# GVProf: A Value Profiler for GPU-based Clusters

**Keren Zhou**[1], Yueming Hao[2], John Mellor-Crummey[1], Xiaozhu Meng[1], and Xu Liu[2]

**[1]Rice University**

**[2]North Carolina State University**

# Value Profiling

- Values and instructions have *invariant*, *predictable*, or *approximate* behavior not eliminated at compile time
- Value profiling finds redundant value accesses and attributes them to source code to pinpoint opportunities for optimizations such as constant propagation, code specialization, and function inlining

# A Motivating Example

- Rodinia/pathfinder

```
void dynproc_kernel(int iteration, int *result, int *wall, ...) {
    for (int i : iteration) {
        result[tx] = shortest + wall[index];
        ...
    }
}
```

int8_t

- The values in the array wall are largely redundant
  - Between [1, 10]
  - Demoting wall to int8_t
  - 1.14x speedup

# GVProf

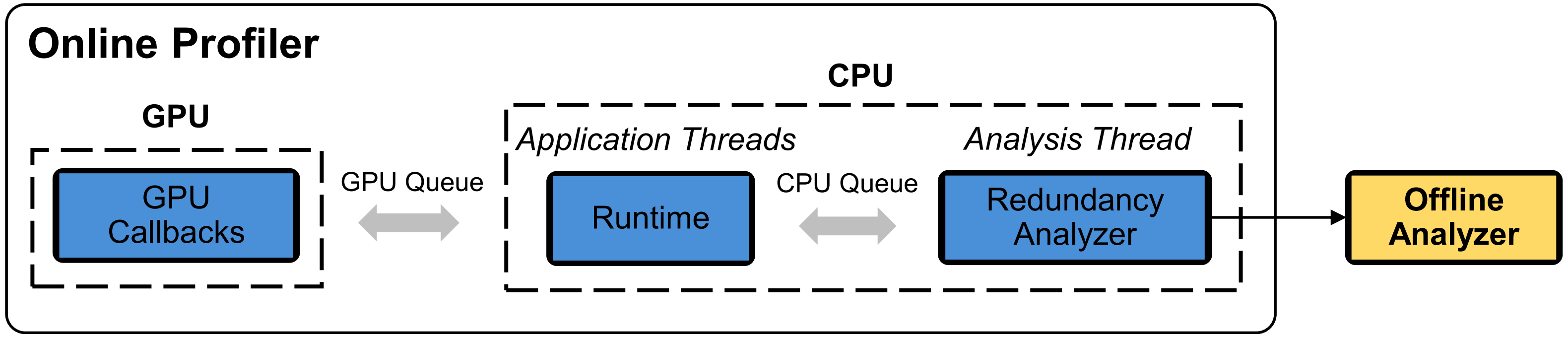
- Past research uses simulators to study value redundancy in GPU programs
  - High overhead
  - Source code recompilation
  - Limited to small benchmarks
- *GVProf* uses binary instrumentation to analyze GPU-accelerated applications with acceptable overhead and pinpoints value redundancies with full calling contexts

# Outline

- Design Overview
- Methodology
- Measurement
- Analysis
- Case Studies
- Contributions and Work in Progress

