Verifying DRF in GPU programs

Tiago Cogumbreiro, UMass Boston

Joint work with

Julien Lange, Royal Holloway, University of London

Dennis Liew Zhen Rong, UMass Boston

Hannah Zicarelli, UMass Boston

March 10, 2021

Brunel, Univerisity of London

Today's talk

UMASS BOSTON

Motivation

- Why do GPUs matter?
- Why we should care about static verification of GPU programs?

Contributions

- By how much do we improve the state of the art?

Our technique

- How we improve the state of the art

Motivation

Why do GPUs matter?

GPUs are everywhere

GPUs are a computing cornerstone of scientific advancement

GPUs in High Performance Computing (HPC)

UMASS BOSTON

Power 8 out of 10 of the Top 10 super computers

	Name	GPU
1	Supercomputer Fugaku	☐
2	Summit	☑
3	Sierra	☑
4	Sunway TaihuLight	☐
5	Selene	☑
6	Tianhe-2A	☑
7	JUWELS Booster Module	☑
8	HPC5	☑
9	Frontera	☑
10	Dammam-7	☑

www.top500.org/lists/top500/2020/11/highs/

Credit: Carlos Jones/ORNL

GPUs powering chemestry

Journal of Molecular Graphics and Modelling 29 (2010) 116–125

Contents lists available at ScienceDirect

Journal of Molecular Graphics and Modelling

journal homepage: www.elsevier.com/locate/JMGM

Topical perspectives

GPU-accelerated molecular modeling coming of age

John E. Stone [a], David J. Hardy [a], Ivan S. Ufimtsev [b], Klaus Schulten [c,*]

[a] Beckman Institute, University of Illinois at Urbana-Champaign, 405 N. Mathews Ave., Urbana, IL 61801, United States
[b] Department of Chemistry, Stanford University, 333 Campus Drive, Stanford, CA 94305, United States
[c] Department of Physics, University of Illinois at Urbana-Champaign, 1110 W. Green, Urbana, IL 61801, United States

ARTICLE INFO

Article history:
Received 9 February 2010
Received in revised form 24 June 2010
Accepted 30 June 2010
Available online 8 July 2010

Keywords:
GPU computing
Molecular modeling
Molecular dynamics
Quantum chemistry
Molecular graphics

ABSTRACT

Graphics processing units (GPUs) have traditionally been used in molecular modeling solely for visualization of molecular structures and animation of trajectories resulting from molecular dynamics simulations. Modern GPUs have evolved into fully programmable, massively parallel co-processors that can now be exploited to accelerate many scientific computations, typically providing about one order of magnitude speedup over CPU code and in special cases providing speedups of two orders of magnitude. This paper surveys the development of molecular modeling algorithms that leverage GPU computing, the advances already made and remaining issues to be resolved, and the continuing evolution of GPU technology that promises to become even more useful to molecular modeling. Hardware acceleration with commodity GPUs is expected to benefit the overall computational biology community by bringing teraflops performance to desktop workstations and in some cases potentially changing what were formerly batch-mode computational jobs into interactive tasks.

© 2010 Elsevier Inc. All rights reserved.

doi:10.1016/j.jmgm.2010.06.010

GPUs powering biology

UMASS BOSTON

BRIEFINGS IN BIOINFORMATICS. VOL 11. NO 3. 323–333
Advance Access published on 7 March 2010

doi:10.1093/bib/bbq006

GPU computing for systems biology

Lorenzo Demattè and Davide Prandi

Submitted: 20th November 2009; Received (in revised form): 30th January 2010

Abstract

The development of detailed, coherent, models of complex biological systems is recognized as a key requirement for integrating the increasing amount of experimental data. In addition, in-silico simulation of bio-chemical models provides an easy way to test different experimental conditions, helping in the discovery of the dynamics that regulate biological systems. However, the computational power required by these simulations often exceeds that available on common desktop computers and thus expensive high performance computing solutions are required. An emerging alternative is represented by general-purpose scientific computing on graphics processing units (GPGPU), which offers the power of a small computer cluster at a cost of ~$400. Computing with a GPU requires the development of specific algorithms, since the programming paradigm substantially differs from traditional CPU-based computing. In this paper, we review some recent efforts in exploiting the processing power of GPUs for the simulation of biological systems.

