TAFE: Thread Address Footprint Estimation for Capturing Data/Thread Locality in GPU Systems

Kishore Punniyamurthy
The University of Texas at Austin
kishore.punniyamurthy@utexas.edu

Andreas Gerstlauer
The University of Texas at Austin
gerstl@ece.utexas.edu

Abstract

In multi-GPU and multi-chiplet GPU systems exhibiting NUMA behavior, information about addresses accessed by threads is crucial for various optimizations such as data/thread co-location and cache/scratchpad memory management. To make optimal decisions and avoid runtime overhead, knowledge about dynamic, potentially data-dependent access patterns should be available before kernel execution. Existing approaches require rewriting of applications or can only capture static, data-independent patterns. In this paper, we propose TAFE, a framework for accurate dynamic thread address footprint estimation of GPU applications. TAFE combines minimal static address pattern annotations with dynamic data dependency tracking to compute threadblock-specific address footprints prior to kernel launch. We propose a low-overhead software mechanism to track dynamic data-dependencies and provide an optional lightweight hardware extension to support transparent tracking.

We evaluate TAFE on different NUMA GPU system configurations. TAFE achieves 91% estimation accuracy across a wide range of access patterns while incurring less than 3% tracking and estimation overhead. We further demonstrate benefits of using TAFE for efficient data/compute co-location. A TAFE-optimized thread/-page mapping, can reduce off-chip traffic by 23% (up to 62%) while requiring only minimal, architecture-oblivious annotations from programmer. Furthermore, a TAFE-optimized system achieves on average 45% and 32% (up to 2x) higher performance compared to an unoptimized baseline and 10% and 22% over existing static, data-independent schemes across multiple system configurations.

CCS Concepts

• **Computer systems organization** → *Parallel architectures.*

Keywords

Data/thread locality, NUMA GPU, data and compute partitioning

ACM Reference Format:
Kishore Punniyamurthy and Andreas Gerstlauer. 2020. TAFE: Thread Address Footprint Estimation for Capturing Data/Thread Locality in GPU Systems. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3410463.3414641

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414641

1 Introduction

General-purpose graphic processing units (GP-GPUs) have become widely used as programmable accelerators for a wide range of application domains. Driven by ever-growing application demands, GPUs are increasingly deployed in multi-GPU and multi-chiplet module (MCM) configurations. Such systems have been proposed in research [9, 41, 45, 54] and industry roadmaps [5]. These types of systems inherently exhibit non-uniform memory accesses (NUMA) behavior and as such are prone to issues, such as exacerbated performance impact of uncoalesced accesses [4] or high performance and energy penalties for inter-module and off-chip accesses [27].

There are numerous prior works aimed at addressing access-dependent bottlenecks, including access-pattern-aware prefetching [30], cache management [31] and page allocation [21, 40, 42, 52]. However, all these solutions require knowledge about data/thread locality, which in turn depends on potentially data-dependent application access patterns that traditionally have to be learnt or predicted during kernel execution. Such online approaches inherently have runtime overhead. Furthermore, since they can only be applied after the kernel has been launched, their benefits are limited and additionally incur costs of applying optimizations at runtime. For instance, moving pages across modules in large systems incurs significant overhead due to resulting off-chip traffic and TLB shootdowns [40].

Enabling spatial locality-based optimizations with low overhead requires knowledge about the address ranges accessed by different threads before the accesses are made, i.e. ideally prior to kernel launch. Some solutions [15, 16, 22, 49] have provided language constructs to associate threads with data they access, but these require the programmer to learn new languages and rewrite the application. The work in [55] proposes extending commonly used languages to capture and leverage data/thread locality using tile semantics. However, it only supports static tiling that can not capture data-dependent access patterns while also requiring the programmer to determine and provide explicit tile dimensions, compute- and data-tile mappings and data sharing information.

In this paper, we propose TAFE, a framework for accurately estimating both static, data-independent and dynamic, data-dependent *thread address footprints (TAFs)* prior to thread and kernel execution. Data-dependent patterns require tracking of input data, which, if not carefully done, can result in overhead proportional to the input problem size. Our framework relies on a combination of easily-derivable code annotations to capture regular, static access patterns with dynamic tracking of input data dependencies. We propose both pure software as well as hardware-assisted mechanisms for lightweight dependency tracking with minimal overhead. We show that with this information, the relation between thread IDs and addresses accessed (the TAFs) can be accurately estimated prior to the

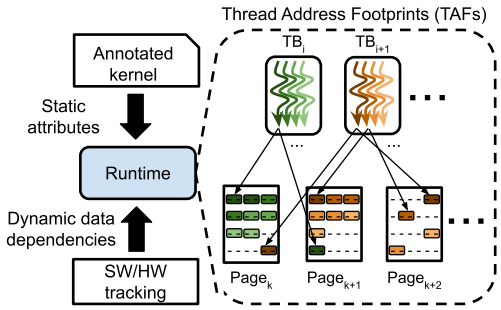


Figure 1: TAFE overview.

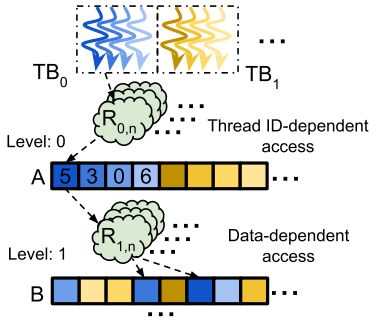


Figure 2: Generic array access patterns.

launch of a GPU kernel. TAFs enable the OS/runtime to derive the spatial sharing of data across threads/warps/threadblocks without explicit input from user. As a case study, we demonstrate that using this information, an OS/runtime can achieve better data/thread co-location in NUMA GPU systems. Such TAFs can in turn also be used for prefetching pages in unified, e.g. CUDA-managed memory systems, transformations of irregular (scatter/gather) patterns into regular remote memory accesses via local page caching and re-arrangement, or reducing coherency traffic [12, 55, 60].

We make the following specific contributions in this paper:

- We introduce the concept of *thread address footprints (TAFs)* and propose *TAF relations* as a compact representation for computing TAFs as a relation of thread IDs and input data dependencies.
- We propose low-overhead software mechanisms and hardware extensions to collect the information required for TAF estimation just-in-time before launch of a kernel.
- We evaluate the estimation accuracy and demonstrate the benefits of our TAFE framework using estimated TAFs for improved data/thread co-location across multiple NUMA GPU configurations.

The paper is organized as follows: We first introduce the concept of TAFs in Section 3. Our TAFE framework and optimizations are described in Sections 4 and 5, respectively. Section 6 discusses evaluation, and the paper concludes with a discussion of the related work followed by summary in Sections 7 and 8, respectively.

2 TAFE Overview

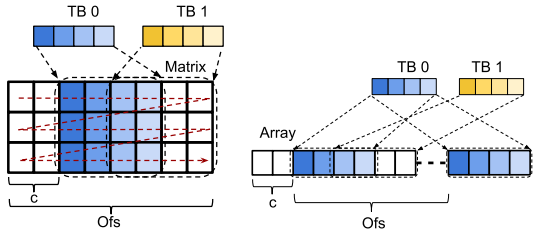
An overview of TAFE is shown in Fig. 1. We focus on arrays as representation of arbitrary data structures contiguously laid out in virtual memory, which are the pre-dominant types of data structures in GPGPU applications. We distinguish between array accesses that are either: 1) *Thread ID-dependent*, where final array addresses are a function of the thread index only, or 2) *Input data-dependent*, where addresses accessed depend on both the thread ID and external input data passed into the thread by the host. The TAFs for the former can be derived from the coefficients of thread indices and offsets used to compute array indices. This information is readily available in the code and can be easily annotated (via static attributes). TAFE uses them to determine the exact addresses accessed by each thread and thereby any spatial sharing of data without needing the programmer to explicitly specify it.

For input-data-dependent accesses, the TAF is typically established through one or more levels of nested and indirect accesses, where an externally initialized primary array is accessed in a specific pattern, and the result is used to compute the index into secondary arrays. Obtaining the TAF for such accesses requires establishing the relationships between 1) the thread ID and the primary array index accessed, 2) the primary array index and the value stored in it, and 3) the primary array value and the secondary array address being accessed. The first and last can be obtained statically in the same way as for purely thread ID-dependent accesses. Tracking 2), however, can only be done dynamically. In order to know the primary array values before thread or kernel execution the primary array values must be known before the thread or kernel launches. We therefore track primary array values and relate primary indices to values stored in them during their initialization. Directly capturing this relationship is, however, infeasible. It would incur overhead that is proportional to size of the primary array itself. Our framework proposes mechanisms to capture data-dependent TAFs and their required information with low overhead. Once a primary array is initialized, TAFE uses the tracked primary array values along with the static information from 1) and 2) to compute the TAF for data-dependent accesses.

