Diogenes: A tool for exposing Hidden GPU Performance Opportunities

Benjamin Welton and Barton P. Miller
Computer Sciences Department
University of Wisconsin – Madison

2018 Performance Tools Workshop – Solitude, UT
July 9th 2018

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Paradyn™

Overview of Diogenes

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

- Unnecessary interactions
- Misplaced interactions

Output is a list of unnecessary or misplaced interactions

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

New Features of Diogenes

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 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.

Impact of Misplaced/Unnecessary Interactions

App Name	App Type	LOC	Original Runtime (Min:Sec)	Percent Reduction	Problems Found
Hoomd-Blue	MDS	112,000	08:36	37%	SY
Qbox / Qball (*)	MDS	100,000	38:54	85%	DD, SY
culBM	CFD	17,000	31:42	27%	SY, JT

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

SY = Synchronization, **DD** = Duplicate Data Transfers, **JT** = JIT Compilation

(*) built with cuFFT

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

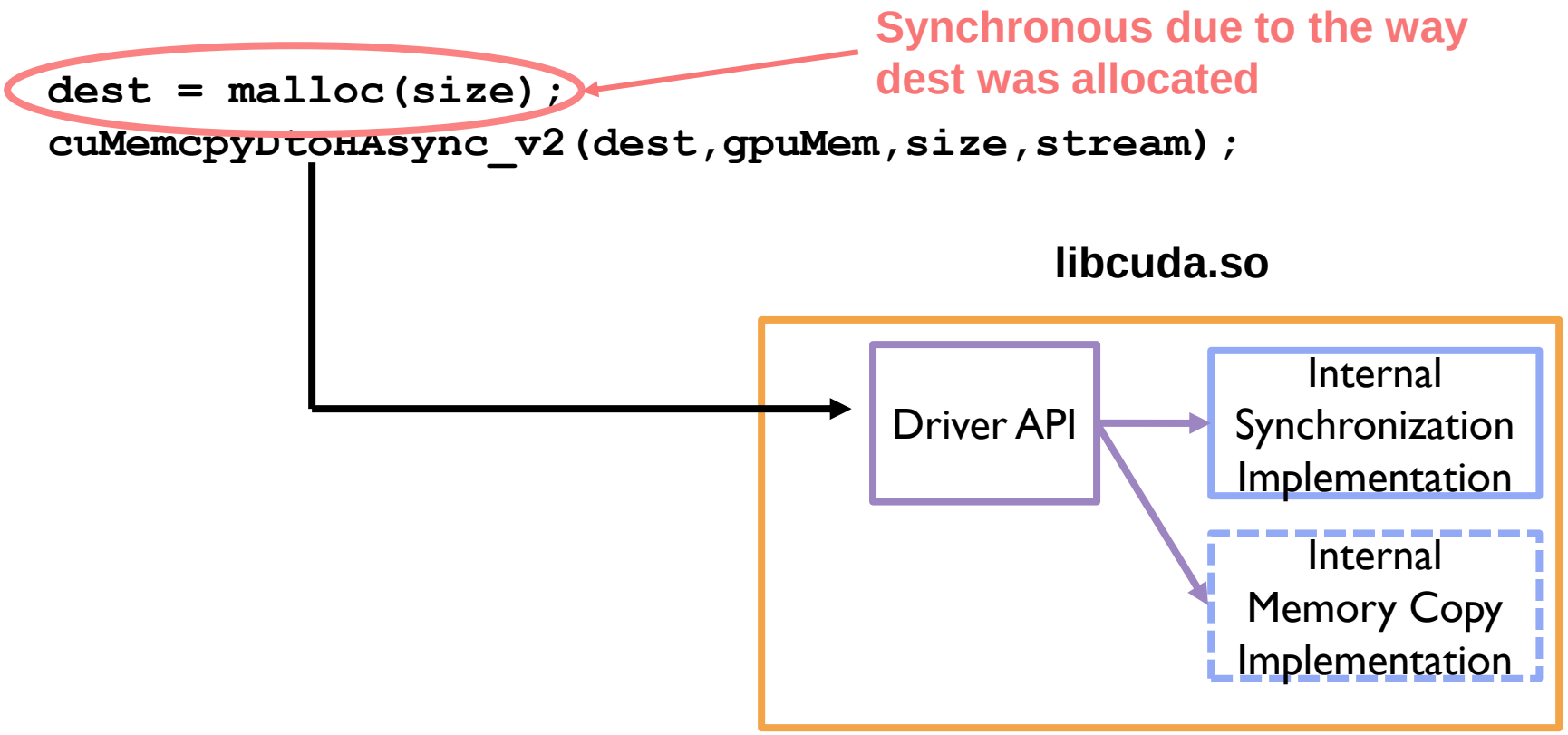
Existing Tools (CUPTI, etc) have collection and analysis gaps preventing detection of issues

