Diogenes: Looking For An Honest CPU/GPU Performance Measurement Tool

Benjamin Welton and Barton Miller

Supercomputing 2019, Denver, CO

November 19th 2019

GPUs are difficult to use

Some of the reasons why include:

- Writing of efficient GPU kernels
- Identification of code suitable for the GPU
- Handling CPU – GPU interactions (data transfers, synchronizations, etc).
- Integrating GPU code into existing CPU code

Very hard for a programmer to get all of these right

Programmers look for help

Turn to performance tools

- Existing tools, such as profilers and tracers, are good at identifying that there is a problem such as the following:
  - GPU is idle
  - CPU is spending time within a CUDA function(s)

Programmers look for help

Where performance tools fail is in identifying the cause of the problem:

- If the operation is required for program correctness (unnecessary synchronization, duplicate data transfers, etc) or placed correctly

Miss problematic behaviors entirely:

- The percentage of time a CUDA function is spending synchronizing vs performing the operation

Overview of Approach

Automatically detect performance issues with CPU-GPU interactions (synchronizations, memory transfers)

- Unnecessary interactions
- Misplaced interactions
- What we do not do:
  - GPU kernel profiling, general CPU/GPU profiling, etc

Output a list of unnecessary or misplaced interactions

- Including an estimate of potential benefit (in terms of application runtime) of fixing these issues.

Overview of Approach

Binary instrumentation of the application and CUDA user space driver for data collection

- Collect information not available from other methods
  - Use (or non-use) of data from the GPU by the CPU
  - Identify hidden interactions
    - Conditional/Implicit interactions (ex. a synchronous cuMemcpyAsync call).
    - Detect and measure interactions on the private API.
  - Directly measure synchronization time
  - Look at the contents of memory transfers

Analysis method to show only problematic interactions.

How problematic are these issues?

Our prior work on exposing problems in GPU applications showed that synchronization and transfer issues were prevalent in GPU applications today.

App Name	App Type	LOC	Original Runtime (Min:Sec)	Percent Reduction	Problems Found
Hoomd-Blue	MDS	112,000	08:36	37%	ES
Qbox	MDS	100,000	38:54	85%	DD, IS
LAMMPS	MDS	208,000	03:34	19%	MP
culBM	CFD	17,000	31:42	27%	IS, JT

MDS = Molecular Dynamics Simulation **CFD** = Computational Fluid Dynamics

ES = Explicit Synchronization, **IS** = Implicit Synchronization
MP = Missed Parallelization, **JT** = JIT Compilation, **DD** = Duplicate Data Transfers.

[Welton2018] Exposing Hidden Performance Opportunities in High Performance GPU Applications

Why do these performance issues exist?

- Large application code base increases the difficulty in identifying inefficient synchronization and memory copy operations.
- Large applications make locally optimal decisions that generate inefficiencies when combined.
- Use of highly optimized GPU libraries hiding synchronization operations and data locations.

The Gap In Performance Tools

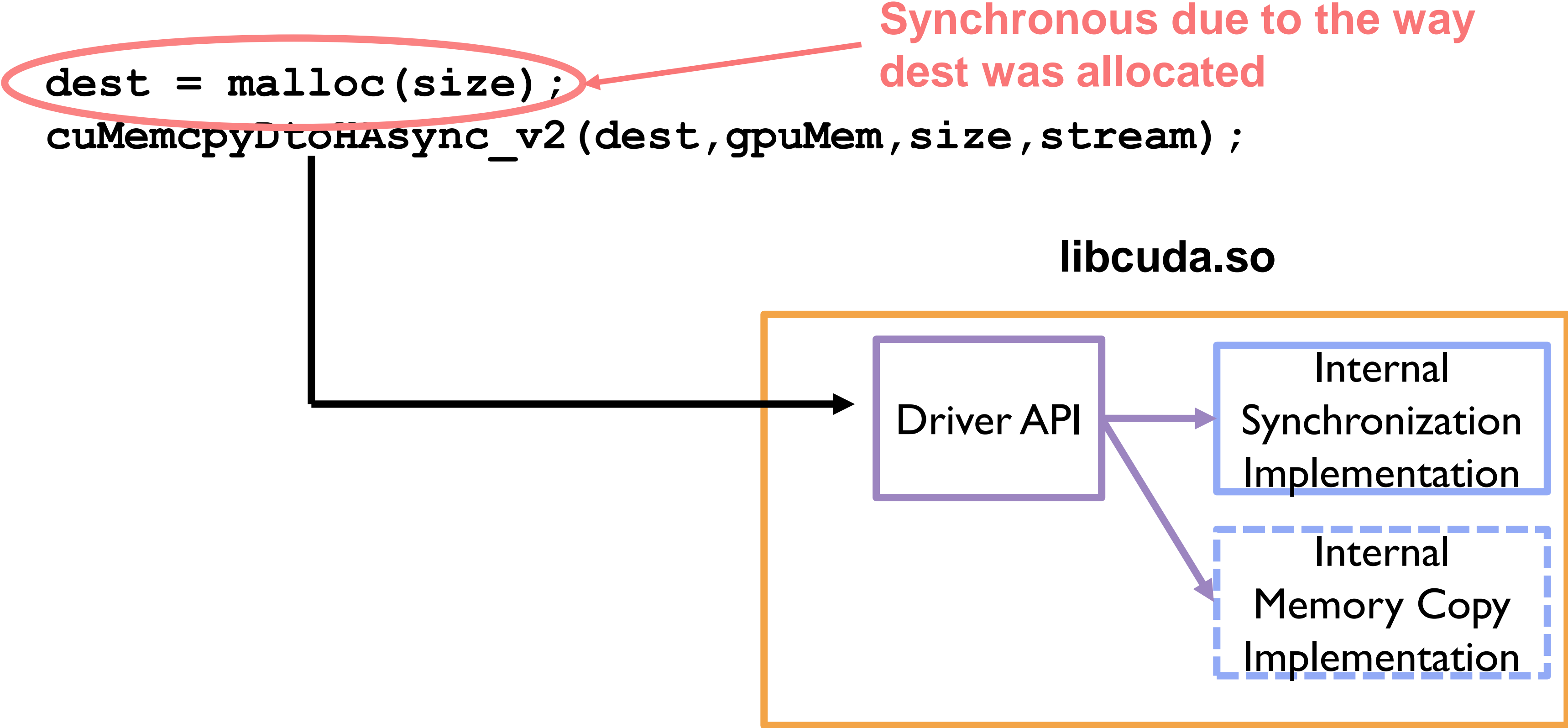
Existing Tools (CUPTI, etc) have collection and analysis gaps preventing detection of issues

- Don't collect performance data on hidden interactions
  - Conditional/Implicit Interactions
  - Private API calls
- Don't determine the necessity of interactions

Result – Interaction issues are not identified resulting in lost performance.

