Concurrency-Aware Register Stacks for Efficient GPU Function Calls

Ni Kang, Ahmad Alawneh, Mengchi Zhang*, Timothy G. Rogers
Elmore Family School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{kang222, aalawneh, zhan2308, timrogers}@purdue.edu

***Abstract*—Since the early days of computers, dividing a program into functions or subroutines has been a common way to manage complexity. Functions make programs easier to read, facilitate code reuse, and provide clean interfaces for separate compilation. However, function calls incur runtime overhead. We quantify the impact of this runtime overhead on GPUs and demonstrate that the register spills/fills required to maintain the function call application binary interface place significant bandwidth and capacity pressure on shared resources.**

To alleviate this overhead, we introduce Concurrency-Aware Register Stacks (CARS), a hardware mechanism that re-purposes segments of the GPU register file as a software-controlled hardware stack. CARS exploits the regularity in function prologue/epilogues to rename registers pushed to the stack with linear base + offset addressing, similar to the baseline GPU. Informed by lightweight call graph analysis and dynamic function behavior, CARS balances the space devoted to register stacks with the concurrency required to hide latency in GPUs.

Without harming function-free programs, CARS improves the performance and energy efficiency of 22 function-calling applications by 26% and 28%, respectively, outperforming idealized GPUs with impractical resources.

***Index Terms*—GPU, Programmability, Register File.**

I. Introduction

The ability to call functions is a cornerstone of any general-purpose language. *General-Purpose Graphics Processing Unit* (GPGPU) programming frameworks [25], [41], [43], [53] have supported device-side function calls (i.e., GPU functions calling GPU functions) since their inception. Historically, GPU codebases have been relatively small, compiled as single objects, where all device-side function calls are inlined [10], [11], [32], avoiding any runtime function calling overhead. However, GPU applications and libraries have grown significantly in size, complexity, and functionality. Figure 1 plots the trend in benchmark suite/library source lines of code (SLOC) and the number of device-side GPU functions implemented in those applications over the past 15 years of CUDA development. As GPU development has expanded, so have the codebases and their use of device functions. For example, the latest NVIDIA-supported machine learning libraries we study, Cutlass [39] and Rapids [9], [36]–[38], contain 3129 and 6348 code files, with 3760 and 27469 device-side function implementations, respectively.

*Work conducted at Purdue University, currently at Meta.

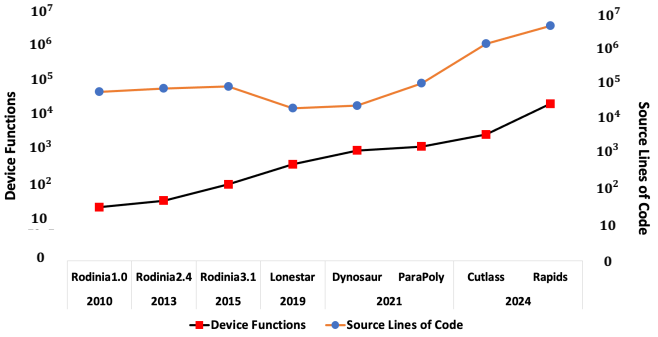


Fig. 1: Number of device functions and source lines of code (SLOC) for different GPU benchmark suites and libraries over the past 15 years, y-axes are in log scale.

Continuing to scale an "always-inline" approach to compiling GPU programs is not sustainable. Although inlining can improve performance by removing function calls and performing inter-procedural optimizations, it also has many downsides. Function inlining decreases software flexibility and can result in counter-intuitive performance cliffs that are difficult to predict statically [34], [49]. From a flexibility standpoint, inlining burdens the binary construction toolchain (often informed by the programmer) with determining which functions should be inlined, requires expensive link-time optimization [19] (LTO) to work with statically linked libraries or separate object compilation, is not possible with dynamically linked libraries, hinders the debuggability of the binary, and provides no runtime flexibility to adjust thread occupancy since the registers for one giant function must be loaded and persistent throughout the life of a warp. From a performance perspective, inlining increases compile/link time and memory consumption and can bloat the application's code size, increasing instruction cache pressure. For example, compiling and linking our suite of function calling workloads (detailed in Section V) with aggressive inlining results in a 10.15× increase in binary construction time and a 1.54× increase in binary size. Contemporary GPU instructions are wide (16B each in Volta/Hopper) since they encode explicit stalling information in each instruction to avoid dynamic dependency checking [13]. This, combined with larger codebases and aggressive inlining, has motivated

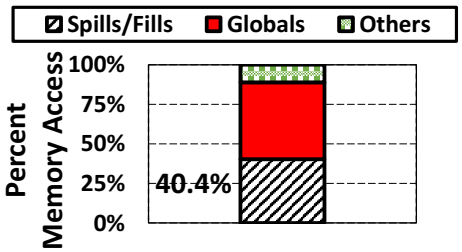


Fig. 2: Memory accesses broken down into spills/fills, globals, and other locals that are not spills/fills. Averaged over 22 function calling GPU apps (Section V) on an NVIDIA V100.

NVIDIA to increase their L1 instruction cache from 4kB in 2008 [59] (GT 200), to 128kB [16] in 2017 (Volta). We argue that the reliance of contemporary GPUs on inlining is hurting the software development process and adds storage resources to the frontend of the pipeline that could be used more effectively elsewhere. This paper proposes a novel mechanism that enables more efficient GPU function calls without relying on the programmer and compiler to remove them completely.

An examination of the GPU function calling *Application Binary Interface* (ABI) reveals that it is surprisingly similar to CPU ABIs despite drastically different bandwidth and register file demands. If a function uses a particular subset of the thread-private registers, it must spill/fill them to and from an in-memory stack. This spilling/filling of registers places increased pressure on shared resources. To understand the magnitude of this pressure, Figure 2 quantifies the spill/fill access frequency averaged across 22 function-calling applications on an NVIDIA V100 GPU. 40.4% of in-core L1D accesses come from moving register state back and forth between the register file and the L1D to enforce the ABI.

These frequent spills/fills, implemented as local memory instructions in GPUs, consume bandwidth and cache capacity that could be devoted to more useful instructions if the register state can stay resident in the register file. Both *bandwidth interference* and *capacity interference* play a role in spill/fill overhead. However, we demonstrate that prior work which focuses on mitigating *capacity interference* [8], [14], [15], [17], [18], [23], [30], [31], [47], [48], [60] cannot avoid the *bandwidth interference* caused by maintaining the ABI with memory instructions. We argue that massively multithreaded machines require a novel register allocation system to keep per-function register state in the register file more often.

Fundamentally, a register is spilled when entering a function because its value inside the function must differ from its value outside the function. On function entry, the architectural register is effectively *renamed*, via store/load instructions, then renamed back to its original value on exit. When performing traditional spills/fills using memory instructions, this renaming is implemented via relatively expensive tag lookups and cache accesses. To free up cache bandwidth and capacity, we propose renaming registers without accessing the memory system by augmenting the GPU's existing base + index register renaming.

We propose *Concurrency-Aware Register Stacks* (*CARS*) to achieve this in-register renaming. At a high level, *CARS* must do two things. First, it must decide how much register file space to devote to hardware stacks, possibly sacrificing concurrency, and handle the case when dynamic stack demand exceeds allocated capacity. Second, it must rename the registers pushed to the hardware stack to an empty region of the allocated register file and restore the name to its original location when it is popped.

CARS augments the register allocator on contemporary GPUs to carve out space for each warp's register stack. Contemporary GPUs examine each warp's register demand to partition the register file when thread blocks are assigned to cores. Using lightweight call graph analysis at link-time, already present in GPU compilers, *CARS* adds to this demand by estimating how much space each warp requires to keep the stack resident in the register file. When new thread blocks are scheduled, *CARS* adapts the space allocated to hardware stacks based on dynamic function calling behavior and the effect of reduced concurrency on runtime. When the demanded stack space exceeds the size allocated by *CARS*, a software trap is triggered to fall back to using local memory accesses.

Once registers are allocated, *CARS* leverages the GPU's existing hardware to perform efficient register renaming. Contemporary GPUs rename physical register locations with simple *base* + *offset* logic that allows physical registers to be assigned to different architectural registers based on kernel demand (i.e., a physical register can be assigned to warp 0:R64 in one kernel and warp 1:R0 in another). *CARS* uses a register frame and stack pointer to offset values for registers renamed on function calls (i.e., those spilled in the baseline). This lightweight renaming mechanism requires minimal changes to existing GPU hardware and avoids the expensive renaming hardware in CPUs and other GPU renaming schemes [14].

