Locality-Aware CTA Scheduling for Gaming Applications

ADITYA UKARANDE, SURYAKANT PATIDAR, and RAM RANGAN, NVIDIA, India

The compute work rasterizer or the GigaThread Engine of a modern NVIDIA GPU focuses on maximizing compute work occupancy across all streaming multiprocessors in a GPU while retaining design simplicity. In this article, we identify the operational aspects of the GigaThread Engine that help it meet those goals but also lead to less-than-ideal cache locality for texture accesses in 2D compute shaders, which are an important optimization target for gaming applications. We develop three software techniques, namely *LargeCTAs*, *Swizzle*, and *Agents*, to show that it is possible to effectively exploit the texture data working set overlap intrinsic to 2D compute shaders.

We evaluate these techniques on gaming applications across two generations of NVIDIA GPUs, RTX 2080 and RTX 3080, and find that they are effective on both GPUs. We find that the bandwidth savings from all our software techniques on RTX 2080 is much higher than the bandwidth savings on baseline execution from inter-generational cache capacity increase going from RTX 2080 to RTX 3080. Our best-performing technique, *Agents*, records up to a 4.7% average full-frame speedup by reducing bandwidth demand of targeted shaders at the L1-L2 and L2-DRAM interfaces by 23% and 32%, respectively, on the latest generation RTX 3080. These results acutely highlight the sensitivity of cache locality to compute work rasterization order and the importance of locality-aware cooperative thread array scheduling for gaming applications.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Graphics processors**;

Additional Key Words and Phrases: Cache locality, CTA scheduling, GPUs, shader programs

ACM Reference format:

Aditya Ukarande, Suryakant Patidar, and Ram Rangan. 2021. Locality-Aware CTA Scheduling for Gaming Applications. *ACM Trans. Arch. Code Optim.* 19, 1, Article 1 (December 2021), 26 pages.
<https://doi.org/10.1145/3477497>

1 INTRODUCTION

The 2D grids of compute shaders are a key part of modern interactive computer graphics. They are used to implement graphics algorithms like raytracing (with DXR 1.1 [42]), denoisers, and a variety of post-processing effects such as screen-space reflections, ambient occlusion, and motion blur. They not only bestow significant photo-realism to game scenes but also make up a significant portion (40%) of a gaming application's average frame time. The stacked bar graph in Figure 1 conveys the percentage of frame-time spent in 2D and non-2D compute calls in the gaming applications we study in this article, with the frame time contribution of 2D compute calls further dissected by

Authors' addresses: A. Ukarande, S. Patidar, and R. Rangan, NVIDIA, C1, Jacaranda, Manyata Embassy Business Park, Outer Ring Road, Bangalore, India; emails: {aukarande, spatidar, rrangan}@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 the author(s) 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](https://permissions.acm.org).

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2021/12-ART1 \$15.00

<https://doi.org/10.1145/3477497>

Fig. 1. Percentage of frame time spent in 2D compute (broken down by primary performance limiter) and other compute.

Fig. 2. Elliptical filters from a reflection denoiser.

their primary performance limiter being memory latency (blue sub-bar), memory bandwidth (yellow sub-bar), or others (green sub-bar)¹. As can be seen, latency and bandwidth limited shaders add up to almost 72% of all 2D compute shaders. In this article, we focus on understanding the memory access patterns of latency and bandwidth limited 2D compute shaders and strive for performance enhancements through improved GPU caching behavior.

The 2D compute shaders typically access large 2D textures with sampling patterns called *filters*. A filter selects one or more texture elements (texels) around a center point. Individual shader program threads fetch the filter texels corresponding to their screen position and perform averaging math on them to reduce them to a single value. Depending on the specific algorithm, these filters can be circles with fixed or random radii, ellipses along varying axes, and so on. Regardless of the exact filter, one common feature shared by these shaders is that their filter regions typically tend to show significant overlap across screen-neighboring thread blocks or **cooperative thread arrays (CTAs)**. For example, Figure 2 shows the elliptical filters in a reflection denoiser from an NVIDIA demo. Given that these filters correspond to individual threads responsible for denoising corresponding screen-mapped pixels, it is not hard to imagine how the filter regions from screen-neighboring CTAs will overlap spatially. The key to exploiting such algorithmic locality is to carefully orchestrate the scheduling of screen-neighboring CTAs to maximize temporal overlap.

However, this is easier said than done. Since modern NVIDIA GPUs cater to a variety of markets from gaming to deep learning and high-performance computing, they are not hyper-optimized for specific memory access patterns. Specifically, the **GigaThread Engine (GE)** of modern NVIDIA GPUs rasterizes all compute grids in a **load-balanced (LB)** row-major fashion, regardless of whether the compute grid is a CUDA grid or a DirectCompute one. The GE’s simple, generic design, although great for maximizing GPU-wide CTA occupancy, poses a few challenges for cache locality of 2D compute shaders. First, LB rasterization prevents screen-neighboring CTAs from being scheduled on the same **streaming multiprocessor (SM)**. Second, non-determinism due to load-balancing prevents software-based CTA remapping algorithms that assume deterministic **round-robin (RR)** rasterization from effectively exploiting L1 locality [24]. And third, row-major rasterization of CTAs allows for inter-CTA filter region overlap only along the *x*-axis, but not along the *y*-axis.

¹We categorize primary limiters based on unit utilization metrics, derived from hardware performance counters [5]. A hardware 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. If the top utilized unit for a given call is L2 or DRAM and its utilization is greater than or equal to 70%, we consider that call as being bandwidth limited. If no single GPU unit’s utilization is more than 70% for a given call, that call is labeled as being latency limited. Otherwise, a call’s limiter is labeled as “others.” At the frame level, we roll up by going over each call in the frame and accumulating the entire time spent in a given call against its primary limiter.

Existing software solutions for inter-CTA locality are tailored to the characteristics of GPGPU compute applications. For example, some techniques optimize for 1D array accesses, or row-major, column-major, or diagonal accesses of 2D arrays commonly seen in GPGPU applications [24], whereas others rely on simple addressing math found in those applications [7, 24]. Since 2D compute shaders in gaming applications exhibit tiled texture access patterns and since such textures' coordinates are typically not statically analyzable, prior techniques cannot be applied directly. Therefore, we explore the following novel techniques in this article:

- *LargeCTAs* increases CTA sizes (and reduces grid size commensurately) to co-locate screen neighboring work tokens in the same SMs, thereby enabling their texture working sets to constructively interfere in the associated L1 caches. Although the locality benefits from larger CTAs are intuitive, they come at the cost of poorer CTA scheduling freedom, besides severely constraining register target selection for the compiler.
- *Swizzle* exploits knowledge about row-major rasterization order to remap CTA identifiers on-the-fly through a simple math-only code sequence to achieve tiled rasterization. The resulting rasterization order increases temporal overlap of the texture working sets from screen-neighboring CTAs. This technique primarily targets L2 locality but has no control over L1 locality.
- *Agents* are persistent CTAs that invoke the original shader program in a long-running loop. Shader invocations from multiple agent warps, collectively called **virtual CTAs (vCTAs)**, effectively map to equivalent-sized regular CTAs of the original kernel. Agents circumvent the hardware rasterizer and precisely control the mapping of screen-mapped vCTAs to SMs. However, unlike prior work [24], our agents do not have a fixed mapping of vCTAs to SMs. They perform work-stealing to ensure LB scheduling and can flexibly target both L1 and L2 locality.

We conclude this section by stating our contributions:

- (1) We describe the high-level operation of NVIDIA's GE, rationalizing its design with runtime statistics and targeted experiments, while also pointing out why its rasterization order is less than ideal for tiled texture access patterns. Besides laying the groundwork for our work, we hope this exposition addresses researchers' long-standing demand for details of the GE [24, 53].
- (2) We explore three novel software techniques to effectively exploit the natural working set overlap found in 2D compute shaders, namely *LargeCTAs*, *Swizzle*, and *Agents*.
- (3) We evaluate these techniques on modern gaming applications across two generations of NVIDIA GeForce GPUs—the RTX 2080 (Turing) and the RTX 3080 (Ampere). Key aspects include:
 - (a) An exhaustive parameter space exploration for all our techniques to identify best achievable upside, labeled BestOf, from customized parameterization. This will appeal to ninja game developers seeking to develop highly optimized shaders.
 - (b) A novel feedback-based heuristic to automatically classify shader programs as being L1 or L2 hit-rate sensitive and, accordingly, apply one of two generic parameter settings. This can lend itself to automation in a GPU driver.

Despite targeting a subset of all 2D compute shaders, our techniques record average BestOf frame-level speedups of up to 4.7% (or 31.2% on the targeted shaders). For these targeted shaders, our best-performing technique reduces L1-L2 and L2-DRAM bandwidth demand by 23% and 32%, respectively, on RTX 3080.

The next section provides a brief introduction to graphics programs. In the section following that, we provide operational details of the GE and discuss how CTA rasterization can impact L1 and L2 cache locality in 2D compute shaders.

2 BACKGROUND

Modern gaming applications are most popularly developed in Direct3D 11 [31], Direct3D 12 [32], OpenGL [44], and Vulkan [45] **application programming interfaces (APIs)**. Without exception, all graphics APIs use a two-level hierarchy to convey work from the CPU to the GPU: an API layer and a shader program layer. The API layer is used to set up GPU state (enable or disable depth testing, blending, etc.), bind resources such as constant buffers, textures (input data), render targets (output data), or general multi-dimensional arrays called *unordered access views* (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 on the SM. Shader programs are typically written in high-level languages such as the high-level shading language (HLSL) used with Direct3D APIs [33], or the OpenGL Shading Language (GLSL) used with OpenGL or Vulkan APIs [19].

A single frame of a modern game typically requires hundreds or even thousands of state setup and GPU work calls. 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. This work focuses on games with a significant frame-time contribution from 2D compute shaders.

The Direct3D specification requires only API call-level memory ordering [29], allows for arbitrary rasterization order for CTAs within a given compute call [28], and provides no ordering guarantees for updates to unordered access views, which serve as the output surfaces of compute calls [30]. The preceding guarantees ensure that different CTA scheduling strategies will not impact updates to application data structures and hence program correctness, as long as all CTA identifiers in any given compute call are ultimately rasterized and scheduled for execution. Other popular compute APIs also allow for arbitrary CTA rasterization orders, and thus techniques discussed in this article can be applied to programs written in those APIs as well [10, 12].