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

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

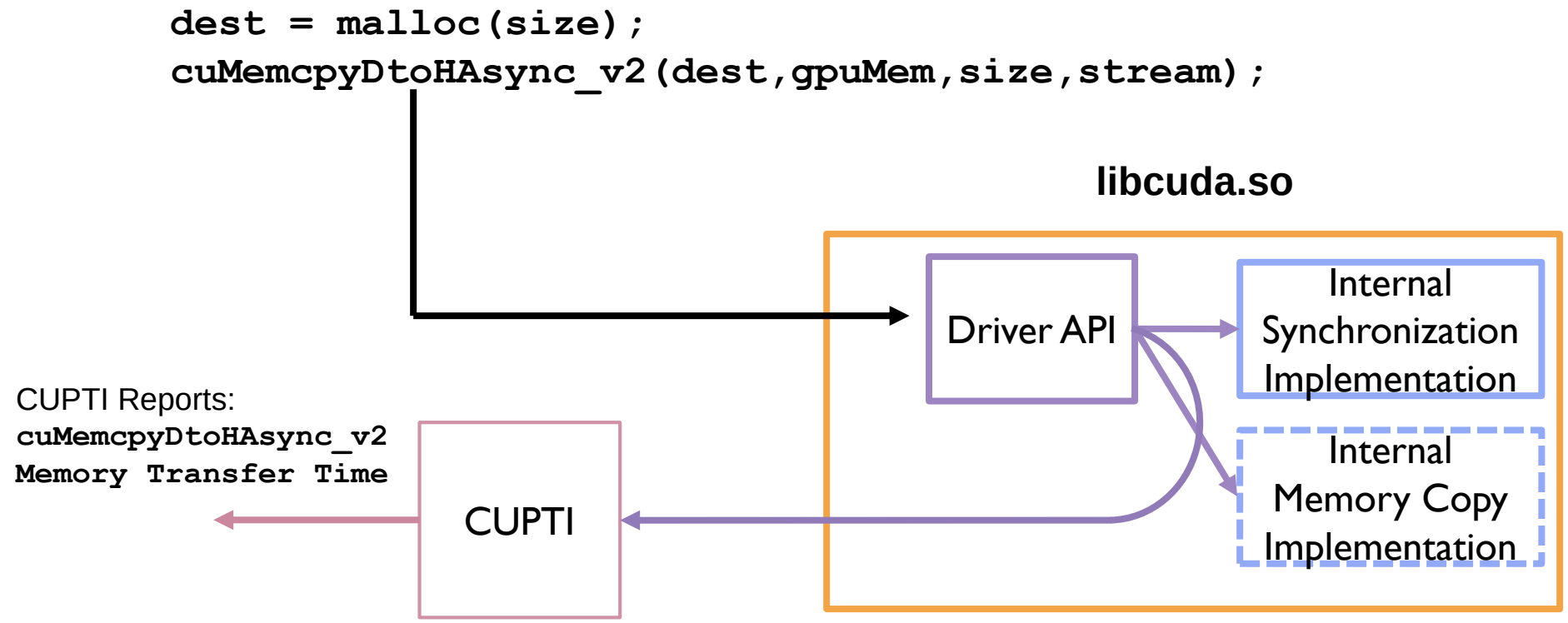
Conditional Interaction

Conditional Interactions are unreported (and undocumented) synchronizations/transfers performed by a CUDA call.