CARS requires minimal changes to the ABI and compiled code. ABI register designations (i.e., persistent, temporary, etc.) are unchanged. Spills and fills are replaced with push/pop instructions that move the register stack pointer when there is space and trigger memory accesses when there is not. Function call instructions save the caller's register frame pointer to the register stack and set the register frame pointer to the current register stack pointer. Upon the function's return, the register stack pointer is set to the current frame pointer, and then the caller's register frame pointer is restored.

In summary, we make the following contributions:

- We perform the first study of direct function calls on GPUs. We illustrate that the spill/fill instructions used to maintain the GPU function calling ABI introduce cache capacity and bandwidth interference.
- To alleviate these bottlenecks, we introduce *Concurrency-Aware Register Stacks* (*CARS*). *CARS* exploits the regularity in stack spill/fill operations to implement a novel register renaming mechanism that allows per-function register state to remain in the register file.
- To balance concurrency with deep register stacks, *CARS* introduces a novel hardware/software prediction mechanism that analyzes the call graph and observes runtime

function call behavior to adapt the size of the register stack reserved for each warp.

On a diverse suite of 22 function-calling applications, we demonstrate that *CARS* improves contemporary GPUs' performance and energy efficiency by 26% and 28%. *CARS* eliminates the spills/fills required to maintain the contemporary ABI to outperform a set of idealized configurations, including cores with unlimited register file and shared memory capacity, an oracle concurrency limiter that maximizes performance in the presence of L1D cache contention [47], and a GPU with a 10× larger L1D cache. *CARS* outperforms these idealized systems by alleviating capacity and bandwidth contention, as opposed to prior work, which focused on capacity alone.

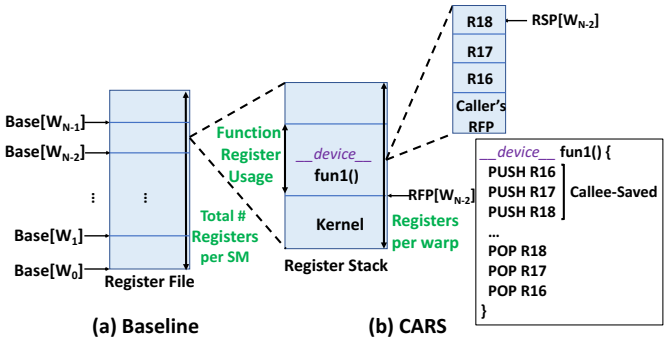


Fig. 3: Register file allocation mechanisms. (a) Depicts the baseline, and (b) illustrates *CARS*' extensions to the baseline.

II. Background and Motivation

Contemporary GPU ISAs implement a function-call ABI similar to contemporary RISC CPU ISAs [3]. Function parameters are passed first by registers, then the stack. The callee must preserve a subset of registers, and GPUs have both direct and indirect branch instructions to call/return from functions. Since GPUs group threads into warps, it is possible for threads in the same warp to call different functions [62], [63], return to different positions in the program or exit the function at different points. In most GPU codes, divergence on function calls/returns is rare.

Unlike traditional single-threaded ISAs, GPU instruction sets assign a variable number of architectural registers to each warp. GPU register files are organized like vector register files, where each thread in a warp is assigned one lane of a vector register. Individual registers are allocated at the warp-granularity (i.e., 128B-wide registers, with 4B for each of the 32 threads in a warp). Four factors can limit the number of thread blocks on each core: number of threads, number of thread blocks, register usage, and shared memory usage. When thread blocks are scheduled on a core, each warp is assigned a contiguous, fixed region of registers, whose size is determined at compile time. For kernels that do not call functions (or have all their functions inlined), the number of registers/warp is simply the registers needed in the main kernel function. For kernels with device-side function calls that are not inlined, each function is compiled individually, spilling/restoring any callee-saved registers it needs as required by the ABI. A maximum of 256 registers (8-bit are used to encode register identifiers in the instruction) are available to any given function. After each device function is compiled and labeled with its required register usage, the linker determines the worst-case register usage at any point in the call graph (i.e., which function uses the most registers). It allocates registers to each warp according to this worst-case.

Register spilling/filling is necessary for three reasons. First, if the compiler does not have full visibility of all the functions in the program (i.e., separate object compilation without link-time optimization), then the callee-saved registers in the ABI provide a guarantee that functions do not interfere with each other's registers. Second, if the compiler does have visibility into all functions, register usage may be too high, either greater than the 256 register maximum or so high that allocating more registers to each warp statically through global register allocation reduces concurrency and hurts performance. The concurrency aspect makes the compiler's life even more challenging since concurrency information (i.e., the size and the number of thread blocks in a kernel) is not known statically. Finally, supporting recursion requires spilling/filling. *CARS* is designed to alleviate all three of these scenarios, relieving the compiler from making difficult spill/fill decisions.

We profile the ABI of contemporary NVIDIA GPUs and find that callee-saved registers come from a contiguous set of architectural registers starting at R16. *CARS*' renaming design benefits from this fact, eliminating the need for an expensive register renaming table. Since threads (and warps) within a core tend to call the same functions simultaneously, this spilling/filling places pressure on the memory system. Even if the GPU were given infinite registers, individual functions would still be required to spill/fill registers to maintain the callee-saved register state defined by the ABI. Our goal is to alleviate pressure on the memory system by keeping multiple copies of architectural registers in the register file such that the contemporary ABI can be maintained.

III. Concurrency-Aware Register Stacks (*CARS*)

To keep the register state resident in the register file on function calls, we propose *Concurrency-Aware Register Stacks* (*CARS*). *CARS* creates a hardware stack for each warp in the register file. Using register frame and register stack pointers, we efficiently rename callee saved registers that require memory access in the baseline. Efficiently implementing *CARS* requires space in the register file to store each warp's stack. Determining how much register state to devote to each warp is non-trivial since increasing each warp's register stack can decrease parallelism. Therefore, we propose a novel allocation and renaming mechanism (Section III-A) that uses static and dynamic information to adapt the space devoted to register stacks in each thread block (Section III-B). *CARS* is generalizable, handling arbitrary control flow and function call depth (Section III-C) by leveraging the reconvergence stack and triggering software traps that fall back to traditional spills/fills when a warp's hardware stack is exhausted.

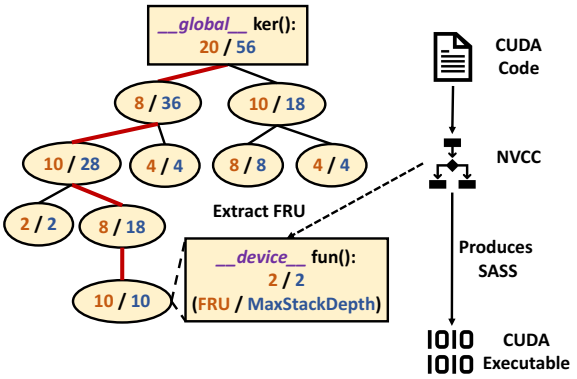


Fig. 4: *CARS* call graph analysis.

A. Register Stack Allocation and Renaming

In this subsection, we discuss how the register file is split among concurrent in-flight warps. An ideal allocator would freely share unused register space among all concurrent warps. However, implementing such an allocator in hardware is complex and costly. We leverage the fact that register spills/fills are implemented as a stack to design an efficient register file allocation mechanism. In the baseline GPU (Figure 3 (a)), each warp is given a contiguous region of the register file, based on static demand. The index for each warp's register (i.e., W0R3) is a simple base + offset computation of the warp's starting index plus the architectural register number. The base and size of each warp's allocation is set when a thread block is assigned to a core and does not change during the life of the thread block. *CARS* maintains this one-time partitioning of the register file among warps in a thread block. However, when the thread block is scheduled, *CARS* optionally assigns additional register space to each warp for an in-register stack. This space is contiguous with the base register allotment, and is not assigned to any registers until functions are called.

To manage register renaming as new functions are called, each warp is assigned a *Register Frame Pointer* (*RFP*) that keeps track of the warp's current function frame and a *Register Stack Pointer* (*RSP*) to keep track of the top of its stack. The operation is similar to a software stack. The *RSP* is increased/decreased when elements are pushed/popped (the stack grows up in our example). As functions are called, the *RFP* moves to the free region after the stack pointer. Figure 3 (b) shows an example function, namely *fun1*. Each warp is allocated a base amount of registers (*kernel* in Figure 3) + extra space for its in-register stacks. We call the additional registers needed for each function the *Function Register Usage* (*FRU*).

