Optimizing GPU-accelerated Applications with HPCToolkit

Keren Zhou and John Mellor-Crummey

Department of Computer Science

Rice University

Problems with Existing Tools

- OpenMP Target, Kokkos, and RAJA generate sophisticated code with many small procedures
  - Complex calling contexts on both CPU and GPU
- Existing performance tools are ill-suited for analyzing such complex programs because they lack a comprehensive profile view
- At best, existing tools only attribute runtime cost to a flat profile view of functions executed on GPUs

Profile View with HPCToolkit

vecAdd.cu

```
__device__
int __attribute__ ((noinline)) add(int a, int b) {
  return a + b;
}


extern "C"
__global__
void vecAdd(int *l, int *r, int *p, size_t N, size_t iter1, size_t iter2) {
  size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
  for (size_t i = 0; i < iter1; ++i) {
    if (idx < N) {
      p[idx] = add(l[idx], r[idx]);
    }
  }
  for (size_t i = 0; i < iter2; ++i) {
    if (idx < N) {
      p[idx] = add(l[idx], r[idx]);
    }
  }
}
```

Loop

Call

Kernel Launch

Top-down view | Bottom-up view | Flat view

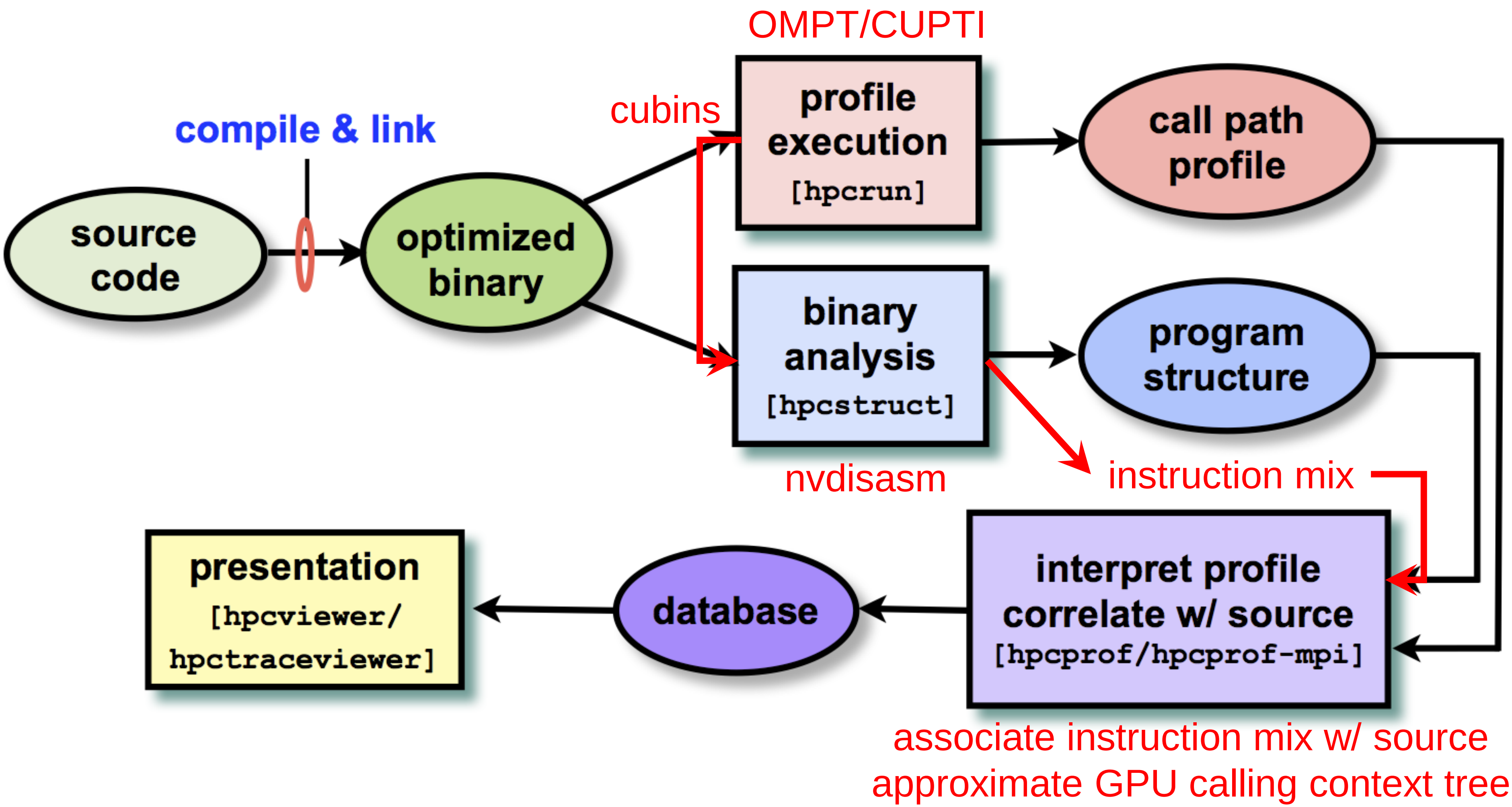
Scope	GPU INST:Sum (I)	GPU INST:Sum (E)	GPU STALL:Sum (I)	GPU STALL:Sum (E)
Experiment Aggregate Metrics	1.93e+05 100 %	1.93e+05 100 %	2.09e+05 100 %	2.09e+05 100 %
<program root>	1.93e+05 100 %		2.09e+05 100 %	
516: main	1.93e+05 100 %		2.09e+05 100 %	
60: main._omp_fn.0	1.93e+05 100 %		2.09e+05 100 %	
loop at main.cu: 83	1.93e+05 100 %		2.09e+05 100 %	
83: <cuda kernel>	1.93e+05 100 %		2.09e+05 100 %	
34: vecAdd	1.93e+05 100 %	9.01e+04 46.8%	2.09e+05 100 %	4.10e+04 19.6%
loop at vecAdd.cu: 16	1.21e+05 62.8%	6.96e+04 36.2%	1.31e+05 62.7%	2.87e+04 13.7%
loop at vecAdd.cu: 11	5.94e+04 30.9%	8.19e+03 4.3%	6.96e+04 33.3%	4.10e+03 2.0%
13: $vecAdd$_Z3addii	1.28e+04 6.6%	1.28e+04 6.6%	1.02e+04 4.9%	1.02e+04 4.9%
loop at vecAdd.cu: 3	8.70e+03 4.5%	8.70e+03 4.5%	7.68e+03 3.7%	7.68e+03 3.7%
vecAdd.cu: 3	2.56e+03 1.3%	2.56e+03 1.3%	1.02e+03 0.5%	1.02e+03 0.5%
vecAdd.cu: 21	1.54e+03 0.8%	1.54e+03 0.8%	1.54e+03 0.7%	1.54e+03 0.7%

