Vortex RISC-V GPGPU System: Extending the ISA, Synthesizing the Microarchitecture, and Modeling the Software Stack

Fares Elsabbagh
Georgia Institute of Technology
Atlanta, Georgia
fsabbagh@gatech.edu

Bahar Asgari
Georgia Institute of Technology
Atlanta, Georgia
bahar.asgari@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, Georgia
hyesoon@cc.gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, Georgia
sudha@gatech.edu

ABSTRACT

The open-source RISC-V instruction set architecture (ISA) has enabled computer architects to propose several innovative processors for a wide range of applications. One of the domains of processor design that can take advantage from RISC-V, but has not seen enough attention, is general-purpose GPU (GPGPU) design. To support the development of open source GPGPU system, we present Vortex, our solution for building single instruction, multiple thread (SIMT) execution using RISC-V. In addition to a synthesizable microarchitecture model, we propose a GPU ISA extension to RISC-V and a software model, in the form of a runtime kernel, which makes Vortex practical and easy to integrate. As a result, Vortex does not require any modifications to the current RISC-V compilers.

ACM Reference Format:
Fares Elsabbagh, Bahar Asgari, Hyesoon Kim, and Sudhakar Yalamanchili. 2019. Vortex RISC-V GPGPU System: Extending the ISA, Synthesizing the Microarchitecture, and Modeling the Software Stack. In *Proceedings of Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019).* , 6 pages.

1 INTRODUCTION

The advent of RISC-V [1, 11, 12], the open-source and free instruction set architecture (ISA), has delivered a new level of freedom in designing hardware architectures. In this new era, computer architects have designed several innovative processors and cores such as BOOM v1 and BOOM v2 [2] out-of-order cores, as well as system-on-chip (SoC) platforms for a wide range of applications. For instance, Gautschi et al. [5] have extended RISC-V to digital signal processing (DSP) for scalable Internet-of-things (IoT) devices. Moreover, vector processors [13] and processors integrated with vector accelerators [9] have been designed and fabricated based on RISC-V. In spite of the advantages of the preceding studies, not enough attention has been devoted to building an open-source general-purpose GPU (GPGPU) system based on RISC-V. Although a couple of recent work have proposed an FPGA accelerator for massively parallel computations (GRVI Phalanx) [6], and a generalized single instruction, multiple thread (SIMT) execution on RISC-V (Simty) [3], none of them have implemented the full-stack by extending the RISC-V ISA, synthesizing the microarchitecture, and modeling the software. We believe that such an implementation is in fact necessary to achieve the level of usability and customizability in massively parallel platforms.

This paper proposes Vortex, a RISC-V GPGPU system, which comprises three main components:

- **ISA extension:** We extend the RISC-V ISA to control warps and threads, and to handle control divergence. We designed the ISA to reduce the microarchitecture and software design complexity without sacrificing functionality or flexibility. (see Section 3).
- **Microarchitecture implementation:** We implement a synthesizable GPGPU model that is highly customizeable and lightweight (see Section 4).
- **Software model:** We implement the software stack, including a runtime kernel, that supports a standalone GPGPU (i.e., host and the kernel are all executed from the same main thread) and a matrix library, to abstract the hardware and provide useability, extensibility, and flexibility (see Section 5).

In this paper, besides synthesizing the model and verifying the functionality, we examine the performance of multi-thread applications and the impact of the number of hardware warps and hardware threads on the overall performance. We run a matrix-matrix multiplication benchmark on various configurations to test performance and synthesize the model on an Intel® Arria® 10 FPGA to examine the cost.

2 TERMINOLOGY

In a SIMT execution model, a group of threads that share a program counter (PC) are grouped in a hardware warp. The threads inside a warp are implemented as single instruction, multiple data (SIMD) lanes. A software warp is the lowest granularity of tasks that can be scheduled by the software. In the software level, a runtime kernel runs closest to the hardware and schedules software warps and contexts (contexts is analogous to a CPU process) onto hardware warps to run GPU kernels. A software warp is allocated and scheduled by the runtime kernel and executes like a one-dimensional CUDA block.

