RegDem: Increasing GPU Performance via Shared Memory Register Spilling

Putt Sakdhnagool
National Electronics and Computer
Technology Center
Thailand
putt.sakdhnagool@nectec.or.th

Amit Sabne
Google Brain
CA, USA
asabne@google.com

Rudolf Eigenmann
University of Delaware
DE, USA
eigenman@udel.edu

ABSTRACT

GPU utilization, measured as *occupancy*, is limited by the parallel threads' combined usage of on-chip resources, such as registers and the programmer-managed shared memory. Higher resource demand means lower effective parallel thread count, and therefore lower program performance. Our investigation found that registers are often the occupancy limiters.

The de-facto nvcc compiler-based approach spills excessive registers to the off-chip memory, ignoring the shared memory and leaving the on-chip resources underutilized. To mitigate the register demand, this paper presents a binary translation technique, called *RegDem*, that spills excessive registers to the underutilized shared memory by transforming the GPU assembly code (SASS). Most GPU programs do not fully use shared memory, thus allowing RegDem to use it for register spilling. The higher occupancy achieved by RegDem outweighs the slightly higher cost of accessing shared memory instead of placing data in registers. The paper also presents a compile-time performance predictor that models instructions stalls to choose the best version from a set of program variants. Cumulatively, these techniques outperform the nvcc compiler with a 9% geometric mean, the highest observed being 18%.

1 INTRODUCTION

Thousands of threads can concurrently reside on GPU multiprocessors (SM). Each SM contains on-chip resources, such as registers and shared memory, used by the running threads. When the cumulative demand of the resources exceeds the resource capacity, some threads need to be suspended, decreasing multiprocessor utilization, called *occupancy* [21].

Achieving maximum occupancy is challenging due to limited on-chip resources. Among all on-chip resources, registers are the most common occupancy limiting factor. For example, NVIDIA Maxwell GPU architectures provide up to 64k 32-bit registers and 2048 resident threads per multiprocessor. Each thread can use only 32 registers if maximum occupancy is desired. Such limitation could easily be surpassed by medium-sized kernels (~100 lines of code). When registers limit occupancy, reducing just a few registers could significantly improve occupancy due to the step-function behavior of occupancy with respect to the kernel's register requirement [23].

The default GPU compiler, nvcc, provides an option to perform aggressive register allocation to emit binaries with fewest spills. nvcc achieves that by choosing instruction sequences that need fewer registers but are less efficient, e.g., it re-materializes [2] expressions. The excessive registers are then spilled to the thread-private, off-chip memory space called *local memory* [25]. While

such method minimizes the overhead of local memory accesses, less efficient binaries are created.

This paper presents an alternative scheme, termed *register demotion*, referred to as *RegDem* hereafter. RegDem increases program occupancy by spilling to the shared memory instead. The approach performs better due to the following reasons: 1) The GPU shared memory is on-chip and software managed. Therefore, the effective latency is substantially less compared to the local memory, even with a hardware-managed cache. 2) Although shared memory is primarily intended for programmer use, it often has enough storage available for register spilling. We observed that, in all our target applications, there is sufficient shared memory space for spilling. 3) As mentioned earlier, nvcc, under the aggressive register allocation option, avoids spilling to local memory as much as possible, owing to its high effective latency. Instead, nvcc produces slower instruction sequences, reducing overall GPU performance. Because RegDem spills to programmer-managed shared memory, it incurrs no such slowdowns.

Prior work has proposed several approaches to reduce register pressure on GPUs [10, 11, 30, 37, 40]. The closest related GPU register allocation algorithm [11] operates on the binary generated by nvcc with aggressive register allocation. This algorithm converts the spills to the faster shared memory, provided sufficient space is available. While this approach, like RegDem, takes advantage of faster memory, it cannot achieve full benefits due to reason (3) above.

RegDem could be efficiently implemented during a common register allocation process. Because the nvcc compiler infrastructure is proprietary, we implemented RegDem in a custom binary translation pass. RegDem determines the count of registers to be spilled, and subsequently translates the user-provided, efficient nvcc-generated binary. The algorithm then compacts the no longer contiguous register space, minimizing the highest-used register number. This is needed because the GPU ISA determines the register usage by this number. Furthermore, because register allocation and instruction scheduling are interacting compiler passes, our optimization considers the effect on the instruction schedule and performs updates where needed. The mechanism also addresses possible register bank conflicts [8] and the allocation of multi-word registers.

While RegDem uses fast memory for spilling, the benefits may not always outweigh spilling overheads. Moreover, when the number of excessive registers is small, the tradeoff between aggressive register allocation and RegDem's overheads becomes non-trivial. To overcome these difficulties, we developed a compile-time performance predictor that analyzes different code variants, using

instruction stalls as performance metric. It considers the combined effect of code efficiency (number of stalls) and occupancy (resulting from the number of registers and shared memory used). We then use this predictor to choose the best code variant.

In summary, the contributions of this paper include:

- A GPU register optimization algorithm, called *RegDem*, which spills excessive registers to shared memory, increasing occupancy and thus performance.
- A compile-time performance predictor, which chooses the best code variant from among different register allocation methods.
- A binary translator for GPU assembly (SASS), named *pyReDe* [1] that implements the proposed techniques.
- An integration and evaluation of the proposed performance predictor. The predictor achieves 99% of the performance of exhaustive search for the best optimization variant.

Our results show that RegDem with the predictor achieve 9% geometric mean speedup over nvcc on nine different benchmarks.

2 BACKGROUND

This section describes the Maxwell GPU architecture.[2] GPU cores are organized into streaming multiprocessors (SMs). Several threadblocks can reside in each SM. Each threadblock consists of a group of threads, called *warps*. Warp represents the SIMD width on GPUs. Overall, 2048 threads can reside in an SM. Occupancy is defined as the ratio of actual threads residing in an SM to the maximum. The GPU scheduling mechanism swaps warps to hide memory latency; lower occupancy results in decreased memory latency hiding, and thereby lower performance. Threads of a given threadblock can access the on-chip programmer-managed cache, termed shared memory.

The 64K 32-bit registers available on each SM are shared among all resident threadblocks. Thus, if the register requirement per thread is high, the cumulative requirement may exceed the available register space, allowing fewer resident threadblocks on the SM. In this manner, the register count can limit occupancy. The achieved occupancy is a step function of the kernel's register count, resulting in occupancy cliffs even when register count changes slightly.

