Exposing Hidden Performance Opportunities in High Performance GPU Applications

Benjamin Welton

Scalable Tools Workshop 2017

Granlibakken, Tahoe, NV

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Paradyn™

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, launching kernels, etc).
- Integrating GPU code into existing CPU code

Summary of Opportunities

- Synchronization
  - Unnecessary synchronization with the GPU reducing CPU – GPU overlap
- Duplicate Data Transfers
  - Unnecessary transfers of data between the GPU and CPU
- Missed Parallelization
  - The conversion of "low benefit" GPU opportunities that result in substantial performance improvements
- Just-in-time compilation of GPU code
  - Incorrect compilation of an application resulting in inefficient generation of native GPU code at runtime.

GPUs are difficult to use

In the abstract, these opportunities are easy to identify.

- And they are when viewed in isolation in small programs.

In practice, these opportunities that seem simple are difficult to identify in real world applications.

Why are they hard to identify?

Size and complexity of the code base

- Difficult for a programmer to manually identify bad behavior across 100K+ LOC

Code evolving over time

- Changes to code bases introduce bad behavior in once innocent code

Use of independently developed libraries with GPU functionality

- Each library is efficient locally but when combined in application result in bad behavior not apparent to the developer

Have to support many different compute architectures (OpenMP, CUDA, Phi, etc).

- Assumptions about usage are made which result in bad behavior when they are not true

Identification of Performance Opportunities

To start to answer this question we looked for unobvious performance opportunities in real applications.

- Identify what performance opportunities exist
- Determine their impact on application performance

Preliminary Results

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.

Summary of Opportunities

- Synchronization
  - Unnecessary synchronization with the GPU reducing CPU – GPU overlap
  - Found in hoomd-blue (37%), QBox (~40%), and cuIBM (7%)
- Duplicate Data Transfers
  - Unnecessary transfers of data between the GPU and CPU
  - Found in QBox (~40%)
- Missed Parallelization
  - The conversion of "low benefit" GPU opportunities that result in substantial performance improvements
  - Found in LAMMPs (19%)
- Just-in-time compilation of GPU code
  - Incorrect compilation of an application resulting in inefficient generation of native GPU code at runtime.
  - Found in cuIBM (~20%).

Characteristics of Opportunities

- Synchronization
  - Synchronization causing a long delay on the CPU
  - CPU computation being delayed unnecessarily
  - No use of data from the GPU by the CPU
- Duplicate Data Transfers
  - Duplicate data contained within the transfer
- Missed Parallelization
  - Loops with long CPU runtimes
  - A sequential memory access pattern for variables within the loop.
- Just-in-time compilation of GPU code
  - Compatibility mismatch between the GPU code contained in the application and the card it is run on.

Characteristics Not Identified by Tools.

- Synchronization
  - Synchronization causing a long delay on the CPU
  - CPU computation being delayed unnecessarily
  - No use of data from the GPU by the CPU
- Duplicate Data Transfers
  - Duplicate data contained within the transfer
- Missed Parallelization
  - Loops with long CPU runtimes
  - A sequential memory access pattern for variables within the loop.
- Just-in-time compilation of GPU code
  - Compatibility mismatch between the GPU code contained in the application and the card it is run on.

Detection Techniques Overview

To automatically identify the four performance issues:

- We must identify the characteristics applications exhibit when they are present.

We propose to identify these characteristics by:

Synchronization

- CPU computation being delayed unnecessarily

  Memory tracing combined with program slicing

- No use of data from the GPU by the CPU

  Memory tracing combined with CPU profiling

Duplicate Data Transfers

- Duplicate data contained within the transfer

  Content based data deduplication

Missed Parallelization

- A sequential memory access pattern for variables within the loop.

  Memory tracing combined with CPU profiling.

Just-in-time compilation of GPU code

- Compatibility mismatch between the GPU code contained in the application and the card it is run on.

  Binary inspection at runtime.

Why do Applications Synchronize?

Synchronize to read the results of GPU computation.

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

- Existing tools give us this information

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.

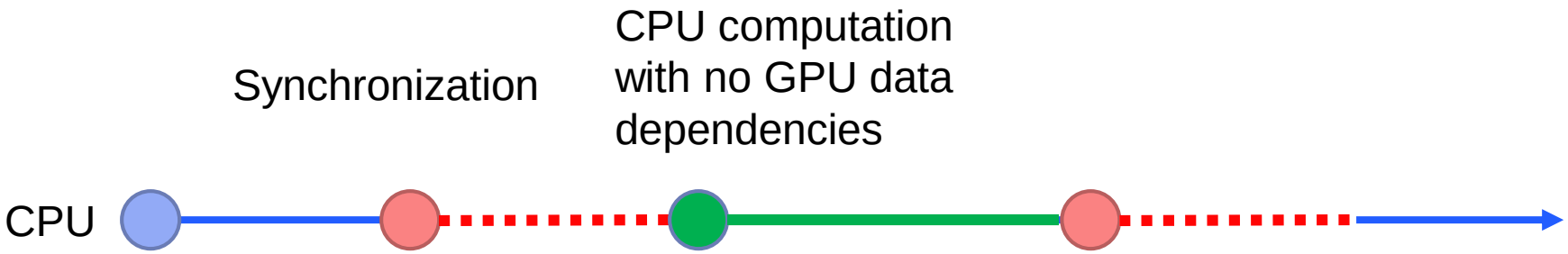
Types of Synchronization Opportunities

We categorize synchronization opportunities into three types.