3 Thread Address Footprints

In order to capture data/thread locality, it is necessary to estimate the address footprint of different threads. In general, threads can access arrays with arbitrary levels of nesting as shown in Fig. 2. In the first level (level 0), thread IDs are used to determine the indices accessed in the first array. In each subsequent level, values stored in primary arrays are used to compute the indices into the next secondary array. At each level l, the indices accessed in different arrays can be captured and represented by relations $R_{l,n} \subseteq (values, index), 0 \leq n < N_l$ that map thread IDs or one or more primary array values to sets of secondary array indices. Each relation is specific to one secondary array, where every array can be accessed using one or more relation. Fig. 2 shows such relations $R_{l,n}$, where l is the nesting level and n is relation ID within level l. The application-specific relations $R_{l,n}$ determine the access patterns and can range from simple identity functions to arbitrary code. TAFs can be established by propagating thread IDs through relations across nesting levels.

In TAFE, we estimate the TAFs of all threads accessing the same data structure by capturing and successively evaluating individual relations throughout all nesting levels. If multiple relations are associated with an array, we assume that all relations contribute towards all nested accesses. In practice, for every thread and threadblock we find their mapping into sets of array indices and use that



(a) Logical layout (b) Physical layout

Figure 3: Thread ID-dependent access.

to determine the addresses and pages accessed. In the following, we identify the information and overhead required to capture TAF relations for different access patterns. TAFE directly supports a restricted set of commonly found relations and approximates others as described further below.

3.1 Thread ID-Dependent Accesses

At first level ($l = 0$), the relations ($R_{0,n}$) take thread IDs as input and define a mapping to array indices for each thread. TAFE natively supports capturing linear relations including iterative accesses within loop control structures of the following form:

$$R_{0,n}(tid) = \{\mathbf{m} \cdot \mathbf{ID}(tid) + c + i \cdot ofs \mid 0 \leq i < I\},$$

where, $R_{0,n}(tid)$ is the set of all indices accessed by thread ID *tid* for a given array, $\mathbf{m}$ is a coefficient vector, $\mathbf{ID}(tid)$ is a function that maps a global thread ID into multi-dimensional block and thread indices, c is a constant, I is the iteration count of the loop and *ofs* is the offset added to the index in each loop iteration. In the simplest case, the array index is computed as an affine function of the thread ID (N and *ofs* are set to 0, i.e. each thread only accesses one array element). The above relation represents patterns where threadblocks access sections of an array separated by an offset iteratively as shown in Fig. 3. Such patterns are commonly seen in linear algebra, dynamic programming or sliding window applications [48].

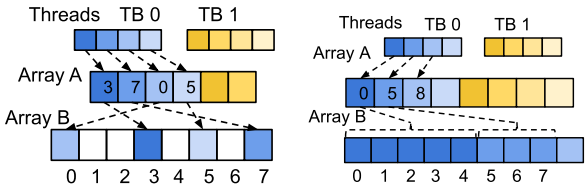
3.2 Input Data-Dependent Accesses

Nested array accesses ($l > 0$) use data stored in primary arrays to determine the secondary indices accessed. There can be one or more primary arrays supplying the input values. TAFE supports two common variants of nested relations $R_{l,n}$:

Indirect In this variant, the result of one memory access (primary access) is used to compute the address for the next memory access (secondary access). TAFE can natively capture patterns with single primary arrays of the form:

$$R_{l,n}(r) = \{m \cdot P[r] + c\}.$$

Here, P is the primary array, r is the primary array index, m is a coefficient, c is a constant, and $R_{l,n}(r)$ maps indices of P into the set of secondary array indices indirectly accessed. Indices r are in turn determined by relations $R_{\text{l-1,q}}$ attached to P in nesting level $l-1$, going recursively back all the way to original thread IDs. This captures patterns like `B[A[i]]` (scatter/gather operations), where the value stored in array A is used to index into array B as shown in Fig. 4a. Such patterns can be seen in algorithms such as link-list traversals, hash search [25], radix sort [24] and COO format-based sparse matrix vector multiplication [20].



(a) Indirect (b) Indirect range

Figure 4: Data-dependent access.

Indirect Range Alternatively, primary arrays can provide bounds of a continuous range of indices accessed in a nested array. TAFE natively supports patterns where two primary arrays provide the start and end indices for accessing a secondary array as follows:

$$R_{l,n}(r_s, r_e) = \{k \mid P_s[r_s] + c_s \leq k < P_e[r_e] + c_e\}.$$

Here, P_s and P_e are the primary arrays containing starting and ending indices (can be identical), r_s and r_e are indices of corresponding start and end elements, c_s, and c_e are constants, and $R_{l,n}(r_s, r_e)$ is the set of secondary indices at level l spanned by the values of start and end elements in P_s and P_e. Fig. 4b shows a specific case in which start and end indices are represented by adjacent values in a single primary array as commonly seen in compact adjacency list representations of graphs [23] and compressed sparse row (CSR) formats [6, 56]. Such patterns are seen in quadratic parallelization based graph traversal algorithms [39], sparse matrix multiplications [8] and shortest path graph algorithm (SSSP, APSP) implementations [23].

While not exhaustive, the relations presented above cover commonly seen access patterns [48]. In general, accesses can have non-linear relations, nested accesses with multiple primary arrays or with specific dependencies across nesting levels, more complex control flow, etc. However, patterns that cannot be directly represented can always be over- or under-approximated by finding the closest match using supported relations (or a combination thereof).

4 TAFE Framework

In this section, we explain our framework for estimation of TAFs in detail. In heterogeneous systems involving CPUs and GPUs, the host usually prepares (and/or initializes) data and then launches kernels across SMs. The required memory for data structures is usually dynamically allocated by the host. Our framework constructs the TAF relations for different data structures before their initialization and passes the information to the OS/runtime, which then use them to estimate TAFs and make optimization decisions before kernel launch. Thread ID-based patterns can be represented using few coefficients and can be statically evaluated to obtain the addresses accessed by threads. Data-dependent access patterns, by contrast require dynamic information (primary array values) to be estimated. Tracking primary arrays can incur overhead proportional to array size if not done carefully.

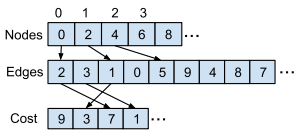
We use a simplified example derived from the *Bfs* benchmark in [17] to demonstrate the working of TAFE throughout this section. The example accesses data stored in CSR format, which is widely used to represent sparse matrices [3, 6] and graphs [7, 56]. A snippet

Table 1: TAFE API.

API	Description
taf_paramSetup (BlkDim, GridDim)	Passes the kernel launch parameters to the framework. The launch parameters passed are stored in the APD header. *BlkDim*, *GridDim*: block and grid dimensions of the application kernel.
taf_register (Base, size, TID_DEP, T, **m**, c, ofs, N, Pri)	This registers a thread ID-dependent TAF relation for an array with our framework. The arguments passed are stored within the APD entry. *Base*: array starting address, *Size*: array size, TID_DEP: thread ID-dependent access pattern, *T*: element size, **m**: pattern coefficients, *c*: constant, *ofs*: loop offset, *N*: loop iteration count, *Pri*: Primary array
taf_register (Base, size, INDIR, T, PBase, m, c, Pri)	Same as above but for Indirect access patterns. *Base*: array starting address, *Size*: array size, *INDIR*: Indirect access pattern, *PBase*: primary array base address, *m*: pattern coefficient, *c*: constant added to primary array values.
taf_register (Base, size, INDIR_RANGE, T, $PBase_s$, c_s, $PBase_e$, c_e, Pri)	Same as above but for Indirect range access patterns. *Base*: array starting address, *size*: array size, *INDIR_RANGE*: Indirect range access pattern, $PBase_s$: base address of primary array with starting indices, c_s: constant added to starting array values. $PBase_e$: base address of primary array with ending indices, c_e: constant added to ending array values.
taf_setRdy (Base)	Indicate that all the information required to evaluate the TAF for an array is available. For a data-dependent pattern, this call needs to be made after the last store to its primary array. This API stalls until all pending stores are completed before marking the array as ready. *Base*: array starting address.
taf_reset (Base)	Remove registered array and reset APD and extent table entries. *Base*: array starting address.

```
int tid = blockIdx.x
          * THREADBLOCK_SIZE +
          threadIdx.x;
int start = Nodes[tid];
int end = Nodes[tid+1];
for (i = start; i < end; ++i) {
  int id = Edges[i];
  Cost[id]++;
}
```

(a) Kernel code



(b) Data dependent accesses

Figure 5: CSR example (*Bfs*).

of the example *Bfs* kernel code is shown in Fig. 5a. Array `Nodes` contains source offsets and is accessed in thread ID-dependent manner. `Edges` contains destination indices, which have data-dependent (indirect range) access patterns dependent on `Nodes`. Finally, the `Cost` array also exhibits data-dependent (indirect) access patterns, but in turn dependent on `Edges`. In the example in Fig. 5b, thread ID 0 accesses `Nodes[0]`, `Edges[0-1]` and `Cost[2-3]`.

