# A Relaxed Criterion for Loop Tiling

Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege

UPMC/INRIA/ENS

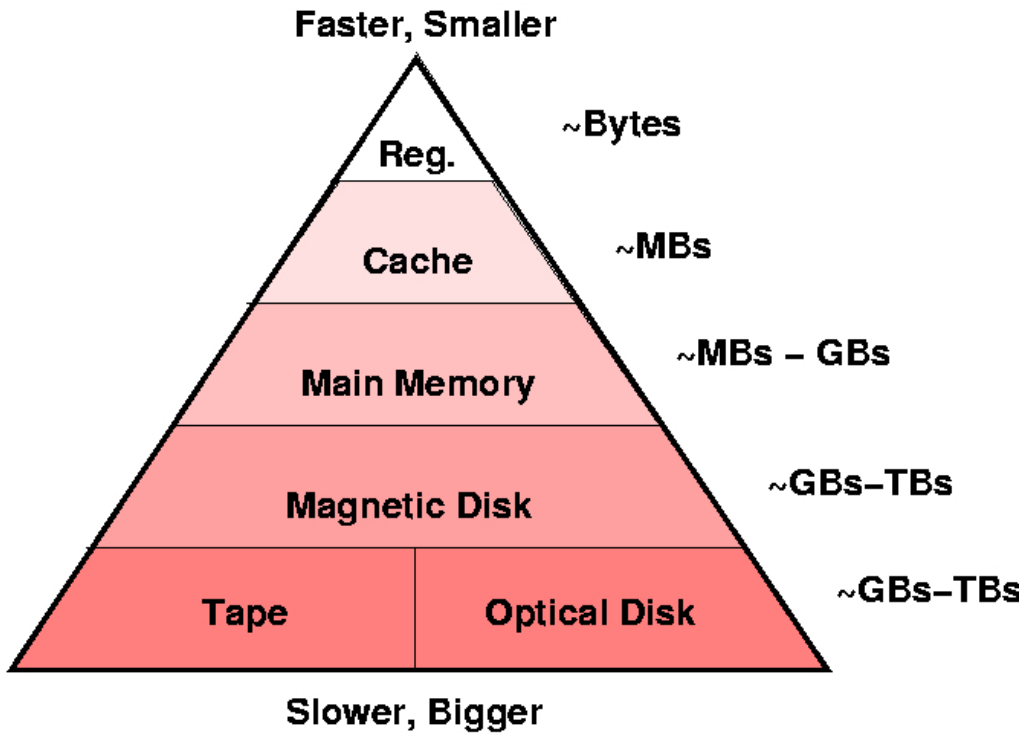
September 22, 2015

# Tiling

- Main benefit: enhance data locality

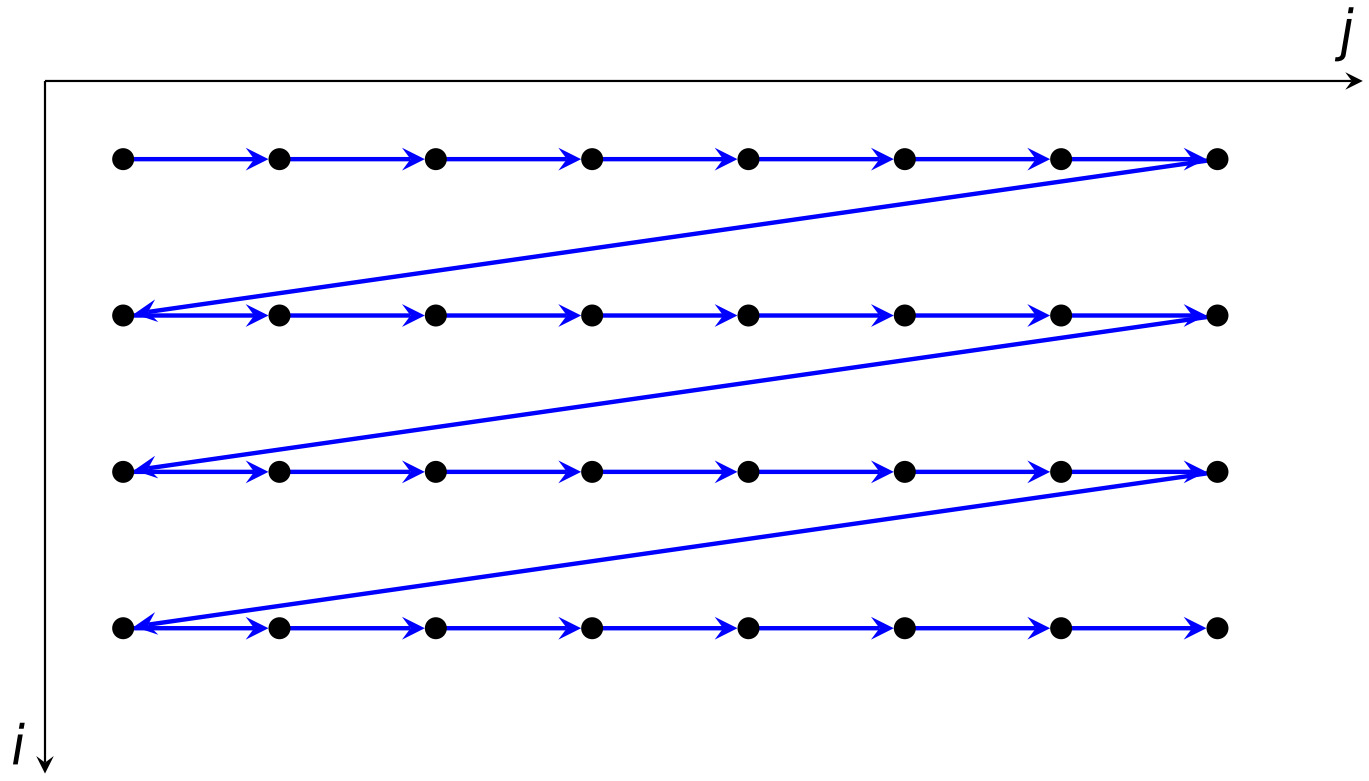
# Tiling

- Main benefit: enhance data locality
- Useful in architectures with a memory hierarchy



# Tiling Example

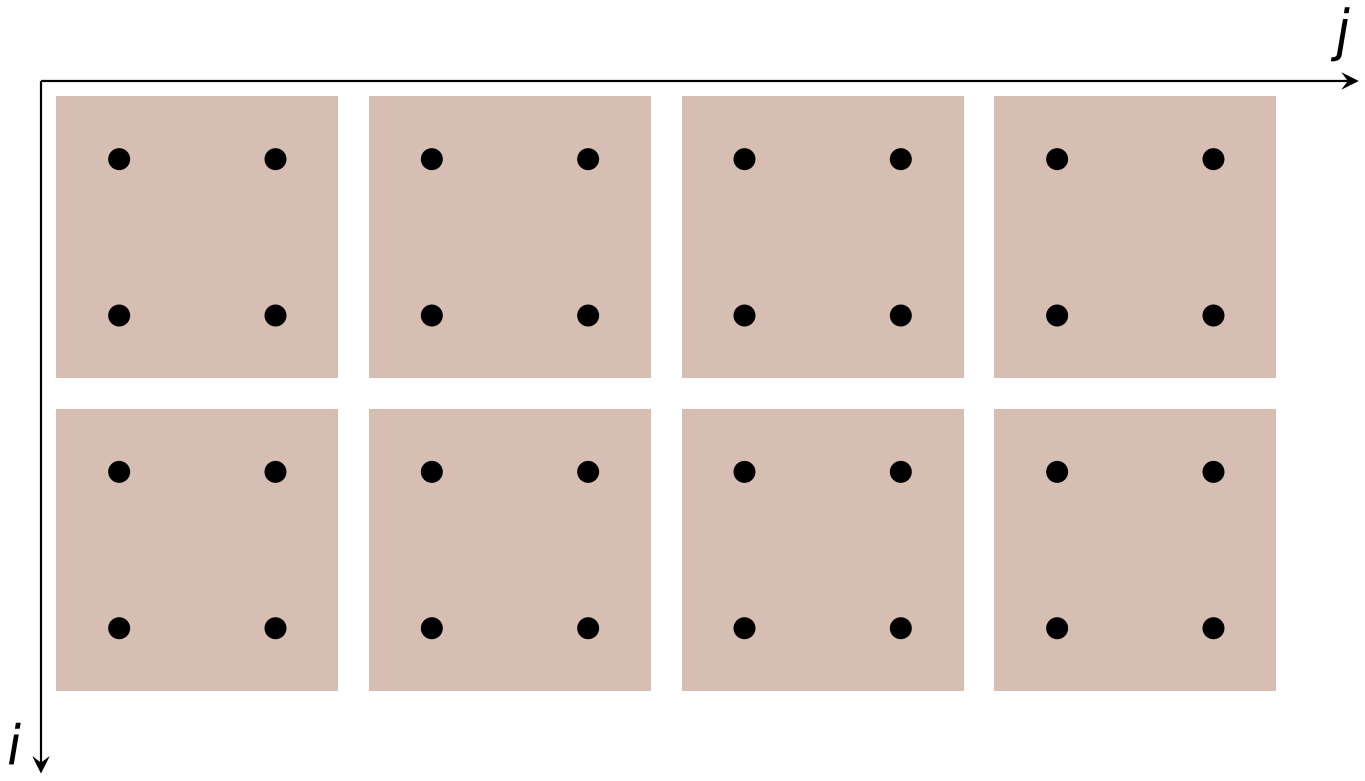
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    C[i][j] = A[j] + B[j];
