Understanding GPGPU Vector Register File Usage

Mark Wyse[*]
wysem@cs.washington.edu
AMD Research, Advanced Micro Devices, Inc.
Paul G. Allen School of Computer Science & Engineering, University of Washington

ABSTRACT

Graphics processing units (GPUs) have emerged as a favored compute accelerator for workstations, servers, and supercomputers. At their core, GPUs are massively-multithreaded compute engines, capable of concurrently supporting over one hundred thousand active threads. Supporting this many threads requires storing context for every thread on-chip, and results in large vector register files consuming a significant amount of die area and power. Thus, it is imperative that the vast number of registers are used effectively, efficiently, and to maximal benefit.

This work evaluates the usage of the vector register file in a modern GPGPU architecture. We confirm the results of prior studies, showing vector registers are reused in small windows by few consumers and that vector registers are a key limiter of workgroup dispatch. We then evaluate the effectiveness of previously proposed techniques at reusing register values and hiding bank access conflict penalties. Lastly, we study the performance impact of introducing additional vector registers and show that additional parallelism is not always beneficial, somewhat counter-intuitive to the "more threads, better throughput" view of GPGPU acceleration.

1. INTRODUCTION

Contemporary graphics processing units (GPUs) are incredibly powerful data-parallel compute accelerators. Originally designed exclusively for graphics workloads, GPUs have evolved into programmable, general-purpose compute devices. GPUs are now used to solve some of the most computationally demanding problems, in areas ranging from molecular dynamics to machine intelligence. The rapid adoption of GPUs into general-purpose computing has given rise to a new term describing these devices and use: General-Purpose GPU (GPGPU) computing. In this context, GPUs are no longer bound to their traditional domain of graphics, but they are commonly viewed as the workhorse for computationally intense applications.

As the use of GPUs has expanded, the architecture of GPGPU devices has evolved. GPGPUs are massively-multithreaded devices, concurrently operating on tens to hundreds of thousands of threads. Unlike CPUs, which target low-latency computation, GPUs excel at high throughput computation. Achieving high throughput requires supporting many threads, each requiring on-chip context. This context typically includes shared memory space, program counters, synchronization resources, and private storage registers. Maintaining context on-chip enables multithreading among the thousands of active threads, with single-cycle context switching between groups of threads. However, the required context consumes millions of bytes, orders of magnitude more than the context of the few threads present in a traditional CPU. The vector register file storage space alone is typically larger than the L1 data caches and consumes as much as 16 MB in a state-of-the-art, fully configured AMD Radeon™ RX "VEGA" GPU [8][9]. With a considerable amount of storage, die area, and energy being consumed by the vector register files, it is important to understand the use of this structure in GPGPU applications so that it may be optimized for performance and/or energy-efficiency.

This paper examines modern GPGPU architectures, focusing on their use of vector general-purpose registers and the vector register subsystem architecture. Our study consists of three main parts. First, we replicate experiments from prior work revealing the vector register usage patterns for a set of compute applications. We confirm the results of prior work, despite modeling a GPGPU architecture based on products from a different device vendor. Second, we evaluate the effectiveness of operand buffering and register file caching as proposed in prior work. Our experiments show these structures to be highly effective at hiding bank access conflict penalties and enabling vector register value reuse. Third, we examine the potential parallelism and occupancy benefit of a GPGPU architecture providing (physically or logically) twice the number of vector general-purpose registers. We show that the benefit of higher wave-level parallelism and device

[*] This work was completed while the author was a Post-Grad Scholar at AMD Research in Bellevue, WA

occupancy is application dependent. For many developers this notion remains counter-intuitive.

The remainder of the paper is organized as follows. Section 2 provides background on GPGPU architecture and execution. Section 3 describes our analysis and simulation methodology. Sections 4, 5, 6, and 7 detail our experimental results. Section 8 covers related work, Section 9 provides thoughts on future research directions, and we conclude in Section 10.

2. BACKGROUND

GPUs are massively-multithreaded processing devices that support over one hundred thousand active threads. Supporting this many active threads requires an architecture that is modular and compartmentalized, as well as a programming model to express data-parallel computation. This section details the GPGPU programming model, describes the hardware execution model, and details the specific GPU architecture used in this study.

2.1 GPGPU Programming Model

GPGPUs use a data-parallel, streaming computation programming model. In this model, a program, or kernel, is executed by a collection of work-items (threads). The programming model typically uses the single instruction, multiple thread (SIMT) execution model. Work-items within a kernel are subdivided into workgroups by the programmer, which are further subdivided into wavefronts by hardware. The work-items within a wavefront are logically executed in lock-step. All work-items within a workgroup may perform synchronization operations with one another. AMD's GCN architecture [2] also includes scalar instructions that are executed on the scalar ALU. These scalar instructions are generated by the compiler, transparent to the programmer, and are intermixed with vector instructions in the instruction stream. Scalar instructions are used for control flow or operations that produce a single result shared by all work-items in a wavefront.

2.2 GPGPU Hardware Execution Model

Modern GPU architectures execute kernels using a SIMD (Single Instruction, Multiple Data) hardware model. As mentioned above, a kernel is composed of many work-items that are collected into workgroups. The workgroup is the unit of dispatch to the Compute Units (CUs), which is the hardware unit responsible for executing workgroups. A CU must be able to support at least one full-sized workgroup, but may be able to execute additional workgroups concurrently if hardware resources allow. All work-items from the same workgroup are executed on the same CU. A GPU device contains at least one CU, but it may contain more

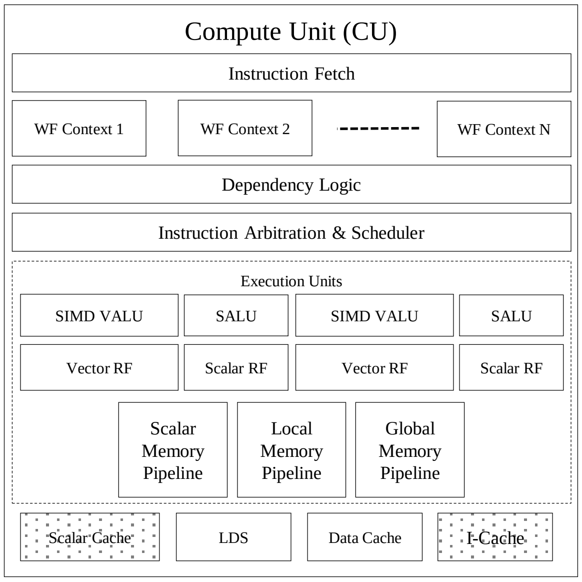

Figure 1. Sample CU Architecture.

to facilitate execution of many workgroups concurrently.

