Advanced Dynamic Scalarisation for RISC-V GPGPUs

Matthew Naylor, Alexandre Joannou, A. Theodore Markettos, Paul Metzger, Simon W. Moore, Timothy M. Jones
Department of Computer Science and Technology, University of Cambridge, UK
{matthew.naylor, alexandre.joannou, theo.markettos, paul.metzger, simon.moore, timothy.jones}@cl.cam.ac.uk

Abstract—**Recently, researchers have proposed the use of the open RISC-V standard as a basis for GPGPU instruction sets, enabling development of unencumbered GPGPU hardware while reusing extensive general-purpose instruction-set, compiler, and software infrastructure where appropriate. In this paper, we identify and overcome a major deficiency in existing SIMT-style RISC-V GPGPUs: the inability to exploit *value regularity* whereby threads executing in lockstep often compute the same or similar intermediate values. As a solution, we propose *advanced dynamic scalarisation*, a set of new microarchitectural features to exploit value regularity without requiring any extensions to the instruction set or compiler. These features include *register-file compression* to reduce on-chip storage requirements in heavily-threaded designs and *parallel scalar and vector pipelines* to increase instruction throughput, and are fully implemented and evaluated in a new, open-source, synthesisable RISC-V GPGPU called SIMTIGHT. Our results show a reduction in register-file storage requirements of 68%, saving 178KB of fast on-chip memory per 2048-thread streaming multiprocessor, and an increase in run-time performance of 20% at low hardware cost.**

Index Terms—**RISC-V, GPGPU, SIMT, value regularity, scalarization, register-file compression**

I. Introduction

Computer architecture research on GPGPUs often relies on the use of software simulators to model proprietary hardware at the intermediate-language level [1, 2]. However, this makes it difficult to evaluate important effects on microarchitecture, synthesis quality, and translation to machine code, which can be easily overlooked [3]. It also focusses engineering effort on the development of abstract simulators that cannot be directly deployed in the field.

Over the last decade, open-source GPGPU hardware has emerged as a promising alternative. While early designs were based either on proprietary instruction sets [4], preventing free deployment of hardware, or custom instruction sets [5, 6, 7], with no access to a mature software stack or compiler, recent designs have started to employ RISC-V [8, 9, 10], avoiding both problems. This is, however, a relatively new initiative and existing SIMT-style RISC-V GPGPUs have limitations. Most notable, in our view, is that they do not exploit *value regularity* whereby threads executing in lockstep (a *warp*) often compute the same or similar intermediate values. This is a major omission. A seminal study [11] reports that 27% of register reads in SIMT workloads yield the same value for each thread in a warp — so-called *uniform* vectors. Similarly, 44% of reads yield values separated by a constant stride — so-called *affine* vectors. Unoptimised, this leads to a large amount of wasted storage, computation, and energy.

Value regularity can be exploited using a technique known as *scalarisation*, which either operates *statically* in the compiler with instruction-set support [12, 13, 14, 15, 16, 17, 18] or *dynamically* in the microarchitecture [11, 19, 20, 21, 22, 23, 24]. For RISC-V GPGPUs, dynamic scalarisation is particularly attractive because it avoids the need to heavily modify the standard RISC-V instruction set and compiler toolchain. On the other hand, dynamic scalarisation has thus far been unable to reduce register-file storage requirements, a key strength of static scalarisation, and has been limited in its ability to improve instruction throughput.

In this paper, we advance the technique of dynamic scalarisation through the following features.

- **Register-File Compression**. We present a compressed register file design in which uniform and affine vectors are detected at run-time and stored compactly in a small scalar register file (SRF) while general vectors are allocated on-demand in a larger, size-constrained vector register file (VRF). The size of the VRF can be set arbitrarily and overflow is handled by dynamically spilling registers to main memory. Our results show that a compressed register file with a quarter-sized VRF has a minimal impact on instruction throughput while reducing register-file storage requirements by 68%, a saving of 178 kilobytes per 2048-thread streaming multiprocessor.
- **Parallel Scalar and Vector Pipelines**. An instruction with uniform or affine inputs is said to be *scalarisable* if it produces a uniform or affine result. We observe that several features of the general SIMT pipeline are unnecessary to process scalarisable instructions, including active thread selection, the vector register file, and the vector data path, motivating separate scalar and vector pipelines. To allow these pipelines to operate in parallel, we maintain a separate warp queue for each. After executing an instruction in either pipeline, *scalarisable instruction prediction* is used to determine whether the next instruction is likely to be scalarisable or not, and to place the currently executing warp in the appropriate queue. Our results show an average execution-time reduction of 20% in a 32-lane configuration while requiring only a single additional execution unit.

These features are fully implemented and evaluated in a new, prototype, open-source, synthesisable, RISC-V GPGPU called SIMTIGHT. Our contributions are as follows.

- Despite significant recent interest in SIMT-style RISC-V GPGPUs, none currently exploit value regularity. We show how this can be achieved and the savings that are possible, without requiring any modifications to the RISC-V instruction set or its compilers.

- We advance the technique of dynamic scalarisation to achieve reduced register-file storage requirements and efficient execution of scalarisable instructions in parallel with the general vector data path.
- We demonstrate high IPC and performance density in a SIMT-style RISC-V GPGPU, a significant improvement upon the state-of-the-art VORTEX design [9].