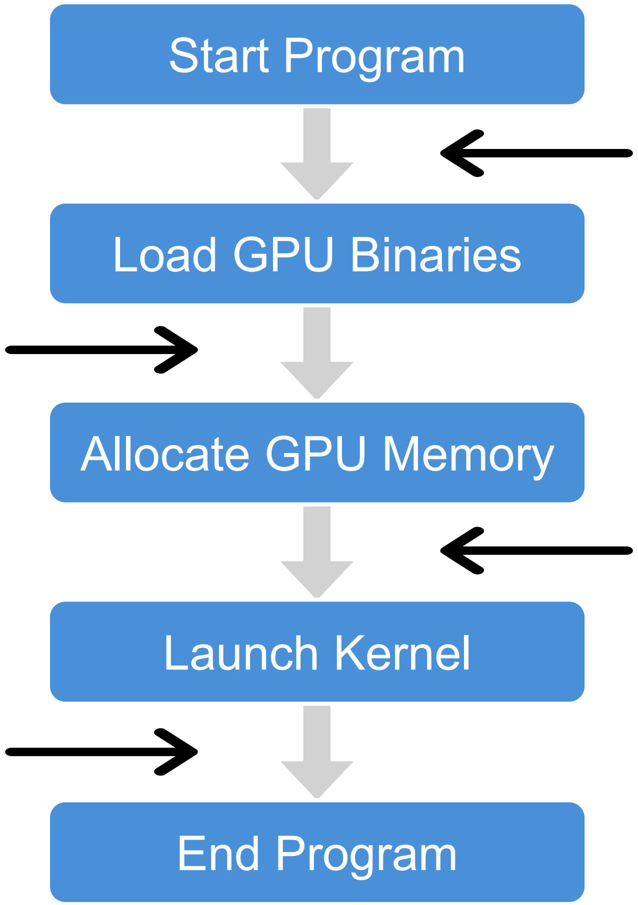
# Design Overview

- Online Profiler
  - CPU
    - Application threads for instrumenting kernels, managing buffers, and recording program calling context and memory objects
    - An analysis thread for on-the-fly analysis of redundancy metrics
  - GPU
    - Callbacks for instrumented GPU instructions
- Offline Analyzer
  - Association of redundancy metrics and program structure



# Workflow

- GVProf uses NVIDIA's Sanitizer API to intercept *binary load*, *kernel launch,* and *memory allocation*




- ❑ Create a background analysis thread to identify value redundancies

- ❑ Read CFGs of GPU functions
- ❑ Map each function's address to a file offset in a binary
- ❑ Add instrumentation callbacks

- ❑ Record calling context of memory allocations
- ❑ Create snapshots of memory allocations

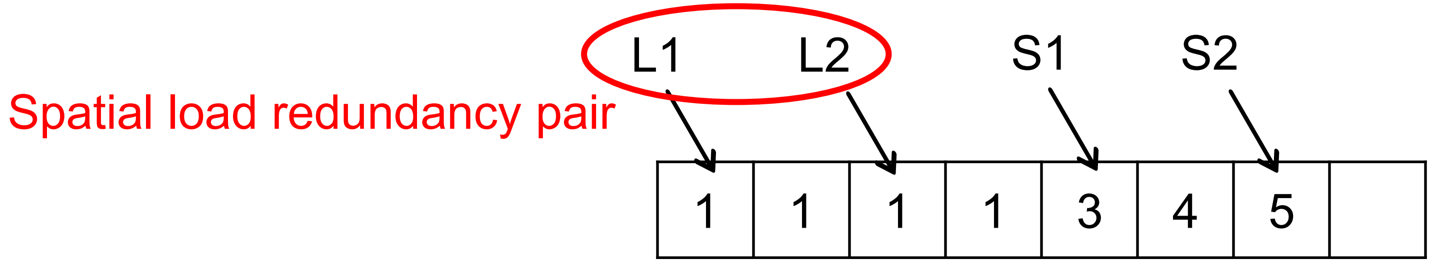
- ❑ Transfer GPU access records to the CPU
- ❑ Enqueue the records for the background analysis thread

# Outline

- Design Overview
- Methodology
- Measurement
- Analysis
- Case Studies
- Contributions and Work in Progress

