Zeroploit: Exploiting Zero Valued Operands in Interactive Gaming Applications

RAM RANGAN, NVIDIA, India

MARK W. STEPHENSON, NVIDIA, USA

ADITYA UKARANDE, NVIDIA, India

SHYAM MURTHY*, University of Wisconsin, USA

VIRAT AGARWAL*, Xilinx, India

MARC BLACKSTEIN, NVIDIA, USA

In this article, we first characterize register operand value locality in shader programs of modern gaming applications and observe that there is a high likelihood of one of the register operands of several multiply, logical-and, and similar operations being zero, dynamically. We provide intuition, examples, and a quantitative characterization for how zeros originate dynamically in these programs. Next, we show that this dynamic behavior can be gainfully exploited with a profile-guided code optimization called *Zeroploit* that transforms targeted code regions into a zero-(value)-specialized fast path and a default slow path. The fast path benefits from zero-specialization in two ways, namely: (a) the backward slice of the *other* operand of a given multiply or logical-and can be skipped dynamically, provided the only use of that other operand is in the given instruction, and (b) the forward slice of instructions originating at the given instruction can be zero-specialized, potentially triggering further backward slice specializations from operations of that forward slice as well. Such specialization helps the fast path avoid redundant dynamic computations as well as memory fetches, while the fast-slow versioning transform helps preserve functional correctness. With an offline value profiler and manually optimized shader programs, we demonstrate that *Zeroploit* is able to achieve an average speedup of 35.8% for targeted shader programs, amounting to an average frame-rate speedup of 2.8% across a collection of modern gaming applications on an NVIDIA® GeForce RTX™ 2080 GPU.

CCS Concepts: • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Profile guided optimization, value specialization, GPUs, shader programs, gaming applications

ACM Reference format:

Ram Rangan, Mark W. Stephenson, Aditya Ukarande, Shyam Murthy, Virat Agarwal, and Marc Blackstein. 2020. *Zeroploit: Exploiting Zero Valued Operands in Interactive Gaming Applications*. *ACM Trans. Archit. Code Optim.* 17, 3, Article 17 (August 2020), 26 pages.
<https://doi.org/10.1145/3394284>

*Contributed to this project while working at NVIDIA, Bangalore, India.

Authors' addresses: R. Rangan and A. Ukarande, NVIDIA Graphics India Pvt. Ltd., C1, Jacaranda, Manyata Embassy Business Park, Outer Ring Road, Bengaluru 560045, Karnataka, India; emails: {rrangan, aukarande}@nvidia.com; M. W. Stephenson, NVIDIA Corporation, 11001 Lakeline Blvd #100, Austin, TX 78717, United States; email: mstephenson@nvidia.com; S. Murthy, University of Wisconsin, Computer Sciences Department, 1210 W Dayton St. Madison, WI 53706, United States; email: smurthy4@wisc.edu; V. Agarwal, Xilinx India, Octave Unit 2A and 2B, Parcel 4, 11th to 16th Floors, Salarpuria Sattva Knowledge City, Survey no 83, 1,HITEC City, Hyderabad 500032, Telangana, India; email: virata@xilinx.com; M. Blackstein, NVIDIA Corporation, 9030 NE Walker Rd, Hillsboro, OR 97006, United States; email: mblackstein@nvidia.com. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1544-3566/2020/08-ART17

<https://doi.org/10.1145/3394284>

ACM Transactions on Architecture and Code Optimization, Vol. 17, No. 3, Article 17. Publication date: August 2020.

1 INTRODUCTION

High-performance gaming processors or graphics processing units (GPUs) are at the heart of the innovation and growth in the gaming industry. The industry has become accustomed to big jumps in GPU performance, generation over generation, for several years now. This “guaranteed” increase in hardware performance has allowed game developers and artists to innovate and use ever more sophisticated algorithms and visual effects to achieve near photorealistic rendering in real time. However, as reality catches up with Moore’s law and GPUs hit the limits of 2D CMOS scaling, GPU performance growth from brute-force hardware scaling will eventually come to an end and will increasingly depend on architectural innovations and software-centric solutions to sustain and support innovations in real-time rendering.

To that end, inspired by prior work on value-dependent code specialization in general purpose programs [7, 32, 49], we explore similar value-specialization opportunities for modern gaming applications. In particular, we focus on exploiting *dynamically* zero-valued register operands in multiplication and similar operations, where the knowledge that one of the source operands is zero can be used to avoid evaluating the backward slice¹ of the *other* source operand. Adapting Calder et al.’s value profiler [6] to data-parallel shader programs of gaming applications, we collect tiny histograms of the most popular dynamic values for every destination operand of top shader programs and find that one of the two register operands of several *key* multiply and similar operations tends to be heavily biased toward being zero at runtime.

The knowledge that one of the input operands of a multiply operation is mostly zero dynamically can be used to rearrange the code to avoid computing the *other* source operand whenever the former is zero, since anything multiplied by a zero results in a zero.² This insight, that the backward slice of the *other* operand can be specialized away coupled with the gaming industry practice of eschewing IEEE-compliance for extra performance (see footnote 2), is at the core of *Zeroploit*. Grant et al. studied this optimization, among many others, in the DyC selective dynamic compilation system [32], but found it to have limited applicability in their application suite.

We revisit the above idea as a profile-guided value specialization optimization in this article. We study it in depth on modern gaming applications and extend it in two novel ways. First, we observe that as a root zero operand is propagated to dependent multiply operations in the former’s forward slice (i.e., the set of instructions directly or indirectly dependent on the root operand), the aforementioned backward slice elimination can be applied to the *other* operands of all those multiply operations as well. Figure 1 illustrates this. We find such recursive backward slice elimination to be very profitable in graphics shader programs. Second, we show that this idea can be generalized and applied to operations other than multiply operations by exploiting dynamic values or value ranges, specific to the semantics of those operations. Here are a few examples:

- Knowledge that either operand of logical-and operations or the numerator of division operations is mostly zero dynamically can be used to avoid computing the other source operand.
- In a conditional move (i.e., C-style ternary operator) [19], the backward slice of the selection predicate operand can be avoided if both its data source operands dynamically compute to the same value.
- The backward slice of the address operand of an atomic increment operation can be marked dead and possibly eliminated, along with the atomic operation itself, when the increment

¹Backward slice of an operand comprises all operations involved in computing that operand.

²Though this is not safe as per IEEE 754 rules [10], since the *other* source operand could evaluate to a floating point NaN, $-\infty$ or $+\infty$, game developers typically allow for such IEEE-unsafe floating point optimizations by explicitly setting the *refactoringAllowed*[20] global flag and dropping the *precise* storage class specifier for variable declarations [24].

Fig. 1. Example program dependency graph illustrating *Zeroploit*'s impact on forward and backward slices.

Fig. 2. Profile-guided optimization (PGO) workflows for deploying *Zeroploit*.

operand is dynamically zero. This is applicable only in systems where memory consistency semantics are not piggybacked on atomic operations.³

In this article, we primarily focus on exploiting dynamic value locality, specifically dynamic zeros, in the aforementioned types of operations. We find that most gaming applications have zero-dependent dynamic work-avoidance opportunities in good measure and such opportunities can be harvested with *Zeroploit* in a profile-guided optimization (PGO) framework. This framework can be implemented with any of the workflows illustrated in Figure 2. Profiling can be done either offline, in a dedicated profiling game run, or online, during actual gameplay (in the background), and the PGO transform itself can be implemented either offline (e.g., manually by game developers in high-level source code) or online in a GPU driver's just-in-time compiler. Our proof-of-concept prototype for *Zeroploit* uses a fully offline approach and mimics the deployment workflow traced by solid lines in Figure 2. Our article's key contributions are:

- (1) Characterization of dynamic value locality in register operands and the origins of dynamic zero-valued operands in shader programs of modern gaming applications.
- (2) Generalizing a prior idea on exploiting dynamic zeros in multiply operations to other operation types.
- (3) A PGO transform called *Zeroploit* to specialize forward and backward slices of such operations, by versioning code into a specialized fast path that takes advantage of zero-valued operands and a default slow path that ensures functional correctness when those operands are not zero dynamically.
 - (a) Outline of compiler analysis and transform for *Zeroploit* to argue that this transform lends itself to automation.

³For example, the Direct3D specification requires only API call-level memory ordering [12], allows for unordered memory accesses within a call [13], does not imply memory fences at atomic operations, and requires explicit memory fences [22] to guarantee visibility across threads within a call. Modern GPUs, such as those from NVIDIA®, handle Direct3D and other graphics APIs efficiently by decoupling memory ordering semantics from atomic functionality and having separate instructions for both, enabling *Zeroploit* to be applied to DirectX Byte Code's *atomic_iadd* [14] on such GPUs.

- (b) Evaluation of *Zeroploit* by manually modifying several shader programs across our application suite. We see an average speedup of 35.8% for targeted shaders on an NVIDIA® GeForce RTX™ 2080 GPU, which translates to an average frame-rate speedup of 2.8%.

Additionally, as a minor, secondary contribution, we present a value-profile driven PGO to avoid certain silent stores in graphics shader programs. Silent stores are store operations whose data values are the same as the values in their corresponding memory locations [43]. Our silent store avoidance technique leverages Application Programming Interface (API)-level guarantees to dynamically detect and avoid a subset of silent stores, specifically those that attempt to write the same value as the one that was used to clear or initialize the corresponding data structure. A simple code transform conditionally avoids such stores dynamically by comparing such stores' data values to their target surface's initialization value. We evaluate this technique on one shader program in one of our applications and show a 109% speedup on the targeted shader, which amounts to a 1.2% full-frame speedup.

The rest of the article is organized as follows. Section 2 provides necessary background information on graphics programming and performance considerations. Section 3 describes our adaptation of Calder et al.'s value profiler [6] for graphics shader programs, characterizes general value locality in gaming applications, and provides qualitative and quantitative insights into the origin of zeros in graphics shader programs. Section 4 discusses how to leverage zero-valued operands in practice with a simple code versioning transform called *Zeroploit* and also outlines the necessary analysis and transform to implement this in a compiler. We evaluate *Zeroploit* on a set of modern gaming applications in Section 5. In Section 6, we describe and evaluate our silent store PGO. Related work is discussed in Sections 7, and Section 8 summarizes key takeaways from this work.

