SUV: Static analysis guided Unified Virtual Memory

Pratheek B
Indian Institute of Science
Bengaluru, India
pratheekb@iisc.ac.in

Guilherme Cox
NVIDIA
Santa Clara, USA
gcox@nvidia.com

Jan Vesely
NVIDIA
Westford, USA
jvesely@nvidia.com

Arkaprava Basu
Indian Institute of Science
Bengaluru, India
arkapravab@iisc.ac.in

***Abstract*—Unified Virtual Memory (UVM) eases GPU programming and enables oversubscription of the limited GPU memory capacity. Unfortunately, UVM may cause significant slowdowns due to thrashing of the GPU memory and overheads of page faults, especially under memory oversubscription.**

We propose to leverage high-level memory access patterns of CUDA applications to reduce the overheads of UVM. We create SUV, a hybrid framework that leverages compiler-inferred (static analysis) memory access semantics to make *proactive* memory management decisions where possible and *selectively* leverages runtime page migrations where needed. It can pin data structures entirely or partially on the GPU memory or CPU's DRAM based on their inferred usefulness and automatically issue software prefetches at kernel boundaries. It also selectively lets parts of data structures migrate on demand onto *reserved* HBM capacity at runtime. SUV reduces execution times of a variety of applications by 74% over UVM under memory oversubscription.

***Index Terms*—GPGPU, UVM, static-analysis, access-counters**

I. Introduction

A significant fraction of today's data-intensive computing happens on Graphics Processing Units (GPUs). While the need to quickly process ever larger amounts of data remains unabated, GPU's onboard high-bandwidth memory (HBM [1]) remains capacity-limited. Toward this, technologies such as NVIDIA's Unified Virtual Memory (UVM), originally envisioned to ease programming, enable oversubscribing HBM by leveraging TBs of DRAM capacity of the host CPU.

UVM can, however, incur large slowdowns under memory oversubscription due to overheads of on-demand page migration. It may thrash the HBM by repeatedly migrating pages back and forth between the HBM and the DRAM as it reacts to runtime memory access streams without semantic knowledge.

Prior works focused on runtime hardware and/or software optimizations to limit UVM's overheads. Most prominent among them include prefetching techniques [1]–[4], and batching of GPU page faults to amortize overheads [5]–[7]. NVIDIA also introduced access counter-based migration to avoid page faults in the critical path [8]. However, they all fail to leverage higher-level memory access patterns of CUDA applications and instead remain myopic to limited runtime information. They also remain susceptible to thrashing under memory oversubscription in the absence of semantic insights into memory access patterns across phases of computing.

[1]For brevity, we henceforth refer to GPU's onboard memory as HBM without loss of generality. It can be GDDR or HBM2/3 technology.

Different from the *runtime-only* optimizations, we propose Static analysis guided UVM (SUV) to harness higher-level access patterns and semantics of CUDA programs. SUV's objective is to maximize the utilization of capacity-limited HBM while minimizing thrashing and page fault overheads.

Towards these goals, SUV employs a *hybrid* framework that couples high-level memory access semantics, statically inferred by a compiler, with *selective* runtime optimizations. Specifically, SUV embodies four key innovations.

First, it places and pins data structures with high *access density* onto the capacity-limited HBM. Data structures with relatively low access density are placed on the DRAM. The access density of a given data structure is the number of GPU memory accesses to it, divided by its size. This helps maximize accesses to the HBM while limiting those to the DRAM over the slow PCIe and avoiding page faults. SUV's compiler pass analyzes access patterns to different data structures to estimate the number of GPU memory accesses a data structure may witness. SUV combines it with runtime information of sizes of data structures to calculate access densities.

Second, we make a key observation that kernels often access *different* parts of large data structures during *different* phases of a kernel's execution. This is common for kernels computing on large matrices. A kernel could be launched with hundreds of thousands of threads, but a GPU may be able to execute only a subset of those threads simultaneously. Thus, only a fraction of such data structures are accessed depending upon the set of threads executing at a given computing phase (time).

To avoid wasting precious HBM capacity, SUV selectively retains only *portions* of such data structures in the HBM through a hybrid of compiler-inferred memory access patterns and runtime page migration. Specifically, it uses static analysis to infer the relation between threads and the parts of a data structure(s) it accesses. It combines this with the runtime information of the number of threads the given GPU can execute simultaneously to find the fraction of a data structure(s) that would be accessed concurrently (a.k.a., *working set*). SUV then reserves HBM capacity to retain only the working set and allow portions of the data structure to migrate onto the reserved HBM capacity on demand. This avoids thrashing of other data structures, besides improving HBM utilization by retaining only the working set.

Third, we also notice that CUDA applications often iteratively launch kernels to process large datasets where the kernels compute on disjoint portions of the dataset in different

iterations (invocations). Our static analysis infers the relation between a loop induction variable and the portions of the dataset that kernels would access in a given iteration. SUV then injects CPU code to explicitly (software) prefetch the data that will be accessed in an upcoming iteration (s) onto HBM while evicting those accessed in a previous iteration (s). This limits page faults while retaining useful data on the HBM.

Finally, not all access patterns are amenable to static analysis, e.g., pointer-chase. SUV's static analysis identifies them. It then deploys a hybrid approach where *only* such data structures are *selectively* migrated onto a reserved portion of HBM leveraging hardware access counters.

SUV requires *no* programmer intervention, *no* new hardware, *nor* does it need profiling. Its implementation has four major components. A compiler pass built into the LLVM framework [9] analyzes the memory access patterns of CUDA kernels. A second compiler pass analyzes the application's CPU (host) code for identifying variables that hold key execution-specific information, e.g., the sizes of data structures and the number of threads. It then injects CPU code to compute various metrics, e.g., access densities, during the program's execution. It does so by combining the results of static analysis with execution-specific information. SUV then invokes appropriate routines implementing memory management policies. The third component is a user-space library implementing and enforcing memory management policies via an enhanced UVM driver. The final component is an extended UVM driver in Linux that helps enforce memory management decisions [10]. The driver supports new capabilities to enable SUV's hybrid strategy, e.g., enabling access-counter-based page migration *only* for chosen virtual address ranges.

SUV's implementation consists of $\sim$3000 LOC in the LLVM compiler framework, $\sim$200 LOC for implementing memory management policies and $\sim$150 LOC for enhancing the UVM's Linux driver. The source code for SUV is available at https://github.com/csl-iisc/SUV-MICRO24.

SUV reduces the execution time of a variety of applications by 45% to 76% on average, over UVM with tree-based prefetching, under various oversubscription levels [11], [12]. We quantitatively compared SUV against hardware access-counter-based page migration [13] and a closely related academic work [14] to find that SUV significantly outperforms them across the board.

Our key contributions are as follows:

- To the best of our knowledge, SUV is the first to harness static (compiler) analysis to enable low-overhead UVM for CUDA applications.
- We propose SUV, a novel hybrid framework that judiciously leverages both compiler-inferred high-level memory access patterns of CUDA applications and selective runtime page migration for chosen memory regions.
- We make multiple key observations.We observe that different parts of a data structure can be accessed at different phases of computing based on the sub-set of GPU threads executing at a given time. We also observe that different iterations can access different parts of a data structure in applications that iteratively launch kernels.
- We build SUV in software by enhancing the compiler, the runtime, and the UVM driver. It requires no programmer intervention or hardware modifications.
- SUV reduces the execution time of applications by 45-74%, on average, without profiling or programmer intervention.

II. Background

GPUs are designed to accelerate data-parallel processing. The compute functions that run on GPU, called kernels, are typically written in CUDA for NVIDIA GPUs [15] [2]. The CPU code launches a kernel with a grid many thousands of threads. The grid is partitioned into equal-sized threadblocks (TBs) comprising up to 1024 threads each. A TB typically contains multiple warps, each with 32 threads that execute in the Single Instruction Multiple Thread model. Each TB executes entirely on a single Symmetric Multiprocessor (SM) on the GPU.

TBs in a grid and threads in a TB are arranged in a three-dimensional space. Each TB in a grid is uniquely identifiable through indices along each axis (`bid.x`, `bid.y`, and `bid.z`). Similarly, each thread in a TB can be uniquely identified by indices along each axis (`tid.x`, `tid.y`, and `tid.z`). GPU kernels typically use these identifiers to index into data structures (e.g., a matrix). Kernels are often launched with hundreds of thousands of threads, all of which may not execute concurrently due to hardware resource constraints. The hardware scheduler concurrently executes subsets of TBs that it can accommodate at a time.

A. Unified Virtual Memory (UVM)

A GPU supports massive memory access bandwidth using onboard High Bandwidth Memory (HBM) [16] or GDDR technology [17]. We generally refer to GPU's onboard memory as HBM for brevity but without loss of generality. By default, the programmer is responsible for allocating and copying necessary data onto HBM. NVIDIA introduced UVM to simplify memory management [18]. Memory allocated using its `cudaMallocManaged` API is automatically migrated between the DRAM (CPU memory) and HBM on demand. Importantly, UVM enables oversubscribing HBM capacity. In the cloud, oversubscribing the limited HBM capacity is key to packing multiple clients onto one GPU.

Page fault-based migration: By default, NVIDIA enables UVM through GPU page faults. A page fault is raised when the GPU accesses a page that is not resident on HBM. The UVM driver on the host CPU is notified of the fault. The driver identifies and migrates the faulting page from the DRAM to HBM. After the driver notifies the GPU of the completion of the fault, the GPU re-executes the faulting instruction.

