Software-Directed Techniques for Improved GPU Register File Utilization

DANI VOITSECHOV, Technion

ARSLAN ZULFIQAR, MARK STEPHENSON, MARK GEBHART, and

STEPHEN W. KECKLER, NVIDIA

Throughput architectures such as GPUs require substantial hardware resources to hold the state of a massive number of simultaneously executing threads. While GPU register files are already enormous, reaching capacities of 256KB per streaming multiprocessor (SM), we find that nearly half of real-world applications we examined are register-bound and would benefit from a larger register file to enable more concurrent threads. This article seeks to increase the thread occupancy and improve performance of these register-bound applications by making more efficient use of the existing register file capacity. Our first technique eagerly deallocates register resources during execution. We show that releasing register resources based on value liveness as proposed in prior states of the art leads to unreliable performance and undue design complexity. To address these deficiencies, our article presents a novel compiler-driven approach that identifies and exploits last use of a register name (instead of the value contained within) to eagerly release register resources. Furthermore, while previous works have leveraged “scalar” and “narrow” operand properties of a program for various optimizations, their impact on thread occupancy has been relatively unexplored. Our article evaluates the effectiveness of these techniques in improving thread occupancy and demonstrates that while any one approach may fail to free very many registers, together they synergistically free enough registers to launch additional parallel work. An in-depth evaluation on a large suite of applications shows that just our early register technique outperforms previous work on dynamic register allocation, and together these approaches, on average, provide 12% performance speedup (23% higher thread occupancy) on register bound applications not already saturating other GPU resources.

CCS Concepts: • **Computing methodologies** → **Graphics processors**; • **Computer systems organization** → **Parallel architectures**; **Embedded hardware**;

Additional Key Words and Phrases: GPU, register file, thread occupancy

ACM Reference format:

Dani Voitsechov, Arslan Zulfiqar, Mark Stephenson, Mark Gebhart, and Stephen W. Keckler. 2018. Software-Directed Techniques for Improved GPU Register File Utilization. *ACM Trans. Archit. Code Optim.* 15, 3, Article 38 (September 2018), 23 pages.

<https://doi.org/10.1145/3243905>

Authors’ addresses: D. Voitsechov, Technion, Electrical Engineering Department, Haifa, 3200003, Israel; email: dani@tce.technion.ac.il; A. Zulfiqar, M. Stephenson, M. Gebhart, and S. W. Keckler, NVIDIA, 2788 San Tomas Expy, Santa Clara, CA 95051, USA; emails: {azulfiqar, mstephenson, mgebhart, skeckler}@nvidia.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.

1544-3566/2018/09-ART38 \$15.00

<https://doi.org/10.1145/3243905>

1 INTRODUCTION

GPUs have been successfully deployed to accelerate a wide array of highly parallel applications, including graphics rendering [16], high-performance computing [29], machine-learning tasks [7] such as image classification [24], speech recognition [15], and natural language processing [8]. These massively parallel processors are capable of achieving a high fraction of their peak performance well in excess of a TeraFLOP [39]. To sustain these peak throughputs, GPUs employ massive multithreading with fast switching between threads to hide instruction execution latencies and maintain high overall throughput.

Today's GPUs are provisioned to concurrently execute thousands of threads, each of which needs memory for registers, scratch-pad, and stack to store its intermediate state. One GPU design challenge is balancing the allocation of chip area between compute (ALU) and memory resources (per-thread state) to maximize Thread Level Parallelism (TLP). Figure 1 plots the speedup obtained on a GTX 1080-like GPU as a function of register file size. Each point in the figure is the speedup normalized to a configuration with a 128KB register file, averaged over 210 compute workloads. The workloads and GPU simulation environment are described in Section 5.5. Figure 1 shows that with more registers available, the SMs are able to run more threads concurrently, and overall application performance improves.

To improve the performance of register bound applications, the size of the register file can be increased in future GPU chips. Unfortunately due to transistor scaling challenges coupled with the inability to grow chip area due to reticle limits, pursuing this strategy in the long term is impractical. Compounding this engineering challenge is the observation shown in Figure 1 that even when the register file size is doubled, performance on current GPUs only improves by 13% on average, motivating exploration of cheaper methods for scaling performance. Our article aims to better use the existing register file capacity by exploiting three program properties: early register release points and similar or narrow operand values. While there is prior art that explores these directions [14, 19, 21, 30], we show that these previous approaches either fall short in delivering robust performance, or their ramifications on thread occupancy (TLP) have not been explored.

In this article, we propose three orthogonal compiler driven approaches that increase the register utilization:

- (1) Early register release identifies and exploits locations in code where a register name will no longer be referenced and therefore safely reallocates these registers to waiting threads.
- (2) De-duplicating scalar operand values identifies registers that hold scalar-values and allocates these values to an efficient scalar register file by the compiler.
- (3) Packing narrow bitwidth operands recognizes and packs multiple narrow operands together in a single 32-bit register.

Current GPUs patiently wait until a collection of threads terminate before reclaiming precious register resources [37]. In a recent work, Jeon et al. [18] propose allocating and deallocating GPU registers during execution based on value liveness information marked in the instruction stream by the compiler. Unmitigated, this approach is prone to deadlocks, which is why the authors propose incorporating a deadlock-avoidance mechanism in their implementation to guarantee forward progress. We demonstrate that this scheme can sometimes cause severe thread throttling leading to diminished performance compared to current GPU designs. Our article proposes an alternative strategy of only deallocating registers at concrete points during a program's execution, because those registers are guaranteed not to be used again. We developed a compiler analysis that statically exposes this common property and designed an architecture to exploit it. We demonstrate that our early register release design outperforms previous work on dynamic

Fig. 1. Performance of a GTX 1080-like GPU with respect to register file size. The speedup is normalized to a configuration with a 128KB register file, averaged over 210 compute workloads.

register allocation [18] delivering performance (thread occupancy) gains as high as 15% (25%) over the baseline GTX 1080-like GPU design.

The additional two techniques exploit the existence of *scalar* and *narrow* operands in programs. While the values contained in a vector operand can vary across threads, scalar operands have the same value across a well-defined subset of threads [26]. Gilani et al. [13] propose detecting these scalar operands dynamically during execution and storing them efficiently in a separate scalar register file. Since the amount of used scalar register resources is unknown at thread launch time, their technique is unable to leverage savings in vector register capacity towards improvements in thread occupancy. Our article exploits the insight that a considerable fraction of scalar operands can be identified statically [26]. By knowing the required register resources at thread launch time, more concurrent threads can be launched commensurate with the number of operands allocated to the scalar register file. We demonstrate that this approach is effective, yielding performance (thread occupancy) gains as high as 72% (~200%).