Fun1 has three callee-saved registers, R16, R17 and R18. The function call instruction pushes the current *RFP* to the stack, the local store instructions originally in fun1 are replaced with push instructions and the local loads with pop. *CARS* uses the indexing logic from the baseline ($Base[W_i] + R_j$, where i is the warp ID and j is the register number), with one small modification. Callee-saved registers (which are a contiguous set of architectural registers, starting at R16) are renamed if they have been pushed to the stack for the current function call. For a given register R_x, if $x >= R16$ and $x < R16 + RSP[W_i] - RFP[W_i]$, then the index for register x becomes: $RFP[W_i] + (x - 16)$. After the number of registers used for a warp exceeds the *registers per warp*, a software trap is triggered that spills/fills to local memory.

B. Determining the Register Stack Size

There are many ways to determine the number of registers to allocate to each warp's stack. More registers per warp means each SM will schedule fewer warps, but the chance of spills/fills decreases. To assign the stack size per warp, we explore three mechanisms:

1) **Low-watermark**: Low-watermark represents our design point with the most concurrency. Each warp is allocated enough register stack space to support *at least* one function call, on top of the base kernel. If another factor is limiting concurrency (i.e. shared memory, #thread blocks, etc.), then the leftover space is divided equally among the warps. In the worst-case, Low-watermark will trigger software spill/fill instructions on almost every function call/return.
2) **High-watermark**: High-watermark is our design point with the least concurrency. A full call graph analysis is performed to determine the register stack size needed to prevent all spills/fills in an acyclic call graph. However, the number of registers per warp of High-watermark can be large, severely limiting parallelism. In applications with cyclic call graphs (i.e., recursive applications), maximum register usage cannot be determined statically. We discuss recursive applications further in Section III-C. To guarantee forward process in High-watermark mode, *CARS* will context switch the register state for warps waiting at barriers if there are other warps in the same thread block waiting.
3) **NxLow-watermark**: NxLow-watermark allocates N *times* Low-watermark registers to each warp, it is a middle ground between High- and Low-watermark.

To make an initial prediction about which allocation mechanism will fit the workload best, we perform a lightweight analysis of the call graph. For each node in the call graph, the compiler computes the number of additional registers needed for each function i.e., the *FRU*. Figure 4 shows an example call graph. We list the FRU for each node in the graph, as well as the *MaxStackDepth*, which is simply the maximum number register that could be demanded by any path to a leaf node from this function.

The root node (i.e. the __global__ kernel function) *FRU* includes all the temporary and global registers available to all the functions. The compiler identifies each function's *FRU* and each function's individual MaxStackDepth. Note that this analysis is only applied to acyclic call graphs, for recursive applications, there is no MaxStackDepth. The bold path in Figure 4 delineates the High-watermark chain for the kernel. From the call graph analysis, Low-watermark would need 30

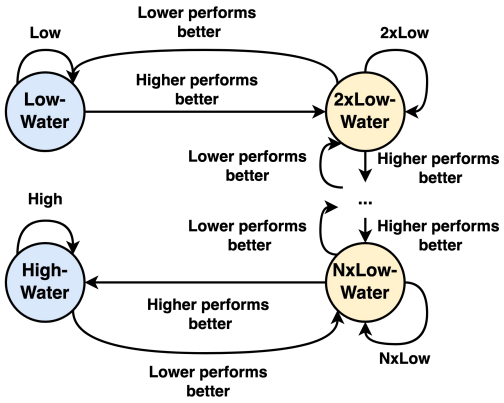


Fig. 5: *CARS*' dynamic register reservation state machine. The performance of threadblocks using different allocation policies is measured and recorded.

registers per warp (20 for the kernel + 10 for the largest function), and High-watermark would need 56 registers (the MaxStackDepth of the root function). To determine the final registers per warp, the other factors that limit occupancy must be known, which happens at kernel launch time. When a kernel is launched, we know what the maximum possible number of registers can be based on other limiting factors. For example, if the kernel is limited by shared memory to 8 warps per core, then the minimum register space available to each warp is $1/8$ of the register file. If this number is greater than the number of registers used in High-watermark, then all warps are given this large allocation, because there is register space to spare. If this number is less than the High-watermark allocation, then a dynamic selection mechanism is used.

For kernels where High-watermark allocation is not possible without limiting occupancy, *CARS* begins with half of the SMs running Low-watermark mode and half running High-watermark mode. After each thread block's execution, *CARS* records the performance. As thread blocks finish, the average thread block performance for both modes is computed. Once one thread block from each of High- and Low-watermark is complete, *CARS* begins employing a state machine, illustrated in Figure 5 to newly spawned thread blocks. If High-watermark performs better, new thread blocks launched on the Low-watermark SM will be launched with a 2xLow-watermark. Conversely, if Low-watermark performs better, the High-watermark SMs will spawn thread blocks with fewer registers allocated. Each thread block's performance at each allocation level is continuously monitored and recorded. At the end of each kernel launch, the average performance of all executed flavors is remembered, and the best-performing allocation represents the starting point for the next invocation of the same named kernel. If the current selection performs worse than the recorded performance of a higher or lower allocation, the allocation policy is adjusted accordingly.

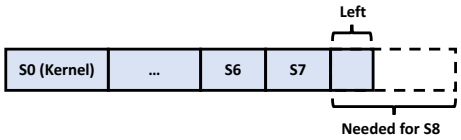


Fig. 6: A circumstance when spilling other stacks is necessary.

C. Handling Control Flow Divergence and Recursion

There are three instances where control flow divergence can affect *CARS*: (1) when warps experiencing control flow divergence call a direct function, (2) when threads within a warp return from a function at different points, and (3) when an indirect call sends threads in the same warp to different functions. In the case of (1), all the threads calling the function will traverse the same control-flow path and full warp-wide registers will be allocated based on the registers requested by the function. This solution wastes register space in the empty lanes, but simplifies the register allocation mechanism, and is consistent with the baseline. For (2), some threads will finish the function early, however, they will not be allowed to proceed until all threads have returned from the function. Similarly, the register stack cannot be released until all threads return. Finally, for (3), indirect functions are relatively rare (typically used to implement virtual functions or function pointers), the challenge is that the compiler does not know what function will be called statically. Here we use the highest register usage count for all the dynamic function calls that belong to the same static function call point to perform the static analysis.

For recursive workloads, we assume only one iteration of the recursive components and assign the MaxStackDepth accordingly. In such a case, High-watermark will not guarantee zero spills/fills, as the static analysis cannot account for the dynamic call depth.

IV. Compiler and Architecture Modifications

This section details our compiler changes in Section IV-A and architectural changes in Section IV-B.

A. Compiler Modifications

The compiler embeds the following information in the binary to enable efficient stack management by the hardware:

- For the kernel function: it embeds the *FRU* into the kernel launch parameter.
- For a device function call: it embeds the *FRU* into each relocatable call instruction so the space for the next frame can be efficiently allocated. To enable the restoration of the frame on the function's return, a push μop is inserted before every call instruction to push the *RFP* of the caller onto the stack. After each relocatable call instruction, the registers to be renamed and allocated are listed in pushes.
- For a device function return: it embeds the *FRU* into each return instruction. Registers are restored and deallocated via pops.

The compiler inserts each function's *FRU* into the call/return instructions so that the hardware knows (before the function

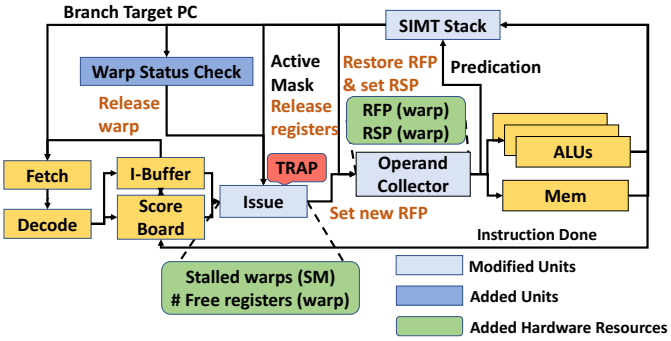


Fig. 7: Overall architecture design.

starts executing) if there is sufficient space on the stack. When space runs out in a warp's register stack (depicted in Figure 6), a software trap will be triggered. In Figure 6, the shaded region shows the total register space available to a particular warp, where *S#* is the stack usage of a particular function. The warp has called seven functions and has little register stack space left. When the warp initiates a call to another function requiring a stack of the size indicated by the dotted outline, insufficient registers remain to accommodate *S8*. *CARS* will keep the top of the stack resident in the register file, evicting the data at the bottom. All register contents within *S1* (not shown on the graph) will be spilled to memory in a wrap-around fashion. If the size of *S1* is still insufficient, more stacks will be spilled. When the function corresponding to *S8* returns, *S8* will be released, and the registers spilled will be filled back when the corresponding function is back in control. In general, every warp utilizes a circular register stack to avoid frequent register spills and fills upon function calls and returns.