1. When the CPU does not access shared data after the synchronization (seen in culBM 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. When CPU computation not dependent on GPU data is delayed by a synchronization (seen in hoomd).

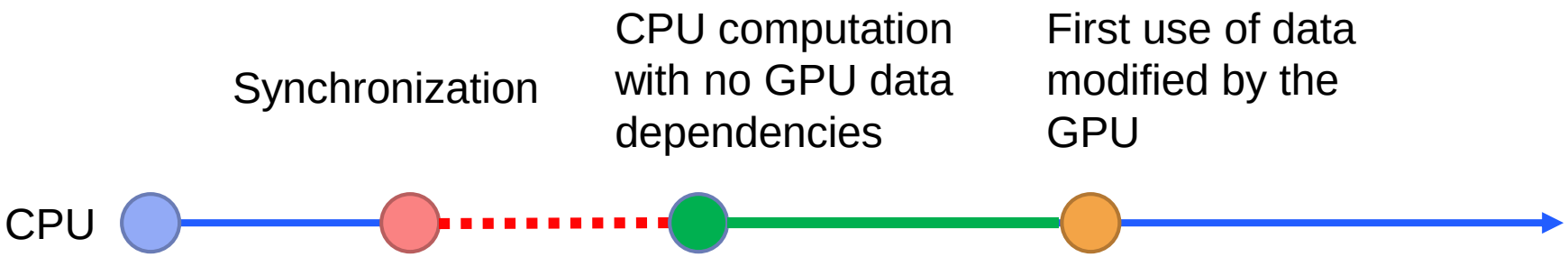
Detection of Synchronization Opportunities

1. When the CPU does not access shared data after the synchronization (seen in culBM and QBox)
   - Synchronization is unnecessary since no CPU computation requires GPU results to compute.



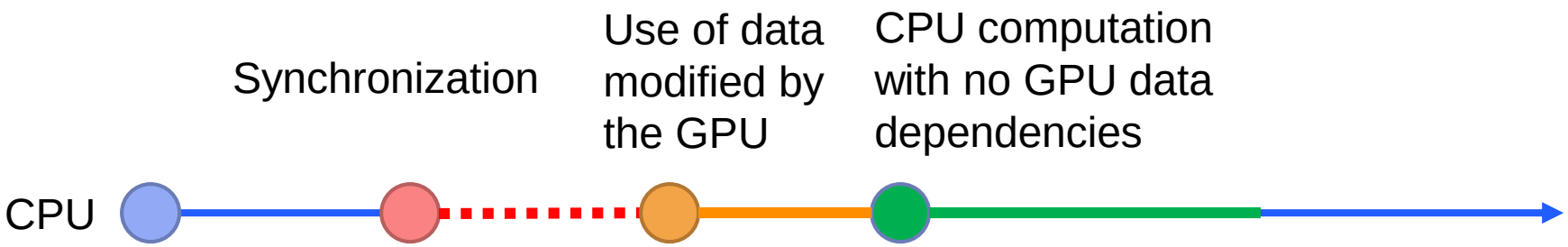
Detection of Synchronization Opportunities

2. When the placement of the synchronization is far from the first access of shared data by the CPU (seen in QBox)
   - Synchronization occurs too early before the results are needed by the CPU.



Detection of Synchronization Opportunities

3. When CPU computation not dependent on GPU data is delayed by a synchronization (seen in hoomd).
   - Moving the CPU computation in front of the synchronization would reduce delay.



Detection of Synchronization Opportunities

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

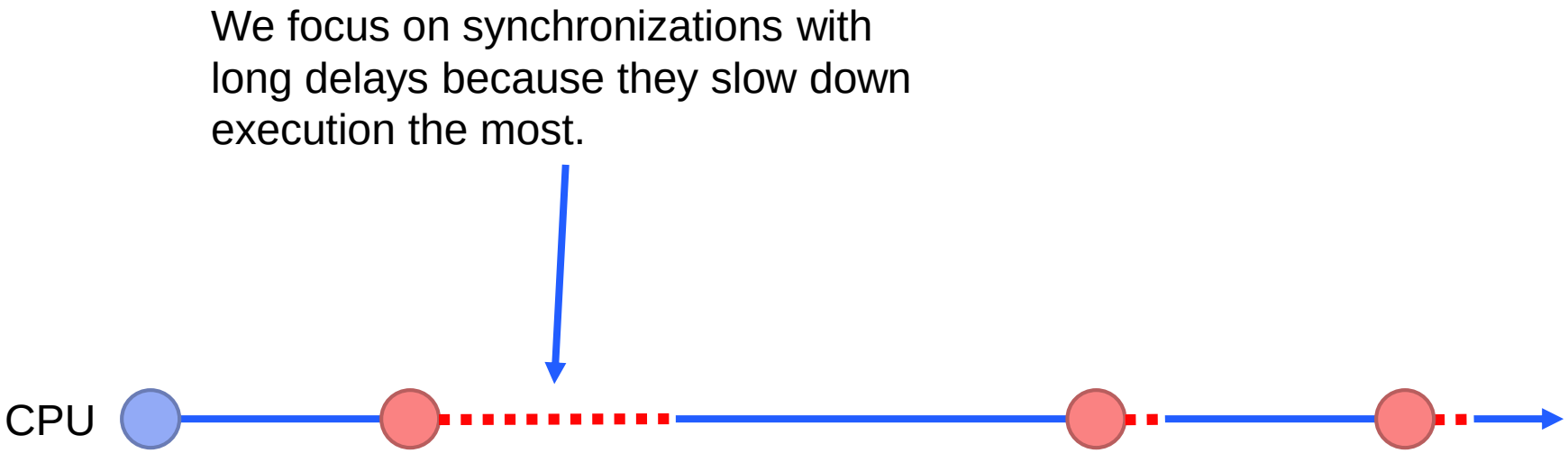
- We must identify if the CPU accesses GPU results and where those accesses occur.

