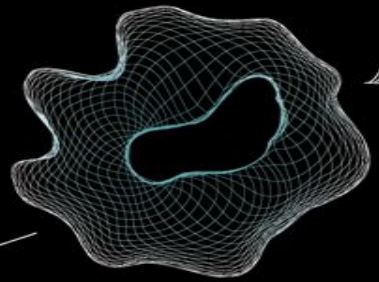
UNIVERSITY OF TWENTE.

codeplay

VerCors + SYCL

Ömer Şakar, PhD Student

UNIVERSITY OF TWENTE, NETHERLANDS



CHEOPS

Overview

- My background
- VerCors
  - Separation Logic
  - PVL (Prototypal Verification Language)
  - SYCL

Formal Methods & Tools

EEMCS Faculty at the University of Twente

- https://www.utwente.nl/en/eemcs/fmt/

UNIVERSITY OF TWENTE.

- Deductive verifier of concurrent and parallel software
- Static analysis of source code
- Supports multiple input languages
- Verification is automated, given user-provided annotations
- Annotations are written as pre/post-condition style contracts
  - Logic based on permission-based separation logic
- There are two tool papers on VerCors:
  - The VerCors tool set: verification of parallel and concurrent software. Blom, S., et al. (IFM 2017)
  - The VerCors verifier: a progress report. Armborst, L., et al. (CAV 2024)

Java/C/C++
OpenCL/CUDA
PVL
SYCL

Viper

Z3

- The explanations use PVL
  - Easiest to explain concepts
  - Language that is best supported
  - Java-like syntax

Functional Correctness

```
int meanRGB(int r, int g, int b){
    return (g + b + r) / 3;
}
```

Functional Correctness

```
requires 0 <= r && r < 256;
```

```
int meanRGB(int r, int g, int b){
   return (g + b + r) / 3;
}
```

Functional Correctness

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;


int meanRGB(int r, int g, int b){
   return (g + b + r) / 3;
}
```

Functional Correctness

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == (r + g + b) / 3;

int meanRGB(int r, int g, int b){
   return (g + b + r) / 3;
}
```

Functional Correctness

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == (r + g + b) / 3;
ensures 0 <= \result && \result < 256;
int meanRGB(int r, int g, int b){
   return (g + b + r) / 3;
}
```

Functional Correctness

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == (r + g + b) / 3;
ensures 0 <= \result && \result < 256;
int meanRGB(int r, int g, int b){
   return (g + b + r) / 3;
}
```

✅

VerCors

Functional Correctness

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == (r + g + b) / 3;
ensures 0 <= \result && \result < 128;
int meanRGB(int r, int g, int b){
   return (g + b + r) / 3;
}
```

Modularity

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == add(g,b,r) / 3;
ensures 0 <= \result && \result < 256;
int meanRGB(int r, int g, int b){
   return add(r,b,g) / 3;
}

requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == r+g+b;
int add(int r, int g, int b) {
   return (r+g+b);
}
```

Modularity

`add` and `meanRGB` are verified separately.

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == add(g,b,r) / 3;
ensures 0 <= \result && \result < 256;
int meanRGB(int r, int g, int b){
   return add(r,b,g) / 3;
}

requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == r+g+b;
int add(int r, int g, int b) {
   return (r+g+b);
}
```

Modularity

`add` and `meanRGB` are verified separately.

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == add(g,b,r) / 3;
ensures 0 <= \result && \result < 256;
int meanRGB(int r, int g, int b){
   return add(r,b,g) / 3;
}
```

Assume the preconditions
Prove the postconditions

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == r+g+b;
int add(int r, int g, int b) {
   return (r+g+b);
}
```

`add` and `meanRGB` are verified separately.

Prove the preconditions
Get the postconditions

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == add(g,b,r) / 3;
ensures 0 <= \result && \result < 256;
int meanRGB(int r, int g, int b){
    return add(r,b,g) / 3;
}
```

Assume the preconditions
Prove the postconditions

```
requires 0 <= r && r < 256;
requires 0 <= g && g < 256;
requires 0 <= b && b < 256;
ensures \result == r+g+b;
int add(int r, int g, int b) {
    return (r+g+b);
}
```

- What do we prove?
  - Functional-correctness properties
  - Data-race freedom and memory safety (by default)
  - Partial correctness (no proof for termination)

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Permission-based Separation Logic

- Permissions to specify ownership
  - Tokens to specify access mode (read/write)
- Specify permissions for every shared location
- Permisisons are fractions between 0 and 1
- $0 < p < 1$ is a read permission
- 1 is a write permission
- For any shared location, at most 1 permission in the whole system

