HPSF
HIGH PERFORMANCE
SOFTWARE FOUNDATION

Efficiently Representing CPU-GPU Performance

July 7, 2025

Jonathon Anderson
Rice University

Overview

- Novel enhancements to performance data representation
  - Same data, but in a smaller size
- Adjustments to improve parallelism in post-mortem analysis
  - Same high-level structure, but written faster
- Ongoing work

Examples: Calling Context Tree (CCT)

HPSF
HIGH PERFORMANCE
SOFTWARE FOUNDATION

```
0.008 main
├─ 0.000 MPI_Barrier
├─ 0.000 MPI_Reduce
└─ 0.007 lulesh.cycle
   ├─ 0.008 LagrangeLeapFrog
   │  ├─ 0.105 CalcTimeConstraintsForElems
   │  ├─ 0.011 LagrangeElements
   │  │  ├─ 0.028 ApplyMaterialPropertiesForElems
   │  │  │  └─ 0.430 EvalEOSForElems
   │  │  │     └─ 1.695 CalcEnergyForElems
   │  │  ├─ 0.014 CalcLagrangeElements
   │  │  │  └─ 0.072 CalcKinematicsForElems
   │  │  └─ 0.060 CalcQForElems
   │  │     └─ 0.052 CalcMonotonicQForElems
   │  └─ 0.324 LagrangeNodal
   │     └─ 0.013 CalcForceForNodes
   │        └─ 0.025 CalcVolumeForceForElems
   │           ├─ 0.162 CalcHourglassControlForElems
   │           │  └─ 0.261 CalcFBHourglassForceForElems
   │           └─ 0.246 IntegrateStressForElems
   └─ 0.004 TimeIncrement
      └─ 0.003 MPI_Allreduce
```