2 BACKGROUND

The purpose of this section is to give a very high-level introduction to graphics programs and their performance considerations for readers who are unfamiliar with the topic. Modern gaming applications are most popularly developed in Direct3D 11 [15], Direct3D 12 [16], OpenGL [37], and Vulkan [38] APIs. Without exception, all graphics APIs use a two-level hierarchy to convey work from the Central Processing Unit (CPU) to the GPU: an API layer and a shader program layer. The API layer is used to setup GPU state (e.g., enable/disable depth testing, enable/disable blending, etc.), bind resources such as constant buffers, textures (read-only input data), render targets (write-only output data), or general multi-dimensional arrays called unordered access views (UAVs, which are read-write), and bind shader programs for use in GPU work calls (which could be graphics draw calls or compute dispatch calls). A GPU work call typically causes one or more threads of a corresponding bound shader program to be run. Shader programs are typically written in high-level languages such as the high-level shading language (HLSL) used with Direct3D APIs [18], or the OpenGL Shading Language (GLSL) used with OpenGL or Vulkan APIs [41]. They are packaged along with an application by compiling them offline to a well-defined high-level assembly such as DXBC [21] or DXIL [25] for DirectX, SPIRV for Vulkan, and so on.

A typical API sequence interleaves various state setup commands with GPU work calls. Several such API commands can be in flight at any given point in the GPU. State setup commands take very little time and most of the GPU time is spent in GPU work calls. A single frame of a modern game typically requires hundreds or even thousands of state setup and GPU work calls. A frame starts with the user's eye position and velocity as inputs. Through successive single calls or groups of calls, it implements various graphics algorithms, some for basic functionality (such as discarding

occluded objects) and others to embellish the basic image to make it more photorealistic with various post-processing effects (e.g., motion blur, depth of field, Bokeh filter, etc.).

The hundreds or thousands of API calls that make up a frame can exhibit vastly different performance characteristics. An API call can be limited by the performance of fixed function units, CPU-GPU data transfer latencies, or the performance of programmable shaders, whose performance can in turn be limited by memory bandwidth, instruction issue rate, or latency of memory loads. The analysis and optimization efforts in this article focus exclusively on the portion of a gaming application's frame time that is dominated by programmable shaders, namely, pixel and compute shaders, which are the two most expensive shader types.

Next, we describe the shader program execution model on NVIDIA® GPUs. Shader programs execute on programmable processing cores called streaming multiprocessors (SMs). When handling shader programs, NVIDIA® GPUs take advantage of inherent data parallelism in graphics workloads and launch several parallel threads of shader programs bound to a given GPU work call. Threads of shader programs are grouped into convenient physical entities called *warps*. A warp can span *at most* 32 threads. Instruction fetch, decode, and scheduling in the SM happens at a warp granularity, while operand fetch, execution in functional units, and register writeback happen at a per-thread granularity. SM execution is most efficient when handling full warps, since all its functional units are fully utilized. The SM implements a single-instruction, multiple threads (SIMT) execution model [28] whereby, each thread of a warp maintains its own program counter (PC). If threads of a warp diverge, i.e., execute at different PCs, then SM datapath utilization suffers. To mitigate this, the SM uses joint hardware-software mechanisms to make diverged threads re-converge at well-defined program points. We will build on the above background in the remaining sections of the article.

3 VALUE LOCALITY IN GRAPHICS SHADERS

In this section, we will first describe how we adapt Calder et al.'s value profiler [6] for graphics shader programs and then characterize register operand value locality in gaming applications.

3.1 Value Profiler

Calder et al. [6] used a top-N value (TNV) table to record up to N <value, count> pairs for each profiled register operand. When an unseen value shows up during profiling and a free entry is available in the TNV table, it gets added directly. If a free entry is not available, then the new value replaces the least-frequently used (LFU) entry. Additionally, to avoid two unseen, but potentially high frequency values kicking each other out repeatedly due to LFU replacement, they also periodically clear the bottom $N/2$ entries (reverse-sorted by their frequencies). Our value profiler is based on their approach, with a few adaptations.

We associate a 128-byte array with all destination register writes in all pixel and compute shaders and collect value profiles on real silicon. Of the 128 bytes, 120 bytes are dedicated to a 15-entry TNV table (4 bytes for value and 4 bytes for frequency count per TNV entry), 4 bytes for an execution count field, and the remaining 4 bytes serve as padding. Two aspects of our profiler worth mentioning are the following:

LFU with probabilistic replacement: Instead of Calder et al.'s approach of clearing half the table periodically, our implementation uses LFU with probabilistic replacement, whereby a probability function decides whether a new incoming value enters the TNV table or not. The probability function is implemented as a simple logical exclusive-or of the value seeking to enter the table and the prevailing value of the GPU clock counter, a random variable for our purposes here, and picking a few lower order bits to control the replacement probability. This was much easier to implement than periodically clearing half the entries of all TNV tables. The above amounts to

```

void TNVRecordValue(unsigned pc, unsigned newVal) {
    // increment execution count at pc
    atomicIncrement(profileData[pc].execCount);
    // linear scan to find matching histogram entry
    if (FindValAndIncrement(pc, newVal) == SUCCESS)
        return;

    // newVal not found, check for free entry
    if (FindFreeEntryAndInstall(pc, newVal) == SUCCESS)
        return;

    // free entry not found - attempt probabilistic
    // replacement
    // first, check probability
    // getClock(): wrapper for PTX's .sreg .u32 %clock
    randomVar = getClock() ^ newVal ^ pc;
    // extract lower 8 bits
    randomVar = randomVar & 255;
    // model 50% probability
    if (randomVar < 127) {
        FindLFUEntryAndInstall(pc, newVal);
    } else {
        // ignore newVal
    }
    return;
}

void installEntry(unsigned index,
                 unsigned pc, unsigned newVal) {
    profileData[pc].histogram[index].count = 1;
    profileData[pc].histogram[index].value = newVal;
}

int FindValAndIncrement(unsigned pc, unsigned newVal) {
    for (i = 0; i < 15; i += 1) {
        if (profileData[pc].histogram[i].value == newVal) {
            atomicIncrement(profileData[pc].histogram[i].count);
            return SUCCESS;
        }
    }
    return FAILURE;
}

int FindFreeEntryAndInstall(unsigned pc, unsigned newVal) {
    for (i = 0; i < 15; i += 1) {
        if (profileData[pc].histogram[i].count == 0) {
            installEntry(i, pc, newVal);
            return SUCCESS;
        }
    }
    return FAILURE;
}

void FindLFUEntryAndInstall(pc, newVal) {
    lfuCount = profileData[pc].histogram[0].count;
    lfuIndex = 0;
    for (i = 1; i < 15; i += 1) {
        if (profileData[pc].histogram[i].count < lfuCount) {
            lfuCount = profileData[pc].histogram[i].count;
            lfuIndex = i;
        }
    }
    installEntry(lfuIndex, pc, newVal);
}

```

Fig. 3. Pseudo-code for updating the TNV table.

random sampling of the entire population of an operand's dynamic values and statistically, the top N values from random sampling are expected to asymptotically approach the top N values from a perfect histogram of all dynamic values seen.

Lock-free design: Our profiler is able to capture the top N values and their likelihoods with good enough accuracy without using any critical section locks. Pseudo-code for recording a dynamic value in the TNV table is given in Figure 3. To update an entry, the profiler performs a linear scan through the TNV table to locate the relevant entry and atomically increments its count field. These atomic increments are not expensive as modern GPUs provide first-class support for atomic integer operations. To add new values to the table, the profiler performs a linear scan to see if a value exists (and fails that check), determines an index to install the value at (either free entry or an entry picked via LFU plus random replacement), sets the value field to the new value, and sets the count field to 1. Our implementation does not hold an explicit valid bit per entry of the TNV table and instead interprets a zero-valued count field to mean that the corresponding entry is free. We write the count field first (to reserve the entry and remove it from the free pool) with a regular store followed by another regular store to the value field. While the above non-atomic value installation is in progress, another thread that wants to install the same value for the same operand can fail its linear scan (since the value field has not yet been updated) and proceed to claim an entirely new entry (since the count field for the previous entry was set to 1 already) to install the same value. Once both the value and count fields are correctly updated, all future updates will hit the lower index entry in their linear scans and increment its count, and any redundant entry at a higher index will eventually get LFU-replaced.

We implement the above value profiler at the DXBC level via static instrumentation and then replace the original DXBC shader programs with their instrumented counterparts. Since each DXBC instruction can write up to four destination register components (x/y/z/w), our infrastructure associates a TNV table for each destination register component and statically adds instrumentation

Fig. 4. Value profiling infrastructure.

App.	Before	After	Drop%
3DMFire	31.01	29.55	4.95%
3DMTime	37.84	34.59	9.40%
ACOD	24.10	19.24	25.22%
Apex	25.06	20.75	20.75%
BF5	22.27	18.22	22.20%
Division2	74.05	59.57	24.30%
DXMD	96.59	77.88	24.03%
FFXV	64.47	48.98	31.63%
FH4	8.64	7.29	18.41%
Metro	53.77	49.64	8.31%
PUBG	56.97	45.47	25.30%
SB	42.80	40.23	6.38%
SOTR	21.86	18.46	18.40%
Average			18.41%

Table 1. Comparison of DXBC Scalar Operation Counts (in Billion) Before and After Post-processing

calls after every DXBC instruction to profile all its destination operands. The infrastructure also modifies the API sequence to create and bind buffers to collect value profiling information, launch GPU work calls with instrumented shader programs instead of the original ones, and copy the buffers from the GPU back to the CPU when a frame has completed. A post-processing script then back-annotates the value profiles to instructions of respective shaders.

Instead of profiling low-level assembly code [61], we choose to profile values at the DXBC level, because our ultimate goal is to perform manual or semi-manual code transforms to optimize shader programs based on value profiles and it is much easier to do such transforms in DXBC than it is in low-level assembly, thanks to DXBC's concise four-component vector instructions simplifying manual code analysis and transform and the ease of working with pre-register-allocated code. Profiling at the DXBC level, while convenient and accurate for performing transforms, has one drawback as far as value locality characterization goes, namely, it does not get the benefit of classical optimizations like constant folding, copy propagation, and dead code elimination, which are performed just-in-time by a GPU driver when transforming DXBC to low-level assembly. From a value locality characterization point of view, this can cause unfair instruction bloat and can make dynamic register operand value locality appear higher than it actually will be in low-level assembly code, due to the flow of compile-time constants through shader programs' dataflow graphs.