4.1 Static Code Annotations

TAFE provides APIs to collect static attributes about arrays from the application and store them in an *access pattern directory (APD)* within OS memory. API calls can be annotated in the host code by the programmer. Alternatively, profiling tools or static compiler analysis can also be used to identify simple patterns and collect the required attributes, e.g. obtaining coefficients, constant loop bounds or offsets. We use manual annotations in our evaluation for simplicity. Note that the programmer does not need to have any knowledge about the architecture of the system on which the application will be executed. Annotated TAFs and TAF relations represent application-specific but architecture-independent information. Therefore, the compiled code can be ported to different systems without further modifications.

API The complete list of TAFE APIs with their parameters is described in Table 1. The snippet of host code corresponding to the *Bfs* kernel launch with TAFE API calls is shown in Listing 1. The application allocates memory for arrays, initializes them and launches the kernel which operates on them. The annotated code required for TAFE is shown in bold. We first setup the kernel parameters for the framework using a `taf_paramSetup()` call. Each of the arrays are then registered with the framework along with the attributes of their access patterns. `Nodes` is registered as an array with its base address, size (`numNodes+1`), thread ID-dependent access (TID_DEP), element-size (`sizeof(int)`) and coefficient vector. The constant, loop offset and loop iteration count are set to 0. `Edges` and `Cost` are similarly registered, except that the base address of their primary

```
taf_paramSetup(dimBlock, dimGrid);
int *Nodes = malloc(numNodes+1)
taf_register(Nodes, numNodes+1, TID_DEP, sizeof(int),
             [1,0,0,THREADBLOCK_SIZE,0,0],0,0,0,1);
taf_setRdy(Nodes);
int *Edges = malloc(numEdges);
taf_register(Edges, numEdges, INDIR_RANGE,
             sizeof(int), Nodes, 0, Nodes, 0, 1);
for (int i = 0; i < numNodes+1; i++) {
/* Initialize "Nodes" array */
    Nodes[i] = prepNodeData(i);
}
taf_setRdy(Edges);
int *Cost = malloc(numNodes);
taf_register(Cost, numNodes, INDIR, sizeof(int),
             Edges, 1, 0, 0);
for (int i = 0; i < numEdges; i++) {
   /* Initialize "Edges" array */
    Edges[i] = prepEdgeData(i);
}
taf_setRdy(Cost);
for (int i = 0; i < numNodes; i++) {
    Cost[i] = prepCostData(i);
}
kernel<<<dimGrid,dimBlock>>>(Nodes, Edges, Cost);
```

Listing 1: CSR example: host code with TAFE APIs.

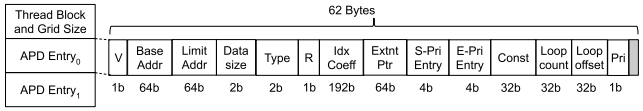


Figure 6: Access pattern directory (APD) and entries.

array is also provided. `Edges` is registered with `Nodes` as both its starting and ending primary arrays to indicate a CSR like format. `Nodes` and `Edges` are set as primary arrays during registration. The `taf_setRdy()` call is used to mark the arrays ready. Unlike thread ID-dependent arrays, data-dependent arrays are marked ready only after its primary array is initialized, as can be seen for `Edge` and `Cost`. `Nodes` and `Edges` are assigned values within the loop, details of which have been omitted for simplicity. The values stored to primary arrays (`Nodes` and `Edges`) are tracked before kernel launch as explained later. Once the entire primary array has been initialized and tracked, the TAF of the dependent arrays can be estimated.

Access Pattern Directory Our framework maintains API-annotated information pertaining to individual access patterns and their TAF relations within APD entries. Fig. 6 shows the APD organization and fields within each entry are described in Table 2. The APD first stores kernel launch parameters (grid and threadblock size) to be

Table 2: APD Entries.

Field	Description
V	Valid bit.
Base Addr	Starting virtual address.
Limit Addr	Ending virtual address.
Data Size	Array element size (8b/16b/32b/64b).
Type	Access pattern. (TID_DEP/INDIR/INDIR_RANGE).
R	Bit indicating information is ready to estimate TAF.
Idx Coeff	Coefficients **m**.
Extnt Ptr	Pointer to Extent table.
S-Pri Entry	Index of APD entry of primary array with starting indices.
E-Pri Entry	Index of APD entry of primary array with ending indices.
Const	Constant c.
Loop Count	Iteration count N.
Loop Offset	Iteration offset *ofs*.
Pri	Bit indicating if the array is primary array.

dimBlock, dimGrid	V	Base Addr	Limit Addr	Data Size	Type	R	Idx Coeff	Extnt Ptr	S-Pri Entry	E-Pri Entry	Const	Loop count	ofs	Pri
Nodes Entry	1	Nodes	Nodes + (numNodes+1)*4	2'b10	TID_DEP	1	[1, 0, 0, THREADBLOCK_SIZE, 0, 0]	$Addr_0$	-	-	0	0	0	1
Edges Entry	1	Edges	Edges + numEdges*4	2'b10	INDIR_ RANGE	1	NA	$Addr_1$	x0	x0	0	0	0	1
Cost Entry	1	Cost	Cost + numNodes*4	2'b10	INDIR	1	NA	-	x1	-	0	0	0	0

Figure 7: CSR example: populated APD table.

used as inputs to TAF relations. It further consists of multiple entries, where each entry is used to capture a specific access pattern of a data structure. If a data structure is accessed with multiple patterns, it can be registered multiple times for different patterns. The APD is stored in memory (where the APD start address, for example, is contained in the Process Control Block), such that it can be accessed during API calls. Each APD entry fits into one cacheline allowing for efficient access.

The populated APD table after array registration for the *Bfs* example is shown in Fig. 7. In addition to the *dimBlock* and *dimGrid* dimensions stored in the header, the APD contains 3 entries, one for each array (`Nodes, Edges and Cost`), which are populated with the information provided during their corresponding (`taf_regsiter()`) calls. In addition, all R bits are set (through `taf_setRdy()` calls), indicating that information required to evaluate their TAF is ready.

4.2 Dynamic Data Dependency Tracking

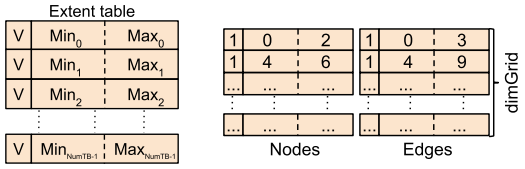
Data-dependent access patterns use values stored in primary arrays to compute the indices for secondary arrays. This requires capturing the relationship between thread indices and primary array values. However, if not carefully done, the overhead involved in tracking primary array values can grow proportional to the array size, E.g. consider a data-dependent access `B[A[tid]]`. Here, the index of array `B` accessed by thread `tid` is `A[tid]`. To track the indices of `B` accessed by a given threadblock TB, all `A[`tid`]`, $tid \in TB$ need to be tracked. This overhead exacerbates in case of nested data-dependent accesses.

Instead of tracking the relation between primary array values and their indices and later relating thread IDs to these indices, we can reformulate the problem as directly tracking the primary array values referenced by every threadblock or group of threadblocks. This problem is similar to the one faced in file systems. A file system directory also faces the challenge of tracking disk block IDs pertaining to individual files while keeping the overhead small to reduce file system wastage. The XFS file system [50] specifically uses extents for managing space. An extent corresponds to one or more adjacent blocks on the disk. Instead of tracking each individual block using bitmaps or a linked allocation, a B+ tree of extents is maintained to indicate which blocks belong to a file or the free list.

Extent Table We propose an extent-based mechanism inspired by XFS to capture the values stored in primary arrays without enduring overhead proportional to the array size. Instead of a tree, we use an extent table (ET) per primary array, organized as list of extents stored in memory as shown in Fig 8a. Since GPUs perform scheduling and mapping at threadblock granularity, we track the primary array values for each threadblock or threadblock group. An extent captures the range of values in the primary array accessed by the threads in one threadblock or threadblock group. Extents contain a valid byte and the tuple of minimum and maximum values stored. To limit storage overhead, we track primary array values using a single extent per threadblock. Stored value is inserted into the extent corresponding to the threadblock which will access the index into which the value is being stored. There is an extent table for every primary array to track the values stored, allowing TAFE to support nested input-data dependent access patterns, since the extent table of immediate primary array is sufficient for estimating the address footprint.

Primary Array Tracking The values stored in primary arrays have to be processed to populate the extent table appropriately. This can be done either by re-reading the primary arrays prior to kernel launch or by overloading their initialization and avoiding additional loads. The approach involving re-reading primary arrays requires 1) iterating across all the thread IDs in the GPU kernel, 2) computing the TAF relations to obtain the indices accessed by each thread, and 3) loading the elements from the indices and inserting them into their corresponding extent. In case of thread ID-dependent primary array, the range of array indices accessed by every threadblock (or threadblock group) is calculated using the attributes from the APD entry and kernel launch parameters. For data-dependent primary arrays, the mapping between threadblock IDs and range of indices accessed by them is present in the extent entries of their nested primary array. Re-reading of primary arrays can be performed as part of `taf_setRdy()` calls before the secondary array is even initialized. Re-reading of primary arrays can be done on the host CPU or by exploiting parallelism (using an additional kernel) on the GPU itself. The latter also supports cases when primary arrays are initialized on the GPU.

