Dimensionality-Aware Redundant SIMT Instruction Elimination

Tsung Tai Yeh
yeh14@purdue.edu
Purdue University
West Lafayette, Indiana, U.S.A.

Roland N. Green
green349@purdue.edu
Purdue University
West Lafayette, Indiana, U.S.A.

Timothy G. Rogers
timrogers@purdue.edu
Purdue University
West Lafayette, Indiana, U.S.A.

Abstract

In massively multithreaded architectures, redundantly executing the same instruction with the same operands in different threads is a significant source of inefficiency. This paper introduces Dimensionality-Aware Redundant SIMT Instruction Elimination (DARSIE), a non-speculative instruction skipping mechanism to reduce redundant operations in GPUs. DARSIE uses static markings from the compiler and information obtained at kernel launch time to skip redundant instructions before they are fetched, keeping them out of the pipeline. DARSIE exploits a new observation that there is significant redundancy across warp instructions in multi-dimensional threadblocks.

For minimal area cost, DARSIE eliminates conditionally redundant instructions without any programmer intervention. On increasingly important 2D GPU applications, DARSIE reduces the number of instructions fetched and executed by 23% over contemporary GPUs. Not fetching these instructions results in a geometric mean of 30% performance improvement, while decreasing the energy consumed by 25%.

CCS Concepts. • **Computer systems organization → Single instruction, multiple data**.

Keywords. GPU, redundant instructions

ACM Reference Format:
Tsung Tai Yeh, Roland N. Green, and Timothy G. Rogers. 2020. Dimensionality-Aware Redundant SIMT Instruction Elimination. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378520

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378520

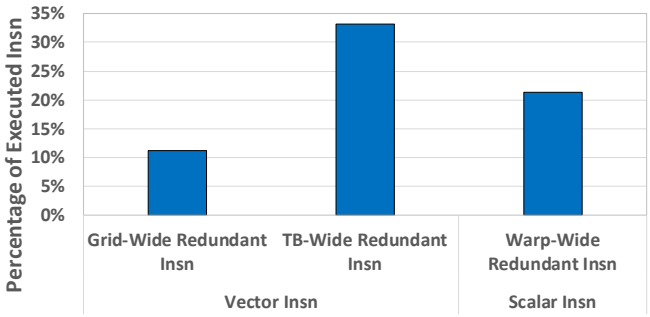

Figure 1. Redundant instructions in each GPU thread grouping level, averaged across the applications listed in Table 1.

1 Introduction

Graphics Processing Units (GPUs) run thousands of concurrent scalar threads based on programmer-defined parallelism. Programmers define a three-dimensional grid of threadblocks (TBs) for each kernel. TBs are three-dimensional arrangements of scalar threads, grouped into warps by the hardware for Single Instruction Multiple Thread (SIMT) execution. Although the programming model for GPUs is SIMT, the underlying datapath is Single Instruction Multiple Data (SIMD). Each warp has a set of private vector registers that store per-thread scalar values in each vector lane. It is well known that grouping threads into warps can create redundancy among the scalar values in vector registers [9, 19, 50]. However, little work has explored identifying and removing redundancy at other levels of the GPU programming model.

To help understand GPU redundancy, Figure 1 shows the results of a limit-study measuring the fraction of redundantly executed instructions at the grid, TB, and warp level. Instructions are classified as redundant at the grid-level when all the grid's warp instructions operate on the same vector operands, implying it need only be executed once for the entire grid. Similarly, Figure 1 plots redundant instructions for TBs if all warp instructions within a TB use the same vector operands. Warp-wide redundancy occurs if all scalar threads in a warp operate on the same scalar value. We find that the most significant opportunity for redundancy elimination exists at the TB level, where on average 33% of instructions need only be executed once per TB. Furthermore, we find that much of this redundancy can be attributed to the thread ID layout in the GPU programming model. GPU languages like

CUDA and OpenCL express parallelism defined along multiple (x, y, and z) axes, which helps programmers naturally map multi-dimensional data to multi-dimensional thread grids. However, this dimensionality has a significant impact on redundancy.

To demonstrate how common multi-dimensional redundancy is, we surveyed 133 applications [4, 6–8, 12, 21, 33–35, 41, 42, 44] running on a commodity NVIDIA Volta GPU. Over 33% of the applications surveyed demonstrated multi-dimensional TB characteristics that create implicit redundancy. Interestingly, we find that this characteristic is more pervasive in applications that make use of optimized libraries (CUDNN, CUBLAS, etc.), where 60% were multi-dimensional. Furthermore, in the apps that have at least one kernel that was multi-dimensional, an average of 71% of the application's execution time is spent in those kernels.

Unlike warp-wide redundant operations that are local to one vector instruction, TB- and grid-wide redundant instructions occur across different vector instructions, each of which occupies space in the instruction pipeline. Eliminating these vector instructions frees space in the pipeline, and reduces pressure on the memory system. In practice, grid-wide redundancy is both difficult to eliminate and less common than TB-wide redundancy. Therefore, we focus this paper on the elimination of TB-wide redundant instructions to improve both performance and energy-efficiency.

We find that most TB-level redundant instructions cannot be fully identified during static compilation, where the size and dimensionality of TBs is not yet known. The layout of thread indices in multi-dimensional TBs creates redundancy in the registers storing thread IDs. This redundancy propagates into dependent instructions, but only if certain TB-sizing conditions are met. A per-kernel runtime check of a TB's dimensions can be used to determine if conditionally redundant instructions are definitely redundant and avoids expensive vector register comparisons at runtime. Based on these observations, we propose *Dimensionality-Aware Redundant SIMT Instruction Elimination* (DARSIE), a TB-centric instruction skipping mechanism that identifies TB-redundant instructions using a combination of static compiler markings and runtime TB-sizing information. Once identified, TB-redundant instructions are skipped by the hardware before they are fetched. DARSIE uses a novel multithreaded register renaming and instruction synchronization technique to share the values from redundant instructions among warps in each TB.

Value sharing and instruction elimination is challenging to solve solely in the compiler or hardware. Alone, the compiler is not able to efficiently coordinate value sharing and instruction skipping between parallel warps. Likewise, it is expensive for hardware to detect redundancy in warp-wide vector registers, and complex to orchestrate a reactive mechanism that identifies skippable future instructions. DARSIE therefore uses a combined compiler/hardware approach to avoid these problems.

Contemporary GPUs from NVIDIA and AMD use a scalar functional unit and register file to perform operations on warp-wide redundant data identified by the compiler [2, 14]. Research has also sought to address warp-level redundancy by masking off lanes in the vector pipeline and skipping partial warp instructions when the SIMD width is less than the warp size [9, 19, 50]. Other work has proposed adding expensive value comparison hardware to the pipeline to remove redundancy at the issue stage, or reduce register file space via compression [20, 22]. In contrast, we use insights gained from creating a taxonomy of redundancy in TBs to identify both new opportunities for instruction skipping and ways to offload redundancy identification to the compiler. This allows DARSIE to avoid expensive value comparisons in hardware, and improve performance by skipping entire vector instructions before they are fetched in the frontend of the pipeline.

This paper makes the following contributions:

- We introduce a new taxonomy of redundancy for GPUs, focusing on the TB granularity. We show that the composition of TB-wide redundancy is highly dependent on the dimensions of the TB, and that thread index layouts in multi-dimensional TBs create ample implicit redundancy.
- We combine a static compiler pass, that marks conditionally redundant instructions, with runtime TB-sizing information to non-speculatively identify redundancy without value comparison hardware.
- We propose DARSIE, which combines our redundancy identification software with novel instruction skipping hardware to ensure that redundant instructions are fetched and executed only once per-TB. DARSIE leverages multithreaded register renaming and selective warp synchronization to share vector registers between warps in a TB, allowing TB-redundant instructions to be skipped in the fetch stage of the pipeline.

