The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs

Nandita Vijaykumar[†§] Eiman Ebrahimi[‡] Kevin Hsieh[†]
Phillip B. Gibbons[†] Onur Mutlu[§†]

[†]Carnegie Mellon University [‡]NVIDIA [§]ETH Zürich

Abstract

Exploiting data locality in GPUs is critical to making more efficient use of the existing caches and the NUMA-based memory hierarchy expected in future GPUs. While modern GPU programming models are designed to explicitly express parallelism*, there is no clear explicit way to express* data locality*—i.e.,* reuse-based *locality to make efficient use of the caches, or* NUMA *locality to efficiently utilize a NUMA system. On the one hand, this lack of expressiveness makes it a very challenging task for the programmer to write code to get the best performance out of the memory hierarchy. On the other hand, hardware-only architectural techniques are often suboptimal as they miss key higher-level program semantics that are essential to effectively exploit data locality.*

In this work, we propose the Locality Descriptor*, a cross-layer abstraction to* explicitly *express and exploit data locality in GPUs. The Locality Descriptor* (i) *provides the software a flexible and portable interface to optimize for data locality, requiring* no *knowledge of the underlying memory techniques and resources, and* (ii) *enables the architecture to leverage key program semantics and effectively coordinate a range of techniques (e.g., CTA scheduling, cache management, memory placement) to exploit locality in a programmer-transparent manner. We demonstrate that the Locality Descriptor improves performance by 26.6% on average (up to 46.6%) when exploiting reuse-based locality in the cache hierarchy, and by 53.7% (up to 2.8X) when exploiting NUMA locality in a NUMA memory system.*

1. Introduction

Graphics Processing Units (GPUs) have evolved into powerful programmable machines that deliver high performance and energy efficiency to many important classes of applications today. Efficient use of memory system resources is critical to fully harnessing the massive computational power offered by a GPU. A key contributor to this efficiency is *data locality*—both *(i) reuse* of data within the application in the cache hierarchy (*reuse-based locality*) and *(ii)* placement of data *close* to the computation that uses it in a non-uniform memory access (NUMA) system (*NUMA locality*) [1–4].

Contemporary GPU programming models (e.g., CUDA [5], OpenCL [6]) are designed to harness the massive computational power of a GPU by enabling explicit expression of *parallelism* and control of *software-managed memories* (scratchpad memory and register file). However, there is no clear explicit way to express and exploit *data locality*—i.e., *data reuse*, to better utilize the hardware-managed cache hierarchy, or *NUMA locality*, to efficiently use a NUMA memory system.

Challenges with Existing Interfaces. Since there is no explicit interface in the programming model to express and exploit data locality, expert programmers use various techniques such as software scheduling [7] and prefetch/bypass hints [8, 9] to carefully manage locality to obtain high performance. However, all such software approaches are significantly limited for three reasons. First, exploiting data locality is a challenging task, requiring a range of hardware mechanisms such as thread scheduling [7, 10–18], cache bypassing/prioritization [8, 19–28], and prefetching [29–34], to which software has *no easy access*. Second, GPU programs exhibit many different types of data locality, e.g., inter-CTA (reuse of data across Cooperative Thread Arrays or thread blocks), inter-warp and intra-warp locality. Often, *multiple* different techniques are required to exploit each type of locality, as a single technique in isolation is insufficient [7, 28, 34–36]. Hence, software-only approaches quickly become *tedious* and difficult programming tasks. Third, any software optimization employing fine-grained ISA instructions to manage caches or manipulating thread indexing to alter CTA scheduling is *not portable* to a different architecture with a different CTA scheduler, different cache sizes, etc [37].

At the same time, software-transparent architectural techniques miss critical program semantics regarding locality inherent in the algorithm. For example, CTA scheduling is used to improve data locality by scheduling CTAs that share data at the same core. This requires knowledge of *which* CTAs share data—knowledge that *cannot* easily be inferred by the architecture [7, 16]. Similarly, NUMA locality is created by placing data close to the threads that use it. This requires a priori knowledge of *which* threads access *what* data to avoid expensive reactive page migration [38, 39]. Furthermore, many architectural techniques, such as prefetching or cache bypassing/prioritization, would benefit from knowledge of the application's access semantics.

A Case Study. As a motivating example, we examine a common locality pattern of CTAs sharing data (*inter-CTA locality*), seen in the `histo` benchmark (Parboil [40]). `histo` has a predominantly accessed data structure (`sm_mappings`). Figure 1 depicts how this data structure ① is accessed by the CTA grid ②. All threads in GPU programs are partitioned into a multidimensional grid of CTAs. CTAs with the same color access the same data range (also colored the same) ③. As depicted, there is plentiful reuse of data between CTAs ④

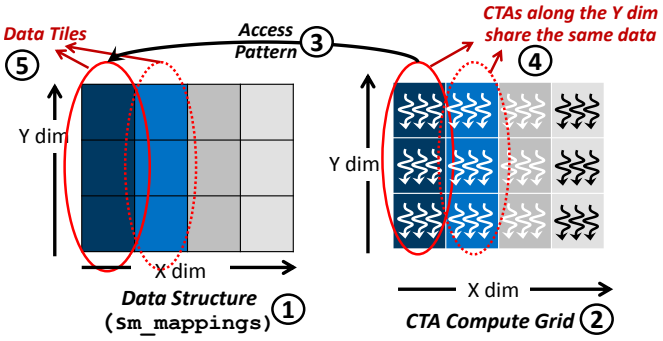

Figure 1: Inter-CTA data locality in `histo` (Parboil).

and the workload has a very deterministic access pattern.

Today, however, exploiting reuse-based locality or NUMA locality for this workload, *at any level of the compute stack*, is a challenging task. The hardware architecture, on the one hand, misses key program information: knowledge of *which CTAs* access the same data ④, so they can be scheduled at the same SM (Streaming Multiprocessor); and knowledge of *which data* is accessed by those CTAs ⑤, so that data can be placed at the same NUMA zone. The programmer/compiler, on the other hand, has *no easy access* to hardware techniques such as CTA scheduling or data placement. Furthermore, optimizing for locality is a tedious task as a *single technique* alone is insufficient to exploit locality (§2). For example, to exploit NUMA locality, we need to coordinate *data placement* with CTA scheduling to place data close to the CTAs that access it. *Hence, neither the programmer, the compiler, nor hardware techniques can easily exploit the plentiful data locality in this workload.*

Our Approach. To address these challenges, we introduce the Locality Descriptor: a *cross-layer* abstraction to *express* and *exploit* different forms of data locality that all levels of the compute stack—from application to architecture—recognize. The Locality Descriptor *(i)* introduces a flexible and portable interface that enables the programmer/software to *explicitly* express and optimize for data locality and *(ii)* enables the hardware to *transparently* coordinate a range of architectural techniques (such as CTA scheduling, cache management, and data placement), guided by the knowledge of *key program semantics*. Figure 2 shows how the programmer or compiler can use the Locality Descriptor to leverage both reuse-based locality and NUMA locality. We briefly summarize how the Locality Descriptor works here, and provide an end-to-end description in the rest of the paper.

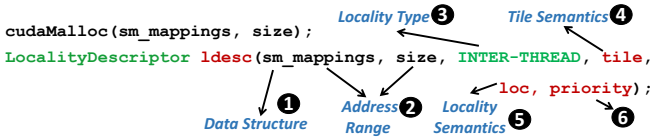

Figure 2: The Locality Descriptor specification for `histo`.

First, each instance of a Locality Descriptor describes a single *data structure's* locality characteristics (in Figure 2, `sm_mappings` ❶) and conveys the corresponding address range ❷. Second, we define several fundamental *locality types* as a contract between the software and the architecture. The locality type, which can be `INTER-THREAD`, `INTRA-THREAD`, or `NO-REUSE`, drives the underlying optimizations used to exploit it. In `histo`, `INTER-THREAD` ❸ describes inter-CTA locality. The locality type ❸ and the *locality semantics* ❺, such as access pattern, inform the architecture to use CTA scheduling and other techniques that exploit the corresponding locality type (described in §3.3). Third, we partition the data structure into *data tiles* that are used to relate data to the threads that access it. In Figure 1, each data range that has the same color (and is hence accessed by the same set of CTAs) forms a data tile. Data tiles and the threads they access are described by the *tile semantics* ❹ (§3.3), which informs the architecture *which CTAs* to schedule together and *which data* to place at the same NUMA zone. Fourth, we use a software-provided *priority* ❻ to reconcile optimizations between Locality Descriptors for different data structures in the same program if they require *conflicting* optimizations (e.g., different CTA scheduling strategies).

We evaluate the benefits of using Locality Descriptors to exploit different forms of both *reuse-based* locality and *NUMA* locality. We demonstrate that Locality Descriptors effectively leverage program semantics to improve performance by 26.6% on average (up to 46.6%) when exploiting reuse-based locality in the cache hierarchy, and by 53.7% (up to 2.8X) when exploiting NUMA locality.

The major **contributions** of this work are:

- This is the first work to propose a holistic cross-layer approach to explicitly *express* and *exploit* data locality in GPUs as a first class entity in both the programming model and the hardware architecture.
- We design the Locality Descriptor, which enables *(i)* the software/programmer to describe data locality in an architecture-agnostic manner and *(ii)* the architecture to leverage key program semantics and coordinate many architectural techniques transparently to the software. We architect an end-to-end extensible design to connect five architectural techniques (CTA scheduling, cache bypassing, cache prioritization, data placement, prefetching) to the Locality Descriptor programming abstraction.
- We comprehensively evaluate the efficacy and versatility of the Locality Descriptor in leveraging different types of reuse-based and NUMA locality, and demonstrate significant performance improvements over state-of-the-art approaches.

2. Motivation

We use two case studies to motivate our work: *(i)* Inter-CTA locality, where different CTAs access the same data and *(ii)* NUMA locality in a GPU with a NUMA memory system.

2.1. Case Study 1: Inter-CTA Locality

A GPU kernel is formed by a *compute grid*, which is a 3D grid of Cooperative Thread Arrays (CTAs). Each CTA, in turn, comprises a 3D array of threads. Threads are scheduled for execution at each Streaming Multiprocessor (SM) at a

CTA granularity. Inter-CTA locality [7, 11, 14–18] is data reuse that exists when multiple CTAs access the same data. CTA scheduling [7, 11, 14–18] is a technique that is used to schedule CTAs that share data at the same SM to exploit inter-CTA locality at the per-SM local L1 caches.