Alternatively, primary array stores can be overloaded to track the stored values without the need for re-reading values from memory. On every overloaded store, this will require 1) computing the array element index into which the value is stored, 2) computing the inverse TAF relation to obtain the index to threadblock mapping, and 3) inserting the value into the corresponding threadblock extent. Array element indices can be computed using their base address and element size (from APD entries). Inverse TAF computation, by contrast, is more complex and can only be evaluated if the TAF relation is invertible, i.e. each primary array index is accessed by a single threadblock. If the TAF relation is non-invertible, we track only one of the threadblocks and other accesses will be false negatives when using overloading. Inverse TAF computation otherwise depends on the access pattern of the primary array. On every store, the element offset (primary array index) is compared against



(a) Layout. (b) CSR example.

Figure 8: Extent table.

the range of array indices corresponding to each threadblock. For thread ID-dependent primary arrays, the range of array indices accessed can be computed from attributes in the APD entry. By contrast for data-dependent primary arrays, the required indices are present in the nested extent entries. On a match, the threadblock ID is selected. This approach works, but results in considerable overhead since the array indices for each threadblock or group need to be computed and compared against the element offset. This penalty scales with the number of threadblocks or groups.

In either case of re-reading or overloading, the stored values are inserted by checking against the extent corresponding to the determined threadblock (or threadblock group) to see if its range already encompasses the value. If not, the extent is expanded accordingly. APD entries of primary arrays contain the starting address of the corresponding extent table (*Extnt Ptr*). ETs for different primary arrays are aligned at cacheline boundaries. The address of the cacheline containing extent for threadblock *TB* can thus be computed as *Extnt Ptr* + ((*TB* / #extents per cacheline) x cacheline size). If the primary array values inserted are not continuous, it can potentially result in over-estimation. *INDIR_RANGE* patterns access contiguous indices, so a single extent is sufficient to perfectly capture them. However, the same is not true for *INDIR* patterns and may result in false positives. For the benchmarks evaluated, we find that 1 extent per threadblock is sufficient and results in only 1.5% average error in TAF estimation. Considering 1 extent per threadblock, a 64B cacheline can hold extents for 7 threadblocks. Each threadblock takes $1 + 2 \times 4 = 9$ bytes of space to store its extent and validity information. For a kernel with 64k threadblocks, each primary array would thus require 0.57MB of memory. Alternatively, stores pertaining to threadblocks assigned to same SM or module can be tracked together reducing the extent table size.

Populated extent tables after initialization of `Nodes` and `Edges` arrays in the *Bfs* example are shown in Fig. 8b, assuming each threadblock consists of 2 threads accessing 2 elements of `Nodes` (i.e., *THREADBLOCK_SIZE* in Listing 1 is 2). Since threadblock 0 will access `Nodes[0-1]`, its first extent in the `Nodes` table covers the value range $(0, 2)$. Similarly, threadblock 1 accesses `Nodes[2-3]` with extent $(4, 6)$. Values in `Nodes` in turn determine the starting indices of the indirect ranges that threadblocks access in `Edges`, where the end of a threadblock's range is determined by the start of the next threadblock's range. Threadblocks 0 and 1 will access `Edges[0-3]` and `Edges[4-7]` resulting in extents $(0, 3)$ and $(4, 9)$, respectively. Indices for `Cost` are determined by the minimum and maximum value of the corresponding threadblock's extent, e.g. threadblock 0 and 1 will be assumed to access `Cost[0-3]` and `Cost[4-9]`. Since the values stored in `Edges[4-7]` are not continuous (see Fig. 5), it results in false positives (`Cost[6]` and `Cost[7]`).

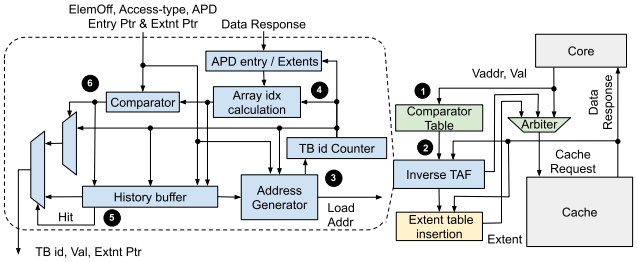


Figure 9: TAFE hardware for tracking primary arrays.

4.3 TAFE Hardware Support

As described previously, primary array values can be tracked purely in software in the host CPU or GPU. This, however, requires software modifications and incurs overhead. We propose an optional lightweight hardware extension to support transparent tracking of primary array stores and reduce software overhead. Arrays are transparently tracked while they are being initialized avoiding additional loads. Optional TAFE hardware extensions are inserted between the cache and core as shown in Fig. 9. They can be potentially combined with prefetchers by sharing logic and functional units [18, 59]. TAFE hardware extension consist of components for comparison, inverse TAF computation and ET insertion.

Comparator Table The comparator table is responsible for detecting stores to primary arrays and computing the element index. As part of registration (`taf_register()`) calls for primary arrays, the framework populates the primary array's base and limit address, element size, access type and a pointer (APD entry pointer for TID_DEP or extent pointer otherwise) into a hardware comparator table. If the primary array itself has a nested data-dependent access pattern, the *Extnt Ptr* of the nested, second-level primary array is also stored. APD entries of primary arrays can be pinned into the cache until kernel launch as part of array registrations for faster access. A single table entry can be reused for tracking multiple primary arrays as long as their initialization is not interleaved, i.e. sequential. An entry is reserved during the registration call of primary array and freed again as a part of the secondary array's `taf_setRdy()` call. If initializations are not sequential, the size of the comparator table determines the maximum number of primary arrays that can be tracked simultaneously in the system. Our results show that tracking 2 primary arrays is sufficient for the benchmarks evaluated.

Every time a store reaches L1 cache, the address is then compared against the address ranges in the comparator table ❶. An address within a tracked range indicates a store to a primary array, and the element offset is computed using the base address and element size. The element offset, value to be stored, access type, pointer (APD entry/extent pointer) and optional nested extent pointer are then passed to an inverse TAF block ❷.

Inverse TAF Computation This block computes the inverse TAF to determine the correct ET entry to be updated. As described previously, this requires computing the index ranges for each threadblock and matching them against the primary array index. In order to reduce the overhead, we exploit the fact that primary array initialization is mostly performed sequentially (within a loop). This means that the threadblock accessing the index on which a store is performed will mostly be same as that of the previous store. Further,

```
if( threadIdx.x >= i+1 && threadIdx.x <= BLOCKSIZE-i-2)) {
  idx = blockIdx.x*sbcols + threadIdx.x + cols*(start+i);
  result[threadIdx.x] = shortest + gpuWall[idx - b]; }
```

Listing 2: Control divergence example (Pathfinder).

we assume that the subsequent primary array indices are accessed by threadblocks in incremental order, i.e. the indices accessed by a threadblock are larger than those accessed by preceding threadblocks (e.g. in TID_DEP, INDIR_RANGE primary access patterns). This avoids the need of iterating across all threadblocks. We use a single-entry history buffer capable of storing one range of array indices (extent) and its corresponding threadblock ID. On the first store to a primary array, the range of array indices are computed for the first threadblock (or threadblock group). In case of a thread ID-dependent primary array, its APD entry is first loaded ❸ and the attributes from the APD entry are used to calculate the range indices ❹. For a data-dependent array, the extent of its nested primary array is loaded (using the extent pointer from comparator metadata) to obtain the indices. The range of indices and the threadblock ID are inserted into the history buffer. On subsequent stores, the primary index is compared against the entry in the history buffer ❺. If it is a hit, the matching overhead is avoided. On a miss, the threadblock ID is incremented and a new range of indices is computed and compared against the element offset ❻ before being updated in history buffer. In our experiments, we observe a >99% history buffer hit rate thereby significantly reducing the overhead.

Extent Table Insertion Finally, the threadblock ID, primary array extent pointer along with value being stored are passed to an extent table insertion block, where the value is inserted into the appropriate entry within the extent table of the primary array. A write-combining buffer is used to merge writes to the same extent reducing the cache transactions.

4.4 Discussion

Benchmarks often have unique access patterns that will lead to estimation inaccuracies when capturing them with our framework. In the following, we discuss some of such application characteristics. While modifying source code is always a solution, we present possible approximations.

Section 3 describes access patterns directly supported by our current TAFE framework. As mentioned there, other patterns can be approximated using a combination of supported patterns to achieve the tightest possible fit. Patterns can be over- or under-approximated depending on the use-case requirements, e.g. to capture non-linear relations in linear form. Consider an access pattern `A[tid*4 + j + i*N]`, $0 \leq j < 4$, $0 \leq i < I$. The tightest fit can be achieved by under-approximating this relation as $\{tid \cdot 4 + i \cdot N \mid 0 \leq i < N\}$. Patterns like `A[tid % N]`, where $N_t > N$ is the total threads in the kernel, can be represented as thread ID dependent relation of the form $\{tid + i \cdot (-N) \mid 0 \leq i < (N_t/N)\}$. Evaluating this relation will result in underflow of indices/addresses, which are filtered by TAFE using *Base* and *Limit* APD entries.

