Scaling Data-Race Freedom Analysis with Array Projections

Paul Maynard   Tiago Cogumbreiro

University of Massachusetts Boston

SERPL 2023

Overview

- Data-Race Freedom in GPU kernels.
- Array projections – nonlinear expressions.
- Limitations of SMT solvers.
- Translating expressions to multidimensional accesses.
- Opens up analysis of a broader variety of programs.

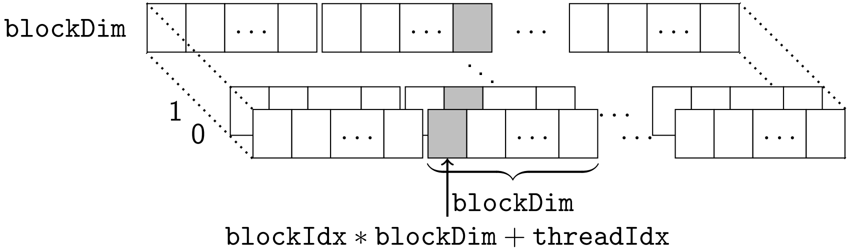
GPU programming

- Massively parallel execution (thousands of threads).
- Blocks of threads.
- Single Instruction, Multiple Threads.
- Thread-global memory and thread-local memory.

GPU programming model

```
void scal(float a, float *x)
{
  int i = blockIdx*blockDim + threadIdx;
  x[i] = a*x[i];
}
```

- Each thread runs this function with the same arguments.
- threadIdx is different for each thread



What is a data-race?

- Concurrent programs with shared memory can have data-races.
- Occur when two threads access the same memory location at the same time.
- At least one thread is writing.
  - One write and one write – data-race.
  - One write and one read – data-race.
  - One read and one read – no data-race.

Data-Race Freedom (DRF)

To prove a program is data-race free:

- Examine all concurrent memory accesses.
- Any accesses between synchronization points.
- Prove that all pairs of accesses are safe with each other.

Checking DRF as SMT formulas

1. Find set of concurrent accesses.
2. For each 2 elements in set, can indices be the same?
3. Translate accesses into an SMT (Satisfiability Modulo Theories) formula.
   - If formula satisfiable, then data-race.
   - If formula unsatisfiable, then DRF.

- Faial – `https://gitlab.com/umb-svl/faial`.

Example – checking DRF

```
void scal(float a, float *x)
{
  int i = blockIdx*blockDim + threadIdx;
  x[i] = a*x[i]; // read x[i], write x[i]
}
```

$$\mathtt{tid}_1 \neq \mathtt{tid}_2 \wedge \mathtt{bid} \cdot \mathtt{bdim} + \mathtt{tid}_1 = \mathtt{bid} \cdot \mathtt{bdim} + \mathtt{tid}_2$$

- *Thread-local variables are instantiated* for 2 symbolic threads. [FSE'10]
- `bid` and `bdim` are constant across threads, so this is solvable.
- No possible values that could satisfy this formula.
- Therefore, it is DRF.

Challenge – Nonlinearity

Projections of multidimensional accesses

- A more complicated problem:

```
void kernel(float* paths, int S, int T) {
  int step = gridDim * blockDim;
  for (int i = threadIdx; i < S ; i += step) {
    for (int t = 0; t < T ; t++) {
        paths[i + S * t] = f(t);
    }         ^^^^^^^^^
  }
}
```

$$i_1 < S \wedge i_2 < S \wedge i_1 + S \cdot t_1 = i_2 + S \cdot t_2$$

- This time the access involves thread local variables.

Nonlinear formulas

$$\mathtt{i}_1 < \mathtt{S} \land \mathtt{i}_2 < \mathtt{S} \land \mathtt{i}_1 + \mathtt{S} \cdot \mathtt{t}_1 = \mathtt{i}_2 + \mathtt{S} \cdot \mathtt{t}_2$$

- This time the variables in the multiplication are thread local.
- Multiplication introduces undecidability.
- SMT solvers can't solve all nonlinear expressions.

Nonlinear formulas

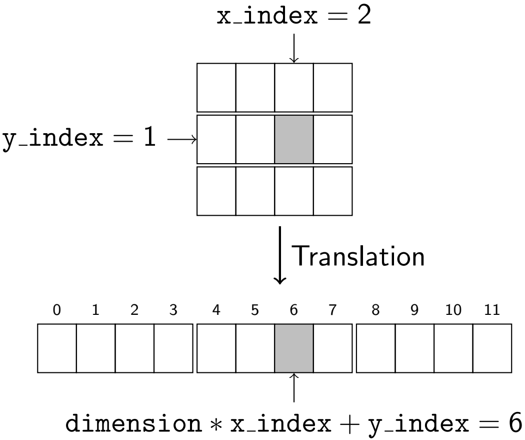
$$\mathtt{i}_1 < \mathtt{S} \wedge \mathtt{i}_2 < \mathtt{S} \wedge \mathtt{i}_1 + \mathtt{S} \cdot \mathtt{t}_1 = \mathtt{i}_2 + \mathtt{S} \cdot \mathtt{t}_2$$

- This time the variables in the multiplication are thread local.
- Multiplication introduces undecidability.
- SMT solvers can't solve all nonlinear expressions.