Tree-based prefetching: A GPU page fault and page migration incur tens of microseconds of delay [6]. To avoid overheads in the critical path, researchers have explored prefetching of pages onto HBM [1], [2], [7]. Most prominent is the Tree-based prefetching (TBP) [11], [12]. TBP creates

[2]We focus on NVIDIA GPUs, but ideas are generally applicable.

a complete binary tree to track allocations within each 2MB region and guide prefetching. A 2MB region is divided into 64KB chunks – the granularity of migration. The leaves represent 64KB regions, the root represents the 2MB region, and the internal nodes represent power-of-two regions between them. Each node in the tree tracks the fraction of HBM-resident memory represented by the subtree rooted at the node. If it is above a threshold (default, 50%), all the other pages in its subtree are prefetched onto the HBM to leverage spatial locality. The UVM driver employs TBP [10].

Access counter-based migration: NVIDIA introduced hardware access counters [11] on GPUs to find frequently accessed DRAM-resident pages. UVM allows DRAM-resident pages to be directly accessed over the PCIe without faulting (remote access). The counters can track the number of remote accesses to a few memory regions [13]. When the count for a region crosses a threshold (default, 256), the driver migrates the memory region to HBM. The default size of a region is 2MB but is configurable to 64KB, 16MB, and 1GB. UVM supports access counter-based migration as an alternative to default page fault-based migration and can be configured at startup.

III. Limitations of prior approaches

We discuss how prior works fall short to motivate SUV. We detail the limitations of runtime approaches, followed by that of the existing static analysis-based technique.

A. Limitations of runtime approaches

The runtime approaches to reduce UVM overheads can be broadly classified into two classes: ① techniques that rely on prefetching and ② hardware access counter-based techniques.

Prefetching techniques: While prefetching techniques, including TBP (§ II-A), can reduce page faults, they cannot alleviate thrashing of the HBM under memory oversubscription. If a kernel's working set is larger than HBM's capacity, it can cause back-and-forth page migration, i.e., thrashing. Further, these techniques are limited to considering access patterns within fixed-sized regions, e.g., 2MB, for prefetching. They cannot leverage access patterns across large memory regions to proactively prefetch more data. TBP is part of our baseline.

Access counter-based migration: Access counters can help identify and migrate frequently accessed pages without incurring faults. However, it cannot prevent thrashing under oversubscription. Thrashing can set in quickly if the migration threshold is configured low. If the threshold is set high, it can cause many slow accesses to the DRAM over the PCIe since the frequently accessed pages would migrate to HBM only after the threshold number of accesses. Access counters also have a large configuration space over threshold and granularity, with different kernels preferring different configurations. Changing the configuration requires reloading the UVM driver.

Importantly, both runtime techniques are inherently *reactive*. They fail to leverage higher-level memory access behavior to perform *proactive* memory management. For example, consider the following access pattern commonly found in linear algebra kernels. Suppose an application iteratively launches kernel(s) and accesses two key data structures. One of the data structures is accessed entirely in every iteration. However, only a *disjoint* portion of the second is accessed in each iteration. Under memory oversubscription, pages from the second data structure may evict useful pages from the first, instead of evicting other pages of the second data structure whose utility has expired in previous iterations. If the semantic knowledge of access patterns had been available, a better strategy would have been to pin the first data structure on HBM. Then, in every iteration, prefetch only the required portions of the second onto the pages whose utility has expired.

B. Limitations of prior static analysis

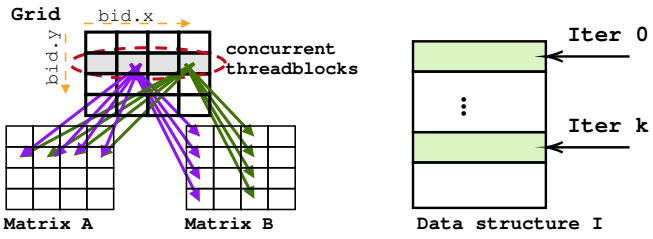
Static analysis can infer high-level memory access patterns that can be leveraged for proactive memory management. There exists one prior work in the supercomputing community that uses static analysis for UVM [14]. We refer to this work as SC. However, SC is tailored for simplistic GPU-accelerated OpenMP programs running on IBM Power 9 systems with cache-coherent NVLink between the DRAM and HBM [19]. Importantly, we noticed significant shortcomings in SC that limit its usefulness for typical CUDA applications.

First, SC considers the *total* access counts to data structures to determine their priorities in placement decisions. This can lead to poor utilization of the HBM. For example, a small and a large data structure (by size) may have similar access counts. Placing the larger one on the HBM would under-utilize its capacity, leading to more DRAM accesses. Further, SC also uses reuse distance across kernel launches as a metric. This can be effective for OpenMP programs where offloading work onto a GPU typically happens at the innermost loop, leading to thousands of launches of small kernels [14]. However, such metrics are insufficient for typical CUDA applications with larger kernels but fewer launches, particularly if each kernel accesses all or most of the key data structures of an application.

Importantly, SC makes memory management decisions for an *entire* data structure. It fails to observe that different parts of a data structure may be accessed during different phases of computing. Such behavior is prevalent in kernels launched with many threadblocks and in applications that iteratively invoke kernels to process different parts of a data set.

Applications processing large datasets often launch kernels with many threadblocks (TBs). All TBs may not execute concurrently due to hardware constraints. Typically, TBs access different parts of data structures indexed by their threadblock ID. Thus, the *parts* of data structures that would be accessed depend upon the TBs scheduled to execute at a given time. Unfortunately, SC is agnostic of this *temporal* access pattern.

We illustrate such behavior with a common access pattern where computation is performed on two matrices, (e.g., matrix multiplication) as shown in Figure 1a (simplified). A kernel is launched with a 2D grid of TBs, which accesses two matrices, A and B, of the same size. The threads in a TB access matrix A in row-major order and matrix B in column-major order, as is typical in many matrix operations. Here, a row of four TBs in the grid (shaded) execute concurrently. Consequently, this



(a) Temporal access. (b) Iterative access.

Fig. 1: Simplified examples of access patterns.

set of TBs will access the entire matrix B but only one of the rows of matrix A. A different row of matrix A will be accessed when a different set of TBs are scheduled for execution.

SC will observe that both matrices witness the same number of accesses and thus allocate them arbitrarily on HBM. A better strategy would be to bring parts of matrix A based on TBs currently executing while keeping the entire B on HBM.

It is also common for GPU-accelerated applications to iteratively launch a kernel(s) to process large data sets. The loop induction variable passed to the kernel typically determines the region of the data set that the kernel will access in a given iteration (Figure 1b). Such access patterns are prevalent in many scientific applications, e.g., Gram Schmidt [20]. Unfortunately, SC considers data structures as a whole and is ignorant of the relation between the iteration number and data accessed by kernels invoked in that iteration. Thus, it makes imprudent decisions when placing data for such applications. Finally, SC ignores data structures that witness access patterns that are unfriendly to static analysis, e.g., pointer-chase.

Summary: Runtime approaches fail to leverage higher-level memory access patterns and cannot avoid thrashing under oversubscription. Existing static analysis is agnostic to temporal and iterative access patterns common in CUDA applications and uses metrics that under-utilizes HBM.

IV. Goals and Design Principles of SUV

The goals of SUV are three-fold: ① Maximally utilize HBM capacity to limit accesses to DRAM. ② Limit overheads of page faults and migrations. ③ Prevent thrashing under memory oversubscription. Importantly, SUV aims to achieve these without programmer intervention or new hardware.

SUV's key idea is to harness high-level memory access patterns of CUDA applications for proactive memory management, which is further aided by reactive runtime techniques and page migration where suitable. The kernels are often structured to leverage GPU's SIMT architecture, allowing compile-time static analysis to yield insights into access patterns. However, the static analysis alone is not sufficient. Execution-specific information that varies across kernel invocations, e.g., sizes of data structure, available HBM capacity, and number of threadblocks, are necessary for memory management. Further, SUV *selectively* employs runtime page migrations to parts of address spaces and leverages the hardware access counters to remain effective in the presence of temporal access patterns and pointer-based data structures, respectively.

This hybrid memory management philosophy of SUV embodies four important innovations, which we discuss next.

SUV uses access density to infer the benefits of placing a data structure on the HBM. Access density is the number of accesses to a given structure divided by its size. SUV sorts data structures by their access densities and pins them (or their parts) on the HBM in descending order of densities. If the HBM is full, the remaining structures are pinned on DRAM to avoid thrashing. Thus, high-density data structures reside on the HBM, limiting slower DRAM accesses and avoiding unnecessary page migrations (goals ①, ②, ③).

Access density alone is inadequate for kernels that access different parts of a data structure based on which TBs execute concurrently at a given time. SUV first uses static analysis to find a kernel's memory accesses that are dependent on the threadblock ID. The analysis also finds out what fraction of a given data structure is accessed by *each* TB. SUV injects CPU code to find the number of TBs in the kernel and the number of TBs that are likely to be executed together on the given GPU. This, along with the static analysis, helps determine the fraction of such a data structure that would be accessed concurrently. We call it the *working set* of that data structure. SUV also marks such data structures as '`Temporal`'.

As before, data structures are considered in descending order of the access densities. However, if SUV encounters one marked '`Temporal`', it reserves the HBM capacity needed to accommodate *only* that data structure's working set. SUV then employs its hybrid strategy, letting portions of such data structures migrate onto HBM on-demand as different sets of TBs execute. This avoids over-provisioning of HBM when only parts of a data structure would be accessed at a given time and brings *only* the necessary portions of it (goal ①). Importantly, it avoids thrashing of other data structures, possibly with higher access density (goal ③).

