G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs

Zhenhong Liu[1], Syed Gilani[2], Murali Annavaram[3], Nam Sung Kim[1]
[1]University of Illinois Urbana-Champaign; [2]AMD; [3]University of Southern California
zliu118@illinois.edu; syed.gilani@amd.com; annavara@usc.edu

Abstract
The GPU has provide higher throughput by integrating more execution resources into a single chip without unduly compromising power efficiency. With the power wall challenge, however, increasing the throughput will require significant improvement in power efficiency. To accomplish this goal, we propose G-Scalar, a cost-effective generalized scalar execution architecture for GPUs in this paper. G-Scalar offers two key advantages over prior architectures supporting scalar execution for only non-divergent arithmetic/logic instructions. First, G-Scalar is more power-efficient as it can also support scalar execution of divergent and special-function instructions, the fraction of which in contemporary GPU applications has notably increased. Second, G-Scalar is less expensive as it can share most of its hardware resources with register value compression, of which adoption has been strongly promoted to reduce high power consumption of accessing the large register file. Compared with the baseline and previous scalar architectures, G-Scalar improves power efficiency by 24% and 15%, respectively, at a negligible cost.

1. Introduction
A GPU typically consists of a large number of cores sharing Fetch, Decode, and Scheduling (FDS) logic. Although designed for graphics processing, GPUs are also used to accelerate data-parallel general-purpose applications. The computational complexity of both graphics and general-purpose applications has been steadily increasing, driving the demand for more GPU computing capability. The improvement of GPU computing capability has been primarily accomplished by integrating more hardware resources and increasing the operating frequency. However, the effectiveness of such techniques is limited by the power and thermal constraints, especially when semiconductor technology scaling approaches its fundamental physics limit [1].

A detailed analysis shows that the execution units and register file are the two most power-consuming components in the GPU and consume about 24% and 16% of total GPU chip power, respectively [2]. In compute-intensive applications, the percentage is even higher. Consequently, a large body of work has been proposed to improve power efficiency of these two power-consuming components, exploiting various characteristics of applications (*e.g.*, [3, 4, 5, 6]).

A GPU register file is typically comprised of 32×4-byte vector registers, each of which can supply source operands for 32 threads in a single warp [7]. It has been observed that 32 registers in a vector register often stores the same (scalar) value [3, 5, 6] or similar values [4, 8] at runtime. Such value characteristics were exploited in two distinct ways to improve power efficiency of GPUs: (1) scalar execution of instructions [3, 5, 6, 8] and (2) register value compression [4].

When an instruction operates only on scalar values, all 32 threads of a warp compute the same value. To reduce redundant computations and thus power consumption of execution pipelines, prior work proposed scalar execution architectures to execute such instructions using a dedicated scalar execution pipeline [3, 5, 6]. The scope of scalar execution was expanded to instructions operating on vector registers storing similar values [8] although the benefit is primarily from operating on scalar values. An orthogonal approach to exploit value similarity was register value compression, which reduces high power consumption of accessing registers [4].

As GPUs began to support more general-purpose applications, the fraction of divergent instructions in contemporary GPU applications has significantly increased, and various optimization techniques have been proposed to efficiently handle them (*e.g.*, [9, 10, 11, 12, 13, 14, 15]). Especially, we observe that many divergent instructions are also eligible for scalar execution (denoted by divergent scalar instructions) if we consider only 4-byte register values in active lanes in a divergent path. Figure 1 shows that 28% of total instructions are divergent instructions and 45% of total divergent instructions are divergent scalar instructions in these benchmarks.

Despite such a high percentage of divergent scalar instructions in contemporary GPU applications, prior architectures do not support scalar execution of divergent instructions. Furthermore, they do not consider scalar execution of special-function instructions such as `sin`, `cos` and `exp`, as implementing a separate Special-Function Unit (SFU) for a scalar execution pipeline incurs a very high chip cost. For example, NVIDIA GTX 480 provides only one 4-lane SFU per Stream Multiprocessor (SM) [7] partly due to its chip cost, while special-function instructions consume 3~24× more energy than

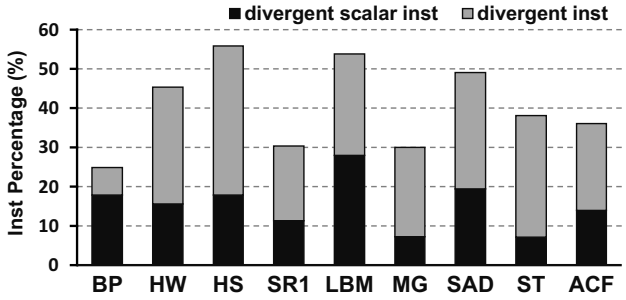

Figure 1: Percentage of divergent instructions and divergent scalar instructions in total instructions.

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.51

IEEE computer society

typical arithmetic/logic instructions [2]. Hence, ignoring divergent code regions while exploiting register value locality will undeniably reduce the effectiveness of prior techniques.

In this paper, we propose G-Scalar, enhanced GPU architecture to cost-effectively support scalar execution of instructions. Prior scalar execution architectures require a dedicated scalar execution pipeline [3, 5, 6], and/or only support scalar execution of non-divergent arithmetic/logic instructions [3, 5, 6, 8]. Consequently, supporting the scalar execution of only non-divergent arithmetic/logic instructions may significantly limit the benefit for many GPU applications. In contrast, G-Scalar can support scalar execution of not only non-divergent arithmetic/logic instructions but also divergent and special-function instructions without any dedicated scalar execution pipeline, as it can share most of hardware resources with register value compression. In particular, if a GPU adopts our low-cost register value compression technique, G-Scalar can support generalized scalar execution practically at no cost. Our key contributions are as follows:

- We show that a significant fraction of instructions is divergent and eligible for scalar execution in many contemporary GPU applications. Therefore, scalar execution of such divergent instructions is critical for pushing the power-efficiency envelope.
- We propose G-Scalar that can support scalar execution of not only non-divergent arithmetic/logic instructions but also divergent and special-function instructions without requiring a dedicated scalar execution pipeline. G-Scalar can improve the power efficiency of a GPU similar to NVIDIA GTX 480 by 24% and 15%, compared with the baseline and previous scalar execution GPU architectures, respectively.
- We propose an enhancement of GPU microarchitecture to cost-effectively support register value compression. This microarchitecture shares most of required hardware resources with G-Scalar, reducing the power consumption of register file by 54% with at most 1% more chip space than the baseline GPU architecture.

The rest of the paper is organized as follows. Section 2 introduces the overall baseline GPU architecture. Section 3 describes a register value compression technique. Section 4 describes how we leverage the hardware resources for register value compression to efficiently support G-Scalar. Section 5 shows our evaluation results. Section 6 discusses related work. Section 7 concludes the paper.

2. Background

2.1 Register File and SIMT Pipelines

We consider a GPU architecture that is similar to NVIDIA GTX 480 as our baseline [7]. The baseline GPU architecture has 15 SMs. To support massive single-instruction multiple-thread (SIMT) execution, each SM in GTX 480 has 32,768 4-byte registers (*i.e.*, 1024 vector registers, each of which consists of 32 4-byte registers). As a typical instruction operates on two or three vector registers, the register file is partitioned into 16 banks. This allows an instruction to access multiple vector registers in a single cycle with single-port SRAM arrays constituting each bank, but necessitates a 16×16 crossbar between banks and 16 operand collectors supplying operands to SIMT execution pipelines [16]. Each bank provides 64 32×4-byte vector registers and consists of eight 64×128-bit single-port SRAM arrays. When the GPU accesses a bank, all eight SRAM arrays are activated. As all the operands are ready for an instruction, a scheduler dispatch the instruction to an appropriate SIMT execution pipeline [16].

Each SM has three types of SIMT execution pipelines: (1) two 16-lane arithmetic/logic, (2) one 16-lane memory, and (3) one 4-lane special-function pipelines. Depending on the width of each execution pipeline, a warp is dispatched to the arithmetic/logic, memory and special-function pipelines over 2, 2, and 8 cycles, respectively. Since each SM has two arithmetic/logic pipelines, up to two arithmetic/logic instructions can be dispatched in a single cycle.