II. BACKGROUND

Single Instruction, Multiple Threads (SIMT) is a parallel execution model, popularised by NVIDIA and AMD GPGPUs, that combines the flexibility of a multi-thread programming model with the efficiency of SIMD hardware. The main idea is to execute multiple hardware threads in lockstep with the aim of exploiting regularity between them. Known collectively as a *warp* (or *wavefront*), these threads can exhibit three main kinds of regularity [19]:

- *control-flow regularity*, where threads in a warp follow the same path through the program;
- *memory-access regularity*, where threads in a warp access neighbouring addresses in memory; and
- *value regularity*, where threads in a warp compute the same or similar intermediate values.

While SIMT processors rely on inter-thread regularity to achieve optimal performance, they nevertheless permit general scalar computation within each thread. Indeed, researchers have argued that a general-purpose scalar instruction set such as RISC-V is well-suited to SIMT execution [8], offering an alternative to today's proprietary and bespoke GPGPU ISAs. One of the main challenges of this approach is how to handle thread *divergence* and *convergence*: when threads in a warp take different paths of a branch, it is desirable to join these threads back together at the earliest opportunity. Existing RISC-V GPGPUs have already addressed this problem, either microarchitecturally [8] or with a small extension to the instruction set [9]. Furthermore, researchers have shown how to extend RISC-V to support graphics processing, and have demonstrated full OpenCL and OpenGL software stacks on top of SIMT-style RISC-V GPGPUs [9, 10].

Collange et al. first identified the prevalence of value regularity in SIMT workloads [11], introducing the term *uniform vector* to refer to a variable that has the same value in every thread in a warp, and *affine vector* to refer to a variable whose value is of the form $base + t \times stride$ for each thread t in a warp with a fixed *base* and *stride*. A uniform vector is a special case of an affine vector with a zero stride. The authors report that, on average over a range of CUDA benchmarks running in simulation, 27% of vectors read from the register file, and 15% of vectors written, are uniform. These numbers rise to 44% and 28% respectively for affine vectors.

Several SIMT designs, including some of today's commercial offerings, exploit value regularity *statically* through a technique known as *scalarisation* [12, 13, 14, 15, 16, 17, 18]. Generally, this comprises three main parts: (1) compiler support for identifying register values that are scalar (uniform or affine) across threads executing in lockstep; (2) a separate architectural register file for holding such values more compactly; and (3) special instructions to process these registers more efficiently. However, on RISC-V GPGPUs, all three parts would require major instruction-set and compiler modifications. Ideally, such modifications should be avoided, where possible, to maximise reuse of existing RISC-V infrastructure.

In this paper, we therefore focus on the use of *dynamic scalarisation* to optimise value regularity microarchitecturally. Although there is already a significant body of research on this topic [11, 19, 20, 21, 22, 23, 24], most of it aims to lower energy usage by reducing the number of register banks that get activated during operand collection and writeback. Existing dynamic approaches do not reduce register file storage requirements, a big challenge for modern GPGPUs with hundreds of thousands of hardware threads [25, 26]. They have also had limited success in improving instruction throughput by freeing up the general vector data path. We discuss these limitations further in Section V.

III. SIMTIGHT

In this section, we introduce the SIMTIGHT GPGPU, starting with the baseline design and then adding register-file compression followed by parallel scalar and vector pipelines.

A. Baseline

SIMTIGHT is a new, prototype, open-source, synthesisable RISC-V GPGPU designed for high performance density on 'classic' GPGPU workloads, i.e., workloads sympathetic to memory-access coalescing rules, and which utilise scratchpad memory for efficient parallel random access and inter-thread communication. It implements RISC-V's RV32IMAZfinx profile, i.e., a 32-bit machine with integer, multiply, atomics, and single-precision floating-point support, with a merged integer and floating-point register file.

Our focus in this paper is on the design of the *streaming multiprocessor* (SM) component of SIMTIGHT, as depicted in Figure 1. It is parameterised by the number of threads per warp, which is equivalent to the number of vector lanes (*NumLanes*) as all threads in a warp can execute in a single cycle. It is also parameterised by the number of warps (*NumWarps*), supporting up to $NumLanes \times NumWarps$ hardware threads in total. It employs a 6-stage processor pipeline fed by a barrel scheduler that switches between warps on every clock cycle. At most one instruction per warp is present in the pipeline at any time, avoiding data and control hazards. Multi-cycle instructions are suspended in the execute stage and resumed in the writeback stage without blocking the pipeline, tolerating high-latency operations such as memory loads, integer division, and floating-point operations. Below, we outline each of the main components of the SM: the six pipeline stages, the coalescing unit, and scratchpad memory.

Stage 1: Warp Scheduling (2 cycles). The pipeline maintains a *suspension bit* for each hardware thread denoting whether or not that thread is awaiting the result of a multi-cycle operation, such as a response from memory. The warp scheduler fairly removes a warp ID from the *warp queue* that