```




: execution order

# Tiling Example

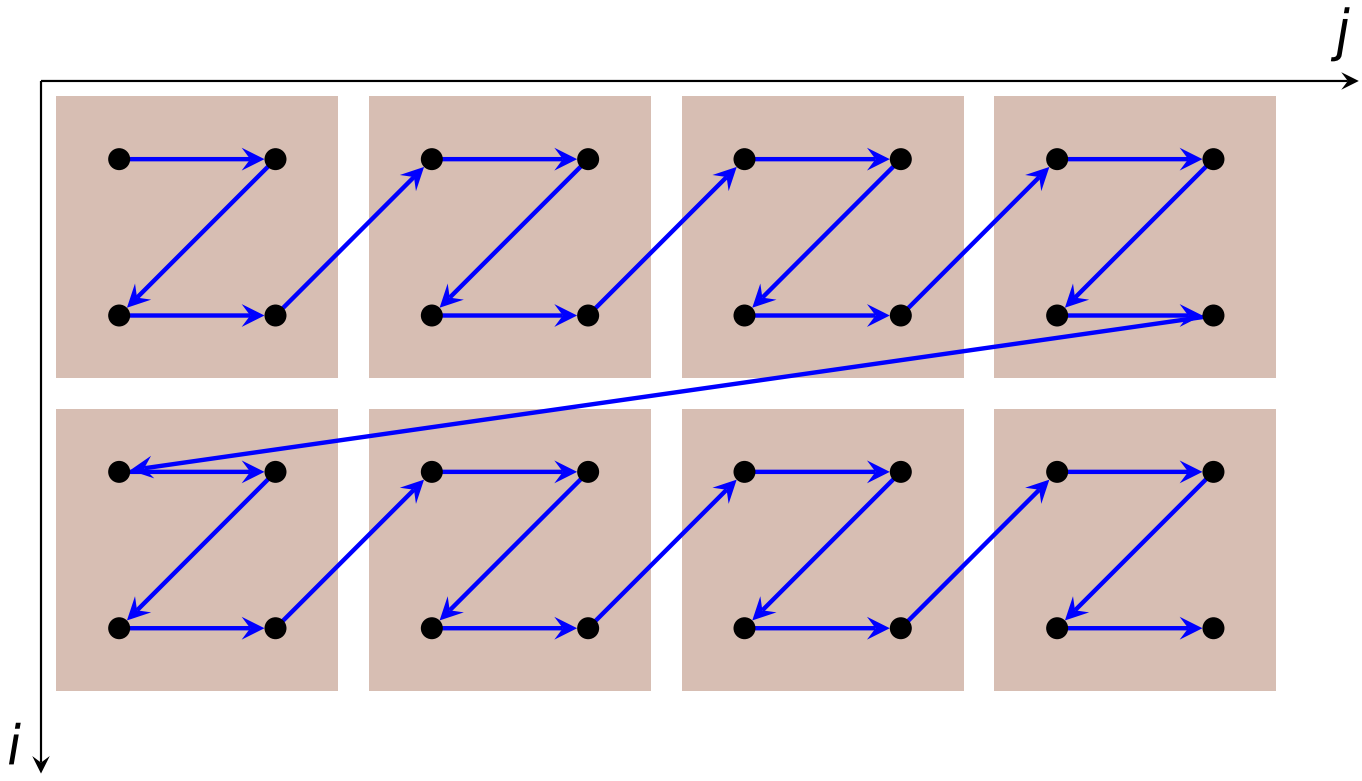
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    C[i][j] = A[j] + B[j];
```




: execution order

# Tiling Example

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    C[i][j] = A[j] + B[j];
```




: execution order

# Permutability Requirements

- To perform tiling we check for permutability

# Permutability Requirements

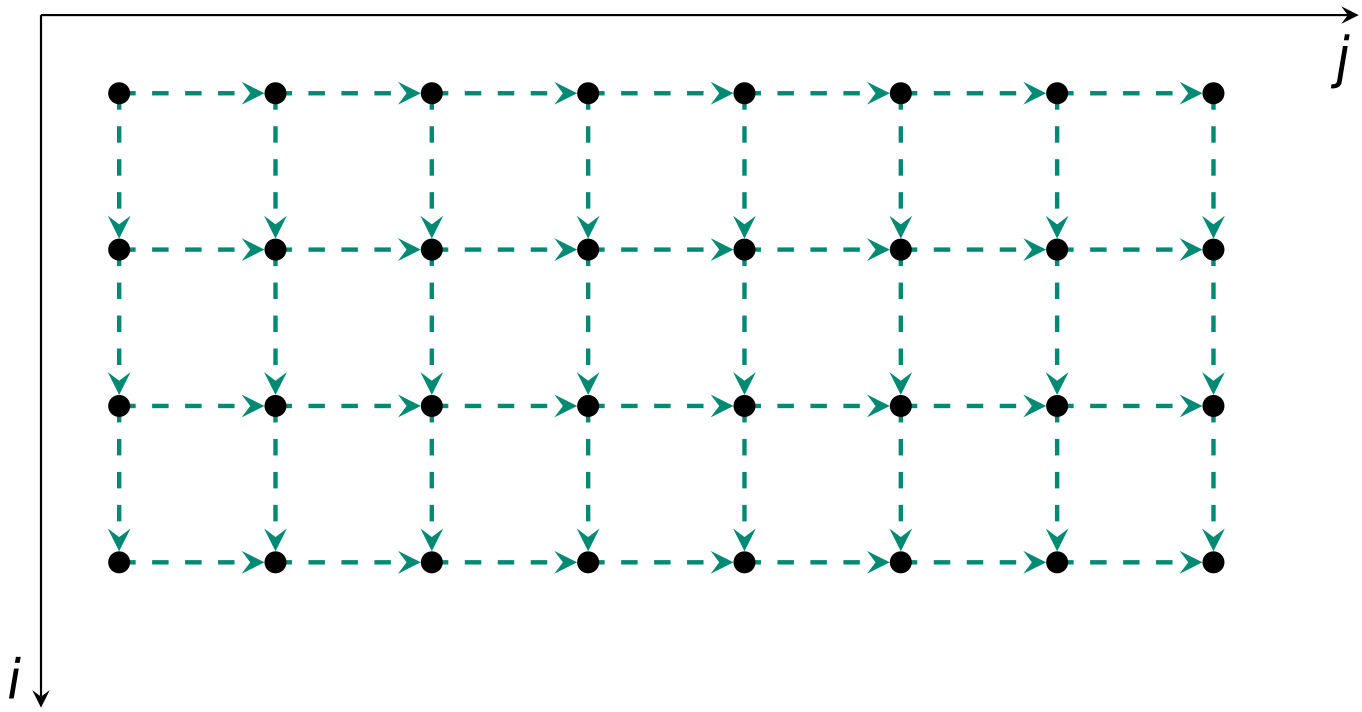