Array projections

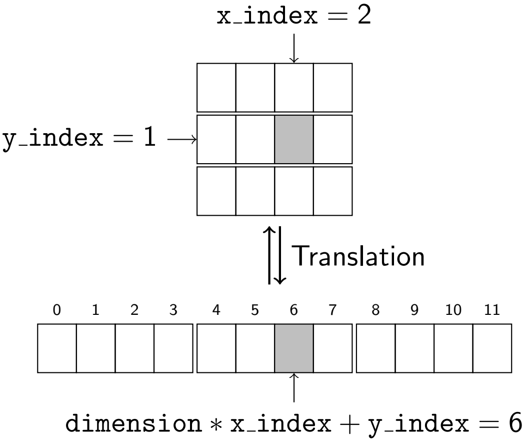
Multidimensional array representation

- Many nonlinear accesses are actually projections of 2d arrays.



Multidimensional array representation

- Many nonlinear accesses are actually projections of 2d arrays.



- Our contribution: To make these solvable by SMT solvers, we can reverse this transformation.

Rewriting accesses

```
void kernel(float paths[][], int S, int T) {
  int step = gridDim * blockDim;
  for (int i = threadIdx; i < S ; i += step) {
    for (int t = 0; t < T ; t++) {
      paths[t][i] = f(t);
    }      ^^^^^^
  }
}
```

$$i_1 < S_1 \wedge i_2 < S_2 \wedge i_1 = i_2 \wedge t_1 = t_2$$

- This formula is now analyzable.

Rewriting projections

How do we rewrite?

1. Identifying patterns (potentially obfuscated or nested).

$$\texttt{dimension} * \texttt{x\_index} + \texttt{y\_index}$$

2. Identifying potential rewrites.
3. Determining which rewrites are sound.
4. Choosing the appropriate rewrite.

$$\mathsf{A[a * b + c]} \implies \mathsf{A[a][c]}$$

or

$$\mathsf{A[a * b + c]} \implies \mathsf{A[b][c]}$$

Identifying projections

- One option: hard-coded patterns.
- Doesn't catch all instances.
  - For example:

    ```
    100 * (a * b + c) + 1
    ```

  - How often do nonstandard instances occur in practice?

Rewrite rules

- Rewrite expressions using series of rules
- Put all expressions into a normal form.
- Identify projections from normal form.
- Example:

```
100 * (a * b + c) + 1
==> 100 * a * b + 100
```

Egraphs

- Equivalence-graphs.
- Track equivalence classes of expressions.
- Allow fast and efficient rewriting of terms. [egg'21]

Soundness and ambiguity

When is rewriting sound?

- Need to make sure we don't remove data-races.
  - Example:

    ```
    A[a * b + c] ==> A[a][c]
    ```

    when

    $$\mathrm{a} = 1 \qquad \mathrm{b} = 5 \qquad \mathrm{c} = 6$$

    ```
    A[1 * 5 + 6] ==> A[1][6]
    A[11]
    A[2 * 5 + 1] ==> a[2][1]
    ```

  - Must guarantee that $c < b$
- Dimensions must be same across all accesses.

Ambiguous expressions

```
100 * a * b + 100 * c
```

- What is the index, and what is the dimension?
  - 6 possible interpretations
- Heuristics:
  - Prefer thread globals and constants as dimension.
  - Prefer loop variables as indices.
  - Soundness rules can help choose:
    - Must have same dimension across all accesses
    - Y-index must be less than the dimension.
- Could send multiple interpretations to the solver.

Open questions/Future work

- Testing/analysis on GPUVerify Cav14 dataset.
  - What patterns exist, and in what proportion?
  - What level of rewriting/searching is needed?
- Can this fix be automatically applied to most programs?
- Is it sound, or are there data-races it eliminates?
- Use in additional contexts
  - Code repair/synthesis.
  - Precondition inference.

Conclusion

- Data-races in GPU programming.
- Array projections.
- Rewriting projections back to multidimensional accesses.
  - Recognizing & rewriting patterns.
  - Soundness of rewrites.
  - Handling ambiguity.