3 CTA SCHEDULING IN NVIDIA GPUS

An NVIDIA GPU has a hierarchical organization. At the top level are one or more **graphics processing clusters (GPCs)**. Each GPC contains multiple texture processing clusters. On modern GPUs such as those belonging to the Turing and Ampere families, each texture processing cluster contains two SMs, each associated with a dedicated texture unit and L1 cache. All GPCs share a common L2 cache accessed over a crossbar.

Compute shaders execute on SMs. The compute work rasterizer or the GE is responsible for rasterizing a compute grid launched from the CPU into compute work tokens called *cooperative thread arrays* (CTAs) and assigning those CTAs to individual SMs. “Rasterization” here simply involves assigning CTA identifiers correctly, ensuring that all CTAs of a compute grid eventually get scheduled for execution on SMs, and tracking their completions. The identifiers assigned to successive CTAs are in accordance with the rasterization order implemented by the GE.

The GE has evolved over the years to meet the demands of new compute features added to NVIDIA GPUs. But its primary goal has remained the same, which is to fill the GPU maximally with compute work (i.e., maximize dynamic CTA occupancy). The GE on modern NVIDIA GPUs, like those based on Pascal, Turing, and Ampere architectures [35, 36, 40], achieves that goal by implementing an LB row-major rasterization. This rasterization behavior is applicable to all compute kernel launches regardless of the software API used (i.e., CUDA, DirectCompute, Vulkan).

Fig. 3. CTA latency heatmaps for 2D compute calls corresponding to two different graphics techniques from two different gaming applications. Blue regions indicate short CTA latencies, whereas dark red regions indicate long CTA latencies. Intermediate colors convey various in-between latencies.

The GE periodically obtains information on availability of CTA resources in individual SMs through a dedicated control network. This information is then used to perform LB CTA scheduling. Prior to the load-balancing step, the GE determines the identifier for the next CTA to be scheduled as per the row-major enumeration. The LB logic goes over CTA resource availability across all SMs and picks the least-loaded SM to send the CTA to. It breaks ties by preferring the SM with the smallest global identifier. The next section provides statistics and arguments in support of this rasterization behavior.

3.1 Rationale

Typically, 2D compute shader programs show non-trivial control flow and dependent memory fetches. Some of the control flow paths may lead to early exits. As a result, there tends to be substantial variability in their CTA latencies, even among screen-neighboring ones.²

This is best understood by plotting latencies of screen-mapped CTAs of 2D compute calls as heatmaps. For example, Figure 3(a) shows the latency heatmap of latencies of all CTAs in a 2D compute shader performing reflection denoising of a 4K image from a scene in the gaming application, BFV. Likewise, Figure 3(b) gives the heatmap of CTA latencies from a screen-space ambient occlusion call in another gaming application. As can be seen, there are sections of heatmaps that are a cool blue indicating very short CTA latencies, sections that are dark red indicating CTAs processing those regions of the screen take really long latencies, and then blocks with various intermediate shades.

A less visually rich but more comprehensive summary of variability in CTA latencies is provided in Figure 4. This figure shows the normalized standard deviation in CTA latencies for all applications. Given such high variability in CTA latencies, it stands to reason that load-balancing is a critical component of the GE's rasterization policy.

To further drive home the importance of LB rasterization, we perform a simple experiment. We implement a variant of our *Agents* or persistent CTAs software approach by turning off its work-stealing features and performing a simple RR rasterization of vCTAs across SMs. The results, shown in Figure 5, are unequivocally bad and reinforce the need for LB rasterization. Intuitively, load-balancing helps the GE react to dynamic CTA latencies and distribute them more evenly across the SMs than a simple screen-mapped assignment of CTAs to SMs. This is because the CTAs belonging to the “red hot” regions of latency heatmaps execute on a handful of SMs and those SMs

²CTA latency is defined as time between launch of first warp of the set of warps that make up a CTA to the retirement of the last warp in that group.

Fig. 4. Normalized standard deviations of CTA lifetimes.

Fig. 5. Performance impact of non-LB rasterization.

Fig. 6. Lost overlap opportunity due to row-major rasterization. Orange squares indicate work tokens or CTAs, whereas blue circles indicate filter regions for a 3×2 region of CTAs on the left top. The arrows show the row-major rasterization order. By the time the CTA for (column 0, row 1) gets scheduled, the overlapping filter region brought into the GPU's caches from (column 0, row 0) will have been evicted.

take longer to complete. Meanwhile, the other SMs will have finished with their allotted CTAs and would be idling, thus bringing down overall GPU utilization and performance.

Last, we note that the GE performs row-major rasterization because it is practically very difficult for the hardware to determine the exact data-access pattern for every compute shader and rasterize accordingly. Although 2D grids make up the bulk of the compute calls in gaming applications, with a relatively smaller contribution from 1D and 3D grids, other domains may see a different distribution across grid types [4]. Consequently, texture or global memory access patterns centered around grid coordinates will vary across these domains. Therefore, the GE implements row-major rasterization, not only because it is a well-understood rasterization order (since most languages lay out multi-dimensional arrays in row-major order), but also because it is simple to implement in hardware.

In the next section, we describe how the preceding aspects of the GE's design impact cache locality for 2D compute shaders.

3.2 Locality Impact

Although LB rasterization in the GE is great for maximizing GPU-wide CTA occupancy, it makes it highly unlikely for screen-neighboring CTAs to go to the same SM and benefit from constructive interference in the private L1 caches. Some researchers have attempted improving L1 locality by assuming strict RR rasterization (i.e., without load-balancing) and remapping CTA identifiers on the fly so that CTAs executing on a given SM get remapped to back-to-back CTA identifiers in grid space, even if the original CTA identifiers they were launched with were far apart. But such remapping strategies were not effective [24], likely due to load-balancing invalidating the strict RR assumption.

Next, the GE's row-major rasterization order poses challenges for tiled texture access patterns typically exhibited by 2D compute shaders. Specifically, maximal data reuse is possible only when most, if not all, of the work tokens or CTAs accessing a particular region of a texture are concurrently active in the GPU. If CTAs whose filters intersect are active or alive at different points in time, the CTAs arriving later might have to refetch the same texture data, since the cache lines fetched from the earlier CTAs will likely have been evicted due to intervening conflict misses from other CTAs. Figure 6 illustrates this by using orange squares for rasterized CTAs and blue circles to represent filter regions for the CTAs on the left end. Each row in the grid is a row of CTAs. By the time the GE returns to rasterize the CTAs at the left end of the screen on the second row, the cache lines corresponding to the filter regions from screen-neighboring CTAs from the first row will likely have been evicted from the GPU's caches.

Thus, whereas LB rasterization impacts L1 locality by steering screen-neighboring CTAs to different SMs, row-major rasterization prevents timely reuse of cached filter data by bringing in filter data for new CTAs along a row and thrashing the L2 cache before data can be reused by screen-neighboring CTAs. In the next section, we present three simple software solutions to effectively exploit working set overlap in 2D compute shaders.

4 SOFTWARE TECHNIQUES

In this section, we evolve three novel software techniques, on top of the native rasterization behavior of the GE, to effectively harvest the natural memory access locality found in 2D compute shaders. The techniques are *LargeCTAs*, *Swizzle*, and *Agents*. Although none of these techniques attempt to capture all inter-CTA cache locality in 2D compute shaders perfectly, the common theme across all of them is that they strive to achieve better spatio-temporal locality of CTAs working on a given screen region than the baseline rasterization order. These techniques change the CTA rasterization order to improve cache locality. Our techniques are suitable for adoption either by game developers at the application source level or for inclusion as a performance optimization in a GPU driver. Whereas the transforms for *LargeCTAs* and *Agents* necessitate API as well as shader program changes, the transform for *Swizzle* involves only shader program changes. Depending on the deployment model, these changes may be performed either offline by game developers at the application source level or online during just-in-time (JIT) compilation by GPU drivers. We describe the techniques in detail below.

4.1 *LargeCTAs*

LargeCTAs is a very simple and intuitive technique that keeps screen-neighboring CTAs together on the same SM by increasing CTA size and commensurately reducing grid size. Since *LargeCTAs* are created statically, the software optimizer (or the programmer) can implement this by changing the CTA size of a targeted shader to span larger squares or rectangles and reduce its grid launch dimensions proportionally. Since a CTA executes wholly in a single SM, the threads of these *LargeCTAs* will access the same L1 cache, achieving higher constructive interference in the L1 and reducing L1 miss-rates. This technique does not target L2 locality in any way.

That said, *LargeCTAs* has two drawbacks. First, the larger the CTA size, the more constrained the compiler's register allocator becomes. For example, the compute shader for a 32-warps CTA can use at most 64 registers. Any additional live operands will need to be spilled. Such spilling can eat into the performance gains from improved L1 locality or at times even negate it and produce slowdowns. Second, *LargeCTAs* can suffer from poor occupancy arising from out-of-order warp completion [54]. A CTA's execution is most efficient from an occupancy standpoint if all its constituent warps retire in close succession. However, this is seldom the case in practice due to significant variability in warp latencies stemming from control flow and dependent memory access

latencies. The GE is unable to reuse such partially retired CTAs' resources to launch a new large CTA unless enough resources are available to launch a full CTA in a given SM. For large CTAs, it takes longer to accumulate enough resources to be able to launch a full CTA. During such periods, the GPU will be under-utilized due to reduced warp occupancy.

One limitation of *LargeCTAs* is that it cannot be used to capture filter data locality in screen regions larger than the largest allowed CTA size, which is 1,024 threads on modern GPUs like RTX 2080 and RTX 3080. Adjacent *LargeCTAs* are subjected to the GE's LB rasterization and will likely be scheduled on different SMs. This is good for GPU-wide CTA occupancy as it enables new *LargeCTAs* to quickly find an SM with enough free resources, even though it means locality of accesses over large filter regions must be sacrificed.

A second limitation of *LargeCTAs* is that it is unable to achieve effective temporal overlap of CTAs working on large filters (i.e., ones whose texture working sets span multiple *LargeCTAs*) at the L2 cache level. Such filter regions lose L2 cache locality due to the GE's row-major rasterization, as explained in Section 3. The next technique, which is orthogonal to *LargeCTAs*, will address this second problem.