2.2 Register Value Compression

For on-chip memory value compression, the base-delta-immediate (BDI) compression scheme [17] has been used [4, 18, 19, 20]. It exploits the observation that values stored in a cache line [17] or a vector register [4] are similar. For example, if eight 4-byte values from a given SIMT pipeline are $\texttt{C04039C0}_{16}$, $\texttt{C04039C8}_{16}$, …, and $\texttt{C04039F8}_{16}$, the first value becomes the base value, and 0_{16}, 8_{16}, …, and 38_{16} become the delta values for the eight 4-byte values. Consequently, a 256-bit (=8×4-byte) value can be compressed to a 96-bit value (*i.e.*, 32-bit base and 8×8-bit delta values, respectively) and stored in a vector register in a compressed format. Such an implementation of the compression hardware for BDI typically requires N 32-bit adders/subtractors [17] where N is the number of 4-byte values in a cache line or a vector register.

3. Register Value Compression

Our key contribution is G-Scalar, generalized scalar execution architecture that supports scalar execution of not only non-divergent arithmetic/logic but also divergent and special-function instructions. Before explaining G-Scalar in depth, however, we need to describe our enhanced GPU microarchitecture to cost-effectively support a register value compression technique, as G-Scalar is built upon this microarchitecture enhancement.

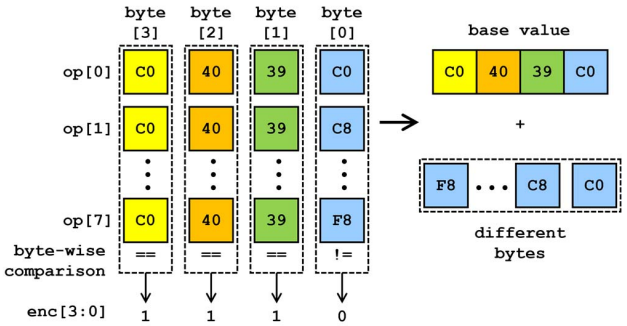

Figure 2: Proposed compression technique. Byte-wise comparison results are store in compression bits `enc[3:0]`. `1` (`0`) means all bytes are (not) the same. `op` denotes an operand stored in a 4-byte register.

3.1 Compression

Exploiting value similarity, we first propose a register value compression technique with a lower cost than BDI [17]. Using the same example in Section 2.2, we illustrate our compression technique in Figure 2. Instead of subtracting the base value from each operand, our compression technique directly compares all 4-byte values in a vector register byte by byte to check whether or not `byte[i]` for every `op` has the same value. In this example, the byte-wise comparison determines that `byte[3]` (=$C0_{16}$), `byte[2]` (=40_{16}), and `byte[1]` (=39_{16}) have the same values across `op[0]`, `op[1]`, …, `op[7]`. That is, $C04039_{16}$ becomes the base value, and `byte[0]` from each `op` (= $F8_{16}$, …, $C8_{16}$, $C0_{16}$) becomes the delta value for each `op`. Consequently, our compression technique stores the 3-byte base value and 8 different bytes with the corresponding encoding bits (1110_2). In our compression technique, the base value can be up to 4 bytes (for a vector register storing a scalar value), and we always use bytes from `op[0]` for the base value for simplicity. Note that our technique is more efficient than BDI in hardware implementation although it does not provide the same compression ratio as BDI in some special cases, especially when the hexadecimal representation of two similar values differs widely.

3.2 Microarchitecture Support

Figure 3 depicts necessary microarchitecture changes to support our register value compression technique for a hypothetical 16-lane SIMT pipeline. In the baseline register file, a bank is comprised of 4 SRAM arrays (8 for a 32-lane SIMT pipeline), each of which stores four 4-byte values. Suppose that we desire to retrieve only `byte[0]` of each 4-byte value to reconstruct sixteen 4-byte values. In a traditional register file, `byte[0]` of sixteen 4-byte values are distributed across all four arrays. Thus, we still need to activate all four arrays.

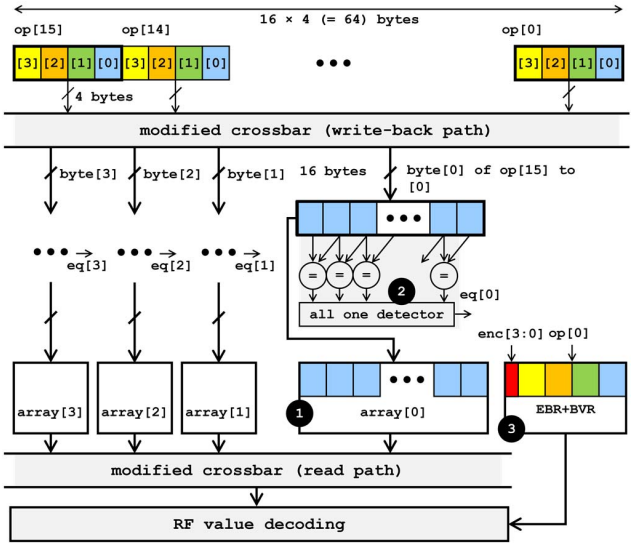

Figure 3: Microarchitecture to support register value compression for 16-lane SIMT. `BVR` and `EBR` denote base value register and compression bit register, respectively. `op`, `enc`, `BVR` and `EBR` denote operand, compression bit, base value register and compression bit register.

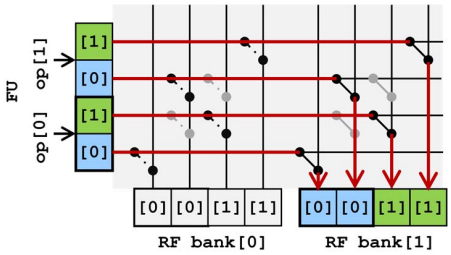

Figure 4: Adapted crossbar for reordering bytes. `FU` denotes a hypothetical 2-lane SIMT pipeline.

To support power-efficient accesses of the register file for our register value compression technique, we propose to reorder bytes such that an array stores only `byte[i]` of all sixteen 4-byte values in a vector register, as illustrated in Figure 3 (❶). This allows us to activate only one array to retrieve/store `byte[i]` of all sixteen 4-byte values. For example, we only activate `array[0]` to retrieve/store `byte[0]` of all sixteen 4-byte values (`op[0]`, `op[1]`, …, `op[15]`) from/to a bank. However, we also need to reorder the bytes back to the standard order before sending them to the SIMT pipeline.

After analyzing the baseline GPU architecture, we propose a slight adaptation of the existing crossbar between banks and operand collectors (cf. Section 2.1) to cost-effectively reorder bytes. Figure 4 illustrates a part of an adapted crossbar (write-back path) designed for a hypothetical 2-lane SIMT pipeline and two banks of 2×2-byte vector registers. The gray switches represent switches in the traditional crossbar and the red arrows depict the flow of values. This adaptation simply rearranges the connection points of the crossbar switches, practically incurring no penalty for performance, power or space. Furthermore, bytes of each 4-byte value corresponding to the base value of a vector register are not stored to the register file after compressing the vector register value, and thus they are not sent over the crossbar when retrieved. Consequently, our compression scheme reduces not only the power consumption of accessing vector registers but also that of sending values through a large crossbar. Lastly, we need minor adaptations in the arbiter and control signals of the switches. Nonetheless, we see that such adaptations are insignificant. Note that we always store bytes to a bank in a reordered way whether or not a vector register is compressed in this crossbar architecture. Hence, values stored in registers is oblivious to the compression technique, significantly simplifying the complexity to control the circuit.

In contrast, the register value compression based on BDI [4] requires a relatively complex interconnect network to pack/unpack compressed/decompressed values, as the delta part can have diverse sizes for different registers, such as 1-byte for register 1 and 3-byte for register 2. As the number of bytes for a single compression operation (*e.g.*, a vector register or a cache line) increases, the complexity of the interconnect network increases even more significantly. Note that BDI was originally proposed for cache value compression where a cache line is comprised of 32 bytes [17], whereas a vector register consists of 128 and 256 bytes for NVIDIA and AMD GPUs, respectively [7, 21].