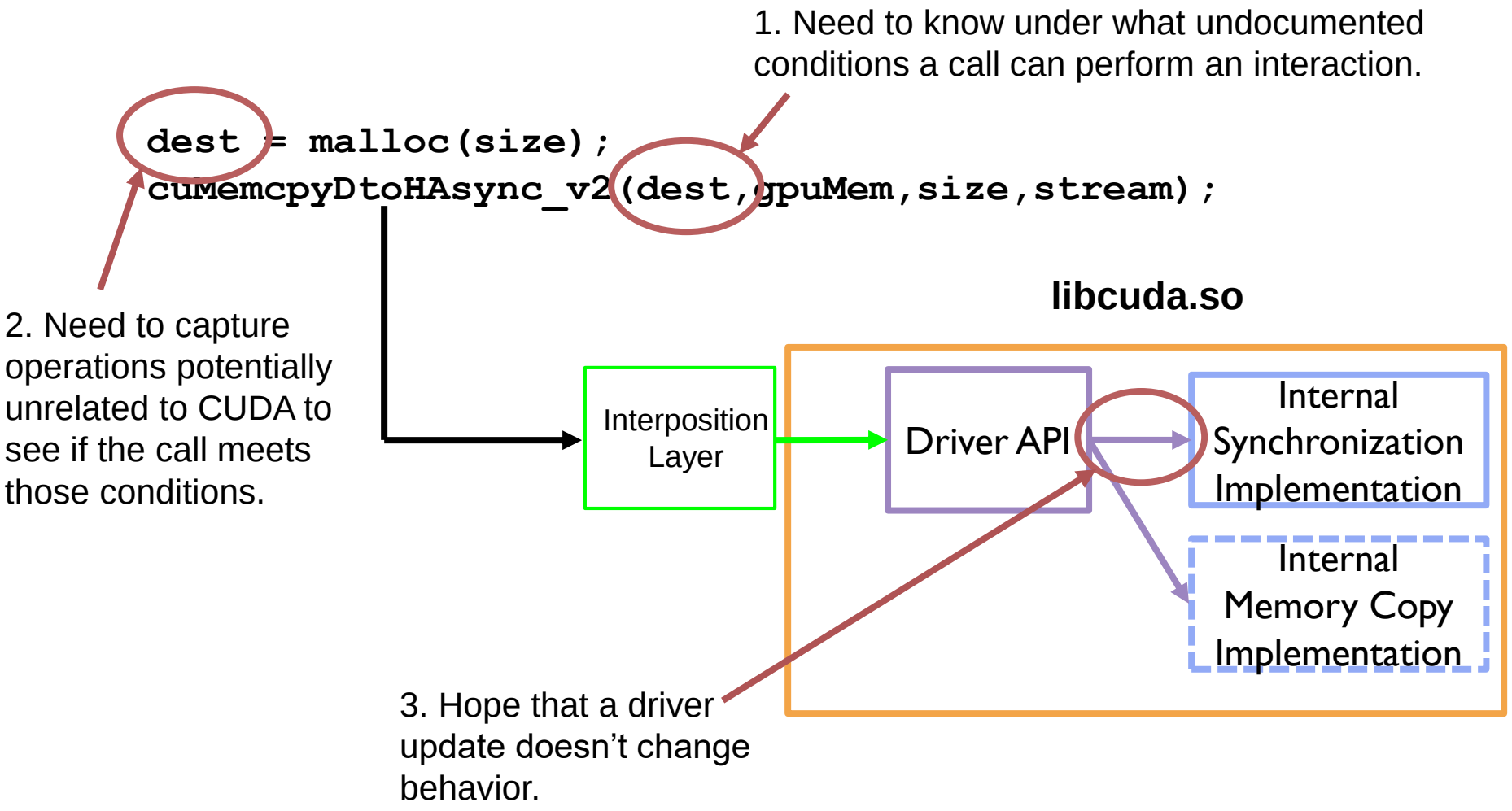
Conditional Interaction Collection Gap

CUPTI doesn't report when undocumented interactions are performed by a call.



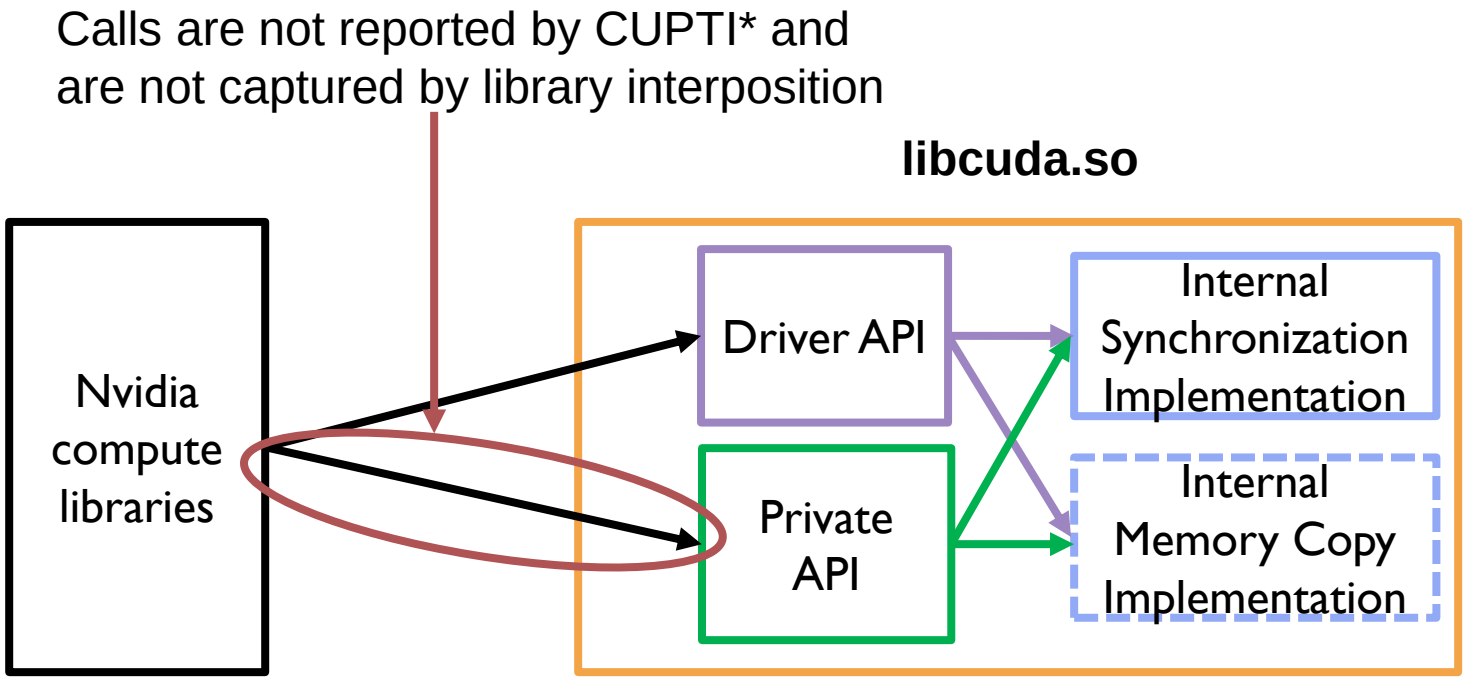
Conditional Interaction Collection Gap

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).