4.2 Swizzle

Recall from Section 3 that row-major rasterization can kick out filter texel data from on-chip caches before they get reused. The *Swizzle* technique's goal is to maximize filter region reuse from the shared L2 cache by carefully determining the screen-space regions that active CTAs in the GPU work on. Since CTAs that work on all parts of a screen are identical in all respects (since they execute the same shader code) except for the identifiers assigned to them, *Swizzle* accomplishes its goal by dynamically swizzling or remapping GE-assigned CTA identifiers in the shader code to new locality-friendly CTA identifiers, which then leads to dramatically improved filter data reuse.

This technique benefits from the GE's load-balancing policy in the same way that *LargeCTAs* does. In addition, *Swizzle* leverages knowledge that the GE's default rasterization order is row-major to perform remapping of CTA identifiers with simple math-only code snippets. Our *Swizzle()* method takes four input parameters, namely *CTAID.xy*, *tileSize*, *tileAxis*, and the *nextCtaDir*, of which the last three together represent a swizzle configuration. *CTAID.xy* conveys the default row-major CTA identifier assigned by the GE. *tileAxis* can take only two possible values, **rowmajor** or **colmajor**. This indicates the axis along which tiled rasterization proceeds.

The *tileSize* parameter specifies one side of a tile in terms of the number of CTAs: either the height or the width depending on whether *tileAxis* is **rowmajor** or **colmajor**, respectively. The other side gets determined automatically at runtime by the maximum GPU-wide CTA occupancy, *gpuCTAs*, allowed.³ A contiguous moving window of *gpuCTAs* traverses the entire screen area in a tiled fashion. For example, if 256 CTAs can be alive at any given point in a GPU for a given shader, and if *tileSize* is specified as 32 CTAs, the other dimension gets automatically regulated to $\frac{256}{32} = 8$ CTAs.

For finer control within a tile, *nextCtaDir* specifies how to locate the next CTA to be rasterized, when rasterization hits the *tileSize* limit in the current row or column. Its possible values are **return-to-start** (RTS or rts) and **boustrophedonic** (BOU or bou). Whereas **return-to-start** returns to the beginning of the column for the next row for row-major traversal, **boustrophedonic** reverses direction every time it hits the last CTA of a row and begins rasterization from the "first" CTA of the next row in the new direction. Column major rasterization is handled similarly as well.

The *tileAxis* and *tileSize* parameters together help to process an entire screen-space in terms of horizontal or vertical strips of CTAs. Any remaining CTAs are handled as part of a residual strip.

³*gpuCTAs* = *numSMs* × *smCTAs*, where *smCTAs* is the maximum number of CTAs that can be live in a single SM at any given point, as determined by resource demand of a given shader.

Fig. 7. Rasterization of *Swizzle* configuration (4, **colmajor**, **boustrophedonic**) through a screen-space of 10×8 CTAs. Each small solid black square indicates a CTA. This example assumes only eight CTAs can be active GPU-wide. Each dashed region indicates a “tile” (i.e., group of CTAs active in the GPU at any point in time).

Fig. 8. Illustration of *Agents* terminology. Tiled rasterization is achieved in this example with **colmajor**, **boustrophedonic** traversal both within and across L2Tiles.

After swizzled rasterization hits the end of a strip, our implementation makes rasterization always resume at the beginning of the next strip. These concepts, along with the contiguous moving window of CTAs are illustrated in Figure 7 for a swizzle tuple of (4, **colmajor**, **boustrophedonic**). Each small solid black square is a CTA. The illustration assumes eight CTAs can be live in the GPU at any point. Rasterization order is boustrophedonic within each column of *tileSize* count number of CTAs. Notice how the shape of the “tiles” represented by the dashed regions changes due to CTAs completing out of order, but still retains contiguity on a best-effort basis as per the intended swizzle pattern. The magenta colored tile toward the bottom of the first column and top of the second column shows how an active tile briefly loses contiguity at screen-space boundaries. In this way, the *Swizzle()* sequence limits the active texture data working set to a tile of CTAs, inside which texture data reuse is expected to be much higher than a set of CTAs from a simple, row-major rasterization order. The procedure to perform swizzling is given in Algorithm 1, which amounts to about 40 assembly instructions. In practice, the overhead of this *Swizzle* logic is well amortized by gains from improved cache locality in latency and bandwidth limited compute shaders.

The 2D compute shaders we studied did not react uniformly to all the swizzle tuples we tried out. Different swizzle tuples were found beneficial in individual shaders. This flexibility to explore such a wide variety of swizzle patterns is enabled by the GE’s simple, well-understood row-major default rasterization order. However, despite its practical usefulness and flexibility, *Swizzle* is unable to address loss of L1 locality since it is not immune to LB rasterization. The next technique, *Agents*, will attempt to capture both L1 and L2 locality by circumventing the hardware rasterizer.

ALGORITHM 1: Swizzle logic.

```

/* isCM(A) checks if A is colmajor. */
I : curID.xy : CTAID.xy, A: tileAxis, TSz: tileSize, D: nextCtaDir
O: SwizID.xy : swizzledCTAID.xy
1 Procedure Swizzle()
2   FlatId  $\leftarrow$  curID.y * gridDim.x + curID.x
3   (SSz, NS, RSSz)  $\leftarrow$  getStripInfo(A, TSz)
4   NCTAsInStrips  $\leftarrow$  SSz * NS
5   S  $\leftarrow$  FlatId / SSz // strip ID
6   C  $\leftarrow$  FlatId % SSz // CTA in strip
7   flag1  $\leftarrow$  (NCTAsInStrips - 1 < FlatId)
8   UseSz1  $\leftarrow$  flag1 ? RSSz : TSz
9   (ID.xy, RevID.xy)  $\leftarrow$  getIDs(C, UseSz1, A)
10  UseDim  $\leftarrow$  isCM(A) ? ID.y : ID.x
11  flag2  $\leftarrow$  (D = RTS || isEven(InfoDim))
12  ID.xy  $\leftarrow$  flag2 ? ID.xy : RevID.xy
13  UseSz2  $\leftarrow$  isCM(A) ? TSz : SSz
14  SwizFlat  $\leftarrow$  S * UseSz2 + ID.y * gridDim.x + ID.x
15  SwizID.x  $\leftarrow$  SwizFlat % gridDim.x
16  SwizID.y  $\leftarrow$  SwizFlat / gridDim.x
17  return SwizID.xy

I : A: tileAxis, TSz: tileSize
O: SSz : strip size, NS: number of strips, RSSz : residual strip size
18 Procedure getStripInfo()
19   SSz  $\leftarrow$  isCM(A) ? (TSz * gridDim.y) : (TSz * gridDim.x)
20   NS  $\leftarrow$  isCM(A) ? (gridDim.x / TSz) : (gridDim.y / TSz)
21   RSSz  $\leftarrow$  isCM(A) ? (gridDim.x % TSz) : (gridDim.y % TSz)
22   return SSz, NS, RSSz

I : C: CTA index in strip, UseSz : size in CTAs to use, A : tileAxis
O: ID.xy : regular ID in strip, RevID.xy : ID in strip reverse order
23 Procedure getIDs()
24   ID.x  $\leftarrow$  isCM(A) ? (C % UseSz) : (C / UseSz)
25   ID.y  $\leftarrow$  isCM(A) ? (C / UseSz) : (C % UseSz)
26   RevID.x  $\leftarrow$  isCM(A) ? (UseSz - ID.x - 1) : (ID.x)
27   RevID.y  $\leftarrow$  isCM(A) ? (ID.y) : (UseSz - ID.y - 1)
28   return ID.xy, RevID.xy

```

ALGORITHM 2: Agents logic.

```

I : W : whole grid size in vCTAs
1 Procedure Agents()
2   NextID  $\leftarrow$  IsZero(ThreadInvCTA) ? GetNextID() :
   GetFromvCTALeaderThread()
3   if NextID > W then
4     return // exit recursion
5   end
6   InvokeOriginalShader(NextID)
7   Agents() // keep thread persistent

O: NextID : next vCTA ID
7 Procedure Steal()
8   if IsAnyOtherSMSLow() then
9     MySMID  $\leftarrow$  FindSlowSM()
10    NextID  $\leftarrow$  GetNextID()
11  else
12    if IsAnyOtherSMNonFast() then
13      MySMID  $\leftarrow$  FindNonFastSM()
14      NextID  $\leftarrow$  GetNextID()
15    else
16      NextID  $\leftarrow$  RasterizeRSG()
17    end
18  end
19  return NextID

I : W : whole grid size in vCTAs
O: NextID : next vCTA ID
15 Procedure RasterizeRSG()
16   NextID  $\leftarrow$  (W - RSGvCTAsNotDone--)
17   return NextID

O: NextID : next vCTA ID
18 Procedure RasterizePSG()
19   cond  $\leftarrow$  IsLeaderInOneTile(LocalvCTAID)
20   NextL1TileID  $\leftarrow$  cond ? GetNewOneTile() :
   GetFromOneLeadervCTA()
21   NextID  $\leftarrow$  ConvertTovCTAID(NextL1TileID)
22   return NextID

O: NextID : next vCTA ID
23 Procedure GetNextID()
24   LocalvCTAID  $\leftarrow$  PSGvCTAsDone[MySMID]++
25   cond  $\leftarrow$  ExceedsQuotaPerSM(LocalvCTAID)
26   NextID  $\leftarrow$  cond ? Steal() : RasterizePSG()
27   return NextID

```

4.3 Agents

Our third technique is based on *Agents* or persistent CTAs. This technique breaks free of the GE and facilitates near-perfect control over CTA scheduling to optimize for both L1 and L2 locality. This technique works by launching a small number of persistent CTAs enough to fill out the GPU [13]. These persistent CTAs or agents invoke the original 2D compute shader in an infinite loop. This loop takes care of rasterizing the original shader's work tokens, now called *virtual CTAs* (vCTAs). Execution breaks out of the loop when there are no more vCTAs left to rasterize. The number of vCTAs to be rasterized per agent is not fixed, neither is the mapping between vCTAs and SMs.

The *Agents* optimization organizes compute work into a three-level hierarchy. vCTAs is a 2D collection of threads. A 2D collection of vCTAs makes up an L1Tile. Finally, a 2D collection of L1Tiles makes up an L2Tile. A given 2D grid of compute worker threads is maximally tiled with L2Tiles. This portion of the grid is referred to as the **perfect sub-grid (PSG)**. Any remaining worker threads (making up the strip at the bottom and another on the right) are considered to be part of a **residual sub-grid (RSG)**. These terms are illustrated in Figure 8.