3 ISA EXTENSION

This section describes our proposed ISA extension that enables building SIMT using RISC-V. To accomplish the desired behaviour, we introduce five new instructions, listed in Table 1, and a thread mask register, to the RISC-V ISA, which were inspired by HAPR

CARRV 2019, June 2019, Phoenix, AZ, USA

ISA [8]. The proposed five new instructions are R-Type instructions and fit in one opcode. These instructions are intended to be the minimal set of instructions to introduce SIMT functionality to an architecture. They are designed to address the three new concepts that SIMT introduces: hardware warps, hardware threads, and control divergence.

3.1 Warp Control

The *wspawn* instruction is intended to address the introduction of hardware warps to the architecture; The instruction sends a signal to the architecture to spawn or wake up a hardware warp at a specific PC value with only thread 0 activated. *wspawn* also copies the value of the %src register from the executing warp to the %dest register of the new warp that's being spawned. If there are no available hardware warps to be spawned, an exception will be thrown.

In addition, to halt the execution of an active hardware warp we utilize the *EBREAK* instruction [11] which sends a signal to the architecture to stop the execution of the current warp. In our implementation, the original intended use of *EBREAK*, causing control to be transferred back to a debugging environment, applies when there's only one active warp remaining.

```
wspawn %dst, %PC, %src
```

3.2 Thread Control

We propose *clone* and *tmc* instructions to address the introduction of SIMD lanes. *tmc* instruction activates and deactivates lanes by controlling the thread mask register for the executing warp. The thread mask register could be read as a Control and Status register (CSR) and contains a bit for each thread singnifying whether the thread is currently active or not. For example, if %NumThreads is equal to five, then lanes zero to four will be activated and the five least significant bits of the thread mask will be set to one. A more complex control flow can be handled by the software as explained in Section 5.3. *clone* instruction copies all of the general purpose registers of thread zero to the general purpose registers of thread %ThreadNum. If either %threadNum or %NumThreads is greater than the number of available hardware threads, an exception will be thrown.

```
clone %ThreadNum
tmc %NumThreads
```

3.3 Control Divergence

split and *join* instructions are used to handle control divergence. Control divergence, which occurs when the threads in the same hardware warp want to follow distinct execution paths, is handled by a hardware immediate postdominator (IPDOM) stack [8]. The %predicate is set by any of the 32 general-purpose registers and it is considered true by the *split* instruction if there's a non-zero value stored in the register. *split* instruction pushes information about the current state of the thread mask and the predication result for all hardware threads into the IPDOM stack and *join* pops out of the IPDOM stack (more details in Section 4.2). If a *join* is executed without a corresponding *split*, an exception will be thrown.

```
split %predicate
join
```

Table 1: Our proposed SIMT ISA extension.

Instructions	Description
wspawn	Spawn a new warp.
clone	Clone register state to a specific thread
tmc	Change the thread mask to activate threads
split	Control flow divergence
join	Control flow reconvergence

4 MICROARCHITECTURE IMPLEMENTATION

The aim of Vortex GPGPU is to make a lightweight GPGPU that is highly customizable to target specific applications. Vortex GPGPU is a 32-bit 5-stage pipeline that supports RISC-V's integer and multiplication extensions (RV32IM) on top of the instructions mentioned in Section 3. Compared to a basic RISC-V pipeline, Vortex has 1) a hardware warp scheduler that contains the PC and thread mask registers and an IPDOM stack for each hardware warp 2) a warp context for each hardware warp that contains a set of register files for each thread in a warp, and 3) a shared memory module, as shown in shown in Figure 1. In addition, the execute stage has multiple execution units to parallelize execution of hardware threads in a warp; However, the number of execution units does not have to be equal to the number of hardware threads or lanes available. Thus, the latency of the execute stage will depend on the number of execution units in the configuration. Vortex supports a fully customizable warp and threads per warp configuration, L1 cache, and a banked shared memory module. An instruction being executed by a warp is only fetched and decoded once, but executes on the operands and destination registers for each hardware thread in a warp.