Keywords: *systems biology; simulation; agent-based modelling; cellular automata; GPGPU; CUDA*

doi:10.1093/bib/bbq006

GPUs power the AI revolution

Autoware.AI

> Autoware.AI is the world's first "All-in-One" open-source software for autonomous driving technology.

22 code results in Autoware-AI/core_perception Sort: Best match

ndt_gpu/src/MatrixDevice.cu
Cuda Last indexed on Oct 15, 2020

ndt_gpu/src/SymmetricEigenSolver.cu
Cuda Last indexed on Oct 15, 2020

vision_darknet_detect/darknet/src/dropout_layer_kernels.cu
Cuda Last indexed on Oct 15, 2020

vision_darknet_detect/darknet/src/col2im_kernels.cu
Cuda Last indexed on Oct 15, 2020

Why we should care about static verification of GPU programs?

GPU programming, a primer

① High-level of parallelism at a reduced cost

(faster processing, lower cost, reduced power consumption)

② Techniques designed for CPUs do not work for GPUs

(hardware assumptions differ: memory available, execution model)

③ GPUs are difficult to program and debug

GPU programming is difficult

UMASS BOSTON

- high degree of parallelism (up to tens of thousand of threads)
- high degree of concurrency (up to 1,024 threads accessing the same array)
- **unconstrained** access to a shared memory (no locks)
- thousands of threads indexing disjoint portions of arrays
- devices are memory constrained (affects debugging techniques)

GPU program example

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM; i+=BLOCK_ROWS)
   {tile[tid.y+i][tid.x] = idata[index_in+i*width];}
  __syncthreads();
  for (int j = 0; j<TILE_DIM; j+=BLOCK_ROWS)
   {odata[index_out+j*height] = tile[tid.x][tid.y+j];}}
```

Source:

- Optimizing matrix transpose in CUDA. NVIDIA CUDA SDK Application Note 18 (2009)

Also in:

- Padding free bank conflict resolution for CUDA-based matrix transpose algorithm. DOI: 10.1109/SNPD.2014.6888709

GPU program example

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM; i+=BLOCK_ROWS)
   {tile[tid.y+i][tid.x] = idata[index_in+i*width];}
  __syncthreads();
  for (int j = 0; j<TILE_DIM; j+=BLOCK_ROWS)
   {odata[index_out+j*height] = tile[tid.x][tid.y+j];}}
```

GPU program example

thread (0,1)

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM; i+=BLOCK_ROWS)
   {tile[ 0 +i][ 1 ] = idata[index_in+i*width];}
  __syncthreads();
  for (int j = 0; j<TILE_DIM; j+=BLOCK_ROWS)
   {odata[index_out+j*height] = tile[ 0 ][ 1 +j];}}
```

thread (1,0)

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM; i+=BLOCK_ROWS)
   {tile[ 1 +i][ 0 ] = idata[index_in+i*width];}
  __syncthreads();
  for (int j = 0; j<TILE_DIM; j+=BLOCK_ROWS)
   {odata[index_out+j*height] = tile[ 1 ][ 0 +j];}}
```

GPU data-races

UMASS BOSTON

Data-race

- Two threads accessing the same array index concurrently
- At least one thread writing

Data-Race Freedom (DRF) analysis

Show that for all possible inputs and executions a program is absent of data-races.

A trivial data-race example (every thread writes to postion 0)

```
A[0] = 1;
```