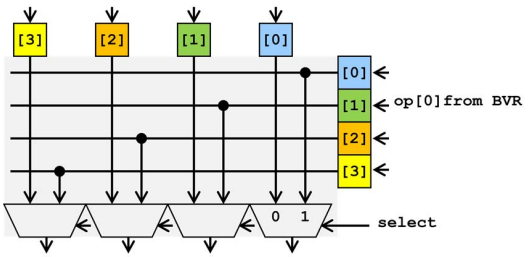

Figure 5: Decompression logic. `select` is the selection signal generated using the four compression bits in `EBR`.

The comparison logic depicted in Figure 3 (❷) generates encoding bits (`enc[3:0]` in Figure 3) that are used not only to determine which array(s) should be activated but also to decompress a compressed vector register value. The comparison logic is almost the same as the logic depicted in prior work [3], but it compares values byte by byte instead of 4-byte word by word. Our circuit-level analysis shows that one cycle is sufficient for the comparison logic to generate its `eq` signal, assuming a typical implementation of all one detector. Then `eq[3:0]` signals are encoded such that `enc[3:0]` bits store 0000_2, (no byte is the same across all 16 4-byte values), 1000_2 (`byte[3]` is the same), 1100_2 (`byte[3:2]` is the same), 1110_2 (`byte[3:1]` is the same), and 1111_2 (`byte[3:0]` is the same or a vector register storing a scalar value). Lastly, we store `enc[3:0]` and the base value of a vector register in an encoding bit register (`EBR` in Figure 3) and a base value register (`BVR` in Figure 3), respectively, as illustrated in Figure 3 (❸); we simply store the first operand value (`op[0]` in Figure 3) of a vector register to a base value register.

To decode sixteen 4-byte operands in the example illustrated in Section 3.1 (*i.e.*, `byte[3:1]` is the same across sixteen 4-byte values), we activate only `array[0]` and a small array storing a 4-byte base value and 4-bit encoding bits (*i.e.*, `BVR` and `EBR`), instead of all four arrays (`array[3:0]`). For a vector register storing a scalar value, we activate only the small array storing the base value and encoding bits. The retrieved bytes from the activated array(s) are reordered back to the standard order after they go through the adapted crossbar. Then the bytes, which are not stored in the register file, are replaced with the corresponding bytes from the base value register by the decompression logic illustrated in Figure 5. In the example depicted in Section 3.1, the decompression logic selects `byte[3]`, `byte[2]`, and `byte[1]` from the base value register, while only `byte[0]` for each operand is from `array[0]` based on the encoding bits ($=1110_2$). For the decompression logic, our conservative circuit-level analysis shows that one cycle is sufficient for the decompression logic. Note that even the baseline GPU deploys some circuits to broadcast a value from the memory pipeline to all SIMT lanes, *e.g.*, when a warp issues a load instruction to the global memory and all threads in the warp attempt to load a value from the same address.

In the example above, a bank for 16-lane SIMT pipelines is comprised of four 16-byte wide arrays. In our experiment, however, a memory compiler synthesizes a bank consisting of eight arrays with 16-byte (= 128-bit) I/O width, when it attempts to satisfy the timing constraint for given capacity (= 8KB), number of ports (= 1), and I/O width (= 1024 bits). That is, we need to use two arrays to store `byte[i]` of all thirty-two 4-byte values of a vector register. Since these two arrays can be activated independently, we can optionally apply our register value compression technique to each half of a vector register separately, denoted by half-register value compression. This increases chances to partially compress more vector registers and support more fine-grain scalar execution for some 32-lane SIMT architectures such as Fermi [7] comprised of 16-lane pipelines at the cost of providing one more set of base value and encoding bit registers per vector register. We will describe how we can exploit this for more fine-grain scalar execution in Section 4.3.

3.3 Handling Branch Divergence

The register value compression technique described in Section 3.1 seamlessly works for non-divergent instructions. For a divergent instruction, only some threads in a warp will be active and they will perform partial writes to a vector register. Consequently, the baseline GPU supports per-word writes to efficiently update vector register values for divergent instructions. For compressed vector registers, however, we cannot perform partial updates unless we decompress it. That is, we always need to retrieve all the values of a vector register before we compress it again, demanding read-modify-write (RMW) operations.

Considering a high performance/energy penalty of such RMW operations, we simply do not encode vector register values for divergent instructions. Although encoding bits are still generated, they are ignored by another bit (denoted by `D` as in "divergent" and affixed to `enc[3:0]`). These encoding bits are used to support scalar execution of divergent instructions described in Section 4.2. Lastly, vector registers, which are already encoded by previous non-divergent instructions, can be still decoded. If a divergent instruction attempts to update values of a compressed vector register, we can employ either hardware- or compiler-assisted techniques.

As a hardware-assisted technique, we can check whether or not the destination vector register of a currently executed instruction is compressed at the scoreboard stage. The scoreboard needs to check the active mask of the instruction and the encoding bits of the destination vector register. If the instruction is divergent and the destination vector register is encoded, the GPU inserts a special register-to-register move instruction retrieving/decompressing the compressed destination vector register value and store it back to the vector register without compressing it. This special move instruction is designed to temporarily ignore the active mask. Prior work reports that the hardware-based technique increases the number of dynamic instructions by only 2% on average [4]. In addition to such a hardware-only approach, a compiler-assisted technique can analyze the lifetime of registers at compile time and identify which registers will store dead values [22]. This information can avoid unnecessary special move instructions. Leveraging such compile-time information, we may further reduce the overhead to less than 2%.

As discussed earlier, a divergent instruction needs to perform partial updates to a vector register. The baseline GPU achieves this by activating write-enable signals associated with active lanes; each array has four write-enable signals for four 4-byte values. Since all 16 bytes of `byte[i]` are stored in one array in our compression technique, we need write-enable signals for each byte in an array. Note that this does not change the number of logical write-enable signals, but each 128-bit wide array needs 16 physical write-enable signals. Analyzing circuit implementations of the write data-path of arrays, we discover that providing 16 write-enable signals do not require any change in the SRAM core. That is, we only need to connect more write-enable signals to write-drivers of the SRAM I/O peripheral block in a finer-grained way. Our circuit-level analysis shows that more write-enable wires increase the area of a large memory array by less than 1%. This is because the I/O circuit has a tight width pitch but a loose height pitch where the write-enable signals run horizontally. Lastly, our compression technique activates all four arrays for a divergent instruction partially updating its destination register, as such a partial update is applied to a decoded vector register value, and each byte of a 4-byte value is distributed across four arrays. In contrast, the baseline architecture may activate fewer arrays for a partial update depending on a given active mask value (`M`). This effect will be accounted for our evaluation.

4. G-Scalar Architecture

Prior work demonstrated that scalar execution of eligible non-divergent arithmetic/logic instructions could significantly improve the performance and power efficiency of GPUs [3, 5]. In this section, leveraging the enhanced microarchitecture for our register value compression, we demonstrate that G-Scalar can support scalar execution of eligible divergent and special-function instructions. This greatly increases the percentage of scalar execution of instructions practically at no cost.

4.1 Efficient Scalar Execution

Our register value compression technique not only reduces the energy consumption of register file and its crossbar but also easily determines how similar the values of a vector register are. A vector register storing a scalar value has its encoding bits (`enc[3:0]`) equal to 1111_2. This allows us to easily support a scalar execution approach proposed by prior work [3] at practically no further hardware cost, because a base value register becomes effectively a scalar register. When a non-divergent instruction operates on two vector registers storing scalar values, only two 4-byte values are sent from the corresponding base value registers to an appropriate SIMT pipeline and only one SIMT execution lane will be active. Similar to prior work [3], scalar execution of an instruction only needs to write-back its computed value to a base value register and sets the encoding bits of the destination vector register to 1111_2. Lastly, when at least one of vector registers for an instruction is not a vector register storing a scalar value, the instruction is not eligible for scalar execution, and the decompression logic automatically decompresses vector registers storing scalar values.