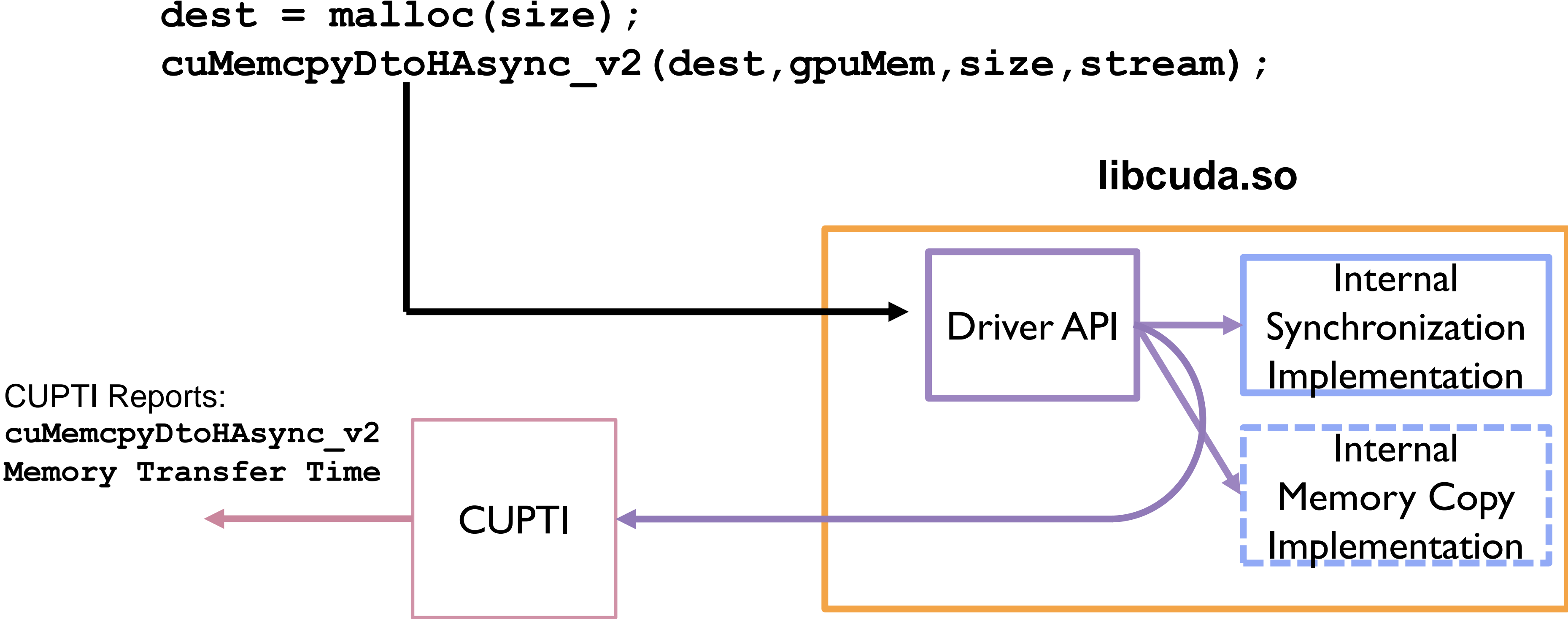
Conditional/Implicit Interaction

Conditional Interactions are unreported synchronizations performed by a CUDA call.