To study the impact of CTA scheduling, we evaluate 48 scheduling strategies, each of which groups (i.e., clusters) CTAs differently: either along the grid's X, Y, or Z dimensions, or in different combinations of the three. The goal of CTA scheduling for locality is to maximize sharing between CTAs at each SM and effectively *reduce* the amount of data accessed by each SM. Hence, as a measure of how well CTA scheduling improves locality for each workload, in Figure 3 we plot the *minimum* working set at each SM (normalized to baseline) across all 48 scheduling strategies. We define *working set* as the average number of uniquely accessed cache lines at each SM. A smaller working set implies fewer capacity misses, more sharing, and better locality. Figure 3 also shows the *maximum* performance improvement among all evaluated scheduling strategies for each benchmark.

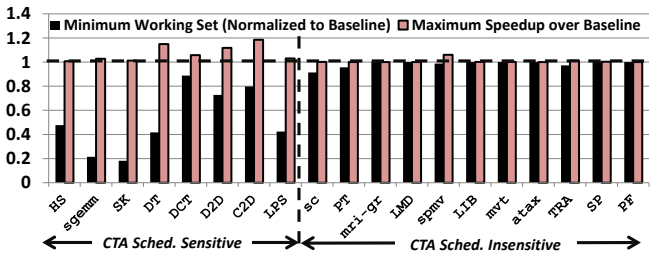

Figure 3: CTA scheduling: performance and working set.

Figure 3 shows that even though CTA scheduling significantly reduces the working set of CTA-scheduling-sensitive applications (on the left) by 54.5%, it has almost no impact on performance (only 3.3% on average across all applications). To understand this minimal impact on performance, in Figure 4 we plot the corresponding increase in L1 hit rate for the specific scheduling strategy that produced the smallest working set (only for the scheduling-sensitive workloads). We also plot the increase in *inflight hit rate*, which we measure as the number of MSHR hits, i.e., another thread already accessed the same cache line, but the line has not yet been retrieved from memory and hits at the MSHRs.

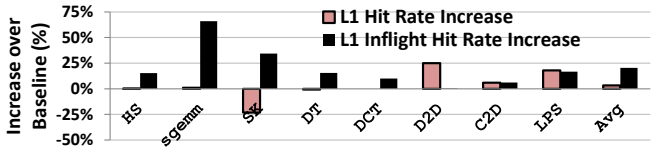

Figure 4: CTA scheduling: L1 hit rate and L1 inflight hit rate.

Figure 4 shows that CTA scheduling has little impact in improving the L1 hit rate (by 3% on average) with the exception of D2D and C2D. This explains the minimal performance impact. CTA scheduling, however, a substantially increases L1 inflight hit rate (by 20% on average). This indicates that even though there is higher data locality due to more threads sharing the same data, these threads wait for the same data at the *same time*. As a result, the increased locality simply causes more threads to *stall*, rather than improving hit rate. Hence, while CTA scheduling is very effective in *exposing* data locality, we still need to address other challenges (e.g., threads stalling together) to obtain performance gains from improved data locality. Furthermore, determining *which* scheduling strategy to use is another challenge, as each application requires a *different* strategy to maximize locality based on the program's sharing pattern.

In summary, to exploit inter-CTA locality *(i)* the hardware-controlled CTA scheduler needs to know *which CTAs access the same data*, to choose an appropriate scheduling strategy (this requires knowledge of program semantics) and *(ii)* a scheduling strategy that *exposes* locality in the cache is *not* necessarily sufficient for translating locality into performance (we need to coordinate other techniques).

2.2. Case Study 2: NUMA Locality

For continued scaling, future GPUs are expected to employ non-uniform memory access (NUMA) memory systems. This can be in the form of multiple memory stacks [3, 4], unified virtual address spaces in multi-GPU/heterogeneous systems [38, 39, 41–45] or multi-chip modules, where SMs and memory modules are partitioned into *NUMA zones* or *multiple GPU modules* [1, 2]. Figure 5 depicts the system evaluated in [1] with four NUMA zones. A request to a remote NUMA zone goes over the lower bandwidth inter-module interconnect, has higher latency, and incurs more traffic compared to *local requests* [1]. To maximize performance and efficiency, we need to control *(i)* how data is placed across NUMA zones and *(ii)* how CTAs are scheduled to maximize local accesses.

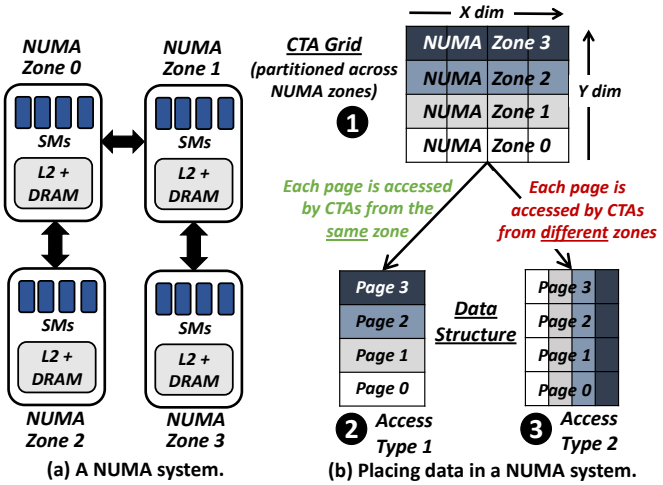

Figure 5: NUMA locality.

To understand why this is a challenging task, let us consider the heuristic-based hardware mechanism proposed in [1], where the CTA grid is partitioned across the 4 NUMA zones such that contiguous CTAs are scheduled at the same SM.

Data is placed at the page granularity (64KB) at the NUMA zone where it is *first accessed*, based on the heuristic that consecutive CTAs are likely to share the same page(s). Figure 5 depicts a CTA grid (❶), which is partitioned between NUMA zones in this manner—consecutive CTAs along the X dimension are scheduled at the same NUMA zone. This heuristic-based mechanism works well for Access Type 1 (❷), where CTAs that are scheduled at the same NUMA zone access the same page(s) (the color scheme depicts which CTAs access what data). However, for Access Type 2 (❷), this policy fails as a single page is shared by CTAs that are scheduled at *different zones*. Two challenges cause this policy to fail. First, suboptimal scheduling: the simple scheduling policy [1] does not *always* co-schedule CTAs that share the same pages at the same zone. This happens when scheduling is *not* coordinated with the application's access pattern. Second, large and fixed page granularity: more CTAs than what can be scheduled at a single zone may access the *same* page. This happens when there are fine-grained accesses by *many* CTAs to each page and when different data structures are accessed by the CTAs in *different* ways. For these reasons (as we evaluate in §6.2), a heuristic-based approach is often ineffective at exploiting NUMA locality.

2.3. Other Typical Locality Types

We describe other locality types, caused by different access patterns, and require other optimizations for locality next.

Inter-warp Locality. Inter-warp locality is data reuse between warps that belong to the same/different CTAs. This type of locality occurs in stencil programs (workloads such as `hotspot` [46] and `stencil` [40]), where each thread accesses a set of neighboring data elements, leading to data reuse between neighboring warps. Inter-warp locality is also a result of misaligned accesses to cache lines by threads in a warp [29,35,47], since data is always fetched at the cache line granularity (e.g., `streamcluster` [46] and `backprop` [46]). Inter-warp locality has short reuse distances [35] as nearby warps are typically scheduled together and caching policies such as LRU can exploit a significant portion of this locality. However, potential for improvement exists using techniques such as inter-warp prefetching [7, 29, 36] or CTA scheduling to co-schedule CTAs that share data [7].

Intra-thread Locality. This is reuse of data by the *same thread* (seen in `LIBOR` [48] and `lavaMD` [46]), where each thread operates on its own working set. Local memory usage in the program is also an example of this type of locality. The key challenge here is *cache thrashing* because *(i)* the overall working set of workloads with this locality type is large due to lack of sharing among threads and *(ii)* the reuse distance per thread is large as hundreds of threads are swapped in and out by the GPU's multithreading before the data is reused by the same thread. Techniques that have been proposed to address cache thrashing include cache bypassing or prioritization (e.g. pinning) of different forms [8, 19–28, 49] and/or warp/CTA throttling [7, 12, 50, 51].

2.4. Key Takeaways & Our Goal

In summary, locality in GPUs can be of different forms depending on the GPU program. Each locality type presents *different* challenges that need to be addressed. Tackling each challenge often requires coordination of *multiple* techniques (such as CTA scheduling and cache bypassing), many of which software has no easy access to. Furthermore, to be effective, some of these techniques (e.g., CTA scheduling, memory placement) require knowledge of *program semantics*, which is prohibitively difficult to infer at run time.

Our goal is to design a holistic *cross-layer* abstraction—that all levels of the compute stack recognize—to express and exploit the different forms of data locality. Such an abstraction should enable connecting a range of architectural techniques with the locality properties exhibited by the program. In doing so, the abstraction should *(i)* provide the programmer/software a simple, yet powerful interface to express data locality and *(ii)* enable architectural techniques to leverage key program semantics to optimize for locality.

3. Locality Descriptor: Abstraction

Figure 6 depicts an overview of our proposed abstraction. The goal is to connect program semantics and programmer intent (❶) with the underlying architectural mechanisms (❷). By doing so, we enable optimization at different levels of the stack: *(i)* as an additional knob for static code tuning by the programmer, compiler, or autotuner (❸), *(ii)* runtime software optimization (❹), and *(iii)* dynamic architectural optimization (❼) using a combination of architectural techniques. This abstraction interfaces with a parallel GPU programming model like CUDA (❺) and conveys key program semantics to the architecture through low overhead interfaces (❻).

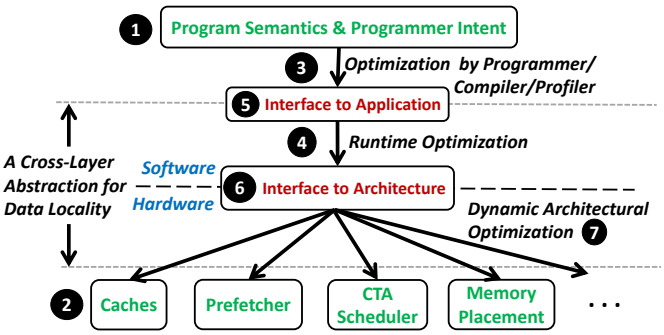

Figure 6: Overview of the proposed abstraction.

