TOWARDS CONCURRENCY REPAIR IN GPU KERNELS WITH RESOURCE COST ANALYSIS

GREGORY BLIKE*

TIAGO COGUMBREIRO

UMASS BOSTON

OCTOBER 14, SERPL23

OVERVIEW

GPU Programming is uncontestedly performant, however, that performance comes with the challenge of avoiding many classes of programming errors, in particular today, data races.

Data races are readily repaired with the introduction of synchronization barriers, but the placement of the barriers must be done with care or they may not fix the data race or do so with a notable performance penalty.

We show that, in many GPU programs, synchronization barriers can be algorithmicly and optimally placed programs using constraint solvers and automatic amoritized resource analysis.

GPU PROGRAMMING

- Highly parallel
- Used in compute intensive tasks like neural networks, cryptography, signal processing, and scientific computing
- Error Prone due to many easy to construct problems

DATA RACES

- multiple threads access the same place with at least one of them being a write
- introduces undefined behavior

A Data Race

```
__global__ void race ( int* A ) {
    int temp = A[ threadId.x ];
    A[ threadId.x + 1] = temp + 1;
}
```

A data race exists:

Thread#1 reads from index 2

Thread#2 writes to index 2.

Synchronization barriers fix these races.

No longer a race

```
__global__ void race ( int* A ) {
    int n = A[ threadId.x ];
    __syncthreads();
    A[ threadId.x + 1] = n + 1;
}
```

All threads must pause at `__syncthreads()` until all threads have reached the same point. We can then consistently reason on the reads and writes.

REPAIRING DATA RACES

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    for (int i = 0; i < t; t++) {
        A[ threadIdx.x ] = temp;
    }
}
```

When considering how to fix a data race, we want to ensure that a synchronization barrier is place between access operations to settle threads into consistency.

TODO reword this

```
__global__ void race ( int * A, int t ) {
  __syncthreads();
  int temp = A [ threadIdx.x + 1 ];
  __syncthreads();
  for (int i = 0; i < t; t++) {
    __syncthreads();
    A[ threadIdx.x ] = temp;
    __syncthreads();
  }
  __syncthreads();
}
```

The Shotgun Approach We can place a barrier at any of these places.

```
__global__ void race ( int * A, int t ) {
    __syncthreads();
    int temp = A [ threadIdx.x + 1 ];
    __syncthreads();
    for (int i = 0; i < t; t++) {
        __syncthreads();
        A[ threadIdx.x ] = temp;
        __syncthreads();
    }
    __syncthreads();
}
```

Several are completely redundent.

How can we selectively and algorithmicly find placements?

Existing tools, like GPURepair, use software synthesis as an approach.

Each possible placement is associated with a boolean variable which is used as part of a MaxSAT search.

Other tools like AuCS use a graph reduction strategy which is similar.???

An advantage of incorporating a constraint solver to handle MaxSAT is that other classes of errors such as barrier divergence are not accidentally added to the solution.

Example of a barrier divergence

```
__global__ void divergence ( int * A ) {
    if( threadId.x == 0 ) {
        __syncthreads();
    } else {
        __syncthreads();
    }
}
```

These are not the same barrier and so threads will have multiple points for all point to arrive at, resulting in a deadlock.

This approach of guess-and-check with an oracle We have noticed that the MaxSAT is overkill.

Use an oracle such as GPUVerify to prove that a program is data race free or contains data races.

If it contains a data race, introduce a barrier. Each position is represented in a boolean formula (as well as the rest of the program). To track and compute viable locations, several tools use SMT solvers to avoid introducing bugs like barrier divergence.

TODO create a diagram for this process

We don't need all of these barriers. Some of them are redundent. Some are conceptually more expensive to use

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    for (int i = 0; i < t; t++) {
        __syncthreads();
        A[ threadIdx.x ] = temp;
    }
}
```

Conceptually we would expect this particular placement is quite expensive because of the loop.

Each tool comes up with its own cost function for determinining placement costs

Eg GPURepair uses

$$(gw * gb) + lw^{ld}$$

- gw is the penalty for a grid-level barrier
- gb is 0 for block-level barriers `__syncthreads()` and 1 for grid-level barriers `g.sync()`
- lw is the penalty for a barrier that is inside a loop
- ld is the loop-nesting depth of the barrier

TODO DIAGRAM

```
__global__ void race ( int * A, int t ) {
    __syncthreads();                  // 2
    int temp = A [ threadIdx.x + 1 ];
    __syncthreads();                  // 2
    for (int i = 0; i < t; t++) {
        __syncthreads();              // 4
        A[ threadIdx.x ] = temp;
        __syncthreads();              // 4
    }
    __syncthreads();                  // 2
}
```

$$\sum(gw * gb) + lw^{ld} = 14$$

As noted before, most barriers are redundent.