4.1 Warps

The warp scheduler in Vortex is implemented as a fine-graine scheduler which is a lightweight scheduler that and hides the latency for branches and jumps. It stores a state that keeps track of which hardware warps are active and it supports the *wspawn* and *ebreak* instructions described in Section 3. In addition, the execution is optimized so that only the currently active warps are scheduled. The rest of the warp state is stored in the warp context in the decode stage. The warp context contains the general purpose register files for each thread and is responsible for supporting the *clone* and *wspawn* instructions.

4.2 Control Divergence

One of the biggest challenges in SIMT processors is control divergence which leaves the pipeline highly underutilized because of its dynamic nature [4]. However, to maintain a simplistic approach, a hardware IPDOM stack [8] is used to handle control divergence. The IPDOM stack works by maintaining a private thread mask register for each warp that stores a mask of the current running threads. When a *split* instruction is reached, the current state of the thread mask and the inverse of the new thread mask, along with PC+4, are pushed onto the IPDOM stack. The next instruction continues with the new thread mask. When *join* instruction is reached,

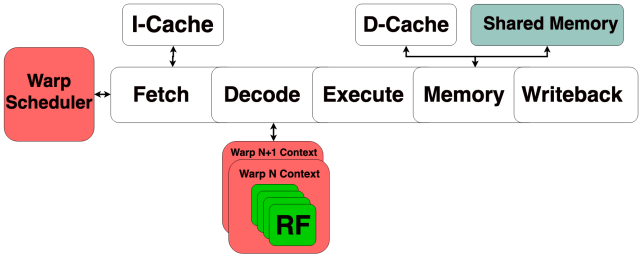

Figure 1: Microarchitecture design of Vortex.

the stack is popped and the pipeline either jumps to the PC popped from the stack, or falls through to the current PC+4.

4.3 Memory

Similar to Volta architecture, Vortex supports both an L1 data cache and shared memory module [7]. The L1 cache implementation is parameterizable by cache size, block size, and number of ways. To maintain a lightweight design, only one memory access request could be sent to the cache in one cycle; This indicates that a 32 threads per warp would take 32 cycles in the memory stage to complete a memory access to the cache in case of a cache hit.

On the other end of the spectrum, the shared memory module is extremely parallelizable. Each hardware thread has a one-cycle access to a different bank. Vortex matrix library is optimized to reduce bank conflicts. The shared memory module is mapped to addresses 0xFF000000 to (0xFF000000 + shared memory size). This address space is interleaved for each bank by word; The word at 0xFF000000 is stored in bank 0, 0xFF000004 is stored at bank 1, ... for a number of banks equal to the number of hardware threads in the configuration.

5 SOFTWARE STACK

Vortex has both a runtime kernel and a matrix library, as shown in Figure 2, which are designed to capitalize on the microarchitecture implementation for the best performance. The Vortex runtime kernel 1) supports a standalone GPGPU (i.e., host and the kernel are all executed from the same main thread) 2) introduces concepts such as software warps which could be scheduled and preempted, similar to threads in CPU programming models, 3) provides basic communication channels between software warps, 4) manages the shared memory address space, and 5) provides synchronization between software warps in the form of centralized barriers [10]. Another major benefit of this software stack is that it does not require any changes to the RISC-V compilers as explained in Section 5.1.

The library is responsible for 1) copying data from hardware cache to shared memory, 2) implementing GPU kernels for matrix operations, and 3) optimizing requests to the kernel to call these GPU kernels. A program running on Vortex could fully rely on the library to enable parallelism in the application. In addition, there is an interface that allows a program to implement its own GPU kernel described in Section 5.2.

5.1 Vortex Runtime Kernel Implementation

To enable Vortex runtime kernel to utilize the new instructions without modifying the existing compilers, we implemented the runtime kernel as an intrinsic library. Thus, the critical sections of the kernel are written in RISC-V assembly. In these critical sections, the kernel spawns new hardware warps, saves and stores software warp contexts, initializes thread registers, and activates/deactivates threads. Since these critical sections have defined entry and exit points where the proposed SIMT instructions (Table 1) are used, these instructions could be encoded manually and inserted in the the assembly code as hex values. This approach achieves the required functionality without restricting the platform or requiring any modifications to the RISC-V compilers. Similarly, control divergence and reconvergence are handled by macros that use C inline assembly to implement the desired functionality as shown in Figure 3.