Hidden Info: NVCC generates a loop at the end of every function

Challenges to Build a Scalable Tool

- GPU measurement collection
  - Multiple worker threads launching kernels to a GPU
  - A background thread reads measurements and attributes them to the corresponding worker threads
- GPU measurement attribution
  - Read line map and DWARF in heterogenous binaries
  - Control flow recovery
- GPU API correlation in CPU calling context tree
  - Thousands of GPU invocations, including kernel launches, memory copies, and synchronizations in large-scale applications

Extend HPCToolkit



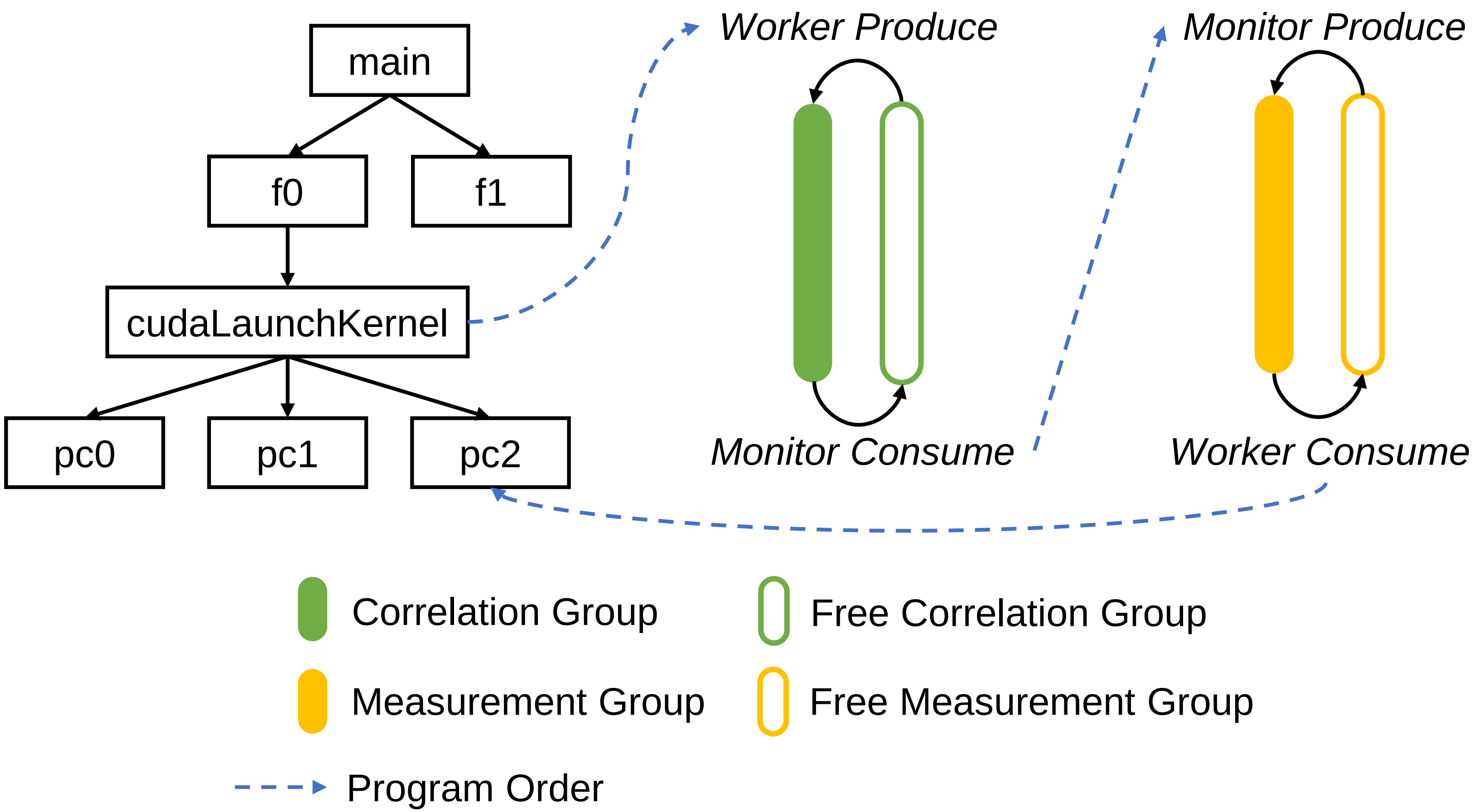
GPU Performance Measurement

- Two categories of threads
  - Worker threads (N per process)
    - Launch kernels, move data, and synchronize GPU calls
  - GPU monitor thread (1 per process)
    - Monitor GPU events and collect GPU measurements
- Interaction
  - **Create correlation:** A worker thread *T* creates a correlation record when it launches a kernel and tags the kernel with a correlation ID *C*, notifying the monitor thread that *C* belongs to *T*
  - **Attribute measurements**: The monitor thread collects measurements associated with *C* and communicates measurement records back to thread *T*

Coordinating Measurements

- Communication channels: wait-free unordered stack groups
- A private stack and a shared stack used by two threads
  - **POP**: pop a node from the private stack
  - **PUSH**(*CAS*): push a node to the shared stack
  - **STEAL**(*XCHG*): steal the contents of the shared stack, push the chain to the private stack