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    __syncthreads();                  // 2
    for (int i = 0; i < t; t++) {
        __syncthreads();              // 4
        A[ threadIdx.x ] = temp;
    }
}
```

$$\sum(gw * gb) + lw^{ld} = 6$$

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    for (int i = 0; i < t; t++) {
        __syncthreads();                    // 4
        A[ threadIdx.x ] = temp;
    }
}
```

$$\sum (gw * gb) + lw^{ld} = 4$$

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    __syncthreads();                            // 2
    for (int i = 0; i < t; t++) {
        A[ threadIdx.x ] = temp;
    }
}
```

$$\sum(gw * gb) + lw^{ld} = 2$$

Our contributions are to take this and improve upon the soundness of the cost model by application of automatic amoritized resource analysis which provides a formal calculus to assigning cost to placements.

our model can analyze the for-loops and determine that a higher bound has a higher cost. and can resolve for (1 .. 10) vs for(1..20) for which would be a superior placement. in addition this gives us formal grounds to prove that certain constructions are universally superior

conjecture placement between these for loops will always be best.

```
__global__ void race ( int * A ) {
    for (int i = 0; i < 10; t++) {
        int temp = A [ threadIdx.x + 1 ];
        __syncthreads();                  // 4
    }
    for (int i = 0; i < 100; t++) {
        __syncthreads();                  // 4
        A[ threadIdx.x ] = temp;
    }
}
```

Existing cost model

$$\sum(gw * gb) + lw^{ld} = 8$$

```
__global__ void race ( int * A ) {
    for (int i = 0; i < 10; t++) {
        int temp = A [ threadIdx.x + 1 ];
        __syncthreads();                            // 1
    }
    for (int i = 0; i < 100; t++) {
        __syncthreads();                            // 1
        A[ threadIdx.x ] = temp;
    }
}
```

With automatic amoritized esource analysis

$$\sum_{i=0}^{9} 1 + \sum_{i=0}^{99} 1 = 108$$

```
__global__ void race ( int * A ) {
    for (int i = 0; i < 10; t++) {
        int temp = A [ threadIdx.x + 1 ];
        __syncthreads();                          // 1
    }
    for (int i = 0; i < 100; t++) {
        __syncthreads();                          // 1
        A[ threadIdx.x ] = temp;
    }
}
```

As a MaxSAT Problem we want a placement that minimizes

$$\min\{\sum_{i=0}^{9} 1, \sum_{i=0}^{99} 1\}$$

Compared to the old cost model this allows us to pick between the loops which barrier is optimal.

$$\min\{10, 100\}$$

vs

$$\min\{4, 4\}$$

```
__global__ void race ( int * A ) {
    for (int i = 0; i < 10; t++) {
        int temp = A [ threadIdx.x + 1 ];
    }
    __syncthreads();
    for (int i = 0; i < 100; t++) {
        A[ threadIdx.x ] = temp;
    }
}
```

This is also a valid placement for an even lower cost (Ie 1). The formalism of resource analysis allows us prove optimality.

Resource Analysis

Resource analysis is similar to runtime algorithm analysis. Fixed costs and variable costs are expressed in formulas and then solved.

In this example,

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    __syncthreads();
    for (int i = 0; i < t; t++) {
        A[ threadIdx.x ] = temp;
    }
}
```

There is a cost associated with this placement, the cost of a barrier.

$$cost_{barrier}$$

In this example,

```
__global__ void race ( int * A, int t ) {
    int temp = A [ threadIdx.x + 1 ];
    for (int i = 0; i < t; t++) {
        __syncthreads();
        A[ threadIdx.x ] = temp;
    }
}
```