Through aggressive instruction skipping, DARSIE is able to reduce the number of instructions fetched and executed in 2D applications by 23%, improving performance by 30%, while decreasing energy consumption by 25%.

2 A taxonomy of GPU redundancy

To help understand why redundancy occurs across vector registers in the same TB, we introduce a new taxonomy of GPU redundancy. We claim that TB-wide redundancy has three classes: *uniform redundancy*, *affine redundancy* and *unstructured redundancy*. Uniform redundancy occurs when every lane in every warp of the TB contains the same scalar value for a particular named register. This typically occurs with shared constants and TB-invariant registers, like TB IDs and dimensions. Affine redundancy occurs when vector

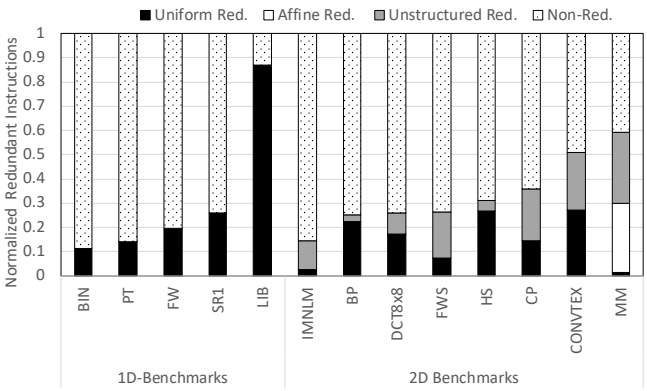

Figure 2. Fraction of dynamically executed TB-redundant instructions. Instructions executed in diverged control flow are considered non-redundant.

registers in different warps have the same value, which can also be represented as a (base,stride) pair. This redundancy occurs naturally in multi-dimensional TBs where consecutive lanes have consecutive threadId values, replicated in the private register space of each warp in the TB. Unstructured redundancy occurs when registers have the same vector values in each warp of the TB, but those values have no discernible pattern. In this paper, we show that all three types of redundancy can be non-speculatively identified at kernel launch time and can be eliminated without expensive runtime comparison hardware.

Figure 2 plots the breakdown of TB-redundant instructions under our new taxonomy for a set of benchmarks using either 1 or 2 dimensional TBs. Instructions are classified based on the type of redundancy in their source registers, and applications are subdivided by their TB dimensionality. Figure 2 shows that both affine and unstructured redundancy is pervasive in 2D TBs, but largely absent in 1D. The non-uniform redundancy in 2D TBs stems from the layout of the tid.x register. Consecutive threads within a warp are assigned consecutive tid.x values. When the TB's x dimension is <= the warp size, the per-lane values in the tid.x register repeat. For example, with a warp size of 4, and a TB with dimensions 4×4, the value in each of the 4 warp's tid.x register is (0,1,2,3). Furthermore, accesses to memory based on affine redundant addresses create unstructured redundancy. We observe that both affine and unstructured instructions can be marked *conditionally redundant* during static compilation, and definitely redundant when the TB's dimensions are known at kernel launch time.

Uniform redundancy is simple and can typically be traced back to a collection of known constant and intrinsic variables. These values are often TB invariant, such as TB IDs, and TB dimensions. We observe that most uniform redundancy is *definitely redundant*, and common in both apps with 1D and 2D TBs, as seen in Figure 2.

Affine and unstructured redundancy is more subtle and stems primarily from the GPU's threadId register values. Furthermore, they are directly related to the dimensions of an application's TBs. As Figure 2 shows, apps with 1D TBs have little to no affine or unstructured redundancy. However, it accounts for a significant portion of total instructions executed for apps with 2D TBs.

To understand how the TB dimensions affect redundancy, consider Figure 3. This pseudo-assembly code reads an integer value from an array with base address 10 and is indexed by each thread's tid.x value. We use a warp size of 4 with this example as well. Figure 3 (a) details the output register values for each instruction in a 1D threadblock with two warps. Each output register is classified based on the pattern it creates across all warps in the TB. In 1D TBs, the tid.x register is laid out sequentially across warps. This creates register values that are affine across the TB (TB-affine), but not redundant between warps. The two instructions that compute the address for the array lookup are also affine and not redundant because they are based on a 1D tid.x. The load instruction reads from the affine address, and the value loaded into R3 is input data-dependent. R3 is neither redundant nor affine in the TB.

Figure 3 (b) illustrates what happens to the same code when the TB is 2D (x=4,y=2). In multi-dimensional TBs, threadIds are assigned to warps by varying the x dimension first, and consecutive threads in a warp will be assigned consecutive tid.x IDs. Both warp 0 and warp 1 have tid.x registers that are affine within the warp, and redundant between warps (affine redundant) in this example. The resulting values computed for R1 and R2 are, therefore, also affine redundant. Since the address computed is redundant between the two warps in the TB, both instructions that load R3 will produce the same value. R3 is an example of unstructured redundancy. Each warp has an identical value, but there is no clear pattern in the values themselves since they are input data-dependent. In this paper, we classify patterns that cannot be represented with a single <base,stride> pair as unstructured redundancy.

Although Figure 3 is a simple example, the conditions under which affine and unstructured redundancy occurs can be generalized. For 2D TBs, each warp's tid.x register will be redundant if the x-dimension is <= the warp size and a power of 2. We refer to this as *conditional redundancy*, since it depends on TB's dimensions that are known at kernel launch time. While this redundancy starts at the threadId registers, it propagates throughout the program through register dependencies. Furthermore, memory instructions that load values from definitely or conditionally redundant addresses will take on the redundancy characteristics of the address they load. These observations also apply to 3D TBs, where both the tid.x and tid.y registers can be conditionally

Memory
Address: [10,14,18,22,26,30,34,38]
Value: [7 , 3, 0,90,55, 8,22,1]

(a) 1D threadblock (xdim=8,ydim=1)

Output Reg. Type	Output Register Value		Instructions
	Warp 0 (tid.x=[0,1,2,3])	Warp 1 (tid.x=[4,5,6,7])	
TB-Affine	[0,4,8,12]	[16,20,24,28]	MUL R1, tid.x, 4
TB-Affine	[10,14,18,22]	[26,30,34,38]	ADD R2, R1, #10
Unrelated	[7,3,0,90]	[55,8,22,1]	LD R3, MEM[R2]

(b) 2D threadblock (xdim=4,ydim=2)

Output Register Value		Output Reg. Type
Warp 0 (tid.x=[0,1,2,3])	Warp 1 (tid.x=[0,1,2,3])	
[0,4,8,12]	[0,4,8,12]	Affine Red.
[10,14,18,22]	[10,14,18,22]	Affine Red.
[7,3,0,90]	[7,3,0,90]	Unstructured Red.

Figure 3. Pseudo-assembly code to read from an integer array with a base address of 10 using tid.x as the index with 1D and 2D TBs. Values in output registers for each instruction are classified based on the pattern they make across the TB. 1D TBs create affine values that are not redundant, while 2D TBs create both affine and unstructured redundant values.

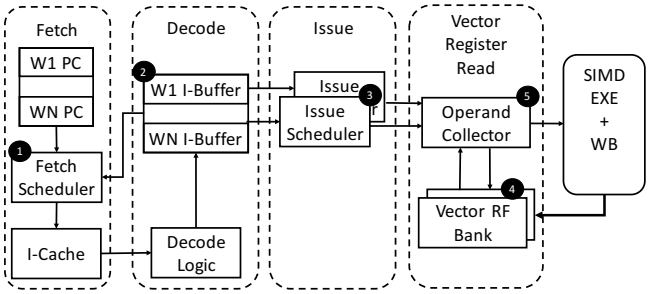

Figure 4. Baseline GPU microarchitecture

redundant. This is the first work to observe that this redundancy can be identified prior to execution and optimized out by hardware at runtime.