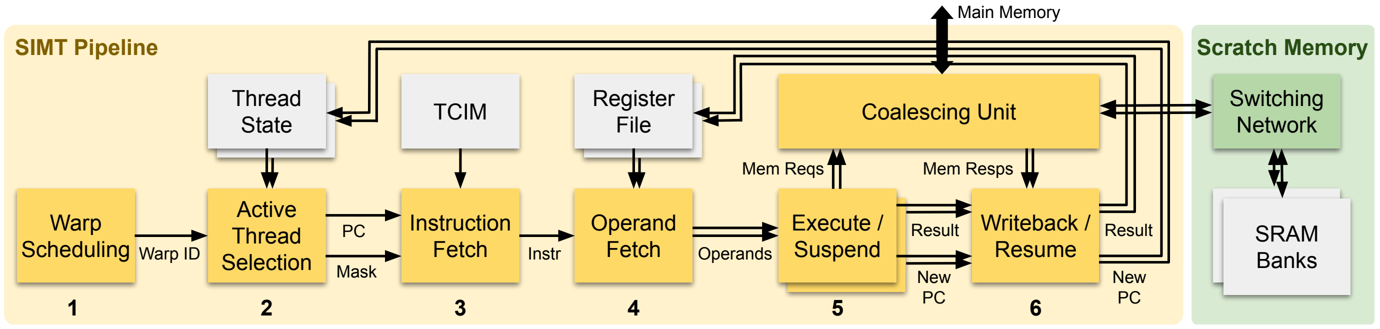

Fig. 1. Diagram of the SIMTIGHT streaming multiprocessor (SM), incuding the pipeline, coalescing unit, and scratchpad memory. Double boxes represent components containing logic or storage that is replicated per vector lane, and double lines represent per-lane wiring.

contains no suspended threads and inserts it into the next stage of the pipeline.

Stage 2: Active-Thread Selection (2 cycles). The pipeline also maintains *thread state* for each hardware thread, including a PC and a *nesting level*. The current warp ID is used to lookup the thread-state memory, and active-thread selection picks a set of threads from the warp that share the maximum nesting level and a common PC. The idea is that each thread's nesting level is incremented when entering a conditional block of code and decremented when leaving it, and prioritising the most deeply nested threads leads to early convergence. This is a simplified version of a stack-less reconvergence design proposed by Collange [8, 27]. Software control of nesting levels is discussed in Section IV.

Stage 3: Instruction Fetch (1 cycle). The common PC shared by the active threads in the warp is used to lookup a tightly coupled instruction memory (TCIM).

Stage 4: Operand Fetch (2 cycles). The register file is implemented as an on-chip memory with two read ports and one write port. The source register IDs in the current instruction together with the warp ID are used to lookup the register file, delivering two 32-bit operands to every lane in a single cycle. These operands are then latched to improve timing. The instruction is also fully decoded in this stage.

Stage 5: Execute / Suspend (1 cycle). The decoded instruction, current PC, and register operands are passed to the per-lane execution units, and the current warp ID is inserted back into the warp queue. Memory load and store instructions issue memory requests to the coalescing unit in this stage. If the instruction is a memory load then each active thread's suspension bit is set.

Stage 6: Writeback / Resume (1 cycle). The per-lane results of the instruction are written to the register file for each active thread in the warp. If no such write is required, then the pipeline opportunistically consumes per-lane responses from the coalescing unit, writes them to the register file, and clears the suspension bit for each corresponding thread. Memory responses for threads in the same warp do not need to arrive at the same time, nor do memory responses need to be in-order with respect to memory requests.

Coalescing Unit. The coalescing unit consumes up to *NumLanes* memory requests per cycle from the pipeline and tries to pack them into a smaller set of wider main memory accsses. It selects the first unsatisfied request (the *leader*) in the warp and applies two selection strategies:

1) the *same-address* strategy selects all requests whose addresses match the leader's; and
2) the *same-block* strategy selects all requests whose addresses are lane-aligned and reside in the same contiguous block of memory as the leader's.

All selected requests are then resolved with a single, wide, burst access to main memory, and the process iterates until all requests are satisfied. This is similar to the coalescing rules from early NVIDIA Tesla devices [28]. For efficient stack access, each thread's stack is interleaved at machine-word (32 bit) granularity such that threads in the same warp accessing the same stack offset will have their accesses coalesced. This interleaving is microarchitectural and invisible to the ISA.

Scratchpad Memory. Any requests that address data in scratchpad memory rather than main memory are forwarded to a set of on-chip SRAM banks via a fast switching network. This network routes requests to the appropriate bank and returns responses back to the originating lane, providing a form of parallel random access. To save area, we use half the number of banks compared to the number of lanes allowing *NumLanes* requests to be satisfied every two cycles in the best case. Large numbers of bank conflicts will reduce throughput significantly, but requests satisfying the same-address strategy (a common case) are coalesced to a single bank access.

B. Register-File Compression

Our register-file compression scheme aims to reduce on-chip storage requirements by exploiting the property that registers will often hold uniform or affine vectors that can be stored compactly. It is parameterised by the amount of space available for storing registers. If this space is ever exhausted, the scheme falls back to main memory by dynamically spilling and fetching registers as required. In this paper, we define the space parameter statically at synthesis time, but it can also be controlled dynamically. Some modern GPUs use unified on-chip memories for register file, scratchpad memory, and cache adjusting the space available to each on a per-application basis [25]. Our compression scheme could be used to dynamically shrink or grow the space available for registers depending, for example, on how much scratchpad memory a *(a) Compressed Register File Write Pipeline*

1: Scalar Lookup. The ID of the register being written is used to lookup the SRF to determine if its *old* value is compressed, and if so, its base (*old.base*) and stride (*old.stride*), and if not, its address (*old.addr*) in the VRF. The value of the first active element (*leader*) in the vector being written (*new*) is determined using a multiplexer.