To address this drawback, a second post-processing tool takes the annotated shader programs as input and performs a rudimentary optimizer's job to mark constant-foldable and copy-propagatable destination register operands. DXBC instructions thus marked are discarded for the below value locality analysis to get as close as possible to the quality of low-level assembly code. Likewise, the resulting value locality metrics ignore compile-time discoverable constants and literal immediates and thus represent true dynamic register operand value locality. Figure 4 illustrates the various steps of our value profiling infrastructure. Table 1 provides the calculated DXBC scalar operation counts before and after the above cleanup optimizations. On an average, we see that our best-effort post-processing cleanup is able to bring down instruction count by 18.41%.

Fig. 5. General operand value locality in gaming applications.

3.2 Register Operand Value Locality

Using the above setup, we characterize general operand value locality⁴ in gaming applications (refer to Section 5 for details on applications chosen) in two different ways.

First, Figure 5(a) gives the weighted average value locality for an instruction across all applications. This graph shows the percentage of dynamic values captured by just the top entry of our 15-entry histogram (labeled VL-1), by the top two entries (labeled VL-2), and by all 15 entries combined (labeled VL-15). It is worth noting that average VL-1 stands at 18.7%, average VL-2 stands at 21.3%, while average VL-15 stands at 25.1%, implying that most instructions showing value locality tend to have one highly prominent value. Next, we plot the cumulative distribution function (CDF) of the normalized dynamic counts of instructions against total value locality (capturable by our 15-entry TNV tables) in Figure 5(b) for all applications. We can see that, on an average (black line with round marker), about 47% of the instructions do not have any value locality. Another item worthy of note is that, as seen from the right end of the CDFs, on an average, about 16% or so of dynamic instructions have value locality upwards of 90%, with some apps having a significant percentage of their dynamic instructions showing 100% value locality. Upon inspecting the 99% to 100% histogram bin, we find that roughly 75% of the instructions with VL-15 above 99% are of one of the following types: memory loads (texture loads), FP arithmetic operations, FP and integer comparisons. Since FP and integer comparison operations have only two possible outcomes, it stands to reason that their VL-15 numbers are at 100%. When analyzing texture loads, we find cases that are fully uniform (thus, even VL-1 would capture those) and those that return a small set of values such that all of them get captured in the 15-entry histogram used by VL-15. FP arithmetic operations with VL-15 higher than 99% typically consume results of prior texture loads that also exhibit similarly high value locality and thus transitively end up getting high VL-15 numbers.

Table 2 gives a breakdown of VL-1 value locality by instruction type in the form $VL(exec)$, where VL is the value locality (VL-1) percentage of a given instruction type and $exec$ is the execution frequency percentage of that instruction type. VL is expressed as a percentage normalized to the total execution count of operations in a given category and $exec$ is the execution frequency of operations in a category as a percentage normalized to the total dynamic instruction count of respective applications. The instruction type categories are:

⁴Lipasti et al. first defined value locality in their 1996 article [46] as “the likelihood of a previously-seen value recurring repeatedly within a storage location,” where storage locations could be either memory locations or registers.

Table 2. Top Value Locality (VL-1) and Execution Frequency by Instruction Type

App.	texture	fpsat	fpmmm	sfu	fparith	fpcmp	intcmp	intlogic	intarith	convert	intmm	cmov
3DMFire	41.7(11.2)	65.7(0.8)	27.9(5.5)	10.7(8.1)	9.2(68.0)	85.2(1.0)	86.1(0.4)	40.4(0.5)	7.3(0.8)	8.6(1.3)	0.0(0.0)	30.7(1.5)
3DMTime	18.9(9.2)	43.9(0.4)	21.2(5.1)	14.5(8.6)	13.4(65.6)	82.8(1.1)	90.6(1.2)	41.2(0.8)	7.6(2.4)	1.9(0.9)	16.7(0.1)	33.5(3.2)
ACOD	36.1(12.0)	45.6(2.8)	25.2(6.1)	22.3(3.3)	18.8(59.9)	88.6(0.6)	89.6(1.1)	26.1(2.6)	11.6(3.1)	24.2(2.5)	0.5(0.6)	30.2(3.5)
Apex	47.1(12.5)	26.8(4.2)	18.7(5.9)	6.6(2.9)	10.5(59.4)	84.3(0.1)	78.4(0.8)	48.1(2.8)	18.2(1.8)	8.5(1.6)	0.4(0.8)	19.4(2.9)
BF5	36.7(18.6)	32.9(2.2)	17.6(4.8)	9.8(2.6)	13.6(57.3)	83.8(0.5)	83.0(1.0)	20.5(2.0)	14.7(5.1)	11.0(2.3)	2.6(0.2)	23.3(1.1)
DXMD	38.2(13.6)	27.7(2.9)	6.4(3.8)	14.6(4.9)	14.2(59.4)	92.6(0.1)	80.0(1.1)	19.0(2.5)	8.7(6.0)	1.8(1.2)	1.4(0.8)	26.7(2.3)
Division2	24.7(10.4)	37.5(2.0)	11.5(3.5)	7.2(2.9)	10.4(63.6)	95.0(1.4)	88.4(0.8)	31.1(2.3)	12.9(3.3)	13.6(3.1)	7.1(0.9)	10.2(4.6)
FFXV	42.9(12.9)	22.3(2.0)	17.5(5.3)	11.8(3.5)	10.5(65.2)	98.9(0.3)	90.7(0.5)	40.2(2.0)	18.9(1.2)	12.0(2.0)	33.5(0.1)	22.8(4.2)
FH4	25.2(10.0)	24.5(1.6)	20.4(3.9)	2.0(6.8)	7.6(64.0)	91.1(0.9)	81.2(0.7)	21.6(2.0)	11.9(2.9)	4.0(3.9)	100(0.2)	12.9(1.5)
Metro	17.5(12.2)	34.8(2.7)	18.2(4.3)	10.2(4.4)	6.5(69.5)	95.1(0.2)	89.6(0.6)	27.5(1.1)	9.5(1.7)	3.6(0.9)	3.5(0.2)	20.2(1.3)
PUBG	56.6(15.4)	39.8(4.0)	18.9(4.2)	16.8(4.0)	26.4(66.7)	78.5(0.3)	95.9(0.3)	49.7(0.9)	35.3(0.9)	19.2(0.6)	0.0(0.0)	35.4(2.5)
SB	29.5(11.6)	26.0(3.3)	8.2(3.5)	4.9(3.9)	10.4(68.3)	85.8(0.6)	88.7(2.6)	37.3(0.4)	9.5(2.9)	6.1(2.1)	0.0(0.0)	60.9(0.3)
SOTR	37.8(12.2)	41.3(4.4)	17.1(2.9)	4.2(4.1)	8.6(59.7)	75.0(0.5)	84.9(1.4)	30.7(3.6)	8.4(3.5)	14.3(2.5)	2.9(1.0)	12.6(2.6)
Average	34.8(12.4)	36.1(2.6)	17.6(4.5)	10.4(4.6)	12.3(63.6)	87.4(0.6)	86.7(1.0)	33.3(1.8)	13.4(2.7)	9.9(1.9)	13.0(0.4)	26.1(2.4)

- *texture*: All memory access operations, including texture and shared memory access operations, are included in this category.
- *fpsat*: Saturating floating point (FP) operations that cap floating point results less than 0.0 to 0.0 and results greater than 1.0 to 1.0. This category also includes floating point rounding operations.
- *fpmmm* and *intmm*: These categories stand for FP and integer min/max operations, respectively.
- *sfu*: SFU stands for special functional unit. Operations covered by this category include division, reciprocal, square root, and transcendental operations like sine, cosine, exponentiation, logarithm, and so on.
- *fparith*: FP arithmetic operations like multiply, add, dot product, and derivative computing instructions are included in this category.
- *fpcmp* and *intcmp*: FP and integer comparison operation categories, respectively.
- *intlogic*: This include all logical and shift operations.
- *intarith*: Integer arithmetic operations.
- *convert*: All conversion operations are grouped in this category, which includes FP to integer, integer to FP, 16-bit FP to 32-bit FP, and 32-bit FP to 16-bit FP operations.
- *cmov*: Conditional move operations [19], which implement C-style ternary operators, are included in this category.

The categories we have picked to characterize value locality are tailored to gaming applications as it helps highlight the relative importance of categories like *sfu*, *fpsat*, *fpmmm*, and *intmm* that are unique to graphics and signal processing applications.

Table 2 is comparable to a similar characterization for general-purpose programs in Reference [6]. One noticeable difference is that the memory loads in graphics applications represented by the *texture* category have an average VL-1 value of 34.8%, whereas loads across FP benchmarks in Reference [6] show an average VL-1 of only 22% (calculated from their table by weighting integer and FP loads' value locality numbers by their execution frequencies and dividing by the sum of their execution frequencies). There are a few other differences as well, but given that we are

Fig. 6. Zero-value locality in gaming applications.

using modern gaming applications from 2019 and their work used SPEC CPU 1992 benchmarks, differences are inevitable.

Interestingly, one strong similarity between prior work on value locality characterization [7] and ours is the undisputed popularity of the value, zero, across all applications (we count FP representations for positive and negative zero as positive zero). The floating point number 1.0 is the second most popular value. These two together account for 61% of the VL-2 metric.

Next, we narrow our focus to just zero values. We define zero-value locality of an operand as the number of times zero showed up in either of the top two entries of its 15-entry histogram, normalized to the count of total writes to the same operand. Figure 6(a) shows the average zero-value locality, weighted over all instructions of an application, across all applications. From this graph, we can see that, on an average, any given instruction has a 11.22% probability of producing zeros and this covers almost half the total average value locality of 25.1% captured by the 15-entry histograms. Figure 6(b) presents CDFs for zero-value locality. Like in Figure 5(b), the y-axis here reports instruction counts normalized to the total dynamic instruction counts of the respective applications, so that it is easy to see what fraction of the total instructions produce zeros. Zero-producing instructions can be:

- zero-propagators—instructions that produce zeros due to one of their source operands being zero, e.g., multiply operations.
- zero-originators—instructions that generate zeros from non-zero source operands as a result of their semantics, e.g., saturating arithmetic operations. This category can be further classified as follows:
 - useless zero-originators—zero-originators that do not have any zero-propagating operations as their consumers. For example, a zero originator whose destination register is consumed only by add operations is a useless zero-originator.
 - useful zero-originators—zero-originators that have at least one zero-propagating consumer. For example, a zero originator that feeds a multiply operation is a useful zero-originator. For our accounting purposes, even if one consumer (out of many) of a zero-originator is a zero-propagator, the former is labeled a useful zero-originator.

