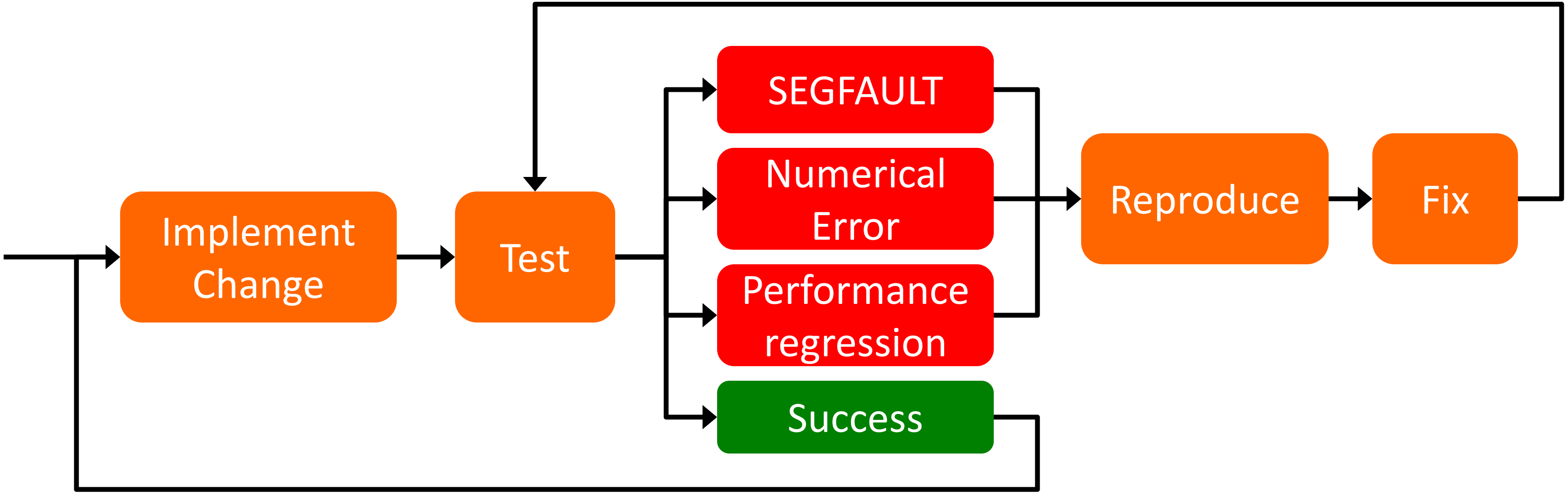
CHIMES: Efficient, Automatic Application Checkpointing as a Powerful Tool for CUDA Development