B. Architectural Design

In this subsection, we discuss the architectural details of the *CARS* design. We first introduce the overall flow, then detail how the existing GPU units are modified and what new units and hardware resources are added. And we finish off by examining the extra area cost and timing overhead.

1) Overall Flow: Figure 7 shows the overall architecture design of *CARS*. Three units are modified: *issue*, the *operand collector*, and the *SIMT stack*. One additional unit is introduced, the *warp status check* unit.

To avoid modifying the global thread block scheduler, *CARS* limits concurrency by blocking the execution of warps not allocated registers in the issue scheduler. When an instruction is in the issue stage, the issue scheduler first checks whether the warp is in a stalled warp list of warps that are not executing because they have not been allocated space in the register file. The instruction is not issued if its corresponding warp is deactivated. For the executing warps, if the instruction issued is a function call, it pushes the caller's *RFP* to the callee's register stack and sets the *RFP* to the current *RSP*. The number of free registers in each warp's register stack is also stored in the issue stage, updated following each call/return. A trap can be triggered if insufficient registers are left for the warp to allocate a register stack for the function call. If triggered, the software initiates register spills and fills.

The *operand collector* is augmented with the register renaming logic, which is responsible for renaming the registers based on the warp's *RFP* (Section III-A). Our augmentation is implemented as a small logic change to the baseline GPU.

After the instruction finishes execution, it goes through the *SIMT Stack* stage. Since it is possible that different threads can return at different points in the function due to control flow divergence, we augment the *SIMT Stack* entries with a single bit to identify whether it is a function call (Section IV-B2). Once all of the threads have returned from a function, the *SIMT Stack* informs the *Operand Collector* unit to set the *RSP* to the current *RFP*, and then the caller's *RFP* is restored. It also informs the issue stage to update the number of free registers.

The instruction information also goes through the *warp status check* unit, which checks whether the warp has finished. If a warp has finished, the unit will inform the issue stage to release another warp if one is waiting.

2) Microarchitecture Details: **Issue**: As described in Section IV-B1, apart from the normal issue operations, *issue* is also responsible for recording the number of free registers left in each warp. This information decides whether to trigger a trap to spill out or fill back a register stack. The unit also stores the IDs for the deactivated warps in the stalled-warp list and stops issuing instructions from these warps. Once it receives the release signal from the *warp status check* unit, it proceeds to release a warp. Moreover, if a barrier is encountered and there are deactivated warps, a trap is activated to switch out a warp's register state, preventing potential deadlocks. A trap is also triggered when insufficient registers exist (Section IV-A).
SIMT Stack: The *SIMT Stack* is an existing structure used to keep track of control flow divergence due to the lock-step execution of multiple threads within one warp. As introduced in Section III-C, there are three instances when control flow divergence can happen for function calls. *CARS* works without issue except when threads from the same warp return at different points in a function. *CARS* cannot release the function's register frame until all threads return. To handle multiple returns, *CARS* adds 1 bit to each SIMT stack entry set for the function call's stack SIMT stack entry. If control flow diverges inside a function call, additional entries will be pushed onto the SIMT stack. When a subset of threads return from the function, *CARS* will delay the frame's deallocation until the SIMT stack is returned to the function call entry (identified by our added bit).
Warp Status Check: The functionality of the *warp status check* checks whether a warp has finished. It is used when there could be warp de-scheduling and reactivating. Every time a warp finishes, it sends the release signal to the *issue* unit and releases one waiting warp from the same kernel.

C. Overall Cost

The hardware resources added to our design are a list of stalled warps (per SM), a register for the number of free registers (per warp), an *RFP* (per warp), and an *RSP* (per

warp). There can be a maximum of 64 concurrent warps running on each SM. Although they cannot all be stalled simultaneously, we still allocate space for 64 in our estimate. The list of stalled warps takes $64 \times 6b = 48B$. The number of bits needed to record the free registers can be formulated as $\log_2 \frac{2048}{\#warps} * \#warps$. Since this function increases monotonically, the max size of the number of registers is $\log_2 \frac{2048}{64} * 64 = 40B$. The cost of *RFP* and *RSP* is $64 * 11b * 2 = 176B$. So, the maximum possible total storage cost is only 264B.

From a timing perspective, in the *issue* stage, *CARS* checks the number of free registers only upon each function call. Rarely, it may perform context switches to avoid register spills/fills; only one kernel we evaluated required this. In this case, we need to check the stalled warps list before issuing the instruction (Section VI-B). The warp status check unit is only initiated when an *EXIT* instruction is issued to check whether a warp has finished. Similarly, this only happens when the kernel requires context switches. The push and pop instructions modify the *RSP* as they pass through the *operand collector*. We add an extra pipeline cycle to the *issue* and *operand collector* stage to model these additional checks, making a worst-case assumption that they affect cycle-level timing. However, we anticipate these simple operations are unlikely to affect the SM's critical path.

V. Experimental Methodology

A. Experiment Setup

We evaluate the effectiveness of *CARS* using Accel-Sim [24] and AccelWattch [21], a cycle-level simulator, and a power modeling framework. The traces are generated using NVBit [57]. Workloads other than Cutlass and Rapids are compiled with one main compilation module and one library module that contains common device functions. We maintain the file structure of Cutlass and Rapids, but inter-module function calls are not inlined. The modules are compiled with the NVCC separate compilation flag (-dc) using CUDA 11.4.

B. Workloads for Experiments

We study 22 representative function calling applications and ML layers from six different benchmark suites: LoneStar [5], Rodinia [6], ParaPoly [62], Department of Energy Apps [22], Cutlass [39] and Rapids [9]. To evaluate *CARS* on recursive applications, we also include a Fibonacci [4] workload in our experiments. Note that FIB is included only as a demonstration that *CARS* executes correctly in the presence of a cyclic call graph, which does not exist in other workloads.

The 22 applications and ML layers cover different application areas, such as model simulations [62], raytracing-based rendering [45], compiler analysis [12], and machine learning [1], [2], [9], [33], [39], [44], [46]. To execute the layers of our DNNs, we use the highly optimized Cutlass library from NVIDIA [39], selecting the best-performing GEMM kernel that uses tensor cores. Cutlass is the most optimized open-source GPU implementation of GEMM operations with performance that rivals the closed-source cuBLAS [40]. We use the cutlass kernels, as opposed to cuBLAS, because Cutlass is open-source, and can be recompiled.

TABLE I: 22 function calling workloads from six different benchmark suites.

Workload	Call Depth	CPKI
LoneStar Benchmark Suite [5]		
Points-to Analysis (PTA)	9	46.11
Delaunay Mesh Refinement (DMR)	1	11.61
Minimum Spanning Tree (MST)	5	20.75
Single-Source Shortest Paths (SSSP)	3	6.30
Rodinia Benchmark Suite [6]		
CFD Solver (CFD)	3	17.48
ParaPoly Benchmark Suite [62]		
Traffic (TRAF)	3	3.13
Game of Life (GOL)	1	7.05
NBody (NBD)	2	21.40
Collision (COLI)	3	19.54
Structure (STUT)	3	10.94
Raytracing (RAY)	4	19.71
Department of Energy Apps [22]		
Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)	3	2.84
Recursive Workload [4]		
Fibonacci (FIB)	8	22.41
MLPerf Workloads [33], [39], [46]		
Bert Linear Transformation Layer (Bert_LT)	5	17.01
Bert Attention Score (Bert_Atscore)	5	17.62
Bert Attention Operation (Bert_Atop)	5	17.48
Bert Fully-Connected Layer (Bert_FC)	5	17.01
Resnet50 Forward Propagation (Resnet_FP)	5	17.04
Resnet50 Weight Gradient (Resnet_WG)	5	16.91
Rapids Workloads [9]		
Support Vector Regression (SVR)	17	47.03
K-Means Clustering (KMEAN)	14	41.23
Random Forests Classifier (RF)	17	47.11

The workloads are massively parallel, and some of them exhibit irregular behaviors. This diversity shows that even though the applications have different properties in other aspects, they all suffer from the memory and bandwidth pressures resulting from register spills and fills. The workloads are listed in Table I, which also lists the call depth and calls per thousand instructions (*CPKI*). The call depth and CPKI numbers vary across the 22 workloads, with Rapids workloads having the highest count. This is due to their heavy usage of multiple CUDA libraries, resulting in increased call depth.

C. Simulation Methodology

We utilize Accel-Sim [24] and AccelWattch [21], a trace-driven simulator and power modeling framework for the experiments, and the main simulation methodology is as below:

Compiler Analysis: We first dump the call graph information using the *-dump-callgraph* nvlink option during compilation to get the register usage information for the base kernels. Then, we analyze the SASS code to find the number of registers spilled for each function, which we label as the *Function Register Usage* (*FRU*). We dump the ELF files from the workloads' binaries and perform static analysis on them to retrieve global (kernel) and device function information. The symbol tables can be parsed and extracted from the ELF file.