Within a CU, the SIMD unit is the hardware component responsible for executing wavefronts. Each wavefront within a workgroup is assigned to a single SIMD within the CU the workgroup is dispatched to. The SIMD unit is responsible for executing all work-items in a wavefront in lock-step. Each SIMD has access to a scalar ALU (SALU), a branch and message unit, and memory pipelines.

The wavefront size is a hardware parameter that may change across architecture generations or between devices capable of executing the same Instruction Set Architecture (ISA) generation. Programmers should not rely on the wavefront size remaining constant across hardware generations and should not have dependencies on a specific wavefront size in their code.

2.3 Baseline GPGPU Architecture

In this section we detail the CU architecture employed in our study. Figure 1 depicts the architecture of the CU we model, which is capable of executing AMD's GCN3 ISA [3]. Without loss of generality, we elect to use AMD's terminology where applicable.

The CU used in our study contains two SIMD Vector ALUs (VALUs), two Scalar ALUs (SALUs), Vector Register Files (VRFs), Scalar Register Files (SRFs), a Local Data Share (LDS), forty wavefront slots, Local Memory (LM), Global Memory (GM), and Scalar Memory (ScM) pipelines, and the CU is connected to scalar, data, and instruction caches. The following subsections detail the main blocks within the CU. Note that the Scalar Cache and I-Cache are shared between

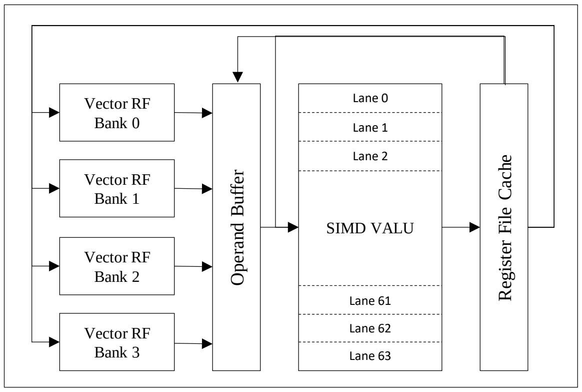

Figure 2. Vector Register File Subsystem Architecture.

multiple CUs, while all other blocks are private per CU.

2.3.1 Wavefront Context

Each CU contains a total of forty wavefront context slots [2]. The wavefront slots are divided equally among the SIMD VALUs, and all instructions from a wavefront are executed by the same SIMD/SALU pair for the duration of the wavefront's life. The wavefront context consists of the program counter, register state information, synchronization and memory counters, and an instruction buffer.

2.3.2 SIMD VALU

Each SIMD within the CU is a sixty-four wide Vector ALU (VALU), capable of issuing for execution one sixty-four wide vector instruction per cycle.

2.3.3 Vector Register File Subsystem

The Vector Register File (VRF) subsystem consists of banked vector register files, containing 1024 64-wide by 32-bit Vector General-Purpose Registers (VGPRs) [3], Operand Buffers (OB) [12][15], and register file caches (RFC) [12]. There is a private VRF, OB, and RFC per SIMD VALU. Figure 2 depicts the various components and operand delivery paths in the VRF subsystem.

2.3.3.1 Banked VRF

The vector register file associated with each SIMD unit contains 128 KB of storage. A CU with two VALUs and two VRFs contains 256 KB of VGPR storage [2]. The VRF comprises multiple SRAM-based banks. Each bank has one read port and one write port, and both a read and a write may occur in the same cycle. In this study, we configure the VRF to have four banks, with each bank holding 128 VGPRs of 64 by 32-bit values. There are 512 VGPRs distributed across the four banks per VRF, with a total of 1024 VGPRs per CU [3]. The bank width matches the wavefront size to facilitate reading and writing an entire VGPR per cycle per bank.

2.3.3.2 Operand Buffer

The Operand Buffer (OB) [12][15] is responsible for reading the vector source operands of each VALU instruction. The primary purpose of the OB is to hide bank access conflict latency penalties. It is a FIFO queue, and instructions enter and leave the OB in-order. However, the OB may read source operands for any instruction present in the FIFO in any cycle (i.e., out-of-order with respect to the execution order). In this study, an oldest-first-then-greedy policy is used to read source operands, but this may be changed in future implementations. The OB attempts to read the operands of the oldest instruction first, but will greedily read operands for younger instructions to avoid bank conflicts or if there are banks with available read ports that contain operands for younger instructions. Source operands are read from the VRF unless the operand exists in the Register File Cache or will be produced by an instruction in the VALU pipeline. Reading all operands for an instruction may take multiple cycles due to bank conflicts. Bank conflicts may occur both within (intra-instruction) and between (inter-instruction) instructions.

2.3.3.3 Register File Cache

The register file cache (RFC) used in this work is inspired by the RFC proposed by Gebhart et al. [12]. The RFC sits between the VALU and the VRF banks. It receives results from the VALU pipeline, forwards those results to the OB and VALU pipeline for future instructions if needed, and lazily writes back results to the VRF.