GPU program example

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM; i+=BLOCK_ROWS)
   {tile[tid.y+i][tid.x] = idata[index_in+i*width];}
  __syncthreads();
  for (int j = 0; j<TILE_DIM; j+=BLOCK_ROWS)
   {odata[index_out+j*height] = tile[tid.x][tid.y+j];}}
```

> Exhibits a data-race: the code after `__syncthreads()` of iteration `i + 1` runs concurrently with the code before `__syncthreads()` of iteration `i`.

- Outer loops is used to measure the benefit of an optimization
- Data-race **corrupts** the data in the array and affects the time measurements

Contributions

Contributions

Theoretical (Part 2)

- a novel analysis of data-race freedom
- a formalization of such analysis using a proof assistant

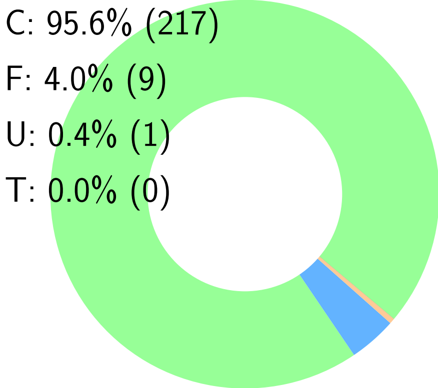
Practical (Part 1)

- an implementation of the analysis (Part 1)
- the largest comparative study of its kind (Part 1)

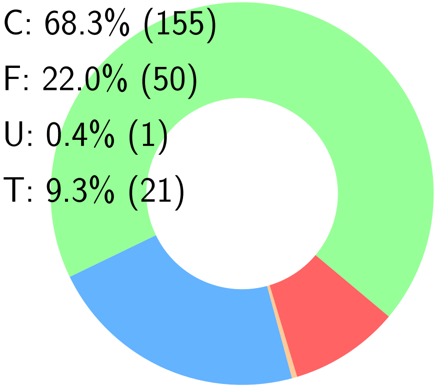
Lowest false-positive rate

- Dataset of 227 data-race free real-world kernels
- Can verify 41% more kernels than others

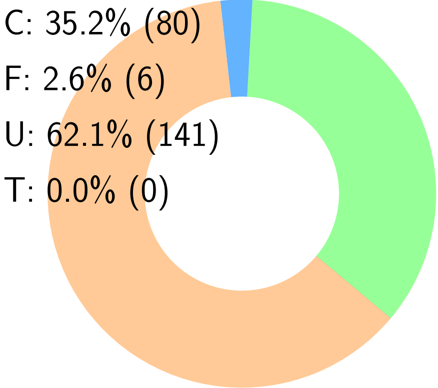
Faial (our tool)



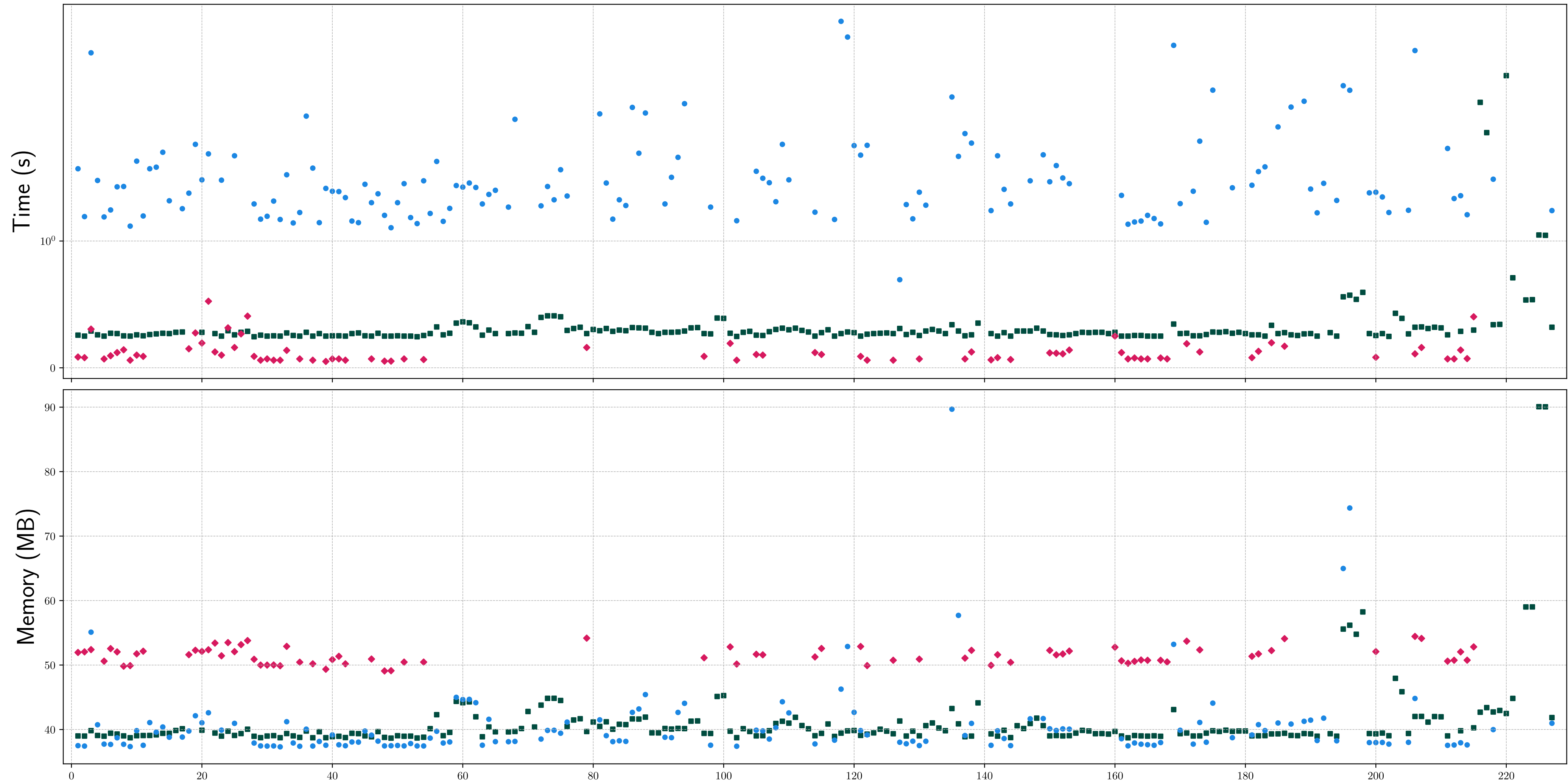
GPUVerify



PUG

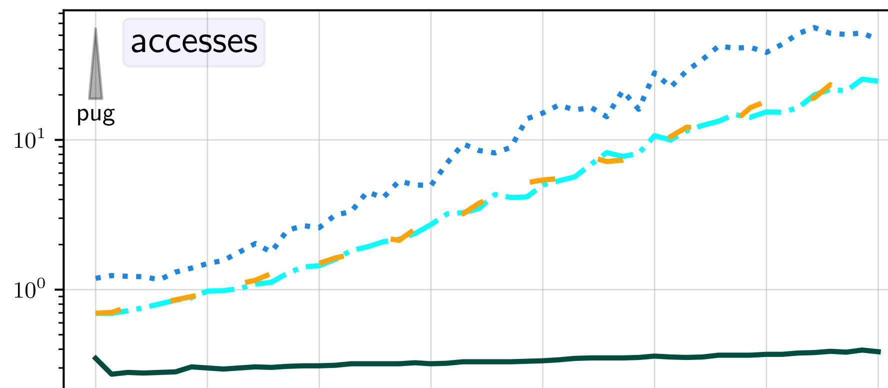


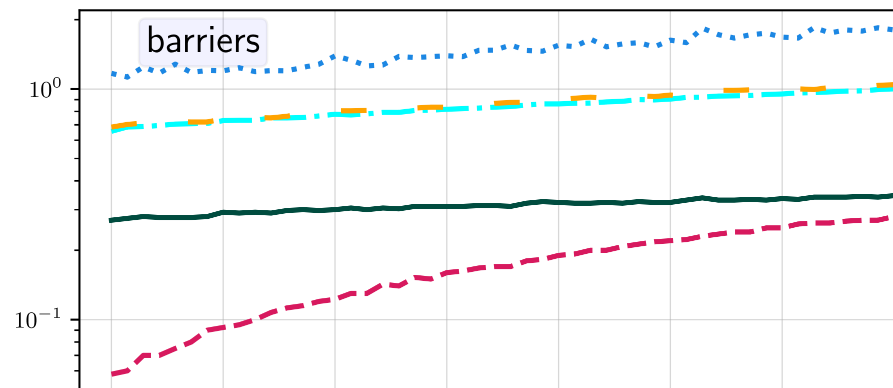
Best compromise time/memory

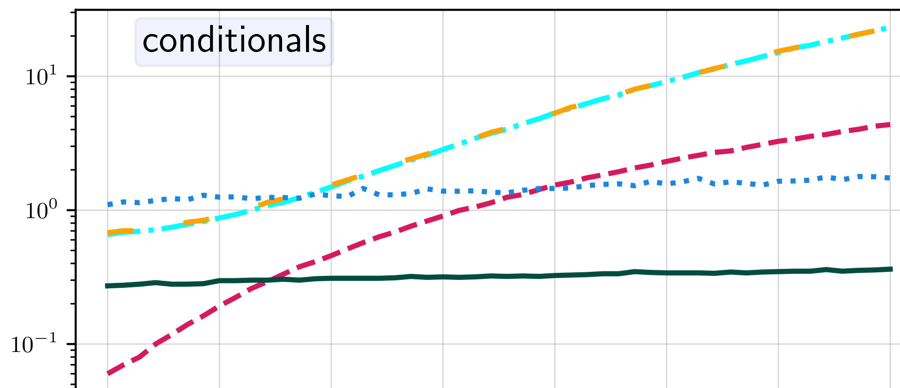


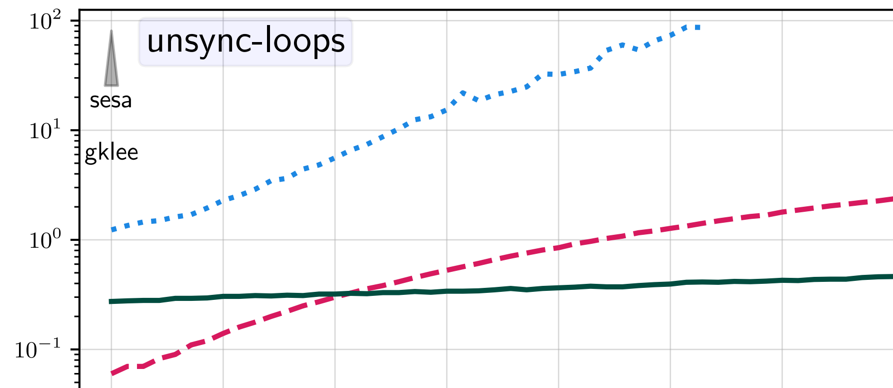
Highest scalability

- Vary the number of constructs from 1 to 50 (250 kernels in total)