- To perform tiling we check for permutability
- Classical loop permutability criterion
  - Each dependence is forward in each loop of the band

# Permutability Requirements

- To perform tiling we check for permutability
- Classical loop permutability criterion
  - Each dependence is forward in each loop of the band
- A dependence is forward if it is oriented from earlier to later iterations

# Permutability Requirement Examples

A[i ][ j ] = f (A[i −1][j ], A[i ][ j −1]);

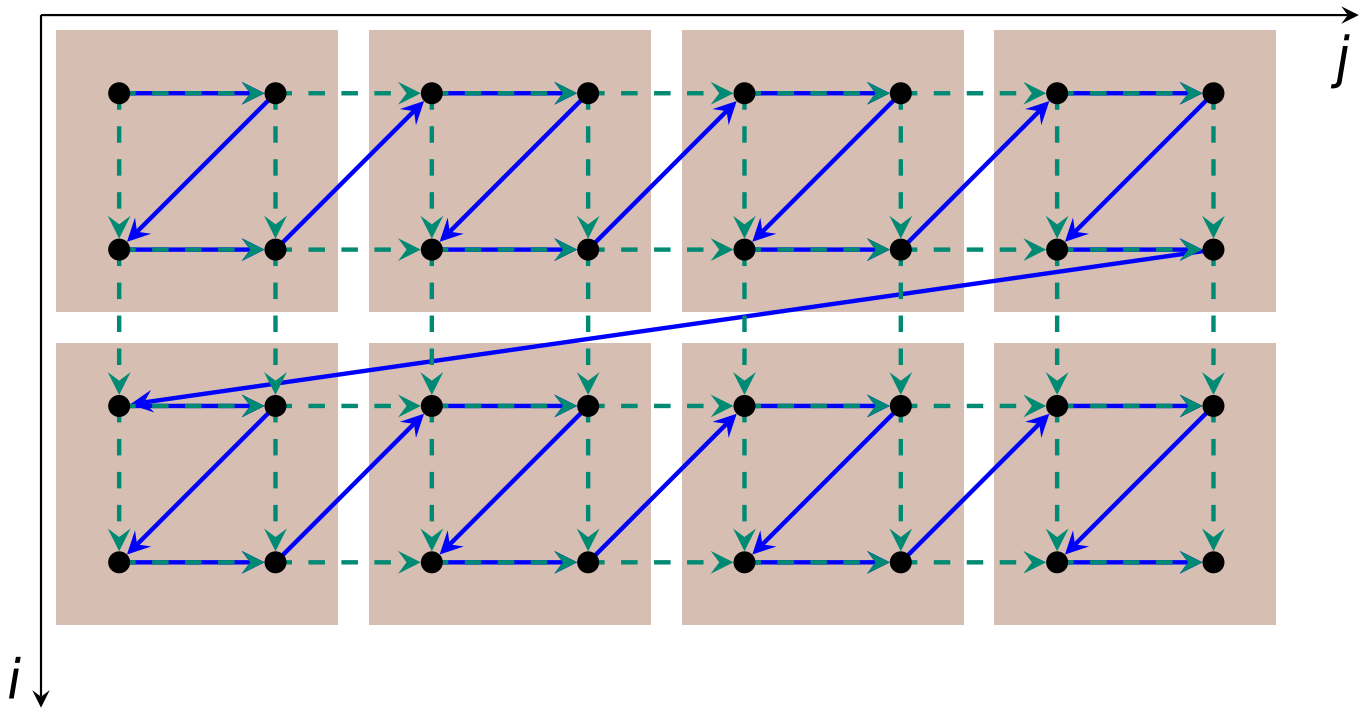



- - - - →: dependence

——→: execution order

# Permutability Requirement Examples

$$A[i][j] = f(A[i-1][j], A[i][j-1]);$$

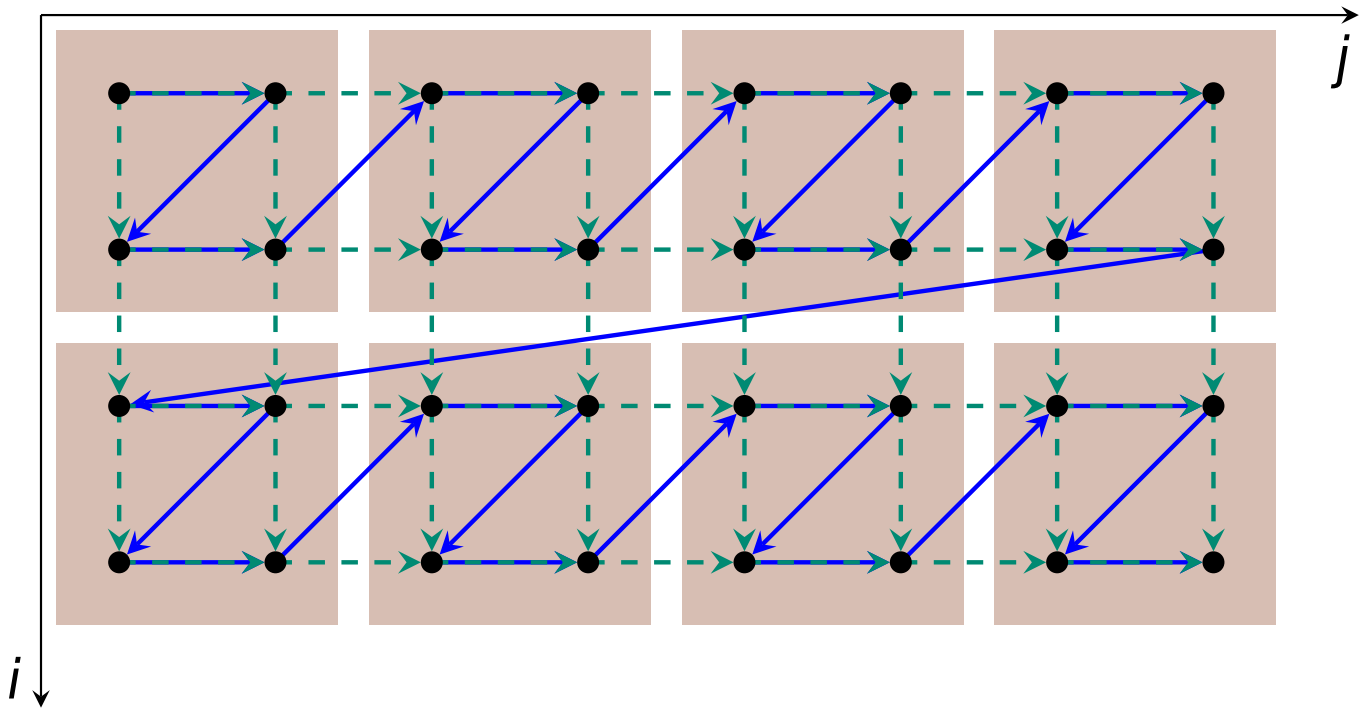



- - - - →: dependence

——→: execution order

# Permutability Requirement Examples

$$A[i][j] = f(A[i-1][j], A[i][j-1]);$$

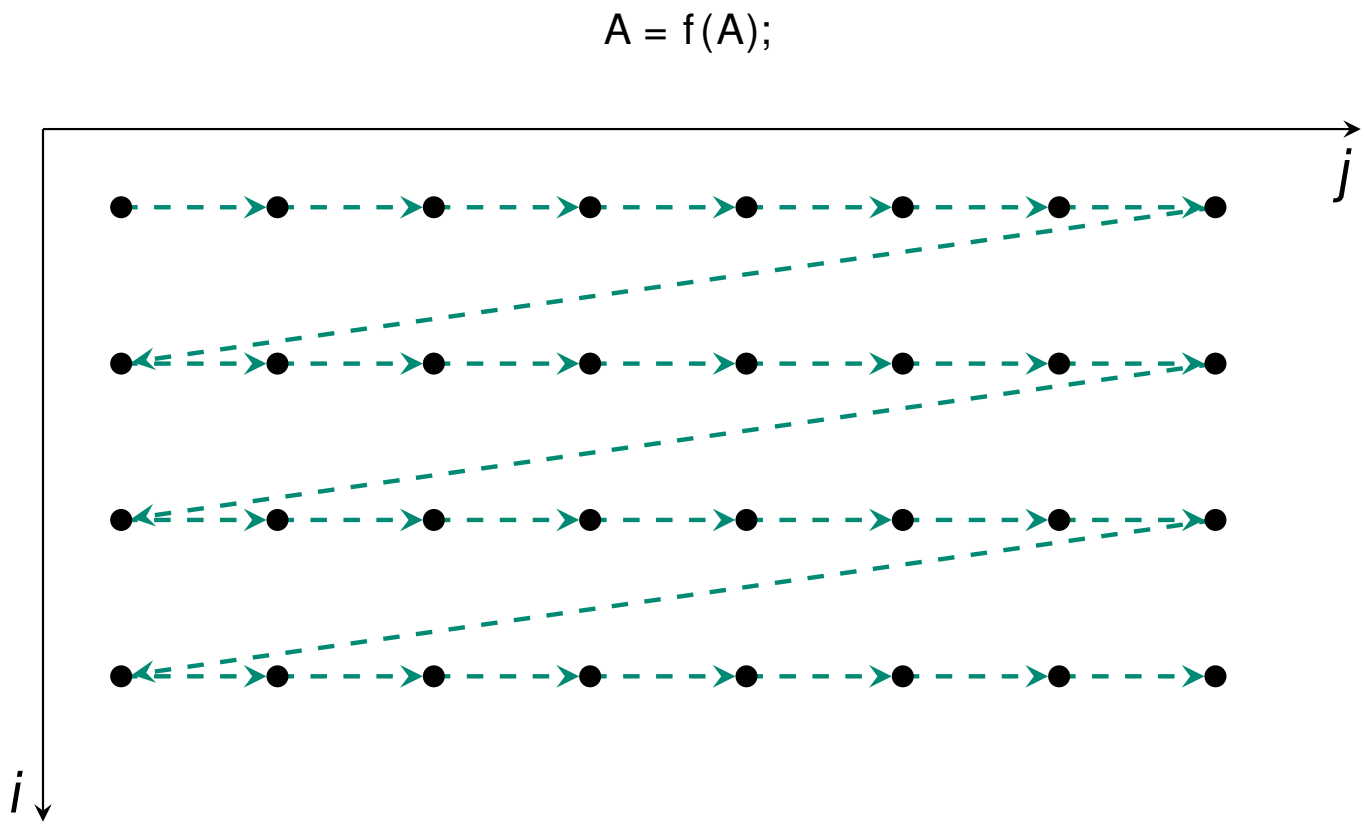



OK

- - - - →: dependence

——→: execution order