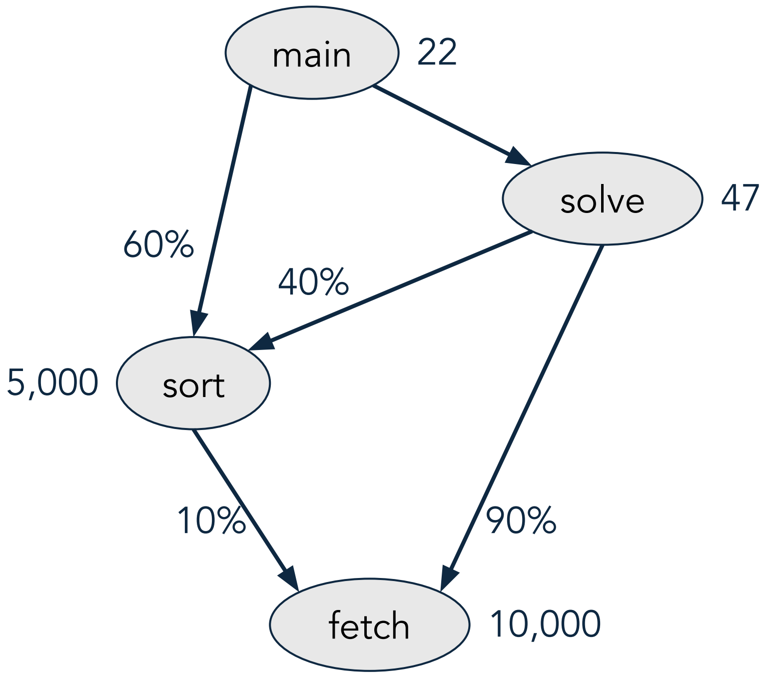
Image credit: LLNL/Hatchet

Examples: Flat Sampling / Flat Vector

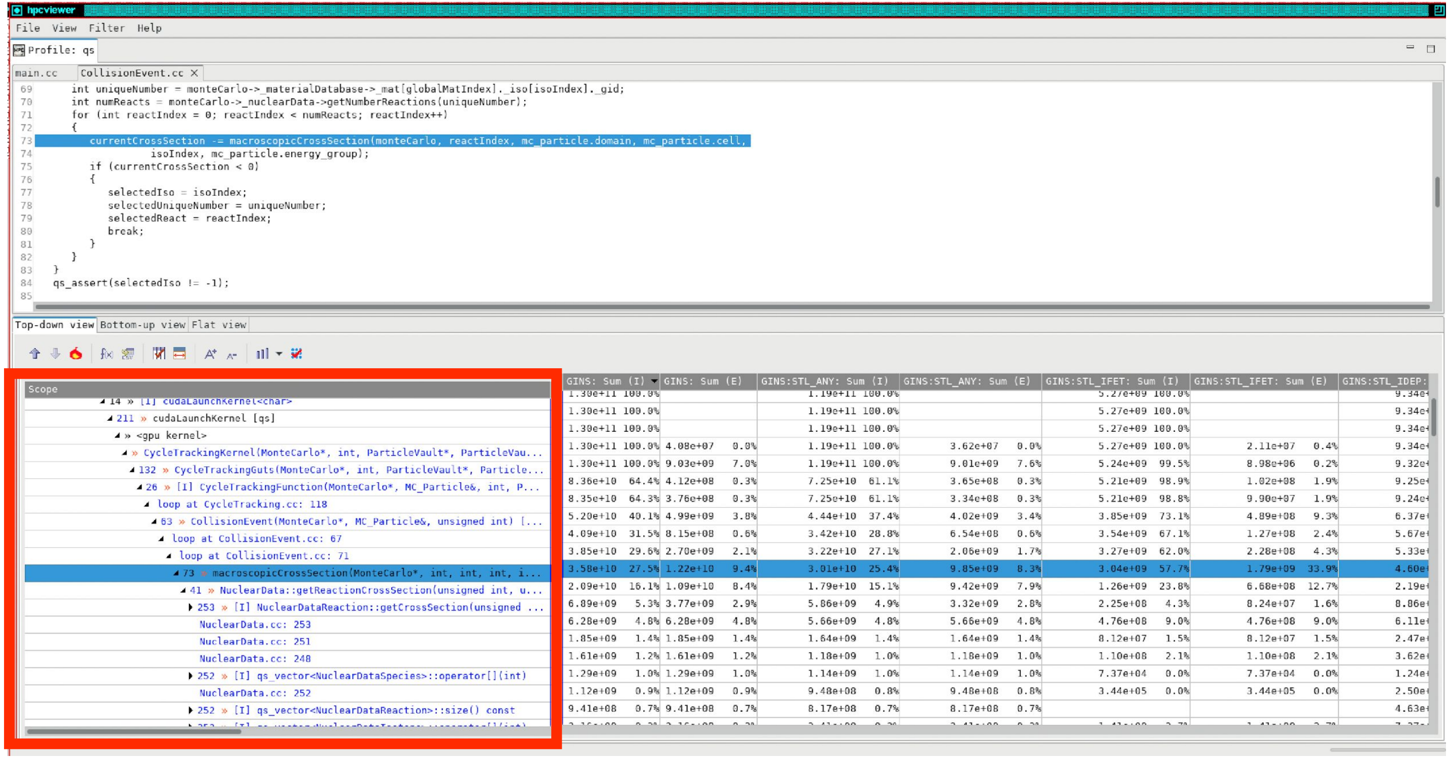
#	Address	Source	Live Registers	mp_sample_count	Sampling Data (All)	Sampling I (Not Issu
22	00007f72 42fa6a50	IADD3 R5, R3, 0x1, RZ	5	40	40	
23	00007f72 42fa6a60	MOV R20, 0x8	6	40	40	
24	00007f72 42fa6a70	IMAD R21, R3, R2, R0	7	16	16	
25	00007f72 42fa6a80	IMAD R5, R2, R5, R0	7	32	32	
26	00007f72 42fa6a90	IMAD R3, R3, R2, -R2	7	34	34	
27	00007f72 42fa6aa0	IMAD.WIDE R16, R21, R20, c[0x0][0x1a0]	8	32	32	
28	00007f72 42fa6ab0	IADD3 R3, R0, R3, RZ	8	46	46	
29	00007f72 42fa6ac0	IMAD.WIDE R10, R5, R20, c[0x0][0x1a0]	9	19	19	
30	00007f72 42fa6ad0	IMAD.WIDE R18, R3, R20, c[0x0][0x1a8]	11	83	83	
31	00007f72 42fa6ae0	LDG.E.64.CONSTANT.SYS R16, [R16]	10	44	44	
32	00007f72 42fa6af0	IMAD.WIDE R26, R21, R20, c[0x0][0x198]	12	44	44	
33	00007f72 42fa6b00	LDG.E.64.CONSTANT.SYS R10, [R10]	12	41	41	
34	00007f72 42fa6b10	IMAD.WIDE R28, R21, R20, c[0x0][0x1a8]	14	38	38	
35	00007f72 42fa6b20	LDG.E.64.CONSTANT.SYS R18, [R18]	14	41	41	
36	00007f72 42fa6b30	IMAD.WIDE R12, R5, R20, c[0x0][0x1a8]	16	52	52	
37	00007f72 42fa6b40	LDG.E.64.CONSTANT.SYS R14, [R26]	17	42	42	
38	00007f72 42fa6b50	LDG.E.64.CONSTANT.SYS R6, [R26+0x8]	19	100	100	
39	00007f72 42fa6b60	LDG.E.64.CONSTANT.SYS R2, [R28+-0x8]	19	137	137	
40	00007f72 42fa6b70	LDG.E.64.CONSTANT.SYS R12, [R12]	19	180	180	
41	00007f72 42fa6b80	LDG.E.64.CONSTANT.SYS R8, [R28+0x8]	21	115	115	
42	00007f72 42fa6b90	LDG.E.64.CONSTANT.SYS R4, [R28]	23	99	99	
43	00007f72 42fa6ba0	IMAD.WIDE R20, R21, R20, c[0x0][0x190]	21	20	20	
44	00007f72 42fa6bb0	LDG.E.64.SYS R22, [R20]	23	177	177	
45	00007f72 42fa6bc0	DADD R24, R16, R10	25	22,607	22,607	21
46	00007f72 42fa6bd0	DMUL R18, R16, R18	25	1,821	1,821	1
47	00007f72 42fa6be0	DADD R24, R24, 1	23	2,683	2,683	2
48	00007f72 42fa6bf0	DADD R16, R14, R6	25	1,320	1,320	1
49	00007f72 42fa6c00	DMUL R2, R14, R2	25	1,338	1,338	1
50	00007f72 42fa6c10	DADD R16, R24, R16	23	1,279	1,279	1
51	00007f72 42fa6c20	DFMA R12, R10, R12, R18	21	427	427	
52	00007f72 42fa6c30	DFMA R2, R6, R8, R2	17	423	423	
53	00007f72 42fa6c40	DFMA R4, R16, R4, -R12	13	2,151	2,151	1

Image credit: Nvidia

Examples: Call Graph

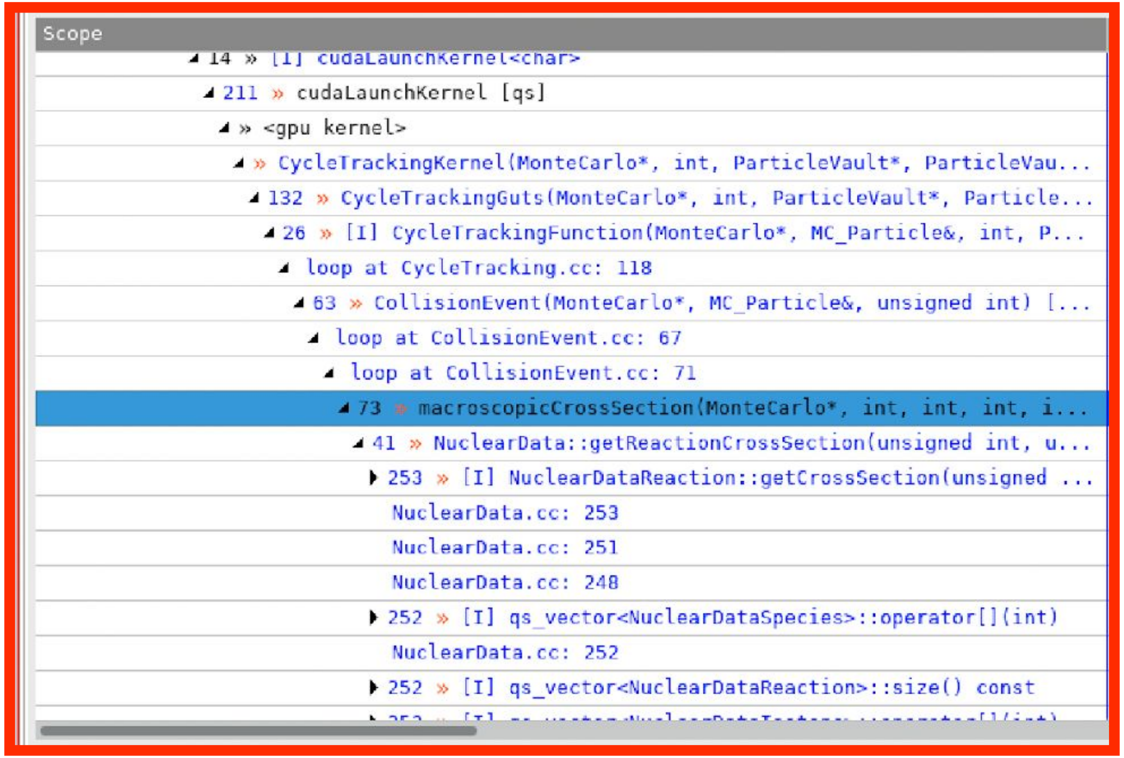


HPCToolkit's Instruction-Level Attribution within GPU Kernels (CCT)