Zero-propagators and useful zero-originators are what we leverage for *Zeroploit*. The name “zero-propagator” actually does not do justice to the immense optimization potential such operations could unlock, because, in addition to propagating zeros forward (down the forward slice

Table 3. Zero-value Locality and Execution Frequency for All Categories of Zero-producers

App.	texture	fpsat	fpm	sfu	fparith	fpcmp	intemp	intlogic	cmov	zeroprop	useless
3DMFire	27.5(1.0)	37.5(0.4)	47.5(0.6)	21.3(0.0)	11.7(0.2)	40.6(0.1)	50.8(0.0)	0.0(0.0)	44.3(0.1)	25.9(4.6)	43.8(14.5)
3DMTime	26.9(0.5)	41.8(0.1)	41.9(0.1)	43.7(0.5)	37.1(1.4)	1.4(0.1)	33.3(0.2)	45.0(0.0)	47.7(2.0)	47.5(3.6)	40.8(17.2)
ACOD	50.3(1.3)	62.9(0.8)	23.9(0.5)	74.0(0.0)	39.5(1.4)	45.7(0.3)	45.5(0.1)	12.3(0.0)	51.5(0.2)	44.1(2.0)	52.3(20.8)
Apex	24.6(0.2)	41.2(1.4)	34.2(0.3)	34.6(0.0)	28.2(1.2)	66.5(0.0)	41.2(0.1)	82.8(0.1)	22.2(0.2)	33.1(1.1)	56.2(15.7)
BF5	60.6(3.5)	31.7(0.3)	49.0(0.2)	3.0(0.1)	32.7(1.1)	47.0(0.2)	26.3(0.1)	33.3(0.0)	77.4(0.1)	33.5(2.1)	48.8(19.3)
DXMD	54.5(1.5)	22.8(0.4)	25.1(0.2)	2.6(0.8)	32.0(1.6)	8.1(0.0)	11.1(0.1)	2.6(0.0)	72.9(0.4)	54.9(3.1)	46.1(17.7)
Division2	50.7(0.3)	45.3(0.6)	21.9(0.1)	10.7(0.1)	30.5(1.4)	31.6(0.1)	0.1(0.0)	23.2(0.0)	31.7(0.1)	40.5(2.2)	46.6(13.2)
FFXV	84.7(0.8)	50.7(0.3)	39.9(0.4)	24.5(0.1)	30.3(1.3)	98.6(0.0)	99.4(0.0)	53.7(0.0)	19.7(0.1)	40.6(1.6)	51.1(14.2)
FH4	27.2(1.3)	34.8(0.3)	49.1(0.9)	0.0(0.0)	30.9(0.8)	69.7(0.1)	12.5(0.1)	0.0(0.0)	78.1(0.0)	19.6(1.6)	33.4(12.4)
Metro	6.1(0.0)	42.7(1.0)	47.7(0.6)	18.6(0.1)	21.0(0.9)	6.6(0.1)	28.1(0.2)	7.6(0.0)	2.3(0.0)	42.6(1.7)	44.7(9.8)
PUBG	89.9(3.1)	52.4(1.1)	42.6(0.3)	19.4(0.2)	79.9(3.6)	0.0(0.0)	96.2(0.0)	0.0(0.0)	49.4(0.2)	50.2(1.3)	64.6(21.8)
SB	8.3(0.2)	33.8(0.8)	10.5(0.0)	5.5(0.0)	15.8(0.6)	98.9(0.0)	0.0(0.0)	0.0(0.0)	0.0(0.0)	43.4(1.0)	50.4(19.3)
SOTR	65.1(0.2)	64.4(1.7)	46.0(0.1)	0.1(0.0)	25.1(0.8)	11.9(0.1)	69.0(0.0)	0.0(0.0)	8.4(0.0)	35.9(0.9)	39.8(14.7)
Average	44.3(1.1)	43.2(0.7)	36.9(0.3)	19.8(0.1)	31.9(1.3)	40.5(0.1)	39.5(0.1)	20.0(0.0)	38.9(0.3)	39.4(2.1)	47.6(16.2)

of the zero-originator), such zero-propagators, by virtue of their semantics, allow elimination of the backward slices of their unrelated, non-zero operands as well.

3.3 Where Do Zeros Originate?

Table 3 provides a breakdown of all zero-producers, with useful zero-originators being split into finer-grained categories based on their instruction types, and zero-propagators and useless zero-originators being grouped coarsely into the last two columns, respectively. As before, each entry of the table is of the form $ZL(exec)$, where ZL is the zero-value locality and $exec$ is the execution frequency. ZL is expressed as a percentage normalized to the total execution count of operations in a given category and $exec$ is the execution frequency of operations in a category as a percentage normalized to the total dynamic instruction count of respective applications.

Most of the useful zero originators owe their existence to a strong desire on the part of graphics algorithm/program developers to avoid control flow and prefer simple and efficient straight-line code, often leveraging graphics-specific instruction set support, to avoid efficiency losses due to potential intra-warp thread divergence as well as to provide GPU compilers with large scheduling scopes to extract maximal instruction level parallelism with local (i.e., basic-block level) instruction schedulers. The category labeled *fpsat* comprises saturating arithmetic operations like *_sat*-suffixed versions of *mul*, *mad*, *div*, and *dp2/dp3/dp4* in the shader model 5 instruction set [21] in addition to FP rounding operations. Graphics algorithms are designed to make efficient use of their data parallel workloads and data parallel execution substrates. Therefore, they prefer to avoid explicit control flow to handle exceptional conditions like overflow, wrap-around, and so on, and instead prefer to use saturating arithmetic [52, 58] when necessary to automatically cap the results of floating point operations to well-defined lower and upper limits (0.0 and 1.0, respectively, in shader model 5). The same control-flow avoidance rationale also applies to the use of floating point min/max operations (labeled *fpminmax*). Most GPU vendors, including NVIDIA, offer first-class support for saturating arithmetic and min/max operations.

The *texture* category is a prominent source of useful zero-originators. It captures cases where texture lookups return zeros. Texture lookups that typically return zeros are those that are written to dynamically in one or more prior GPU work calls and consumed in one or more later GPU work

Fig. 7. Partially zero texture.

```

1  lt r12.xyzw, r11.xyzw, cb0[0][2].www
2  and r12.xyzw, r12.xyzw, l(1.0)
3  mad r13.xyzw, -r11.xyzw, r3.zzzz, l(1.0)
4  mul r12.xyzw, r12.xyzw, r13.xyzw

```

Fig. 8. Commonly seen idiom: compare $\rightarrow$ logical-and with 1.0 $\rightarrow$ multiply.

calls. Such textures are referred to as dynamic textures or rendered textures, as opposed to artist-created static textures, which are part of a game's assets and are not changed dynamically. Zeros in a dynamic texture can arise due to one of the following two reasons. In some cases, the destination register of zero-originator or zero-propagator of a prior GPU work call could get written to that surface. Or, the surface could be initialized to zero with an explicit API *clear()* call earlier in the frame and then conditionally written to in later GPU work calls before being consumed as a texture. This is a common idiom in graphics programming and many dynamic textures, both color and depth surfaces, get created and updated in this fashion. When such surfaces are read in a consumer work call, the locations that were not conditionally written to will return the initial value of zero. The shadow accumulation texture from an NVIDIA® Nsight™ [27] capture of one of our applications, BF5, in Figure 7 (© Electronic Arts Inc.) is a good example of a partially zero texture that will return zeros dynamically for texture loads going to the black areas.

Results of comparison operations (categories *intcmp* and *fpcmp*) and logical operations *intlogic* occasionally tend to be used in zero-propagators like multiply operations, thus making them useful zero-originators. Instead of writing a 1.0 or 0.0 to a variable in short if-else blocks, a standard code pattern seen in the output of Microsoft's fxc compiler [17] is to logical-and the result of a comparison, which can be 0xffff_ffff or 0x0, with 1.0, and use the result in future multiply operations. Figure 8 shows an example snippet from a 3DMT shader. Notice the result of the less-than comparison operation *lt* in line 1 is logically-anded with 1.0 in line 2 before being used as a multiplicand in line 4. Here, if *lt* is dynamically zero with a high likelihood, it is tagged as a useful zero originator.

Logical-or operations or FP add operations are commonly used to accumulate the results of multiple texture lookups. If all those texture lookups return zeros, then the logical-or or FP add operations can get counted as useful zero-originators if their results are used in zero-propagators (the original texture lookups' results do not get used in a zero-propagator and thus such texture lookups get counted under useless zero-originators).

Later, in Section 5, we will see how this relatively tiny percentage of useful zero-originators and zero-propagators are exploited by *Zeroploit* to produce good performance gains across our target applications. In the next section, we will describe the *Zeroploit* code transform that will seek to leverage this wealth of zeros to produce optimized codes.

4 THE ZEROPLOIT TRANSFORM

Zeroploit is a profile-guided code transform that may be applied at any level (i.e., high-level language to intermediate representation to target assembly language). The following sub-sections will

<pre> r0 = expensiveWork() // r1 == 0, 50% of the time r1 = expressionAvblEarly() r2 = r0 x r1 </pre> <p>(a) Original.</p>	<pre> r1 = expressionAvblEarly() if (r1 == 0) { // fast path // specialization hint r1 = 0 r0 = expensiveWork() r2 = r0 x r1 } else { // slow path r0 = expensiveWork() r2 = r0 x r1 } </pre> <p>(b) After Zeroploit.</p>	<pre> r1 = expressionAvblEarly() if (r1 == 0) { // fast path r2 = 0 } else { // slow path r0 = expensiveWork() r2 = r0 x r1 } </pre> <p>(c) After Zeroploit + specialization.</p>
--	---	---

Fig. 9. Example illustrating *Zeroploit*. Here, *r1* is the versioning variable and *r0* is the *other* operand, whose backward slice is *expensiveWork()*.

present the building blocks of the transform, heuristics, challenges, and finally outline a compiler algorithm to achieve this transform automatically in practice.

4.1 Key Concepts

It is best illustrated with the canonical example in Figure 9. A typical opportunity for *Zeroploit* will have three main ingredients, namely:

- (1) A program variable that is produced by a useful zero-originator and is consumed by a zero-propagator. This variable (or its equivalent) will become the *versioning variable* in the transformed code, based on whose dynamic value, execution will jump to either a fast path or a slow path.
- (2) The *other* operand of the zero-propagator is very expensive to compute.
- (3) The expression computing the versioning variable is available early enough. At a minimum, this expression must be independent of the *other* operand and in general, going backwards up the program dependency graph, the farther up the backward slices of the versioning variable and the *other* operand intersect (or not intersect at all), the better.