Max Grossman, Vivek Sarkar

Rice University

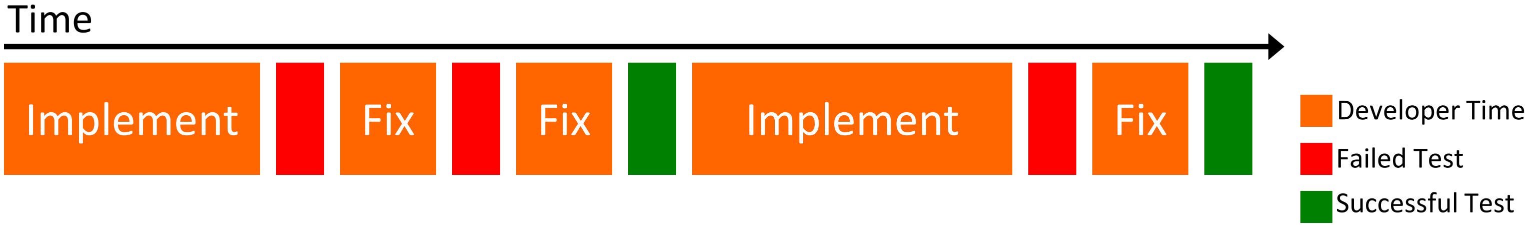
Motivation

Software development is incremental and reactive



Motivation

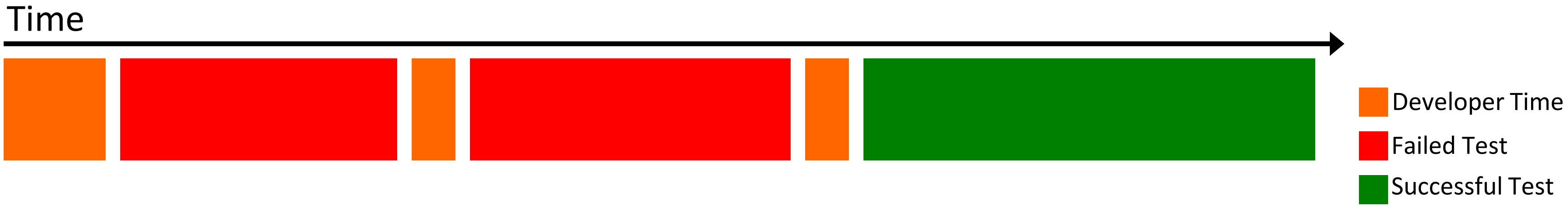
In many cases, this workflow isn’t a problem and does not waste developer time.



Still allows for rapid iteration on a feature.

Motivation

For large-scale applications and simulations, thoroughly testing new code and reproducing bugs dominates development time



There are techniques to counter this to some extent (e.g., unit testing, dataset subsetting) but as the complexity of an application increases, it becomes more and more difficult to thoroughly verify correctness quickly.

Motivation

In particular, let’s focus on three types of regressions:

1. Performance
2. Correctness
3. Numerical accuracy

Detecting/diagnosing these regressions can be made simpler and faster with **checkpoints of in-memory application state** that can be replayed.

This talk will discuss in-progress work on a powerfully simple framework for general-purpose checkpointing of C and CUDA programs, on top of which more powerful tools are planned.

What is a checkpoint?

A checkpoint contains most of the application state required to resume an arbitrary C/CUDA application from an intermediate point-in-time:

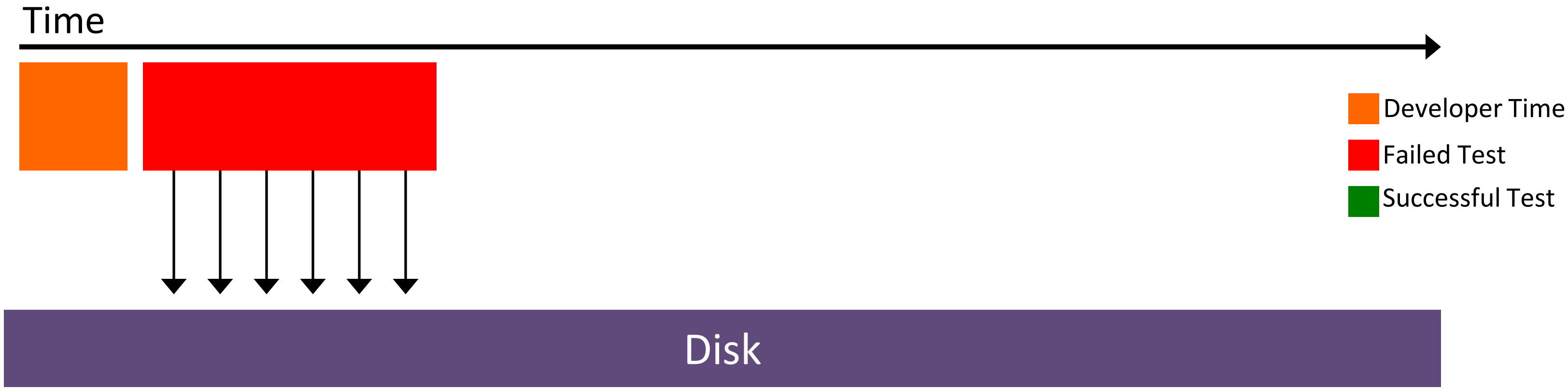
1. Per-thread stack variables
2. Heap allocations
3. CUDA device memory allocations
4. Global variables
5. Per-thread stack trace

A checkpoint is persisted on-disk for later recall.