The function call graph analysis and *CARS* prediction are then conducted as introduced in Section III-A.

Register Spills and Fills Trap Triggering: We use Accel-Sim to implement the core design of *CARS*, ensuring that our implementation identifies the moments to initiate register spills/fills, and seamlessly injects the corresponding instructions for trap triggering.

D. Techniques Studied

Baseline (V100): We simulate the NVIDIA V100 GPU as our baseline design, including all the register spills and fills introduced by nvcc. The V100 configuration in [24] is used.
Idealized Virtual Warps: We model an idealized version of a previous work, Zorua [56], where the GPU has an unlimited number of registers, shared memory, and thread block slots.
10MB L1: We extend the cache size for each SM from 128KB to 10MB, which is large enough to eliminate the overhead due to cache capacity misses for most of the workloads.
Static Wavefront Limiter (Best-SWL): This scheduling technique limits the number of concurrent warps on each SM core. We perform experiments on six warp counts (1, 2, 3, 4, 8, and 16) and choose the highest performing limit number, which outperforms *CCWS*. This technique limits parallelism to avoid cache capacity oversubscription [47].
***Concurrency-Aware Register Stacks* (*CARS*)**: Our main design as described in Sections III and IV.

VI. Experimental Results

In this section, we evaluate *CARS* and related work and investigate the causes of the observed performance.

A. Performance Improvement Analysis

Figure 8 shows the performance comparison for 22 workloads among *Baseline (V100)*, *Idealized Virtual Warps*, *10MB L1*, *Best-SWL* and *CARS*. Compared to the baseline, *CARS* demonstrates the most performance improvement at a geomean 26%. The register spills/fills problem cannot be simply resolved by assigning unlimited hardware resources to the core, eliminating cache capacity misses, or limiting the SM warp count to avoid cache oversubscription. To further analyze the results, Table II enumerates the varied reasons why *CARS* improves the performance of our workloads.

1) L1D Capacity and Contention: We analyze the performance improvement of *CARS* using the *10MB L1* and *Best-SWL* configurations here specifically. *10MB L1* can reduce the cache misses due to the capacity limit, while *Best-SWL* shows the best-performing configuration that has been fine-tuned for cache contention purposes by controlling the number of in-flight warps. Compared to the baseline, DMR, MST, CFD, GOL, STUT, and Resnet_FP show performance improvement with both of the configurations, while Bert_LT, Bert_FC and Resnet_WG only show speedup with 10MB L1.

Because *CARS* can effectively eliminate the majority of register spills/fills memory accesses, it can alleviate pressure on cache capacity. Figure 9 demonstrates the reduction in memory accesses with *CARS* compared to the baseline. On average, the fraction of register spills/fills decreases by 40%. The global load and store accesses are unaffected since *CARS* only reduces local memory instructions. Some of the workloads use local memory for other purposes. Each workload in this class significantly reduces memory access, contributing to the observed speedup.

Also, because *CARS* can limit concurrency in exchange for a higher number of registers allocated per warp, the speedup also results from alleviating inter-warp cache capacity interference by reducing local memory accesses and limiting parallelism. DMR demonstrate more pronounced performance improvements with the *Best-SWL* and *10MB L1* configurations. This occurs because *10MB L1* and *Best-SWL* can capture more global access locality, given the prevalence of global accesses in those workloads.

The *10MB L1* configuration effectively accounts for the improvement observed in Bert_LT, Bert_FC, and Resnet_WG for *CARS*, whereas *Best-SWL* shows minimal speedup for most layers. This discrepancy arises because ML workloads generally have larger data footprints that *Best-SWL* fails to accommodate. Nevertheless, *CARS* delivers performance similar to the *10MB L1* in these scenarios by mitigating the spills/fills.

2) L1D Bandwidth Contention: To further explain *CARS*' performance improvement, we explore an ideal ALL-HIT configuration, where every register spill/fill memory access hits in the L1D cache without traversing the cache. To keep the L1D cache bandwidth usage the same, all accesses still suffer the cache hit latency. We show the performance comparison of ALL-HIT and *CARS* in Figure 10.

We observe that the workloads experiencing the cache capacity and contention problems generally have outstanding speedup with ALL-HIT. However, for some workloads, like PTA, SSSP, TRAF, FIB, and Rapids, while increasing the cache capacity does not have much effect on performance, ALL-HIT explains the performance improvement of *CARS*. This is due to the high absolute value of the number of memory accesses, with the majority being local memory accesses resulting from register spills/fills. Due to the interference caused by these local memory accesses, the global data cannot consistently reside in the cache, leading to cache misses.

Figure 11 plots L1D bandwidth contention versus time. Here, we show a timeline graph of the L1 cache bandwidth on global and local accesses for both the baseline and *CARS* for one of the kernels in PTA. For this kernel, *CARS* does not eliminate all the register spills/fills traffic. The average global memory access bandwidth is also shown on the graph, and this number in *CARS* is 98% higher than in the baseline. With less interference of register spills/fills, the core pipeline can issue more global memory instructions per unit time.

3) Low Occupancy: For applications like Bert_Atscore and Bert_Atop, neither the 10MB L1 cache, Best-SWL, nor ALL-HIT configurations exhibit speedup comparable to *CARS*. Our observation is that these workloads have low occupancy on each SM, so there are not enough warps running concurrently on the SM to hide the memory load latency entirely. The reason why Bert_Atscore and Bert_Atop have lower

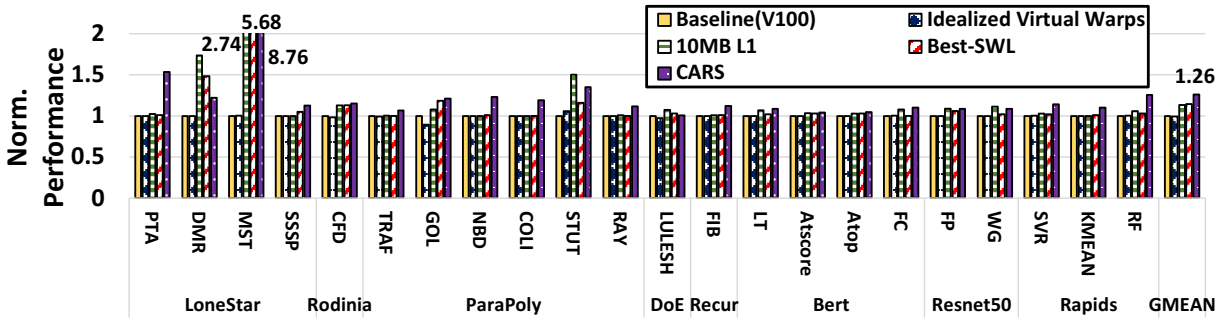


Fig. 8: Performance of Idealized Virtual Warps, 10MB L1, Best-SWL and *CARS*, normalized to the performance of the original applications with register spills and fills.

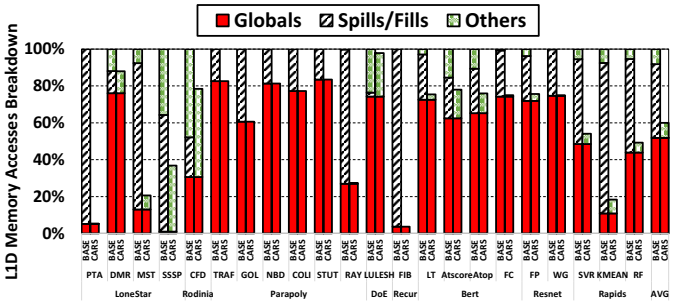


Fig. 9: Memory accesses broken down into spills/fills, globals, and other locals that are not spills/fills.

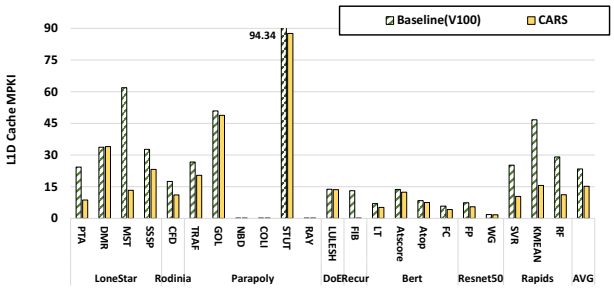


Fig. 12: L1D MPKI for Baseline (V100) and *CARS*.

