Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics

Blaise Tine
btine3@gatech.edu
Georgia Institute of Technology

Fares Elsabbagh
fsabbagh@gatech.edu
Georgia Institute of Technology

Krishna Yalamarthy
kyalamarthy@gatech.edu
Georgia Institute of Technology

Hyesoon Kim
hyesoon.kim@gatech.edu
Georgia Institute of Technology

ABSTRACT

The importance of open-source hardware and software has been increasing. However, despite GPUs being one of the more popular accelerators across various applications, there is very little open-source GPU infrastructure in the public domain. We argue that one of the reasons for the lack of open-source infrastructure for GPUs is rooted in the complexity of their ISA and software stacks. In this work, we first propose an ISA extension to RISC-V that supports GPGPUs and graphics. The main goal of the ISA extension proposal is to minimize the ISA changes so that the corresponding changes to the open-source ecosystem are also minimal, which makes for a sustainable development ecosystem. To demonstrate the feasibility of the minimally extended RISC-V ISA, we implemented the complete software and hardware stacks of Vortex on FPGA. Vortex is a PCIe-based soft GPU that supports OpenCL and OpenGL. Vortex can be used in a variety of applications, including machine learning, graph analytics, and graphics rendering. Vortex can scale up to 32 cores on an Altera Stratix 10 FPGA, delivering a peak performance of 25.6 GFlops at 200 Mhz.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**.

KEYWORDS

reconfigurable computing, computer graphics, memory systems.

ACM Reference Format:
Blaise Tine, Fares Elsabbagh, Krishna Yalamarthy, and Hyesoon Kim. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480128

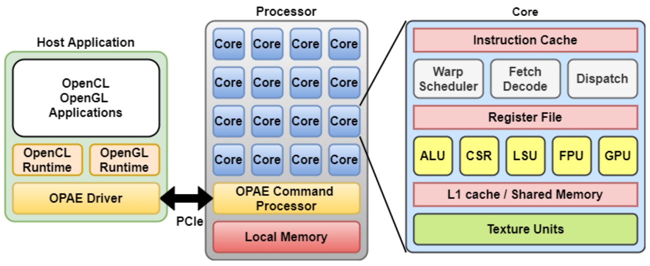

Figure 1: Vortex framework overview.

1 INTRODUCTION

The emergence of data-parallel architectures and general-purpose graphics processing units (GPGPUs) has enabled new opportunities to address the power limitations and scalability of multi-core processors [25], allowing for 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 a relatively low energy cost, leading the current race for energy efficiency (Green500 [30]) and application support with their accelerator-centric parallel programming models [57] [50].

Architecture research on GPGPUs has mainly focused on simulations [10] [54] [31] [63] [41] [23] that model the hardware architecture at the Intermediate Language (IL) level (PTX [52], HSAIL [56]) because of the lack of open-source hardware implementation. Simulating complex hardware at the IL level can obfuscate several aspects of the micro-architecture that have a substantial impact on performance [32]. The recent introduction of full-system ISA-based GPU model simulations [55] has closed the evaluation gap with actual hardware but still remains limited as it does not cover other important areas such as run-time evaluation, power efficiency, reliability, and detailed microarchitecture evaluation that can be pursued when using RTL-level implementation. Several implementations of open-source GPGPU hardware [1] [21] [16] [4] [17] [11] have been proposed that provide a detailed micro-architectural description of various GPGPU's components. However, these implementations lack a detailed description of the cache subsystem, which is one of the most performance-critical components in the GPGPU. Also, the ISA used in those implementations is custom or proprietary, restricting application support and wide adoption.

Two recent technological trends provide an opportunity to revisit and expand open-source GPGPUs for hardware research today:(1) The emergence of high-end FPGAs in the consumer market. Today's high-capacity FPGAs with floating-point DSPs and large memory provide high computational capability at a lower energy budget that makes implementing a full-feature GPGPU with a detailed cache subsystem operating at a reasonable speed a possibility. (2) The advent of RISC-V [7] with its free, open, and extensible ISA, provides a new level of freedom in designing hardware architectures at a lower cost that leverages its rich ecosystem of open-source software and compiler tools. Adopting the RISC-V ISA for a GPGPU processor architecture presents a solid base for wide-range adoption.

Today, graphics acceleration remains an important research area, as the demand for high-speed higher-quality real-time rendering [39, 46, 64, 65] continues to grow. The current area of GPU computation for gaming moving to the cloud with Google Stadia [29] and Microsoft xCloud [49] presents new challenges for graphics computation, including real-time latency, and scalability, as well as security. However, to the best of our knowledge, no open-source graphics pipeline infrastructure exists that integrates the entire software and hardware stacks.

In this paper, we introduce Vortex[1], an open-source RISC-V-based soft GPU for high-end FPGAs (Figure 1). In this work, we aim to explore the design and implementation of a GPGPU on modern FPGAs. The challenges of this task are: First, identifying the subset of the GPGPU ISA that covers the essential capabilities of the SIMT execution model across modern GPGPUs and still fit on FPGA. Second, identifying an effective way to implement the GPGPU microarchitecture on top of the RISC-V ISA while maintaining compatibility with the standard. Third, exploring the microarchitecture suited for FPGAs that maximizes resource utilization.

We particularly focused on minimizing our ISA extension for two reasons: (1) to utilize as much of the existing open-source hardware and software ecosystem, and (2) to provide a sustainable development ecosystem. We argue that one of the most beneficial findings from this work is that by adding only six new instructions to the standard RISC-V ISA, The Vortex processor can execute GPGPU applications and also accelerate the 3D graphics pipeline.

In addition to the standard SIMT microarchitecture components, Vortex implements a detailed high-bandwidth non-blocking cache subsystem using a multi-ported multi-bank architecture optimized for FPGAs. It also integrates a PCIe-based command processor for communicating with a host processor like conventional GPGPUs. The platform also implements a robust compiler, driver, and application stack supporting OpenCL. We also extended the microarchitecture by implementing texture sampling units [66], allowing the platform to support graphics rendering. Vortex was designed from the ground up using elastic pipelines [22] [53], providing consistency across the design and enabling design patterns that make the code more accessible and extensible for research.

This paper makes the following key contributions:

- We showcase a taxonomy of current GPGPU ISAs and propose a minimal subset that covers the essential SIMT microarchitectural capabilities.
- We describe Vortex's SIMT microarchitecture, its texture unit implementation, and its rasterization pipeline.
- We detail the implementation of Vortex high-bandwidth non-blocking cache using a multi-ported multi-bank architecture optimized for FPGAs.
- We demonstrate the effectiveness of an elastic pipeline on large multi-threaded architectures and how it is leveraged to scale the processor up to 32 cores while preserving a good operating clock frequency.

[1]The Vortex's project is available at http://vortex.cc.gatech.edu.

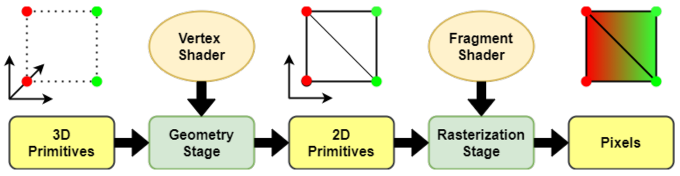

Figure 2: Overview of the graphics pipeline.

- We present an evaluation of a PCIe-based soft GPU framework on a modern FPGA.

2 BACKGROUND ON GRAPHICS

Figure 2 illustrates the main stages of the programmable 3D graphics pipeline:

Geometry Stage: In this stage, incoming vertices from the application are transformed to screen-space triangle primitives using a programmable vertex shader.

Rasterization Stage: Triangles entering this stage are traversed pixel-by-pixel, invoking a fragment shader that generates the color that is rendered to the destination buffer.

Texturing: A fragment shader stage where the pixel color is combined with texture data (texels). The inputs to the texturing stage are the normalized texel coordinates and the filtering mode. Multiple filtering techniques are used - point, linear, bilinear, and trilinear. Point sampling returns the nearest texel at the input location, whereas bilinear returns an interpolated value of the four nearest texels. Trilinear filtering combines the bilinear filtering of adjacent texture surfaces of different resolutions (mipmaps) (see Figure 3).