- Wait-free because **PUSH** fails at most once when a concurrent thread **STEALs** contents of the shared stack

Worker-Monitor Communication



GPU Metrics Attribution

- Attribute metrics to PCs at runtime
- Aggregate metrics to lines
  - Relocate cubins’ symbol table
    - Initial values are zero
    - Function addresses are overlapped
  - Read .debug_lineinfo section if available
- Aggregate metrics to loops
  - nvdisasm –poff –cfg for all valid functions
  - Parse dot files to data structures for Dyninst
  - Use ParseAPI to identify loops

GPU API Correlation with CPU Calling Context

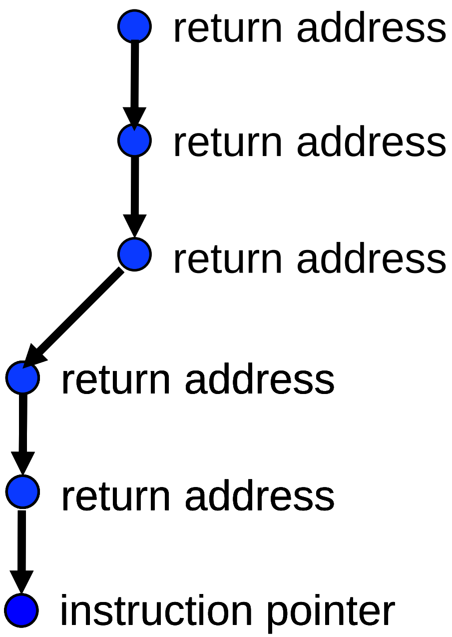
- Unwind a call stack from API invocations, including kernel launches, memory copies, and synchronizations
- Query an address's corresponding function in a global shared map
- Applications have deep call stacks and large codebase
  - Nyx: up to 60 layers and 400k calls
  - Laghos: up to 40 layers and 100k calls

Fast Unwinding

- Memoize common call path prefixes
  - Temporally-adjacent samples in complex applications often share common call path prefixes
  - Employ eager (mark bits) or lazy (tramopoline) marking to identify LCA of call stack unwinds
- Avoid costly access to mutable concurrent data
  - Cache unwinding recipes in a per thread hash table
- Avoid duplicate unwinds
  - Filter CUDA Driver APIs within CUDA Runtime APIs

Memoizing common call path prefixes

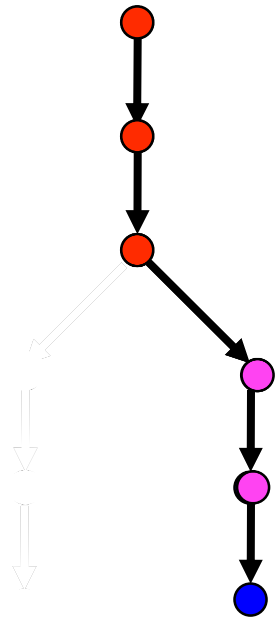
Call path sample



Eager LCA
Arnold & Sweeny, IBM TR, 1999.