3.1. Design Goals

We set three goals that drive the design of our proposed abstraction: *(i) Supplemental and hint-based only*: The abstraction should be an optional add-on to optimize for *performance*, requiring no change to the rest of the program, nor should it impact the program's *correctness*. *(ii) Architecture-agnosticism*: The abstraction should abstract away any low-level details of the architecture (e.g., cache size, number of SMs, caching policy). Raising the abstraction level improves portability,

reduces programming effort, and enables architects to flexibly design and improve techniques across GPU generations, transparently to the software. *(iii) Generality and flexibility*: The abstraction should flexibly describe a wide range of locality types typically seen in GPU programs. It should be easy to extend what the abstraction can express and the underlying architectural techniques that can benefit from it.

3.2. Example Program

We describe the Locality Descriptor with the help of the `histo` (Parboil [40]) workload example described in §1. We begin with an overview of how a Locality Descriptor is specified for `histo` and then describe the key ideas behind the Locality Descriptor's components. Figure 7 depicts a code example from this application. The primary data structure is `sm_mappings`, which is indexed by a function of the thread and block index only along the *X* dimension. Hence, the threads that have the same index along the *X* dimension access the same part of this data structure.

```
__global__ void histo_main_kernel(…){
    ...
    unsigned int local_scan_load = blockIdx.x * blockDim.x +
threadIdx.x;
    ...
    while (local_scan_load < num_elements) {
        uchar4 sm = sm_mappings[local_scan_load];
        local_scan_load += blockDim.x * gridDim.x;
        ...
    }
}
```

Data is shared by all threads/CTAs with the same X index

Figure 7: Code example from `histo` (Parboil).

Figure 8 depicts the data locality in this application in more detail. ① is the CTA grid and ② is the `sm_mappings` data structure. The CTAs that are colored the same access the same data range (also colored the same). As §1 discusses, in order to describe locality with the Locality Descriptor abstraction, we partition each data structure in *data tiles* ③ that group data shared by the same CTAs. In addition, we partition the CTA grid along the X dimension into *compute tiles* ④ to group together CTAs that access the same data tile. We then relate the compute and data tiles with a *compute-data* mapping ⑤ to describe which compute tile accesses which data tile. Figure 9 depicts the code example to express the locality in this example with a Locality Descriptor. As §1 describes, the key components of a Locality Descriptor are: the associated data structure (❶), its locality type (❷), tile semantics (❸), locality semantics (❹), and its priority (❺). We now describe each component in detail.

3.3. An Overview of Key Ideas and Components

Figure 10 shows an overview of the components of the Locality Descriptor. We now describe the five key components and the key insights behind their design.

3.3.1. Data Structure (❶) We build the abstraction around the program's *data structures* (each specified with its base address and size). Each instance of the Locality Descriptor describes the locality characteristics of a single data structure. Designing the Locality Descriptor around the program's data structures is advantageous for two reasons. First, it ensures

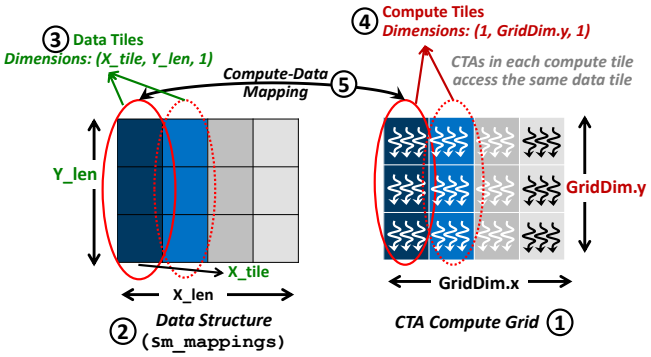

Figure 8: Data locality and compute/data tiles in `histo`.

```
1 uchar4 *sm_mappings;
2 size_t size = X_len * Y_len * sizeof(uchar4);
3 cudaMalloc(sm_mappings, size);
4 Tile_t tile((X_tile, Y_len, 1), (1, GridSize.y, 1), (1, 0, 0))
    ❸ Tile semantics   Data Tile   Compute Tile   Compute-Data Map
5 LocalitySemantics_t loc(COACCESSED, REGULAR, X_len);
    ❹ Locality semantics   Sharing Type   Access Pattern   Stride
6 LocalityDescriptor ldesc(sm_mappings, size, INTER_THREAD, tile,
                                                        loc, 1);
    ❶ Data structure   ❷ Locality type   ❺ Priority
```

Figure 9: Locality Descriptor example for `histo`.

architecture-agnosticism as a data structure is a software-level concept, easy for the programmer to reason about. Second, it is natural to tie locality properties to data structures because in GPU programming models, *all* threads typically access a given data structure in the same way. For example, some data structures are simply streamed through by all threads with no reuse. Others are heavily reused by groups of threads.

3.3.2. Locality Type (❷) Each instance of the Locality Descriptor has an explicit *locality type*, which forms a contract or *basis of understanding* between software and hardware. This design choice leverages the known observation that locality type often determines the underlying optimization mechanisms (§2). Hence, software need only specify locality type and the system/architecture transparently employs a different set of architectural techniques based on the specified type. We provide three fundamental types: *(i)* `INTRA-THREAD`: when the reuse of data is by the same thread itself, *(ii)* `INTER-THREAD`: when the reuse of data is due to sharing of data between different threads (e.g., inter-warp or inter-CTA locality), and *(iii)* `NO-REUSE`: when there is no reuse of data (NUMA locality can still be exploited, as described below). If a data structure has multiple locality types (e.g., if a data structure has both *intra-thread* and *inter-thread* reuse), multiple Locality Descriptors with different types can be specified for that data structure. We discuss how these cases are handled in §4.

3.3.3. Tile Semantics (❸) As data locality is essentially the outcome of how computation accesses data, we need to express the *relation between compute and data*. To do this, we first need a *unit* of computation and data as a basis. To this end, we partition the data structure into a number of *data tiles (D-Tiles)* and the compute grid into a number of *compute tiles (C-Tiles)*. Specifically, a D-Tile is a 3D range of data elements

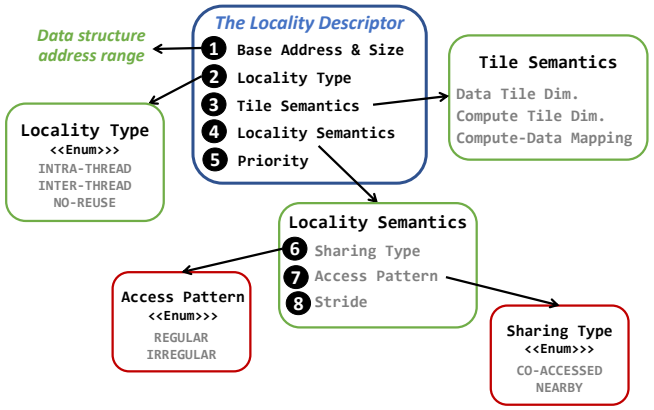

Figure 10: Overview of the Locality Descriptor.

in a data structure and a C-Tile is a 3D range of threads or CTAs in the 3D compute grid (e.g., ③ and ④ in Figure 8).

This design provides two major benefits. First, it provides a *flexible and architecture-agnostic* scheme to express locality types. For example, to express `INTER-THREAD` locality, a D-Tile is the range of data shared by a set of CTAs; and each such set of CTAs forms a C-Tile. To express `INTRA-THREAD` locality, a C-Tile is just a single thread and the D-Tile is the range of data that is reused by that single thread. Second, such decomposition is *intrinsic* and conceptually similar to the existing hierarchical tile-based GPU programming model. Tile partitioning can hence be done easily by the programmer or the compiler using techniques such as [38, 39]. For irregular data structures (e.g, graphs), which cannot be easily partitioned, the Locality Descriptor can be used to describe the entire data structure. This imposes little limitation as such data structures exhibit an irregular type of locality that cannot be easily described by software.

We further reduce complexity in expression by stipulating only an *imprecise* description of locality. There are two primary instances of this. First, we use a simple 1:1 mapping between C-Tile and D-Tile. This is a non-limiting simplification because data locality is fundamentally about *grouping* threads and data based on sharing. If multiple C-Tiles access the same D-Tile, a bigger C-Tile should simply be specified. In an extreme case, where the entire data structure is shared by all threads, we should only have one C-Tile and one D-Tile. In another case, where there is only intra-thread locality (no sharing among threads), there is a natural 1:1 mapping between each thread and its working set. This simplification would be an approximation in cases with irregular sharing, which is out of the Locality Descriptor's scope because of the high complexity and low potential. Second, C-Tile and D-Tile partitioning implies that the grouping of threads and data needs to be *contiguous*—a C-Tile cannot access a set of data elements that is interleaved with data accessed by a different C-Tile. This is, again, a non-limiting requirement as contiguous data elements are typically accessed by neighboring threads to maximize spatial locality and reduce memory traffic. If there is an interleaved mapping between C-Tiles and the D-Tiles they access, the C-Tiles and D-Tiles can be approximated by merging them into bigger tiles until they are contiguous. This design drastically reduces the expression complexity and covers typical GPU applications.

Specifically, the tile semantics are expressed in three parts: *(i) D-Tile dimensions*: The number of data elements (in each dimension) that form a D-Tile. Depending on the data structure, the unit could be any data type. In the `histo` example (③ in Figure 8), the D-Tile dimensions are `(X_tile, Y_len, 1)`, where `X_tile` is the range accessed by a single C-Tile along the X dimension, `Y_len` is the full length of the data structure (in data elements) along the Y dimension. *(ii) C-Tile dimensions*: The number of CTAs in each dimension that form a C-Tile. The compute tile dimensions in the `histo` example (④ in Figure 8) are `(1, GridDim.y, 1)`: 1 CTA along the X dimension, `GridDim.y` is the length of the whole grid along the Y dimension, and since this is a 2D grid, the Z dimension is one. *(iii) Compute-data map*: We use a simple function to rank which order to traverse C-Tiles first in the 3D compute grid as we traverse the D-Tiles in a data structure in $X \rightarrow Y \rightarrow Z$ order. For example, the mapping function `(3,1,2)` implies that when D-Tiles are traversed in the $X \rightarrow Y \rightarrow Z$ order, and the C-Tiles are traversed in the $Y \rightarrow Z \rightarrow X$ order. In our `histo` example, this mapping (⑤ in Figure 8) is simply `(1,0,0)` as the C-Tiles need only be traversed along the X dimension. This simple function saves runtime overhead, but more complex functions can also be used.

