GVPROF: A Value Profiler for GPU-Based Clusters

Keren Zhou
Department of Computer Science
Rice University
Houston, USA
keren.zhou@rice.edu

Yueming Hao
Department of Computer Science
North Carolina State University
Raleigh, USA
yhao24@ncsu.edu

John Mellor-Crummey
Department of Computer Science
Rice University
Houston, USA
johnmc@rice.edu

Xiaozhu Meng
Department of Computer Science
Rice University
Houston, USA
xm13@rice.edu

Xu Liu
Department of Computer Science
North Carolina State University
Raleigh, USA
xliu88@ncsu.edu

***Abstract*—GPGPUs are widely used in high-performance computing systems to accelerate scientific and machine learning workloads. Developing efficient GPU kernels is critically important to obtain "bare-metal" performance on GPU-based clusters. In this paper, we describe the design and implementation of GVPROF, the first value profiler that pinpoints value-related inefficiencies in applications running on NVIDIA GPU-based clusters. The novelty of GVPROF resides in its ability to detect temporal and spatial value redundancies, which provides useful information to guide code optimization. GVPROF can monitor production multi-node multi-GPU executions in clusters. Our experiments with well-known GPU benchmarks and HPC applications show that GVPROF incurs acceptable overhead and scales to large executions. Using GVPROF, we optimized several HPC and machine learning workloads on one NVIDIA V100 GPU. In one case study of LAMMPS, optimizations based on information from GVProf led to whole-program speedups ranging from 1.37x on a single GPU to 1.08x on 64 GPUs.**

***Index Terms*—High performance computing, Performance analysis, Parallel programming, Supercomputers**

I. INTRODUCTION

General-purpose graphics processing units (GPGPU) have become a popular accelerator on HPC platforms. Among the latest Top 500 list [1], more than 100 supercomputers employ CPU+GPU heterogeneous architectures; at present, two of the top three supercomputers—Summit and Sierra—employ NVIDIA Volta GPUs to provide most of the FLOPS on their compute nodes. Scientific software packages and machine learning workloads leverage GPUs to deliver superior performance.

Obtaining "bare-metal" performance on GPUs is challenging, especially for HPC applications. Performance tools such as NVProf [2], Nsight Systems [3], Nsight Compute [4], HPCToolkit [5], and TAU [6] help pinpoint code inefficiencies. While these tools can identify common GPU inefficiencies such as poor data locality in memory hierarchies, data/control divergences, and various instruction execution stalls, they overlook an important inefficiency category: redundant computation and data movement involving the same values.

Prior work [7]–[9] has shown that values stored in memory have *temporal* and *spatial* redundancies in CPU codes. Temporal value redundancy indicates that the same (or approximately the same) value overwrites the same memory location. One can exploit temporal value redundancy by removing redundant computations and redundant data movement. Spatial value redundancy indicates that nearby memory locations share the same (or approximately the same) value. One can exploit spatial value redundancy via memoization [10], [11] (i.e., remember the value computed and reuse it if the same computation is performed on an adjacent location) and data compression [7] (i.e., compress repeated values with a sparse data structure). Value redundancy provides unique opportunities for code optimization and approximate computing.

Value redundancies exist in GPU code as well. There have been a surge of approaches [12]–[15] to exploit redundant values on GPUs. However, most of them rely on special hardware to identify and bypass redundant computation, which does not apply to existing HPC systems. Alternatively, one can use detailed simulation (e.g., GPGPUSim [16]) or compiler-based instrumentation (e.g., CUDAAdvisor [17], LLVM [18]) to study redundant values. However, simulation-based approaches are either time-consuming or require the availability of source code for recompilation, which makes them unsuitable for HPC applications that employ external close-sourced libraries, such as cuBLAS [19] and cuDNN [20].

To complement and improve upon existing approaches, we developed GVPROF. To the best of our knowledge, GVPROF is the *first* value profiler for NVIDIA GPUs to explore both temporal and spatial value redundancies in HPC applications. GVPROF leverages NVIDIA's Sanitizer API [21] to instrument memory access instructions in GPU binaries. GVPROF can identify memory accesses that manipulate redundant values and provide detailed information, including source code attribution with full calling contexts, data object details, and various derived metrics. We evaluate GVPROF on several benchmarks and applications. Most of the examples we studied use a single GPU. However, we evaluated Sandia's

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

```
1 ▶ power[ty][tx] = power[index];
2    for (int i : iteration) :
3      temp_t[ty][tx] = temp[ty][tx] +
4 ▶      step_div_Cap * (power[ty][tx] + ...
```

Listing 1: A GPU kernel in Rodinia [23] *hotspot* benchmark has high temporal value redundancy. The loop invariant value `power[ty][tx]` is repeatedly loaded from memory.

```
1 void dynproc_kernel(int *wall, ...) :
2   for (int i : iteration) :
3 ▶   result[tx] = shortest + wall[index];
```

Listing 2: A GPU kernel in Rodinia [23] *pathfinder* benchmark has high spatial value redundancy. The values in the array `wall` are largely redundant.

LAMMPS [22] molecular dynamics simulator across several nodes of the Summit supercomputer, showing that GVPROF can monitor multi-GPU executions.

In the rest of this section, we describe examples that exhibit value redundancies to motivate the need for GVPROF, highlight our contributions, and outline the organization of the rest of the paper.

A. Value Redundancies in GPU Kernels

We investigated two of the Rodinia [23] benchmarks to illustrate the potential for exploiting value redundancies in GPU kernels. Listing 1, part of a GPU kernel in *hotspot*, shows that each thread loads the same value from `power[ty][tx]` repeatedly in the loop, where `ty` and `tx` are GPU thread indices and `power[ty][tx]` is loop invariant. Temporal value redundancy occurs because the NVCC compiler fails to promote the value into a register even with the highest (-O3) optimization option. To eliminate the redundant loads, we use a scalar `p` to cache `power[index]` at line 1, forcing the value to be stored in a register, rather than shared memory, and calculate `step_div_Cap * p` before the loop. This simple optimization yields a 1.04× speedup.

Listing 2, part of the GPU kernel in *pathfinder*, shows high spatial redundancies on line 3 associated with the array `wall`. Further investigation showed that `wall` is a large array with millions of elements. However, the element values are always between one and ten assigned by its input-independent initialization function; no updates to `wall` occur in the kernel execution. We optimize *pathfinder*'s performance by demoting the element type in `wall` from a 32-bit integer to a 8-bit integer to improve the efficiency of coalesced memory transactions. This change yields a 1.14× speedup. We can employ a sparse data structure to compress redundant elements in `wall` for additional performance.

In this paper, we study other examples from the Rodinia benchmark suite, some of the CUDA SDK samples [24], and four HPC and machine learning codes. We find that their load and store operations produce up to 61% and 37% redundant values, respectively, which motivates the need for GVPROF to explore value-related redundancies in GPU codes.

It is worth noting that not all redundancies in an execution need be eliminated. In our experience, a high fraction of redundancy typically indicates performance inefficiencies and demands a further investigation for code optimization.

B. Paper Contributions

In this paper, we make the following contributions:

- We systematically study temporal and spatial value redundancies in GPU codes for both memory loads and stores, and propose various techniques for optimization.
- We describe the design of GVPROF, the first value profiler for NVIDIA GPUs (to the best of our knowledge). GVPROF can monitor production HPC and machine learning applications to identify value redundancies.
- We design GVPROF to provide useful performance insights, including derived redundancy metrics, full calling contexts, and a data-centric view for instructions and data objects (i.e., arrays) involved in value redundancies.
- We minimize the overhead of GVPROF to support measurement of large-scale executions by employing asynchronous analysis and hierarchical sampling.
- We employ GVPROF on ORNL's Summit supercomputer to measure a multi-node multi-GPU execution of LAMMPS. Guided by GVPROF, we improved the performance of several HPC and machine learning workloads running on a single V100 GPU and obtained nontrivial speedups.

C. Paper Organization

This rest of the paper is organized as follows. Section II reviews related work and distinguishes GVPROF. Section III describes the methodology of GVPROF. Section IV describes details of implementation. Section V quantifies GVPROF's accuracy and overhead. Section VI presents the analysis of four codes using GVPROF. Finally, Section VII presents our conclusions and outlines some plans for future work.

II. Related Work

There exist several approaches to exploring value redundancies in CPU codes. Hardware-based approaches [25]–[31] introduce new hardware components to detect and eliminate redundant computations and memory operations. Software-based approaches, such as RedSpy [8], LoadSpy [7], and Witch [9] perform value profiling by leveraging binary rewriters (e.g., Intel Pin [32]) or performance monitoring units and debug registers available only in CPU architectures. GVPROF uses instrumentation-based measurement like RedSpy and LoadSpy. However, there are significant differences in the approaches to identifying value redundancies because GVPROF: (1) handles sophisticated thread coordination across a massive number of SIMT threads on GPUs, (2) partitions the framework into GPU data collection and CPU on-the-fly analysis to avoid excessive memory overhead on GPUs, and (3) understands the data type (e.g., float or integer) and unit size (e.g., 32 or 64 bits) of values loaded or stored by memory

instructions using data flow analysis whereas CPU registers are typically associated with fixed types.