Next, SUV's compile-time static analysis identifies applications that iteratively launch kernel(s) with different iterations (i.e., different invocations of the same kernels), accessing disjoint parts of large data structure(s). SUV infers the relation between the address offset accessed in such kernels and the induction variable of the loop in the CPU (host) code that iteratively launches the kernels. This identifies if only parts of data structure(s) are accessed in a given iteration. At loop boundaries, SUV then prefetches parts of such structures to be accessed in an upcoming iteration, evicting data from an earlier iteration. This avoids page faults and thrashing while minimizing slow accesses to DRAM (goals ①, ②, and ③).

Finally, it may *not* be possible to discern all access patterns statically, e.g., pointer-chase and input-dependent accesses. SUV leverages hardware access counter-based page migration *only* for data structures that witness such access patterns – creating a hybrid strategy. All data structures that lend themselves to static-analysis-guided memory management continue to benefit from the aforementioned three strategies. For those where static analysis fails to discern access patterns, SUV selectively deploys access counter-based page migration.

Summary: Ⓐ SUV pins data structures with higher access density on the HBM and those with the lowest densities on the DRAM to maximally utilize HBM while avoiding thrashing. Ⓑ For data structures that are accessed in parts based on TBs being executed, SUV employs on-demand page migration onto the HBM reserved to accommodate only its working set. Ⓒ In applications that iteratively launch kernels, it leverages the relation between iteration count and data to be accessed in an iteration to issue software prefetches.Ⓓ Finally, for data structures that are unfriendly to static analysis, SUV relies on access-counter-guided page migration.

V. Implementing SUV

SUV does not need user intervention nor manual code modification. It is built in the LLVM compiler framework [9], [21] to perform source code analysis and to inject (host) code as necessary. SUV extends NVIDIA's UVM Linux driver [10] with new APIs for enforcing memory management decisions.

SUV's implementation has four major components, as depicted in Figure 2. ① An LLVM compiler pass, named SUV-Kern, analyses memory operations (loads/stores) in CUDA kernels to infer memory access patterns. ② A second LLVM pass, called SUV-Host, analyzes the host (CPU) code to identify kernel launches, (virtual) memory regions passed to the kernels [3], and variables containing key execution-specific information (e.g., grid dimension) It then injects CPU code that would compute key metrics needed for memory management decisions (e.g., access density, working set) at runtime and invoke routines implementing memory management policies. ③ SUV-Mgmt, written as C/C++ functions, implements the memory management policies and invokes appropriate driver APIs to enforce them. ④ Finally, SUV extends the UVM driver to create necessary APIs for enforcing its decisions. We now detail the implementation of each of these components.

A. SUV-Kern: CUDA kernel analysis

SUV-Kern, implemented in the LLVM framework, analyzes the LLVM IR (intermediate representation) [22] of CUDA kernels to identify (static) loads and stores (henceforth called memops), the data structures they access, and their access patterns. It performs the required analysis to help infer virtual addresses a memop would access at runtime. Further, if a memop is within a loop, it helps infer the loop bounds.

We explain the workings of SUV-Kern with a simple LLVM IR snippet for the expression `int val = arr[bid.x * bdim.x + tid.x]` (listing 1). Here, it identifies the `load` at line 4 as a memop and `arridx` as the address it accesses, defined at line 3. The `gep` instruction [23] (line 3) computes the effective address, given a pointer (`arr`) and offset (`idx`). From line 3, SUV-Kern infers that the memop accesses the data structure pointed by `arr` and `idx` is the index (offset) into it. It creates an expression tree (Figure 3a) by traversing the statements in lines 1-2 in reverse order. As the variables

[3]We use the terms (virtual) memory region and data structure interchangeably for ease of explanation.

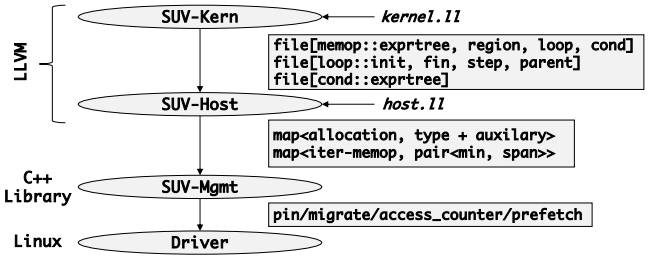


Fig. 2: High level workflow of SUV.

```
1  %mul1 = mul i32, %bidx, %bdimx
2  %idx = add i32, %mul1, %tidx
3  %arridx = gep i32, ptr %arr, i64 %idx
4  %val = load i32, ptr %arridx
```

Listing 1: LLVM IR for computing index in a simple load.

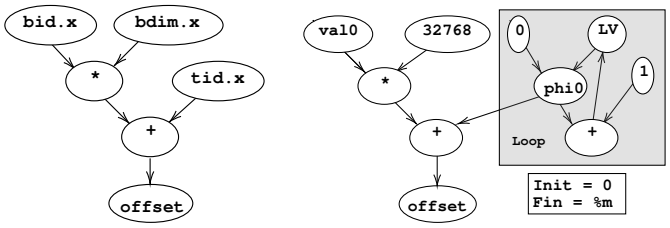
in the expression tree (e.g., `tid.x`) assume real values at runtime, evaluating the tree yields the offset (address) into the data structure (memory region) pointed to by `arr`.

SUV-Kern also identifies if a memop is inside a loop (or nested loop) and extends its expression tree with relevant loop information. Figure 3b shows such a (simplified) tree, along with the expression tree for its loop induction variable (shaded box). The loop bounds, i.e., variables containing the initial and final values of the loop variable, are also associated with the tree (rectangle at bottom). Here, the offset is computed from `val0`, constant 32768, and the loop variable `LV`. The value of `LV` increases by 1 every iteration while `0` (Init) and `%m` (Fin) are the loop bounds. Note that the absolute values of loop bounds may not be known statically. There, SUV-Kern creates the expression trees for loop bounds (not shown) that can be traversed at runtime to compute them.

Besides loop details, SUV-Kern keeps branch information. If a memop resides in the taken or the fall-through path of a conditional branch, SUV-Kern tags that memop with the expression tree of the branch condition. It also notes whether the memop is within the taken or the fall-through path. This information is later used during kernel launch to determine the fraction of threads that would execute the given memop.

Finally, SUV-Kern identifies pointer chase patterns by detecting loops where the value loaded from memory in one iteration is used in computing addresses in the next iteration. Such data structures are marked '`Unknown`'. By default, all data structures are classified as '`Density`' since SUV first tries to use access density to place data structures. However, they can later be re-classified based on access patterns.

Outcome: SUV-Kern outputs a file (Figure 2) that contains the following information for *every* kernel in the application. For every memop of a given kernel, it keeps an ID, the expression tree for the address it accesses, the data structures it accesses, and if the memop is within a loop in the kernel, then it keeps the ID of the encompassing loop. For memops within a branch, it keeps the expression tree for the branch condition and whether it resides in the taken or the fall-through path. Separately, for each loop in the kernel, SUV-Kern keeps the



(a) A simple memory access. (b) Memory access inside loop.

Fig. 3: Example expression trees generated by SUV-Kern.

loop ID and expression trees for its initial, final, and step values. It also tracks nested loops. We will later see how other components of SUV utilize this information.

B. SUV-Host: CPU code analysis and metric computation

SUV-Host, also implemented in LLVM, performs three key tasks. It analyzes the LLVM IR of the host (CPU) code to identify memory allocations, kernel launches, and the variables holding key execution-specific information, e.g., grid dimensions. It also identifies if a kernel (s) is launched iteratively and tracks the associated loop induction variables. Next, SUV-Host injects CPU IR (code) to compute various metrics, e.g., access density, at runtime using results from static analysis. Finally, SUV-Host inserts calls to pre-defined memory management routines (SUV-Mgmt). We detail each of these sub-components.

1) Identifying callsites and arguments: We use a simplified example (listing 2) to describe the working of SUV-Host. First, SUV-Host traverses the IR to identify memory allocations (line 1, mallocManaged). For each allocation, it inserts IR (not shown) to record the size of the allocation (here, `%sz`) and the pointer (virtual address) it would return at runtime (here, `%arr`). This information is later used in calculating metrics, e.g., access densities. Further, SUV-Host finds all aliases to the pointer (`%arr`) as different variables can refer to the same memory allocation.

SUV-Host then identifies kernel launches and their arguments. In listing 2, it identifies the kernel launch in line 6 (cudaLaunchKernel). It records the variables containing grid (`%gdimxy`, `%gdimz`) and block dimensions (`%bdimxy`, `%bdimz`) and associates them with the callsite (`%kcall`). It notes the kernel invoked, identified by the variable `kern` here. The CUDA runtime passes arguments to the kernel by packing them into a structure. SUV-Host unpacks the structure (%args) and traverses the IR to identify the variables assigned to the arguments. For example, it will identify variable `%arr` as the pointer to the data structure used by the kernel. These are recorded in an internal map in LLVM for later consumption.