3.3.4. Locality Semantics (❹) This component describes the type of reuse in the data structure as well as the access pattern. §4 describes how this information is used for optimization. This component has two parts: Sharing Type (❻) and Access Pattern (❼). There are two options for Sharing Type (❺) to reflect the typical sharing patterns. `COACCESSED` indicates that the *entire* D-Tile is shared by all the threads in the corresponding C-Tile. `NEARBY` indicates that the sharing is more irregular, with nearby threads in the C-Tile accessing nearby data elements (the form of sharing seen due to misaligned accesses to cache lines or stencil-like access patterns [7, 35]). Sharing type can be extended to include other sharing types between threads (e.g., partial sharing). Access Pattern (❻) is primarily used to inform the prefetcher and includes whether the access pattern is `REGULAR` or `IRREGULAR`, along with a stride (❽) within the D-Tile for a `REGULAR` access pattern.

3.3.5. Priority (❺) Multiple Locality Descriptors may require *conflicting* optimizations (e.g., different CTA scheduling strategies). We ensure that these conflicts are rare by using a conflict resolution mechanism described in §4. When a conflict *cannot* be resolved, we use a software-provided *priority* to give precedence to certain Locality Descriptors. This design gives the software more control in optimization, and ensures the key data structure(s) are prioritized.

4. Locality Descriptor: Detailed Design

We detail the design of the programmer/compiler interface (❺ in Figure 6), runtime optimizations (❹), and the architectural interface and mechanisms (❷ and ❼)—CTA scheduling, memory placement, cache management, and prefetching.

4.1. The Programmer/Compiler Interface

The Locality Descriptor can be specified in the code after the data structure is initialized and copied to global memory. Figure 9 is an example. If the semantics of a data structure change between kernel calls, its Locality Descriptor can be re-specified between kernel invocations.

The information to specify the Locality Descriptor can be extracted in three ways. First, the compiler can use static analysis to determine forms of data locality, *without* programmer intervention, using techniques like [7, 16] for inter-CTA locality. Second, the programmer can annotate the program (as was done in this work), which is particularly useful when the programmer wishes to hand-tune code for performance and to specify the *priority* ordering of data structures when resolving potential optimization conflicts (§3.3). Third, software tools such as auto-tuners or profilers [52–54] can determine data locality and access patterns via dynamic analysis.

During compilation, the compiler extracts the variables that determine the address range of each Locality Descriptor, so the system can resolve the virtual addresses at run time. The compiler then summarizes the Locality Descriptor semantics corresponding to these address ranges and places this information in the object file.

4.2. Runtime Optimization

At run time, the GPU driver and runtime system determine how to exploit the locality characteristics expressed in the Locality Descriptors based on the specifics of the underlying architectural components (e.g., number of SMs, NUMA zones). Doing so includes determining the: *(i)* CTA scheduling strategy, *(ii)* caching policy (prioritization and bypassing), *(iii)* data placement strategy across NUMA zones, and *(iv)* prefetching strategy. In this work, we provide an algorithm to coordinate these techniques. Both, the set of techniques used and the algorithm to coordinate them, are extensible. As such, more architectural techniques can be added and the algorithm can be enhanced. We first describe the algorithm that determines *which* architectural techniques are employed for different locality types, and then detail each architectural technique in the following subsections.

Figure 11 depicts the flowchart that determines which optimizations are employed. The algorithm depicted works based on the three *locality types*. First, for `INTER-THREAD`Locality Descriptors, we employ CTA scheduling (§4.3) to expose locality. We also use other techniques based on the *access pattern* and the *sharing type*: *(i)* For `COACCESSED` sharing with a `REGULAR` access pattern, we use guided stride prefetching (§4.5) to overlap the long latencies when many threads are stalled together waiting on the same data; *(ii)* For `COACCESSED` sharing with a `IRREGULAR` access pattern, we employ cache prioritization using *soft pinning* (§4.4) to keep data in the cache long enough to exploit locality; *(iii)* For `NEARBY` sharing, we use simple nextline prefetching tailored to the frequently-occurring access pattern. Second, for an `INTRA-THREAD` Locality Descriptor, we employ a thrash-resistant caching policy, *hard pinning* (§4.4), to keep a part of the working set in the cache. Third, for a `NO-REUSE` Locality Descriptor, we use cache bypassing as the data is not reused. In a NUMA system, *irrespective of the locality type*, we employ CTA scheduling and memory placement to minimize accesses to remote NUMA zones. If there are conflicts between different data structures, they are resolved using the priority order, as described in §4.3 and §4.6.

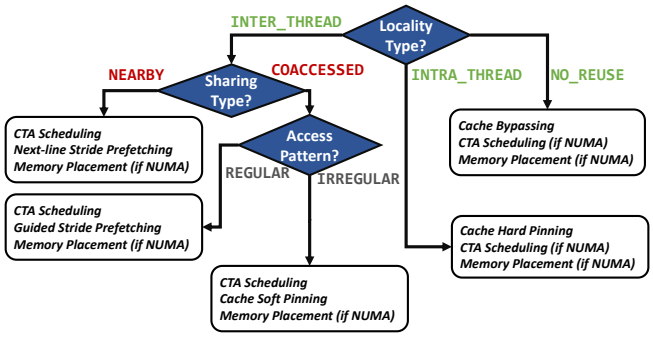

Figure 11: Flowchart of architectural optimizations leveraging Locality Descriptors.

4.3. CTA Scheduling

Figure 12 depicts an example of CTA scheduling for the CTA grid (❶) from our example (`histo`, §3.2). The default CTA scheduler (❷) traverses one dimension at a time ($X \rightarrow Y \rightarrow Z$), and schedules CTAs at each SM in a round robin manner, ensuring load balancing across SMs. Since this approach does *not* consider locality, the default scheduler schedules CTAs that access the same data at *different* SMs (❷).

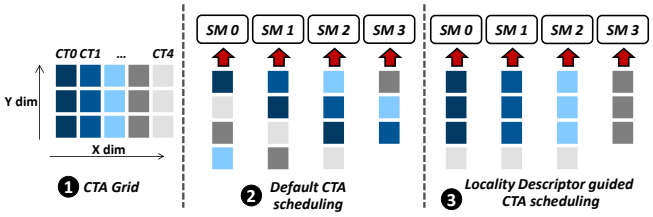

Figure 12: CTA scheduling example.

The Locality Descriptor guided CTA scheduling (❸) shows how we expose locality by grouping CTAs in each C-Tile into a cluster. Each cluster is then scheduled at the same SM. In this example, we spread the last C-Tile (CT4) across three SMs to trade off locality for parallelism. To enable such application-specific scheduling, we need an algorithm to use the Locality Descriptors to drive a CTA scheduling policy that *(i)* schedules CTAs from the same C-Tile (that share data) together to expose locality, *(ii)* ensures all SMs are fully occupied, and *(iii)* resolves conflicts between multiple Locality Descriptors. We use Algorithm 1 to form *CTA clusters*, and schedule each formed cluster at the same SM in a non-NUMA system.[1] In a NUMA system, we first *partition* the CTAs across the different NUMA zones (see §4.6), and then use Algorithm 1 *within* each NUMA zone.

[1]This algorithm optimizes only for the L1 cache, but it can be extended to optimize for the L2 cache as well.

Algorithm 1 Forming CTA clusters using Locality Descriptors

1: **Input:** $LDesc_{1...N}$: all N Locality Descriptors, sorted by priority (highest first)
2: **Output:** $CLS = (CLS_X, CLS_Y, CLS_Z)$: the final cluster dimensions
3: **for** $i = 1$ to N **do** ▷ Step 1: Split C-Tiles into 2 to ensure each *LDesc* has enough C-Tiles for all SMs (to load balance)
4: **while** CT_NUM($LDesc_i$) < SM_{NUM} and CT_DIM($LDesc_i$) != (1, 1, 1,) **do**
5: Divide the C-Tile of $LDesc_i$ into 2 along the largest dimension
6: **end while**
7: **end for**
8: $CLS \leftarrow$ CT_DIM($LDesc_1$) ▷ Each cluster is now formed by each of the highest priority *LDesc*'s C-Tiles after splitting
9: **for** $i = 2$ to N **do** ▷ Step 2: Merge the C-Tiles of lower priority *LDesc*s to form larger clusters to also leverage locality from lower priority *LDesc*s
10: **for** d in (X, Y, Z) **do**
11: $MCLS_d \leftarrow CLS_d \times$ MAX (FLOOR(CT_DIM($LDesc_i$) / CLS_d) , 1) ▷ Merge C-Tiles along each dimension
12: **end for**
13: **if** CT_NUM($MCLS$) $\geq SM_{NUM}$ **then** ▷ Ensure there are enough C-Tiles for all SMs
14: $CLS \leftarrow MCLS$
15: **end if**
16: **end for**

The algorithm first ensures that each Locality Descriptor has enough C-Tiles for all SMs. If that is not the case, it splits C-Tiles (lines 3–7), to ensure we have enough clusters to occupy all SMs. Second, the algorithm uses the C-Tiles of the highest priority Locality Descriptor as the initial CTA clusters (line 8), and then attempts to merge the lower-priority Locality Descriptors (lines 9–16).[2] *Merging* tries to find a cluster that also groups CTAs with shared data in other lower-priority Locality Descriptors while keeping the clusters larger than the number of SMs (first step). By scheduling the merged cluster at each SM, the system can expose locality for multiple data structures. The GPU driver runs Algorithm 1 before launching the kernel to determine the CTA scheduling policy.

4.4. Cache Management

The Locality Descriptor enables the cache to distinguish reuse patterns of different data structures and apply policies accordingly. We use two caching mechanisms that can be further extended. First, *cache bypassing* (e.g., [8, 19–27]), which does not insert data that has no reuse (NO-REUSE locality type) into the cache. Second, *cache prioritization*, which inserts some data structures into the cache with higher priority than the others. We implement this in two ways: *(i)* hard pinning and *(ii)* soft pinning. *Hard pinning* is a mechanism to prevent cache thrashing due to large working sets by ensuring that part of the working set stays in the cache. We implement hard pinning by inserting all hard-pinned data with the highest priority and evicting a specific cache way (e.g., the 0th way) when *all* cache lines in the same set have the highest priority. Doing so protects the cache lines in other cache ways from being repeatedly evicted. We use a timer to automatically reset all priorities to *unpin* these pinned lines periodically. *Soft pinning*, on the other hand, simply prioritizes one data structure over others without any policy to control thrashing. As §4.2 discusses, we use hard pinning for data with INTRA-THREAD locality type, which usually has a large working set as there is very limited sharing among threads. We use soft pinning for data with INTER-THREAD locality type to ensure that this data is retained in the cache until other threads that share the data access it.