3 Baseline

Figure 4 presents our baseline microarchitectural model. Each cycle, a fetch scheduler (❶ in Figure 4) in the frontend of the pipeline initiates a fetch for one of the warps assigned to the core. This scheduler uses loose-round-robin (LRR) prioritization, and initiates instruction cache (I-cache) fetches based on which warps have empty instruction buffer (I-Buffer ❷) entries. Each warp has a two-entry I-buffer that is used to decouple the SIMT frontend (which fetches one PC for an entire 32-thread warp) from the SIMD backend. Each cycle, multiple issue schedulers select at most two instructions from one warp each for execution. Warps are statically partitioned among these issue schedulers.

Once selected to issue, instructions must read their source operands (32-element vector registers with 32-bit elements) from a highly banked register file (❹). To avoid excessive bank conflicts and facilitate high bandwidth, an operand collector [27] schedules register file reads in a way that limits stalls. These operand collectors are the inputs to the execution stage of the pipeline. The mapping between <named vector register, warp ID> pairs and physical vector register contents is programmable, and based on a mapping table initialized when a TB is launched on an SM. This is necessary since each warp can be assigned a different number of registers at compile time.

There is little documentation on how this register mapping is achieved, so we make the assumption that blocks of registers are assigned to warps using a simple base register + length mapping table. This avoids storing a unique <named vector register, warp> pair for each physical vector register. We also assume this mapping is done in the operand collection phase. Contemporary GPUs (like the Pascal card we model) do not have scoreboard logic embedded in the core, but rather encode dependencies in their instruction stream. Variable-cycle memory instructions are controlled via *depbar* instructions that ensure source registers read by instructions have received responses to their memory requests before a dependent instruction is issued.

4 Dimensionality-Aware Redundant SIMT Instruction Elimination

DARSIE ensures that TB-redundant instructions are fetched and executed only once in each TB. As all instructions are 64-bits in length, redundant ones can be skipped in the frontend of the pipeline by simply adding eight to the program counter. However, detecting and marking redundancy, sharing values of skipped instructions, properly handling redundant load instructions, and handling branches with the potential for divergence requires more care. Therefore, we start by presenting DARSIE's operation at a high level in Section 4.1. Here, we introduce our extensions to the compiler that marks instructions as either redundant, conditionally redundant, or non-redundant. This is explained in full detail in Section 4.2. We next describe our changes to the microarchitecture, explained fully in Section 4.3. We additionally describe how DARSIE handles the skipping of load instructions, and its

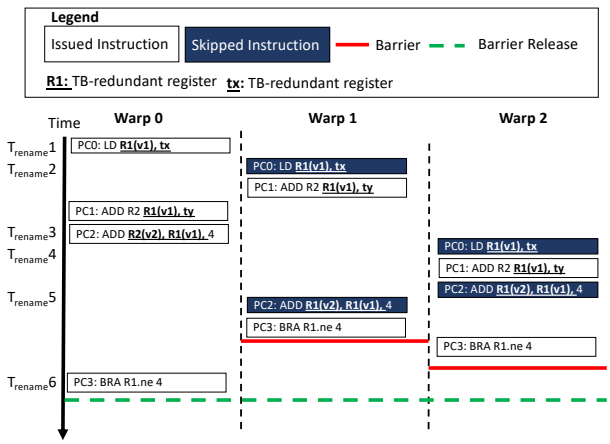

Figure 5. DARSIE's Instruction Skipping Flow: Branch instructions always force a TB-wide barrier to determine what the majority-path is. In this example, TBs are three warps wide.

operation in the presence of divergence. These are explained fully in Sections 4.4 and 4.5 respectively.

4.1 High level operation

DARSIE operates by first declaring PCs as *skippable*. Using a novel compiler pass, we detect when values are conditionally-redundant across a TB. At kernel-launch time, every static PC in the program is marked at as either TB-redundant or not. The hardware itself treats uniform, affine redundant, and unstructured redundant instructions the same. Section 4.2 details these compiler markings. The compiler analysis also assumes that warps in a TB are executed in lock-step, such that there is only one version of each TB-redundant instruction at any given time. There are two options in hardware to ensure that warps read the correct version of a TB-redundant register without enforcing costly lock-step execution on every instruction: (1) Synchronize the warps in the TB when any warp writes to a TB-redundant register. This ensures there is only ever one live version of a particular TB-redundant register. (2) Store multiple versions of each TB-redundant register such that each warp can progress at a different pace. As long as warps are executing on the same control-flow path (hence executing the same instructions in the same order), the correct register version for each warp can be attained by keeping track of writes to the register. When warps in the TB take different control flow paths, DARSIE continues operating on the path taken by the majority of warps. To avoid excessive synchronization, DARSIE adopts solution (2).

In hardware, DARSIE elects a single warp from a TB to execute a redundant instruction, and share the result with other warps in the TB. We call the warp that executes the instruction the *leader warp*. Warps that skip the instruction and read the value are referred to as *follower warps*. Subsequent instructions in follower warps that are dependent on the result of an eliminated instruction access the leader warp's values through a multithreaded register renaming mechanism. The existing programmable GPU register mapping mechanism helps facilitate this process, as each warp has a configurable number of physical registers based on the per-thread register demands of an application. Our updates to this mechanism are described in Section 4.3.1.

Figure 5 provides a visual overview of DARSIE's operation. In Figure 5, warp 0 becomes the leader for instruction PC0 at T_{rename} 1 because it arrives first. Warp 0 executes the instruction and stores the result as R1(v1), since this is the first write to R1. At time T_{rename} 2, warp 1 skips PC0, updating its register mapping table to point to R1(v1) which warp 0 just produced. Warp 1 then executes PC1, which is a true vector instruction (since ty is not TB-redundant). Warp 0 does the same. At time T_{rename} 3, warp 0 writes to R1 again. Each time a redundant register is written, we create a new version of the register tagged with the number of times it has been written by this TB. At this point, there are now 2 active versions of R1 across this TB. At T_{rename} 4, warp 2 finally reaches PC0, skips the instruction then executes PC1 using the old (v1) version of R1. Warp 2 uses R1(v1) for its source operand because there has only been 1 write to R1 in warp 2's instruction stream. Warp 2 then skips PC2, and increments the number of writes the warp has seen to R1. At T_{rename} 5, warp 1 skips PC2. Since all the warps in the TB are done with R1(v1), it can be released. Warp 1 executes the branch instruction, then waits for all other warps in the TB to reach this branch. Synchronization at branches is necessary to ensure that all warps skipping instructions in a TB execute on the same control-flow path. Warps that diverge off the majority control-flow path are no longer considered for skipping. Similarly, warps with intra-warp control-flow divergence do not participate in instruction skipping. Note that, except at branch instructions, the warp scheduling order in DARSIE is not prescribed. The scheduler will still make throughput-oriented decisions. The ordering in this example is therefore one of many that are possible.

