DrGPUM: Guiding Memory Optimization for GPU-Accelerated Applications

Mao Lin
University of California, Merced
Merced, California, USA
mlin59@ucmerced.edu

Keren Zhou
OpenAI
San Francisco, California, USA
kerenzhou@openai.com

Pengfei Su
University of California, Merced
Merced, California, USA
psu9@ucmerced.edu

ABSTRACT

GPUs are widely used in today's computing platforms to accelerate applications in various domains. However, scarce GPU memory resources are often the dominant limiting factor in strengthening the applicability of GPU computing. In this paper, we propose DrGPUM, the first profiler that systematically investigates patterns of memory inefficiencies in GPU-accelerated applications. The strength of DrGPUM, when compared to a large class of existing GPU profilers, is its ability to *(1)* correlate problematic memory usage with data objects and GPU APIs, *(2)* identify and categorize object-level and intra-object memory inefficiencies, and *(3)* provide rich insights to guide memory optimization.

DrGPUM works on fully-optimized and unmodified GPU binaries, requires no modification to hardware or OS, and features a user-friendly GUI, which makes it attractive to use in production. Our evaluation with well-known benchmarks and applications shows DrGPUM's effectiveness in identifying memory inefficiencies with moderate overhead. Eliminating these inefficiencies requires less than nine source lines of code modifications and yields significant reductions in peak memory usage (up to 83%) and/or significant performance improvements (up to 2.48×). Our optimization patches have been confirmed by application developers and upstreamed to their repositories.

CCS CONCEPTS

• **General and reference** → **Metrics**; **Measurement**; • **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Parallel architectures**.

KEYWORDS

GPUs, GPU profilers, Memory management, CUDA

ACM Reference Format:
Mao Lin, Keren Zhou, and Pengfei Su. 2023. DrGPUM: Guiding Memory Optimization for GPU-Accelerated Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582044

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582044

1 INTRODUCTION

GPUs are widely used in modern and emerging computing platforms to accelerate applications in various domains, including deep learning (DL) and high performance computing (HPC). However, scarce GPU memory resources are often the dominant limiting factor in enhancing the applicability of GPU computing. For DL, the trend of developing deeper and wider neural networks demands significant GPU memory resources. For instance, training a trillion-parameter model with 3D parallelism requires up to 320 high-end NVIDIA A100 GPUs [53] as each A100 GPU only offers 80 GB on-board memory [11]. For HPC, the trend of pursuing high-resolution simulation of climate projections, quantum chromodynamics, and molecular dynamics is accompanied by the ever-increasing usage of GPU memory. For instance, a global climate model needs to be deployed to a supercomputer of 4888 NVIDIA P100 GPUs to achieve a 1 km resolution [18].

There is a long line of research and engineering work in the DL community that overcomes the hurdle of GPU memory capacity by either reducing peak memory usage of GPU-accelerated applications or incorporating the CPU memory as an external buffer (including DRAM and NVRAM). Such techniques primarily include recomputation [6, 23, 34, 52, 66], swapping [30, 52, 56, 57, 66], and compression [7, 17, 17, 31, 51]. The recomputation-based approaches reclaim temporarily-unused data in the forward propagation phase and then recompute them in the backward propagation phase. The swapping-based approaches copy temporarily-unused data to CPUs in the forward propagation phase and copy them back to GPUs in the backward propagation phase. These approaches, however, often suffer from nontrivial performance degradation due to expensive recomputation and frequent CPU-GPU data transfers. The compression-based approaches employ lossless or lossy compression algorithms to compress intermediate data (e.g., activation maps) and decompress them when needed. Unfortunately, the former suffers from a limited compression ratio, while the latter is prone to accuracy loss.

Unlike the aforementioned approaches that specifically target deep neural networks (DNNs), unified memory [26] is a general solution applicable to a broad range of GPU-accelerated applications. On NVIDIA GPUs, unified memory has been supported since CUDA 6.0, which enables GPUs to access both CPU and GPU memories without explicit data movement. On the downside, however, it can bring a severe performance loss (up to 10× slowdown [40]) due to expensive page migrations.

In summary, existing solutions lack wide applicability as they are designed for a specific class of GPU workloads (e.g., DNNs), incur severe performance degradation due to expensive recomputation/data

transfers/page migrations, and suffer from a low compression ratio or accuracy loss.

1.1 Proposed Work

To complement the missing pieces in existing approaches, we explore memory optimization opportunities with DrGPUM, a novel object-centric GPU memory profiler that consists of macroscopic object-level analysis (e.g., arrays and tensors) and microscopic intra-object analysis (e.g., individual elements in an array/tensor.)

- *Macroscopic object-level analysis.* DrGPUM tracks and collects object-level accesses across GPU APIs (i.e., memory allocation, deallocation, copy, and set, and kernel launch APIs), and presents them in a topological order to identify a class of inefficiencies at the granularity of data objects.
- *Microscopic intra-object analysis.* DrGPUM monitors intra-object accesses (i.e., memory accesses in a data object) to identify never- and infrequently-accessed regions in a data object for space compression and discern hot regions from cold ones in a data object for better data placement in the GPU memory hierarchy.

Together, they present a comprehensive view of memory inefficiencies (including memory wastage and inefficient memory accesses) for GPU-accelerated applications and provide actionable insights for code optimization.

Although memory profilers are abundant in the GPU community, including vendor-provided profilers [3, 19, 41, 44, 47–49] and research profilers [2, 20, 35, 55, 59, 67, 69, 73], they tend to monitor GPU memory usage from only a macroscopic point of view. Without correlating data objects with GPU APIs, they provide little actionable insight for developers. To the best of our knowledge, TensorBoard [19] is the only object-centric GPU profiler. However, it applies to TensorFlow-based DL workloads only and instruments program source code to monitor tensors (i.e., data objects). Thus, this approach is not application-independent. Additionally, it lacks a systematic investigation of patterns of memory inefficiencies (described in Section 3) and thus cannot provide intuitive optimization insights needed for developer action or detect as many memory inefficiencies as DrGPUM can.

DrGPUM differs fundamentally from the existing solutions in several aspects. First, it is independent of applications (e.g., TensorFlow), models (e.g., DNNs), and compilers and works on fully-optimized and unmodified GPU binaries. Second, it requires no source code modification or recompilation, which is suitable for production software that requires cumbersome dependencies for compilation and proprietary code packages without source code. Last but not least, it provides intuitive program insights and suggestions to enable object-level and intra-object optimizations. To the best of our knowledge, DrGPUM is the first general and object-centric profiler to guide memory optimization for GPU-accelerated applications.

In the remainder of this section, we first describe two motivating examples, showing the unique optimization opportunities detectable only via DrGPUM. We then summarize the contributions of this paper.

1.2 Motivating Examples

We study Laghos [14] and MiniMDock [63], two DOE applications, and identify the following memory inefficiencies.

Object-level inefficiency: late deallocations. Laghos solves the time-dependent Euler equations of compressible gas dynamics. Listing 1 shows the problematic code snippet. Vector `q_dx` (line 2) is a member variable of class `QUpdate` and resides in the GPU memory. We find that the last access to `q_dx` occurs in function `UpdateQuadratureData()` at line 7, whereas `q_dx` is not released until the end of the program execution. For memory saving, one can proactively invoke a GPU memory deallocation API (e.g., `cudaFree`) to release `q_dx` just after `UpdateQuadratureData()` is finished. This optimization reduces the memory peak by 18%. Identifying this kind of inefficiency requires tracking all GPU APIs that access `q_dx`.

Intra-object inefficiency: overallocations. MiniMDock is a particle-grid based protein-ligand molecular docking tool. Listing 2 shows the problematic code snippet, which always allocates a maximum constant-size memory chunk for array `pMem_conformations` on the GPU (line 2) without considering the actual size specified by program inputs (line 7). As a result, many elements in `pMem_conformations` are never accessed. By adjusting the allocated memory size based on the inputs, one is able to achieve up to a 64% reduction in peak memory usage. Identifying this kind of inefficiency requires monitoring accesses to individual elements in `pMem_conformations`.

```
1 class QUpdate {
2 ▶ Vector q_dx;
3   ...
4 }
5 qupdate = new QUpdate(...);
6 // The last function that accesses q_dx.
7 ▶ qupdate->UpdateQuadratureData(...);
```

Listing 1: Late memory deallocations in Laghos. The last access to member variable `q_dx` of class `QUpdate` occurs in function `UpdateQuadratureData()`, whereas `q_dx` is not released until the end of the program execution.

```
1 void setup_gpu_for_docking(...) {
2 ▶ size_population = MAX_NUM_OF_RUNS * MAX_POPSIZE *
      GENOTYPE_LENGTH_IN_GLOBMEM * sizeof(float);
3   cudaMalloc((void**)& (pMem_conformations), size_population);
4 }
5 ...
6 void docking_with_gpu(..., Dockpars& mypars) {
7 ▶ size_population = mypars->num_of_runs * mypars->pop_size *
      GENOTYPE_LENGTH_IN_GLOBMEM * sizeof(float);
8   cudaMemcpy(pMem_conformations, cpu_init, size_population,
        cudaMemcpyHostToDevice);
9 }
```

Listing 2: Memory overallocations in `MiniMDock`. Array `pMem_conformations` is always allocated with a maximum constant-size memory chunk without considering the actual size specified by program inputs.

1.3 Contribution Summary

In this paper, we make the following contributions:

- Investigate and categorize patterns of memory inefficiencies involved in GPU code, which serves as the foundation of DrGPUM.