To do this we plan on:

1. Performing an initial profiling run of the application to identify synchronizations with long delays.
2. In a separate profiling run, we will Identify the CPU computation accessing shared data and where to move the synchronization
    1. Identify the memory locations containing shared data on the CPU
    2. Identify the instructions that access this data.

Detection of Synchronization Opportunities

1. Existing profilers are used to identify synchronizations with long delays



Detection of Synchronization Opportunities

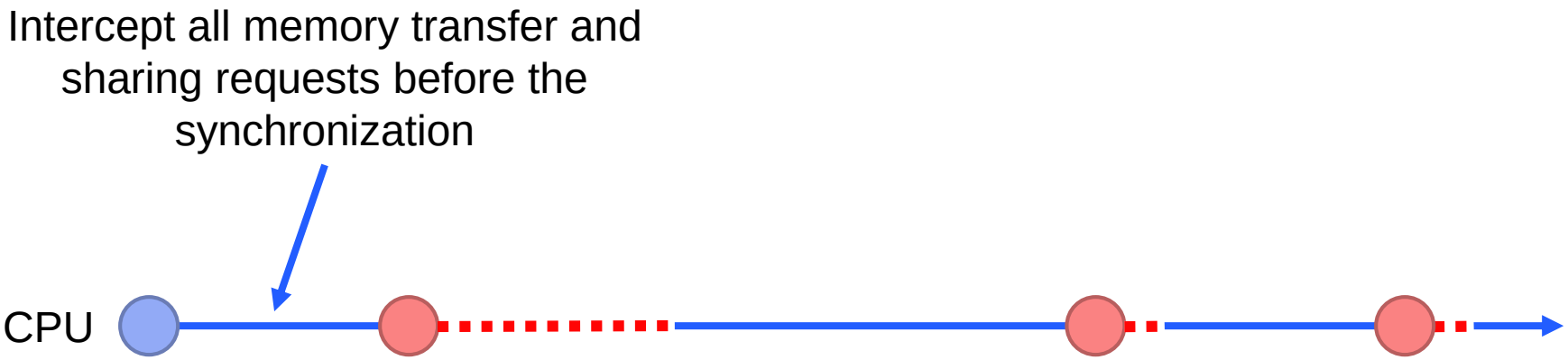
2. We need to identify CPU computation accessing data that can be modified by the GPU (shared data) and where to move the synchronization
   A. Identify where GPU results are stored in the CPU
   B. Identify what CPU instructions access these locations

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

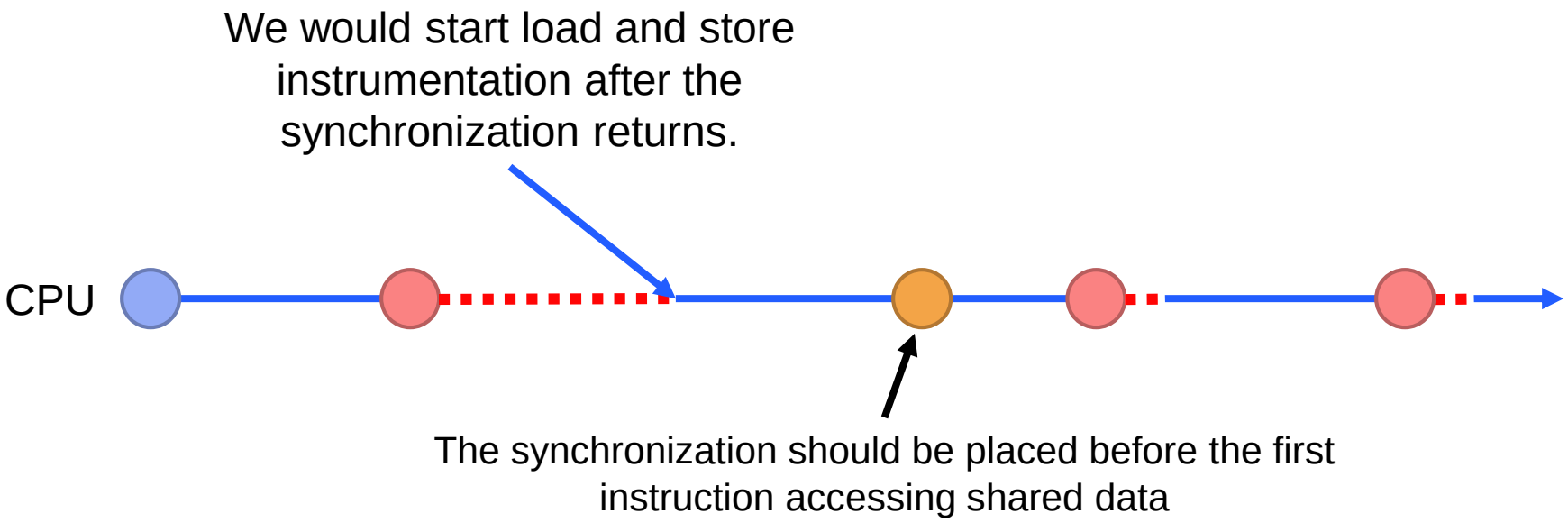
- GPU results are only stored in locations the CPU explicitly specifies via function call before the synchronization
- Intercepting these calls will give us the locations in CPU memory that will contain GPU results.