4.2 Compiler annotations

DARSIE's static compilation phase detects both definitely and conditionally redundant registers and instructions. In relation to our taxonomy described in Section 1 and Section 2, uniform redundant values are always definitely redundant. Affine and unstructured redundant values are conditionally redundant. The compiler starts by identifying all the intrinsic values known to be uniform across a TB. The values we consider in this work are: blockIdx, blockDim, scalar constants, global kernel input parameters and the base value of shared memory. These values are all marked as definitely redundant. Next, the compiler marks intrinsic registers that are conditionally redundant. Based on our observations in

```
// Loading Parameters and thread ID
      PC
DR    0x000      cvt.u32.u16 $r2, %ctaid.y;
...
CR    0x068      cvt.u32.u16 $r4, $r0.lo // $r0.lo: tid.x
...
CR    0x0f0      add.u32 $10, $r4, 0x0000040c
...
CR    0x108      shl.b32 $ofs3, $r10, 0x00000007
...
// Start of loop
V     0x150      l0x00000150: ld.global.u32 $11, [$r1]

// Unrolled Loop
CR    0x178      mov.u32 $r0, s[$ofs3+0x0000];
CR    0x180      add.u32 $ofs4, $ofs3, 0x00000080;
V     0x188      mad.f32 $r10, s[$ofs2+0x0000], $r0, $r10;
CR    0x190      mov.u32 $r0, s[$ofs4+0x0000];
CR    0x198      add.u32 $ofs4, $ofs3, 0x00000100;
V     0x200      mad.f32 $r10, s[$ofs2+0x0004], $r0, $r10;

...

V     0x480      set.le.s32.s32 $p0/$o127, $r8, $r9;
V     0x488      add.u32 $r1, $r1, 0x00000080;
V     0x490      add.u32 $1, $1, 0x0000080;
V     0x498      add.u32 $r5, $r6, $r5
V     0x500      @$p0.ne bra 10x00000150;

// End of loop
...
```

Figure 6. Example of compiler marking TB-redundant instructions for matrix multiply kernel. DR:Definitely Redundant, CR:Conditionally Redundant

Section 2, both the threadIdx.x and threadIdx.y registers are conditionally redundant and depend on the TB dimensions known at runtime. All studied applications use 2D TBs at most, as is typical for GPU workloads. We therefore limit the analysis to only threadIdx.x. All other registers are considered true vector registers. The compiler then creates the program-dependence graph and iteratively propagates our redundancy information through registers and instructions. Unlike previous works that focus exclusively on finding affine and uniform instructions [11, 19, 45], we introduce and exploit *conditional* redundancy as well. As detailed in Section 1, conditionally redundant instructions comprise a significant portion of total executed instructions for applications with 2D TBs. Load instructions that access redundant or conditionally redundant addresses (and their corresponding output registers) are also marked. If more than one of our three redundancy definitions (redundant, conditionally redundant or vector) reaches a source operand of an instruction, we assign the weakest of the definitions.

This analysis assumes that warps within the TB proceed through the program in lock-step. Since enforcing this requirement can be expensive (or impossible if warps traverse different control-flow paths), we rely on hardware to create the illusion that warps leveraging DARSIE are proceeding in lockstep (described in Section 4.3.3) with respect to TB-redundant operations.

Promoting conditionally redundant registers to definitely redundant requires runtime information about a TB's dimensions. For example, if the there are 32 threads in the x-dimension, then the tid.x value per warp will vary from [0 to 31]. Likewise, if there are 16 threads in the x-dimension, each warp will have tid.x IDs [0 to 16]. In both cases, values based on tid.x will be redundant across the TB, and repeat for every warp. Conditionally redundant instructions are evaluated at kernel launch time based on the kernel's specified TB size, and are static for the duration of the kernel. This marking could either be implemented in the GPU driver's JIT-ing finalization pass, or determined with a minor hardware modification that compares conditionally redundant instructions to the launched TB size. Supporting CUDA dynamic parallelism, where the GPU launches kernels to itself is possible with latter option, as the code does not need to be recompiled. In either case, the check simply tests if the kernel has 2D TBs, and that the width of the x-dimension is a power of 2, and less than or equal to the warp size. If so, conditionally redundant instructions are marked as definitely redundant, or are otherwise marked as true vector instructions. We note that the majority of multi-dimensional GPU applications meet the above x-dimension criteria. Of the 128 unique 2D kernels from the application surveyed in Section 1, only one fails to meet this requirement.

Figure 6 illustrates the compiler annotations made for the matrix multiply kernel. Note that this code is register-allocated PTXPlus code, which is used for all our experiments. As shown in Figure 6, the value *threadIdx.x* propagates to the register *$ofs3*. Thus, the unrolled loop in the program contains 2 redundant instructions and one true vector operation. This highlights the granularity at which redundancy elimination takes place using DARSIE.

The compiler can only mark static instructions as being skippable based on an analysis that assumes that threads in a TB are executing in lock-step. It is up to the hardware fetch scheduler to ensure that all warps in the TB skip the same version of the redundant instruction. For example, if a redundant instruction is in a loop, all threads skipping the instruction must be on the same iteration of the loop. This ensures that dependencies between loop iterations are maintained for each warp. Ordering is enforced using either register-versioning or forcing the hardware to barrier when TB-redundant registers are written.

We note that the compiler changes in DARSIE do not change the instruction stream in any way, other than adding hints about redundancy. This is dissimilar to techniques like DAC [45] that completely transform the compiled code into a format requiring hardware support for affine and non-affine instruction streams. If the hardware does not support DARSIE, the markings are simply ignored. Likewise, the hardware does not require the compiler to support DARSIE. Binaries compiled without DARSIE markings will run seamlessly on DARSIE hardware, but without instruction skipping.

We encode the three-state <vector, conditionally redundant, redundant> classification in two bits of the GPU's virtual ISA (PTX in NVIDIA) that is produced by the static compiler. GPUs employ a two-step compilation process where the virtual ISA in the binary is transformed into the real machine instruction set (known as SASS) when the kernel is launched. Although the encoding of SASS is proprietary, reverse engineering efforts indicate that there are many unused bits in this 64-bit RISC-like ISA [31]. We use one of these extra bits to encode if an instruction is TB-redundant or not, which is known when the SASS is loaded into the GPU. Two bits would be required to maintain the three states of redundancy if the decision is delayed until after the code is JIT compiled.

4.3 DARSIE microarchitecture

Figure 7 shows the changes to the microarchitecture needed to support DARSIE. We add the instruction skipping hardware in the fetch stage, consisting of a PC coalescer (Ⓐ in Figure 7), a PC Skip Table (Ⓑ) and fixed-size adders to allow each PC to be incremented by 8 (Ⓒ).

4.3.1 Remapping registers . To enable dynamic remapping of vector registers, we add a register renaming table that is probed prior to looking up the register in the baseline's linear register renaming table (Ⓓ). This allows follower warps to read register values produced by leader warps. Registers in contemporary GPUs are not mapped on a per-thread basis, but rather on a per-warp basis. There is a 1:1 correspondence between vector lanes and threads within the warp, and cross-lane communication is generally not supported outside of special instructions. DARSIE does not change this assumption and remaps whole vector registers. Our register renaming table contains one entry for every currently renamed register in each warp. The register rename table, version table, and physical register freelist implement the versioning detailed in Figure 5.

The register rename table maps <warp,reg#> pairs to this warp's <reg#,version#> pair. A separate version table (Ⓔ) stores the <reg#,version#> to physical register mapping. Both the version and rename table are banked on a per-TB basis. When a kernel with TB redundancy is launched, we allocate a portion of the physical register file space for renaming. Many GPU applications are not limited by the register file size, so this allocation will not typically affect occupancy. How much register space to consume could be made on an application basis. In this work, we allow DARSIE to consume up to 32 vector registers per TB for renaming. DARSIE uses as many registers as it can before affecting occupancy when registers are limited.