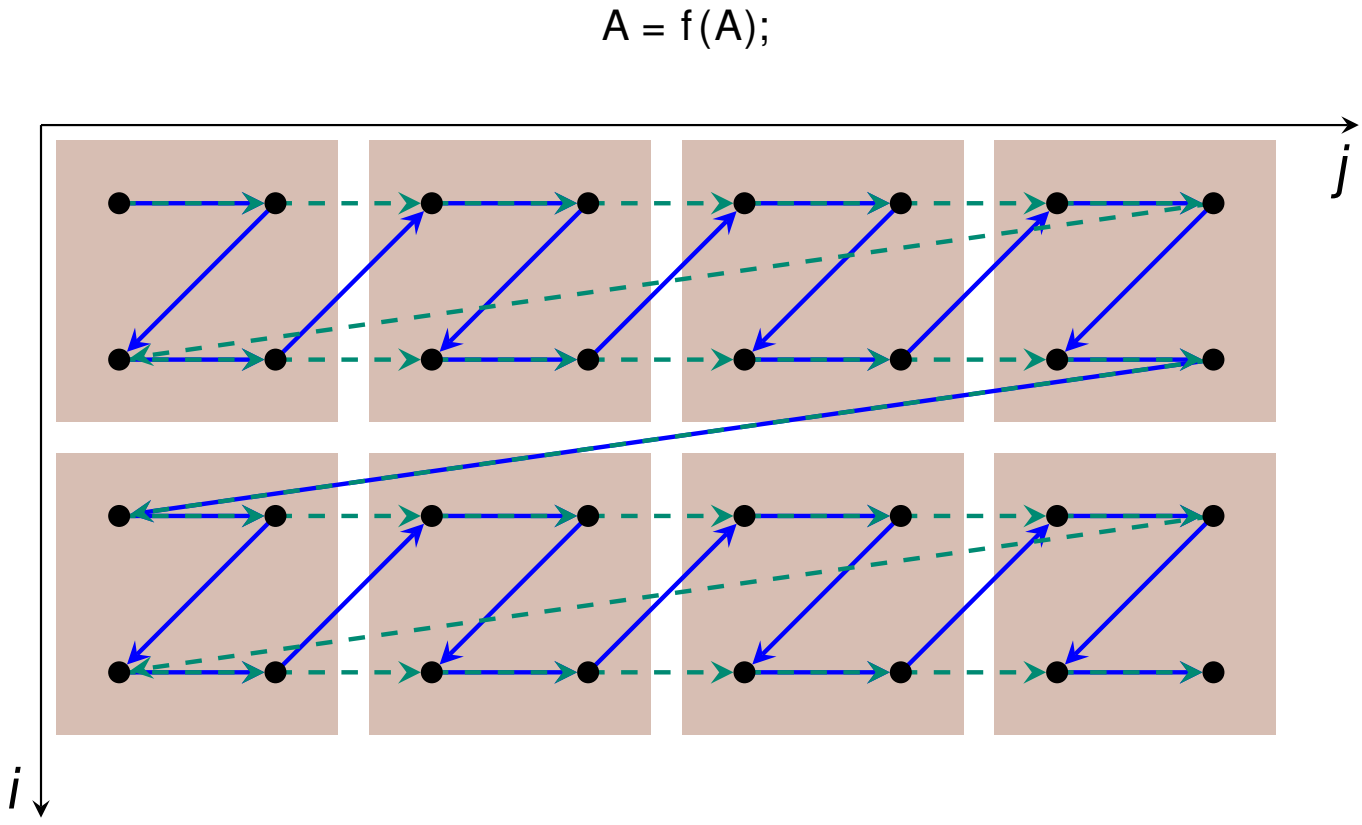
# Permutability Requirement Examples




- - - - →: dependence (only some dependences shown)

——→: execution order

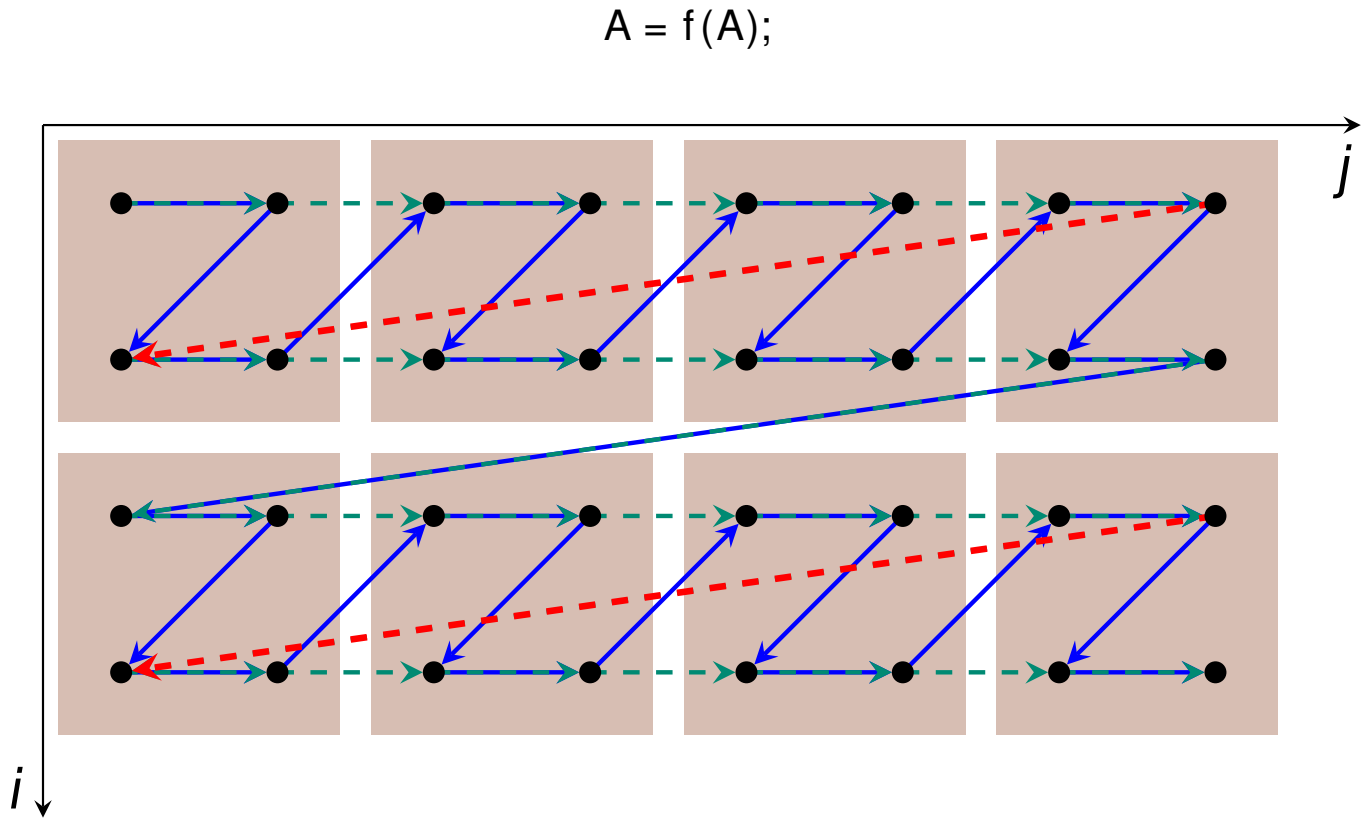
# Permutability Requirement Examples




- - - - →: dependence (only some dependences shown)

——→: execution order

# Permutability Requirement Examples



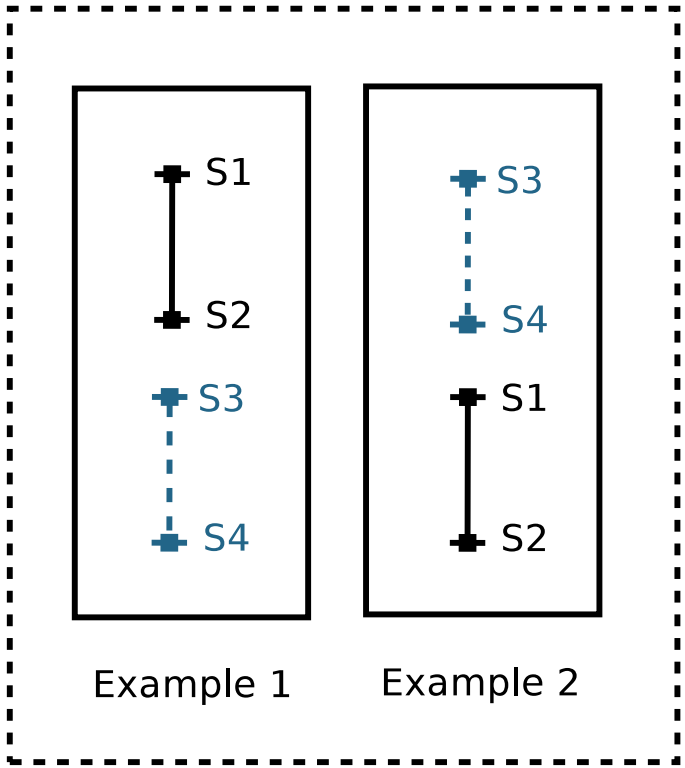

NOT OK

- - - - →: dependence (only some dependences shown)

——→: execution order