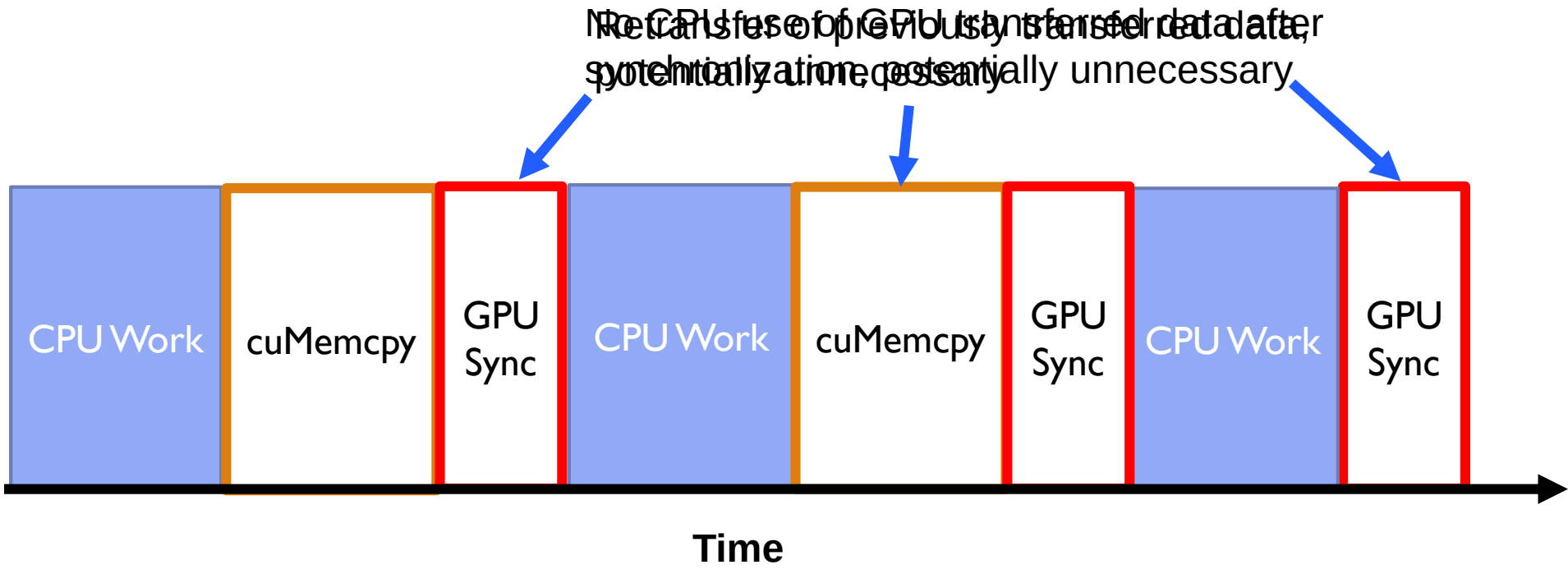


*Fun Fact: CUPTI sets its callbacks through the Private API

Gap in Analysis

Just supplying profiling/tracing information leaves analysis to developers

- Ideally a developer would only see potentially problematic operations.



Diogenes Collection and Analysis Techniques

1. Identify and time interactions
   - Including hidden synchronizations and memory transfers

   Binary Instrumentation of libcuda to identify and time calls performing synchronizations and/or data transfers
2. Determine the necessity of the interaction
   - If the interaction is necessary for correctness, is it placed in an efficient location?

   Synchronizations: A combination of memory tracing, CPU profiling, and program slicing

   Duplicate Data Transfers: Content based data deduplication approach.
3. Provide an estimate of the fixing the bad interactions

   Diogenes uses a new Feed Forward Instrumentation workflow for data collection and analysis

Diogenes – Workflow

Diogenes uses a newly developed technique called feed forward instrumentation:

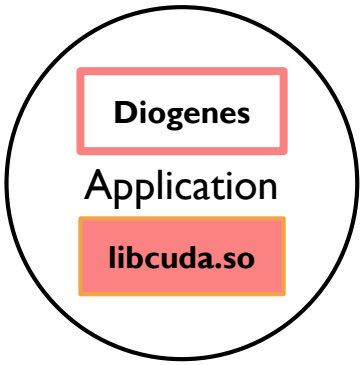
- The results of previous instrumentation guides the insertion of new instrumentation.

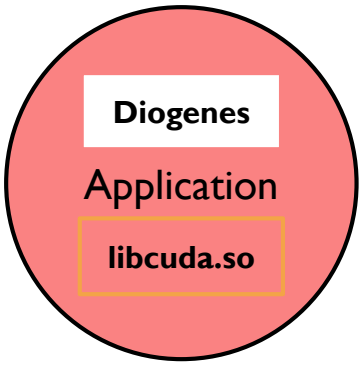
Step 1
Measure execution time of the application (without instrumentation)

Step 2
Instrument libcuda to identify and time synchronizations and Memory Transfers

Step 3
Instrument application to determine necessity of the operation.

Step 4
Offline analysis of data collected in Step's 1-3 to identify problematic calls and potential savings





Call	Type	Potential Savings
…	…	…
…	…	…

Diogenes performs each step automatically (via a launcher)

Step 1: Measure execution time of the application (without instrumentation).

- Interested in total application execution time
- Used by Step 4 (Analysis) to produce a potential savings estimate relative to normal application execution time.