We allocate our renamed register space in a strided fashion across the vector RF banks at kernel launch and maintain a physical register freelist (Ⓕ). Physical registers are freed when a register version number is no longer in flight for

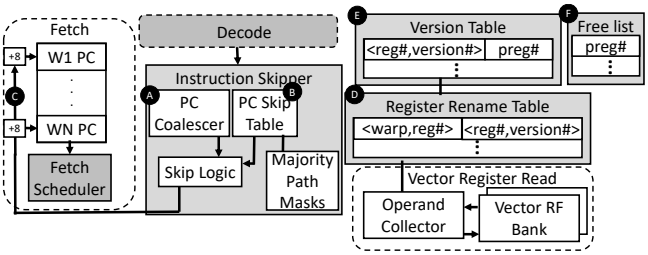

Figure 7. Detailed breakdown of DARSIE uarch operation.

the TB. When the freelist empties, synchronization must be performed to ensure all required versions of a register are available. Our evaluation accounts for the increase in register bank conflicts that occur when all follower warps attempt to read from the renamed register's space.

4.3.2 PC skip table . DARSIE skips instructions in the front of the pipeline before the I-cache is probed. To achieve this, we add hardware that acts in parallel with the fetch scheduler, to skip some instructions while initiating a fetch for another. This effectively increases our throughput at fetch without increasing the width of any of existing structures like the fetch scheduler and I-cache. The instruction skipper relies on our compiler annotations to decide which instructions should be skipped, and the PC Skip table (Ⓑ) controls the skipping logic. Each entry in the Skip PC table contains five fields:

1. **PC** : The program counter that should be skipped
2. **Warps waiting bitmask** : A mask that indicates which warps are waiting at this PC to skip it. Required if synchronizing between warps.
3. **Majority-path bitmask**: A mask with 1-bit per warp in a TB that indicates which warps are executing on the majority-path.
4. **IsLoad**: A bit that is 1 if this instruction is a load instruction. This is necessary since load instructions must be removed from the Skip PC Table if a store is executed, or if global atomics/synchronization events occur.
5. **LeaderWB**: A bit that is 1 if the leader warp has written back the redundant-register value. To ensure correct operation, follower warps must wait for the leader to writeback before they can leave the skippable instruction.

4.3.3 Achieving the illusion of lockstep execution . Our skipping mechanism is dependent on all warps having the same branch history as the leader warp (i.e., all warps following the same control-flow path). To ensure this condition, we synchronize TBs at branch instructions. The path with the majority of warps will continue skipping. Warps on any other path will not. We store 1-bit per warp to indicate if it is on the TB-majority path. When warps deviate from

the path, their bit is cleared. These bits are all set back to one upon the execution of *syncthreads* instructions which require the entire TB to be in sync.

4.3.4 PC coalescer. A PC coalescer is used to minimize the skip table read port requirements (Ⓐ). The PC coalescer acts like the global memory coalescer in the load/store unit, except instead of coalescing global memory addresses to cache lines, it coalesces PCs based on exact matches. This helps limit the number of accesses made to the Skip PC Table each cycle. The PC skip table contains one entry for each PC currently being skipped. We experimentally determine that the PC coalescer reduces the port requirement on the PC skip table to 2, while providing both reasonable throughput and minimal area and energy overheads.

4.3.5 Instruction skipping flow . After TB-redundant instructions are decoded, the PC skip table is probed to see if they are currently being skipped. If there is no PC skip table entry and the accessing warp is on the majority-path, it becomes the leader warp. Upon the creation of a new leader warp, an entry is created in the PC skip table, a new physical register is taken from the freelist and allocated in the version table, the LeaderWB bit is cleared, and the leader updates its register's version number in the register renaming table. If there are no other entries in the freelist, the warps waiting bitmask is updated to indicate that this instruction will act as a synchronization point. Only warps on the majority control-flow path can skip instructions. When the leader writes back a TB-redundant value, it updates the LeaderWB bit.

When another warp in the TB gets to the PC being skipped, the PC skip table is probed, and an entry is found. If the warps waiting bitmask is empty, and the leaderWB bit is set, this follower warp is able to skip the instruction. If the warps waiting bitmask has a non-zero value, it is updated to indicate that the new warp is now waiting for all other warps in the TB to reach the TB-skippable instruction. The follower warp then updates its version number in the register renaming table to reflect the fact that it needs to read a newer version of this register. If this was the last warp in the TB using a particular register version number, the physical register is returned to the freelist. The PC of the skipping warp is then incremented by 8 (Ⓒ). If synchronization is necessary, the warp is removed from the fetch scheduler.

As more warps from the same TB reach the PC to be skipped, their PCs are incremented, and their registers' versions updated. If synchronization is necessary, we determine if all the warps on the majority control-flow path have arrived at the instruction to be skipped by matching the warps waiting bitmask with the majority-path bitmask for this TB. Once all follower warps in the TB have skipped the instruction, it is removed from the PC Skip Table.

When warps leave the majority path, they copy their redundant register values into their warp-private space, and clear their state in the register renaming and version table. We also note that the execution of warps is different than execution of individual scalar threads, in that a warp may proceed in both branch directions using the SIMT stack. Our technique is not applied in the presence of SIMD divergence (see Section 4.5). If SIMD divergence is encountered on a branch, the warp is no longer considered for skipping and is removed from the majority path.

4.4 Skipping load instructions

Skipping load instructions presents a unique challenge since memory dependence information is not embedded in the decoded instruction. Without complex and potentially expensive memory dependence tracking hardware, we cannot guarantee that a store instruction does not update memory at the location a skipping load instruction reads. To simplify the design, complexity and size of our proposed hardware, DARSIE avoids the memory dependence problem by removing load PCs from the skip table when one of two events happens: **(1) This TB executes any store instruction**: Stores are relatively infrequent, so we conservatively assume that any store can update memory referenced by any load instruction to be skipped. **(2) Any global communication primitives are executed**: Our baseline GPU does not guarantee any particular memory ordering between TBs executing on different SMs, or TBs on the same SM, unless global communication primitives are used. When we detect that an SM executes any instruction used to perform global communication, such as global atomic instructions, we remove all global load PCs from the skip table. In our benchmarks, and the in the bulk of contemporary GPU workloads, these global communication primatives are not used.

4.5 Handling SIMD divergence

DARSIE specifically targets highly regular code that does not exhibit large levels of SIMD divergence. This is the common case for GPU applications. We therefore simplify our design by not skipping instructions with inactive threads in its active mask. While we note that divergent workloads exist, prior work from industry has shown them to be a minority of contemporary applications run by GPU customers [36]. We evaluated the effects of allowing diverged instructions to be considered redundant, but found that it provided minimal returns. We note that warp-level control-flow divergence is different from SIMD divergence. Warp-level divergence indicates that the entire warp took a different execution path, and not just some threads within a warp. If warp-level divergence occurs, instruction skipping is still possible among warps that traverse the majority control-flow path.