- Develop macroscopic object-level and microscopic intra-object analyses and integrate them into DrGPUM to perform multi-scale memory profiling on the GPU.
- Show that DrGPUM can monitor fully-optimized and unmodified GPU binaries and provide rich insights to guide memory optimization. Such insights include an object-centric view, liveness analysis, call paths of GPU APIs, inefficiency patterns, as well as metrics that can prioritize identified inefficiencies to facilitate optimization.
- Demonstrate the effectiveness of DrGPUM in identifying various forms of memory inefficiencies in well-known benchmarks and applications. Eliminating these inefficiencies requires less than nine source lines of code modifications and yields significant reductions in peak memory usage and/or significant performance improvements.

2 RELATED WORK

There exist many optimization and profiling techniques for assisting GPU application developers in tuning their code, such as saving memory [6, 29, 31, 56, 66], eliminating redundant memory accesses [61, 62, 67, 71, 72], reducing memory access latency [33, 37], increasing parallelism [28, 70], and minimizing synchronization overhead [16, 24, 74]. This section only reviews the techniques that are closely related to memory optimization and profiling on GPUs.

2.1 Memory Optimization

Recomputation, swapping, and compression are widely-used memory optimization techniques on GPUs. *Recomputation* was first proposed to explore a tradeoff between time and space overheads in Automatic Differentiation [22]. Recent work has extended recomputation to develop deeper and wider DNNs. Gruslys et al. [23] leveraged dynamic programming in conjunction with memoization and recomputation to find an optimal balance between memory usage and recomputation cost. Superneurons [66] performs layer-wise recomputation; an improved variant of this work was proposed in a later research [52], which performs tensor-wise recomputation to achieve fine-grained memory management.

Swapping-based approaches swap data between GPU and CPU memories. vDNN [56] offloads tensors that are not used in the near future to the CPU memory and prefetches them to the GPU memory when reuse is required. Capuchin [52] makes swapping decisions on the fly based on dynamic tensor access patterns. Unlike vDNN and Capuchin that focus on swapping tensors, SwapAdvisor [30] swaps both tensors and parameters with a custom-designed genetic algorithm to achieve a higher memory-saving ratio.

Compression-based approaches directly reduce the size of certain data on GPUs and restore them when needed without data transfers between CPUs and GPUs. Lal et al. [36] developed a Huffman-based lossless memory compressor based on a probability estimation of symbols. COMET [31] adopts an error-bounded lossy compression algorithm to reduce peak memory usage in convolutional neural network (CNN) training. JPEG-ACT [17] extends an existing JPEG-based lossy compression algorithm to CNNs by exploiting the similarities between images in computer vision and tensors in CNNs. Compared to lossless compression algorithms, lossy ones achieve a higher compression ratio with a certain level of accuracy loss.

DrGPUM differs from the aforementioned approaches. First, as a profiler, DrGPUM reports inefficiencies and provides optimization suggestions but does not modify application code, whereas the aforementioned approaches do. Based on DrGPUM's report, users make optimization choices, which can include recomputation, swapping, compression, and many others. Second, DrGPUM is generally applicable to GPU-accelerated applications, whereas the aforementioned approaches typically target a specific class of GPU workloads.

2.2 Memory Profiling

Most popular GPU profilers in industry and academia employ profiling interfaces such as CUPTI API [10], Sanitizer API [8], PC sampling [12], SASSI [60], or NVBit [65] available for NVIDIA GPUs to collect and analyze memory metrics. They, however, do not correlate data objects with GPU APIs, thus missing out on many optimization opportunities that DrGPUM can identify via object-level and intra-object analyses.

Zhou et al. [71] proposed GVProf, a value-aware profiler to identify redundant memory accesses; their follow-up work [72] improves on GVProf's profiling techniques to identify more redundancy patterns. Diogenes [67] identifies duplicate memory copies by monitoring values copied from CPUs to GPUs. Orthogonal to them, DrGPUM targets value-agnostic inefficiencies. Besides, GVProf and Diogenes detect inefficient memory accesses only; DrGPUM not only detects inefficient memory accesses but also memory wastage. CUDAAdvisor [58] instruments memory instructions using an LLVM compiler pass to measure reuse distance and memory divergence. DARSIE [69] leverages an LLVM compiler pass to identify redundant memory instructions. Unlike them, DrGPUM is independent of compilers and can monitor legacy and proprietary code packages that are only available in the form of binary files.

3 PATTERNS OF MEMORY INEFFICIENCIES AND OPTIMIZATION GUIDANCE

Our observation, which is justified by myriad cases under investigation, is that the following ten patterns of memory inefficiencies pervasively exist in GPU-accelerated applications. They fall into two categories based on the granularity of inefficiencies — object-level and intra-object memory inefficiencies. The former focuses on accesses to individual data objects by treating the data object as a whole and the latter focuses on accesses to individual elements in each data object. Table 1 summarizes patterns of memory inefficiencies found by DrGPUM in popular GPU programs, including benchmarks and applications. In the remaining section, we elaborate on each pattern and their respective optimization guidance.

3.1 Object-level Memory Inefficiencies

Definition 3.1 (Early Allocation)**.** A data object O is allocated before GPU APIs $\mathcal{A}_1$ and $\mathcal{A}_2$ [1]. O matches the *early allocation*

[1] GPU APIs include memory allocation, deallocation, copy, and set, and kernel launch.

Table 1: Patterns of memory inefficiencies found in popular GPU programs.

Programs		Object-level inefficiency patterns							Intra-object inefficiency patterns		
		Early Allocation	Late Deallocation	Redundant Allocation	Unused Allocation	Memory Leak	Temporary Idleness	Dead Write	Overallocation	Non-uniform Access Frequency	Structured Access
Rodinia [5]	huffman	✓	✓	✓	✓		✓				
	dwt2d	✓	✓	✓	✓		✓	✓			
PolyBench [21]	2MM	✓	✓	✓							
	3MM	✓	✓	✓			✓				
	Gram-Schmidt	✓	✓				✓			✓	✓
	BICG	✓	✓	✓						✓	
PyTorch [50]		✓	✓	✓	✓		✓				
Laghos [14]		✓	✓	✓	✓		✓	✓			
Darknet [54]		✓	✓	✓	✓	✓	✓	✓			
XSBench [64]						✓			✓		
MiniMDock [63]		✓	✓		✓		✓		✓		
SimpleMultiCopy [13]		✓	✓				✓	✓			

pattern if $\mathcal{A}_2$ is the first GPU API that accesses O and $\mathcal{A}_1$ is executed earlier than $\mathcal{A}_2$ [2].

Definition 3.2 (Late Deallocation)**.** A data object O is deallocated after GPU APIs $\mathcal{A}_1$ and $\mathcal{A}_2$. O matches the *late deallocation* pattern if $\mathcal{A}_1$ is the last GPU API that accesses O and $\mathcal{A}_2$ is executed later than $\mathcal{A}_1$.

The early allocation and late deallocation patterns are prevalent in GPU-accelerated applications. The main reason is that application developers tend to *eagerly* allocate memory in batches (typically at the beginning of a program) and *lazily* reclaim allocated memory in batches (typically at the end of a program), leading to memory wastage. For memory saving, the best course of action is to defer a data object's allocation until the time just before the first GPU API that accesses this data object is executed and reclaim the allocated memory as soon as GPU APIs do not access it.

Definition 3.3 (Redundant Allocation)**.** $\mathcal{A}_1$ is the last GPU API that accesses a data object O_1 and $\mathcal{A}_2$ is the first GPU API that accesses another data object O_2. O_2 matches the *redundant allocation* pattern with O_1 if $\mathcal{A}_1$ ends before $\mathcal{A}_2$ starts and O_1 does not exceed $\mathcal{X}$% of O_2 in size.

Reusing an already-allocated data object not only saves memory but also improves performance because it reduces the frequency of performing expensive GPU memory allocation operations (i.e., calling `cudaMalloc` family of functions). The optimization is straightforward; one can reuse the already-allocated data object to avoid allocating a new one of (approximately) equal size. It is worth noting that $\mathcal{X}$ is user-tunable; based on our experiments, we set $\mathcal{X}$% to 10%.

Definition 3.4 (Unused Allocation)**.** A data object O matches the *unused allocation* pattern if it is not accessed by GPU APIs.

Definition 3.5 (Memory Leak)**.** A data object O matches the *memory leak* pattern if it is not deallocated by the end of execution.

Based on case studies, we observe that the unused allocation and memory leak patterns are fairly common, especially in large codebases where memory allocations can be hidden deep and go unnoticed if not carefully designed or periodically monitored. For optimization, one should remove unused allocations and guarantee that memory allocation and deallocation operations always appear in pairs to avoid memory leaks.

Definition 3.6 (Temporary Idleness)**.** $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ are the i^{th} and $(i+1)^{th}$ GPU APIs that access a data object O. If at least $\mathcal{X}$ GPU APIs are executed between $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$, O matches the *temporary idleness* pattern in this time range.

A data object is temporarily idle during the execution of GPU APIs that do not access it. The temporary idleness pattern is prevalent in DNNs. For instance, some activation maps accessed in the forward propagation phase are not accessed by subsequent forward GPU APIs until the backward propagation phase. One can reduce memory consumption by offloading temporarily-idle data objects from the GPU memory to the CPU memory and bringing them back when reuse is required. $\mathcal{X}$ is user-tunable; based on our experiments, we set $\mathcal{X}$ to 2.

Definition 3.7 (Dead Write)**.** A data object O matches the *dead write* pattern if no intervening GPU APIs access O between two memory copies, two memory sets, or one memory copy and one memory set (or vice versa) that write O.

A dead write occurs when a write operation, such as a memory copy or set, to a data object is overwritten by the following write operation without any use of the data object in between. This pattern exposes useless memory operations during program execution. For optimization, one can remove dead writes without affecting program results.

3.2 Intra-object Memory Inefficiencies

Definition 3.8 (Overallocation)**.** A data object O matches the *overallocation* pattern if less than $\mathcal{X}$% elements in O are accessed by GPU APIs.