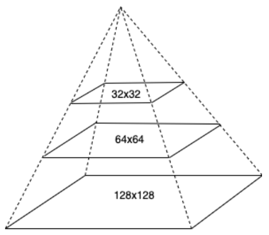

Figure 3: Texture mipmaps.

Modern GPUs support two types of rendering architectures: 1) immediate-mode rendering, where triangle primitives are issued for rasterization in the order they are produced, and 2) tile-based rendering [60], where the geometry outputs are subdivided and rasterized on a per-tile basis to reduce memory footprint. Rasterization is the most compute and memory intensive stage in the GPU, mainly dominated by texture sampling, which are memory-bound [34]. Modern GPUs execute shaders on a multi-threaded processor, and rasterization is done using fixed-function hardware. It is also possible to implement the graphics stack entirely in software using the GPU compute pipeline [43] while maintaining reasonable performance (1.5-8x slowdown). Larrabee [58] first experimented with this solution by only accelerating texture sampling and moving the rest of the pipeline to software. Their texture sampling unit supported all filtering modes, including mipmapping [26]. We opted for a software rendering approach following Larrabee where only texture sampling is accelerated due to limited area on the FPGA. Software rendering is also useful for Vortex in that it enables the flexibility of exploring various rendering algorithms on the platform. Vortex differs from Larrabee in that only rasterization is

ISA	Memory Model	Threading Model	Register File	Thread Control	Synchro-nization	Flow Control	ALU Operations	Memory Operations	GPU Operations
RDNA [2]	GDS, LDS Constants Global	Workgroup Wavefront 32/64 threads	Vector/Scalar 256 VGPRs 106 SGPRs	end threads thread mask	barrier wait_cnt data dep	branch theead mask	arithmetic conditional bitwise	load store prefetch	interpolate tex-sampler
GCN [3]	GDS, LDS Constants Global	Compute unit Wavefront 64 threads	Vector/Scalar 256 VGPRs 102 SGPRs	end threads thread mask	barrier wait_cnt data dep	branch theead mask split/join	arithmetic conditional bitwise	load store prefetch	interpolate tex-sampler
PTX [52]	Shared, Texture Constants Global	Grid/CTA Warp 32 threads	Scalar	predicate	barrier membar	branch predicate	arithmetic conditional bitwise	load store prefetch	tex-sampler tex-load tex-query
GEM [35]	SW Managed	Root thread Child tread	256-bit Vec 128 GRFs predicate	send msg	Wait Fence	branch SPF Regs split/join	arithmetic conditional bitwise	load store	interpolate tex-sampler
PowerVR [61]	Global Common St Unified St	USC 32 threads	Vector 128-bit	predicate	fence	branch predicate	arithmetic conditional bitwise	load store	tex-sampler iteration alpha/depth
Vortex	Shared Global	Compute Unit Wavefront	Scalar 32-bit	thread mask	Barrier Flush	Split/Join	arithmetic conditional bitwise	load store	tex-sampler

Table 1: Comparing mainstream GPU ISAs with Vortex.

offloaded to the FPGA, allowing the geometry processing to execute concurrently on the host processor for load balancing.

3 GPU ISA

3.1 Taxonomy of GPGPU ISA

Table 1 shows a comparative evaluation of the different ISAs: Nvidia PTX [52][2], AMD RDNA [2], AMD CGN [3], Intel GEM [35], and PowerVR mobile GPU [61]. We excluded debugging, exception handling, and other systems management instructions.

The Threading Model: AMD GCN implements 64-thread wavefronts that are grouped into compute units (CU). RDNA extended GCN's compute units with a WorkGroup that comprises two CUs. It also introduces a new mode for 32-thread wavefronts. PTX uses Warp structures to represent wavefronts, each having 32 threads, and cooperative thread array (CTA) structures representing a group of warps. CTAs are grouped into grids. GEN architecture is CPU-centric with root threads that are dispatched and managed by hardware, and child threads that are spawned dynamically from their parent root thread during shader execution. PowerVR defines a Unified Shading Cluster (USC) structure that groups multiple threads.

The Memory Model: In addition to global and constant memories, AMD GPUs implement a dedicated local memory (LDS) that is shared by all threads within a workgroup and a global shared memory (GDS) across all workgroups. PTX has one shared memory structure available at the CTA level and an additional dedicated memory space for textures. GEM ISA only defines a global memory space as on traditional CPU architectures, leaving its management and organization up to software. On PowerVR, shared memory is modeled by two register banks: a unified store local to ALUs and a common store local to a USC.

Register Files: All ISAs support SIMD vector registers, with AMD having a separate scalar register file. On RDNA, 256 32-bit vector registers and 106 32-bit scalar registers are accessible to shader programs. GEM has larger 128 256-bit vector registers per thread and supports predication with predicate registers. PowerVR has 128-bit SIMD vector registers and predication is also supported.

Thread Control: GEM ISA uses message-passing instructions to handle thread communication with other hardware components inside the processor. It is used to control thread spawn and termination. AMD uses a thread mask to control threads' activation and provides a dedicated ENDPGM instruction for terminating a wavefront. PTX uses predication to control thread activation.

Synchronization: Barrier and memory fence are supported on all architectures. AMD ISA defines an explicit WAIT_CNT instruction for flushing previously issued instructions and data dependency counter instructions (VM_CNT, VS_CNT). GEM uses message passing for thread synchronization and memory fence. PTX provides explicit *barrier* and *membar* instructions for thread synchronization and memory fence, respectively.

Flow Control: Standard branch instructions are provided on all ISAs. For the special cases of control-flow divergence, predication or thread masks can be used by applications to control thread activation. GCN and GEM provide explicit split/join instructions for compilers to annotate the code blocks at divergent and convergent points, respectively.

ALU Operations: Standard integer and floating-point arithmetic operations are supported on all ISAs. Double, single, and half-precision floating-point formats are also supported, with the

[2] We should note that using PTX to infer the underlying ISA description is an approximation.

exception of PowerVR, which doesn't have double precision. Vector-specific instructions are also supported for shuffling elements or performing a reduction operation.

Memory Operations: GEM ISA implements memory load/store and atomic operations via message passing. Prefetching is done in hardware automatically. In addition to standard load/store operations, RDMA, CGN, and PTX ISAs provide explicit memory prefetching instructions.

GPU Operations: Texture sampling instructions are defined on all ISAs, the same as for non-texture resources like depth and stencil buffers. PTX adds explicit instructions for loading pre-filterd texture data and querying texture states. On GEM, all texture query and filtering operations are handled via message passing. RDNA, CGN, and GEM provide explicit instructions for interpolating gradient values. PowerVR has dedicated graphics instructions for pixel iteration, alpha testing, and depth testing.

In summary, most GPGPU architectures that support the SIMT execution model share the following features: 1) some threading and memory hierarchy, 2) thread control and synchronization structures, and 3) memory synchronization. In designing the Vortex ISA, we couldn't support predication because of RISC-V dependency. To support thread divergence, we couldn't rely on using registers to store the divergence stack as it is done in AMD GPUs because RISC-V doesn't have enough free registers. We opted for an explicit split/join instruction within the internal hardware architecture. We also opted to support a texture sampling instruction for graphics workloads because texture lookup operations are usually a performance bottleneck in the software rendering pipeline. For memory synchronization, we leveraged the RISC-V fence instruction.

3.2 Vortex ISA

Vortex extends the RISC-V ISA to support GPGPUs by adding six new instructions: *wspawn*, *tmc*, *split*, *join*, *bar*, and *tex*, as shown in Table 2. They are all RISC-V R-Type instructions and fit in one opcode. They provide minimal ISA addition to handle wavefront activation, thread control, control divergence, synchronization, and texture filtering, the essential computational primitives to support SIMT execution model and graphics processing.

Wavefront Control: We propose a *wspawn* instruction to activate a number of wavefronts at a specific program's PC value, enabling multiple instances of that program to execute independently.

Thread Control: We propose a *tmc* instruction to activate or deactivate threads within a wavefront via a thread mask register, which is also accessible via the control status registers (CSRs).

Control Divergence: We propose the *split* and *join* instructions to handle control divergence. The *split* instruction pushes information about the current state of the thread mask and the branch predication result for all threads into a hardware-immediate post-dominator (IPDOM) stack [40], and the *join* instruction pops this out during reconvergence.