SUV-Host identifies a kernel as being iteratively launched if the callsite is inside a loop (e.g., lines `2-8` in listing 2). For iteratively launched kernels, it identifies the (host) loop induction variables and loop bounds. Here, line `3` assigns the loop induction variable (`%iter`) value 0 if the statement is reached from the label `body` or the value of the variable `%inc1` if it is reached from the label `end`. SUV-Host records

```
1  %c = call @mallocManaged(ptr %arr,i64 %sz)
2  body:
3    %iter = phi i32 [0, body], [%inc1, end]
4    %args.arr = %arr
5    %args.k = %iter
6    %kcall = call i32 cudaLaunchKernel(ptr %
         kern, i64 %gdimxy,  i32 %gdimz,  i64
         %bdimxy,  i32 %bdimz,  ptr %args)
7    %inc1 = add i32 %iter, 1
8  end:
9  %bool1 = icmp eq i32 %inc1, %loopiters
```

Listing 2: Example host side LLVM IR.

that the loop variable `%iter` is passed to the kernel as the argument `%k`. Finally, it associates the iterative kernel launch with the loop bounds. For example, it will identify `%loopiters` as the variable containing the final value of the loop variable. SUV-Mgmt later uses these for prefetching at the kernel launch boundary.

2) Code injection for metric computation: SUV-Host injects CPU code (IR) that computes three metrics during an execution. It computes the access density, working set sizes of data structures, and the 'span' of memory accesses for iteratively launched kernels. We define the span as the range of address offsets accessed by memop in an iteration.

Calculating access density: Before every kernel callsite, SUV-Host injects IR to compute the access count of each memop of that kernel. It obtains the mapping of memops to the memory regions they access from the output of SUV-Kern. SUV-Host then aggregates the access counts of all memops accessing a given memory region and divides them by the size of the memory region to get the corresponding access density. The access densities are recorded in a runtime map.

Listing 3 shows a simplified example of *injected* IR to compute access density for a data structure accessed by two memops in the kernel. In this example, the first memop is the one whose expression tree is shown in Figure 3b. Lines 1-3 show the injected IR that calculates its access count. The number of threads is obtained by multiplying variables containing block and grid dimensions (line 1). Here, all threads in the kernel execute the given memop. Further, since this memop is executed in a loop, the access count is multiplied by the loop bound of its enclosing loop (here, `%m`) as identified by SUV-Kern. The second memop (not shown) is outside the loop. Accordingly, the injected code calculates its access count in lines 1 and 4. The IR then adds the access counts of these memops and divides them by the size of the memory region to arrive at the access density (lines 5-6).

In cases where the loop bounds cannot be determined even at the time of kernel launch, e.g., when the bounds are data-dependent, the access density or patterns of the memory operations within these loops are not calculated. SUV deploys its hybrid strategy by classifying such data structures as '`Unknown`' and relying on hardware access counters to migrate pages *only* for that virtual address region.

For memops tagged with a branch condition, SUV-Host injects IR to compute the branch condition. The injected IR

```
1  %num_thread = mul i32 %gdim.x, %bdim.x
2  %loopbound = %m
3  %acc_cnt1 = mul i32 %num_thread, %loopbound
4  %acc_cnt2 = %num_thread
5  %acc_cnt = add i32, %acc_cnt1, %acc_cnt2
6  %acc_den = div i32 %acc_cnt, %size
```

Listing 3: Example injected IR for computing access density.

computes the fraction of threads that would execute the memop based on whether it resides on the taken or the fall-through path and the evaluation of the branch condition. Then, it injects IR to scale the access density of the memop, initially estimated without considering branches, by the proportion of threads that follow the path where the memop is located. Note that if the branch condition is data-dependent, i.e., depends on data loaded during kernel execution, SUV cannot evaluate the condition, potentially overestimating the access density.

Estimating working set size: SUV-Host injects code before every kernel callsite to estimate the working set size (WSS) of each data structure that the kernel accesses. Based on the estimated WSS and the size of the data structure, SUV-Host may classify a data structure as '`Temporal`'.

The WSS captures the fraction of a data structure likely to be accessed by concurrently executing threadblocks (TBs). SUV-Host calculates WSS using two pieces of information: ① the number of TBs that a given GPU is likely to execute concurrently, and ② the fraction of data accessed by one TB.

First, SUV-Host injects IR to find the number of SMs and the number of threads per SM on the GPU. The injected code then divides the number of threads per SM by the number of threads per TB (available from the kernel launch arguments) to find the number of TBs per SM. It is multiplied by the number of SMs to estimate the number of TBs that execute concurrently. If this is more than half the total number of TBs, data structures cannot exhibit significant temporal behavior in the given kernel invocation, and we do no further analysis.

For kernels that may exhibit temporal behavior, SUV-Host identifies the fraction of each data structure accessed by *a single* TB. It parses the output file of SUV-Kern to find the memops that access the given data structure. For each such memop, SUV-Host injects (host) IR to find the minimum and maximum offset within the region that it accesses. The injected IR traverses the memop's expression tree (available in SUV-Kern's output) from leaves to the root, calculating the min and max at each node. The minimum and maximum values a node assumes depend on the variable in the node. For example, the block dimensions dictate its min and max values of threadID. If the given memop is within a loop inside the kernel, its loop bounds determine the min and max. Alternatively, a node can represent a constant, in which case both min and max are the constant itself. The min and max at an intermediate node are calculated using that of its children. The difference between the min and max values at the root of the expression tree provides the address range (offset) that a given memop can access. Further, since we calculate the min and max for a single TB, block IDs at any node are set to a constant.

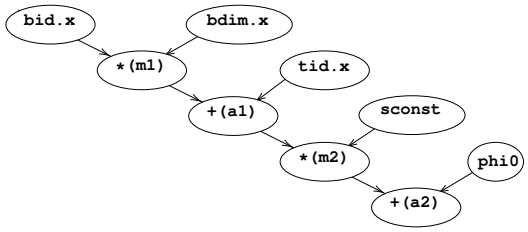


Fig. 4: Example expression tree of a memop inside a loop.

```
1  %max.bid.x = i32 0
2  %min.bid.x = i32 0
3  %max.tid.x = sub i32 %bdim.x, 1
4  %min.tid.x = i32 0
5  %max.phi0 = %m
6  %min.phi0 = i32 0
7  %max.m1 = mul i32 max.bid.x, bdim.x
8  %min.m1 = mul i32 min.bid.x, bdim.x
9  ...
10 %max.a2 = add i32 max.a2, max.phi0
11 %min.a2 = add i32 min.a2, min.phi0
12 %wss = sub %max.a2, %min.a2
```

Listing 4: Calculating working set size of a memop.

We show WSS calculation using an example expression tree of a memop inside a loop in the kernel (Figure 4). In Figure 4, `phi0` is a *kernel*'s loop induction variable, and `sconst` is a constant. We name the operator nodes (in parenthesis) for ease of explanation. Listing 4 shows the injected IR for computing the min and max offsets for this example. First, it identifies the min and max values that the leaves of the tree can assume. The thread ID `tid.x` can take a max value of (`bdim.x` -1). This value is available in the kernel's threadblock dimension at runtime. The max value of the kernel loop variable `phi0` is the loop bound '`%m`' (line 5). Similarly, the injected IR computes the min values (lines 2, 4, 6). The tree is traversed from leaves to root to calculate the min and max at each node using those of its children. For example, the max value at the root (`max.a3`, line 10) is the sum of the max of its children (`a2` and `phi0`) since the root node has a '+' operator. Similarly, the line 11 calculates the min.

SUV-Host injects similar code to find the min and max offset for *every* memop that accesses the given region. The min and max offset of *any* access on that region is then calculated as the min and max offset over all the memops accessing it. The difference between this min and max gives the region's working set size (WSS) considering *one* TB. The injected code then multiplies this WSS by the number of concurrent TBs to arrive at the final WSS of the region.

If the WSS is smaller than a fraction (here, 0.5) of that data structure's size, then the region is marked '`Temporal`'. Otherwise, most of the data structure is accessed by the TBs concurrently. SUV-Host calculates and records WSS for all data structures (data structures) accessed by a kernel and tags them as '`Temporal`' as applicable.

Note that static analysis cannot estimate the WSS for indirect (data-dependent) memory access. SUV detects indirect access by checking if the expression tree contains a load. Since

SUV cannot calculate the WSS of data structures with indirect accesses, those structures are classified as '`Density`' and, thus, unable to leverage their temporal behavior (if any).

Calculating 'span' for iteration-dependent memops: Next, SUV-Host analyzes iteratively launched kernels. SUV keeps only portions of data structures that will be accessed in an upcoming iteration(s), i.e., invocation of a kernel, in the HBM while evicting data that has lost its utility. SUV-Host injects host IR to find parts of data structures that memops would access in a given iteration of a kernel launch. SUV finds the memops in an iteratively launched kernel where the accessed address depends on the iteration. SUV-Host inspects the expression trees of memops generated by the SUV-Kern to find if any of its nodes contain the host loop variable (i.e., iteration number). Such *iteration-dependent* memops access different offsets within a data structure across different iterations.

SUV-Host injects (host) IR to find the 'span' of offsets within a data structure that such memops access. The span is the range of addresses within a data structure that a memop accesses in a *given* iteration. The injected IR traverses the memop's expression tree from leaves to the root, computing the min and max values each node can take in the same way it estimates the WSS, but with one difference. Recall that SUV-Host estimates WSS for one threadblock and, thus, assigns a constant value for blockID. However, while estimating the span, the max and min values of blockIDs are determined by the kernel's grid dimensions. For example, `bid.x` (blockID along the x-axis) takes max and min values of `gdim.x -1` (kernel launch parameter) and 0, respectively. Further, the host loop induction variable gets its value from the kernel launch parameters during execution. The difference between the min and max values of the root node of the expression tree represents the span for the given iteration.