The overallocation pattern occurs if a data object holds many elements while only a small portion of elements are accessed. Not only do unnecessary elements waste memory, but also they result in suboptimal data placement in the GPU memory hierarchy. NVIDIA Ampere and Hopper architectures provide L2 cache residency controls for users [43, 46]; with this facility, one can keep/evict data in/from the L2 cache. However, due to the limited L2 cache size, users could fail to make an overallocated data object that exhibits

[2]Note that a GPU memory allocation/deallocation API allocates/deallocates memory for a data object but does not access that data object.

Table 2: Optimization guidance on memory overallocations. We recommend that users focus their optimization efforts on data objects with both low percentages of accessed elements and memory fragmentation.

% of accessed elements	% of fragmentation	Optimization guidance
Low	Low	Easy to optimize and shrinking/freeing unaccessed memory yields nontrivial benefit to memory saving.
High	Low	Shrinking/freeing unaccessed memory yields little benefit to memory saving.
Low	High	Difficult to optimize because unaccessed elements are scattered all over the data object.
High	High	No action on memory saving.

strong locality of reference fit into the L2 cache, leading to unnecessary cache misses. For optimization, one needs to track every access to a data object to locate unaccessed elements and then works out a solution to shrink/free them. $\mathcal{X}$ is user-tunable, which quantifies the severity of memory wastage in O. Besides, we compute the percentage of memory fragmentation for O with the following equation, which quantifies the difficulty of shrinking/freeing unaccessed memory.

$$\mathcal{F}rag_O = 1 - \frac{\text{Size of the largest unaccessed memory chunk in } O}{\text{Size of unaccessed memory in } O} \quad (1)$$

Guided by these two metrics, our optimization guidance is shown in Table 2. Only data objects with both low percentages of accessed elements and memory fragmentation are worth efforts for optimization. Based on our experiments, we investigate a data object iff both percentages are less than 80%.

Definition 3.9 (Non-uniform Access Frequency)**.** A data object O matches the *non-uniform access frequency* pattern at a GPU API $\mathcal{A}$ if the variance [3] of access frequencies of different elements in O is higher than $\mathcal{X}$%.

As on-chip memory, shared memory and cache enjoy much lower latency than global memory (~100× speedup [25]). However, due to their limited sizes, oftentimes, only a small portion of a data object can reside in shared memory and cache. By tracking access frequencies of individual elements in a data object, one can discern hot slices from cold ones within the data object and then place the hot ones in shared memory and cache to accelerate memory accesses. $\mathcal{X}$ is user-tunable; based on our experiments, we set $\mathcal{X}$% to 20%.

Definition 3.10 (Structured Access)**.** A data object O matches the *structured access* pattern if each GPU API $\mathcal{A}_i$ accesses a slice of O and these slices do not overlap.

The structured access pattern exposes a unique memory-saving opportunity — replace a single memory allocation with multiple smaller memory allocations whose lifetimes do not overlap. PolyBench/GramSchmidt [21] falls into such an example. We find that array `R_gpu` is split into many disjoint slices and each slice is accessed by a distinct instance of GPU kernel `gramschmidt_kernel3` (invoked in a loop). For memory saving, instead of allocating a large chunk of memory for the entire data object once and for all, one can allocate memory for each slice separately. For example, one can allocate memory for a slice just before it is accessed by a kernel instance, free the allocated memory just after the kernel instance is finished, and repeat this process until all kernel instances are finished.

[3] We employ the coefficient of variation metric [68] to compute variance.

4 DRGPUM WORKFLOW

DrGPUM is versatile and practical for production use: it works on fully-optimized and unmodified GPU binaries, does not require any hardware or OS modification, and features a user-friendly GUI. Figure 1 shows the workflow of DrGPUM, which consists of an online data collector, an online pattern detector, an offline analyzer, and an offline GUI.

Online data collector. The data collector leverages NVIDIA's Sanitizer API [8] to intercept every GPU API to collect object-level information (i.e., a data object's lifetime, start address, and size) and instrument every memory instruction to collect intra-object information (i.e., which element(s) in a data object a memory instruction touches). Also, it collects call paths of GPU APIs to provide deep insights.

Online pattern detector. The pattern detector analyzes the data obtained from the data collector to detect patterns of memory inefficiencies. The detector performs object-level pattern detection on the GPU to accelerate the analysis and minimize CPU-GPU memory traffic, and performs intra-object pattern detection on the CPU/GPU in an adaptive fashion to avoid exhausting the GPU memory. The output is composed of object-level and intra-object inefficiency patterns, along with their respective call paths and optimization suggestions.

Offline analyzer. The analyzer extracts line mapping information (e.g., source code lines, file names, and file paths) from the DWARF [15] debugging sections in executables and dynamically loaded libraries, and associates them with inefficiency patterns obtained from the pattern detector. Also, the analyzer pinpoints data objects involved in memory peaks, which are highlighted by DrGPUM's GUI. This way, DrGPUM narrows down the investigation scope to help users focus on data objects appearing on critical paths. In our experiments, DrGPUM reports data objects involved in the top two memory peaks, which is user-tunable.

Offline GUI. The GUI is built atop an existing web-based graphical interface, Perfetto UI [39], which supports popular data formats (e.g., JSON and Protocol Buffers) and runs in mainstream browsers (e.g., Chrome, Firefox, Safari, and Microsoft Edge). Figure 7 in Section 7.1 shows an example of the GUI, which visualizes data objects' lifetimes, a topological order of GPU APIs, as well as inefficiency patterns and optimization suggestions correlated with data objects. We defer the explanation of the GUI details to that section.

5 METHODOLOGY AND IMPLEMENTATION

In this section, we describe the implementation of DrGPUM's core components — data collector and pattern detector. DrGPUM's multi-scale analysis includes object-level and intra-object analyses.

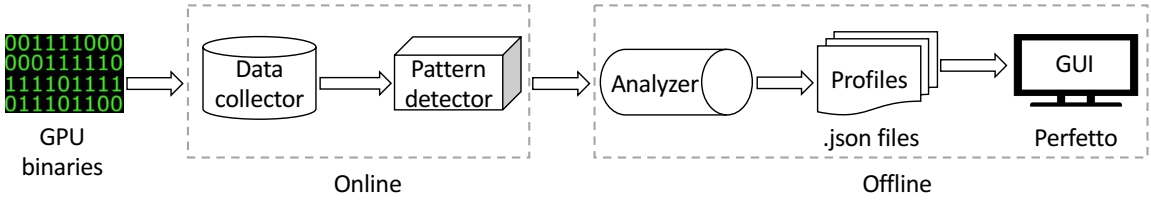

Figure 1: The Workflow of DrGPUM.

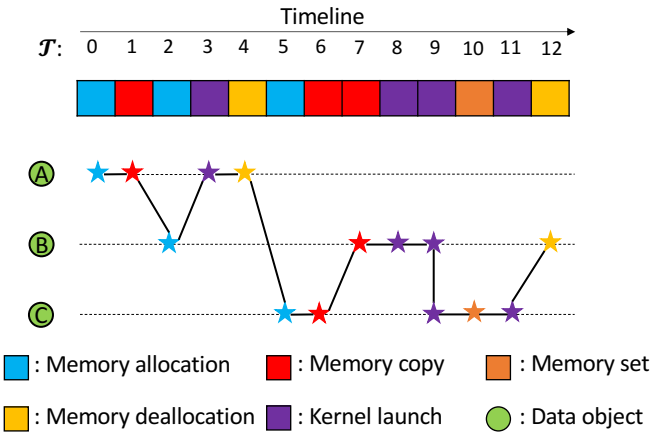

Figure 2: The mental model of a timestamp-augmented object-level memory access trace (denoted by the thick solid line). On the trace, each pentagram represents access to a data object (denoted by the circle), along with a GPU API (denoted by the rectangle) that accesses this data object at the timestamp $\mathcal{T}$. Data object B matches the early allocation and late deallocation patterns, and data object C matches the memory leak and temporary idleness patterns.

The former analyzes accesses to individual data objects by treating the data object as a whole and the latter analyzes accesses to individual elements in each data object.

5.1 Identifying Object-level Memory Inefficiencies

The key component of DrGPUM's object-level analysis is a memory access trace, as shown in Figure 2, which is a unique view among all existing GPU profilers. On the trace, each pentagram represents access to a data object (denoted by the circle). The sequence of rectangles represents GPU APIs invoked at different timestamps $\mathcal{T}$ that access the corresponding data objects below. DrGPUM associates each GPU API invocation with a timestamp, which is aligned with its invocation order. To construct the memory access trace, DrGPUM keeps track of active data objects and GPU APIs. DrGPUM intercepts memory allocation and deallocation API invocations to record necessary information for each data object. At the invocation of each memory allocation API, DrGPUM obtains the address range of the allocated data object as the key, unwinds the call path using libunwind [42] as the value, and inserts them into a memory map $\mathcal{M}$. At the invocation of each memory deallocation API, DrGPUM removes the corresponding record from $\mathcal{M}$. At the invocation of each memory copy, memory set, or kernel launch API, DrGPUM looks up $\mathcal{M}$ with addresses this invocation touches to determine which data objects have been accessed.

Automating pattern detection. With the help of a timestamp-augmented memory access trace, DrGPUM can automate the analysis of memory inefficiencies and provide actionable optimization insights. Consider O as a data object, and assume that O is allocated and deallocated at $\mathcal{T}_O^{alloc}$ and $\mathcal{T}_O^{free}$, and the first and last GPU APIs that access O are invoked at $\mathcal{T}_O^{first}$ and $\mathcal{T}_O^{last}$. DrGPUM walks through the memory access trace from the node with $\mathcal{T}_O^{alloc}$ to the node with $\mathcal{T}_O^{free}$ and checks O's inefficiency patterns based on the following rules:

- *Early Allocation:* If there exist GPU API invocations between $\mathcal{T}_O^{alloc}$ and $\mathcal{T}_O^{first}$.
- *Late Deallocation:* If there exist GPU API invocations between $\mathcal{T}_O^{last}$ and $\mathcal{T}_O^{free}$.
- *Unused Allocation:* If O is not accessed by GPU APIs between $\mathcal{T}_O^{alloc}$ and $\mathcal{T}_O^{free}$.
- *Memory Leak:* If there is no GPU memory deallocation API associated with O.
- *Temporary Idleness:* If there exist at least two (user-tunable) GPU API invocations between two consecutive GPU APIs that access O.
- *Dead Write:* If O is not accessed by GPU APIs between two memory copies, two memory sets, or one memory copy and one memory set (or vice versa) that write O.