Synchronization: We propose a *bar* instruction to synchronize wavefront execution at barrier locations. A barrier is released when an expected number of wavefronts reach it. In addition, the barrier ID encodes whether it has local scope (intra-core) or global scope (inter-core).

Instructions	Description
wspawn %numW, %PC	Wavefronts activation
tmc %numT	Thread mask control
split %pred	Control flow divergence
join	Control flow reconvergence
bar %barID, %numW	Wavefronts barrier
tex %dest, %u, %v, %lod	Texture sampling/filtering

Table 2: Proposed RISC-V Vortex ISA extension.

Texture Filtering: We propose a *tex* instruction for texture lookup. The instruction follows the R4 type format of RISC-V ISA, currently used for FMA operations. It has three source operands, namely, *u, v, lod*, which specify the normalized coordinates of the source texel and the texture mipmap to use for the lookup. Other texture states (dimension, format, filtering mode, addressing mode, and memory address) are configurable via CSRs.

4 HARDWARE IMPLEMENTATION

4.1 Vortex Microarchitecture

Figure 4 details the various components of the Vortex microarchitecture, which implements a standard five-stage in-order RISC-V pipeline augmented by the following SIMT hardware components: 1) *hardware wavefront scheduler* that contains the PC, thread mask registers, and an IPDOM stack - 2) *banked GPRs* that contain the general-purpose registers for each thread in each wavefront - 3) *high-bandwidth caches* with parallel access by the threads in the active wavefront - 4) *barrier control module* for wavefront-level synchronization. The processor implements a scalable architecture that allows clustering of multiples cores with optional L2 and L3 caches. A command processor (AFU) manages the onboard memory system and the communication with the host processor via PCIe.

4.1.1 Wavefront Scheduler. The wavefront scheduler in the fetch stage decides what to fetch from the I-cache (see Figure 4). It has two components: 1) a set of wavefront masks to choose the wavefront to schedule next and 2) a wavefront table that includes private information for each wavefront. The scheduler uses four thread masks: 1) an active wavefront mask, each bit indicating whether or not a wavefront is active, 2) a stalled wavefront mask indicates which warps should not be scheduled temporarily, 3) a barrier mask for stalled wavefronts waiting at a barrier instruction, and 4) a visible wavefront mask to support hierarchical scheduling policy [51]. In each cycle, the scheduler selects one wavefront from the visible wavefront mask and invalidates that wavefront. When a visible wavefront mask is zero, the active mask is refilled by checking which wavefronts are currently active and not stalled.

4.1.2 Threads Masks and IPDOM Stack. To support SIMT, a thread mask register and an IPDOM stack have been added to the hardware, similar to other SIMT architectures [28]. When a split instruction is executed by a wavefront, the predicate value for each thread is evaluated. In the case of divergence, 1) the current thread mask is pushed into the IPDOM stack a as fall-through; 2) the active threads with false predicate are pushed into the stack with the next PC;

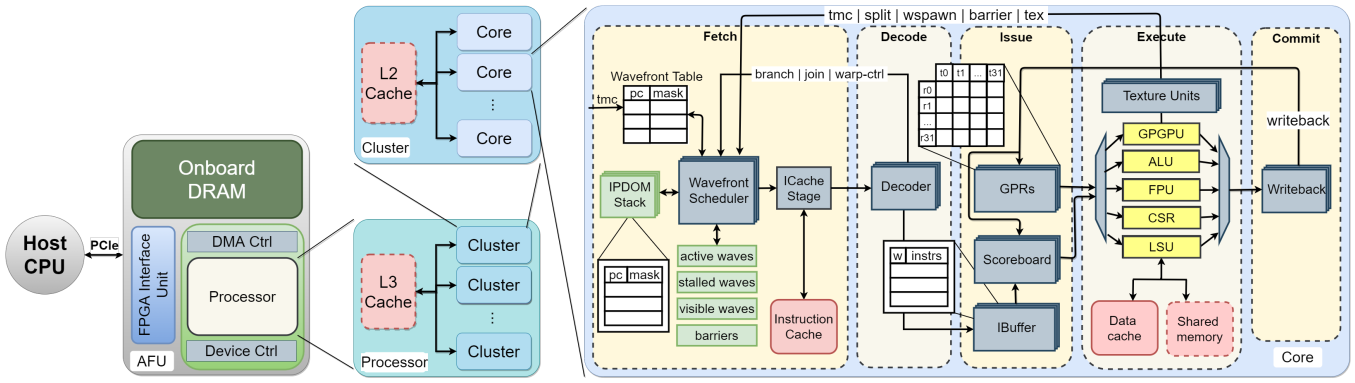

Figure 4: Vortex microarchitecture.

and 3) execution resumes with the thread mask set to the active threads with *true* predicate. When a join instruction is executed, the stack is popped and the thread mask is set to the stored value. If the popped entry it is not a fall-through, execution resumes at the stored PC.

4.1.3 Wavefront Barriers. Barriers are provided in the hardware to support synchronization between wavefronts. A barrier table keeps the following information for each entry: 1) a counter of the number of wavefronts left that need to execute the barrier, and 2) a mask of wavefronts stalled by the barrier. A similar table is also used for global barriers in multi-core configurations where the MSB of the barrier ID indicates global scope. When a barrier instruction is executed, the processor updates the barrier counter and mask accordingly. If the counter is zero, the mask is used to release the stalled wavefronts.

4.1.4 Memory system. Each core has an instruction cache and data cache. An optional shared memory is also available that can act as scratchpad memory or a stack depending on the application. Cores can be grouped into a cluster that can optionally be attached to a shared L2 cache. Clusters can share an optional L3 cache. Flush operations among caches are provided as a means of providing weak coherent memory space.

4.2 3D Graphics Support

Algorithm 1: Trilinear Filter

```
1: function TRILINEAR(stage, u, v, lod)
2:     a ← TEX(stage,u,v,lod)
3:     b ← TEX(stage,u,v,lod+1)
4:     return LERP(a, b, FRAC(lod))
5: end function
```

4.2.1 Hardware Texture Filtering . The hardware implements configurable texture units for graphics support. Each texture unit implements point sampling and bilinear sampling on 1D and 2D textures given *(u, v)* source coordinates and a *lod* operand to specify the level of detail in the texture. Advanced filtering algorithms like trilinear or anisotropic filtering are implemented as pseudo-instructions,

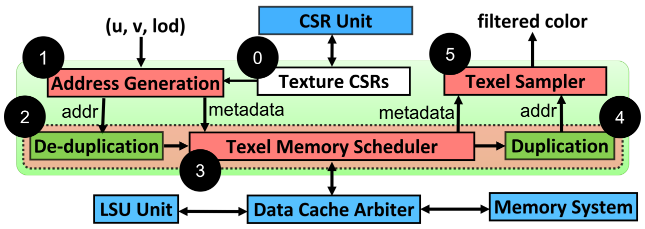

Figure 5: Texture unit microarchitecture.

invoking multiple *tex* instructions to average of filtering operations across mipmaps (see algorithm 1). The implementation supports various texture formats and texture wrap modes as defined by OpenGL[12].

4.2.2 Texture Unit Microarchitecture. Figure 5 shows the microarchitecture of a texture unit. It implements three main stages - the texture address generator ❶, the texture memory system ❶ ❸ ❹, and the texture sampler ❺. The device is configured via CSRs by the kernel, and the number of active texture states is configurable.

When a *tex* instruction is issued to the texture unit, the *u, v, lod* arguments are used to retrieve the relevant control information for the texture operation from the CSRs ⓿. The mipmap-specific base address, along with wrap and stride information from the CSRs, are passed to the address generator, where, given the filtering mode, point or bilinear, the (u, v) values are converted to texel addresses (single for point and quad for bilinear) for all the threads in parallel ❶. These texel addresses, along with metadata - wavefront-id, format, and blend values - are passed to the texture memory unit. The texture memory unit first de-duplicates memory accesses that are repeated across threads ❷. The batch of unique addresses, along with instruction metadata, are passed to the texel memory scheduler for issue to the data cache ❸. Upon the cache response, the returned texels are duplicated and piped into a buffer waiting to feed the texture sampler ❹. Only when all the texels in the batch have returned does the scheduler begin servicing the next batch. The texel sampler performs a format conversion and a two-cycle bilinear interpolation on incoming texels. Finally, a filtered RGBA color is generated per thread and sent out of the texture unit ❺.