Our current implementation supports loop structures, but only in thread-ID dependent accesses at level 0. Branching behavior with control divergence can be over-approximated by capturing relations for all possible branches. Listing 2 shows an example from the *Pathfinder* benchmark, where *result* and *gpuWall* are only accessed by some threads. We over-estimate this access by registering the TID_DEP patterns in the branch unconditionally for all threads.

In the worst case, unsupported patterns require marking the entire array as part of the footprint. To achieve higher accuracy, TAFE APIs can be extended to natively incorporate additional patterns, including non-linear and multi-input relations, more complex nesting dependencies or loop control structures in nested levels.

Applications with multiple kernels are currently supported in TAFE by tracking the access patterns of the entire application as a whole in a single APD. This can accurately capture the individual kernel patterns if they have the same threadblock and grid dimensions. Applications consisting of kernels with varying dimensions (e.g. *Lud*) can be approximated by either identifying a threadblock and grid dimension that allows non-contradictory mappings across kernels or using the dominant kernel dimensions. Alternatively, TAFE can be extended to support kernels with different dimensions by tracking each kernel separately (requires an APD per kernel).

Our current TAFE implementation also has some limitations beyond accuracy effects. Obtaining TAFs requires primary arrays to be initialized before and not modified during kernel execution. Furthermore, our proposed optional hardware tracking currently only supports nested primary array accesses with a single TID_DEP pattern or INDIR_RANGE pattern using the same starting and ending arrays and hence cannot support arbitrary nested accesses.

5 Optimizations using TAFs

The OS/runtime takes the APD and ET information to evaluate the TAF relations and compute the TAFs for different kernel data structures. Given the estimated TAFs, the OS/runtime can determine the set of array indices (in other words, addresses) accessed by each threadblock during the memory allocation phase itself. Based on this information, the runtime or hardware can perform optimizations, e.g. data/compute co-location, cache/scratchpad memory management. In this paper, we evaluate the benefit of using TAFs for co-locating threadblocks/pages in NUMA GPU systems.

5.1 Threadblock Mapping

Prior work has shown that mapping contiguous threadblocks within SMs benefits locality [41]. However, for multi-dimensional kernel grids, to capture maximum locality, threadblocks accessing the closest array elements should be mapped together. For thread ID-dependent arrays, we map threadblocks in row-, column- or depth-major order depending on the relative step size defined by block ID coefficients along x, y and z dimensions. Threadblocks are equally divided across SMs for load-balancing. Since most applications have similar trends across different arrays, the decision can be made based on the array registered first. Data is later mapped/allocated to complement the determined threadblock mapping.

If data is already mapped before threadblock mapping is decided (e.g. threadblock remapping across multiple kernel calls), threadblocks should be mapped to the module containing pages accessed by them. The module to which a page is mapped can be identified from its physical address, requiring a address translation per page or OS support as done in [51]. Using the page mapping and TAFs, appropriate threadblocks can be co-located with their data-pages.

For kernels with data-dependent arrays, extent entries provide the range of indices accessed by each threadblock. This information

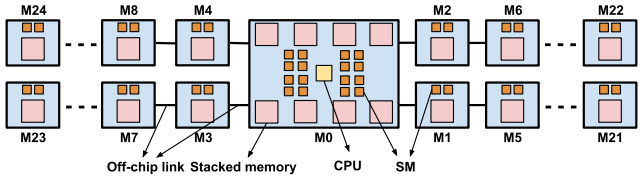


Figure 10: Simulated exascale node configuration.

can be used to find groups of threadblocks with maximum locality. However, this incurs higher extent table size (proportional to threadblock count). Alternatively, a specific order (e.g. x-dimension first) can be selected, allowing the runtime to determine the exact threadblocks mapped to each module. The primary array values now can be tracked per module instead of every threadblock, reducing the extent table size overhead (proportional to the number of modules). However, using such coarse grain tracking would mean relying solely on data mapping to reduce off-chip accesses.

5.2 Data Allocation and Mapping

Once the threadblock mapping has been determined (coefficient-based or pre-selected), the runtime uses estimated TAFs to allocate memory to modules while reducing off-chip data movement. This requires computing TAFs and performing page allocation before arrays are initialized on the host CPU as doing otherwise would require remapping of pages, which defeats the purpose of optimizing the mapping before kernel launch. TAFE supports TAF estimation and optimal page mapping prior to host initialization as all the required information is available during `taf_setRdy()` calls.

Since the focus of this work is not on data partitioning heuristic, we use a straightforward allocation algorithm: If a page is accessed by only one module, it is allocated in that specific module. If a page is accessed by multiple modules, it is mapped to the module with the shortest distance to all other accessing modules. All page mappings are subject to memory capacity restrictions. A qualitative measure of temporal locality can easily be passed to TAFE by the user as in [55]. Since the mapping and allocation is done during a runtime call, it allows different partitioning and mapping schemes of varying complexity to be employed [19, 28, 29, 33].

6 Evaluation

We evaluate TAFE for 2 different NUMA GPU system configurations: 1) A multi-GPU/-chiplet system with identical modules interconnected by off-chip links and 2) a heterogeneous multi-chiplet module (MCM) configuration representing recently proposed exascale node architectures (Fig. 10) [38, 47, 54]. The configurations consist of a host CPU and GPU with unified memory distributed across interconnected modules. Details of our simulated configurations are provided in Table 3. We use gem5-gpu [46] (VI_hammer_fusion) for simulating our system. We augmented the CUDA runtime to include APIs required by TAFE. Support for the new APIs and for TAFE hardware modifications are made in the simulator. We model an APD with 16 entries, extent tables with 1 extent per threadblock, and a comparator table with 2 entries. The memory allocated by the application is uniformly distributed across HMC stacks.

We evaluated TAFE for 12 applications mostly from the *Rodinia-nocopy* [17] benchmark suite. The list of applications including lines of code (LOC) added for TAFE annotations and storage overhead of

Table 3: Simulation parameters.

Configuration 1: Multi-GPU/-chiplet	
NUMA modules	8 (all identical)
SM count	64 (8 per module)
Memory stacks	8 (1 per module)
Topology	Tree
Configuration 2: Exascale node	
NUMA modules	25
SM count	64 (16 in M0, 2 each in others)
Memory stacks	32 (8 in M0, 1 each in others)
Topology	Ext: Tree, Central pkg: Fully connected
Memory	
Bandwidth	Local: 160 GB/s, off-chip: 80 GB/s [26]
Memory model	HMC_2500 [11, 57]
Core Configuration	
CPU	OOO model, 2Ghz
SM	1.4Ghz, 48kB shared memory [2]
Cache	L1:16kB, 4-way (pvt.), L2:1MB, 16-way (shared)

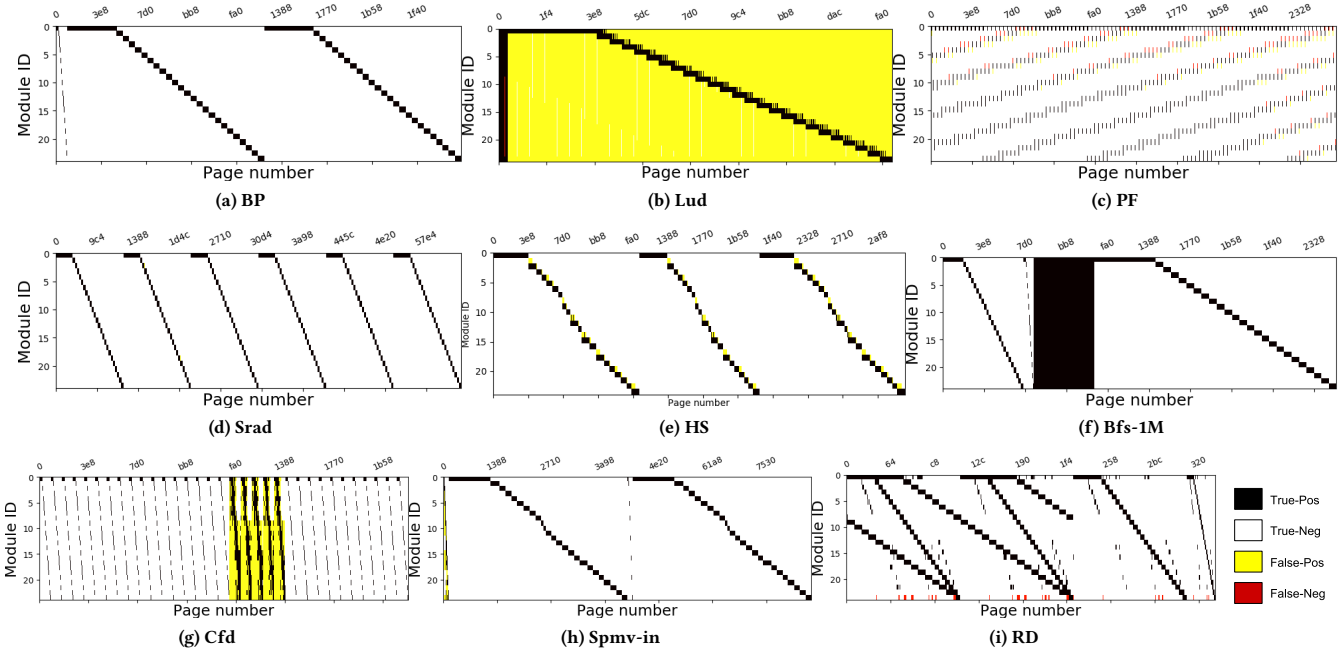
Table 4: Benchmarks.