HPCToolkit's Instruction-Level Attribution within GPU Kernels (CCT)

HPSF HIGH PERFORMANCE SOFTWARE FOUNDATION

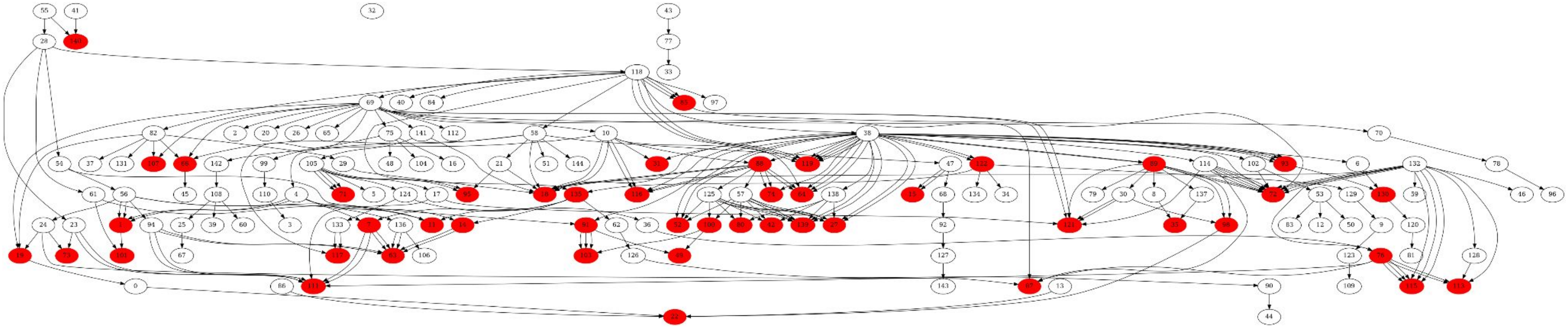


HPCToolkit's Measurement Approach

- HPCToolkit uses different representations (for measurement) depending on the performance measured
  - CPU performance: attribute samples to full calling context (CCT)
    - Application stack is unwound at runtime
  - GPU performance: attribute flat samples to instructions within each GPU kernel
    - Limitation of the GPU vendor runtime APIs
    - AMD GPU PC samples: correlation ids enable mapping to full CPU calling context of kernel launch
    - Nvidia & Intel GPU PC samples: can't be correlated to CPU calling context without a performance hit
      - GPU APIs don't provide correlation id; correlation accomplished by serializing all GPU kernels
- HPCToolkit's post-mortem analysis converts each of the representations to CCT

HPCToolkit's CCT-based Attribution

- Flat samples are apportioned across plausible call paths
  - Attributed to static call graph, then expanded to CCT
  - If a node has multiple callers (red), it's cost is distributed based on call counts
  - CCT contains multiple nodes for each GPU function, one for each plausible caller
    - …Recursively to cover all plausible call paths



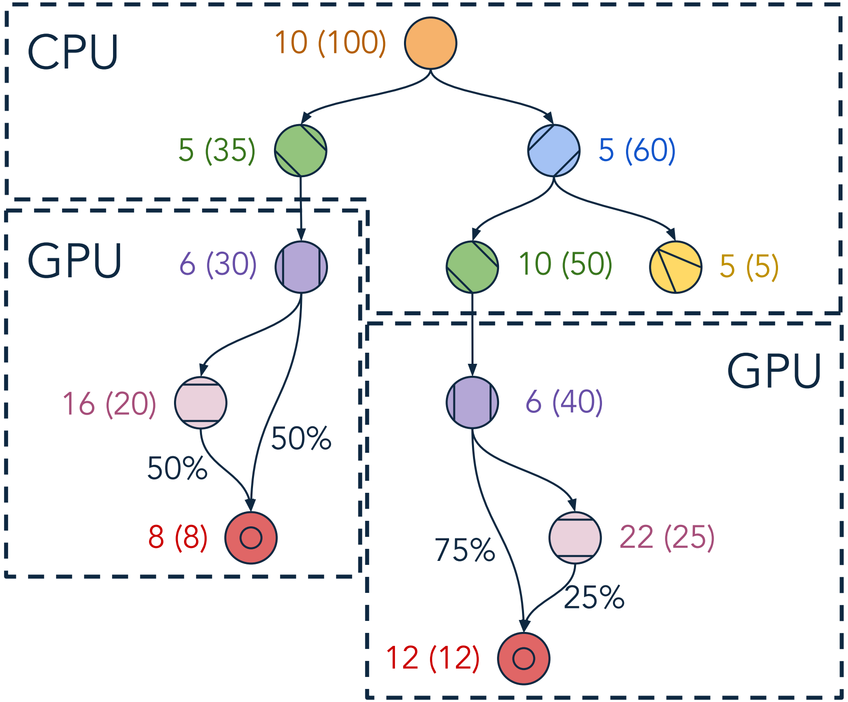
Call graph for one GPU kernel of Quicksilver (w/o optimization)