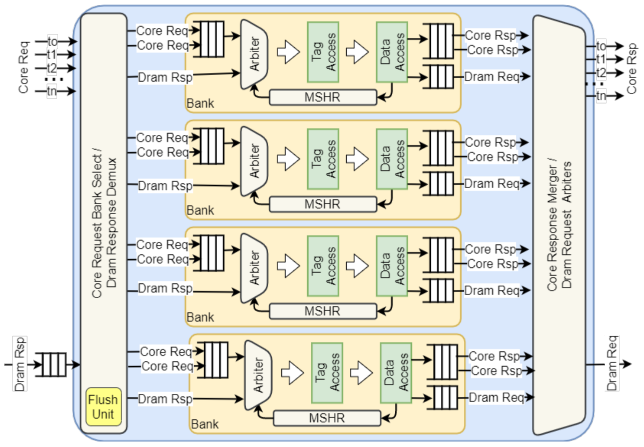

Figure 6: High-bandwidth cache.

Algorithm 2 Virtual Ports Assignment

```
for $i \leftarrow 1$ to $NUM\_REQS$ do
    $m \leftarrow (req.line[i] == bank.line[\text{req.bank}[i]])$
    if (req[$i$]) $ports[i\%NUM\_PORTS] \leftarrow m$
end for
```

This sampler closely resembles the sampler in [68], the difference being that their implementation runs on a different mobile graphics API with custom bit-widths, whereas our sampler supports OpenGL color formats. The texel sampler implements only bilinear filtering. Point sampling is executed using bilinear filtering with blend values of 0. Although point sampling would have only taken one cycle, the overhead of muxing and synchronization required to support a variable-latency sampler delay is not worth a single cycle gain. The texture unit microarchitecture is inspired by [27] and [68].

4.3 High-Bandwidth Caches

Modern GPGPUs [15] [48] [14] today integrate non-blocking high-bandwidth (NBHB) caches to mitigate the memory pressure, allowing the cache subsystem to process multiple independent requests concurrently. NBHB caches implemented on FPGAs use different techniques to reduce the high cost of ports in memory devices: 1) multi-banking [38], the common solution, partitions the cache into single-ported banks, which introduces bank conflicts; 2) virtual multi-porting or multi-pumping [20] exploits the higher clock speed of memory devices to process multiple requests using bus time-sharing. This solution is constrained by the clock speed of the memory to operate at 2x the base frequency; 3) the Live-value Table (LVT) [42] approach replicates the memory for each read and write port and maintains separate LVT storage to keep track of the memory block holding recently written addresses. LVT caches have higher area and storage requirements compared to the previous approaches. Our implementation use a hybrid solution that extends multi-banking with virtual ports exploiting cache line locality.

Figure 6 describes the high-bandwidth cache microarchitecture used in Vortex. It is a multi-banked, non-blocking pipelined cache architecture. Each bank maintains its own miss status holding register (MSHR) to reduce miss rate, a solution adapted from [8]. The pipeline has four-stages: 1) schedule, where the next request into the pipeline is selected from the incoming core request, the memory fill, or the MSHR entry, with priority given to the latter; 2) tag access; a single-port access to the tag store; 3) data access, single-port access to the data store; 4) response, handling core response back to the core. At the back-end is the bank merger where outgoing responses from the banks are coalesced based on their request tag. The front-end of the cache is the bank selector where the incoming core requests are assigned to individual banks based on their address. The bank selector also resolves bank conflicts by selecting a single request going into a bank at the time. If virtual ports are enabled, the bank selector will coalesce requests that map to the same bank and the same cache line. Algorithm 2 shows the pseudo-code of the virtual port selection where a modulo operation is used to update the matching valid bit of each port. Using virtual ports in this scheme is efficient in two ways: 1) minimal storage is needed for the virtual ports as we only need to store the word offsets for each port in the MSHR; 2) the output of the data access, which is a full block, can now be fully utilized during reads. A deadlock inside the cache can occur in two ways: 1) when the MSHR is full and a new request is already in the pipeline, and 2) when the memory request queue is full and there is an incoming memory response. We mitigate the MSHR deadlock by using an early full signal before a new request is issued. We mitigate the memory deadlock similarly by ensuring that its request queue never fills up.

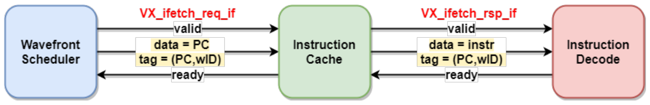

Figure 7: Elastic pipeline request.

4.4 Elastic Pipelines

Vortex was designed with the primary goal for architecture research; it was important at the beginning to set the foundations that would make it easier to maintain and modify the hardware architecture. We originally explored using a hardware construction language (HCL) [9] [5] [62] but reverted back to using Verilog for greater adoption and reach. We implemented Vortex from the ground up enforcing elastic [22] [53] [33] design patterns across all main architecture components, sub-components, including libraries (arbiters, muxes, crossbars, etc.). Maintaining this consistency throughout the codebase makes it possible to support the following features: 1) extensibility: the elastic handshake protocol is simple and intuitive, allowing flexibility for easy extensions, and 2) tracing and debugging support: elastic-based pipeline requests are assigned tags, which consist of the instruction PC and wavefront identifier that track the life cycle of instructions and other request types inside the processor. We leveraged SystemVerilog's Interface construct to implement all the elastic connections in the design. Figure 7 illustrates an example for the instruction fetch request issued from wavefront scheduler as it enters the instruction cache and exits as a new response interface carrying the fetched instruction while still preserving its original tag as it enters the decode stage.

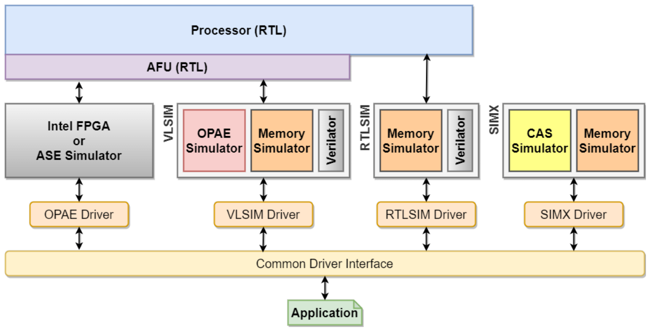

Figure 8: Vortex simulation stack.

4.5 Hardware Simulation

Vortex integrates an advanced simulation infrastructure to validate the implementation and perform design-space exploration. Figure 8 shows the Vortex simulation stack, which includes four simulation environments: 1) OPAE driver uses Intel's proprietary AFU Simulation Environment (ASE) [36] to simulate the full design; 2) VLSIM driver uses Verilator [59] to simulate the full RTL design and implements the AFU interface and memory simulation in software; 3) RTLSIM driver simulates the processor RTL without the command processor (AFU) to emulate SOC environment where the host and accelerator share the same memory interface; 4) SIMX driver implements a cycle-level simulator for Vortex and is ideal for architecture design-space exploration. All drivers share a common API that applications use when executing on the platform, whether it is targeting the actual FPGA or a specific simulator.

5 SOFTWARE SUPPORT

5.1 Vortex Driver Stack

The Vortex software stack primarily integrates a driver for handling the kernel interface to access the FPGA via the PCIe bus. Figure 9 shows the FPGA driver connections.

We use OPAE (Open Programmable Acceleration Engine) [36], a lightweight user-space open-source C library, as a driver to provide abstractions of FPGA resources as a set of features accessible for software running on the host. It configures the FPGA, read/write instructions, and data to/from the RAM present on the FPGA. It uses the CCI-P (Core Cache Interface) protocol to assign a shared memory space, accessible by the Accelerator Functional Unit (AFU) and host, for data transfer. The data is read from the shared space and written into FPGA local memory. Vortex is then reset to start execution, and once the operation is complete, the result is stored in local memory. The result data is then moved from local memory to the shared space accessible by the host using MMIO.