Benchmark	Input	Access Pattern	TAFE Storage	TAFE LOC
Backprop (BP) [17]	256k	TID_DEP	256 B	25
Lud [17]	2048	TID_DEP	192 B	26
NN [17]	640k	TID_DEP	128 B	13
Pathfinder (PF) [17]	Default [46]	TID_DEP	192 B	19
Srad [17]	Default [46]	TID_DEP	768 B	32
Hotspot (HS) [17]	4096	TID_DEP	192 B	13
Bfs-1M [17]	1M nodes	TID_DEP, INDIR & INDIR_RANGE	1136 B	40
Bfs-4M [17]	4M nodes	TID_DEP, INDIR & INDIR_RANGE	2.8 kB	40
Cfd [17]	0.2M	TID_DEP & INDIR	942 B	52
Spmv-in [8]	inline [1]	TID_DEP, INDIR & INDIR_RANGE	4.7 kB	42
Spmv-ser [8]	serena [1]	TID_DEP, INDIR & INDIR_RANGE	12.4 kB	42
Radix Sort (RD) [24]	256k	TID_DEP & INDIR	1098 B	27

TAFE metadata (APD and extent tables) are shown in Table 4. The applications are written/compiled in CUDA v3.2. Radix sort [24] is executes the prefix sum in the host machine instead of the GPU. The applications are annotated with the TAFE API calls to obtain the required information while maintaining the original functionality. Except for over-approximations to capture control divergence in *Hotspot*, *Lud* and *Pathfinder* (with 1, 4 and 1 instances, respectively), all patterns encountered in benchmarks are directly supported by TAFE. Results are obtained by running each application (in addition to all host CPU instructions prior to first kernel launch) for 1B GPU instructions or to completion, whichever occurs first.

6.1 Address Footprint Accuracy

The primary purpose of TAFE is to estimate address footprints accurately prior to kernel launch. We begin by studying the accuracy with which TAFE is able to estimate the address footprints for different benchmarks. Evaluated benchmarks are not affected by limitations of hardware-assisted tracking. As such, software- and hardware-based tracking have the same accuracy. Fig. 11 shows TAFE accuracy in form of a color map for Exascale node configuration. *NN* has a very similar color map as *Backprop*, *Bfs-4M* and *Spmv-ser* have similar characteristics as *Bfs-1M* and *Spmv-in* respectively and are omitted for space reasons. For every combination of page and module, we show when TAFE correctly estimated whether a page is actually accessed by a module (*true-pos*, shown in black) or not (*true-neg*, shown in white). We also show pages that were accessed but that TAFE captured otherwise (*false-neg*, shown in red),



(a) BP (b) Lud (c) PF (d) Srad (e) HS (f) Bfs-1M (g) Cfd (h) Spmv-in (i) RD

Figure 11: TAF tracking accuracy.

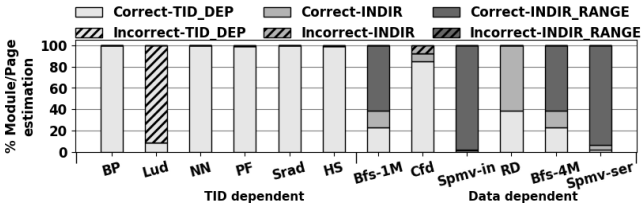


Figure 12: TAF estimation accuracy.

and pages not accessed but tracked by TAFE as accessed (*false-pos*, shown in yellow). The first 2 categories indicate correct tracking by TAFE, while the latter 2 indicate wrong tracking.

It can be seen that for most benchmarks, TAFE is able to track the address footprint accurately, including those with data-dependent accesses. The quantitative distribution of TAFE's estimation accuracy by access patterns for configuration 1 is further summarized in Fig. 12. TAFE has on average 91% accurate estimations across all benchmarks. In data-dependent benchmarks, TAFE can estimate TAFs with on average 1.5% (and up to 8%) error. Most errors are due to conservative over-estimation under control divergence (*Pathfinder, Hotspot*) or limits of extent-based data tracking (*Cfd, Spmv-in*). *Lud* has a significant incorrect estimations because it has multiple kernel calls with varying launch parameters and control divergence over memory accesses, which does not accurately map to the patterns currently supported by TAFE.

We measured the average value sparsity across indirectly (INDIR) accessed array indices for irregular graph and sparse matrix benchmarks. The value sparsity $1 - \frac{Num.\ indices\ accessed}{(Max.\ index\ value - Min.\ index\ value)}$ for *Bfs-1M, Bfs-4M, Spmv-in and Spmv-ser* is 0.98, 0.99, 0.95 and 0.99, respectively. Note that TAFE accuracy ultimately depends on the granularity at which TAFs are requested. TAFE achieves high accuracy at page/module granularity for sparse applications since the probability that at least one word from each page within the range will be accessed is high, resulting in a near-contiguous range.

6.2 TAFE Overhead

In this section, we discuss the overhead of implementing TAFE. Estimating TAFs for thread ID-dependent access patterns can be done using static annotations and incurs overhead of API calls and memory space to store the APD table. Data-dependent access patterns can be tracked either in software or using optional hardware as described in Section 4. For software, we use re-reading of immediate primary arrays to process the values stored. This approach is more flexible and avoids limitations of inverse TAF computations. We evaluate overhead of software tracking in the host CPU or GPU and hardware-assisted tracking in the CPU.

Table 5 summarizes the cost for TAFE hardware support. We use McPAT [35] to estimate the area overhead to be 2% of an Alpha21364 reference host core. Existing prefetcher components can be shared with TAFE tracking logic to further reduce the hardware overhead. CPU prefetchers [18, 59] already include hardware for tracking cache accesses, functional units for computing prefetch addresses and issue loads. Note that TAFE hardware overhead is independent of the number of SMs and data size, and the information obtained can be reused for different optimizations. The 3 stages for tracking stores (comparator table, inverse TAF and ET insertion) can happen in parallel to cache accesses and can be pipelined to support best-case throughput of 1 store per cycle on history buffer hits. Only one extra arbiter delay is added on the critical path of cache accesses. In rare cases of history buffer misses, the tracking pipeline and cache accesses need to be stalled until the miss is served. We assume an inverse TAF miss delay of 20 cycles for our evaluations.

Table 5: TAFE hardware costs.

Component	Storage	Logic	Delay
Comparator Table	68 B	4 CMP, 1 ADD, 1 SHFT	1 cyc. (2 pipeline stages)
Inverse TAF	133 B	2 MUL, 2 ADD, 1 counter, 1 CMP	Hit: 1 cyc., Miss: 8 cyc. + load
ET Insertion	34 B	2 CMP, 1 ADD	1 cyc.
Pipeline	76 B	control/stall logic	-

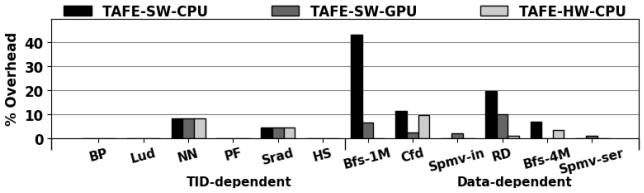


Figure 13: TAFE runtime overhead.

Fig. 13 compares the TAFE overhead when implemented in software within the host CPU (TAFE-SW-CPU) or GPU (TAFE-SW-GPU) and when using optional hardware support in the host CPU (TAFE-HW-CPU) for exascale node configuration. Overhead is compared against host CPU runtime without TAFE. For thread ID-dependent benchmarks that do not require primary array tracking, both hardware and software versions of TAFE incur the same small overhead for API calls. For data-dependent benchmarks, TAFE-SW-CPU in experiences larger overhead (up to 43%) for many benchmarks (*Bfs-1M, Cfd, RD*) due to the inability of the host CPU to hide the latency. TAFE-SW-GPU has low overhead (3% on average and max. of 10%) for most cases, indicating that TAFE-SW-GPU is an efficient and also more flexible approach to track primary array values. TAFE-HW-CPU allows transparent tracking and incurs the lowest overhead (2% on average), but requires extra hardware.