The *Agents* optimization takes nine input parameters. *vCTADim.xy* specifies a vCTA's size in terms of threads. *L1TileDim.xy* specifies L1Tile size in terms of the number of vCTAs along both axes. *L2TileDim.xy* specifies L2Tile size in terms of L1Tiles. In addition to the preceding three size controls, the *Agents* optimization takes two additional knobs per level of the hierarchy to specify traversal axis as well as how to locate the next element after hitting size limits. They take the same values as their *Swizzle* counterparts. Accordingly, *vCTAAxis*, *vCTANext*, *L1TileAxis*, *L1TileNext*, *L2TileAxis*, and *L2TileNext* make up the remaining six input parameters.

The implementation has separate global memory resident counters to work on the PSG and the RSG. For the PSG, it maintains an array of per-SM counters, `PSGvCTAsDone[Num_SMs]`, to keep track of the number of vCTAs rasterized by each SM and a scalar counter, `PSGL1TilesDone`, to keep track of the number of L1Tiles rasterized GPU-wide. These counters are initialized to zero. For the RSG, it just maintains a single counter, `RSGvCTAsNotDone`, that is initialized to the count of vCTAs in the RSG, which is calculated based on input parameters.

The outline of the *Agents* algorithm is given in Algorithm 2. In each iteration of the main loop, each persistent CTA rasterizes one or more vCTAs with appropriate, locality-friendly identifiers for its constituent threads. Initially, these identifiers belong to the PSG, and after the PSG has been fully rasterized, the RSG is rasterized. To determine thread identifiers, exactly one leader thread in a vCTA executes a `GetNextID()` function to obtain a flattened vCTA identifier within the current L1Tile. This process first increments the corresponding SM's `PSGvCTAsDone[MySMID]`, where `MySMID`, the SM identifier, is initialized from a special register exposed through internal intrinsics [38], but may get reassigned later in the algorithm due to work-stealing.

The algorithm then dynamically determines, on a first-come-first-serve basis, if the current vCTA is a leader vCTA by using simple modulo arithmetic (i.e., for SM m , if `PSGvCTAsDone[m]` modulo $(L1TileDim.x * L1TileDim.y)$ is zero, the current vCTA is considered a leader). If the current vCTA is a leader, it must also reserve a new L1Tile by atomically incrementing the global counter `PSGL1TilesDone`. The resulting flattened L1Tile identifier is saved in dedicated per-SM locations for communication to later-arriving non-leader vCTAs belonging to the same L1Tile. Using this flattened L1Tile identifier and *L2TileDim.xy*, individual threads of a vCTA then determine their final vCTA and thread identifiers efficiently with a handful of math operations. This information is communicated among threads of a vCTA either via intra-warp shuffles [38] in case of one-warp vCTAs or barrier-synchronized shared memory accesses for larger vCTAs.

Agents is able to vary vCTA and L1Tile sizes to flexibly target L1 or L2 locality with the desired warp scheduling freedom. We define **per-SM screen-space region (PSSSR)** as the L1Tile in terms of threads (i.e., $vCTADim.xy \times L1TileDim.xy$). The GPU-wide working set at any given point is the PSSSR size times the number of SMs. As we discuss later in the article, L1 hit-rate sensitive shaders prefer a larger PSSSR, whereas L2 hit-rate sensitive ones prefer to keep the GPU-wide working set small enough to fit within the L2 and thus prefer smaller PSSSRs. For a given PSSSR size, smaller vCTAs are expected to have more scheduling freedom and slightly poorer L1 locality than larger vCTAs.

The preceding functionality helps the *Agents* technique successfully hand out vCTA identifiers that are contiguous and within the desired tiled regions at both the L1 and L2 cache levels. Next, to achieve LB execution of vCTAs, *Agents* employs dynamic work-stealing. To explain its operation, we first define the notions of fast and slow SMs. Since all SMs work collectively on the PSG, we set a per-SM *FastThreshold* as $\frac{\text{TotalvCTAsInPSG}}{\text{NumSMs}}$. We empirically determined that 90% of *FastThreshold* can be considered as a good *SlowThreshold*. If the number of vCTAs processed by any SM x , given by `PSGvCTAsDone[x]`, reaches the *FastThreshold*, SM x is considered to be fast and is eligible for stealing from other SMs. If an SM's counter reaches the *SlowThreshold*, then that SM is considered to be *not* slow. To start with, all SMs are considered slow. To steal work, the fast SMs first look for

any slow SMs in the system and steal from them. If there are no slow SMs, they steal from an SM that has not been marked as fast yet. Stealing work from an SM y simply involves incrementing `PSGvCTAsDone[y]` and conditionally incrementing `PSGL1TilesDone` as described earlier. If no SMs are available to steal from, in applicable cases (i.e., compute calls that give rise to RSG based on their grid size and $L2TileDim.xy$), a fast SM proceeds to decrement the `RSGvCTAsNotDone` counter to grab a vCTA identifier from the RSG, which serves as a global shared work pool.

Atomic updates to the preceding counters are not globally ordered or synchronized. However, any resulting data races in updating these counters during work-stealing do not lead to incorrect rasterization. Data races due to multiple stealer SMs atomically incrementing a slow SM's `PSGvCTAsDone` counter can cause the latter's counter to exceed the *SlowThreshold*. But the stealer SMs will still go ahead and steal work from the latter, a non-fast SM now, even though other slow SMs may be available. The *Agents* algorithm automatically recovers from this momentary violation of its heuristic bounds since future work-stealing attempts will observe the preceding SM as a non-fast SM and proceed to steal work from other slow SMs. Similarly, when one or more stealer SMs' concurrent atomic increments cause a stealee SM's counter to exceed the *FastThreshold*, the algorithm moves the stealer SM(s) to the RSG, making this type of data race irrelevant as well. Note that the preceding discussion on data races pertains specifically to the global memory counters used in the *Agents* algorithm. As explained in Section 2, all the CTA scheduling techniques discussed in this article, including *Agents*, are in conformance with the Direct3D specification and thus do not impact updates to application data structures.

We measure about 5% performance overhead at the kernel level from the preceding scheduling logic in directed testing. The preceding overhead is dominated by time spent on atomic updates of the various global memory resident counters. That said, the overhead is more than amortized by the concerted L1 and L2 locality improvements from *Agents*, as evidenced by the speedups in Section 6.

5 METHODOLOGY

Table 1 provides details of the gaming applications evaluated in this article. We use single-frame traces, captured by other teams within NVIDIA, from either a built-in benchmark (where available) or from actual gameplay of each gaming application. These are captured with an internal frame-capture tool, similar to publicly available tools like Renderdoc [18] and Nsight [37], among others. These single-frame captures, called *APICs*, contain all the information needed to replay a game frame—that is, both the API sequence (including all relevant state) as well as shader programs used in individual calls. We use a total of 10 APICs for our evaluation.

On average, we target a little more than 70% of all 2D compute shaders that are latency or bandwidth limited from Figure 1 for our evaluation. This amounts to 20% of average frame time.⁴ We use automatic prototyping tools to perform shader and API modifications to transform baseline APICs offline to create new, suitably modified APICs for each configuration of all our locality-improvement techniques. Shader modifications are performed in high-level assembly such as Microsoft's DXBC or DXIL languages. Correctness is verified by ensuring that the output image matches a reference image for each application's APIC as well as ensuring functional statistics like total number of texture lookups remain the same across baseline and modified runs.

We perform our experiments on a GPU from the Turing family, RTX 2080, and a GPU from the latest Ampere family, RTX 3080. We configure GPU core and DRAM frequencies such that both GPUs have a similar compute-to-memory-bandwidth ratio (i.e., $\frac{\text{number of SMs} \times \text{core frequency}}{\text{DRAM bandwidth}}$) to

⁴Recall from Section 1 that 2D compute calls make up 40% of the average frame time and 72% of these calls are latency or bandwidth limited 2D calls. Thus, targeted region is 70% of 40% $\times$ 72%, which comes to 20%.

Table 1. Gaming Applications Evaluated in This Article

Application	Short Name	Resolution	API	#Shaders Targeted
Crysis Remastered	Crysis	1,440p	dx11	1
Microsoft Flight Simulator	FltSim	4K	dx11	1
Control	Control	4K	dx12	1
CyberPunk 2077 RT OFF	CP77	4K	dx12	1
Metro DLC	Metro	1,440p	dxr1.0	1
BattleField V	BFV	4K	dxr1.0	1
Watch Dogs Legion	WDL	1,440p	dxr1.0	2
CyberPunk 2077 RT ON	CPRT	4K	dxr1.0	2
Fortnite	Fort	1,440p	dxr1.0	1
Minecraft	Mine	1,440p	dxr1.1	3

Table 2. Configuration Details of Our Test GPUs

GPU	RTX 2080	RTX 3080
Core frequency setting	1,650 MHz	1,605 MHz
Number of SMs	46	68
Max L1 cache capacity (per SM)	64 KB	128 KB
L2 cache capacity (per GPU)	4 MB	5 MB
Max L2-DRAM bandwidth (per GPU)	448 GB/s	640 GB/s

facilitate easy comparison of the performance of our software techniques across the two GPUs. Table 2 summarizes the salient configuration details of our test GPUs. More information about these GPUs can be obtained from their respective product whitepapers [36, 40]. We measure GPU-only performance with accurate in-house profiling tools, after locking clocks as described earlier and power state to production active gameplay settings. GPU-time measurements for all techniques subsume their respective overheads. All runs use a recent internal branch of NVIDIA's GeForce Game Ready driver and use production settings for driver and compiler optimizations.

6 EVALUATION

We note a few caveats before presenting the results. First, for simplicity, we do not consider any 2D compute shaders using cooperative shared memory communication in our evaluation. This allows us to flexibly vary CTA or vCTA sizes without worrying about preserving correct shared memory semantics. Although such cases can be handled by our techniques with additional engineering effort, they are relatively rare in gaming applications and thus we do not handle them for this work.⁵ Second, even though our techniques target only select 2D compute kernels of the chosen game frames, we present full-frame performance speedups from these techniques in Figure 9, as that is the primary yardstick for gaming performance. We summarize average kernel-level metrics at the end of this section. Third, in addition to speedups for individual configurations, each technique's graph also includes a BestOf bar that conveys the best speedup over all configurations for each APIC for a given technique. Last, note that the y -axis ranges of these graphs vary across techniques.