5 Methodology

We use GPGPU-sim v4.0 [1] and GPUWattch [24] to estimate the performance and energy consumption of DARSIE respectively. We simulate our applications using PTXPlus,

Table 1. Applications studied

Name	Abbr.	TB dim	Name	Abbr.	TB dim
binomial-Option [32]	BIN	(256,1)	ImageDenois-ingNLM [32]	INLM	(16,16)
pathfinder [8]	PT	(1024,1)	Backprop [8]	BP	(16,16)
fastWalsh-Transform [32]	FW	(256,1)	DCT8x8 [32]	DCT	(8,8)
SRADV1 [8]	SR1	(512,1)	Floyd-Warshall [7]	FWS	(16,16)
LIB [4]	LIB	(256,1)	HotSpot [8]	HS	(16,16)
			CP[4]	CP	(16,8)
			convolution-Texture [32]	CONVTEX	(16,16)
			MatrixMul [32]	MM	(32,32)

C: CUDA SDK [32], P: Parboil benchmark [41], R: Rodinia benchmark suite [8], I: GPGPU-sim distribution benchmark [4]. P: Pannotia benchmark [7]

Table 2. Baseline GPU

Parameters	Values
GPU	Pascal (GTX1080Ti), 28 SMs, 64 warps/SM
	32 thread blocks/SM
SM	32 SIMD Width, 2K vector registers per SM
Scheduler	4 warp scheduler/SM, GTO scheduling
L1	96KB shared memory/SM
Register	14.2pJ/read 25.9pJ/write [24]

and extension of NVIDIA's virtual ISA PTX that is converted from the native machine ISA SASS. We use the NVIDIA GTX-1080Ti Pascal GPU as our baseline. Table 2 describes our baseline, and is verified to have a 90% correlation to the real card. We swept different warp schedulers and observed that these regular applications are insensitive to scheduler choice, with GTO being the best performing option. We implement DARSIE's compiler pass inside GPGPU-Sim on register-allocated PTXPlus code, similar to the methodology used by Wang et al. [45]. We use Cacti 7.0 [47] to model the energy and area overhead of DARSIE's additional hardware components. The PC skip table has 2 read ports, 1 write port and 1 entry per TB. Each SM also contains a register rename table, which has 32 entries per TB, based on the maximum number of registers renamed in our applications. We compare DARSIE as described in Section 4 with two previously proposed techniques:

Uniform Vector (UV): UV [50] is a recently proposed technique to remove redundant inter-warp instructions. UV makes use of an instruction reuse buffer [40] to eliminate instructions that read uniform scalar register values. UV prevents instructions from executing at the issue stage of the pipeline after being loaded into the instruction buffer. It does not consider non-uniform redundant vectors, and does not skip memory operations. We choose this technique to compare against because it is the only related work to remove redundancy without major pipeline modification. A more detailed description of UV is located in Section 7.

Idealized Decoupled Affine Computation (DAC): DAC-IDEAL [45] proposes a compiler and hardware mechanism that detects affine (not necessarily redundant) operations.

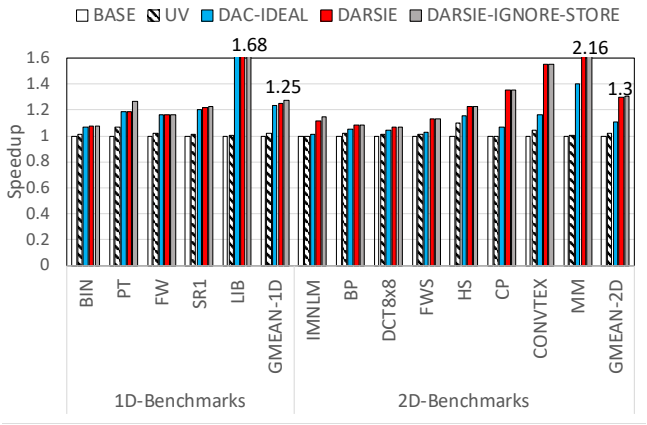

Figure 8. Performance of DARSIE against prior work. Speedup is normalized to the baseline GPU.

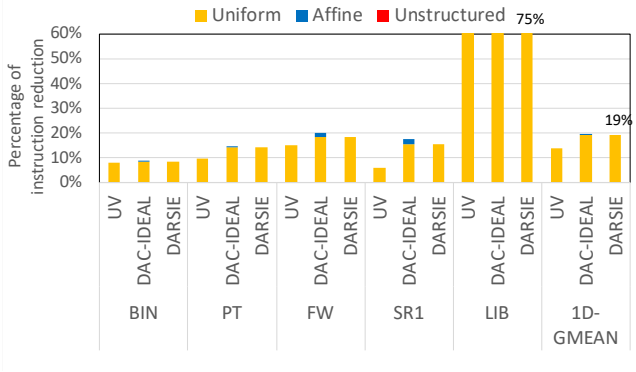

Figure 9. Percent reduction in 1D benchmark instructions versus the baseline

The DAC compiler separates instructions into affine and non-affine streams, and synchronizes the two when the vector stream reads values from the affine stream. We model an idealized version of DAC by detecting affine instructions at runtime, and assuming that all affine instructions (both redundant and otherwise) will be executed only once. We also assume there is no synchronization cost between affine and non-affine instruction streams. This implementation was validated to be as good or better at instruction reduction compared to the original results in [45]. We choose DAC-IDEAL to compare against because it covers both uniform and affine redundancy, and is the most recently proposed technique.

6 Experimental Results

The following subsections evaluate the performance and energy-efficiency of DARSIE, the effects of synchronization and provide an area estimate.

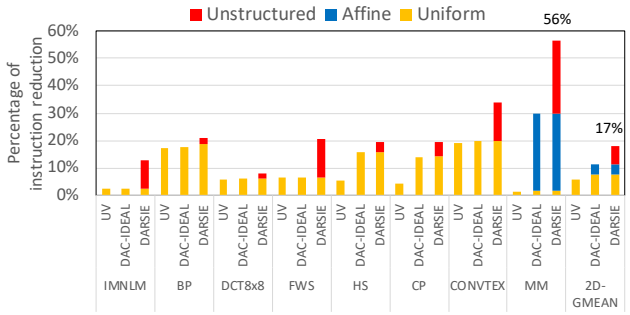

Figure 10. Percent reduction in 2D benchamrk instructions versus the baseline

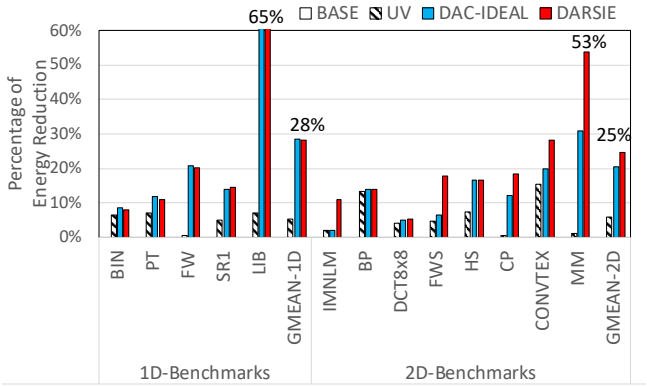

Figure 11. Percent Energy Reduction versus the baseline

6.1 Performance and energy

Figure 8 compares the speedup of UV [50], DAC-IDEAL [45], and DARSIE over our baseline GPU. DARSIE achieves a geometric mean speedup of 1.3, significantly better than UV (1.02) and DAC-IDEAL (1.11) for benchmarks with 2D TBs. DARSIE-IGNORE-STORE doesn't reset the skip table in the occurrence of store instructions. DARSIE significantly outperforms the two alternatives because of the elimination of the unstructured redundancy in 2D benchmarks. As mentioned in Section 1, neither UV nor DAC-IDEAL remove unstructured redundancy. UV is typically limited by fetch throughput since it can only remove uniform redundancy at the issue stage. DARSIE has significantly higher instruction skipping bandwidth because it can skip multiple instructions in a single fetch cycle with only an increment of the PC. DAC-IDEAL's performance with 1D TB applications is roughly equal to DARSIE's since it is similarly able to remove all uniform and affine-redundant instructions. To evaluate the effect store instructions have on performance, DARSIE-IGNORE-STORE doesn't reset the skip table when store instructions occur and demonstrates that the performance impact is minimal. Further investigation reveals that stores usually occur at the end of the register-use chain. Therefore; the value in the register is typically not used again after the store, so clearing it's redundancy data has little effect on DARSIE's performance. Since DARSIE remaps follower warps to the same register bank, it does cause additional register file bank conflicts. However, we find that artificially removing all DARSIE-induced bank conflicts results in just a 1% performance improvement.