Although the key high-level concept of identifying and executing scalar instructions is the same as previous scalar execution architectures [3, 5, 6], G-Scalar has notable advantages over previous scalar execution architectures. First, G-Scalar effectively provides 16 banks for scalar values because each bank has its own small array for base value registers. In contrast, we observe that a single bank for scalar values in prior scalar execution architectures can be a performance bottleneck for applications with many instructions eligible for scalar execution. An instruction can access all of its vector registers in parallel if there is no bank conflict. However, it always takes multiple cycles for an instruction eligible for scalar execution to retrieve its register values because there is only one bank for scalar values. Furthermore, when one warp issues a scalar instruction, we observe that other warps are also likely to issue scalar instructions. This is because the warps are executed roughly at the same pace. Therefore, there can be a burst of scalar instructions, all of which wait for accessing a single bank for scalar values at the operand collectors. If all operand collectors are allocated, the warp schedulers must stop issuing instructions from ready warps. With semiconductor technology scaling, future GPUs also tend to have more hardware resources, such as larger register file with more banks and more SIMT execution pipelines. Thus, relying on only a single bank for scalar values may not be a scalable approach.

Second, G-Scalar does not need separate scalar execution pipelines. Instead, it leverages existing SIMT execution pipelines and clock-gates all but one lane for scalar execution [2]. Note that prior work estimates that the cost of supporting per-lane clock gating is very small [2]. We identify a few reasons to obviate from implementing separate scalar execution pipelines. Since the front-end can schedule and issue only up to two instructions per cycle, a separate scalar execution pipeline may bring only little performance improvement because the front-end will soon become a performance bottleneck. Furthermore, implementing even one more SFU incurs a considerable penalty of chip power and space. For example, a GTX 480 GPU has only four SFUs per SM because each SFU consumes a large amount of chip power and space. This is why a previous architecture implements the scalar execution pipeline only for arithmetic/logic instructions. In contrast, G-Scalar can support scalar execution for any vector instructions, because it uses SFUs that are already in the SIMT pipelines.

4.2 Scalar Execution of Divergent Instructions

No prior scalar execution architecture can support scalar execution of divergent instructions [3, 5, 6, 8]. However, we observe that approximately 50% of executed instructions are divergent in some GPU applications such as `lbm` [23] and `heartwall` [24]. Meanwhile, we discover that 4-byte values, which are associated with active lanes in a divergent path, are often the same in a vector register. Suppose that a vector register has 8 values, `AAABAABC`. We cannot consider that such a vector register stores a scalar value for a non-divergent instruction. Nonetheless, we may consider that the vector register stores a scalar value for a divergent instruction with an active mask ("`M`") value equal to `10101100` (*i.e.*, `A-A-AA--`), as shown in Figure 6. To support scalar execution for such

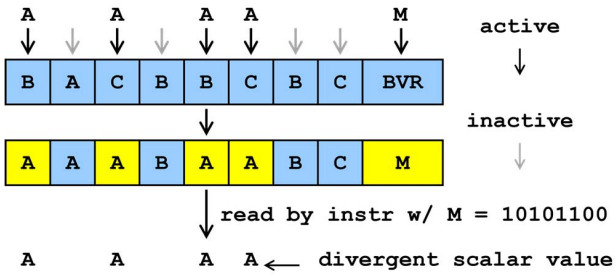

Figure 6: An example for divergent scalar value. `M` denotes active mask. A vector register of 8 values is updated partially first, then read by another divergent instruction.

divergent instructions, we need to slightly adapt the comparison logic to correctly compare the (partial) write-back values of a vector register updated by a divergent instruction and detect whether a given divergent instruction is eligible for scalar execution.

As depicted in Figure 3 (❷), our comparison logic for compression compares each byte of a 4-byte write-back value to that of a 4-byte write-back value in its neighboring lane. Such an implementation makes the comparison of partial write-back values impossible for a divergent instruction, since the inactive lanes of a divergent instruction do not have any valid write-back values. This breaks the comparison chain in the comparison logic. However, we tackle this challenge based on the following observation: providing a 4-byte value from any active lane for values of inactive lanes should not change the outcome of comparison, because we only check whether or not all active lanes have the same write-back value. Suppose that a given SIMT execution pipeline has four lanes, while `lane[0]`, `lane[1]`, and `lane[3]` are active. The comparison of `op[0]`, `op[1]`, and `op[3]` is equivalent to that of `op[0]`, `op[1]`, `op[0]`, and `op[3]`. We can accomplish such a comparison by slightly adapting the comparison logic illustrated in Figure 3 (❷).

Figure 7(a) shows the adapted comparison logic. We have a shared byte-wide bus that can be driven by a write-back value from only one of active lanes. The logic detecting the leading one of a given 16-bit active mask value (`M`) (*e.g.*, `L[15:0]` = 0100000000000000_2 for `M[15:0]` = 0100000011111010_2) can enable only one tristate buffer connected to the write-back value from the first active lane (*i.e.*, `lane[14]`). This allows us to send a write-back value from an active lane to all inactive lanes just for a comparison purpose and check whether or not active lanes operate on the same 4-byte value. Since the comparison logic has enough timing slack (cf. Section 3.1), the adapted comparison logic can still complete a comparison in one cycle according to our circuit-level analysis.

As described in Section 3.3, we do not compress the destination register of a divergent instruction. Nonetheless, we still generate its encoding bits and store them to its encoding bit register. This is to indicate whether or not 4-byte values, which correspond to active lanes of subsequent divergent instructions, are the same (*i.e.*, a vector register of a divergent scalar value). For example, `if(r1==r2)` in Figure 7(b) (❶) starts a branch divergence; Figure 7(b) (❷) and (❸) illustrate two following divergent paths. Since `r2=r2*2` stores a (divergent) scalar value with respect to a given active mask value in Figure 7(b) (❷), the corresponding encoding bit register (`EBR(r2)`) is set to 1111_2 indicating that `r2` stores a divergent scalar value. However, `r2` stores a scalar value only with respect to the current active mask value (`M=10001111`). The following instruction (`r1=abs(r2)`) in Figure 7(b) (❸) is on the other divergent path and operates on `r2`. Although `EBR(r2)` indicates that `r2` contains a scalar value, we cannot perform scalar execution for `r1=abs(r2)` because the encoding bits of `r2` are invalid with respect to the current active mask (`M=01110000`). From this example, we can see that the operand values in the active lanes (`-2`, `0`, `1`) are not the same indeed.

To correctly determine whether or not a current instruction operates on vector registers of divergent scalar values with respect to its active mask, we need to remember which lanes were compared to generate the corresponding encoding bits (*i.e.*, the active mask of the previous instruction that wrote its values to the register). Since we do not encode values of a divergent destination register, we do not need to store its base value to the corresponding base value register. Exploiting this, we propose to store the associated active mask to its base value register (Figure 7(b) (❷)).