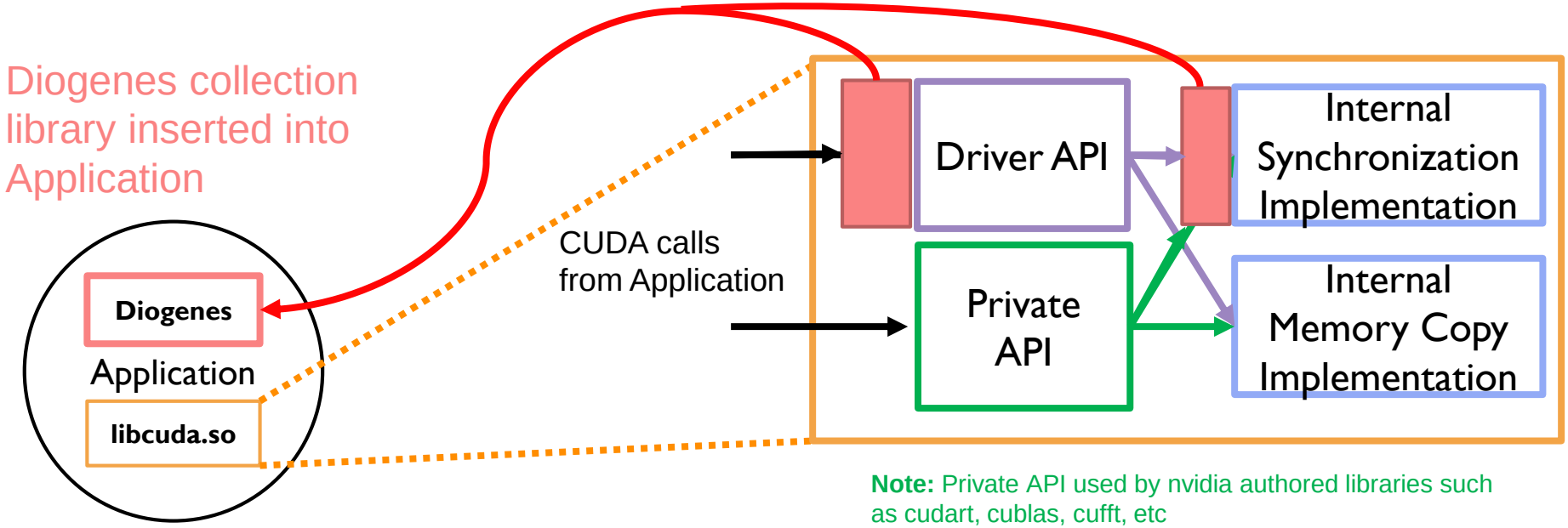
Step 2: Instrument libcuda to identify and time Synchronizations and Memory Transfers

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

Wrap entire public API to capture and time cuda calls



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

Diogenes performs each step automatically

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

Employs two detection techniques:

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

Why do Applications Synchronize?

Synchronize to read the results of GPU computation.

- Synchronization waits for all updates to shared data to be written by the GPU.

Shared data is data that can accessed by the GPU.

- Shared memory pages between the CPU and GPU and memory transfers from the GPU.

If we can determine that the CPU does not access the results from the GPU, the synchronization is unnecessary.

A Synchronization Opportunity Exists When…

CPU computation is being delayed unnecessarily

- If some CPU computation after a synchronization does not need GPU results to compute correctly.

No use of data from the GPU by the CPU

- If the CPU is not accessing GPU results, the CPU does not need to synchronize.

If either of these characteristics is true, the synchronization is unnecessary or misplaced.

To identify these characteristics, we must obtain:

The amount of time the CPU is blocked at a synchronization.

- Binary instrumentation to determine the amount of time a call spends synchronizing.

The locations of GPU results (shared data) on the CPU.

- Use memory tracing to obtain this information

The CPU instructions that access that data.

- Use memory tracing and program slicing to obtain this information.

Types of Synchronization Opportunities

We categorize synchronization opportunities into three types.

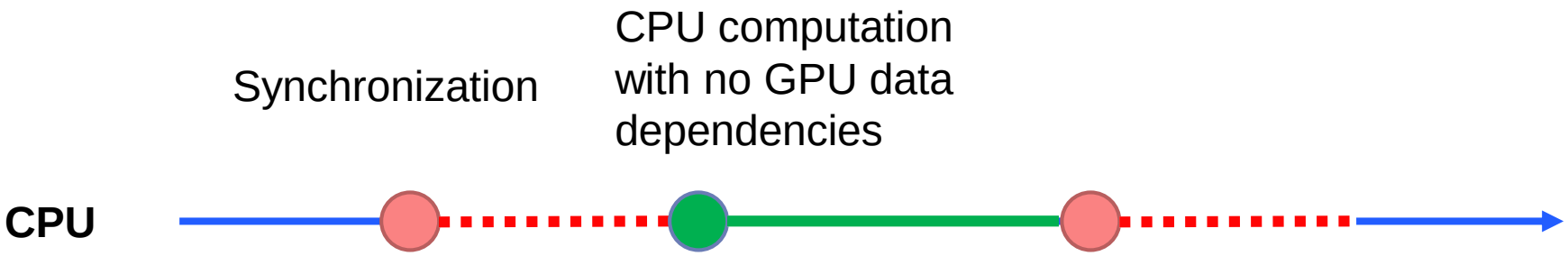
1. When the CPU does not access shared data after the synchronization (seen in cuIBM and QBox)
2. When the placement of the synchronization is far from the first access of shared data by the CPU (seen in QBox).
3. [Future Work] When CPU computation not dependent on GPU data is delayed by a synchronization (seen in hoomd).

