A Fine-grained CPU-GPU Analysis Framework

Yueming Hao
yhao24@ncsu.edu
North Carolina State University

Contributors:

Keren Zhou, John Mellor-Crummey , Xiaozhu Meng, Xu Liu

GPU Profilers Lack

GPU Profilers

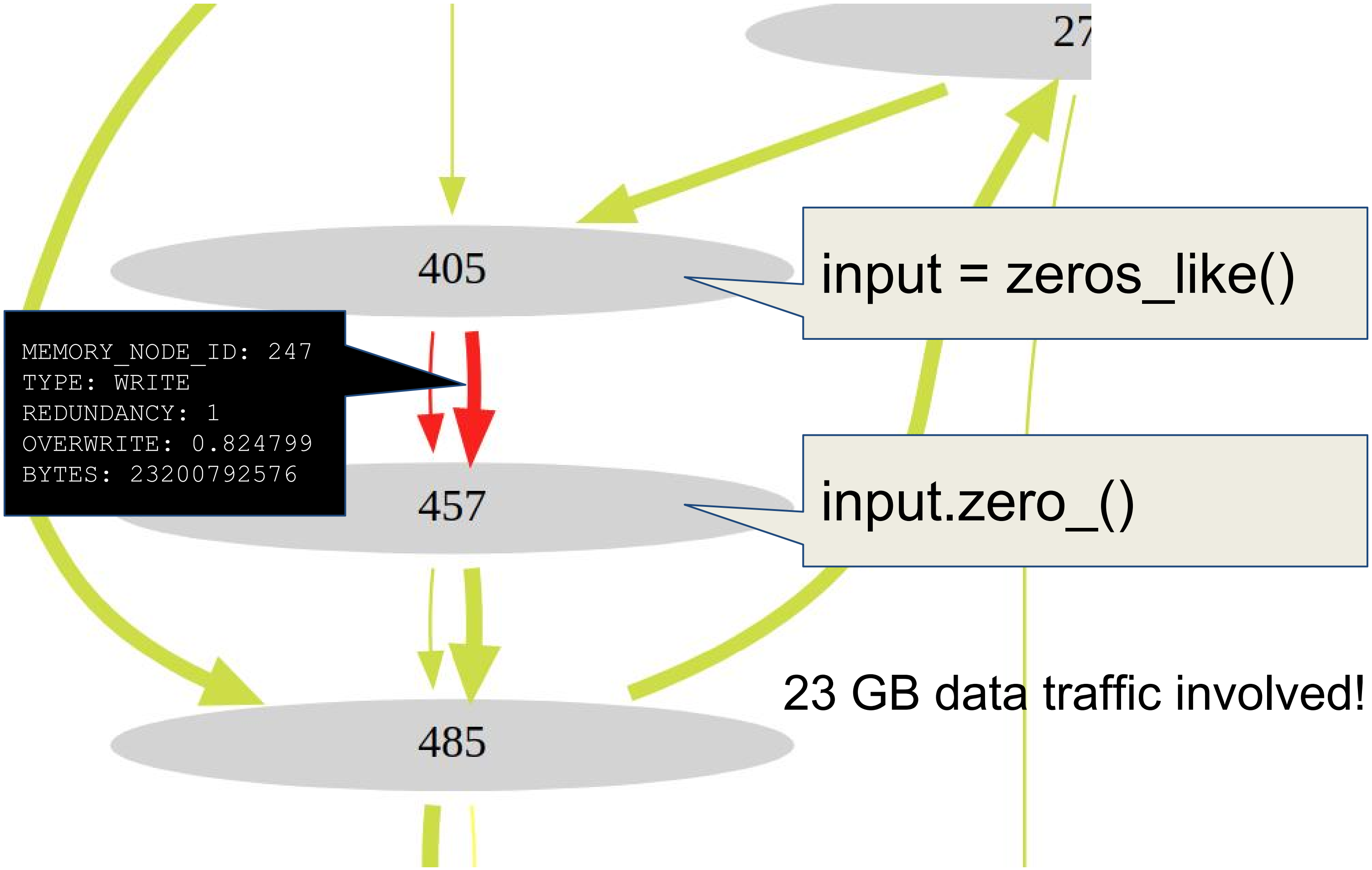
- NVIDIA: Nsight Compute, Nsight Systems
- AMD: Rocm Profiler

Limitations:

- hotspot analysis only
- no value profiling

Our goal is to develop a value profiler for GPUs!