The registers are organized across banks, and a warp accessing registers from the same bank causes higher access latency due to bank conflicts. The nvcc compiler allocates registers in a manner that reduces bank conflicts. Users can limit the number of registers used by a kernel by specifying the limiting number using --maxrregcount flag.

Each SM uses a hardware-managed L1 cache. Because several threadblocks share the cache, it faces higher contention. The shared memory, on the other hand, avoids the contention. Shared memory is allocated either statically, or dynamically, which means the allocation sizes only become apparent during the GPU kernel launch. The shared memory is organized into banks; threads in a warp accessing memory in the same bank see longer latencies. It is the programmer's responsibility to avoid such access patterns.

---

3 REGISTER DEMOTION

RegDem's goal is to reduce the register usage of a GPU kernel, such that the kernel achieves a higher occupancy level. Section 3.1 describes the challenges involved. Sections 3.2 and 3.3 describe spilling algorithms, while Section 3.4 presents post-spilling optimizations.

RegDem begins with two entities: the GPU executable and a count of registers to spill. Figure 1 shows overall process. RegDem contains an automatic utility that chooses different register counts to spill such that different occupancy cliffs could be achieved and the spills can fit in the available shared memory. Alternatively, the user may specify the register count to be spilled.
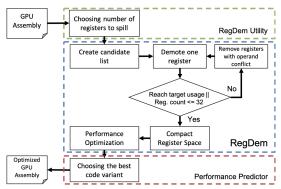

Figure 1: Register Demotion Process

Next, from the GPU executable, RegDem generates a list of candidate registers for demotion using certain selection criteria. In each iteration, one register is picked from this list and is demoted to shared memory. We will refer to the corresponding memory locations as **demoted registers** and to the demotion code as **demoted loads and stores**. This process repeats until the kernel reaches a desired occupancy level or the kernel's register usage falls below 32 registers, at which point further demotion offers no occupancy benefits.

3.1 Key Challenges

To realize the described approach the RegDem algorithm must address the following issues:

(1) **Shared Memory Bank Conflicts:** GPU shared memory consists of 32 banks. If multiple threads in a warp access different single words (32-bits) in the same bank, the accesses get serialized, increasing memory latency. The demoted loads and stores therefore must avoid such *shared memory bank conflicts*.

(2) **Operand Conflicts:** In GPU architectures, each shared memory access requires an explicit load/store instruction, thus needing a temporary register to access demoted registers. An *operand conflict* occurs when two registers are operands of the same instruction. Demoting both registers in that situation would require two temporaries, creating additional register pressure.

(3) **Multi-word Data Types:** All GPU registers are single-word, 32-bit wide. Storing multi-word data, such as double-precision floating point numbers, requires an *aligned* sequence of registers in which the leading register must be even numbered. Moreover, this requirement creates *register aliases* [33]. For example, if R8 is being used as a double-word register, then R9 is being used implicitly, in spite of no explicit reference in the binary.

(4) **Managing Instruction Barriers:** Maxwell was the first GPU ISA to introduce six different instruction barriers for synchronizing long-latency instructions [8]. Demoted loads and stores must therefore set barriers to signal their completion. Careful handling of these barriers, avoiding interference with existing synchronization, is essential for correctness. Furthermore, a good choice of the barriers is important for reducing stalls.

(5) **Using contiguous register numbers:** The architecture determines the register count of a GPU kernel by the highest register number being used. E.g., if a kernel uses register R15 but not R0 to R14, the GPU will still reserve 16 registers per thread. The spilled registers will create *gaps* in the register space, which will need to be compacted.

(6) **Register Bank Conflicts:** The GPU register file is split into banks. If an instruction tries to access two or more registers from the same bank, the accesses will be serialized, due to a *register bank conflict* [8]. We found that such conflicts can increase computation time by as much as 12%. Temporary registers allocated by RegDem, and during compaction should avoid register conflicts.

(7) **Instruction Scheduling:** Because demoted loads and stores are inserted in a previously schedule-optimized executable, additional instruction scheduling opportunity presents itself.

(8) **Choosing Candidates for Register Demotion:** The choice of registers to be demoted can greatly impact the performance, due to the cost of memory accesses and potential operand conflicts.

3.2 Register Spilling to Shared Memory

Shared Memory Allocation for Spilling: The allocated shared memory for spilling must avoid shared memory bank conflicts. RegDem's mechanism to that end is captured in Figure 2a. Each demoted register is assigned a contiguous space in memory, where successive threads own consecutive words. For a kernel with n threads per thread block, each demoted register allocates $n \times 4$ bytes of consecutive memory. The shared memory location of the r-th demoted register of the t-th thread of the thread block can be computed as

$$location = \underbrace{(t \times 4)}_{\text{base address}} + \underbrace{s + (r \times n \times 4)}_{\text{register offset}} \qquad (1)$$

where s is the static allocation size of the shared memory used by the kernel, rounded up to the nearest multiple of 4 bytes (shared memory bank alignment). Note that all values are known at compile-time except t, which is known at runtime. Because contiguous 32-bit shared memory values are placed in different banks, the above organization guarantees that all threads in a warp will access different shared memory banks as shown in Figure 2b. Our implementation dynamically allocates memory for demoted registers to separate user and demotion-allocated memory spaces.

Shared Memory Access Mechanism: GPUs use base-plus-offset addressing mode for shared memory accesses. Load (LDS) and store (STS) instructions have the following format.
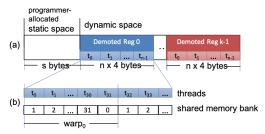
```
LDS RV, [RA+offset];
STS [RA+offset], RV;
```


Figure 2: Demoted Registers in Shared Memory: (a) Shared memory allocation with k demoted registers for a kernel with n-threads per thread block and s bytes of statically allocated shared memory. t_i denotes the i-th thread in a thread block. (b) Mapping between threads inside a warp and shared memory banks when the last byte of the static allocation resides in the 0-th memory bank.

RA is a register that contains the base address of a shared memory location and offset is an immediate value for indexing. RV denotes a register that contains the value of the target shared memory location. Both load and store instructions require barriers for synchronization. For a load instruction, read and write barriers are required to prevent RA and RV from write-after-read and read-after-write hazards, respectively. A store instruction only requires a read barrier to avoid a write-after-read hazard.