5.2 OpenCL Compiler and Runtime

OpenCL is the main parallel API supported on Vortex. We used the POCL [37] open-source framework to implement the compiler and runtime software for OpenCL. The POCL compiler back-end was modified to generate kernel programs targeting the Vortex ISA and the POCL runtime was modified to access the Vortex driver, enabling communication with the FPGA via PCIe.

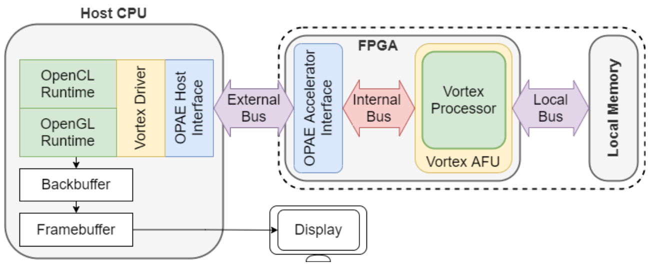

Figure 9: Vortex driver stack and frame buffer connection.

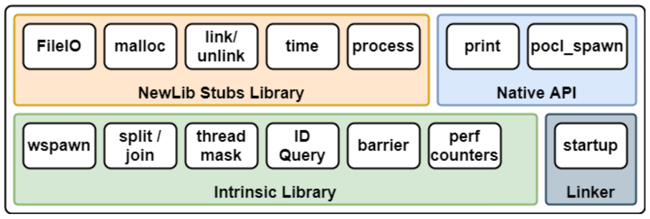

Figure 10: Runtime system for Vortex.

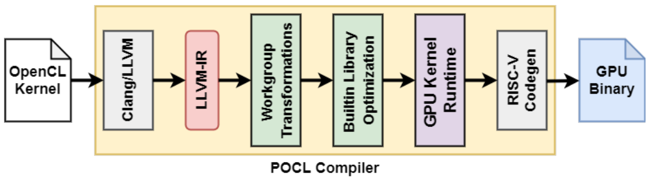

Figure 11: Vortex binary generation steps for OpenCL applications.

5.3 Vortex Native Runtime

The Vortex software stack implements a native runtime that exposes the new SIMT functionalities provided by the RISC-V ISA extension and basic resource management API to kernel programs running on Vortex. Figure 10 shows an overview of the runtime system. We statically link the runtime library with OpenCL kernels during POCL compilation.

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 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.

5.4 POCL Backend Compiler

The POCL back-end compiler is responsible for generating the OpenCL kernel binaries given their source code, as shown in Figure 11. We modified POCL to achieve the following goals: (1) support RISC-V by adding new devices and compiler support (the details of RISC-V support is discussed in [13]), (2) support new Vortex instructions, (3) integrate with Vortex runtime system.

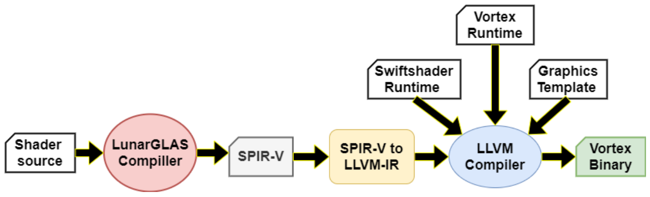

Figure 12: Shader compilation pipeline.

```
1  int main(kernel_arg_t* arg) {
2    // configure texture unit
3    csr_write(TEX_ADDR(0),   arg->src_ptr);
4    csr_write(TEX_MIPOFF(0), 0);
5    csr_write(TEX_WIDTH(0),  arg->srcW);
6    csr_write(TEX_HEIGHT(0), arg->srcH);
7    csr_write(TEX_FORMAT(0), arg->format);
8    csr_write(TEX_WRAP(0),   arg->wrap);
9    csr_write(TEX_FILTER(0), arg->filter);
10
11   shader_state_t state;
12   state.arg    = arg;
13   state.tileW  = arg->dstW;
14   state.tileH  = arg->dstH;
15   state.deltaX = 1.0f / arg->dstW;
16   state.deltaY = 1.0f / arg->dstH;
17
18   // launch rendering tasks
19   spawn_tasks(shader, state);
20 }
```

Figure 13: A sample code kernel with texture rendering.

5.5 Graphics Support

The Vortex graphics API implements the OpenGL-ES specification with the geometry processing running on the host processor and the rasterization pipeline running as a kernel on the Vortex parallel architecture. Running geometry processing on the host allows the accelerator to fully utilize its processing resources for the more compute-and-memory-intensive rasterization tasks. The rasterizer implements basic point, line, and triangle primitives, and fragment processing including depth, stencil, fog, and alpha tests. Texture sampling is accelerated via the new *tex* instruction, which executes as part of the fragment shader. The rasterizer's implementation follows Larrabee [58]'s tile-rendering algorithm, with the rasterization tiles generated on the host.

Figure 12 shows an overview of the compilation pipeline for Vortex programs, which also includes a step for compiling the graphics shaders. The LunarGLASS [47] compiler internally uses LLVM [44] Clang as part of its front-end to process the source kernel code into the LLVM IR (through SPIR-V to LLVM IR conversion). The LLVM-IR program is passed to the LLVM compiler with additional information, including the Vortex runtime and the graphics kernel template, to generate the final Vortex binary. Figure 13 shows a code-snippet of a kernel that invokes a shader with texture filtering. The texture sampler states are programmed via CSRs (lines 3-9); then, the kernel spawns the shader execution on the available hardware threads (line 19).

6 EVALUATION

6.1 Experimental Setup

Our evaluation setup consisted of a 3.5 GHz Intel Xeon E5-1650 for the host processor. For the benchmarks, we use a subset of the Rodinia [19] OpenCL kernels. We classified the benchmarks into a compute-bounded group that includes *sgemm*, *vecadd*, and *sfilter*, and a memory-bounded group that includes *sxapy*, *nearn*, *gaussian*, and *bfs*. To evaluate the texture engine, we use three synthetic benchmarks to exercise the supported filtering modes, including point sampling, bilinear filtering, and trilinear filtering. The texture benchmarks all use a 1080p source texture as input and renders it into a destination render target of the same size. We synthesized Vortex RTL on both Intel Aria 10 GX FPGA and Intel Stratix 10 FPGAs with speed grade 2.

6.2 Microarchitecture

6.2.1 Design Space Configurations. In Vortex design, we can increase the data-level parallelism by either increasing the number of threads or increasing the number of wavefronts. 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 entries. Increasing the number of wavefronts does not require increasing the number of ALUs because the ALUs are multiplexed among wavefronts. Increasing the number of wavefronts involves the following changes to the hardware: 1) increasing the logic for the wavefront 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 wavefront table. It is important to note that the cost of increasing the number of wavefronts is dependent on the number of threads in that wavefront; thus, increasing wavefronts for larger thread configurations becomes more expensive. This is because the size of each GPR table, IPDOM stack, and wavefront table is dependant on the number of threads.

	4W-4T	2W-8T	8W-2T	4W-8T	8W-4T
LUT	21502	36361	16981	37857	24485
Regs	32661	54438	24343	57614	34854
BRAM	131	238	77	247	139
f(MHz)	233	224	225	224	228

Table 3: Synthesis results for different core configurations.

Table 3 shows the area costs of various configurations of a processor core as we increase the number of wavefronts (i.e. 4W, 8W) or the number of threads (i.e. 4T, 8T). Figure 14 shows the corresponding performance at the different configurations. Moving from a 4W-4T configuration[3] to a 2W-8T configuration, maximizing

[3]the configuration is per core.