Motivation: Pytorch-Deepwave

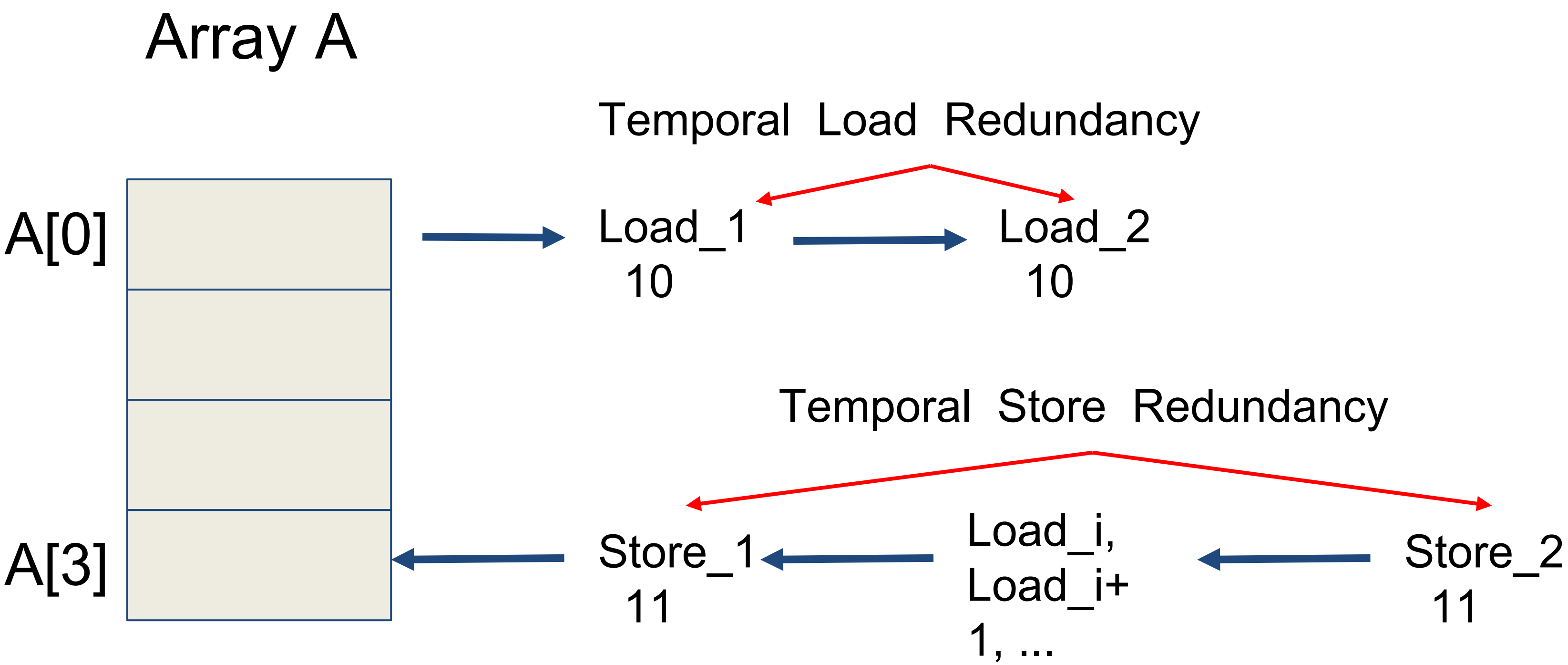


Case Study: Pytorch-Deepwave

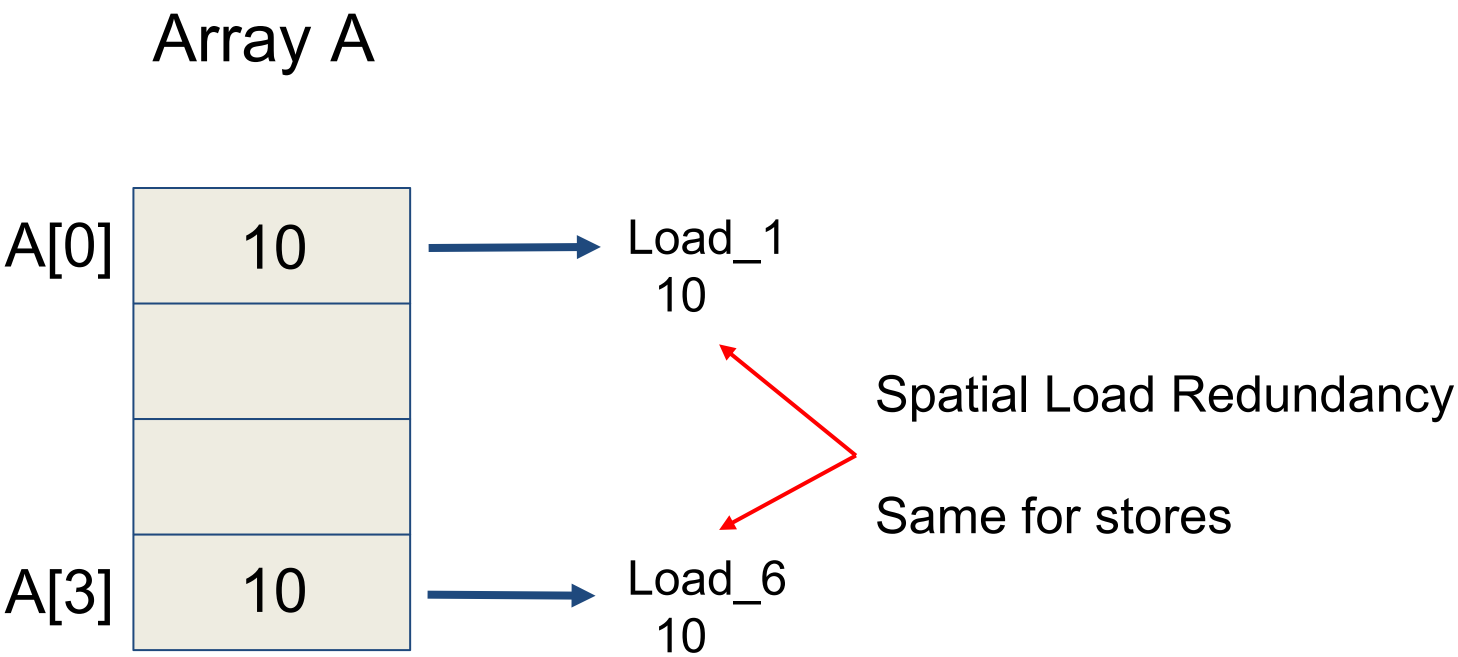
```
void replication_pad3d_backward_out_cuda_template(...){
  gradInput.resize_as_(input);
  gradInput.zero_();
  ...}
Tensor replication_pad3d_backward_cuda(...){
- auto gradInput = at::zeros_like(input,
      LEGACY_CONTIGUOUS_MEMORY_FORMAT);


  replication_pad3d_backward_out_cuda_template(gradInput,
        gradOutput, input, paddingSize);
  ...
}
```