For example, based on the trace in Figure 2, DrGPUM identifies that data object B matches the early allocation pattern because there exist four GPU API invocations between the time when B is allocated ($\mathcal{T}$ = 2) and the time when the first GPU API that accesses B is invoked ($\mathcal{T}$ = 7). Also, DrGPUM identifies that B matches the late deallocation pattern because there exist two GPU API invocations between the time when the last GPU API that accesses B is invoked ($\mathcal{T}$ = 9) and the time when B is freed ($\mathcal{T}$ = 12).

Detecting redundant allocations. Detecting the redundant allocation pattern is different from detecting the others. A data object can reuse the memory of another data object freed before it is allocated, while it can also be reused by others allocated after it is freed. To address this challenge, we developed a one-pass algorithm that scans the memory access trace once to suggest potential data object reuse pairs, as illustrated in Figure 3. ① For each data object, we extract the first and last GPU APIs (including memory set, memory

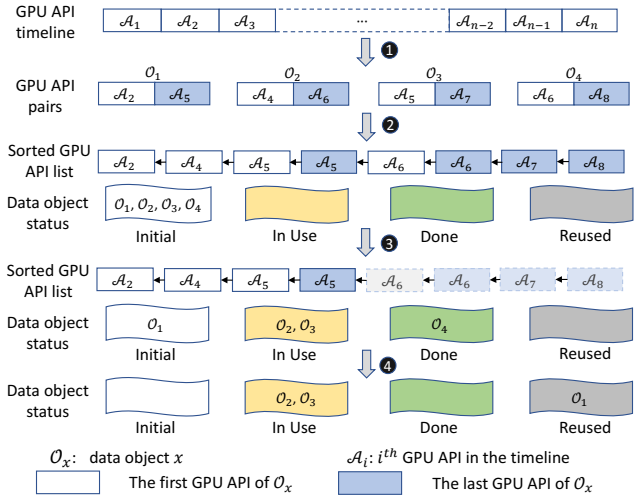

Figure 3: An example of identifying redundant allocations.

copy, and kernel launch APIs) that access it from the memory access trace. ② For data objects whose sizes are within a user-defined percentage threshold of difference (10% in our experiments), we sort their first and last GPU APIs based on each GPU API's timestamp, and the last GPU API is placed after the first GPU API if they have the same timestamp. For example, in Figure 3, the last GPU API $\mathcal{A}_5$ of O_1 is placed after the first GPU API $\mathcal{A}_5$ of O_3. ③ We traverse the sorted GPU API list from the tail to the head. During the traversal, each data object can be changed from one status to another. *(a)* `Initial` represents a data object's initial status. *(b)* `In Use` indicates that a data object's last GPU API has been visited, but its first GPU API has not. *(c)* `Done` indicates that a data object's first and last GPU APIs have been visited. *(d)* `Reused` indicates a data object has been reused, so it is no longer available for reuse by other objects (but it can still reuse others.). ④ Once a data object O_i turns to the `Done` status, we select the closest data object O_j to it from the left with the `Initial` status and recommend that O_i reuses O_j's memory. After O_j has been selected for reuse, its status becomes `Reused`. In Figure 3, when the first GPU API $\mathcal{A}_6$ of O_4 is visited on the GPU API list, O_4's status is changed to `Done`. Then, we claim that O_4 can reuse O_1's memory and change O_1's status to `Reused`.

5.2 Identifying Intra-object Memory Inefficiencies

DrGPUM's intra-object analysis instruments every memory instruction (i.e., memory read and write) in GPU binaries to track accesses to individual elements within a data object. Upon the execution of each instrumented memory instruction, DrGPUM obtains its accessed memory range (i.e., start address and size) to determine which object has been accessed and which elements in this object have been accessed.

- *Overallocation:* DrGPUM maintains a bitmap for each data object, of which each bit corresponds to the access status of an element in the data object. The initial status of each bit in a data object's bitmap is "untouched" (0). Upon the execution of an instrumented memory instruction, DrGPUM sets the states of corresponding bits to "accessed" (1) in the accessed data object's bitmap. After all GPU APIs are finished, DrGPUM examines each data object's bitmap to compute the percentage of accessed elements and identifies data objects whose accessed percentages are less than a user-defined percentage threshold (80% in our experiments).
- *Structured Access:* Different from the overallocation pattern, DrGPUM creates multiple bitmaps for a data object, where each bitmap represents memory accesses of a GPU API. For each data object, if its bitmaps of different GPU APIs do not overlap, DrGPUM reports the structured accessed pattern.
- *Non-uniform Access Frequency:* DrGPUM maintains a hashmap for each data object, where each cell stores the access frequency of an element in the data object. Upon the invocation of a GPU API $\mathcal{A}$, DrGPUM zeros out hashmaps of data objects this GPU API will access. Upon the execution of an instrumented memory instruction, DrGPUM increases the access frequencies of corresponding elements by one in the accessed data object's hashmap. After $\mathcal{A}$ is finished, DrGPUM examines each accessed data object's hashmap to compute the variance of access frequencies of individual elements. If the variance is higher than a user-defined percentage threshold (20% in our experiments), DrGPUM reports the non-uniform access frequency pattern and plots the hashmap as a histogram to facilitate optimization. DrGPUM repeats the above steps until all GPU APIs are finished.

5.3 Handling Multi-stream GPU Applications

While DrGPUM can sequence GPU APIs according to their invocation order for single-stream applications, the scenario of multi-stream applications, where GPU APIs dispatched on different streams are executed concurrently, complicates matters. DrGPUM addresses this challenge by constructing a *dependency graph* to track dependencies across GPU APIs and establish a topological order for them.

Definition 5.1 (Dependency Graph). A dependency graph $G = (V, E)$ is a directed acyclic graph, where V is the set of vertices and E is the set of edges.

- Each vertex $v \in V$ represents a GPU API, i.e., a memory allocation, memory deallocation, memory copy, memory set, or kernel launch.
- An edge $e_{i,j} \in E$ exists from v_i to v_j if one of the following holds:
  - v_i and v_j pertain to the same stream and v_j is v_i's immediate successor. In this case, $e_{i,j}$ denotes the intra-stream execution dependency between v_i and v_j.
  - v_i allocates/writes a data object O_x, v_j reads O_x, and no $v_k \in V$ writes O_x following v_i and before v_j. In this case, $e_{i,j}$ is labeled with a read operation for v_j, denoting a *read-after-write* (RAW) dependency on O_x between v_i and v_j.
  - v_i allocates/writes a data object O_x, v_j frees/writes O_x, and no $v_k \in V$ reads/writes O_x following v_i and before v_j. In this case, $e_{i,j}$ is labeled with a free/write operation for v_j, denoting a *write-after-write* (WAW) dependency on O_x between v_i and v_j.
  - v_i reads a data object O_x, v_j frees/writes O_x, and no $v_k \in V$ writes O_x following v_i and before v_j. In this case, $e_{i,j}$ is labeled

with a free/write operation for v_j, denoting a *write-after-read* (WAR) dependency on O_x between v_i and v_j.

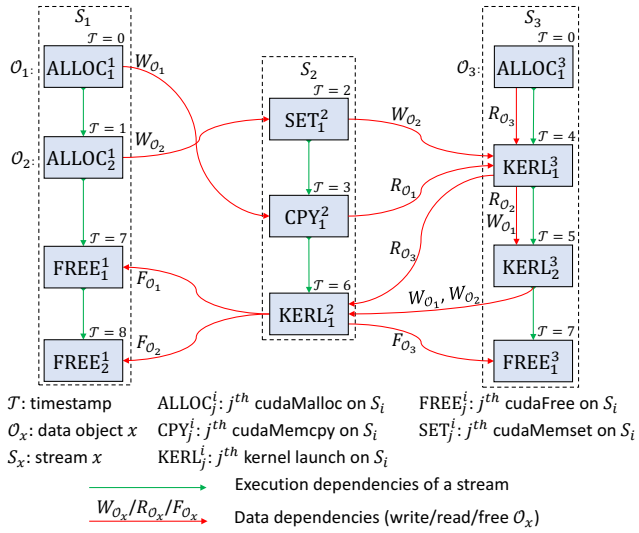

Figure 4: The mental model of a timestamp-augmented dependency graph, where vertices represent GPU APIs (denoted by rectangles), and edges represent data dependencies (denoted by red edges) and intra-stream execution dependencies (denoted by green edges). The red edges indicate the execution order for inter-stream GPU APIs, while the green edges indicate the execution order for intra-stream GPU APIs.

Figure 4 shows an example of constructing a dependency graph based on Definition 5.1, where vertices represent GPU APIs and edges represent dependencies across GPU APIs. DrGPUM tracks two kinds of dependencies: RAW/WAW/WAR data dependencies (denoted by red edges) and intra-stream execution dependencies (denoted by green edges). The former indicates the execution order for GPU APIs pertaining to different streams, while the latter indicates the execution order for GPU APIs pertaining to the same stream (GPU APIs are executed sequentially within a stream.). To track data dependencies, DrGPUM monitors the flow of data objects across GPU APIs. To track intra-stream execution dependencies, DrGPUM monitors GPU API invocations within each stream, where the invocation order is consistent with the execution order.