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r, write) ** Perm(g, write) ** Perm(b, write);
    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context
    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r,      )
    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r,   1\1)
    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r, write)
    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r, write) ** Perm(g, write) ** Perm(b, write);
    ensures r == 0 && g == 0 && b == 0;
    void clear() {
        r = 0;
        g = 0;
        b = 0;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r,1\2);
    ensures \result == r;
    int getR() {
        return r;
    }
}
```

Data-race freedom

```
class RGB {
    int r;
    int g;
    int b;

    context Perm(r,read);
    ensures \result == r;
    int getR() {
        return r;
    }
}
```

Data-race freedom

```
class RGB {
   int r;
   int g;
   int b;

  requires Perm(r,3\4);
  ensures  Perm(r,1\4);
  int getR() {
      return r;
  }
}

void main() {
   RGB rgb = new RGB();

   int r = rgb.getR();
   r = rgb.getR();
   assert r == 0;
}
```

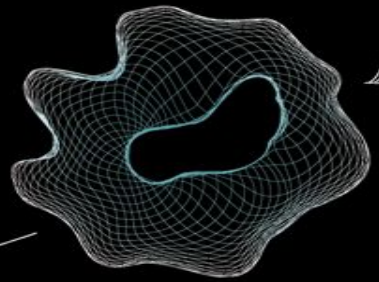
UNIVERSITY OF TWENTE.

codeplay

Verification of Matrix Multiplication in SYCL

Ömer Şakar, PhD Student

UNIVERSITY OF TWENTE, NETHERLANDS



CHEOPS

Overview

- Matrix Multiplication
- Verification Concepts/Techniques
- Matrix Multiplication in SYCL

Matrix Multiplication

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{A} \times \mathbf{B} = \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Matrix Multiplication

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{A} \times \mathbf{B} = \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Matrix Multiplication

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{A} \times \mathbf{B} = \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Matrix Multiplication

- Kernel 1: Initialize matrix a
- Kernel 2: Initialize matrix b