2: Affine Detection. Each element of *new* is compared against *leader*. If exactly equal then the vector is compressible (uniform). Alternatively, if *new* satisfies the equations

$$\begin{aligned} new_i \;\%\; (\mathit{NumLanes} \times s) &= i \times s \\ \lfloor new_i / (\mathit{NumLanes} \times s) \rfloor &= \lfloor leader / (\mathit{NumLanes} \times s) \rfloor \end{aligned}$$

for every lane i and any stride $s \in \{1, 2, 4\}$ then the vector is compressible (affine). Speculatively, the *old* compressed value (if it exists) is expanded to a full vector *exp* such that $exp_i = old.base + i \times old.stride$.

3a: Vector Deallocation. If *new* is compressible then it is written to the SRF. In this case, if *old* was not compressed then *old.addr* is pushed onto the *free slot stack*.

3b: Vector Allocation. If *new* is not compressible and *old* was compressed then *new* is written to the VRF at an address popped from the free slot stack. In this case, if *new* represents a partial write then its inactive elements are replaced with the corresponding elements of *exp* before being written to the VRF. The SRF is updated to point to the newly allocated vector. The dynamic spill mechanism (Figure 3) ensures that the free slot stack will never be empty.

3c: Vector Update. If *new* is not compressible and *old* was not compressed then *new* is written to the VRF at address *old.addr*.

(b) Compressed Register File Read Pipeline

1: Scalar lookup. The ID of the register being read is used to lookup the SRF to determine if its *current* value is compressed, and if so, its value *current.base* and *current.stride*, and if not, its address *current.addr* in the VRF.

2: Vector lookup. If the register is compressed then it is expanded to the vector $current.base + i \times current.stride$ for each vector lane i. Otherwise, the VRF is looked up at address *current.addr*.

Fig. 2. The compressed register file uses a 3-stage write pipeline (a) and a 2-stage read pipeline (b).

particular application is using or how effective the compression is in a particular application.

Our compressed register file comprises a scalar register file (SRF) and a size-constrained vector register file (VRF). For every architectural vector register, the SRF either holds a compressed vector (base and stride) or a pointer to a register in the VRF. We encode the stride using two bits, allowing representation of the four most common stride values (0, 1, 2, and 4) at low hardware cost. Vectors in the VRF are allocated and deallocated dynamically using a *free-slot stack* that tracks unused locations in the VRF. Subject to some minor constraints discussed below, the size of the VRF can be set arbitrarily.

Read / Write Pipelining. We introduce a 3-stage pipeline on the write path of the compressed register file to compress vectors and to allocate/deallocate vectors in the VRF, as detailed in Figure 2(a). We also introduce a 2-stage pipeline on the read path to decompress vectors in the SRF and to follow the indirection to the VRF when required, as detailed in Figure 2(b). Both the write and read pipelines require read access to the SRF for two register operands so the SRF requires four read ports in total. We achieve this using two on-chip memories, each with two read ports. Writes to the SRF go to both memories so the two always contain identical data. This provides the necessary bandwidth at the cost of doubling the SRF size, but this cost is expected to be small in comparison to the size of the VRF.

Dynamic Spilling. Our dynamic spill mechanism ensures that the VRF never overflows. When the number of unused vector registers falls below the number of warps, the SIMT processor pipeline enters *spill mode*. This threshold guarantees that there are enough available registers for every in-flight instruction to write a vector result if required, given that only one instruction per warp can be in-flight at any time. In spill mode, the pipeline operates largely as normal except for the modifications detailed in Figure 3(a). Subsequently, if a fetched instruction requires access to a spilled register, it passes through the pipeline but does not execute; instead, it unspills the required register and tries again to execute the next time the warp is scheduled. The pipeline modifications for unspilling are detailed in Figure 3(b).

Compressed Stack Cache. Information about compressed vectors can be passed from the register file to the memory subsystem for further storage savings. As a proof of concept, we have added a small component to our coalescing unit to cache writes of *compressed* vectors. The RISC-V architecture is constrained to 32 general-purpose integer registers, or 16 under the E extension, and the compiler will insert code to spill registers to memory if these limits are exceeded. We reduce memory bandwidth by caching many of these spills at low hardware cost. The component is tailored to caching small GPU-thread stacks: it only applies to loads and stores to stack memory and it does not cache load misses.

C. Parallel Scalar and Vector Pipelines

An instruction is said to be *scalarisable* if its result can be computed on a single execution unit rather than being issued to all vector lanes. Many instructions with uniform or affine inputs are scalarisable. In this section, we are interested in how we can execute scalarisable instructions in parallel with non-scalarisable ones, potentially doubling throughput at the cost of only one additional vector lane / execution unit.

(a) Pipeline Modifications for Spilling

1: Warp Scheduling. When the number of unused vector registers falls below the spill threshold (equal to the number of warps), the pipeline enters *spill mode*.

4: Operand Fetch. The pipeline selects a register to spill using either a round-robin or pseudo-least-recently-used spill policy. The chosen register is then fetched from the register file, instead of the fetched instruction's operands. For correct operation, the chosen register is never one of the source or destination registers needed by the fetched instruction; this guarantees that the processor cannot enter a livelock whereby registers get spilled and immediately unspilled without executing any instructions. As a result, the smallest permitted size for the VRF is three vector registers per warp plus the spill threshold — a maximum compression factor of $8\times$ for the VRF. The chosen register must also hold an uncompressed vector.

5: Execute. In spill mode, the pipeline does not actually execute the fetched instruction. Instead, it issues a memory write request for the register to spill. Spilled registers are stored in a region of main memory big enough to hold the entire uncompressed register file.