The RFC holds data for one or more VGPR sized entries. Each entry is one complete 64-wide by 32-bit VGPR. The RFC is an on-demand allocation and eviction cache, with strict LRU eviction and replacement. The RFC's primary purposes are: (a) forwarding source operands to the OB and VALU pipeline, thereby reducing the number of VRF reads, and (b) to hide the latency penalty of bank write access conflicts.

The RFC can provide up to three VGPRs of 64 32-bit values to the instruction being dispatched from the OB to the VALU pipeline over forwarding paths. This path operates similar to bypass paths in traditional computational pipelines, and allows source operands to be delivered directly to the VALU pipeline without waiting for the values to be read and written from the VRF, saving access time and energy.

Operands may also be forwarded from the RFC to the OB as the OB attempts to read source operands for instructions. This path is activated when an existing RFC entry is evicted, and may further reduce source operand reads performed from the VRF. The evicted RFC entry is provided to every instruction in the OB that needs it.

2.3.4 Scalar ALU and Scalar Register File

AMD's GCN architecture includes a Scalar ALU (SALU) to handle execution of scalar instructions. Unlike vector instructions that operate on each individual work-item in a wavefront, scalar instructions are executed once for all work-items in a wavefront. The primary purpose of scalar instructions is to handle control flow and perform thread independent computation for the wavefront.

Our CU model includes two Scalar ALUs, with each SALU being associated to one of the VALUs. Each SALU has a private Scalar Register File (SRF) containing 800 32-bit Scalar General-Purpose Registers (SGPRs) [3][8]. The SGPRs are assigned at dispatch time to wavefronts being executed by the VALU/SALU pair.

2.3.5 Memory Subsystem

The memory subsystem used in the baseline architecture in this paper is modeled after the GCN device architecture [2][3][8][9]. In this setup, a CU contains a private L1 vector data cache and Local Data Share (LDS) scratchpad memory. A CU shares a scalar data cache and instruction cache with a collection of other CUs in the system. All three caches (vector, scalar, instruction) are supported by a shared L2 cache, which in turn connects to main memory.

The vector L1 data cache is a 16 KB, 16-way set associative, 64-byte cache block SRAM cache [2]. The shared instruction cache is a 32 KB, 8-way set associative, 64-byte cache block SRAM cache [2]. The L2 cache is a 512 KB, 16-way set associative, 64-byte cache block SRAM cache [17]. The L2 cache unifies the scalar data, vector data, and instruction caches, and is connected to system memory.

Each CU also contains a 64KB Local Data Share (LDS). The LDS is a software managed cache, with 32 banks [3][8][9]. This structure provides a high-bandwidth, low-latency, software managed memory and acts as a data cache bandwidth amplifier.

3. METHODOLOGY

This section describes the benchmark analysis and simulation methodologies used for the experiments presented.

3.1 Benchmark and Kernel Analysis

In this subsection we detail the static and dynamic analysis performed to assess register usage and dependency characteristics.

3.1.1 Kernel Analysis

To evaluate dispatch limits for the benchmarks under study, we rely on data produced by the compiler and disassembly tools for AMD's GCN3 ISA [3][4]. These tools provide the number of vector and scalar general-purpose registers required per work-item and wavefront, respectively. Simulation (methodology below) provides the number of workgroups executed per kernel dispatch. Combining these data with architectural parameters of our system, we are able to determine the resources that limit kernel and workgroup dispatch and evaluate dispatch limits as architectural parameters are varied.

3.1.2 Dynamic Register Profiling

We use the gem5 simulator [14], which includes a modified version of AMD's APU gem5 model [7] (details below) to collect register reuse and producer-consumer data. Simple implementation of the register file system is sufficient to provide both the number of consumers per value producer and the distance between producer and consumer for all vector register values.

3.2 The gem5 Simulator

The gem5 simulator is an execution-driven, cycle-level microarchitecture simulator that is capable of executing real ISAs on simulated microarchitectures. AMD's recent APU extension has added support for GPU Compute Units within the simulation framework. The APU model is compatible with gem5's system call emulation (SE) mode, where system calls invoked by simulated applications are either emulated in the simulator or passed to the host for execution. In this study, we use an updated version of the AMD APU compute model that faithfully implements the GCN3 ISA and runs an unmodified, publicly-released ROCm [6] version 1.1 software stack, with only kernel driver functionality being emulated. We simulate an APU with one CPU and a single CU to stress the CU and VRF to the greatest extent.

The following subsections detail the instruction readiness, dispatch, and execution flow in our Compute Unit implementation. The remaining structures (VALU, SALU, VRF, SRF, etc.) are implemented faithfully to the descriptions provided in Section 2.3 above.

3.2.1 Wavefront and Instruction Readiness

As described in the GCN3 architecture, each SIMD VALU has many associated wavefronts it is responsible for executing, with our simulated architecture supporting twenty wavefronts per SIMD. Every cycle, each SIMD evaluates all wavefronts for readiness. A wavefront is deemed ready to execute if it is active, not waiting for a synchronization operation to complete, has at least one instruction to execute, and all true (RAW) register dependencies are resolved. Register dependencies are tracked using a scoreboard indicating which, if any, source operands have not been produced yet by the various functional units (VALU, SALU, memory pipelines) and are busy. Each wavefront that is ready is presented to the instruction dispatch unit as a candidate for execution.

3.2.2 Instruction Dispatch and Execution

After wavefronts are checked for readiness, the instruction dispatch unit selects and attempts to schedule for execution up to one wave per execution resource. The execution resources are the VALUs, SALUs, and memory pipelines. Each cycle, a scheduler selects a candidate wave for each resource, typically using an oldest-first policy.

To be dispatched for execution, a selected wave must first gather all source operands from the register files. Each non-scalar instruction is sent to the vector register file, with VALU operations requesting a slot in the Operand Buffer, and vector memory (VMEM) instructions requesting access to the VRF banks. VMEM instructions receive priority for reading VRF banks. Once operands are read from the register files, the appropriate execution resources are checked for readiness. An execution resource may disallow instruction issue due to certain conditions, such as issue period limitations or full buffers (e.g., for vector memory coalescing).