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.

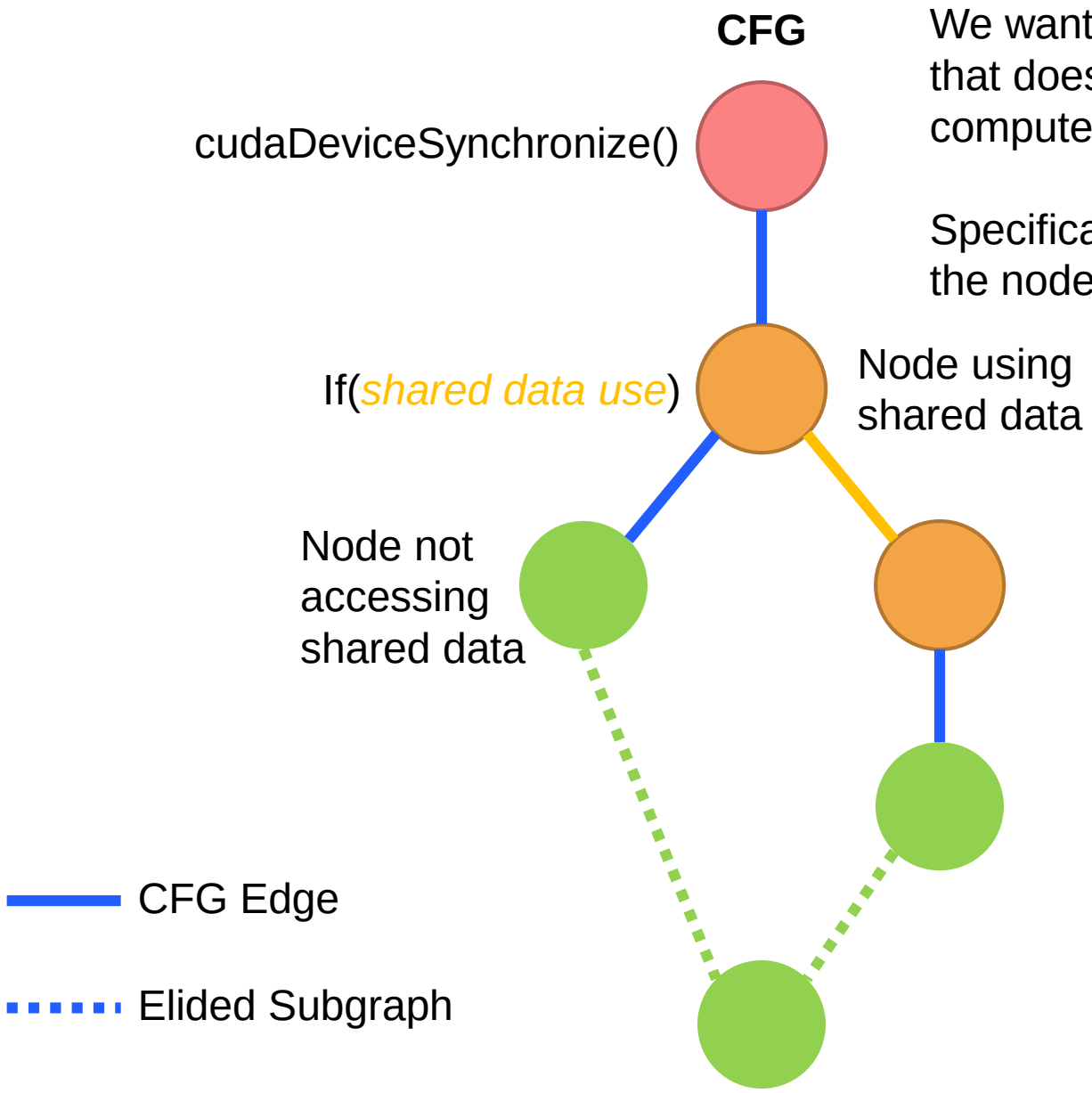


Detection of Synchronization Opportunities

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.