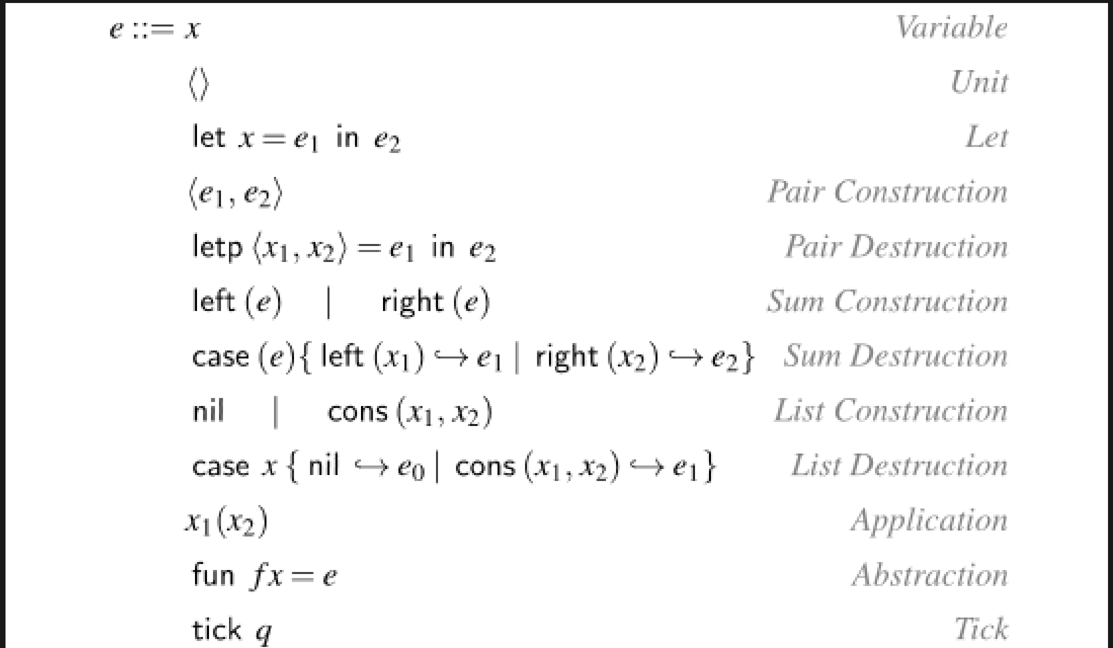
The number of times that `__syncthreads()` is dependent on the value of `t`. So we can write this as summation expression.

$$\sum_{i=0}^{t} cost_{barrier}$$

We would like to continue this analysis to give us a tigher bound on the cost of placing synchronization barriers. We can then prove the optimality of placement.

Automatic Amoritized Resource Analysis (Hoffmann and Jost) gives us a calculus for composing these analysis.

Hoffmann and Jost created a small programming language to reason about types annoations for arbitrary resources in programs. And the semantics to



$e ::= x$	*Variable*
$\langle\rangle$	*Unit*
$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	*Let*
$\langle e_1, e_2\rangle$	*Pair Construction*
$\mathsf{letp}\ \langle x_1, x_2\rangle = e_1\ \mathsf{in}\ e_2$	*Pair Destruction*
$\mathsf{left}\,(e) \quad \vert \quad \mathsf{right}\,(e)$	*Sum Construction*
$\mathsf{case}\,(e)\{\ \mathsf{left}\,(x_1) \hookrightarrow e_1 \mid \mathsf{right}\,(x_2) \hookrightarrow e_2\}$	*Sum Destruction*
$\mathsf{nil} \quad \vert \quad \mathsf{cons}\,(x_1, x_2)$	*List Construction*
$\mathsf{case}\ x\,\{\ \mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\,(x_1, x_2) \hookrightarrow e_1\}$	*List Destruction*
$x_1(x_2)$	*Application*
$\mathsf{fun}\ f x = e$	*Abstraction*
$\mathsf{tick}\ q$	*Tick*

The type system allows us to compose elemental costs to complex costs. This is the automatic in automatic amoritized analysis basic resource is a tick Show how embed our cost model into Hoffmann's calculus our tick is the `__syncthreads()`

We still need to prove that these costs are sound for this model. Show an unsound example of the other authors' cost model compares to this one.

Assuming we have the AARA. We can use a computer algebra system (maxima) to provide inequalities to relate cheaper or more expensive placements. We may even have constant values.

Using the boolean variables, like with GPURepair, we can use MaxSAT to find a placement that satisfies the oracle as well as uses the least cost according to our precise placement cost model. We use an SMT solver with the features (Z3).

Further work would include investigaing other modes of repair such as rearranging statements to repair multiple data races simultaniously.

Loop interaction and bounding resource usage

```
__global__ void race ( int * A, int t ) {
    for (int i = 0; i < t; t++) {
        int temp = A [ threadIdx.x + 1 ];
        __syncthreads();
        A[ threadIdx.x ] = temp;
    }
}
```

Still has data race for

$$t > 1$$

!

Conclusion

- Static analysis allows us to form solutions to repairing data races in GPU programs.
- Solutions are not necessarily unique.
- We have adopted Automatic Amoritized Resource Analysis's calculus to GPU Synchronization
- Using the resource analysis we are able to reason about that allows us to pick an optimal placement of a barrier.