After all source operands and execution resources are ready, an instruction is deemed ready for execution. At this point all non-vector ALU operations will be issued for execution. VALU operations must make one final request to the register file cache to allocate slots for destination registers. If the RFC is unable to allocate slots for the instruction, VALU instruction issue will stall. Once the RFC accepts the destination slot allocation request, the VALU operation will be issued to the pipeline for execution.

At the end of instruction execution, the destination values will be written into the RFC and the scoreboard updated to indicate result data are available for use. Register file write-back operations occur lazily from the RFC. Memory loads are enqueued in the memory pipelines and return data in variable latencies depending on memory system behavior and contention. Loads update the scoreboard and write-back results to the VRF once data return from the memory system.

3.3 Benchmarks

Table 1 lists the benchmarks used in this study. These applications are obtained from the AMD compute applications GitHub [1] [5]. The applications used in this study represent common kernels from HPC and scientific computing workloads that are of interest in the GPGPU community.

The selected applications are written using the heterogeneous compute (HC) C++ API. Source code is compiled using the heterogeneous compute compiler (HCC) [4], which is based on Clang and LLVM. HCC is an open source compiler for heterogeneous compute applications that target the ROCm stack.

Benchmark	**Description**
Array-BW	Memory streaming
Bitonic Sort	Parallel Merge Sort
CoMD	DOE Molecular-dynamics algorithms
FFT	Digital signal processing
HPGMG	Ranks HPC systems
MD	Generic Molecular-dynamics algorithms
SNAP	Discrete ordinates neutral particle transport application
SpMV	Sparse matrix-vector multiplication
XSBench	Monte Carlo particle transport simulation

Table 1. Description of evaluated workloads.

4. VECTOR REGISTER USAGE

Prior works [10] [12] have examined the usage of vector register values in GPGPU architectures and concluded that most values produced are consumed a small number of times within a small instruction window from the producer instruction, and many registers do not contain live values for significant portions of execution.

The authors of [12] claim up to 70% of values are read only once, and only around 10% of values are read more than 2 times. This prior study evaluates an architecture modeled after those from NVIDIA, thus it

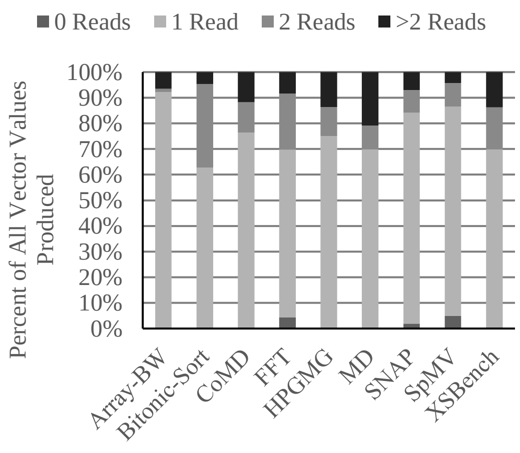

Figure 3. Number of reads per vector register value.

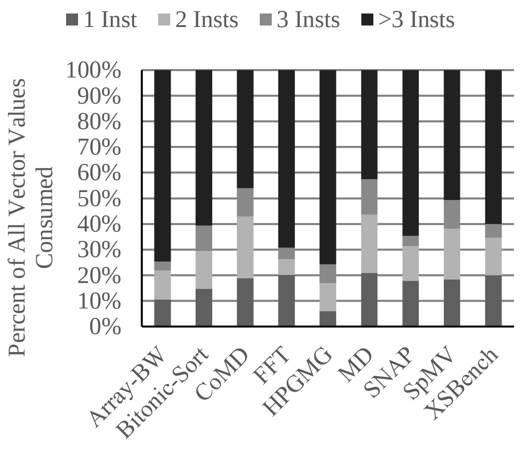

Figure 4. Lifetime of vector register values.

is worth asking the question: do the same patterns hold for AMD GPUs and GCN3 ISA code?

We replicate two of their studies and find that 60-90% (75% average) of vector values produced are read exactly once, and 4-13% (10% average) of vector values produced are read more than twice, as shown in Figure 3. Further, we find 24-57% (40% average) of all values consumed were produced within 3 instructions prior, as shown in Figure 4. In both experiments, our results are in line with prior work. While not overly surprising, this adds confidence to the remainder of our evaluation and experiments, and confirms that the codes being used for evaluation exhibit similar behavior despite differences in implementation, compilers, and ISA.

Prior studies also examine the liveness of register values over the course of execution [10] [11]. The authors' conclusions are that many registers are short-lived and thus contain no live values for a significant percentage of a kernel's execution. We replicated their results (data not shown) and confirm that the applications used in our study exhibit similar behavior. No application utilized all of its compiler-allocated registers, and all applications had register usage patterns with significant variation in the number of live register values throughout execution.

5. Reducing the Number of Register File Reads

One function of the OB and RFC is to reduce the number of reads from the vector register file. As described above, the RFC is able to both recycle operands to the OB and forward source operands to the VALU pipeline at instruction dispatch. Each of these paths reduces the number of VGPR reads performed from the main register file by the OB. In this experiment, we examine the number of reads required by the OB for VALU instructions that can be saved by resizing the RFC, and discuss the performance and implementation implications of such changes.

Figure 5 shows the number of reads saved for each RFC configuration. We sweep RFC sizes from 2 through 512 total entries, with the y-axis showing the percentage of vector sources that are provided by the RFC to the OB, or equivalently, the number of VRF reads saved (higher is better) for VALU instructions. Figure 7 shows the relative performance for each RFC configuration. The y-axis is IPC normalized to an RFC with eight entries (higher is better).