Detection of Synchronization Opportunities

1. When the CPU does not access shared data after the synchronization (seen in culBM and QBox)

CPU Example

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

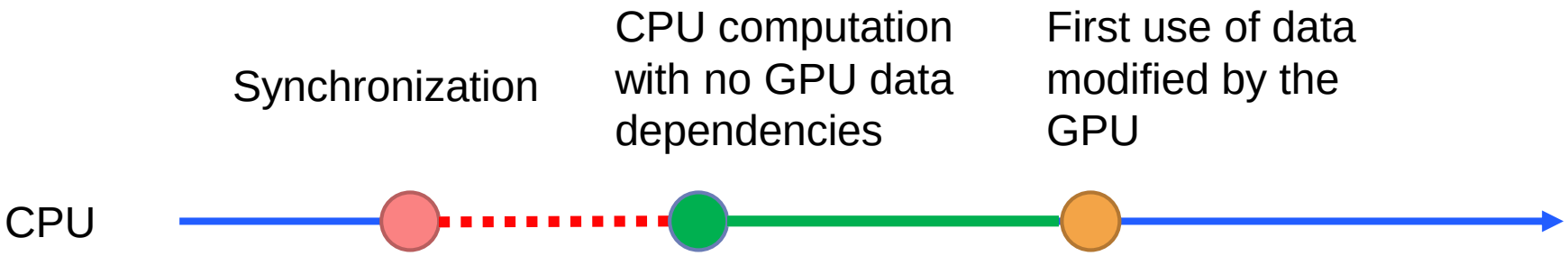


Detection of Synchronization Opportunities

2. When the placement of the synchronization is far from the first access of shared data by the CPU (seen in QBox)

CPU Example

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

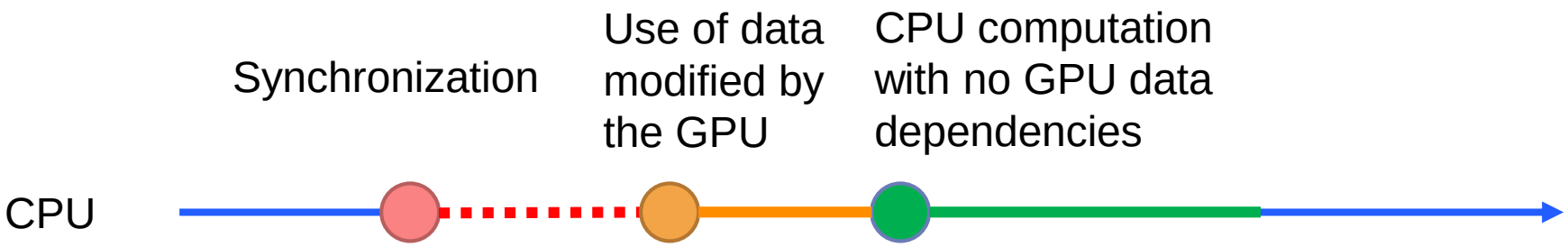


Detection of Synchronization Opportunities

3. When CPU computation not dependent on GPU data is delayed by a synchronization (seen in hoomd).

CPU Example

```
Synchronization();
for(x : GPUData) {
    // Use GPU Data
}
for (…) {
    // Work with no GPU dependencies
}
```



Detection of Synchronization Opportunities

The first two types of synchronization opportunities are identified using profiling and memory tracing.

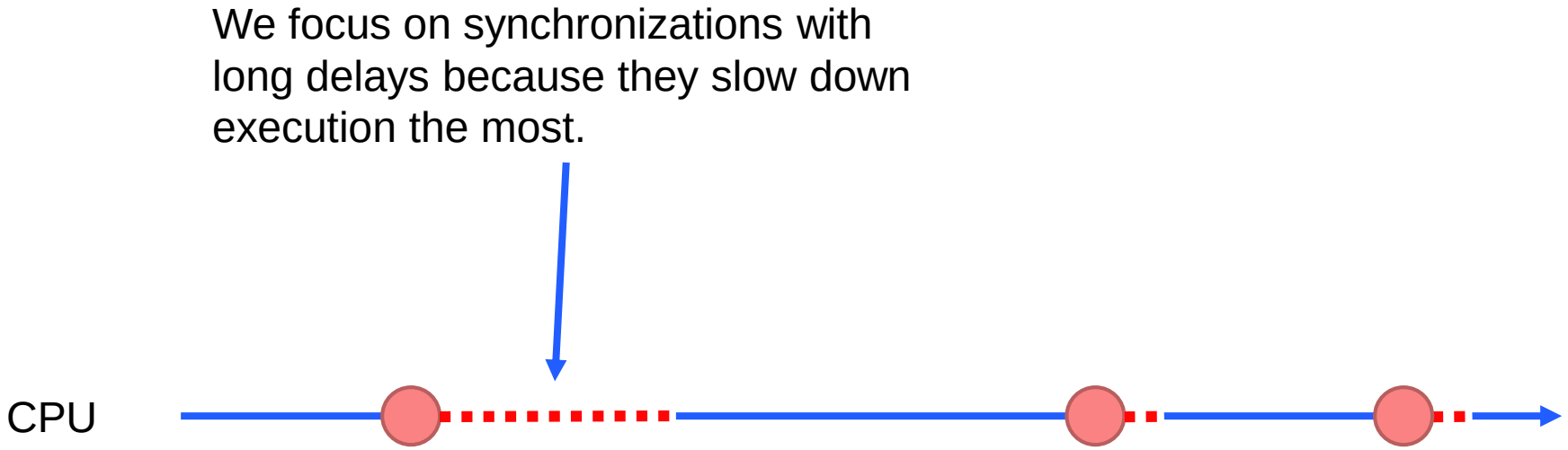
- We must identify if the CPU accesses GPU results and where those accesses occur.

To do this we:

1. Performing an initial profiling run of the application to identify synchronizations with long delays (Performed by Step 2).
2. Identify the CPU computation accessing shared data and where to move the synchronization
   A. Identify the memory locations containing shared data on the CPU
   B. Identify the instructions that access this data.