Depending on whether or not a register is updated by a non-divergent or divergent instruction (`D` = `0` or `1`), the interpretation of `enc[3:0]` may change and the corresponding base value register may store a base or active mask value. When `D` is set to `1`, we do not actually compress register values. Consequently, we ignore the corresponding encoding bits

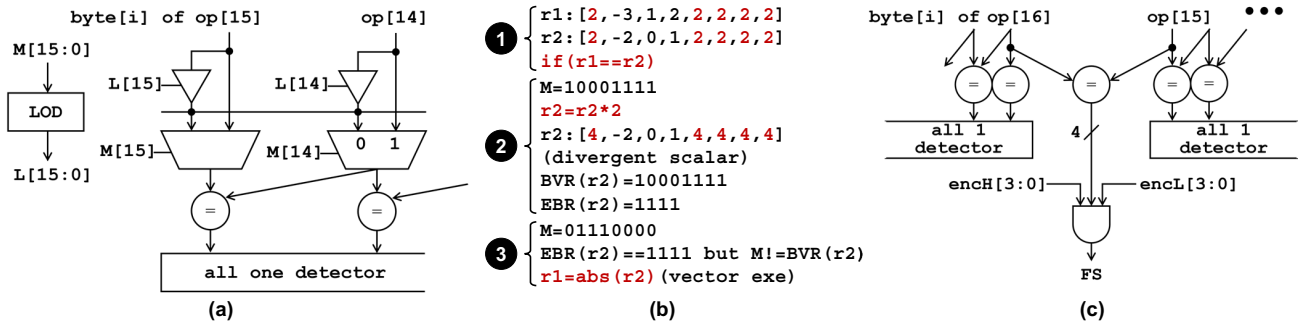

Figure 7: (a) The adapted comparison logic for divergent instructions. (b) An example of checking active mask (denoted by `M`). (c) The adapted comparison logic for half-warp scalar execution. `FS` **denotes "full scalar".**

and bring all values from the register file. However, when D and enc[3:0] are set to 1 and 1111_2, respectively, we compare the active mask value of a current instruction with the value of a base value register corresponding to a vector register that the instruction operates on (Figure 7(b) (❸)). If these two values are matched, the source vector register contains a (divergent) scalar value. Note that we still need to bring all 16 operand values from the register file, but we activate only one lane for scalar execution. Otherwise, we cannot perform scalar execution for the instruction although enc[3:0] is set to 1111_2.

Lastly, scalar execution of divergent instructions is the same as non-divergent scalar instructions (*i.e.*, only one lane is active), but retrieving/storing values from/to the register file is different. As we disable the register value compression for divergent instructions, we store a scalar value to the register file without compressing the register value. In particular, we leverage an existing mechanism depicted in Figure 3 to broadcast a divergent scalar value to the write paths associated with all active lanes. Although a source vector register stores the same value with respect to a given active mask, we still retrieve the value from the register file.

4.3 Half-warp Scalar Execution

As described in Section 3.1, we can optionally compress each half of a vector register separately, providing one more pair of base value and encoding bit registers for each vector register. Leveraging a half-warp execution architecture in some GPUs [7] and our half-register value compression technique, we can support half-warp scalar execution. Suppose that enc[3:0] of the first half of a vector register (denoted by encL[3:0]) is 1100_2 but enc[3:0] of the second half of the vector register (encH[3:0]) is 1111_2. In previous scalar execution architectures, we cannot support scalar execution of such an instruction, as some values of the first half of the vector register are not the same. In contrast, we can still support scalar execution of such an instruction for the second-half warp.

In some occasions, the first and second halves of a vector register are scalar each, but they have two distinct scalar values. In contrast to traditional scalar execution architectures, G-Scalar can support scalar execution for each half warp. To indicate whether or not the first and second halves have the same value, we need another flag bit (FS in Figure 7(c)); the modified comparison logic for this architecture is depicted in Figure 7(c). This is necessary for scalar execution of a full-warp using only one lane for all 32 threads.

Note that the enhanced microarchitecture for half register value compression can seamlessly handle the write-back from scalar execution of a half warp. Moreover, it allows us to support half-warp scalar execution at practically no further hardware cost. Considering the complexity and benefit, we support half-warp scalar execution only for non-divergent instructions in this study. Lastly, this half-warp scalar execution allows even future GPUs with wider SIMT pipelines (*i.e.*, fewer full-warp scalar instructions) to continuously benefit from scalar execution. However, implementing one more set of base value and encoding bit registers increases the hardware cost of the register file from 3% to 7%.

5. Evaluation

5.1 Methodology

We use GPGPUSim 3.2.2 [25] and GPUWattch [2] to evaluate the effectiveness of G-Scalar and our register value compression technique. GPGPUSim is configured to model a GPU architecture similar to NVIDIA GTX480 [7]. The key configuration parameters are tabulated in Table 1. GPUWattch is also configured to estimate the power consumption of G-Scalar GPU.

We use 17 benchmarks from Parboil [23] and Rodinia [24] benchmark suites that represent diverse GPU applications (cf. Table 2). Although one of our key contributions is scalar execution of divergent instructions, we exclude exceptionally divergent benchmarks (*e.g.*, myocyte), while including a fair number of non-divergent benchmarks (*e.g.*, mri-q, sgemm, spmv). Lastly, we exclude unusually memory-intensive benchmarks (*e.g.*, bfs), as the performance and power consumption are dominated by the memory subsystem, whereas scalar execution architecture aims to improve power efficiency of compute-intensive applications.

Figure 8 plots value similarity of vector registers in these benchmarks. We collect values of vector registers by executing the benchmarks. We compare all thirty-two 4-byte values of each vector register byte by byte. "*n*-byte" denotes that the first *n* most significant bytes of all 4-byte values in a vector register are the same. As seen in Figure 8, the average percentage of (non-divergent) scalar, 3-, 2-, and 1-byte categories are 36%, 17%, 4%, and 7%, respectively.

We synthesize the compressor and decompressor logic with a commercial 40nm standard cell library. The results shown in Table 3 include the additional 1024-bit pipeline registers for compressor and decoder each. Since we have one

Table 1: Simulator configuration.

# of SMs	15	Registers per SM	128KB
SM Frequency	1.4GHz	Register File Bank	16
NoC Frequency	0.7GHz	OC per SM	16
Warp Size	32	Schedulers per SM	2
SIMT EXE Width	16	L1$ per SM	16KB
Threads per SM	1536	Memory Channels	6
CTAs per SM	8	L2$ Size	768KB

Table 2: Benchmarks.

Rodinia		Parboil	
Benchmark	**Abbr.**	**Benchmark**	**Abbr.**
b+tree	**BT**	cutcup	**CC**
backprop	**BP**	lbm	**LBM**
heartwall	**HW**	mri-grid	**MG**
hotspot	**HS**	mri-q	**MQ**
leukocyte	**LC**	sad	**SAD**
pathfinder	**PF**	sgemm	**MM**
srad_1	**SR1**	spmv	**MV**
srad_2	**SR2**	stencil	**ST**
		tpacf	**ACF**

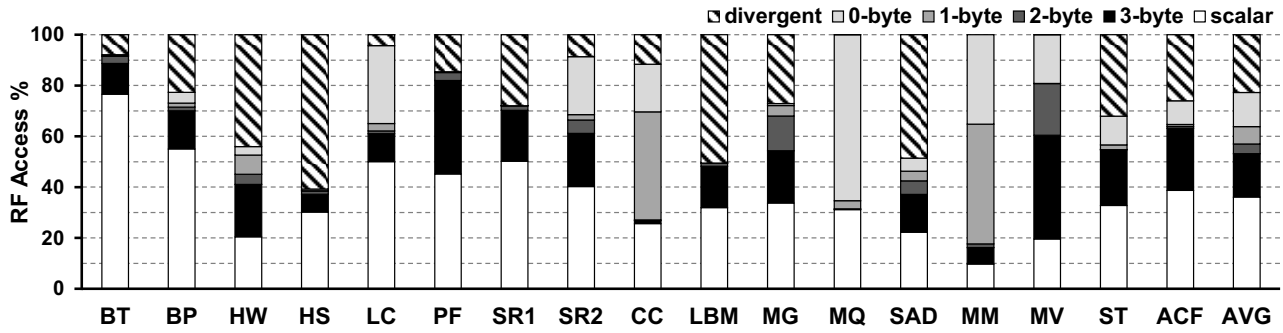

Figure 8: RF access distribution for operand values. "scalar" means all the operands are identical and "*n*-byte" means the first *n* MSBs of the operand values are the same. "divergent" means the operands are accessed by a divergent instruction.

Table 3: Estimation of encoder/decoder area, delay and power consumption including that of the pipeline registers at 1.4GHz. The encoder includes the broadcasting logic depicted in Figure 7.