[2] Although the algorithm starts by optimizing the highest priority LDesc, it is designed to find a scheduling strategy that is optimized for *all* LDescs. Only when no such strategy can be found (i.e., when there are conflicts), is the highest priority LDesc prioritized over others.

4.5. Prefetching

As §2 discusses, using CTA scheduling alone to expose locality hardly improves performance, as the increased locality causes more threads to stall, waiting for the *same critical data* at the *same time* (see the L1 inflight hit rate, Figure 3). As a result, the memory latency to this critical data becomes the performance bottleneck, since there are too few threads left to hide the memory latency. We address this problem by employing a hardware prefetcher guided by the Locality Descriptor to prefetch the *critical data* ahead of time. We employ a prefetcher only for INTER-THREAD Locality Descriptors because the data structures they describe are shared by multiple threads, and hence, are more critical to avoid stalls. The prefetcher is triggered when an access misses the cache on these data structures. The prefetcher is instructed based on the access pattern and the sharing type. As Figure 11 shows, there are two cases. First, for NEARBY sharing, the prefetcher is directed to simply prefetch the next cache line. Second, for COACCESSED sharing with a REGULAR access pattern, the prefetched address is a function of *(i)* the access stride, *(ii)* the number of bytes that are accessed at the same time (i.e., the width of the data tile), and, *(iii)* the size of the cache, as prefetching too far ahead means more data needs to be retained in the cache. The address to prefetch is calculated as: `current address + (L1_size/(number_of_active_tiles*data_tile_width) * stride)`. The `number_of_active_tiles` is the number of D-Tiles that the prefetcher is actively prefetching. The equation decreases the prefetch distance when there are more active D-Tiles to reduce thrashing. This form of controlled prefetching avoids excessive use of memory bandwidth by only prefetching data that is shared by many threads, and has high accuracy as it is informed by the Locality Descriptor.

4.6. Memory Placement

As §2.2 discusses, exploiting locality on a NUMA system requires coordination between CTA scheduling and memory placement such that CTAs access local data within each NUMA zone. There are two major challenges (depicted in Figure 5 in §2.2): *(i)* how to partition data among NUMA zones at a fine granularity. A paging-based mechanism (e.g., [1]) does not solve this problem as a large fixed page size is typically ineffective (§2.2), while small page sizes are prohibitively expensive to manage [55], and *(ii)* how to partition CTAs among NUMA zones to exploit locality among *multiple* data structures that may be accessed differently by the CTAs in the program. To address these two challenges, we use a *flexible* data mapping scheme, which we describe below, and a CTA partitioning algorithm that leverages this scheme.

Flexible Fine-granularity Data Mapping. We enhance the mapping between physical addresses and NUMA zones to enable data partitioning at a flexible granularity, smaller than a page (typically 64KB). Specifically, we use consecutive bits within the physical address itself to index the NUMA zone (similar to [3] in a different context). We allow using a *different* set of bits for different data structures. Thus, each data structure can be partitioned across NUMA zones at a *different* granularity.[3] Figure 13 shows how this is done for the example in §2.2. As the figure shows, CTAs in each NUMA zone ❶ access the same page (64KB) for data structure A ❷, but they only access the same *quarter-page (16KB)* for data structure B ❸. If we partition data across NUMA zones only at the page granularity [1], most accesses to data structure B would access remote NUMA zones. With our mechanism, we can choose bits 16-17 (which interleaves data between NUMA zones at a 64KB granularity) and bits 14-15 (which interleaves data at a 16KB granularity) in the physical address to index the NUMA zone for data structures A and B respectively. Doing so results in all accesses to be in the *local* NUMA zone for *both* data structures.

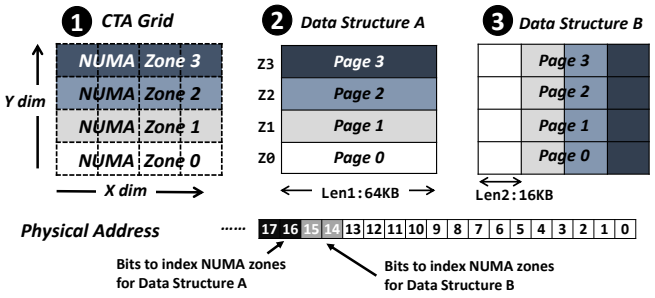

Figure 13: Memory placement with Locality Descriptors.

This design has two constraints. First, we can partition data only at a power-of-two granularity. Our findings, however, show that this is not a limiting constraint because *(i)* regular GPU kernels typically exhibit power-of-two strides across CTAs (consistent with [3]); and *(ii)* even with non-power-of-two strides, this approach is still reasonably effective compared to page-granularity placement (as shown quantitatively in §6.2). Second, to avoid cases where a data structure is *not* aligned with the interleaving granularity, we require that the GPU runtime align data structures at the page granularity.

CTA Scheduling Coordinated with Data Placement. To coordinate memory placement with CTA scheduling, we use a simple greedy search algorithm (Algorithm 2) that partitions the CTAs across the NUMA zones and selects the most effective address mapping bits for each data structure. The algorithm is described in detail in our extended technical report [56]. We provide a brief overview here.

[3]We limit the bits that can be chosen to always preserve the minimum DRAM burst size (128B) by always specifying a size between 128B-64KB (bits 7-16). We always use bit 16/17 for granularities larger than 64KB as we can flexibly map virtual pages to the desired NUMA zone using the page table. We enable flexible bit mapping by modifying the hardware address decoder in the memory controller.

The algorithm evaluates the efficacy of all possible address mappings for the data structure described by the highest-priority Locality Descriptor (line 4). This is done by determining which N consecutive bits between bit 7-16 in the physical address are the most effective bits to index NUMA zones for that data structure (where N is the base-2-log of the number of NUMA zones). To determine which mapping is the most effective, the algorithm first determines the corresponding CTA partitioning scheme for that address mapping using the `NUMA_PART` function (line 5). The `NUMA_PART` function simply schedules each C-Tile at the NUMA zone where the D-Tile it accesses is placed (based on the address mapping that is being tested). The 1:1 C-Tile/D-Tile compute mapping in the Locality Descriptor gives us the information to easily do this. To evaluate the effectiveness or *utility* of each address mapping and the corresponding CTA partitioning scheme, we use the `COMP_UTIL` function (line 7). This function calculates the ratio of local/remote accesses for each mapping.

Algorithm 2 CTA partitioning and memory placement for NUMA

1: **Input:** $LDesc_{1...N}$: all N Locality Descriptors, sorted by priority (highest first)
2: **Output 1:** CTA_NPART: the final CTA partitioning for NUMA zones
3: **Output 2:** $MAP_{1...N}$: the address mapping bits for each $LDesc$
4: **for** $b\_hi = 7$ to 16 **do** ▷ Test all possible mappings for the highest-priority $LDesc$
5: $CTA\_PART_{b\_hi} \leftarrow$ `NUMA_PART`$(LDesc_1, b\_hi)$ ▷ Partition the CTAs based on the address mapping being evaluated
6: $best\_util\_all \leftarrow 0$ ▷ $best\_util\_all$: the current best utility
7: $util_{b\_hi} \leftarrow$ `COMP_UTIL`$(N, LDesc_1, CTA\_PART_{b\_hi}, b\_hi)$ ▷ Calculate the utility of the CTA partitioning scheme + address mapping
8: **for** $i = 2$ to N **do** ▷ Test other $LDescs$
9: $TMAP_i \leftarrow 7$ ▷ $TMAP$: temporary mapping
10: $best\_util \leftarrow 0$ ▷ $best\_util$: the utility with the best mapping
11: **for** $b\_lo = 7$ to 16 **do** ▷ Test all possible address mappings
12: $util \leftarrow$ `COMP_UTIL`$(N-i+1, LDesc_i, CTA\_PART_{b\_hi}, b\_lo)$ ▷ Calculate overall best mapping
13: **if** $util > best\_util$ **then**
14: $TMAP_i \leftarrow b\_lo$; $best\_util \leftarrow util$ ▷ update the best mapping
15: **end if**
16: **end for**
17: $util_{b\_hi} \leftarrow util_{b\_hi} + best\_util$ ▷ update the new best utility
18: **end for**
19: **if** $util_{b\_hi} > best\_util\_all$ **then**
20: $MAP \leftarrow TMAP$; $MAP_1 \leftarrow b\_hi$;
21: $best\_util\_all \leftarrow util_{b\_hi}$; $CTA\_NPART \leftarrow CTA\_PART_{b\_hi}$
22: **end if**
23: **end for**

Since we want a CTA partitioning scheme that is effective for *multiple* data structures, we also evaluate how *other data structures* can be mapped, based on each CTA partitioning scheme tested for the high-priority data structure (line 8). Based on which of the tested mappings has the highest overall utility, we finally pick the CTA partitioning scheme and an address mapping scheme for each data structure (line 12).

The GPU driver runs Algorithm 2 when all the dynamic information is available at run time (i.e., number of NUMA zones, CTA size, data structure size, etc.). The overhead is negligible because: *(i)* most GPU kernels have only several data structures (i.e., small N), and *(ii)* the two core functions (`NUMA_PART` and `COMP_UTIL`) are very simple due to the 1:1 C-Tile/D-Tile mapping.

The Locality Descriptor method is more flexible and versatile than a first-touch page migration scheme [1], which *(i)* requires demand paging to be enabled, *(ii)* is limited to

a *fixed* page size, *(iii)* always schedules CTA in a fixed manner. With the knowledge of how CTAs access data (i.e., the D-Tile-C-Tile compute mapping) and the ability to control and coordinate both the CTA scheduler and flexibly place data, our approach provides a powerful substrate to leverage NUMA locality.

5. Methodology

We model the entire Locality Descriptor framework in GPGPU-Sim 3.2.2 [57]. To isolate the effects of the cache locality versus NUMA locality, we evaluate them separately: we evaluate reuse-based locality using an existing single-chip non-NUMA system configuration (based on Fermi GTX 480); and we use a futuristic NUMA system (based on [1]) to evaluate NUMA-based locality. We use the system parameters in [1], but with all compute and bandwidth parameters (number of SMs, memory bandwidth, inter-chip interconnect bandwidth, L2 cache) scaled by 4 to ensure that the evaluated workloads have sufficient parallelism to saturate the compute units. Table 1 summarizes the major system parameters. We use GPUWattch [58] to model GPU power consumption.