Shortcomings of CCT-based Attribution

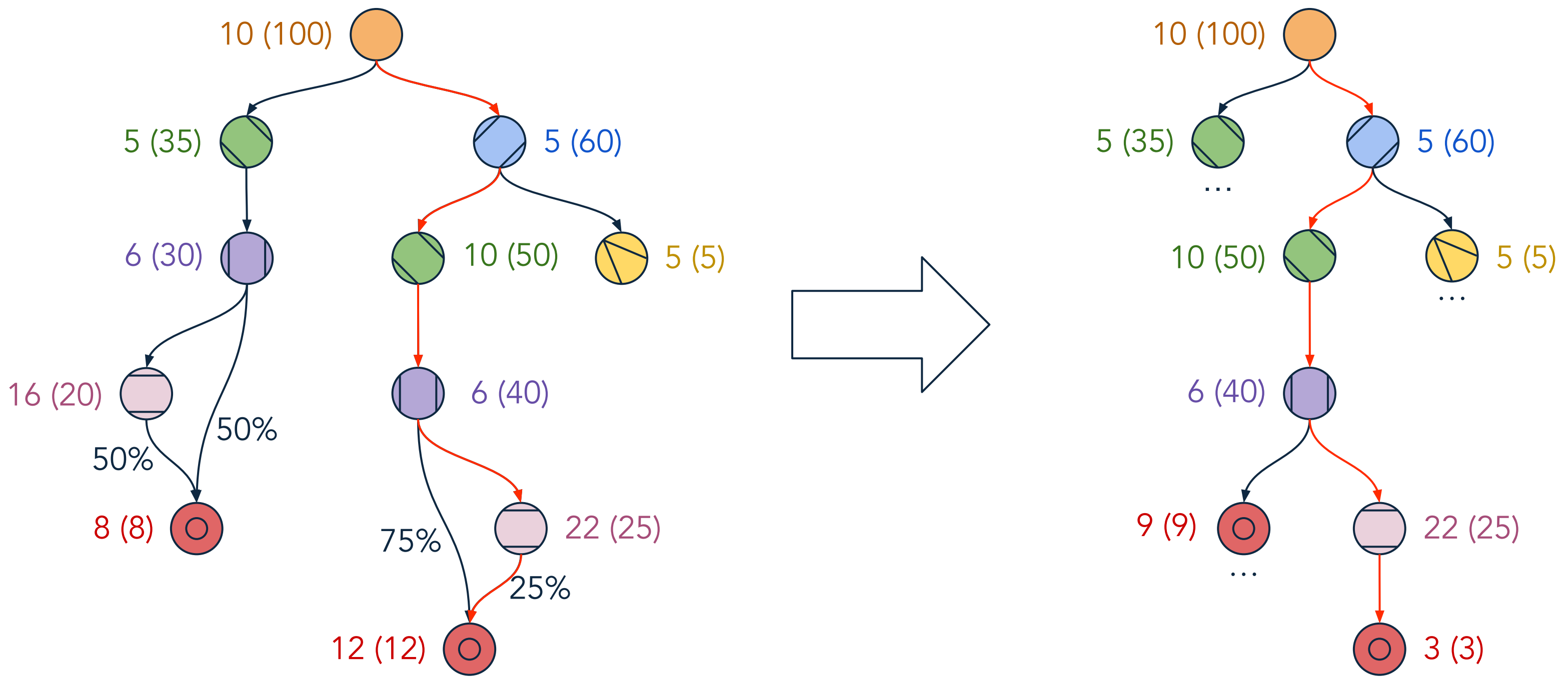
- Mercury: Monte Carlo transport app developed at LLNL
  - ~2K functions in one GPU kernel
  - ~500K call paths to a single, widely-used leaf function
  - One function called cuda_div_… >100 times
- In practice, HPCToolkit fails to construct GPU CCTs for Mercury
  - Unfeasibly slow post-mortem analysis, reconstructs contexts for only 1 instruction per second!
  - Why? CCT explodes in size during analysis!
    - Need to distribute performance ~500K ways, for each instruction in one function
    - → Need a better data representation that won't explode

Novel Graph-based Data Representation

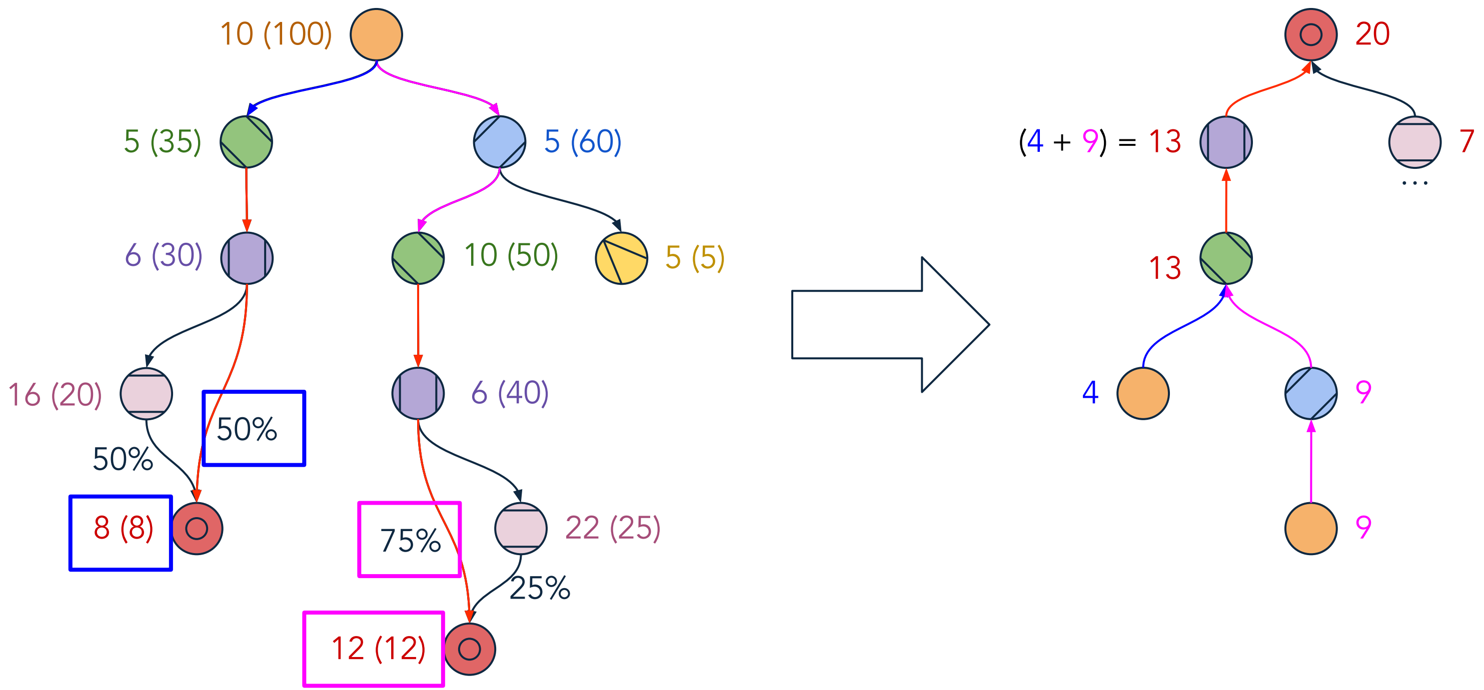
- Directed (Calling) Context Graph
  - Generalization of other representations
  - Nodes = unique measured contexts
  - Edges = control flow (e.g. calls)
  - Edge weights = apportion between callers
  - Node values = X (Y)
    - X = Node exclusive cost
    - Y = Node inclusive cost
- Efficient for both CPU & GPU performance
  - …But with no visible distinction between the two