	Decompressor	Compressor
Area (μm^2)	7332	11624
Delay (ns)	0.35	0.67
Power (mW)	15.86	16.22

decompressor for each operand collector and one compressor for each SIMT execution pipeline, we need 16 decompressor and 4 compressor per SM. These compressor and decompressor increase the chip power and space by 0.32W (1.6%) and 0.16mm^2 (0.7%) respectively, compared to the baseline SM. Considering the small cost of chip power and space, we do not place-and-route the compressor and decompressor implementations, as the conclusion of this study will not notably changes. Lastly, as our compression technique is simpler than BDI, our compressor logic associated wires consume only 19% and 30% of prior work [4].

In order to estimate the access energy of register file, we use a memory compiler, which determines the size of an array constituting a 64×1024-bit bank for given capacity (= 8KB), number of ports (= 1), and I/O width (= 1024 bits). This gives us a bank comprised of 8×128-bit arrays with 128-bit I/O width, agreeing to the size used for prior work [26]. For 32-bit base value registers, 4-bit encoding bit registers, and `D` and `FS` bits, we synthesize a 64×38-bit array. The energy consumption of accessing a 38-bit register in this array is 5.2% of that of accessing an entire 1024-bit vector register in a bank. This register file architecture needs practically no modification to the baseline register file architecture except for only extra write-enable signals. The memory array for base value registers, encoding bit registers, and `D` and `FS` bits increases the size of register file by ~3%.

Lastly, the encoding bits should be known before reading the register file to determine which arrays should be activated. This increases the pipeline latency by one cycle, but the throughput of register file is not affected as we pipeline such an operation. In total, we increase the GPU pipeline latency by 3 cycles (*i.e.*, 1 cycle each for (1) compressing a register value; (2) decoding a register value; and (3) accessing a base value register, encoding bit register and flag bits).

5.2 Instructions Eligible for Scalar Execution

Figure 9 shows the percentage of instructions eligible for scalar execution. "ALU scalar" is the baseline architecture, which only supports scalar execution of non-divergent arithmetic/logic instructions. "all scalar" covers special-function and memory (load/store) instructions eligible for scalar execution atop "ALU scalar." "half-scalar" includes instructions eligible for half-warp scalar execution atop "all scalar." Finally, "divergent scalar" includes divergent instructions eligible for scalar instructions atop "half-scalar." "ALU scalar" covers only 22% of total dynamic instructions on average. Atop "ALU scalar," we can cover 7%, 2%, and 9% more instructions when we cover scalar execution of special-function, memory, half-warp, and divergent instructions, respectively. Compared with "ALU scalar," G-Scalar can almost double the number of instructions eligible for scalar execution, increasing the number of scalar instructions to 40%.

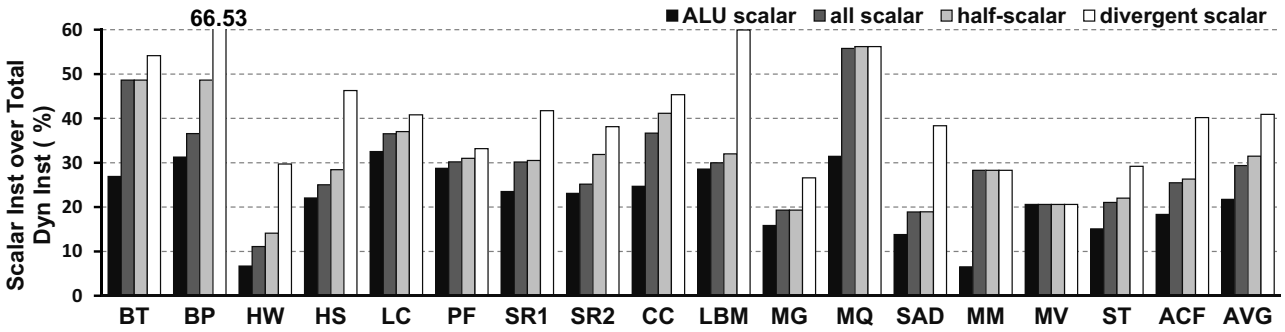

Figure 9: Percentage of instructions eligible for scalar execution.

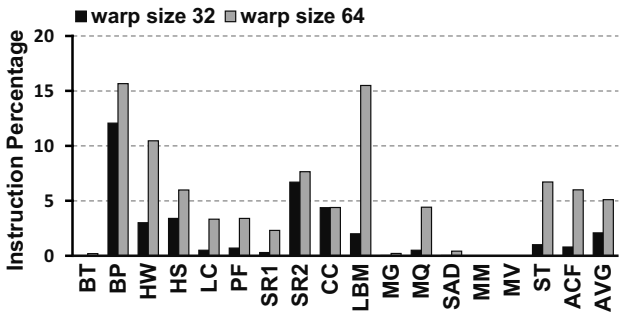

Figure 10: Percentage of instructions eligible for "half-scalar" execution for different warp sizes. For warp size of 64, we keep the same 16-thread checking granularity so it should actually be "quarter-scalar" instruction.

Although special-function instructions contribute to only 3% of total dynamic instructions eligible for scalar execution, they consume 3~24× more energy than other floating-point instructions [2], and SFUs contribute up to 60% of the power consumption of GPUs in some benchmarks [27]. Therefore, supporting scalar execution for special-function instructions can considerably reduce GPU power consumption.

For a memory instruction eligible for scalar execution, all the threads in a warp attempt to access a value at the same address. That is, memory instructions eligible for scalar execution cannot improve the performance of the memory system, as the memory pipeline already has logic to coalesce memory requests from multiple threads in a warp. Nonetheless, scalar execution of memory instructions can still reduce power consumption of computing target memory addresses.

The instructions eligible for half-warp scalar execution contribute to 12% of total executed instructions for `BP`. Although half-warp scalar execution is optional, it shows that the number of scalar instructions also depends on the granularity of checking vector registers storing scalar values. That is, if we increase the warp size to 64 and keep the 16-thread checking granularity, the average number of "half-scalar" ("quarter-scalar" in this case) will increase to 5% as shown in Figure 10. Note that 64-thread warps are supported by AMD GPUs, which execute 64-thread wavefront instructions using 16-lane SIMD pipelines in the Graphics Cores Next (GCN) architecture [21]. For some benchmarks, the number of instructions eligible for "half-scalar" execution increases significantly. One reason is that two scalar instructions with different operand values (from 32-thread warps) may be organized into one instruction from a 64-thread warp. Therefore, such instructions from those 64-thread warps are no longer scalar instructions but become "half-scalar" instructions. If the warp size further increases in the future, the half-scalar execution can be attractive to maintain the benefit of scalar execution.

The number of divergent scalar instructions depends on the total number of divergent instructions. Intuitively, benchmarks with many divergent instructions have more instructions eligible for divergent scalar execution. For example, `HS`, `LBM` and `SAD` have 17%, 30% and 19% divergent scalar instructions, respectively. Especially for `LBM`, supporting divergent scalar instructions can double the number of instructions eligible for scalar execution, compared with the previous scalar execution architectures.