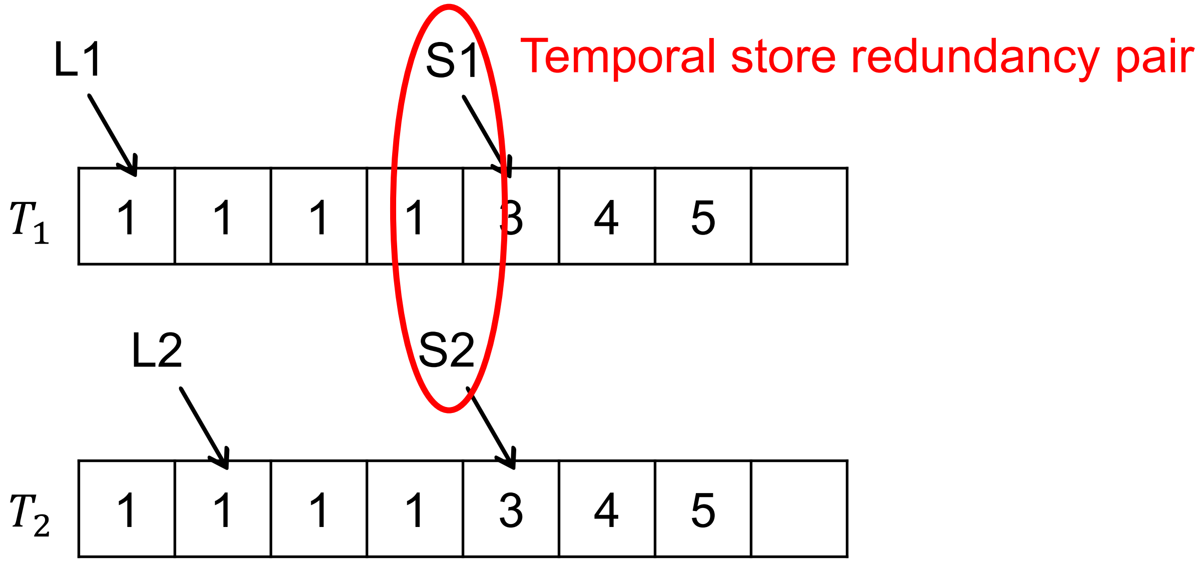
# Spatial Redundancy

- Spatial load redundancy
  - A memory load L2 is redundant *iff* it loads a value v from address A2, and another memory load L1 loads v from address A1, and A2 and A1 are in the memory range of a data object allocated by a GPU memory allocation
- Spatial store redundancy
  - A memory store S2 is redundant *iff* it stores a value v to address A2, and another memory store S1 stores v to address A1, and A2 and A1 are in the memory range of a data object allocated by a GPU memory allocation



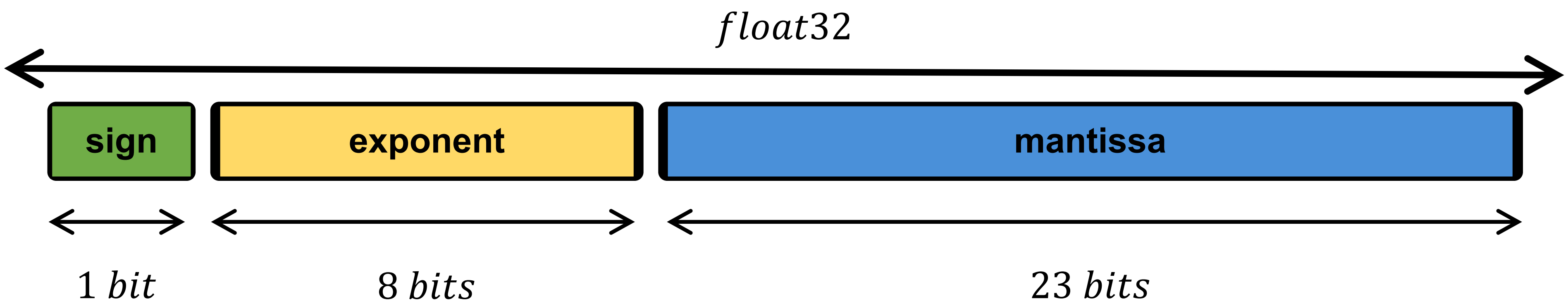
# Temporal Redundancy

- Temporal load redundancy
  - A memory load L2 is redundant *iff* it loads a value v from address A, and the previous memory load L1 from A also loaded v
- Temporal store redundancy
  - A memory store S2 is redundant *iff* it stores a value v to address A, and the previous memory store S1 also stored v to A



# Approximate Redundancy

- For floating point values, we adjust the length of the mantissa to compute approximate redundancy
  - $value = sign\ 2^{exponent} \times mantissa$