- Out of 5 tools, the only that **scales linearly** (time) (PUG, GPUVerify, GKlee, SESA)



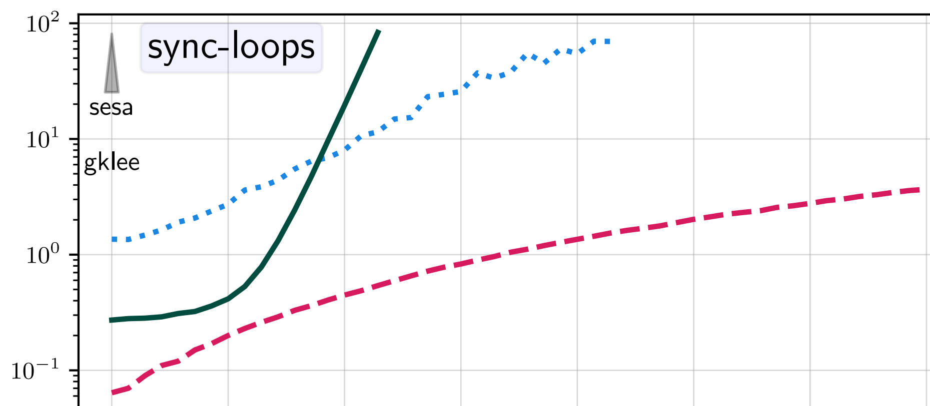




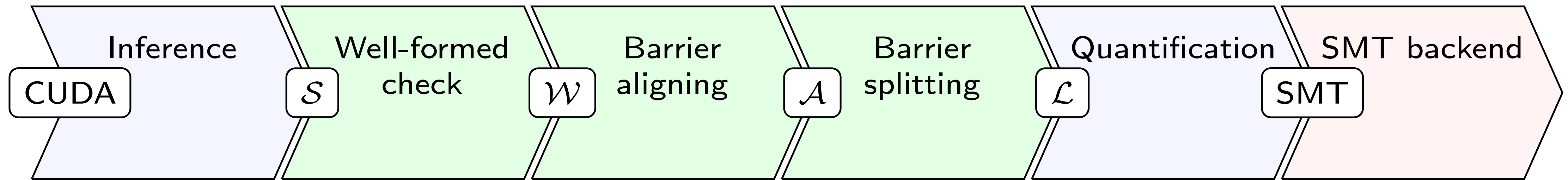


Limitations of our analysis

- Cannot handle more than 13 nested synchronized loops
- 3rd out of 5 tools
- We found a maximum nesting level of 3 in our experiments



Our approach

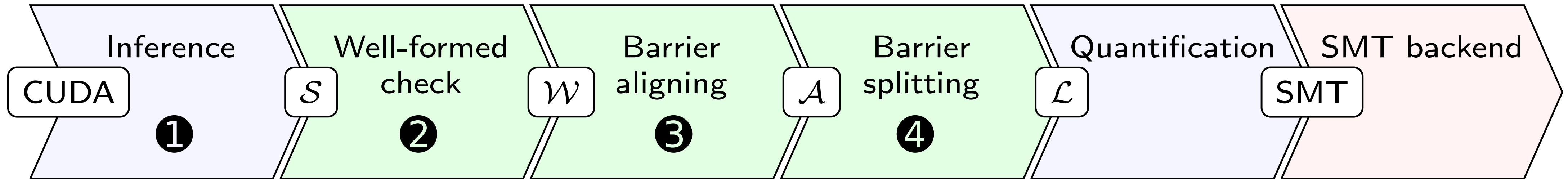


- A behavioral type (syntax+semantics)
- Results on the correctness of the analysis
- Mechanized proofs using the Coq proof assistant (18,000 LOC)

Simplified running example

- A CUDA example, which simplifies our initial example
- Exhibits the same root cause (data-race)