As seen above, accessing a demoted register requires two additional registers. First register holds the value of the *base address* from equation (1). We will refer to this register as RDA (short for demoted base address register). We use dynamic addressing for demoted register access because the value of t is only known at runtime. The second register is used to hold the value of the demoted register. This register will be referred to as RDV (short for demoted value register). Therefore, at least two registers must be added to the program. More than one value registers can be used so that multiple demoted registers can be accessed simultaneously. However, doing so will increase register pressure. The RegDem algorithm therefore limits the number of value registers to one. Section 3.4 will describe scenarios where more value registers can be employed. Note that for multi-word data, the value register count is increased based upon the data width.

Algorithm: RegDem's spilling algorithm renames the register to be demoted with RDV and places the associated demoted load-/store instructions next to the demoted register's accesses. Most importantly, the algorithm updates the instruction schedule by inserting barriers on the demoted loads/stores, which impacts both performance and correctness. Figure 3 shows a simplified version of the algorithm, which caters to single-word registers only.

The main algorithm (lines 3–31) spills one register at a time until the register usage reaches the target level or falls below 32, where it no longer limits occupancy. In each iteration, a register r is picked from the candidate list and its shared memory location is computed (lines 4–5). Next, the algorithm searches and replaces r with the RDV register (line 10). A shared memory instruction for accessing/updating the demoted register value is then inserted (lines 11–29). After all occurrences of r are replaced, the algorithm removes all candidates that have operand conflicts with r (line 31).

For write accesses (lines 11–19), a store instruction ($inst_{sts}$) is placed immediately after the write instruction ($inst$) to update the

```
1  Input: Program p, Size of thread block n
2  candidate_list ← CreateDemotionCandidate(p)
3  while p.reg_count > target_usage AND p.reg_count > 32 do
4  |   r ← dequeue(candidate_list)
5  |   offset ← demoted_reg_count × n × 4 + p.shared_size
6  |   foreach instruction inst ∈ p do
7  |   |   if inst is jump OR label then
8  |   |   |   ResetBarrierTracker(tracker)
9  |   |   if r ∈ inst.regs then
10 |   |   |   rename r in inst with RDV
11 |   |   |   if r is destination register then
12 |   |   |   |   inst_sts ← "STS [RDA+offset], RDV;"
13 |   |   |   |   if inst is high-latency instruction AND ¬ inst.write_barrier
   |   |   |   |     then
14 |   |   |   |   |   inst.write_barrier ← GetBarrier(tracker)
15 |   |   |   |   inst_sts.Wait(inst.write_barrier)
16 |   |   |   |   inst_sts.read_barrier ← GetBarrier(tracker)
17 |   |   |   |   Add inst_sts after inst
18 |   |   |   |   inst_next ← NextInstruction(p, inst_sts)
19 |   |   |   |   inst_next.Wait(inst_sts.read_barrier)
20 |   |   |   if r is operand then
21 |   |   |   |   inst_lds ← "LDS RDV, [RDA+offset];"
22 |   |   |   |   inst_lds.read_barrier ← GetBarrier(tracker)
23 |   |   |   |   inst_lds.write_barrier ← GetBarrier(tracker)
24 |   |   |   |   inst.Wait(inst_lds.read_barrier)
25 |   |   |   |   inst.Wait(inst_lds.write_barrier)
26 |   |   |   |   Add inst_dem before inst
27 |   |   |   |   inst_prev ← PrevInstruction(p, inst_lds)
28 |   |   |   |   if inst_prev is demoted store then
29 |   |   |   |   |   inst_lds.waitBarrier(inst_prev.read_barrier)
30 |   |   UpdateBarrierTracker(tracker, inst)
31 |   RemoveOperandConflict(candidate_list, r)
32 Function UpdateBarrierTracker(tracker, inst)
33 |   if inst.read_barrier then
34 |   |   tracker[inst.read_barrier].inst ← inst
35 |   |   tracker[inst.read_barrier].stall ← 0
36 |   if inst.write_barrier then
37 |   |   tracker[inst.write_barrier].inst ← inst
38 |   |   tracker[inst.write_barrier].stall ← 0
39 |   foreach barrier b ∈ tracker do
40 |   |   tracker[b].stall ← tracker[b].stall + inst.stall
41 |   foreach barrier b ∈ inst.wait do
42 |   |   tracker[b] ← NULL
43 Function GetBarrier(tracker)
44 |   min_barrier ← NULL; min_stall ← GL_MEM_STALL + 1
45 |   foreach barrier b ∈ tracker do
46 |   |   if tracker[b] == NULL then return b
47 |   |   if tracker[b].inst is global memory inst then
48 |   |   |   stall ← GL_MEM_STALL - tracker[b].stall
49 |   |   else if tracker[b].inst is shared memory inst then
50 |   |   |   stall ← SH_MEM_STALL - tracker[b].stall
51 |   |   if min_stall > stall then
52 |   |   |   min_barrier ← b; min_stall ← stall
53 |   return min_barrier
```

Figure 3: RegDem Algorithm: Each iteration of the main algorithm (lines 3–31) removes one register from the program. The `UpdateBarrierTracker` function keeps track of barrier usage. The `GetBarrier` function uses this information to select the barrier that will cause the least stalls.

demoted register's value in shared memory. The algorithm ensures that (1) updating RDV has completed before storing its value to the demoted register and (2) RDV is not rewritten before writing its value to the demoted register has completed. Similarly, for read accesses (lines 20–29), a load instruction ($inst_{lds}$) is placed before the instruction that accesses the demoted register ($inst$). The algorithm inserts barriers, ensuring that the shared memory load is completed before RDV is used by the next instruction. If the instruction before the demoted load ($inst_{prev}$) is a demoted store, the algorithm ensures that RDV is free before the demoted register is loaded.

Note from above that two barriers are used by each demoted load/store instruction to synchronize with other instructions. The barriers are limited in count; only six barriers exist on Maxwell and Pascal architectures. Therefore, if the barrier placed on a demoted load/store instruction was already occupied by a different instruction, additional stalls are introduced. A poor choice of a barrier may result in a wait of as many as 200 cycles, if that barrier is busy. To minimize the synchronization overhead, the RegDem algorithm presents a *barrier tracker* to monitor barrier usage. The tracker records the last instruction that will set the barrier and estimates the number of cycles passed since the setting instruction was executed as shown in the UpdateBarrierTracker() function. The algorithm obtains a new barrier through the GetBarrier() function. The function returns a free barrier if available, or else it returns the barrier that generates minimum stalls, which is determined by the type of setting instruction and the number of cycles passed since that instruction. Estimating the cycle count accurately is crucial to reducing the stall count. A key observation helps us estimate the stall count statically: GPU architectures require that barriers are cleared before jump instructions, and hence can only span basic blocks. The barrier tracker therefore can mark barriers that are assigned before a jump instruction to be definitely available after the jump. For straight line code, the tracker estimates cycle counts per instruction based on instruction latencies. For example, our current implementation sets the latency of device memory access (GL_MEM_STALL) to 200 cycles[22], and the latency of shared memory access (SH_MEM_STALL) to 24 cycles, based on the stalls caused by register read-after-write dependencies [21].