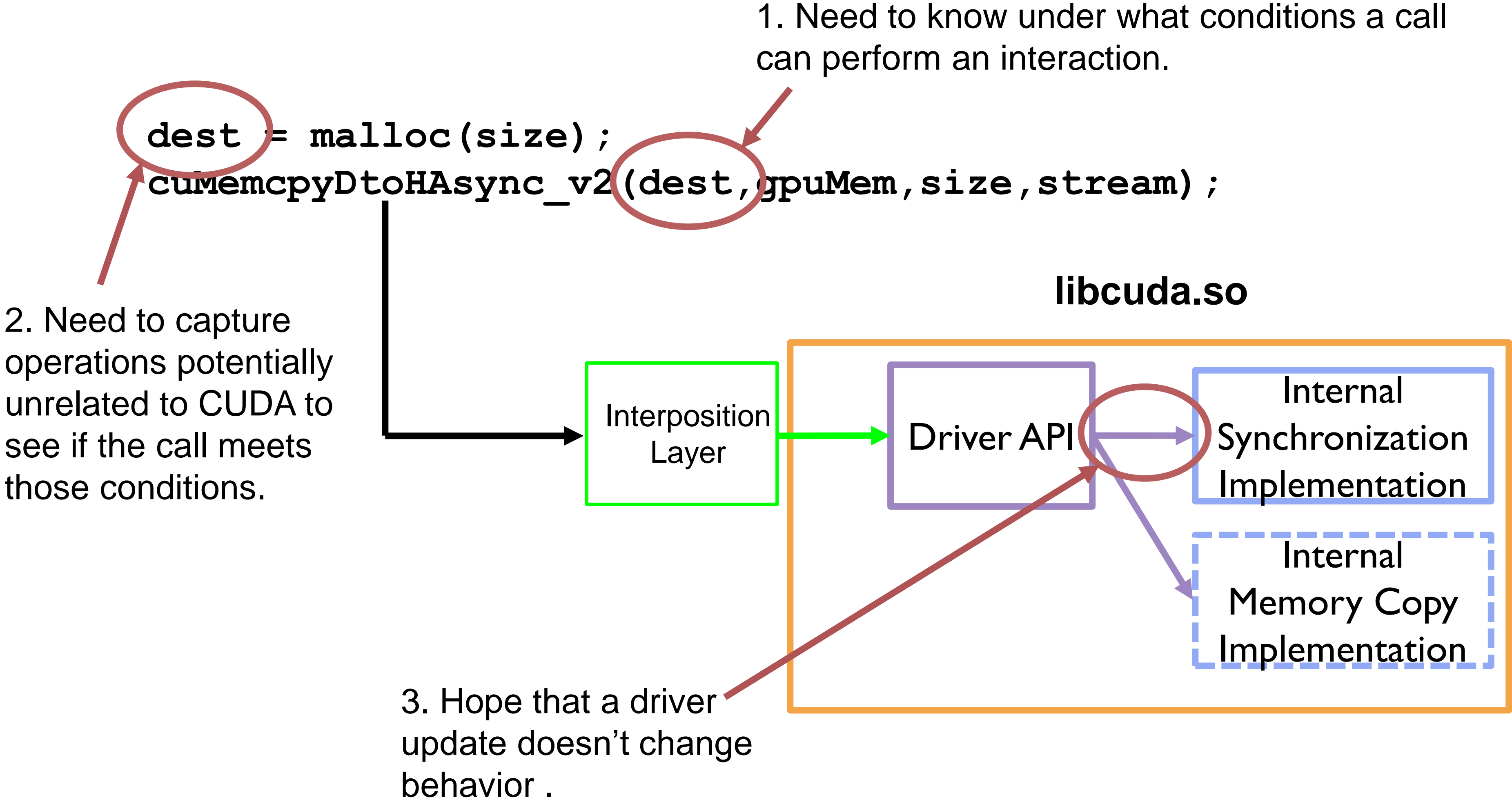
Conditional Interaction Collection Gap

CUPTI doesn't report when conditional/implicit interactions are performed by a call.



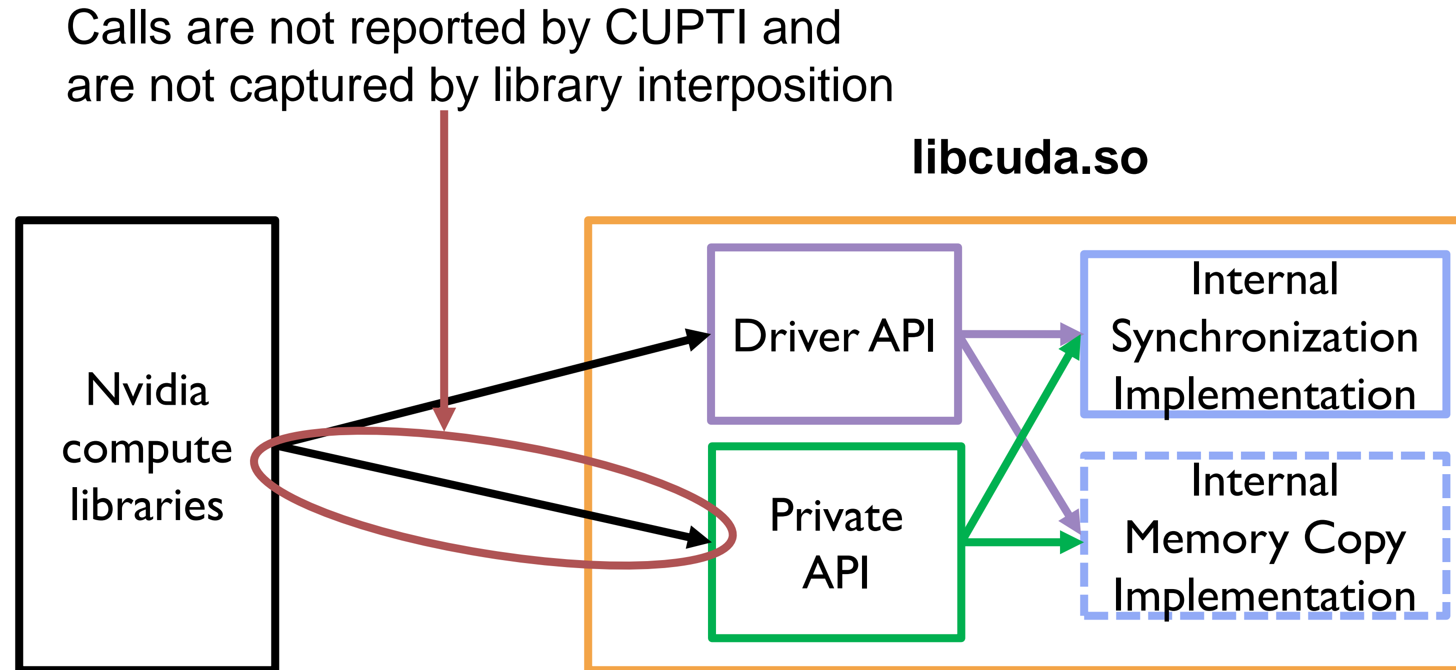
Conditional Interaction Collection Gap

Hard to detect with library interposition approaches due to:



The Private API

Large private API used by Nvidia compute libraries (cufft, cublas, cudnn, etc) which has all the capabilities of the public API (and many more).



Diogenes – Workflow

Diogenes uses a newly developed technique called feed forward instrumentation

- The results of previous instrumentation guides the insertion of new instrumentation.

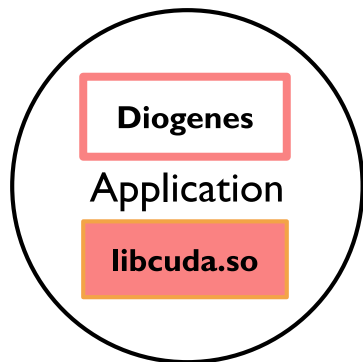
Diogenes performs each step automatically (via a launcher)

Diogenes – Workflow

Diogenes uses a newly developed technique called feed forward instrumentation

- The results of previous instrumentation guides the insertion of new instrumentation.