- Kernel 3: Calculate c=axb

Matrix Multiplication

Ghost state

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication

Ghost state

Program state

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Ghost state

$\mathbf{A} =$ $\mathbf{B} =$

Matrix Multiplication

Ghost state

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

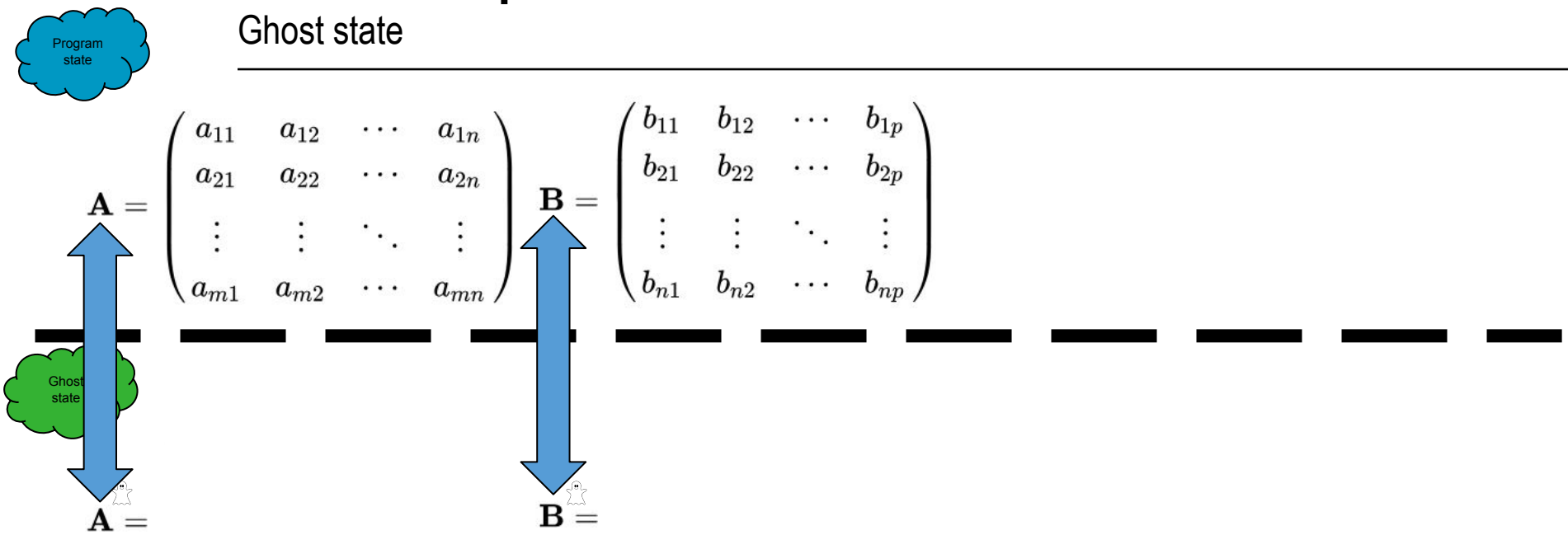
Ghost state

seq<int>

$\mathbf{A} =$ $\mathbf{B} =$

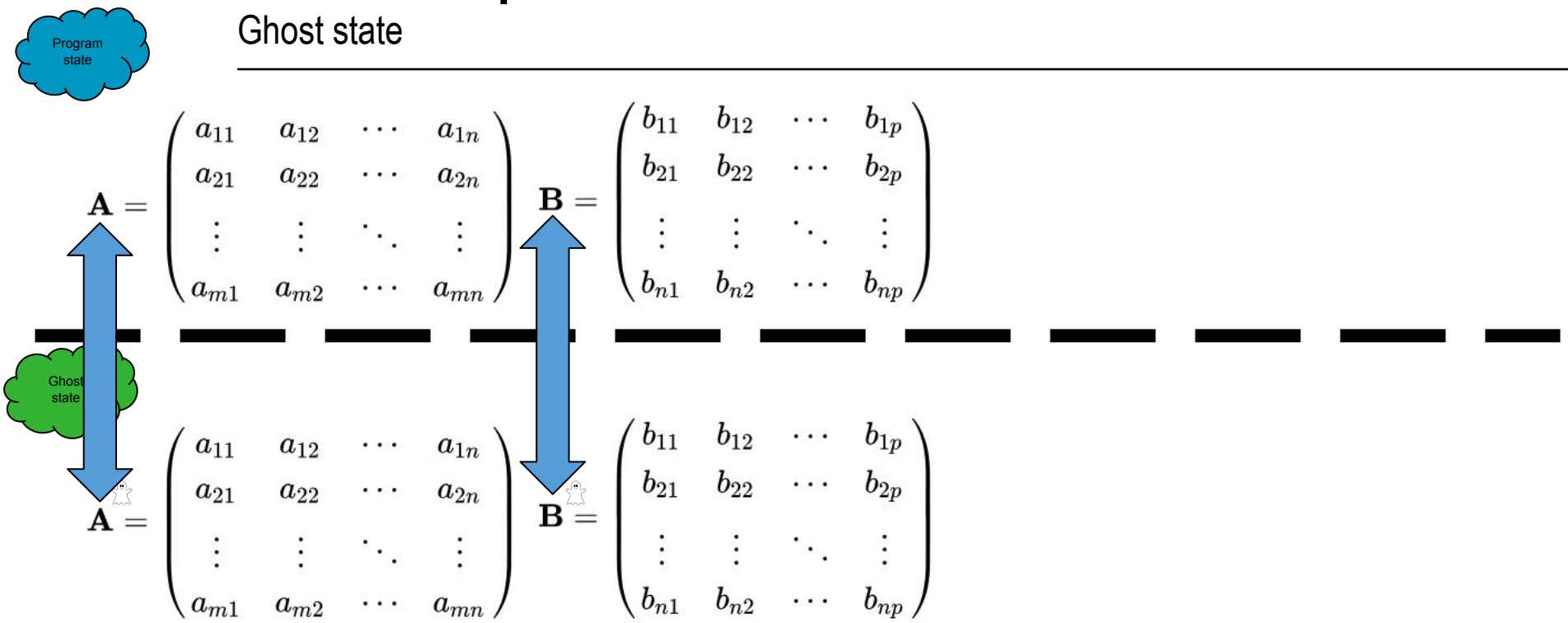
Matrix Multiplication

Ghost state



Matrix Multiplication

Ghost state



Matrix Multiplication

Ghost state

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Ghost state

seq<int>

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication

Ghost state

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

Ghost state

seq<int>

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication

Loop invariant

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

```
sum = 0
```

Ghost state

seq<int>

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication

Loop invariant

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

`sum = a11*b11`

Ghost state

seq<int>

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication

Loop invariant

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

sum = $a_{11}*b_{11}+a_{12}*b_{21}$

Ghost state

seq<int>

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication

Loop invariant

Program state

int*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

sum = $a_{11}*b_{11}+a_{12}*b_{21}$ … $a_{1n}*b_{n1}$

Ghost state

seq<int>

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix Multiplication in SYCL

VSCode

Matrix Multiplication in SYCL

- Current improvement points of SYCL support
  - Currently only buffers/accessors are supported
    - USM support is a possible followup
  - The proof takes ~4 minutes
    - Also proven the concrete case
    - Parsing C++ takes significant time, ~1 minute