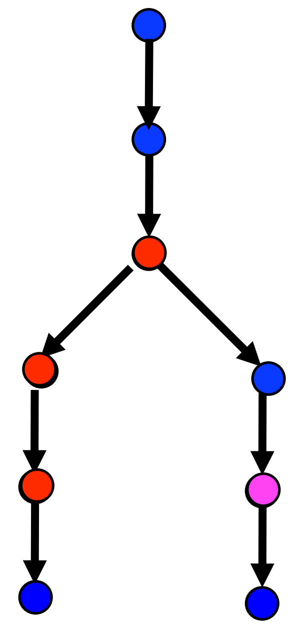
Lazy LCA
Froyd et al, ICS05.

Eager LCA



- **mark frame RAs while unwinding**
- **return from marked frame clears mark**
- **new calls create unmarked frame RAs**
- **mark frame RA during next unwind**
- **prior marked frames are common prefix**

Lazy LCA



- **mark innermost frame RA**
- **return from marked frame moves mark**
- **new calls create unmarked frames**
- **mark frame RA during next unwind**
- **prior marked frame indicates common prefix**

Analysis Methods

- Heterogenous context analysis
- GPU calling context approximation
- Instruction mix
- Metrics approximation
  - Parallelism
  - Throughput
  - Roofline
  - …

Heterogenous Context Analysis

- Associate GPU metrics with calling contexts
  - Memory copies
  - Kernel launches
  - Synchronization
  - …
- Merge CPU calling context tree with GPU calling context tree
  - CPUTIME > Memcpy shows implicit synchronization

CPU Importance

- $\mathrm{CPU_{IMPORTANCE}}$:

Ratio of a procedure's time to the whole execution time

$$\boxed{\mathrm{Max}\left(\frac{\mathrm{CPU_{TIME}} - \mathrm{SUM}(\mathrm{GPU\_API_{TIME}})}{\mathrm{CPU_{TIME}}}, 0\right)} \times \boxed{\frac{\mathrm{CPU_{TIME}}}{\mathrm{EXECUTION_{TIME}}}}$$

Ratio of a procedure's pure CPU time.
If more time is spent on GPU than CPU, the ratio is set to 0

- $\mathrm{GPU\_API_{TIME}}$:
  - $\mathrm{KERNEL_{TIME}}$: *cudaLaunchKernel*, *cuLaunchKernel*
  - $\mathrm{MEMCPY_{TIME}}$: *cudaMemcpy*, *cudaMemcpyAsync*
  - $\mathrm{MEMSET_{TIME}}$: *cudaMemset*
  - …

GPU API Importance

- $\text{GPU\_API}_{\text{IMPORTANCE}}$

$$\frac{\text{GPU\_API}_{\text{TIME}}}{\text{SUM}(\text{GPU\_API}_{\text{TIME}})}$$

Consider the importance of the memory copy to all the GPU time

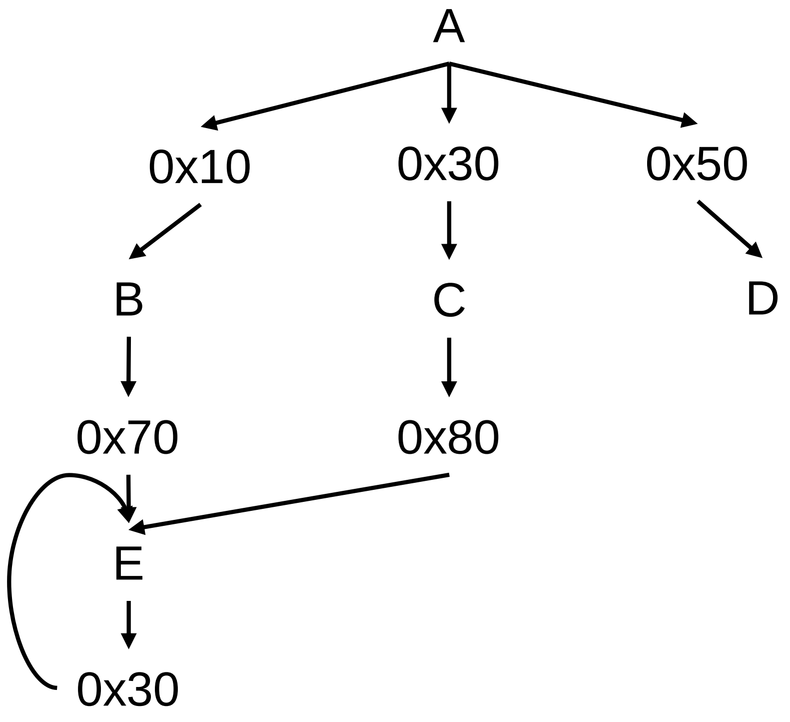
- Find which type of GPU API is the most expensive
  - Kernel: optimize specific kernels with PC Sampling profiling
  - Other APIs: apply optimizations based on calling context

GPU Calling Context Tree

- Problem
  - Unwinding call stacks on GPU is costly for massive parallel threads
  - No available unwinding API
- Solution
  - Reconstruct calling context tree using call instruction samples

