ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator

Bahar Asgari, Ramyad Hadidi,
Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili

Georgia Tech

comparch

CASL

SYNERGY

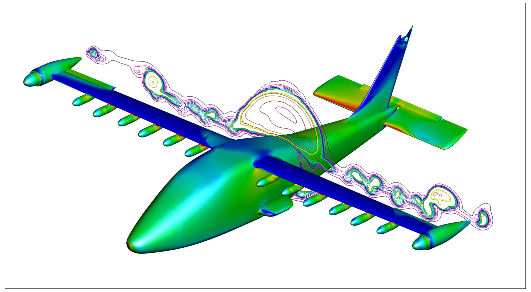

TODAY IS

Modeling impacts our lives and future!

Hurricanes

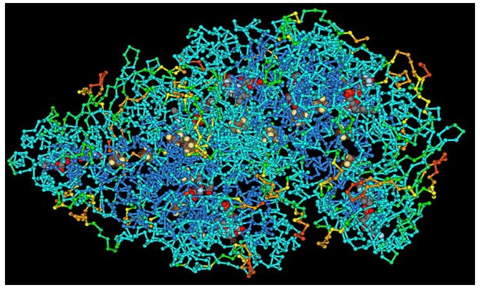
nasa.gov

Aerodynamic



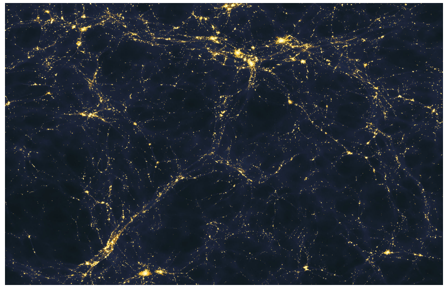
nasa.gov

Macromolecules



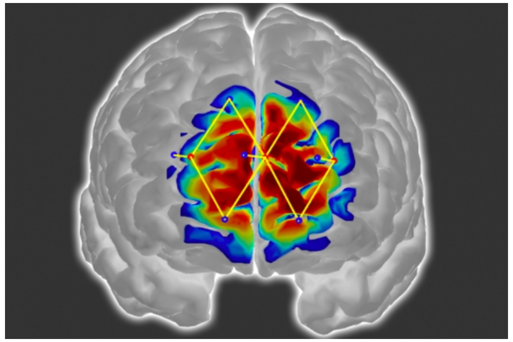
wikipedia.org

Universe



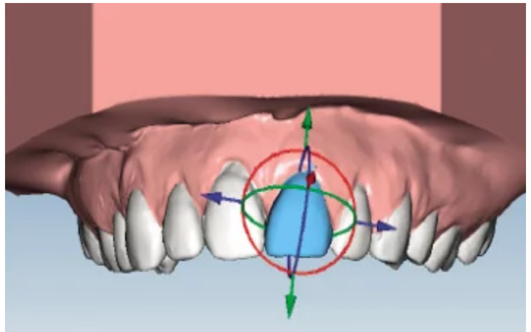
ucl.ac.uk

Pain



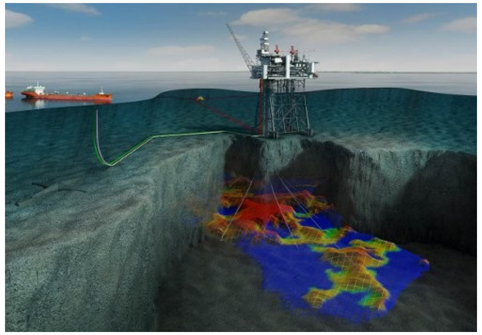
mit.edu

Orthodontics



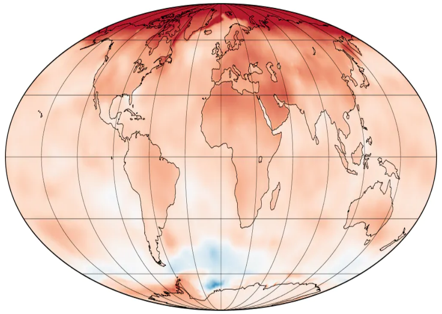
planmeca.com

Oil and Gas



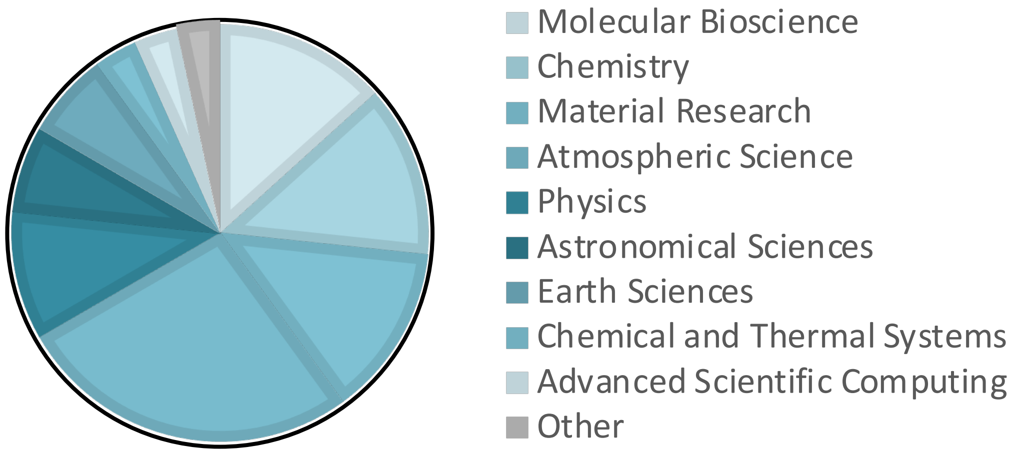
enwa.com

Global Warming



washingtonpost.com

Modeling is **costly!**

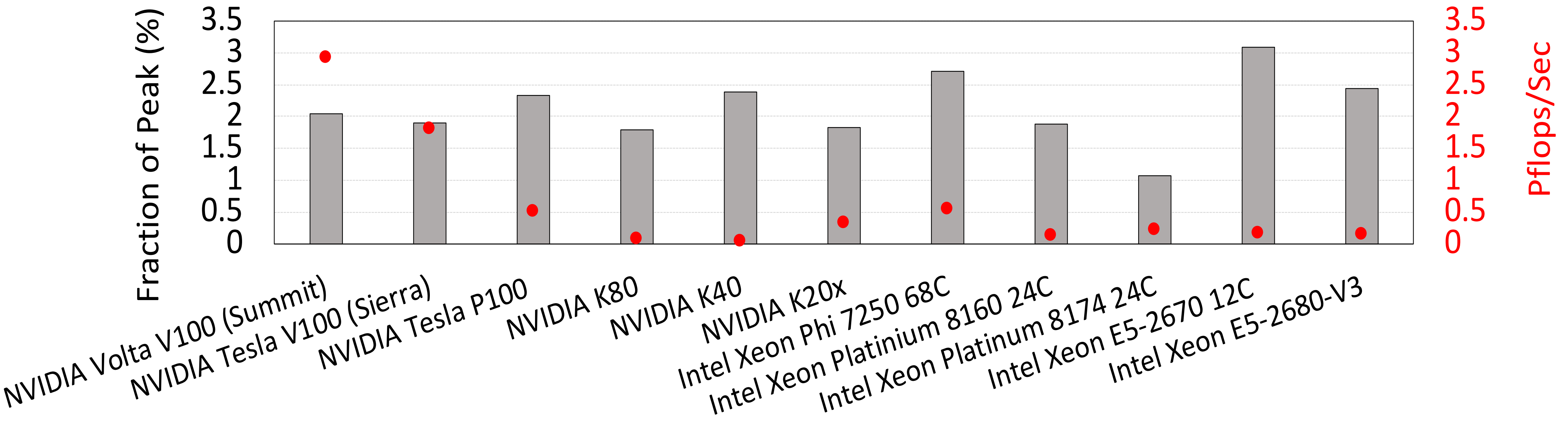


>96% of supercomputer workloads![1]

$3.5 million per year only for power and cooling one system[2]!

[1] Workloads of Kraken, housed in the Oak Ridge National Lab.
[2] Tianhe-1A

Modeling is **slow**, even with optimizations!



They utilize < **3%** of peak performance

Data obtained from HPCG ranking.

We propose **Alrescha**[1]

a **fast** and **low-cost** solution

for executing scientific problems

[1]Alrescha (/ælˈriːʃə/) is a binary star system in the equatorial constellation of Pisces

Outline

- Using PDEs for modeling and key challenges
- Alrescha
  - Main contributions
  - Storage format
  - Reconfigurable microarchitecture
  - Broad applications
- Results
- Conclusions

Outline

- **Using PDEs for modeling and key challenges**
- Alrescha
  - Main contributions
  - Storage format
  - Reconfigurable microarchitecture
  - Broad applications
- Results
- Conclusions

Partial differential equations (PDEs)

- PDEs are used for modeling.
- PDEs are transformed to $\boldsymbol{Ax} = \boldsymbol{b}$.
- Solving PDEs

Direct methods: Exact but too slow 🤔

- Cholesky method
- Are not used for large sparse problems

Focus of this paper →

Iterative methods: Fast and converges 😃

- Conjugated Gradient (CG)
- Fast execution ➔ more iterations ➔ exact results

PDE Characteristics and Challenges

- PDEs are **sparse**
- Iterative solvers include **data-dependency**
- Limited parallelism:
  - **Dependencies** limit using high memory bandwidth
- We cannot simply add more bandwidth to gain performance

Dependencies in solving PDEs

Symmetric Gauss Seidel (SymGS) is the main kernel

Simplified mathematical expression is $$\boldsymbol{x}_i = \sum_{j=0}^{columns} \boldsymbol{A}_{ij}^T \times \boldsymbol{x}_j$$

Which includes a nested loop:

- Iterations of **outer** loop are **data-dependent**
- Iterations of **inner** loop can run in **parallel**

This creates bottleneck

```
for i = 0 to rows
  for j = 0 to columns
       sum += A[i][j] * x[j]
  x[i] = update(sum)
```

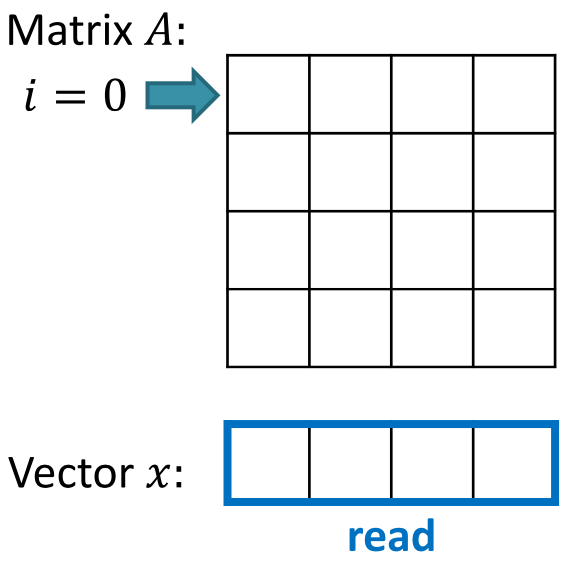
The equation and pseudo are the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
      sum += A[i][j] * x[j]
  x[i] = update(sum)
```



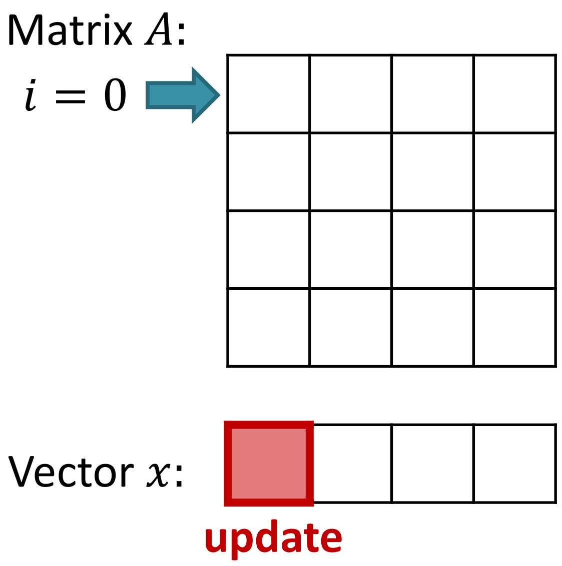
The pseudo is the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
      sum += A[i][j] * x[j]
  x[i] = update(sum)
```

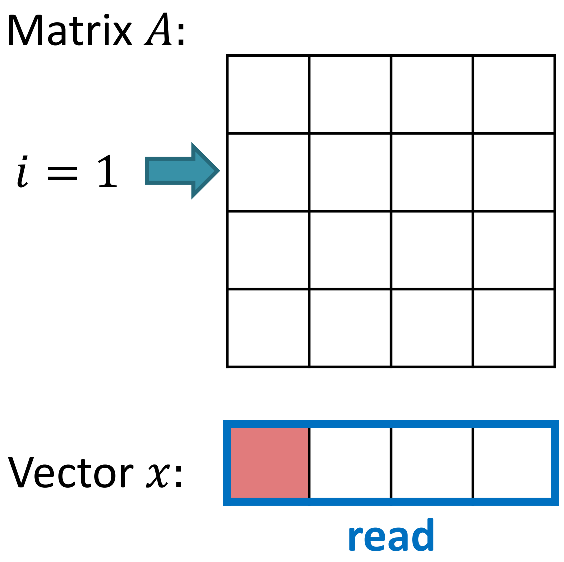


Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
      sum += A[i][j] * x[j]
  x[i] = update(sum)
```

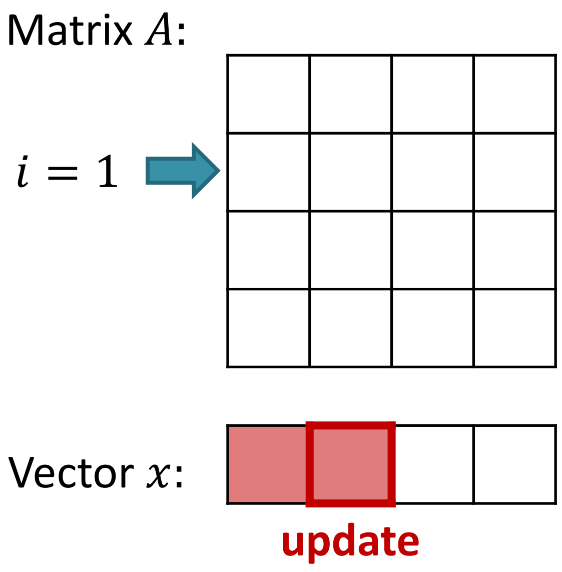


The pseudo is the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x




```
for i = 0 to rows
  for j = 0 to columns
       sum += A[i][j] * x[j]
  x[i] = update(sum)
```

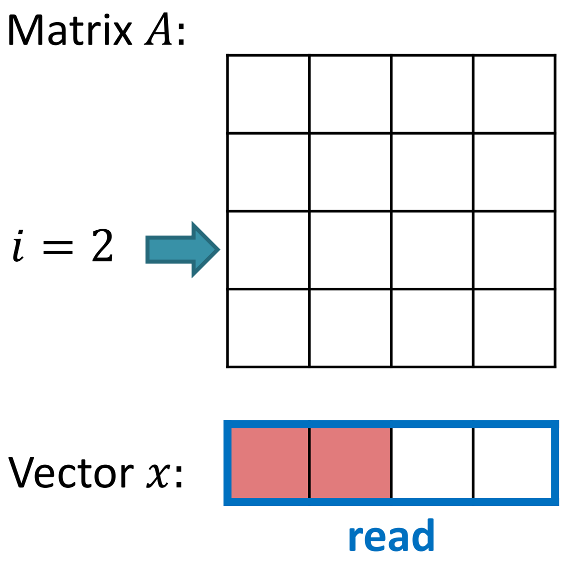
The pseudo is the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
      sum += A[i][j] * x[j]
  x[i] = update(sum)
```



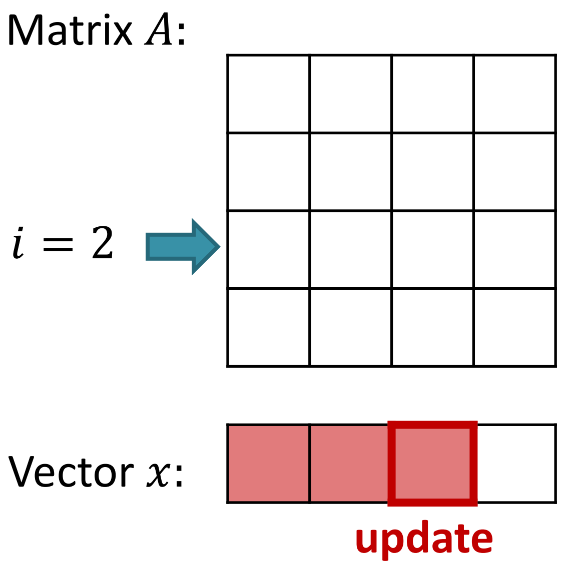
The pseudo is the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
        sum += A[i][j] * x[j]
  x[i] = update(sum)
```



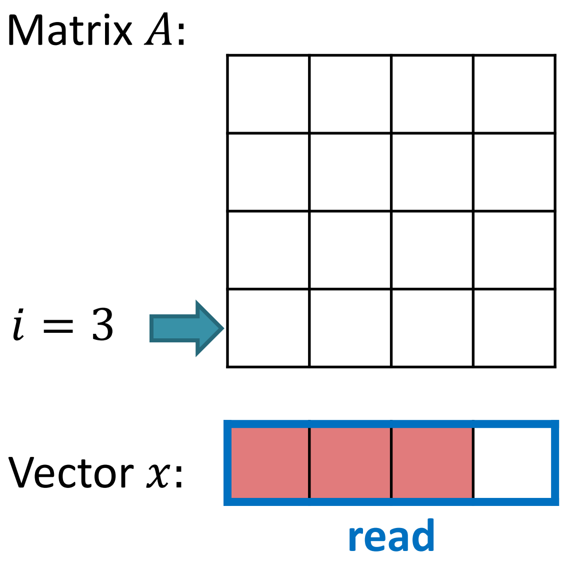
The pseudo is the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
      sum += A[i][j] * x[j]
  x[i] = update(sum)
```



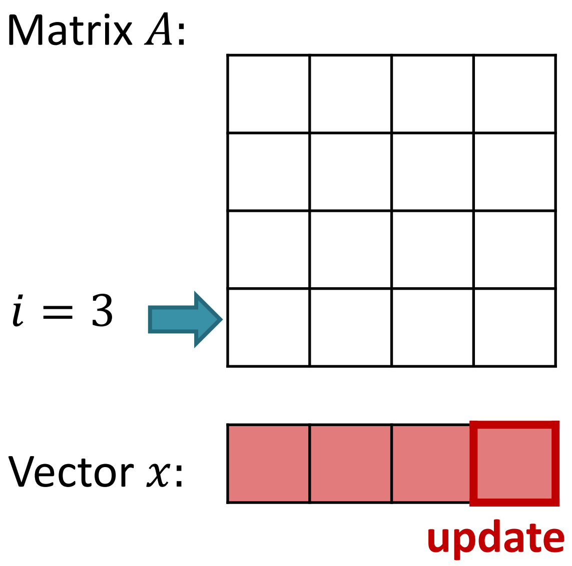
The pseudo is the extremely simplified version of SymGS

Why data-dependent?