For the ReplicationPad operator in the backward phase,
RTX 2080Ti: 1.07x speedup
A100: 1.04x speedup
https://github.com/pytorch/pytorch/pull/48890 The PR has been merged.

Temporal Redundancy



Spatial Redundancy



GVProf: A value profiler for GPU-based clusters. [SC 2020]
Zhou Keren, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu.

Fine-Grained Value Patterns

	Array A	Array A	Array A
A[0]	1	1	0
	1	1	0
	1	1	0
	1	1	0
	2	1	0
A[5]	3	1	0
	Frequent Values	Single Value	Single Zero

Fine-Grained Value Patterns

Array A

A[0]	1
A[1]	2
A[2]	3
A[3]	4
A[4]	5
A[5]	6

Structured Values
y = kx + b

Array A
Data type: int32

A[0]	1
	127
	32
	-120
	15
A[5]	12

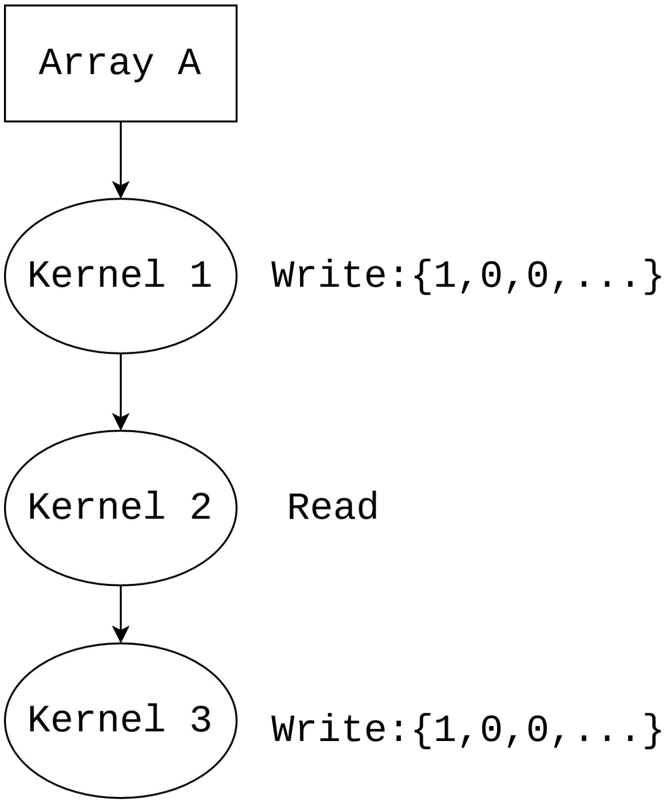
Heavy Type

Array A
Data type: float32

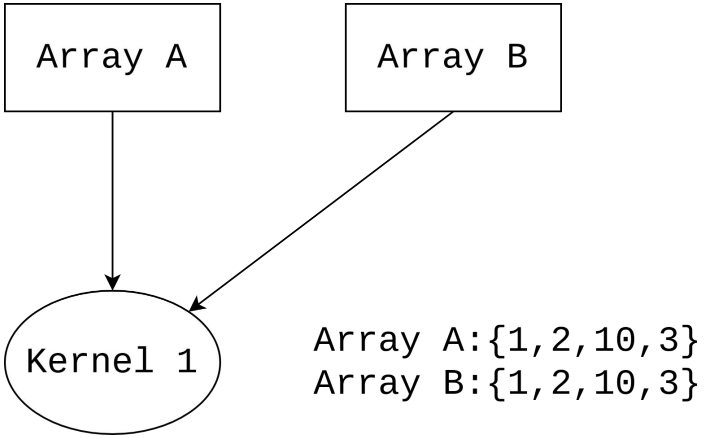
A[0]	1.0001
	1.0003
	1.0
	1.0
	1.0006
A[5]	1.000004