On GPUs, prior work mostly explores value redundancies via hardware approaches. Xiang et al. [13] design a hardware instruction reuse buffer to skip instructions with uniform values. Kim et al. [14] propose microarchitectural mechanisms to handle instructions composed of either uniform or affine value structures. Wang and Lin [12] introduce a method to identify affine compute instructions and decouple them from the regular SIMT instruction pipeline. Unlike such approaches that utilize specialized hardware components to identify the value redundancies, GVPROF does not require any hardware extensions. GVPROF can be deployed in existing GPU-based clusters or data centers.

The most closely-related work to GVPROF is the software profiling techniques used on GPU architectures. NVIDIA provides many tools [2]–[4] to measure performance metrics from the kernel level to the whole program level. Intel [33] provides GTPin to instrument GEN ISA binaries. Academic efforts [5], [6], [34]–[37] provide similar insights but scale to large HPC applications. Knobloch et al. [38] survey such tools. Aforementioned profilers employ performance counters or PC sampling [39] available in NVIDIA GPUs to provide useful performance insights. However, none of these tools can perform value profiling as GVPROF does.

While GVPROF leverages code from the open-source HPCToolkit performance tools [5] to attribute metrics to CPU calling contexts and presents GPU redundancy metrics using HPCToolkit's hpcviewer GUI, GVPROF is distinct from HPCToolkit in four ways. (1) GVPROF employs binary instrumentation for measurement of GPU programs; HPCToolkit does not. (2) GVPROF uses GPU queues to communicate GPU records to CPUs; in contrast, HPCToolkit uses NVIDIA's CUPTI API [40] to monitor and report GPU activities. (3) GVPROF provides a measurement substrate that correlates redundancy metrics with instruction addresses in GPU code; HPCToolkit uses CUPTI for all fine-grain attribution of metrics. (4) GVPROF employs kernel sampling and block sampling to reduce GPU measurement overhead; HPCToolkit uses CUPTI as a measurement black box.

Compiler-based profilers can provide additional insights. Yeh et al. [15] use an LLVM compiler pass to identify redundant instructions. CUDAAdvisor [17] is able to monitor every GPU memory access by adding instrumentation while compiling with LLVM. CUDA Flux [41] leverages LLVM to instrument GPU kernels for instruction characterization. Unlike these approaches, GVPROF is compiler-independent and can monitor external libraries that have only binary code available.

Other tools employ binary-level approaches. Welton and Miller [34], [42] inspect GPU APIs for performance issues in HPC applications. They instrument CPU binary code and check redundant data copies between CPUs and GPUs. Unlike GVPROF, they do not investigate redundancies inside GPU kernels. Moreover, their approaches require running a program multiple times, whereas GVPROF only needs to run the code once. GPU binary instrumentation frameworks, such as NVBit [43], SASSI [44], and Sanitizer API [21] all from NVIDIA, provide a rich set of APIs for instruction-level inspection. They can serve as the foundation of building a value profiler like GVPROF. GVPROF is built atop Sanitizer API and is the first GPU value profiler to the best of our knowledge.

III. METHODOLOGY

GVPROF works on heterogeneous systems with x86 or POWER host CPUs and NVIDIA GPUs. It investigates the values produced and used in GPU kernels and identifies the following four value redundancies.

Definition III.1 (Temporal Load Redundancy)**.** A memory load $L2$ is redundant iff it loads a value $v2$ from address A, and the last memory load $L1$ from A loads $v1$, where $v1 = v2$.

Definition III.2 (Temporal Store Redundancy)**.** A memory store $S2$ is redundant iff it stores a value $v2$ to address A, and the last memory store $S1$ stores $v1$ to A, where $v1 = v2$.

Definition III.3 (Spatial Load Redundancy)**.** A memory load $L2$ is redundant iff it loads a value $v2$ from address $A2$, and another memory load $L1$ loads $v1$ from address $A1$, where $v1 = v2$, and $A2$ and $A1$ are in the memory range of a data object allocated by a GPU memory allocation.

Definition III.4 (Spatial Store Redundancy)**.** A memory store $S2$ is redundant iff it stores a value $v2$ to address $A2$, and another memory store $S1$ stores $v1$ to address $A1$, where $v1 = v2$, and $A2$ and $A1$ are in the memory range of a data object allocated by a GPU memory allocation.

To identify temporal value redundancies, GVPROF reasons about the value generated by each memory instruction instance and compares the last value at the target memory location with the newly generated value. If the two values are the same, GVPROF records the program counters of two involved memory accesses $\langle PC_{old}, PC_{new} \rangle$ as a pair of redundancy and accumulates redundancy metrics with it. To identify spatial value redundancies, GVPROF intercepts data allocations, and associates memory accesses with data objects allocated in GPU memory. If a memory access produces the same value as a prior access to the same data object, a spatial redundancy occurs. GVPROF records allocation contexts of the data object and program counters of memory accesses $\langle C_{data}, PC_{access} \rangle$ and accumulates associated redundancy metrics.

To enable GVPROF for production HPC systems, we address several challenges. (1) We employ an efficient data collection mechanism to parallelize analysis and execution on GPUs with a massive number of threads. (2) We adopt a hierarchical sampling scheme to reduce measurement overhead to a reasonable level. (3) As GPU assembly does not provide access kind information for memory instructions (i.e., integer or float of different sizes), we devise a novel bidirectional slicing method to extract access kind information. We elaborate on the details of our approach in the next section.

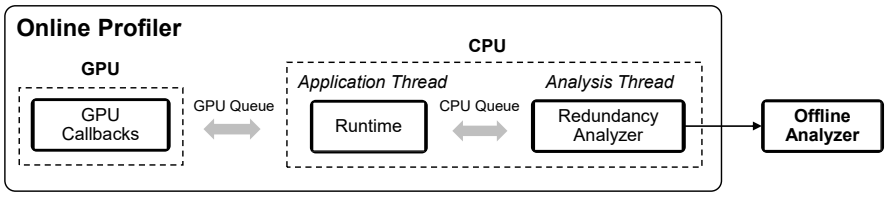

Fig. 1: Overview of GVPROF.

Utilization of GVPROF*:* GVPROF does not need any manual effort to produce the profiling report. Moreover, it runs a program once to identify all temporal and spatial value redundancies for memory loads and stores. GVPROF pinpoints performance inefficiencies and attributes them to full calling contexts. As a dynamic tool, GVPROF may observe different redundancy information in different parallel executions or runs with different inputs. One can use GVPROF to profile HPC applications with typical inputs that produce execution behaviors of interest. Programmers and compilers can use the aforementioned techniques, such as memoization and data compression to bypass the redundant computation. In Section VI, we show some examples of optimizing value redundancies with the information provided by GVPROF.

IV. IMPLEMENTATION

Figure 1 shows the major components of GVPROF: an online profiler that monitors the GPU code execution to collect data and perform online analysis, and an offline analyzer that aggregates profiles from multi-GPUs and visualizes them in a GUI. The online profiler further consists of three components. The GPU callbacks instrument analysis functions in GPU binaries to collect data during execution. The runtime system in the address space of the monitored programs on the host CPU manages the collected GPU data and gleans extra information for analysis, such as memory allocation and calling contexts. The redundancy analyzer spawns a helper thread to perform necessary analysis on the GPU data (i.e., identifying value redundancies in this paper) and passes analysis results to the runtime system. Queues exist between the three online components to interchange the data efficiently. The entire architecture of GVPROF is modular, and one can easily extend it for other analyses.

In the rest of this section, we elaborate on the design of each online component and the offline analyzer.

A. GPU Callbacks for Data Collection

GVPROF's GPU component consists of three parts: a set of callbacks to collect data via instrumentation, a queue that transfers data to CPUs for analysis, and a hierarchical sampler that reduces monitoring overhead.

a) Callbacks: GVPROF utilizes Sanitizer API [21] to instrument callback functions at each thread block entrance and exit, as well as after each memory access instruction. GVPROF uses the following approach for the memory access callback functions. GVPROF can distinguish load and store instructions. If a memory access instruction is a memory store, GVPROF directly obtains the value provided by Sanitizer API. Otherwise, GVPROF obtains the effective address accessed by the instruction and then reads the value stored in the memory by dereferencing the effective address according to the size and type (i.e., global, local, or shared) of the instruction. Typically, references to local and shared memory use 32-bit addresses, while references to global memory use 64-bit addresses. All information associated with this memory access, such as program counter (PC), value, effective address, and access size, forms a record, and GVPROF inserts the record into the GPU queue. Values collected at this stage are a binary sequence, with no type information. We refer to the values here as *raw* values.

b) GPU queue: For each GPU stream, GVPROF creates a queue shared between a GPU and a CPU to hold and transfer records collected with the callbacks. To distinguish from other queues, we call it the GPU queue. The size of the GPU queue is configurable by users. By default, we set the queue with 72 MB, which is a good tradeoff between runtime and memory overhead from our experiments.

