warps two and three. Warps will stay in the Active Mask until they set their thread mask's value to zero, or warp zero utilizes wspawn to deactivate these warps.

C. Threads Masks and IPDOM Stack

To support the thread concepts provided in Section IV-A, a thread mask register and an IPDOM stack have been added to the hardware similar to other SIMT architectures [8]. The thread mask register acts like a predicate for each thread, controlling which threads are active. If the bit in the thread mask for a specific thread is zero, no modifications would be made to that thread's register file and no changes to the cache would be made based on that thread.

The IPDOM stack, illustrated in Figure 5 is used to handle control divergence and is controlled by the split and join instructions. These instructions utilize the IPDOM stack to enable divergence as shown in Figure 3.

When a split instruction is executed by a warp, the predicate value for each thread is evaluated. If there is only one thread active, or all threads agree on a direction, the split acts like a nop instruction and does not change the state of the warp. When there is more than one active thread that contradicts on the value of the predicate, three microarchitecture events

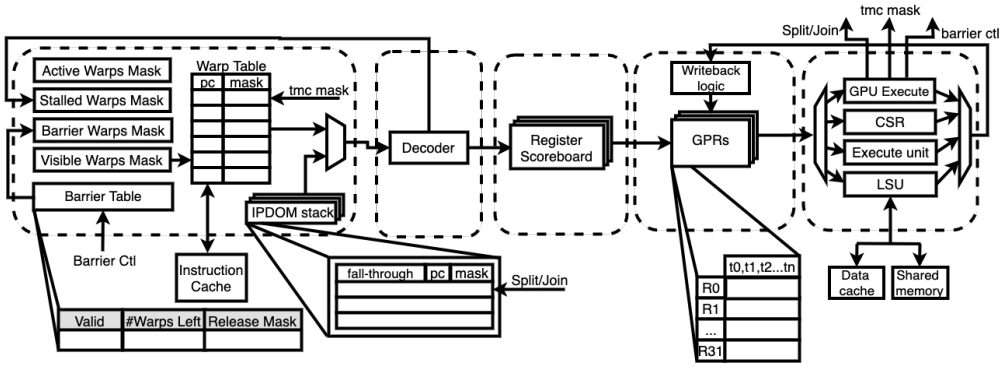


Fig. 5: Vortex Microarchitecture.

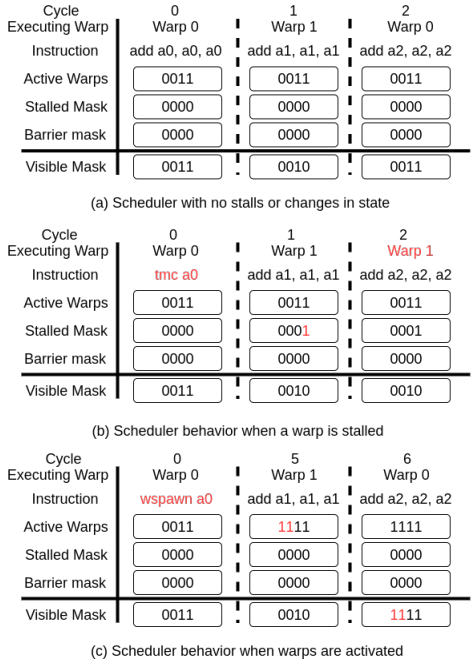


Fig. 6: This figure shows the Warp Scheduler under different scenarios. In the actual microarchitecture implementation, the instruction is only known the next cycle, however it's displayed in the same cycle in this figure for simplicity.

occur: 1) The current thread mask is pushed into the IPDOM stack as a fall-through entry, 2) The active threads that evaluate the predicate as false are pushed into the stack with PC+4 (i.e., size of instruction) of the split instruction, and 3) The current thread mask is updated to reflect the active threads that evaluate the predicate to be true.