Approximate Values

Coarse-Grained Value Patterns

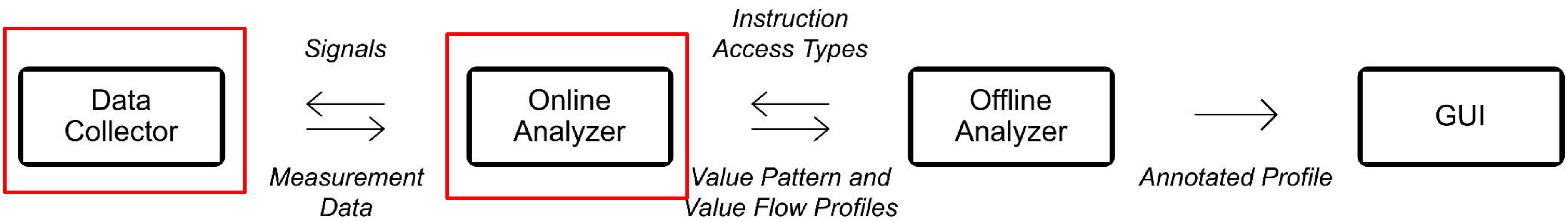


Redundant Values



Duplicate Values

ValueExpert Overview



Online analyzer processes collected performance data to identify value patterns and build value flow graphs. This component dispatches

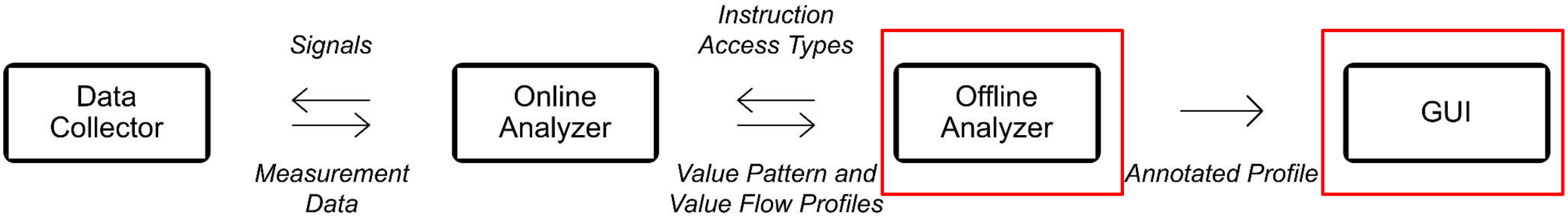
- preprocess of coarse-grained value patterns on GPUs
- all other analysis works on CPUs

Data collector utilizes NVIDIA's Sanitizer API to instrument

- each GPU memory instruction to obtain its touched memory addresses, value loaded/stored to the memory addresses
- GPU APIs, including memory copy, memory set, and kernel launch

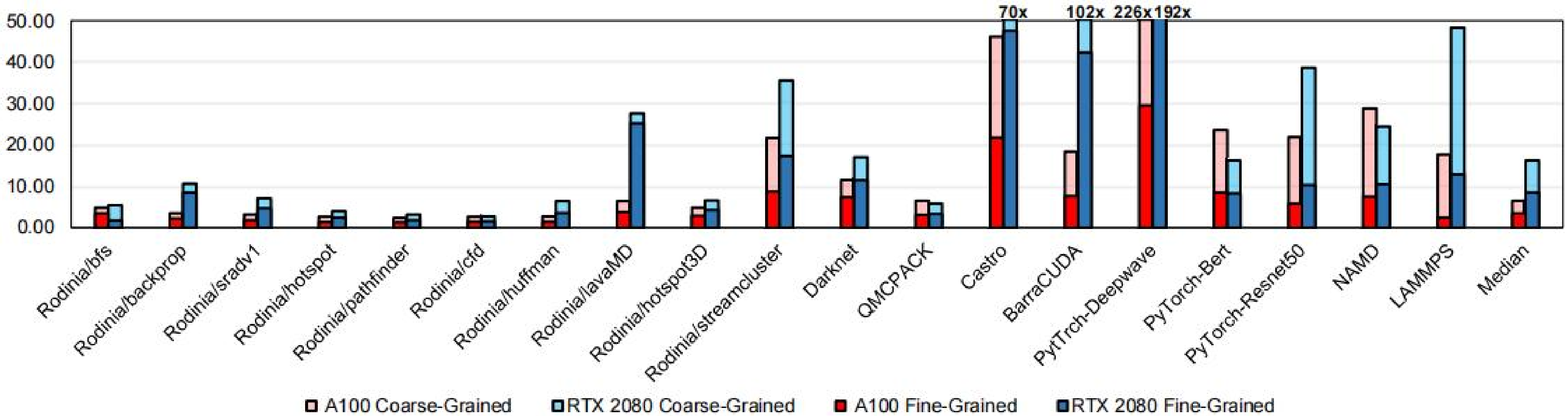
ValueExpert: exploring value patterns in GPU-accelerated applications. [ASPLOS 2022]
Zhou, Keren, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu.

ValueExpert Overview



The offline analyzer mainly analyzes CPU and GPU binaries to obtain information about line mapping etc.