At each iteration of the outer loop, we

- Read entire x
- Update one element of x

```
for i = 0 to rows
  for j = 0 to columns
      sum += A[i][j] * x[j]
  x[i] = update(sum)
```



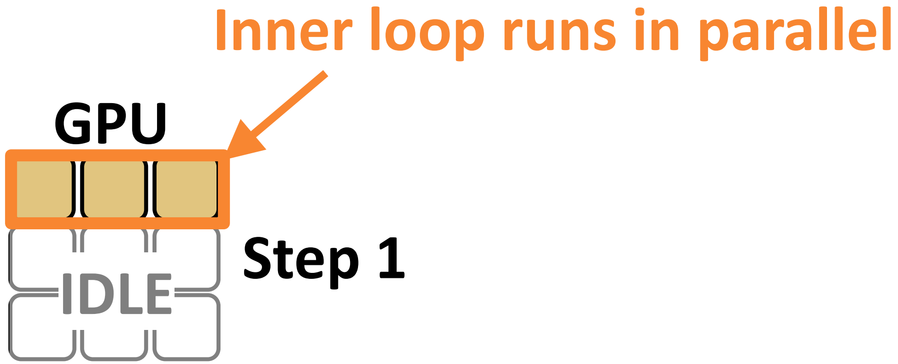
Cannot utilize parallelism of GPU

Timeline of GPU:

```
for i = 0 to rows
    for j = 0 to columns
    x[i] = ...
```

Iterations of outer loop

`i = 4` | `for j = 0 to columns` | `x[4]=...`



Cannot utilize parallelism of GPU

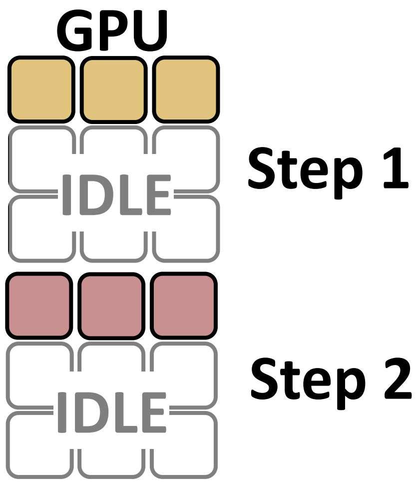
Timeline of GPU:

```
for i = 0 to rows
    for j = 0 to columns
    x[i] = ...
```

Iterations of outer loop

```
i = 4   for j = 0 to columns   x[4]=...

i = 5   for j = 0 to columns   x[5]=...
```



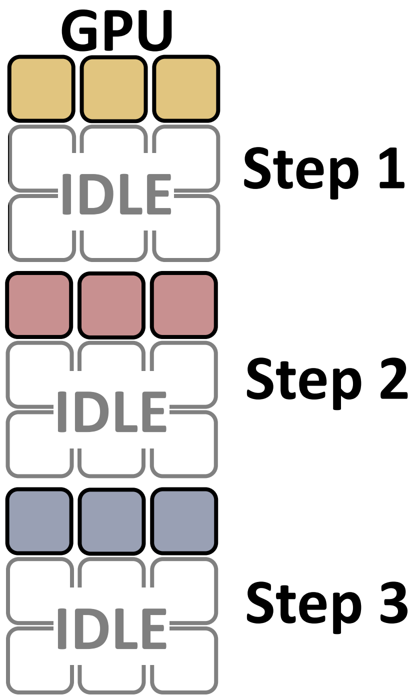
Cannot utilize parallelism of GPU

Timeline of GPU:

```
for i = 0 to rows
      for j = 0 to columns
      x[i] = ...
```

Iterations of outer loop

i = 4	for j = 0 to columns	x[4]=...
i = 5	for j = 0 to columns	x[5]=...
i = 6	for j = 0 to columns	x[6]=...

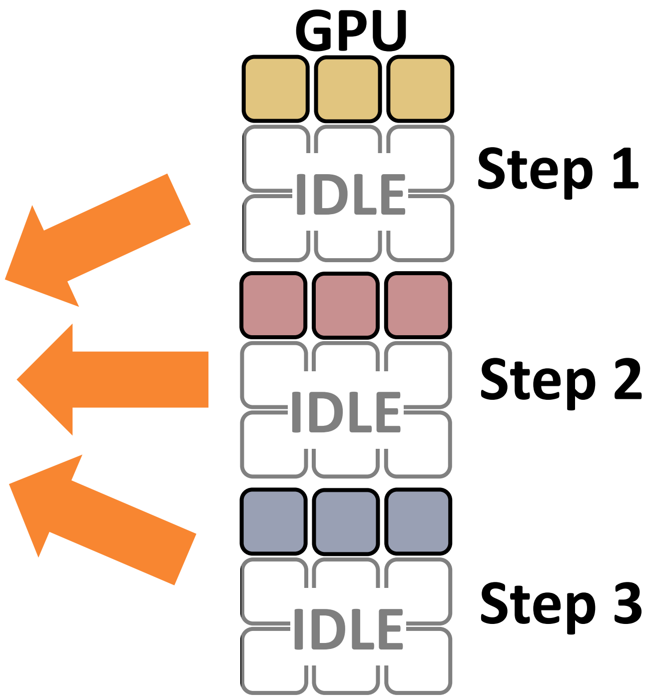


Key challenge

Timeline of GPU:

```
for i = 0 to rows
    for j = 0 to columns
    x[i] = ...
```

Iterations of the outer loop are **not parallel**



Optimization cannot help

Timeline of GPU with unrolling and blocking:

```
for i = 0 to rows
    for j = 0 to columns
    x[i] = ...
```

Unroll the outer loop &
Break down the inner loop

i = 4	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[4]=...
i = 5	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[5]=...
i = 6	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[6]=...

Optimizations similar to graph coloring

Optimization cannot help

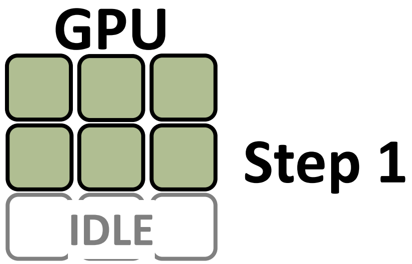
Timeline of GPU with unrolling and blocking:

```
for i = 0 to rows
      for j = 0 to columns
      x[i] = ...
```

Unroll the outer loop &
Break down the inner loop

i = 4	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[4]=...
i = 5	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[5]=...
i = 6	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[6]=...

No dependency here
They can run in parallel



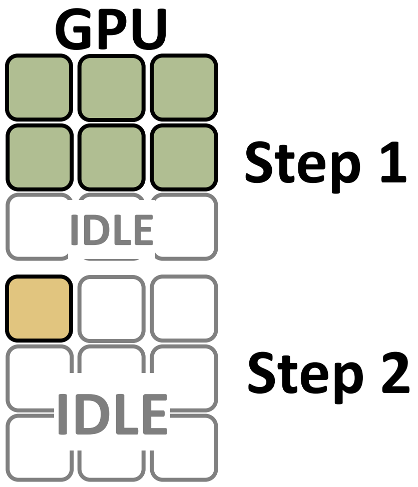
Optimizations similar to graph coloring

Optimization cannot help

Timeline of GPU with unrolling and blocking:

```
for i = 0 to rows
      for j = 0 to columns
      x[i] = ...
```

Unroll the outer loop &
Break down the inner loop



i = 4	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[4]=...
i = 5	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[5]=...
i = 6	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[6]=...

Optimizations similar to graph coloring

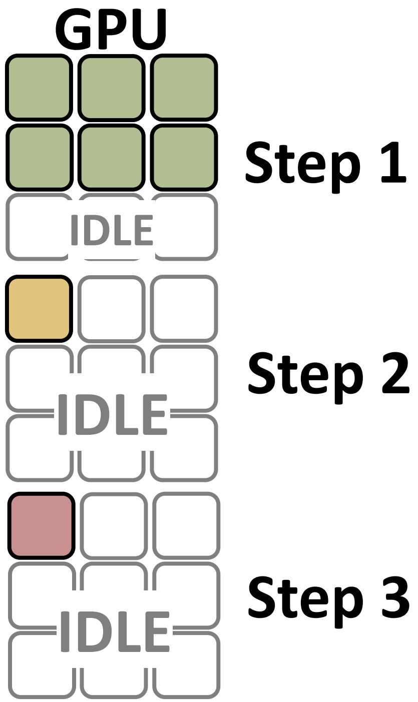
Optimization cannot help

Timeline of GPU with unrolling and blocking:

```
for i = 0 to rows
      for j = 0 to columns
      x[i] = ...
```

Unroll the outer loop &
Break down the inner loop

i = 4	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[4]=...
i = 5	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[5]=...
i = 6	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[6]=...



Optimizations similar to graph coloring

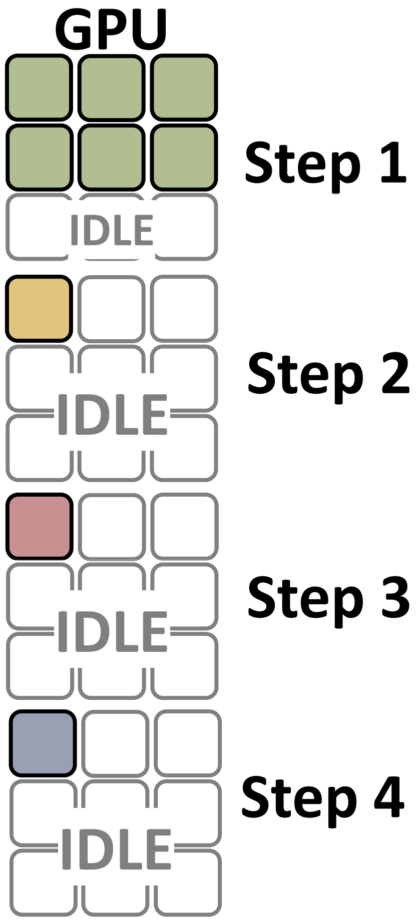
Optimization cannot help

Timeline of GPU with unrolling and blocking:

```
for i = 0 to rows
    for j = 0 to columns
    x[i] = ...
```

Unroll the outer loop &
Break down the inner loop

i = 4	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[4]=...
i = 5	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[5]=...
i = 6	for j = 0 to 3	j = 4, 5, 6	j = 7 to columns	x[6]=...

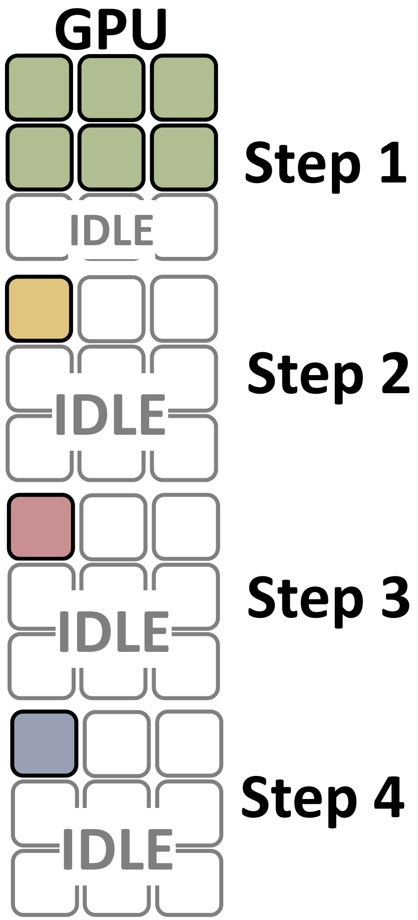


Key challenge

Timeline of GPU with unrolling and blocking:

More parallelism

But the **dependencies** still create the **bottleneck**

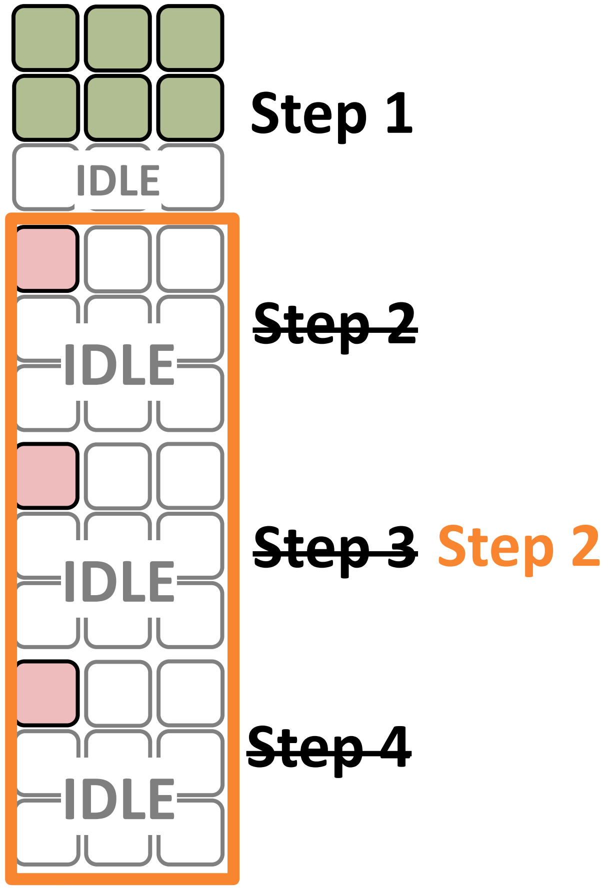


Key insight

We accelerate the dependent operations

We use a partially reconfigurable hardware

to execute both parallel and dependent part



We cannot resolve dependencies

but,

We can execute them in one step!

Alrescha

Divides a large SymGS into:

- **Parallel GEMV**[1]
- **Small data-dependent SymGS**[2]

Reorders the operations:

- First, GEMV
- Then, SymGS

GEMV

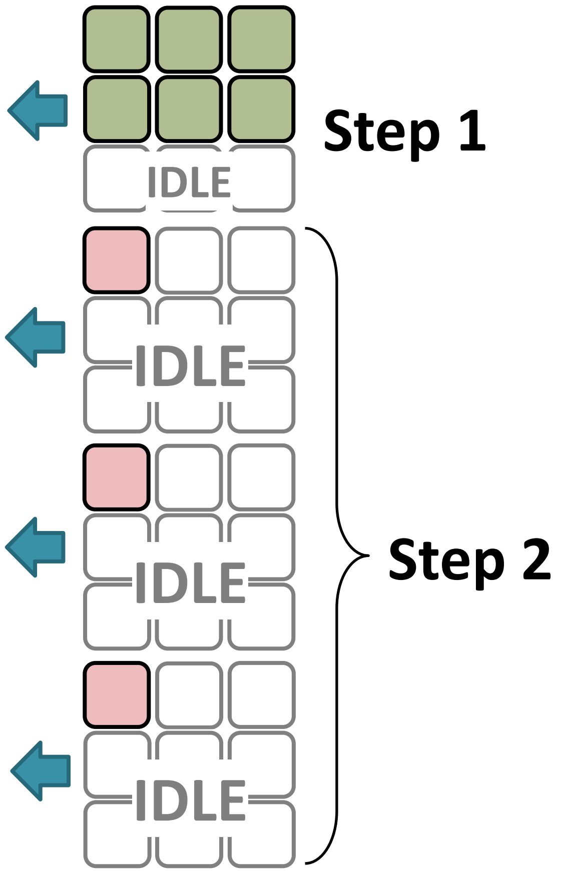
$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

SymGS

$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$



[1] GEMV: General matrix vector multiplication
[2] SymGS: Symmetric Gauss Seidel

Outline

- Using PDEs for modeling and key challenges
- **Alrescha**
  - **Main contributions**
  - Storage format
  - Reconfigurable microarchitecture
  - Broad applications
- Results
- Conclusions

Contributions of Alrescha

1. To take advantage of reordering

- Fast execution of data-dependent SymGS

$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

Dependencies still exist
Alrescha implements them fast!

Contributions of Alrescha

1. To take advantage of reordering

- Fast execution of data-dependent SymGS
- Fast switching between GEMV to SymGS

This must be fast!

Alrescha uses a LIFO[1]

$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

$$\overline{\boldsymbol{x}_4} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

$$\overline{\boldsymbol{x}_5} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x}_4} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x}_6} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x}_4} + A^T_{65} \times \overline{\boldsymbol{x}_5} + A^T_{66} \times x_6$$

[1] Last in first out (LIFO) buffer

Contributions of Alrescha

1. Reordering the operations
   - Fast execution of data-dependent SymGS
   - Fast switching between GEMV to SymGS

2. Lightweight reconfigurable architecture
   - A fixed reduction engine
   - A small reconfigurable hardware

$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

They differ slightly
Both need reduction!

$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

Contributions of Alrescha

1. Reordering the operations

- Fast execution of data-dependent SymGS
- Fast switching between GEMV to SymGS

2. Lightweight reconfigurable architecture

- A fixed reduction engine
- A small reconfigurable hardware

3. A new storage format

- To sustain the desired orders

$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

Contributions of Alrescha

1. Reordering the operations
    - Fast execution of data-dependent SymGS
    - Fast switching between GEMV to SymGS
2. Lightweight reconfigurable architecture
    - A fixed reduction engine
    - A small reconfigurable hardware
3. A new storage format
    - To sustain the desired orders
4. Broad applications
    - Because we have a reduction engine!

$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

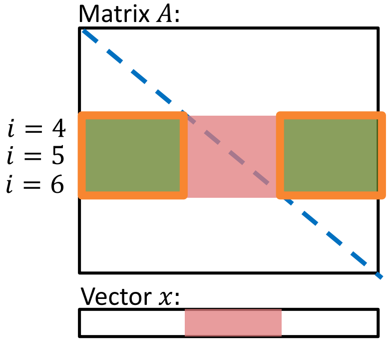
$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

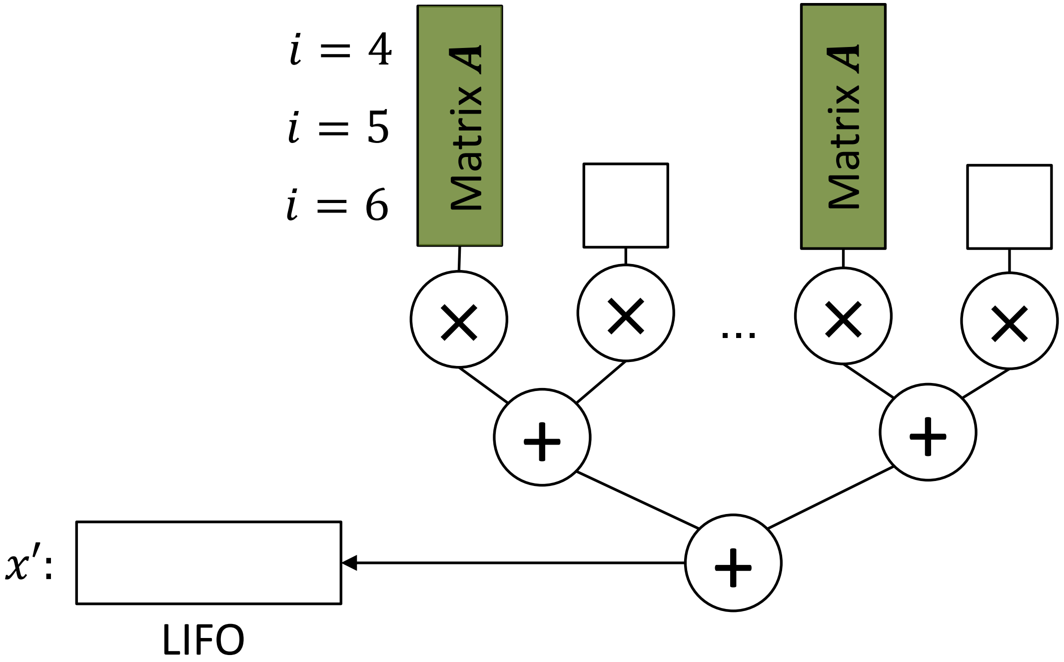
$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

Reordering the operations

- First, Alrescha
  - Executes the GEMV
  - Produces intermediate results (partial sum, $\boldsymbol{x}'$)
  - Pushes x' into a LIFO to reuse them fast, later





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

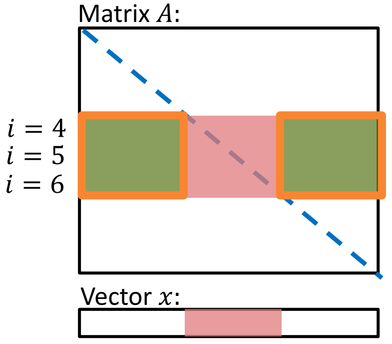
$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44}\times x_4 + A^T_{45}\times x_5 + A^T_{46}\times x_6$$

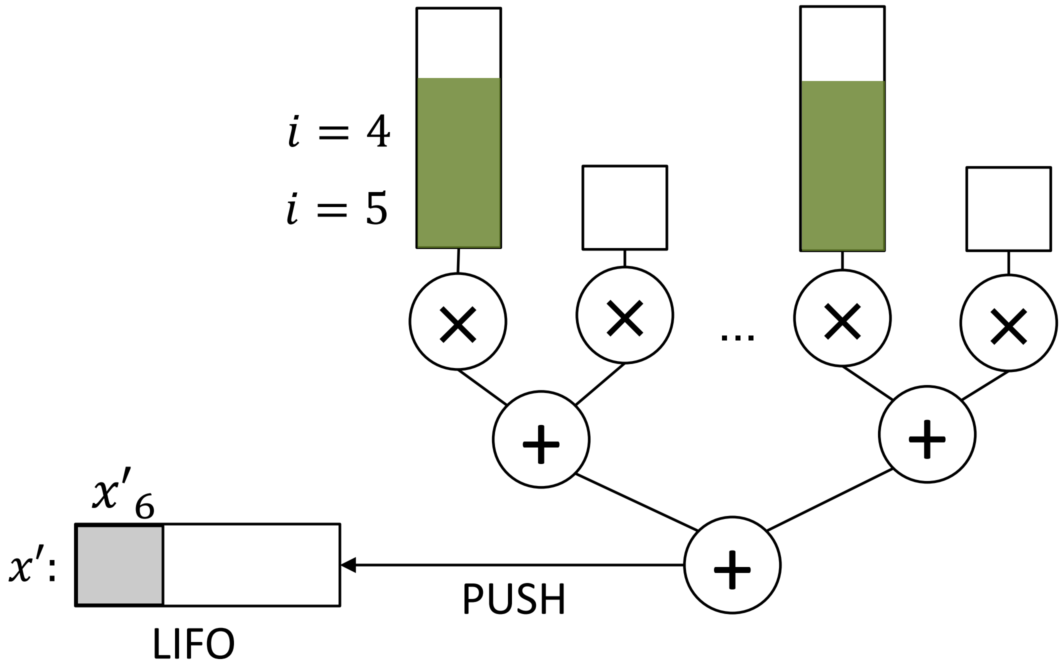
$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54}\times\overline{\boldsymbol{x_4}} + A^T_{55}\times x_5 + A^T_{56}\times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64}\times\overline{\boldsymbol{x_4}} + A^T_{65}\times\overline{\boldsymbol{x_5}} + A^T_{66}\times x_6$$