When a join instruction is executed, an entry is popped out of the stack which causes one of two scenarios: 1) If the entry is not a fall-through entry, the PC is set to the entry's PC and the thread mask is updated to the value of the entry's mask, which enables the threads evaluating the predicate as false to follow their own execution path, and 2) If the entry is a fall-through entry, the PC continues executing to PC+4 and the thread mask is updated to the entry's mask, which is the case when both paths of the control divergence have been executed.

D. Warp Barriers

Warp barriers are important in SIMT execution, as it provides synchronization between warps. Barriers are provided in the hardware to support global synchronization between work-groups. Each barrier has a private entry in barrier table, shown in Figure 5, with the following information: 1) Whether that barrier is currently valid, 2) the number of warps left that need to execute the barrier instruction with that entry's ID for the barrier to be released, and 3) a mask of the warps that are currently stalled by that barrier. However, Figure 5 only shows the per core barriers. There is also another table on multi-core configurations that allows for global barriers between all the cores. The MSB of the barrier ID indicates whether the instruction uses local barriers or global barriers.

When a barrier instruction is executed, the microarchitecture checks the number of warps executed with the same barrier ID. If the number of warps is not equal to one, the warp is stalled until that number is reached and the release mask is manipulated to include that warp. Once the same number of warps have been executed, the release mask is used to release all the warps stalled by the corresponding barrier ID. The same method works for both local and global barriers; however, global barrier tables have a release mask per each core.

TABLE I: Proposed SIMT ISA extension.

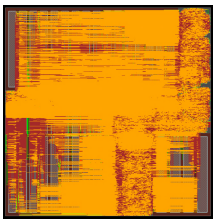
Instructions	Description
wspawn %numW, %PC	Spawn W new warps at PC
tmc %numT	Change the thread mask to activate threads
split %pred	Control flow divergence
join	Control flow reconvergence
bar %barID, %numW	Hardware Warps Barrier

V. Evaluation

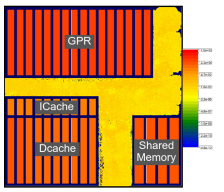
This section will evaluate both the RTL Verilog model for Vortex and the software stack.

A. Micro-architecture Design space explorations

In Vortex design, we can increase the data-level parallelism either by increasing the number of threads or by increasing the



(a) GDS Layout



(b) Power density distribution across Vortex

Fig. 7: GDS layouts for our Vortex with 8 warp, 4 thread configuration (4KB register file). The design was synthesized for 300Mhz and produced a total power output of 46.8mW. 4KB 2 ways 4 banks-data cache, 8KB with 4 banks-shared memory, 1Kb 2 way cache, one bank-I cache.

number of warps. Increasing the number of threads is similar to increasing the SIMD width and involves the following changes to the hardware: 1) Increasing the GPR memory width for reads and writes, 2) Increasing the number of ALUs to match the number of threads, 3) increasing the register width for every pipeline stage after the GPR read stage, 4) increasing the arbitration logic required in both the cache and the shared memory to detect bank conflicts and handle cache misses, and 5) increasing the number of IPDOM enteries.

Whereas, increasing the number of warps does not require increasing the number of ALUs because the ALUs are multiplexed by a higher number of warps. Increasing the number of warps involves the following changes to the hardware: 1) increasing the logic for the warp scheduler, 2) increasing the number of GPR tables, 3) increasing the number of IPDOM stacks, 4) increasing the number of register scoreboards, and 5) increasing the size of the warp table. It's important to note that the cost of increasing the number of warps is dependant on the number of threads in that warp; thus increasing warps for bigger thread configurations becomes more expensive. This is because the size of each GPR table, IPDOM stack, and warp table are dependant on the number of threads.

Figure 8 shows the increases in the area and power as we increase the number of threads and warps. The number is normalized to 1 warp and 1 thread support. All the data includes 1Kb 2 way instruction cache, 4 Kb 2 way 4 banks data cache, and an 8kb 4 banks shared memory module.

B. Benchmarks

All the benchmarks used for evaluations were taken from the Rodinia [5], a popular GPGPU benchmark suite.[1]

C. simX Simulator

Because the benchmarks used in Rodinia Benchmark Suite have large data-sets that took Modelsim a long time to simulate, we used simX, a C++ cycle-level in-house simulator for Vortex with a cycle accuracy within 6% of the actual

[1]The benchmarks that are not evaluated in this paper is due to the lack of support from LLVM RISC-V.