Operations on the GPU queue are atomic. To minimize contention, GVPROF only allows the first active lane in a warp to request an empty slot in the queue. GVPROF then uses a warp shuffle operation to broadcast the slot location to every active lane in the warp. All active threads write their records to the slot concurrently. All lanes are synchronized when the writing is done. GVPROF uses the first active lane to inform the GPU queue that a slot is filled. Besides memory records, the GPU queue also holds records about thread block entrances and exits.

Once the GPU queue is full, the GPU kernel is paused while all records in the queue are transferred to the CPU. To achieve this, GVPROF intercepts kernel launch APIs. Upon each kernel launch, GVPROF locks the current stream to prevent other CPU threads from submitting kernels to the same stream. After a kernel is submitted to the GPU, GVPROF leverages a priority stream to check whether the GPU queue is full repeatedly. If full, GVPROF copies all records in the queue to the CPU for further analysis, clears the queue, and resumes kernel execution for more data collection. Meanwhile, GVPROF tracks the number of active threads at thread block exits. Once all threads are inactive (i.e., the current kernel is finished), GVPROF releases the stream lock.

c) Hierarchical Sampling: While GPU memory overhead of GVPROF is bounded by the size of the GPU queue, the runtime overhead is proportional to the number of instrumentation callbacks invoked. We observed that most HPC

applications employ iterative and data-parallel programming models; behaviors across different GPU kernel instances and across different thread blocks are similar. GVPROF employs a new hierarchical sampling mechanism to significantly reduce runtime overhead but minimize measurement accuracy loss, which we quantify in Section V. The hierarchical sampling consists of *kernel sampling* and *block sampling*.

Kernel Sampling: Kernel sampling monitors a subset of instances of the same GPU kernel. GVPROF uses the launching context to uniquely identify a GPU kernel. For example, if the same kernel is launched in two different calling contexts, GVPROF treats them as two different "kernels" instead of one. The calling context determination is done in the runtime system on the host CPU, which is described in the next section. GVPROF ensures that each kernel is sampled at least once. GVPROF exposes a command line interface for users to specify the sampling period.

Block Sampling: Even though GVPROF applies kernel sampling, a large number of thread blocks used in a kernel instance can still incur large overhead. Block sampling is used in complement to the kernel sampling. Block sampling randomly monitors thread blocks within a kernel instance. In the GPU callbacks, GVPROF checks whether a block should be monitored according to its block ID b. GVPROF makes the decision with two more parameters: A sampling threshold P defined by users and a random number r generated upon each kernel launch. GVPROF monitors a block b only if $(b + r)\%P == 0$.

B. Runtime System on Host CPU

The runtime system accepts data collected from GPUs via the GPU queue and passes it to the analysis thread via the CPU queue. Moreover, the runtime system determines the calling context for each kernel launch, obtains source code attribution (used for profile presentation), and maintains a list of active data objects (used for the spatial redundancy analysis).

a) Collecting Calling Contexts and Attributing to Source Code: The calling context at kernel launch is necessary for kernel sampling and redundancy attribution. GVPROF intercepts kernel launch APIs and determines the calling context for each kernel launch. GVPROF uses a combination of libunwind [45] and online binary analysis [46] to unwind the call stack to glean calling contexts and maintain them in a compact calling context tree [47]. Source code attribution is critical for understanding what optimizations might be applicable. However, Sanitizer API can only return the PC of the monitored memory access as an absolute address in memory, which cannot directly map to the source code. To address this issue, GVPROF captures GPU binaries and obtains the virtual accesses of all functions in the binaries via a Sanitizer API `sanitizerGetFunctionPcAndSize()`. GVPROF then uses function names to associate each function's virtual address with its offset in the GPU binaries. Once the runtime system of GVPROF interprets the access records collected via the GPU callbacks, it uses binary search to locate the function enclosing the program counter (PC) of each monitored instructions. GVPROF translates each PC to its relative offset in a GPU binary, which can then be mapped to the source code using line mapping information in the GPU binary.

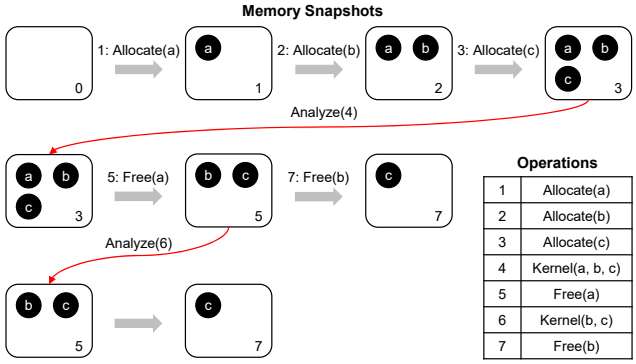

Fig. 2: Following the operations in sequence, the figure shows how memory snapshots are changed. After allocating a, b, c, the analysis thread is woken up and tries to analyze kernel 4. It first looks up data objects in snapshot 3, which is the closest to kernel 4. Then all the memory snapshots before 3 are erased. Likewise, after *7: Free(b)* the analysis analyzes kernel 5. After analysis, it erases snapshot 3 but keeps snapshots 5 and 7.

b) Tracking Data Objects on GPU: Monitoring data object allocation and free is necessary for analyzing spatial value redundancies. GVPROF uses Sanitizer API to intercept standard GPU memory allocation and free operations to track all active GPU data objects, including their allocated memory ranges and allocation contexts. For custom allocators, we could extend GVProf to leverage LLNL's GOTCHA library [48] to intercept and track allocations. The runtime system passes the list of all active data objects to the analysis thread via the CPU queue.[1] Because GVPROF's analysis thread is running asynchronously, its runtime system needs to keep the list up to date for all kernels.

GVPROF employs a novel approach – *memory snapshots* to track data objects across kernels efficiently. A memory snapshot contains all active GPU data objects. A new memory snapshot is created when a GPU memory allocation (e.g., `cudaMalloc`) or free (e.g., `cudaFree`) occurs; all the memory snapshots are stored in a map.[2] GVPROF assigns a unique ID *opid* for each snapshot and each kernel launch; *opid* always increments. The creation of memory snapshots follows two rules.

- Data allocation. When a GPU data object is allocated, GVPROF forks a new memory snapshot from the latest memory snapshot and inserts the new data object.
- Data reclamation. When a GPU data object is freed, GVPROF forks a new memory snapshot from the latest memory snapshot and removes the deleted data object.

Figure 2 shows a detailed example of how GVPROF maintains memory snapshots.

[1]We call it the CPU queue to distinguish from the GPU queue.

[2]These maps could be implemented more efficiently with a partially persistent data structure [49].

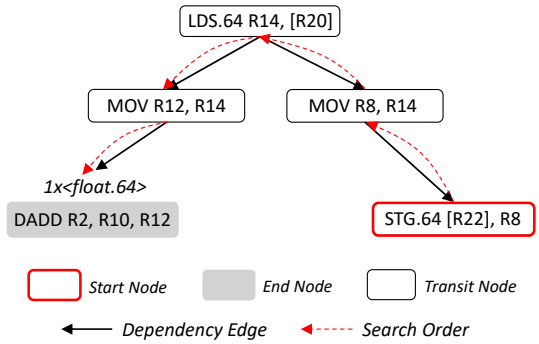

Fig. 3: An example of access kind analysis. The search starts at `STG.64` with *Backward* order until reaching `LDS`. The search proceeds in *Forward* order and terminates at `DADD`.

When the analysis thread investigates the current memory snapshot, it first obtains $opid_k$ of the current kernel launch and then to get the memory snapshot with $opid_m$, where $opid_m$ has the greatest value but smaller than $opid_k$ in the map. After the analysis, GVProf removes memory snapshots whose $opid$s are smaller than $opid_k$, except the one with $opid_m$. A global lock is used on the map to avoid data races between application threads and the analysis thread. Since GVProf shrinks the map whenever the analysis is done, snapshot lookup is fast even for complex code with tens of thousands of memory allocations.

Tracking memory allocations only works for data allocated in global memory. GVProf cannot identify individual data allocated in shared and local memory because there is no explicit allocation API to intercept. As a workaround, GVProf does not distinguish data objects allocated in the shared and local memories.