5.2 Writing New GPU Kernels

One of the main goals of this project is to provide a platform that is customizable enough to be able to support specific applications, as mentioned in Section 4. An interface was designed to easily write and develop GPU kernels without having to worry about the architecture design or kernel implementation.

To write a GPU kernel, the header must be in the format described in Figure 4. It's important to note that unlike CUDA's implementation, wid (Warp ID) and tid (Thread ID) are stored in the general-purpose registers and are handled by the software. Any possible control divergent statements requires the use of __if and __else macros described in Figure 3. In addition, to issue a request to the runtime kernel to call this GPU kernel, vx_spawnWarps should be used.

Note that requested NumW (i.e., the number of software warps) and NumT (i.e., number of software threads) in Figure 4 could be more than the available hardware resources, since vx_spawnWarps is referring to software not hardware warps, as explained in Section 5.3.

5.3 Execution Model

The programming model of Vortex is very similar to that of CUDA's. Consider the hardware to be configured with 4 warps and 32 threads per warp Vortex configuration and the software to contain a GPU kernel called *vx_mat_add* as shown in Figure 4. The example below describes the control flow of a program requesting 1024 software warps each with 32 threads active to call *vx_mat_add*. The maximum number of software warps and software threads a user could request is 2^{32}. Figure 5 shows the pseudo-code for the steps described below.

(1) Hardware Warp 0: The main program will do a system call to the kernel to execute *vx_mat_add* as shown in Figure 4, with NumW set to 1024 warps and NumT set to 32 threads.
(2) Hardware Warp 0: The runtime kernel will store the context for the current warp. It will then round-robin the 1024 kernel calls requested by the main program to the runqueue of warps 1,2,3 and itself.

(3) Hardware Warp 0: The runtime kernel will use *wspawn* instruction described in section 3.1 to wake up hardware warps 1, 2, and 3.
(4) Hardware Warps 0,1,2,3: the runtime kernel will check if there are any available tasks in the runqueue. If there is, it will go to step 5 otherwise it will go to step 6.
(5) Hardware Warps 0,1,2,3: The runtime kernel will:
  (a) use the memory manager to allocate the stacks for 32 threads requested by the main program
  (b) utilize the *clone* instruction to set the stack pointer, return address, argument zero tid (i.e. thread id), and argument one wid (i.e. warp id) registers to the correct values for each hardware thread. It does this by 1) Storing the register state of thread zero 2) Setting thread zero's registers to thread one's desired register values 3) cloning thread zero's register state to thread one's registers 4) Repeating Steps 2 and 3 for the desired number of threads and 5) Restoring the register state for thread zero.
  (c) Utilize the *tmc* instruction to activate the first 32 hardware threads as requested by the main program. It will then call *vx_mat_add* which will return to step 4 when it returns.
(6) Warps 1,2,3: Free their stack memory by calling the memory manager then halt execution. Warp 0: Wait until warps 1, 2, and 3 are halted, load the context stored in step 2, then return to the main program.

In addition to the example given above, there are other variations supported by the kernel by setting different flags.

Complex Control Flow: In the example above, wid will have a range of {0,...,NumW-1} and tid will have a range of {0,...,NumT-1}. However, as described in Section 5.2, wid and tid are controlled by the software which allows for a more flexible control flow. An input flag could be set as an input to vx_spawnWarps to only allow odd or even IDs. For more control, a number generator in the form of a lambda expression could also be passed as an input and would be used to generate warp and thread IDs.

Asynchronous Requests: In the example above, the main program made a synchronous call to the runtime kernel, blocking hardware warp 0 until the request has been completed. However, in step 1 warp 0 could have called the runtime kernel asynchronously by setting an input flag to vx_spawnWarps. Thus, the tasks in step 2 would have been divided across hardware warps 1, 2, and 3 and hardware warp 0 would have returned back to the main program.