Step 1
Measure execution time and identify functions performing synchronizations

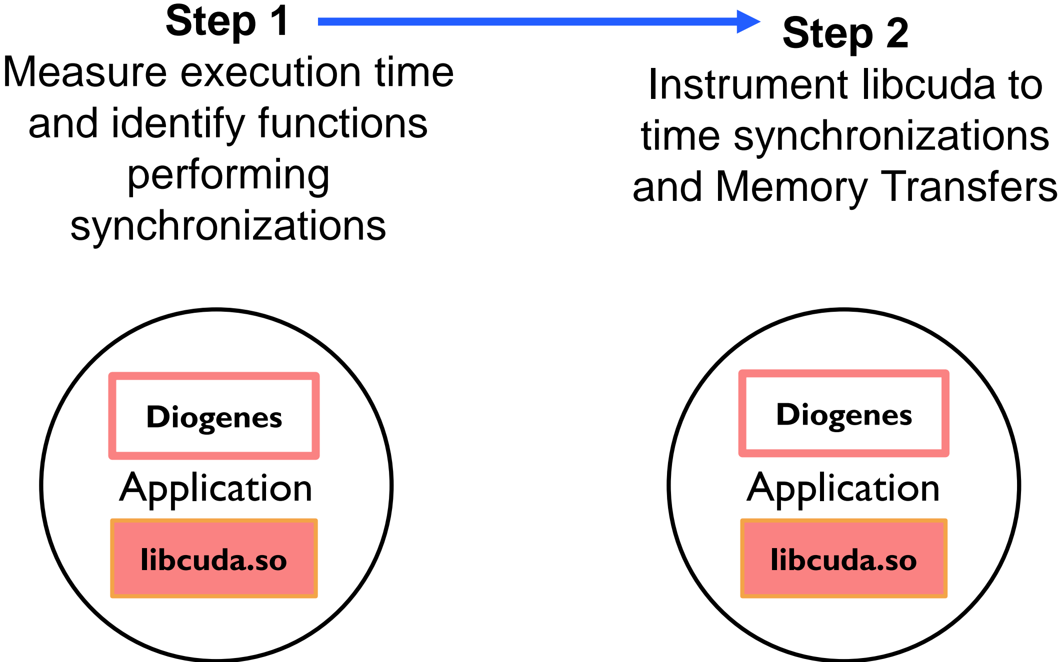


Diogenes performs each step automatically (via a launcher)

Diogenes – Workflow

Diogenes uses a newly developed technique called feed forward instrumentation

- The results of previous instrumentation guides the insertion of new instrumentation.

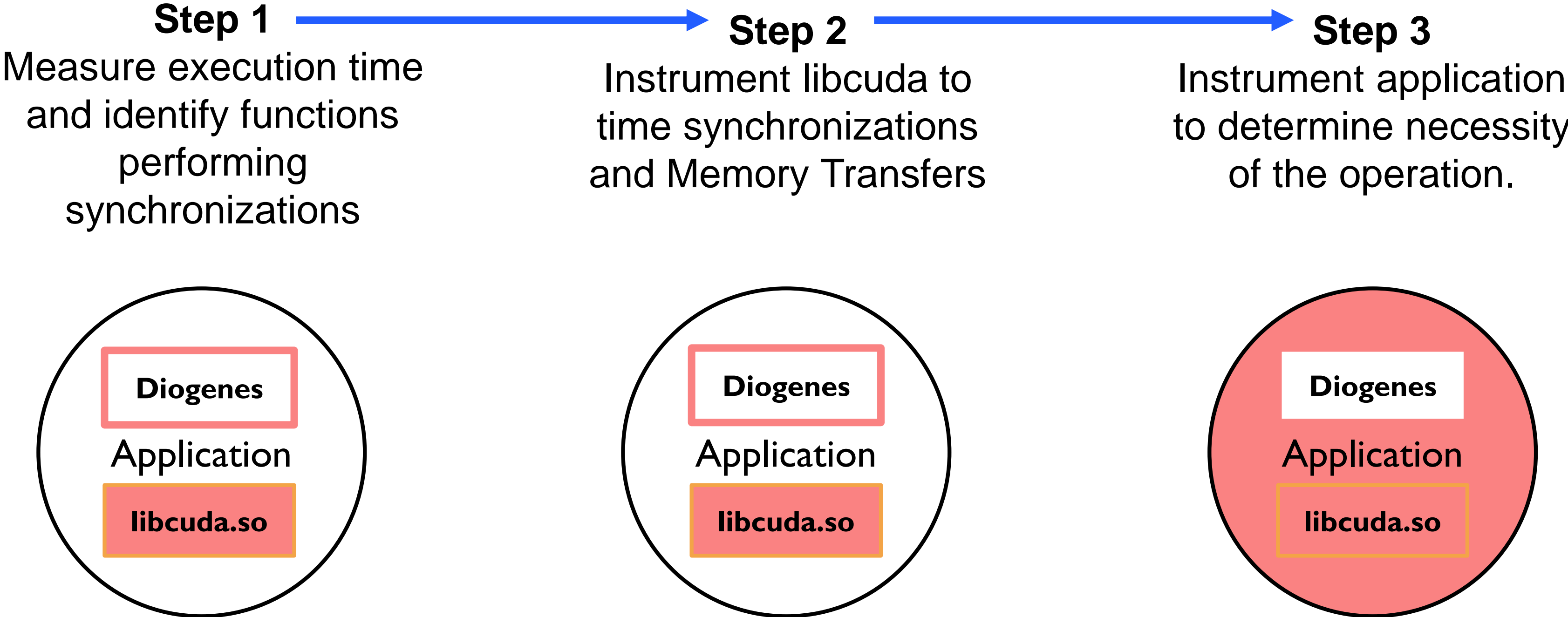


Diogenes performs each step automatically (via a launcher)

Diogenes – Workflow

Diogenes uses a newly developed technique called feed forward instrumentation

- The results of previous instrumentation guides the insertion of new instrumentation.