A checkpoint can be loaded into the original application to resume execution from some intermediate point-in-time.

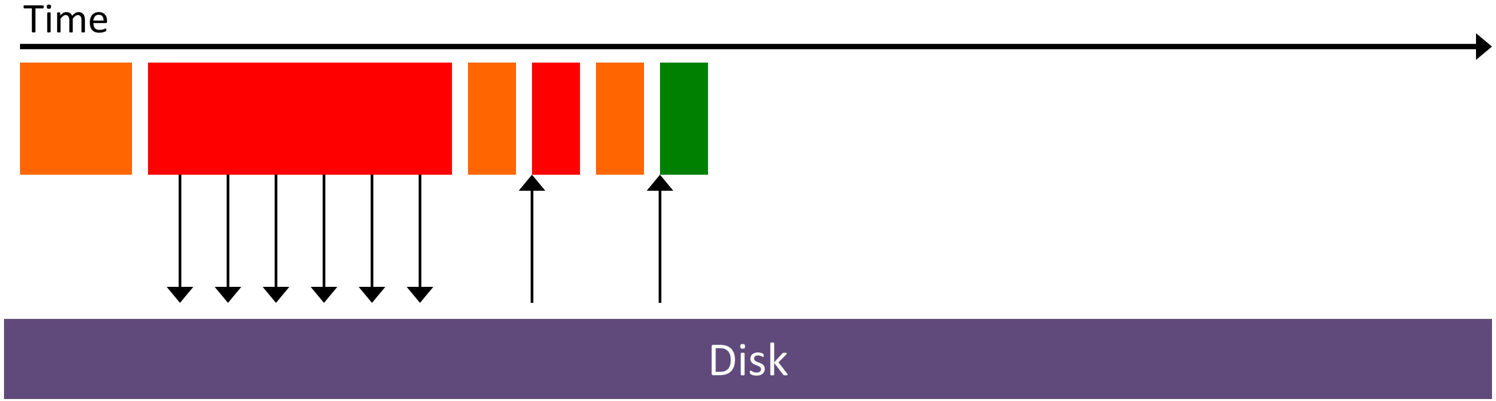
How are checkpoints useful

During application execution, checkpoints are periodically created and persisted.



How are checkpoints useful

When reproducing an error or testing a fix only the most recent checkpoint needs to be loaded, drastically cutting down on time-to-verify and making it simple to add a new regression test for this error.