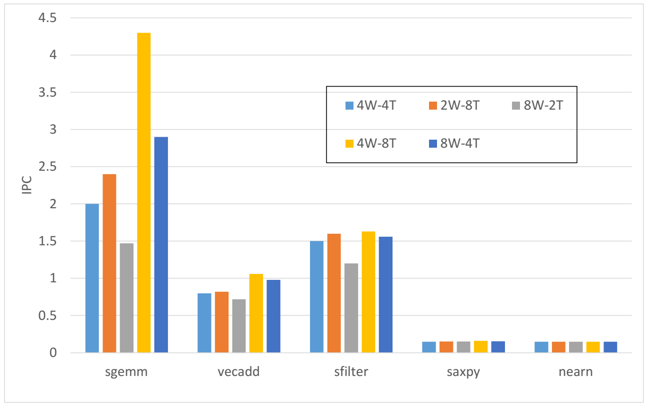

Figure 14: IPC results for different core configurations.

threads, introduces a 69% area cost increase in LUT and registers, as well as a speedup of 20% for sgemm. However, changing the configuration to 8W-2T, maximizing wavefronts, generates cheaper hardware, about a 27% area reduction. This comes with a reduction in performance in terms of IPC, 36% for sgemm in the extreme case. The 8W-4T configuration has some performance gains and a relatively less expensive area. We picked 4W-4T primarily to allow scaling to 16/32 cores on the target FPGAs while achieving good performance.

6.2.2 Area Cost. We managed to fit a baseline processor configuration with up to 16 cores on the Intel Arria 10 (A10) and up to 32 cores on the Intel Stratix 10 (S10) FPGA where we reached scaling up to 32 cores at a 200 MHz clock speed.

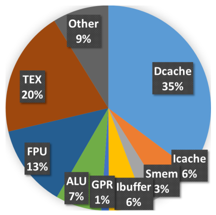

Figure 15: Area distribution.

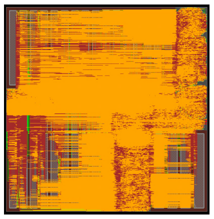

Figure 16: GDS layout.

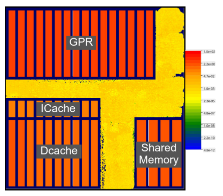

Figure 17: Power density.

cores #	ALM (%)	Regs	BRAM (%)	DSP	fmax MHz	FPGA
1	13	78K	10	2	234	A10
2	19	111K	15	5	225	A10
4	30	176K	25	9	223	A10
8	53	305K	45	19	210	A10
16	85	525K	83	38	203	A10
32	70	1057K	23	20	200	S10

Table 4: Hardware synthesis for all core configurations.

Table 4 shows the synthesis summary of the processor at different core configurations, and a breakdown of the area utilized by the main components is shown in Figure 15. At eight cores, 53% of Arria 10 FPGA's logic is utilized and that cost is occupied primarily by the texture units and caches (16KB for L1 caches and shared memory). The FPU area is relatively low because we utilize the existing floating-point DSP blocks on the device for FMA computations.

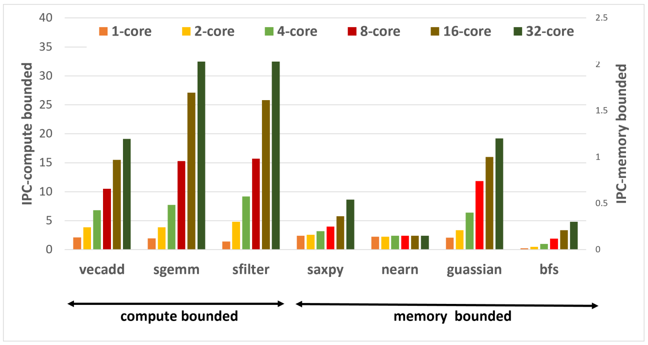

Figure 18: Vortex performance scaling.

6.2.3 Performance Scaling. Figure 18 shows the performance scaling of the Vortex processor at various core configurations on the FPGA in terms of IPC. For the compute-bounded benchmarks, the IPC increases almost linearly with the addition of cores into the processor. For the memory-bounded benchmarks, the results still see some IPC increase with the core count, with the exception of the *nearn* program, which is also compute-bound with an expensive long-latency floating-point square-root operation inside its kernel.

6.3 High-bandwidth Cache

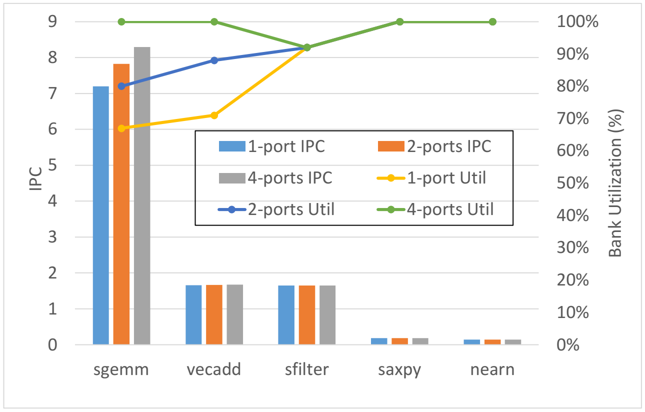

Figure 19: The effect of multi-port caches.

We analyzed the performance of our high-bandwidth caches for our baseline 4W-4T processor configuration. For this setup, we focused only on single-core performance and varied the number of virtual ports on the data cache bank. We need to point out that only the data-cache implements virtual-multi-porting. The instruction

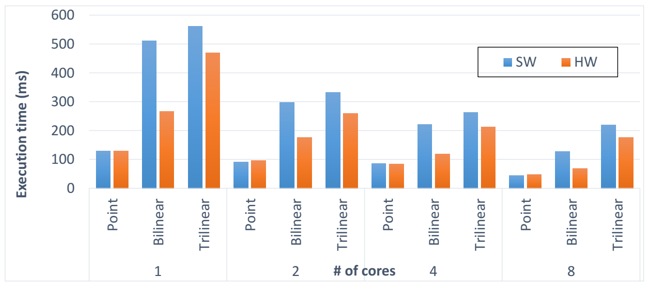

Figure 20: HW Texture acceleration vs software.

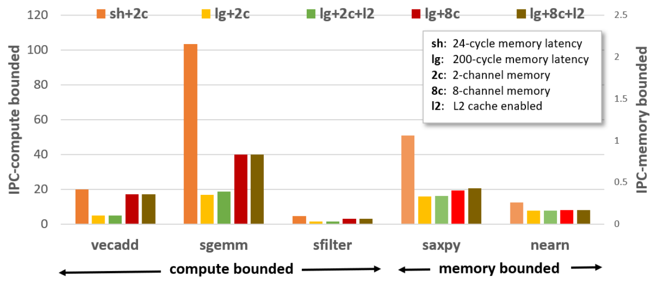

Figure 21: The effect of memory scaling on performance.

cache doesn't need it since SIMT execution needs to fetch only one instruction at a cycle. Table 5 shows the synthesis summary of a 4-bank data cache, with 1-port, 2-port, and 4-port virtual multi-porting enabled. four ports is the maximum setting possible, which improves the worst-case scenario where all four requests go to the same bank and occupy the four individual virtual ports on that bank. The port increase from one to two adds a 9% increase in logic area and from one to four adds a 25% increase. Figure 19 shows the data cache bank utilization for each virtual port configuration. A 100% bank utilization means that all requests that were issued did not directly experience bank conflicts and that all stalls originated from the banks' input FIFOs being full. *sgemm* and *vecadd* are the two benchmarks that mainly experienced high bank conflict with bank utilization at 67% and 71%, respectively. Increasing the number of virtual ports linearly increases the bank utilization of those benchmarks up to 100%. Figure 19 shows each benchmark performance for each virtual port configuration, and we observe that *sgemm* considerably benefits from this optimization vecadd IPC also increases by a slight amount, but the change doesn't show well due to chart scale. The 2-port configuration has the best balance between improved utilization and cost.

	1-port	2-port	4-port
LUT	10747	11722	13516
Registers	13238	13650	14928
BRAM	72	72	72
Frequency (MHz)	253	250	244

Table 5: Virtual multi-ported 4-bank cache synthesis results.