Finally, we investigate an approach that redesigns the register file along with the associated datapath to support narrow subword reads and writes. This approach is motivated by the observation that most integer operands in GPU applications dynamically use a small fraction of a register’s bitwidth [42]. By extending a previously proposed static bitwidth analysis [4], the register allocator in our compiler is able to pack multiple subword operands into a single 32-bit register. Compared to the dynamic subword packing design proposed in Reference [13], static allocation can enable higher thread occupancy in GPUs as the required register resources are known at thread launch time. Our approach leads to overall performance (thread occupancy) gains of up to 60% (~200%).

Our article also demonstrates that these three techniques complement each other. That is, individually each approach often fails to free enough registers to improve occupancy, but together they synergistically free enough registers to launch additional parallel work. Combining the three proposed approaches yields TLP improvements as high as 205% leading to performance gains of up to 100%.

2 MOTIVATION

Modern GPUs such as those made by AMD [1, 2] and NVIDIA [36, 39] are capable of executing bulk synchronous programs that are expressed as *kernels*. Using NVIDIA’s terminology for CUDA, kernels decompose into collections of threads known as cooperative thread arrays (CTAs). CTAs themselves consist of distinct 32-element sets of threads called *warps* that are tightly coupled at the architectural level. The threads of a warp execute in SIMT fashion. As part of the CUDA programming model, threads within a CTA can share a common software-managed memory (shared memory). A GPU contains one or more streaming multiprocessors (SMs), and the *CTA scheduler* maps an entire CTA to a single SM. A single SM on a GTX 1080 (see Table 1) can simultaneously

Table 1. Occupancy Resources of An SM

SM version 6.1 in CUDA (GTX 1080 [38])	
Occupancy limiter	Value
Register file size	256KB
Shared memory size	96KB
Maximum warps	64
Maximum cooperative thread arrays (CTAs)	32

execute as many as 32 CTAs, but as we discuss in the next section, several kernel and architectural characteristics can limit parallelism.

2.1 Occupancy and Its Limiters

The properties of a given kernel and the target architecture determine how efficiently the kernel will run. The metric for efficiency that we use in this article is *occupancy*, which is defined to be the ratio of the number of threads that are active in an SM to the maximum number of threads that the SM supports. A CTA's resource demands are known at kernel *launch* and include the number of registers, amount of shared memory, the number of warps, and so on, required to execute all of the threads in the CTA. Likewise, the SMs of a given architecture are provisioned with a fixed amount of register file (RF), shared memory, warp slots, and so on.

Table 1 highlights a subset of the SM resources, which also happen to be the primary *occupancy limiters*, in some NVIDIA Pascal GPUs (including GTX 1080). We can see that if a CTA uses 48KB of shared memory, then even for the GTX 1080, which is provisioned to run up to 32 CTAs per SM, at most two CTAs can concurrently run per SM (because the two CTAs use *all* of the available shared memory). An SM can be provisioned with higher occupancy resources to support more thread-level parallelism (TLP). However, this results in increased power and area consumption leading to a higher costs.

2.2 The Importance of the Register File

The four resources in Table 1 are the most common occupancy limiters, but how important is each resource overall? To answer that question we conducted an extensive study using 210 kernels that dominate the execution time of 70 compute applications (from Rodinia [6], Parboil [43], Lonestar [5], OpenCV [17], etc.) for the SM configuration highlighted in Table 1. The number of registers per thread, shared memory usage, size of CTA, and grid size are the possible occupancy limiters that determine the maximal occupancy of a kernel. We have extracted these occupancy limiters for the 210 representative kernels with NVIDIA's freely available nvprof profiling tool [40]. Table 2 presents the results, which clearly show that the register file is by far the single biggest occupancy limiter (47%) observed in workloads. This study plainly motivates our work's focus on the register file. The next section describes the three techniques proposed in our article to improve the efficiency of the register file.

3 OPTIMIZING RF UTILIZATION

This section presents the three approaches proposed in our article along with the key application-level insights highlighting their effectiveness. For each technique, we also present the compiler analysis needed to enable it. Section 4 describes the implementation details of the baseline GPU architecture and our enhancements.

3.1 Early Register Release

GPUs allow the allocation of a large number of architectural registers per thread. For example, NVIDIA's Pascal GPUs allow each thread to use up to 255 registers. As we have seen, however,

Table 2. Occupancy Limiters Observed in 210 Representative Kernels for the SM Configuration Shown in Table 1, Extracted with NVIDIA's nvprof Profiling Tool [40]

Occupancy limiter	% of Workloads
Register file	47.6
Register file + Warps	28.9
Warps	16.8
CTAs	1.4
Shared memory + Warps	<1
Register file + Shared memory	<1
Shared memory	<1
Other combinations	5

Some workloads are limited by more than one resource; these workloads do not benefit from increasing just one resource and require increase in two or more resources to improve occupancy. Occupancy limiter combinations are marked with a "+" in the table.

Fig. 2. Register usage profile in the object detection workload from the OpenCV library [17].

heavy per-thread register usage limits occupancy. GPU architectures allot a CTA's register resources when the architecture schedules the CTA. The allotment conservatively provisions for registers that *statically* might be needed but may not be dynamically used. The architecture frees the entire set of registers allotted to a warp only when *all* threads of that warp have exited [37]. By profiling applications for the number of simultaneously live registers, we can get an upper bound for how wasteful the de facto GPU policy is.

Figure 2 shows how register usage varies with time for an object detection workload from the OpenCV library [17]. These phases of high and low register usage are quite natural in programs with loops. The de facto GPU policy allocates 31 registers for the entire duration of the kernel. The solid line in the figure plots the number of live registers, where a register is live if it could be read before its next write. The figure demonstrates that the number of live registers can vary dramatically as a program executes [18]. The object detection workload has several points during the execution of the kernel where 10 or fewer registers are live. In recent prior work, Jeon et al. [18] have proposed virtualizing the register file, which lets the GPU allocate and deallocate registers at a very fine granularity to match an application's dynamic register needs at any point in the application's execution. As an example, for the workload in Figure 2, the per-thread register usage of Reference [18] would approximate the set of live registers at any given point, oscillating between allocating and deallocating as registers die and become live again (the solid line).

We pursue a simpler strategy of only deallocating registers, leveraging the key insight revealed from our profiling work that many kernels reach points early in their execution where they *permanently* cease to access a large fraction of their allotted registers. These points quite often arise at the beginning of control flow changes in a program where two mutually exclusive execution paths have different behaviors. This observation led us to develop a simple static analysis that determines where these points are. The dashed line in Figure 2 shows the results of our analysis that computes the set of registers that conservatively might be used again. These are referred to as *Register Working Sets* (RWS) in the rest of the article. Thus our technique initially allocates all required registers when a CTA is scheduled, but we allow the warps of a CTA to eagerly deallocate registers that are not members of the register working sets *during* execution. The gap between the dashed and solid lines on the right side of Figure 2 reflects the compiler’s conservative computation of the register working set and the potential for a path not taken by the program to use more registers than the solid line shows.