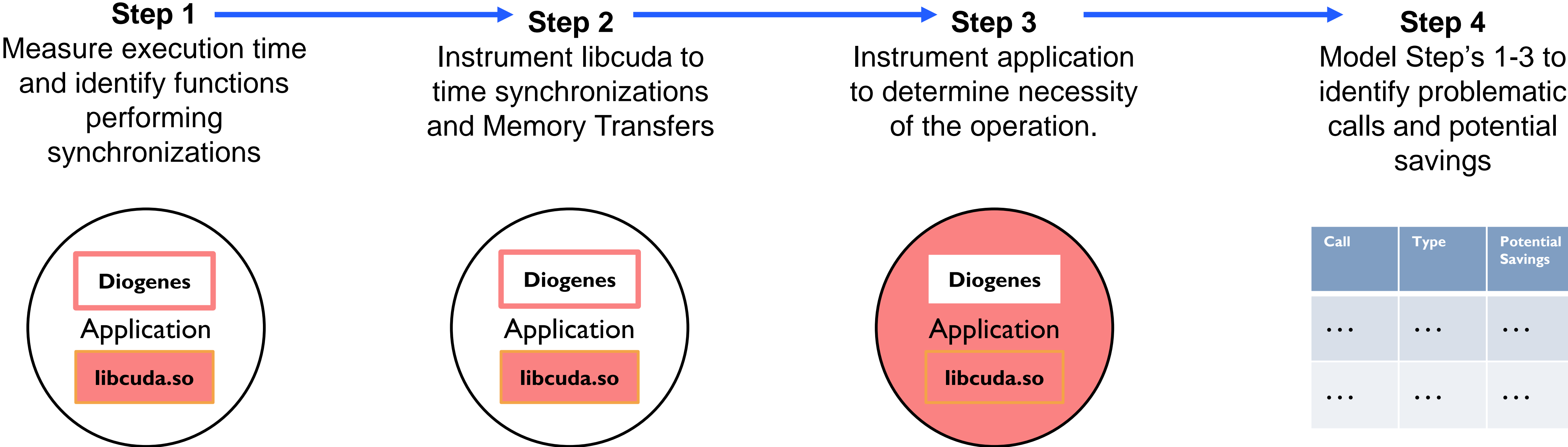


Diogenes performs each step automatically (via a launcher)

Diogenes – Workflow

Diogenes uses a newly developed technique called feed forward instrumentation

- The results of previous instrumentation guides the insertion of new instrumentation.



Diogenes performs each step automatically (via a launcher)

Step 1: Measure execution time and identify functions performing synchronizations

Capture (and store) application execution time

- Needed for modeling expected benefit

Identify functions performing synchronization operations

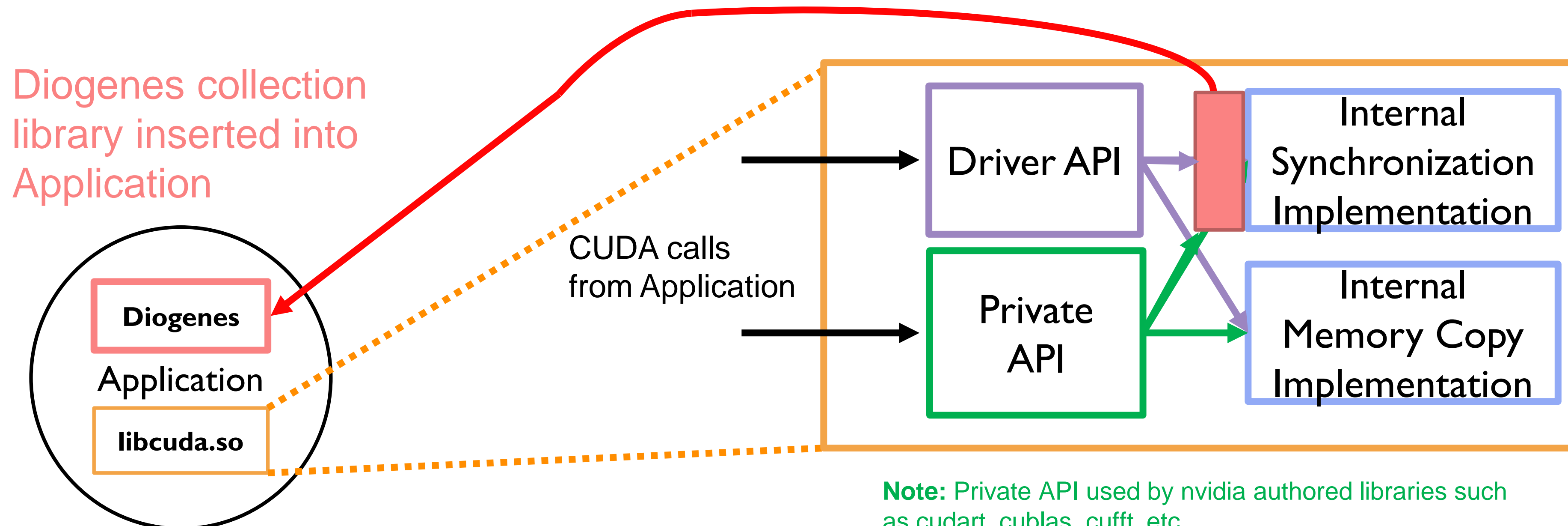
- Need to be captured in advance to know what functions to time in step 2
  - We don’t know before running the program what calls will be performing synchronizations

Capturing CUDA Synchronizations

Wrap the internal synchronization function to collect performance data hidden from other tools (CUPTI/Library Interposition)

- Time the synchronization delay directly
- Catch private API synchronization calls
- Captures conditional synchronization operations