Reordering the operations

- First, Alrescha
  - Executes the GEMV
  - Produces intermediate results (partial sum, $\boldsymbol{x}'$)
  - Pushes x' into a LIFO to reuse them fast, later





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

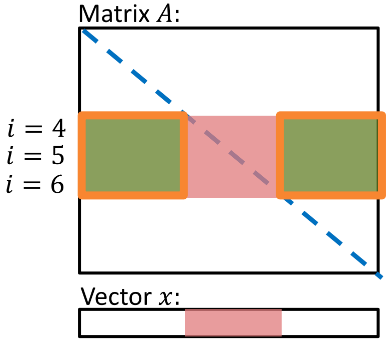
$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

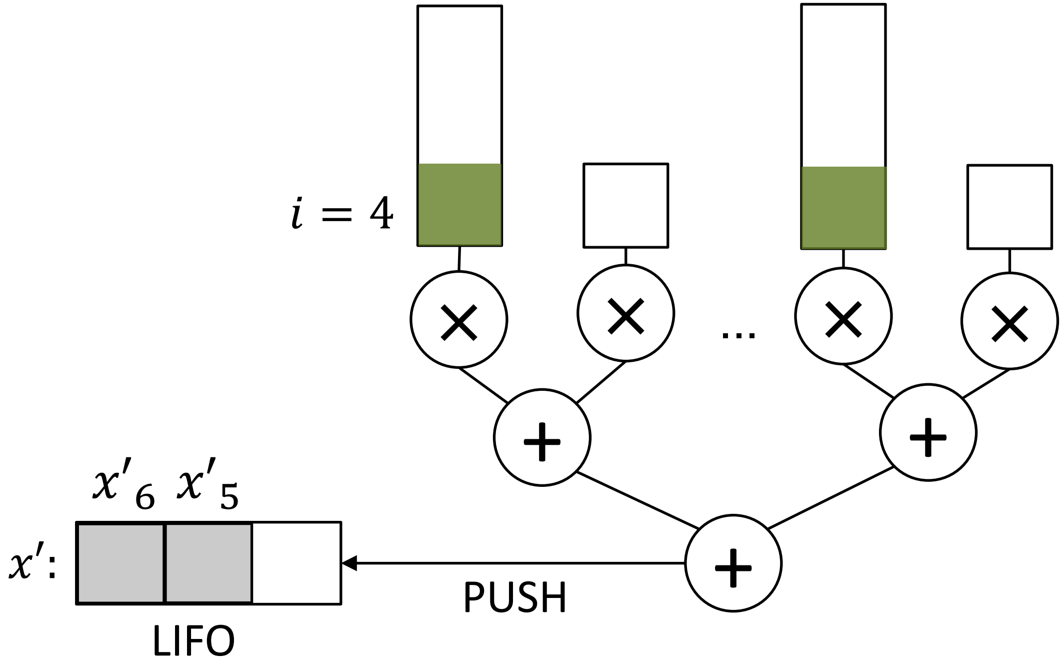
$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

Reordering the operations

- First, Alrescha
  - Executes the GEMV
  - Produces intermediate results (partial sum, $\boldsymbol{x}'$)
  - Pushes x' into a LIFO to reuse them fast, later





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

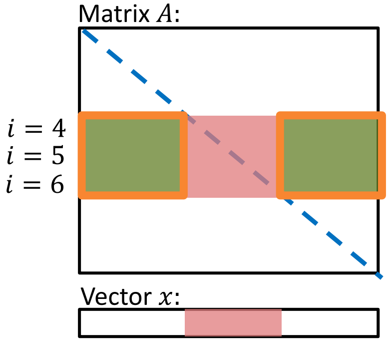
$$\overline{x_4} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

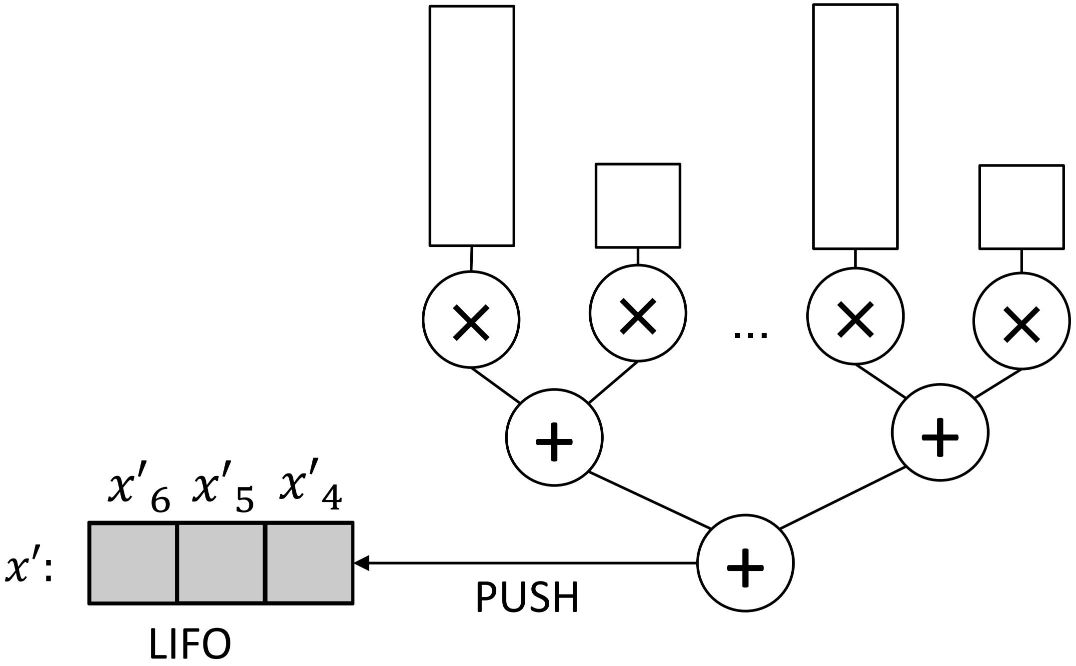
$$\overline{x_5} = x'_5 + A^T_{54} \times \overline{x_4} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{x_6} = x'_6 + A^T_{64} \times \overline{x_4} + A^T_{65} \times \overline{x_5} + A^T_{66} \times x_6$$

Reordering the operations

- First, Alrescha
  - Executes the GEMV
  - Produces intermediate results (partial sum, $\boldsymbol{x}'$)
  - Pushes x' into a LIFO to reuse them fast, later





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

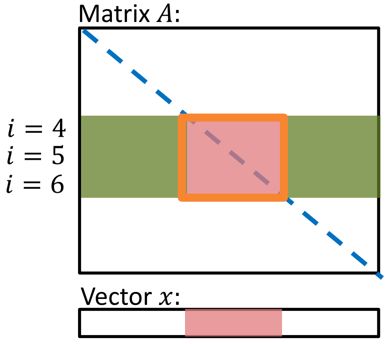
$$\overline{\boldsymbol{x}_4} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

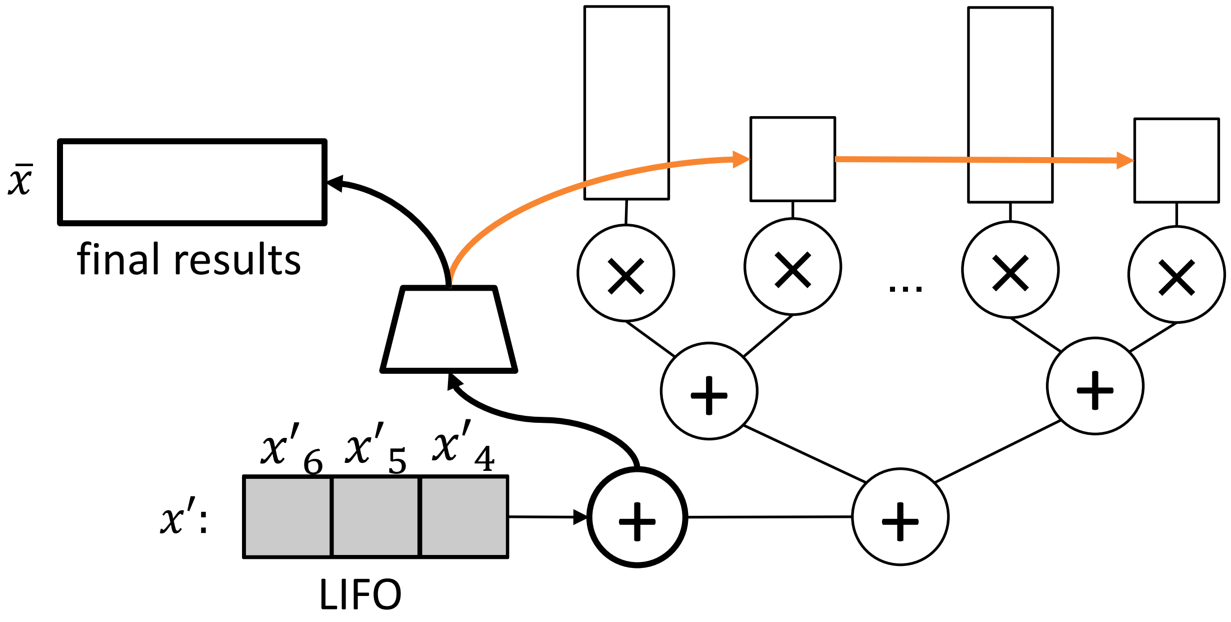
$$\overline{\boldsymbol{x}_5} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x}_4} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x}_6} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x}_4} + A^T_{65} \times \overline{\boldsymbol{x}_5} + A^T_{66} \times x_6$$

Reordering the operations

- Then, Alrescha
  - Executes SymGS using **same reduction tree**
  - Pops x' from the LIFO to reuse them fast
  - **Reuses new $\bar{x}$ immediately through shift connections**





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

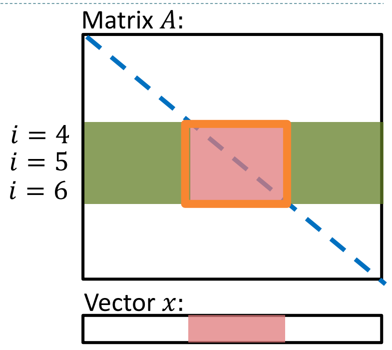
$$\bar{x}_4 = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

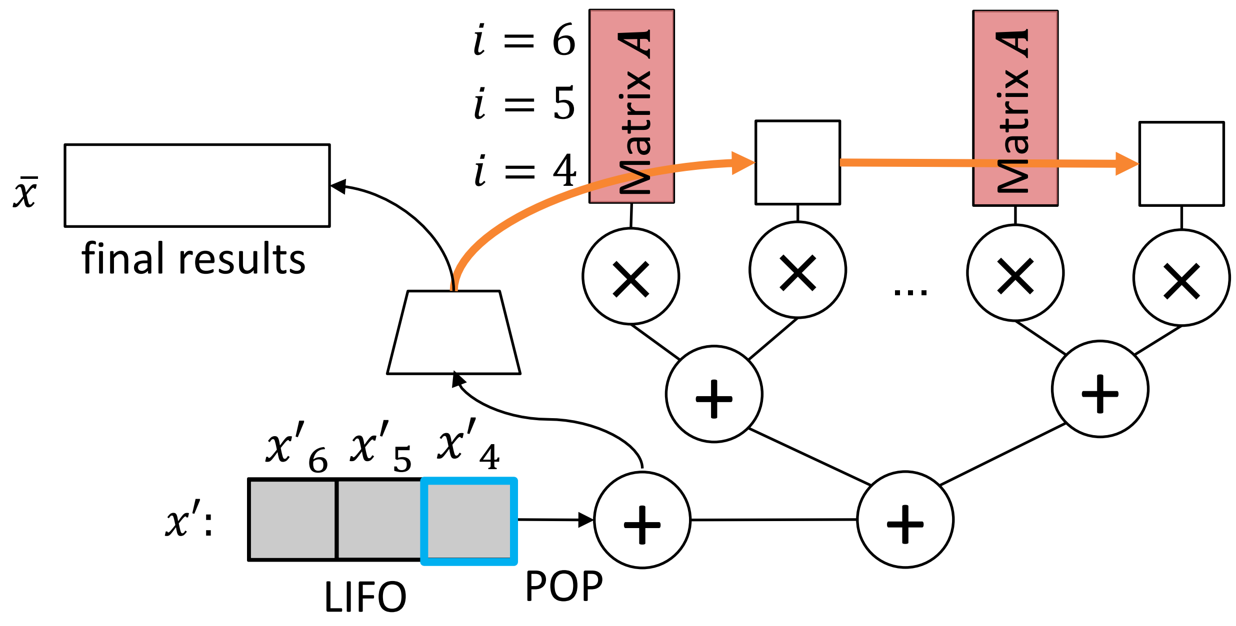
$$\bar{x}_5 = x'_5 + A^T_{54} \times \bar{x}_4 + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\bar{x}_6 = x'_6 + A^T_{64} \times \bar{x}_4 + A^T_{65} \times \bar{x}_5 + A^T_{66} \times x_6$$

Reordering the operations

- Then, Alrescha
  - Executes SymGS using **same reduction tree**
  - Pops x' from the LIFO to reuse them fast
  - **Reuses new $\overline{x}$ immediately through shift connections**





$$x'_i = \sum_{i\neq 4,5,6} A^T_{ij} \times x_j$$

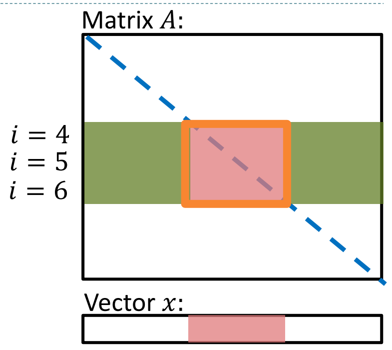
$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44}\times x_4 + A^T_{45}\times x_5 + A^T_{46}\times x_6$$

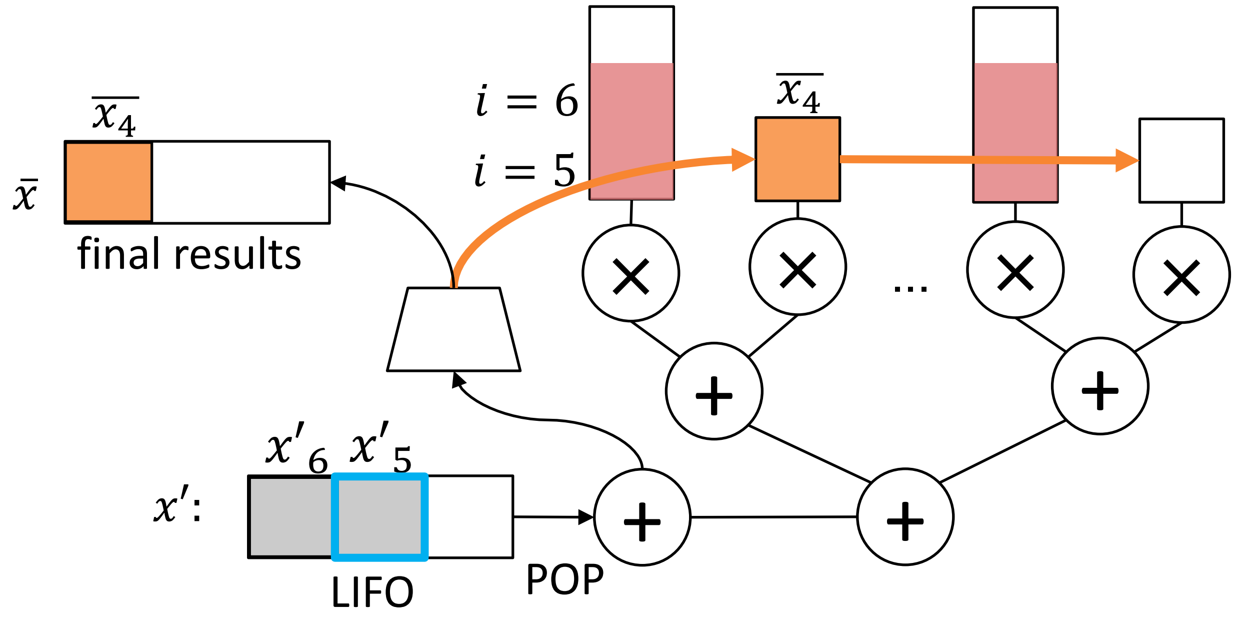
$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54}\times \overline{\boldsymbol{x_4}} + A^T_{55}\times x_5 + A^T_{56}\times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64}\times \overline{\boldsymbol{x_4}} + A^T_{65}\times \overline{\boldsymbol{x_5}} + A^T_{66}\times x_6$$

Reordering the operations

- Then, Alrescha
  - Executes SymGS using **same reduction tree**
  - Pops x' from the LIFO to reuse them fast
  - **Reuses new $\overline{x}$ immediately through shift connections**





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

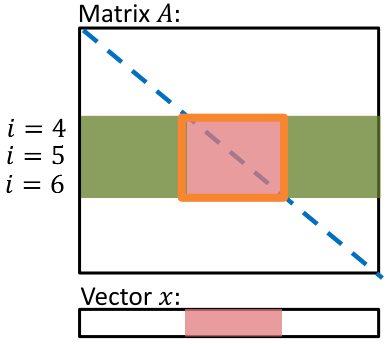
$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

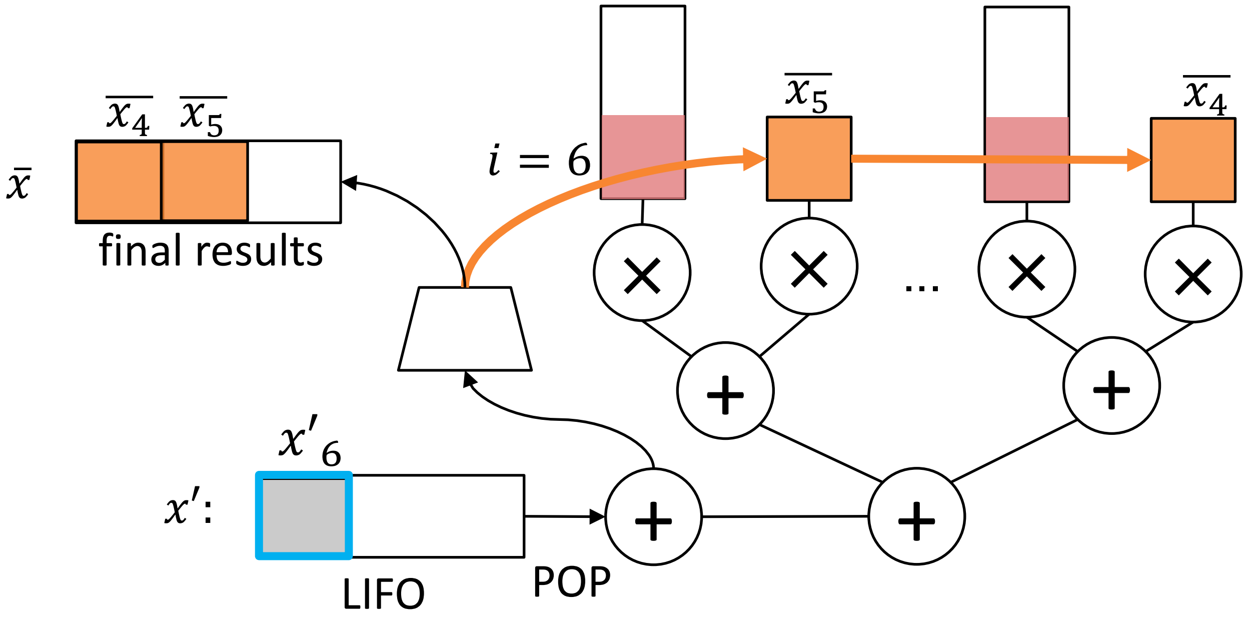
$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