6: Writeback. The register is marked as *spilled* in the SRF. The VRF is now known to contain one additional unused vector register.

(b) Pipeline Modifications for Unspilling

4: Operand Fetch. When any of the source or destination registers for the current instruction are marked as spilled, one of those registers is selected for unspilling.

5: Execute. If a register has been selected for unspilling, the current instruction is not executed. Instead, a memory read request is issued for the desired register. The warp is suspended, just as it would be for a normal load instruction. The next time the warp is scheduled, the instruction will have one fewer registers that need to be unspilled, ensuring forward progress.

Fig. 3. SIMT processor pipeline modifications to support dynamic register spilling (a) and unspilling (b).

For an instruction to be scalarisable, we require all threads in the warp to be active. This is a common though not strictly necessary requirement (an alternative is explored by Liu et al. [22]) but it captures the common case and permits a cheap implementation. Consequently, the active-thread selection stage of the SIMT pipeline is unnecessary for processing scalarisable instructions as the active threads are implied. Furthermore, scalarisable instructions do not require access to the vector register file or the vector data path. We therefore opt for completely separate scalar and vector pipelines. Each pipeline is fed by a separate warp queue, and warps are moved between the two using *scalarisable-instruction prediction*.

Separate Pipelines. We introduce a dedicated scalar pipeline that runs in parallel with the main vector pipeline. This requires two additional read ports and one additional write port to the SRF. The additional write port is already available as the SRF is implemented using an on-chip memory with two read ports and two write ports, and the additional read ports are provided by replication of the relatively small SRF. We also make use of a TCIM with two read ports to enable parallel instruction fetch.

Prediction Table. We introduce a *scalarisable-instruction prediction table* that maps each instruction address to a single bit denoting whether that instruction was scalarisable the last time it executed. This table is written to in the operand latch stage of the main vector pipeline, when the current instruction and all its operands are known. In the execute stage, when the address of the *next* instruction to execute is known, the prediction table is looked up. If the table returns true then the warp is inserted into a *scalar warp queue* rather than the general vector warp queue.

The scalar pipeline is similar to a standard five-stage pipeline, except for an initial scheduling stage that selects warps from the scalar warp queue. When the current instruction and all its operands are known, the scalar pipeline checks whether or not the scalarisable prediction was correct. If not, execution is aborted and the warp is moved back to the main vector warp queue. If so, the instruction is executed and, once we know the address of the next instruction, the prediction table is again looked up. Depending on the result of the lookup, the warp is either moved back to the main vector warp queue, or reinserted into the scalar warp queue.

Conditions for Scalarisation. To move a warp from the vector warp queue to the scalar warp queue, we require not only that the next instruction is predicted as scalarisable but also that all lanes are active and that the current instruction does not diverge the warp. This avoids partial writes to the register file, ensuring that the scalar pipeline only requires write access to the SRF and not the VRF. We also require that the operands for the current instruction are stored in the SRF and satisfy one of the following criteria: (1) all operands are uniform; or (2) the instruction is an *add* instruction with one uniform operand and one affine operand. In our experience affine addition is by far the common case, but this could be generalised in future. We do not yet support load and store instructions in the scalar pipeline, which is another avenue for future improvement.

IV. Evaluation

In this section, we introduce our experimental setup and then evaluate the performance of SIMTight and its main features over a range of benchmark kernels. All artefacts used for evaluation are available online [29].

Benchmarks. We have developed a thin software layer called NoCL that supports writing CUDA-style compute kernels in plain C++ (no special compute language is required). This allows CUDA and OpenCL benchmarks to be easily ported to SIMTight. An example NoCL kernel is shown in

```
0  struct Histogram : Kernel {
1   int len; unsigned char* in; int* out;
2
3   void kernel() {
4     // Allocate bins in shared local memory
5     int* bins = shared.alloc<int>(256);
6     // Initialise bins
7     for (int i = threadIdx.x; i<256; i += blockDim.x)
8         bins[i] = 0;
9     __syncthreads();
10    // Accumulate bins
11    for (int i = threadIdx.x; i<len; i += blockDim.x)
12        atomicAdd(&bins[in[i]], 1);
13    __syncthreads();
14    // Write bins to global memory
15    for (int i = threadIdx.x; i<256; i += blockDim.x)
16        out[i] = bins[i];
17  }
18 };
```

Fig. 4. NoCL kernel to compute the histogram of a given byte array. Lines 6–16 are identical compared to the CUDA version of the same kernel.

Configuration	**Use DSP Blocks?**	**Fmax (MHz)**	**Area (ALMs)**	**Area (DSPs)**
Baseline - FP	Yes	207	48K	66
Baseline	Yes	204	94K	297
Baseline + RFC	Yes	196	100K	297
Baseline + RFC + PP	Yes	194	103K	299
Baseline - FP	No	205	69K	0
Baseline	No	205	176K	0
Baseline + RFC	No	191	181K	0
Baseline + RFC + PP	No	189	188K	0

Fig. 5. Synthesis results for a single 32-lane 64-warp SIMTIGHT SM on an Intel Stratix 10 FPGA with and without floating-point (FP), register-file compression (RFC), and parallel scalar and vector pipelines (PP).

Figure 4. The main area where NoCL differs from CUDA is that it requires the programmer to explicitly mark divergence and convergence points in the program. This amounts to incrementing and decrementing the per-thread nesting level (Section III-A) before and after each conditional block using NoCL primitive functions. Details can be found in the NoCL manual [29].