Step 1: Construct Static Call Graph

- Link call instructions with corresponding functions

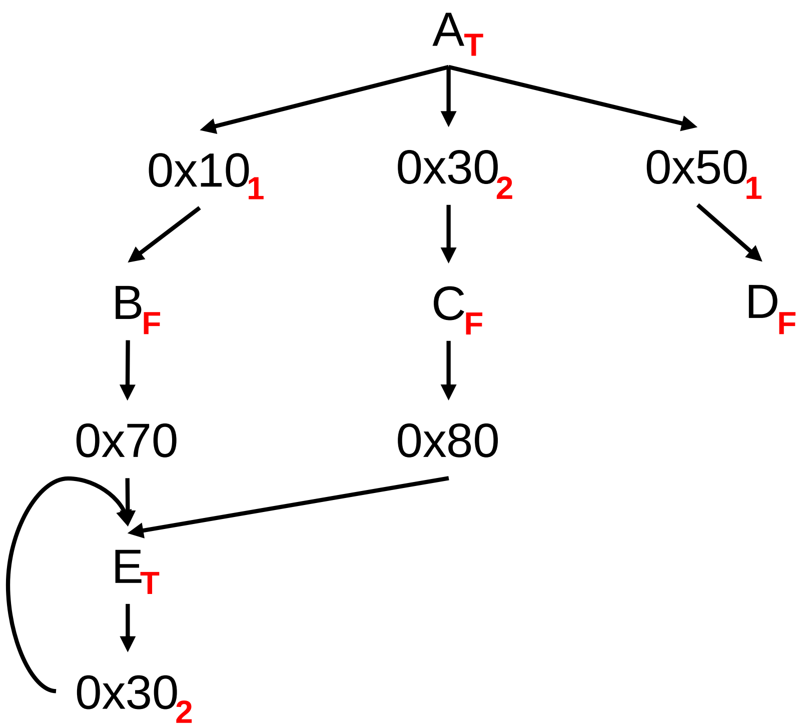


Step 2: Construct Dynamic Call Graph

- Challenge
  - Call instructions are sampled (**Unlike gprof**)
- Assumptions
  - If a function is sampled, it must be called somewhere
  - If there are no call instruction samples for a sampled function, we assign each potential call site one call sample

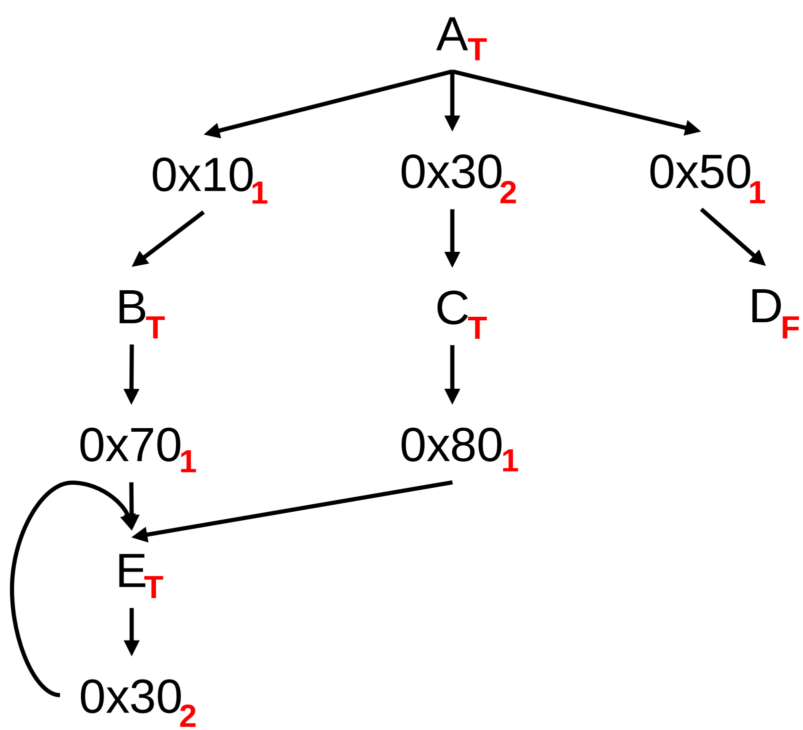
Step 2: Construct Dynamic Call Graph

- Assign call instruction samples to call sites
- Mark a function with *T* if it has instruction samples, otherwise *F*