Table 1: Major parameters of the simulated systems.

Shader Core	1.4 GHz; GTO scheduler [50]; 2 schedulers per SM Round-robin CTA scheduler
SM Resources	Registers: 32768; Scratchpad: 48KB, L1: 32KB, 4 ways
Memory Model	FR-FCFS scheduling [59, 60], 16 banks/channel
Single Chip System	15 SMs; 6 memory channels; L2: 768KB, 16 ways
Multi-Chip System	4 GPMs (GPU Modules) or NUMA zones; 64 SMs (16 per module); 32 memory channels; L2: 4MB, 16 ways; Inter-GPM Interconnect: 192 GB/s; DRAM Bandwidth: 768 GB/s (192 GB/s per module)

We evaluate workloads from the CUDA SDK [48], Rodinia [46], Parboil [40] and PolybenchGPU [61] benchmark suites. We run each kernel either to completion or up to 1B instructions. Our major performance metric is instruction throughput (IPC). From the workloads in Table 2, we use cache-sensitive workloads (i.e., workloads where increasing the L1 by 4× improves performance more than 10%), to evaluate reuse-based locality. We use memory bandwidth-sensitive workloads (workloads that improve performance by more than 40% with 2× memory bandwidth), to evaluate NUMA locality.

6. Evaluation

We evaluate the Locality Descriptor's efficacy and versatility using two use cases: *(i)* leveraging *reuse-based locality* to improve cache hit rates in §6.1 and *(ii)* improving NUMA locality (§6.2) by placing data close to threads that use it in a NUMA system.

6.1. Reuse-Based (Cache) Locality

We evaluate six configurations: *(i)* `Baseline`: our baseline system with the default CTA scheduler (§4.3). *(ii)* `BCS`: a heuristic-based CTA scheduler based on BCS [11], which schedules two consecutive CTAs at the same SM. *(iii)* `LDesc-Sched`: the Locality Descriptor-guided CTA scheduler, which uses the Locality Descriptor semantics and algorithm (§4.3). Compiler techniques such as [7, 16] can produce the same benefits. *(iv)* `LDesc-Pref`: the Locality Descriptor-guided prefetcher, as described in §4.5. Sophisticated classification-based prefetchers such as [36], can potentially obtain similar benefits. *(v)* `LDesc-Cache`: the Locality Descriptor-guided cache prioritization and bypassing scheme (§4.4). *(vi)* `LDesc`: our proposed scheme, which uses the Locality Descriptor to distinguish between the different locality types and selectively employs different (scheduling, prefetching, caching, and data placement) optimizations based on Figure 11.

Table 2: Summary of Applications

Name (Abbr.)	Locality Descriptor types (§3.3)
Syrk (SK) [61]	`INTER-THREAD (COACCESSED, REGULAR), NO-REUSE`
Doitgen (DT) [61]	`INTER-THREAD (COACCESSED, REGULAR), NO-REUSE`
dwt2d (D2D) [46]	`INTER-THREAD (NEARBY, REGULAR)`
Convolution-2D (C2D) [61]	`INTER-THREAD (NEARBY)`
Sparse Matrix Vector Multiply (SPMV) [40]	`INTRA-THREAD,` `INTER-THREAD (COACCESSED, IRREGULAR)`
LIBOR (LIB) [48]	`INTRA-THREAD`
LavaMD (LMD) [46]	`INTRA-THREAD,` `INTER-THREAD (COACCESSED, REGULAR)`
histogram (HS) [40]	`INTER-THREAD (COACCESSED, REGULAR)`
atax (ATX) [61]	`NO-REUSE, INTER-THREAD (COACCESSED, REGULAR)`
mvt (MVT) [61]	`NO-REUSE, INTER-THREAD (COACCESSED, REGULAR)`
particlefilter (PF) [46]	`NO-REUSE`
streamcluster (SC) [46]	`NO-REUSE, INTER-THREAD (NEARBY)`
transpose (TRA) [48]	`NO-REUSE`
Scalar Product (SP) [48]	`NO-REUSE`
Laplace Solver (LPS) [48]	`NO-REUSE, INTRA-THREAD`
pathfinder (PT) [46]	`NO-REUSE`

Figure 14 depicts the speedup over `Baseline` across all configurations. `LDesc` improves performance by 26.6% on average (up to 46.6%) over `Baseline`. `LDesc` always performs either as well as or better than any of the techniques in isolation. Figure 15 shows the L1 hit rate for different configurations. `LDesc`'s performance improvement comes from a 41.1% improvement in average hit rate (up to 57.7%) over `Baseline`. We make three observations that provide insight into `LDesc`'s effectiveness.

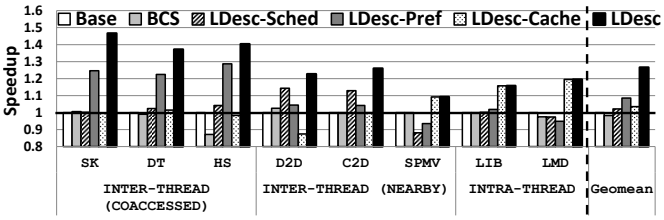

Figure 14: Normalized performance with Locality Descriptors.

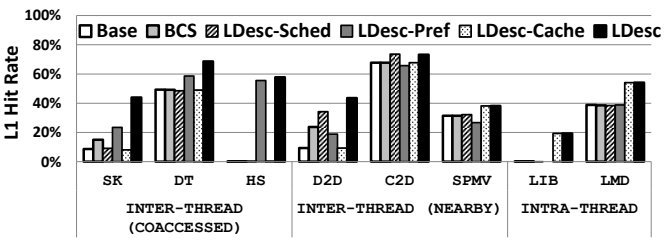

Figure 15: L1 hit rate with Locality Descriptors.

First, different applications benefit from *different* optimizations. Applications with INTER-THREAD type of locality (SK, DT, HS, D2D, C2D) benefit from CTA scheduling and/or prefetching. However, LIB, LMD, SPMV do not benefit from CTA scheduling as there is little inter-CTA reuse to be exploited. Similarly, prefetching significantly hurts performance in these workloads (LIB, LMD, SPMV) as the generated prefetch requests exacerbate the memory bandwidth bottleneck. As a result, significant performance degradation occurs when the working set is too large (e.g., when there is no sharing and only INTRA-THREAD reuse, as in LMD and LIB) or where the access patterns in the major data structures are not sufficiently regular (SPMV). Cache prioritization and bypassing is very effective in workloads with INTRA-THREAD reuse (LIB, LMD), but is largely ineffective and can even hurt performance in workloads such as D2D and HS when a non-critical data structure or too many data structures are prioritized in the cache. Since LDesc is able to distinguish between locality types, it is able to select the best combination of optimizations for each application.

Second, a single optimization is very often *insufficient* to exploit locality. For the INTER-THREAD applications (SK, DT, HS, D2D, C2D), LDesc-guided CTA scheduling significantly reduces the L1 working set (by 67.8% on average, not graphed). However, this does *not* translate into significant performance improvement when scheduling is applied by itself (only 2.1% on average). To understand why, we plot the L1 in-flight hit rate in Figure 16 for the INTER-THREAD COACCESSED workloads (SK, DT, HS): we see a 17% average increase as a result of more threads accessing the same data. These threads wait on the same shared data at the *same time*, and hence cause increased stalls at the core. The benefit of increased locality is thus lost. Prefetching (LDesc-Pref) is an effective technique to alleviate effect. However, prefetching by itself significantly increases the memory traffic and this hinders its ability to improve performance when applied alone. When combined with scheduling, however, prefetching effectively reduces the long memory latency stalls. Synergistically, CTA scheduling reduces the overall memory traffic by minimizing the working set. For the INTER-THREAD NEARBY workloads (C2D, D2D), CTA scheduling co-schedules CTAs with overlapping working sets. This allows more effective prefetching between the CTAs for the critical high-reuse data. In the cases described above, prefetching and CTA scheduling work better synergistically than in isolation, and LDesc is able to effectively combine and make use of multiple techniques depending on the locality type.

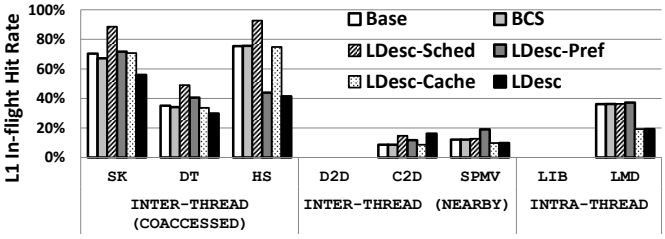

Figure 16: L1 in-flight hit rate with Locality Descriptors.

Third, LDesc-guided CTA scheduling is significantly more effective than the heuristic-based approach, BCS. This is because LDesc tailors the CTA scheduling policy for each application by clustering CTAs based on the locality characteristics of each data structure (§4.3). Similarly, the LDesc prefetcher and replacement policies are highly effective, because they leverage program semantics from the Locality Descriptor (§4.4 and §4.5).

Impact on Energy Consumption. Our energy evaluations show that LDesc reduces the overall system energy consumption by 4.0% on average. In contrast, LDesc-Pref *increases* energy consumption by 5.7%. LDesc-Sched and LDesc-Cache have a negligible impact on energy (<1%).

Conclusions. We make the following conclusions: *(i)* The Locality Descriptor is an effective and versatile mechanism to leverage reuse-based locality to improve GPU performance and energy efficiency; *(ii)* Different locality types require different optimizations—a single mechanism or set of mechanisms do not work for all locality types. We demonstrate that the Locality Descriptor can effectively connect different locality types with the underlying architectural optimizations. *(iii)* The Locality Descriptor enables the hardware architecture to leverage the program's locality semantics to provide significant performance benefits over heuristic-based approaches such as the BCS scheduler.

6.2. NUMA Locality

To evaluate the benefits of the Locality Descriptor in exploiting NUMA locality, Figure 17 compares four different mechanisms: *(i)* Baseline: The baseline system which uses a static XOR-based address hashing mechanism [62] to randomize data placement across NUMA zones. *(ii)* FirstTouch-Distrib: The state-of-the-art mechanism proposed in [1], where each page (64KB) is placed at the NUMA zone where it is first accessed. This scheme also employs a heuristic-based distributed scheduling strategy where the compute grid is partitioned equally across the NUMA zones such that *contiguous* CTAs are placed in the same NUMA zone. *(iii)* LDesc-Placement: The memory placement mechanism described in §4.6 based on the semantics of the Locality Descriptors, but *without* the accompanying CTA scheduling strategy. *(iv)* LDesc: The Locality Descriptor-guided memory placement mechanism with the coordinated CTA scheduling strategy (§4.6). We draw two conclusions from the figure.