After construction of the dependency graph, we employ Kahn's topological sorting algorithm [32] in conjunction with a global timestamp $\mathcal{T}$ to annotate the *topological order* of vertices in the graph. The topological order ensures that, for any given directed edge, the execution of the origin vertex precedes that of the destination vertex. We elaborate on major steps of the algorithm as follows. ① DrGPUM initializes $\mathcal{T}$ to zero. ② DrGPUM searches the dependency graph for vertices with an in-degree of zero, assigns $\mathcal{T}$ to them, and deletes them along with their outgoing edges from the graph. ③DrGPUM updates in-degrees of vertices adjacent to the deleted vertices. ④ DrGPUM increases $\mathcal{T}$ by one. ⑤ DrGPUM repeats steps ②-④ until the graph is empty. In the end, every vertex is annotated with a timestamp, denoting its topological order in the graph.

With the help of a timestamp-augmented dependency graph as shown in Figure 4, DrGPUM can obtain the *inefficiency distance* between any two dependent vertices (i.e., dependent GPU APIs) by computing the difference between their timestamps, which helps quantify the severity of identified inefficiencies. For example, in Figure 4, data object O_1 is allocated at $\mathcal{T}[\text{ALLOC}_1^1] = 0$, and the first access to O_1 occurs at $\mathcal{T}[\text{CPY}_1^2] = 3$; thus, O_1 matches the early allocation pattern and the inefficiency distance is 3.

5.4 Handling Custom GPU Memory APIs

Existing DL frameworks, such as PyTorch [50] and TensorFlow [1], typically pre-allocate a large chunk of GPU memory, used as a memory pool, at the beginning of the execution. Then, users can request and release tensors from the memory pool using custom GPU memory APIs with low overhead. As NVIDIA's Sanitizer API has no visibility into custom GPU memory APIs, we developed a memory profiling interface for PyTorch and integrate it into DrGPUM. Our memory profiling interface registers a callback through PyTorch's `ThreadLocalDebugInfo` utility to monitor every allocation and deallocation operations on PyTorch's memory pool. At each memory operation, we associate it with a call path consisting of Python frames to identify where this operation occurred and update the total amount of allocated and reserved memory.

5.5 Accelerating Pattern Analysis

Without optimization, DrGPUM can easily exhaust scarce GPU memory and introduce high time overhead. For instance, without optimization, DrGPUM incurs up to a 1170× time overhead when monitoring Darknet [54]. We propose several strategies to reduce DrGPUM's time and memory overheads.

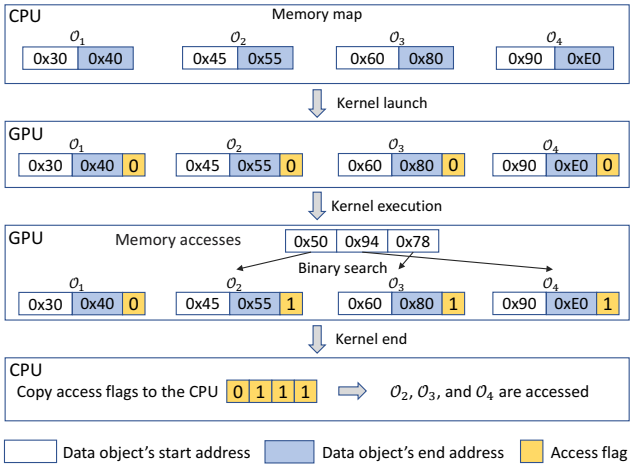

Figure 5: An example of identifying data objects accessed by kernels on the GPU.

Accelerating object-level analysis. Naively, we can record information about every GPU memory access in a buffer and copy the buffer to the CPU when it is full to construct the memory access trace. However, this method can result in frequent CPU-GPU data

Table 3: The configurations of two GPU platforms: NVIDIA RTX 3090 and A100 GPUs

CPU	GPU	GPU memory	System	GPU driver	CUDA Toolkit	GCC	Compiler options
Intel(R) Xeon(R) 4316	RTX 3090	24 GB GDDR6X	Linux 5.4	460.32	11.2.1	9.4.0	-g -lineinfo -O3 -arch sm_86
AMD EPYC 7402	A100	40 GB HBM2	Linux 4.18	460.27	11.1.1	8.3.1	-g -lineinfo -O3 -arch sm_80

transfers and long execution time on the CPU. To accelerate object-level analysis, we offload major steps of constructing the memory access trace to the GPU, as shown in Figure 5. We maintain a memory map $\mathcal{M}$ on the CPU, which records the address range of every active data object. We copy $\mathcal{M}$ to the GPU at each kernel launch [4] and associate each entry with a hit flag, representing whether the kernel accesses this data object. The initial status of every hit flag is "untouched" (0). For each GPU memory access, we adopt a binary search to locate which data object the address belongs to in $\mathcal{M}$ and set the corresponding hit flag to "accessed" (1). After the kernel is finished, we copy all hit flags to the CPU to identify which data objects have been accessed by the kernel. We gained nontrivial speedup from this design. For instance, the object-level analysis time of Darknet is reduced from 1.5 hours to 12 seconds.

Accelerating intra-object analysis. For intra-object analysis, DrGPUM needs to fill each memory access record into the corresponding cell of bitmaps/hashmaps. In the following, we refer to the bitmap/hashmap as the access map. DrGPUM implements two modes to collect information: *(a)* copying all accessed addresses from the GPU to the CPU and updating access maps on the CPU; *(b)* updating access maps on the GPU and copying the final results back to CPU after each kernel is finished. Option (b) is much faster than option (a) since updates can be done via atomic operations by massive threads on the GPU. However, the size of access maps in option (b) is subject to the GPU memory capacity; option (a) does not impose this limitation in most cases. We have to ensure the total size of access maps and active data objects does not exceed the GPU memory capacity. To this end, DrGPUM computes the total size of access maps and data objects before each kernel launch. If the total size is smaller than the GPU memory capacity, DrGPUM selects option (b); otherwise, it selects option (a). This method strikes a good balance between profiling overhead and profiling applicability. In addition, DrGPUM adopts kernel sampling and kernel whitelist [71] to further reduce its time and memory overheads. The former is based on an observation that code behaviors typically remain similar across different instances of the same GPU kernel. The latter allows users to specify one or more GPU kernels of interest for monitoring.

5.6 Discussion

First, like other dynamic profilers, DrGPUM's profiling output is input-dependent. Therefore, we recommend using typical program inputs for representative profiles. Second, DrGPUM identifies memory inefficiencies and provides optimization suggestions but does not implement automatic inefficiency fixing mechanisms. Third, DrGPUM does not incur false positives; all reported memory inefficiencies are real inefficiencies. However, users' postmortem analysis is necessary to make appropriate and safe optimization choices. Last but not least, DrGPUM is built atop NVIDIA's Sanitizer API, which works on NVIDIA GPUs only. However, the proposed methodology is generally applicable to other GPU brands if necessary binary instrumentation engines are available.

[4]There is no need to copy $\mathcal{M}$ to the GPU at the invocation of a memory copy or set API because NVIDIA's Sanitizer API provides facilities to directly obtain the accessed address range.

6 EVALUATION

We evaluated DrGPUM on two GPU platforms shown in Table 3. We applied DrGPUM on well-known HPC benchmarks, such as Rodinia [5], PolyBench [21], XSBench [64], and SimpleMultiCopy [13], as well as DL and HPC applications, such as PyTorch [50], Darknet [54], Laghos [14], and MiniMDock [63].

Overhead measurement. Figure 6 shows the overhead of DrGPUM, which is measured as the ratio of the execution time of a program with DrGPUM enabled to the execution time of its native execution. To reduce system noises, we run every program 10 times and compute the average. DrGPUM incurs moderate overhead for both object-level and intra-object analyses. For object-level analysis, DrGPUM has a median overhead of 1.45× and 1.30×, and a geometric mean overhead of 2.19× and 2.28× on RTX 3090 and A100, respectively. For intra-object-analysis, DrGPUM has a median overhead of 3.55× and 4.13×, and a geometric mean overhead of 3.66× and 3.31× on RTX 3090 and A100, respectively. We observe three takeaways. First, A100 enjoys lower overhead over programs that involve many memory accesses (e.g., Polybench/2MM) because of its higher bandwidth than RTX 3090. Second, MiniMDock suffers the highest overhead on both machines because it involves significant memory accesses and GPU API invocations. The former results in high overhead for intra-object analysis and the latter results in high overhead for object-level analysis. Third, the A100 machine incurs noticeably higher overhead over Rodinia/dwt2d than the RTX 3090 machine because most execution time in Rodinia/dwt2d is spent on the CPU side and the AMD EPYC 7402 CPU offers worse performance than the Intel(R) Xeon(R) 4316 CPU.

Program optimization. Table 4 summarizes the memory inefficiencies reported in GPU programs using DrGPUM, and peak memory reductions and speedups obtained by eliminating the inefficiencies. From the table, we can see that nontrivial peak memory reductions and speedups are achieved with just a few source lines of code modifications. To guarantee optimization correctness, we ensure the optimized code does not change program semantics for any inputs and passes validation tests. Note that all the inefficiencies were found and fixed by a graduate student familiar with CUDA programming but with no prior knowledge of the profiled programs. Thanks to DrGPUM's intuitive optimization guidance, that student typically spent ~1.5 hours fixing each inefficiency.