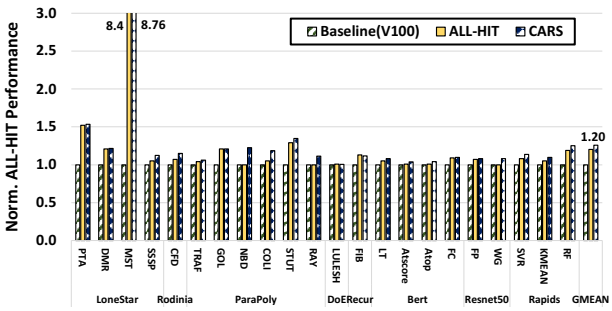
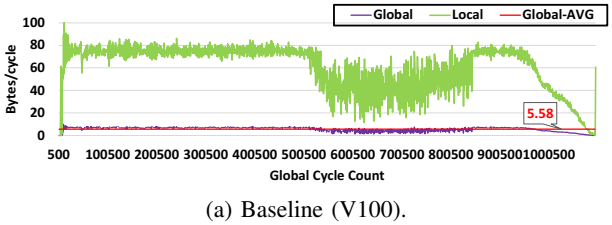
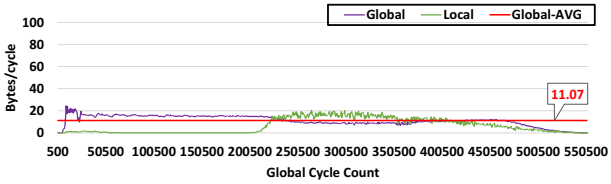


Fig. 10: Performance improvement of an ALL-HIT cache.



(a) Baseline (V100).



(b) *CARS*.

Fig. 11: Global and local bandwidth comparison for baseline and *CARS* for one of the kernels of PTA.

occupancy compared to the other layers is that they involve smaller-sized GEMMs due to the multi-head design [55]. With *CARS*, the substantial reduction in local memory accesses significantly reduces the number of cycles stalled due to the load dependencies for these two workloads.

We plot MPKI (misses per thousand instructions) in Figure 12 to further indicate the speedup gained by *CARS*. On average, *CARS* experiences a 35% reduction in MPKI. While most of the applications exhibit a notable decrease, NBD, COLI, and RAY show low MPKI values. These three workloads and LULESH demonstrate only a marginal reduction compared to the baseline. To explain this, we plot Figure 13 to show the breakdown of different types of instructions for both *CARS* and the baseline. We can see that the fraction of register spills/fills instructions is low for these four workloads.

In general, among the 22 workloads, MST performs best. While it may not exhibit the most significant reduction in memory accesses, *CARS* effectively mitigates issues related to its limited L1D capacity and inter-warp cache resource contention, which can be indicated by its high reduction of L1D cache miss frequency. On the contrary, LULESH barely has any speedup, attributed to its relatively low count of absolute register spills/fills instructions and memory accesses.

GOL performs worse than the baseline with *Idealized Virtual Warps*, and this is because not limiting the number of concurrent thread blocks running on the core can thrash the cache. Generally speaking, *CARS* out-performs the realistic related work *Best-SWL*, and it also out-performs the unrealistic *10MB L1*, *Idealized Virtual Warps* and *ALL-HIT* configurations.

TABLE II: Main factors contributing to the improved performance of *CARS* across the 22 workloads studied.

Workload	*CARS* Main Speedup Factors
PTA	L1D bandwidth contention
DMR	L1D capacity and contention
MST	L1D capacity and contention
SSSP	L1D bandwidth contention
CFD	L1D capacity and contention
TRAF	L1D bandwidth contention
GOL	L1D capacity and contention
NBD	L1D bandwidth contention
COLI	L1D bandwidth contention
STUT	L1D capacity and contention
RAY	L1D bandwidth contention
LULESH	Low total local memory access count
FIB	L1D bandwidth contention
Bert_LT	L1D capacity
Bert_Atscore	Low occupancy
Bert_Atop	Low occupancy
Bert_FC	L1D capacity
Resnet_FP	L1D capacity and contention
Resnet_WG	L1D capacity
SVR	L1D bandwidth contention
KMEAN	L1D bandwidth contention
RF	L1D bandwidth contention

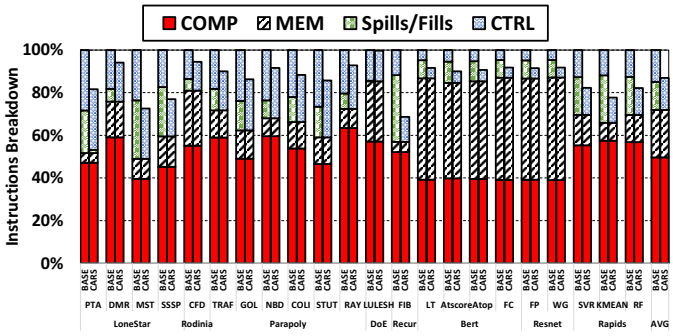


Fig. 13: Instruction frequency, normalized to the baseline.

B. Allocation Mechanisms and Context Switching

Figure 14 shows the performance improvement of the kernels within one iteration of PTA, normalized to the baseline. Since each kernel can have different High-watermark and Low-watermark register allocations, the number of possible allocation mechanisms varies. The x-axis represents various kernels, each with its distinct allocation mechanisms. Over half of the kernels show no performance improvement over the baseline, primarily due to the absence of any register spills/fills within these kernels. Consequently, Low- and High-watermark yield the same results. High-watermark performs worse than Low-watermark in K3, K14 and K21. To explain this, High-watermark for these kernels needs to perform context switches due to the existence of barriers, which incur register spills/fills traffic. K1 also needs context switches, but the High-watermark configuration delivers better performance. This is attributed to the high function call depth of K1, where Low-watermark needs to spill/fill the allocated stack frequently to accommodate the depth of functions. K2 and K13 show barely any performance differentiation across various configurations, primarily due to their shallow function call depth and the small disparity between High- and Low-watermark. Additionally, for K22, the total number of thread blocks to be dispatched is lower than the total number of SMs. Thus, High- and Low-watermark dispatch the same number of thread blocks on each SM for these three kernels, so the performance does not change.

Out of all the kernels we evaluated across the 22 applications, only one requires context switches, which is K1 in PTA. Three reasons account for why just this single kernel opts for context switches. First, the absence of barriers obviates the need for context switches in kernels. Second, few kernels choose High-watermark, as Low-watermark and NxLow-watermark configurations are already effective in reducing most register spills/fills without compromising parallelism. Only kernels that either have a high function call depth or inter-warp cache contention would prefer High-watermark. Lastly, as shown in Figure 14, *CARS* avoids selecting High-watermark if context switches have a detrimental impact.

C. Register Spill/Fill Frequencies

TABLE III: Software trap handling frequency and severity.

Workload	Fraction of functions that invokes trap handler	# of bytes spilled/filled per function call
PTA	0.014%	0.78

We show applications that still have register spills/fills after applying *CARS* in Table III. While the fraction of functions with spills/fills reflects only the proportion of functions that undergo stack spills/fills due to insufficient registers, # bytes spilled/filled per function call includes both the context switches and stack spills/fills.

There are two reasons why only PTA exercises spilling with *CARS*. First, if Low-watermark is preferred, there typically exists a minimal number of register spills/fills. Second, in the case where Low-watermark still produces a substantial number of register spills/fills, and if the application allows for a sacrifice of parallelism without encountering any barriers, High-watermark is preferred, which means the complete elimination of register spills/fills in non-recursive kernels. Therefore, register spills/fills only happen when there is a trade-off between parallelism and the number of registers allocated for each warp, or when context switches occur. Our recursive FIB benchmark does not spill/fill because the call depth with our input is low. Increasing the call depth of FIB by changing the input triggers spills/fills with *CARS*.

D. Energy Efficiency

Figure 15 plots the energy efficiency of *10MB L1*, *Best-SWL*, *Idealized Virtual Warps* and our *CARS* design. On average, CAR is 28% more energy efficient. The figure also matches what we found in the previous figures, which show that MST suffers most from register spills/fills. However, MST does not get as much energy reduction as its performance

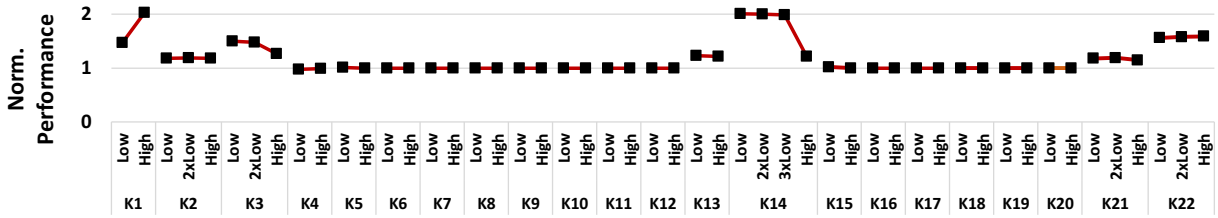


Fig. 14: *CARS* performance improvement over different allocation mechanisms for PTA.