```
1 for (int r = 0; r < N; r++) {
2   for (int i = 0; i<M; i++)
3     { tile[tid] = ...;}
4   __syncthreads();
5   for (int j = 0; j<M; j++)
6     {... = tile[tid+j];}}
```




```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<M; i++)
    { tile [ tid ] = ...;}
    __syncthreads();
  for (int j = 0; j<M; j++)
    {... = tile [ tid+j ];}}
```
❶

```
for^S r in 0..N {
  for^U i in 0..M { wr[tid] }
  sync;
  for^U j in 0..M { rd[tid + j] }
}
```
❷

```
for^U i in 0..M { wr[tid] }
sync;
for^S r in 1..N {
  for^U j in 0..M { rd[tid + j] }
  for^U i in 0..M { wr[tid] }
  sync; }
for^U j in 0..M { rd[tid + j] }
```
❸

$$\forall r, j_1, i_1, j_2, i_2 \colon 1 \leq r < N \wedge 0 \leq j_1 < M \wedge 0 \leq i_1 < M \wedge 0 \leq j_2 < M \wedge 0 \leq i_2 < M$$

$$\implies \{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$
❹

Memory access protocols

- Behavioral types for SIMT/SPMD that capture memory accesses
- One type per array. Capture: accesses, synchronization, structured loops
- Distinguish between synchronized/unsynchronized loops

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<M; i++)
    {tile[tid] = ...;}
  __syncthreads();
  for (int j = 0; j<M; j++)
    {... = tile[tid+j];}}
```
❶

```
for^S r in 0..N {
  for^U i in 0..M { wr[tid] }
  sync;
  for^U j in 0..M { rd[tid + j] }
}
```
❷