3.1.1 Our Approach versus Full Virtualization [18]. Launching new threads while dynamically allocating and deallocating registers between them as proposed in Reference [18] can induce deadlock in otherwise sound applications. Elaborating further, consider the workload example shown in Figure 2, and assume that it is executing on a GPU where each SM has 40 physical registers and can support two concurrent CTAs. For simplicity, assume that each CTA contains a single warp. Using an un-bound dynamic allocation and deallocation scheme, two CTAs would start executing, each holding zero registers initially. Suppose both these CTAs are prioritized equally for execution by the SM such that they simultaneously reach 18% of their execution time. At this point in execution, both CTAs will be holding 20 registers each equaling the capacity of the register file leading to a circular wait dependency for acquiring resources and resulting in deadlock. To avoid deadlocks, Reference [18] proposes a deadlock avoidance scheme in which warps from at least one CTA are always allocated the maximal register amount (Figure 2) to guarantee their forward progress. This approach however can cause all other CTAs to throttle once register file resources are exhausted; only one CTA advances while the others sit idle, which clearly can lead to drastic performance degradations. Our early register release solution based on the concept of register working sets never leads to a deadlock, because a warp only releases resources that are statically guaranteed to be unused in its remaining execution.

Higher thread occupancy can sometimes lead to diminished performance due to increased contention in any shared on-chip storage structures (e.g., caches). To compare the performance potential of these two approaches independent of these thrashing side effects, we conducted an idealized simulation where instruction execution was modeled without overheads (i.e., single-cycle completion latency). We define “SM utilization” in this idealized evaluation as the total time to execute all the warps in these two approaches normalized to baseline execution time. Figure 3 presents SM utilization for 210 CUDA workloads, which demonstrates that the full virtualization design can deliver higher potential speedups than our approach but exhibits severe utilization drops due to its throttling mechanism in 35% of the workloads. Our technique does not suffer from these utilization cliffs offering reliable performance albeit to a lesser extent than in Reference [18], but avoiding all of its dynamic renaming and throttling complexity. Overall, our design provides 30% better performance potential than in Reference [18].

3.1.2 Determining Register Working Sets. The register working set at any program point is defined as the set of registers that might be read or written by executing the kernel from the given point. To illustrate, consider the control flow graph in Figure 4. This simple three basic block snippet shows to the right of each instruction the set of live registers (in blue), and the register working set (in gold). In particular, note that at the conditional backward branch (instruction [9]) in *BB1*

Fig. 3. Normalized utilization efficiency of full virtualization design [18] versus our approach (higher is better).

Fig. 4. Register working set example.

only R0 is live. However, if the branch is taken, then R0, R2, R3, and R4 will be used again, and thus these register names cannot be released. If, however, the branch falls through to BB2, then the analysis tells us that R3, and R4 can be released, because they are no longer in the register working set. Throughout the remainder of the article, we refer to registers that will no longer be used as *releasable*.

Register working sets are determined by performing a backwards dataflow analysis. However, unlike traditional liveness, when computing the input to an instruction, we do not subtract an instruction's definitions from the output set. The complete set of dataflow equations for the analysis follows:

$$\begin{aligned}
 RWS_{out}[last] &= \emptyset \\
 RWS_{in}[s] &= USES[s] \cup RWS_{out} \\
 RWS_{out}[s] &= \bigcup_{p \in succ[s]} RWS_{in}[p].
 \end{aligned}$$

In the equations above, $USES[s]$ is the set of source registers used in instruction s , $RWS_{in}[s]$ contains the register working set immediately before instruction s executes, and $RWS_{out}[s]$ contains the register working set immediately after instruction s executes. The algorithm proceeds backwards up the program's dependence graph iteratively computing the RWS for each instruction as

Fig. 5. Fraction of workloads in which warps have the given register working set size at various points of execution.

a function of its own operand uses and those of its successors. We compute the register working set by simply applying the above equations until the analysis reaches a fixed point [35].

The cardinality of the register working set is monotonically non-increasing with respect to the dynamic execution of the program. In other words, registers that are not in the register working set at a given instruction point are guaranteed not to be used in all reachable paths of that program from that point on. Hence, they can be safely deallocated during execution, thereby freeing up register resources for other unexecuted threads. The dashed line in Figure 2 shows the dynamic register working set behavior of the object detection kernel from the OpenCV library [17]. As this figure demonstrates, our proposed scheme would allow warps to release 19 registers once they reach $\sim 85\%$ of execution time making these resources available to other unexecuted warps so that they may begin executing early. The figure also makes the monotonically non-increasing property of register working sets apparent. The dynamic register working set size will likely be greater than the dynamic liveness set size (as in Figure 4).

The candidate applications that benefit most from our technique are those that are register occupancy limited (i.e., the register file size is the *only* TLP limiter in hardware) and where the register working set size decreases early in the execution of a thread. The sooner a thread is able to release its register resources, the earlier the architecture can apportion those registers to schedule other threads. To judge how many workloads can benefit from our proposed dynamic register deallocation technique, we examined the register working set profiles of a wide range of compute workloads. Figure 5 demonstrates the fraction of workloads that exhibit different numbers of releasable registers as a function of completed execution time. This figure shows that for more than 40% of the workloads, at least 10% of their registers are releasable when their warps complete 80% of their execution. Furthermore, for about 33% of the workloads, as many as 20% of their registers are releasable once they complete 80% of their execution. This highlights promising performance potential for our proposed technique.

3.2 Deduplicating Scalar Operand Values

Because GPUs operate in SIMT fashion, a single instruction dictates the behavior of multiple threads (up to 32 for NVIDIA's GPUs). Since each thread has its own register set, common values across threads will be replicated in the register file. Consider a simple instruction such as, `MOVI R1, 8`, which moves the value eight into R1. Each of the active threads in the warp has its own copy of R1 even though all versions of R1 contain the same value after the instruction has completed. These operands in a GPU program that have the same uniform value across all the active threads of a warp are defined as scalar. AMD's family of architectures can reduce redundant computation by scheduling common operations on a dedicated scalar unit [1]. Different register

allocation policies that exploit the scalar register file can affect occupancy and performance on these GPUs [23].

Lee et al. showed that a large fraction of operands in GPU programs are scalar [26], and they presented an analysis that statically identifies these operands. We implemented their scalarization algorithm in our compiler and modified the register allocator to place these scalar operands in a special-purpose scalar register file. The algorithm can be stated as a standard dataflow analysis. The analysis initially assumes that all register-allocatable operands are *invariant* and then proceeds to identify and (forward) propagate sources of variance (e.g., `tid.x`). That is, if any of an instruction's source operands is *variant*, then the instruction's destination operands will also be marked as *variant*. When the algorithm has converged, all operands marked *invariant* are statically guaranteed to not depend on any invariant computations, and thus can be safely allocated in the scalar register file. For every register we allocate to the scalar register file, we free a (32× larger) warppide register from the vector register file, leading to improved utilization. While the idea of using a scalar register file is not novel, we include the approach, because it effectively improves register file utilization. Furthermore, to the best of our knowledge, our work is the first that compares the occupancy of a vector-only register file to that of a split scalar and vector register file.