Reordering the operations

- Then, Alrescha
  - Executes SymGS using **same reduction tree**
  - Pops x' from the LIFO to reuse them fast
  - **Reuses new $\bar{x}$ immediately through shift connections**





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

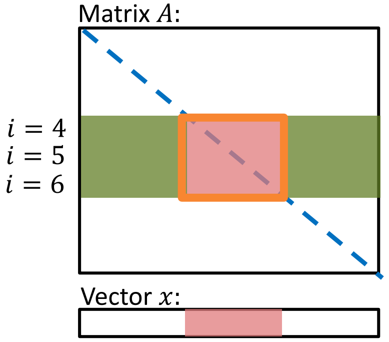
$$\overline{x_4} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

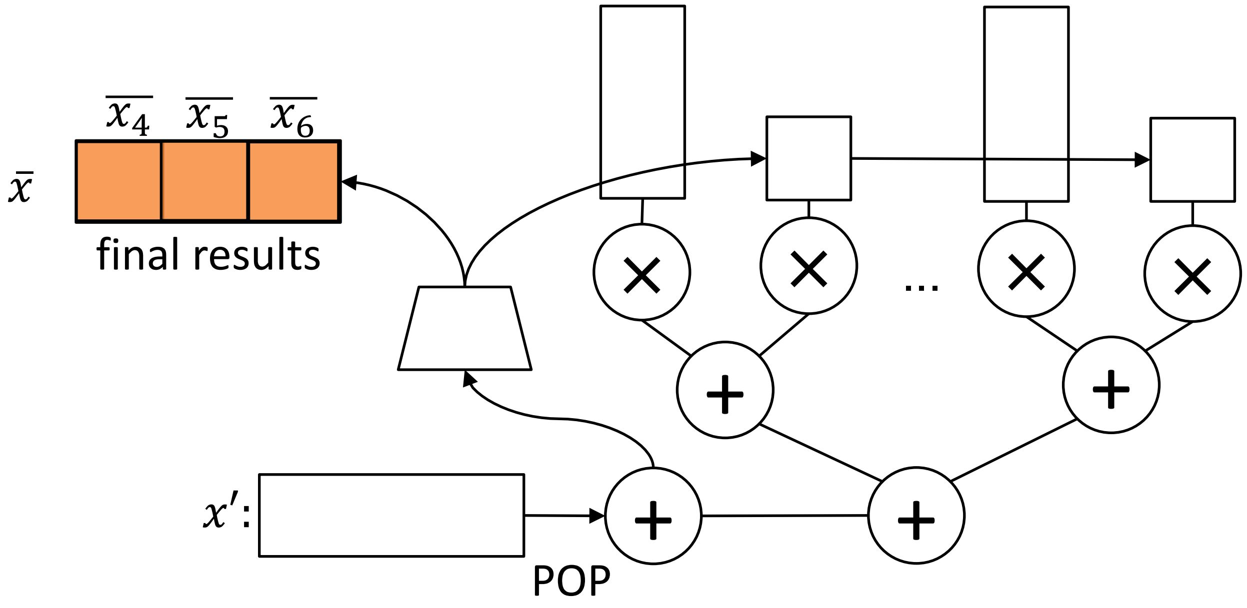
$$\overline{x_5} = x'_5 + A^T_{54} \times \overline{x_4} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{x_6} = x'_6 + A^T_{64} \times \overline{x_4} + A^T_{65} \times \overline{x_5} + A^T_{66} \times x_6$$

Reordering the operations

- Then, Alrescha
  - Executes SymGS using **same reduction tree**
  - Pops x' from the LIFO to reuse them fast
  - **Reuses new $\bar{x}$ immediately through shift connections**





$$x'_i = \sum_{j \neq 4,5,6} A^T_{ij} \times x_j$$

$$\overline{\boldsymbol{x_4}} = x'_4 + A^T_{44} \times x_4 + A^T_{45} \times x_5 + A^T_{46} \times x_6$$

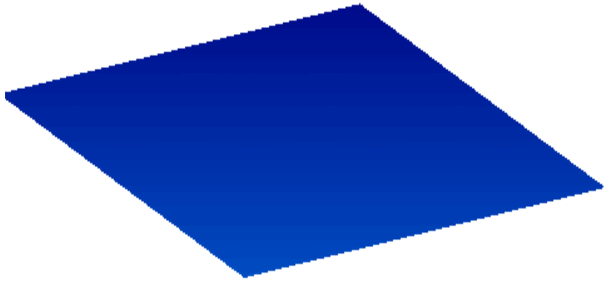
$$\overline{\boldsymbol{x_5}} = x'_5 + A^T_{54} \times \overline{\boldsymbol{x_4}} + A^T_{55} \times x_5 + A^T_{56} \times x_6$$

$$\overline{\boldsymbol{x_6}} = x'_6 + A^T_{64} \times \overline{\boldsymbol{x_4}} + A^T_{65} \times \overline{\boldsymbol{x_5}} + A^T_{66} \times x_6$$

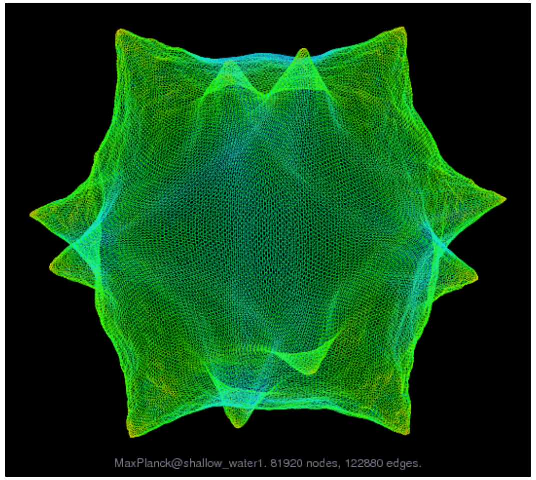
Putting them together for **sparse** matrices

What: Matrix $\boldsymbol{A}$ in $\boldsymbol{Ax} = \boldsymbol{b}$

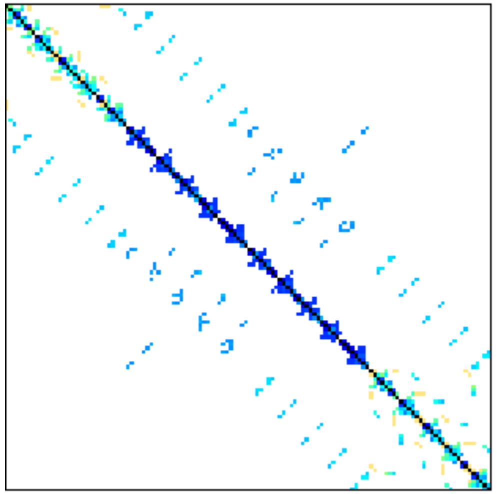
Why: Not all the points in a 3D-grid are occupied



Shallow-water equations[1]
(a set of PDEs)



Discretized to a 3D grid [2]



Matrix A

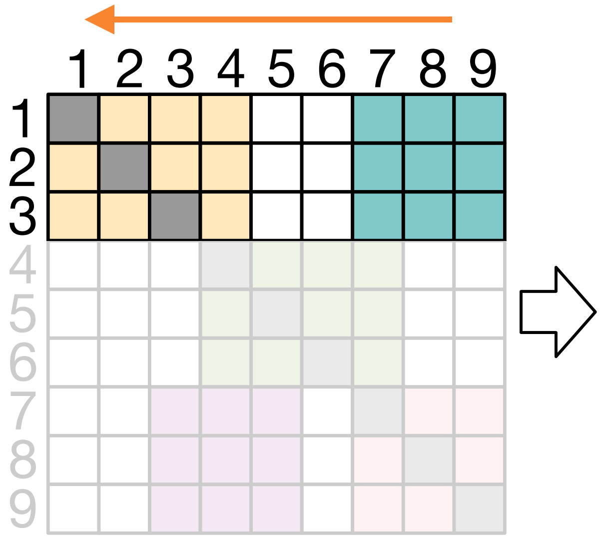
[1] https://en.wikipedia.org/wiki/Shallow_water_equations
[2] From Max-Plank Institute of Meteorology

Putting them together for **sparse** matrices

Alrescha accesses the vector $\boldsymbol{x}$ from cache (1KB)

Alrescha **streams** non-zero blocks[1] of matrix $\boldsymbol{A}$:

- First, $GEMV$ non-diagonals blocks
- Then, $SymGS$ on diagonal blocks



Order of operations

$GEMV$

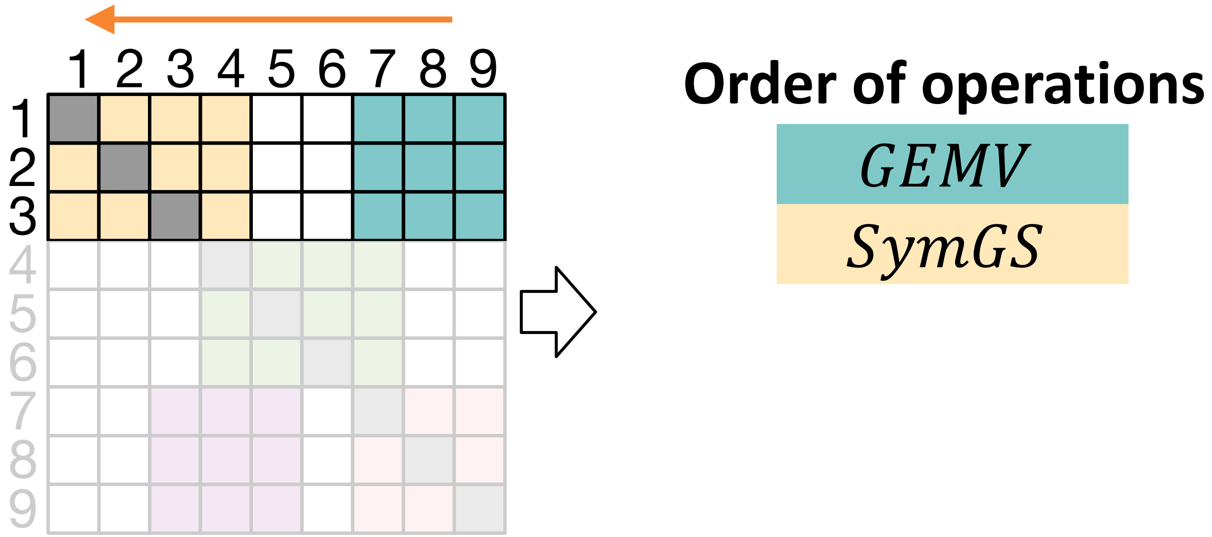
[1] As shown in prior work, the target scientific problems have block structure.

Putting them together for **sparse** matrices

Alrescha accesses the vector $\boldsymbol{x}$ from cache (1KB)

Alrescha **streams** non-zero blocks[1] of matrix $\boldsymbol{A}$:

- First, $GEMV$ non-diagonals blocks
- Then, $SymGS$ on diagonal blocks



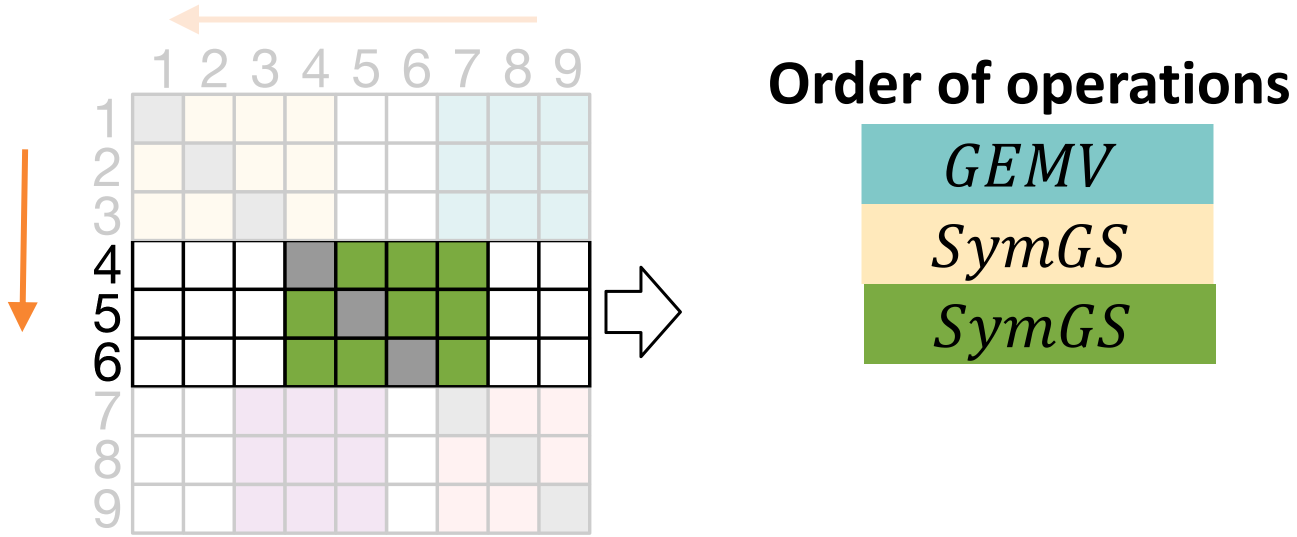
[1] As shown in prior work, the target scientific problems have block structure.

Putting them together for **sparse** matrices

Alrescha accesses the vector $\boldsymbol{x}$ from cache (1KB)

Alrescha **streams** non-zero blocks[1] of matrix $\boldsymbol{A}$:

- First, $GEMV$ non-diagonals blocks
- Then, $SymGS$ on diagonal blocks



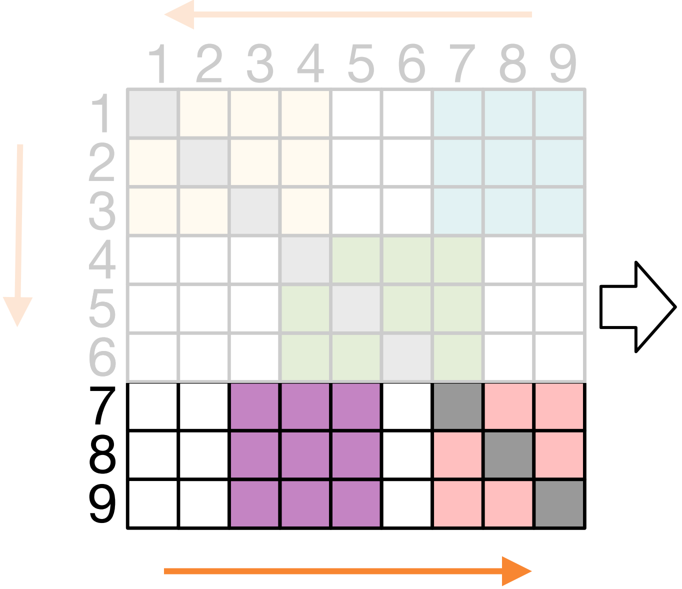
[1] As shown in prior work, the target scientific problems have block structure.

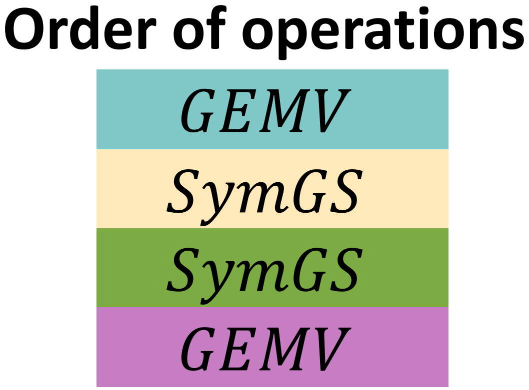
Putting them together for **sparse** matrices

Alrescha accesses the vector $\boldsymbol{x}$ from cache (1KB)

Alrescha **streams** non-zero blocks[1] of matrix $\boldsymbol{A}$:

- First, $GEMV$ non-diagonals blocks
- Then, $SymGS$ on diagonal blocks





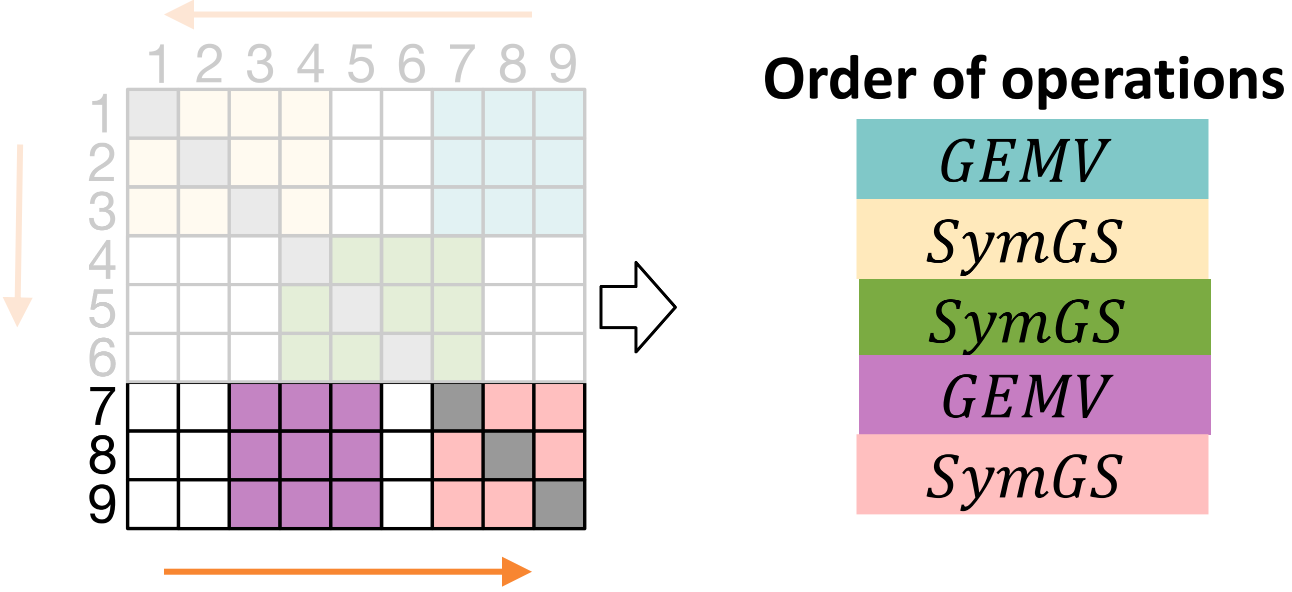
[1] As shown in prior work, the target scientific problems have block structure.

Putting them together for **sparse** matrices

Alrescha accesses the vector x from cache (1KB)

Alrescha **streams** non-zero blocks[1] of matrix A:

- First, $GEMV$ non-diagonals blocks
- Then, $SymGS$ on diagonal blocks



[1] As shown in prior work, the target scientific problems have block structure.