The majority of benchmarks used for evaluation have been transcribed to NoCL from existing kernels published by NVIDIA. To capture the strengths and weaknesses of SIMTIGHT, we have obtained a range of benchmarks with varying levels of inter-thread regularity.

Experimental Setup. Following modern NVIDIA devices, we use 64 warps and 32 threads per warp providing 2048 hardware threads in total per SM. We obtain all results on a Terasic DE10-Pro development board with an Intel Stratix-10 FPGA holding a single SIMTIGHT SM connected to a single DDR4 DIMM. On the software side, we use version 12.2 of the standard GCC compiler targeting RISC-V.

Baseline. Figure 6 shows the instruction throughput of a single 32-lane 64-warp SIMTIGHT SM. In several benchmarks, the IPC approaches the number of vector lanes, which is the upper bound on run-time performance. As expected, performance suffers in benchmarks involving significant thread divergence and/or non-coalesceable memory access patterns, notably SPMV and VecGCD.

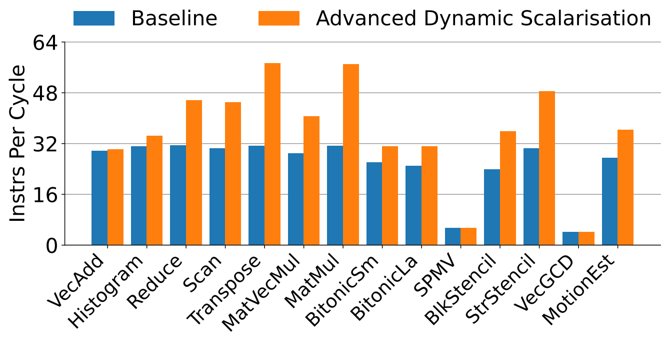

Fig. 6. Instruction throughput for a single 32-lane SIMTIGHT SM. Many workloads achieve an IPC approaching the number of vector lanes, and surpass it using parallel scalar and vector pipelines.

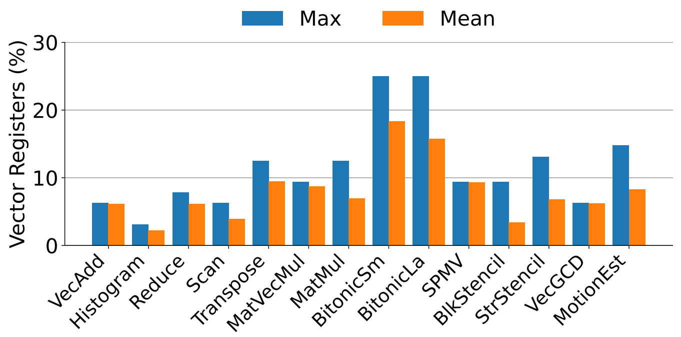

Fig. 7. Proportion of registers stored as vectors in the VRF (lower is better). Remaining registers are stored compactly as scalars in the SRF.

A useful comparison point for the SIMTIGHT baseline is the recently published RISC-V VORTEX GPU [9], which is also evaluated on a Stratix-10 FPGA. On the VecAdd benchmark, VORTEX achieves an IPC of 20 using 32 $\times$ 4-lane 4-warp SMs (128 execution units) running at 200MHz. This amounts to 4 billion instructions per second (BIPS) compared to 5.4 BIPS from a single SIMTIGHT SM (32 execution units) on the same benchmark. Excluding functionality that SIMTIGHT does not support, this VORTEX configuration is over $4.5\times$ larger than SIMTIGHT, so the performance density of SIMTIGHT on the VecAdd benchmark is over $6\times$ higher.

Register File Compression. The uncompressed register file in a 32-lane 64-warp RV32I SM contains 2,048 32-element vector registers. Figure 7 shows the proportion of these registers that actually get stored as vectors, i.e., in the VRF, when using register-file compression. The remaining registers are all stored compactly as scalars in the SRF. Over all benchmarks, the geometric mean of the maximum proportions of registers stored in the VRF is just 12%. Figure 8 shows the storage savings and performance overheads of register-file compression for various VRF size limits. In particular, a quarter-sized VRF containing just 512 vector registers reduces register-file storage requirements by 68% with only a 1% impact on execution cycles and main-memory accesses. For comparison, we run RV32E-compiled benchmarks on the baseline, halving the number of architectural registers per thread statically from 32 to 16 at the cost of increased register spilling. This reduces register-file storage requirements by only

VRF Size (Vector Registers)	Total Storage (Kilobits)	Compression Ratio	Cycle Overhead	Main Memory Access Overhead
1024	1202	1 : 0.57	0.8%	0.0%
512	672	1 : 0.32	1.0%	1.3%
256	407	1 : 0.19	9.5%	47.9%

Fig. 8. Storage savings and geometric-mean overheads due to register-file compression (without the compressed stack cache) for a half-size, quarter-size, and eighth-size VRF relative to the baseline (which has 2048 vector registers occupying 2097 kilobits of storage).

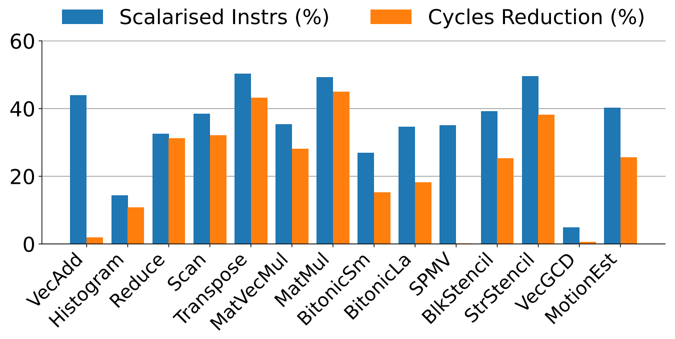

Fig. 9. Overall, parallel scalar and vector pipelines enable 31% of instructions to be scalarised for a 24% reduction in execution time (clock cycles).