Detection of Synchronization Opportunities

1. Binary instrumentation inserted into the internal synchronization function of libcuda used to time synchronization delay. (Performed by Step 2)



Detection of Synchronization Opportunities

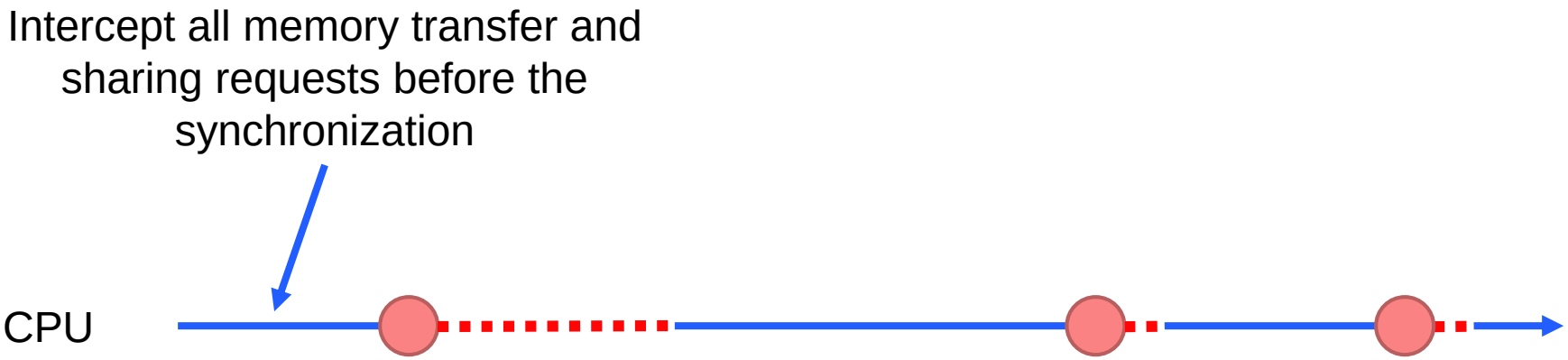
2. We need to identify CPU computation accessing data that can be modified by the GPU (shared data) and where to move the synchronization
   A. Identify where GPU results are stored in the CPU
   B. Identify what CPU instructions access these locations

The first location to access shared data after the synchronization will be the location where they synchronization should be moved to.

Detection of Synchronization Opportunities

2.A Identify where GPU results are stored in the CPU

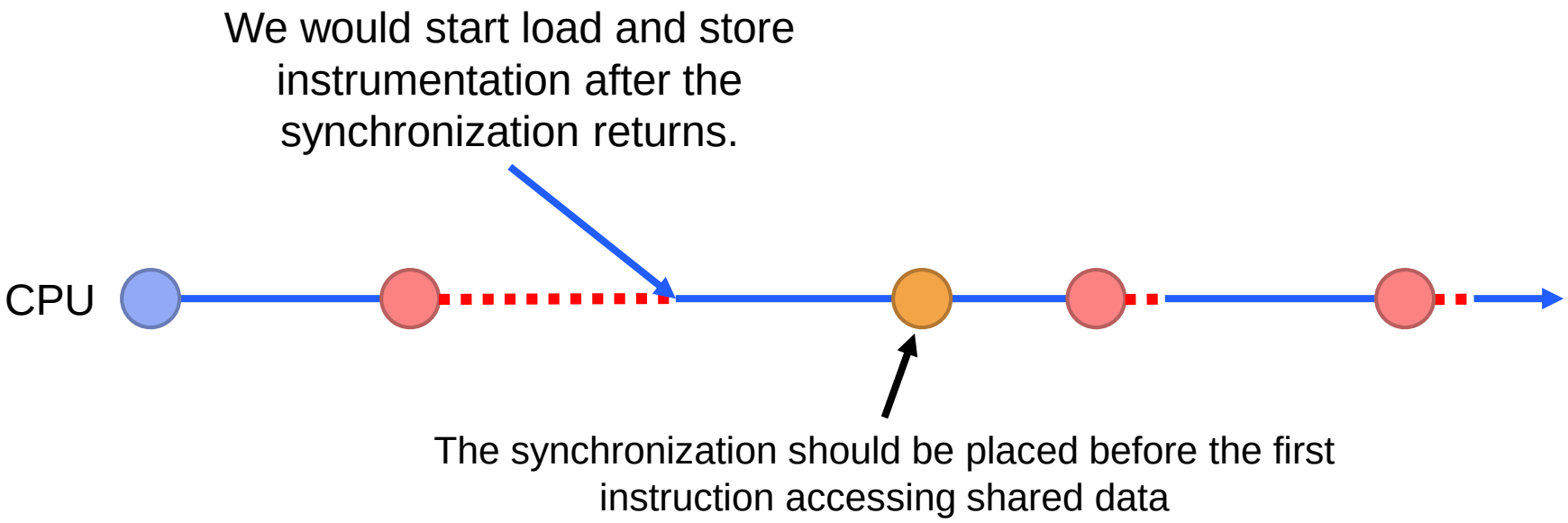
- GPU results are only stored in locations the CPU explicitly specifies via function call before the synchronization
- Intercepting these calls will give us the locations in CPU memory that will contain GPU results.