3.3 Packing Narrow Bitwidth Operands

Prior work has shown that programmer specified datatypes are often grossly overprovisioned [3, 4, 41]. Even for GPUs, a value-profiling study shows that integer-typed narrow operands are common [42]. In this work, we statically infer operand bitwidths and augment our register allocator to pack multiple, simultaneously live narrow operands into a single physical register. Using the design presented in Section 4.3, these narrow operands can be inexpensively packed together into a single 32-bit register, and thus can reduce the overall number of registers required per thread.

Our static analysis, which is based on the work of Budiú et al. [4], employs a conservative forward and backward data-flow analysis that uses instruction semantics to infer the values of individual bits produced by an instruction. Our implementation uses abstract interpretation to expose definite bit-values for operands. Each operand bit can be in one of five states that we have defined as:

- $\perp$ - The bit's value is uninitialized.
- 0 - The bit's value is zero.
- 1 - The bit's value is one.
- $\top$ - The bit's value is unknown.
- X - The bit's value is not subsequently used, and therefore is dead.

As an example of how the forward propagation proceeds, consider the following code snippet that computes a 64-bit offset for `(threadIdx.x + 1) * 4`:

```
(1) S2R.U vr252 = 52
    vr252 = [00000000000000000000TTTTTTTTTT] (11 bits)
(2) ADD.I vr131 = vr252, 1
    vr131 = [00000000000000000000TTTTTTTTTT] (12 bits)
(3) SHR.I vr261 = vr131, 31
    vr261 = [000000000000000000000000000000] (zero)
(4) SHF.U vr302 = vr131, vr261, 2, .L.C.U64
    vr302 = [000000000000000000000000000000] (zero)
(5) SHF.U vr301 = vrZero, vr131, 2, .L.C.U64
    vr301 = [0000000000000000TTTTTTTTTTTT00] (14 bits)
```

Fig. 6. Baseline GPU architecture.

The S2R instruction (1) moves `threadIdx.x` into `vr252`, which can be up to 11 bits. The analysis cannot know statically what the actual value is, and therefore it sets the least-significant 11 bits to $\top$. However, it *can* infer that the most-significant 22 bits are “0”. The next instruction (2) simply increments `vr252`, which results in a possible carry, and therefore `vr131` is at most a 12-bit quantity. Instructions (3) and (4) are required to conditionally zero out the top 32 bits of the 64-bit result. Instruction (3) shifts a 12-bit quantity right 31 times, which our analysis correctly infers yields zero. Likewise the analysis infers that the funnel shift in (4) yields zero. The funnel shift in (5) shows that the approach can also infer concrete values for the least significant bits, even where more significant bits are not known. In addition to shifts, there are dozens of instructions that allow us to infer the values of certain bits. Budiou et al. [4] includes a more complete treatment of the analysis, including the backward data-flow analysis.

Our implementation uses the inferred bitvalues to compute minimum-bitwidth per operand annotations. Furthermore, we extend Reference [4] to record for each operand whether the value contained therein is provably non-zero. We do so, because this additional information helps refine bit-values for a few common integer intrinsics in the NVIDIA ISA.

We modified the register allocator to consider operand bitwidth information and to allow multiple subword operands with overlapping live ranges to share a physical register. Our allocator is capable of allocating operands at the byte level, allowing up to four operands to share a single 32-bit register.

The analysis performs well on applications that contain both a significant number of integer operations and operations that allow the analysis to deduce bit-values, but typically performs poorly on applications that are floating-point heavy. The analysis performs extremely well on `mummergpu` [6], which contains almost exclusive integer computation as well as abundant shift and logical operations with which the analysis can infer definite bit-values. For `mummergpu` 55% of the dynamic register writes can be satisfied with 16-bit registers, including the 46% that require only 8-bit registers. However, in many cases the analysis is only able to prove narrowness for fewer than 5% of the dynamic register writes.

4 RF OPTIMIZATION IMPLEMENTATION

Figure 6 shows the model of our baseline GPU that is composed of schedulers for launching work (kernel, CTA) onto the many simultaneous multiprocessors (SMs) that are interconnected with caches and high-bandwidth DRAM memory. Our article focuses on the design of the SM and its interactions with the CTA scheduling logic. The CTA scheduler tracks the occupancy resources

Fig. 7. A rename table is used to translate an architectural batch of registers to a physical register batch. Our design adds counters to this table to track register releases.

of individual SMs (# of free warp slots, CTA slots, registers, and free shared memory capacity) and performs the task of launching new threads on an SM once it has enough free occupancy resources.

The lower right of Figure 6 shows the SM architecture. Each SM is provisioned with resources to execute up to a fixed maximum number of warps and CTAs. These resources include a large register file for maintaining thread context, a software-controlled cache (shared memory), and a hardware-managed L1 data cache. The consecutive threads of a CTA that are mapped to an SM are bundled into warps, and the active threads within a warp are executed on functional unit (SIMT) lanes. The front-end of the SM pipeline primarily consists of an instruction cache and decode logic. The warp scheduler orchestrates the scheduling of decoded warp instructions for execution. We model the occupancy parameters of the SM to match those found in a modern NVIDIA GPU, as highlighted in Table 1. Figure 6 shows the sizes of the SM structures (warp slots, CTA slots, register file, shared memory) in our modeled GPU configuration.

Modern GPUs allocate and deallocate register resources in batches [37], which has important implications on the design of our register file enhancements. Specifically, a GPU requires a translation mechanism that performs the mapping between architectural registers (and their corresponding batches) to physical registers (belonging to their corresponding physical batches) in hardware. We assume a simple rename table, shaded in Figure 6, to perform these translations before the register file is accessed. The number of entries in the rename table along with the size of each entry is highlighted in Figure 7.

4.1 Early Register Release

The dynamic register deallocation technique proposed in our article involves several components. The first component is a compiler pass that implements the analysis of Section 3.1.2 to annotate each instruction boundary where registers can be released with its associated set of releasable registers. The second component is a mechanism for the SM to create batches of releasable registers; the SM consults the compiler’s annotations to dynamically track the registers that will no longer be used. Third, when the SM has aggregated a complete batch of releasable registers, the entire batch is deallocated and made available to the CTA scheduler so that it may be reassigned to new threads. In the final step, the CTA scheduler launches unscheduled threads when it acquires enough register resources (and other necessary resources) to satisfy their demands.