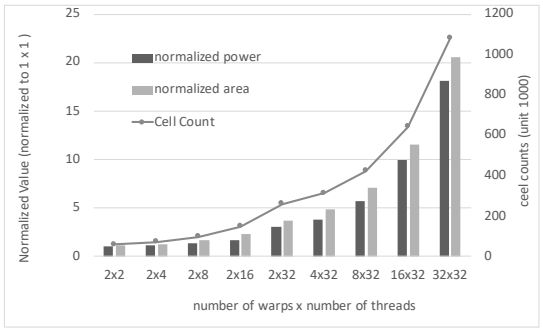


Fig. 8: Synthesized results for power, area and cell counts for different number of warps and threads

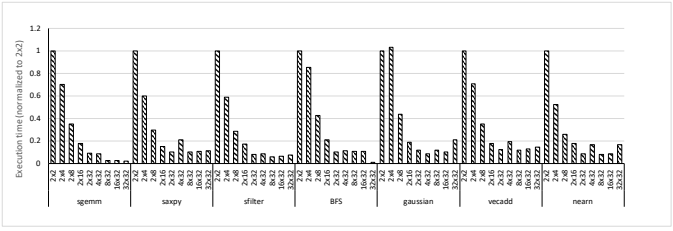


Fig. 9: Performance of a subset of Rodinia benchmark suite (# of warps x # of threads

Verilog model. Please note that power and area numbers are synthesized from the RTL.

D. Performance Evaluations

Figure 9 shows the normalized execution time of the benchmarks normalized to the 2warps x 2threads configuration. As we predict, most of the time, as we increase the number of threads (i.e., increasing the SIMD width.), the performance is improved, but not too much from increasing the number of warps. Some benchmarks get benefits from increasing the number of warps such as bfs, but in most of the cases increasing the number of warps is not translated into performance benefit. The main reason is that to reduce the simulation time, we warmed up caches and reduced the data set size, thereby the cache hit rate in the evaluated benchmarks was high. Increasing the number of warps is typically useful to hide long latency operations such as cache misses by increasing TLP and MLP; Thus, the benchmark that benefited the most from the high warp count is BFS which is an irregular benchmark.

As we increase the number of threads and warps, the power consumption increases but they do not necessarily produce more performance. Hence, the most power efficient design points vary depending on the benchmark. Figure 10 shows a power efficiency metric (similar to performance per watt) which is normalized to the 2 warps x 2 threads configuration. The results show that for many benchmarks, the most power efficient design is the one with fewer number of warps and 32 threads except for the BFS benchmark. As we discussed earlier since BFS benchmark gets the best performance from the 32 warps x 32 threads configuration, it also shows the most power efficient design point.

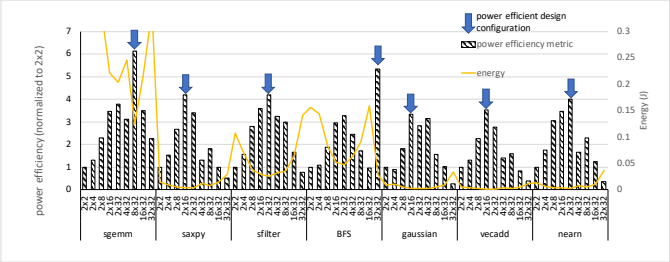


Fig. 10: Power efficiency (# of warps x # of threads (Power efficiency and Energy)

E. Placement and layout

We synthesized our RTL using a 15-nm educational library. Using Innovus, we also performed Place and route (PnR). Figure 7 shows the GDS layout and the power density map of our Vortex processor. From the power density map, we observe that the power is well distributed among the cell area. In addition, we observe the memory including the GPR, data cache, instruction icache and the shared memory have a higher power consumption.