Outcome: By the end of SUV-Host's pass, each data structure has at least one classification – '`Density`', '`Temporal`', or '`Unknown`'. They can also be marked '`Iterative`' when accessed by an iteratively launched kernel. SUV-Host also associates each region with its access density and WSS, where applicable. Further, for each iteration-dependent memop in iteratively launched kernels, it keeps the min offset of access within the region and the span (offset range). These are calculated for each kernel by the IR injected by SUV-Host during execution and are stored in runtime maps for the next component (SUV-Mgmt) to perform memory management.

C. SUV-Mgmt: Memory management policies

SUV-Mgmt makes memory management decisions during execution using the analysis performed by SUV-Kern and SUV-Host. It then enforces those decisions by calling appropriate driver APIs. SUV-Mgmt is implemented in C++. Updating policies does not need changes to LLVM, ensuring a clean separation between the mechanism and the policy. SUV-Mgmt is implemented in two key routines. The first manages data structures (memory regions) at each kernel invocation. The second is responsible for prefetching parts of '`Iterative`' data structures at iteration boundaries.

Algorithm 1 Per- kernel invocation decision routine

```
1:  avail := getAvailableHBM()
2:  for ds ∈ DataStructures do
3:      if DecisionMap[ds] == Unknown then
4:          Reserve (size[ds]/footprint) * avail
5:          Enable Access Counters
6:  SortDataStructuresByAccessDensity()
7:  for ds ∈ DataStructures do
8:      if avail == 0 then
9:          DRAMPIN(ds); continue
10:     if DecisionMap[ds] == Temporal then
11:         if AccessDensity[ds] > threshold then
12:             ONDEMAND(ds, wss[ds])
13:             allocatedSize = wss[ds]
14:             avail = avail − allocatedSize
15:             continue
16:         else DRAMPIN(ds)
17:     pinSize = min(size[ds], avail)
18:     HBMPIN(ds, pinSize)
19:     DRAMPIN(ds + pinSize, size − pinSize)
20:     allocatedSize = pinSize
21:     avail = avail − allocatedSize
```

Routine for per-kernel invocation policy: Algorithm 1 shows the high-level pseudo-code for the routine invoked by injected IR before every kernel launch. It iterates over all data structures accessed by the kernel and makes memory management decisions based on each data structure's classifications.

First, the algorithm determines the available HBM capacity (line 1). Then, it identifies the data structures of type '`Unknown`', if any. It is not possible to statically determine their access density or working set. Thus, it sets aside HBM capacity in proportion to the ratio of their sizes to the kernel's total memory footprint (line 4) and enables access counter-based migration via enhanced driver API (line 5).

It then iterates over all other data structures in the *decreasing order* of their access densities (lines 6,7). If HBM is full, the data structure is pinned on the DRAM (lines 8-9). For data structures of type '`Temporal`' with access density above a threshold (here, 5[4]), it reserves capacity on the HBM equal to its WSS. It allows it to migrate on demand at runtime following the hybrid strategy (lines 10-15). If the density is below the threshold, the data structure is pinned on the DRAM (line 16). For other types of data structures (not '`Temporal`'), it pins it on the HBM if space is available (lines 17-18). If the HBM capacity is exhausted, the rest is pinned on the DRAM (line 19).

Routine for iteration-dependent prefetching: Data structures marked '`Iterative`' are considered for software-directed prefetching at host loop iteration boundaries. Algorithm 2 shows the simplified pseudo-code for the corresponding routine. For each data structure of type '`Iterative`', it iterates over the memops that access different parts of the data structure across different iterations (lines 1-2). For each such memop, it looks up the map generated by SUV-Host

[4]The threshold (parameterized) ensures that the migration cost is amortized. We empirically determined the default value using a micro-benchmark.

(Section V-B) to find the corresponding min offset and span. It adds the base virtual address of the data structure to the min to find the starting address to prefetch for the given iteration. The span is the amount of memory to be prefetched (lines 3-4). It invokes driver APIs to prefetch and pin the corresponding portions of the data structure on the HBM (line 5). It also finds data that were brought in a previous iteration but lost their utility and evicts such data to the DRAM (not shown). If the span is smaller than 2MB, we optimize by prefetching for several iterations together to amortize overheads (not shown).

D. Enhanced UVM driver for enforcing memory management

SUV-Mgmt enforces its decisions through an *enhanced* version of NVIDIA's UVM driver in Linux [10]. The driver must support five key actions. ① It must allow placing and pinning of data on HBM. However, the driver currently does not support pinning on HBM. SUV introduces a new API, `suvPinOnGPU`, which pins a given virtual address range on the HBM. It places the corresponding pages on the HBM and removes them from the driver's LRU list to avoid their eviction. ② The driver allows pinning of data on DRAM using a new API, `suvPinOnHost`. To avoid GPU page faults on un-initialized DRAM-resident data, it first does `memset` on them. Then, it calls the existing `SetAccessedBy` API to map DRAM-resident pages onto GPU page tables. ③ To allow on-demand migration, SUV unsets all memory management policies on a memory region, allowing it to migrate via page faults. ④ For prefetching, SUV uses an existing API, `CudaPrefetchAsync` to prefetch address regions to HBM at iteration boundaries. ⑤ SUV creates two new APIs to support its hybrid design, where it may pin some memory regions while others can migrate using access counters. A new API `suvEnableAccessCounter` enables/disables access counter-based migration. The second API, `suvSetNoMigrateRegion`, requests the driver to ignore access counter notifications to specific virtual address regions, disallowing them from migrating.

E. Putting it all together

In summary, SUV-Kern analyses CUDA kernel code to generate expression trees for all (static) kernel memory operations, identifies the data structures they access and then saves these in a file (Figure 2). SUV-Host analyses CPU code to identify data structures and kernel launches, and then appropriately injects CPU code (IR) to compute metrics using execution-specific information and generated expression trees. The injected code saves the relevant information in runtime maps and invokes appropriate routines of SUV-Mgmt that make memory management decisions using relevant metrics. Finally, it invokes appropriate APIs of the enhanced UVM driver to enforce its decisions.

Algorithm 2 Per-iteration prefetch function

```
1: for ds ∈ HasIterDataStructures do
2:     for memop ∈ IterMemops[ds] do
3:         start = getStart(memop, iter)
4:         span = getSpan(memop)
5:         PrefetchAndPinOnGPU(start, span)
```

TABLE I: List of benchmarks with size and types of memory allocations. Types represented by I for '`Iterative`', D for '`Density`', U for '`Unknown`' and T for '`Temporal`'.

Abbr.	Benchmark	Types	Size (MB)
2DC	2-D convolution of matrix [20]	D	8192
AN	Alexnet inference [25]	D, T	3504
BFS	Breadth first search [24]	D, U	2610
BCG	Kernel of BiCGStab Linear Solver [20]	D	4096
BTR	B-tree lookup query [27]	U, D	5120
DTG	Doitgen linear algebra kernel [20]	I, D	8192
FDT	Finite difference in time domain [20]	D	6912
FW	Floyd Warshall for shortest paths [26]	I, D	4096
GMM	Simple matrix multiplication [20]	D, T	6912
GRM	Gramschmidt linear algebra kernel [20]	I, D	3072
HEL	Hellinger algorithm for similarity [26]	D, T	6912
MM	Tiled matrix multiplication [26]	D, T	5760
MVT	Matrix vector multiplication [20]	D	4096
SN	Squeezenet inference [25]	D, T	4976
XSB	XSBench neurtron transport algorithm [26]	D, U	3884

VI. Evaluation

We evaluated SUV on a system with an NVIDIA 3090 GPU (24GB version), connected to an AMD Ryzen 7950X CPU via an x16 PCIe 4.0 interconnect. We use LLVM 15.0 compiler [9] and NVIDIA's open-source Linux UVM driver (version 525) [10]. We choose a diverse set of applications from Polybench [20], Rodinia [24], Tango [25], and HecBench [26] suites for evaluation. These applications cover diverse access patterns, and have large memory footprints. They also have a mix of single, multiple, and iteratively launched kernels. Table I lists them with their memory footprints, and the different types of data structures they possess. For e.g., SUV identifies that `MM` has both '`Temporal`' and '`Density`' data structures.

In our evaluation, an oversubscription factor of *x* percent implies the application's memory footprint is *x* percent larger than the available HBM. To create different levels of oversubscription across different applications, we reserve portions of the HBM using `cudaMalloc`, as done in previous studies [2]. The reserved memory is unavailable to the application under evaluation and thus creates oversubscription.

A. Performance of SUV

We first compare the performance of SUV with the default page-fault-based UVM (baseline). We report end-to-end runtimes, excluding the initialization and post-processing on the CPU (e.g., file open/close). The measured time for SUV includes all its overheads due to the injected code and calls to the driver. Figure 5 reports the runtime of the applications under 15, 30, and 50 percent memory oversubscription. Each application has two bars under every oversubscription level representing runtimes under SUV and UVM, normalized to the runtime under *zero* (no) oversubscription. Lower is better.

First, note that SUV reduces runtime over UVM by large margins (e.g., 60%, on average, under 30% oversubscription).