At small RFC sizes (2 or 4 entries), we observe that up to 25% of all possible reads required by the OB from the VRF are avoided. At these sizes, we observe performance degradation, caused by the timing of RFC entry allocation in our simulator implementation. An RFC slot is allocated when an instruction is dispatched to the VALU pipeline, and because the pipeline latency is larger than the number of RFC slots for small sizes, instruction issue stalls on RFC allocation.

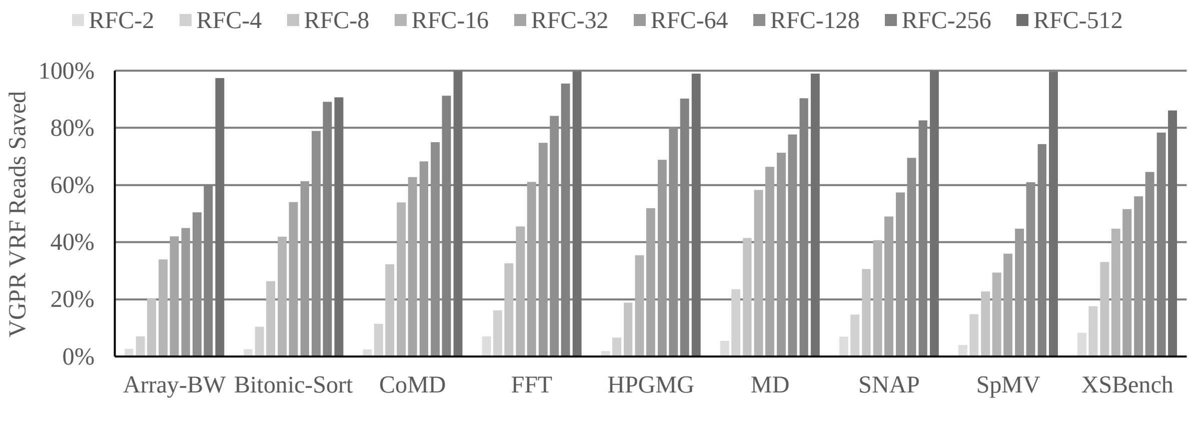

Figure 5. Number of VGPR reads saved by RFC forwarding paths.

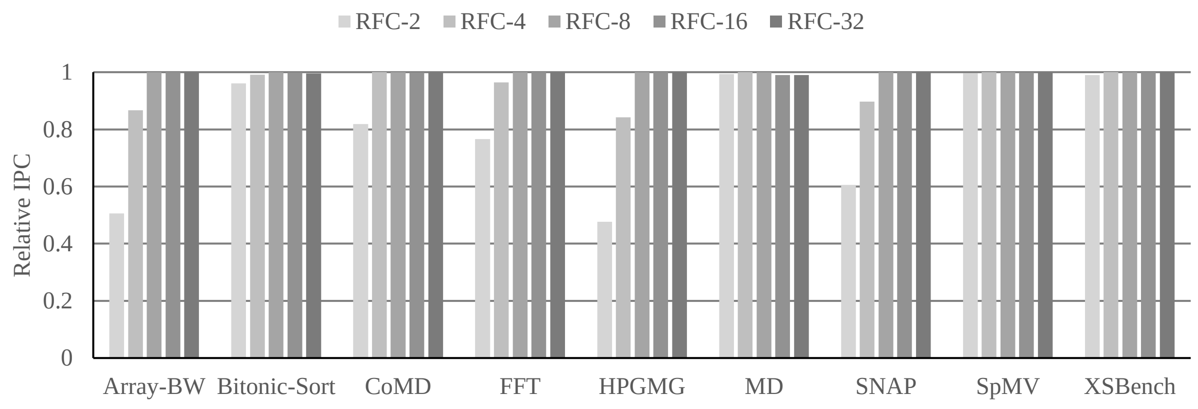

Figure 6. Relative performance (IPC) for RFC sizes. Performance at sizes greater than 32 is stable at a relative IPC of 1.

Pipeline bubbles are introduced, diminishing performance. This behavior is an artifact of the simulator implementation, and correcting it is left as future work.

At larger RFC sizes (16 or more entries), we observe the number of saved reads increases significantly with RFC size. More than half of the benchmarks studied are able to serve more than 80% of all required VGPR operands for VALU instructions from the RFC at the largest RFC configuration. Although the RFC is as large as the VRF at size 512, not all source operands will be captured for reuse by the RFC. Any value loaded by a memory instruction into VGPRs must be read from the VRF instead of the RFC to ensure correctness, until the physical VGPR loaded to is overwritten by a VALU instruction and the result stored in the RFC. Additionally, some VGPR values are initialized by hardware at wavefront dispatch. Until the physical VGPRs holding these values are overwritten by a VALU instruction, if they ever are, those values will be read from the VRF, not the RFC. At larger RFC sizes, no performance benefit is observed. This implies that the RFC and OB are effective at hiding latency penalties from bank access conflicts at default sizes of eight RFC entries and four OB entries (details in Section 6). Although increasing the RFC size does not lead to better performance, it may lead to reduced VRF access energy as more operands can be provided by the lower energy forwarding paths. However, there are trade-off costs in implementation (area, power, and latency) as RFC size increases that may make larger RFCs less energy-efficient.

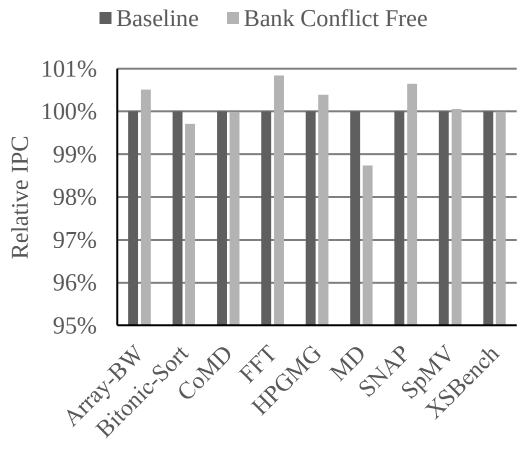

Figure 7. Relative performance comparing baseline architecture to a bank conflict free configuration (Note y-axis begins at 95%).