Top-down Analysis Example



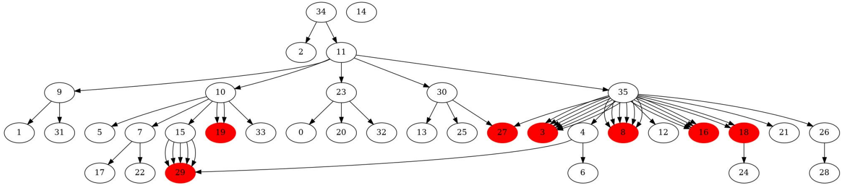
Bottom-up Analysis Example



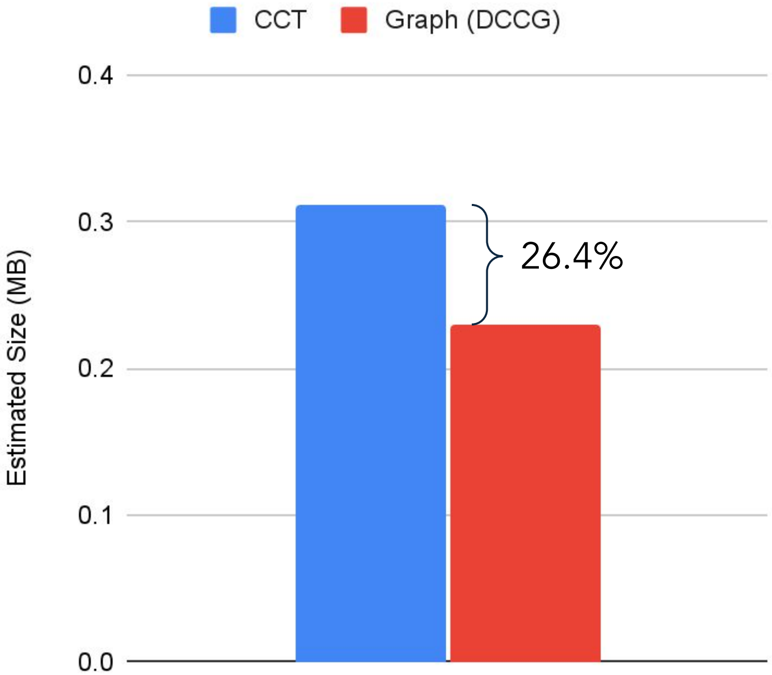
Estimated Size Reduction

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

- Quicksilver: proxy app for Mercury
  - 36 functions (nodes in the DCCG)
  - 58 nodes in the static CCT
- Estimated size reduction: 37.9%
  (Assuming uniform instruction count per function)



Static call graph of one GPU kernel

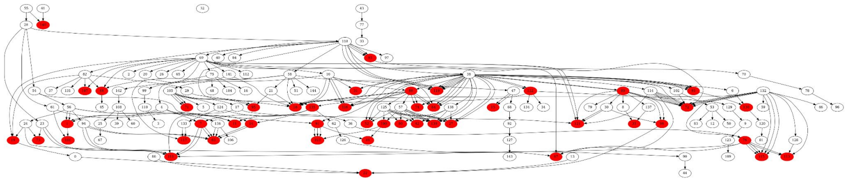


(Estimate using actual instruction count per function)

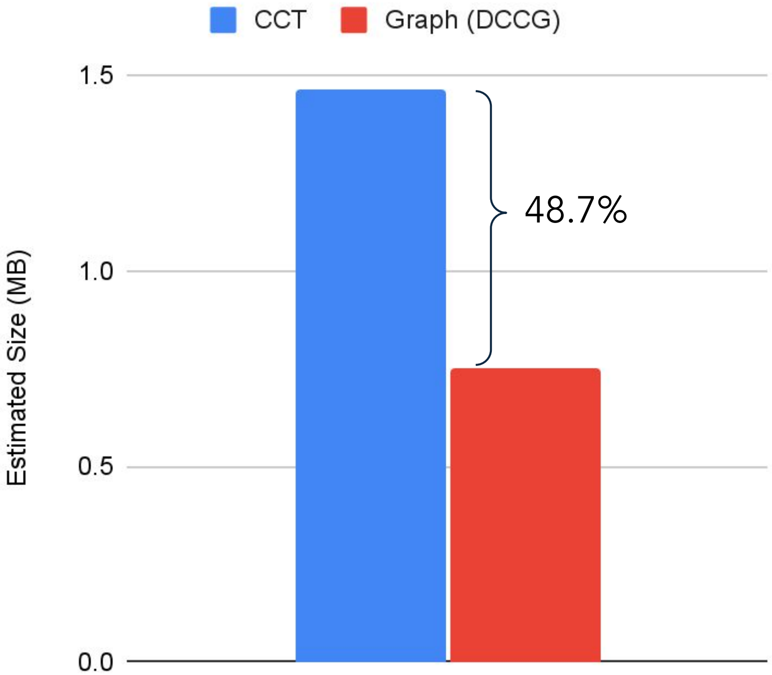
Estimated Size Reduction

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

- Quicksilver w/o optimization
  - 145 functions (nodes in the DCCG)
  - 409 nodes in the static CCT
- Estimated size reduction: 64.5%
  (Assuming uniform instruction count per function)



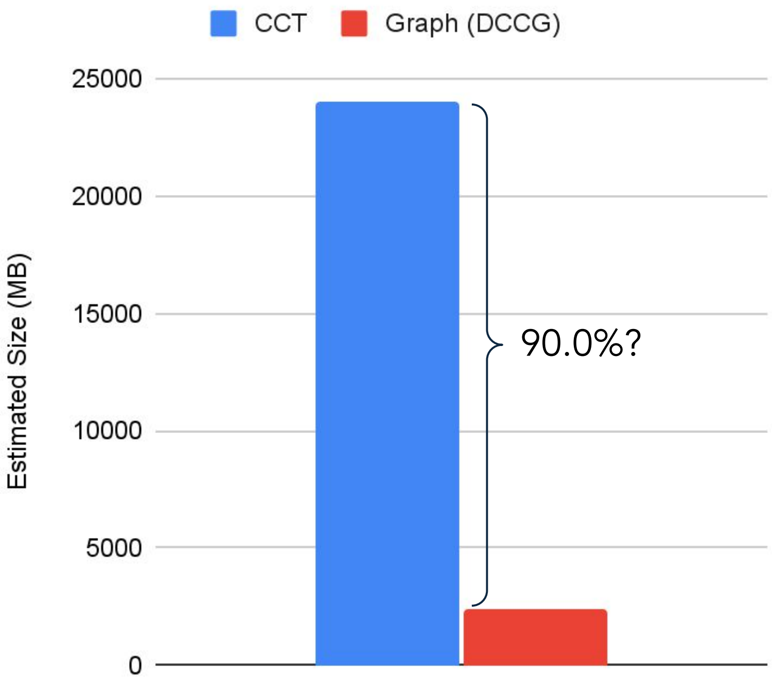
Static call graph of one GPU kernel



(Estimate using actual instruction count per function)