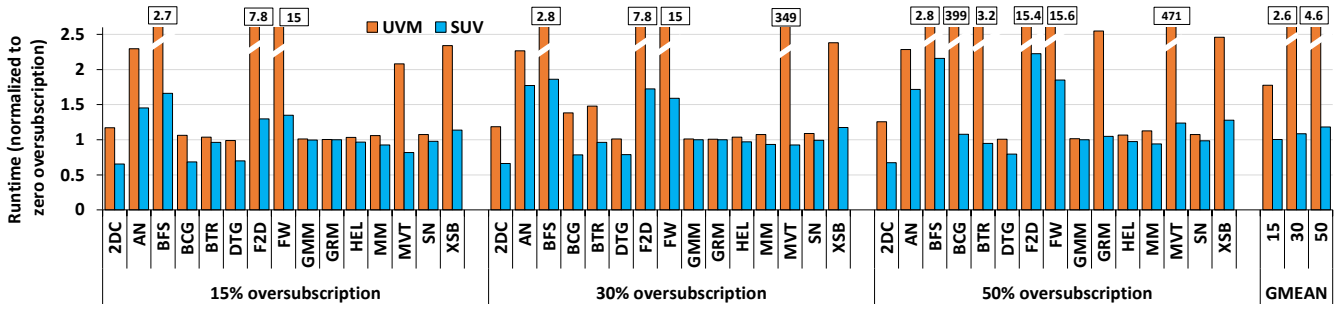


Fig. 5: Normalized runtime (lower is better) with various oversubscription levels.

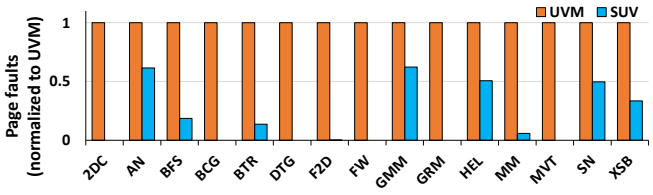


Fig. 6: Reduction in page faults with SUV over UVM.

Under UVM, it would be challenging to run several applications, e.g., FW, MVT, even under limited memory oversubscription due to unreasonably high overheads. SUV allows running them with limited overheads even under relatively higher oversubscription. SUV even outperforms zero oversubscription for some applications, e.g., 2DC. We will now explain the reasons behind SUV's performance.

The applications FW, MVT, and FDT (under all oversubscription levels), and GRM, BCG, BTR, BFS, and XSB (under 50% oversubscriptions) incurred manifold slowdowns. This is because the active working sets of these applications fail to fit in HBM under these oversubscription levels and suffer from thrashing under UVM. SUV avoids thrashing and also limits slow accesses to DRAM. All applications significantly benefit from SUV's approach of pinning data structures with high access density on the HBM and those with the least on DRAM to avoid unnecessary page migrations while serving majority of the accesses from the fast HBM. Further, GRM and FW iteratively launch kernels to process different parts of a large data structure. They benefit significantly from SUV's software prefetching which only brings portions of such data structures needed in an upcoming iteration to avoid memory pressure. BTR, XSB, and BFS are testimonies to SUV's hybrid strategy. The pointer-based tree data structure of BTR, the memory region where XSB performs binary search, and the graph data structures in BFS (edge list) are migrated to the HBM guided by access counters. Meanwhile, low access density arrays (statically inferred) are pinned on the DRAM to avoid contention for the limited HBM capacity.

SUV improves performance even when applications do not thrash under UVM, e.g., DTG, MM, GMM, HEL, 2DC. The application DTG speeds up by 21% under 50% over subscription thanks to SUV's judicious software prefetching portions of its iteratively accessed data structures. The key data structures of MM, GMM, and HEL are 'Temporal' type. They benefit from SUV's hybrid strategy whereby it employs on-demand page migrations for *portions* of those data structures to reserved HBM capacity, avoiding eviction of useful data. The application, 2DC, speeds up by 43% thanks to the pinning of data structures as per their access densities. This limits page migrations and costly DRAM accesses. The performance for MM, 2DC, and DTG under SUV can even be better than with zero memory oversubscription under UVM. This is possible due to SUV's proactive pinning of key data structures on the HBM guided by the static analysis before the kernel starts executing. This avoids page faults that would have otherwise happened under UVM.

Applications AN and SN benefit from SUV's access density guided data placement and hybrid strategy on-demand page migration for 'Temporal' data structures. They launch many kernels, but only a few of those need a significant amount of memory and, thus, suffer from the overheads of memory oversubscription. They witness overall speedups of up to 30%, while individual kernels speed up by up to 2×.

In summary, SUV significantly outperforms UVM. SUV reduces the average (geomean) runtime over UVM between 44% to 74% across various oversubscription levels.

Figure 6 reports the normalized number of page faults under SUV and UVM with 50% oversubscription to substantiate SUV's ability to avoid thrashing. We observe that the number of faults reduces between 37% to 100% with SUV. Notice that applications such as MM, GMM, SN, and AN incur many faults even under SUV. This is because SUV allows page fault-based on-demand migration of 'Temporal' data structures in these applications. They speed up under SUV by not evicting useful pages of other data structures, unlike UVM.

B. Comparison with alternatives

We quantitatively compared SUV against ① SC [14] (§III-B), and ② NVIDIA's access counter-based page migration (AC) [8], [13]. While the SC originally worked only for OpenMP programs, we adapted it into the LLVM framework to work for CUDA kernels. We verified its implementation.

Figure 7 presents normalized runtimes of the applications under the alternatives at 50 percent oversubscription. We normalize runtimes over SUV.

Comparison with SC: SUV significantly outperforms SC. Applications GMM, MM, and AN access different parts of their key data structures at different phases of computing (i.e.,

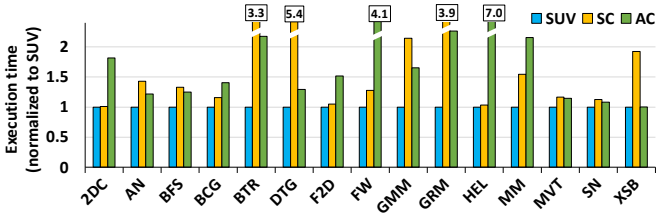


Fig. 7: Normalized runtime of alternatives (lower is better).

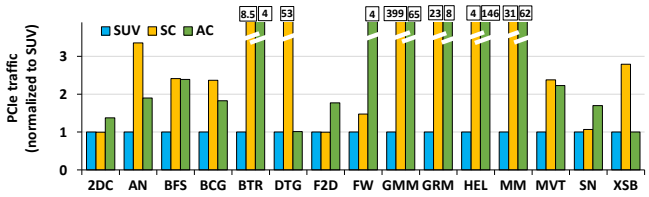


Fig. 8: Normalized PCIe traffic at 50% oversubscription.

'Temporal'). SC, being agnostic of such temporal behavior, under-utilizes HBM capacity, causing costly DRAM accesses over the PCIe and page faults. In contrast, SUV brings only portions of data structures onto a reserved HBM capacity, outperforming SC by 52%, on average. Figure 8 shows PCIe traffic normalized to that of SUV. As expected, these applications incur significantly more PCIe traffic under SC.

Applications GRM, FW, and DTG run 69% faster under SUV. These applications iteratively launch kernels and benefit from SUV's proactive software prefetching at iteration boundaries, bringing only necessary data for upcoming iterations onto the HBM. However, SC is agnostic of such iterative behavior and incurs several times more PCIe traffic (Figure 8). For MVT and BCG, SC places data structures with lower access density on HBM since it uses access count, unlike SUV. This leads to under-utilization of the HBM capacity. BFS has several data structures with unknown access density. Unlike SC, SUV dynamically leverages access counters for those data structures to migrate them onto HBM. On average, applications speed up by 40% with SUV over SC and incur 81% less PCIe traffic.

Comparison with AC: Applications GRM, FW, FDT, and BTR perform poorly with access counter-based migration (AC) compared to SUV (due to thrashing). GRM and FW iteratively launch kernel(s) and cause thrashing of HBM under AC. SUV's strategy of keeping only the data needed in upcoming iteration (s) in HBM avoids thrashing. FDT's large working set thrashes with AC, but SUV's pinning alleviates the thrashing. BTR has one pointer-chase data structure (B-tree) and two arrays (query and results) that are lightly accessed. SUV pins these arrays on DRAM, allowing more HBM capacity for the B-tree. Similarly, BFS pins lightly accessed regions on the DRAM, allowing more HBM capacity for the remaining structures. This shows the importance of SUV's hybrid strategy. XSB performs almost identically to AC since most memory regions rely on access-counter-based migration.

The applications MVT, BCG, DTG, GMM, MM, HEL, 2DC, SN, and AN do not suffer from thrashing but lose the opportunity to migrate heavily accessed pages early since pages migrate only after witnessing at least a 'threshold' (here, default 256).In contrast, SUV uses high-level semantics to place and pin pages proactively. Across fifteen applications, SUV reduces runtime by 43%, on average. Further, from Figure 8, we observe that applications incur 77% less traffic over PCIe under SUV, on average. We found similar trends under 15 and 30% oversubscription. We omit those due to space constraints.

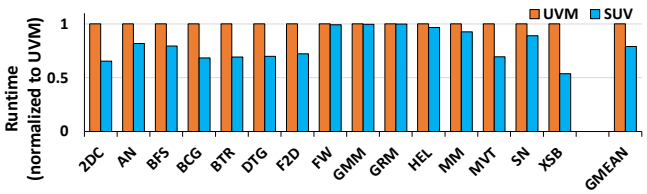


Fig. 9: SUV and UVM under zero oversubscription.