The performance gain of DARSIE is not always proportional to the number of instructions eliminated. Some memory-bound applications have a high number of redundant compute operations, but few redundant memory accesses. For example, DARSIE improves the performance of FWS by 13%, despite the fact that 21% of its instructions are skipped. This is because memory operations dominate the application runtime but are not redundant. Conversely, MM has a significant number of unstructured-redundant accesses to shared memory. MM tiling causes multiple warps in one TB to access the same shared memory blocks with affine memory addresses. This results results in excessive affine and unstructured redundancy.

Figures 9 and 10 plot the number of instructions eliminated by DARSIE and prior work. In Figure 10, DARSIE decreases instructions by a geometric mean of 17% in 2D TB benchmarks. UV [50] is able to remove uniform redundant instructions, but doesn't improve performance. UV removes instructions in the execute stage of the pipeline, requiring them to still be fetched and decoded. In applications like LIB, the fetch bandwidth becomes the bottleneck. Since both DARSIE and DAC eliminate instructions before they are fetched, they are able to see significant performance gains in LIB. DAC-IDEAL [45] eliminates redundant instructions by a geometric mean of 11%. We make the idealized assumption that DAC-IDEAL is able to remove all affine values in both 2D and 1D applications, but is not able to remove the unstructured redundancy we identify in this paper. We also assume that DAC-IDEAL is able to remove *non-redundant* affine values that occur in 1D applications for example, tidx.x in Figure 3(a). Only DARSIE removes unstructured redundant instructions, which accounts for the improvements over UV and DAC-IDEAL in 2D TB benchmarks. As a result, DARSIE is able to match the performance of DAC-IDEAL on 1D benchmarks, while outperforming DAC in 2D applications.

Figure 11 shows the total energy consumption of UA, DAC-IDEAL and DARSIE is normalized to the baseline GPU. DARSIE reduces energy by a geometric mean 25%, while UV and DAC-IDEAL reduce energy by a geometric mean of 7% and 20% respectively. This improvement can be traced back to our microarchitecture preventing redundant instructions from even probing the I-cache and saves energy throughout the pipeline. The overhead of DARSIE is only 0.95% of the dynamic energy consumption. Most of the overhead comes from accessing the PC Skip Table, majority path mask and register rename table. This minimal energy overhead stems

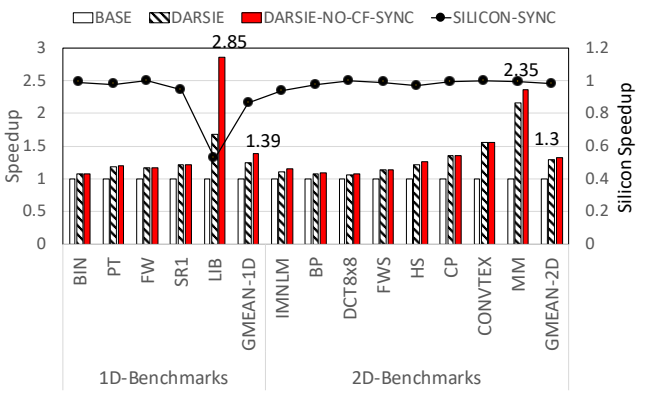

Figure 12. Effects of Synchronization.

from the small size of the added hardware (roughly 82 bytes for the majority path mask, and 84 bytes per TB bank).

6.2 Effect of Synchronization

Figure 12 presents the performance of an idealized DARSIE, *DARSIE-NO-CF-SYNC*, that has no DARSIE-related synchronization. To measure the effect of DARSIE's synchronization overhead, without any of DARSIE's benefits, on a real machine, we instrumented the applications with __syncthreads() calls at basic-block boundaries and measured their performance. SILICON-SYNC in Figure 12 plots the effect synchronization has on performance on a silicon NVIDIA Pascal Titan X GPU. The overhead in most applications is small. Interestingly, many of the 2D applications already had __syncthread() operations at basic block boundaries, limiting DARSIE's synchronization effects. On LIB, there is a 50% performance reduction becasue the baseline application contains no __sycthreads(). However, the 75% instruction reduction DARSIE provides (Figure 10) on LIB makes up for the overhead.

6.3 Area Estimation

The three major sources of area in DARSIE are the PC Skip Table, majority path mask and register renaming/version tables. One PC Skip Table entry includes a PC value, the warp waiting bitmask (which consists of one bit for each warp that can be allocated on one TB), a bit to indicate if this instruction is a memory load (IsLoad) and a bit to indicate if the leader warp's output register has been written back (LeaderWB). One TB is allocated 8 PC skip table entries that are replaced dynamically. These fields consume 82 bits: 48 bits for the PC + 32 bits for the warp mask (since there are at most 32 warps can be allocated by one thread block) + 1 bit for the IsLoad flag + 1 bit for LeaderWB. The PC Skip Table is 256 entries based on their being at most 32 TBs in one SM, and consumes 20092 bits (2624 bytes). DARSIE allocates one majority path mask entry for one TB in one SM. These fields cost 32 bits for the warp bitmask. The total size of majority path mask is $32 \times 32 = 1024$ bits (128 bytes). We conservatively estimate that each entry in the register rename and version table consists of 21 bits: 8 bits for the named register (CUDA allows 255 potential named registers per thread) + 8 bits for physical register tag + 5 bits for the version numbers. DARSIE allocates 32 entries for one TB, based on the max register usage of our workloads (32). These entries therefore consume: 21×32 (entries per TB) $\times$ 32 (TBs in one SM) = 21504 bits (2688 bytes). Altogether, the additional structures consume an additional 5.31 kB (2.1% of the Pascal GPU register file size).

7 Related Work

Instructions operating on identical data has long been observed in CPUs [5, 25, 26, 29, 40, 46]. Recent GPU work [9–11, 19, 45, 49, 50, 52] has targeted the removal of GPU instructions. Recent work by Wang and Lin [45] proposes Decoupled Affine Computation (DAC) that uses the compiler to identify and isolate an affine instruction stream that is run on a separate pipeline from the SIMT instruction stream. DAC captures the run-ahead effect of Decoupled Access Execution [39] as well as achieves a reduction in SIMT instructions by computing affine *base* + *stride* values only as needed in the affine stream. In contrast, DARSIE exploits redundant instructions, which are fundamentally different than affine instructions. The unstructured redundancy eliminated by DARSIE cannot be eliminated with affine function units. Xiang et al. [50] identified inter-warp uniform values in the decode stage, and skips selective, uniformly redundant instructions using an instruction reuse buffer [40]. Unlike Xian et al.'s design, DARSIE skips redundant instructions before they are fetched, based on the pre-emptive detection of TB-level redundancy. Kim et al. [19] presents a fine-grain(FG-SIMT) execution engine to tackle instructions composed of *affine* and *uniform* value structures. This FG-SIMT architecture aims to improve performance and energy efficiency for irregular kernels, focusing primarily on scalar instructions.

Lee et al. [22] proposes compressing GPU vector registers to save energy. Esfeden et al. [3] proposes a register packing mechanism using renaming to that helps save energy and increase performance by combining reads to multiple registers into a single access. GPU compiler work on scalarization [10, 18, 23, 51] discovers invariant instructions, and re-allocates registers to improve performance.