C. Analysis Thread

GVProf spawns a helper thread on the host CPU to perform necessary analysis concurrently with the program execution. The helper thread interchanges data with the runtime system via the CPU queue. The helper thread accepts the records collected on the GPU and sends back analysis results. The helper thread extracts the value of an instruction's raw value according to its access kind (e.g., integer/float, 32/64 bits) and performs value analysis.

a) Extracting Values According to Access Kind: The raw value obtained for each GPU memory access is a sequence of binary bits, with no type information. A memory instruction can interpret a raw value in different ways. For example, a `STS.64` instruction always stores 64-bit data to shared memory; the 64-bit data can be two 32-bit values or a single 64-bit value to shared memory, and the data type can be either float or integer. To distinguish values in different types, GVProf defines *access kind* to characterize memory accesses. An access kind is a triple associated with one memory access: *unit_size*, *vec_size*, and *data_type*, where *unit_size* indicates the size of each value, *vec_size* indicates the number of values accessed, and *data_type* specifies the value type, either float or integer. Once GVProf knows the access kind of each memory instruction, it can correctly interpret the bit sequence as values.

The high-level idea is that GVProf creates a dependency graph for each function based on its def-use chains and searches along the chains until access kind triples can be identified. While it is similar to the techniques used for type inference on CPU executables [50], our analysis is designed to track arithmetic data types (i.e., float vs. integer) with special handling for GPU instructions. Our analysis consists of three components:

- Access kind defining instruction. GVProf derives access kinds from a subset of GPU instructions. On NVIDIA GPUs, `CONVERT` instructions are used to convert between integer and float, or the same data type of different sizes; operands in `FLOAT` and `INTEGER` instructions are float and integer data types, respectively. GVProf can obtain the access kind for a value by directly decoding these instructions.
- Dependency graph creation. GVProf creates a dependency graph for each function based on its def-use chains and searches along the chains until access kind triples can be identified.
- Bidirectional search. GVProf employs both *Forward* and *Backward* orders to traverse along def-use chains using depth-first search. The search reverses the order when it encounters a memory load or store instruction because we do not track access kind through memory.

Figure 3 shows an example of the searching algorithm. As the `STG.64` instruction cannot tell its access kind, GVProf checks the def instruction—`MOV` instruction that defines `R8`. Since the `MOV` instruction cannot tell its access kind, GVProf keeps searching for the def instruction that defines `R14`, which is the `LDS` instruction. However, the `LDS` instruction is not labeled with its access kind either; it depends on how `R14` is used. GVProf then reverses the search direction to check the uses of `R14`. Finally, GVProf encounters the `DADD` instruction, which uses a single float 64 value in its registers. GVProf propagates the access kind back to both `LDS.64` and `STG.64` instructions.

GVProf uses program slicing in Dyninst [51] to create def-use chains for each GPU function and performs access kind analysis as a one-time procedure when a GPU binary is loaded. Our approach does not guarantee to recover all access kinds. For example, if the target register of a load instruction is immediately stored to memory and has no further use, GVProf is unable to identify the load instruction's unit size and data type. In such a case, GVProf can assign a default unit size and data type predefined by users. In Section V, we show that GVProf can achieve high access kind analysis accuracy without human intervention.

b) Value Redundancy Analysis: GVProf identifies both temporal and spatial value redundancies.

Temporal redundancy: GVProf creates two tables for each GPU kernel to detect temporal value redundancies, as shown in Figure 4. The *Last Seen Table* records each thread's last access value at an address. The *Redundant Pairs Table* records PC pairs involved in the redundancy. For each access

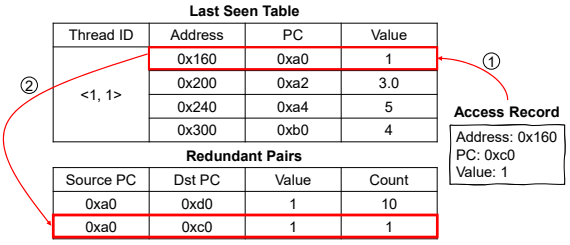

Fig. 4: Data structures for detecting temporal value redundancy. The Thread ID column contains a pair $\langle blockId, threadId \rangle$ that uniquely identifies each GPU thread.

record, GVPROF first checks whether its effective address is in the *Last Seen Table*. If not, GVPROF inserts a record into the table. Otherwise, GVPROF compares the current value with the last value and reports redundancies in the *Redundant Pairs Table* if the two values are the same. The address in the *Last Seen Table* is then updated to the current address.

One challenge arises due to the large number of threads used on GPUs. For instance, the total number of threads in NVIDIA Volta V100 is up to 2^{41}; the *Last Seen Table* with all thread records can easily overflow GPU memory. We minimize the size of the *Last Seen Table* by leveraging the fact that the number of active threads on a GPU is limited to the number of stream multi-processors (e.g., 80 in Volta V100). Thus, upon a `BLOCK_EXIT` record in the GPU data, GVPROF removes records of all inactive threads from the *Last Seem Table* as they are never used again.

Spatial redundancy: Similarly, GVPROF identifies and accumulates spatial value redundancies. Instead of checking values loaded from or stored to the same effective address, GVPROF checks values within the same data object. GVPROF leverages memory snapshots captured by the runtime system to create a map from data allocation contexts to allocated object memory ranges. GVPROF then attributes memory accesses to the data objects that enclose the effective addresses of the memory accesses. GVPROF reports the spatial redundancies to data objects as well as the related memory accesses.

D. Offline Analysis

GVPROF's offline component aggregates profiles from different GPUs, calculates redundancy metrics, and visualizes results in a GUI.

a) Profile aggregation: Each CPU thread that invokes GPU APIs produces a profile file, regardless of the number of streams used, GVPROF coalesces profiles across CPU threads and processes for the whole execution. Such a compact view of profiles can scale the analysis of program executions to a multi-node multi-GPU cluster. The coalescing procedure is applied for the kernels with the same calling context, following two rules: (1) temporal redundancy pairs $\langle PC_1, PC_2 \rangle$ and $\langle PC_3, PC_4 \rangle$ are merged if $PC_1 = PC_3$ and $PC_2 = PC_4$; and (2) spatial redundancy pairs $\langle C_{data}, PC \rangle$ and $\langle C'_{data}, PC' \rangle$ are merged if $C_{data} = C'_{data}$ and $PC = PC'$. The redundancy metrics are also accumulated during coalescing. The profile coalescing overhead grows linearly with the number of CPU threads and processes used by the monitored program. GVPROF can leverage the reduction tree technique [52] to parallelize the merging process. GVPROF typically requires $<$10 seconds to produce the aggregate profiles for all of our case studies.

b) Metric derivation: GVPROF derives metrics for both temporal and spatial value redundancies.

$$TR_{k,c} = \frac{TC_{k,c}}{N_{k,c}} \tag{1}$$

Equation 1 calculates the temporal redundancy ratio of a given context c (e.g., an instruction, a function, or the entire kernel) for a kernel k as the faction of the number of temporally redundant accesses $TC_{k,c}$ over the number of memory accesses $N_{k,c}$. One can apply this equation to either memory loads or stores.

$$SR_{k,o} = \frac{SC_{k,o}}{N_{k,o}} \tag{2}$$

Equation 2 computes the spatial redundancy ratio for any data object o in a kernel k as the fraction of the occurrences of the value redundant accesses to object o over the total number of accesses to o. Likewise, this equation also applies to memory loads and stores.

c) Result presentation: GVPROF provides two views for users. The *profile view* employs a Java-based GUI to show redundancy metrics associated with the program source code for different calling contexts. It is particularly useful to pinpoint redundancies in specific contexts. Figure 7 shows a snapshot of the GUI.

The *memory view* is a data-centric view that presents values accessed at top redundant PCs for different data objects. One can use the *memory view* to investigate the value distribution of a specific PC or an entire data object.

E. Portability

The implementation of GVProf is modular. By changing its instrumentation engine from NVIDIA's Sanitizer API to Intel's GTPin [33], we could adapt GVProf to work on Intel GPUs. At present, AMD does not provide an instrumentation tool for its GPU binaries, so GVProf does not support AMD GPUs.

V. Evaluation

We evaluated GVPROF on the Summit supercomputer at Oak Ridge National Laboratory. Each Summit node has 2 POWER9 CPU processors and 6 NVIDIA Volta GPUs with 96GB GPU memory in total and with the following system software: Linux 4.14.0, NVIDIA CUDA Toolkit 10.1.243, NVIDIA Driver 418.116, and GCC 6.4.0. We evaluated GVPROF on several Rodinia benchmarks [23], several CUDA SDK samples [24], and the HPC applications described below.

- Darknet [54] is an open-source neural network framework. We studied Darkent on the `yolov3-tiny` model [56] with a `cuBLAS` backend.