6.3 TAFE-Based Optimizations

We further study the benefits of utilizing TAFE for data/thread co-location for both system configurations. We model page co-location with compute by swapping pages prior to writing data rather than modifying memory allocation to ensure all untracked pages are in their default locations. We compare our framework against a *Base* configuration that has no knowledge about data/thread locality and uses the default threadblock scheduling and page mapping. Furthermore, we evaluate the benefits of tracking data-dependent access patterns (*Bfs-1M, Cfd, Spmv-in, Radix, Bfs-4M and Spmv-ser*) by comparing against a TAFE-TID scheme, which only tracks static, thread ID-dependent access patterns. This is similar to prior work [55]. In TAFE-TID, data-dependent data structures are treated as being accessed by all modules.

Off-Chip Link Traffic Fig. 14 shows the off-chip link traffic across different benchmarks normalized against *Base* for both configurations. The off-chip link traffic is the total amount of data moved across all off-chip (inter-module) links during the course of the application. As expected, the overall traffic decreases due to improved mapping. On average, TAFE reduces the off-chip link traffic by by 33% (up to 80%) for the multi-GPU/-chiplet and 23% (and up to 62%) for exascale node configuration. In data-dependent benchmarks, TAFE reduces traffic by 34% and 24% on average while TAFE-TID reduces it by 13% and 10%, respectively.

Warp Memory-Load Latency Fig. 15 compares the average time taken by a warp-instruction reading global memory to complete (all 32 thread requests must complete) for both configurations. For the multi-GPU/-chiplet case, both TAFE and TAFE-TID achieve similar reduction in warp memory-load latency (on average 21%), but in some cases TAFE-TID achieves higher reduction than TAFE (e.g. *Bfs-1M, Cfd*). This is because the latency is dominated by on-chip accesses for such cases. For exascale node system, TAFE-TID reduces warp memory-load latency by 29% on average while TAFE reduces it by 36%. In data-dependent benchmarks, TAFE-TID and TAFE achieve 16% and 30% reduction, respectively.

Performance Fig. 16 shows the average SM IPC across different benchmarks normalized against the *Base* case for both configurations. On an average, TAFE provides 45% (up to 2.1x) and 32% (and up to 2x) IPC improvement for multi-GPU/-chiplet and exascale node cases, respectively. This improvement is both because of fewer off-chip accesses and lower off-chip link congestion. In multi-GPU/-chiplet case, TAFE achieves on avg. 10% (up to 32%) higher performance than TAFE-TID for data-dependent benchmarks, while it performs 22% (up to 31%) better than TAFE-TID in exascale node case. The improvement is lower in multi-GPU/-chiplet system because the off-chip accesses have lower performance impact due to the smaller number of distributed modules.

7 Related work

Many prior works have investigated data/thread co-location, shared memory management, compiler analysis or programmer annotations to optimize data/thread locality.

A wide range of programming languages have been proposed to allow explicit expression of locality [13, 15, 16, 22, 36, 49, 53, 58]. Such approaches require the programmers to be familiar with a new language and re-write the application. By contrast, TAFE can be applied over existing programming models, requiring programmers to only provide simple hints on how existing data structures are accessed without affecting the functionality of the application.

Several works have proposed means to allow programmers to provide hints about access patterns. [55] presents a way to express data locality within applications. The authors propose descriptors consisting of *C-tiles* and *D-tiles* to let programmers explicitly specify the inter- and intra-thread locality, which is then utilized to perform data and compute mapping for NUMA systems. Similarly, [14] presents user-level APIs allowing the programmer to specify access patterns. These works are able to capture static, thread ID-dependent accesses, but they do not support complex dynamic, data-dependent accesses. While TAFE also relies on programmer annotated APIs, it does not require explicit mapping and data sharing information from programmer like[55] and can capture dynamic data-dependent access patterns. [48] handles specific complex access patterns using an *index mapper* that performs analyses by re-reading the entire primary data structure. This solution was proposed for discrete GPU systems, where reloading primary array by the host will result only in local accesses. However, such an approach will incur significant overhead in systems with unified memory architectures, limiting scalability. TAFE can handle a wider range of access patterns precisely with little overhead even for unified memory architecture systems.

Many prior works [21, 40, 42, 44, 52] use compiler annotations or hardware profiling to dynamically move pages in NUMA systems. Re-mapping pages at kernel execution time becomes infeasible as

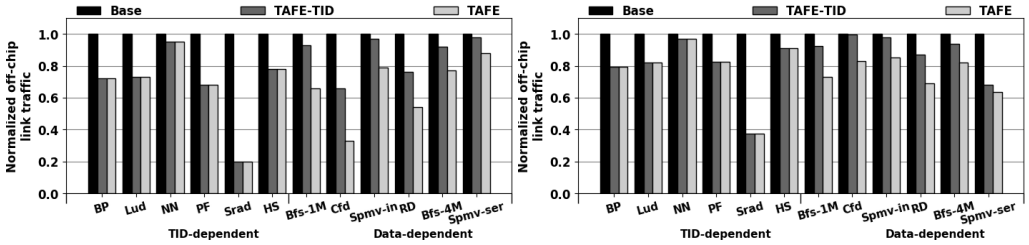


Figure 14: Off-chip traffic. Left: Multi-GPU/-chiplet, Right: Exascale node

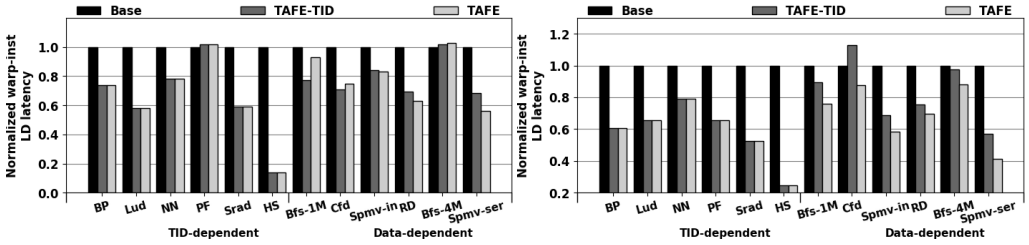


Figure 15: Warp memory load latency. Left: Multi-GPU/-chiplet, Right: Exascale node

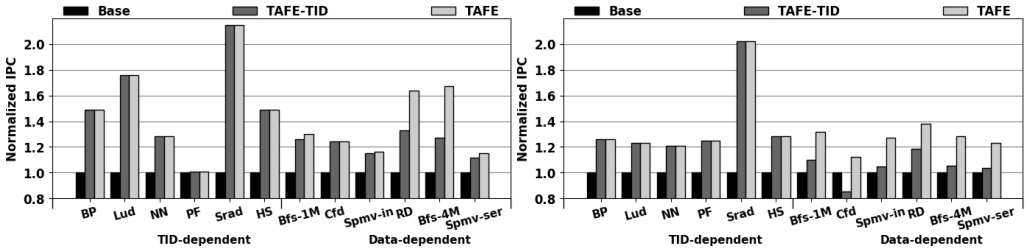


Figure 16: IPC. Left: Multi-GPU/-chiplet, Right: Exascale node

the system size scales. TAFE allows appropriately mapping pages during data allocation and does not rely on dynamic re-mapping of pages. [26] proposes a data mapping solution for discrete memory systems where the access patterns are profiled to identify the mapping, and data copy to device memory is delayed for that duration. Such a solution cannot be applied in systems with unified memory between host and device (like in [54]). TAF enables deciding the optimal/near-optimal mapping just-in-time during data allocation, and is therefore applicable to systems with both unified and discrete memory. Similarly, compiler and runtime support to allow thread remapping at runtime has been proposed [10, 32, 51]. But like dynamic page re-mapping, re-mapping GPU threadblocks during kernel execution incurs overhead and will limit the scalability.

Prior works [34, 37, 43] have proposed compiler analysis but unlike TAFE they don't support data-dependent accesses.

8 Summary and Conclusions

In this paper, we target the problem of estimating, in a scalable manner, the relationship between threads and both static, data-independent and dynamic, data-dependent addresses accessed by them prior to kernel launches in GPGPU systems. We introduce the concept of thread address footprints (TAFs) and TAF relations as a means to concisely represent this relationship. We propose TAFE, a framework for evaluating TAF relations and estimating TAFs. Our framework constructs TAF relations using a combination of static attributes and dynamic data dependencies collected through application code and dependency tracking. We show that dynamic data dependencies can be tracked in GPU software with low overhead and further provide an optional hardware extention to capture data-dependencies. The TAFs can be used by a OS/runtime to compute TAFs prior to kernel launch. We evaluate TAFE's accuracy and demonstrate its benefits by using TAF information for data/threadblock co-location for multiple NUMA GPU system configurations. However, TAFE's benefits are not limited to aforementioned optimizations. Address footprints can also be used for other optimizations like prefetching pages in unified, e.g. CUDA-managed memory systems, transformations of irregular patterns into regular remote memory accesses via local page caching, or reducing coherency traffic [12, 55, 60]. Our evaluations show that TAFE achieves 91% estimation accuracy across all benchmarks. Moreover, TAFE can accurately track data-dependent access patterns (only 1.5% average error) while incurring low (3%) overhead. Performing page/threadblocking mapping using TAFE improves performance by 45% and 32%. For data-dependent applications, TAFE assisted mapping can achieve 10% and 22% speedup compared to data-independent schemes across different configurations.