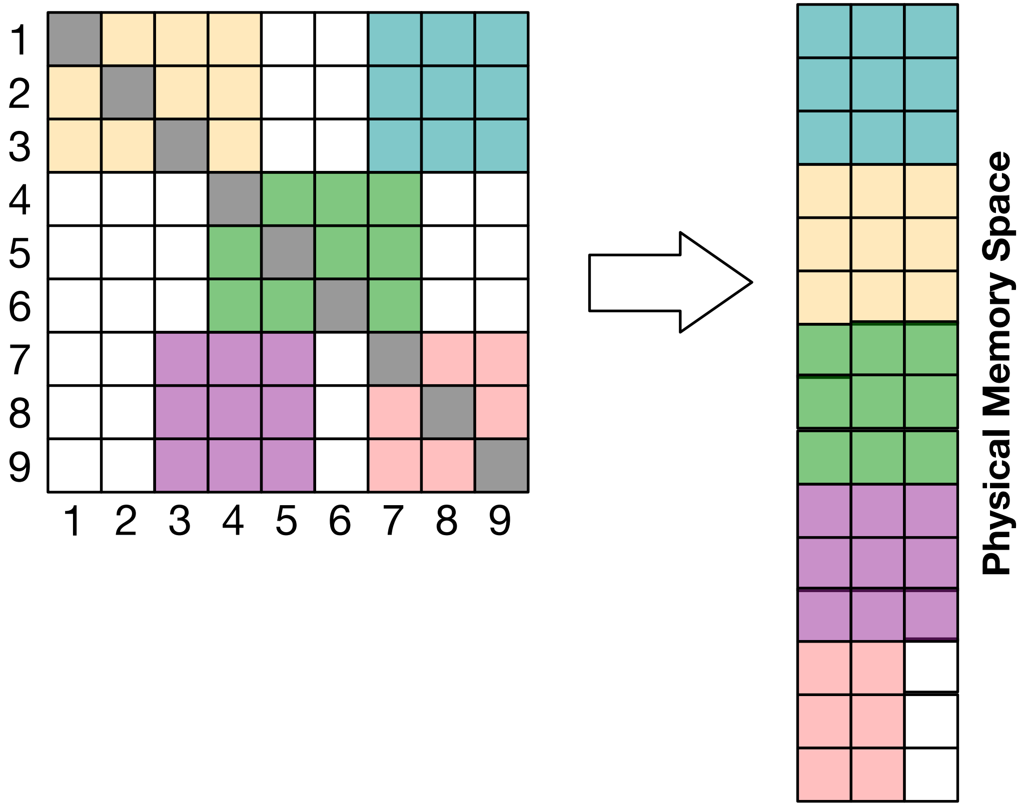
Outline

- Using PDEs for modeling and key challenges
- **Alrescha**
  - Main contributions
  - **Storage format**
  - Reconfigurable microarchitecture
  - Broad applications
- Results
- Conclusions

Storage format

Similar to BCSR[1] with linear overhead

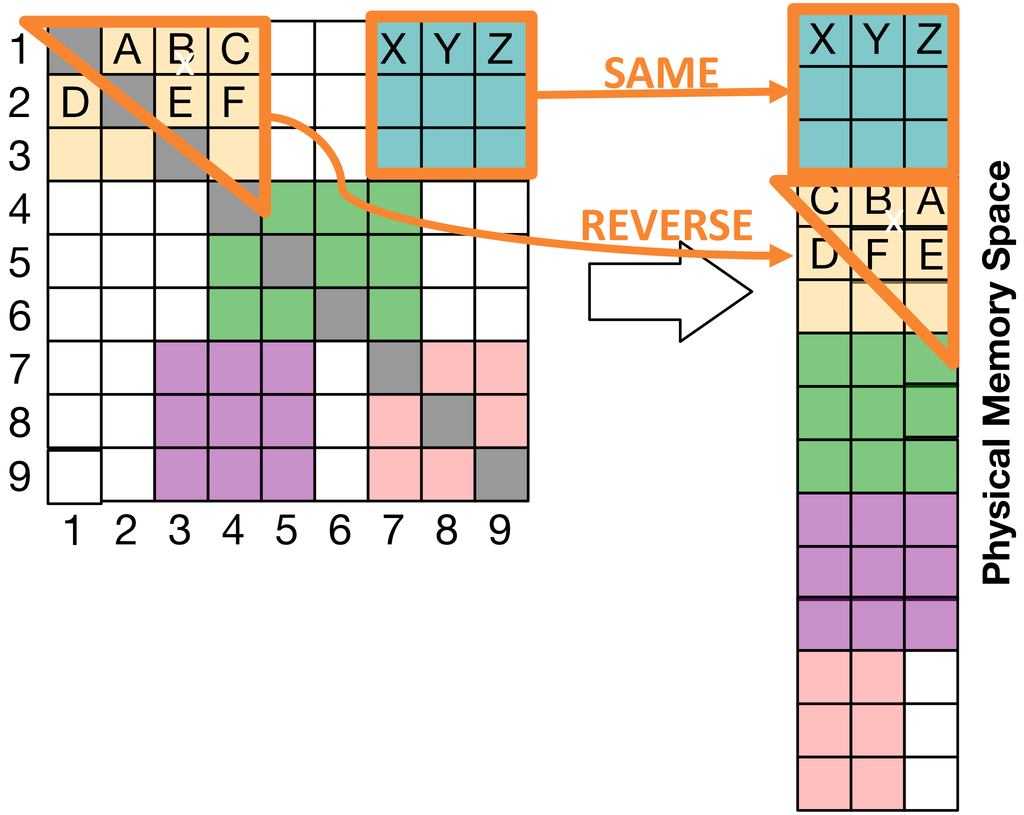
- Order of blocks:
  - Same as order of operations



Storage format

Similar to BCSR[1] with linear overhead

- Order of blocks:
  - Same as order of operations
- Order of elements:
  - Non-diagonal blocks: original
  - Up triangle of diagonal blocks: reverse

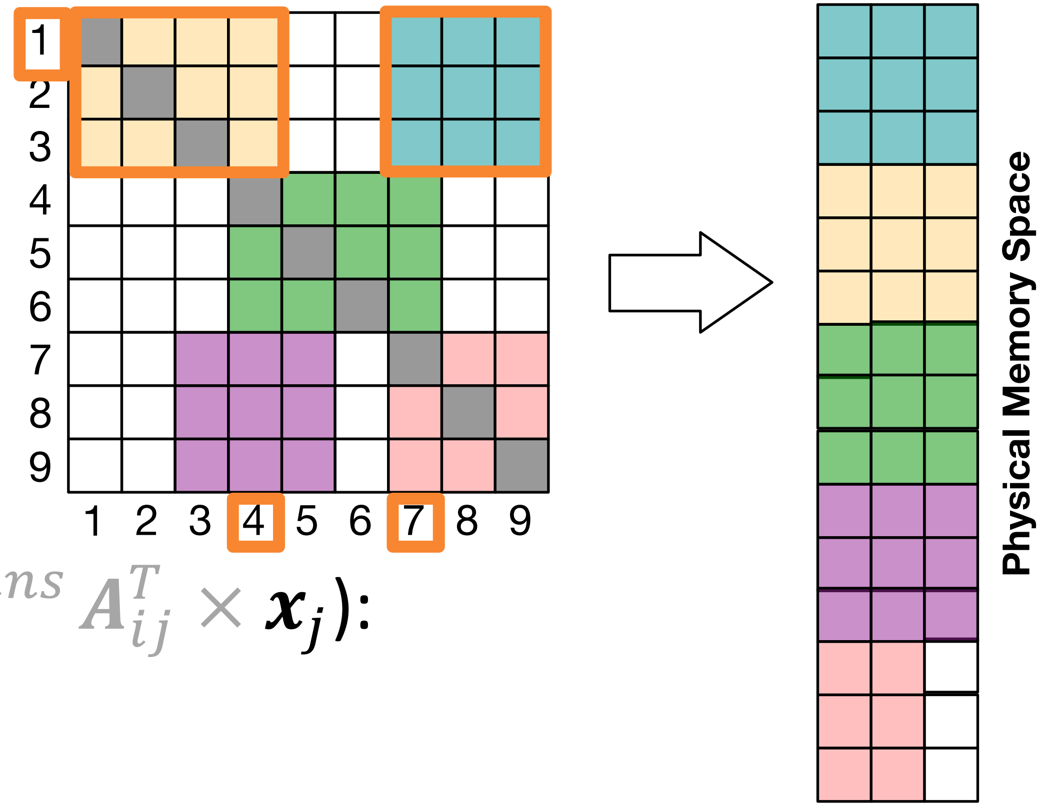


Blocked compressed sparse row (CSR)

Storage format

Similar to BCSR[1] with linear overhead

- Order of blocks:
  - Same as order of operations
- Order of elements:
  - Non-diagonal blocks: original
  - Up triangle of diagonal blocks: reverse
- Indexing (for cache access $\boldsymbol{x}_i = \sum_{j=0}^{columns} A_{ij}^T \times \boldsymbol{x}_j$):
  - Input indices: 7, 4,
  - Output indices: 1,

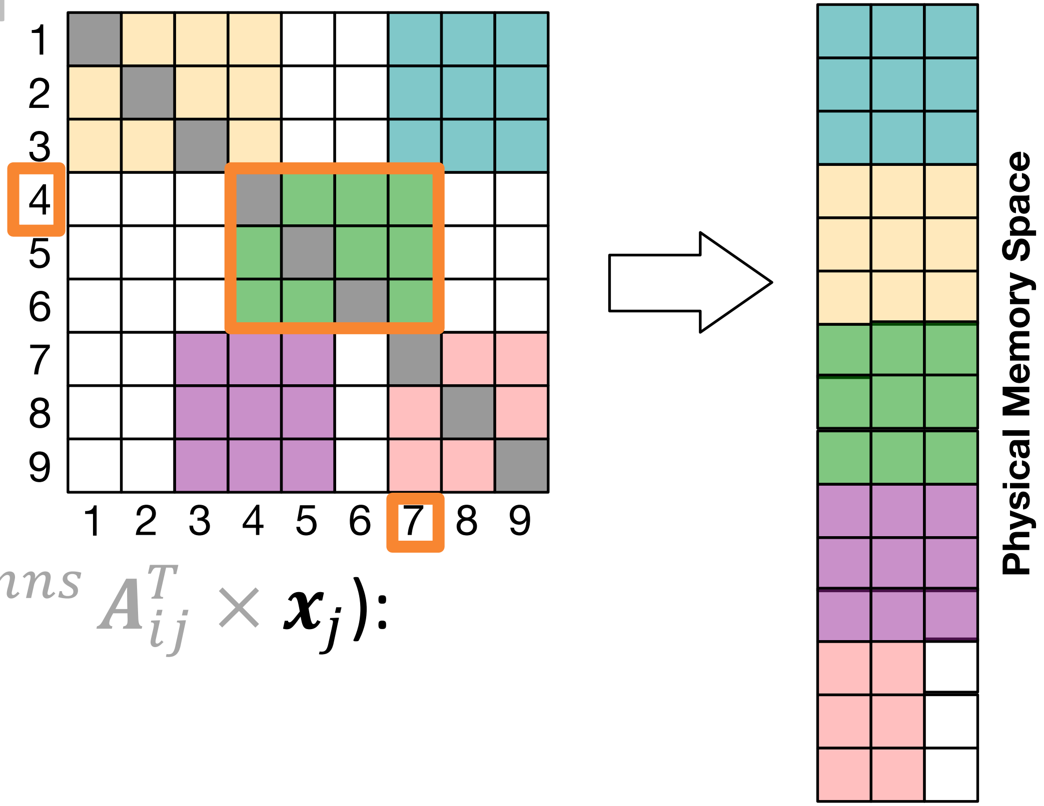


Blocked compressed sparse row (CSR)

Storage format

Similar to BCSR[1] with linear overhead

- Order of blocks:
  - Same as order of operations
- Order of elements:
  - Non-diagonal blocks: original
  - Up triangle of diagonal blocks: reverse
- Indexing (for cache access $\boldsymbol{x}_i = \sum_{j=0}^{columns} A_{ij}^T \times \boldsymbol{x}_j$):
  - Input indices: 7, 4, 7,
  - Output indices: 1, 4,

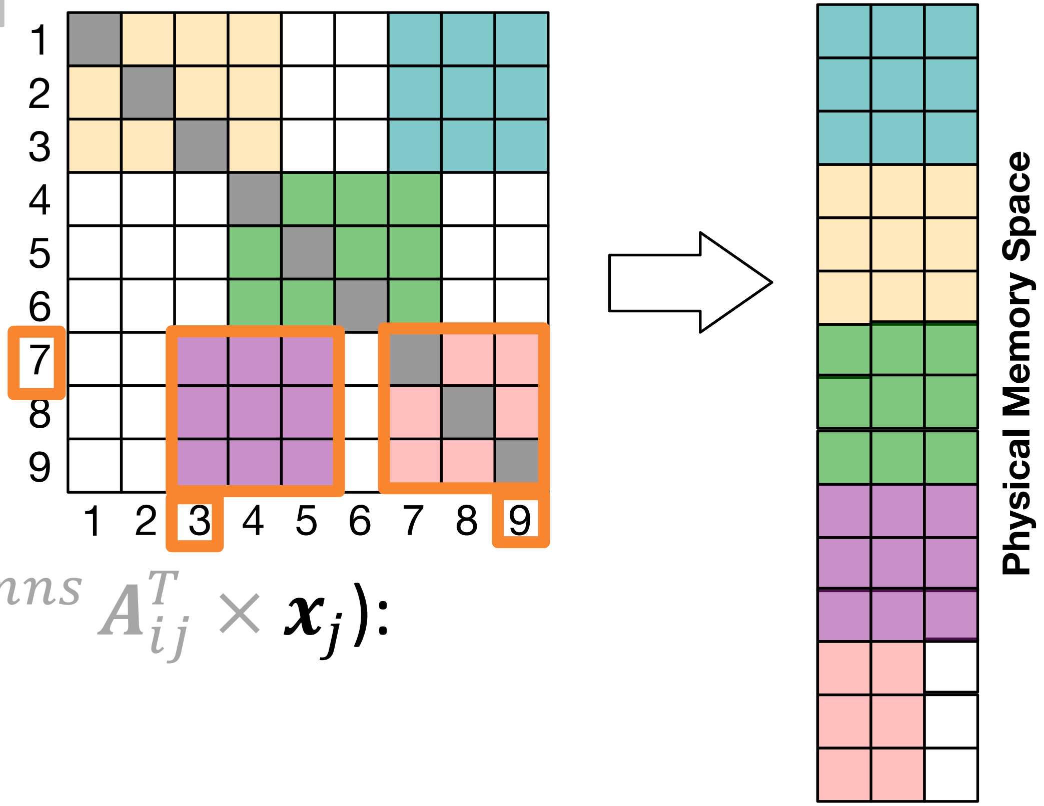


Blocked compressed sparse row (CSR)

Storage format

Similar to BCSR[1] with linear overhead

- Order of blocks:
  - Same as order of operations
- Order of elements:
  - Non-diagonal blocks: original
  - Up triangle of diagonal blocks: reverse
- Indexing (for cache access $\boldsymbol{x}_i = \sum_{j=0}^{columns} A_{ij}^T \times \boldsymbol{x}_j$):
  - Input indices: 7, 4, 7, 3, 9
  - Output indices: 1, 4, 7



[1] Blocked compressed sparse row (CSR)

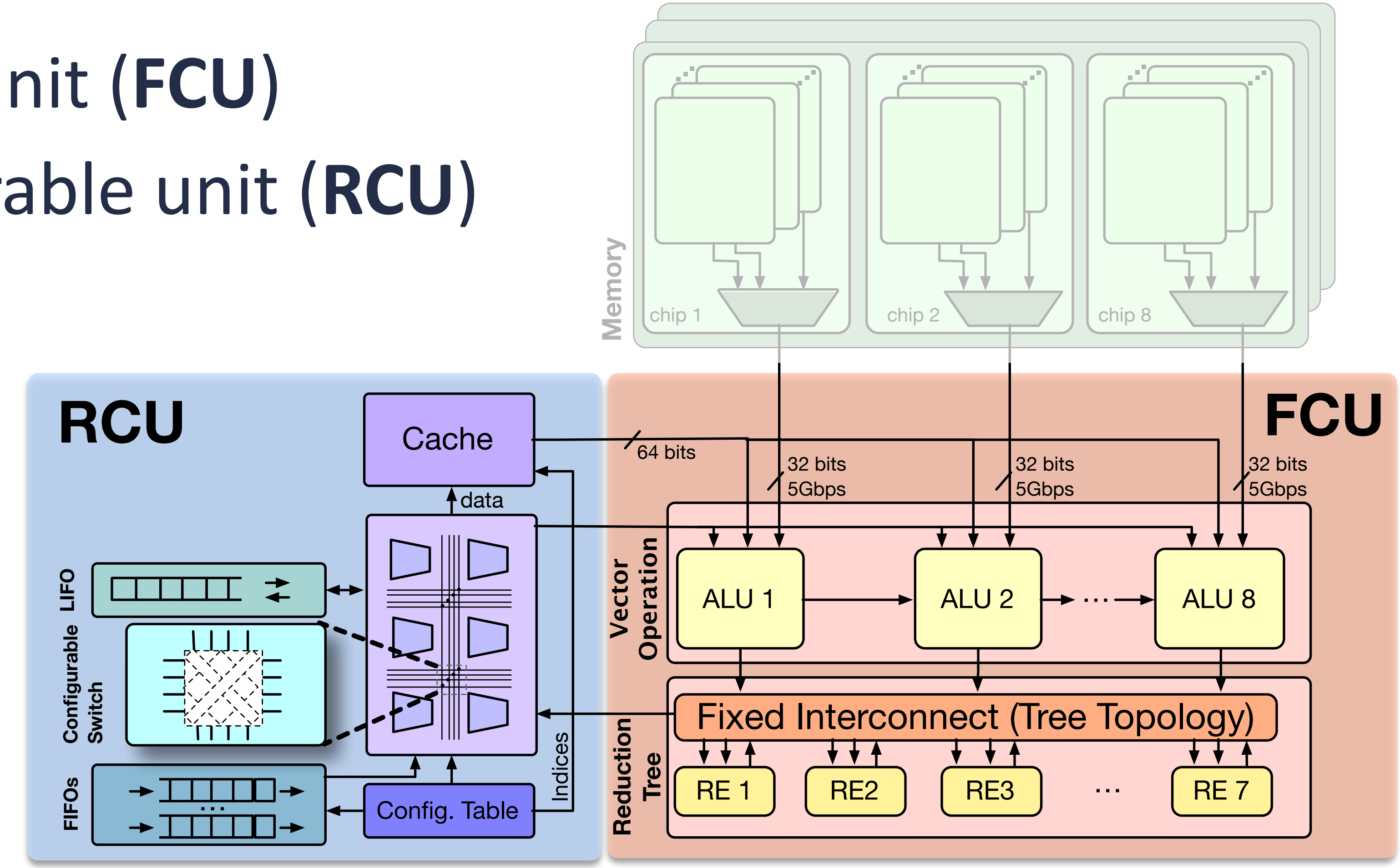
Outline

- Using PDEs for modeling and key challenges
- **Alrescha**
  - Main contributions
  - Storage format
  - **Reconfigurable microarchitecture**
  - Broad applications
- Results
- Conclusions

Lightweight reconfigurable microarchitecture

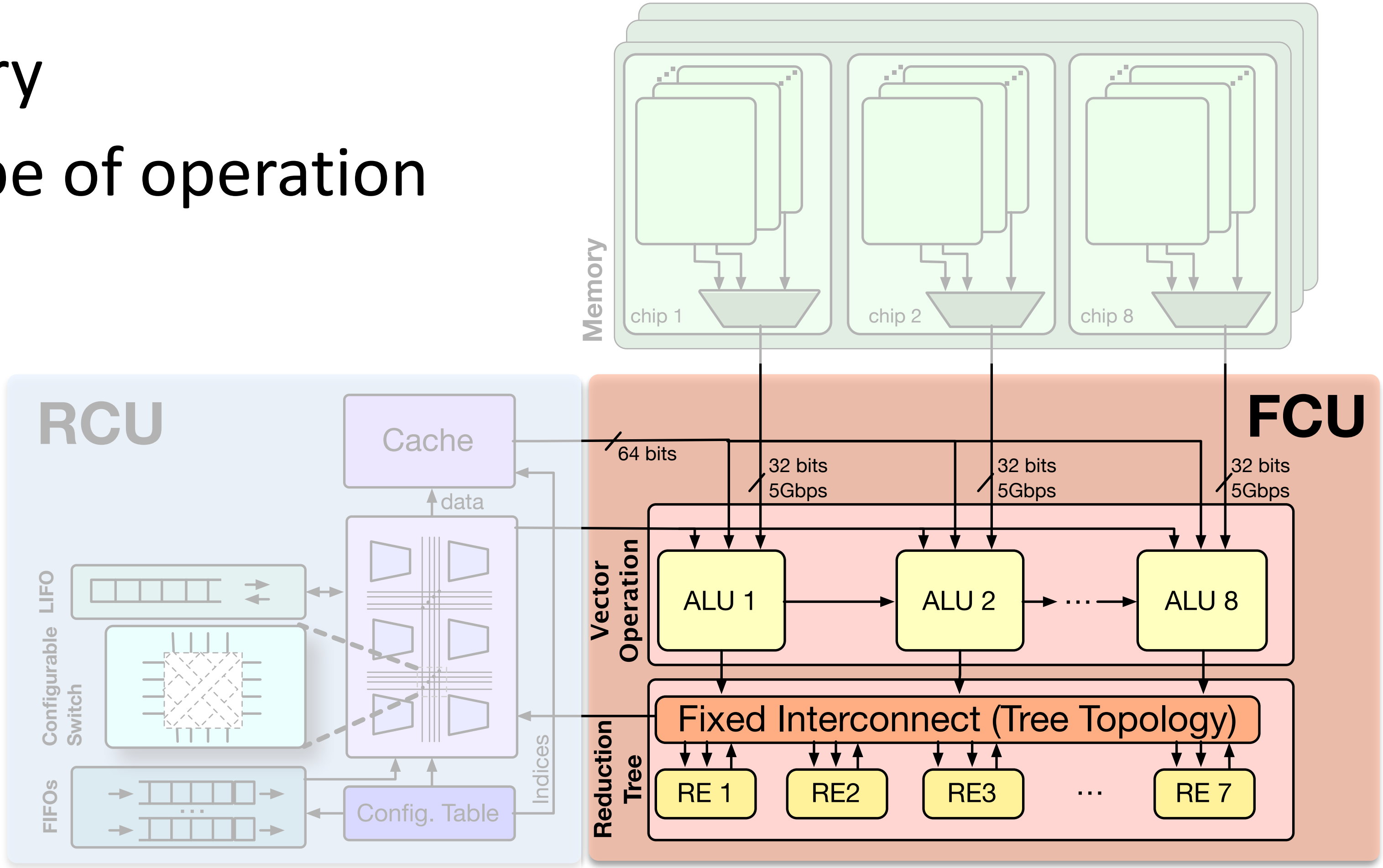
Alrescha includes:

- A fixed compute unit (**FCU**)
- A small reconfigurable unit (**RCU**)



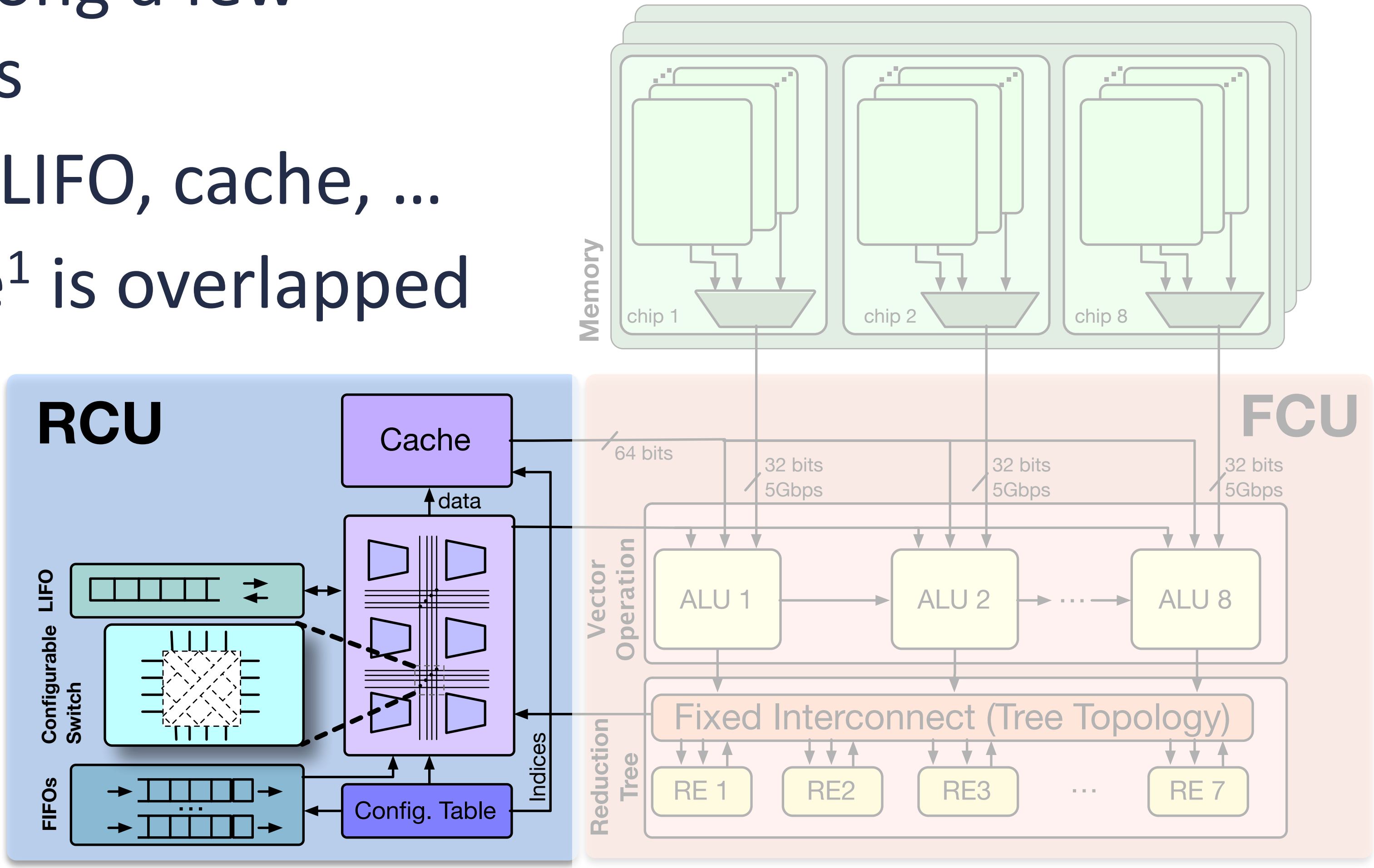
Fixed compute unit (FCU)

- Applies reduction on the streamed data
  - Directly from memory
  - Regardless of the type of operation



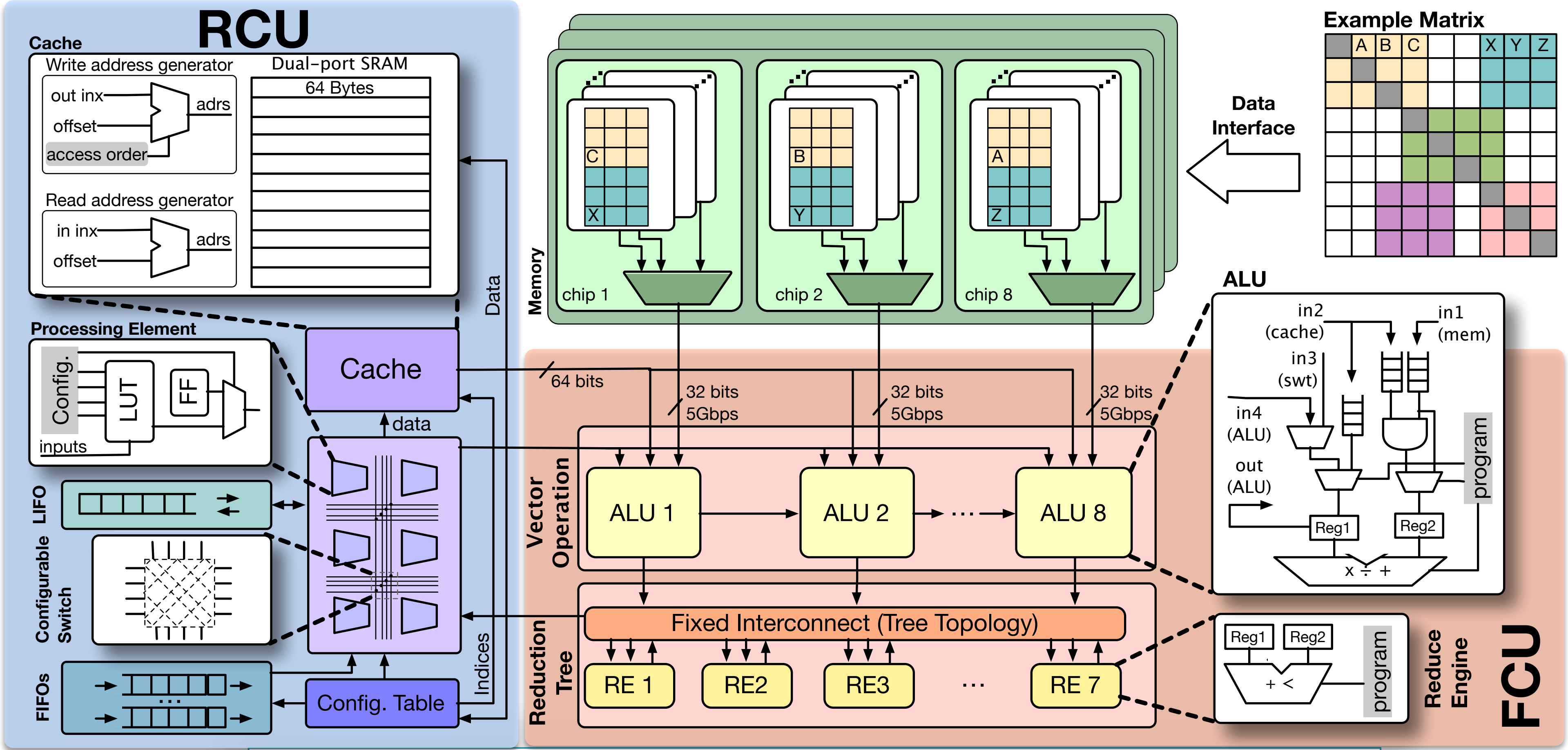
Reconfigurable compute unit (RCU)

- Interconnections among a few simple compute units
- Connections: FCU -> LIFO, cache, …
- Reconfiguration time[1] is overlapped with draining FCU
- Small and fast in both ASIC/FPGA



[1]Based on Xilinx Virtex-4 numbers using 90nm technology, it takes ~0.3 ns

Lightweight reconfigurable microarchitecture



For more details please refer to paper

Outline

- Using PDEs for modeling and key challenges
- **Alrescha**
  - Main contributions
  - Storage format
  - Reconfigurable microarchitecture
  - **Broad applications**
- Results
- Conclusions

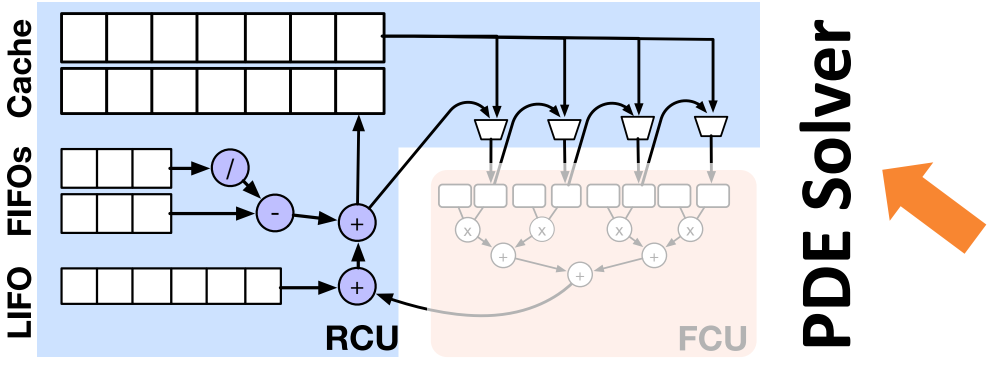
Broad Applications

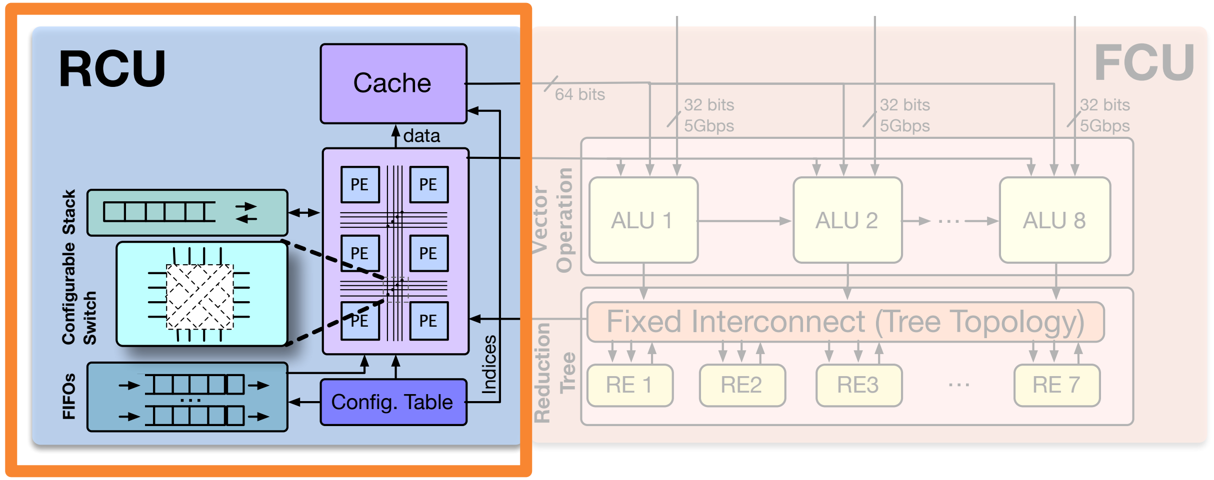
Alrescha accelerates **other sparse algorithms**, because of

- The common reduction engine
- The lightweight (partial) reconfigurable microarchitecture

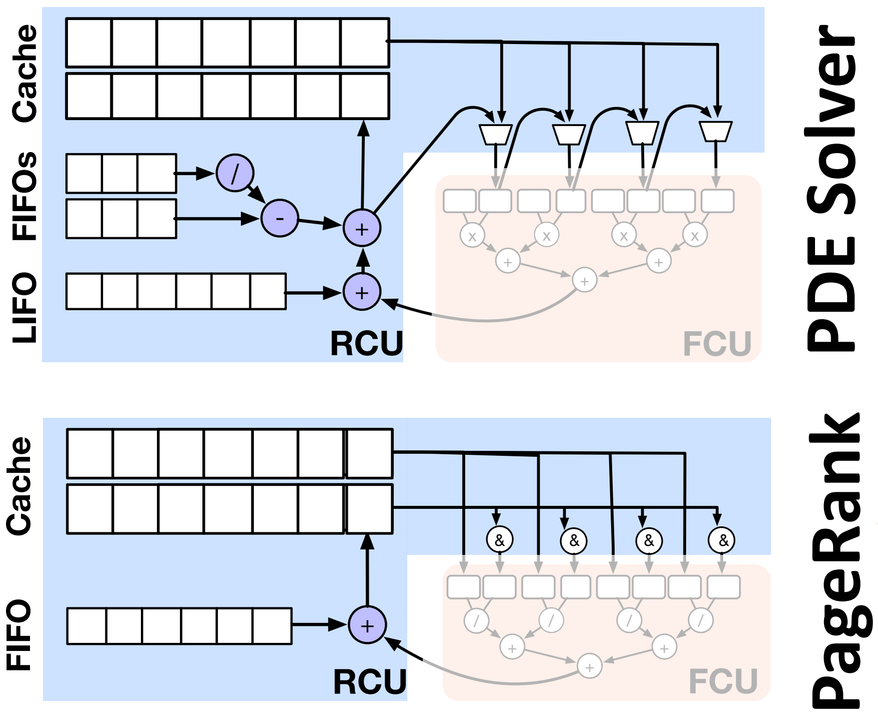
[1] SpMV: Sparse matrix vector multiplication

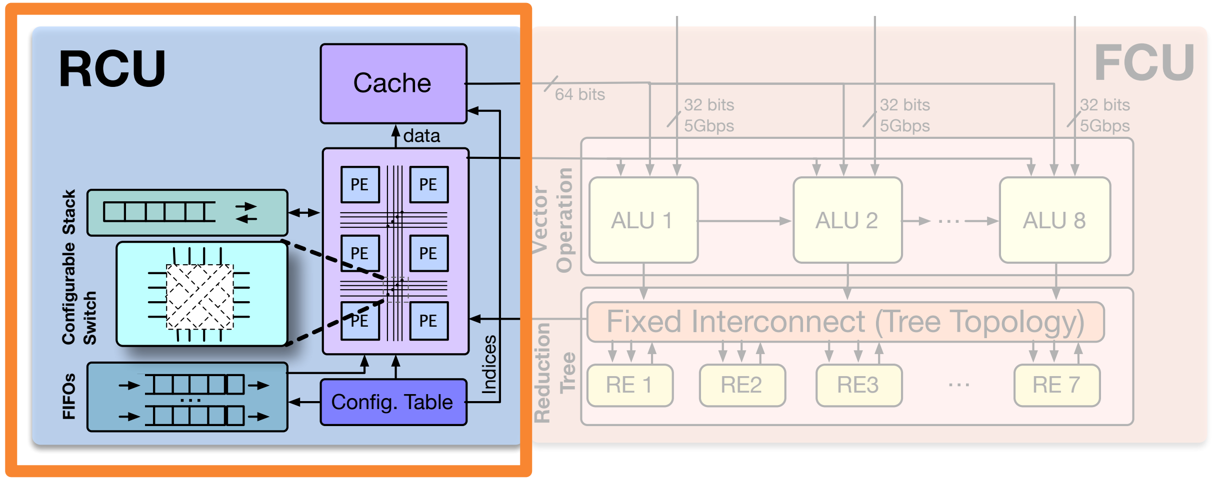
Broad Applications



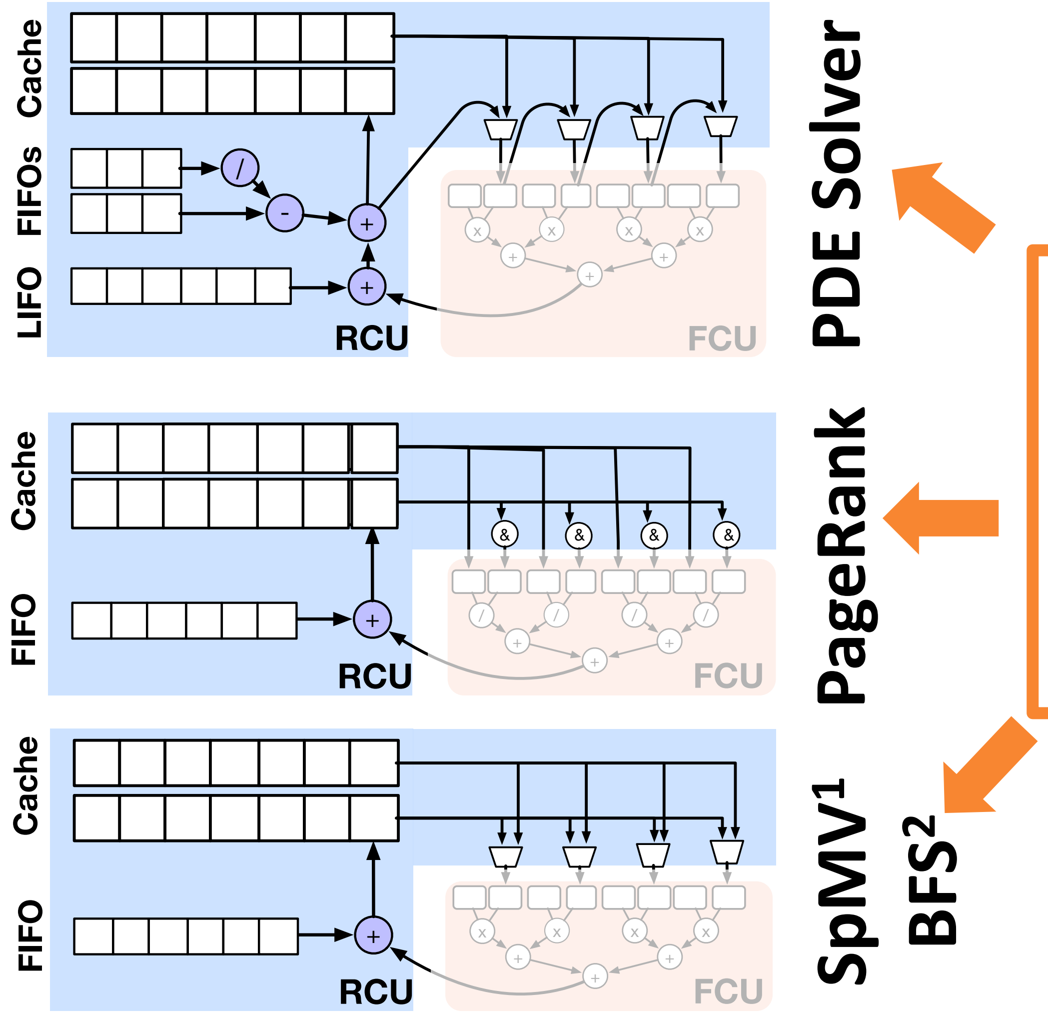


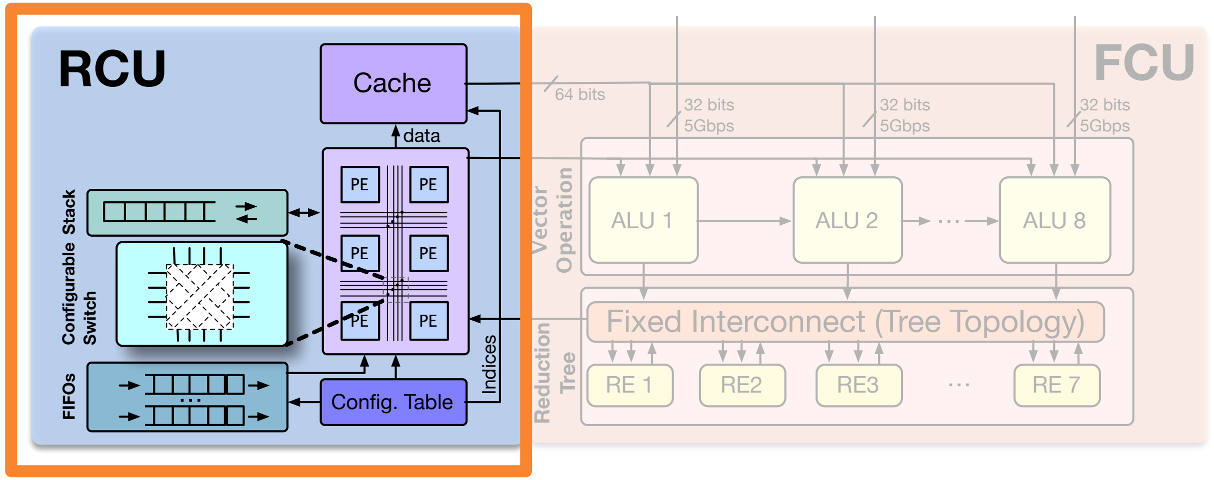
Broad Applications





Broad Applications





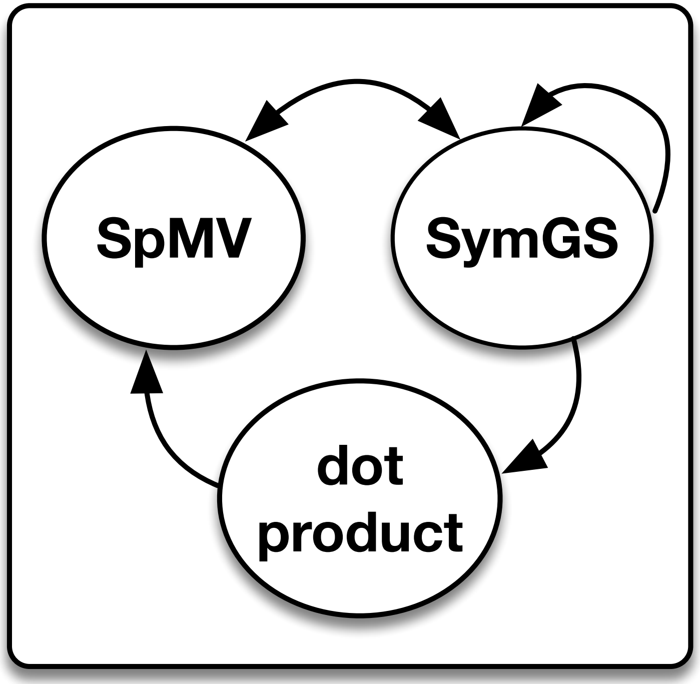
[1] SpMV: Sparse matrix-vector multiplication