The example code in Figure 9(a), which is assumed to satisfy all three pre-requisites, can be *Zeroploit*-transformed to the code in Figure 9(b) as follows. First, *Zeroploit* identifies a set of operations that will be affected by either forward or backward slice (including recursive backward slice) specialization based on the versioning variable being zero. The region of code from the first affected operation to the last forms the versioning scope. Next, it duplicates operations in the versioning scope. One of the copies or versions, which will become the specialized, fast path, is prefixed with an explicit assignment of zero to the versioning variable to enable the compiler back-end to easily notice the specialization opportunity in that scope and apply classical optimizations such as constant folding, constant propagation, dead code elimination, and so on, to name a few, to instructions in that path. The other version will serve as the default, unspecialized, slow path. Finally, a conditional branch predicated on the versioning variable being equal to zero is added to dynamically steer execution to the fast path or the slow path. Figure 9(c) shows the final optimized code after the fast path has been specialized by a compiler back-end. While the transform shown in Figure 9 is similar in principle to Muth et al.'s approach [49], the key distinction lies in the first step where we determine the versioning scope. Given our focus on exploiting zeros, we make sure to include both forward and backward slices of zero-propagators in the versioning scope, whereas Muth et al. focused on harvesting generic value locality and restricted their approach to include only forward slices of versioning variables.

Note, while the above steps are suitable when versioning based on just one variable, difficult tradeoffs arise when dealing with multiple versioning variables. Handling them will involve evaluating and comparing costs and benefits for various possibilities. Section 4.4 will take cost-benefit tradeoffs into account and describe a more nuanced procedure to perform versioning.

The *Zeroploit* transformation is straightforward enough that it can be done by hand by game developers or low-level optimization experts, or can be automated in a compiler. Given *Zeroploit*'s reliance on classical compiler optimizations like constant propagation, constant folding, dead code elimination, and so on, to achieve specialization benefits, *Zeroploit* is expected to be run as one of the early passes in a compiler, prior to the aforementioned classical optimizations, or where possible, even at a higher-level code. We plan to implement *Zeroploit* as one of the first passes in our low-level back-end compiler's intermediate representation (IR) so that the optimization can be applied in a front-end agnostic manner, to shader programs written in HLSL or GLSL.

4.2 Identifying Versioning Variables

The previous sub-section assumed the existence of a pre-identified versioning variable upon which the rest of the transform was built. In this sub-section, we will see how to go about identifying profitable versioning variables.

Identifying profitable versioning variables is somewhat of a tricky problem. In a given program's value profile, any number of variables may be detected as having a high likelihood of being dynamically zero. That does not make them all good candidates for *Zeroploit*. Since *Zeroploit*'s sole objective is to avoid dynamically redundant work, the choice of the versioning variable must maximize that optimization objective. It is important to set a certain minimum threshold for the amount of work that *Zeroploit* would avoid (weighted in terms of instruction count or latency or estimated savings in texture accesses or other metrics) and filter out candidates that fall short of that threshold. Mathematically, if $zeroProb$ is the probability of a versioning variable being zero, $fastPathCost$ is some cost metric (latency or instruction count) of the fast path, $slowPathCost$ is the cost of the default slow path, then the inequality, $zeroProb \times fastPathCost + (1 - zeroProb) \times slowPathCost < slowPathCost$, must hold in order for the transform to be profitable. But, this is an optimistic model. The question of whether it is profitable to introduce control flow in GPU shader programs, needs more nuanced handling and modeling, as will be seen in Section 4.3.

Since zeros propagate transitively through a chain of multiply or logical-AND operations, the optimizer must trace backwards up through a program's dependency graph to find the root variable that originates zeros and version based on the useful zero-originator, rather than zero-propagators like multiply operations later in the dependency chain to maximize specialization benefits.

At times, it may be beneficial to combine independent versioning conditions into a single versioning variable by logically AND'ing the two versioning conditions and simultaneously specializing for two (or more) independent variables being zero. However, since the probability of two independent events jointly occurring is $p1 * p2$, where $p1$ and $p2$ are the probabilities of the respective events occurring individually, the joint probability of the combined versioning condition will be lower than the individual probabilities. Therefore, the optimizer must carefully tradeoff the benefits of deeper specialization brought about by combining variables and specializing simultaneously for multiple variables being zero, albeit at a reduced taken probability against independently specializing for each of those variables in a shallow manner, but at a higher taken probability. Figure 10 illustrates this tradeoff. In Figure 10(b), we can see that deeper specialization allows specialization of a function call $func()$ at an 81% probability, with the implication being that expensive texture lookup operations ($Tex()$) will execute the remaining 19% of the time, whereas in the shallow-specialized code in Figure 10(a), both texture lookup operations are guaranteed to not execute for 90% of the time. Note that, it is possible to modify the code in Figure 10(a) to specialize

<pre> If (cond1) { // 90% probability R0 = R1 = 0 } else { R0 = Tex() R1 = R0 x 0.5 } If (cond2) { // 90% probability R2 = R3 = 0 } else { R2 = Tex() R3 = R2 x 0.5 } R1 = R1 + R3 func(R1) </pre> <p style="text-align: center;">(a) Shallow specialization.</p>	<pre> If (cond1 & cond2) { // 90% x 90% == 81% probability R0 = R1 = 0 R2 = R3 = 0 R1 = R1 + R3 // statically evalutes to 0 func(0) // deeper specialization } else { R0 = Tex() R2 = Tex() R1 = R0 x 0.5 R3 = R2 x 0.5 R1 = R1 + R3 func(R1) } </pre> <p style="text-align: center;">(b) Deep specialization from combined versioning.</p>
---	---

Fig. 10. Example illustrating choices in handling independent versioning conditions. The optimizer has to judge the relative benefit of deeper specialization at reduced probability over shallow specialization at a higher probability.

the function call $func(R1)$ for $R1==0$ at the same 81% probability. But it comes at the cost of an additional branch, which must be weighed against the extra specialization benefit.

4.3 Performance Challenges

Divergent control flow: Introducing versioning branches or materializing control flow from conditional moves is not without its performance risks. Thanks to NVIDIA® GPUs' SIMT execution model, even if all but one thread of a warp take the fast path and a single thread takes the slow path, the effective latency seen by an instruction that is dependent on this transformed region would be $fastPathCost + slowPathCost$, whereas in the baseline code, the dependent instruction would have seen only $slowPathCost$. So, in cases where the specialization objective is to reduce instruction issue slots, the versioning condition must be passed through an intra-warp *vote* instruction [28], which effectively does a logical AND of the branching condition across all threads of a given warp. The result of the *vote* will be warp-convergent by definition. A branch based on the *vote*'s result will ensure that the fast path will be taken only if all threads of a warp agree to take the fast path, otherwise the slow path is taken. However, if the specialization objective is to reduce bandwidth either from the processing core to the texture unit or to any of the on-chip caches or to main memory, then it might be profitable to allow for divergent branching in Zeroploit-transformed code. Thus, the optimizer must use heuristics to decide when it is desirable to prevent divergent branching.

Short versioning scopes: Even with highly convergent versioning branches, short versioning scopes can splinter straight line basic blocks into multiple smaller basic blocks. A compiler could handle this in two different ways, both of which are far from ideal. It could either retain the branches and the small basic blocks as is, and run the risk of adversely impacting local instruction scheduling by restricting scheduling scopes and adding dynamic branch execution overhead. Otherwise, the compiler could choose to heuristically if-convert [2] short basic blocks to create larger straight-line code regions. This will negate one of the key specialization benefits, namely, redundant instruction elimination, since both the specialized and unspecialized paths' instructions will need to be fetched and issued on architectures like NVIDIA® GPUs. From our experience, it is profitable to create short versioning scopes, only if the baseline code's performance is

bandwidth-limited (main memory, texture unit, or any of the SM's low-throughput functional units) and the specialization objective is to overcome those bandwidth limiters. Barring such situations, optimizers should prefer larger versioning scopes, spanning, at least ten or more scalar operations.

Instruction cache working set bloat: Though we have not encountered this particular performance problem in practice in our manual prototypes, we discuss it briefly here for completeness. While large versioning scopes help maximize specialization benefits, creating large versioning scopes naively can theoretically hurt performance by bloating the instruction cache working set. If the slow path gets taken even at a small, but high enough frequency, then its instructions can destructively interfere at the instruction cache with the fast path's instructions. This can be avoided by constraining versioning scope sizes to not exceed a machine-specific upper bound.

4.4 Outline of Compiler Algorithms

Zeroploit is a profile-guided optimization. We first identify a set of *Zeroploit* candidates via runtime profiling, which informs the subsequent optimization. As we previously discussed, a developer can consult the runtime profile to determine which variables are frequently zero, and how such zero values enable removing forward and backward slices of the computation.

In this section, we sketch a compiler approach that automatically applies *Zeroploit* by following the same basic strategy a human developer would, starting by finding root candidates. By winnowing the set of possible candidates to only those that are not themselves dependent on another candidate, we simultaneously increase the scope of the specializable forward slice, and we reduce the time needed to perform subsequent analyses.

With a set of root candidates to consider, our approach first determines the benefit of each candidate without actually transforming the code. For ease of explanation, however, we describe the analyses that estimate benefit as if they transform the code. At a high-level, we estimate benefits by first finding the set of instructions that become specialized constants (the forward slice), and then by finding the set of instructions that can be removed, because they are no longer required to produce said constants (the backward slice). When each candidate's benefit is calculated, our approach greedily transforms the program by specializing the best candidates, combining them where appropriate.

To estimate the benefit of each candidate, we perform the following analyses:

- (1) Versioning check hoisting: This optional step aims to determine legal, "earlier" locations in the program to which to hoist the computation needed to perform the candidate's versioning check. Hoisting the versioning check can increase the scope of the computation's backward slices. There are several well-known code motion approaches (e.g., loop invariant code motion, partial redundancy elimination, and unification) that can be adapted to perform this task.
- (2) Region identification: This step identifies the set of basic blocks, $\mathcal{R}$, in which any specialization is legal for a given candidate, and it depends on the location of the versioning check. The set is formally defined as $\{b \mid b \in G \text{ and } v \in \text{Dom}(b)\}$, where G represents the set of basic blocks in the control flow graph, v is the block with the versioning check, and Dom is the function that returns the *dominators* of the given block. We can copy the blocks to a new set $\mathcal{S} \leftarrow \mathcal{R}$, and insert an if-then-else hammock that branches either to $\mathcal{R}$ or $\mathcal{S}$, depending on the outcome of the versioning check. We will specialize the instructions in $\mathcal{S}$.
- (3) Constant propagation: Within $\mathcal{S}$, we perform a standard forward constant propagation analysis. When propagating constants, if the definition of a source operand reaches from

a block $b \notin \mathcal{S}$ then we conservatively assume the operand is not constant (i.e., it is $\top$). This analysis discovers the set of instructions for which all destination results will provably return constants.