Marking registers for release. Using the register working set analysis presented in Section 3.1.2, the compiler can annotate the code at locations where it knows that one or more registers will cease to be used for the remainder of the program. Our design adds “release” instructions to the ISA to communicate information about releasable registers during program execution. For cases where fewer than six registers are released, we use a new instruction that explicitly releases up to five registers (REL RA, RB, RC, RD, RE). For cases where the compiler can release more registers,

we use an instruction that breaks the 255 architectural registers into eight 32-register segments, and communicates the segment and an associated bitvector that specifies which registers in the segment to release (`REL segment, [32-bit vector]`).

In our design, the compiler prefers inserting release instructions at the beginning of basic blocks to mark releasable registers at these program locations. The releasable registers are identified using the static analysis presented in Section 3.1.2. For the example program in Figure 4, the compiler places a release instruction at the beginning of *BB2* with registers *R3* and *R4* marked for release. Since, basic blocks *BB0* and *BB1* have no releasable registers, the compiler forgoes placement of release instructions at the beginning of these blocks. Within basic blocks, aggressively inserting too many release instructions can lead to significant performance overheads. After evaluating many different heuristics, our chosen compiler strategy inserts a release instruction once five or more releasable registers are identified within a basic block. We discuss a design alternative that can trade hardware overhead for performance at the end of this section.

Releasing registers during execution. During program execution, our design dynamically deallocates registers that are marked as releasable. However for two reasons, these freed registers cannot be made available immediately to the CTA scheduler to be allocated to new threads:

- GPUs allocate warp-wide vector registers to instructions. Thus a register must be released by all threads of warp before it can be deallocated.
- Modern GPUs allocate and deallocate register resources in batches as discussed above. Therefore, the individual registers that are released during execution must be tracked in the SM until a batch amount is available for deallocation.

To meet these constraints, our design adds counters to the rename table to track individual register releases by threads of a warp during execution until an entire batch of warp-wide registers is freed. These special counters are highlighted in Figure 7 and are sized to be $\log_2(\text{warp size} \times \text{batch size})$ bits. The counters are set to their maximum value when the batch is allocated to a new warp. When an architectural register that is marked for release is encountered, the value of the counter stored for its corresponding batch is decremented by a value equal to the number of active threads in the warp instruction. Once the counter value indicates (by underflowing) that all threads have released their batch, then our program analysis guarantees that the architectural registers belonging to its batch will not be used again by the warp. In this case, the CTA can reclaim the batch of physical registers corresponding to the batch of released architectural registers.

Deallocating batched register resources and launching new threads. During execution, as a batch of registers is released in the SM, it is deallocated from the owner thread and added to the “free list” structure of Figure 7 that tracks available physical register batches in the SM. The CTA scheduler is then notified of these available register resources. Once the scheduler determines that an SM has enough available resources, it can launch new threads on it.

Design alternatives. The compiler strategy used in our design offers robust performance by allowing register release even in potentially divergent control flow paths. However, this fine-grained approach can be relaxed to release registers at coarser locations in the program, leading to reduced implementation complexity but at the cost of lower performance. For example, one alternative design point releases registers at *convergence* points in the control flow graph (using the `REL` instruction). By restricting register release to program convergence points, the number of bits used in the rename table counters can be reduced. Since the cost of the additional state in the rename table of our design is quite small, releasing registers at coarser program locations does not offer a worthwhile tradeoff.

Fig. 8. Modifications (shaded blocks) to the SM datapath to enable efficient storage of (a) scalar and (b) narrow bitwidth operands.

4.2 Deduplicating Scalar Operand Values

Scalar operands, by definition, contain the same uniform value across all the active threads of a warp. Allocating these scalar operands in the vector register file results in redundant duplicated values occupying precious register file resources. Typically, these scalar register operands store user-defined constants, special uniform values loaded from a constant RAM, special registers (e.g., `blockIdx.x` in CUDA), or results of operations in which the source operands are strictly scalar. For example, memory reads where the base and offset operands used for address calculation are both scalar. Using the static analysis presented in Section 3.2, the scalar operands of a program can be identified and allocated to a separate scalar register file that maintains only one (32b) value for all the active threads of a warp. Figure 8(a) shows the incorporation of the scalar register file into the SM datapath. To support reads and writes to the scalar register file, the ISA of the GPU is extended to support scalar source and destination operands in instructions.

4.3 Packing Narrow Bitwidth Operands

Because modern graphics processors are designed to sustain high throughput for single-precision floating-point operations, the SM datapath is optimized for reading, writing and storing 32-bit operands per thread. In these architectures, narrow operand values (less than 32-bits) typically have to be explicitly packed together (e.g., using intrinsics) to exploit higher throughput math and storage allocation benefits. Motivated by the observation that integer register operands (e.g., loop trip count variables) are typically over-provisioned with respect to their declared datatypes, we propose a design that makes narrow operand read, write, and allocation capabilities available to code that does not explicitly pack narrow operand values. Using the static analysis presented in Section 3.3, the narrow integer operands in a program can be identified and packed into the register file using the design shown in Figure 8(b). By adding packing and unpacking logic to the ALU datapath, narrow operand values can consume fewer bits in the register file, such as 16 bits per thread in Figure 8(b). The packing and unpacking logic consists of shifters and sign extension circuitry to convert the source and destination operands into the proper formats for execution and storage. Note that most of this packing and unpacking circuitry already exists in GPU ALU pipelines accessible via a single instruction or a sequence of operations. Our design simply moves the packing and unpacking logic to an earlier stage in the pipeline such that all ALU instructions incorporate the ability to operate on narrow source and destination operands.

Table 3. Baseline Simulator Configuration

Parameter	Value
SM execution model	In-order
SM warp size	32-threads
SIMT lanes per SM	128 single-precision
SM L1/texture cache	48KB, 128B line size
SM DRAM bandwidth	32 Bytes/cycle
SM shared memory bandwidth	128 Bytes/cycle
SM occupancy configurations	(Table 1)
Max # of arch. registers per thread	255 (same as CUDA)
CTA scheduling policy	Round-robin
Register alloc./ dealloc. batch size	8 warp-wide registers

Figure 8(b) demonstrates our approach for packing 16-bit narrow operands. In this design, we extend the GPU ISA to support a “*Hi*” or “*Lo*” designation with each source and destination register operand of an ALU operation to select its upper or lower half, respectively.

While the design shown in Figure 8(b) enables packing of two 16-bit narrow subwords, it can be extended to support even finer-grained packing (e.g., four 8-bit subwords) by deploying narrower SRAM arrays in the register file banks and additional shifting logic in the packing/unpacking circuitry. The ISA can also be extended to allow appropriate subword granularity referencing of source and destination operands.