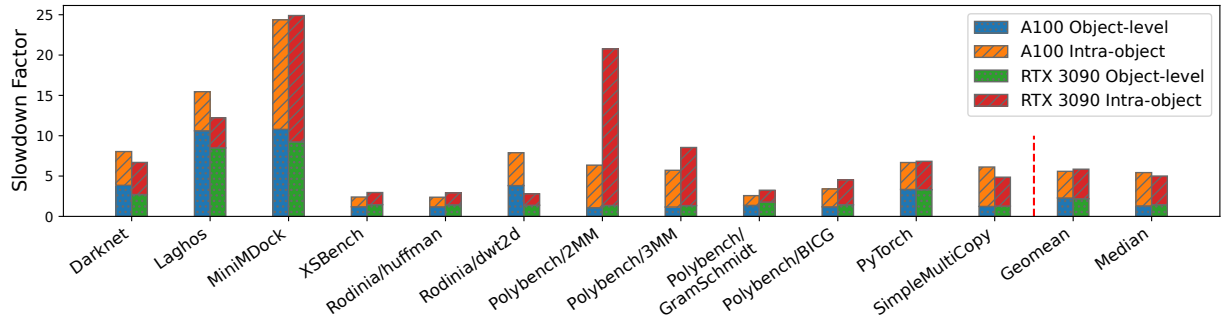

Figure 6: DrGPUM's overhead on benchmarks and applications. In object-level analysis, DrGPUM monitors all GPU APIs without sampling for each benchmark and application. In intra-object analysis, DrGPUM monitors the GPU kernel that has the largest memory footprint and sets the kernel sampling period to 100 for each benchmark and application.

Table 4: Overview of peak memory reductions and performance gains guided by DrGPUM.

Programs		Inefficiencies		Optimizations				Domains
		Data objects	Patterns	SLOC mod.	Peak mem. reductions*	Speedups RTX 3090	Speedups A100	
Rodinia	huffman	d_cw32 d_sourceData	UA LD	2 2	67%	–	–	Lossless compression
	dwt2d	c_r_out c_g_out backup c_g	EA RA UA TI	4 2 4 5	48%	–	–	Image/video compression
Poly-Bench	2MM	A_gpu B_gpu D_gpu	LD RA EA	2 5 4	40%	–	–	Matrix multiplication
	3MM	A_gpu C_gpu E_gpu F_gpu	RA LD TI EA	5 2 4 4	57%	–	–	Matrix multiplication
	Gram-Schmidt	R_gpu	SA NUAF	6 4	33% –	– 1.39×	– 1.30×	Gram-Schmidt decomposition
	BICG	s_gpu q_gpu	NUAF NUAF	8 8	–	2.06×	2.48×	Linear solver
PyTorch		columns	UA	3	3%	–	–	Deep learning
Laghos		q_dx q_dy	LD LD	2 2	35%	–	–	LAGrangian solver
Darknet		l.weights_gpu l.output_gpu l.delta_gpu	DW EA UA	1 3 2	83%	–	–	Deep learning
XSBench		GSD.concs GSD.index_grid	ML OA	1 8	63%	–	–	Neutronics
MiniMDock		pMem_ conformations	OA	2	64%	–	–	Molecular biology
SimpleMultiCopy		d_data_in1 d_data_out1 d_data_in2 d_data_out2	TI EA LD LD	4 2 2 2	50%	–	–	Data communication

EA: Early Allocation, LD: Late Deallocation, RA: Redundant Allocation, UA: Unused Allocation, ML: Memory Leak, TI: Temporary Idleness, DW: Dead Write, OA: Overallocation, NUAF: Non-uniform Access Frequency, SA: Structured Access.
*: The optimization yields the same peak memory reduction for each program on RTX 3090 and A100.

7 CASE STUDIES

This section dives into several case studies listed in Table 4 to show how DrGPUM guides optimization. We ran each program with its default input unless otherwise specified.

7.1 SimpleMultiCopy

SimpleMultiCopy [13] is a multi-stream sample program released with NVIDIA CUDA Toolkit [9]. Figure 7 shows a snapshot of DrGPUM's web-based GUI that visualizes the profile of SimpleMultiCopy for intuitive analysis. The GUI consists of three panes. The top pane shows the topological order of GPU APIs in the timeline. The middle pane presents data objects related to the top two memory peaks in SimpleMultiCopy, along with GPU APIs that access these data objects. The bottom pane provides rich information for each GPU API, including call paths, inefficiency patterns, inefficiency distances, and optimization suggestions. This figure highlights that data object `d_data_out1` matches the early allocation pattern because it is allocated at `ALLOC(0, 2)`, which is three GPU APIs (i.e., `ALLOC(0, 3)`, `SET(0, 2)`, and `ALLOC(0, 4)`) before its first-touch GPU API `KERL(0, 1)`. For memory saving, DrGPUM suggests deferring the allocation of `d_data_out1` just before `KERL(0, 1)`.

The middle pane also involves other inefficiency patterns, of which detailed insights are not shown in the figure due to the page limit. ① Data object `d_data_in1` is temporarily idle during the execution of `ALLOC(0, 2)`, `ALLOC(0, 3)`, `SET(0, 2)`, and `ALLOC(0, 4)`; DrGPUM suggests freeing `d_data_in1` just before `ALLOC(0, 2)` and reallocating it just after `ALLOC(0, 4)`. ② Data objects `d_data_in2` and `d_data_out2` match the late deallocation pattern; DrGPUM suggests freeing them immediately after their respective last-touch GPU APIs.

By incorporating the aforementioned optimizations, we cut the peak memory usage by 50% on RTX 3090 and A100.

7.2 Darknet

We applied DrGPUM to profile the inference phase of the YOLOv4 [4] model with Darknet, a popular DL framework. DrGPUM identifies that array `l.weights_gpu` matches the dead write pattern. Further investigation shows that `l.weights_gpu` is initialized twice without an intervening read in the forward phase of each convolutional layer, as shown in Listing 3. The first initialization (i.e., the dead write) occurs in function `cuda_make_array()` at line 2, which allocates memory for `l.weights_gpu` on the GPU and then initializes it with `l.weights`, an array on the CPU. The second initialization occurs in function `cuda_push_array()` at line 7, which initializes `l.weights_gpu` again. To eliminate the unnecessary initialization, as per Darknet's API manual, we change the first parameter of `cuda_make_array()` to zero (line 3) such that it only allocates memory on the GPU without initialization.

DrGPUM also identifies other inefficiency patterns in Darknet, which are not shown in Listing 3. Array `l.output_gpu` matches the early allocation pattern; it is allocated in the networking parsing phase (i.e., the beginning of the execution) but unused until the forward phase of convolutional layers. Array `l.delta_gpu` matches the unused pattern; it is allocated in the Route layer but unused during the entire inference phase. After optimizing all the identified inefficiencies, we reduce Darknet's memory peak by 83% on RTX 3090 and A100.

7.3 Polybench/GramSchmidt and BICG

Polybench/GramSchmidt [21] is a method for orthonormalizing a set of vectors in an inner product space. DrGPUM reports that array `R_gpu` matches the structured access pattern at GPU kernel `gramschmidt_kernel3`. Our investigation reveals that `gramschmidt_kernel3` is invoked in a hot loop and accesses a slice of `R_gpu` in each invocation. These slices are the same size and do not overlap, as shown in Figure 8. To save memory, before entering the loop, we request GPU memory for a slice of `R_gpu` instead of for the whole `R_gpu` and reuse it across different invocation instances of `gramschmidt_kernel3`. This optimization reduces the memory peak by 33% on RTX 3090 and A100. DrGPUM further reports that the variance of access frequencies of individual slices in `R_gpu` is 58%, indicating `R_gpu` matches the non-uniform access frequency pattern. For optimization, we sort the slices based on their access frequencies and place the top 60% hottest slices in shared memory, which yields a 1.39× speedup on RTX 3090 and a 1.30× speedup on A100, respectively. It is worth noting that an even higher speedup can be achieved by fine-tuning the percentage of slices that reside in shared memory.

Using DrGPUM, we find the same inefficiency pattern in arrays `s_gpu` and `q_gpu` in Polybench/BICG [21], a linear system solver. By performing the same optimization on `s_gpu` and `q_gpu`, we obtain a 2.06× speedup on RTX 3090 and a 2.48× speedup on A100, respectively.

7.4 PyTorch

Resnet50 [27] is a 50-layer convolutional neural network. We profiled it on PyTorch [50]. DrGPUM reveals that tensor `columns` matches the unused allocation pattern, which is created and assigned memory at line 2 in Listing 4. Further investigation shows that `columns` is accessed only when the binary condition in the ternary operator `?:` at line 6 is true. Thus, we can conditionally bypass the memory allocation for `columns` (line 4) when the binary condition in `?:` is false. Adding this simple conditional check reduces the peak memory usage of convolutional layers by 3% on RTX 3090 and A100. This patch has been upstreamed to the PyTorch repository.

7.5 XSBench

XSBench [64], a mini-app developed by the Argonne National Laboratory, models the Monte Carlo neutron transport algorithm. DrGPUM reports that array `GSD.index_grid` matches the overallocation pattern. The percentage of accessed elements in `GSD.index_grid` is 5%, indicating most elements are unaccessed. Our investigation reveals that `GSD.index_grid` consists of many equal-sized chunks and each GPU thread accesses one chunk only. Consequently, most chunks are untouched. To save memory, we reclaim all untouched memory chunks. Our optimization yields a 63% reduction in peak memory usage on RTX 3090 and A100, and has been upstreamed to the XSBench repository.