5.3 Power Efficiency Improvement

Figure 11 shows the power efficiency in terms of normalized IPC/W. In "ALU scalar," a scalar register file is also used to reduce power consumption of register file. When all of our proposed techniques are combined, we can improve the power efficiency by 24% on average. Compared with "ALU scalar," G-Scalar further improves the power efficiency by 15%. Amongst all the benchmarks, `BP` shows very high (79%) power efficiency improvement. `BP` is very compute-intensive, and the total power consumption of the GPU is over 100W from GPUWattch [2]. Over 50% of the GPU power is consumed by execution units and register files. Especially, SFUs alone consume more than 25% of the total power although only 14% of total dynamic instructions are SFU instructions. With a very high percentage of special-function scalar instructions in `BP` (~60%), we can significantly reduce the power consumption (43%). The SFU power consumption is reduced to less than 10% of the baseline architecture. One of the reasons is that each thread of `BP` needs to compute 2.0 to the n^{th} in floating-point for n iterations. G-Scalar can execute such instructions as scalar instructions. However, some benchmarks show significant improvement in the number of scalar instructions but their efficiency improvement is not proportional. For example, more than 40% of total dynamic instructions are eligible for scalar execution in `LBM`, but the power efficiency improvement is less than 20%. The primary reason

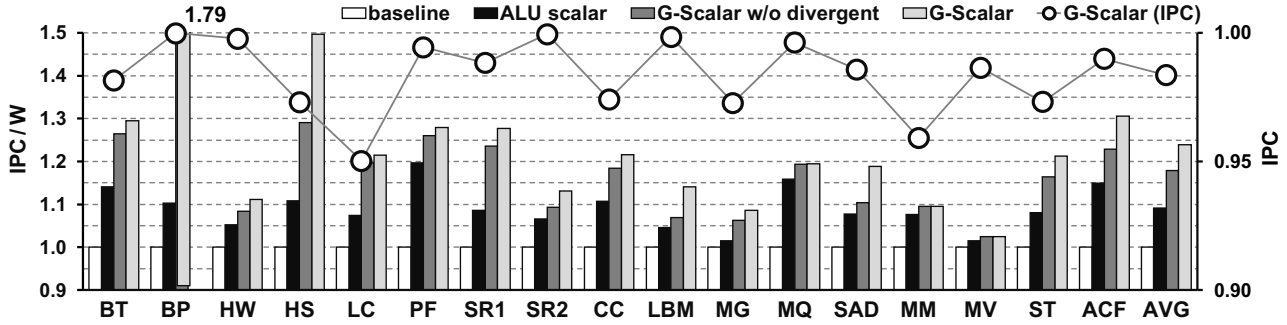

Figure 11: Normalized GPU power efficiency and performance. "ALU Scalar" and "G-Scalar w/o divergent" enable scalar execution on ALU pipeline and all three types of pipelines, respectively. "G-Scalar" extends scalar execution to half-scalar and divergent-scalar instructions. "G-Scalar (IPC)" shows the performance impact of adding 3 cycles of latency in normalized IPC.

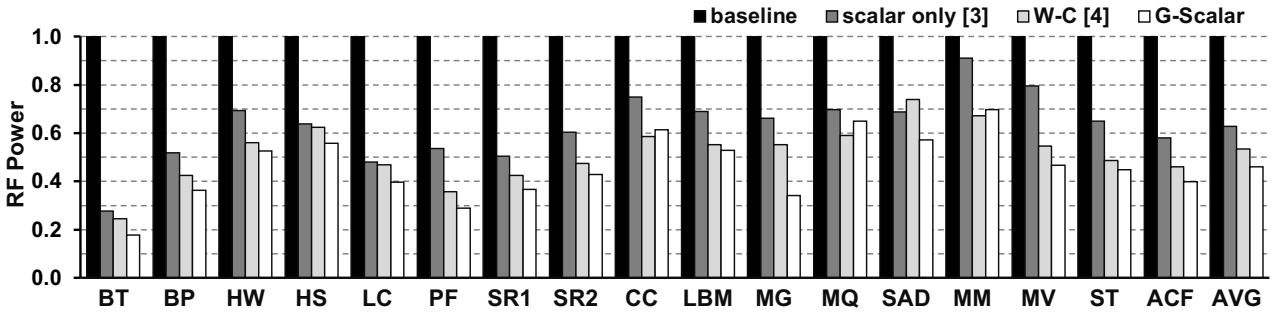

Figure 12: Normalized RF dynamic power consumption. "scalar only" is a scalar RF technique [3] and "W-C" is a warped-compression technique [4].

is that those benchmarks are rather memory-intensive. A significant fraction of their dynamic power is consumed by memory subsystems, such as L2 cache, memory controller and DRAM.

Exploiting value similarity in a vector register, we show the power consumption of register file in Figure 12. More specifically, we compare the power consumption of our register value compression technique with that of scalar-only register file [3] and recent register value compression techniques [4]. The scalar register file consumes 37% less power than the baseline register file, whereas our register file consumes 54% less power on average. That is, our register file consumes about 17% less power than the scalar register file (46% vs 63%). In some benchmarks such as `MG` and `MV`, where there are relatively fewer scalar values and many 3-byte and 2-byte accesses, our register value compression technique can reduce the power consumption of register file by more than 40% compared with the scalar register file technique. Our register compression scheme also reduces more power consumption than the prior register value compression technique [4] at a lower penalty of chip power and space. The chips space required by our compression technique is only 52% of the chip space demanded by the prior compression scheme [4] according our logic synthesis of both compression techniques. Lastly, as our compression technique uses a separate small array to store base values, it often activates one fewer array of register file than the prior compression technique for the same compression ratio, and the average compression ratio of our compression technique is 2.17, whereas that of BDI is 2.13 when the same input sets for the benchmarks are used. Furthermore, the simplicity of our compression technique in fact facilitates G-Scalar execution at practically no further cost.

In our experiment, we use 32-bit unsigned integers for all address computations. However, recent GPUs began to support larger DRAM capacity. Consequently, GPUs need to use 64-bit integers for address computations if they support the DRAM capacity larger than 4GB. In this case, we can obtain more power reduction with our register value compression technique. As mentioned earlier, it is very likely that only few LSBs are different for addresses in a warp. If the addresses are 64-bit, we can have more bytes with the same value and thus more power reduction. For data types that are smaller than 4 bytes (short integer and character types), our scheme can at least avoid the unnecessary access to the sign/zero extended bytes.

Lastly, as the current trend deploys a larger register file for GPUs and uses 8- and 10-transistor memory cells to tolerate every-increasing process variations [28], the energy consumption of register file is to further increase. This can justify adoption of a register value compression technique due to power and thermal constraints that do not scale well, which in turn can support G-Scalar practically at no cost.

5.4 Performance Impact

As we described in Sections 3.2 and 5.1, we increase the latency of pipelines by 3 cycles to account for cycles for reading encoding bits, decompressing and compressing a vector register. Figure 11 also shows the performance impact of G-Scalar after increasing the latency of baseline pipelines by 3 cycles. On average, the performance degradation is only 1.7%. The additional latency primarily increases the chance of pipeline stall due to data dependency. Although there is no data bypassing in GPUs, a large number of active warps can still effectively hide this latency [29]. Amongst all the evaluated benchmarks, `LC` shows the most significant performance degradation because it has an insufficient number of warps to effectively hide latency while utilizing many long latency instructions (*e.g.*, integer `DIV`). These two factors together make `LC` more sensitive to the 3-cycle latency increase, but still shows more than 20% IPC/Watt improvement.

6. Related Work

Scalar execution. Prior work proposed scalar execution architectures [3, 5, 6]. Xiang *et al.* proposed to exploit inter-warp scalar opportunity [5]. Yilmazer *et al.* further proposed to group more than one scalar instructions together and execute them in an execution pipeline in a batch [30]. However, all these scalar execution architectures only exploit scalar opportunity amongst non-divergent instructions. In contrast, we extend the concept of scalar instructions to divergent instructions.

A scalar unit is also supported by AMD GPUs, but they are to manage and optimize control flows operating on execution mask and vector conditional code [31]. Wong *et al.* proposed to apply approximation techniques by exploiting value similarity and relaxing the constraints for scalar execution [8]. The scalar execution can improve performance by executing scalar instructions in fewer cycles than vector instructions [5].

In NVIDIA Fermi and AMD GCN architectures [7, 21], a warp is executed over multiple cycles due to the limited width of SIMT execution pipelines. For example, it takes at least 8 cycles to execute a special-function instruction. If the instruction is executed as a scalar instruction, it takes as low as only one cycle to execute. Lastly, since G-Scalar can almost double the number of instructions eligible for scalar execution, other architectures (*e.g.*, [18]), which exploit such instructions to improve performance and/or reduce power consumption, can benefit from G-Scalar.