Programs	Redundancy ratios				Sampling errors	Overhead		Access kind coverage
	Temporal load	Temporal store	Spatial load	Spatial store		w/o sampling	w/ sampling	
backprop	15.8%	0.0%	21.0%	5.2%	0.0%	66x	3.7x	100.0%
bfs	11.0%	0.0%	52.6%	15.2%	0.1%	31x	1.9x	100.0%
hotspot	3.6%	0.0%	12.4%	5.3%	0.2%	12.6x	2.6x	100.0%
sradv1	2.4%	0.0%	8.2%	3.3%	0.2%	8x	4.4x	100.0%
dct8x8	0.0%	0.0%	1.2%	0.6%	0.0%	443.8x	7.5x	99.5%
dwt2d	17.3±0.1%	7.9±0.1%	33.0%	24.9%	1.7%	5.2x	4.2x	100.0%
dxtc	32.3%	0.0%	1.1%	0.2%	0.1%	781.6x	13.1x	98.0%
reduction	45.9±2.6%	31.2±1.0%	60.8%	37.0%	4.8%	7.7x	7.7x	100.0%
pathfinder	5.1%	2.5%	1.9%	0.2%	0.0%	24.8x	1.7x	100.0%
histogram	0.0%	0.0%	12.2%	12.7%	0.0%	1188.2x	14.7x	100.0%
euler3d	0.0%	0.0%	28.8%	7.5%	0.1%	31.6x	1.8x	96.4%
lammps	7.9±0.1%	2.3±0.1%	15.7±0.2%	5.8±0.2%	–	–	44.7x	–
darknet	5.4%	1.2%	9.0%	2.2%	–	–	29.4x	–
quicksilver	27.3%	18.2%	43.4%	23.4%	–	–	769.2x	–
laghos	1.9%	5.2%	18.0%	11.8%	–	–	28.1x	–
average	11.7%	4.6%	21.3%	10.4%	0.7%	236.4x	62.3x	99.4%
median	5.4%	0.0%	15.7%	5.8%	0.1%	31x	7.5x	100.0%

TABLE I: Evaluation of redundancy ratios, sampling errors, profiling overhead, and access kind coverage for benchmarks and applications. The redundancy ratios are averaged across five runs. We do not report standard deviations that are exactly zero. We do not report the overhead and redundancy ratios of applications without sampling because GVPROF does not halt on these applications when sampling is disabled. Moreover, we only report access kind coverage for benchmarks because obtaining ground truth manually for real applications is tedious.

Redundancy Type	Redundancy Metrics	Problems	Optimization Methods	Programs	Inputs	Execution time	Speedup
Spatial	100% Single value	Read/write constant values	Constant replacement	LAMMPS [22]	bench/lj.in	2.71s	1.47x
					/src/USER-INTEL/TEST/in.intel.lj	11.20s	1.37x
				Laghos [53]	square01_quad.mesh	17.22ms	1.00x
					cube01_hex.mesh	82.17ms	0.99x
				sradv1 [23]	502 458	21.57us	1.46x
					1004 916	75.48us	1.26x
Spatial	High ratio ($\geq 50\%$) single value	Waste compute resources	Conditional load/store	Darknet [54]	yolov3-tiny.cfg	28.79us	1.60x
				backprop [23]	65536	29.57us	1.93x
					131072	54.90us	2.42x
Spatial	Frequent occurrences of a few distinct values	Waste memory bandwidth	Compression	pathfinder [23]	100000 100 20	104.46us	1.14x
					100000 200 20	202.90us	1.14x
Spatial	Close values	Waste compute/memory resources	Approximate compute	hotspot [23]	data/temp_512	16.84us	1.29x
Temporal	Memory access in a loop	Failed register promotion	Scalar replacement	dxtc [23]	data/lena_std.ppm	1.66ms	1.02x
				hotspot [23]	data/temp_512	16.84us	1.04x
Temporal	Memory access in a device function	Missing function inlining	Inline substitution	Quicksilver [55]	default input	1.16s	1.35x
					CORAL-p1.inp	42.27s	1.68x

TABLE II: Performance insights into value redundancies produced by GVPROF. We report kernel time speedup for all cases except for LAMMPS, which uses whole program speedups. We evaluated program speedups with different inputs if available.

- Quicksilver [55] is a DOE proxy application for solving a dynamic Monte Carlo particle transport problem. Quicksilver has a single big GPU kernel, in which a number of device functions are used. We studied Quicksilver with its default input released with the software.
- LAMMPS [22] is a molecular dynamics code for large-scale materials modeling. LAMMPS uses Kokkos [57] to provide an abstraction for different accelerators. We profiled LAMMPS using its Lennard-Jones liquid benchmark released together with the software package.
- Laghos [53] is a DOE mini-app that solves the time-dependent Euler equations of compressible gas dynamics. We ran Laghos with its `square01_quad.mesh` input.

In the remaining section, we describe insights based on the analysis results reported by GVPROF, analysis accuracy, and overhead on a single GPU execution. One exception is the scalability evaluation of GVPROF in Section V-C, for which we used up to 64 GPUs across 11 Summit nodes.

A. Value Redundancy Analysis

Table I quantifies temporal and spatial value redundancy ratios for both memory loads and stores throughout the entire program execution. We can see that value redundancies are common in GPU codes, both benchmarks and applications. Table II summarizes the insights we obtained using GVPROF to analyze programs with high value redundancy ratios. Note that the insights are based on the statistics associated with specific contexts in the *profile view* and the *memory view*, rather than the overall metrics presented in Table I. In the rest of this section, we overview these insights to highlight the root causes of value redundancies. We analyze the four applications in detail in Section VI.

a) Insights for spatial value redundancy: GVPROF quantifies spatial value redundancies, which suggests the following four optimizations described in Table II.

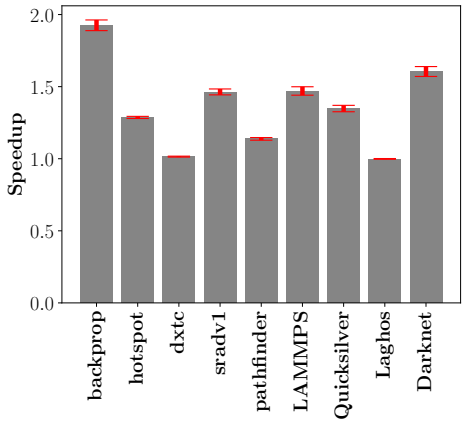

Fig. 5: Speedups averaged across ten runs with error bars.

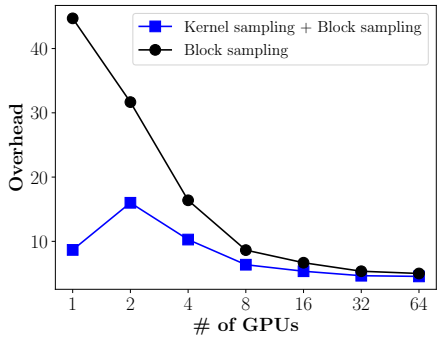

Fig. 6: Scalability of GVPROF's overhead across multiple GPUs and multiple Summit nodes.

- When the spatial value redundancy ratio of an array is 100%, indicating accesses to this array's elements have the same value. One can hoist this value to a scalar. LAMMPS, Laghos, and *sradv1* fall into this category.
- When the spatial value redundancy ratio of an array is high ($> 50\%$) with a single value, one can memoize the results and skip computations on the same value. Darknet, *dwt2d*, and *backprop* are in this category.
- When the spatial value redundancy ratio of an array is high due to the frequent occurrences of a few distinct values, one can compress either the element type (e.g., from 32-bit integer to 8-bit integer) or the array itself with a sparse format. *pathfinder* is in this category.
- When the spatial value redundancy ratio of an array is high due to approximate values, if approximate computing is allowed, one can bypass the computation of similar values. *hotspot* belongs to this category.

b) Insights for temporal value redundancy: There are two possible optimizations for temporal value redundancies. First, if the redundancies occur in a loop, a compiler may fail to promote loop-invariant values into registers due to unknown loop trip count, missing access index constraints, or register limitations. One can cache the values explicitly to avoid repeated memory loads and stores. Laghos and *dxtc* are in this category. Second, if the redundancies occur at a function's prologue or epilogue, it means invoking the function involved storing or loading redundant values on local memory to reserve registers. One can inline the function if possible, to avoid redundant local memory operations. Quicksilver is in this category.

c) Optimization workflow: One can start with the profile view to check GPU code with high redundancies. Next, one can leverage the memory view to track down redundant values and instruction PCs at these hotspots. Finally, one can apply optimizations suggested in Table II for some common redundancy patterns.

d) Optimization speedup analysis: Figure 5 presents the averaged speedups of codes we studied. For some codes, the performance improvements from eliminating value redundancies were small (e.g., 1.02× speedup of dxtc); for other codes, eliminating value redundancies had a much larger effect (e.g., 2.42× speedup of backprop). In general, eliminating value redundancies offers an opportunity for improving performance. However, the speedup from eliminating redundancies in an execution depends on the program and in some cases its input data. In practice, applications can spend much time on GPUs. For some programs, we see large speedups. For example, our optimizations reduce Quicksilver's kernel time from 42.27s to 25.11s.