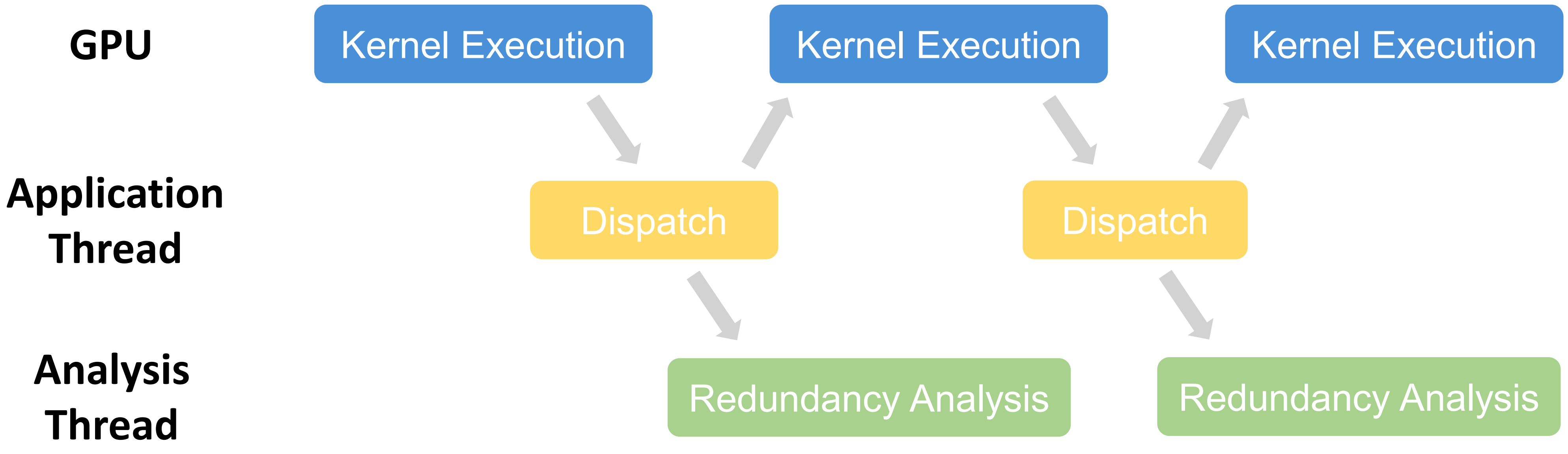

- Example
  - $85.0000125 = 2^6 \times 01010100000000000000010b$
  - $85.0 = 2^6 \times 01010100000000000000000b$
  - $85.0000125 \approx 85.0$ only consider the leading 21 bits of mantissa

# Outline

- Design Overview
- Methodology
- Measurement
- Analysis
- Case Studies
- Contributions and Work in Progress

# Processing Pipeline

- Overlap kernel execution and value analysis
  - GPU and application threads communicate via a GPU queue
  - Application threads and the analysis thread communicate via a CPU queue



# Hierarchical Sampling

- For applications that employ iterative and data parallel models, behaviors across different GPU kernel invocations and blocks are similar
- Kernel sampling
  - Monitor a subset of kernel invocations with the same invocation context
- Block sampling
  - Monitor a subset of a kernel invocation's thread blocks

# GPU Binary Instrumentation and CPU-GPU Communication

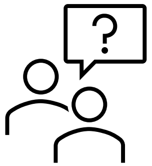
- At binary load time, add instrumentation at *memory access*, *thread block enter*, and *thread block exit*
- When instrumentation executes
  - Each warp reserves a slot for a record in the queue with atomicAdd
  - Each active thread in a warp writes its entry in the record
  - Each warp pushes the record into the queue
- The GPU signals the CPU to drain the queue
  - When the queue is full
  - When the GPU kernel is complete

# Outline

- Design Overview
- Methodology
- Measurement
- Analysis
- Case Studies
- Contributions and Work in Progress