6.4 Texture Sampling

Our evaluation of the texture acceleration is based on synthetic benchmarks that directly exercise the custom hardware. We evaluated point sampling, bilinear sampling, and trilinear sampling. As discussed in Section 4.2.1, trilinear sampling is implemented as a pseudo-instruction around the accelerated bilinear sampler. We compare Vortex acceleration (HW) with a rendering pipeline with no acceleration where the texture unit is implemented fully in software(SW). Figure 20 shows the performance difference between software and hardware texture acceleration for different processor core configurations. We observe that the point-sampling difference is very negligible across all core configurations. This is expected because, as mentioned in Section 4.2.1, point sampling acceleration shares the sample filter back-end with bilinear sampling to reduce area cost along with the fact that the feature is not commonly used. Also, the source texture used in this experiment has an RGBA format, meaning format conversion is unnecessary, causing the point-sampling software code to turn into a simple copy operation. The bilinear filter, on the other hand, shows more improvement, with an almost 2x speed up on a single core where the memory bandwidth is less saturated. As the core count increases, that speed is slightly reduced due to memory bandwidth. Trilinear filtering also better with hardware acceleration although the gains are not as strong when compared with bilinear filtering, mainly due to memory bandwidth since trilinear doubles the number of requests to the memory. Looking at texture acceleration standalone, we also observe the effect of memory contention as the number of cores increase.

6.5 Using Vortex in Architecture Research

The Vortex infrastructure provides a complete implementation of a GPU stack on an FPGA that enables the exploration of full-system optimizations across the application, compiler, driver, and hardware stacks in both desktop and SoC environments. To the best of our knowledge, this is the first soft GPU implementation that supports a PCIe interface, which opens new scenarios that deal with CPU/GPU communication, command buffer management, and kernel offloading. Its high-bandwidth cache subsystem connected to the FPGA multi-bank memory system (2 on A10 and 8 on S10) provides a solid platform for exploring memory optimizations. Vortex can be easily extended to evaluate on HBM based FPGAs [67] to further evaluate different memory systems. The simulation tools in Section 4.5 enable the design-space exploration of more complex architectures that cannot fit on FPGAs. Figure 21 shows the effect of memory scaling for a 16-core, 16-wavefront, 16-thread processor configuration as we increase the memory latency and bandwidth using SIMX (Section 4.5) with the baseline RTL design parameters.

6.6 Porting Vortex to ASIC Design Flow

A solid simulation platform coupled with a comprehensive FPGA prototyping environment provides a robust infrastructure for exploring ASIC development. Early during Vortex development [24], we synthesized an 8-wavefront-4-thread single-core Vortex configuration using a 15-nm educational cell library, obtaining a 46.8mW design running at 300 MHz. (See Figure 16 and Figure 17 for the

GPGPU	ISA	Exec Model	Cache System	Memory System	Graphics Suppport	Threads x Cores	RTL	Host Interface	Software Stack	Cycle-level Simulation
HWACHA	RISCV	Vector	L1,L2	Simulated	No	N/A	Yes	No	N/A	No
Simty	RISCV	SIMT	No	No	No	1x1	Yes	No	N/A	No
MIAOW	AMD	SIMT	No	Simulated	No	N/A	Yes	N/A	OpenCL	No
FlexiGrip	Custom	SIMT	sharedm	Simulated	No	32x1	Yes	SoC	Custom	No
FGPU	Custom	SIMT	L2	FPGA	No	64x8	Yes	SoC	Custom	No
NyuziRaster	Custom	SIMT	L1,L2	FPGA	Fixed-Function Rasterizer	4x1	Yes	N/A	Custom	No
Vortex	**RISCV**	**SIMT**	**sharedm L1,L2,L3**	**FPGA**	**Shaders Texture Units**	**16x32**	**Yes**	**PCIe**	**OpenCL OpenGL**	**Yes**

Table 6: Comparisons of open-source GPPGUs.

GDS layout and power design distribution, respectively). However, Vortex's microarchitecture was optimized for FPGAs, and porting the design to ASIC requires changes to address platform differences with FPGAs such as clock tree, reset distribution, power management, memories, and performance, which is outside the scope of our current work.

7 RELATED WORK

Table 6 contrasts Vortex with other open-source GPGPU implementations, highlighting the provided features and performance characteristics. The details about each project and comparison with Vortex are summarized below.

7.1 RISC-V extension to support GPGPU/GPU

HWACHA [45] and ARA [18] are RISC-V-based co-processors that implement an SIMD execution model, where vector instructions are streamed into vector lanes. Their design is based on the open-source RISC-V Vector ISA Extension proposal [6] taking advantage of its vector-length agnostic ISA and its relaxed architectural vector registers.

Simty [21] implements a specialized RISC-V architecture that supports SIMT execution similar to Vortex. However, in the authors' work, only the microarchitecture was implemented as a proof of concept without any software stack.

7.2 FPGA based GPUs

MIAOW [11] is an FPGA soft GPU that implements the AMD Southern Islands GPGPU ISA and is capable of running unmodified OpenCL-based applications. The authors proposed a partial architecture in which most of the on-chip network and memory subsystem are simulated. Their main goal was to provide the closest realistic implementation of the reference architecture for the components written in RTL. On the other hand, the goal of Vortex is not to replicate a specific GPGPU architecture but instead to provide a complete comparable implementation that is optimized for FPGAs. Furthermore, MIAOW doesn't support graphics.

FlexiGrip [4], FGPU [1], and Harmonica [40] are also soft GPUs that are implemented for FPGAs. They all have an SIMT-based architecture, but they have their own custom ISA, which requires porting existing applications and benchmarks. They do not support graphics.

7.3 Soft GPUs with rendering

NyuziRaster [17] is an open-source soft GPU with graphics rendering support. NyuziRaster integrates a simple multi-threaded in-order processor that supports a custom ISA. NyuziRaster doesn't implement any texture unit and does texture sampling completely in software. NyuziRaster implements a fixed-function rasterizer with no programmable shader support. Vortex supports programmable shaders via OpenGL that execute as parallel tiles on its compute platform. It also has hardware accelerated texture sampling. NyuziRaster can support up to four threads in its processor design, while Vortex can scale up to 512 total threads on FPGA.

8 CONCLUSION

By leveraging the fast-growing open-source community around RISC-V and the open-source LLVM and POCL compilers, Vortex tries to present a holistic approach for GPGPU research that explores new ideas at any part of the hardware and software stacks. With its minimal ISA extensions, Vortex implements GPGPU functionality and 3D graphics acceleration. This, along with its high-bandwidth caches, and its elastic pipeline, enables a design that achieves a high frequency on FPGAs. A configurable RTL and a tightly coupled runtime stack allow for quick yet flexible experimentation, which we hope is evident from the variety of evaluation metrics we presented. We believe this will allow for increasingly diverse and complex workloads to be deployed on Vortex, leading to research on more realistic and meaningful scenarios. For future work, we plan to extend Vortex's compiler and runtime software to support CUDA and Vulkan APIs. The support for the ASIC design flow is also an essential roadmap to chip fabrication.

9 DEDICATION

We dedicate this work to the memory of our great advisor, colleague, mentor, and dear friend, Prof. Sudhakar Yalamanchili. The project was started with his vision of building an open source GPU framework for research.

ACKNOWLEDGMENTS

This work was partially supported by Oak Ridge National Laboratory and SiliconArts. We gratefully acknowledge the support of Intel Corporation and NSF CCRI 2016701, NSF CNS 1815047 for providing FPGA resources. We would like to thank Ruei Ting Chien, Kanghong Yan, Will Gulian, Jaewoong Sim, Liam Cooper, Xingyang Li, Malik Burton, Carter Montgomery, Da Eun Shim, Priyadarshini Roshan, Jaewon Lee, Ethan Lyons, Roye Eshed, Taejoon Park, Lingjun Zhu, Sung Kyu Lim, Han Ruobing for their contribution to the Vortex project development. We also thank HPArch group members, Jeff Young, Seyong Lee, Jeff Vetter, Chad Kersey, and the anonymous reviewers for their feedback on improving the paper.

REFERENCES

[1] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. 2016. FGPU: An SIMT-architecture for FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 254–263.

[2] AMD. [n.d.]. RDNA 1.0 Instruction Set Architecture. https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf.

[3] AMD. [n.d.]. RDNA 1.0 Instruction Set Architecture. http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf.

[4] Kevin Andryc, Murtaza Merchant, and Russell Tessier. 2013. FlexGrip: A soft GPGPU for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT).* IEEE, 230–237.

[5] Arvind. 2003. Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification Invited Talk. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03).* IEEE Computer Society, Washington, DC, USA, 249–. http://dl.acm.org/citation.cfm?id=823453.823860

[6] Krste Asanovic. [n.d.]. *RISC-V Vector Extension.* https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

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

[8] Mikhail Asiatici and Paolo Ienne. 2019. Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 310–319.