C. Analyzing overheads of SUV

SUV injects code for computing necessary metrics and invokes memory management routines. These overheads are included in the reported runtimes. Further analysis showed that these contribute less than 1% of runtimes. SUV adds 3% additional CPU instructions to the executable files, on average. Finally, the extra LLVM passes in SUV together add up to 5 seconds of additional compilation time. However, this is only a one-time cost for generating the binaries.

D. SUV under zero oversubscription

Even under zero oversubscription, i.e., when an application's memory footprint fits in the HBM capacity, SUV can improve upon UVM by prefetching and pinning data structures onto HBM, avoiding page faults during kernel execution. Figure 9 compares UVM and SUV at zero oversubscription. Applications like MVT, BTR, and DTG, which spend considerable time in page fault servicing under UVM, gain the most from SUV. SUV improves performance by 20 % on average.

VII. Extending SUV for closed-source libraries

Many GPU applications invoke closed-source kernels from libraries such as cuDNN. However, SUV needs to analyze kernels' source to infer memory access patterns, and thus, SUV is not immediately applicable to such applications. Fortunately, SUV's core design philosophies remain equally applicable, and its implementation needs only a minor extension to work with applications using closed-source kernels. Specifically, we divide the tasks of SUV between the library developer and the application writer who invokes kernels. All of SUV's components, *except* SUV-Host, remain unaltered and are deployed by the library developer. The library developer uses SUV-Kern to perform analysis of the kernels she writes and provides SUV-Mgmt and an enhanced driver that the application developer should use. Further, the library developer exposes a wrapper for each kernel that takes two additional arguments beyond the original: ① a boolean specifying whether the given kernel is invoked iteratively by the host code, and ② if yes, the second parameter passes the loop induction variable.

The functionality of SUV-Host is split into two – SUV-Host-app and SUV-Host-lib. The application writer runs SUV-Host-app only on the host code. SUV-Host-app analyzes the host code to find iterative kernel invocations, tracks induction variables, and replaces the original kernel invocation with a call to the wrapper with the two additional arguments mentioned above. At the library developer site, SUV-Host-lib uses SUV-Kern's output and injects host code as before to compute metrics such as access density and invoke SUV-Mgmt routines. SUV-Mgmt makes calls to the enhanced driver as usual. Finally, it invokes the original kernel with its arguments.

We implemented this proposed version of SUV to work with closed-source libraries. To demonstrate the feasibility, we created closed source libraries containing tiled matrix multiplication kernel `MM` and kernels from the `DTG` application. We invoked the kernels from an application program while deploying the enhanced SUV framework. As expected, SUV provided upto 15% improvement with `MM` over different dimensions of the matrices and up to 21% with `DTG`.

VIII. Related work

Tyler and Ge [28], [29], Chien et al. [30], and Gayathri et al. [31] analyzed the costs and benefits of demand paging and tree-based prefetching. They provide insights into UVM, but not mechanisms to automatically improve performance. Zheng et al. [6] describe various hardware and software techniques to improve UVM's performance. Mosaic [32] proposed application-transparent UVM with multiple page sizes. HPE [33] and CHPE [4] study eviction policies for UVM. Kim et al. [34] increase page fault batch size and perform pre-eviction to reduce UVM overheads. ETC [7] studied prefetching and eviction policies under UVM oversubscription. Ganguly et al. [35] [1] and Go et al. [2] used page fault history to detect access patterns and choose migration/eviction/prefetching policies. Long et al. [3] improve tree-based prefetching using deep learning. Ganguly et al. [36] propose enhanced access counter hardware and policies for dynamically determining migration thresholds. Agarwal et al. [37], [38] use profiling for page placement, maximizing bandwidth utilization in heterogeneous GPU systems. DynaMap [39] uses instrumentation to identify and migrate heavily accessed pages at runtime. Unlike these, SUV leverages high-level memory access patterns inferred by static analysis and requires no hardware changes.

DeepUM [40], Sentinal [41], SwapAdvisor [42], Capuchin [43] use profiling to optimize DNN training. G10 [44] uses tensor vitality analysis to migrate tensors between GPU, host and storage. They leverage idiosyncrasies of DNN training that are not generally applicable. There have been several static analysis-based optimizations for GPU programs but only one focuses on UVM. Several works [45]–[48] used linearity of address computation to reduce instruction count. Jablin et al. [49], DyManD [50], Tung et al. [51] and SuperNeurons [52] automate memory transfers, but without oversubscription. Several works have studied compiler-assisted prefetching for CPUs [53]–[55]. To the best of our knowledge, only one prior work [14] used static analysis for UVM. We detailed its shortcomings in Section III-B.

Locality Descriptor [56] allows programmers to provide locality hints for TB scheduling, cache management and prefetching. Unlike SUV, it needs hardware modifications and programmer hints. Khiary et al., [57] used static analysis to inform TB scheduling and page placement on multi-GPU systems. While some of our observations regarding the temporal locality have similarities, we exploit this observation to improve HBM utilization under oversubscription. We also make additional novel observations regarding the importance of access density and locality across iterations and build a hybrid of static and runtime techniques to exploit it.

Finally, several techniques [58]–[71] have been proposed to limit address translation overheads on GPUs. These techniques are complementary to SUV and may be used in conjunction.

IX. Conclusion

We propose to leverage high-level memory access patterns in GPU applications inferred through static analysis to optimize unified virtual memory. Our proposal, SUV, embodies several key innovations, including pinning data structures using access density to maximally utilize HBM, on-demand migrations of parts of data structure onto reserved HBM to avoid thrashing, and software prefetches for iteratively launched kernels. SUV requires no new hardware, nor does it need programmer intervention. Yet, SUV significantly outperforms the alternatives by wide margins across various levels of memory oversubscriptions.

X. Acknowledgments

We thank anonymous reviewers for the constructive feedback. Pratheek is partially supported by the Intel India Ph.D. Fellowship. He gratefully acknowledges the Microsoft travel award enabling participation in the MICRO 2024 conference. This work is partially supported by generous research grants from VMware Inc., AMD Inc., and by Google faculty awards.

Appendix

A. Abstract

We provide the source code of SUV as an artifact. SUV consists of 2 LLVM passes, runtime (provided as a header file), and enhanced UVM driver. The artifact contains the LLVM passes, header file, enhanced driver, and workloads.

After compiling LLVM (along with our passes) and installing the enhanced driver, please run the provided scripts to produce the results from the paper.

B. Artifact check-list (meta-information)

- **Compilation: LLVM**
- **Run-time environment: Linux, with Nvidia UVM driver**
- **Hardware: Nvidia 3090**
- **Metrics: Runtime, page faults, PCIe bandwidth**
- **Output: Runtime, page faults**
- **Experiments:**
- **How much disk space required (approximately)?: 150-200 GBs**

- **How much time is needed to prepare workflow (approximately)?: 4 hours**
- **How much time is needed to complete experiments (approximately)?: 48 hours**
- **Publicly available?: Yes**
- **Archived (provide DOI)?: https://doi.org/10.5281/zenodo.13743918**

C. Description

All script commands (in boxes) assume that it is in the base directory of the artifact.

1) How to access: Please download and unzip the archived artifact or clone the repository from here.

2) Hardware dependencies:

- Nvidia 3090 (24GB)
- PCIe v4.0 interconnect
- AMD Ryzen 7950X
- 128 GB RAM (for LLVM compilation)

3) Software dependencies:

- LLVM 15.0 (provided)
- Linux 6.2 (with sudo permissions)
- Nvidia UVM driver 525 (open source version)
- CUDA 11.8
- TMUX (or screen)

Our scripts require reloading the driver. Therefore, we recommend setting passwordless sudo to enable fire-and-forget for the scripts.

D. Setting paths

Please open a TMUX (or screen) session and run the following commands. Ensure that CUDA 11.8 is in the path.

Edit (we used nano editor) startup.sh to point to point to the bin directory of LLVM. For example, if the folder suv-ae is in the home folder of username `p`, then edit the startup.sh file to contain the following: Save the file by pressing Ctrl-X, followed by Y (on nano editor).

```
nano startup.sh
export PATH=/home/p/suv-ae/llvm/build/bin/:$PATH
export SUVHOME=/home/p/suv-ae/
```

Source the startup.sh file to ensure that LLVM and clang are in the path.

```
source startup.sh
```

E. Installation

First, install the open source flavor of UVM driver (ver. 525) from Nvidia. Please follow the instructions here for the installation, and here for additional details.

Compile and load the extended UVM driver from our folder.

```
cd open-gpu-kernel-modules
make modules -j
cd ..
bash driver_change.sh 0 64k 1
```

Compile LLVM (provided with our artifact). Note that this will also compile our LLVM passes. Please refer to LLVM requirements for required packages for building LLVM.

```
bash makerunllvm.sh
cd llvm/build
ninja
```

Note that LLVM compilation (ninja) may fail during linking on machines with less memory. In case this happens, just run ninja again (and no other command).

Generating data: The required data sets for all workloads except BFS are provided with the artifact. To generate the graph used with BFS, please run the following from the base directory.

```
cd eval/bfs/inputGen
make
bash gen_dataset.sh
cp graph16M.txt ..
```

F. Experiment workflow

Now simply run the following command to compile, run, and parse all the main results.

```
bash all.sh
```

To perform the compilation, running, and parsing steps individually, please follow the following.

