Redesigning the Vortex GPU ISA: 64-Bit Instruction Words and Conflict-Aware Register Allocation

Ruohan Richard Yan
UC Berkeley
yrh@berkeley.edu

Shashank Anand
UC Berkeley
shashank.anand@berkeley.edu

Abstract

The Vortex open source GPU's ISA and compiler stack contain performance bottlenecks due to a lack of architectural registers and prevalence of register bank conflicts. To mitigate these bottlenecks, we propose two key innovations: a 64-bit instruction set to expand register availability and a conflict-aware register allocation mechanism. We modify the LLVM stack to integrate our changes to the ISA and to the backend. Our evaluation, statically on the binary and dynamically using a custom ISA emulator demonstrate significant improvements, including a 99% reduction in register bank conflicts and elimination of register spills. These advancements lay the groundwork for exploring a high-performance GPU architecture.

1 Introduction

GPUs have come into the spotlight in recent years as a result of the skyrocketing demand for parallel compute required to do large language model inference [5], ushering in paradigm shifts in GPU architecture design. However, the two major players - NVIDIA and AMD - have proprietary designs with implementation details walled up. To facilitate GPU microarchitecture research, there had been numerous efforts to create a GPU research vehicle, ranging from simulators [3] and open source hardware [9][2]. In particular, a team at Georgia Tech has proposed Vortex [9], an open source GPGPU design written in SystemVerilog, which uses an extended version of the RISC-V ISA.

However, using the Vortex ISA led to two notable challenges: register spilling and register bank conflicts. This project aims to solve these issues by rethinking the instruction encoding, and extending upon their existing LLVM modifications to perform bank-aware register allocation.

In this report, we first detail the two challenges, and how our approach mitigates them. We discuss our ISA design and compiler modifications necessary to implement the new ISA. We then give a brief overview of our evaluation methodology, followed by preliminary data showing the effectiveness of our changes. Finally, we provide an outlook of what will follow after this project.

2 Motivation

The Vortex software stack utilizes LLVM, defining several new instructions intended to support SIMT execution, including warp and thread mask control. A pass was also added to insert split and join instructions at intra-warp branches, pushing and popping from the IPDOM stack. However, other aspects of their compiler stack remain largely unchanged from standard LLVM with a RISC-V backend. Outside of the handful of extra intrinsics, the GPU cores execute base RV32IMF instruction set. This design decision can be justified due to the complexity of designing a new dedicated GPU ISA and building a software stack around it.

One of our previous work under submission, Virgo, aimed to marry instances of the Vortex core with a matrix multiplication accelerator in a comprehensive SoC framework [4]. As part of our work, we implemented a FlashAttention kernel running matrix multiplication on Gemmini, the accelerator, and softmax on the Vortex SIMT core. We noticed two notable pain points when optimizing for performance and energy efficiency.

2.1 Register Spills

When writing a GEMM kernel, tiling is a common technique used to maximize reuse at each hardware memory level, such as the levels of caches as well as the register file [8], With larger tiles comes more operand reuse, and consequently fewer trips to more expensive memories like DRAM, improving arithmetic intensity and energy efficiency. The register file, usually the level closest to the execution unit, is critical to last-level reuse and overall utilization of the execution units. Importantly, higher levels of caches may have the benefit of having their latency hidden by lower levels of caches through well timed (pre)fetches, but the register file has no fallback; register spills directly lead to stalls and thus wasted cycles. At 32 architectural registers and no vector registers, the Vortex ISA provides very little wriggle room when it comes to tiling operands in the register file. This is contrast to register files found in NVIDIA and AMD GPUS; for example, CDNA3 details 128 scalar general purpose registers [1], with another 128 reserved for other purposes such as constants.

In our experience, GEMM kernels for Vortex is constrained to a very small innermost block size. When introducing a tightly coupled tensor core similar to that found in NVIDIA's Volta and Ampere generations [6] [7], we had to downsize the block size to 8x8 at FP16, as the tensor cores fetch only from the 32-entry floating point register file. The integer register file also faces significant pressure calculating the addresses from and to which each thread loads and stores: at

multiple tile levels, address generation involves many loop indices at different levels and dimensions.

2.2 Register Bank Conflicts

Vortex was designed from the ground up to target FPGAs, not ASICs. This had led to an FPGA-centric register file design; specifically, the register files were implemented using block RAMs found only on FPGAs, with asynchronous (combinational) reads. Large pieces of memory on ASIC designs use SRAMs instead, where reads are usually synchronous, outputting data once clock cycle after address input. In our previous work, we resorted to using area- and power-inefficient flip-flop arrays to emulate Vortex's BRAM based register file behavior.