Estimated Size Reduction

- Mercury
  - ~2K functions (nodes in the DCCG)
  - ~500K nodes in the static CCT (for one function)
- Estimated size reduction: 99.6%?
  (Estimate using rough guess based on limited info)

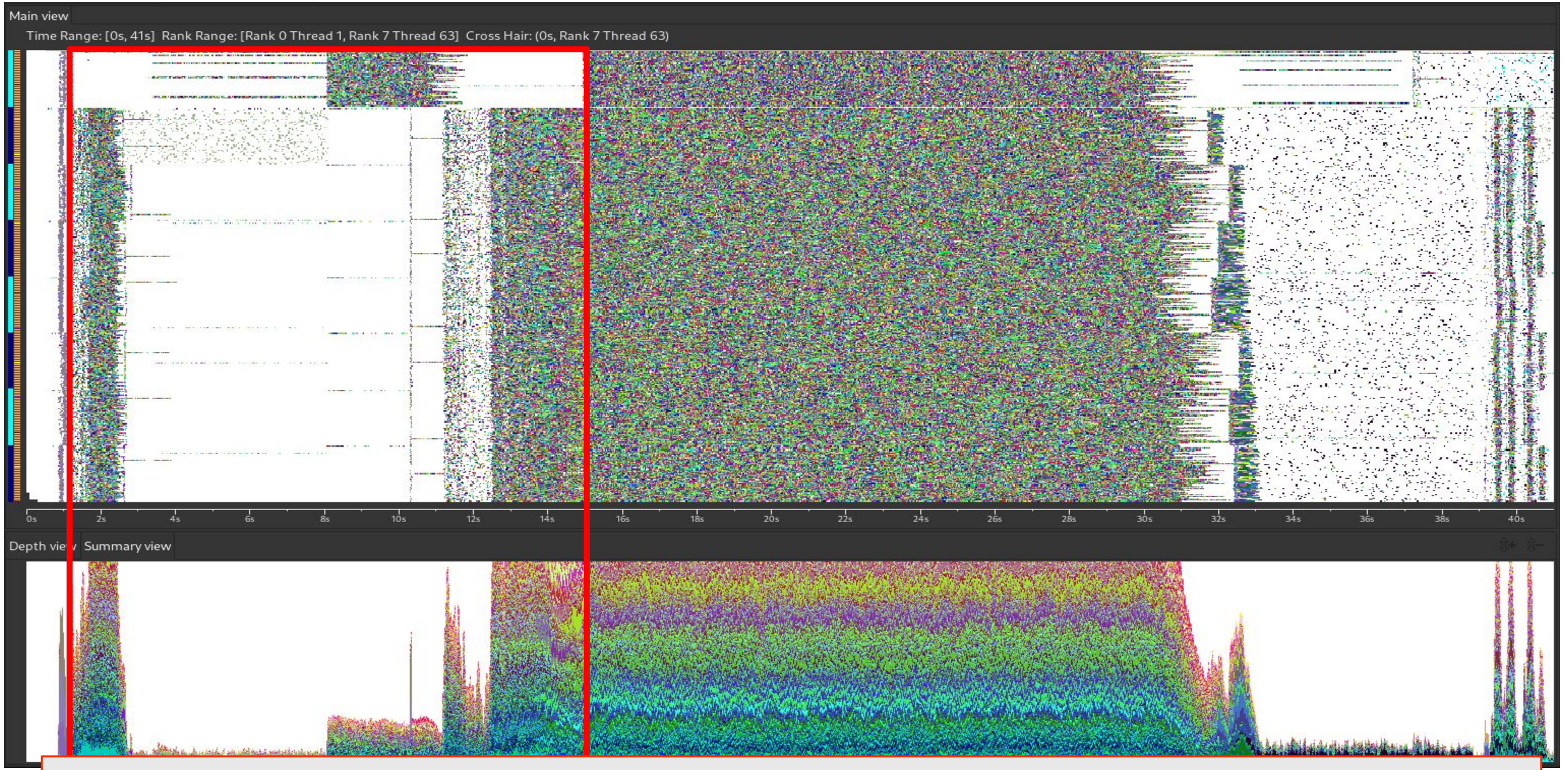


(Complete guess based on left estimate)

Overview

- Novel enhancements to performance data representation
  - Same data, but in a smaller size
- Adjustments to improve parallelism in post-mortem analysis
  - Same high-level structure, but written faster
- Ongoing work

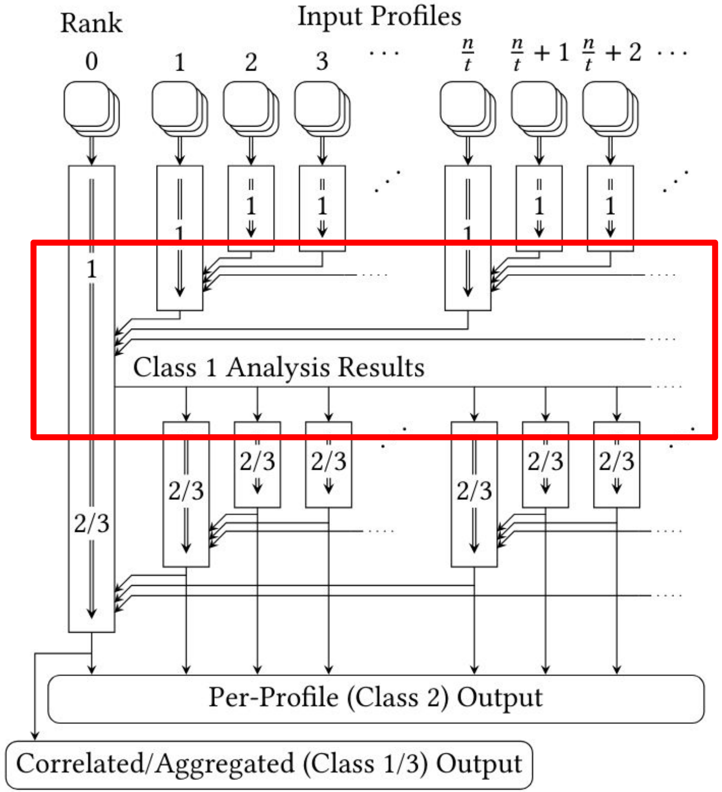
Parallel Bottleneck Of hpcprof-mpi



Unify ~2K CCTs across 8 nodes using 8 MPI ranks x 64 threads

Removing Communication

- Class 1: unique identifiers
  - Mostly for CCT nodes
  - All ranks use the same id to refer to a CCT node
- CCT nodes are unified based on “unique” info
  - Caller node, instruction offset, source line, etc.
  - Only merge nodes that match exactly
- We only need consensus… what about hashes?
  - Same context → same hash in all MPI ranks
  - Hashing local data is much faster than MPI synchronization/communication
- …But only useful if hashes don’t collide