We provide one compiled binary for each configuration (workload, oversubscription factor, and algorithm (UVM, SUV, SC)). Compile all binaries by running the following command. This can take upto 4 hours.

```
bash compile0.sh
bash compile.sh
```

Once all the binaries are compiled, run everything with run.sh. This process can take about 48 hours to complete.

```
bash run0.sh
bash run.sh
```

Run parse.sh to parse everything and obtain CSV files in the base folder.

```
bash parse1.sh
bash parse2.sh
bash parse3.sh
```

G. Evaluation and expected results

Figure 5, 6 and 7 can be reproduced from the artifact by running the above commands. The `all.sh` script above generates three CSV files, corresponding to the three main figures in the paper.

Please copy the contents of the CSV files onto the first column in corresponding page in the provided Excel sheet, `suv.xslx`, to get the normalized data.

H. Experiment customization

Figures 8 (PCIe traffic) can also be reproduced, but requires additional runs, since the PCIe traffic monitoring tool is only capable of measuring one direction (either RX or TX) at a time. To enable PCIe monitoring, set `NVML_PROFILER` to 1 in `penguin-sc.h` and `penguin-suv.h`. Then set `NVML_TX` to 1 to measure TX traffic and to 0 to measure RX traffic. The PCIe traffic is reported in the output of the program execution. Please use the helper script `set_profiler_pcie.sh` provided as follows. The first argument enables/disables PCIe traffic profiling, while the second argument sets the direction (1 for TX, 0 for RX).

```
bash set_profiler_pcie.sh 1 1
```

Then run

```
bash compile_pcie.sh 1   # for TX
bash run_pcie.sh 1   # for TX
```

We have provided a convenience script, `pcie.sh` to compile, run and parse the experiments for PCIe traffic.

To change the oversubscription ratio, please edit the `compile.sh` and `run.sh` files, and change the `oversub` array in both files.

I. Notes

The obtained results may be slightly different based on conditions like system load.

REFERENCES

[1] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 224–235.

[2] S. Go, H. Lee, J. Kim, J. Lee, M. K. Yoon, and W. W. Ro, "Early-adaptor: An adaptive framework forproactive uvm memory management," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 248–258.

[3] X. Long, X. Gong, B. Zhang, and H. Zhou, "Deep learning based data prefetching in CPU-GPU unified virtual memory," *Journal of Parallel and Distributed Computing*, vol. 174, pp. 19–31, apr 2023. [Online]. Available: https://doi.org/10.1016%2Fj.jpdc.2022.12.004

[4] Q. Yu, B. Childers, L. Huang, C. Qian, H. Guo, and Z. Wang, "Coordinated page prefetch and eviction for memory oversubscription management in gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 472–482.

[5] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

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

[7] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 49–63. [Online]. Available: https://doi.org/10.1145/3297858.3304044

[8] NVIDIA, "Nvidia tesla v100 gpu architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2018.

[9] LLVM, "Compiling cuda with clang," https://llvm.org/docs/CompileCudaWithLLVM.html, 2023.

[10] NVIDIA, "open-gpu-kernel-modules," https://github.com/NVIDIA/open-gpu-kernel-modules/tree/main, 2023.

[11] N. Sakharnykh, "Everything you need to know about unified memory," https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf, 2018.

[12] NVIDIA, "open-gpu-kernel-modules," https://github.com/NVIDIA/open-gpu-kernel-modules/blob/main/kernel-open/nvidia-uvm/uvm_perf_prefetch.c, 2023.

[13] NVIDIA, "open-gpu-kernel-modules," https://github.com/NVIDIA/open-gpu-kernel-modules/blob/main/kernel-open/nvidia-uvm/uvm_gpu_access_counters.c, 2023.

[14] L. Li and B. Chapman, "Compiler assisted hybrid implicit and explicit gpu memory management under unified address space," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356141

[15] NVIDIA, "Cuda toolkit," https://developer.nvidia.com/cuda-toolkit, 2023.

[16] Micron, "Hbm3e," https://www.micron.com/products/ultra-bandwidth-solutions/hbm3, 2023.

[17] Micron, "Gddr6: The next-generation graphics dram," https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tned03_gddr6.pdf, 2017.

[18] M. Harris, "Unified memory for cuda beginners," https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.

[19] IBM, "Ibm power system ac922 introduction and technical overview," https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf, 2018.

[20] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[21] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis and transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[22] LLVM, "Llvm language reference manual," https://llvm.org/docs/LangRef.html, 2023.

[23] LLVM, "The often misunderstood gep instruction¶," https://llvm.org/docs/GetElementPtr.html#the-often-misunderstood-gep-instruction, 2024.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[25] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed characterization of deep neural networks on gpus and fpgas," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 12–21. [Online]. Available: https://doi.org/10.1145/3300053.3319418

[26] Z. Jin and J. S. Vetter, "A benchmark suite for improving performance portability of the sycl programming model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 325–327.

[27] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance gpu b-tree," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 145–157. [Online]. Available: https://doi.org/10.1145/3293883.3295706

[28] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for gpu accelerated computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3480855

[29] T. Allen and R. Ge, "Demystifying gpu uvm cost with deep runtime and workload analysis," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 141–150.

[30] S. Chien, I. Peng, and S. Markidis, "Performance evaluation of advanced features in cuda unified memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019, pp. 50–57.

[31] R. Gayatri, K. Gott, and J. Deslippe, "Comparing managed memory and ats with and without prefetching on nvidia volta gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 41–46.

[32] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A gpu memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 136–150. [Online]. Available: https://doi.org/10.1145/3123939.3123975

[33] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "Hierarchical page eviction policy for unified memory in gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 149–150.

[34] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpus for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

[35] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in cpu-gpu unified memory," in *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021, pp. 1212–1217.

[36] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 451–461.

[37] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 607–618. [Online]. Available: https://doi.org/10.1145/2694344.2694381

[38] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 354–365.

[39] C.-H. Chang, A. Kumar, and A. Sivasubramaniam, "To move or not to move? page migration for irregular applications in over-subscribed gpu memory systems with dynamap," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456727.3463766

[40] J. Jung, J. Kim, and J. Lee, "Deepum: Tensor migration and prefetching in unified memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 207–221. [Online]. Available: https://doi.org/10.1145/3575693.3575736

[41] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 598–611.

[42] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1341–1355. [Online]. Available: https://doi.org/10.1145/3373376.3378530

[43] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based gpu memory management for deep learning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 891–905. [Online]. Available: https://doi.org/10.1145/3373376.3378505

[44] H. Zhang, Y. Zhou, Y. Xue, Y. Liu, and J. Huang, "G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 395–410. [Online]. Available: https://doi.org/10.1145/3613424.3614309

[45] D. Ha, Y. Oh, and W. W. Ro, "R2d2: Removing redundancy utilizing linearity of address generation in gpus," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589039

[46] T. T. Yeh, R. N. Green, and T. G. Rogers, "Dimensionality-aware redundant simt instruction elimination," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1327–1340. [Online]. Available: https://doi.org/10.1145/3373376.3378520

[47] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for gpgpu performance, energy efficiency, and opportunistic reliability enhancement," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 433–442. [Online]. Available: https://doi.org/10.1145/2464996.2465022

[48] K. Wang and C. Lin, "Decoupled affine computation for simt gpus," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 295–306.

[49] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 142–151. [Online]. Available: https://doi.org/10.1145/1993498.1993516

[50] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 165–174. [Online]. Available: https://doi.org/10.1145/2259016.2259038

[51] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "Automatic gpu memory management for large neural models in tensorflow," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–13. [Online]. Available: https://doi.org/10.1145/3315573.3329984

[52] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–53. [Online]. Available: https://doi.org/10.1145/3178487.3178491

[53] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, p. 40–52, apr 1991. [Online]. Available: https://doi.org/10.1145/106975.106979

[54] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133382.2133384

[55] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito, "Compiler-based data prefetching and streaming non-temporal store generation for the intel(r) xeon phi(tm) coprocessor," in *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1575–1586.

[56] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 829–842.

[57] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-centric data and threadblock management for massive gpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1022–1036.

[58] J. Lowe-Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proceedings of International Symposium on High-Performance Computer Architecture*, ser. HPCA '14, 02 2014, pp. 568–578.

[59] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[60] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 637–650. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173194

[61] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering translation bandwidth with virtual caching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 113–127. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173195

[62] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 161–171.

[63] A. Jaleel, E. Ebrahimi, and S. Duncan, "Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3309710

[64] X. Tang, Z. Zhang, W. Xu, M. T. Kandemir, R. Melhem, and J. Yang, "Enhancing address translations in throughput processors via compression," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 191–204. [Online]. Available: https://doi.org/10.1145/3410463.3414633

[65] T. Baruah, Y. Sun, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Valkyrie: Leveraging inter-tlb locality to enhance gpu performance," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 455–466. [Online]. Available: https://doi.org/10.1145/3410463.3414639

[66] J. B. Kotra, M. LeBeane, M. T. Kandemir, and G. H. Loh, "Increasing gpu translation reach by leveraging under-utilized on-chip resources," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1169–1181. [Online]. Available: https://doi.org/10.1145/3466752.3480105

[67] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-gpus via sharing and spilling aware tlb design," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1154–1168. [Online]. Available: https://doi.org/10.1145/3466752.3480083

[68] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *SIGPLAN Not.*, vol. 53, no. 2, p. 503–518, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173169

[69] B. Pratheek, N. Jawalkar, and A. Basu, "Improving gpu multi-tenancy with page walk stealing," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 626–639.

[70] P. B, N. Jawalkar, and A. Basu, "Designing virtual memory system of mcm gpus," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '22. IEEE Press, 2023, p. 404–422. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00036

[71] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular gpu applications," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 352–363. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00036