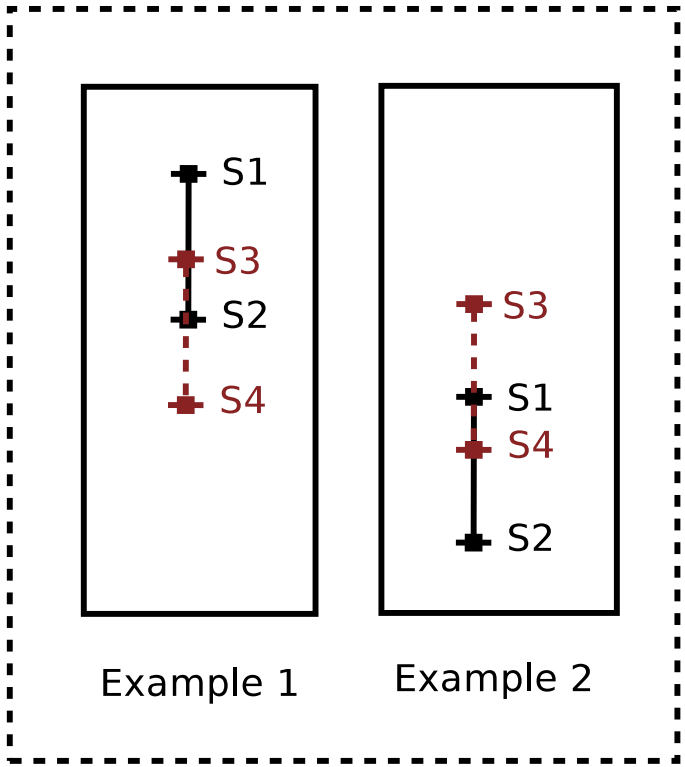
# Loop Transformation Legality

A loop transformation is correct if live ranges do not interfere after the transformation



# Loop Transformation Legality

A loop transformation is correct if live ranges do not interfere after the transformation

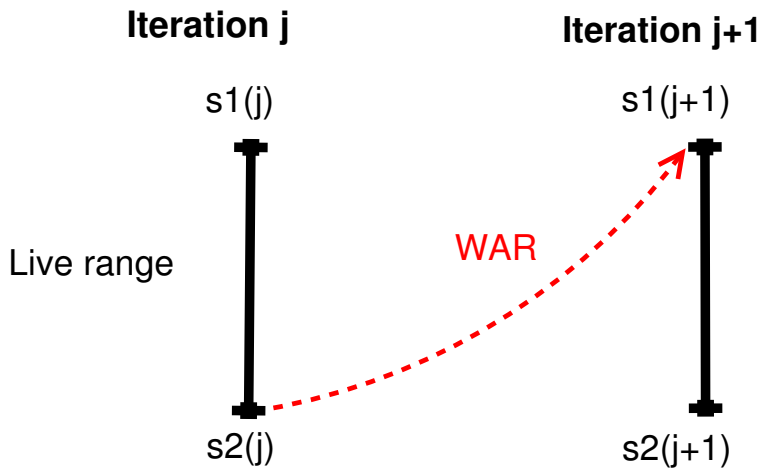


# True and Falce Dependence

- Types of dependences
  - true dependences: write → read
  - false dependences
    - anti dependence: read → write