5 RF UTILIZATION EVALUATION

To evaluate our proposed techniques, we extended an NVIDIA proprietary simulator that models simultaneous multiprocessors (SMs) interconnected with a CTA scheduler for distributing work and a memory system as shown in Figure 6. Our simulator is capable of executing NVIDIA’s native SM6.1 ISA. In addition to supporting execution-driven simulation, the simulator incorporates a timing component for modeling detailed cycle-level behavior in the microarchitecture. Table 3 describes the key simulation parameters used in our evaluation.

We model the SM occupancy configuration after actual NVIDIA GPUs (Table 1). We evaluated the SM design on 210 workload kernels drawn from a variety of categories including high-performance computing (HPC), computer vision, and machine learning. These application kernels were compiled using NVIDIA’s production compiler tools. We implemented the static analyses described in Section 3 in the compiler, and the simulator was modified to model the designs presented in Section 4.

5.1 Results Summary

Figure 9 shows the occupancy and performance improvements obtained using our proposed techniques on over 200 CUDA workload kernels. While the range of improvements in occupancy and performance varies across the different workloads, the figure demonstrates that our compiler-aided techniques are able to deliver speedups as high as 2×. This figure also highlights several interesting behaviors. First, our three techniques collectively increase the effective register file size, leading to significant improvements in occupancy in many of the workloads. The occupancy improvement often leads to performance speedups, albeit by a smaller amount. However, there are cases where increased occupancy does not improve performance. Furthermore, a very small percentage of the workloads (those on the far left in Figure 9(top)) experience performance degradations with our techniques.

Fig. 9. (Top) Performance and occupancy improvements obtained over the baseline GPU on over 210 CUDA workloads. (Bottom) Enlargement of graph for the 75 *candidate workloads*.

To better understand these results, we categorized the workloads into three main classes, highlighted in Figure 9. A workload is classified as *non-register file sensitive* if it does not have the register file as the sole occupancy limiter. The *memory bandwidth limited* workloads do not benefit from additional threads, because the DRAM interface is already saturated. The *candidate workloads* are those where the register file is the only occupancy limiter and where memory bandwidth is not a bottleneck.

The non-register file sensitive workloads are unable to map more concurrent threads with improvements in register file size as they are bottlenecked on a non register file hardware resource (Section 2). Although memory bandwidth limited workloads in Figure 9 sometimes do show improvements in occupancy (TLP), that occupancy increase does not translate into performance. In fact, the performance on some workloads decreases as occupancy is increased (workloads on the left of Figure 9). Mapping more threads in an already memory bottlenecked workload results in higher cache and DRAM contention, reducing overall performance. On NVIDIA GPUs, performance-conscious programmers optimize the tradeoff between occupancy and memory system pressure by recompiling the program with different values for the `-maxregcount` flag. While a developer could explore this tradeoff space when using our compiler-based register file optimization techniques, performance degradations are rare and minor, limiting the necessity of controlling the compiler in this way.

The candidate workloads are sensitive to register file size and are *not* bottlenecked on a GPU resource such as memory bandwidth. These workloads can yield higher performance with increased occupancy as mapping more concurrent threads will generally lead to better latency-hiding behavior during execution. The rest of our evaluation focuses on these candidate workloads, which are

Fig. 10. Dynamic register working set behavior of Multiresolution Blend and SGEMV.

shown in greater detail at the bottom of Figure 9. As this figure demonstrates, our three techniques deliver improvements across many kernels, providing an average 12% speedup (23% occupancy increase) across the entire candidate workloads set.

The rest of this section explores the specific benefits of each of the three techniques. For ease of presentation, we picked 20 workload examples with varying characteristics to explain the performance trends. However, each graph we present includes a geometric mean over the entire candidate workloads set to illustrate overall improvements.

5.2 Early Register Release

Compared to a current GPU where register resources are held throughout the execution of a warp, our technique releases registers early, effectively making the register file appear larger than it is. We define a new metric called *effective register file (RF) increase*, which is the ratio between the sum of register lifetimes (in terms of elapsed cycles for complete kernel execution by all threads) in a baseline GPU to the sum of register lifetimes when using early register release. Workloads that have good register working set behavior and release a significant number of registers early in a warp's execution will exhibit high effective register file size improvements. Alternatively, a workload with poor register working set behavior may not show any improvement.

Figure 10 shows the dynamic register working set behavior of two candidate workloads, Multiresolution Blend and SGEMV. Multiresolution Blend exhibits better register working set behavior than SGEMV, because it releases 45% of its registers early. Figure 11 shows that this behavior leads to an effective RF increase of 1.6× for Multiresolution Blend compared to no improvement for SGEMV. The figure demonstrates in many applications effective RF increase is translated to higher occupancy, which then leads to higher performance (as much as 15% on SFM). Across all candidate workloads, our early register release technique provides an average 2.7% higher performance. Supporting dynamic register release requires 2KB of extra storage to implement the special counters in the rename table (Section 5.6). Section 5.5 further discusses the area overheads for our register file optimizations.

5.3 Deduplicating Scalar Operand Values

The baseline GPU allocates registers for all active threads of a warp (occupying 32×32b) in the register file regardless of the values contained within them. Our second technique statically identifies scalar operands in a program and allocates them to a separate scalar register file that maintains only one (32b) value for *all* the active threads of a warp. Using this approach, redundant scalar

Fig. 11. The improvements enabled by the early register release technique. The effective RF, occupancy, and performance are increased on average (geometric mean) by 6.2%, 4.2%, and 2.7% respectively.

Fig. 12. Gains by deduplicating scalar operands. The effective RF, occupancy, and performance are increased on average (geometric mean) by 19%, 12.3%, and 6.7% respectively.

values can be removed from the main register file, effectively increasing its size. Figure 12 shows the effective RF increase, together with occupancy and performance improvements over the baseline of our design with a 4KB scalar register file. The effective RF size increases for many workloads (e.g., Thrust scan, Coral UMT, and CoMD), which results in more concurrent warps (higher occupancy) executing on the SM, and overall improved performance. However, some applications see an increase in effective RF size but there is no improvement in occupancy or performance (Coral NAMD, SpMv, DeepBench, Conv, etc.). These cases arise because occupancy is not a linear function of available resources but rather a step-function where a new CTA can only be launched if the technique is able to free enough register resources in an SM as required by a CTA. The scalar

Fig. 13. Improvements from narrow operand packing. The effective RF, occupancy, and performance are increased on average (geometric mean) by 3.6%, 2.6%, and 1.7% respectively.

deduplication optimization delivers performance improvements as high as 72%, with an average 6.7% across all of the candidate workloads.

We also performed a sensitivity study on the size of the scalar register file, reducing it from 4KB to 3KB and 2KB. Compared to 4KB, the smaller scalar register files diminish performance benefits to 5.7% and 4.5%, respectively. Since the 4KB scalar register file only increases the total register file area by 1.5% while providing a 20% effective RF increase, spending some additional area on the larger scalar register file is a worthwhile tradeoff. Our results also showed that increasing the scalar register file size beyond 4KB did not improve occupancy or performance for the workloads we studied.