Furthermore, Vortex duplicates register data three times for up to three register reads per cycle, emulating a piece of memory with three read ports. This is acceptable in FPGAs, as BRAMs are abundant distributed across the die. On an ASIC design, however, duplicating data has significant area and power implications. Two read port SRAMs come at a steep cost, added only by the inclusion of a write port.

If naively storing all registers in a single SRAM, the ensuing structural hazard stalls the entire pipeline, requiring two or more cycles to collect operands for many instructions. The realistic solution is splitting each register into a few SRAM banks, providing parallel access to a certain degree. However, without proper compiler optimization, the full benefits from banking registers cannot be realized.

3 Implementation

3.1 64-bit Instruction Word ISA Design

RISC-V, as an ISA designed for CPUs, deliberately chose a short and mostly fixed width instruction width to simplify decoding logic and minimize instruction cache footprint [10]. The rationale no longer holds in GPUs. A single GPU warp contains 32 threads in NVIDIA GPUs, all of which share the same instruction stream and thus largely amortizes the cost of higher instruction fetch traffic. Oftentimes, different warps co-occupying the same hardware datapaths execute the same kernel roughly in sync, which also lowers instruction cache capacity demands. While NVIDIA's GPU ISA is kept opaque, AMD's CDNA3 does also use 64-bit instructions, justifying our decision [1].

After deliberation, we have settled on an ISA design that keeps assembly compatiblity with Vortex, but with a wider instruction word at 64 bit to provide some desirabale features: * 8-bit register fields allow up to 256 registers, from the previous 32; * 2 extra bits of opcode in addition to the existing 7; * Up to 4 source operands and 1 destination operand in a single instruction; * 32-bit immediate for I, S and SB type instructions; * 4 reserved bits for a predicate register.

We show the instruction encodings in Figure **??**.

3.2 LLVM Backend Modifications

We modified the RISC-V 32-bit instruction generation logic directly, instead of adding on top of it, since we do not mix 32-bit and 64-bit instruction words. Some notable implementation details:

- We define the new instruction formats and encodings in TableGen, replacing existing ones. New formats are added, including R-types with more than 3 source operands, as well as I-types with two source operands and a 24-bit immediate; for those with additional operands, we add new DAG patterns. We do not implement new instructions under this format, but we added and tested inline assembly directives for them. We modify existing defined instructions to use the appropriate new format.
- We amend the assembly parser and assembly backend, adding support for new immediate types, and remove logic such as implicit zeros in the original pc-relative instructions. We patch the fixup process to generate the correct bits to insert into the instruction.
- We extend the number of general purpose registers to 128; of the new 96, there are 16 additional `a` registers, 32 additional `t` registers, and 48 `s` registers. We extend the number of floating point registers to 64; duplicating each type (8 `fa`, 12 `fs`, 12 `ft` registers). Eventually, we hope to unify scalar and floating point registers through the RISC-V `zfinx` extension.
- We adjust the calling convention to add new callee-saved `s` registers.
- We change the immediate generation logic so long immediates loads into registers no longer require the two-instruction pair `LUI+ADDI`. We change the address load logic, which generates `AUIPC+JALR`, fixing `AUIPC` to always generate a offset of 0, and delegating all offset to the 32-bit immediate in `JALR`.
- We add the additional register names and aliases to `clang`, required for handling inline register specifications like `register int x asm ("t32")`.
- We reflect the encoding changes in the RISC-V target-specific code in the `lld` linker, which handles address relocations. In particular, `lld` requires custom handling of immediate insertion into the instructions where a relocated label is present and thus left blank by `clang`.

3.3 Register Bank Conflict Aware Allocation

We modified the register allocation backend of LLVM to take into account register bank conflicts while deciding to assign a register.

LLVM contains three register allocation backends: Base, Linear Scan, and Greedy. Base is a toy allocator. Linear scan and greedy are used in practice, with linear scan being used with the `-O1` flag while greedy is used for all higher levels.

```
R-type:
63  60   59   58    52 51 44 43 36 35 28 27 20 19    17 16 9 8     7 6      0
[pred] [resv] [funct7] [rs4] [rs3] [rs2] [rs1] [funct3] [rd] [opext] [opcode]
  4b     1b      7b      8b    8b    8b    8b     3b     8b     2b      7b

I2-type:
63  60 59          36 35          28 27 20 19    17 16 9 8     7 6      0
[pred] [ imm[23:0] ] [ imm[31:24] ] [rs1] [funct3] [rd] [opext] [opcode]
  4b        24b            8b          8b     3b     8b     2b      7b

S/SB-type:
63  60 59          36 35 28 27 20 19    17 16           9 8     7 6      0
[pred] [ imm[23:0] ] [rs2] [rs1] [funct3] [ imm[31:24] ] [opext] [opcode]
  4b        24b        8b    8b     3b          8b          2b      7b
```

Figure 1. Instruction encodings of the 64-bit wide ISA.