Overhead



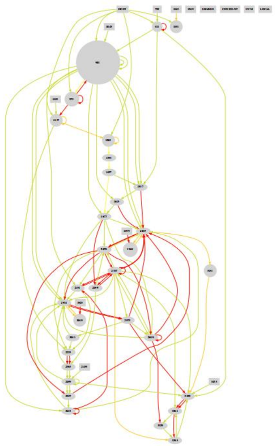
Median overhead:

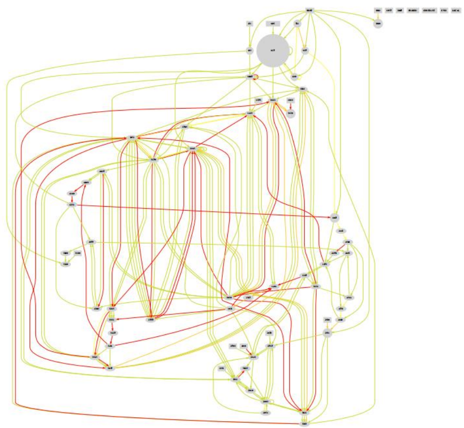
- 7.35× on RTX 2080 Ti
- 7.81× on A100

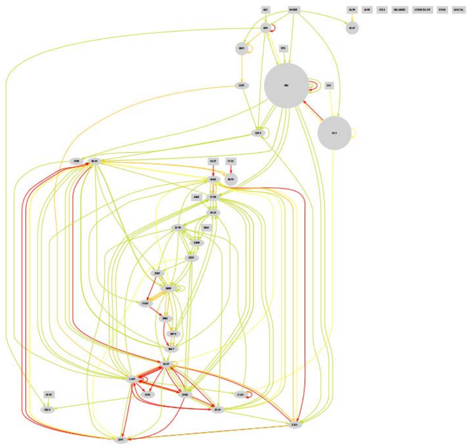
Optimization Results

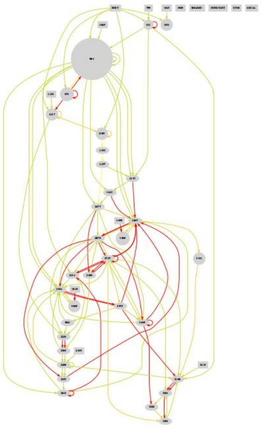
Applications	RTX 2080Ti		A100	
	Kernel Speedup	Memory Speeup	Kernel Speedup	Memory Speeup
Rodinia/backprop	8.18X	1.01X	1.67X	1.01X
Rodinia/sradv1	1.52X	1.03X	1.11X	1.06X
Rodinia/pathfinder	1.13X	4.21X	1.37X	3.27X
Rodinia/hotspot3D	2.00X	1.00X	1.99X	0.99X
Darknet	1.06X	1.82X	1.05X	1.73X
Castro	1.27X	1.00X	1.24X	1.02X
BarraCUDA	1.06X	1.13X	1.06X	1.13X
PyTorch-Deepwave	1.07X	1.01X	1.04X	1.00X
PyTorch-Bert	1.57X	1.01X	1.59X	1.00X
PyTorch-Resnet50	1.02X	1.00X	1.03X	0.98X
LAMMPS	-	6.03X	-	5.19X
Geometric Mean	1.58X	1.34X	1.39X	1.28X
Median	1.29X	1.01X	1.11X	1.02X