Extension for Multi-word Data: We can extend the algorithm to demote registers containing multi-word data. Recall that multi-word data requires an aligned series of registers. To accommodate this requirement, the algorithm chooses RDV to be even-numbered and adds extra registers for padding if needed. In our implementation, each register in the series is treated individually and uses the same allocation scheme as single-word registers. This allocation scheme allows each register to be accessed separately while avoiding bank conflicts. When accessing multi-word demoted registers, multiple load/store instructions are inserted.

3.3 Register Compaction

Recall that the last physical register number present in the code determines the register usage of the kernel. Demoted registers may still count as used until this number is reduced. Compaction achieves that effect. Our algorithm uses a data structure called *relocation space*, which utilizes an array for performing virtual register movement. Figure 4 shows this data structure. Each slot
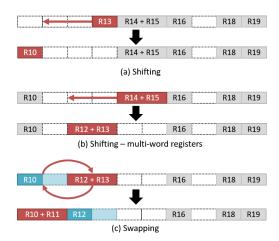
Figure 4: Register Relocation Space and Register Compaction: A multi-word register is represented by a series of registers connected using '+' signs. (a) Single-word register shifting. (b) Double-word register shifting. The register cannot be moved to the first gap because of alignment restriction. (c) A double-word register is swapped with a swapping window of size 2 (marked in cyan) containing a single-word register and a gap.

represents one physical register present in the program. Multi-word registers occupy multiple slots based on their size and are represented as single registers. This representation prevents the algorithm from breaking register aliases when compressing the register space. A register gap is represented by an empty slot.

The algorithm pushes the gaps toward the end of the register space by using two operations, *shifting* and *swapping*. Figure 4 shows an example of the relocation space and its operations. Shifting moves the next available register to fill the gaps (Figure 4a). Occasionally, the shifting operation cannot move the multi-word register to the available gap due to register alignment (Figure 4b). In this situation, swapping applied after the shifting operation moves the register to the closest possible gap. The swapping operation exchanges the multi-word register with the registers in the *swapping window* (Figure 4c). The swapping window starts from the location of the multi-word register and grows toward the lower-numbered location. The size of swapping windows is determined by the size of the multi-word registers.

3.4 Post-Spilling Optimizations

This section discusses optimization opportunities presented after applying RegDem's shared memory spilling.

3.4.1 Avoiding Register Bank Conflicts. When register operands in the same instruction belong to the same register bank, their accesses get serialized. Because RDV is chosen before RegDem begins, bank conflicts may get introduced during the demotion process.

The first strategy is to choose RDV from the bank that generates the least number of conflicts. This is achieved with a small addition to the presented algorithm, keeping track of the conflicts caused by RDV. Register bank conflict avoidance is also added to the compaction algorithm (Section 3.3). The mechanism searches for registers from the same bank to fill gaps. Swapping is performed within a window

of size equal to the number banks in the register file, i.e., four. This modification can lose efficiency when an even-numbered gap is filled by a single-word register, leaving a small gap. We revert to the original algorithm in that case since reducing register count is the top priority.

3.4.2 Performance Enhancements. RegDem adds load/store instructions conservatively for lack of global analysis. The following optimization passes improve the code in these situations:

Eliminating redundant demote code: This pass reduces demoted register access overhead by tracking the recent value of the RDV register and eliminating subsequent loads of the same, if the value is still alive. For example, if two consecutive instructions read the same demoted register, a demoted load will be placed before each of them. This optimization will remove the second load. Similarly, demoted stores are removed if their target demoted register will be updated again in the near future.

Updating instruction schedule: This pass reschedules demoted loads and stores to reduce instruction synchronization overhead. Demoted loads are hoisted as early as possible, updating the associated instruction barriers. The analysis also removes barriers from demoted stores if the RDV register will not be updated before the memory store completes.

Substituting Value Register: To keep register pressure low, only one RDV register is reserved for accessing demoted registers. Hence, only one demoted register can be in use at any given time, restricting the window within which demoted loads/stores can be hoisted. To enlarge this window, the optimization analyzes register liveness in each basic block, identifying other free registers as local temporaries. Then, the optimization substitutes RDV inside the block with these temporaries, allowing multiple demoted registers to be in use simultaneously without increased register pressure.

3.4.3 Choosing Candidate Registers for Demotion. We use three strategies for choosing candidate registers. Each strategy estimates register access counts, and candidates are chosen in ascending order of the access count. The first strategy makes a simple pass through the assembly code and counts the number of static accesses for each register. The second strategy traverses the CFG to count register accesses of each basic block. For basic blocks inside a loop, the access count is multiplied by a generic value of 10. The third strategy takes operand conflicts into consideration. It chooses candidates in ascending order of their operand conflicts. The performance predictor described in the next section (Section 4) chooses from among these three strategies.

4 COMPILE-TIME PERFORMANCE PREDICTOR

In some corner cases, the RegDem benefits may not outweigh the spilling overheads. When the spill count is small, the tradeoff between aggressive register allocation and overheads of RegDem becomes non-trivial. With the optimizations described in Section 3.4, the difficulty of choosing the best performing code variant increases further.

To make that decision, we developed a compile-time performance predictor that analyzes GPU binaries and selects the best code, also

considering non-RegDem variants. The predictor approximates program performance using instruction stalls as performance metric. The predictor considers both explicit stalls, presented in instruction annotations [8, 14], and implicit overheads from (i) memory accesses latencies, (ii) variations in instruction throughput across different instruction types, and (iii) loop and function constructs. Figure 5 shows the performance predictor algorithm. It performs three main steps:

Step one (lines 3–22) traverses through the program control flow graph (CFG) and estimates the stall cycles in each basic block. The algorithm collects stalls generated by each instruction in the block and adjusts these stalls based on instruction throughput and memory accesses. GPU instructions could have different throughput based on available resources, e.g., Maxwell GPUs have 128 FP32 and 4 FP64 cores. Instructions with less resources would experience more stalls due to higher contention. The predictor factors in occupancy and instruction throughput, using the following equation:

$$stall = inst_{stall} \times occupancy \times \frac{MAX\_THROUGHPUT}{inst_{throughput}} \quad (2)$$

$inst_{stall}$ denotes the stall cycles per the instruction annotation. The term $\frac{MAX\_THROUGHPUT}{inst_{throughput}}$ estimates the contention of lower-throughput instructions, where $MAX\_THROUGHPUT$ denotes the maximum instruction throughput and $inst_{throughput}$ denotes the throughput of the instruction. For Maxwell GPUs, the value of $MAX\_THROUGHPUT$ is 128 instructions/cycle. $occupancy$ is used for estimating the number of threads waiting for the resource.