Memory Transfer Wrapping is future work

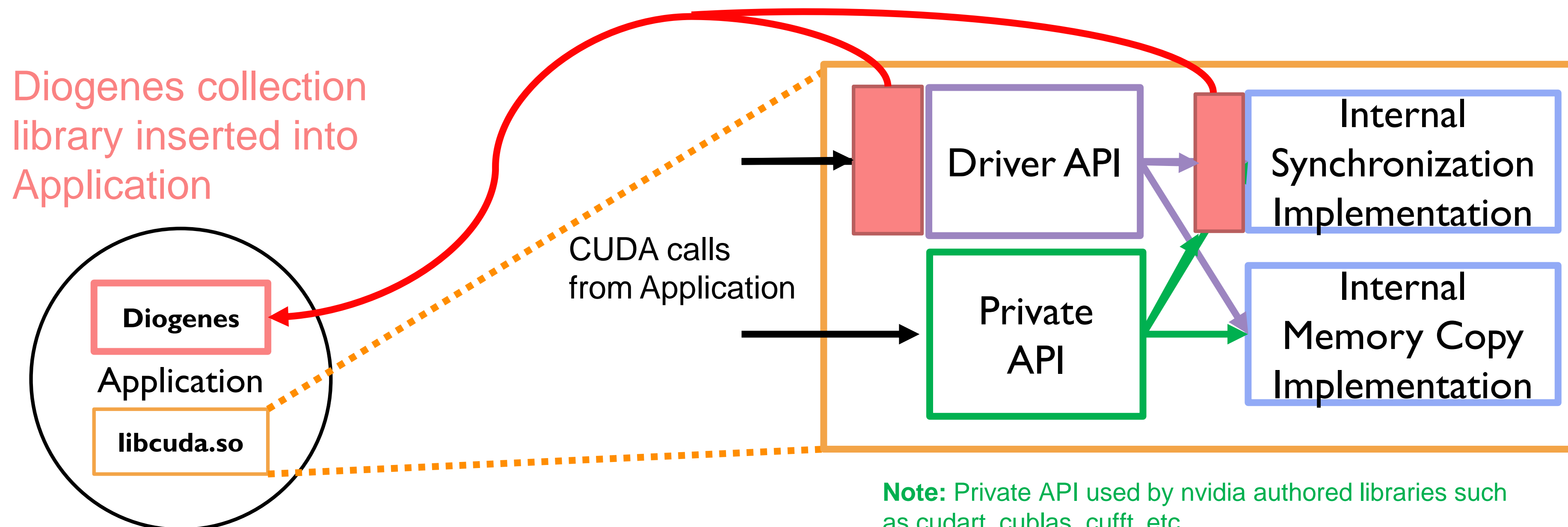


Note: Private API used by nvidia authored libraries such as cudart, cublas, cufft, etc

Step 2: Instrument libcuda to time Synchronizations and Memory Transfers

Wrap public API functions to performing synchronizations and memory transfers and record execution time

Wrap internal synchronization function and capture synchronization time.



Note: Private API used by nvidia authored libraries such as cudart, cublas, cufft, etc

Step 3: Instrument application to determine necessity of the operation.

Employs two detection techniques:

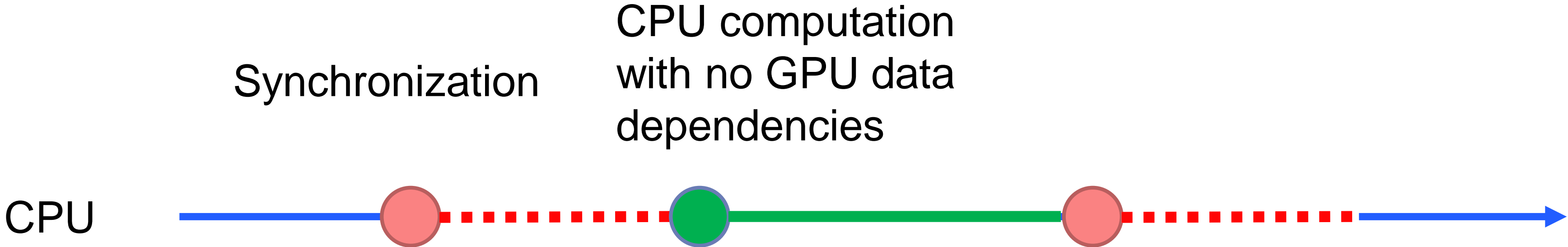
- Identification of unnecessary/misplaced synchronizations
- Identification of duplicate data transfers

Detection of Synchronization Opportunities

1. When the CPU does not access shared data after the synchronization

CPU Example

```
Synchronization();
for(…) {
   // Work with no GPU dependencies
}
Synchronization();
```

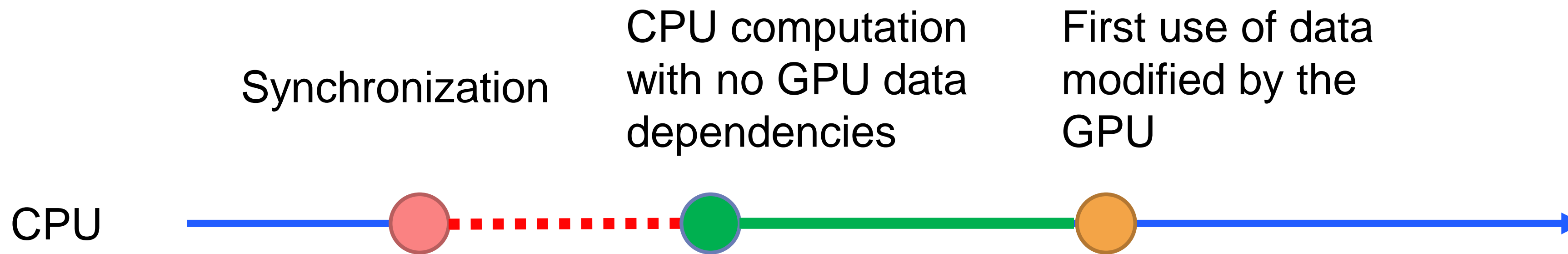


Detection of Synchronization Opportunities

2. When the placement of the synchronization is far from the first access of shared data by the CPU

CPU Example

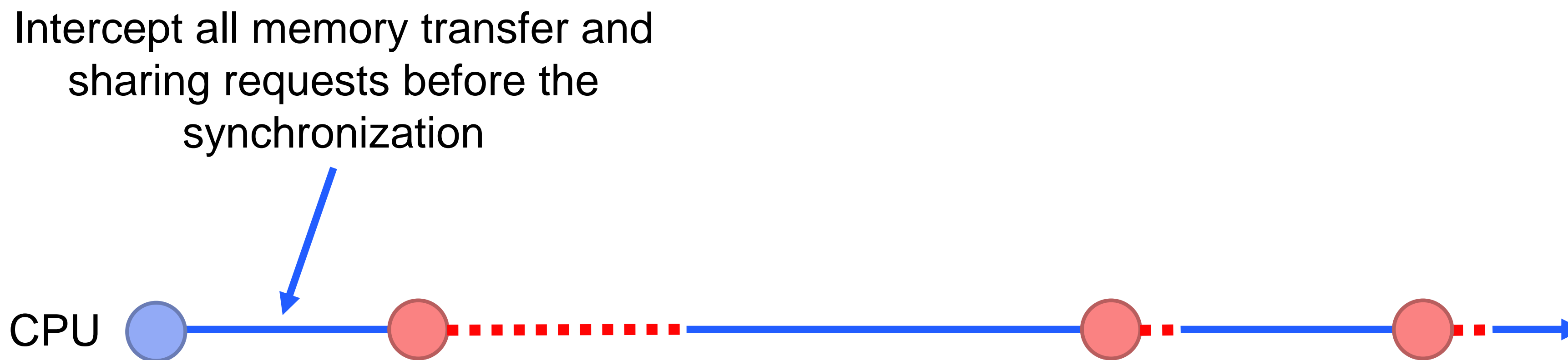
```
Synchronization();
for(…) {
   // Work with no GPU dependencies
}
result = GPUData[0] + …
```