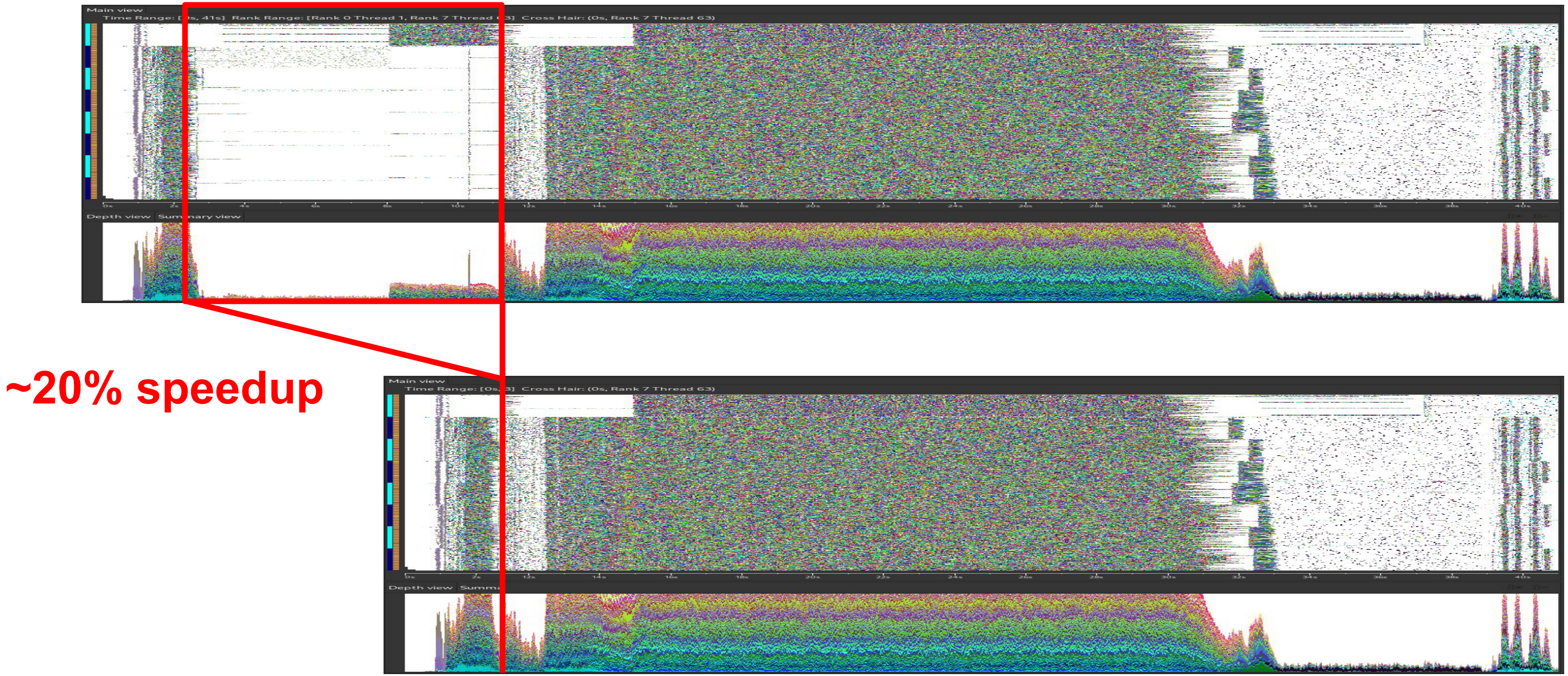
Effective Identifiers Using Hashes

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

- We can avoid hash conflict resolution in practice, assuming:
    - Maximum limit of ~4 billion calling contexts
        - Based on HPCToolkit's current 32-bit identifiers
    - Each node is a hash of "its (unique) location in the graph"
- 128-bit hashes (statistically) never conflict (w/ >99.9999% probability)
    - Used for communication and validation during post-mortem analysis
- 64-bit hashes are expected to have a small number of conflicts
    - Used to identify each graph node and edge on-disk
    - Disambiguate contexts at read time using graph traversal
- For performance, always leave hash conflicts unresolved
    - If a conflict causes validation errors, just restart the process (<0.0000001% chance)
    - Otherwise leave it be, disambiguate at read time

Est. Speedup of Post-mortem Analysis

HPSF
HIGH PERFORMANCE
SOFTWARE FOUNDATION



Overview

- Novel enhancements to performance data representation
  - Same data, but in a smaller size
- Adjustments to improve parallelism in post-mortem analysis
  - Same high-level structure, but written faster
- Ongoing work

HPCToolkit’s Performance Atlas

- Next evolution of the HPCToolkit database format
- Development is ongoing
    - Atlas library implementation: ~60%
    - Integration with hpcprof: <5%
    - Integration with hpcviewer: 0%
- Will be available as a separate library
    - First-party support for external clients

Recap

HPSF HIGH PERFORMANCE SOFTWARE FOUNDATION

- Novel enhancements to performance data representation
  - New graph-based representation for performance data
  - Expect significantly smaller than CCT-based representation
- Adjustments to improve parallelism in post-mortem analysis
  - Hash-based identifiers to avoid MPI communication during analysis
  - Expect ~20% speedup over MPI consensus algorithm
- Atlas library implementation is ongoing
  - Making steady progress towards initial working implementation
  - Will be available as a separate library to ease adoption