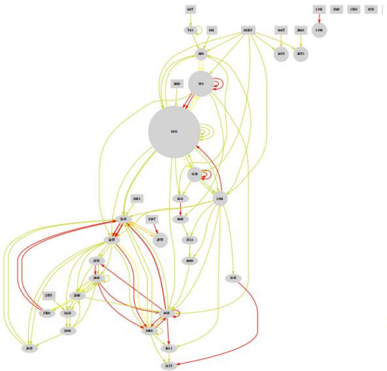
PyTorch Benchmarks

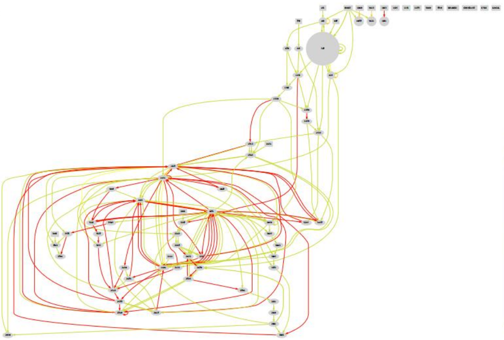


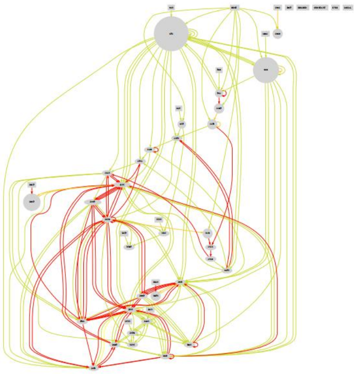


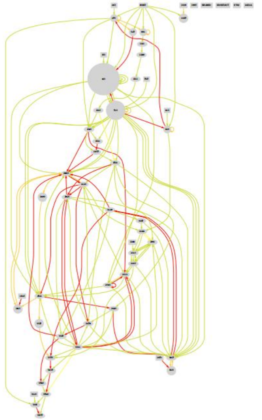




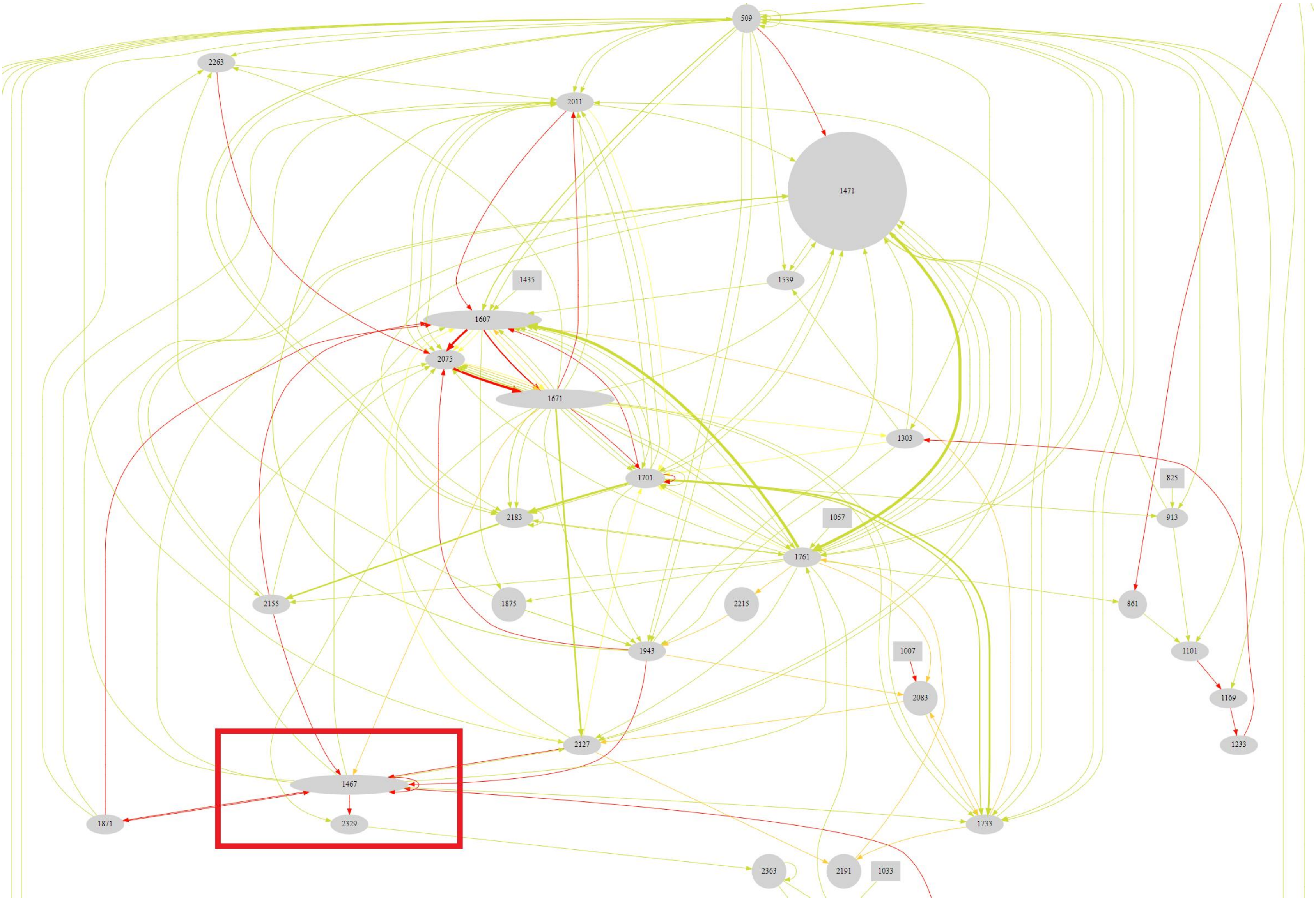




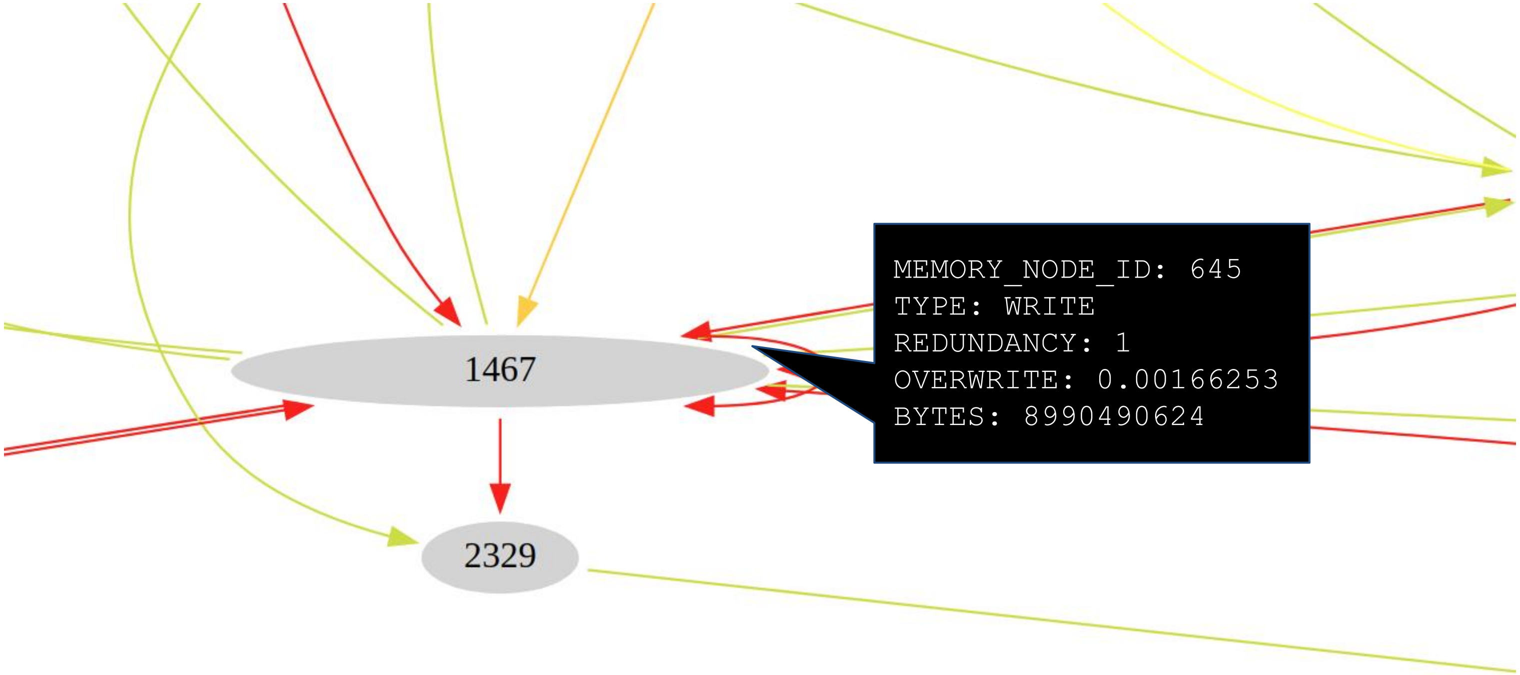




Case Study: PyTorch-Resnet50



Case Study: Pytorch-Resnet50



Case Study: Pytorch-Resnet50

```
void THNN_(SpatialConvolutionMM_updateOutput)(...) {
+ if (bias) {
    if (ones->dim() != 2 || ones->size(0)*ones->size(1) <
          outputHeight*outputWidth) {
      THCTensor_(resize2d)(state, ones, outputHeight, outputWidth);
      THCTensor_(fill)(state, ones, ScalarConvert<int, scalar_t>::
           to(1)); }
+ }
}
```

input × filter + bias

1.02× and 1.03× speedups for convolution layers on RTX 2080 Ti and A100

https://github.com/pytorch/pytorch/pull/48540 The PR has been merged.

What about the CPU-GPU interactions?

Interesting Topics

- CPU-GPU managed memory
- CPU-GPU data transmission
- GPU-GPU data transmission
- CPU-GPU shared memory hierarchy