Lee *et al.* proposed to analyze instructions eligible for scalar executions at compile-time [31]. However, there are several drawbacks using such a compiler-assisted method. First, a compiler-assisted method is limited by compile-time information and cannot exploit value similarity originated from executing load instructions. Most instructions with high dynamic power consumptions operate on values that are loaded from the memory and thus unavailable at the compile-time. Running AAA game titles (a classification term in game industry) using an in-house simulator for AMD GPUs, we observe that such a compiler-assisted method captured 24% less instructions eligible for scalar execution than G-Scalar. Second, such a compiler-assisted method is purely for code analysis. Therefore, microarchitectural and/or architectural modifications are required to support scalar execution. Therefore, we believe G-Scalar is orthogonal to such methods. Collange *et al.* proposed methods to dynamically detect uniform and affine vectors in GPUs [32], which is tag-based and limited to some of the registers. Kim *et al.* proposed affine execution unit to explore the affine instructions [33]. However, it is only applicable to limited type of instructions.

Reducing power consumption of register files. Gebhart *et al.* proposed a register file caching technique for GPUs to reduce the power consumption of register file by reducing the number of register file accesses. It utilizes a small per-thread cache to store frequently used register values [22]. Lee *et al.* proposed a BDI-based register file compression technique [4]. Our register file compression technique not only reduces the power consumption of register file but also that of crossbar and execution units, reducing the number of bytes required per register file access. Gilani *et al.* proposed to store scalar values separately in a much smaller and more power-efficient scalar register file [3]. We use base value registers and reuse the existing hardware resources of GPUs to exploit varying degrees of value similarity (1-byte, 2-bytes, 3-bytes or scalar) across threads.

7. Conclusion

As GPUs began to support more general-purpose applications, the fraction of divergent instructions in contemporary GPU applications has significantly increased. In this paper, we demonstrate that (1) many divergent instructions are also eligible for scalar execution if we consider only operand values in active lanes in a divergent path and (2) special-function instructions, many of which are also eligible for scalar execution, consumes a large fraction of execution power. However, prior scalar execution architectures cannot support scalar execution of these instructions. Tackling such limitations of prior scalar execution architectures, we propose G-Scalar, generalized scalar execution architecture along with a low-cost register value compression technique. G-Scalar can support scalar execution of not only conventional non-divergent arithmetic/logic but also divergent and special-function instructions. Furthermore, if GPUs adopt our low-cost register value compression technique, G-Scalar is practically free, as it is architected to (1) share most of hardware resources with our register value compression technique and (2) reuse existing hardware resources of SIMT execution pipelines for scalar execution instead of implementing dedicated scalar execution pipelines. Our evaluation shows that G-Scalar, which consumes only 1% more chip space than the baseline GPU, can double the number of instructions eligible for scalar execution. This in turn improves power efficiency of GPUs by 24% and 15% compared with the baseline and previous scalar execution architectures, respectively. Lastly, our register value compression technique alone can reduce the power consumption of register file by 54%.

Acknowledgement

This work is supported in part by grants from NSF (CCF-1600986 and CNS-1600669). Nam Sung Kim has a financial interest in AMD and Samsung Electronics.

References

[1] "The International Technology Roadmap for Semiconductors," [Online]. Available: http://www.itrs.net/.

[2] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt and V.J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in International Symposium on Computer Architecture (ISCA), 2013.

[3] S.Z. Gilani, N.S. Kim and M.J. Schulte, "Power-Efficient Computing for Compute-Intensive GPGPU Applications," in International Symposium on High Performance Computer Architecture (HPCA), 2013.

[4] S. Lee, K. Kim, G. Koo, H. Jeon, W.W. Ro and M. Annavaram, "Warped-Compression: Enabling Power Efficient GPUs through Register Compression," in International Symposium on Computer Architecture (ISCA), 2015.

[5] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. Hsu and H. Zhou, "Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement," in International Conference on Supercomputing (SC), 2013.

[6] Y. Yang, P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong and H. Zhou, "A Case for a Flexible Scalar Unit in SIMT Architecture," in International Parallel and Distributed Processing Symposium (IPDPS), 2014.

[7] "Fermi Architecture Whitepaper," Nvidia, [Online]. Available:http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[8] D. Wong, N.S. Kim and M. Annavaram, "Approximating Warps with Intra-warp Operand Value Similarity," in

International Symposium on High Performance Computer Architecture (HPCA), 2016.

[9] T.G. Rogers, D.R. Johnson, M. O'Connor and S.W. Keckler, “A variable warp size architecture,” in International Symposium on Computer Architecture (ISCA), 2015.

[10] B. Coutinho, D. Sampaio, F.M.Q. Pereira and W. Meira Jr., “Divergence Analysis and Optimizations,” in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011.

[11] N. Brunie, S. Collange and G. Diamos, “Simultaneous Branch and Warp Interweaving for Sustained GPU Performance,” in International Symposium on Computer Architecture (ISCA), 2012.

[12] A. ElTantawy, J.W. Ma, M. O’Connor and T.M. Aamodt, “A Scalable Multi-path Microarchitecture for Efficient GPU Control Flow,” in International Symposium on High Performance Computer Architecture (HPCA), 2014.

[13] W.W.L. Fung, I. Sham, G. Yuan and T.M. Aamodt, “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” in International Symposium on Microrchitecture (MICRO), 2007.

[14] M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation,” in International Symposium on Computer Architecture (ISCA), 2013.

[15] M. Rhu and M. Erez, “The Dual-Path Execution Model for Efficient GPU Control Flow,” in International Symposium on High Performance Computer Architecture (HPCA), 2013.

[16] S. Liu, J.E. Lindholm, M.Y. Siu, B.W. Coon and S.F. Oberman, “Operand Collector Architecture.” US Patent 7,834,881 B2, 06 11 2010.

[17] G. Pekhimenko, V. Seshadri, O. Mutlu, P.B. Gibbons, M.A. Kozuch and T.C. Mowry, “Base-Delta-Immediate Compression: Practical Data Compression for On-chip Caches," in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2012.

[18] J. Zhan, M. Poremba, Y. Xu and Y. Xie, "NoΔ: Leveraging Delta Compression for End-to-End Memory Access in NoC Based Multicores," in Asia and South Pacific Design Automation Conf. (ASP-DAC), 2014.

[19] G. Pekhimenko, T.C. Mowry and O. Mutlu, “Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency,” in International Symposium on Microrchitecture (MICRO), 2013.

[20] D. J. Palframan, N.S. Kim and M.H. Lipasti, “COP: To Compress and Protect Main Memory,” in International Symposium on Computer Architecture (ISCA), 2015.

[21] “AMD Graphics Cores Next (GCN) Architecture White Paper,” Advanced Micro Devices, [Online]. Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[22] M. Gebhart, D.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm and K. Skadron, “Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors,” in International Symposium on Computer Architecture (ISCA), 2011.

[23] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu and W.-m. W. Hwu, “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, 2012.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee and K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing,” in International Symposium on Workload Characterization (IISWC), 2009.

[25] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong and T.M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009.

[26] J. Lim, N.B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili and W. Sung, "Power Modeling for GPU Architectures Using McPAT," ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 19, no. 3, 06 2014.

[27] H. Zhang, M. Putic and J. Lach, "Low Power GPGPU Computation with Imprecise Hardware," in Design Automation Conference (DAC), 2014.

[28] H. Yamauchi, "A Discussion on SRAM Circuit Design Trend in Deeper Nanometer-Scale Technologies," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 18, no. 5, pp. 763 - 774, 2009.

[29] S.-Y. Lee and C.-J. Wu, "Characterizing the Latency Hiding Ability of GPUs," in International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.

[30] A. Yilmazer, Z. Chen and D. Kaeli, "Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs," in International Parallel and Distributed Processing Symposium (IPDPS), 2014.

[31] Y. Lee, R. Krashinsky, V. Grover, S.W. Keckler and K. Asanović, "Convergence and Scalarization for Data-Parallel Architectures," in International Symposium on Code Generation and Optimization (CGO), 2013.

[32] S. Collange, D. Defour and Y. Zhang, "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations," in Euro-Par 2009: Parallel Processing Workshops, 2009.

[33] J. Kim, C. Torng, S. Srinath, D. Lockhart and C. Batten, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," in in International Symposium on Computer Architecture (ISCA), 2013.