    - output dependence: write → write

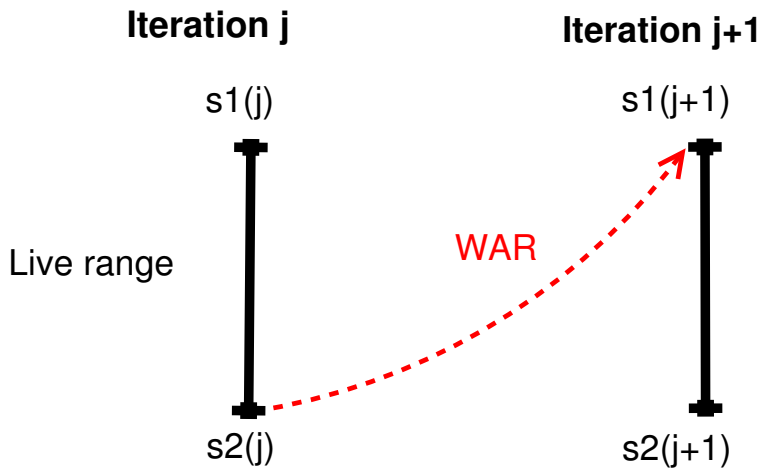
# True and Falce Dependence

- Types of dependences
  - true dependences: write → read
  - false dependences
    - anti dependence: read → write
    - output dependence: write → write
- False dependences
  - are caused by memory reuse
  - prevent live ranges from overlapping



# True and Falce Dependence

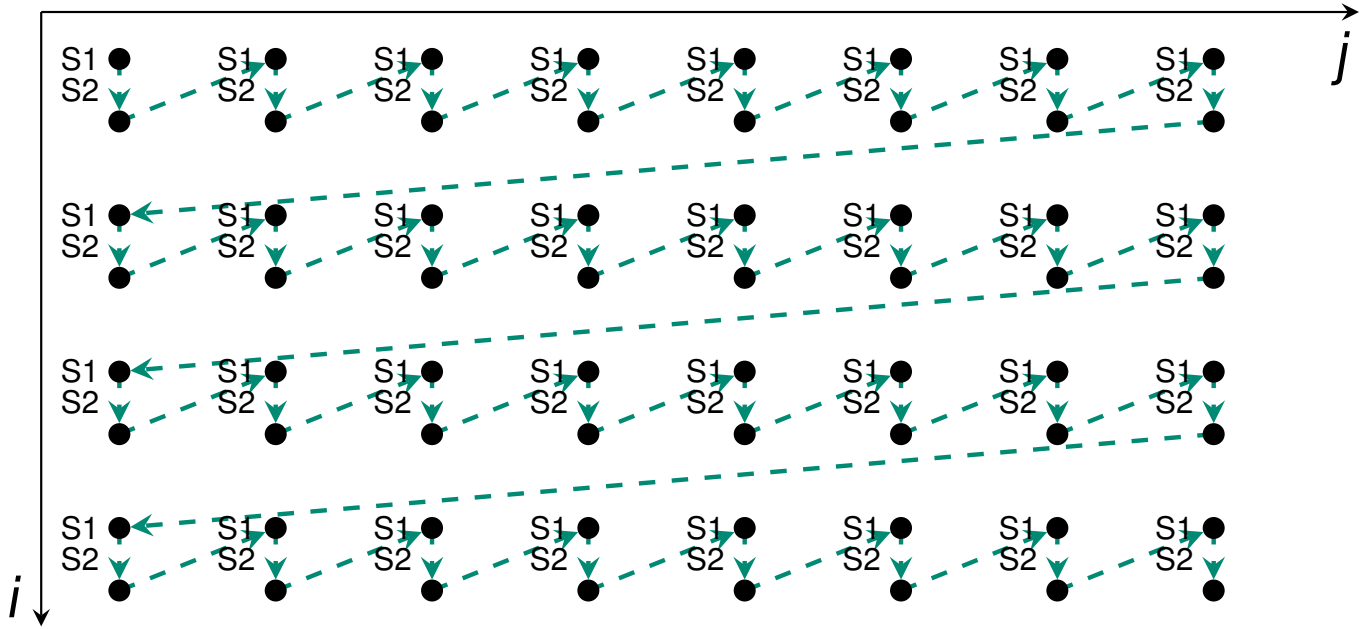
- Types of dependences