[2] BFS: Breadth-first search

Broad Applications

Alrescha is the first multi-kernel sparse accelerator

- Problems including different kernels (e.g., SpMV and SymGS)



A PDE Solver

Outline

- Using PDEs for modeling and key challenges
- Alrescha
  - Main contributions
  - Storage format
  - Reconfigurable microarchitecture
  - Broad applications
- **Results**
- Conclusions

Experimental Setup

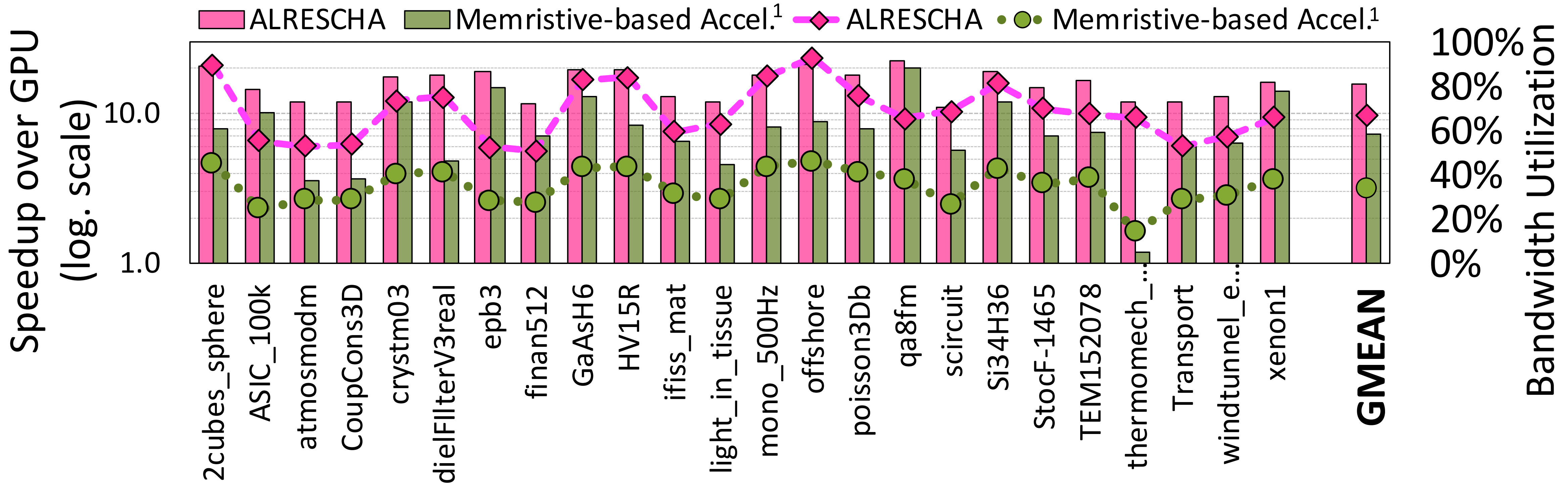
- Implementation:
  - Preprocessing: Matlab
  - Simulation: Cycle-accurate C++ simulator
- Benchmarks
  - Algorithms: PCG, SpMV, BFS, SSSP, PageRank
  - Datasets: Sparse matrices from SuiteSparse collection[1]
- Baselines
  - CPU: Intel Xeon E5
  - GPU: NVIDIA Tesla K40c
  - State-of-the-art accelerators: Memristive[2], OuterSPACE[3], GraphR[4]
  - Memory bandwidth is similar among comparisons

[1] https://sparse.tamu.edu/
[2] B. Feinberg et al. ISCA'18
[3] S. Pal, et al. HPCA'18
[4] L. Son, et al. HPCA'18

Speedup for scientific workloads

Alrescha resolves performance bottleneck of PDE solvers

- 2.1x speedup compared to emerging technologies
- 15.6x speedup over GPU

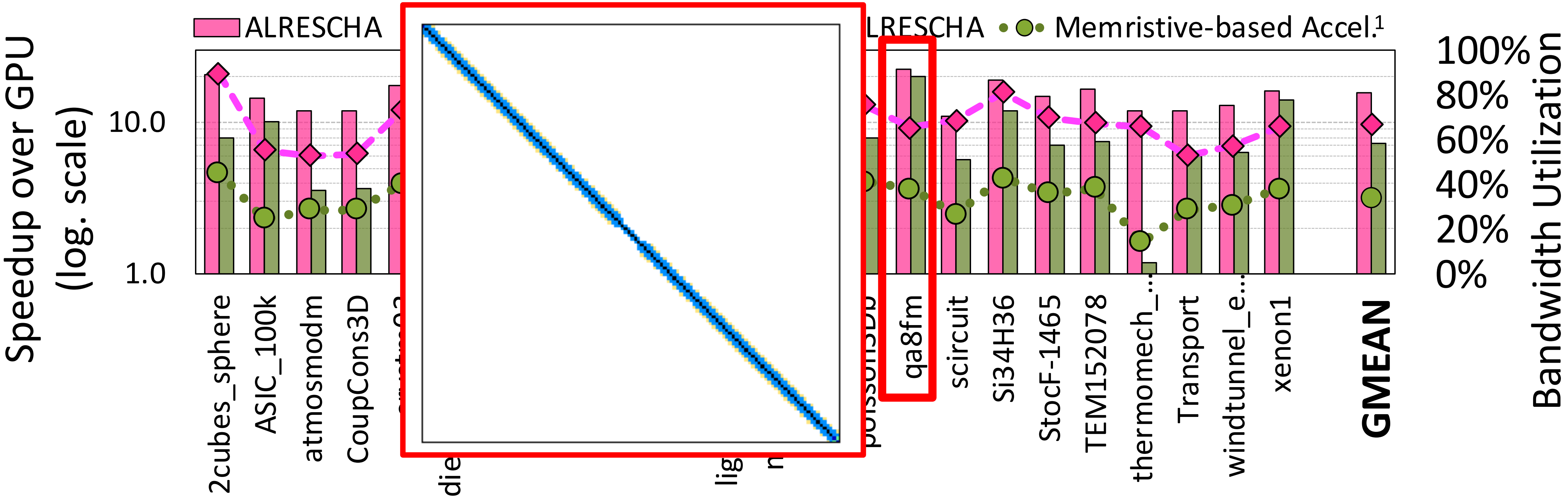


[1] B. Feinberg et al. "Enabling scientific computing on memristive accelerators," ISCA'18

Speedup for scientific workloads

Alrescha resolves performance bottleneck

- 2.1x speedup compared to emerging technologies
- 15.6x speedup over GPU

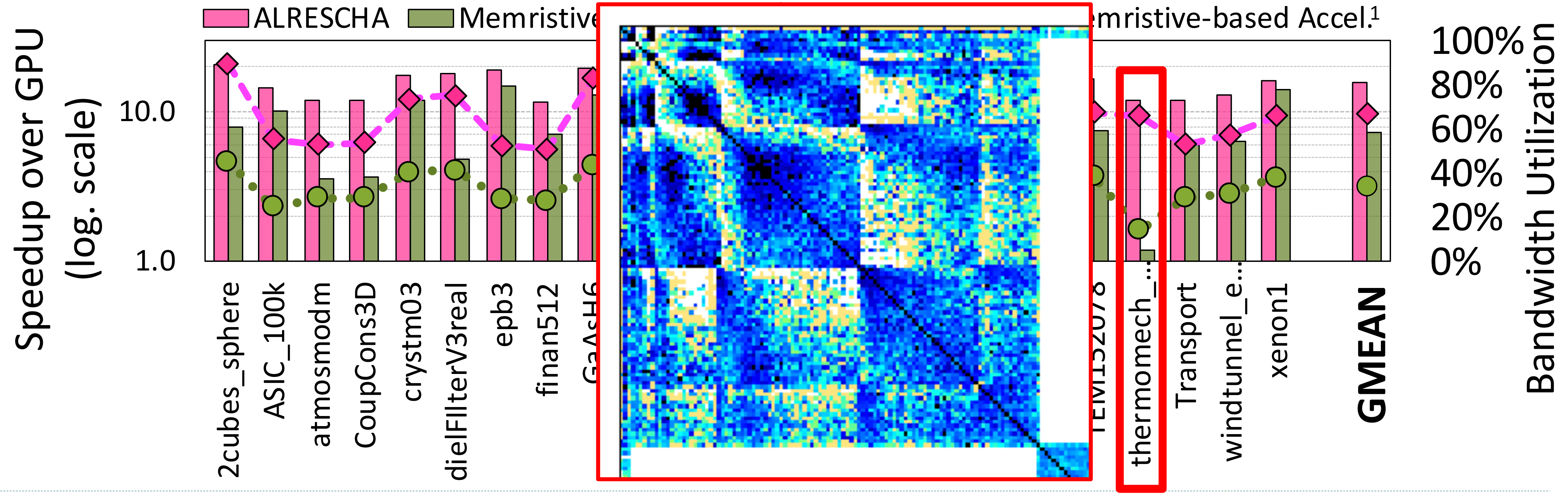


[1] B. Feinberg et al. "Enabling scientific computing on memristive accelerators," ISCA'18

Speedup for scientific workloads

Alrescha resolves performance bottleneck

- 2.1x speedup compared to emerging technologies
- 15.6x speedup over GPU

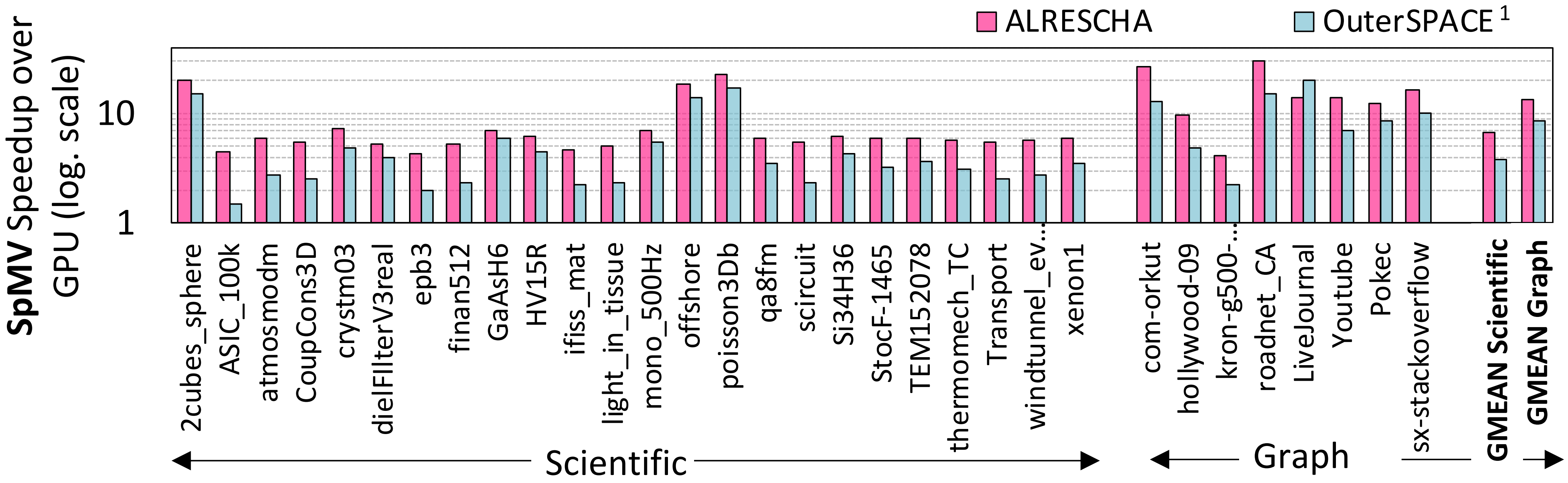


[1] B. Feinberg et al. "Enabling scientific computing on memristive accelerators," ISCA'18

Speedup for Other Applications

Alrescha provides better reusability

- 13.6x speedup over GPU for scientific workloads
- 6.9x speedup over GPU for graph workloads



[1] S. Pal, J. Beaumont, et al. "Outerspace: An outer product based sparse matrix multiplication accelerator," HPCA'18

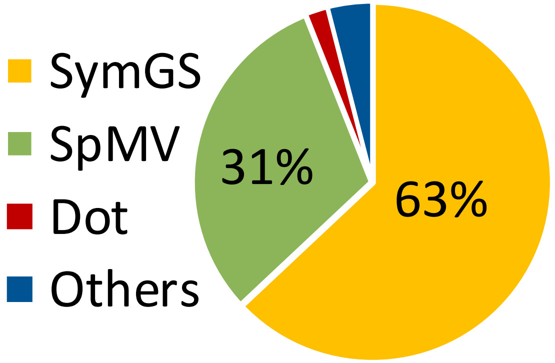
Outline

- Using PDEs for modeling and key challenges
- Alrescha
  - Main contributions
  - Storage format
  - Reconfigurable microarchitecture
  - Broad applications
- Results
- **Conclusions**

Conclusions and Further Impacts

Alrescha:

- Accelerates the **key kernels** of scientific problems



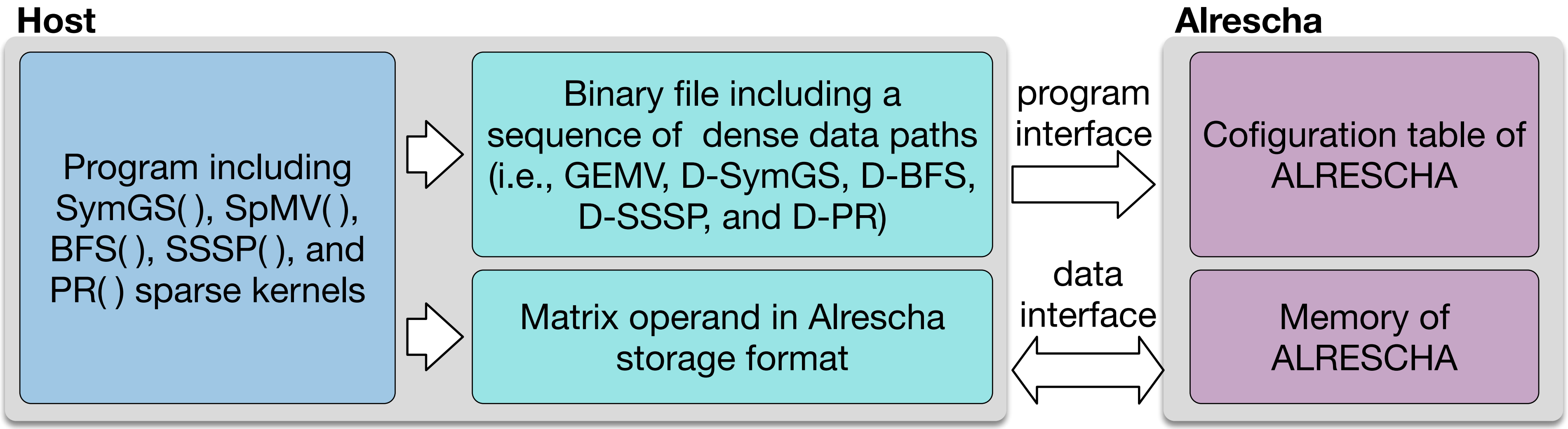
- Is the **first multi-kernel** sparse accelerator
- Has **broad applications** (e.g., scientific problems, graph, SpMV)
- Does not require emerging technologies

Further Impacts and Applications:

- Can accelerate any sparse problem that includes reduction operations
- Can leverage **partial reconfigurability** of FPGAs

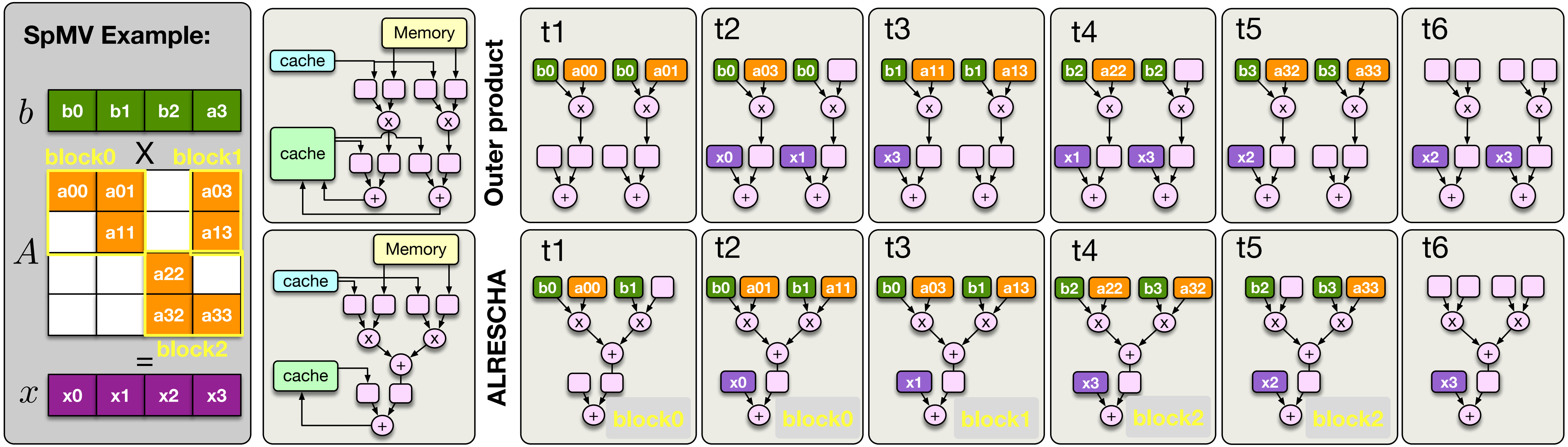
Backup Slides

How does Alrescha work?



Back to list

How Alrescha provides locality in output?