B. Analysis Accuracy

a) Accuracy for access kind analysis: To quantify the accuracy of GVPROF's access kind inference, we define *access kind coverage* as the ratio of the number of correctly identified memory instructions over the total number of memory access instructions in all GPU kernels for a given program; 100% coverage means our access kind inference is always correct. We obtained the ground truth of access kind for an instruction by manually checking every memory instruction in a GPU binary. It is worth noting that we evaluated coverage only for benchmarks because obtaining the ground truth for large kernels in applications is tedious and error-prone. Table I shows that our bidirectional slicing method for access kind inference yields high access kind coverage. Accuracy losses arise mainly due to two causes. On the one hand, `nvdisasm` as the foundation of GVPROF fails to decode machine instructions in GPU binaries (e.g., *euler3d*). On the other hand, GVPROF's access kind inference fails on some load instructions whose values are immediately stored to memory with no further use (e.g., *dxtc* and *dct8x8*).

b) Accuracy of block sampling: Table I shows the block sampling errors with the default sampling period 50, quantified as the average across four redundancy ratios with the comparison to the ratios with no sampling enabled. GVPROF chooses this default sampling period as a good trade-off between accuracy and overhead from multiple experiments with different sampling periods, including 2, 3, 4, 8, 10, 16, 32, 50, 64, and 100. From the table, we can see that GVPROF yields high sampling accuracy. Benchmark *reduction* has the highest sampling error (4.8%) because it has imbalanced work distribution across blocks.

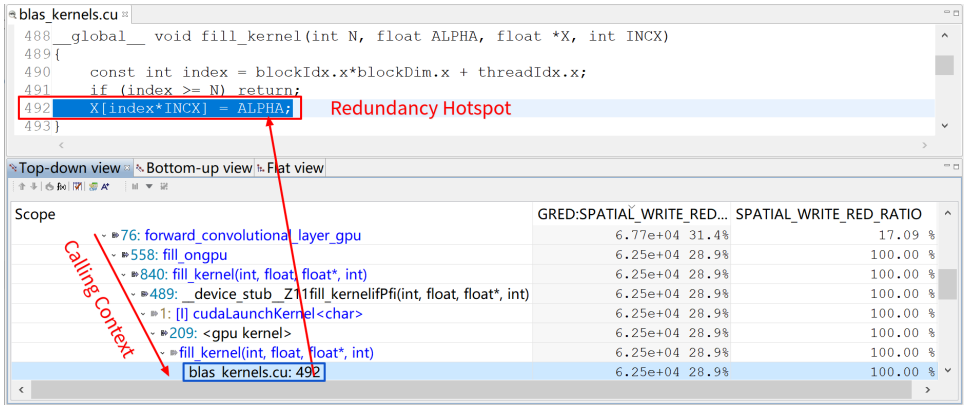

Fig. 7: The GUI of GVProf shows the calling context of the `fill` kernel invocation with high spatial store value redundancy. The top pane shows the source code; the bottom left pane shows the full calling contexts; and the bottom right pane displays the redundancy count (`SPATIAL_WRITE_RED`) and the redundancy ratio (`SPATIAL_WRITE_RED_RATIO`) of each context (e.g., source lines and kernels).

c) False positives: GVProf has no false positives; that is, all the value redundancies identified are real, measured with fine-grained instrumentation of all memory loads and stores in GPU kernels. However, not all value redundancies can or should be eliminated. Sometimes, redundancies may be input-dependent. Sometimes removing redundancies may be more costly than leaving them alone.

C. Measurement Overhead

Table I shows that without sampling, GVProf can incur 1000× overhead and even more in HPC codes. This high overhead is due to the large number of GPU access records generated by the massive number of memory accesses for monitoring and the use of a large number of blocks for computation. Fortunately, GVProf's block sampling mechanism significantly reduces the overhead to a reasonable level, which is typically 7.5× for benchmarks. HPC applications suffer from higher overhead due to their complex kernel designs, but it is worth noting that without sampling, GVProf runs forever to collect profiles. To further reduce the overhead for HPC applications that consist of tens of thousands of kernel instances, GVProf can employ kernel sampling.

Typically, the overhead of GVProf does not change program behaviors. While a program that performs event-driven processing as messages arrive from its communication partners might have its behavior affected by the order of message arrival, in most cases, we do not believe that the characteristics of GPU computations performed by a program would often be affected by such effects. We show standard deviations from five runs of our redundancy measurements in Table I. The small standard deviations of our redundancy measurements indicate the small measurement variance across different runs.

D. Multi-GPU

Figure 6 shows the scalability of GVProf. We evaluated GVProf in a strong scaling experiment with LAMMPS running up to 64 GPUs across 11 Summit nodes. We ran LAMMPS with MPI, with each MPI process using a GPU. As shown in the figure, GVProf's overhead drops with more GPUs when using block sampling because more processes are involved in processing the measurement data from the fixed amount of kernel computation (due to strong scaling), which accelerates online analysis. Kernel sampling can further reduce the overhead and converge with block sampling when using 64 GPUs. Notably, kernel sampling's overhead increases when moving from one GPU to multiple GPUs because every process must sample every kernel calling context at least once. As the number of GPUs increases, the workload of each process is reduced due to strong scaling.

VI. Case Studies

With the information from GVProf, we further optimized HPC and machine learning applications, such as Darknet [54], Quicksilver [55], LAMMPS [22], and Laghos [53]. Like Section V, we profiled and optimized these applications on Summit with a single GPU. All applications were compiled with `-O3` optimization.

A. Darknet

Darknet implements convolution using the lowering method [58], which transforms each image in a mini-batch into an input matrix, utilizes cuBLAS to perform matrix multiplication with a filter matrix. As shown in Figure 7, GVProf reports 28.9% spatially redundant stores in a `fill` kernel with 100% redundancy ratio. The `fill` kernel is applied before matrix multiplication $A \times B$ to set the output matrix C to zeros. The computation follows the format: $C \leftarrow A \times B + C$, which introduces many spatial redundancies on loading C. Thus, we removed the `fill` kernel invocation and computed $C \leftarrow A \times B$ to avoid redundant loads and stores. This optimization yields a 1.02× speedup for the entire program.

GVProf further pinpoints that 99% spatially redundant loads are in `cuBLAS`'s matrix multiplication kernel. To investigate the root cause, we obtained the redundancy ratio for each convolution layer individually and found that the first convolution layer incurs the most spatial redundancy.

```
1  bool CollisionEvent(..) :
2    for (int isoIndex = 0; isoIndex < numIsos; isoIndex++) :
3      for (int reactIndex = 0; reactIndex < numReacts;
4        reactIndex++) :
5        currentCrossSection -= macroscopicCrossSection(
6          isoIndex, reactIndex, ...);
7
8  double macroscopicCrossSection(
9    int isoIndex, int reactIndex, ...) :
10   microscopicCrossSection = monteCarlo->_nuclearData->
11 ▶   getReactionCrossSection(reactIndex, isoIndex, ...);
12
13 double NuclearData::getReactionCrossSection(
14   int reactIndex, int isoIndex, int group) :
15 ▶ qs_assert(isoIndex < _isotopes.size());
16 ▶ qs_assert(reactIndex < _isotopes[isoIndex]._species[0].
17     _reactions.size());
18   return _isotopes[isoIndex]._species[0].
19 ▶   _reactions[reactIndex].getCrossSection(group);
```

Listing 3: Temporal value redundancies in Quicksilver.

GVPROF shows that 50% values loaded from shared memory in this kernel are zeros. This is because `cuBLAS` uses a general tiling approach for tall-and-thin matrices in the first convolution layer of `yolov3-tiny`, which causes many zero values in shared memory. For optimization, we employed a fast implementation for tall-and-thin matrix multiplication [59] to better tile matrices. This optimization reduced spatially redundant loads from 41% to 6%, yielding a 1.60× speedup for the matrix multiplication kernel.

B. Quicksilver

GVPROF identifies both temporal and spatial value redundancies in Quicksilver.

a) Optimizing temporal value redundancies: Listing 3 shows the code that GVPROF highlights for two temporal redundancies. The first one is the `qs_assert` on line 19 and 20, which account for 20.9% of total temporally redundant loads. Further investigation shows that the function `getReactionCrossSection` enclosing the two assertions is called in a loop nest on line 5. Because `isotope_index` and `_isotopes` are both loop invariant, we can either hoist the assertions out of the loop or remove the assertions in the released version. We removed the two assertions and obtained a 1.10× speedup for the kernel.