6.1 LargeCTAs

Figure 9(a) and (b) show the performance of different large CTAs with respect to each application's baseline performance across RTX 2080 and RTX 3080, respectively. Most baseline 2D compute shaders use an 8×8 CTA size—that is, 64 threads organized as 8×8 and spanning two

⁵Although *Swizzle* will not require any additional changes to support shaders with native shared memory usage, *LargeCTAs* and *Agents* will require maintaining multiple shared memory allocations corresponding to the multiple original logical CTAs that may be part of a single, larger physical CTA, and redirecting shared memory requests to appropriate allocations.

(a) RTX 2080: *LargeCTAs*. (b) RTX 3080: *LargeCTAs*. (c) RTX 2080: *Swizzle*. (d) RTX 3080: *Swizzle*. (e) RTX 2080: best *LargeCTAs* and *Swizzle*. (f) RTX 3080: best *LargeCTAs* and *Swizzle*. (g) RTX 2080: *Agents*. (h) RTX 3080: *Agents*.

Fig. 9. Frame-level performance speedups for all techniques across both RTX 2080 and RTX 3080 GPUs.

32-thread warps. Therefore, on an ad hoc basis, we consider any CTA size that is 16×16 or larger as an application of the *LargeCTAs* technique. Besides testing two square-shaped large CTA sizes (16×16 and 32×32), we evaluate an “**isoreg**” setting that uses a large CTA size based on baseline shader’s register usage. For example, if baseline shader used 80 registers, which would give rise to 24 concurrent warps per SM [39], the corresponding **isoreg** *LargeCTAs* experiment would use a 32×24 CTA size.

WDL sees a big slowdown with 16×16 CTA size. That is because its baseline shader uses a 32×32 CTA size and is very sensitive to L1 cache locality. Thus, a smaller CTA size like 16×16 slows it down. In contrast, on Crysis larger CTA sizes results in slowdowns for a reason not related to locality. The targeted Crysis shader uses a 16×16 CTA size for its baseline and implements a ray-tracing technique called *SVOGI* [50], which exhibits significant amounts of control flow divergence. Large CTA sizes for such highly divergent shaders suffer from the out-of-order warp completion issue described in Section 4.1. BFV, Metro, and Control respond positively on both GPUs. Mine slows down with 32×32 on 3080 due to increased out-of-order warp completions on 3080 than 2080 on one of the targeted shaders.

6.2 Swizzle

We evaluate a total of 12 configurations for both chips. Since the performance difference between individual configurations is relatively small, to save space, we present only six configurations’ results per graph. For both chips, we evaluate ($tileAxis = \{\text{colmajor}, \text{rowmajor}\} \times (tileSize = \{8, 16, 32\}) \times (nextCtaDir = \{\text{return-to-start (rts)}, \text{boustrophedonic (bou)}\})$). For RTX 2080, we report **colmajor** results, and for RTX 3080, we report **rowmajor** results in Figure 9(c) and (d), respectively. The BestOf bars for both chips capture the best performance over all 12 configurations. The CTA size was maintained the same as the respective baseline compute shaders. The only change was the application of a specific *Swizzle* configuration. As can be seen from the figures, this simple technique is very effective across several applications, with BFV and Metro showing big gains. Working with NVIDIA’s DevTech engineers, we have successfully integrated *Swizzle* into a few shipping games [6]. Although, on an average, the different *Swizzle* configurations perform about the same, individual APICs show sensitivity to specific configurations. This is understandable since different *Swizzle* configurations react differently to individual 2D compute shaders’ filter regions.

Recall that whereas *LargeCTAs* targets L1 locality, *Swizzle* targets L2 locality. The performance of these techniques for a given application give us an idea of the application’s sensitivity to L1 or L2 hit-rates. For example, Control shows 3.5% and 2.4% BestOf speedups from *LargeCTAs* on 2080 and 3080, respectively. Interestingly, it sees a 3.5% speedup from *Swizzle* as well on 2080 but experiences a minor slowdown on 3080. This is because L2 hit-rate matters less on 3080 (Ampere), which provides $2\times$ larger cacheable L1 than 2080 (Turing) for compute shaders not using shared memory, resulting in Control shifting from being jointly sensitive to L1 and L2 hit-rates on 2080, to being less sensitive to L2 hit-rate on 3080. Crysis, which did not benefit from *LargeCTAs*, responds well to *Swizzle* on both GPUs, indicating L2 sensitivity. BFV and Metro benefit from both techniques, although speedups from *Swizzle* are more prominent, again indicating L2 sensitivity.

6.3 Combining *LargeCTAs* and *Swizzle*

Since the *LargeCTAs* and *Swizzle* techniques are orthogonal in their impact and operation, we next study the performance of jointly applying *LargeCTAs* and *Swizzle*. Since studying a full cross-product of *LargeCTAs* sizes and *Swizzle* configurations can lead to a combinatorial explosion, we restrict ourselves to studying the impact of different *LargeCTAs* configurations mentioned earlier in combination with the best-known *Swizzle* configuration for each targeted compute shader. The results of this study are given in Figure 9(e) and (f) for both GPUs. Control, and

to a small extent, BFV and Metro, reap the combined benefits of both *LargeCTAs* and *Swizzle*. For other applications, the performance of the combined technique mostly tracks *Swizzle*-only performance. This enables the BestOf average of *LargeCTAs* plus *Swizzle* to be better than either of those techniques individually.

6.4 Agents

Next, we measure the performance of the *Agents* technique across the following configurations: ($vCTADim.xy = 8 \times 4$) $\times$ ($L1TileDim.xy = \{1 \times 1, 2 \times 4, 4 \times 8, 8 \times 8, 8 \times 16\}$) and ($vCTADim.xy = 32 \times 32$) $\times$ ($L1TileDim.xy = \{1 \times 1, 2 \times 1, 2 \times 2\}$), for a total of eight basic configurations. For each of these configurations, we searched the optimization space across other parameter settings, namely $L2TileDim.xy$, the traversal axes and next element direction parameters for vCTAs, L1Tiles, and L2Tiles. The results for each of the eight basic configurations correspond to the best identified setting for these other parameters on a per APIC basis. All configurations use 32×32 persistent CTAs, regardless of the original CTA size or the vCTA size targeted. This means all original shaders will have to be register-allocated in 64 registers or less, spilling if needed, to enable 32 active warps per SM [39]. Figure 9(g) and (h) present performance results for *Agents*.

Agents is able to effectively target both L1 and L2 locality as can be seen from good speedups on BFV, Metro, and Control. Although most applications' performance is comparable between *Agents* and *LargeCTAs+Swizzle*, the former's BestOf average ends up better than the latter, thanks mainly to CrysIs. *Agents* uses its flexible vCTA sizing to perform CrysIs's SVOGI kernel work in units of 8×4 vCTAs and 1×1 L1Tiles, ensuring maximal warp occupancy in the face of divergent behavior while also getting *Swizzle*-like L2 locality. Larger vCTAs do not work well for CrysIs and similar applications because they are subject to barrier-synchronization penalties when exchanging vCTA identifier information via shared memory. *Agents* for CrysIs performs better on RTX 3080 than on RTX 2080 due to RTX 3080's larger L2 cache allowing *Agents*'s tiled scheduling logic to retain more of its dynamic working set on-chip, leading to better performance on CrysIs's L2-sensitive shader.

On CP77, CPRT, and WDL, *Agents* is able to realize good performance gains by breaking free of NVIDIA GPUs' maximum CTA size of 1,024 threads by using 32×32 vCTAs and multi-vCTA L1Tiles (2×1 or 2×2) to effectively target 64×32 or 64×64 screen-space regions from a single SM while concurrently shaping active L2 working set to tile-shaped regions made up of these large L1Tiles. Fort is interesting in that it is L1 sensitive and prefers a large PSSSR (64×32), but prefers it as 8×8 L1Tiles of 8×4 vCTAs rather than 2×1 L1Tiles of 32×32 vCTAs, to benefit from the good warp scheduling freedom of 8×4 vCTAs. For most applications, depending on whether an application is sensitive to L1 or L2 hit-rate, its BestOf bar can be seen to match either a large or small PSSSR, respectively.

Next, to understand the impact of *Agents*'s work-stealing features, we disable cross-SM work-stealing and uniformly apportion the RSG, where applicable (6 out of 10 applications), among all agents instead of using the RSG as a global shared work pool. Table 3 presents the performance speedups of BestOf *Agents* with and without work-stealing features over baseline on RTX 3080 in rows labeled *A* and *A – WS*, respectively. Overall, *A – WS* improves performance by 3.8% over baseline compared to the 4.7% upside shown by the full *Agents* algorithm on RTX 3080. Thus, on an average, work-stealing features contribute an absolute 0.9% or a relative 19.1% to *Agents*'s frame-level speedup.⁶

⁶Note that even though both *A – WS* and the naive rasterization experiment in Figure 5 have load-balancing features turned off, *A – WS* is still 3.8% faster than the baseline on the average compared to the 9% average slowdown seen in the latter, thanks to *Agents*'s tiling-aware rasterization dramatically reducing overall CTA latencies and the magnitude of their standard deviations in the *A – WS* runs. These results reinforce the importance of locality-aware CTA scheduling.

Table 3. *Agents*: Impact of Work-Stealing Features on RTX 3080 (in %)

Configuration	BFV	Control	CP77	CPRT	Crysis	FltSim	Fort	Metro	Mine	WDL	Mean
A	7.7	2.7	3.3	1.5	14.0	0.4	3.8	6.5	3.9	3.2	4.7
A – WS	7.3	2.8	2.5	1.4	13.9	0.4	2.3	6.3	2.4	-1.2	3.8

Row A: Speedup of BestOf *Agents* over baseline; Row A – WS: Speedup of BestOf *Agents* without work-stealing features over baseline.