Concurrent Execution of GPU Kernels: In the example given above, the main program requested to spawn warps to *vx_mat_add*. However, if the main program made that request asynchronously, it could request to spawn warps to another GPU kernel before the exeuction of *vx_mat_add* ends; Thus, both of these GPU kernels would be concurrently scheduled by the runtime kernel.

6 RESULTS

6.1 Evaluation Methodology

1, 2, 4, 8, 16, 32 warps/threads per warp configurations are evaluated and the have been successfully [1] synthesized on an Intel® Arria® 10 FPGA on 10AX115U3F45I2SG device for RV32I using Quartus

[1] 16, 32 warps with 32 threads per warp configurations failed due to routing congestion and limited ALMs respectively.

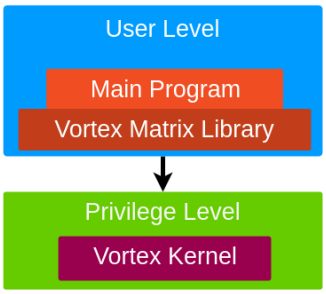

Figure 2: Software stack of Vortex.

```
#define __if(cond) \\
    register bool b asm("t0") = cond; \\
    asm("split t0"); \\
    if (b) {

#define __else \\
    } else {

#define __endif \\
    } \\
    asm("join");
```

Figure 3: Control divergent __if __else macros pseudo-implementation.

```
void vx_mat_add(unsigned tid, unsigned wid)
{
    arg_t * args = (arg_t *) vx_get_arg_t();

    unsigned * x_ptr = args->x;
    unsigned * y_ptr = args->y;
    unsigned * z_ptr = args->z;

    unsigned num_cols = args->num_cols;

    int idx = (wid * num_cols) + tid;
    __if(tid < num_cols)
    {
        z_ptr[idx] = x_ptr[idx] + y_ptr[idx];
    }
    __else
    __end_if

    return;
}
NumW = 1024 software warps
NumT = 32 software threads per software warp
vx_spawnWarps(NumW, NumT, vx_mat_add, (&args));
```

Figure 4: A CUDA like GPU kernel interface.

```
#define AVA_WARPS 4
queue<Task> runqueues[AVA_WARPS];

void vx_spawnWarps(numW, numT, func, args)
{
    // Only warp 0 thread 0 is active.
    if (vx_sigsetjmp()) // Storing context
         return;

    // Round robin tasks to Warps 0, 1, 2, and 3.
   vx_distribute_tasks(runqueues, numW, numT, func, args);

    for (cur_warp = 1; cur_warp < AVA_WARPS; cur_warp++)
    {
       // Warp 0 communicating the hardware wid using t0
       register int hardware_warp asm("t0") = cur_warp;
        // Warp 0 requesting Warps to spawn at SCHEDULE
         wspawn("t0", &&SCHEDULE, "t0");
    }
    // Warp 0 setting its own Hardware wid to 0.
    {register int hardware_warp asm("t0") = 0;}

    // Now thread 0 of warps 0,1,2, and 3 are active
    SCHEDULE:
    register int hardware_warp asm("t0");
    Queue<Task> & myQueue = runqueues[hardware_warp];

    // This loop will iterate for numW/AVA_WARPS
    //   iterations for each hardware warp
    while(!myQueue.is_empty())
    {
        Task t = myQueue.dequeue();
        vx_scheduleTask(t);
    }

    vx_barrier(0); // Synchronize all hardware warps

    if (hardware_warp != 0) EBREAK; // Halt execution

    // Now only warp 0 thread 0 is active.
    vx_siglongjmp(); // Returns back to main program
}
```

Figure 5: vx_spawnWarps pseudo-Implementation. This code assumes a Vortex configuration of four hardware warps.