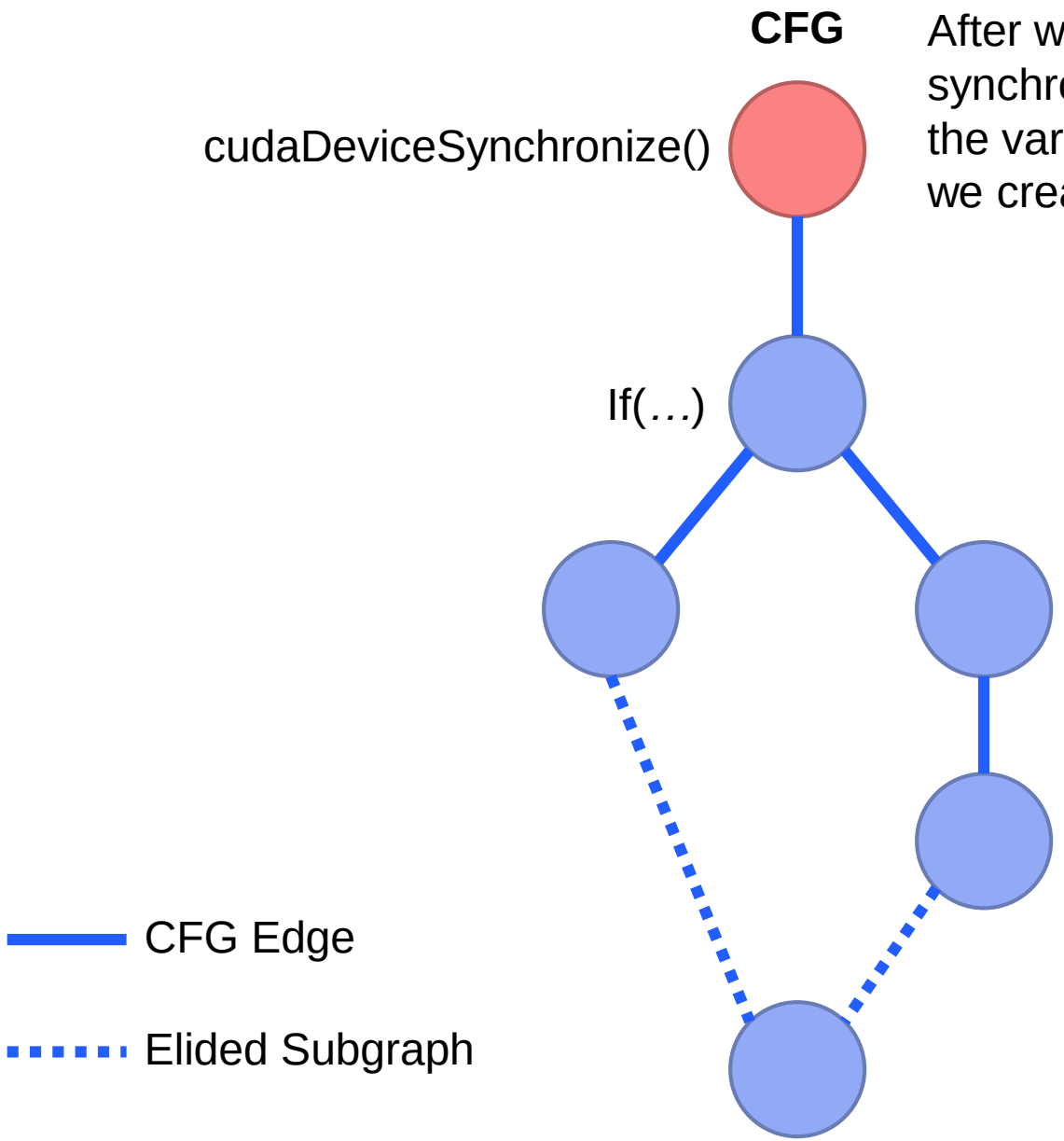
Program Slicing



We want to identify computation that does not need GPU data to compute correctly.

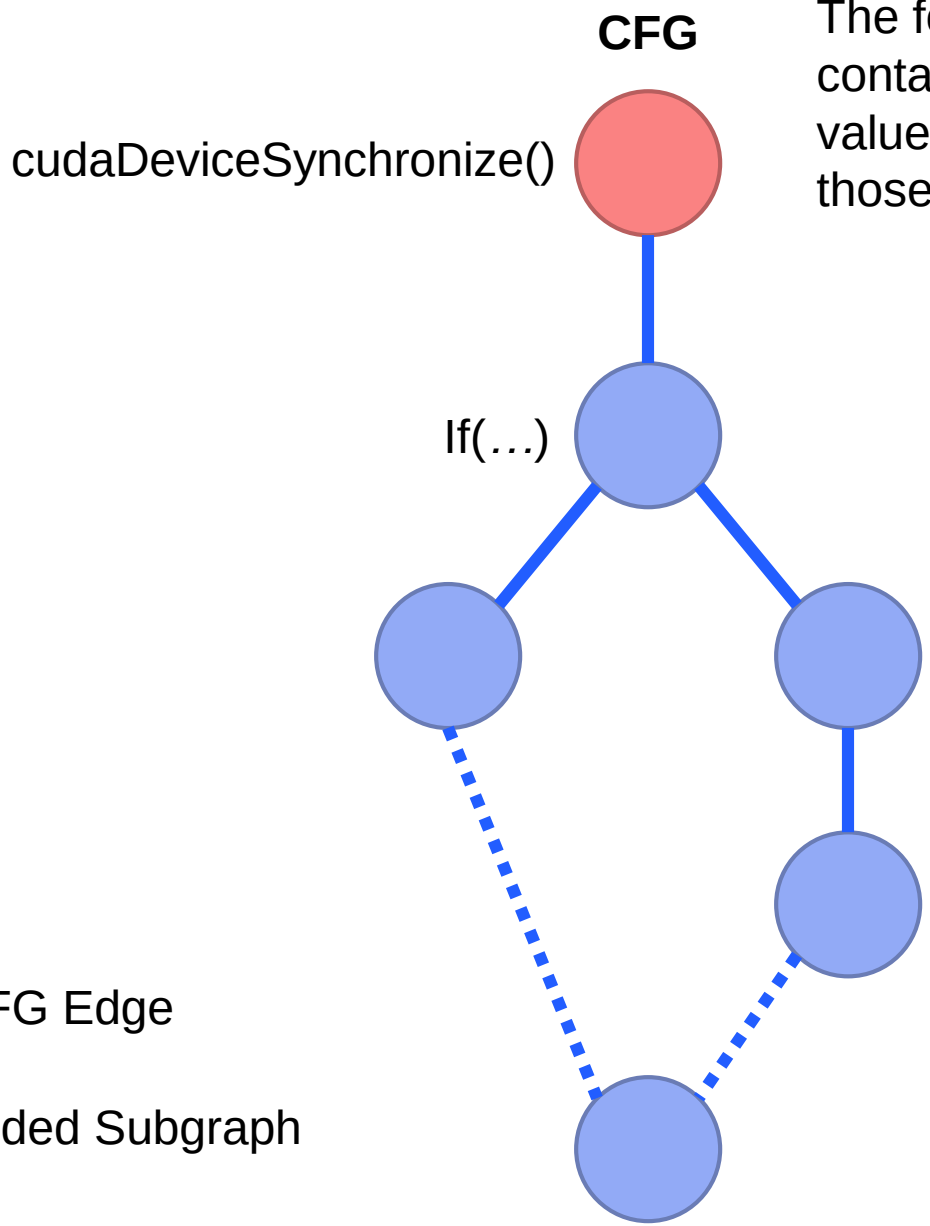
Specifically the number and size of the nodes in green.