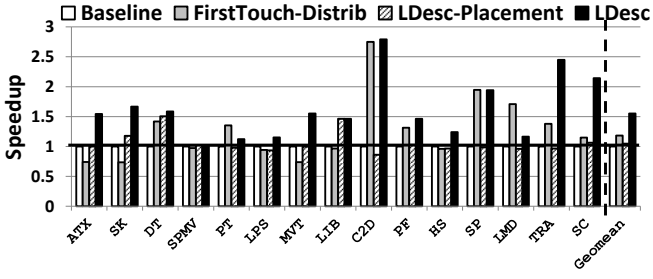

Figure 17: NUMA Locality: Normalized performance.

Conclusion 1. LDesc is an effective mechanism in NUMA data placement, outperforming Baseline by 53.7% on average (up to 2.8×) and FirstTouch_Distrib by 31.2% on average (up to 2.3×). The performance impact of NUMA placement is primarily determined by two factors: *(i) Access efficiency* (plotted in Figure 18), which is defined as the fraction of total memory accesses that are to the local NUMA zone (higher is better). Access efficiency determines the amount of traffic across the interconnect between NUMA zones as well as the latency of memory accesses. *(ii) Access distribution* (plotted in Figure 19) across NUMA zones. Access distribution determines the effective memory bandwidth being utilized by the system—a non-uniform distribution of accesses across NUMA zones may lead to underutilized bandwidth in one or more zones, which can create a new performance bottleneck and degrade performance.

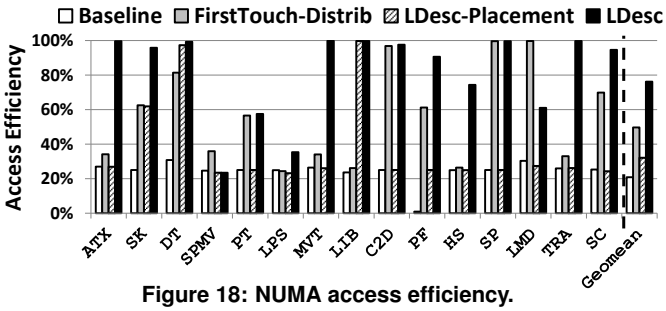

Figure 18: NUMA access efficiency.

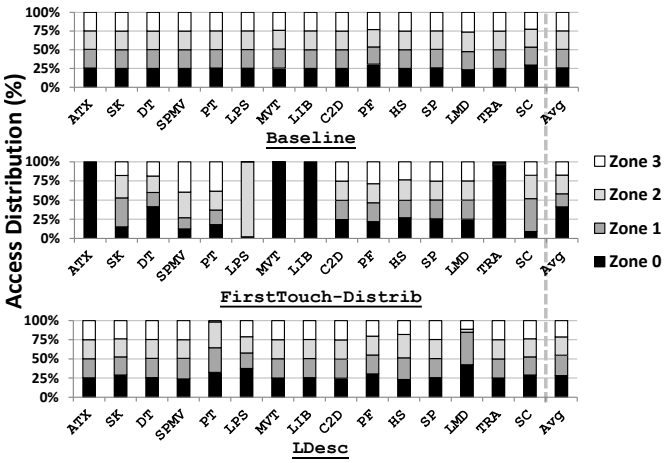

Figure 19: NUMA zone access distribution.

The static randomized mapping in Baseline aims to balance access distribution across NUMA zones (with an average distribution of ∼25% at each zone), but is *not* optimized to maximize access efficiency (only 22% on average). FirstTouch-Distrib on the other hand, has higher access efficiency in some workloads (e.g., SP, PT) by ensuring that a page is placed where the CTA that accesses it first is scheduled (49.7% on average). However, FirstTouch-Distrib is still ineffective for many workloads for three reasons: *(i)* Large page granularity (64KB) often leads to high skews in access distribution when pages are shared between many CTAs, e.g., ATX, MVT, LIB (Figure 19). This is because a majority of pages are placed in the NUMA zone where the CTA that is furthest ahead in execution is scheduled. *(ii)* FirstTouch-Distrib has low access efficiency when the heuristic-based scheduler does *not* schedule CTAs that access the same pages at the same NUMA zone (e.g., DT, HS, SK). *(iii)* FirstTouch-Distrib has low access efficiency when each CTA irregularly accesses a large number of pages because data *cannot* be partitioned between the NUMA zones at a fine granularity (e.g., SPMV).

LDesc interleaves data at a fine granularity depending on how each data structure is partitioned between CTAs and schedules those CTAs accordingly. If a data structure is shared among more CTAs than what can be scheduled at a single zone, the data structure is partitioned across NUMA zones, as LDesc favors parallelism over locality. Hence, LDesc tries to improve access efficiency while reducing skew in access distribution in the presence of a large amount of data sharing. As a result, LDesc has an average access efficiency of 76% and access distribution close to 25% across the NUMA zones (Figure 19). LDesc is less effective in access efficiency in cases where the data structures are irregularly accessed (SPMV) or when non-power-of-two data tile sizes lead to imperfect data partitioning (LPS, PT, LMD, HS).

Conclusion 2. From Figure 17, we see that LDesc is *largely ineffective* without coordinated CTA scheduling. LDesc-Placement retains the LDesc benefit in reducing the skew in access distribution (not graphed). However, *without* coordinated CTA scheduling access efficiency is very low (32% on average).

Energy Consumption. Our energy evaluations show that LDesc consumes 9.1% less energy on average (up to 46.6% less) than Baseline and 9.0% less energy than FirstTouch-Distrib (up to 32.6%).

We conclude that the Locality Descriptor approach is an effective strategy for data placement in a NUMA environment by *(i)* leveraging locality semantics in intelligent data placement *and* CTA scheduling and *(ii)* orchestrating the two techniques using a single interface.

7. Related Work

To our knowledge, this is the first work to propose a cross-layer abstraction that enables the software/programmer to flexibly express and exploit different forms of data locality in GPUs. This enables leveraging program semantics to transparently coordinate architectural techniques that are critical to improving performance and energy efficiency. We briefly discuss prior work related to different aspects of our proposal:

Improving Cache Locality in GPUs. There is a large body of research that aims to improve cache locality in GPUs using a range of hardware/software techniques such as CTA scheduling [7, 10–18], prefetching [29–33, 36, 63], warp scheduling [50, 64, 65], cache bypassing [8, 19–27], and other cache management schemes [13, 28, 34–36, 47, 66–77]. Some of these works orchestrate multiple techniques [7, 13, 31, 34–36, 72, 76] to leverage synergy between optimizations. However, these prior approaches are either hardware-only, software-

only, or focus on optimizing a single technique. Hence, they are limited *(i)* by what is possible with the information that can be solely *inferred* in hardware, *(ii)* by existing software interfaces that limit what optimizations are possible, or *(iii)* in terms of the range of optimizations that can be used. In contrast, the Locality Descriptor provides a new, portable and flexible interface to the software/programmer. This interface allows easy access to hardware techniques in order to leverage data locality. Furthermore, all the above prior approaches are largely orthogonal to the Locality Descriptor as they can use the Locality Descriptor to enhance their efficacy with the knowledge of program semantics.

The closest work to ours is ACPM [35], an architectural cache management technique that identifies intra-warp/inter-warp/streaming locality and selectively applies cache pinning or bypassing based on the detected locality type. This work is limited to the locality types that can be inferred by hardware, and it does *not* tackle inter-CTA locality or NUMA locality, both of which require a priori knowledge of program semantics and hardware-software codesign.

Improving Locality in NUMA GPU Systems. A range of hardware/software techniques to enhance NUMA locality have been proposed in different contexts in GPUs: multiple GPU modules [1], multiple memory stacks [3], and multi-GPU systems with unified virtual addressing [4, 38, 39, 42–44]. We already qualitatively and quantitatively compared against `FirstTouch-Distrib` [1] in §6.2. Our memory placement technique is similar to the approach taken in TOM [3]. In TOM, frequent power-of-two strides seen in GPU kernels are leveraged to use consecutive bits in the address to index a memory stack. TOM, however, *(i)* is the state-of-the-art technique targeted at near-data processing and does *not* require coordination with CTA scheduling, *(ii)* relies on a profiling run to identify the index bits, and *(iii)* does *not* allow using different index bits for different data structures. Techniques to improve locality in multi-GPU systems [4, 38, 39, 42–44] use profiling and compiler analysis to partition the compute grid and data across multiple GPUs. These works are similar to the Locality Descriptor in terms of the partitioning used for forming data and compute tiles and, hence, can easily leverage Locality Descriptors to further exploit reuse-based locality and NUMA locality in a single GPU.

Expressive Programming Models/Runtime Systems/Interfaces. In the context of multi-core CPUs and distributed/heterogeneous systems, there have been numerous software-only approaches that allow explicit expression of data locality [78–86], data independence [81–83, 86] or even tiles [87, 88], to enable the runtime to perform NUMA-aware placement or produce code that is optimized to better exploit the cache hierarchy. These approaches *(i)* are software-only; hence, they do not have access to many architectural techniques that are key to exploiting locality and *(ii)* do not tackle the GPU-specific challenges in exploiting data locality. These works are largely orthogonal to ours and can use Locality Descriptors to leverage hardware techniques to exploit reuse-based locality and NUMA locality in GPUs.

Expressive Memory (XMem) [89] is a cross-layer interface to communicate program semantics from the application to the system software and hardware architecture. XMem is similar in spirit to the Locality Descriptor in providing a new expressive abstraction to bridge the semantic gap between software and hardware. However, XMem is primarily designed to convey program semantics to aid general memory optimization in CPUs. Expressing locality in GPUs imposes different design challenges, requires describing a different set of semantics, and requires optimizing a different set of architectural techniques, leading to a very different cross-layer design for the abstraction.