18.0. In these tests, the number of execution units are always kept equal to the number threads in a warp for each configuration. The number of cycles are simulated with a two-way set associative L1 cache with 16 Kbytes, block size of 256 bytes, and a 16-Kbytes shared memory. The number of banks is always kept the same as the number of threads in a warp. To examine the performance of Vortex for various configurations, we use a simple matrix multiplcation benchmark: 1) initialize two 32×32 matrices, 2) multiply both matrices, and 3) print the result.

Table 2: Some of the important functions implemented in the Vortex software model.

Function Name	Description
vx_spawnWarps	Calls the kernel to spawn warps
vx_sm_malloc	Allocates a shared memory segment
vx_sm_free	Frees a shared memory segment
vx_mat_cpy	Copy matrix to shared memory
vx_mat_mult	Execute matrix multiplication
vx_mat_add	Execute matrix addition

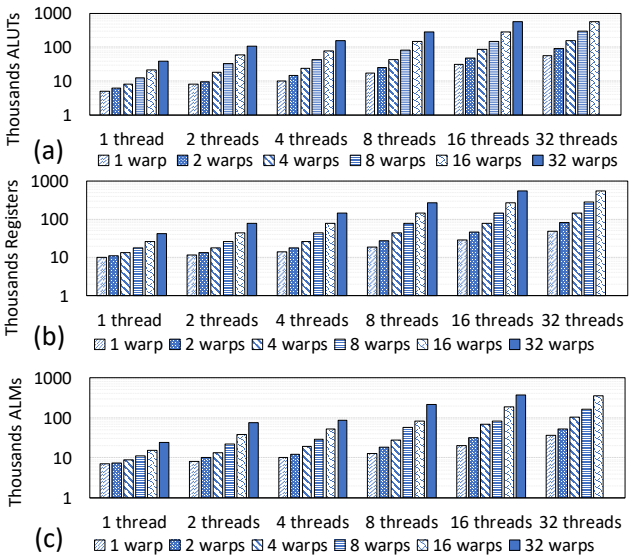

Figure 6: The cost of various Vortex configurations on an Intel® Arria® 10 FPGA in terms of the number of (a) ALUTs, (b) registers, and (c) ALMs, as a function of number of warps and number of threads per warp.

6.2 Resource Utilization

Both Figures 6 and 7 show that the hardware resources required for adding threads is more than that of adding warps. For example, consider the following two configurations: (a) 4-warp with 32 threads per warp and 32 execution units and (b) 32-warps with 4 threads per warp and 4 execution units. (a) requires 19% more ALMs, 2% more ALUTs, and 3% more registers than (b). While both of these configurations require the same number of general purpose registers, *4 warps * 32 threads/warp * 32 registers* or *32 warps * 4 threads/warp * 32 registers*, (a) is configured to have more execution units than (b) which explains the difference in resource utilization.

6.3 Performance

The hardware cost of adding a thread versus adding a warp has a considerable effect on performance. Configuration (a) described in Section 6.2 has a 25% lower fmax than configuration (b), thus favoring a high warp count configuration. This is also caused by the higher number of execution units and register fanout for configuration (a).

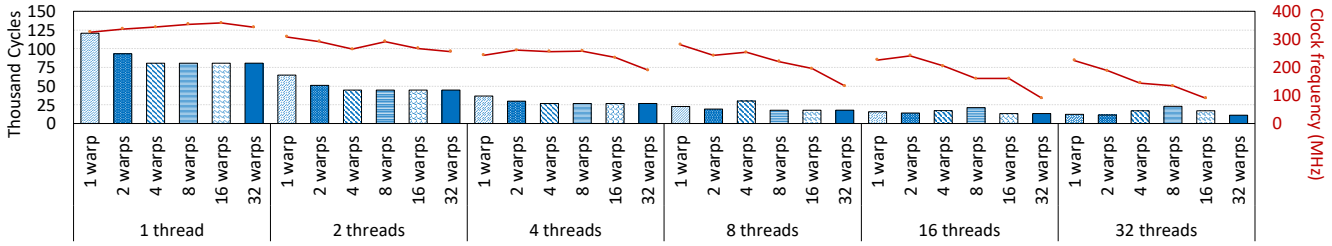

Figure 7: The number of cycles and the fmax for each Vortex configuration on an Intel® Arria® 10 FPGA while multiplying two 32×32 matrices then printing the result.