Program Slicing



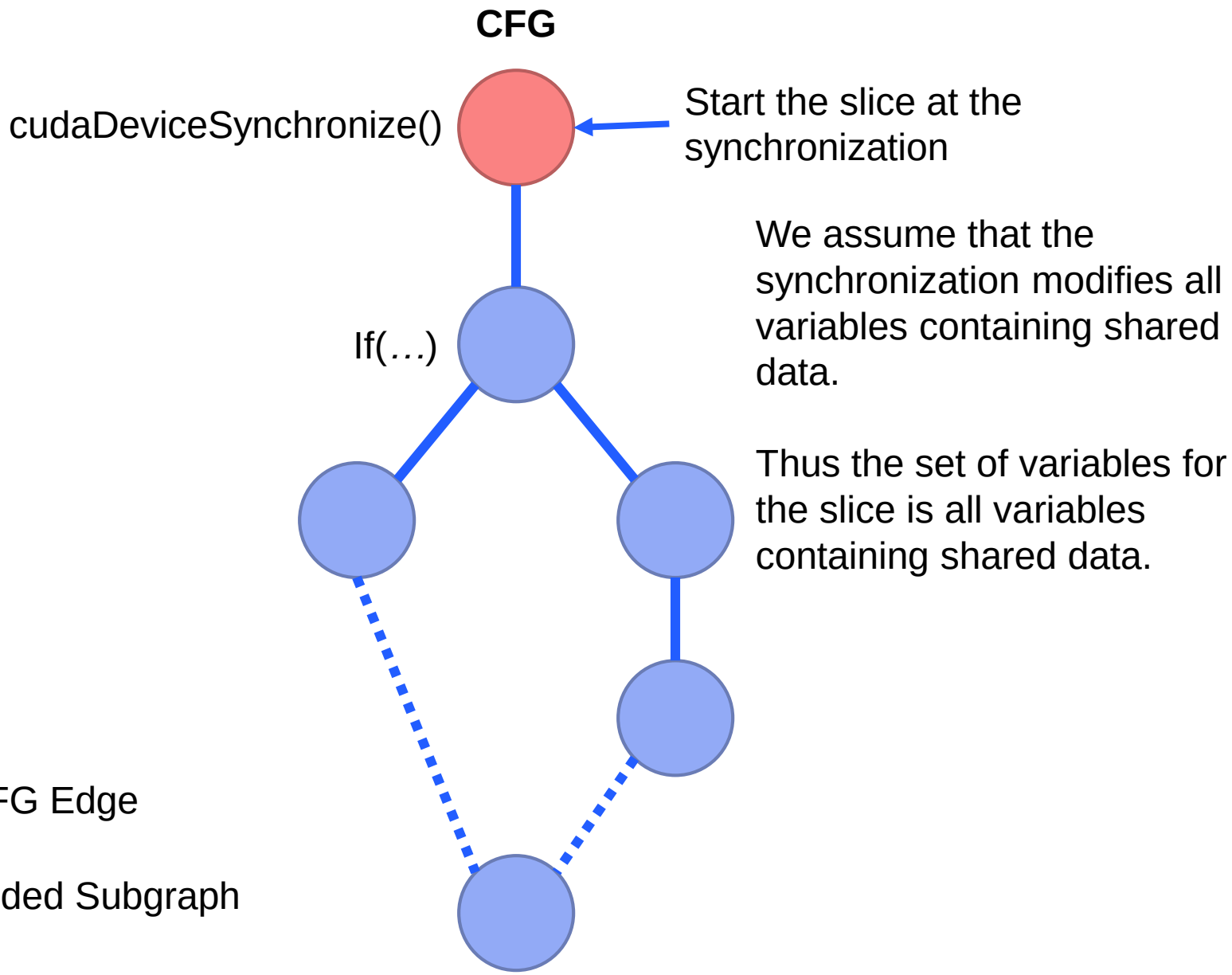
After we have identified synchronizations with long delay and the variables containing shared data we create a forward program slice.