# Spatial Redundancy Metrics

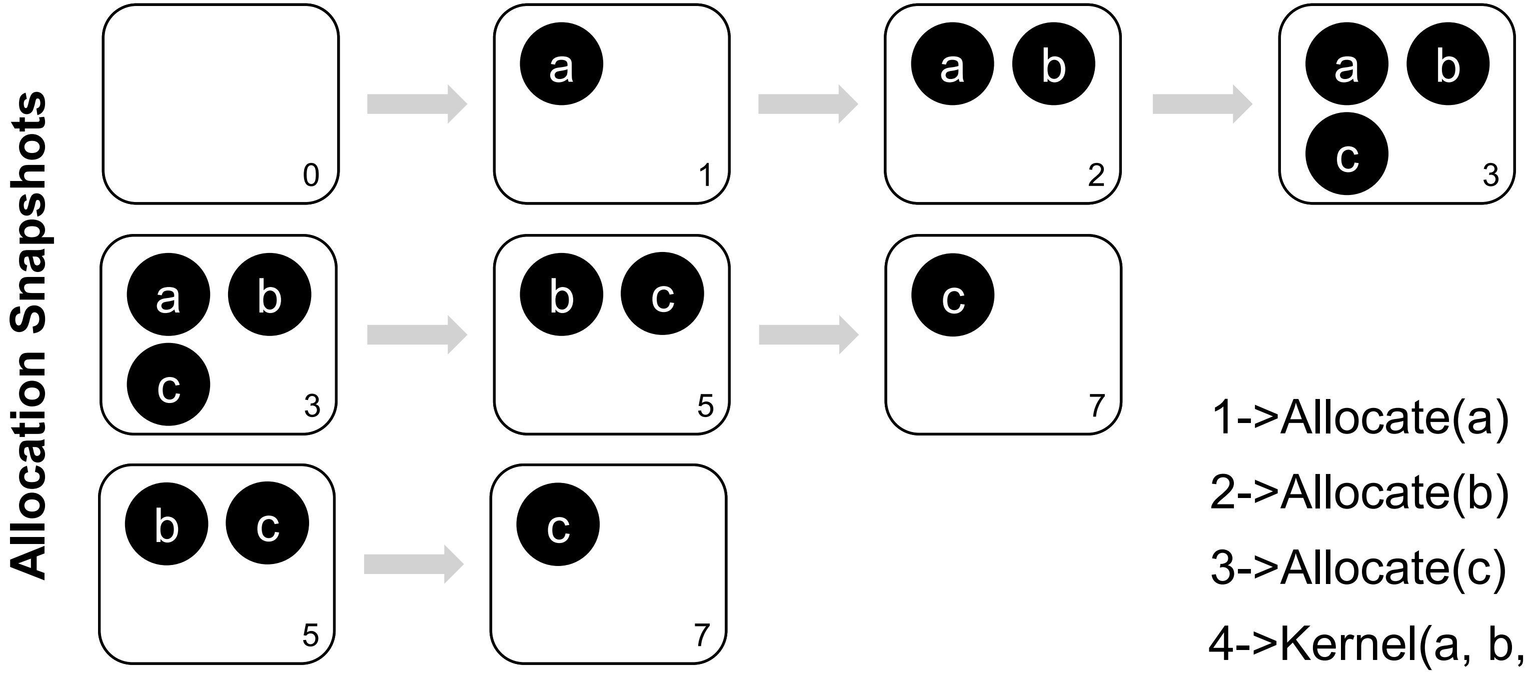
- $SR_{k,o,v} = \frac{SC_{k,o,v}}{N_{k,o}}$
  - The spatial redundancy rate $SR$ of a data object $o$ within kernel $k$ with value $v$
  - $SC_{k,o,v}$
    - Spatial redundancy count of a data object $o$ within kernel $k$ with value $v$
  - $N_{k,o}$
    - The total number of memory accesses of a data object o within kernel $k$
- Insights
  - 100% single value
    - Load/Store constant values
  - High ratio of single value
    - Common computation




- How do we identify data objects using memory addresses?
- How do we compare and interpret values?

# Identify Data Objects

- The analysis thread and GPU memory allocations are asynchronous
  - Record an *allocation snapshot* after each memory allocation and free
  - Look up the closest allocation snapshot




**Operations**

| | | |
|---|---|---|
| 1->Allocate(a) | Analyze(4) | Analyze(6) |
| 2->Allocate(b) | 5->Free(a) | |
| 3->Allocate(c) | 6->Kernel(b, c) | |
| 4->Kernel(a, b, c) | 7->Free(b) | |

# Identify Memory Access Type

- The raw value obtained for each GPU memory access is a sequence of binary bits, with no type information
  - Unit size
    - The length of each element accessed
  - Vector size
    - The number of elements accessed
  - Data type
    - Float/Integer

$< float.64 >$

DADD R4, R11, R12

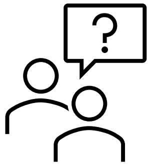
STG.128 [R1], R4

$vec\_size = 2$
$unit\_size = 64$
$data\_type = float$

- Use backward slicing to identify memory access types
- The algorithm and a concrete example are described in the paper