8. Conclusion

This paper demonstrates the benefits of an *explicit abstraction* for data locality in GPUs that is recognized by all layers of the compute stack, from the programming model to the hardware architecture. We introduce the Locality Descriptor, a rich cross-layer abstraction to explicitly express and effectively leverage data locality in GPUs. The Locality Descriptor *(i)* provides the software/programmer a flexible and portable interface to optimize for data locality without any knowledge of the underlying architecture and *(ii)* enables the architecture to leverage program semantics to optimize and coordinate multiple hardware techniques in a manner that is transparent to the programmer. The key idea is to design the abstraction around the program's data structures and specify locality semantics based on how the program accesses each data structure. We evaluate and demonstrate the performance benefits of Locality Descriptors from effectively leveraging different types of reuse-based locality in the cache hierarchy and NUMA locality in a NUMA memory system. We conclude that by providing a flexible and powerful cross-cutting interface, the Locality Descriptor enables leveraging a critical yet challenging factor in harnessing a GPU's computational power, data locality.

Acknowledgments

We thank the ISCA 2018 reviewers for their valuable suggestions. We thank the members of NVIDIA Research for their feedback, especially Dave Nellans and Evgeny Bolotin. We acknowledge the support of our industrial partners, especially Google, Huawei, Intel, Microsoft, and VMware. This work was supported in part by SRC, NSF, and ETH Zürich.

References

[1] A. Arunkumar *et al.*, "MCM-GPU: Multi-chip-module GPUs for continued performance scalability," in *ISCA*, 2017.
[2] U. Milic *et al.*, "Beyond the socket: NUMA-aware GPUs," in *MICRO*, 2017.
[3] K. Hsieh *et al.*, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *ISCA*, 2016.
[4] G. Kim *et al.*, "Toward standardized near-data processing with unrestricted data placement for GPUs," in *SC*, 2017.
[5] NVIDIA, "CUDA Programming Guide," 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[6] AMD, "AMD Accelerated Parallel Processing OpenCL Programming Guide," in *www.developet.amd.com/GPU/AMDAPPSDK*, 2011.

[7] A. Li *et al.*, "Locality-aware CTA clustering for modern GPUs," in *ASPLOS*, 2017.
[8] X. Xie *et al.*, "An efficient compiler framework for cache bypassing on GPUs," in *ICCAD*, 2013.
[9] NVIDIA, "PTX ISA Version 6.0," in *http://docs.nvidia.com/cuda/parallel-thread-execution/#cache-operators*.
[10] V. Narasiman *et al.*, "Improving GPU performance via large warps and two-level warp scheduling," in *MICRO*, 2011.
[11] M. Lee *et al.*, "Improving GPGPU resource utilization through alternative thread block scheduling," in *HPCA*, 2014.
[12] O. Kayıran *et al.*, "Neither more nor less: optimizing thread-level parallelism for GPGPUs," in *PACT*, 2013.
[13] A. Jog *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," in *ASPLOS*, 2013.
[14] J. Wang *et al.*, "Laperm: Locality aware scheduler for dynamic parallelism on GPUs," in *ISCA*, 2016.
[15] X. Gong *et al.*, "TwinKernels: an execution model to improve GPU hardware scheduling at compile time," in *CGO*, 2017.
[16] L. Chen *et al.*, "Improving GPGPU performance via cache locality aware thread block scheduling," *CAL*, 2017.
[17] B. Lai *et al.*, "A cache hierarchy aware thread mapping methodology for GPGPUs," *TACO*, 2015.
[18] B. Wu *et al.*, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *ICS*, 2015.
[19] Y. Tian *et al.*, "Adaptive GPU cache bypassing," in *GPGPU*, 2015.
[20] C. Li *et al.*, "Locality-driven dynamic GPU cache bypassing," in *ICS*, 2015.
[21] A. Li *et al.*, "Adaptive and transparent cache bypassing for GPUs," in *SC*, 2015.
[22] Y. Liang *et al.*, "An efficient compiler framework for cache bypassing on GPUs," *ICCAD*, 2015.
[23] S. Mittal, "A survey of cache bypassing techniques," *JLPEA*, 2016.
[24] S. Lee *et al.*, "Ctrl-C: Instruction-aware control loop based adaptive cache bypassing for GPUs," in *ICCD*, 2016.
[25] C. Zhao *et al.*, "Selectively GPU cache bypassing for un-coalesced loads," in *ICPADS*, 2016.
[26] X. Xie *et al.*, "Coordinated static and dynamic cache bypassing for GPUs," in *HPCA*, 2015.
[27] D. Li *et al.*, "Priority-based cache allocation in throughput processors," in *HPCA*, 2015.
[28] R. Ausavarungnirun *et al.*, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.
[29] J. Lee *et al.*, "Many-thread aware prefetching mechanisms for GPGPU applications," in *MICRO*, 2010.
[30] A. Sethia *et al.*, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in *PACT*, 2013.
[31] H. Jeon *et al.*, "CTA-aware prefetching for GPGPU," *Computer Engineering Technical Report Number CENG-2014-08*, 2014.
[32] P. Liu *et al.*, "Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads," *TACO*.
[33] N. Lakshminarayana *et al.*, "Spare register aware prefetching for graph algorithms on GPUs," in *HPCA*, 2014.
[34] A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," in *ISCA*, 2013.
[35] G. Koo *et al.*, "Access pattern-aware cache management for improving data utilization in GPU," in *ISCA*, 2017.
[36] Y. Oh *et al.*, "APRES: Improving cache efficiency by exploiting load characteristics on GPUs," in *ISCA*, 2016.
[37] N. Vijaykumar *et al.*, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.
[38] J. Cabezas *et al.*, "Automatic parallelization of kernels in shared-memory multi-GPU nodes," in *ICS*, 2015.
[39] J. Cabezas *et al.*, "Automatic execution of single-GPU computations across multiple GPUs," in *PACT*, 2014.
[40] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois, at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.
[41] N. Agarwal *et al.*, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *HPCA*, 2015.
[42] R. Sakai *et al.*, "Towards automating multi-dimensional data decomposition for executing a single-GPU code on a multi-GPU system," in *CANDAR*, 2016.
[43] J. Kim *et al.*, "Achieving a single compute device image in OpenCL for multiple GPUs," in *PPOPP*, 2011.
[44] J. Lee *et al.*, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *PACT*, 2013.
[45] A. Ziabari *et al.*, "UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs," *TACO*, 2016.
[46] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[47] G. Koo *et al.*, "Revealing critical loads and hidden data locality in GPGPU applications," in *IISWC*, 2015.
[48] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples
[49] R. Ausavarungnirun *et al.*, "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," ser. ASPLOS, 2018.
[50] T. Rogers *et al.*, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.
[51] O. Kayiran *et al.*, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
[52] A. Davidson *et al.*, "Toward techniques for auto-tuning GPU algorithms," in *IWAPC*, 2010.
[53] M. Khan *et al.*, "A script-based autotuning compiler system to generate high-performance CUDA code," *TACO*, 2013.
[54] K. Sato *et al.*, "Automatic tuning of CUDA execution parameters for stencil processing," in *Software Automatic Tuning*, 2011.
[55] R. Ausavarungnirun *et al.*, "Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes," in *MICRO*, 2017.
[56] N. Vijaykumar *et al.*, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs," in *CMU SAFARI Technical Report No.2018-008*, 2018.
[57] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[58] J. Leng *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ISCA*, 2013.
[59] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.
[60] W. Zuravleff *et al.*, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," 1997, US Patent 5,630,096.
[61] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to GPU codes," in *InPar*, 2012.
[62] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.
[63] N. Vijaykumar *et al.*, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.
[64] S. Lee *et al.*, "CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," in *ISCA*, 2015.
[65] Y. Zhang *et al.*, "Locality based warp scheduling in GPGPUs," *Future Generation Computer Systems*, 2017.
[66] H. Choi *et al.*, "Reducing off-chip memory traffic by selective cache management scheme in GPGPUs," in *GPGPU*, 2012.
[67] X. Chen *et al.*, "Adaptive cache management for energy-efficient GPU computing," in *MICRO*, 2014.
[68] M. Khairy *et al.*, "Efficient utilization of GPGPU cache hierarchy," in *GPGPU*, 2015.
[69] B. Wang *et al.*, "DaCache: Memory divergence-aware GPU cache management," in *ICS*, 2015.
[70] J. Kloosterman *et al.*, "WarpPool: Sharing requests with inter-warp coalescing for throughput processors," in *MICRO*, 2015.
[71] M. Khairy *et al.*, "SACAT: Streaming-aware conflict-avoiding thrashing-resistant GPGPU cache management scheme," *TPDS*, 2017.
[72] K. Kim *et al.*, "IACM: Integrated adaptive cache management for high-performance and energy-efficient GPGPU computing," in *ICCD*, 2016.
[73] Y. Zhang *et al.*, "Locality protected dynamic cache allocation scheme on GPUs," in *Trustcom/BigDataSE/ISPA*, 2016.
[74] W. Jia *et al.*, "Characterizing and improving the use of demand-fetched caches in GPUs," in *ICS*, 2012.
[75] S. Mu *et al.*, "Orchestrating cache management and memory scheduling for GPGPU applications," *VLSI*, 2014.
[76] Z. Zheng *et al.*, "Adaptive cache and concurrency allocation on GPGPUs," *CAL*, 2015.
[77] J. Gaur *et al.*, "Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads," in *MICRO*, 2013.
[78] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005.
[79] B. Chamberlain *et al.*, "Parallel Programmability and the Chapel Language," *IJHPCA*, 2007.
[80] K. Fatahalian *et al.*, "Sequoia: Programming the memory hierarchy," in *SC*, 2006.
[81] M. Bauer *et al.*, "Structure slicing: Extending logical regions with fields," in *SC*, 2014.
[82] S. Treichler *et al.*, "Realm: An event-based low-level runtime for distributed memory architectures," in *PACT*, 2014.
[83] E. Slaughter *et al.*, "Regent: A high-productivity programming language for HPC with logical regions," in *SC*, 2015.
[84] Y. Yan *et al.*, "Hierarchical place trees: A portable abstraction for task parallelism and data movement." in *LCPC*, 2009.
[85] G. Bikshandi *et al.*, "Programming for parallelism and locality with hierarchically tiled arrays," in *PPOPP*, 2006.
[86] M. Bauer *et al.*, "Legion: Expressing locality and independence with logical regions," in *SC*, 2012.
[87] J. Guo *et al.*, "Programming with tiles," in *PPOPP*, 2008.
[88] D. Unat *et al.*, "Tida: High-level programming abstractions for data locality management," in *ISC*, 2016.
[89] N. Vijaykumar *et al.*, "A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory," in *ISCA*, 2018.