VI. Related Work

ARA [3] is a RISC-V Vector Processor that implements a variable-length single-instruction multiple-data execution model where vector instructions are streamed into vector lanes and their execution is time-multiplexed over the shared execution units to increase energy efficiency. Ara design is based on the open-source RISC-V Vector ISA Extension proposal [1] taking advantage of its vector-length agnostic ISA and its relaxed architectural vector registers. Maximizing the utilization of the vector processors can be challenging, specifically when dealing with data dependent control flow. That is where SIMT architectures like Vortex present an advantage with their flexible scalar-threads that can diverge independently.

HWACHA [15] is a RISC-V scalar processor with a vector accelerator that implements a SIMT-like architecture where vector arithmetic instructions are expanded into micro-ops and scheduled on separate processing threads on the accelerator. An advantage that Hwacha has over pure SIMT processors like Vortex is its ability to overlap the execution of scalar instructions on the scalar processor which increases hardware complexity for hazard management.

Simty [6] processor implements a specialized RISC-V architecture that supports SIMT execution similar to Vortex, but with different control flow divergence handling. In their work, only microarchitecture was implemented as a proof of concept and there was no software stack, and none of GPGPU applications were executed with the architecture.

VII. Conclusions

In this paper we proposed Vortex that supports an extended version of RISC-V for GPGPU applications. We also modified OpenCL software stack (POCL) to run various OpenCL kernels and demonstrated that. We plan to release the Vortex RTL and POCL modifications to the public.[2] We believe that an Open Source version of RISC-V GPGPU will enrich the RISC-V echo systems and accelerate other researchers that study GPGPUs in wider topics since the entire software stack is also based on Open Source implementations.

References

[1] K. Asanovic, *RISC-V Vector Extension*. [Online]. Available: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

[2] K. Asanović *et al.*, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[3] M. A. Cavalcante *et al.*, "Ara: A 1 GHz+ scalable and energy-efficient RISC-V vector processor with multi-precision floating point support in 22 nm FD-SOI," *CoRR*, vol. abs/1906.00478, 2019.

[4] C. Celio *et al.*, "Boomv2: an open-source out-of-order risc-v core," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[5] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," ser. IISWC '09. IEEE Computer Society, 2009.

[6] S. Collange, "Simty: generalized simt execution on risc-v," in *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*, 2017, p. 6.

[7] J. J. Corinna Vinschen, "Newlib," http://sourceware. org/newlib, 2001.

[8] W. W. L. Fung *et al.*, "Dynamic warp formation and scheduling for efficient gpu control flow," ser. MICRO 40. IEEE Computer Society, 2007, pp. 407–420.

[9] M. Gautschi *et al.*, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[10] G. Gobieski *et al.*, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," ser. MICRO '52. ACM, 2019, pp. 670–684.

[11] J. Gray, "Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 17–20.

[12] Green500, "Green500 list - june 2019," 2019. [Online]. Available: https://www.top500.org/lists/2019/06/

[13] P. Jaaskelainen *et al.*, "Pocl: Portable computing language," *International Journal of Parallel Programming*, pp. 752–785, 2015.

[14] P. O. Jäskeläinen *et al.*, "Opencl-based design methodology for application-specific processors," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2010, pp. 223–230.

[15] Y. Lee *et al.*, "A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, Sep. 2014, pp. 199–202.

[16] Y. Lee *et al.*, "A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 2014, pp. 199–202.

[17] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, Aug 2009, pp. 1–314.

[18] V. Narasiman *et al.*, "Improving gpu performance via large warps and two-level warp scheduling," ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 308–317.

[19] NVIDIA, "Cuda binary utilities," NVIDIA Application Note, 2014.

[20] A. Waterman *et al.*, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," EECS Department, UC Berkeley, Tech. Rep., 2014.

[21] A. Waterman *et al.*, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[22] B. Zimmer *et al.*, "A risc-v vector processor with tightly-integrated switched-capacitor dc-dc converters in 28nm fdsoi," in *2015 Symposium on VLSI Circuits (VLSI Circuits)*. IEEE, 2015, pp. C316–C317.

[2] currently the github is private for blind review.