Table 4. Summary Points and Statistics

Stat	GPU	L	S	C	A
Frame-level BestOf speedup	2080	1.4%	2.9%	3.2%	3.5%
	3080	1.4%	2.4%	3.0%	4.7%
Kernel-level BestOf speedup	2080	10.2%	23.7%	25%	27%
	3080	11.3%	19.3%	25.4%	31.2%
Kernel-level L1-L2 bandwidth reduction for BestOf configs	2080	16.3%	2.8%	11.5%	5.6%
	3080	14.6%	2.7%	18.8%	22.9%
Kernel-level L2-DRAM bandwidth reduction for BestOf configs	2080	13.4%	43.2%	40.9%	37.0%
	3080	9.8%	36.1%	38.8%	31.7%

L, *LargeCTAs*; S, *Swizzle*; C, L+S combined; A, *Agents*.

6.5 Putting It All Together

Table 4 summarizes key performance statistics from our cache locality optimization space exploration. We report average baseline-normalized bandwidth reduction between the L1 and the L2 (L1-L2) and between the L2 and the DRAM (L2-DRAM) for the BestOf configurations in the preceding table for both GPUs, while presenting application-wise BestOf speedups and bandwidth breakdowns in Figure 10 at the shader (or equivalently kernel) level for just RTX 3080 to save space.⁷ Some application kernels record speedups even with negative bandwidth reductions (i.e., bandwidth increase) at either the L1-L2 level or the L2-DRAM level. The reason for this will become apparent later in this section.

Overall, the BestOf results for *LargeCTAs+Swizzle* and *Agents* look promising since both these techniques capture L1 and L2 locality. *Agents* is the better of the two due to its ability to flexibly size vCTAs and L1Tiles to target the desired PSSSR size. However, as seen in previous sections, individual applications respond positively to certain settings of these techniques and not so positively to others. To practically realize these techniques' BestOf speedups in a production driver, good heuristics are necessary to automatically determine optimal kernel-specific settings for these techniques. For example, for *Agents*, it is necessary to determine *a priori* whether a 2D compute shader is sensitive to L1 hit-rate or L2 hit-rate and accordingly determine PSSSRs. Next, we describe a feedback-driven approach to determine L1 or L2 hit-rate sensitivity and build our heuristics on top of that.

6.5.1 Heuristic Development. We sketch a potential heuristic based on the absolute baseline bandwidth numbers in Table 5. We observe that application kernels fall into two categories based on the value of L2-DRAM bandwidth normalized to L1-L2 bandwidth, a proxy for the baseline's L2 miss-rate (columns marked *R* in the table). Admittedly overfitting to our small suite of applications across our two test GPUs, we find that an ad hoc *R* threshold of 15% appears to neatly

⁷We note a few caveats. First, bandwidth refers to raw byte count here. Second, for Mine, CPRT, and WDL, where we target multiple kernels, kernel-level speedup and bandwidth are calculated by aggregating targeted kernels' runtimes and bandwidth. Third, the bandwidth reduction sub-bars are not additive since they are normalized to different baseline statistics; the stack just makes for easy visualization.

Fig. 10. Kernel-level performance (top) and bandwidth reduction (bottom) on RTX 3080 for BestOf configurations. L, *LargeCTAs*; S, *Swizzle*; C, L+S combined; A, *Agents*.

split our application kernels into two groups—either L1 sensitive or L2 sensitive, which matches our experimental results quite well. Such a heuristic will categorize kernels with a high R -value ($\geq 15\%$), namely BFV, Crysis, Metro, and Mine, in blue text, as L2-sensitive, and the remaining six applications that are lower than the 15% threshold, in red text, as L1-sensitive.

Accordingly, we evolve a heuristic for *Agents* by picking small PSSRs for the L2-sensitive applications and big PSSRs for the L1-sensitive ones. The equivalent for *LargeCTAs+Swizzle* would be to pick larger *LargeCTAs* for L1-sensitive applications (either **isoreg** or *32x32*) and prefer smaller *LargeCTAs* for L2-sensitive applications (**16x16**). Since we have a full cross-product of runtime data for all settings of each technique for all applications, application of a heuristic setting here simply means selecting the speedup corresponding to that setting for all applications and averaging them.

Results for various heuristic settings for *Agents* are presented in Figure 11. The first cluster, BestOf (fixed L2-A-D), shows the performance impact of fixing *L2TileDim.xy* to 256×128 threads, *vCTAAxis*, *L1TileAxis*, and *L2TileAxis* to **colmajor**, and *vCTANext*, *L1TileNext*, and *L2TileNext* to **boustrophedonic**, while allowing the other parameters to vary and calculating the resulting best performance. We see that fixing L2Tile size, the axis parameters, and the next tile direction parameters results in speedups that are 90.3% and 93.6% of the BestOf speedups across RTX 2080 and RTX 3080, respectively.

For the remaining clusters, we maintain the preceding fixed parameterization for the aforementioned seven parameters and evaluate the impact of vCTA and L1Tile sizes. These clusters explore sensitivity to both generic PSSSR sizes, with internal flexibility in vCTA and L1Tile sizes, as well

Table 5. Baseline Bandwidth Demand

	RTX 2080			RTX 3080		
App.	P	Q	R	P	Q	R
BFV	2.73	1.72	63%	2.94	1.4	48%
CP77	15.35	0.14	1%	14.39	0.13	1%
CPRT	8.08	1.09	13%	7.69	0.88	11%
Control	6.91	0.71	10%	5.79	0.69	12%
Crysis	3.16	1.38	44%	3.95	1.31	33%
FltSim	3.02	0.01	0%	2.76	0.01	0%
Fort	11.39	0.71	6%	9.93	0.56	6%
Metro	2.1	1.11	53%	2.36	0.93	39%
Mine	7.25	1.18	16%	6.68	0.97	15%
WDL	3.14	0.34	11%	3.19	0.27	8%

$$P = \text{L1-L2 in GB}, Q = \text{L2-DRAM in GB}, R = \frac{Q \times 100\%}{P}.$$

Fig. 11. Heuristic settings search for *Agents*.

as fixed vCTA and L1Tile sizes making up PSSSRs. The three clusters prefixed with “PSSSR” in the figure correspond to generic PSSSR settings, whereas the following clusters correspond to concrete vCTA and L1Tile settings that make up the corresponding PSSSRs.

In the first category, we find that a (bigPSSSR, smallPSSSR) combination of $(32 \times 32, 8 \times 4)$ achieves about 83% of the BestOf speedups for both RTX 2080 and RTX 3080, whereas a more relaxed setting captured by $(\geq 32 \times 32, < 32 \times 32)$ gets about 91% of the BestOf speedups on both GPUs. These results can help ninja game developers, who go to great lengths to squeeze out additional performance from their shaders, prune their optimization search space and achieve close to BestOf speedups, provided they have knowledge of R -values for a given shader.

However, the preceding information is not enough for automation in a GPU driver because PSSSR settings can be arrived at in different ways. For example, a PSSSR setting of 32×32 can be formed by 32×32 vCTAs making up 1×1 L1Tiles or 8×4 vCTAs making up 4×8 L1Tiles. Our targeted kernels are sensitive to the exact vCTA and L1Tile size. Thus, for potential future automation in a driver, explicit settings for vCTA and L1Tile sizes making up a PSSSR setting are needed. From Figure 11, we see that $(32 \times 32^2 \times 1, 8 \times 4^1 \times 1)$ and $(32 \times 32^2 \times 1, 8 \times 4^2 \times 4)$ are the best settings for RTX 2080 and RTX 3080, respectively. Each setting is specified as $(vCTADim.xy * L1TileDim.xy)$. These settings achieve 79% of the BestOf speedups for *Agents* for both the GPUs, amounting to 2.7% and 3.7% frame-level speedups on RTX 2080 and RTX 3080, respectively. For *LargeCTAs+Swizzle* combined, we found that preferring **isoreg** for L1-sensitive applications and 16×16 for L2-sensitive applications obtained 94% and 92% of the BestOf speedups for that technique, amounting to 3% and 2.7% frame-level speedups on RTX 2080 and RTX 3080, respectively.

The preceding exercise shows that an R -values based automatic framework along with the heuristic settings identified earlier can potentially realize most of the BestOf speedups. Both *Agents* and *LargeCTAs+Swizzle* show good frame-level speedups even with heuristically derived fixed parameterizations of the respective techniques.

6.5.2 What Makes R -values Based Heuristics Work? Intuitively, since texture lookups return in order from the texture unit (L1) to the SM, head-of-line blocking due to L1 misses means the steady state texture L1 lookup latency subsumes the average L2 lookup latency. Since R approximates the baseline’s L2 miss-rate, it is a good indicator of average L2 lookup latency and, therefore, average texture L1 lookup latency. The higher the R -values, the higher is the texture L1 lookup latency. In such scenarios, improvement in L2 locality can help improve performance. However, a low R -values means the baseline’s L2 performance is good to begin with and so L1 hit-rate matters more for performance.

Going back to Figure 10, *Crysis* sees a speedup despite a negative reduction in L1-L2 traffic due to it being L2 sensitive. Likewise, *FltSim* performs well despite negative L2-DRAM bandwidth savings due to it being L1 sensitive. For *Mine*, an L2-sensitive case that speeds up, our analysis of performance counters and shader codes suggests that the L2-DRAM traffic increase in the *Agents* run is due to spill traffic brought about by increased liveness from *Agents*-related variables and not due to texture read traffic. On RTX 2080, even though *LargeCTAs+Swizzle*’s average bandwidth savings is better than the average bandwidth savings with *Agents*, *Agents* still has a slight performance edge. This seeming anomaly can be understood from the recently introduced notions of L1 and L2 sensitivity and RTX 2080 bandwidth savings data available with us (but not included for want of space). *Agents* on RTX 2080 shows negative L1-L2 bandwidth savings on all four L2-sensitive applications, whereas *LargeCTAs+Swizzle* shows healthy positive L1-L2 bandwidth savings by picking 16×16 or higher CTA sizes. Although this does not affect overall performance (since these applications are L2-sensitive), it makes the L1-L2 average bandwidth savings number for *Agents* look

bad. A similar story repeats in the other direction too—that is, *Agents* showing lower L2-DRAM bandwidth savings than *LargeCTAs+Swizzle* on three L1-sensitive applications, namely *FltSim*, *Fort*, and *WDL*, which causes the *average* bandwidth savings numbers to look bad but does not affect performance.