6. Register Bank Conflicts In a Compute Optimized GPU Architecture

The VGPRs are physically stored in a banked register file to provide high-bandwidth access without the overhead of multi-ported register files. Each bank is the width of a GCN3 wavefront, or 64 32-bit entries, or equivalently, one VGPR wide, and can read and write one VGPR per cycle.

In both the GCN3 and our simulated architecture, both intra- and inter-instruction bank conflicts may occur. Intra-instruction read conflicts occur when two or more source operands in the same instruction reside in the same physical VGPR bank. Inter-instruction read conflicts occur when multiple instructions have operands residing in the same bank and attempt to read them in the same cycle. Given our GPU architecture detailed above, the natural question to ask is: how effective are the RFC and OB at hiding the latency penalties of bank access conflicts?

We answer this question by performing a limit study on the performance benefit of removing bank conflicts. As discussed in the Section 5, the RFC and OB significantly reduce the number of accesses to the

	Max WG/CU by Resource			
Application	**WF**	**VGPR**	**SGPR**	**Limiter**
Array-BW	2	1	12	VGPR
Bitonic-Sort	2	4	12	WF
CoMD	40	28	100	VGPR
FFT	40	12	66	VGPR
HPGMG	2	1	4	VGPR
MD	2	1	6	VGPR
SNAP	40	14	100	VGPR
SpMV	20	21	100	WF
XSBench	10	4	25	VGPR

Table 2. Maximum Workgroups per CU per application when only limited by Wavefront Slots (WF), VGPRs, or SGPRs.

VRF. However, these structures are also meant to hide the latency penalty of bank access conflicts. Specifically, the OB acts as a read buffer to opportunistically read operands when banks become available and prevent the insertion of bubbles into the VALU pipeline. To assess the effectiveness of the OB at hiding conflict penalties, we compare our baseline architecture to one without bank conflicts in the VRF. We reconfigure the simulator to place each VGPR in its own VRF bank. By definition, two different VGPRs will not conflict with one another in this setup.

Figure 6 shows the results of this experiment and displays normalized performance of the baseline and bank conflict free configurations. The y-axis is performance in IPC, normalized to the baseline configuration (higher is better). As shown, there is negligible change in IPC between the two configurations (less than +/- 1% typically), indicating that the RFC and OB are well-suited at handling all bank access conflicts encountered during dynamic execution.

There are a few different conclusions that may be drawn from these results. Perhaps most obvious is that the RFC and OB are effective at hiding the latency penalty of any conflicts that occur. This may be due to the OB's ability to gather operands out-of-order with respect to instruction issue order, or that the RFC's forwarding paths are able to adequately reduce the number of VRF accesses required by the OB. Fewer required accesses means less register file pressure and a lower probability of bank conflicts. Another possible explanation is that the applications studied have lower than expected dynamic register usage. Although most of the applications are limited by VGPRs in dispatch (see Section 7), the dynamic usage may not be as great as the static demand. It is possible that codes in other domains may have greater dynamic register usage.

7. DISPATCH LIMITS AND WAVE-LEVEL PARALLELISM

In this section we examine the resource requirements for workgroup dispatch for each benchmark studied and examine the impacts of increased resource availability in terms of performance and parallelism.

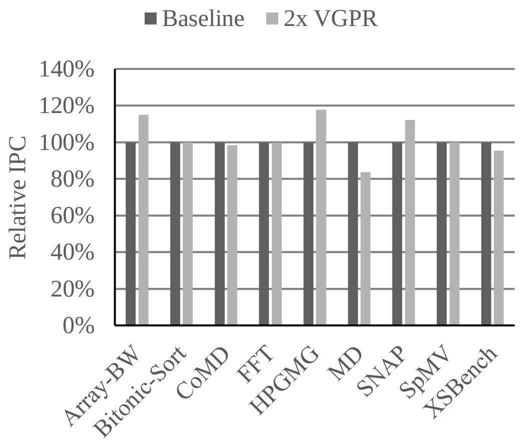

Figure 8. Relative performance with 2x VGPR per SIMD.

7.1 Workgroup Dispatch Limits

Launching, or dispatching, a kernel for execution on a GPU requires a set of resources to be available. These resources are wavefront slots, vector registers, scalar registers, and scratch memory. For our applications, we find that the majority are vector register limited for dispatch. Table 2 shows the results of offline analysis on the compute kernels and lists the maximum number of workgroups that can be dispatched per CU when only considering one resource at a time (other resources assumed infinite). Each application has a single compute kernel, except FFT, which has both a forward and inverse FFT kernel. The FFT kernels, however, have identical resource requirements and are executed sequentially. The three columns give the maximum number of workgroups per CU when limited only by wavefront slots (WF), VGPR availability (VGPR), and SGPR availability (SGPR). The final column (Limiter) lists which resource limits dispatch and prevents further workgroups from being dispatched. Of the nine applications we study, only Bitonic-Sort and SpMV are not VGPR limited for dispatch.

Because many applications are limited by VGPR availability, it is natural to ask: does providing additional VPGRs result in improved wave-level parallelism and/or performance?