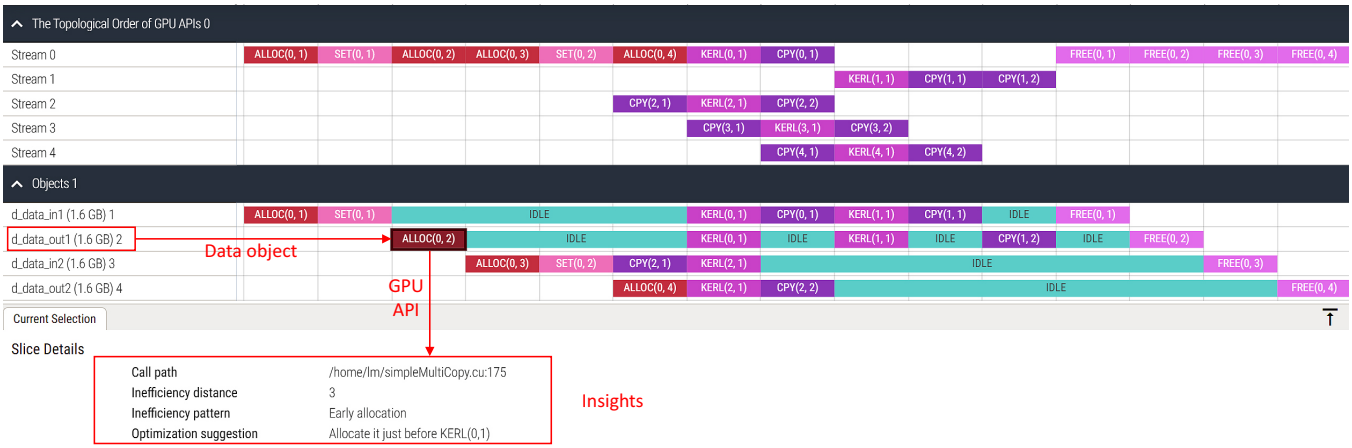

Figure 7: DrGPUM's report for SimpleMultiCopy, showing that data object `d_data_out1` matches the early allocation pattern with detailed optimization insights. ALLOC(i, j): j^{th} cudaMalloc on stream i, SET(i, j): j^{th} cudaMemset on stream i, CPY(i, j): j^{th} cudaMemcpy on stream i, FREE(i, j): j^{th} cudaFree on stream i, KERL(i, j): j^{th} kernel launch on stream i.

```
convolutional_layer make_convolutional_layer(...) {
- l.weights_gpu = cuda_make_array(l.weights, l.nweights);
+ l.weights_gpu = cuda_make_array(0, l.nweights);
   ...}
...
void push_convolutional_layer(...) {
   cuda_push_array(l.weights_gpu, l.weights, l.nweights);
   ...}
```

Listing 3: The dead write pattern in Darknet. `l.weights_gpu`, an array on the GPU, is initialized twice without an intervening read in the forward phase of each convolutional layer.

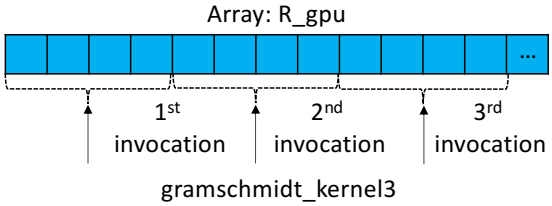

Figure 8: The mental model of the structured access pattern in GramSchmidt. `gramschmidt_kernel3` is invoked in a loop and accesses a slice of array `R_gpu` in each invocation. These slices do not overlap.

7.6 MiniMDock

DrGPUM detected the overallocation pattern described in Section 1.2. DrGPUM reports that array `pMem_conformations` is the largest data object in MiniMDock, of which only 2.4E-3% elements are accessed and memory fragmentation is as low as 4.89E-3%. By applying the optimization described in Section 1.2, we obtain a 64% reduction in peak memory usage on RTX 3090 and A100. This patch has been upstreamed to the MiniMDock repository.

```
void slow_conv2d_forward(...) {
- auto columns = at::empty({nInputPlane * kW * kH, outputHeight *
      outputWidth}, input.options());
+ at::Tensor columns;
+ if (requires_columns) {
+   columns = at::empty({nInputPlane * kW * kH, outputHeight *
      outputWidth}, input.options());}
   auto gemm_in_ptr = requires_columns ? columns.data_ptr<scalar_t
      >() : input_n.data_ptr<scalar_t>();
}
```

Listing 4: The unused allocation pattern in PyTorch. Tensor `columns` is always assigned with memory without considering whether the assigned memory is used.

7.7 Laghos

Using DrGPUM, we confirmed and optimized the late deallocation pattern described in Section 1.2. Besides, DrGPUM highlights that another member variable `q_dy` of class `QUpdate` suffers from the same inefficiency and guides the same optimization. Together, these two optimizations deliver a 35% reduction in peak memory usage on RTX 3090 and A100, and have been confirmed by Laghos developers.

7.8 Comparison with State-of-the-art Tools

We ran ValueExpert [72] and Compute Sanitizer (with the *memcheck* substrate) [8] on every benchmark and application studied in this paper to compare memory inefficiencies detected by each tool. Table 5 draws an overview of the comparison results, from which we can make the following key conclusions:

1 Except for unused allocations, ValueExpert failed to detect any inefficiencies reported by DrGPUM. ValueExpert targets value-aware memory inefficiencies (e.g., consecutive writes of the same value to the same memory location), while DrGPUM targets value-agnostic memory inefficiencies.
2 Except for memory leaks, Compute Sanitizer failed to detect any inefficiencies reported by DrGPUM. Compute Sanitizer is highly specialized for detecting various forms of memory errors, such as memory leaks, out-of-bound memory accesses, and misaligned memory accesses, but not memory inefficiencies. It is worth

Table 5: DrGPUM vs. state-of-the-art tools: whether the inefficiencies detected by DrGPUM could be detected by other tools.

Inefficiency patterns	DrGPUM	ValueExpert	Compute Sanitizer
Early Allocation	Yes	No	No
Late Deallocation	Yes	No	No
Redundant Allocation	Yes	No	No
Unused Allocation	Yes	Yes*	No
Memory Leak	Yes	No	Yes
Temporary Idleness	Yes	No	No
Dead Write	Yes	No	No
Overallocation	Yes	No	No
Non-uniform Access Frequency	Yes	No	No
Structured Access	Yes	No	No

*: While ValueExpert does not report unused allocations, users can reason about them with ease based on ValueExpert's profiling output.

noting that Compute Sanitizer can detect memory leaks caused not only by the host-side `cudaMalloc` but also the device-side `malloc` [45], while DrGPUM can detect the former only.

8 CONCLUSIONS AND FUTURE WORK

This paper presents DrGPUM, the very first profiler that systematically investigates patterns of memory inefficiencies in GPU-accelerated applications and provides rich insights to guide optimization. DrGPUM can identify both object-level and intra-object memory inefficiencies, which helps isolate memory usage problems in complex codebases. DrGPUM works on fully-optimized and unmodified GPU binaries, requires no modification to hardware or OS, and features a user-friendly GUI, making it attractive to production software. Guided by DrGPUM, we are able to optimize several GPU benchmarks and applications. The results show significant peak memory reductions and/or significant performance improvements. Our optimization patches have been confirmed or accepted by application developers.

We envision two future directions. One is to enable DrGPUM to support TensorFlow. The other is to explore memory inefficiencies beyond GPU code. We will investigate both CPU and GPU code to identify memory inefficiencies that reside in CPU-GPU interactions, such as page-level false sharing in unified memory.

ACKNOWLEDGEMENTS

We thank all the anonymous reviewers for their valuable comments. This work was supported by National Science Foundation under CNS-2125732.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes DrGPUM and benchmarks, along with instructions to generate results for Table 1, Table 4, Figure 6, and Figure 7 on NVIDIA A100 and RTX 3090 GPUs. The memory peak reduction and DrGPUM's profiling overhead for each benchmark are averaged among 10 runs.

We provide a docker image with pre-installed prerequisites to simplify the experiment workflow. Users can also start from scratch.

A.2 Artifact Check-list (Meta-information)

- **Benchmarks: Laghos@54977f7, AlexeyAB/darknet@b4d03f8, XSBench@ff7e9d4, miniMDock@3811014, PyTorch@34b0285, Rodinia v3.1, polybench-gpu@08b2fb0**
- **Run-time environment: Linux x86-64 systems.**
- **Hardware: NVIDIA Volta GPUs and later generations.**
- **Metrics: Memory inefficiency patterns, memory peak reductions, and profiling overhead**
- **Output: An example of DrGPUM GUI. A file that contains memory peak reductions for all benchmarks. A file that contains profiling overhead for all benchmarks.**
- **How much disk space required (approximately)?: 300 GB**
- **How much time is needed to prepare workflow (approximately)?: 4 hours**
- **How much time is needed to complete experiments (approximately)?: 5 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: BSD-3**
- **Archived (provide DOI)?: doi.org/10.5281/zenodo.7588406**

A.3 Description

A.3.1 How to Access. The artifact is published on Zenodo [38].

A.3.2 Hardware Dependencies. DrGPUM currently only works on NVIDIA Volta GPUs and generations above. We have tested DrGPUM's correctness and performance on machines equipped with NVIDIA A100 and RTX 3090 GPUs. To reproduce the results in the paper, we suggest the reviewers use the same GPUs and a machine with at least 300 GB of available disk space.

A.3.3 Software Dependencies.

- NVIDIA CUDA driver: ≥ 460.27
- CUDA Toolkit: 11.1.1 and above
- GCC: 8.3.1 and above
- Linux Kernel: 4.18.0 and above

A.4 Installation

Decompress the packages and launch a docker instance.

```
7za x drgpum_ae_image.tar.7z
7za x docker_drgpum_home.7z
docker load -i drgpum_ae_image.tar
docker run --rm --runtime=nvidia -it -v \
`pwd`/docker_drgpum_home:/root  drgpum_ae
```

Install DrGPUM and benchmarks (4 hours).

```
cd /root
# [gpu_arch]=80 if you are using A100
# [gpu_arch]=86 if you are using RTX 3090
./scripts/install.sh [gpu_arch]     # ~4 hours
```

A.5 Experiment Workflow

Reproduce memory peak reductions in Table 4 and inefficiency pattern detection in Table 1 (1 hour).

```
cd /root && ./scripts/tables.sh     # ~1 hour
cat ./results/memory_peak.txt
cat ./results/patterns.txt
```

Reproduce overhead in Figure 6 and DrGPUM GUI in Figure 7 (4 hours). It will generate an *overhead.pdf* file and a *liveness.json* file under /root/results/.

```
cd /root
./scripts/overhead.sh   # generate overhead.pdf (~4 hours)
./scripts/generate_gui.sh   # generate liveness.json (~5 mins)
```

DrGPUM GUI is built atop Perfetto (https://ui.perfetto.dev/). To view DrGPUM GUI on Perfetto, click on *Open trace file* on the left pane of Perfetto and then upload *liveness.json*.