- (4) Dead code elimination: Source operands are not required to produce a statically known constant, and therefore, newly discovered constant expressions in $\mathcal{S}$ might obviate the need to compute their (useless) source operands. Therefore, we run a backward dead code elimination pass to remove backward slices of computation in $\mathcal{S}$. We augment a standard dead code removal analysis to also take zero-propagator semantics into consideration, where a reaching, provably zero operand obviates the need to compute the other operand.

At the end of this process, we can estimate the overall savings of specialization by considering the set of instructions in $\mathcal{S}$ that can be replaced with a literal move or can be completely eliminated. We can also use the extent of the optimized instructions to sink the versioning check to capture the full benefits of specialization while reducing the amount of redundancy between $\mathcal{S}$ and $\mathcal{R}$. Our benefit estimate depends on the mix of instructions that we can specialize, the frequency of time that the versioning variable is zero, the resultant code bloat, and so on. Determining a good heuristic to wisely choose which candidates to transform is ongoing research. Choosing which subset of candidates to apply the transform to, even given the estimated savings of each, is a challenging problem. We are considering simple greedy approaches that apply the best individual candidates in turn, as well as more complicated approaches that attempt to unify candidates (and their respective specialized regions).

While an automatic *Zeroexploit* implementation is beyond the scope of this article, we evaluate *Zeroexploit* in this article using a manual approach, involving a fair amount of trial and error, for both opportunity identification and code transformation. The next subsection describes how we identify profitable candidates and distills our empirical insights into benefit function heuristics that a future automatic framework might incorporate.

4.5 Heuristics for Benefit Estimation

With *Zeroexploit*, we seek to reduce two fundamental workload metrics, namely, instruction issue slots and texture accesses. Issue slot bloat due to intra-warp divergence can be avoided by using a warp-wide *vote* instruction (see Section 4.3). However, this can thwart texture fetch savings by steering execution away from the specialized path under divergent versioning conditions. So, our goal here is to strike the right tradeoff between issue slot reduction with warp-convergent branching and texture access reduction with simple thread-level branching. For a given versioning scope associated with a versioning variable $\mathcal{V}$, benefit B from *Zeroexploit* can be modeled as $M \times \Delta I + N \times \Delta T$, where ΔI is the net reduction in warp-granular issue slots and ΔT is the net reduction in thread-granular texture accesses from specializing the given versioning scope for $\mathcal{V} == 0$. An automatic framework can calculate ΔI and ΔT , given zero value probability information for $\mathcal{V}$, in a straightforward manner. Weights M and N can be set based on the overall optimization objective for a given shader program and will apply uniformly to all candidate opportunities in a program.

To determine a shader's optimization objective, we first determine its hardware unit utilization metrics by reading hardware performance counters from a profiling run [4]. A unit's utilization is defined as the maximum observed throughput over all its internal pipelines, with each pipeline's throughput being expressed as a percentage of its peak throughput value. Based on unit utilizations, we label a shader as being instruction issue limited, texture (or memory) bandwidth limited, or latency limited as follows. If the SM is the top utilized unit and its utilization is greater than 80%, then we label it issue limited. If the top utilized unit is one of the memory system units (texture unit, L2 cache, or memory) and its utilization is greater than 80%, then we label it bandwidth

Table 4. Gaming Applications Evaluated in This Paper

Application	Short Name	Resolution	dx11	dx12	Anandtech	Guru3d	PCgamer
3DMark Firestrike GT1	3DMFire	1440p	✓			✓	
3DMark Timespy GT2	3DMTime	1440p		✓		✓	
Assassin's Creed Odyssey	ACOD	1440p	✓		✓		✓
Apex Legends	Apex	1440p	✓				
BattleField5	BF5	1440p		✓		✓	
Deux Ex Mankind Divided	DXMD	4K		✓		✓	
The Division 2	Division2	4K		✓	✓		✓
Final Fantasy XV	FFXV	4K	✓				
Forza Horizon 4	FH4	1440p		✓	✓		✓
Metro Exodus	Metro	4K	✓		✓	✓	✓
PlayerUnknown's Battlegrounds	PUBG	4K	✓				
Strange Brigade	SB	4K		✓	✓	✓	✓
Shadow of the Tomb Raider	SOTR	1440p		✓	✓	✓	✓

limited. The 80% threshold is an empirically established value, beyond which non-trivial performance improvements are typically possible only by reducing the relevant type of traffic. Shaders that do not fall into the above categories are labeled latency limited.

For instruction issue limited shaders, issue slot reduction is the sole optimization objective, which can be met by a setting of $\langle M, N \rangle = \langle 1, 0 \rangle$. During benefit estimation and transformation, convergent branching must be used and short versioning scopes avoided, as noted in Section 4.3.

For the other two categories, texture access reduction is the primary optimization objective, to both save bandwidth and lower average memory access latencies. Divergent branching is allowed, even if it makes ΔI negative, as long as it helps reduce texture fetches. For bandwidth limited shaders, setting $\langle M, N \rangle = \langle 0, 1 \rangle$ to give credit only for texture fetch reduction will work best. For latency limited shaders, in lieu of complex, instruction schedule-aware latency computations, a simple heuristic that assigns $\langle M, N \rangle = \langle 1, 25 \rangle$ to give credit for both issue slot and texture fetch reduction might work well in practice and reflects our manual approach. The much higher weight for N is to approximate the average number of issue slots wasted per warp due to texture load-to-use stalls on modern NVIDIA® GPUs.

We apply the *Zeroploit* transform to all versioning scopes in a given shader with $B > 0$.

5 EVALUATION

Gaming applications do not have well-established benchmarks like the SPEC benchmarks. So, we picked the most popular applications from the latest review cycles from three popular review sites, namely, Anandtech [59], Guru3D [33], and PCgamer [62]. We also added a few more popular DX11 gaming applications from previous review cycles to have an even mix of DX11 and DX12 applications. Table 4 lists the gaming applications evaluated in this article.

Using an internal frame-capture tool that is similar in functionality to publicly available tools like Renderdoc [40], NVIDIA® Nsight™ [27], and so on, a single random frame is captured from an in-built benchmark run or from actual gameplay of each gaming application, depending on whether the game has an in-built benchmark. These frame-captures capture both the API sequence (including all relevant state) as well as shader programs. We then run the value profiler from Section 3 on these frames, collect and back-annotate value profiles for all pixel and compute shader programs in all frames. We manually look for *Zeroploit* opportunities in shaders with 2% or higher

Fig. 11. Speedups.

frame time contribution. Where opportunities exist, we manually implement *Zeroploit* at the DirectX byte-code (DXBC) level and run them using the shader-replacement feature of our internal frame capture tool.

We use a recently released NVIDIA® GeForce Game Ready driver [26] for our performance experiments. These runs use production settings for driver and compiler optimizations, which include classical optimizations, like constant folding, dead code elimination, loop unrolling, instruction scheduling, and so on, in addition to various machine-specific optimizations. We verify correctness of our manual transforms by ensuring bit-exactness in the final rendered images. We run our experiments on an NVIDIA® GeForce RTX™ 2080 GPU, locked to base clock settings of 1,515 MHz for the GPU core and production DRAM frequency settings. Full specification of this GPU can be found in Reference [53]. An in-house profiler allows us to accurately measure the GPU time taken for individual calls as well as for the whole frame. Figure 11(a) shows full-frame speedups achieved by *Zeroploit*. Full-frame speedup, the primary metric of interest, is dependent on the frame-time contribution of targeted shader programs and the speedups produced by *Zeroploit* in those targeted regions. The latter is captured in Figure 11(b). The arithmetic mean averages for full-frame and targeted region speedups stand at 2.8% and 35.8%, respectively. We note that these averages are on the higher side due to high outlier data points. Excluding PUBG, the average full-frame speedup comes to 1.9% for the twelve remaining applications, which is still practically very attractive. Planned future work, with the aid of an automatic *Zeroploit* implementation, will evaluate how well these averages hold across a wider collection of applications.

Figure 12 shows how two fundamental metrics, namely, warp instruction count and texture fetch count, and two supporting metrics, namely, average compute shader (CS) warp latency and average pixel shader (PS) warp latency, reduced between the baseline and optimized versions at the full-frame level. Higher reduction percentage conveys that *Zeroploit*'s specialization benefit was higher. Comparing this graph with the one in Figure 11(a), we can see that most applications' frame-level speedups correlate well with either instruction count reduction or texture fetch reduction, the two fundamental metrics *Zeroploit* is expected to influence. On an average, we see a 5.6% reduction in dynamic instruction count and a 6.2% reduction in texture loads, a few percent points higher than the average full-frame performance speedup of 2.8%. Future work could study if the additional reduction in instruction count and texture loads could result in measurable energy savings.

5.1 Discussion

All applications had at least one manually identifiable and exploitable *Zeroploit* opportunity. Across all applications, the list of exploitable useful zero-originators predominantly belonged to either

Fig. 12. Change in key metrics with Zeroploit.

```

ps_5_0
dcl_globalFlags refactoringAllowed
dcl_resource_texture3d (float,float,float,float) t0
dcl_uav_typed_texture3d (float,float,float,float) u0
dcl_input_ps_siv linear noperspective v0.xy, position
dcl_temps 3
ftou r0.xy, v0.yxxx
mov r0.w, l(0)
mov r1.xyz, l(1.000000,1.000000,1.000000,0)
mov r0.z, l(0)
loop
  ige r1.w, r0.z, l(128)
  breakc_nz r1.w
  store_uav_typed u0.xyzw, r0.xzyz, r1.xzyz
  ld_indexable(texture3d) r2.xyzw, r0.xzyz, t0.xzyz
  mul r1.xzyz, r1.xzyz, r2.xzyz
  iadd r0.z, r0.z, l(1)
endloop
ret

```

Fig. 13. Particle shadows shader from 3DMFire.

fpcmp or *texture* categories (defined in Section 3). *fpcmp*, *logical-and*, *fpmnm*, and *intlogic* categories each gave rise to one exploitable candidate. Graphics techniques that benefitted from our manual optimization spanned g-buffer passes, screen-space reflections, global illumination, lighting, and post-processing passes such as motion blur and temporal anti-aliasing. One interesting Zeroploit opportunity that is unique to graphics applications is discussed in detail below.