Detection of Synchronization Opportunities

2.B Identify what CPU instructions access these locations

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

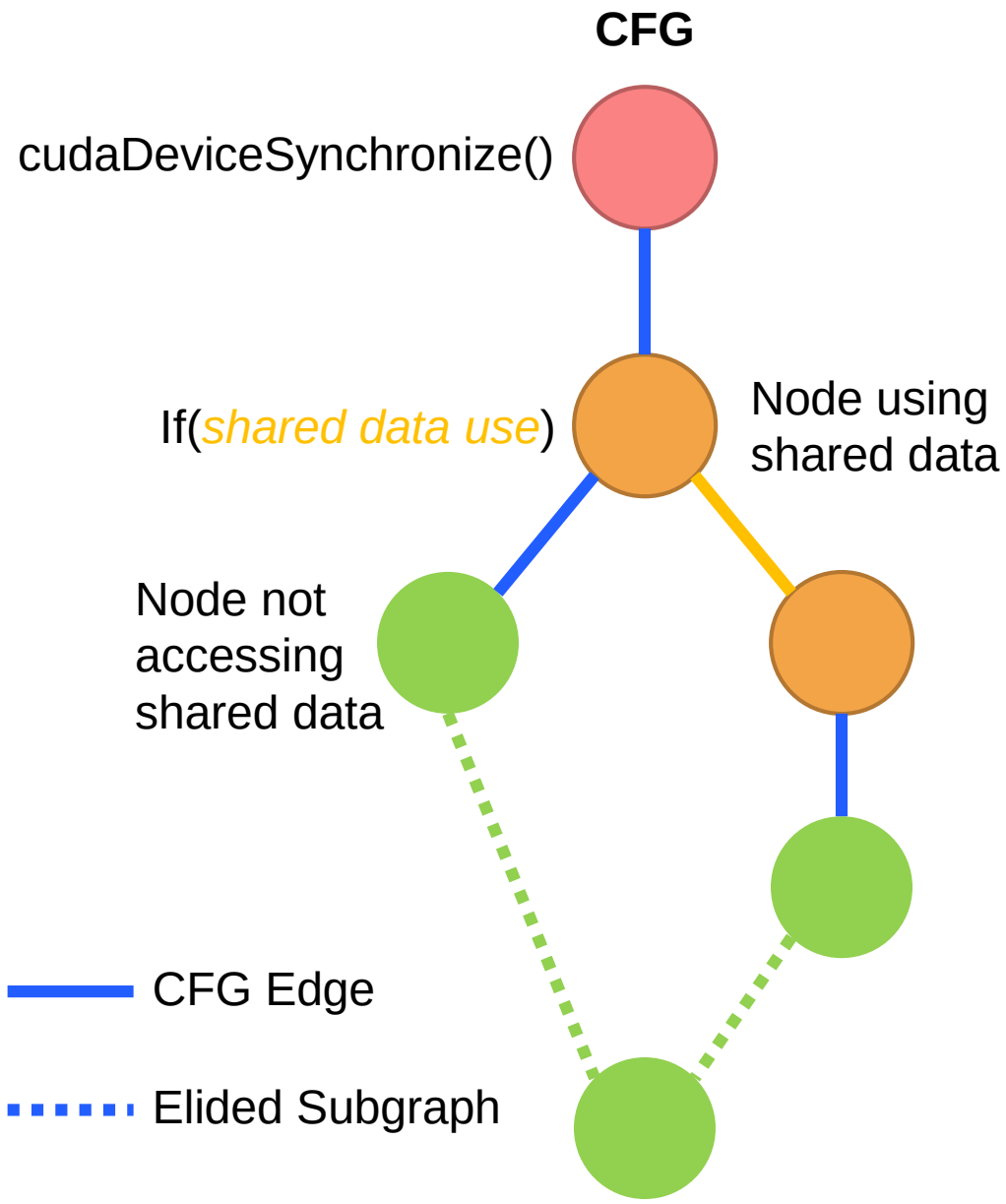


Detection of Synchronization Opportunities

The third type of synchronization opportunity requires identifying CPU computation that does not need GPU results to compute correctly.

1. Perform an initial profiling run on the application to identify synchronizations with long delays.
2. In a separate profiling run, identify the variables that contain shared data.
3. Use program slicing with these variables to identify instructions that may be affected by their values.
   - Identifies the CPU computation that may require GPU data to compute.

Program Slicing



Synchronization opportunity #3 (CPU computation with no dependencies being delayed) is a work in progress.

```
Synchronization();
for(x : GPUData) {
   // Use GPU Data
}
for (…) {
   // Work with no GPU dependencies
}
```

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: Analysis

- Only show problematic memory transfers/synchronizations (identified by Step 3).
- For each of problematic operations Diogenes outputs
  - The type of problem identified (synch, data transfer)
  - A stacktrace 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.

Diogenes - What's Working

Duplicate data detection and the following synchronization issues:

Type 1 - Working

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

Type 2 - Working

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

Type 3 – Early Development (>3 Months out)

```
Synchronization();
for(x : GPUData) {
   // Use GPU Data
}
for (…) {
   // Work with no GPU dependencies
}
```

Current Status of Diogenes

Prototype is working on x86_64 but there are some in progress components:

- Power architecture support (maybe this week?)
- Merging data deduplication and synchronization analysis.

Diogenes is still a work in progress and may encounter issues (such as instrumentation overhead) in the future.

Open Questions

- What is the real world impact on performance of hidden interactions?
  - We believe impact to be high, but we need to gather concrete performance results.
- What other interesting profiling/tracing events are unreported?
  - Are there unified memory events that are not reported?
- What other high level analysis techniques could be used with instrumentation of the GPU user space driver and CPU?