Detection of Synchronization Opportunities

Identify where GPU results are stored in the CPU

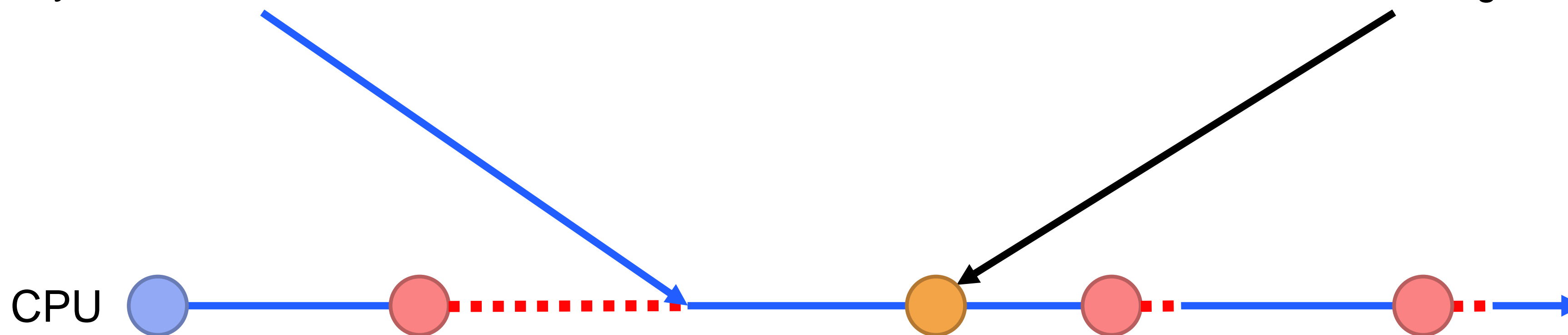
- GPU results are only stored in locations the CPU explicitly specifies via function call before the synchronization
- Intercepting these calls will give us the locations in CPU memory that will contain GPU results.



Detection of Synchronization Opportunities

Identify what CPU instructions access these locations

- Instrumenting load and store operations can identify the instructions accessing shared data locations.



Duplicate Data Transfers

- The characteristic we need to identify is:
  - Duplicate data contained within the transfer
- A content based data deduplication approach will be used to identify these transfers.

Detection of Duplicate Data Transfers

The content based deduplication approach consists of four steps:

1. Intercept the memory transfer requests using library interposition.
2. We create a hash of the data being transferred.
3. Compare the hash to past transfers
4. If there is a match, we mark the transfer as a duplicate.

An initial profiling run of the application using this deduplicator will be run to identify duplicate transfers

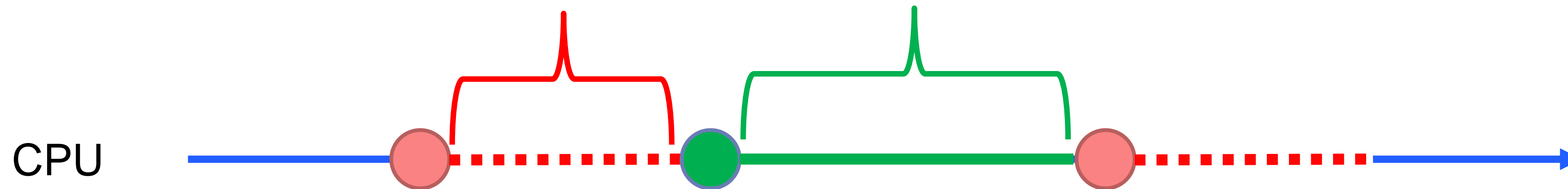
Step 4: Predictive Performance Model

- Only show problematic memory transfers/synchronizations (identified by Step 3).
- For each of problematic operations Diogenes outputs
  - A stack trace of the point
  - The time that could be saved by moving/removing it
    - Calculated using timing data from Step 2
  - The percentage of total execution time that could potentially be saved.

Predictive Performance Model

For synchronizations*, time saved estimate is generated by:

MIN(Sync Delay Time, Time To Next Sync)



For memory transfers, estimate is the time of the transfer.

If a memory transfer has a problematic synchronization as well, the estimates generated for both problems are combined.

* Note: for certain calls where removal of the operation also removes an implicit synchronization, we add that operation/overhead time to the savings

Experiments

Tested Diogenes diagnosis capability on four applications

- cumf_als - Matrix Factorization Library (IBM/UC Berkeley)
- cuIBM – Computational Fluid Dynamics application (Boston University)
- AMG – Algebraic Multigrid Solver (LLNL)
- Rodinia – Gaussian Benchmark (UVA).

All experiments were run on LLNL Ray (Coral EA POWER 8 system), CUDA version 9.2

Diogenes Predictive Accuracy Overview

App Name	App Type	Diogenes Estimated Benefit (% of Exec)	Actual Benefit by Manual Fix (% of Exec)
cumf_als	Matrix Factorization	10.0%	8.3%
AMG	Algebraic Solver	6.8%	5.8%
Rodinia	Gaussian Benchmark	2.2%	2.1%
culBM	CFD	10.8%	17.6%

Estimates for the prominent problems in each application we corrected.