[9] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE.* 1212–1221. https://doi.org/10.1145/2228360.2228584

[10] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software.* IEEE, 163–174.

[11] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, and Pradip Valathol. 2015. Miaow-an open source rtl implementation of a gpgpu. In *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII).* IEEE, 1–3.

[12] Lars Bishop. 2006. OpenGL ES 1.1, 2.0 and EGL. In *ACM SIGGRAPH 2006 Courses.* 3–es.

[13] Tine Blaise, Seyong Lee, Jeff Vetter, and Hyesoon Kim. 2021. Bringing OpenCL to Commodity RISC-V CPUs. In *2021 Workshop on RISC-V for Computer Architecture Research (CARRV).*

[14] Ian Bratt. 2015. The arm® mali-t880 mobile gpu. In *2015 IEEE Hot Chips 27 Symposium (HCS).* IEEE, 1–27.

[15] John Burgess. 2020. Rtx on—the nvidia turing gpu. *IEEE Micro* 40, 2 (2020), 36–44.

[16] Jeff Bush, Philip Dexter, Timothy N Miller, and Aaron Carpenter. 2015. Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 173–182.

[17] Jeff Bush, Mohammad A Khasawneh, Khaled Z Mahmoud, and Timothy N Miller. 2016. NyuziRaster: Optimizing rasterizer performance and energy in the Nyuzi open source GPU. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 204–213.

[18] Matheus A. Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2019. Ara: A 1 GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI. *CoRR* abs/1906.00478 (2019). arXiv:1906.00478

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC).* 44–54.

[20] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. 2012. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines.* IEEE, 17–24.

[21] Sylvain Collange. 2017. Simty: generalized SIMT execution on RISC-V. In *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017).* 6.

[22] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. 2010. Elastic systems. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010).* IEEE, 149–158.

[23] Victor Moya Del Barrio, Carlos González, Jordi Roca, Agustín Fernández, and E Espasa. 2006. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software.* IEEE, 231–241.

[24] Fares Elsabbagh, Blaise Tine, Priyadarshini Roshan, Ethan Lyons, Euna Kim, Da Eun Shim, Lingjun Zhu, Sung Kyu Lim, and Hyesoon Kim. 2020. Vortex: OpenCL Compatible RISC-V GPGPU. *CoRR* abs/2002.12151 (2020). arXiv:2002.12151 https://arxiv.org/abs/2002.12151

[25] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA).* 365–376.

[26] Jon P Ewins, Marcus D Waller, Martin White, and Paul F Lister. 1998. Mip-map level selection for texture mapping. *IEEE Transactions on Visualization and Computer Graphics* 4, 4 (1998), 317–329.

[27] Kayvon Fatahalian. [n.d.]. Lecture 15: Optimizing Data Access in the Graphics Pipeline. http://cs348k.stanford.edu/fall18/lecture/gfxmemory.

[28] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. IEEE Computer Society, 407–420. https://doi.org/10.1109/MICRO.2007.12

[29] Google. 2019. Google Stadia. https://stadia.google.com/.

[30] Green500. 2019. Green500 list - June 2019. https://www.top500.org/lists/2019/06/

[31] Ayub A Gubran and Tor M Aamodt. 2019. Emerald: graphics modeling for SoC systems. In *Proceedings of the 46th International Symposium on Computer Architecture.* 169–182.

[32] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. 2018. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 608–619. https://doi.org/10.1109/HPCA.2018.00058

[33] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic cgras. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays.* 171–180.

[34] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. 1998. Prefetching in a texture cache architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware.* 133–ff.

[35] Intel. [n.d.]. Intel Graphics Hardware Specifications. https://01.org/linuxgraphics/documentation/hardware-specification-prms.

[36] Intel. 2018. the Open Programmable Acceleration Engine (OPAE). https://01.org/opae.

[37] Pekka Jaaskelainen, Carlos Sanchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. POCL: Portable Computing Language. http://portablecl.org. *International Journal of Parallel Programming* (2015), 752–785.

[38] Mohammad Reza Kakoee, Vladimir Petrovic, and Luca Benini. 2012. A multi-banked shared-l1 cache architecture for tightly coupled processor clusters. In *2012 International Symposium on System on Chip (SoC).* IEEE, 1–5.

[39] Michael Kenzel, Bernhard Kerbl, Wolfgang Tatzgern, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing. *PACMCGIT* 1, 2 (2018), 28:1–28:17. https://doi.org/10.1145/3233303

[40] Chad D. Kersey, Hyesoon Kim, and Sudhakar Yalamanchili. 2017. Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia) *(MEMSYS '17).* ACM, 49–59. https://doi.org/10.1145/3132402.3132426

[41] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology* (2012).

[42] Charles Eric LaForest and J Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. 41–50.

[43] Samuli Laine and Tero Karras. 2011. High-performance software rasterization on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. 79–88.

[44] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[45] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. 2014. 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)*. 199–202. https://doi.org/10.1109/ESSCIRC.2014.6942056

[46] Alexander Lier, Marc Stamminger, and Kai Selgrad. 2018. CPU-style SIMD ray traversal on GPUs. In *HPG ’18*.

[47] LunarG. 2019. LunarGLASS Shader Compiler Stack. https://www.lunarg.com/.

[48] Mike Mantor. 2012. AMD Radeon™ HD 7970 with graphics core next (GCN) architecture. In *2012 IEEE Hot Chips 24 Symposium (HCS)*. IEEE, 1–35.

[49] Microsoft. 2019. Microsoft XCloud. https://www.xbox.com/en-US/xbox-game-streaming/project-xcloud/.

[50] A. Munshi. 2009. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. 1–314. https://doi.org/10.1109/HOTCHIPS.2009.7478342

[51] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling *(MICRO-44)*. ACM, 308–317. https://doi.org/10.1145/2155620.2155656

[52] NVIDIA. 2010. PTX: Parallel thread execution ISA version 2.3. http://developer.nvidia.com/compute/cuda.

[53] Rafael T Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. 2016. Fluid-Pipelines: Elastic circuitry without throughput penalty. In *Logic Synthesis (IWLS), Proceedings of the 2016 International Workshop on.*

[54] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. 2014. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters* 14, 1 (2014), 34–36.

[55] Kyle Roarty and Matthew D Sinclair. 2020. Modeling Modern GPU Applications in gem5. In *gem5 Users Workshop*.

[56] Ben Sander and AMD SENIOR FELLOW. 2013. HSAIL: Portable compiler IR for HSA.. In *Hot Chips Symposium*. 1–32.

[57] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[58] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)* 27, 3 (2008), 1–15.

[59] Wilson Snyder. [n.d.]. Verilator. https://www.veripool.org/wiki/verilator.

[60] Rys Sommefeldt. 2015. A look at the PowerVR graphics architecture: Tile-based rendering.

[61] Imagination Technologies. [n.d.]. PowerVR Instruction Set Reference. Rev 1.0. http://cdn.imgtec.com/sdk-documentation/PowerVR+Instruction+Set+Reference.pdf.

[62] Blaise-Pascal Tine, Sudhakar Yalamanchili, and Hyesoon Kim. 2020. Tango: an optimizing compiler for Just-In-Time RTL simulation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 157–162.

[63] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 335–344.

[64] Elena Vasiou, Konstantin Shkurko, Erik Brunvand, and Cem Yuksel. 2019. Mach-RT: A Many Chip Architecture for Ray Tracing. In *High-Performance Graphics - Short Papers*, Markus Steinberger and Tim Foley (Eds.). The Eurographics Association. https://doi.org/10.2312/hpg.20191188

[65] Ingo Wald, Will Usher, Nate Morrical, Laura Lediaev, and Valerio Pascucci. 2019. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *High-Performance Graphics - Short Papers*. https://doi.org/10.2312/hpg.20191189

[66] Li-Yi Wei. 2004. Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 55–63.

[67] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da. 2017. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. *Xilinx Whitepaper* (2017).

[68] Hoi-Jun Yoo, Jeong-Ho Woo, Ju-Ho Sohn, and Byeong-Gyu Nam. 2010. *Mobile 3D graphics SoC: From algorithm to chip*. John Wiley & Sons.