50% yet incurs greater overheads: a 7% increase in execution cycles along with a 61% increase in main-memory accesses.

Compressed Stack Cache. We find that a 12KB compressed stack cache reduces main memory traffic by a geomean of 52%, suggesting that a significant amount of main memory accesses arise from static register spilling. Note that, in many benchmarks, the majority of memory accesses are to scratchpad memory rather than main memory.

Parallel Scalar and Vector Pipelines. Figure 9 shows the impact of parallel pipelines on execution time. In many cases, the reduction in cycles closely follows the number of scalarised instructions, suggesting that both pipelines are well utilised. The two main exceptions are VecAdd and SPMV, where memory-access latency is a limiting factor due to high DDR4 bandwidth utilisation and non-coalesceable access patterns respectively. Overall, the geometric-mean reduction in clock-cycle execution time is 24%. Accounting for the small dip in Fmax compared to the baseline (Figure 5), the geometric-mean reduction in wall-clock execution time is 20%.

Logic Area. To measure logic-area overhead, we disable the use of DSP blocks on the FPGA to obtain a single ALM count representing all logic used, as shown in 5. This gives a logic area overhead 7% per SM for advanced dynamic scalaristion.

V. Related Work

We review existing work on dynamic scalarisation and GPU register-file compression, contrasting it with our own work.

Detecting Value Regularity. Existing dynamic scalarisation techniques use one of two methods to detect scalar values: *inference* or *comparison*. Approaches based on inference [11, 19, 20] use a set of axioms to introduce uniform/affine vectors along with a set of inference rules that propagate uniform/affine operands to uniform/affine results. In contrast, approaches based on comparison [21, 22, 23, 24] detect uniform/affine vectors using an array of comparators in the writeback stage of the pipeline. Inference has the advantage of requiring less logic and energy by avoiding the comparison network, while comparison has the advantage of detecting more uniform/affine vectors, such as those loaded from memory. Both approaches are effective, and SIMTIGHT takes the comparison approach for simplicity: it allows the entire detection subsystem to be abstracted out of the pipeline and into the register file. Our main focus is on how to exploit value regularity rather than how to detect it, which we explore in the remainder of this section.

Redundant Register Accesses. Several dynamic scalarisation efforts aim to exploit uniform/affine vectors by storing them more compactly in the register file, reducing the energy consumption of register loads and stores. This is achieved either by introducing an additional, narrower, more power-efficient scalar register file alongside the main vector register file [19, 20, 21], or by activating fewer banks in the vector register file on lookup [22, 23, 24]. However, in all these approaches the register file still contains one physical register for every architectural register; there is no storage reduction.

Redundant Storage. Only prior work by Collange and Kouyoumdjian [30] exploits dynamic scalarisation to reduce on-chip storage requirements. They propose to add an *affine vector cache* alongside the general L1 vector cache to achieve a 59% larger usable cache capacity for the same amount of onchip storage. The approach is particularly affective at optimising compiler-inserted register spills of affine vectors and inspired us to develop SIMTIGHT's compressed stack cache. As future work, the authors suggest moving affine compression into the register file to increase compression benefits, which is precisely what we have done in SIMTIGHT. Our register-file compression design also enables other optimisations such as parallel scalar and vector pipelines.

Redundant Computation. An instruction that has uniform/affine operands can in many cases be executed on a single execution unit, saving power [19, 20, 21, 22]. Furthermore, in SIMT microarchitectures where the number of vector lanes is smaller than the warp size, and hence where each warp must be serialised over multiple cycles, scalarisable instructions can be reduced to a single cycle of execution, improving throughput [19, 20]. However, this technique does not apply to SIMT cores that support single-cycle execution of warps, and does not exploit parallelism.

Gilani et al. [21] propose to use the dual-issue feature of their baseline GPGPU model to execute scalarisable and vector instructions in parallel. However, their design has a major limitation: their baseline serialises warps over multiple clock cycles and they do not lift this restriction for scalarisable instructions, i.e., scalarisable instructions require multiple cycles to complete, even though they execute in parallel with vector instructions. Using the dual-issue feature of the SIMT pipeline for dynamic scalarisation is also rather wasteful as scalarisable instructions permit a simpler processing path. Resources such as active-thread selection, the vector register file, and the vector data paths are wasted on scalarisable instructions.

GPU Register File Compression. GPU register-file compression has been explored outside the context of value regularity and dynamic scalarisation. A common observation [31, 32, 33] is that while the compiler assigns each thread the maximum number of registers that are ever simultaneously live, only a subset of these may be live for long periods of execution. By incorporating liveness information from the compiler into the instruction stream, hardware can allocate registers on demand, reducing the average register-file occupancy. If register storage becomes exhausted, a mechanism called *warp throttling* is used, whereby some warps get suspended and swapped out to main memory, freeing space for the remaining warps.

Our dynamic register spilling scheme may provide some of the benefits of these approaches without the need for compiler-inserted liveness information: our least-recently-used spill policy will naturally evict registers that have not been used for period of time, if the limits of the VRF are reached.