Thus, the notions of L1 and L2 sensitivity are useful both for heuristic design as well as explaining tricky performance behavior. Building on this approach, and possibly taking into account other characteristics, it is likely one might be able to devise more robust and better-performing heuristics. Optionally, iterative compilation or autotuning approaches [1, 3, 9, 20, 52] can be employed to achieve close to BestOf performance by walking through the parameter search space heuristically across frames in live gameplay and discovering optimal parameter settings for our cache locality techniques. Autotuning can also be used to avoid introducing regressions when applying our CTA scheduling techniques generically across all shaders in a frame. However, such explorations are beyond the scope of this work, which primarily seeks to demonstrate that significant cache locality improvements are possible atop the GEs in shipping GPUs through careful CTA scheduling. We believe the results in this section adequately accomplish that goal.

7 DISCUSSION

Despite the NVIDIA Turing architecture packing in useful architectural enhancements to improve latency-limited performance [43], we see that 2D compute shaders are mostly latency-limited on both Turing and Ampere GPUs. We have shown that it is possible to improve the performance of latency and bandwidth-limited compute shaders on these GPUs through software-orchestrated cache locality optimizations.

The GE was designed to ensure maximal dynamic CTA occupancy across all SMs. The current GE rasterization policy helps accomplish that goal handsomely. However, although the GE has retained that basic behavior over the years, on-chip cache capacities have increased over the generations. In NVIDIA's GeForce series, Ampere (2020 release) increases cacheable L1 and L2 capacities by up to 2× and 1.25×, respectively, over Turing (2018 release), which itself increases cacheable L1 and L2 capacities by up to 2.7× and 2×, respectively, over Pascal (2016 release).

From the data in Table 5, we can calculate that average baseline L1-L2 and L2-DRAM bandwidth demand reduced by 1% and 12.9%, respectively, from RTX 2080 to RTX 3080, thanks to the cache capacity increase from Turing to Ampere. Compared to that, all software techniques reduce average L1-L2 and L2-DRAM bandwidth demand by much higher margins on RTX 2080 as seen in Table 4, highlighting the power of locality-aware CTA scheduling vis-à-vis cross-generational cache capacity increase. These scheduling techniques are just as effective on RTX 3080 and able to further reduce bandwidth demand over and above the cross-generational improvement. Thus, we argue that to make effective use of increased cache capacity, it is all the more important to devise cache locality-aware CTA scheduling mechanisms on top of the GE.

Our best-performing technique, *Agents*, achieves BestOf speedups of 3.5% and 4.7% on RTX 2080 and RTX 3080, respectively. To contextualize these speedups, consider the fact that it is not uncommon for GPU hardware refreshes within an architecture family to yield similar average speedups as our techniques on gaming applications while using more processing cores, memory bandwidth, and power [46, 49]. The proposed software techniques, despite their overhead, can help gamers squeeze more performance out of their current GPUs and provide frame-rate boosts that approach those from simple GPU hardware refreshes but without these software optimizations.

8 RELATED WORK

We first quantitatively compare the performance of our techniques with two important prior approaches—**block CTA scheduling (BCS)** [23] and locality-aware clustering [24], both of which

Fig. 12. Comparison with BCS [23] and the locality-aware clustering techniques of Li et al. [24] on RTX 3080.

have successfully sped up GPGPU compute applications through cache locality improvements. BCS is a hardware technique that allocates two consecutive row-major CTAs to the same SM to increase L1 data reuse. This is very effective for row-major access patterns. We model BCS in software as a special case of *LargeCTAs* by simply doubling the baseline CTA size along the x -direction. For baseline CTA size of 32×32 , since current GPU hardware limitations prevent us from creating larger CTAs, we mimic BCS by looping over the baseline code twice and suitably calculating thread and CTA identifiers programmatically. In all cases, we reduce grid dimension by $2 \times$ in the x -direction.

We also implement **redirection-based clustering (RD)** and **agent-based SMID clustering (CLU)** techniques for row-major and column-major clustering as described in the work of Li et al. [24]. Since Li et al. do not discuss implementation details of tiled clustering but indicate it as a possibility, we implement a maximal tile clustering strategy for both RD and CLU, whereby the entire 2D screen-space is maximally partitioned into as many tiles as SMs in RTX 3080 (which is 68) and any residual CTAs are handled via row-major clustering. We report the BestOf numbers across the three different clustering possibilities for each technique for a given application. We study the impact of clustering techniques in isolation and do not evaluate it in combination with orthogonal approaches like prefetching, throttling, or cache bypassing like Li et al. do in their work.

Figure 12 compares speedups of BestOf of *LargeCTAs*, *Swizzle*, *LargeCTAs+Swizzle*, and *Agents*, with BCS, BestOf of redirection-based clustering (RD), and BestOf of SMID clustering (CLU) over baseline on RTX 3080. BCS gives an average performance improvement of 0.62% over baseline, compared to an average of 1.4% for our equivalent technique, *LargeCTAs*. RD and CLU, unfortunately, do not perform well on our applications.⁸ At the L1-L2 and L2-DRAM interfaces, average shader-level bandwidth savings for RD is -2% and -98% , respectively, whereas for CLU it is $+23\%$ and -83% , respectively. Both clustering techniques significantly worsen L2-DRAM bandwidth due to not managing L2 locality explicitly and causing temporally close by L2 lookups to go to disjoint regions of a grid's screen-space. Although such access patterns are harmless for kernels whose working sets fit within the L2, they thrash the L2 in gaming shaders which have significantly larger working sets.⁹ RD experiences a smaller average frame slowdown of 3.4% due to it benefiting from the GE's LB scheduling, whereas CLU, despite improving L1-L2 bandwidth, sees a 9% average slowdown, due to imbalanced execution stemming from a static, uniform distribution of work tokens across SMs (same fate as Figure 5). In contrast, our equivalent techniques *Swizzle*

⁸For CLU, we had to barrier-synchronize the warps of an agent for each iteration of its outer loop to prevent early-exiting virtual warps from moving ahead to subsequent iterations and thrashing the L1. This helped improve CLU from a 13% average slowdown to the 9% slowdown seen in Figure 12.

⁹For example, multiple 1,440p or 4K textures are accessed by most compute shaders, compared to the single 512×512 or smaller test image inputs from the CUDA SDK [41] for IMD and DCT, the top-performing kernels for CLU [24].

and *Agents* work well in gaming applications. *Swizzle* targets only L2 locality very successfully for tiled texture accesses in 2D compute shaders. *Agents* uses a combination of sophisticated tiling and load-balancing strategies to effectively target L1 and L2 cache locality for 2D compute calls in gaming applications.

The rest of this section compares our work qualitatively with other relevant prior work.

8.1 Software Scheduling Techniques

Researchers have parallelized array-intensive loops in a cache hierarchy aware fashion on multi-core processors using compile-time mappings of iterations to cores [17, 27]. In contrast, CTA-to-SM mappings in our techniques are either implicit (*LargeCTAs*) or derived dynamically through low-overhead means (*Swizzle* and *Agents*). Software approaches that duplicate and re-layout data arrays to improve cache locality in applications with irregular memory accesses [22, 57] are not suited to real-time computer graphics.

Persistent CTAs have been previously used in GPU workloads to enable LB execution [2, 8], uberkernel support [48], and improved global synchronization [55], among other uses [13]. Wu et al. [53] implement a flavor of persistent CTAs to improve cache locality across kernel launches. To the best of our knowledge, our *Agents* implementation, driven by insights gained from *LargeCTAs* and *Swizzle*, is unique in its simultaneous pursuit of L1 and L2 cache locality as well as LB execution through work-stealing within a single kernel.

8.2 Hardware Enhancements for GPU Cache Locality

The hardware approach of Chen et al. [7] steers CTA to SMs based on locality estimates derived from executing address computation kernels extracted from the original GPGPU programs. Such address kernel extraction is very difficult in compute shaders in games, since it might involve dependent memory lookups and control flow, including loops.

Nah et al. [34] show improved cache locality in simulations for virtual reality stereo rendering by mapping tiles for left and right eyes to the same shader core. Virtual reality stereo effectively renders twice near-identical images, with only the left and right eyes' viewpoints differing between the two rendering calls. This is different from and orthogonal to our effort here to improve the efficiency of individual compute shader calls. Other hardware enhancements have targeted cache locality across kernel launches for parent-child kernels [51] or generic dependent kernels [15]. Several researchers have explored cache bypassing to improve GPU cache locality [21, 25, 26, 47, 56, 58]. Others have studied the impact of warp throttling on locality [14, 24]. OWL enhances the SM's warp scheduler to be CTA-aware and makes it prefer warps from a subset of the inflight CTAs repeatedly to improve intra-CTA locality [16]. Cooperative caching network (CCN), a ring network among L1 caches of a GPU, was proposed to reduce L2 bandwidth demand [11]. These techniques are orthogonal to our work on CTA scheduling and could co-exist with CTA scheduling enhancements to exploit inter-CTA cache locality.

9 CONCLUSION

The GE's LB row-major rasterization order maximizes GPU-wide CTA occupancy in the face of highly variable CTA latencies while ensuring design simplicity. However, this default rasterization behavior leads to less-than-ideal cache locality for tiled texture access patterns of 2D compute shaders. We demonstrate that simple, low-overhead software techniques can be used on top of the current GE design to improve cache locality and speed up game frames by up to 3.5% and 4.7% on average on RTX 2080 and RTX 3080, respectively, through significant bandwidth reduction for targeted kernels at both the L1-L2 and L2-DRAM interfaces. On our suite of applications, the bandwidth reduction from our CTA scheduling techniques on RTX 2080 is much more than the

reduction brought about by Turing-to-Ampere cross-generational cache capacity increase, highlighting the importance of locality-aware CTA scheduling.

Although we have not studied the efficacy of our CTA scheduling techniques in applications beyond gaming in this article, we conjecture that any algorithm exhibiting inter-CTA locality due to tiled access patterns will likely benefit from our techniques (e.g., image processing with convolution filters). Although more research is needed to validate that conjecture, we identify the following generic insights from our work that may be useful across application domains, namely:

- It may be profitable to simultaneously optimize for both L1 and L2 locality (as demonstrated by *LargeCTAs+Swizzle* and *Agents*) rather than focusing exclusively on one or the other. For such techniques, our Rvalue based L1 vs L2 sensitivity categorization may help effectively prune the CTA scheduling parameter search space and guide the development of effective heuristics.
- For applications whose CTAs exhibit variable latencies, any locality-aware CTA scheduling technique should either make use of the GE's (or equivalent hardware's) load-balancing support (like *Swizzle* and *LargeCTAs* do) or resort to work-stealing like approaches (as *Agents* does) to strive for improved cache locality while ensuring LB execution.