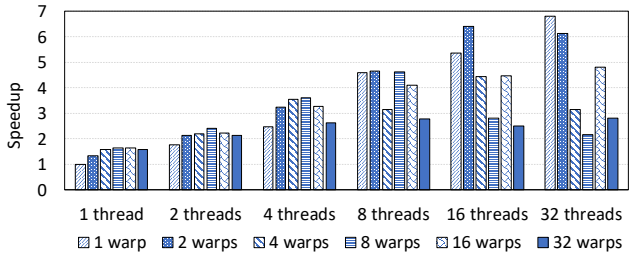

Figure 8: The speedup of every configuration when performing matrix multiplication relative to the one warp one thread configuration.

However, when we look at the number of cycles illustrated in Figure 7, we observe that the combination of a high latency cache and a fine-grained warp scheduler might be a big bottleneck. Figure 7 shows that (a) executed in 40% less cycles than (b), favoring a high thread count configuration and illustrating that adding more warps is causing a lower pipeline utilization. A better warp scheduler and a higher cache port number would result in much better performance. For example, a configuration of 32 warps and 4 threads, each cache access would stall that warp for 4 cycles; A good warp scheduler would not stall the pipeline and schedule the other 31 warps, resulting in a higher pipeline utilization.

Figure 8 shows the overall speedup of all configurations relative to 1 warp and 1 thread per warp configuration. It shows that the best speedup is a 1 warp and 32 threads per warp configuration, favoring an expensive but highly parallel configuration over an underutilized one.

7 CONCLUSION AND FUTURE WORK

This paper introduced Vortex general-purpose GPU (GPGPU) system, which includes an ISA extension, a synthesizable microarchitecture implementation, and a software model. The ISA extension is compromised of only 5 new instructions and the software model includes a runtime kernel and a library that do not require any compiler modifications. The microarchitecture implementation has already been synthesized on Arria 10 FPGA with an fmax ranging from 340 MHz to 90 MHz based on the configuration. In the future, we plan to implement a better hardware warp scheduler and a cache module that allow for a better pipeline utilization. We will also be looking at configurations with higher warp and thread count and comparing the performance to other SIMT and SIMD processors. In addition, we plan on extending Vortex runtime kernel and library with more GPU kernels, better resource utilization, and provide support for some popular libraries like OpenCV.[2]

8 ACKNOWLEDGMENT

We would like to thank Blaise Tine and Ahmed Elsabbagh for their constant feedback, guidance, and expertise on the project.

REFERENCES

[1] Asanović, K., and Patterson, D. A. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[2] Celio, C., Chiu, P.-F., Nikolic, B., Patterson, D. A., and Asanovi, K. Boomv2: an open-source out-of-order risc-v core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)* (2017).

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

[4] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 407–420.

[5] Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F. K., and Benini, L. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25*, 10 (2017), 2700–2713.

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

[7] Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. Dissecting the nvidia volta gpu architecture via microbenchmarking. *CoRR abs/1804.06826* (2018).

[8] Kersey, C. D., Kim, H., and Yalamanchili, S. Lightweight simt core designs for intelligent 3d stacked dram. In *Proceedings of the International Symposium on Memory Systems* (New York, NY, USA, 2017), MEMSYS '17, ACM, pp. 49–59.

[9] Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanović, V., and Asanović, K. 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)* (2014), IEEE, pp. 199–202.

[10] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 21–65.

[11] Waterman, A., Lee, Y., Patterson, D. A., and Asanovi, K. The risc-v instruction set manual. volume i: User-level isa, version 2.0. Tech. rep., CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[12] Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62 116* (2011).

[13] Zimmer, B., Lee, Y., Puggelli, A., Kwak, J., Jevtic, R., Keller, B., Bailey, S., Blagojevic, M., Chiu, P.-F., Le, H.-P., 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)* (2015), IEEE, pp. C316–C317.

[2]The following is the link to the GitHub page for the project: https://github.gatech.edu/casl/Vortex