Memory access stalls are estimated by tracking the time between barrier set and register use, then taking the maximum of this time and the memory latency. The algorithm uses the barrier tracker and memory latency described in Section 3 for analysis.

Step two (lines 23–28) updates the stall count of every basic block inside a loop. The block stall count is multiplied by a generic $LOOP\_FACTOR$; the current implementation sets this value to 10, which is a plausible static estimate for a value that would need dynamic analysis. This method weighs loops higher than straight-line code.

Step three (lines 29–31) estimates the overall stalls by summing stall cycles of all basic blocks. This approximation is needed because branch decisions are not known statically. Furthermore, both branch targets must be considered because the GPU SIMD approach results in serial execution of branch taken/not taken paths if even a single thread executes the respective path. The algorithm proceeds interprocedurally, estimating the CFGs of inner functions first.

To compare code variants generated by the different register allocation methods, the predictor considers stalls as well as program occupancy. Improving occupancy normally yields diminishing returns, and degrades performance at worst [35]. The predictor reflects this behavior in the estimated execution time using the following equation to adjust the result $stall_{count}$ from Figure 5.

$$stall_{program} = \frac{f(occupancy)}{f(occupancy_{max})} \times stall_{count} \quad (3)$$

$stall_{program}$ represents an estimated execution time of the code variant in stall cycles. The term $\frac{f(occupancy)}{f(occupancy_{max})}$ computes the slow-down caused by the lower occupancy, where $occupancy_{max}$ is the maximum occupancy across code variants and $f(x)$ is a function

```
1  Input: Program p, Program CFG cfg
2  Output: Estimated stall cycle stall_count
3  for block ∈ cfg do
4      block.stall ⟵ 0
5      for inst ∈ block.instructions do
6          inst.stall ← inst.stall × p.occupancy × (MAX_THROUGHPUT / inst.throughput)
7          if inst.read_barrier then
8              tracker[inst.read_barrier].inst ⟵ inst
9              tracker[inst.read_barrier].stall ⟵ 0
10         if inst.write_barrier then
11             tracker[inst.write_barrier].inst ⟵ inst
12             tracker[inst.write_barrier].stall ⟵ 0
13         for w ∈ inst.wait_barriers do
14             if tracker[w].inst is global access then
15                 if tracker[w].stall < GL_MEM_STALL then
16                     block.stall ⟵ block.stall + GL_MEM_STALL −
                           tracker[w].stall
17             else if tracker[w].inst is shared access then
18                 if tracker[w].stall < SH_MEM_STALL then
19                     block.stall ⟵ block.stall + SH_MEM_STALL −
                           tracker[w].stall
20         for bar ∈ barriers do
21             tracker[bar].stall ⟵ tracker[bar].stall + inst.stall
22         block.stall ⟵ block.stall + inst.stall
23 for block ∈ cfg in breath-first order do
24     for edge ∈ block.edge do
25         if edge is backward then
26             loop ⟵ GetLoop(block, edge)
27             for b ∈ loop.blocks do
28                 b.stall ← b.stall × LOOP_FACTOR
29 stall_count ⟵ 0
30 for block ∈ cfg in breath-first order do
31     stall_count ⟵ stall_count + block.stall
```

Figure 5: Performance Estimation Algorithm

used for estimating the execution time at $x\%$ occupancy. $f(x)$ was determined empirically, using compute-intensive microbenchmarks across various thread block sizes. The occupancy of the microbenchmarks is controlled by modifying register usage, measuring only the impact of occupancy on performance. The predictor uses theoretical occupancy in the computation, which can be computed from the thread block size of the user input [23].

5 EVALUATION

This section evaluates the presented RegDem technique and performance predictor. We compare the performance of RegDem with the default nvcc code generation and the closest research alternative [11]. Next, we measure the impact of the optimization options. Lastly, we evaluate the added gain by the performance predictor.

5.1 Experimental Settings

We evaluated our techniques on an Ubuntu Linux 14.04.3 system with a quad-core Intel Core i7-6700K processor running at 4.00 GHz, 16 GB of main memory, and a Maxwell-based GeForce GTX Titan X GPU with 12 GB device memory.

Recall that registers can limit occupancy if and only if a kernel requires more than 32 registers per thread. Only applications where

Table 1: Details of Benchmark Kernels used in Performance Evaluation.

Bench-mark	Kernel	Input	#Thread blocks	Threads / block	Shared memory	# Registers Used		# Registers Spilled[1]		Achieved Occupancy[2]	
						orig	target	nvcc	RegDem	orig	RegDem
cfd	cuda_compute_flux	fvcorr.193K	1008	192	0B	68	56	10	14	0.35	0.54
qtc	QTC_device	8192 points	1538	64	512B	55	48	8	10	0.51	0.57
md5hash	FindKeyWithDigest_Kernel	1680M keys	93790	256	0B	33	32	0	3	0.70	0.94
md	compute_lj_force	73728 atoms	228	256	0B	34	32	1	5	0.75	0.83
gaussian	d_recursiveGaussian_rgba	32K×10K px	500	64	0B	43	40	1	5	0.58	0.62
conv	convolutionColumnsKernel	4K×4K px	16384	128	0B	35	32	0	5	0.73	0.98
nn	nearest_neighbor_search	200K 7d pts	1024	192	1.52KB	35	32	0	5	0.55	0.72
pc	compute_correlation	200K 7d pts	1024	256	2.03KB	36	32	2	6	0.54	0.72
vp	search_kernel	200K 7d pts	2048	256	2.03KB	34	32	0	4	0.52	0.68

[1] Number of registers spilled / demoted by nvcc and RegDem when restricting register usage to the specified target.
[2] Achieved occupancy of the kernel measured by nvprof profiler before (*orig*) and after RegDem.