5.4 Packing Narrow Bitwidth Operands

Using a static analysis, our third technique identifies and packs multiple narrow bitwidth operands into a single 32-bit (thread) register leading to higher effective RF size over the baseline GPU. Figure 13 shows this effective RF increase together with occupancy and performance improvements when we employ an 8-bit subword design where four such subwords can be packed together into a single 32-bit register (Section 4.3). As this figure demonstrates, only three workloads (SFM, Thrust Scan, CoMD) exhibit occupancy improvements and speedups due to the register file savings provided by this technique. Although this technique is able to free register resources for most of the other workloads (Coral NAMD, Graphcut, VGG, AlexNet, SpMv), the freed registers still fall short of the number required to launch a new CTA. On average, the narrow subword packing technique, on its own, only delivers a 1.6% speedup (2.5% occupancy improvement) across all the candidate workloads.

5.5 Combining All Three Techniques

The results in Sections 5.2–5.4 demonstrate that applying individual optimizations can yield occupancy and performance improvements for some applications, but little to no benefit for others. While any one technique may not free up enough registers to launch new threads, the synergy of all three techniques often frees up enough registers to launch an additional CTA, increasing both parallelism and performance. Figure 14 demonstrates the effective register file size, occupancy,

Fig. 14. Improvements from all three techniques. The effective RF, occupancy, and performance are increased on average (geometric mean) by 31%, 22.8%, and 12.3% respectively.

and performance improvements obtained over the baseline GPU when combining all three of our approaches. The combination is effective on many workloads, delivering occupancy and performance speedups as high as 3× and 2× respectively. On average, our approaches enable 23% higher GPU occupancy, leading to 12% speedup over all the candidate workloads. Furthermore, our RF efficiency techniques are able to increase the effective register file size by an average 31% compared to the baseline GPU. While this article converts the improved register file efficiency into performance, it could instead be used to design a smaller register file that maintains the same performance level as the baseline GPU.

5.6 Hardware Overheads

Our techniques add a total of 6KB of state to the SM—2KB for the rename table counters and 4KB for the scalar register file. The additional 6KB of registers correspond to about 2.3% of additional SM area. While one could apply that 6KB to build a larger (262KB) register file, our experiments show that this configuration provides no performance improvements on our workloads over an SM with a 256KB register file.

As demonstrated in Figure 8(a), the scalar RF is connected to the crossbar network and operand collectors that are used in current GPUs to communicate/broadcast the intermediate-values/constants stored in the Vector RF/Constant RAMs. This additional connection into the crossbar switch adds minimal area to the GPU datapath.

As explained in Section 4.3, current GPUs limit the use of narrow integer operands to specific instructions in the ISA. Hence, the logic required to pack/unpack narrow bitwidth operands is typically already included in existing ALU pipelines. Our design simply moves this pack/unpack logic to an earlier stage of the ALU pipeline leading to minimal area overheads. However, incorporating the pack/unpack logic before the ALU pipeline can increase its timing path. If this additional timing delay cannot be absorbed in existing datapath timing slack, then additional pipeline stages may need to be added. Since, GPUs are latency-tolerant processors, our evaluations show negligible performance degradation if we increase the ALU pipeline latencies by one to two cycles.

Our results, which are presented in Figure 14, show that our optimizations convert the small amount of chip area presented in this section into significant performance gains that are not achievable by simply making the register file a little larger.

6 RELATED WORK

There is a rich body of work aimed at improving the performance, area, and energy consumption of register files for many types of processors such as CPUs [21, 32], VLIW processors [45, 46], and more recently for GPUs [10, 11]. This section discusses the most relevant previous work.

Out-of-order rename optimizations. Most modern CPU processors employ register renaming to improve instruction-level parallelism. Approaches to remove inefficiencies pertaining to register renaming include early release of physical registers using a hardware-only mechanism [33, 34] or by using software-directed techniques [21, 32]. Moudgill et al. propose releasing a physical register at an earlier point in the pipeline rather than waiting for the instruction that writes its mapped architectural register to commit [34]. Jones et al. [21] and Martin et al. [32] apply compiler techniques to explicitly mark operands that will not be used again. The goal behind all of these techniques is to allow out-of-order processors to reclaim the physical register associated with a logical register as soon as possible. Lo et al. [31] evaluate a similar approach for SMT processors. Note that the register working set concept introduced in our article is different from the value liveness measure used in these prior approaches for defining dead registers. Specifically, there is no differentiation between logical and physical registers in our approach such that when a physical register is deallocated, its logical definition is guaranteed to be dead for the rest of execution. This register can then be claimed by other threads without deadlock concerns.

Exploiting operand properties. Prior work on CPU register files has also explored techniques to improve utilization, area, and energy consumption by exploiting various operand value properties. To reduce the number of physical registers employed for renaming, Lipasti et al. [30] propose embedding small value operands in the rename table instead of storing them in physical registers. Ergin et al. [9] present schemes to pack narrow-operands together in the register file to improve utilization. González et al. [14] discuss a content-aware register file that stores short and long values separately to reduce area and power consumption. This design also allows operands to share identical subsets of consecutive bits by storing a value once and using indirection pointers to it. Jourdan et al. [22] propose an approach where multiple logical registers can be mapped to the same physical register if they share the same value. The design presented in Reference [13] also proposes efficient storage of scalar and narrow operand values. However, since these operand types are detected dynamically, their approach cannot be used to improve occupancy. Extending research in this area, ours is the first work that leverages statically inferred operand value properties for providing occupancy and performance improvements in GPUs.

GPU register files. GPU register files are large and expensive, both in terms of area and energy consumption. A variety of recent work aimed to improve the efficiency of register files in GPUs. Gebhart et al. [10] propose using a small register file cache to improve the energy consumption of register accesses. In their follow-on work [11], the authors improve the efficiency of their design by replacing the cache with another level of register file. Data movement in the two-level register file hierarchy is orchestrated by the compiler. The authors in Reference [12] propose a unified memory design from which the register file, scratch-pad memory and cache can be carved out of the same SRAM according to the needs of the target workload. Jing et al. [20] extend this idea by enabling on-chip memory resources to be partitioned dynamically between register file and cache instead of at kernel launch time. Our early register release technique can be applied to these prior approaches to further reduce the register file footprint. Similarly to our approach, the authors in Reference [18] propose releasing registers early during execution. However, as shown in our article, the thread throttling mechanism required in their design for deadlock avoidance can cause significant performance degradations.

Synergistic related work. Recent related work has investigated techniques that ultimately reduce memory bandwidth pressure [27, 28] and would therefore synergistically broaden the set of

applications for which improved occupancy yields improved performance. While we do not evaluate the energy-related benefits of our technique, the orthogonal work presented in References [25, 44] nicely highlights the importance of the register file for power optimization.