Recent approximate computing research [30, 48] concentrates on removing the execution of similar value structures to reduce energy consumption. Daniel et al. [48] observed operand value similarity within a warp. Their approximating warp micro-architecture [48] can both detect value similarity, and reduce the execution of identical data across SIMT lanes. G-scalar [28] found that 45% of divergent instructions are eligible for scalarization. G-scalar [28] compares values of the registers and compresses them to reduce the usage

Table 3. Comparison of DARSIE to related work

	WIR [20]	G-Scalar[28]	UV [50]	GP-SIMT [19]	DAC [45]	DARSIE
Uniform Redundany	✓	✓	✓	✓	✓	✓
Affine Redundancy				✓	✓	✓
Unstructured Redundancy						✓
Min. Pipeline Modifications			✓			✓

of the register file. Concurrent work on Warp Instruction Resuse (WIR) [20] saves energy by reusing registers with identical operand values across warps through a signature-based renaming mechanism. Unlike DARSIE, WIR relies on a complex, hardware-based redundancy detection mechanism, and is still bottlenecked on fetch/issue bandwidth.

Volkov [43] restructured GPU programs to reduce their occupancy and improve performance. They manually collapsed code for a particular application, and demonstrated that an increase in register density resulted in improved performance. Yang et al. [51] performed a similar merging to manage parallelism in an optimizing pass of the compiler. Work on TB compaction uses TB-wide synchronization [13] to help orchestrate thread packing in code.

Contemporary GPUs are designed to perform well when executing regular kernels with limited control-flow divergence. As a result, these regular applications are the most commonly run in the field today [36]. Although these applications can be computationally-dense, we demonstrate that they also execute a significant number of redundant operations. In contrast to the body of orthogonal work on improving GPUs in the presence of irregular, cache- and scheduling-sensitive workloads [15–17, 37, 38], DARSIE is designed to target common contemporary workloads which prior work has demonstrated are insensitive to locality-optimizing techniques that focus on scheduling [37].

8 Conclusion

In this paper we detail the root cause of massively multi-threaded redundancy at the programming language level, and quantitatively explore how much redundancy exists at the grid, threadblock and warp levels. We show that a significant portion of redundancy in GPU applications is TB-wide. Moreover, we observe that much of the seemingly unstructured redundancy that occurs at runtime can be non-speculatively identified based on TB sizing information known at kernel launch time. Using a novel compiler pass for conditional redundancy and an aggressive instruction-skipping microarchitecture that skips instructions in fetch, our proposed DARSIE design both increases performance and decreases energy consumption by 30% and 25% respectively.

DARSIE is a vertical solution that delegates each aspect of complexity to the appropriate system level. Static compilation techniques are first leveraged to propagate our newly observed *conditionally redundant* registers, then simple TB sizing information available at kernel launch time finalizes the static compiler's incomplete picture. Our light-weight hardware modifications then provides what the compilation system cannot: the illusion of TB-wide lockstep execution, efficient access to warp-private registers and the ability to skip instructions from multiple warps before they are fetched.

Acknowledgments

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Tor M. Aamodt. 2012. GPGPU-Sim 3.x Manual. http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual. (accessed March 30, 2017).

[2] AMD. 2016. AMD Graphics Cores Next (GCN) Architecture. [Online]. Available: https://www.amd.com/documents/gcn_architecture_whitepaper.pdf. (accessed April 5, 2019).

[3] Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael Abu-Ghazaleh. 2019. CORF: Coalescing Operand Register File for GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 701–714.

[4] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA Workloads Using A Detailed GPU Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174.

[5] J. Adam Butts and Guri Sohi. 2002. Dynamic Dead-Instruction Detection and Elimination. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 199–210.

[6] Cy Chan, Didem Unat, Michael Lijewski, Weiqun Zhang, John Bell, and John Shalf. 2013. Software Design Space Exploration for Exascale Combustion Co-design. In *International Supercomputing Conference (ICS)*. 196–212.

[7] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding Irregular GPGPU Graph Applications. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 185–195.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 44–54.

[9] Zhongliang Chen and David Kaeli. 2016. Balancing Scalar and Vector Execution on GPU Architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 973–982.

[10] Zhongliang Chen, David Kaeli, and Norman Rubin. 2013. Characterizing Scalar Opportunities in GPGPU Applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 225–234.

[11] Sylvain Collange, David Defour, and Yao Zhang. 2009. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. In *European Conference on Parallel Processing(Euro-Par)*. 46–55.

[12] 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)*. 63–74.

[13] W. W. L. Fung and T. M. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 25–36.

[14] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[15] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. 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)*. 395–406.

[16] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 332–343.

[17] Onur Kayundefinedran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither More nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 157–166.

[18] A. Kerr, G. Diamos, and S. Yalamanchili. 2012. Dynamic Compilation of Data-parallel Kernels for Vector Processors. In *International Symposium on Code Generation and Optimization (CGO)*. 23–32.

[19] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 130–141.

[20] Keunsoo Kim and Won Woo Ro. 2018. WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 389 – 402.

[21] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. 2009. Lonestar: A Suite of Parallel Irregular Programs. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 65–76.

[22] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-Compression: Enabling Power Efficient GPUs through Register Compression. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 502–514.

[23] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanovic. 2013. Convergence and Scalarization for Data-parallel Architectures. In *International Symposium on Code Generation and Optimization (CGO)*. 1–11.

[24] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 487–498.

[25] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the Value Locality of Store Instructions. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 182–191.

[26] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 138–147.

[27] S. Liu, J.E. Lindholm, M.Y. Siu, B.W. Coon, and S.F. Oberman. 2010. Operand Collector Architecture. https://www.google.com/patents/US7834881 US Patent 7,834,881.

[28] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim. 2017. G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 601–612.

[29] Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T. Chong. 2010. Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 337–348.

[30] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 127–139.

[31] NIRVANA. 2016. Maxas SASS Assembler. https://github.com/NervanaSystems/maxas. (accessed Aug 1, 2018).

[32] NVIDIA. 2015. CUDA. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/. (accessed March 5, 2017).

[33] NVIDIA. 2016. NVIDIA CUDA SDK 4.2. [Online]. Available: https://developer.nvidia.com/cuda-downloads. (accessed March 30, 2017).

[34] NVIDIA. 2018. NVIDIA CUDA SDK 10.0. [Online]. Available: https://developer.nvidia.com/cuda-downloads. (accessed April 4, 2019).

[35] PolyBench:. 2016. The Polyhedral Benchmark Suite. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet/software/polybench. (accessed March 30, 2017).

[36] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A Variable Warp Size Architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 489–501.

[37] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 72–83.

[38] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-Aware Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 99–110.

[39] James E. Smith. 1984. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (Nov. 1984), 289–308. https://doi.org/10.1145/357401.357403

[40] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic Instruction Reuse. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 194–205.

[41] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[42] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[43] Vasily Volkov. 2010. Better Performance at Lower Occupancy. In *Proceedings of the GPU technology conference, GTC*, Vol. 10. 16.

[44] J. Wang and S. Yalamanchili. 2014. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 51–60.

[45] Kai Wang and Calvin Lin. 2017. Decoupled Affine Computation for SIMT GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 295–306.

[46] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 47–61.

[47] S. J. E. Wilton and N. P. Jouppi. 1996. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits* 31, 5 (May 1996), 677–688.

[48] Daniel Wong, Nam Sung Kim, and Murali Annavaram. 2016. Approximating Warps with Intra-warp Operand Value Similarity. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 176–187.

[49] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, , Dong Qunfeng, and Huiyang Zhou. 2014. A Case for a Flexible Scalar Unit in SIMT Architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 93–102.

[50] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. 2013. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *Proceedings of the International Conference on Supercomputing (ICS)*. 433–442.

[51] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 86–97.

[52] Ayse Yilmazer, Zhongliang Chen, and David Kaeli. 2014. Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 103–112.