Going forward, we plan to explore both static heuristics as well as online autotuning approaches in our production driver to deliver close-to-BestOf speedups in live gameplay environments. We also plan to study the energy impact of these cache locality improvement techniques. A third potential future direction is to leverage insights gained from this work to pursue GE design enhancements to jointly optimize for GPU-wide CTA occupancy as well as cache locality.

ACKNOWLEDGMENTS

We thank our article's anonymous reviewers and Emmett Kilgariff for their valuable feedback. Michael Fetterman encouraged us to explore the use of persistent CTAs for locality-aware scheduling. Many thanks to NVIDIA's Devtech and QA teams for providing the application traces used in this article, and the entire Applied Architecture team for their support during this work.

REFERENCES

- [1] B. Aarts, M. Barreleteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, et al. 1997. OCEANS: Optimizing com-ilers for embedded applications. In *Proceedings of the 3rd International Euro-Par Conference*. 1351–1356.
- [2] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)*. 145–149. <https://doi.org/10.1145/1572769.1572792>
- [3] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys* 51, 5 (Sept. 2018), Article 96, 42 pages. <https://doi.org/10.1145/3197978>
- [4] Ali Bakhoda, George Yuan, Wilson Fung, Henry Wong, and Tor Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. 163–174. <https://doi.org/10.1109/ISPASS.2009.4919648>
- [5] Louis Bavoil. 2019. The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload. Retrieved August 25, 2021 from <https://devblogs.nvidia.com/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/>.
- [6] Louis Bavoil. 2020. *Optimizing Compute Shaders for L2 Locality Using Thread Group ID Swizzling*. Retrieved August 25, 2021 from <https://developer.nvidia.com/blog/optimizing-compute-shaders-for-l2-locality-using-thread-group-id-swizzling/>.
- [7] L. Chen, H. Cheng, P. Wang, and C. Yang. 2017. Improving GPGPU performance via cache locality aware thread block scheduling. *IEEE Computer Architecture Letters* 16, 2 (2017), 127–131.
- [8] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. 2010. Dynamic load balancing on single- and multi-GPU systems. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS'10)*. 1–12. <https://doi.org/10.1109/IPDPS.2010.5470413>

- [9] Keith D. Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing* 23 (2002), 7–22.
- [10] NVIDIA Corporation. 2021. 2.2 thread hierarchy. In *CUDA C++ Programming Guide*. Retrieved August 25, 2021 from https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
- [11] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2016. Cooperative caching for GPUs. *ACM Transactions on Architecture and Code Optimization* 13, 4 (Dec. 2016), Article 39, 25 pages. <https://doi.org/10.1145/3001589>
- [12] The Khronos OpenCL Working Group. 2021. 3.2.2 execution of kernel-instances. In *The OpenCL. Specification*. Retrieved August 25, 2021 from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.
- [13] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of the 2012 Innovative Parallel Computing Conference (InPar'12)*. 1–14. <https://doi.org/10.1109/InPar.2012.6339596>
- [14] Hsien-Kai Kuo, Ta-Kan Yen, B. C. Lai, and J. Jou. 2013. Cache capacity aware thread scheduling for irregular memory access on many-core GPGPUs. In *Proceedings of the 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC'13)*. 338–343. <https://doi.org/10.1109/ASPDAC.2013.6509618>
- [15] Muhammad Huzaifa, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D. Sinclair, and Sarita V. Adve. 2020. Inter-kernel reuse-aware thread block scheduling. *ACM Transactions on Architecture and Code Optimization* 17, 3 (Aug. 2020), Article 24, 27 pages. <https://doi.org/10.1145/3406538>
- [16] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 395–406. <https://doi.org/10.1145/2451116.2451158>
- [17] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhnag. 2010. Cache topology aware computation mapping for multicores. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 74–85. <https://doi.org/10.1145/1806596.1806605>
- [18] Baldur Karlsson. 2019. Renderdoc v1.5. Retrieved August 25, 2021 from <https://renderdoc.org/docs/index.html>.
- [19] John Kessenich, Dave Baldwin, and Randi Rost. 2017. The OpenGL® Shading Language. Retrieved August 25, 2021 from <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf>.
- [20] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. 1999. A feasibility study in iterative compilation. In *Proceedings of the 2nd International Symposium on High Performance Computing*. 121–132.
- [21] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPU. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 307–319. <https://doi.org/10.1145/3079856.3080239>
- [22] B. Lai, H. Kuo, and J. Jou. 2015. A cache hierarchy aware thread mapping methodology for GPGPUs. *IEEE Transactions on Computers* 64, 4 (April 2015), 884–898.
- [23] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 260–271.
- [24] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-aware CTA clustering for modern GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 297–311. <https://doi.org/10.1145/3037697.3037709>
- [25] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. 1–12. <https://doi.org/10.1145/2807591.2807606>
- [26] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS'15)*. 67–77. <https://doi.org/10.1145/2751205.2751237>
- [27] J. Liu, Y. Zhang, W. Ding, and M. Kandemir. 2011. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'11)*. 161–170. <https://doi.org/10.1109/CGO.2011.5764684>
- [28] Microsoft Corporation. 2015. Anatomy of a Compute Shader Dispatch Call. Retrieved August 25, 2021 from https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm#18.6.320Anatomy20of20a20Compute20Shader20Dispatch20Call.
- [29] Microsoft Corporation. 2015. Fixed Order of Pipeline Results. Retrieved August 25, 2021 from https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm#4.220Fixed20Order20of20Pipeline20Results.
- [30] Microsoft Corporation. 2015. Unordered Access Views. Retrieved August 25, 2021 from https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm#UAVs.

- [31] Microsoft Corporation. 2018. Direct3D 11 Graphics. Retrieved August 25, 2021 from <https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11>.
- [32] Microsoft Corporation. 2018. Direct3D 12 Graphics. Retrieved August 25, 2021 from <https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics>.
- [33] Microsoft Corporation. 2018. High Level Shading Language. Retrieved August 25, 2021 from <https://docs.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hls>.
- [34] Jae-Ho Nah, Yeongkyu Lim, Sunho Ki, and Chulho Shin. 2017. Z^2 traversal order: An interleaving approach for stereo rendering on tile-based GPUs. *Computational Visual Media* 3 (2017), 349–357. <https://doi.org/10.1007/s41095-017-0093-5>
- [35] NVIDIA Corporation. 2016. NVIDIA GeForce GTX 1080—Gaming Perfected. Retrieved August 25, 2021 from https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.
- [36] NVIDIA Corporation. 2018. NVIDIA Turing GPU Architecture—Graphics Reinvented. Retrieved August 25, 2021 <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>.
- [37] NVIDIA Corporation. 2019. Nsight 2019.6. Retrieved August 25, 2021 from <https://developer.nvidia.com/nsight-graphics>.
- [38] NVIDIA Corporation. 2019. Parallel Thread Execution ISA: Application Guide. Retrieved August 25, 2021 from https://docs.nvidia.com/pdf/ptx_isa_6.5.pdf.
- [39] NVIDIA Corporation. 2020. CUDA Occupancy Calculator. Retrieved August 25, 2021 from <https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html>.
- [40] NVIDIA Corporation. 2020. NVIDIA Ampere GA102 GPU Architecture—The Ultimate Play. Retrieved August 25, 2021 from <https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf>.
- [41] NVIDIA Corporation. 2021. CUDA Samples. Retrieved August 25, 2021 from <https://docs.nvidia.com/cuda/cuda-samples/index.html>.
- [42] Amar Patel, Shawn Hargreaves, and Rys Sommefeldt. 2020. DirectX Raytracing (DXR) Functional Spec. Retrieved August 25, 2021 from <https://github.com/microsoft/DirectX-Specs/blob/master/d3d/Raytracing.md>.
- [43] Ram Rangan, Naman Turakhia, and Alexandre Joly. 2020. Countering load-to-use stalls in the NVIDIA turing GPU. *IEEE Micro* 40, 6 (2020), 59–66.
- [44] The Khronos Group Inc. n.d. OpenGL Overview. Retrieved August 25, 2021 from <https://www.opengl.org/documentation/>.
- [45] The Khronos Group Inc. 2018. Vulkan Overview. Retrieved August 25, 2021 from <https://www.khronos.org/vulkan/>.
- [46] Bill Thomas. 2019. Tech Radar. Retrieved August 25, 2021 from <https://www.techradar.com/in/news/amd-radeon-rx-5700-vs-rx-5700-xt-whats-the-best-amd-gpu-for-you>.
- [47] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15)*. 25–35. <https://doi.org/10.1145/2716282.2716283>
- [48] Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics (HPG'10)*. 29–37.
- [49] Michelle Rae Uy. 2019. Tech Radar. Retrieved August 25, 2021 from <https://www.techradar.com/news/nvidia-geforce-rtx-2080-super-vs-rtx-2080-what-has-changed>.
- [50] Umut Uyurkulak, Willem Andreas Haan, Sean Braganza, Brian Dilg, and Collin. 2020. CRYENGINE V manual—Voxel-based global illumination. CRYTEK. Retrieved August 25, 2021 from <https://docs.cryengine.com/pages/viewpage.action?pageId=25535599>.
- [51] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. LaPerm: Locality aware scheduler for dynamic parallelism on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 583–595. <https://doi.org/10.1109/ISCA.2016.57>
- [52] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC'98)*. 1–27.
- [53] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. 119–130. <https://doi.org/10.1145/2751205.2751213>
- [54] Ping Xiang, Yi Yang, and Huiyang Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 284–295. <https://doi.org/10.1109/HPCA.2014.6835939>
- [55] Shucai Xiao and Wu Chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS'10)*. 1–12. <https://doi.org/10.1109/IPDPS.2010.5470477>

- [56] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 76–88. <https://doi.org/10.1109/HPCA.2015.7056023>
- [57] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 369–380. <https://doi.org/10.1145/1950365.1950408>
- [58] Xian Zhu, Robert Wernsman, and Joseph Zambreno. 2018. Improving first level cache efficiency for GPUs using dynamic line protection. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP'18)*. Article 64, 10 pages. <https://doi.org/10.1145/3225058.3225104>

Received March 2021; revised May 2021; accepted July 2021