The notable differences between linear scan and greedy are that greedy splits intervals, and allows backtracking and reassigning existing assignments in an attempt to find a better coloring.

We augment the greedy register allocator. The greedy allocator heuristically attempts to solve the graph coloring problem by using the interference graph. The interference graph contains edges between virtual registers that have overlapping live ranges. We create a new graph, the bank conflict graph, by adding edges between virtual registers that are used as operands in the same instruction: that is, those that could potentially be read in parallel from non-conflicting register banks in a single cycle. Most importantly, the interference graph is a hard constraint: we cannot assign the same physical register to overlapping virtual registers. However, the bank conflict graph is a soft constraint: it is beneficial to have non-conflict register assignments, however it may not always be possible, and bank conflicting register assignments are functionally correct. Thus, we modify the register allocator to take into account the bank conflict graph, and attempt to find a coloring that minimizes bank conflicts.

Some implementation details are:

- We integrate a new analysis pass that creates the bank conflict graph per machine function by iterating through all machine instructions.
- We modify the greedy allocator's interference checker to query the bank conflict graph and try to assign non-conflicting registers. We prevent the physical register search loop from terminating early until it either finds a conflict free physical register, or exhausts the register space.
- We ensure that special register assignments and register hints are unaffected.
- We implement a new register abstraction to group registers together as a bank. This is internally represented as a class callled `RegisterStripe`. For evaluation purposes, we have defined four register banks: GPRS0 through 3 and FPRS0 through 3, simply grouping every 4th GP/FP register into a bank.
- We modify a tablegen parser to generate target specific code that uses our new `RegisterStripe` class.

4 Evaluation

4.1 Methodology

We wrote two common integer linear algebra kernels: matrix multiplication and vector add. We compiled them with our compiler to baremetal code, without using the C runtime or Vortex runtime. They both target a hardware configuration with 4 warps and 4 lanes, for a total of 16 threads.

Our vecadd implementation is fairly straightforward; two statically-allocated arrays each of length 2048 are added element-wise in a loop, with the result stored to another static location. At the start of the kernel, we spawn 16 threads, which all work on contiguous sections; each thread therefore has a data stride of 16 words. We unroll the loop by 16 using a compiler pragma directive.

Our gemm implementation is slightly more complicated. We tile the output matrix on the i and j dimension, computing a single 4×4 block at once. For each k, an outer product is computed; they are summed across the k dimensions in register and written back in the end. Different threads are assigned different blocks. This is not the most optimal way of GEMM on a SIMT architecture, but we believe it suffices for evaluation purposes, as the kernel gains data reuse from a large inner tile. The size of the problem is set to $64 \times 64 \times 64$.

We enabled the `-O2` optimization level to use the greedy register allocation algorithm, which our pass is added in; using `O2` also avoids storing variables on the stack if possible.

In terms of register banking, we opted for a 4-bank design, with each register belonging to their register number modulo 4. Since the different types of registers mostly occupy contiguous register numbers, this approach distributes the different types to different banks equally, which we believe

Kernel	Baseline	Bank-unaware	Bank-aware
vecadd	1	0	0
gemm	12	18	4

Table 1. Static analysis for register bank conflicts.

is a sensible scheme. Since our approach can adapt to any register bank assignment, this decision is done without loss of generality.

We evaluate the compiled kernels with three different compilers:

- **Baseline**: equivalent to the original Vortex compiler, with 32 total registers;
- **Bank-unaware**: enlarged 128-entry register file, but without bank-aware register allocation;
- textbfBank-aware: includes both of our optimizations.

4.2 Static Analysis

In static analysis, we examine the compiler assembly output when compiling these two kernels. For all instructions that utilize more than one register, We check if the source operands belong to the same register bank; if so, we register an instance of bank conflict. Our kernels do not feature instructions that use more than two operands. Where the destination register lives does not matter, since in hardware the bank write port will likely be separate. This approach is somewhat limited, since examining the assembly direclty means loops and conditionals are not evaluated. In Table 1, we show the bank conflict results:

For `vecadd`, the results are rather uninformative, since the register pressure was small to begin with; there is no regression of our algorithm in this case. For `gemm`, we observe an uptick when we enlarged the register file; this is expected since the reduced stack spill resulted in more register usage, which caused more bank conflicts. With bank-aware allocation, the number of conflicts dropped down to just 4.