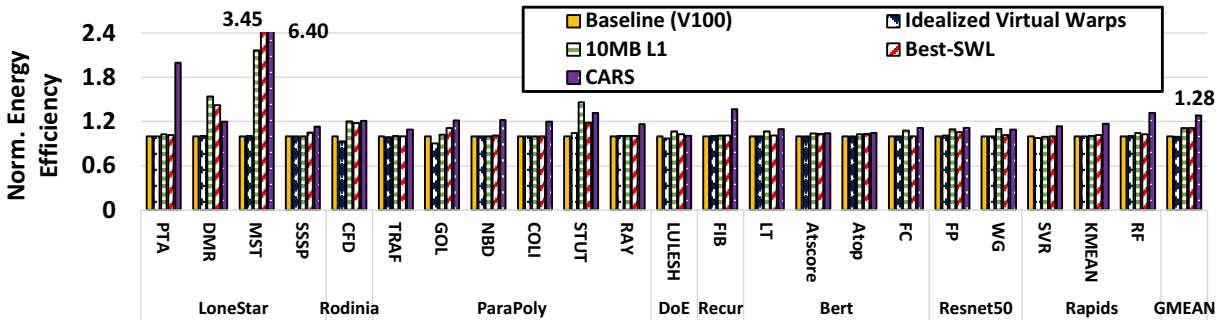


Fig. 15: Energy efficiency (higher is better), normalized to the V100 baseline.

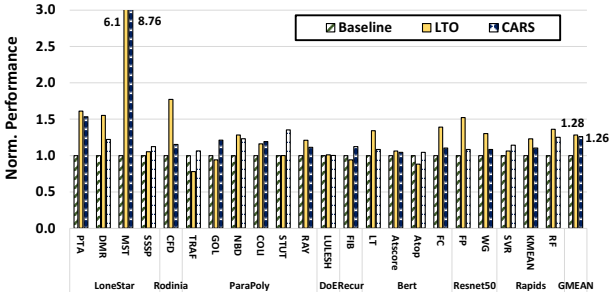


Fig. 16: Performance of the fully inlined (LTO) code and *CARS*, normalized to the baseline V100.

improvement, and this is because *CARS* can limit concurrency. Although the L1D cache pressure is relieved by not having as many warps running on the core simultaneously, more core resources are used to run the less paralleled workload. However, the total energy used still decreases due to the execution time reduction. Overall, the energy gains are slightly better than performance owing to both a reduction in static energy (caused by a reduced runtime) and a reduction in core activity by not moving spills and fills around.

E. Sensitivity Studies

Although the goal of *CARS* is to perform efficient function calls at runtime, avoiding the downsides and lack of flexibility full inlining places on the program, understanding the performance of *CARS* versus a statically inlined version of the code helps to put the work in context and provides a best-case scenario for *CARS* in some workloads. Figure 16 plots the performance of fully inlining all functions (using LTO) and *CARS*. As shown in Figure 16, the fully-inlined code performs 28% better than the baseline on average, while *CARS* has a 26% improvement. Most applications perform better with fully-inlined code, as function inlining offers more optimization opportunities beyond just eliminating register spills/fills. However, there are several applications where the fully-inlined code performs worse. This is because fully inlining increases the instruction footprint, which raises the GPU's front-end pressure, and in some applications, the concurrency-limiting properties of *CARS* reduce inter-warp memory contention that will continue to be present in the fully-inlined code.

Figure 17 plots the performance of the baseline and *CARS* as L1D cache ports are scaled from from 2x to 8x. On average, the three bandwidth settings have a speedup of 1.02x, 1.03x, and 1.03x over the baseline, compared to *CARS*' speedup of 1.28x, 1.29x, and 1.29x at the same bandwidth level. Only applications sensitive to cache hit bandwidth, without significant miss pressure, see a rise in performance, as cache miss bandwidth and resources remain constant.

Figure 18 plots the performance of *CARS* when applied to an Ampere RTX 3070 GPU. Most applications exhibit a speedup similar to V100, except MST, which shows a lower speedup. On the V100, *CARS* selects the High-watermark configuration, maximizing register locality while achieving sufficient concurrency. On the RTX 3070, changes in occupancy cause *CARS* to select the Low-watermark mode, resulting in more memory accesses and a slightly lower speedup. However, *CARS*' overall speedup is resilient on a more recent architecture.

VII. Related Work

GPU Programmability: A body of work exists on providing CPU-like programmability for GPUs. Support for a file system abstraction [52], network stack [26], and more advanced memory management [51] are examples of this. ParaPoly [62] created a set of workloads to emphasize the

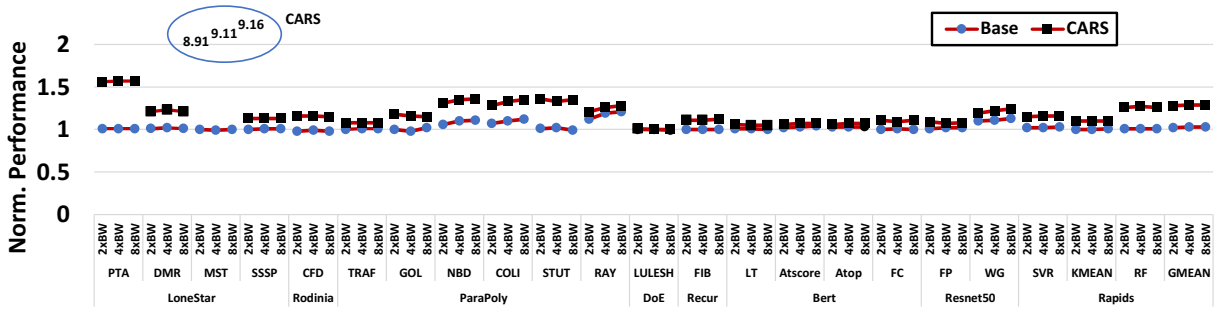


Fig. 17: Performance of increased L1 cache bandwidth and *CARS* sensitivity, all normalized to the baseline.

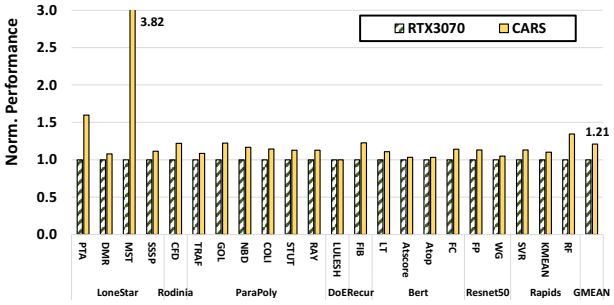


Fig. 18: *CARS*' performance on an Ampere RTX3070.

bottlenecks of GPU virtual function calls, and techniques have been proposed to mitigate the effects of calling virtual functions on GPUs [63]. Lee et al. [30] exploit orthogonal programming characters, such as inter-CTA locality and concurrent execution, to improve performance.

Functions Optimizations at Link and Runtime: NVIDIA introduced separate compilation with link-time optimization [35] in CUDA 11.2, so that the performance of applications with multiple source files and libraries does not suffer. A large number of research works are devoted to compilation optimization; for example, Chen et al. proposed AutoFDO [7], a Feedback based link-time optimization that can tolerate the staleness of the profile used by the next release, and Johnson et al. proposed ThinLTO [19], a scalable framework designed for Cross-Module Optimization.

Techniques for Managing CPU Register Files: Register renaming [54] resolves false data dependencies by remapping logical registers to physical ones. Oehmke et al. propose a new register file architecture that treats the physical register space as a cache of a much larger architectural register space [42], reducing memory accesses. SVF [29] by Lee et al., similar to *CARS*, focuses on stack accesses. Zhuang et al. [64] suggest a hardware-managed register allocation for embedded systems. Yan and Zhang [61] introduce virtual registers to ease register pressure. Jordan et al. [20] propose a software-managed stack cache, using a scratchpad memory. All these works explore architecture limited to a few threads. An alternative hardware mechanism to avoid explicit spill/fill accesses is to use register windows [27], [28], [58]. However, register windows see little use today due to inefficiencies in aggressive register management, resulting in wasted registers which are a precious resource in GPUs.

Techniques for Managing GPU Register File: Xie et al. proposed the CRAT compiler framework [60] to coordinate register allocation for TLP and spill registers to shared memory. Sakdhnagool et al. proposed RegDem [50], which identifies register allocation as a main factor in limiting occupancy and mitigates register demand in function-free programs. Jeon et al. proposed register file virtualization [14], utilizing register renaming to reduce register file size and save energy. Vijaykumar et al. proposed Zorua [56] to virtualize GPU on-chip resources, such as registers, scratchpad memory, and thread slots. All of these prior works attempt to make better use of on-chip resource capacity, while *CARS* explicitly targets both the capacity and bandwidth bottleneck introduced by excessive spills/fills on function calls.