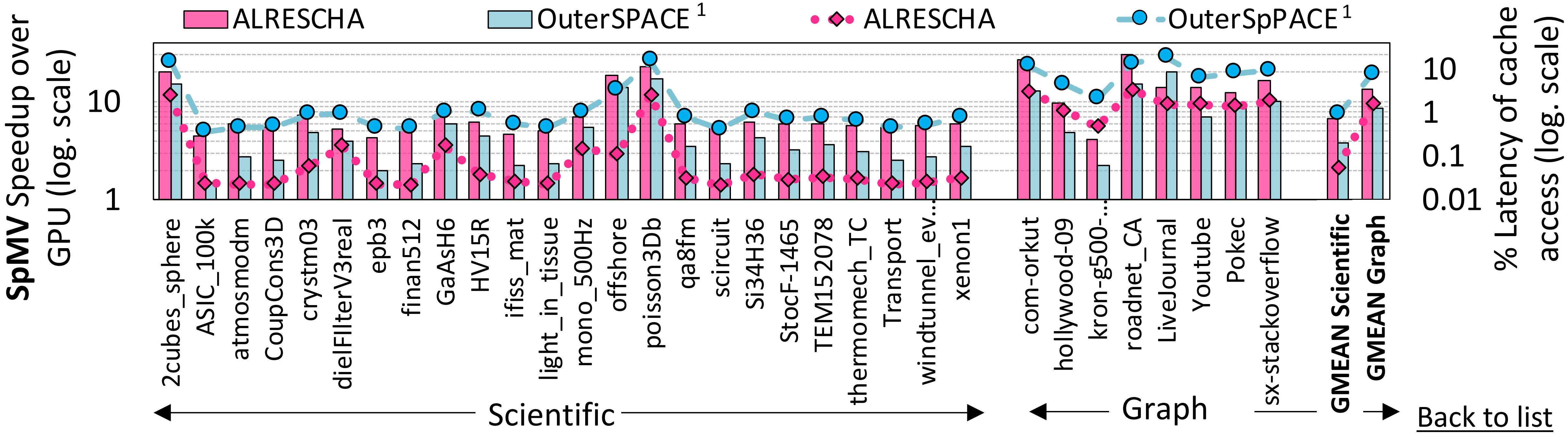


Back to list

Speedup for Other Applications

Alrescha provides better reusability

- 13.6x speedup over GPU for scientific workloads
- 6.9x speedup over GPU for graph workloads



Back to list

[1] S. Pal, J. Beaumont, et al. “Outerspace: An outer product based sparse matrix multiplication accelerator,” HPCA’18

How do the exact mathematical expressions look like?

$$x_j^t = \frac{1}{A_{jj}^T} - (b_j - \sum_{i=1}^{j-1} A_{ij}^T \times x_i^t - \sum_{i=j+1}^{n} A_{ij}^T \times x_i^{t-1})$$

$$x_j^t = (\frac{1}{A_{jj}^T} - b_j) + (\sum_{i=1}^{j-1} A_{ij}^T \times x_i^t + \sum_{i=j+1}^{n} A_{ij}^T \times x_i^{t-1})$$

Back to list

How do you program Alrescha?

Algorithm 1 Convert Algorithm

```
function CONVERT(KernelType, A_nxn, ω)
    A_nxn: sparse matrix, ω : block width
    DP: Data path type
    l2r: left to right, r2l: right to left
    Inx_in := 0, Inx_out := 0
    Blocks[] = Split(A, ω) // partitions A to ω × ω blocks
    m = n/ω
    for (i = 1, i < m, i++) do
        for (j = 1, i < m, j++) do
            if (nnz(Blocks[i, j]) > 0) then
                if KernelType != SymGS then
                    DP = KernelType.DataPath
                    Inx_in = i.ω, Inx_out = j.ω
                    Order = l2r
                    Op = port1 // the operand vector
                else
                    if (i != j) then
                        DP = GEMV
                        Inx_in = j.ω
                        Inx_out = −1 // no write to cache
                        Order = l2r
                        if (i > j) then
                            Op = port2 //which is x^(t−1)
                        else
                            Op = port1 //which is x^t
                    else
                        DP = D-SymGS
                        Inx_in = j.ω, Inx_out = (i + 1).ω
                        Order = r2l
                        Op = port2 //which is x^(t−1)
                Add2Table(DP, Inx_in, Inx_out, Order, Op)
```

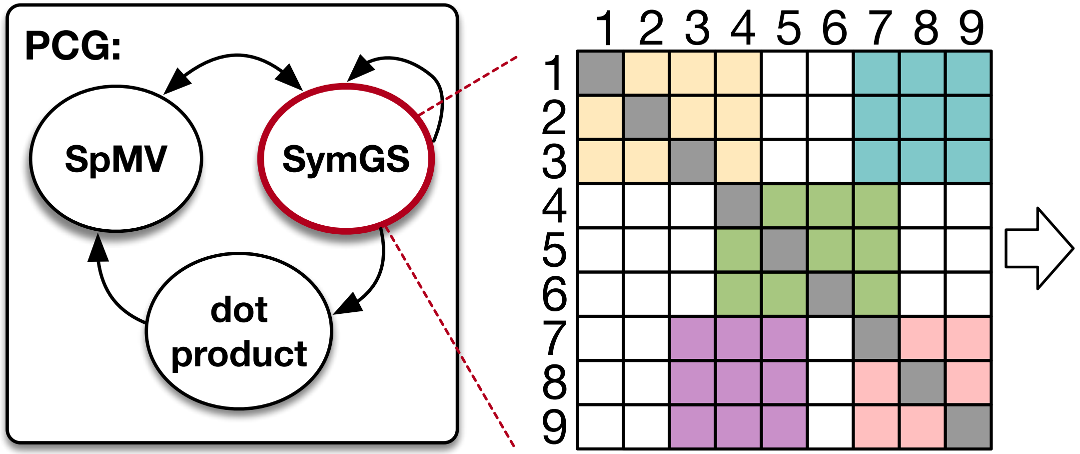
Back to list

Why Alrescha can execute other sparse problems?

Sparse Kernel	Sparse Application	Dense Data Paths	Phase 1 (vector operation)				Phase 2 (reduce)	Phase 3 (assign)
			vector operand1	vector operand2	vector operand3	operation		
SymGS	PDE solving	D-SymGS/GEMV	a row of coefficient matrix	the vector from iteration (i-1)	the vector at iteration (i)	multiplication	sum	apply operation with A^T and b_j and update vector
SpMV	PDE solving and graph	GEMV	a row of coefficient matrix	the vector from iteration (i-1)	N/A	multiplication	sum	sum and update the vector
Page Rank	Graph	D-PR	a column of adjacency matrix	the out-degree vector of vertices	the rank vector at iteration (i-1)	AND/division	sum	rank vector update
BFS	Graph	D-BFS	a column of adjacency matrix	the frontier vector	N/A	sum	min	compare and update distance vector
SSSP	Graph	D-SSSP	a column of adjacency matrix	the frontier vector	N/A	sum	min	compare and update distance vector

Back to list

What is configuration table?



DP	Inx_{in}	Inx_{out}	$Order$	Op
$GEMV$	7	-	$l2r$	x^{t-1}
$D-SymGS$	4	1	$r2l$	x^{t-1}
$D-SymGS$	7	4	$r2l$	x^{t-1}
$GEMV$	3	-	$l2r$	x^{t}
$D-SymGS$	9	7	$r2l$	x^{t-1}

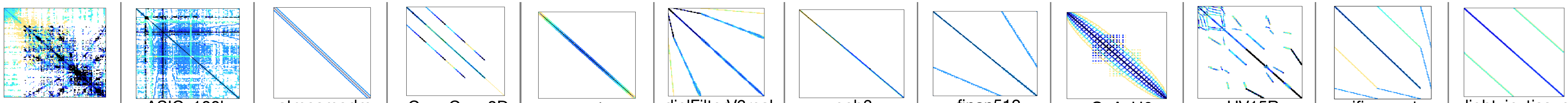
What is the details of Alrescha & baseline config.

Floating point	double precision (64 bits)
Clock frequency	2.5 GHz
Cache	1KB, 64-Byte lines, 4-cycle access latency
RE latency	3 Cycles (sum: 3, min: 1)
ALU latency	3 Cycles
Memory	12 GB GDDR5, 288 GB/s

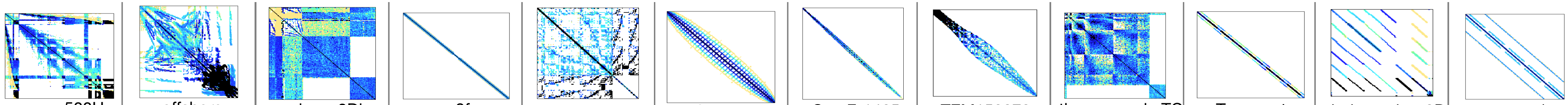
GPU baseline	
Graphics card	NVIDIA Tesla K40c, 2880 CUDA cores
Architecture	Kepler
Clock frequency	745MHz
Memory	12 GB GDDR5, 288 GB/s
Libraries	Gunrock [37] and CUSPARSE
Optimizations	row reordering (coloring) [8], ELL format
CPU baseline	
Processor	Intel Xeon E5-2630 v3 8-core
Clock frequency	2.4 GHz
Cache	64 KB L1, 256 KB L2, 20 MB L3
Memory	128 GB DDR4, 59 GB/s
Platforms	CuSha [39], GridGraph [38]

Back to list

How do the scientific workloads look like?



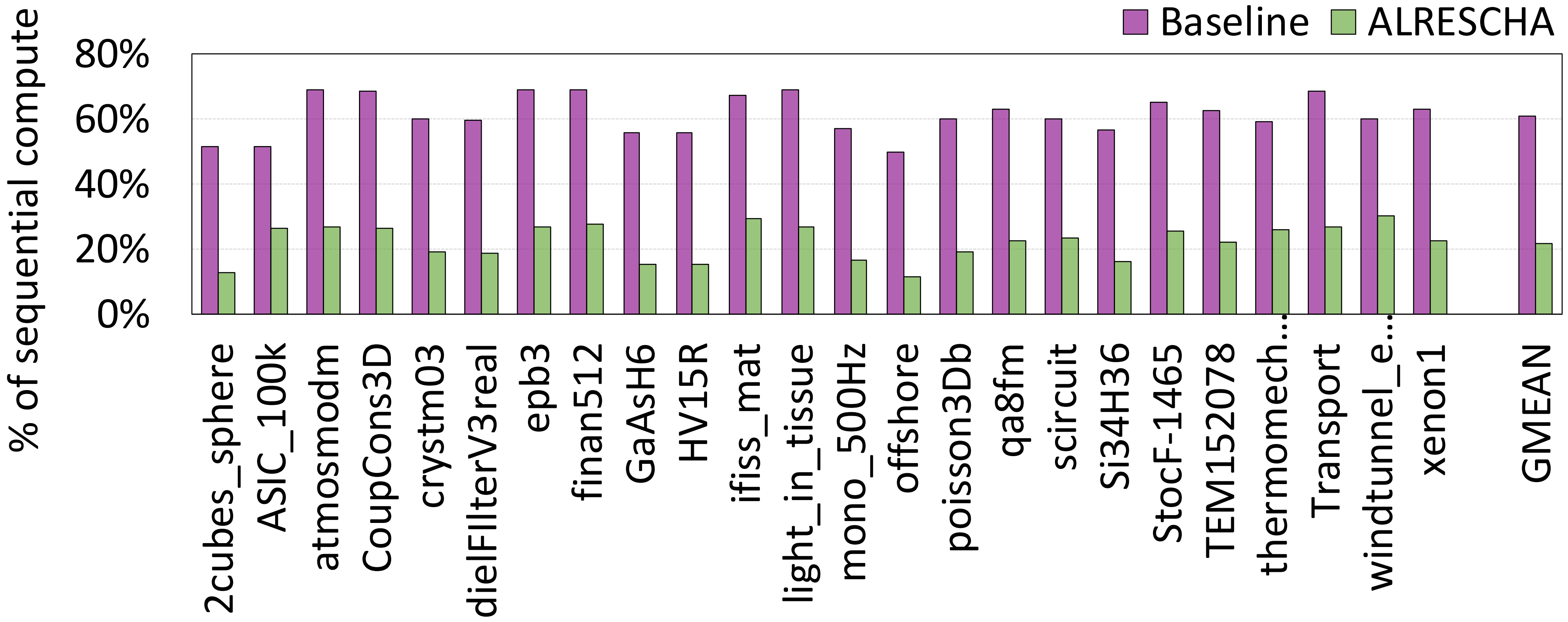
Name	2cubes_sphere	ASIC_100k	atmosmodm	CoupCons3D	crystm	dielFilterV3real	epb3	finan512	GaAsH6	HV15R	ifiss_mat	light_in_tissue
Row/Col	101,492	99,340	1,489,752	416,800	24,696	1,102,824	84,617	74,752	61,349	2,017,169	96,307	29,282
Kind	Electromagnetic	Circuit Simul.	Fluid Dynamics	Structural Prob.	Materials Prob.	Electromagnetic	Thermal Prob.	Economic Prob.	Chemistry Prob.	Fluid Dynamics	Fluid Dynamics	Electromagnetics
NNZ	1,647,264	954,163	10,319,760	17,277,420	583,770	89,306,020	463,625	596,992	3,381,809	283,073,458	3,599,932	406,084



Name	mono_500Hz	offshore	poisson3Db	qa8fm	scircuit	Si34H36	StocF-1465	TEM152078	thermomech_TC	Transport	windtunnel_ev3D	xenon1
Row/Col	169,410	259,789	85,623	66,127	170,998	97,569	1,465,137	152,078	102,158	1,602,111	40,816	48,600
Kind	Acoustics Prob.	Electromagnetic	Fluid Dynamics	Acoustics Prob.	Circuit Simul.	Chemistry Prob.	Fluid Dynamics	Electromagnetics	Thermal Prob.	Structural Prob.	Fluid Dynamics	Materials Prob.
NNZ	5,036,288	4,242,673	2,374,949	1,660,579	958,936	5,156,379	21,005,389	6,459,326	711,558	23,487,281	803,978	1,181,120

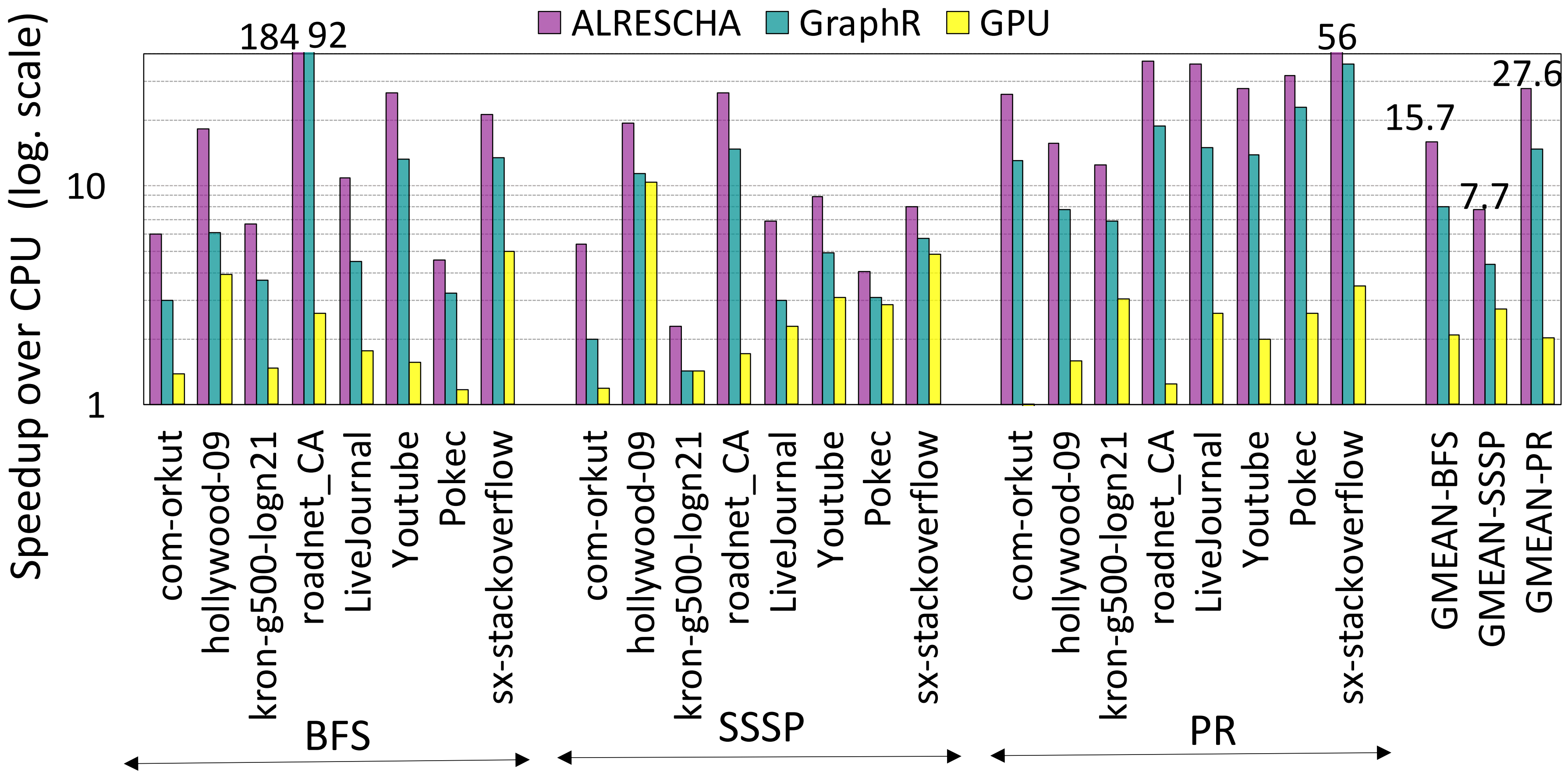
Back to list

How much Alrescha reduces sequential computations?



Back to list

Can Alrescha accelerate graph algorithms?

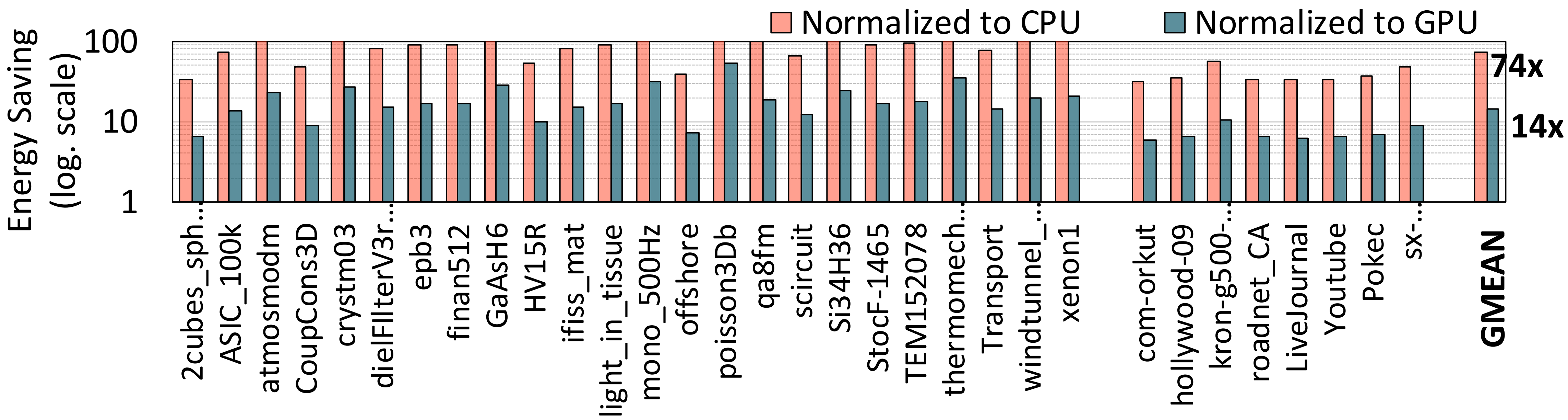


Back to list

Energy Consumption

Alrescha substitutes many memory accesses with

- Computations
- Local cache accesses



Back to list

Comparison with state-of-the-art accelerators

		GraphR [24]	OuterSPACE [18]	Memristive-Based Accelerator [25]	Row Reordering Matrix Coloring [8]	Alrescha (our work)
Application Domain		Graph	Graph (only SpMV)	PDE solver	PDE solver	**Graph and PDE solver**
Hardware	**Multi-Kernel Support**	✗	✗	✗	✗	✓
	BW Utilization	Low	Moderate	Low	Moderate	**High**
	NOT Transferring Meta-data	✗	✗	✗	✗	✓
	Processing Type	ReRAM Crossbar	PEs connected in a high-speed crossbar	heterogeneous Memristive crossbar	GPU Instruction	**Fixed vector processor and a small reconfigurable switch**
	Cache Optimizations For Frequently-Used Vectors	N/A	✗	N/A	✗	✓
	Reconfigurability	✗	Only for cache hierarchy	✗	N/A	✓
Techniques	**Storage Format**	4×4 COO	CSR	multi-size blocks (64×64, 128×128, 256×256, 512×512)	ELL	**8×8 blocking with fine-grained in-block ordering**
	Resolving Limited Parallelism	N/A	N/A	✗	✓ (Instruction-level, limited by sparsity pattern)	✓

Back to list