4.3 Emulation Results

For a more comprehensive and accurate test, we managed to cobble together a preliminary ISA emulator written entirely in Rust named Cyclotron. This is intended to be a functional model for our future GPU core design implementing this ISA.

Cyclotron fetches directly from a compiled ELF file, and is currently capable of executing the base I and M extension instructions, along with the vortex thread mask and warp spawn intrinsics. It is also able to fetch from and write to basic CSR registers, such as the warp ID or global thread ID. Intra warp branching support (pushing and popping the IPDOM stack) is currently unimplemented, but they are not featured in our two simple kernels. Cyclotron instantiates a maximum of 16 threads in 4 warps, all of which share a perfect global memory backed by the ELF. At the start of the

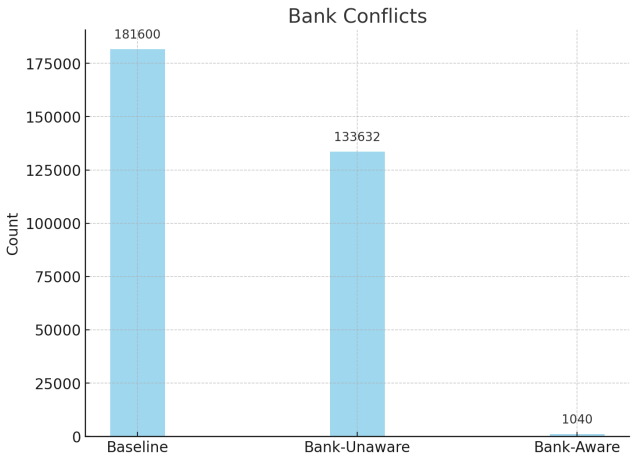

Figure 2. Bank conflicts.

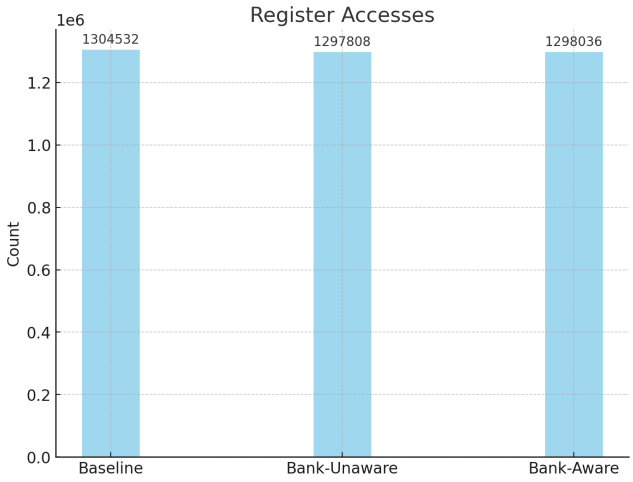

Figure 3. Register accesses.

execution, a warp spawn instruction in the kernel spawns all 4 warps required to compute the GEMM.

We added hooks to the ALU so that all necessary register reads are logged. Checks in the global memory read and write routines are added to log accesses to memory addresses belonging to the stack. Figure 3 shows the amount of register accesses for context, followed by the number of bank conflicts shown in Figure 2.

The number of register accesess was reduced by under 1% when introducing new registers. However, the number of bank conflicts went down by 26.4% in the process as well, even without the bank aware allocation pass. This is likely due to the baseline needing to frequently access stack, which oftentimes uses fixed registers like `sp`, effectively eliminating one of the four banks as a viable scratch register allocation option. With the added pass however, the number of conflicts drops dramatically by 99.2%.

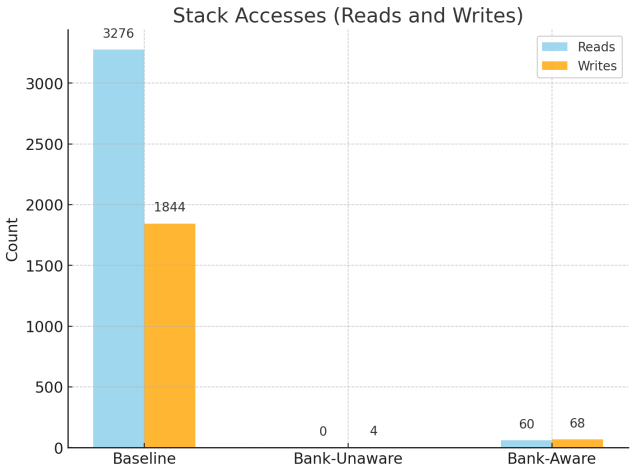

Figure 4. Stack accesses.