A.6 Evaluation and Expected Results

The reproduced results for Table 1, Table 4, Figure 6, and Figure 7 are expected to match the corresponding results in the paper. Table 1 shows inefficiency patterns and Table 4 shows memory peak reductions. Figure 6 shows DrGPUM's profiling overhead. Figure 7 shows a snapshot of DrGPUM GUI that visualizes the profile of SimpleMultiCopy. Our PRs for PyTorch 79183, XSBench 24, and miniMDock 2 have been accepted by the developers.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22 (2010), 685–701.

[3] AMD Inc. 2022. ROCm Profiler. https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html.

[4] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv preprint arXiv:2004.10934* (2020).

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. https://doi.org/10.48550/ARXIV.1604.06174

[7] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. 2020. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 926–939. https://doi.org/10.1109/ISCA45697.2020.00080

[8] NVIDIA Corp. 2022. Compute Sanitizer. https://docs.nvidia.com/cuda/compute-sanitizer.

[9] NVIDIA Corp. 2022. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[10] NVIDIA Corp. 2022. CUPTI. https://docs.nvidia.com/cupti.

[11] NVIDIA Corp. 2022. H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100.

[12] NVIDIA Corp. 2022. PC Sampling View. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#pc-sampling.

[13] NVIDIA Corp. 2022. SimpleMultiCopy - Simple Multi Copy and Compute. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/0_Introduction/simpleMultiCopy.

[14] Veselin A Dobrev, Tzanio V Kolev, and Robert N Rieben. 2012. High-order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5 (2012), B606–B641. https://doi.org/10.1137/120864672

[15] Michael Eager. 2012. The DWARF Debugging Standard. https://www.dwarfstd.org.

[16] Ahmed ElTantawy and Tor M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 375–388. https://doi.org/10.1109/HPCA.2018.00040

[17] R. David Evans, Lufei Liu, and Tor M. Aamodt. 2020. JPEG-ACT: Accelerating Deep Learning via Transform-based Lossy Compression. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 860–873. https://doi.org/10.1109/ISCA45697.2020.00075

[18] Oliver Fuhrer, Tarun Chadha, Torsten Hoefler, Grzegorz Kwasniewski, Xavier Lapillonne, David Leutwyler, Daniel Lüthi, Carlos Osuna, Christoph Schär, Thomas C. Schulthess, and Hannes Vogt. 2018. Near-global Climate Simulation at 1 KM Resolution: Establishing a Performance Baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development* 11, 4 (May 2018), 1665–1681. https://doi.org/10.5194/gmd-11-1665-2018

[19] Google LLC. 2022. TensorBoard. https://www.tensorflow.org/tensorboard.

[20] Anton V Gorshkov, Michael Berezalsky, Julia Fedorova, Konstantin Levit-Gurevich, and Noam Itzhaki. 2019. GPU Instruction Hotspots Detection Based on Binary Instrumentation Approach. *IEEE Trans. Comput.* 68, 8 (2019), 1213–1224. https://doi.org/10.1109/TC.2019.2896628

[21] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning A High-level Language Targeted to GPU Codes. In *2012 innovative parallel computing (InPar)*. IEEE, 1–10. https://doi.org/10.1109/InPar.2012.6339595

[22] Andreas Griewank. 1989. On Automatic Differentiation. In *Mathematical Programming: Recent Developments and Applications*. Kluwer Academic Publishers, 83–108.

[23] Audrūnas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation through Time. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 4132–4140.

[24] Manish Gupta, Daniel Lowell, John Kalamatianos, Steven Raasch, Vilas Sridharan, Dean Tullsen, and Rajesh Gupta. 2017. Compiler Techniques to Reduce The Synchronization Overhead of GPU Redundant Multithreading. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062212

[25] Mark Harris. 2013. Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc.

[26] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[28] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[29] Zhongzhe Hu, Junmin Xiao, Zheye Deng, Mingyi Li, Kewei Zhang, Xiaoyang Zhang, Ke Meng, Ninghui Sun, and Guangming Tan. 2022. MegTaiChi: Dynamic Tensor-Based Memory Management Optimization for DNN Training. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 25, 13 pages. https://doi.org/10.1145/3524059.3532394

[30] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. https://doi.org/10.1145/3373376.3378530

[31] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2021. COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy Compression. *Proc. VLDB Endow.* 15, 4 (dec 2021), 886–899. https://doi.org/10.14778/3503585.3503597

[32] A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (nov 1962), 558–562. https://doi.org/10.1145/368996.369025

[33] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. 2016. Warped-Preexecution: A GPU Pre-execution Approach for Improving Latency Hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 163–175. https://doi.org/10.1109/HPCA.2016.7446062

[34] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Vfs_2RnOD0H

[35] Andreas Knüpfer, Christian Rössel, Dieter Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Competence in High Performance Computing 2011*. Springer Berlin Heidelberg, 79–91.

[36] Sohan Lal, Jan Lucas, and Ben Juurlink. 2017. E2MC: Entropy Encoding Based Memory Compression for GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1119–1128. https://doi.org/10.1109/IPDPS.2017.101

[37] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 67–77. https://doi.org/10.1145/2751205.2751237

[38] Mao Lin, Keren Zhou, and Pengfei Su. 2023. Reproduction Package for Article "DrGPUM: Guiding Memory Optimization for GPU-Accelerated Applications". https://doi.org/10.5281/zenodo.7588406

[39] Google LLC. 2022. Perfetto: System Profiling, App Tracing and Trace Analysis. https://perfetto.dev.

[40] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training Deeper Models by GPU Memory Optimization on TensorFlow. In *Proceedings of ML Systems Workshop in NIPS*.

[41] Meta Inc. 2022. PyTorch Profiler. https://pytorch.org/docs/stable/profiler.html.

[42] David Mosberger and Dave Watson. 2022. The libunwind project. https://www.nongnu.org/libunwind.

[43] NVIDIA Corp. 2020. Ampere Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth.

[44] NVIDIA Corp. 2022. DCGM: Manage and Monitor GPUs in Cluster Environments. https://developer.nvidia.com/dcgm.

[45] NVIDIA Corp. 2022. Dynamic Global Memory Allocation and Operations. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations.

[46] NVIDIA Corp. 2022. Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth.

[47] NVIDIA Corp. 2022. Nsight Compute. https://developer.nvidia.com/nsight-compute.

[48] NVIDIA Corp. 2022. Nsight Systems. https://developer.nvidia.com/nsight-systems.

[49] NVIDIA Corp. 2022. NVProf: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint arXiv:1912.01703* (2019).

[51] Shaurya Patel, Tongping Liu, and Hui Guan. 2021. FreeLunch: Compression-based GPU Memory Management for Convolutional Neural Networks. In *2021 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 1–8. https://doi.org/10.1109/MCHPC54807.2021.00007

[52] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. https://doi.org/10.1145/3373376.3378505

[53] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[54] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. https://pjreddie.com/darknet.

[55] James Reinders. 2005. VTune Performance Analyzer Essentials. *Intel Press* (2005).

[56] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. VDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 18, 13 pages.

[57] Shriram S.B., Anshuj Garg, and Purushottam Kulkarni. 2019. Dynamic Memory Management for GPU-Based Training of Deep Neural Networks. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 200–209. https://doi.org/10.1109/IPDPS.2019.00030

[58] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/3168831

[59] Sameer S. Shende and Allen D. Malony. 2006. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (2006), 287–311. https://doi.org/10.1177/1094342006064482

[60] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/2749469.2750375

[61] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. 2019. Pinpointing Performance Inefficiencies in Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 818–829. https://doi.org/10.1145/3338906.3338923

[62] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant Loads: A Software Inefficiency Indicator. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 982–993.

[63] Mathialakan Thavappiragasam, Aaron Scheinberg, Wael Elwasif, Oscar Hernandez, and Ada Sedova. 2020. Performance Portability of Molecular Docking Miniapp On Leadership Computing Platforms. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 36–44. https://doi.org/10.1109/P3HPC51967.2020.00009

[64] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of A Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[65] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[66] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. 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* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3178487.3178491

[67] Benjamin Welton and Barton P. Miller. 2019. Diogenes: Looking for An Honest CPU/GPU Performance Measurement Tool. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 21, 20 pages. https://doi.org/10.1145/3295500.3356213

[68] Wikipedia. 2022. Coefficient of Variation. https://en.wikipedia.org/wiki/Coefficient_of_variation.

[69] Tsung Tai Yeh, Roland N. Green, and Timothy G. Rogers. 2020. *Dimensionality-Aware Redundant SIMT Instruction Elimination*. Association for Computing Machinery, New York, NY, USA, 1327–1340. https://doi.org/10.1145/3373376.3378520

[70] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. 2016. Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 609–621. https://doi.org/10.1109/ISCA.2016.59

[71] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2020. GVProf: A Value Profiler for GPU-Based Clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16. https://doi.org/10.1109/SC41405.2020.00093

[72] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: Exploring Value Patterns in GPU-Accelerated Applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 171–185. https://doi.org/10.1145/3503222.3507708

[73] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU Performance Advisor Based on Instruction Sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 115–125. https://doi.org/10.1109/CGO51591.2021.9370339

[74] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *The World Wide Web Conference (WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2494–2504. https://doi.org/10.1145/3308558.3313508

Received 2022-10-20; accepted 2023-01-19