7.2 Increasing Wave-Level Parallelism and Performance with Additional VGPRs

General-purpose registers are shown to be the limiting factor in workgroup dispatch for seven of the nine benchmarks in this study. GPGPUs are throughput accelerators, and it is often thought that improving the amount of available work or occupancy will result in

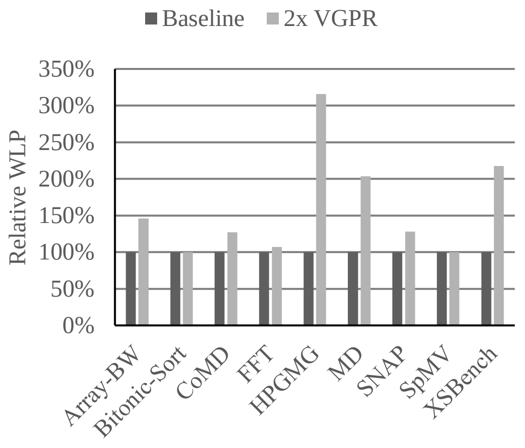

Figure 9. Relative WLP with 2x VGPR per SIMD.

improved performance. However, prior work has shown that additional parallelism may not always be beneficial [16].

In this experiment, we assess the benefits of having additional VGPRs available per SIMD VALU and CU. For the benchmarks studied, do the additional VGPRs (a) allow more workgroups to be dispatched and increase the wave-level parallelism (WLP), and (b) if WLP is increased, is there a resulting performance benefit?

To estimate the best-case improvement, we modify our simulator configuration to have twice the number of VGPRs per SIMD/CU (1024 VGPR per SIMD, 512 KB per CU), but do not modify any timing for VGPR access. Assuming no timing penalty for a larger VRF is optimistic, especially when the register files are as large as those used in GPUs, but it allows us to study the upper bound benefit of additional VGPRs.

Figure 8 and Figure 9 show the results of our experiment. Figure 8 displays relative performance in IPC, for the baseline and 2x VRF configurations, normalized to the baseline. The y-axis is improvement in IPC over the baseline, and higher is better. Figure 9 shows the realized wave-level parallelism (WLP) for the baseline and 2x VRF configurations. The y-axis is the average WLP observed normalized to the baseline configuration, and a value greater than one indicates greater observed WLP. We measure WLP by counting the number of already active wavefronts when each new wavefront is dispatched, then averaging this count from all wavefront dispatches.

Our experimental results are mixed. All seven applications that are VGPR limited for dispatch observe increased average WLP. However, the performance benefit of higher WLP is mixed. CoMD, MD, and XSBench experience performance degradation, FFT sees no performance change, and Array-BW, HPGMG, and SNAP see improved performance.

For the benchmarks with a performance loss, averaging around 7% worse IPC, we suspect memory access divergence is at fault. Prior analysis (data not shown) revealed MD and XSBench have greater memory divergence per memory instruction. These two benchmarks also have long tails in their memory access latency distributions compared to other applications, as shown in Figure 10. We only show a subset of the benchmarks for clarity in the figure; however, the benchmarks not shown have CDF's that fall between those shown for FFT and SpMV. The divergence in memory requests increases the number of post-coalescing memory requests per instruction and appears to cause an increase in average memory access

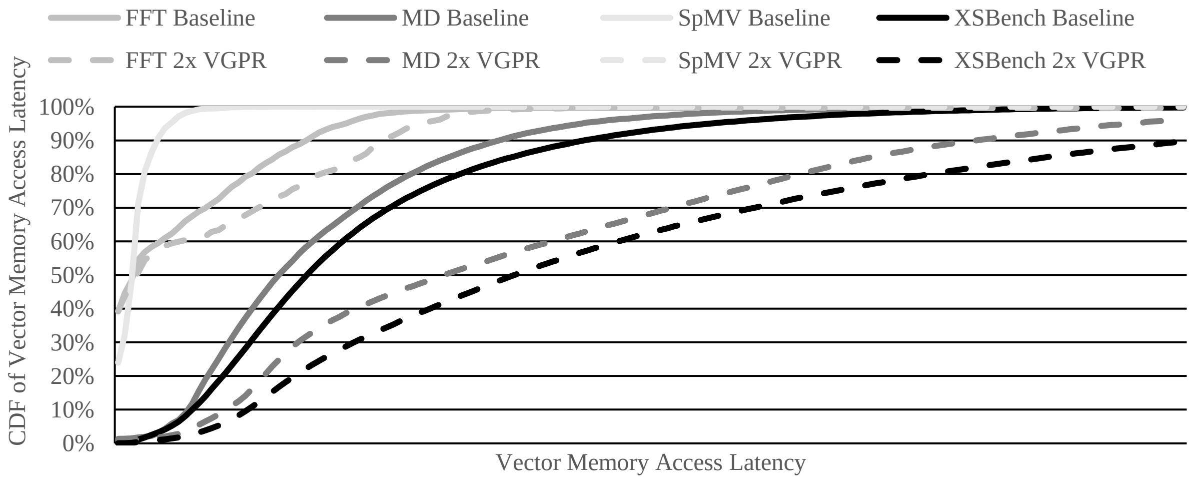

Figure 10. CDF of Vector Memory Access Latency - Selected Benchmarks.

latency. This issue is exacerbated when additional wavefronts are active due to an increased VRF size. The increased memory access time reduces the rate at which the coalescer may issue additional requests to the memory system, and creates backpressure on instruction issue, manifesting as coalescer access stalls when attempting to dispatch instructions. It is possible that a different organization or banking of the memory caches could alleviate some or all of the long tail of memory access latencies. Investigating this is left as future work.

In summary, we show that a subset of applications studied achieve greater WLP when twice the VGPRs are available per SIMD/CU. However, performance improvement is not guaranteed under increased WLP.

8. RELATED WORK

The most similar works, and in fact the works we attempt to replicate studies from, are those of Gebhart et al. [12], Jeon et al. [10], and Yu et al. [11].

Gebhart et al. [12] evaluate the usage of vector registers in terms of the number of consumers per value produced and the value lifetime of certain values. We replicate the former, and perform a similar experiment on vector value lifetime. Gebhart et al. also propose the Register File Cache that we study here. Their system includes an Operand Buffer to gather operands from the register file and feed them to the vector ALU pipeline. We largely confirm the results of the authors' study.