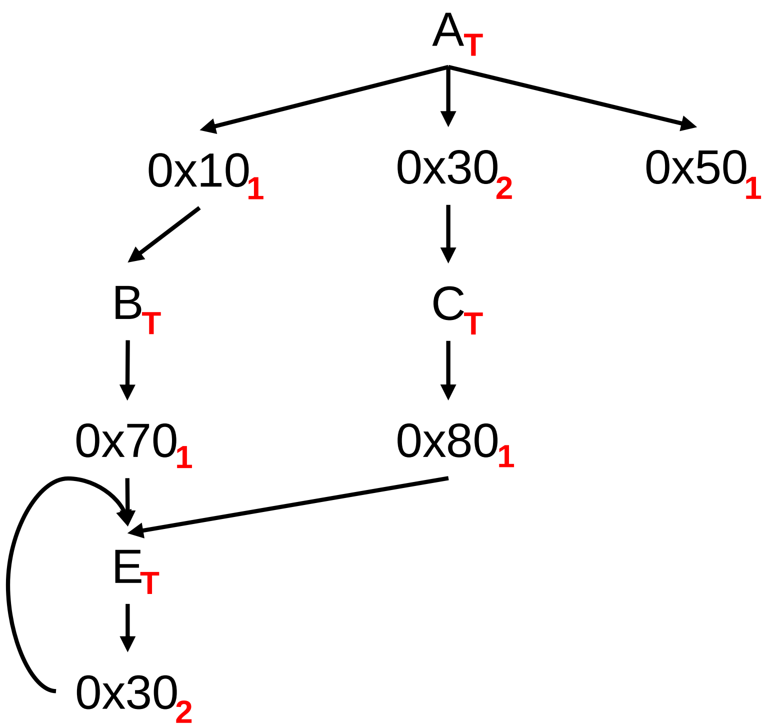
Step 2: Construct Dynamic Call Graph

- Propagate call instructions
  - At the same time change function marks
  - Implemented with a queue



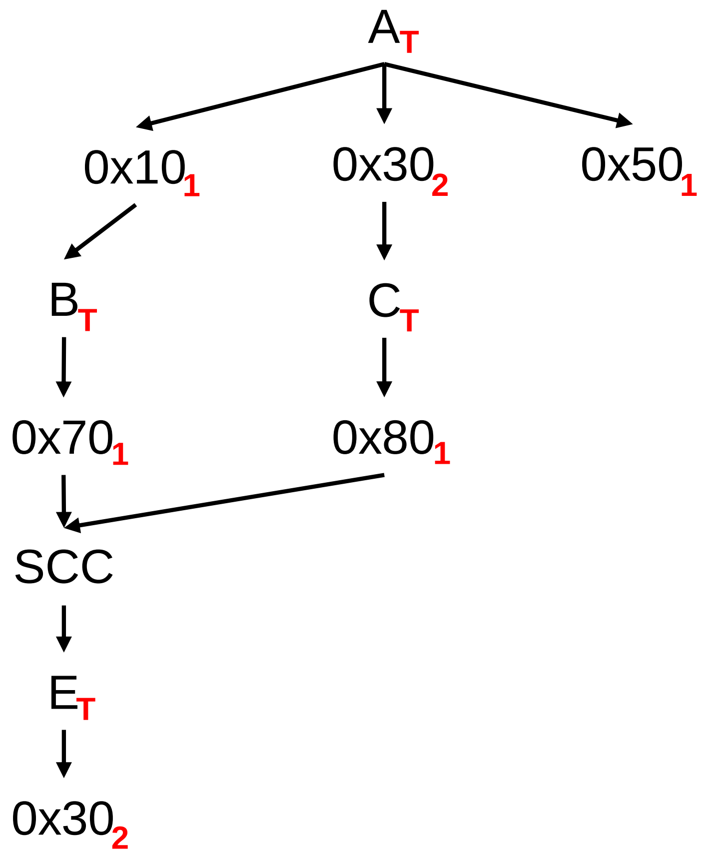
Step 2: Construct Dynamic Call Graph

- Prune functions with no samples or calls
- Keep call instructions



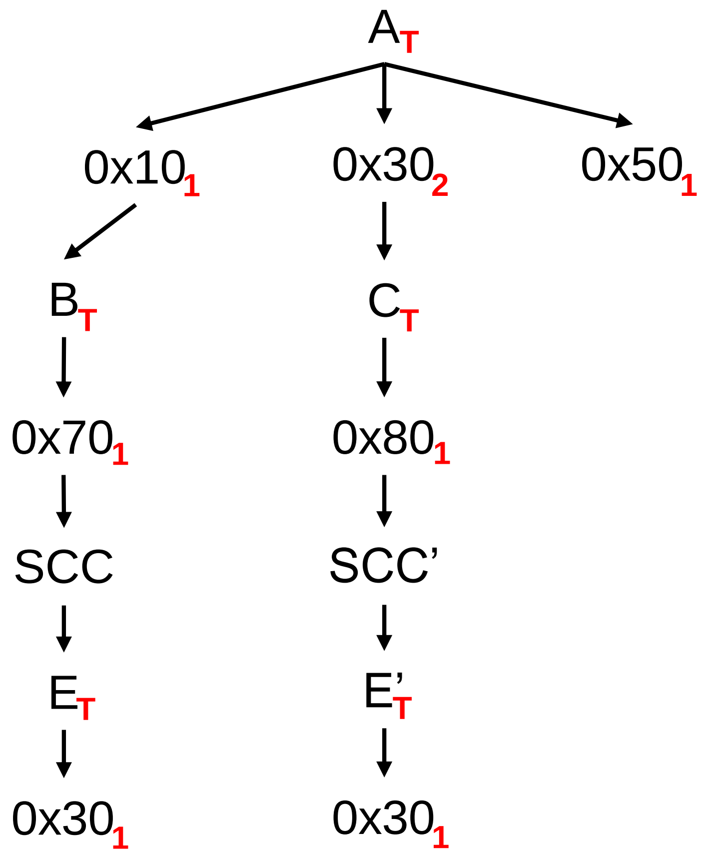
Step 3: Identify Recursive Calls

- Identify SCCs in call graph
- Link external calls to SCCs and unlink calls inside SCCs