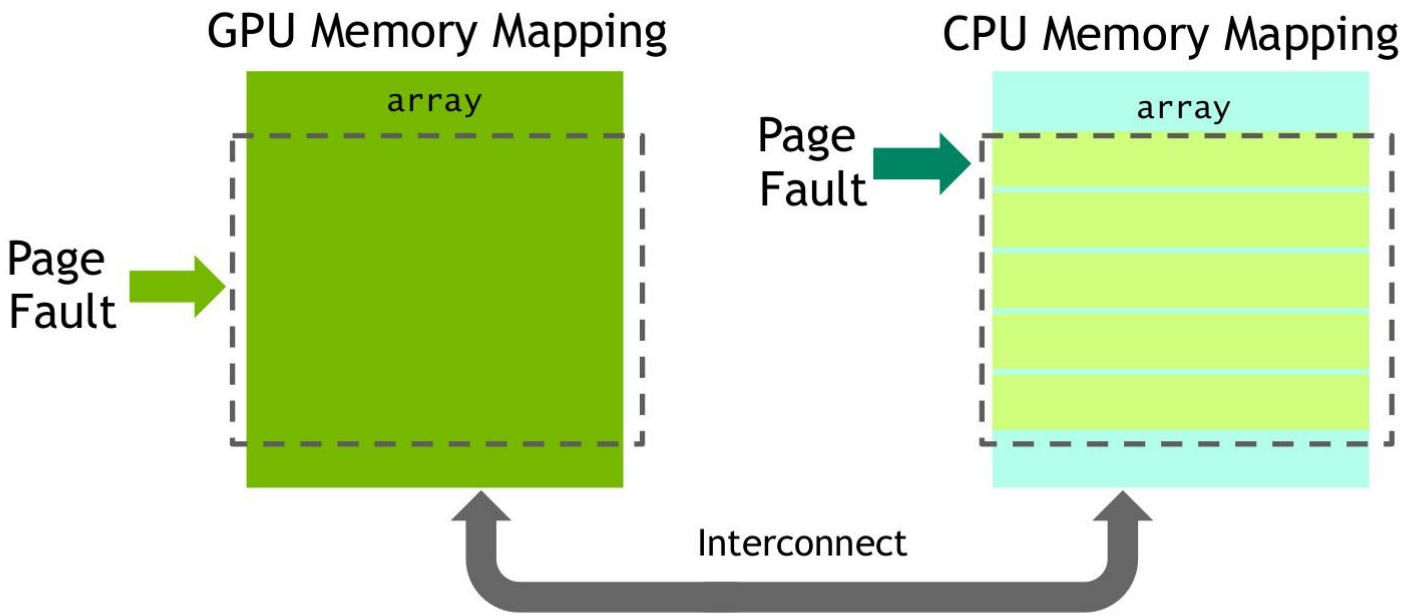
How Managed Memory Works

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

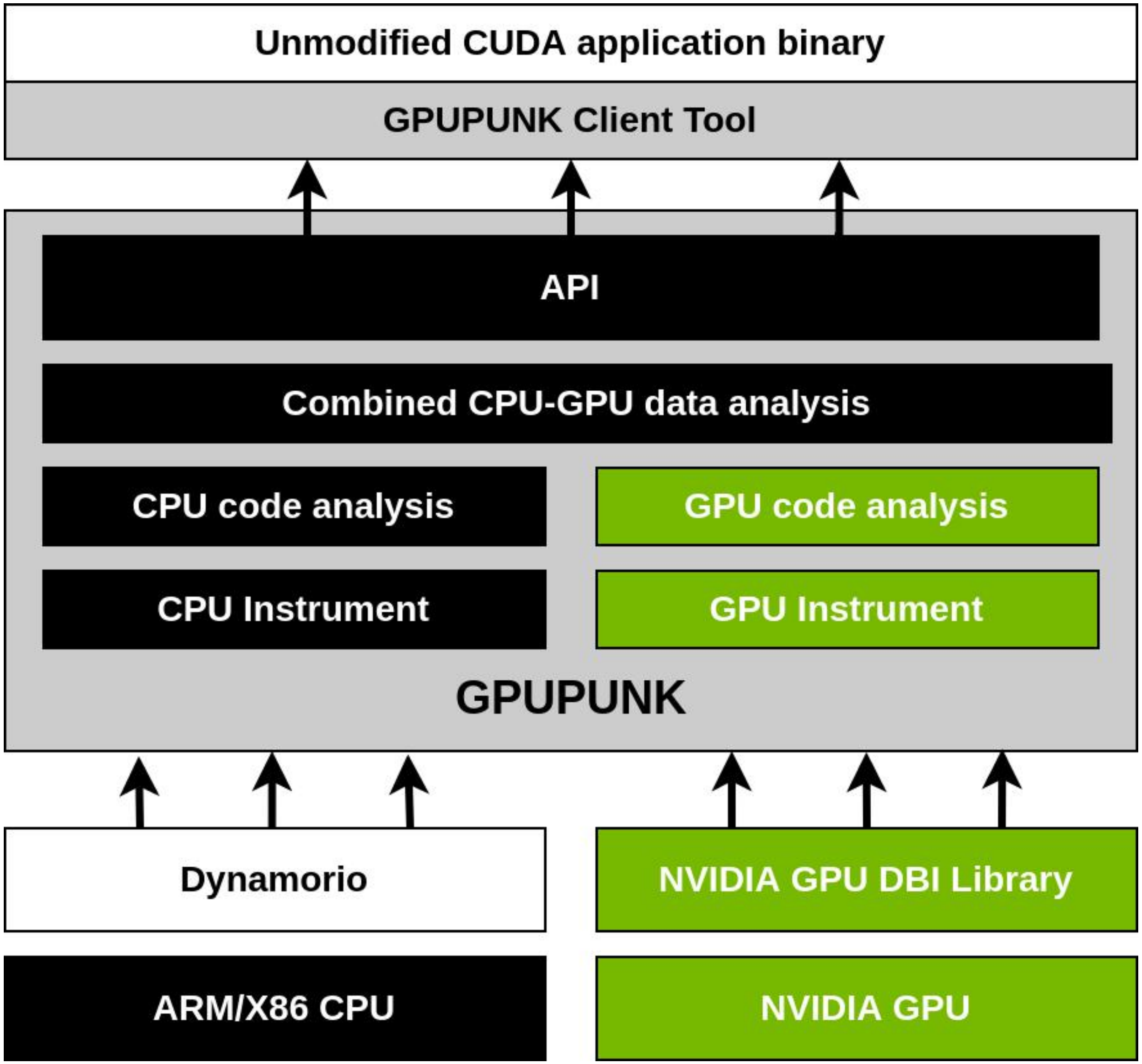
```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



Potenional frequent true shaing and false sharing!

Source: https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf

GPUPUNK Overview



Implementation Details

- Combined Fine-grained CPU-GPU CCT
  - Build the combined accurate CCT based on instrumentation data, inlucding device function calls
- Combined CPU-GPU Data Analysis
  - Instrument all CPU memory accesses and GPU memory accesses
  - Estimate page faults invoked by memory accesses
  - Analyze potentional page false sharing

Reduce Overhead

- GPU Preprocesor
  - Use a small kernel preprocess data collected from GPU
  - Expose those APIs to users
- Minimizing CPU-GPU Data Transfers
  - Use an adaptive copy mechanism to switch between different copy strategies
- Kernel Filtering
  - Supports monitoring a subset of GPU kernels specified by users
- Kernel and Block Sampling
  - Supports monitoring a subset of GPU kernel and block executions specified by users

Conclusion

- GVProf & ValueExpert
  - https://github.com/GVProf/GVProf
- GPUPUNK(work in progress)
  - https://github.com/FindHao/gpupunk

Questions?