Table 2: Benchmark Description

Benchmark suite	Benchmark	Description
Rodinia [4]	cfd	An unstructured grid solver for three-dimensional Euler equations.
SHOC [5]	qtc	A quality threshold clustering algorithm
	md5hash	A brute force search to find a key with a MD5 digest.
	md	An N-body computation computing the Lennard-Jones potential.
CUDA Toolkit-Imaging [24, 26]	gaussian	A Gaussian blur using Deriche's recursive method.
	conv	A separable convolution filter for 2D image.
FSM [18]	nn	Nearest neighbors search of the input points in the metric space using kd-tree.
	pc	Computing two-point correlation of each points in the input data.
	vp	Nearest neighbors search of the input points in the metric space using vantage point trees.

Table 3: Code Variant Comparison

	nvcc (base-line)	Reg-Dem	local	local-shared	local-shared-relax
spilled memory space	-	shared	local	shared	shared
target register usage	★	†	†	32	†
use nvcc to spill registers	-	-	✓	✓	✓
convert local to shared mem.	-	-	-	✓	✓
demote reg. to shared memory	-	✓	-	-	-

★: not restricted, †: set to the target register usage specified in Table 1.

register pressure limits occupancy will benefit from RegDem. This is the case in nine applications of the four benchmark suites in Table 2. RegDem has no effect on the other applications.

The baseline versions for all benchmarks are created with nvcc and the benchmark-provided compiler flags. The optimized versions use the techniques described in this paper. RegDem extracts assembly code from a .cubin file, performs the optimizations, and regenerates assembly code. The MaxAs tool [8] then inserts the optimized code into the original .cubin file. We used nvcc version 6.5, the latest version supported by MaxAs. NVIDIA's nvprof profiler was used to measure the average execution time of the kernels across five runs.

5.2 Achieved Occupancy

Table 1 shows achieved occupancy of the benchmarks before (*orig*) and after RegDem. On average, RegDem improves occupancy by 27%. Benchmarks with larger thread block size could see higher improvements owing to the step-function nature of occupancy.

5.3 Code Variants used in Performance Evaluations

We consider four code variants in addition to RegDem, shown in Table 3. The *nvcc* version represents the *baseline* performance. It is compiled with nvcc and the default compiler flags provided by the benchmarks.

The *local* variant uses nvcc with --maxrregcount flag, forcing the compiler to use aggressive register allocation and spill excessive

registers to local memory. This is the state-of-the-art method for restricting register usage. The register count is set to be the same as in RegDem.

Local-shared and *local-shared-relax* realize the technique presented by Hayes et al. [11], which converts spill code from *local* to shared memory. The only difference is the target register usage. The *local-shared* variant strictly follows the [11] implementation. The target register usage is set to 32 registers, allowing kernels to execute at maximum occupancy, and relies on tuning the thread block size to achieve the best performance. This is unlike our approach, where the number of spilled registers is the only parameter tuned. We consider this version the *closest research alternative*. For fair comparison, *local-shared-relax* relaxes the register usage restriction and sets it to the same value used in RegDem. We call this variant the *enhanced research alternative*.

5.4 Performance Results

Figure 6 shows the speedups of RegDem and alternatives over the baseline nvcc variants. This experiment applies the best combination of optimization options presented in Section 3.4, found through exhaustive search over all combinations.

Overall, RegDem performs the best in seven of the nine benchmarks when compared to other spilling techniques. RegDem achieves up to 1.18x speedup over the baseline nvcc, with a geometric mean of 1.07x, and shows significant improvement over the closest research alternative (*local-shared*), where RegDem obtains 1.19x geometric mean speedup. In contrast, *local*, *local-shared*, *local-shared-relax* achieve 1.03x, 0.90x, and 1.05x geometric mean speedups over the baseline implementations, respectively.

5.5 Discussion

We classify the benchmarks into three groups based on their characteristics. The first group, cfd and qtc, requires a significant number of registers to be spilled. Local memory spilling is unable to improve
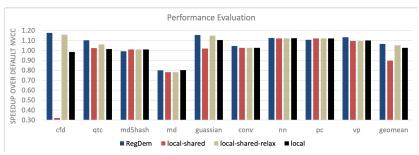
Figure 6: Performance Evaluation of RegDem: The bars show speedups over default baseline `nvcc` versions. The rightmost bars show the geometric mean speedup of various versions. RegDem uses the techniques presented in this paper. *local, local-shared,* and *local-shared-relax* represent the alternative techniques for GPUs, with *local-shared* being the closest alternative. Overall, RegDem obtains a 7% geometric mean speedup over default `nvcc` and outperforms the other alternative techniques in 7 benchmarks.
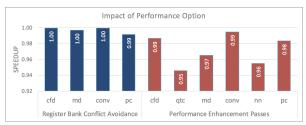

Figure 7: Impact of the Post-Spilling Optimizations: Slowdown obtained by disabling individual performance options. A lower speedup indicates that the performance option has higher impact. On average, performance enhancement passes and register bank conflict avoidance show 3% and less than 1% performance impact, respectively.
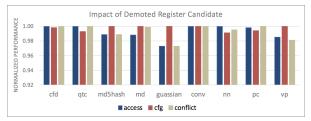

Figure 8: Impact of Register Candidate: The chart shows normalized performance over the *best* candidate selection strategy. The strategy with 1.0x speedup is the best strategy for that benchmark. Overall, the *cfg* strategy obtains the best performance.

the performance due to high access latencies. RegDem also benefits from the maximized single-thread performance [25] of `nvcc`'s default register allocation and shows substantial improvement over other spilling alternatives, which rely on the aggressive allocation.

The second group includes the remaining benchmarks except `md`, wherein a few registers are spilled to reach the target occupancy. Hence, the spilled register access overhead is less noticeable. In several benchmarks, instead of spilling registers, `nvcc` performs allocation in a manner that avoids spilling, but degrades single-thread performance (Table 1). This results in an occupancy gain without spilling overhead, which we call *zero spilling*.

The `md5hash` benchmark exemplifies the effect of zero spilling in the alternative approaches, resulting in slightly better performance than RegDem. However, zero spilling also comes with a drawback, as it sacrifices single-thread performance for occupancy. Such example is shown in `vp`, where the performance achieved from zero spilling is below that of RegDem. Profiling results indicate that the number of dynamic instructions increased significantly (about 3% per warp), owing to the reduced single-thread performance. Assembly inspection also showed that the additional instructions have high stall count (13 cycles).