The data-race protocol

```
for^S r in 0..N {
  for^U i in 0..M { wr[tid] }
  sync;
  for^U j in 0..M { rd[tid + j] }
}
```

```
// r = 0
for^U j in 0..M           // for (int j = 0; j<M; j++)
  {rd[tid+j]};             //   _ = tile[tid+i];
// r = 1
for^U i in 0..M           // for (int i = 0; i<M; i++)
  {wr[tid]}                //   tile[tid] = _;
```

Proving that data-race exists

UMASS BOSTON

- Interpret **unsynchronized** loops as forall-binders:
  - compare one iteration of each loop of each thread
  - collapses all the iterations of a single loop into one
- One formula per thread; **data-race:** $t_1 = 0, t_2 = 1, j_1 = 1, M > 1$: `rd[1]` and `wr[1]`

```
// r = 0
for^U j in 0..M        // for (int j = 0; j<M; j++)
  {rd[tid+j]};         //   _ = tile[tid+i];
// r = 1
for^U i in 0..M        // for (int i = 0; i<M; i++)
  {wr[tid]}            //   tile[tid] = _;
```

$$\forall j_1, i_1, j_2, i_2 : 0 \leq j_1 < M \land 0 \leq i_1 < M \land 0 \leq j_2 < M \land 0 \leq i_2 < M \implies$$

$$\{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$

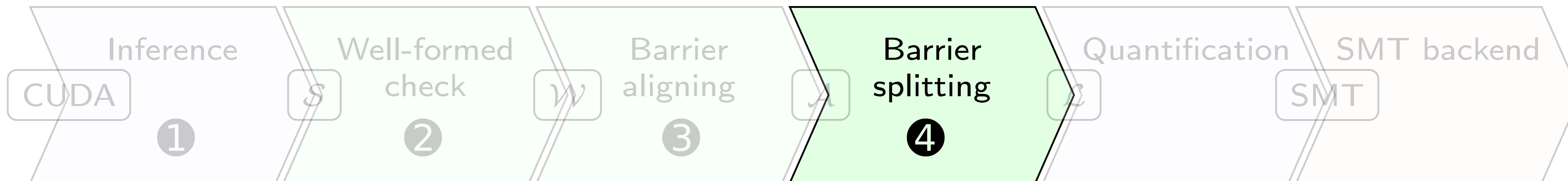
Aligning protocols

- We define a notion of **aligned** protocols, where accesses do not "leak" across iterations
- We show that all protocols can be aligned (modulo notion of well-formedness)
- **Intuition:** unfold loop and rearrange accesses

```
for^S r in 0..N {
  for^U i in 0..M { wr[tid] }
  sync;
  for^U j in 0..M { rd[tid + j] }
}
```
❷

```
for^U i in 0..M { wr[tid] }
sync;
for^S r in 1..N {
  for^U j in 0..M { rd[tid + j] }
  for^U i in 0..M { wr[tid] }
  sync; }
for^U j in 0..M { rd[tid + j] }
```
❸

Splitting protocols




```
forU i in 0..M { wr[tid] }
sync;
forS r in 1..N {
  forU j in 0..M { rd[tid + j] }
  forU i in 0..M { wr[tid] }
  sync; }
forU j in 0..M { rd[tid + j] }
```

❸

$$\forall r, j_1, i_1, j_2, i_2 : 1 \leq r < N \land 0 \leq j_1 < M \land 0 \leq i_1 < M \land 0 \leq j_2 < M \land 0 \leq i_2 < M$$

$$\implies \{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$

❹

Splitting protocols

UMASS BOSTON

- Syntax-oriented extraction of unsynchronized fragments
- **Compositional** analysis (no data-races between fragments)
- Synchronized loop variables can also be interpret as a forall-binder
- However, the binder must be shared by both threads (ie, only one r variable shared by both threads)

Conclusion

- Behavioral types being used to enforce data-race freedom
- A compositional analysis, formally proved
- Large experimental evaluation (229 real-world + 258 synthetic = 487 kernels)
- Used our tool to confirm data-races found in the wild
- Our approach is more scalable and more precise (fewer false-positives) than related work
- Source code and proofs available in a free software license

https://gitlab.com/umb-svl/faial
https://gitlab.com/umb-svl/faial-coq