For stack accesses, we count separate the number of reads and writes, shown in Figure 4.

With the added registers, spilling has been completely eliminated. The only accesses for the bank-unaware compiler are the ones at the start and end saving and restore the return address register; there's a small amount of inaccuracy towards the end of the emulation due to the emulation terminating when the warp scheduler sees the first warp returning from main, so the stack reads to restore `ra` was not counted (this would add 4 loads).

For the bank aware case, there is a slight uptick in stack accesses. This is due to the compiler exhausting all `a` and `t` registers that don't cause bank conflicts, and resorting to `s` registers, which are callee saved in RISC-V callling convention. The tradeoff of whether some stack acceses for less register bank conflicts here is unclear and warrants future investigation. Whether it's beneficial to increase the number of temporary registers and decrease the number of saved registers will also require investigation using a more complicated GPU kernel. Nevertheless, this situation does not involve stack spills either, as the stack accesses are not caused by running out of allocatable registers.

5 Future Work

We plan to continue work on this compiler stack to support our overall research goal of developing a new high-performance GPU SoC design. In the near term, there are some clear goals we want to resolve:

- We need to compile a C runtime library, be it GNU or LLVM, into our new ISA. This will allow us to write kernels that uses routines like `memset`, and also allow us to compile the Vortex runtime.
- The relocation code in `lld` is missing the optimization pass for merging the `AUIPC/JALR` pair. This was needed in base RISC-V since `AUIPC` provides the upper 20 bits of the PC-relative jump, while `JALR` provides the lower 12; in our LLVM, the immediate in `AUIPC` is set to 0 as a temporary fix. The pair can be replaced with a single `JAL` instruction, now supporting the full 32-bit offset.
- The emulator is still missing many features, such as branch divergence support and floating point operations. We also wish to add timing annotations for more accurate simulations.
- As previously mentioned, we need to investigate the optimal number of temporary and saved registers. We need quantitative data to decide in cases where we face the decision of whether to allocate a bank-conflicting unsaved register or a non-conflicting saved register.
- We need to merge the general purpose register file and the floating point register file, through the `zfinx` extension. This simplifies the architecture, and is more straightforward when implementing a single unified physical register file in hardware.

6 Conclusion

In this report, we highlighted performance challenges we faced when using the RISC-V-based Vortex ISA, namely register spills and bank conflicts. We proposed a wider instruction set to provide more architectural registers, and a bank conflict aware register allocation pass to solve these challenges. We implemented both proposals in LLVM, modifying the RISC-V backend. We evaluated the resultant compiler with two integer kernels by static analyzing the assembly, as well as by collecting statistics from running an emulation on an ISA emulator that we wrote in Rust. The results show that register conflicts have been reduced by more than 99%, and that register spills to stack have been eliminated. We aim to continue working on this compiler as part of our GPU building effort.

References

[1] Advanced Micro Devices, Inc. 2024. *"AMD Instinct MI300" Instruction Set Architecture.* Technical Report. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf

[2] Caroline Collange. 2017. Simty : generalized SIMT execution on RISC-V Sylvain Collange Inria sylvain. https://api.semanticscholar.org/CorpusID:27721335

[3] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[4] Hansung Kim, Ruohan Yan, Joshua You, Tieliang Vamber Yang, and Yakun Sophia Shao. 2024. Virgo: Cluster-level Matrix Unit Integration in GPUs for Scalability and Energy Efficiency. arXiv:2408.12073 [cs.AR] https://arxiv.org/abs/2408.12073

[5] NVIDIA. 2023. Demystifying AI Inference Deployments for Trillion-Parameter Large Language Models. https://developer.nvidia.com/blog/demystifying-ai-inference-

deployments-for-trillion-parameter-large-language-models/ Accessed: 2024-12-17.

[6] NVIDIA Corporation. 2017. *NVIDIA Tesla V100 GPU Architecture*. Technical Report. NVIDIA Corporation. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[7] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report. NVIDIA Corporation. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[8] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. https://github.com/NVIDIA/cutlass

[9] Blaise Tine, Fares Elsabbagh, Krishna Praveen Yalamarthy, and Hyesoon Kim. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-GraphicsResearch. *CoRR* abs/2110.10857 (2021). arXiv:2110.10857 https://arxiv.org/abs/2110.10857

[10] Andrew Waterman. 2016. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. *University of California, Berkeley* (2016). https://people.eecs.berkeley.edu/~krste/papers/waterman-ms.pdf