GVPROF also pinpoints the epilogue of the function `getReactionCrossSection` and its caller `macroscopicCrossSection` (both shown in Listing 3), accounting for 30.2% of total temporally redundant loads. This is because these two functions are called in a loop nest, introducing redundant local memory store and load operations to spill and restore unchanged values, such as loop trip count (e.g., `numIsos`), to make registers available for the callee. We inlined these two functions into their caller to avoid redundant local memory accesses and obtained an extra 1.10× speedup to the kernel.

b) Optimizing spatial value redundancies: GVPROF identifies 61.5% spatial store redundancies in the class constructor for `MC_Distance_To_Facet` (not shown in the paper), in which values are initialized to zeros.

Hotspot 1

794: loop at create_atoms.cpp	21.4%
795: loop at create_atoms.cpp	21.4%
796: loop at create_atoms.cpp	21.4%
797: loop at create_atoms.cpp	21.4%
831: LAMMPS_NS::AtomVecAtomicKokkos::create_atom(int, double*)	21.4%
795: LAMMPS_NS::AtomVecAtomicKokkos::grow(int)	21.4%
75: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	6.0%
232: Kokkos::DualView(…)	6.0%
679: Kokkos::resize(…)	6.0%
74: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	6.0%
73: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	4.1%
69: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	1.3%
70: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	1.3%
71: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	1.3%
68: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	1.3%

Hotspot 2

398: loop at atom_vec_atomic_kokkos.cpp:	30.9%
398: LAMMPS_NS::AtomVecAtomicKokkos::grow(int)	30.9%
75: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	9.6%
74: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	7.4%
73: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	6.7%
69: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	2.1%
70: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	2.1%
71: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	1.8%
68: LAMMPS_NS::MemoryKokkos::grow_kokkos(…)	1.2%

Fig. 8: Two hot spatially redundant stores in LAMMPS due to function `Kokkos::resize`. `Kokkos::resize` is invoked from every call to `LAMMPS_NS::MemoryKokkos::grow_kokkos`. Note: We collapse long function names generated by Kokkos template wrappers.

Spatial store redundancy is high because an array of `MC_Distance_To_Facet` is constructed repeatedly in a loop (*loop1*). There are three data members in `MC_Distance_To_Facet`: `distance`, `facet`, and `subfacet`. However, in *loop1*, `facet` and `subfacet` are never used so we can compress the `MC_Distance_To_Facet` array to a `double` array for `distance` only to reduce spatial store redundancy.

After this optimization, GVPROF identifies the `distance` array accounting for 10.5% spatially redundant loads in another loop (*loop2*). GVPROF reports that a large double value assigned in *loop1* dominates values in the array. Further investigation shows that Quicksilver uses the `distance` array to save the nearest distance to each facet of a cell in *loop1*, and finds the minimum value of the array in *loop2* and discards the `distance` array. We optimized the code by fusing the two loops and compressing the `distance` array to a single variable to hold the minimum value. The two spatial redundancy optimizations yield a 1.12× additional speedup.

c) Optimization summary: To summarize, our optimization guided by GVPROF on both temporal and spatial value redundancies yields a 1.35× speedup to Quicksilver's kernel.

C. LAMMPS

GVPROF reports 52.3% spatial value redundant stores are in `Kokkos::resize` kernel under two hot calling contexts, as shown in Figure 8. `Kokkos::resize` is called repeatedly in loop nests to increase the size of multi-dimensional arrays. Resizing an array requires allocating a new piece of memory and initializing it to zero, thus, resulting in spatial store redundancy. Since arrays grow dynamically, LAMMPS defines an array growth factor that indicates how much additional space will be added to a dynamic array each time it resizes. To avoid frequent allocation and initialization, we increased

GPUs	1	2	4	8	16	32	64
lj.in	1.47×	1.31×	1.30×	1.23×	1.04×	1.05×	1.05×
in.intel.lj	1.37×	1.39×	1.37×	1.33×	1.20×	1.19×	1.08×

TABLE III: Speedups of LAMMPS on varying numbers of GPUs.

the default factor at the two hotspots from 0.01 to 0.1 and 0.8 respectively.

Table III shows the speedups of this optimization for two different inputs. With one GPU, we obtained a $1.47\times$ speedup and a $1.37\times$ speedup for the entire program. When the workload is partitioned over more GPUs, speedup falls off but always remains greater than $1\times$. As the workload is distributed across more processes, the size of the arrays on each process is reduced so that the benefit of using a large array growth factor decreases. It is also worth noting that the speedups for executions with the larger input (in.intel.lj) fall off more slowly than those for the smaller input (lj.in) as the workload is partitioned across a larger number of GPUs.

D. Laghos

GVPROF identifies a temporal load redundancy due to memory aliases in `cuKernelDot` kernel that performs a dot product of vectors `x` and `y`. In some context, `x` and `y` point to the same array, so the same values are loaded twice. As an optimization, we can check whether `x` and `y` point to the same array. If yes, we only load the value once for the computation. Although the optimization did not result in a significant speedup for the input we studied, other inputs that cause manipulation of larger arrays may benefit from the redundancy elimination.

VII. Conclusions and Future Work

This paper introduces GVPROF, the first value profiler on GPUs, to detect value-related redundant computation in HPC applications. GVPROF identifies temporal and spatial value redundancies for both memory loads and stores and provides detailed information that is useful to guide optimization, including calling contexts, data objects, and source code attribution. To monitor production HPC applications, GVPROF employs various optimizations to reduce its overhead, especially for multi-GPU and/or multi-node executions. With insights provided by GVPROF, we were able to optimize several HPC and machine learning code bases, most notably improving the whole-program performance of LAMMPS. GVPROF [60] is open-source and has been employed on the Summit supercomputer.

We envision two enhancements to GVPROF. First, we will explore value redundancies across GPU kernels. Second, we will parallelize the analysis in GVPROF's helper thread to reduce tool's overhead.

VIII. Acknowledgement

We would like to thank Aurelien Chartier at NVIDIA for fixing bugs and providing new functions in Sanitizer API. We also thank the reviewers for proofreading the paper and providing helpful comments.

This research was supported by Ken Kennedy Institute ExxonMobil Graduate Fellowship and by the Exascale Computing Project (17-SC-20-SC) - a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work is partially supported by a Google gift and a Google Faculty Research Award, as well as the Thomas F. and Kate Miller Jeffress Memorial Trust, Bank of America, Trustee and any specified Program donor (if applicable). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] (2019) Top 500 List. [Accessed March 26, 2020]. [Online]. Available: https://www.top500.org

[2] (2019) The user manual for nvidia profiling tools for optimizing performance of cuda applications. [Accessed March 26, 2020]. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide

[3] NVIDIA Corporation. (2020) Nvidia nsight systems. [Accessed March 26, 2020]. [Online]. Available: https://developer.nvidia.com/nsight-systems

[4] ——. (2020) NVIDIA Nsight Compute. [Accessed March 26, 2020]. [Online]. Available: https://developer.nvidia.com/nsight-compute

[5] K. Zhou, M. Krentel, and J. Mellor-Crummey, "A tool for top-down performance analysis of gpu-accelerated applications," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 415–416.

[6] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[7] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant loads: A software inefficiency indicator," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 982–993.

[8] S. Wen, M. Chabbi, and X. Liu, "Redspy: Exploring value locality in software," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 47–61.

[9] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 332–347.

[10] J. Mostow and D. Cohen, "Automating program speedup by deciding what to cache," in *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'85. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1985, p. 165–172.

[11] L. Della Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 607–622. [Online]. Available: https://doi.org/10.1145/2814270.2814290

[12] K. Wang and C. Lin, "Decoupled affine computation for simt gpus," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 295–306, 2017.

[13] 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*, 2013, pp. 433–442.

[14] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural mechanisms to exploit value structure in simt architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 130–141.

[15] 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*, 2020, pp. 1327–1340.

[16] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[17] D. Shen, S. L. Song, A. Li, and X. Liu, "Cudaadvisor: Llvm-based runtime profiling for modern gpus," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 214–227.

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

[19] NVIDIA Corporation. (2020) NVIDIA cuBLAS. [Accessed March 26, 2020]. [Online]. Available: https://developer.nvidia.com/cublas

[20] ——. (2020) NVIDIA cuDNN. [Accessed March 26, 2020]. [Online]. Available: https://developer.nvidia.com/cudnn

[21] ——. (2020) NVIDIA Compute Sanitizer. [Accessed March 26, 2020]. [Online]. Available: https://docs.nvidia.com/cuda/compute-sanitizer/index.html

[22] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," Sandia National Labs., Albuquerque, NM (United States), Tech. Rep., 1993.