Program Slicing

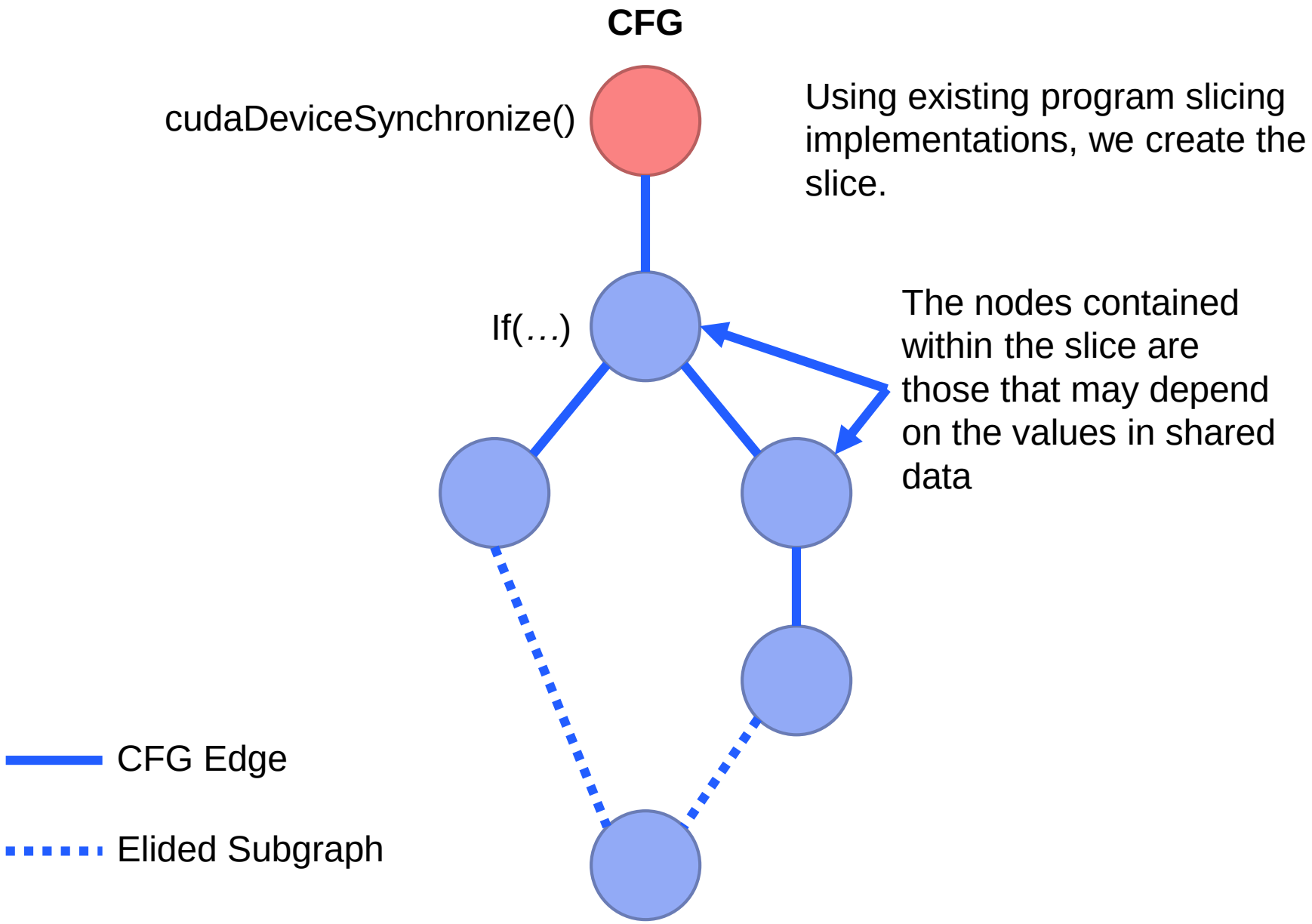


The forward program slice will contain all instructions with values that may depend on those stored in shared data.