Acknowledgments

The authors would like to thank the reviewers for their comments. This work was partially supported by the National Science Foundation (NSF) under grant CCF-1725743.

References

[1] [n.d.]. SuiteSparse Matrix Collection. https://sparse.tamu.edu/.

[2] 2009. *GPGPUSim v3.2.2. GTX 480 Configuration.*

[3] 2016. clSPARSE documentation. http://clmathlibraries.github.io/clSPARSE/index.html.

[4] 2017. CUDA OPTIMIZATION TIPS, TRICKS AND TECHNIQUES. http://on-demand.gputechconf.com/gtc/2017/presentation/s7122-stephen-jones-cuda-optimization-tips-tricks-and-techniques.pdf.

[5] 2019. Intel's Xe for HPC: Ponte Vecchio with Chiplets, EMIB, and Foveros on 7nm. https://www.anandtech.com/show/15119/intels-xe-for-hpc-ponte-vecchio-with-chiplets-emib-and-foveros-on-7nm-coming-2021.

[6] 2019. NVIDIA cuSPARSE CUDA Toolkit documentation. https://docs.nvidia.com/cuda/cusparse/index.html.

[7] 2019. NVIDIA nvGRAPH CUDA Toolkit documentation. https://docs.nvidia.com/cuda/nvgraph/index.html.

[8] A. Aithal and S. Bharadwaj. 2018. Sparse-Matrix-Vector-Multiplication documentation. https://github.com/aneesh297/Sparse-Matrix-Vector-Multiplication.

[9] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C. J. Wu, and D. Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *International Symposium on Computer Architecture (ISCA)*. 320–332.

[10] S. Augonnet, C.and Thibault and R. Namyst. 2010. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *International Conference on Parallel Processing (ICPP)*. 56–65.

[11] E. Azarkhish, C Pfister, D. Rossi, I. Loi, and L. Benini. 2017. Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 1 (Jan. 2017).

[12] A. Basu, S. Puthoor, S. Che, and B. M. Beckmann. 2016. Software Assisted Hardware Cache Coherence for Heterogeneous Processors. In *International Symposium on Memory Systems (MEMSYS)*. 279–288.

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[14] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin. 2015. Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[15] B. L. Chamberlain, D. Callahan, and H. P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 519–538.

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

[18] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. 2004. A General Framework for Prefetch Scheduling in Linked Data Structures and Its Application to Multi-Chain Prefetching. *ACM Trans. Comput. Syst.* 22, 2 (May 2004), 214–280. https://doi.org/10.1145/986533.986536

[19] A. T. Chronopoulos, D. Grosu, A. M. Wissink, M. Benche, and J. Liu. 2003. An efficient 3D grid based scheduling for heterogeneous systems. *J. Parallel Distrib. Comput.* 63, 9 (Sept. 2003), 827–837. https://doi.org/10.1016/S0743-7315(03)00112-6

[20] Hoang-Vu Dang and Bertil Schmidt. 2012. The Sliced COO Format for Sparse Matrix-Vector Multiplication on CUDA-enabled GPUs. In *International Conference on Computational Science (ICCS)*.

[21] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 381–394.

[22] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *ACM/IEEE Conference on Supercomputing (SC)*. 4–4.

[23] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *International Conference on High Performance Computing (HiPC)*. 197–208.

[24] M. Harris, S. Sengupta, and J. D. Owens. [n.d.]. CUDA-based Parallel Radix Sort. http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/radix_sort_cuda.cc.

[25] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. 2007. Efficient Gather and Scatter Operations on Graphics Processors. In *ACM/IEEE Conference on Supercomputing (SC)*. 1–12.

[26] K. Hseih, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems. In *International Symposium on Computer Architecture (ISCA)*. 204–216.

[27] X. Jian, P. K. Hanumolu, and R. Kumar. 2017. Understanding and Optimizing Power Consumption in Memory Networks. In *International Symposium on High Performance Computer Architecture (HPCA)*. 229–240.

[28] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392.

[29] H. Khaleghzadeh, H. Deldari, R. Reddy, and A. Lastovetsky. 2018. Hierarchical Multicore Thread Mapping via Estimation of Remote Communication. *J. Supercomput.* 74, 3 (March 2018), 1321–1340. https://doi.org/10.1007/s11227-017-2176-6

[30] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram. 2018. CTA-Aware Prefetching and Scheduling for GPU. In *International Parallel and Distributed Processing Symposium (IPDPS)*. 137–148.

[31] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPU. In *International Symposium on Computer Architecture (ISCA)*. 307–319.

[32] K. Kyriakopoulos, A. T. Chronopoulos, and L. Ni. 2007. An optimal scheduling scheme for tiling in distributed systems. In *International Conference on Cluster Computing*. 267–274.

[33] A. L. Lastovetsky and R. Reddy. 2007. Data Partitioning with a Functional Performance Model of Heterogeneous Processors. *The International Journal of High Performance Computing Applications* 21, 1 (2007), 76–90. https://doi.org/10.1177/1094342006074864

[34] C. Li, Y. Yang, Z. Lin, and H. Zhou. 2015. Automatic data placement into GPU on-chip memory resources. In *International Symposium on Code Generation and Optimization (CGO)*. 23–33.

[35] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman a nd Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture*. 469–480.

[36] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. 2008. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 287–296.

[37] W. Ma and G. Agrawal. 2010. An integer programming framework for optimizing shared memory use on GPUs. In *International Conference on High Performance Computing*. 1–10.

[38] NNSA ASC 2014 PI MEETING. 2014. The AMD FASTFORWARD Project. https://asc.llnl.gov/fastforward/AMD-FF.pdf.

[39] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 117–128.

[40] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *International Symposium on High Performance Computer Architecture (HPCA)*. 126–136.

[41] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans. 2017. Beyond the Socket: NUMA-aware GPUs. In *International Symposium on Microarchitecture (MICRO)*. 123–135.

[42] M. Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *International Conference on Parallel Architecture and Compilation (PACT)*. 188–200.

[43] Yunheung Paek, Jay Hoeflinger, and David Padua. 2002. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 1 (Jan. 2002), 65–109. https://doi.org/10.1145/509705.509708

[44] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. 2014. Compiler Support for Selective Page Migration in NUMA Architectures. In *International Conference on Parallel Architectures and Compilation (PACT)*. 369–380.

[45] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G.H. Loh. 2017. There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes. In *International Symposium on Computer Architecture (ISCA)*. 678–690.

[46] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. [n.d.]. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letters* 14, 1 ([n. d.]), 34–36.

[47] K. Punniyamurthy and A. Gerstlauer. 2020. Off-Chip Congestion Management for GPU-based Non-Uniform Processing-in-Memory Networks. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 282–289.

[48] E. Rubin, E. Levy, A. Barak, and T. Ben-Nun. 2014. MAPS: Optimizing Massively Parallel Applications Using Device-Level Memory Abstraction. *ACM Trans. Archit. Code Optim.* 11, 4, Article 44 (Dec. 2014). https://doi.org/10.1145/2680544

[49] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. 2015. Regent: A High-productivity Programming Language for HPC with Logical Regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[50] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. 1996. Scalability in the XFS File System. In *Conference on USENIX Annual Technical Conference*.

[51] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy. 2017. Data Movement Aware Computation Partitioning. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[52] P. Tsai, C. Chen, and D. Sánchez. 2018. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. *International Symposium on Microarchitecture (MICRO)* (2018), 641–654.

[53] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, Ann S. Almgren, and John Shalf. 2016. TiDA: High-Level Programming Abstractions for Data Locality Management. In *International Supercomputing Conference (ISC)*. 116,135.

[54] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. B. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Mesani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan. 2017. Design and Analysis of an APU for Exascale Computing. In *International Symposium on High Performance Computer Architecture (HPCA)*. 85–96.

[55] N. Vijaykumar, E. Ebrihami, K. Hseih, P. B. Gibbons, and O. Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs. In *International Symposium on Computer Architecture (ISCA)*. 829–842.

[56] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1 (Aug. 2017). https://doi.org/10.1145/3108140

[57] C. Weis, A. Mutaal, O. Naji, M. Jung, A. Hansson, and N. Wehn. 2017. DRAMSpec: A High-Level DRAM Timing, Power and Area Exploration Tool. *Int. J. Parallel Program.* 45, 6 (Dec. 2017).

[58] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. 2009. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *International Conference on Languages and Compilers for Parallel Computing (LCPC)*. 172–187.

[59] Chia-Lin Yang and Alvin R. Lebeck. 2001. *The Push Architecture: a Prefetching Framework for Linked-Data Structure.* Ph.D. Dissertation. USA. AAI3095642.

[60] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*. 345–357.