VIII. Conclusion

We perform the first quantitative analysis of direct function calls on GPUs, demonstrating that register spills/fills are a problem. We propose *Concurrency-Aware Register Stacks* for efficient GPU function calls, a lightweight register stack mechanism for massively parallel architectures. By balancing concurrency with spills and fills *CARS* eliminates much of the data movement between the register file and memory required to maintain the GPU's RISC-like ABI. *CARS* uses a lightweight call graph analysis and an adaptive prediction mechanism to choose among allocation schemes that balance concurrency and register stack locality.

Experimental results demonstrate that *Concurrency-Aware Register Stacks* delivers 1.26× performance improvement with 28% better energy efficiency, adding negligible area overhead. Notably, *Concurrency-Aware Register Stacks* outperforms several idealized stackless systems with an impractical amount of hardware resources. As the first paper to study hardware for direct function call support on GPUs, *CARS* opens a new design space for optimizing function calls on GPUs.

Acknowledgment

We would like to thank our anonymous reviewers for their feedback, which helped improve this paper. This work was supported, in part, by NSF CCF #1943379 (CAREER) and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA.

REFERENCES

[1] "BERT — huggingface.co," https://huggingface.co/docs/transformers/model_doc/bert, [Accessed 23-02-2024].

[2] "Pretrained Models 2024; Transformers 3.4.0 Documentation — huggingface.co," https://huggingface.co/transformers/v3.4.0/pretrained_models.html, [Accessed 23-02-2024].

[3] User Guide for AMDGPU Backend. https://llvm.org/docs/AMDGPUUsage.html#non-kernel-functions.

[4] "Fibonacci GPU," https://github.com/branstanley/Fibonacci_GPU, 2016, accessed July 15, 2023.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151. [Online]. Available: https://doi.org/10.1109/IISWC.2012.6402918

[6] 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 *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[7] D. Chen, T. Moseley, and D. X. Li, "AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications," in *International Symposium on Code Generation and Optimization (CGO)*, 2016, pp. 12–23.

[8] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu, "Adaptive Cache Bypass and Insertion for Many-core Accelerators," in *Proceedings of International Workshop on Manycore Embedded Systems*, 2014, pp. 1:1–1:8.

[9] N. Corporation, "cuml: Rapids machine learning library," *GitHub Repository*, 2024. [Online]. Available: https://github.com/rapidsai/cuml

[10] D. Das, "Function Inlining versus Function Cloning," *ACM SIGPLAN Notices*, vol. 38, no. 6, pp. 23–29, 2003.

[11] D. Detlefs and O. Agesen, "Inlining of Virtual Methods," in *Proceedings of the 13th European Conference on Object-Oriented Programming*, 1999. [Online]. Available: https://dl.acm.org/doi/10.5555/646156.679839

[12] M. Emami, "A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing C Compiler," 1993.

[13] S. Gray, "Nervana systems maxas," https://github.com/NervanaSystems/maxas/wiki/Introduction, [Accessed 23-02-2024].

[14] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU Register File Virtualization," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015, pp. 420–432.

[15] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014, pp. 272–283.

[16] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," 2018.

[17] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013.

[18] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[19] T. Johnson, M. Amini, and X. David Li, "ThinLTO: Scalable and Incremental LTO," in *International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 111–121.

[20] A. Jordan, S. Abbaspour, and M. Schoeberl, "A Software Managed Stack Cache for Real-Time Systems," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 319–326. [Online]. Available: https://doi.org/10.1145/2997465.2997488

[21] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, and T. G. Rogers, "AccelWattch: A Power Modeling Framework for Modern GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. ACM, 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3466752.3480063

[22] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "LULESH Programming Model and Performance Ports Overview," Tech. Rep. LLNL-TR-608824, December 2012.

[23] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[24] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, 2020. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00047

[25] Khronos Group, "OpenCL," http://www.khronos.org/opencl/, 2013.

[26] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*, 2014. [Online]. Available: https://doi.org/10.1145/2963098

[27] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[28] T. Lang and M. Huguet, "Reduced Register Saving/Restoring in Single-Window Register Files," 1986. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/381711.381714

[29] H.-S. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson, "Stack Value File: Custom Microarchitecture for the Stack," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 5–14.

[30] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014, pp. 260–271.

[31] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014, pp. 175–186.

[32] R. Leupers and P. Marwedel, "Function Inlining Under Code Size Constraints for Embedded Processors," in *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051)*. IEEE, 1999, pp. 253–256.

[33] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia, "Mlperf training benchmark," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 336–349. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2020/file/411e39b117e885341f25efb8912945f7-Paper.pdf

[34] S. McFarling, "Procedure Merging with Instruction Caches," vol. 26, no. 6, pp. 71–79, 1991.

[35] M. Murphy and A. Sundaram. (2021) Improving gpu application performance with nvidia cuda 11.2 device link time optimization. [Online]. Available: https://developer.nvidia.com/blog/improving-gpu-app-performance-with-cuda-11-2-device-lto/

[36] NVIDIA, "cudf," *GitHub repository*, 2024.

[37] NVIDIA, "cuGraph," *GitHub repository*, 2024.

[38] NVIDIA, "cuspatial," *GitHub repository*, 2024.

[39] NVIDIA. (2024) Nvidia cutlass library. [Online]. Available: https://github.com/NVIDIA/cutlass?tab=readme-ov-file

[40] "NVIDIA cuBLAS," https://developer.nvidia.com/cublas, NVIDIA Corp., 2016, accessed August 6, 2016.

[41] NVIDIA Corp., "NVIDIA CUDA C Programming Guide," https://docs.nvidia.com/cu-da/cuda-c-programming-guide/index.html, NVIDIA Corp., 2020, accessed August 6, 2020.

[42] D. Oehmke, N. Binkert, T. Mudge, and S. Reinhardt, "How to Fake 1000 Registers," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005, pp. 12 pp.–18.

[43] Open-source community, "OpenAcc," https://www.openacc.org/, OpenACC.org, 2019, accessed April 15, 2019.

[44] S. Pati, S. Aga, N. Jayasena, and M. D. Sinclair, "Demystifying BERT: Implications for Accelerator Design," 2021.

[45] Peter Shirley, "Ray Tracing in One Weekend," https://github.com/petershirley/raytracinginoneweekend, 2018, accessed Aug 20, 2018.

[46] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[47] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012.

[48] ——, "Divergence-Aware Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.

[49] A. Romano and W. Wang, "When Function Inlining Meets WebAssembly: Counterintuitive Impacts on Runtime Performance," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 350–362. [Online]. Available: https://doi.org/10.1145/3611643.3616311

[50] P. Sakdhnagool, A. Sabne, and R. Eigenmann, "RegDem: Increasing GPU Performance via Shared Memory Register Spilling," 2019.

[51] S. Shahar, S. Bergman, and M. Silberstein, "Activepointers: A case for software address translation on gpus," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016. [Online]. Available: https://doi.org/10.1145/3007787.3001200

[52] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013. [Online]. Available: https://doi.org/10.1145/2499368.2451169

[53] O. source community, "Numba: JIT Compiler from Python to Fast Machine Code." http://numba.pydata.org, Open-source community, 2022, accessed July 6, 2022.

[54] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11,

[61] J. Yan and W. Zhang, "Virtual Registers: Reducing Register Pressure without Enlarging the Register File," in *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and* no. 1, pp. 25–33, 1967.

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[56] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[57] O. Villa, M. Stephenson, D. Nellans, and S. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358307

[58] D. W. Wall, "Register Windows vs. Register Allocation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: Association for Computing Machinery, 1988, p. 67–78.

[59] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010, pp. 235–246.

[60] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling Coordinated Register Allocation and Thread-Level Parallelism Optimization for GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, New York, NY, USA, 2015, p. 395–406. [Online]. Available: https://doi.org/10.1145/2830772.2830813

Compilers, ser. HiPEAC'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 57–70.

[62] M. Zhang, A. Alawneh, and T. G. Rogers, "Characterizing Massively Parallel Polymorphism," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 205–216.

[63] ——, "Judging a Type by Its Pointer: Optimizing GPU Virtual Functions," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 241–254. [Online]. Available: https://doi.org/10.1145/3445814.3446734

[64] X. Zhuang, T. Zhang, and S. Pande, "Hardware-Managed Register Allocation for Embedded Processors," ser. LCTES '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 192–201. [Online]. Available: https://doi.org/10.1145/997163.997191