Backup Slides

Determining Hash Sizes

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

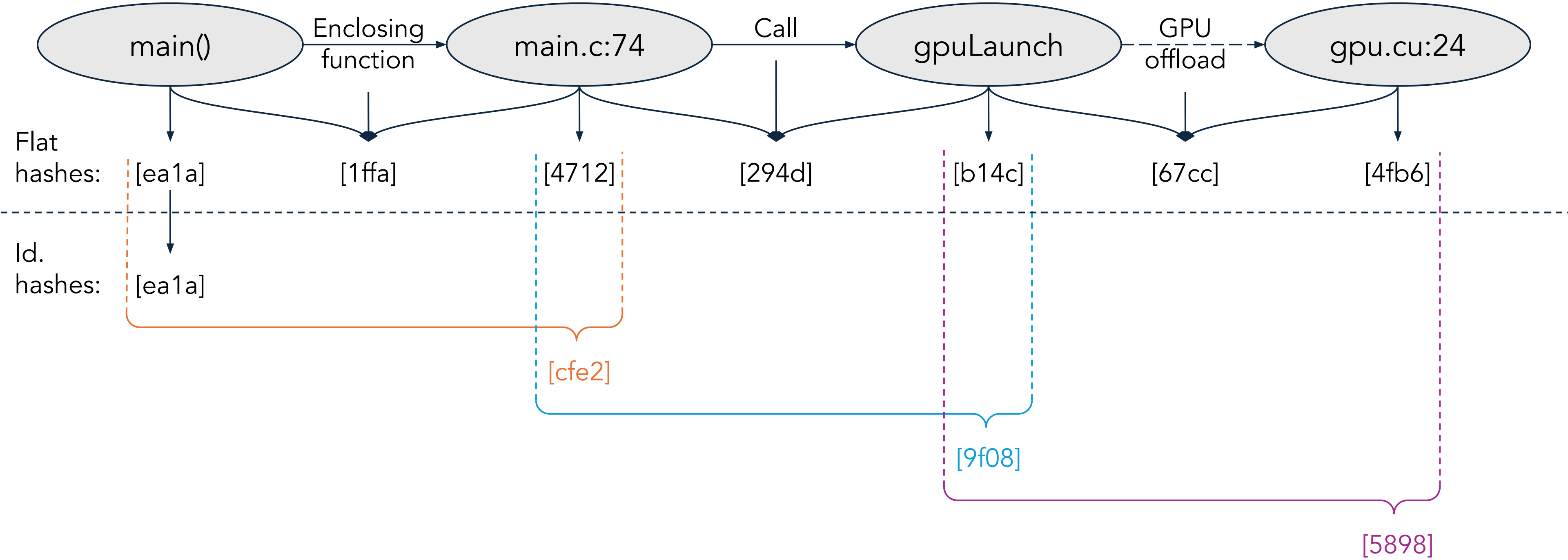
- 128-bit hashes are sufficient to identify 2^{32} items uniquely with >99.9999% probability:
  a. Actually only need 83 bits but round up to 128 for technical convenience
  b. Need at least 32 bits for 2^{32} distinct hash values
  c. With 32×2 bits, probability of collision among 2^{32} items $P \approx 50\% = 2^{-1}$
     - From approximate solution to the generalized Birthday Problem
  d. Adding 19 bits multiplies collision probability by 2^{-19}, so collision probability is now 2^{-20}
     - Each extra bit reduces probability by ~½
  e. $2^{-20} < 0.0000001\%$, so probability of no collision is >99.9999%

`ff16d9b8 671ac65c 30bb8ae6 519d6fdb`

- `ff16d9b8`: 32 bits to store 2^{32} distinct hashes, many many hash collisions
- `671ac65c`: ×2 bits to reduce probability of collision to ~50%
- `30bb8`: +19 bits reduce probability to 2^{-20}
- `ae6 519d6fdb`: Rounded up to 128 bits for technical convenience

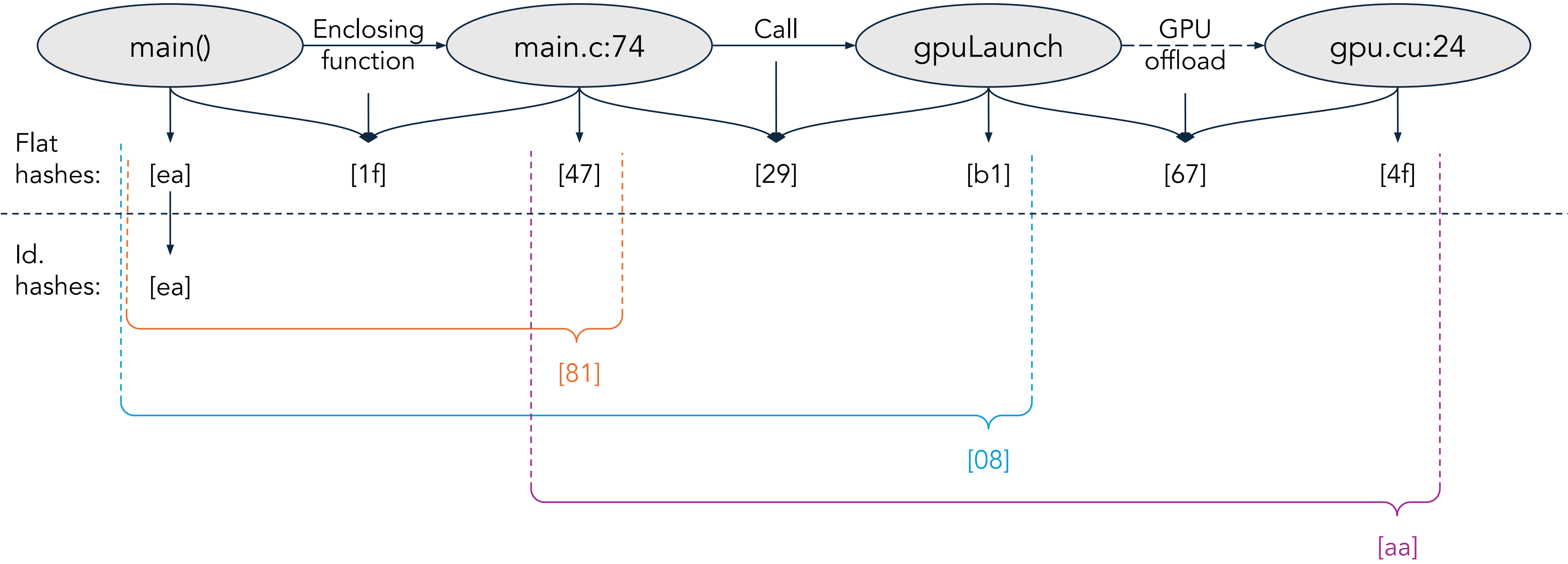
Identifier Hash Construction (128-bit)

- Identifier hash for a DCCG node is a Merkle hash combining ids for dominators + flat hashes of node/edge content
  - Dominators are from measurement data, e.g. CPU stack unwind, GPU launch site + instruction offset
  - Edges may not be in DCCG, instead artificial edges just for hash construction (e.g. GPU offload)



Identifier Hash Construction (64-bit)

- Identifier hash for a DCCG node is a Merkle hash combining ids for dominators + flat hashes of node/edge content
  - Dominators are from measurement data, e.g. CPU stack unwind, GPU launch site + instruction offset
  - Edges may not be in DCCG, instead artificial edges just for hash construction (e.g. GPU offload)



Atlas: Storage Backend-agnostic Design

- Atlas is formed out of “pages”
  - Independent contiguous data blobs
  - Structured data, serialized w/ FlatBuffers (zero-copy)
  - Pages can be stored together or separate
- Pages refer to each other via descriptors
  - Can be anything: file offset, index, content hash, etc.
  - Heavily inspired by OCI images, but more general
- (Will be) widely compatible across storage media
  - POSIX/Lustre file
  - DAOS/Rabbit object store
  - OCI registry
  - NoSQL/key-value database
  - …etc.