Jeon et al. [10] perform similar analysis for register value lifetime. Noting that register values are short-lived and register files are underutilized due to static waste, the authors propose virtualizing registers to enable an architecture with half the number of physical vector registers, while maintaining performance. In contrast, we examine the potential benefits of providing twice the number of vector registers to increase parallelism and performance.

Yu et al. [11] propose using a register stash to remove the vector register dispatch limit on GPUs. Their study also examines the usage patterns of registers and kernel dispatch limits. Their register stash technique claims to solve the underutilization of the register files, enable architectures with smaller register files, and improve performance when maintaining a constant register file size. Counter to their work, we find that providing additional registers to increase parallelism does not always improve performance.

The primary difference between our work and prior work is our use of a hardware ISA (GCN3) simulator instead of a virtual ISA (PTX) simulator. AMD's GCN3 ISA is available publicly, making it possible to faithfully implement the instructions executed by hardware devices. It is unclear how wide the gap is between PTX and the hardware microcode executed on NVIDIA architectures, and what effect this has on the results of these studies.

9. FUTURE DIRECTIONS AND RESEARCH

The analysis and experiments performed in this study provide multiple directions for future research. First, it may be interesting to extend the study of register value usage and lifetime to a much broader set of applications across many domains. Second, there are implementation details in our simulator that would benefit from improvement. Although we do not believe the results of this study would change significantly, confirming our intuition is worth the effort. Lastly, we propose investigating the impact of the compiler and generated code on the use of Operand Buffering and Register File Caches. The inclusion of these structures is the largest departure from the AMD GCN architecture in our study, and it is possible that the results of our study will change if the compiler is aware of these structures.

Expanding from the scope of this work, we are interested in exploring future research related to the compute fabric and memory hierarchy design. Can we eliminate software-managed scratchpads, which are only understood by expert programmers, while maintaining the performance of these structures? Is the fused multiply add SIMD architecture the most advantageous for GPGPU applications? How do we extract and exploit scalar execution from data-, thread-, or task-parallel programs? While many of these questions have been explored in some manner already, we believe opportunities still remain.

10. CONCLUSIONS

This paper examined modern GPGPU architectures, focusing on their use of vector general-purpose registers. We successfully replicated results from prior studies, showing vector register values are used only a few times and within a short window of the producer. Notably, we confirmed their results on an architecture based on AMD's latest offerings instead of NVIDIA's architecture. We then examined the effectiveness of operand collection and caching structures proposed in prior literature. Our experiments concluded that these structures are highly effective at hiding bank access conflict penalties and reducing the number of required register file reads through the use of operand forwarding. Last, we studied the limiting resources for workgroup dispatch and evaluate the performance and parallelism impact of having additional vector general-purpose registers. We found that some applications effectively utilize the additional registers to increase wave-level parallelism, but not all applications that do so are able to improve performance with increased parallelism. We concluded the work with suggestions for future research related specifically to this work and to the field of general-purpose compute acceleration.

ACKNOWLEDGMENT

Many thanks are due to my mentors and colleagues at AMD Research, specifically Brad Beckmann, Tony Gutierrez, and John Kalamatianos, for their insights, feedback, and overall helpfulness throughout this project.

11. REFERENCES

[1] AMD. Compute Applications. GitHub Repository, 2017. http://github.com/AMDComputeLibraries/ComputeApps. Accessed: February 27, 2017.

[2] AMD. Graphics Core Next (GCN) Architecture. AMD Whitepaper, 2012. http://amd.com/Documents/GCN_Architecture_witepaper.pdf. Accessed: September 20, 2017.

[3] AMD. Graphics Core Next Architecture, Generation 3. AMD Technical Manual, 2016. http://gpuopen.com/wp-content/uploads/2016/08/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf. Accessed: September 20, 2017.

[4] AMD. HCC. https://www.github.com/RadeonOpenCompute/hcc/wiki. Accessed: September 20, 2017.

[5] AMD. HCC Sample Applications, GitHub Repository, 2016. http://github.com/RadeonOpenCompute/HCC-Example-Application. Accessed: September 20, 2017.

[6] AMD. “ROCm: Platform for Development, Discovery and Education Around GPU Computing”. http://gpuopen.com/compute-product/rocm. Accessed: September 20, 2017.

[7] AMD. The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5. http://www.gem5.org/wiki/images/f/fd/AMD_gem5_APU_simulator_micro_2015_final.pptx. 2015.

[8] AMD. “Vega” Instruction Set Architecture Reference Guide. http://developer.amd.com/wordpress/media/2017/08/Vega_Shader_ISA_28July2017.pdf. Accessed: Sept. 20, 2017.

[9] AMD Radeon Technologies Group. Radeon’s next-generation Vega architecture. http://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf. Accessed: Oct. 3, 2017

[10] H. Jeon, G.S. Ravi, N. S. Kim, M. Annavaram. GPU Register File Virtualization. MICRO-48. December 2015.

[11] L. Yu, Y. Pei, T. Chen, M. Wu. Architecture Supported Register Stash for GPGPU. JPDC, 89, pp. 25-36. 2016.

[12] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, K. Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. ISCA’11. June 2011.

[13] M. Gebhart, S. W. Keckler, W. J. Dally. A Compile-Time Managed Multi-Level Register File Hierarchy. MICRO’11. December 2011.

[14] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. The gem5 Simulator. SIGARCH Computer Architecture News, 39(2), pp. 1–7, 2011.

[15] NVIDIA. NVIDIA’s Next Generation CUDA Compute Architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed: Sept. 19, 2017.

[16] V. Volkov. Better Performance at Lower Occupancy. NVIDIA GTC, September 22, 2010. http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf. Accessed: Oct. 10, 2017.

[17] Walton, Mark. “Sixth time lucky: AMD details the Carrizo APU.” https://arstechnica.com/information-technology/2015/06/sixth-time-lucky-amd-details-the-carrizo-apu/. Accessed: Dec. 8, 2017.