- Tried to be as careful as possible to alter only the problematic operation

Profiler Comparison for cumf_als

Operation	NVProf Profiled Time		HPCToolkit Profiled Time		Diogenes Est Benefit	
	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos
cudaDeviceSynchronize	745s (52%)	1	628s (24%)	1	1s (0%)	3
cudaFree	275s (18%)	2	258s (10%)	2	214s (15%)	1
cudaMalloc	218s (17%)	3	230s (9%)	3		
cudaMemcpy	158s (11%)	4	119s (4%)	4	30s (2%)	2

Profiler Comparison for cumf_als

Operation	NVProf Profiled Time		HPCToolkit Profiled Time		Diogenes Est Benefit	
	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos
cudaDeviceSynchronize	745s (52%)	1	628s (24%)	1	1s (0%)	3
cudaFree	275s (18%)	2	258s (10%)	2	214s (15%)	1
cudaMalloc	218s (17%)	3	230s (9%)	3		
cudaMemcpy	158s (11%)	4	119s (4%)	4	30s (2%)	2

NVProf and HPCToolkit report 745s and 628s spent in cudaDeviceSynchronize

Diogenes reports no potential savings

Manually verified by removing cudaDeviceSynchronize calls

Profiler Comparison for cumf_als

Operation	NVProf Profiled Time		HPCToolkit Profiled Time		Diogenes Est Benefit	
	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos
cudaDeviceSynchronize	745s (52%)	1	628s (24%)	1	1s (0%)	3
cudaFree	275s (18%)	2	258s (10%)	2	214s (15%)	1
cudaMalloc	218s (17%)	3	230s (9%)	3		
cudaMemcpy	158s (11%)	4	119s (4%)	4	30s (2%)	2

Expected benefit order differs from profiled time

Diogenes does not report times on cudaMalloc

- Does not perform a CPU/GPU synchronization in most use cases (on modern drivers)

Profiler Comparison for cuIBM

Operation	NVProf Profiled Time		HPCToolkit Profiled Time		Diogenes Est Benefit	
	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos
cudaFree	Profiler Crashed		447s (12%)	1	421s (22%)	1
cudaLaunchKernel			395s (12%)	2		
cudaMalloc			382s (10%)	3		
cudaDeviceSynchronize			170s (5%)	4	136s (7%)	2
cudaMemcpyAsync			163s (5%)	5	80s (4%)	3
cudaFuncGetAttributes			154s (4%)	6		
cudaStreamSynchronize			52s (1%)	7	4s (0%)	4

Differences in cudaMemcpyAsync and cudaStreamSynchronize

Profiler Comparison for cuIBM

Operation	NVProf Profiled Time		HPCToolkit Profiled Time		Diogenes Est Benefit	
	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos	Time (% of Exec)	Profile Pos
cudaFree	**Profiler Crashed**		447s (12%)	1	421s (22%)	1
cudaLaunchKernel			395s (12%)	2		
cudaMalloc			382s (10%)	3		
cudaDeviceSynchronize			170s (5%)	4	136s (7%)	2
cudaMemcpyAsync			163s (5%)	5	80s (4%)	3
cudaFuncGetAttributes			154s (4%)	6		
cudaStreamSynchronize			52s (1%)	7	4s (0%)	4

Diogenes excludes non-synchronous and non-transfer calls

Profiler Comparison for Rodinia

Operation	NVProf Profiled Time Time (% of Exec)	NVProf Profiled Time Profile Pos	HPCToolkit Profiled Time Time (% of Exec)	HPCToolkit Profiled Time Profile Pos	Diogenes Est Benefit Time (% of Exec)	Diogenes Est Benefit Profile Pos
cudaThreadSynchronize	**6.05s (94%)**	**1**	5.01s (75%)	1	0.13s (2%)	1
cudaMemcpy	**0.01s (1%)**	**2**	0.07s (1%)	2	0.06 (1%)	2
cudaFree	**< 0.01s (1%)**	**3**	< 0.01s (1%)	3	< 0.01s (1%)	3

NVProf and HPCToolkit both report >75% of execution time spent in cudaThreadSynchronize

Diogenes reports limited potential savings (~2%)

- Verified by removed cudaThreadSynchronize (actual savings 2.2%)

Diogenes Predictive Accuracy Overview

App Name	App Type	Diogenes Estimated Benefit (% of Exec)	Actual Benefit by Manual Fix (% of Exec)
cumf_als	Matrix Factorization	10.0%	8.3%
AMG	Algebraic Solver	6.8%	5.8%
Rodinia	Gaussian Benchmark	2.2%	2.1%
culBM	CFD	10.8%	17.6%

culBM's and cumf_als had synchronization issues that were symptoms of larger problems

- Memory management issues (cudaMalloc/cudaFree)
- Asynchronous transfer issues (synchronous cudaMemcpyAsync)

Fixing the cause of these issues can result in much larger benefit

- Removing the malloc, using cudaMallocHost to allocate memory to be used with cudaMemcpyAsync, etc.

Current Status of Diogenes

Prototype is working on Power 8 architectures

- Including on the current GPU driver versions used on LLNL/ORNL machines
- Power was initial target because we were working with LLNL on Coral early access machines

What about x86?

- Should run on x86 with modifications
  - Work on a port is in progress

Current Status of Diogenes

Ncurses interface for exploring Diogenes analysis

Diogenes Overview Display

Time(s) (% of execution time)
421.716s (22.52%) Fold on cudaFree
150.353s (8.03%) Sequence starting at call ....
136.150s (7.27%) Fold on cudaDeviceSynchronize
98.803s (5.28%) Sequence starting at call ...
80.938s (4.32%) Fold on cudaMemcpyAsync
...
Back/Previous
Exit

Expansion of Problem

Time(s) (% of execution time)
421.716s(22.52%) Fold on cudaFree
  202.985s(10.84%) thrust::detail::contiguous_storage<...>
    Conditionally unnecessary (see: conditions)
  113.375s(6.06%) thrust::pair<...>
    Conditionally unnecessary (see: conditions)
  65.258s(3.49%) void cusp::system::detail::generic::multiply<...>
    Conditionally unnecessary (see: conditions)
  ...

Diogenes – Overhead/Limitations

Overhead:

- 6x-20x application run time
- Benefit from improvements to binary parsing in the binary instrumentation tool Dyninst
  - Parse overhead declined from > 15 minutes to a few minutes for large (>100 MB) binaries.

Limited to programs with a single GPU context (one user thread is responsible for all cuda calls)

Questions?

Papers:

- Exposing Hidden Performance Opportunities in High Performance GPU Applications
  - Best Paper at CCGrid 2018, Available on http://paradyn.org/
- Diogenes: Looking For An Honest CPU/GPU Performance Measurement Tool
  - This paper (SC19)
- Untitled Autocorrection paper

Diogenes Github: http://github.com/bwelton/diogenes