The `md` benchmark is the only benchmark that does not achieve improvement with any of the techniques. The key distinction of `md` is that it uses double-precision floating point numbers, and hence the FP64 ALUs become the performance bottleneck. Improving occupancy by the described optimizations increases the execution time of the critical path, as more threads need to wait for the FP64 ALUs.

5.6 Impact of Post-Spilling Optimizations

This section analyzes the impact of the performance options presented in Section 3.4. We used the best combination (*RegDem* from Figure 6) as the baseline for the analysis.

Figure 7 evaluates the register bank conflict avoidance and the performance enhancement passes. We measured the impact by disabling individual options and observing the performance change. Benchmarks that do not benefit from any option are not shown.

Register bank conflict avoidance has an impact of less than 1%. Although, `MaxAs` reports that the algorithm could avoid an average of 36% of the conflicts, the ratio of the instructions with conflict to the total number of instructions is low and therefore does not significantly affect program performance.

The performance enhancement pass improves the performance by up to 5% and by approximately 3% on average. We have observed that value register substitution is rarely employed. This stems from the rarity of code sections where free registers are available. Hence, only a small portion of the program can take advantage of the pass.

We also evaluated the impact of choosing candidate registers for demotion, described in Section 3.4.3 (Figure 8). Overall, the *cfg* strategy gives the best result. The advantage of the *cfg* approach is that it considers access overhead inside loops; however, it could result in over-estimating the overhead, and thus avoiding good candidates.

5.7 Compile-time Performance Prediction

This section evaluates the performance predictor. We compare the performance achieved by the predictor to an oracle that knows the
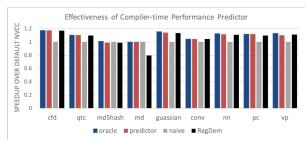
Figure 9: Effectiveness of Compile-time Performance Predictor: The chart shows the speedup of each benchmark using an oracle and the performance predictor. The oracle and the predictor achieves a geometric mean speedup of 1.10x and 1.09x, respectively. Thus, the predictor can achieve 99.0% performance compared to the oracle.

best variant of the benchmark, including original, *local*, *local-shared*, and *local-shared-relax* from Figure 6.

The predictor estimates the performance of every code variant. For RegDem, the predictor estimates all performance option combinations, including the post-RegDem optimizations presented in Section 3.4. To break ties, it chooses the one with the highest number of performance options enabled, counting on potential benefits of the enabled options. We also compared the predictor with a naive scheme (*naive*) that statically counts stall cycles, and RegDem with all performance enhancements enabled (*RegDem*).

Figure 9 shows the results. The geometric mean speedup achieved by the oracle is 1.10x while the predictor obtains a geometric mean speedup of 1.09x over the default nvcc versions, achieving 99.0% of the results of exhaustive search. The predictor also helps avoid the worst-case scenario, where applying the optimization degrades the performance.

In seven of the nine benchmarks, the predictor correctly chose the spilling technique with the best performance. We consider this good accuracy for static performance prediction. One notable case is the md benchmark, which all optimizations fail to improve. The predictor correctly assesses the situation and chooses the low-occupancy variant.

The current predictor is biased toward optimizations that directly impact instruction stalls, such as removing and rescheduling memory instructions. This bias is due to limited consideration of instruction scheduling. A runtime method could improve the accuracy further, forfeiting the benefit of a compile-time solution. In future work, we will explore using microbenchmarks to build a database of instruction interactions, capturing effects of scheduling more accurately.

6 RELATED WORK

Generally, register spilling is performed during register allocation. Prior work has extensively studied strategies for CPUs [3, 9, 16, 27, 28, 33, 36]. These algorithms aim for high register usage to maximize single-thread performance. Using the same approach on GPUs could lead to lower occupancy and reduced performance.

Several approaches have been proposed to improve GPU register usage. Many of them exploit smarter register allocation [10, 11, 30, 37, 40]. Similar to our work, Hayes and Zhang [11] proposed a register allocation algorithm for GPUs that utilizes the shared memory for spilling. Their approach turns local memory spills into shared

memory allocations. As shown in the evaluation section, this approach experiences reduced single-thread performance. Moreover, the approach does not handle multi-word data. Sampaio et al. [30] proposed divergence-aware register allocation, which reduces register pressure by placing common data in non-register memory spaces. Xie et al. [37] proposed a compiler framework that puts spilled registers into shared memory; however, their method applies the optimization at the PTX level and requires additional hardware support for register allocation. Hayes et al. further extended their work in Orion [10] with a register allocation scheme that spills registers to shared and local memory, instead of converting local memory spills. In contrast, RegDem does not fully reallocate the entire register space, which could interfere with transformations performed by prior optimization passes. Also their approach applies to the older Kepler ISA alone, and hence does not take instruction barriers into consideration. Additionally, our approach can be applied as a stand-alone optimization.

Resource virtualization is another direction for reducing register pressure. Yan and Zhang [38, 39] use virtual register files to realize such effect on CPUs. On GPUs, Zorau [34] uses resource virtualization to manage multiple on-chip resources, including registers and shared memory. Jeon et al. [12] proposed register file virtualization to share physical registers across GPU warps but did not use shared memory as a spill target. These approaches require hardware support, which is unavailable in current GPUs.

Prior work has proposed several auto-tuning systems for GPUs [6, 7, 13, 15, 17, 19, 29, 31]. However, the majority of these contributions rely on runtime information for performance tuning. Several authors have also studied offline performance analysis for GPUs. Baghsorkhi et al. [1] proposed a work-flow graph for compiler-based performance analysis. Meng et al. [20] predict GPU performance from the CPU skeleton code. Unlike these two schemes, our performance predictor uses low-level information to estimate the performance of GPU programs. Sim et al. [32] proposed an offline performance analysis, which estimates the potential benefit to GPU programs. In contrast to our approach, Sim's method aims to find the bottleneck of the GPU program, while our scheme compares the performance of different code variants. Our performance estimation can be seen as complementary to these approaches, allowing the analysis from multiple angles for higher-accuracy performance models.

7 CONCLUSION

This paper proposed *RegDem:* an assembly-level GPU register optimization method for improving program occupancy by reducing register pressure. The optimization moves excessive registers to the on-chip shared memory, finding a good tradeoff between register use and occupancy. The optimization addresses issues such as bank conflicts and interactions with instruction scheduling. The paper also introduced three optimizations to further improve the resulting code of RegDem. The presented techniques work well in an automatic, stand-alone binary translator. Nevertheless, tighter integration with the nvcc compiler could yield further improvements, especially through better interactions of register allocation, instruction scheduling, and instruction selection. Further opportunities lie in improving performance prediction through runtime methods.