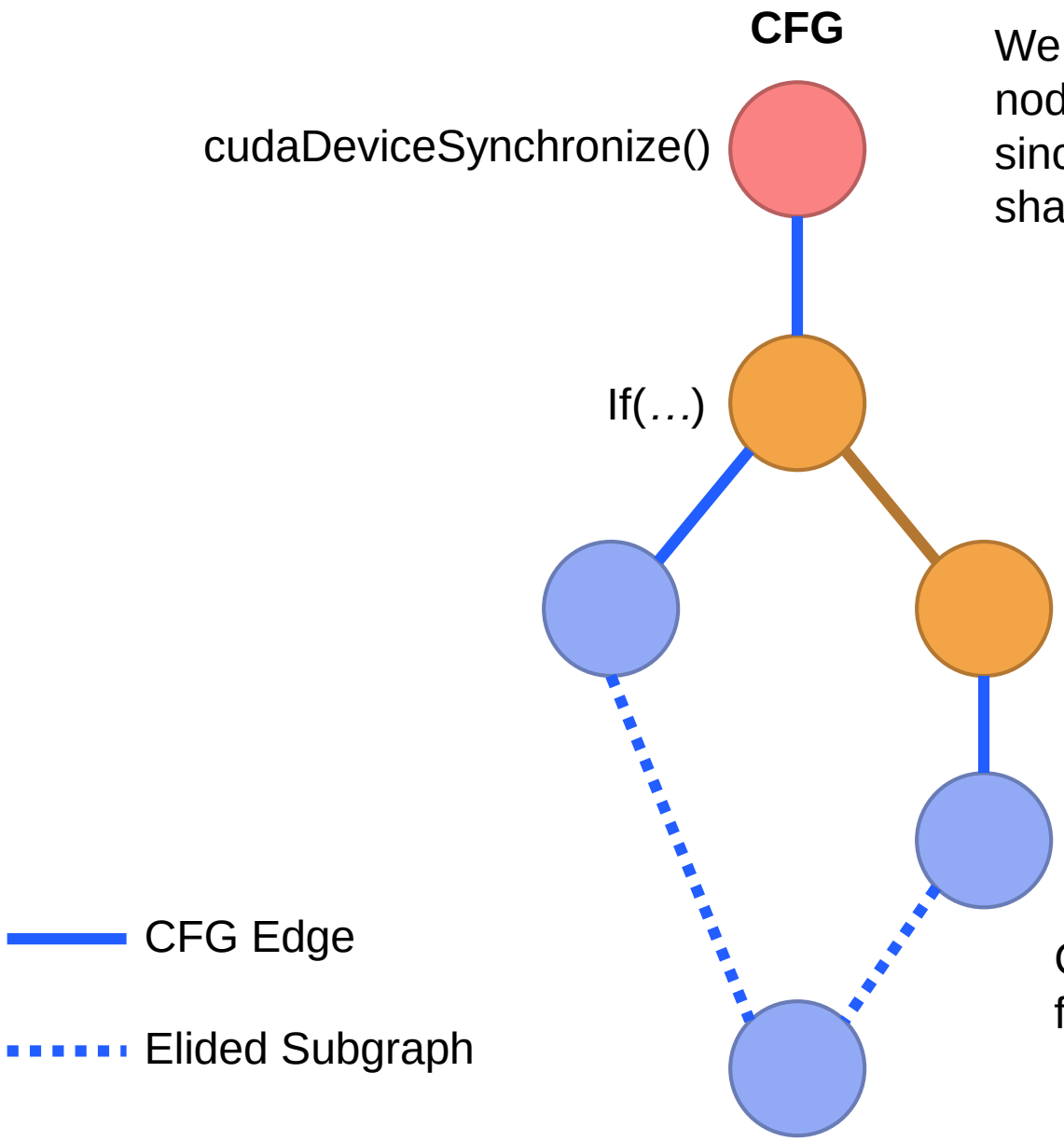
Program Slicing



Program Slicing



Program Slicing



We are really interested in the nodes not contained in the slice since they do not depend on shared data.

If a large number of nodes exist outside of the slice, a synchronization opportunity exists.

Computation can be moved in front of the synchronization

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

Correction of Duplicate Transfers

We cannot remove the duplicate transfers using binary modification

- No guarantee that the transfer will be a duplicate on subsequent executions of the same call.

Our correction would perform content based data deduplication on all transfers that were identified as containing duplicates.

- If a duplicate is detected, we instruct the GPU to make a local copy of the data.
- Saves a data transfer over the PCI-E bus.

Open Questions/Issues

- Can we hash data fast enough for content based deduplication to be feasible?
  - Preliminary Results: Approximately 3% execution time overhead from hashing (tensorflow using xxhash32)
- Can we determine if an identified duplicate transfer is always a duplicate to eliminate the need for a check on subsequent executions?

Detection of Unobvious Parallelization

- Two characteristics are needed for an unobvious parallelization opportunity to exist:
  - Loops with long execution times
  - A sequential memory access pattern for variables accessed within the loop.

LAMMPS Performance Analysis

- 20% of the execution time was spent in a single loop that was not parallelized.

This loop was not parallelized because GPUs do not perform well when multi-level pointers are present

```
for (int i = 0; i < nlocal; i++) {
    if (mask[i] & groupbit) {
        double dtfm;
        dtfm = dtf / mass[type[i]];
        v[i][0] += dtfm * f[i][0];
        v[i][1] += dtfm * f[i][1];
        v[i][2] += dtfm * f[i][2];
    }
}
```

Detection of Unobvious Parallelization

We want to identify if the memory access patterns within long running loops are vectorizable.

We want to know if the memory access within the loop are favorable to vectorization

```
for (int i = 0; i < nlocal; i++) {
    if (mask[i] & groupbit) {
        double dtfm;
        dtfm = dtf / mass[type[i]];
        v[i][0] += dtfm * f[i][0];
        v[i][1] += dtfm * f[i][1];
        v[i][2] += dtfm * f[i][2];
    }
}
```

Detection of Unobvious Parallelization

We will use load and store instrumentation to identify vectorizable memory access patterns:

- We will capture the addresses used to load and store values to/from main memory.
- If a continuous region of memory can be formed from these addresses, we consider the loop parallelizable.

Open Questions/Issues

- How long of an execution time must a loop have before it is considered favorable to convert to the GPU?
- Are there other characteristics that exist in unobvious parallelization opportunities that may better help identify their presence?

Just-in-time Compilation

- We need to identify when an application has been compiled with GPU code incompatible with the device used.

Just-in-time Detection

- We can detect a mismatch by comparing the GPU code architectures present in the application with those compatible with the card.
  - On linux each GPU code architecture is stored in its own ELF section of the application binary
    - The ELF header contains a the name of the architecture the code is compiled for.
  - The architecture of the GPU card in use can be queried from the device.

Summary

- Our research focuses on the development of techniques to detect and exploit GPU performance opportunities
- We have so far identified four performance opportunities in four applications
- We have proposed four solutions to detect and automatically correct the performance opportunities we have identified.