Step 4: Transform Call Graph to Calling Context Tree

- Apportion each function's samples based on samples of its incoming call sites



Instruction Mixes

- Map opcodes and modifiers to instruction classes
- Memory ops
  - class.[memory hierarchy].width
- Compute ops
  - class.[precision].[tensor].width
- Control ops
  - class.control.type
- …

Metrics Approximation

- Problem
  - PC sampling cannot be used in the same pass with CUPTI Metric API or PerfWork API
  - Nsight-compute runs 47 passes to collect all metrics for a small kernel
- Solution
  - Derive metrics using PC sampling and other activity records
  - E.g. instruction throughput, scheduler issue rate, SM active ratio

Experiments

- Setup
  - Summit compute node: Power9+Volta V100
  - hpctoolkit/master-gpu
  - cuda/10.1.105
- Case Studies
  - Laghos
  - Nekbone

Laghos-CUDA

- Pinpoint performance problems in profile view by *importance metrics*
  - CPU takes 80% execution time
    - *mfem::LinearForm::Assemble* only has CPU code, taking 60% execution time
  - Memory copies can be optimized by different methods based on their calling context
    - Use memory copy counts and bytes to determine if using pinned memory with help
    - Eliminate conditional memory copies
    - Fuse memory copies into kernel code

Laghos-CUDA

- Original result: 32.9s
  - 11.3s on GPU computation and memory copies
- Optimization result: 30.9s
  - 9.0s on GPU computation and memory copies
- Overall improvement: 6.4%
- GPU code section improvement: 25.6%

Laghos-RAJA

- Pinpoint synchronization
  - Kernel launch in CUDA is asynchronous, but Laghos uses RAJA synchronous kernel launch
  - Use asynchronous RAJA kernel launch
- Bad compiler generated code with RAJA template wrapper
  - rMassMultAdd<3,4>: RAJA version has 4x STG instructions as the CUDA version. ¼ STG instructions within a loop use the same address.
  - Store temporary values in local variables

Laghos-RAJA

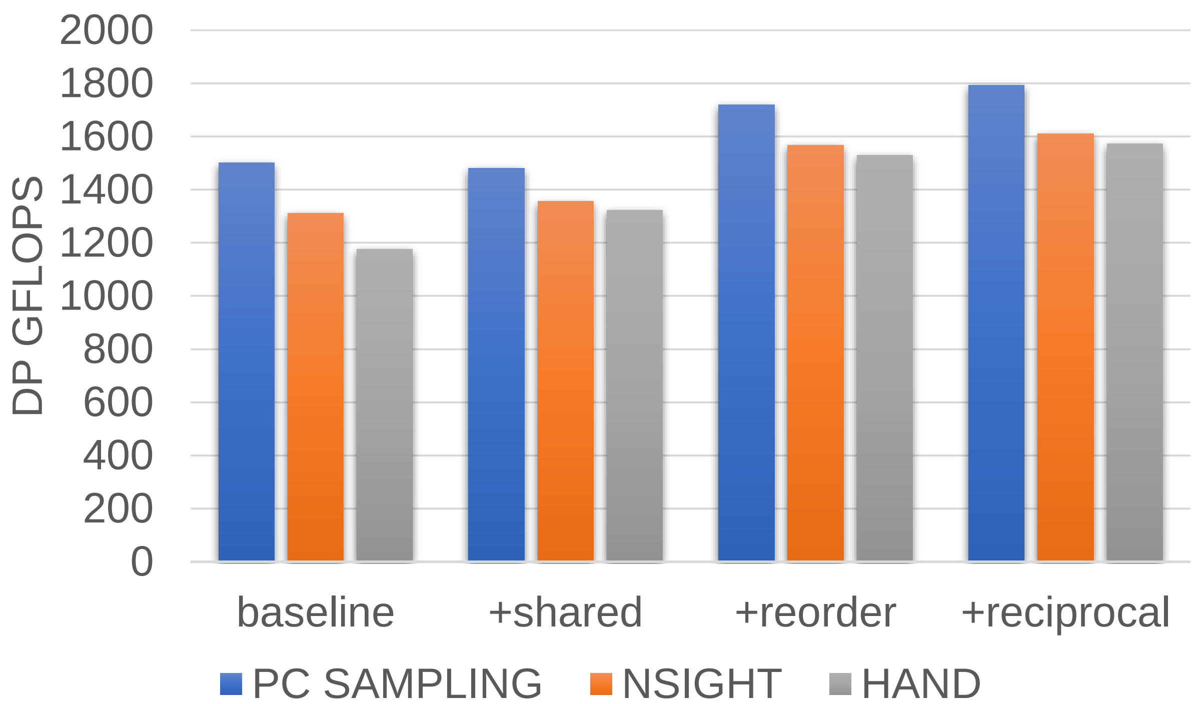
- Original result: 41.0s
  - 19.47s on GPU computation and memory copies
- Optimization result: 32.2s
  - 10.8s on GPU computation and memory copies
- Overall improvement: 27.3%
- GPU code section improvement: 80.2%