Alpha-blending optimization: Pixel shaders typically have a four-channel output, RGBA, which stands for red, green, blue, and alpha channels. Typically, the alpha channel gets interpreted as a transparency control value and some algorithms use the alpha value to perform an operation called blending, whereby, a value produced by a pixel shader is blended according to a programmer-specified blending equation, with the value already present in the current render target in memory, for the corresponding pixel position. In SB’s alpha-blending calls, the blending equation is set up as $dst.rgb = src.rgb \times src.alpha + dst.rgb \times (1 - src.alpha)$, where *src.rgb* and *src.alpha* refer to RGB and alpha values produced by the pixel shader and *dst.rgb* refers to the RGB values in the render target. If *src.alpha* is zero, then *src.rgb* becomes irrelevant and *dst.rgb* remains unchanged. We exploit this opportunity by returning early from the pixel shader program and avoiding computation of color channel values, if *src.alpha* is zero dynamically. This contributes 10% of SB’s frame-level speedup. This Zeroploit opportunity is unique to graphics shader programs, as operands that seem unrelated (i.e., the RGB channels and the A channel) from an inspection of the pixel shader program, can become (multiplicatively) related due to API-specified blend equation. To exploit this opportunity in an automatic framework, driver software must process the API-specified blend state and append the blending equation to the shader program code as assembly instructions, so that a Zeroploit-enabled backend compiler can recognize the opportunity and effect the transform.

5.2 Cross-validation

In graphics animation, since scenes change very gradually to give the appearance of continuity, we expect that the speedups demonstrated above will be applicable for 100 or so frames in the vicinity of each of our random candidate frames. But beyond that vicinity, speedups may go up or down. It is possible that as a player moves to different scenes or levels of a game, dynamic value locality might change, causing execution to go down the slow path more often than not, or the relative importance of the shader programs we targeted could diminish, resulting in below par speedups. Likewise, it is possible that dynamic value locality could be more favorably biased toward the fast path or the targeted shaders’ frame-time contributions could increase, possibly resulting in above

Fig. 14. Cross-validating against new frame-captures.

par speedups. In the same vein, if value profiles of more frames are available, then it will be possible to optimize more shader programs or target more opportunities in a given shader program. Doing full justice to these questions would not only require an automatic compiler implementation for *Zeroploit* but also automating profile collection for several far-apart frames during gameplay or in dedicated profile collection runs. Our single-frame-capture-based research prototype is unfortunately not able to scale to full gameplay. Aforementioned automation is beyond the scope of this article, and we plan to pursue that as future work. However, as due diligence, we obtain one additional random single frame-capture for four of our thirteen applications, DXMD, Metro, FFXV, and PUBG, and cross-validate *Zeroploit* by evaluating the performance of the *Zeroploit*-ed shader programs, based on value profiles of the original frames (labeled *frame0*), on the new frames (labeled *frame1*).

Results of these cross-validation experiments are given in Figure 14. From Figure 14(a), we can see that except for DXMD, frame-level speedup reduces a bit in the other three applications on the new frames compared to the original frames. But that is not because our manually optimized shaders are less effective in *frame1* of these applications. As can be seen in Figure 14(b), the targeted shaders perform at par or better on *frame1* than on *frame0* in three of the four chosen applications, indicating dynamic value locality is biased more in favor of the fast paths in those applications. We found that though the targeted shaders performed well, their frame-time contributions reduced, resulting in lower frame-level speedups. Other scenes in these games may respond differently and yield higher speedups. These experiments demonstrate that *Zeroploit*, as a PGO that transforms programs based on value profiles of random frames of a gaming application, can work quite well across large parts of that application to yield good speedups.

6 SILENT STORE OPTIMIZATION

In this section, we describe a value-profile driven PGO to perform targeted, software-only silent store avoidance and evaluate it on one of our applications, 3DMFire. Silent stores are stores that do not change the contents of their memory locations [43]. Our approach targets a subset of an application's silent stores, specifically those that write the same value as the value used to clear or initialize the surface they are writing to. Unlike prior approaches [43, 44], our approach, given its restricted scope, does not require a memory load and comparison of the loaded value with the store's data value to determine if a store is silent or not. In fact, it does not even need the memory address of the store operation. Instead, our approach leverages knowledge about graphics API call sequence to effect a code transform to dynamically avoid targeted silent stores in a non-speculative manner with 100% accuracy. First, the optimizer (game developer or driver software) ensures the presence of a prior `clear()` API call that bulk-clears an entire surface, say S , to a well-defined

initial value. Second, the optimizer ensures that no intervening API calls write to S by inspecting the resources bound to those intervening calls and ensuring that none of them match S . This is sufficient, because, unlike general-purpose programs, wherein any user-mode store operation can write to any part of user memory space, graphics shader programs have no way of reading or writing to resources that are not explicitly bound to the corresponding API calls. Last, in the targeted call, the optimizer makes the stores in the shader program conditional on the data operand not being equal to the prior clear value.

For example, in the particle shadows simulation shader from 3DMFire [47] shown in Figure 13, the same value (of 1.0f) as a prior clear call gets stored to a 3D unordered access view [23] or UAV surface at a high likelihood. Direct3D specification allows accesses to UAVs to happen in any order across the threads of an API call [23]. So, even if data races arise, the above transform will not be in violation of the Direct3D specification. The silent store optimization described above improved this 3DMFire shader by 109% and yielded a frame-level speedup of 1.2%. Performance improved from savings in memory write bandwidth from conditionally avoiding the silent stores. Note, this speedup is independent of and additive with the 1.15% frame-level speedup for 3DMFire from *Zeroploit*, which was realized in a different shader.

This value-dependent optimization is unique to graphics applications and can be effectively implemented by developers or driver software. In the above case, the `clear()` was already present in the API stream. As a potential enhancement, on GPUs that support efficient compressed clears [5, 48], the optimizer could introduce explicit `clear()` calls, where missing, to clear surfaces to the most popular value that is likely to be written to that surface based on prior frames' value profiles and then conditionally skip stores of that value in future calls as described above. Future work will examine the prevalence of such opportunities across a range of applications.

7 RELATED WORK

Short-circuit evaluation is a programming language feature that dynamically skips later sub-expressions in an evaluation sequence, depending on the outcome of earlier sub-expressions. This is routinely used for logical expressions [3, 35], but not for arithmetic expressions [36]. Unlike *Zeroploit*, short-circuit evaluation is not a PGO, but rather an unconditionally applied transform. It operates at a single expression granularity and does not perform forward or backward slice specialization.

Exploiting zero-valued operands of multiply operations is not new. Richardson identified multiply-by-0 as a type of trivial computation and evaluated upside from cutting down the hardware functional unit latency of such multiply operations with a simple instruction-set simulator [54]. Grant et al. explored specializing targeted multiply operations, identified via annotations, for dynamic zeros and eliminated backward-slices of the *other* operands of such multiply operations as dynamic dead-code, as one of the many optimizations for their DyC selective dynamic compilation system [32]. They found it applicable only in 3 of the 11 dynamic regions they studied, whereas we find it profitable across our application suite. From their description, it seemed like though the DyC system normally considered only forward slices for specialization, they make a limited exception for eliminating targeted backward slice operations for dynamically dead assignments. It is not clear if their implementation had any restrictions on how many levels of backward slice operations it would target or if it could identify transitive multiply operations in a root dynamic zero's forward slice and recursively attempt to eliminate those multiply operations' backward slices. In contrast, *Zeroploit*'s profile-guided static approach, by virtue of hoisting the root versioning variable as far up as possible in a shader program, is able to apply forward and backward slice specialization at maximal versioning scopes. *Zeroploit* takes advantage of explicit permission granted by game developers to perform IEEE-unsafe optimizations such as eliminating

backward slices of multiply-by-zero candidates and all our code transforms are productizable. It is not clear if similar guarantees were available to the DyC work.

More generally, value-dependent code specialization has shown promise in partial evaluation systems for functional and imperative language programs [11, 39], general purpose programs [7, 32, 49, 55], embedded software [8], Java just-in-time compilers [29, 57], and so on. While our article has similar goals as those prior efforts, it advances value-based code specialization research in two significant ways. First, we apply value-based code specialization to shader programs of gaming applications and evaluate it on a production GPU, which has not been attempted before, to the best of our knowledge. Next, while aforementioned efforts focused on specializing forward slices of versioning variables for generic values, *Zeroexploit* focuses on exploiting zero-valued source operands of multiply and similar operations to eliminate backward slices of unrelated and independent expression trees (i.e., not directly dependent on the zero-valued operands) by taking advantage of such operations' semantics and achieves both forward and backward slice specialization. Unlike prior specialization efforts, *Zeroexploit* needs to be aware of the SIMT execution model of GPUs and tradeoff warp-convergent execution for specialization benefits as appropriate.

Several researchers have proposed microarchitectural techniques to take advantage of value locality by reusing results from a result cache [54, 60] and designing value predictors to enable speculative execution [30, 45, 46]. Researchers have studied value locality or redundancy in GPGPU compute kernels to detect scalar variables [9, 31, 42], affine computations [42], and value similarity [63] across threads of SIMT warps, thread-block (TB) dimensionality dependent redundancy across threads of TBs [64], and have proposed hardware-only or hardware-software techniques to gainfully exploit such value locality.

Finally, we compare our work on exploiting zeros in modern gaming applications with prior art on leveraging sparsity (i.e., memory loads and computations returning zeros) in deep neural networks (DNNs). Several researchers have exploited sparsity to improve DNN performance in accelerator architectures [1, 34, 50, 65], GPUs [51], and general-purpose processors [56] through hardware enhancements. Like these efforts, *Zeroexploit* seeks to improve performance by leveraging zero valued operands. However, unlike them, *Zeroexploit* uses code specialization to accomplish its goal without relying on any hardware support. Unlike *Zeroexploit*, they make no attempt to eliminate backward slices of multiply operations and focus on optimizing load and multiply operations' forward slices.

8 CONCLUSIONS

There is an urgent need for architectural and software innovations to sustain gaming application performance growth, which will in turn sustain innovations up the gaming software stack and continue the gaming industry's impressive march toward real-time photorealistic rendering. Toward that goal, we explored value-profile-based code specialization for gaming applications in this article and showed that it holds potential. First, we characterized register operand value locality and zero locality in shader programs of gaming applications. Next, we described a novel profile-guided optimization called *Zeroexploit* that exploits dynamically zero-valued source operands of multiply and similar operations and leveraged such operations' semantics to optimize away backward slices of the *other*, unrelated operands of such instructions. *Zeroexploit*, in addition to reaping the benefits of traditional forward-slice specialization due to root zero operands, is able to recursively eliminate backward slices of all multiply and similar operations in such root zero operands' forward slices. We showed that, despite targeting only select shader programs, *Zeroexploit* is able to achieve an average frame-rate improvement of 2.8% and an average speedup of 35.8% for targeted regions in our application suite. We also described a simple value-profile driven silent store optimization and

demonstrated a 109% speedup on a single targeted shader of one application, amounting to 1.2% frame-rate speedup on that application.