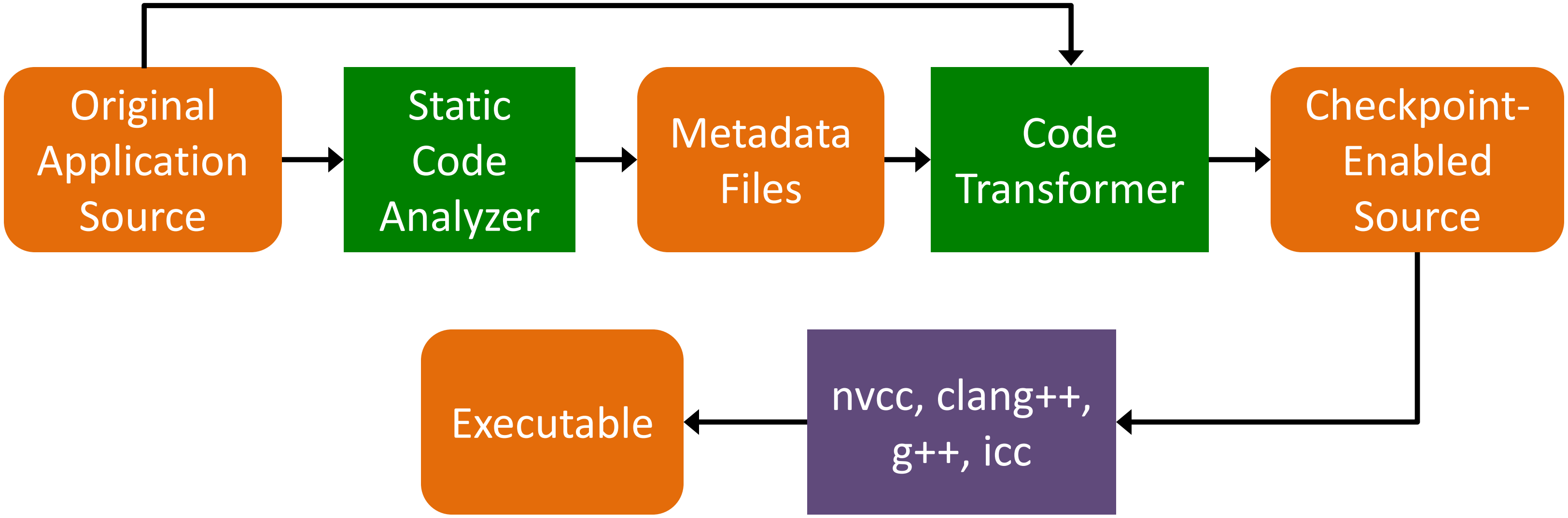
Functional Requirements

For CHIMES to be useful, it must:

1. Minimize performance impact on the executing application in terms of processor cycles, PCIe bandwidth, memory, disk bandwidth. Interference with the running application must be kept to a minimum to make this a feasible tool.
2. Minimize compiler and environmental dependencies. No compiler, operating system, virtualized environment requirements. Currently tested on MacOS and Linux; clang++, g++, nvcc.
3. Maximize usability and generality of the API and workflow to make it simple to build more complex tools on top of CHIMES.

CHIMES Implementation

CHIMES can be split into 3 components: a static code analyzer, a source-to-source code transformer, and a runtime library.



CHIMES Implementation

At runtime, the application is launched as usual:

```
$ ./a.out
```

When it completes, a number of checkpoint files have been created:

```
$ ls -la
-rw-r--r--   1 jmg3  staff  12783565 Feb  7 01:00 chimes.0.ckpt
-rw-r--r--   1 jmg3  staff  12783565 Feb  7 01:00 chimes.1.ckpt
-rw-r--r--   1 jmg3  staff  12783565 Feb  7 01:00 chimes.2.ckpt
...
```

CHIMES Implementation

The application can be re-launched from an intermediate point-in-time using an environment variable to load a specific checkpoint:

```
$ CHIMES_CHECKPOINT_FILE=chimes.5.ckpt ./a.out
```

The runtime library detects the resume, restores stack and heap state from the provided checkpoint file, and continues execution.

Demo

Current restrictions (AKA planned work)

- Support for CUDA texture memory and constant memory
- Support for multi-GPU applications
- Support for distributed applications
- Custom user resume logic
- OpenCL, OpenACC, pthreads, etc.
- Improve portability of checkpoints across platforms and across versions of the same application
- Continued expansion of testing suite
- Continued performance improvements

Potential Use Cases

- Efficiently reproducing application correctness errors
- Comparing checkpoints from legacy applications and CUDA ports makes it easier to detect and diagnose numeric errors (due to different compilers, architectures, etc.)
- Checkpoints allow you to rapidly iterate on hotspot optimizations or newly detected performance regressions
- Checkpoints may simplify resilient application development by making executable migration and resume easier

Conclusion

General-purpose checkpointing of applications is a powerful tool for a number of use cases.

This is a work in-progress, but is already demonstrating promising results on examples of basic to intermediate complexity.

Future work will expand the scope of C/CUDA features supported and build high-level tools on top of this checkpointing utility.

Contact: Max Grossman, jmg3@rice.edu