Nekbone

- Use PC sampling to associate stall reasons and instruction mixes with GPU calling context, loops, and lines
- Problems and optimizations
  - **Memory throttling**: high frequency global memory requests do not always hit cache. +*shared memory*
  - **Memory dependency**: compiler (-O3) does not reorder global memory read properly to hide latency. +*reorder global memory read*
  - **Execution dependency**: complicated assembly code for integer division. +*precompute reciprocal to simplify division*

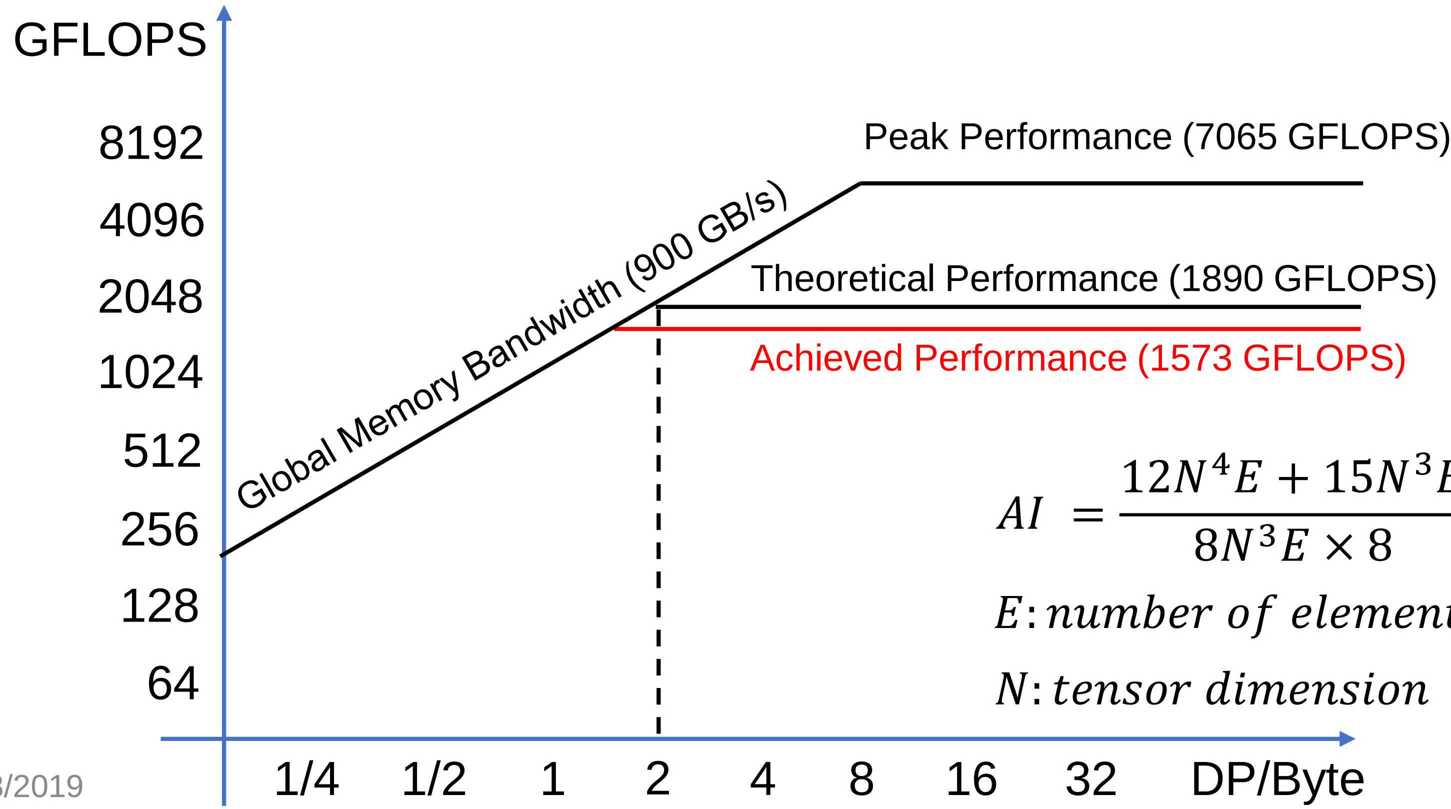
Nekbone Optimizations and Predictions

- Optimization result: +34%
- Prediction errors: the first one +26% because of predicates; others are within +13%



Nekbone Roofline Analysis

- 83% of peak performance
  - Could obtain +19% by fusing multiply and add on the assembly level



$$AI = \frac{12N^4E + 15N^3E}{8N^3E \times 8} \approx 2.1$$

$E: number\ of\ elements$

$N: tensor\ dimension$

Summary

- HPCToolkit pinpoints performance problems for both large-scale applications and individual kernels
- HPCToolkit provides insights for finding problems in compiler-generated GPU code, resource usage, synchronization, parallelism level, instruction pipeline, and memory access patterns
- HPCToolkit collects measurement data efficiently
  - Without PC sampling: comparable with nvprof
  - With PC sampling: 6x speedup

Next Steps

- Build an intelligent performance advisor that advises on specific lines and variables, choosing principal metrics that impact performance
- Study synchronization costs in MPI-OpenMP-CUDA hybrid programs