# Temporal Redundancy Metrics

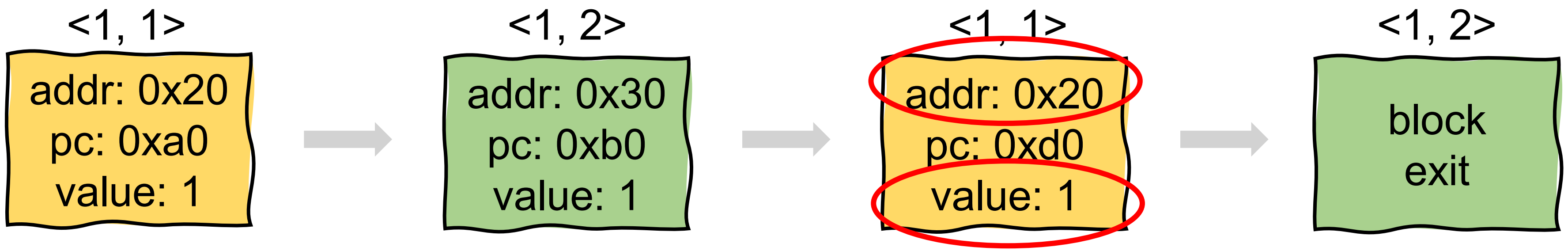
- $TR_{k,i,v} = \frac{TC_{k,i,v}}{N_{k,i}}$
  - The temporal redundancy rate $TR$ at instruction $i$ within kernel $k$ with value $v$
  - $TC_{k,i,v}$
    - Temporal redundancy count at instruction $i$ within kernel $k$ with value $v$
  - $N_{k,i}$
    - The total number of memory accesses at instruction $i$ within kernel $k$
- Insights
  - High redundancy in a loop
    - Value not in a register
  - High redundancy in device function
    - Failed to inline function




- How do we keep track of memory access records of each thread?

# Analysis of Temporal Redundancy

- The analysis thread identifies temporal redundancies within each GPU thread by scanning its access records and keeping only information about redundancies




**Last Seen Table**

| Thread ID | Address | PC | Value |
|---|---|---|---|

**Redundant Pairs**

| Source PC | Dst PC | Value | Count |
|---|---|---|---|

# Outline

- Design Overview
- Methodology
- Measurement
- Analysis
- Case Studies
- Contributions and Work in Progress

# Case Studies

- Platform
  - Summit supercomputer
  - Up to 64 NVIDIA Volta V100 GPUs
- Benchmark
  - Rodinia
    - A collection of parallel programs
  - Darknet/cuBLAS
    - An open-source deep learning framework
  - Quicksilver
    - A DOE proxy application for solving a dynamic Monte Carlo particle transport problem

  Single GPU

  - LAMMPS
    - A molecular dynamics code for large-scale materials modeling

  Up to 64 GPUs

# Evaluation of GVProf

- Measurement overhead
  - Up to 1000x without sampling
  - 7.5x in average with block sampling
- Sampling accuracy
  - 0.7% error in average with block sampling
- Optimizations
  - GVProf does not have false positives
    - But not all value redundancies can or should be eliminated
  - Achieved speedups from 1.02x to 2.42x

# Darknet

- 50% spatial load redundancy on shared memory with zeros
  - The first layer of YOLOv3-tiny has channel size 16 so that it only requires a 128x16 tile on shared memory
  - cuBLAS 128x32 matrix multiplication kernel uses a 128x32 tile on shared memory
  - Half of the shared memory is filled with zeros
- Achieved 1.60x speedup by employing a fast implementation for tall-and-thin matrices

# Quicksilver

- 20.9% temporal load redundancy in *qs_assert* to check boundary conditions
  - *qs_assert* is enclosed in a non-inlined device function invoked in a loop and checks loop invariant values
  - Achieved 1.10x speedup by hoisting the *qs_assert* out of the device function
- 30.2% temporal load redundancy in the epilogue of *getReactionCrossSection* and *macroscopicCrossSection*
  - The two non-inlined device functions are called in a loop, introducing redundant local memory store and load operations to spill and restore unchanged values
  - Achieved 1.10x speedup by inlining these two functions into their caller

# LAMMPS

- 52.3% spatial redundant stores with zeros in a deep calling context
  - Kokkos resizes an array by allocating a new piece of memory and initializing it to zero
  - Achieved 1.47x speedup by increasing the array growth factor to reduce the calls to *Kokkos::resize()*

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

# Outline

- Design Overview
- Methodology
- Measurement
- Analysis
- Case Studies
- Contributions and Work in Progress

# Contributions and Work in Progress

- GVProf highlights
  - identifies temporal and spatial value redundancies for both memory loads and stores;
  - provides detailed information to guide optimization, including calling contexts, data objects, and source code attribution;
  - employs various optimizations to reduce its overhead
- Work in progress
  - Track value changes regarding the whole program execution
    - memset/memcpy
    - Inter kernels
  - Analyze value patterns for each data object
    - Type misuse
    - Immutable values