Our work has proven that there is tangible performance in understanding and exploiting operand value locality, particularly zeros, in modern gaming applications. However, in this article, we implemented *Zeroploit* manually, with the aid of an offline value profiler in a research prototype. More work is needed to implement and test an automatic compiler pass for *Zeroploit* and determine how best to productize this optimization along with a value profiler, within the constraints of GPUs' runtime systems and JIT compilers and without affecting interactive gameplay performance, while still ensuring a net performance win. We plan to work on the above topics going forward.

ACKNOWLEDGMENTS

We thank our article's anonymous reviewers as well as Nitya Ranganathan for their valuable comments that have strengthened the article. Many thanks to Slava Gostrenko and Xingyong LV for tools support, to NVIDIA's Devtech, QA, and DX driver performance teams for providing the application traces used in this work, and the entire Applied Architecture team for their support during this work.

REFERENCES

- [1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 1–13.
- [2] J. R. Allen, Ken Kennedy, Carrie Porterfield, Joe Warren, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83)*. 177–189.
- [3] Bruce W. Arden, Bernard A. Galler, and Robert M. Graham. 1962. An algorithm for translating boolean expressions. *J. ACM* 9, 2 (Apr. 1962), 222–239.
- [4] Louis Bavoil. 2019. *The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload*. Retrieved from <https://devblogs.nvidia.com/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/>.
- [5] Chris Brennan. 2016. *Delta Color Compression Overview*. Retrieved from <https://gpuopen.com/dcc-overview/>.
- [6] Brad Calder, Peter Feller, and Alan Eustace. 1997. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 259–269.
- [7] Brad Calder, Peter Feller, and Alan Eustace. 1999. Value profiling and optimization. *J. Instruct. Level Parallel*. 1 (Mar. 1999). Retrieved from <https://www.jilp.org/vol1/v1paper2.pdf>.
- [8] Eui-Young Chung, B. Luca, G. DeMicheli, G. Luculli, and M. Carilli. 2002. Value-sensitive automatic code specialization for embedded software. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 21, 9 (Sep. 2002).
- [9] Sylvain Collange. 2011. *Identifying Scalar Behavior in CUDA Kernels*. Technical Report HAL-00555134.
- [10] Microprocessor Standards Committee. 2019. *754-2019-IEEE Standard for Floating-Point Arithmetic*. Retrieved from <https://ieeexplore.ieee.org/servlet/opac?punumber=8766227>.
- [11] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volansche. 1996. A uniform approach for compile-time and run-time specialization. In *Selected Papers from the International Seminar on Partial Evaluation*.
- [12] Microsoft Corporation. 2015. *Fixed Order of Pipeline Results*. Retrieved from https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm#4.2%20Fixed%20Order%20of%20Pipeline%20Results.
- [13] Microsoft Corporation. 2015. *Unordered Access Views*. Retrieved from https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm#UAVs.
- [14] Microsoft Corporation. 2018. *Atomic Iadd*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/atomic-iadd--sm5---asm>.
- [15] Microsoft Corporation. 2018. *Direct3D 11 Graphics*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11>.
- [16] Microsoft Corporation. 2018. *Direct3D 12 Graphics*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics>.
- [17] Microsoft Corporation. 2018. *Effect-Compiler Tool*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dtools/ffc>.

- [18] Microsoft Corporation. 2018. *High Level Shading Language*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hlsl>.
- [19] Microsoft Corporation. 2018. *movc (sm4-asm)*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/movc--sm4---asm>.
- [20] Microsoft Corporation. 2018. *Shader Model 4 Assembly (DirectX HLSL)-dcl_globalFlags*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/dcl-globalflags>.
- [21] Microsoft Corporation. 2018. *Shader Model 5 Assembly (DirectX HLSL)*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/shader-model-5-assembly--directx-hlsl>.
- [22] Microsoft Corporation. 2018. *Sync*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/sync--sm5---asm>.
- [23] Microsoft Corporation. 2018. *Unordered Access Buffer or Texture*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-cs-resources#unordered-access-buffer-or-texture>.
- [24] Microsoft Corporation. 2018. *Variable Syntax*. Retrieved from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hlsl-variable-syntax>.
- [25] Microsoft Corporation. 2019. *DirectX Intermediate Language*. Retrieved from <https://github.com/Microsoft/DirectXShaderCompiler/blob/master/docs/DXIL.rst>.
- [26] NVIDIA Corporation. 2019. *GeForce Game Ready Driver, Version 441.41-WHQL*. Retrieved from <https://www.geforce.com/drivers/results/155060>.
- [27] NVIDIA Corporation. 2019. *Nsight 2019.6*. Retrieved from <https://developer.nvidia.com/nsight-graphics>.
- [28] NVIDIA Corporation. 2019. *Parallel Thread Execution ISA: Application Guide*. Retrieved from https://docs.nvidia.com/pdf/ptx_isa_6.5.pdf.
- [29] Igor Costa, Pericles Alves, Henrique Nazare Santos, and Fernando Magno Quintao Pereira. 2013. Just-in-time value specialization. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. 1–11.
- [30] Freddy Gabbay and Avi Mendelson. 1996. *Speculative Execution Based on Value Prediction*. Technical Report. Technion-Israel Institute of Technology, EE Department, TR 1080.
- [31] S. Z. Gilani, N. S. Kim, and M. J. Schulte. 2013. Power-efficient computing for compute-intensive GPGPU applications. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 330–341.
- [32] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. 293–304.
- [33] Hilbert Hagedoorn. 2019. *NVIDIA GeForce RTX 2080 SUPER (8GB Founder)*. Retrieved from <https://www.guru3d.com/articles-pages/geforce-rtx-2080-super-review,1.html>.
- [34] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 243–254.
- [35] H. D. Huskey and W. H. Wattenburg. 1961. Compiling techniques for boolean expressions and conditional statements in ALGOL 60. *Commun. ACM* 4, 1 (Jan. 1961), 70–75.
- [36] Randall Hyde. 2006. *Writing Great Code, Volume 2: Thinking Low-Level, Writing High-Level*. No Starch Press, Chapter 13, 427–435.
- [37] The Khronos Group Inc.[n.d.]. *OpenGL Overview*. Retrieved from <https://www.opengl.org/documentation/>.
- [38] The Khronos Group Inc.2018. *Vulkan Overview*. Retrieved from <https://www.khronos.org/vulkan/>.
- [39] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Upper Saddle River, NJ.
- [40] Baldur Karlsson. 2019. *Renderdoc v1.5*. Retrieved from <https://renderdoc.org/docs/index.html>.
- [41] John Kessenich, Dave Baldwin, and Randi Rost. 2017. *The OpenGL Shading Language*. Retrieved from <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf>.
- [42] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural mechanisms to exploit value structure in SMT architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 130–141.
- [43] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the value locality of store instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*.
- [44] Kevin M. Lepak and Mikko H. Lipasti. 2000. Silent stores for free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*. 22–31.
- [45] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'96)*. 226–237.

- [46] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*. 138–147.
- [47] Future Mark. 2019. *3DMARK® Technical Guide*. Retrieved from <https://s3.amazonaws.com/download-aws.futuremark.com/3dmark-technical-guide.pdf>.
- [48] D. K. McAllister, S. E. Molnar, Jr. J. F. Duluk, E. M. Kilgariff, P. R. Brown, C. J. Amsinck, J. M. O'Connor, J. M. Burgess, G. A. Muthler, and J. Robertson. 2012. Zero Bandwidth Clears. United States Patent No. 8330766.
- [49] Robert Muth, Scott A. Watterson, and Saumya K. Debray. 2000. Code specialization based on value profiles. In *Proceedings of the 7th International Symposium on Static Analysis (SAS'00)*. Springer-Verlag, London, 340–359.
- [50] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Bruce Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 27–40.
- [51] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2016. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In *Proceedings of the 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES'16)*. 33:1–33:10.
- [52] Alex Peleg, Sam Wilkie, and Uri Weiser. 1997. Intel MMX for multimedia PCs. *Commun. ACM* 40, 1 (1997).
- [53] Tech Powerup. 2018. *NVIDIA GeForce RTX 2080*. Retrieved from <https://www.techpowerup.com/gpu-specs/geforce-rtx-2080.c3224>.
- [54] Stephen E. Richardson. 1992. *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation*. Technical Report SMLI TR-92-1. Sun Microsystems Laboratories, Inc.
- [55] S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. 2000. Characterizing coarse-grained reuse of computation. In *Proceedings of the ACM Workshop on Feedback Directed and Dynamic Optimization*.
- [56] Sanchari Sen, Shubham Jain, Swagath Venkataramani, and Anand Raghunathan. 2018. SparCE: Sparsity aware general-purpose core extensions to accelerate deep neural networks. *IEEE Trans. Comput.* 68, 6 (Nov. 2018).
- [57] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. 2005. Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. 327–343.
- [58] Nathan T. Slingerland and Alan Jay Smith. 2000. *Multimedia Instruction Sets for General Purpose Microprocessors: A Survey*. Technical Report UCB/CSD-00-1124. University of California, Berkeley, Computer Sciences Division.
- [59] Ryan Smith. 2019. *The NVIDIA GeForce RTX 2080 Super Review: Memories of the Future*. Retrieved from <https://www.anandtech.com/show/14663/the-nvidia-geforce-rtx-2080-super-review/3>.
- [60] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*. 194–205.
- [61] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 185–197.
- [62] Jarred Walton. 2019. *NVIDIA GeForce RTX 2080 Super Review*. Retrieved from <https://www.pcgamer.com/nvidia-geforce-rtx-2080-super-review/>.
- [63] D. Wong, N. S. Kim, and M. Annavaram. 2016. Approximating warps with intra-warp operand value similarity. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 176–187.
- [64] Tsung Tai Yeh, Roland N. Green, and Timothy G. Rogers. 2020. Dimensionality-aware redundant SIMT instruction elimination. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 1327–1340.
- [65] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

Received January 2020; revised April 2020; accepted April 2020