Warp throttling is an interesting alternative to dynamic register spilling. On the one hand, it may suffer from less memory thrashing in highly space-constrained configurations. On the other hand, it would lower the latency tolerance of the core due to fewer active warps. Warp throttling also requires an advanced run-time system that is capable of saving and restoring warp contexts at arbitrary points during execution.

VI. Conclusion

Value regularity occurs extensively in SIMT-style RISC-V GPGPUs, and advanced dynamic scalarisation is highly effective at exploiting it. We have demonstrated significant on-chip storage savings, memory-access reductions, and run-time performance improvements, all at low hardware cost and without modifications to the instruction set or compiler. Emerging RISC-V GPGPUs can therefore exploit value regularity while continuing to benefit from substantial infrastructure reuse across the RISC-V ecosystem.

Acknowledgements. Thanks to Jianyi Cheng for his improvements to the SIMTight build process. Our work was supported by the UK EPSRC under the *CAPcelerate Project* (EP/V000381/1) and the *Chrompartments Project* (EP/X015963/1), both part of the Digital Security by Design (DSbD) Programme and the DSbDtech initiative. Additional data related to this publication is available at https://doi.org/10.17863/CAM.111868. The latest version of SIMTight is available at https://github.com/CTSRD-CHERI/SIMTight.

References

[1] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS 2009*.
[2] H. Kim *et al.*, "MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide," 2012, commit `94942a2`, accessed 2024-02-02. [Online]. Available: https://www.github.com/gthparch/macsim
[3] A. Gutierrez *et al.*, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *HPCA 2018*.
[4] R. Balasubramanian *et al.*, "Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU," *ACM Transactions on Architecture and Code Optimisation*, vol. 12, no. 2, 2015.
[5] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *FPT 2013*.
[6] M. Al Kadi, B. Janssen, and M. Huebner, "FGPU: An SIMT-Architecture for FPGAs," in *FPGA 2016*.
[7] J. Bush *et al.*, "NyuziRaster: Optimizing rasterizer performance and energy in the Nyuzi open source GPU," in *ISPASS 2016*.
[8] C. Collange, "Simty: generalized SIMT execution on RISC-V," in *CARRV 2017*.
[9] B. Tine *et al.*, "Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics," in *MICRO 2021*.
[10] B. Tine *et al.*, "Skybox: Open-Source Graphic Rendering on Programmable RISC-V GPUs," in *ASPLOS 2023, Volume 3*. Association for Computing Machinery.
[11] C. Collange, D. Defour, and Y. Zhang, "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations," in *Euro-Par 2009*.
[12] C. Collange, "Identifying scalar behavior in CUDA kernels," ENS Lyon, Research Report, 2011. [Online]. Available: https://hal.science/hal-00555134
[13] Y. Lee *et al.*, "Convergence and scalarization for data-parallel architectures," in *CGO 2013*.
[14] K. Wang and C. Lin, "Decoupled affine computation for SIMT GPUs," in *ISCA 2017*.
[15] Z. Chen and D. Kaeli, "Balancing Scalar and Vector Execution on GPU Architectures," in *IPDPS 2016*.
[16] A. Yilmazer, Z. Chen, and D. Kaeli, "Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs," in *IPDPS 2014*.
[17] Z. Jia *et al.*, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking," 2019, arXiv 1903.07486.
[18] Advanced Micro Devices (AMD), *Southern Islands Series Instruction Set Architecture 1.1*, 2012.
[19] J. Kim *et al.*, "Microarchitectural mechanisms to exploit value structure in SIMT architectures," in *ISCA 2013*.
[20] P. Xiang *et al.*, "Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement," in *ICS 2013*.
[21] S. Z. Gilani, N. S. Kim, and M. Schulte, "Power-efficient computing for compute-intensive GPGPU applications," in *PACT 2012*.
[22] Z. Liu *et al.*, "G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs," in *HPCA 2017*.
[23] S. Lee *et al.*, "Warped-Compression: Enabling Power Efficient GPUs through Register Compression," in *ISCA 2015*.
[24] S. Lee *et al.*, "Improving Energy Efficiency of GPUs through Data Compression and Compressed Execution," *IEEE Transactions on Computers*, vol. 66, no. 05, 2017.
[25] M. Gebhart *et al.*, "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor," in *MICRO 2012*.
[26] S. Mittal, "A Survey of Techniques for Architecting and Managing GPU Register File," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, 2017.
[27] C. Collange, "Stack-less SIMT reconvergence at low cost," ENS Lyon, Research Report, 2011. [Online]. Available: https://hal.science/hal-00622654
[28] E. Lindholm *et al.*, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, 2008.
[29] M. Naylor, "Research data supporting 'Advanced Dynamic Scalarisation for RISC-V GPGPUs'," https://doi.org/10.17863/CAM.111868, 2024.
[30] C. Collange and A. Kouyoumdjian, "Affine Vector Cache for memory bandwidth savings," ENS de Lyon, Research Report, 2011. [Online]. Available: https://ens-lyon.hal.science/ensl-00649200
[31] H. Jeon *et al.*, "GPU Register File Virtualization," in *MICRO 2015*.
[32] L. Yu *et al.*, "Architecture supported register stash for GPGPU," *Journal of Parallel and Distributed Computing*, vol. 89, 2016.
[33] F. Khorasani *et al.*, "RegMutex: Inter-Warp GPU Register Time-Sharing," in *ISCA 2018*.