[23] 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)*. Ieee, 2009, pp. 44–54.

[24] NVIDIA Corporation. (2020) NVIDIA CUDA Samples. [Accessed March 26, 2020]. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[25] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII. New York, NY, USA: ACM, 1996, pp. 138–147. [Online]. Available: http://doi.acm.org/10.1145/237090.237173

[26] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. Washington, DC, USA: IEEE Computer Society, 1996, pp. 226–237. [Online]. Available: http://dl.acm.org/citation.cfm?id=243846.243889

[27] K. M. Lepak and M. H. Lipasti, "On the Value Locality of Store Instructions," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, Jun 2000, pp. 182–191.

[28] ——, "Silent Stores for Free," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000, pp. 22–31.

[29] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 127–139.

[30] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelganger: A Cache for Approximate Computing," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 50–61.

[31] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-free Value Prediction with Safe-to-approximate Loads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 62, 2016.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[33] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, "Fast computational gpu design with gt-pin," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 76–86.

[34] B. Welton and B. Miller, "Exposing hidden performance opportunities in high performance gpu applications," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 301–310.

[35] D. Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knapfer, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, C. Rassel, P. Saviankou, D. Schmidl, S. Shende, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Springer Berlin Heidelberg, 2012, pp. 85–97.

[36] H. Zhang, "Data-centric performance measurement and mapping for highly parallel programming models," Ph.D. dissertation, University of Maryland—College Park, 2018.

[37] H. Zhang and J. Hollingsworth, "Understanding the performance of GPGPU applications from a data-centric view," in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, Nov 2019, pp. 1–8.

[38] M. Knobloch and B. Mohr, "Tools for gpu computing–debugging and performance analysis of heterogenous hpc applications," *Supercomputing Frontiers and Innovations*, vol. 7, no. 1, pp. 91–111, 2020.

[39] NVIDIA Corporation, "NVIDIA PC sampling view," http://docs.nvidia.com/cuda/profiler-users-guide/index.html#pc-sampling.

[40] ——. (2020) NVIDIA CUPTI. [Accessed May 26, 2020]. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/index.html

[41] L. Braun and H. Fröning, "Cuda flux: A lightweight instruction profiler for cuda applications," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) Workshop, collocated with International Conference for High Performance Computing, Networking, Storage and Analysis (SC2019)*, 2019.

[42] B. Welton and B. P. Miller, "Diogenes: Looking for an honest cpu/gpu performance measurement tool," 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/10.1145/3295500.3356213

[43] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 372–383.

[44] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.

[45] D. Mosberger-Tang, "`libunwind`," http://www.nongnu.org/libunwind.

[46] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan, "Binary analysis for measurement and attribution of program performance," in *Proc. of the 2009 ACM PLDI*. NY, NY, USA: ACM, 2009, pp. 441–452.

[47] M. Arnold and P. F. Sweeney, "Approximating the calling context tree via sampling," IBM, Tech. Rep. 21789, 1999.

[48] (2020) Gotcha. [Accessed March 26, 2020]. [Online]. Available: https://github.com/LLNL/GOTCHA

[49] H. Kaplan, "Persistent data structures," in *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2018, pp. 511–527.

[50] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, May 2016. [Online]. Available: https://doi.org/10.1145/2896499

[51] U. of Wisconsin-Madison. Dyninst. [Accessed January 26, 2020]. [Online]. Available: https://github.com/dyninst/dyninst

[52] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *SC 2010 International Conference for High-Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, November 2010.

[53] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012.

[54] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[55] (2020) Quicksilver. [Accessed March 26, 2020]. [Online]. Available: https://github.com/LLNL/Quicksilver

[56] (2020) YOLO: Real-Time Object Detection. [Accessed March 26, 2020]. [Online]. Available: https://pjreddie.com/darknet/yolo/

[57] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[58] Y. Jia, "Learning semantic image representations at a large scale," Ph.D. dissertation, UC Berkeley, 2014.
[59] C. Rivera, J. Chen, N. Xiong, S. L. Song, and D. Tao, "Ism2: Optimizing irregular-shaped matrix-matrix multiplication on gpus," *arXiv preprint arXiv:2002.03258*, 2020.
[60] (2020) GVProf. [Accessed August 28, 2020]. [Online]. Available: https://github.com/Jokeren/GVProf

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We profiled and optimized part of Rodinia benchmark v3.1, part of CUDA 10.1 SDK Samples, Quicksilver (Github commit@af27b3d), AlexeyAB/darknet (Github commit@342a8d1), Laghos (Github commit@368a914), and LAMMPS (Github commit@aa2b885) on ORNL's Summit supercomputer with GVProf. We ran all applications with one GPU and one process, except for LAMMPS which used up to 64 GPUs and 64 processes.

Each Summit node has 2 POWER9 CPU processors and 6 NVIDIA Volta GPUs with 96GB GPU memory in total and with the following system software: Linux 4.14.0, NVIDIA CUDA Toolkit 10.1.243, NVIDIA Driver 418.116, and GCC 6.4.0.

The performance of each application was averaged among 10 runs.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: Github:
↪ https://github.com/Jokeren/GVProf
Artifact name: GVProf
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Summit, Nvidia Tesla v100, IBM POWER9 CPUs

Operating systems and versions: Redhat 7.6 running Linux kernel 4.14.0

Compilers and versions: GCC 6.4.0, NVCC 10.1.243

Applications and versions: Rodinia benchmark v3.1, CUDA 10.1 SDK Samples, Quicksilver (Github commit@af27b3d), AlexeyAB/darknet (Github commit@342a8d1), Laghos (Github commit@368a914), and LAMMPS (Github commit@aa2b885)

Libraries and versions: CUDA Toolkit 10.1.243, IBM Spectrum MPI 10.3.1.02rtm0

Key algorithms: bidirectional slicing

Input datasets and versions: Quicksilver: built-in input. AlexeyAB/darknet: ./data/dog.jpg, and 'https://pjreddie.com/media/files/yolov3-tiny.weights'. Laghos: ./data/square01_quad.mesh. LAMMPS: bench/in.lj.

ARTIFACT EVALUATION

Verification and validation studies: The optimizations we performed to avoid value redundancies (e.g., loop invariant code motion) are well known to be semantics preserving.

Rodinia v3.1:

backprop: ./backprop 65536 ./backprop 131072

bfs: ./bfs ../data/graph1MW_6.txt

hotspot: ./hotspot 512 2 2 ../data/temp_512 ../data/power_512 output.out

sradv1: ./srad 1 0.5 502 458 ./srad 1 0.5 1004 916

dct8x8: ./dct8x8

dwt2d: ./dwt2d 192.bmp -d 192x192 -f -5 -l 3

dxtc: ./dxtc ../data

reduction: ./reduction −type=double −n=1024

pathfinder: ./pathfinder 100000 100 20 ./pathfinder 100000 200 20

histogram: ./histogram

euler3d: ./euler3d ../data/fvcorr.domn.097K

Quicksilver@af27b3d: We profiled and optimized Quicksilver using its built-in input and 'Examples/CORAL2_Benchmark/CORAL-p1.inp' with one GPU and one process. Quicksilver was compiled with -O3 optimization.

AlexeyAB/darknet@34a8d1: We profiled and optimized Darknet using './data/dog.jpg' as input with one GPU and one process. Pretrained weights were downloaded from 'https://pjreddie.com/media/files/yolov3-tiny.weights'. Darknet was compiled with -O3 optimization and cuBLAS enabled.

Laghos@368a914: We profiled and optimized Laghos using './data/square01_quad.mesh' and './data/cube01hex.mesh' as input with one GPU and process. Laghos was compiled with -O3 optimization.

LAMMPS@aa2b885: We ran LAMMPS using 'bench/in.lj' and 'examples/COUPLE/simple/in.l'. We tested GVProf's overhead using different numbers of GPUs from 1 to 64 across 11 nodes, with one GPU per MPI process. We optimized LAMMPS' performance with one GPU. LAMMPS was compiled with -O3 optimization.

Accuracy and precision of timings: We profiled and optimized benchmarks and applications on Summit. And the performance was calculated by averaging the execution time among 10 runs.

Kernel time: For programs that have internal timing utilities, we used them to measure kernel time. Otherwise, we used nvprof to measure kernel time.

Whole program time: We used the time command of Linux to get the whole programs' running time.

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: We ran the experiments several times in multiple batch jobs. The results show that our program optimizations yield stable performance improvements.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. We ran the experiments several times in multiple batch jobs. The results show that our program optimizations yield stable performance improvements.

Eliminating redundant data movement and computation is beneficial and not dependent on any particular GPU architecture. We have tested GVProf on NVIDIA Volta V100 and RTX 1650 GPUs.