Such methods would need to carefully consider the overheads of making the decisions – in this case selecting from among several code variants – in the critical program execution path. The presented static method avoids these overheads and performs well in practice.

REFERENCES

[1] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An Adaptive Performance Modeling Tool for GPU Architectures. *SIGPLAN Not.* 45, 5 (Jan. 2010), 105–114. https://doi.org/10.1145/1837853.1693470

[2] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. *SIGPLAN Not.* 27, 7 (July 1992), 311–321. https://doi.org/10.1145/143103.143143

[3] G. J. Chaitin. 1982. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. ACM, New York, NY, USA, 98–105. https://doi.org/10.1145/800230.806984

[4] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. 2010. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10) (IISWC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/IISWC.2010.5650274

[5] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

[6] Andrew Davidson and John Owens. 2012. *Toward Techniques for Auto-tuning GPU Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 110–119. https://doi.org/10.1007/978-3-642-28145-7_11

[7] Yuri Dotsenko, Sara S. Baghsorkhi, Brandon Lloyd, and Naga K. Govindaraju. 2011. Auto-tuning of Fast Fourier Transform on Graphics Processors. *SIGPLAN Not.* 46, 8 (Feb. 2011), 257–266. https://doi.org/10.1145/2038037.1941589

[8] Scott Gray. 2017. MaxAs. https://github.com/NervanaSystems/maxas/. (2017). [Online; accessed 1-April-2017].

[9] Sebastian Hack, Daniel Grund, and Gerhard Goos. 2006. Register Allocation for Programs in SSA-Form. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 247–262. https://doi.org/10.1007/11688839_20

[10] Ari B. Hayes, Lingda Li, Daniel Chavarría-Miranda, Shuaiwen Leon Song, and Eddy Z. Zhang. 2016. Orion: A Framework for GPU Occupancy Tuning. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA, Article 18, 13 pages. https://doi.org/10.1145/2988336.2988355

[11] Ari B. Hayes and Eddy Z. Zhang. 2014. Unified On-chip Memory Allocation for SIMT Architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 293–302. https://doi.org/10.1145/2597652.2597685

[12] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU Register File Virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 420–432. https://doi.org/10.1145/2830772.2830784

[13] Wenhao Jia, Elba Garza, Kelly A. Shaw, and Margaret Martonosi. 2015. GPU Performance and Power Tuning Using Regression Trees. *ACM Trans. Archit. Code Optim.* 12, 2, Article 13 (May 2015), 26 pages. https://doi.org/10.1145/2736287

[14] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826

[15] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. 2013. A Script-based Autotuning Compiler System to Generate High-performance CUDA Code. *ACM Trans. Archit. Code Optim.* 9, 4, Article 31 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400690

[16] Philipp Klaus Krause. 2013. *Optimal Register Allocation in Polynomial Time*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/978-3-642-37051-9_1

[17] S. Lee and R. Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2010.36

[18] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*. 2:1–2:12. https://doi.org/10.1145/2925426.2926261

[19] Yixun Liu, E. Z. Zhang, and X. Shen. 2009. A cross-input adaptive framework for GPU program optimizations. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–10. https://doi.org/10.1109/IPDPS.2009.5160988

[20] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. 2011. GROPHECY: GPU performance projection from CPU code skeletons. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[21] NVIDIA. 2017. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide. (2017). [Online; accessed 2-April-2017].

[22] NVIDIA. 2017. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. (2017). [Online; accessed 2-April-2017].

[23] NVIDIA. 2017. CUDA Occupancy Calculator. https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls. (2017). [Online; accessed 9-April-2018].

[24] NVIDIA. 2017. CUDA Toolkit Documentation - CUDA Samples. http://docs.nvidia.com/cuda/cuda-samples. (2017). [Online; accessed 1-April-2017].

[25] NVIDIA. 2017. NVIDIA CUDA Compiler Driver NVCC. http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/. (2017). [Online; accessed 2-April-2017].

[26] Victor Podlozhnyuk. 2013. *Image Convolution with CUDA*. Technical Report. NVIDIA Corporation.

[27] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sept. 1999), 895–913. https://doi.org/10.1145/330249.330250

[28] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register Allocation by Puzzle Solving. *SIGPLAN Not.* 43, 6 (June 2008), 216–226. https://doi.org/10.1145/1379022.1375609

[29] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. 2012. Effects of Compiler Optimizations in OpenMP to CUDA Translation. In *Proc. of the International Workshop on OpenMP, IWOMP*. http://engineering.purdue.edu/paramnt/publications/iwomp12.pdf

[30] Diogo Nunes Sampaio, Elie Gedeon, Fernando Magno Quintão Pereira, and Sylvain Collange. 2012. *Spill Code Placement for SIMD Machines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 12–26. https://doi.org/10.1007/978-3-642-33182-4_3

[31] Katsuto Sato, Hiroyuki Takizawa, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2010. *Automatic Tuning of CUDA Execution Parameters for Stencil Processing*. Springer New York, New York, NY, 209–228. https://doi.org/10.1007/978-1-4419-6935-4_13

[32] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 11–22. https://doi.org/10.1145/2145816.2145819

[33] Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004. A Generalized Algorithm for Graph-coloring Register Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 277–288. https://doi.org/10.1145/996841.996875

[34] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2016.7783718

[35] V. Volkov. 2010. Better performance at lower occupancy.. In *Proceedings of the GPU Technology Conference*.

[36] Christian Wimmer and Michael Franz. 2010. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 170–179. https://doi.org/10.1145/1772954.1772979

[37] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/2830772.2830813

[38] Jun Yan and Wei Zhang. 2007. *Virtual Registers: Reducing Register Pressure Without Enlarging the Register File*. Springer Berlin Heidelberg, Berlin, Heidelberg, 57–70. https://doi.org/10.1007/978-3-540-69338-3_5

[39] Jun Yan and Wei Zhang. 2008. Exploiting Virtual Registers to Reduce Pressure on Real Registers. *ACM Trans. Archit. Code Optim.* 4, 4, Article 3 (Jan. 2008), 18 pages. https://doi.org/10.1145/1328195.1328198

[40] Yi-Ping You and Szu-Chieh Chen. 2015. Vector-aware Register Allocation for GPU Shader Processors. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '15)*. IEEE Press, Piscataway, NJ, USA, 99–108. http://dl.acm.org.ezproxy.lib.purdue.edu/citation.cfm?id=2830689.2830703