7 CONCLUSION

GPUs devote the majority of their on-die memory storage to large register files that store the state of tens of thousands of in-flight threads. In spite of this overwhelming provisioning of resources, our study shows that the vast majority of workloads (47%) in modern GPUs are still limited from launching more parallel work because of a scarcity in available register file resources. While simply scaling register files to ever larger sizes is conceptually straightforward, the area and cost ramifications of this approach make it increasingly impractical. Compounding this challenge is the observation demonstrated in this article that nearly half of the workloads would see no benefit from a larger register file, because they are limited on a variety of other hardware bottlenecks. This observation motivated our approach to instead improve the efficiency of the *existing* register file.

This article presents three compiler-based techniques that enable efficient allocation/ deallocation of register resources in GPUs leading to improved TLP and performance. Our first technique modifies the traditional liveness algorithm to mark last read/write use of registers in *all* reachable executions of the program. Register resources are then eagerly reclaimed at these marked locations instead of when the thread finishes execution. We show that our early register release approach offers higher overall performance over the state of the art [18] while avoiding many of its hardware complexities. Our remaining two techniques deploy static analyses to identify scalar and narrow operands in the program that are then allocated more efficiently in hardware. An in-depth performance evaluation shows that all three techniques, when combined, deliver speedup gains as high as 100%. On average, we observe 12% speedup on register-bound workloads not saturated on GPU resources.

REFERENCES

- [1] AMD. 2012. AMD Graphics Cores Next (GCN) Architecture. Retrieved from https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.
- [2] AMD. 2016. Dissecting the Polaris Architecture. Retrieved from <http://radeon.wpengine.netdna-cdn.com/wp-content/uploads/2016/08/Polaris-Architecture-Whitepaper-Final-08042016.pdf>.
- [3] David Brooks and Margaret Martonosi. 1999. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'99)*. 13–22.
- [4] Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. 2000. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the European Conference on Parallel Processing*. 969–979.
- [5] Martin Burtcher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC'12)*. 141–151.
- [6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC'09)*.
- [7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. <https://arxiv.org/abs/1410.0759>. (2014).
- [8] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. 2011. Natural language processing (almost) from scratch. <https://arxiv.org/abs/1103.0398>. (2011).
- [9] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the International Symposium on Microarchitecture (MICRO'04)*. 304–315.
- [10] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'11)*. 235–246.

- [11] Mark Gebhart, Stephen W. Keckler, and William J. Dally. 2011. A compile-time managed multi-level register file hierarchy. In *Proceedings of the International Symposium on Microarchitecture (MICRO'11)*. 465–476.
- [12] Mark Gebhart, Stephen W. Keckler, Bruce Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO'12)*. 96–106.
- [13] Syed Zohaib Gilani, Nam Sung Kim, and Michael J. Schulte. 2013. Power-efficient computing for compute-intensive GPGPU applications. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 330–341.
- [14] R. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, and M. Valero. 2004. A content aware integer register file organization. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*. 314–324.
- [15] A. Graves and J. Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. In *Neural Networks* 18, 5–6 (2005), 602–610.
- [16] John F. Hughes, Andries Van Dam, James D. Foley, and Steven K. Feiner. 2014. *Computer Graphics: Principles and Practice*. Pearson Education.
- [17] Itseez. 2015. Open Source Computer Vision Library. Retrieved from <https://github.com/itseez/opencv>.
- [18] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the International Symposium on Microarchitecture (MICRO'15)*. 420–432.
- [19] N. Jing, H. Liu, Y. Lu, and X. Liang. 2013. Compiler assisted dynamic register file in GPGPU. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'13)*. 3–8.
- [20] Naifeng Jing, Jianfei Wang, Fengfeng Fan, Wenkang Yu, Li Jiang, Chao Li, and Xiaoyao Liang. 2016. Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO'16)*.
- [21] Timothy M. Jones, MFR O'Boyle, Jaume Abella, Antonio Gonzalez, and Oguz Ergin. 2005. Compiler directed early register release. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 110–119.
- [22] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. 1998. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the International Symposium on Microarchitecture (MICRO'98)*. 216–225.
- [23] Charu Kalra. 2016. *Design and Evaluation of Register Allocation on GPUs*. Master's thesis. Northeastern University.
- [24] A. Krizhevsky, I. Sutskever, and G. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS'16)*.
- [25] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-compression: Enabling power efficient GPUs through register compression. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. 502–514.
- [26] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanovic. 2013. Convergence and scalarization for data-parallel architectures. In *International Symposium on Code Generation and Optimization (CGO'13)*. 1–11.
- [27] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'15)*. 17:1–17:12.
- [28] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. 2015. Priority-based cache allocation in throughput processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'15)*. 89–100.
- [29] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 8, 2 (2008), 39–55.
- [30] Mikko H. Lipasti, Brian R. Mestan, and Erika Gunadi. 2004. Physical register inlining. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*. 325–337.
- [31] Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Henry M. Levy, and Dean M. Tullsen. 1999. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.* 10, 9 (1999), 922–933.
- [32] Milo M. Martin, Amir Roth, and Charles N. Fischer. 1997. Exploiting dead value information. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 125–135.
- [33] Teresa Monreal, Víctor Viñals, Antonio González, and Mateo Valero. 2002. Hardware schemes for early register release. In *Proceedings of the International Conference on Parallel Processing*. 5–13.
- [34] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. 1993. Register renaming and dynamic speculation: An alternative approach. In *Proceedings of the International Symposium on Microarchitecture (MICRO'93)*. 202–213.
- [35] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

- [36] NVIDIA. 2012. Kepler GK110. Retrieved from <http://www.nvidia.co.uk/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.
- [37] NVIDIA. 2016. CUDA Occupancy Calculator. Retrieved from http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.
- [38] NVIDIA. 2016. NVIDIA GeForce GTX 1080 Whitepaper. Retrieved from http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.
- [39] NVIDIA. 2016. NVIDIA Tesla P100. Retrieved from <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.
- [40] NVIDIA. 2016. Visual Profiler User's Guide. Retrieved from <http://docs.nvidia.com/cuda/profiler-users-guide>.
- [41] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. 108–120.
- [42] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. 185–197.
- [43] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, Urbana.
- [44] D. Wong, N. S. Kim, and M. Annavaram. 2016. Approximating warps with intra-warp operand value similarity. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'16)*. 176–187.
- [45] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. 2000. Two-level hierarchical register file organization for VLIW processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO'00)*. 137–146.
- [46] Yanjun Zhang, Hu He, and Yihe Sun. 2005. A new register file access architecture for software pipelining in VLIW processors. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'05)*. 627–630.

Received March 2018; revised June 2018; accepted July 2018