  - true dependences: write → read
  - false dependences
    - anti dependence: read → write
    - output dependence: write → write
- False dependences
  - are caused by memory reuse
  - prevent live ranges from overlapping




- Dependences adjacent to live ranges

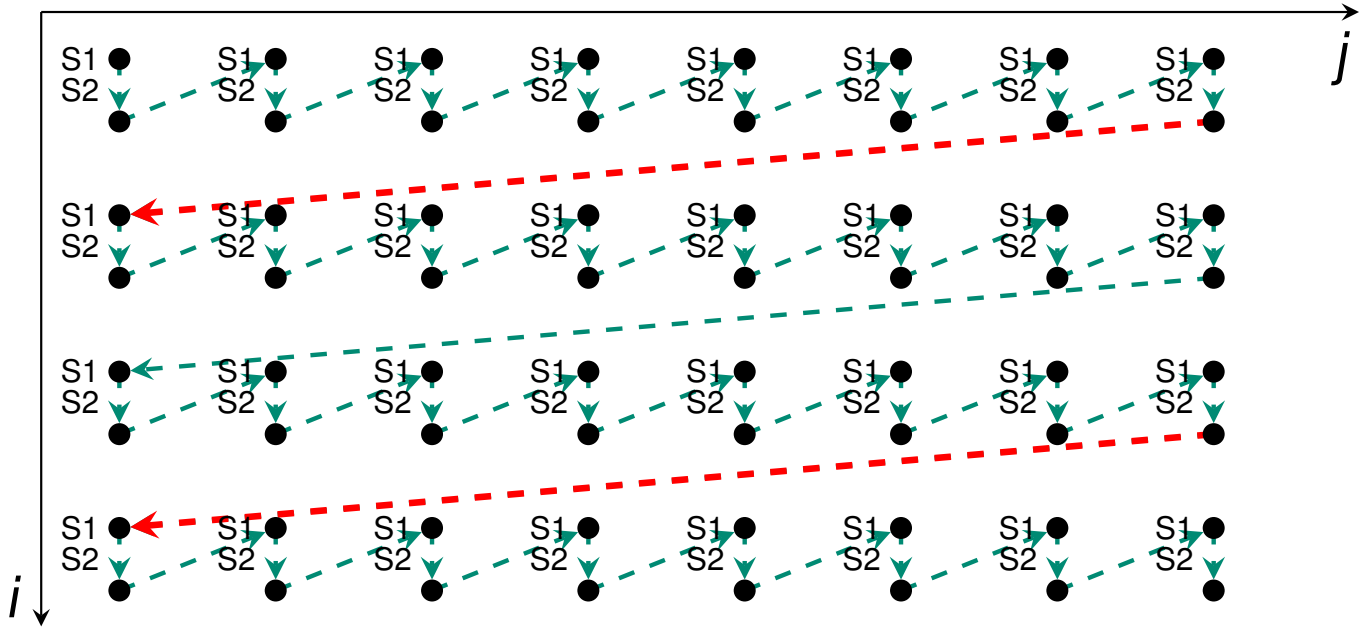
# False Dependences prevent Tiling

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
S1:   t = A[i];
S2:   B[i][j] = t;
    }
```



# False Dependences prevent Tiling

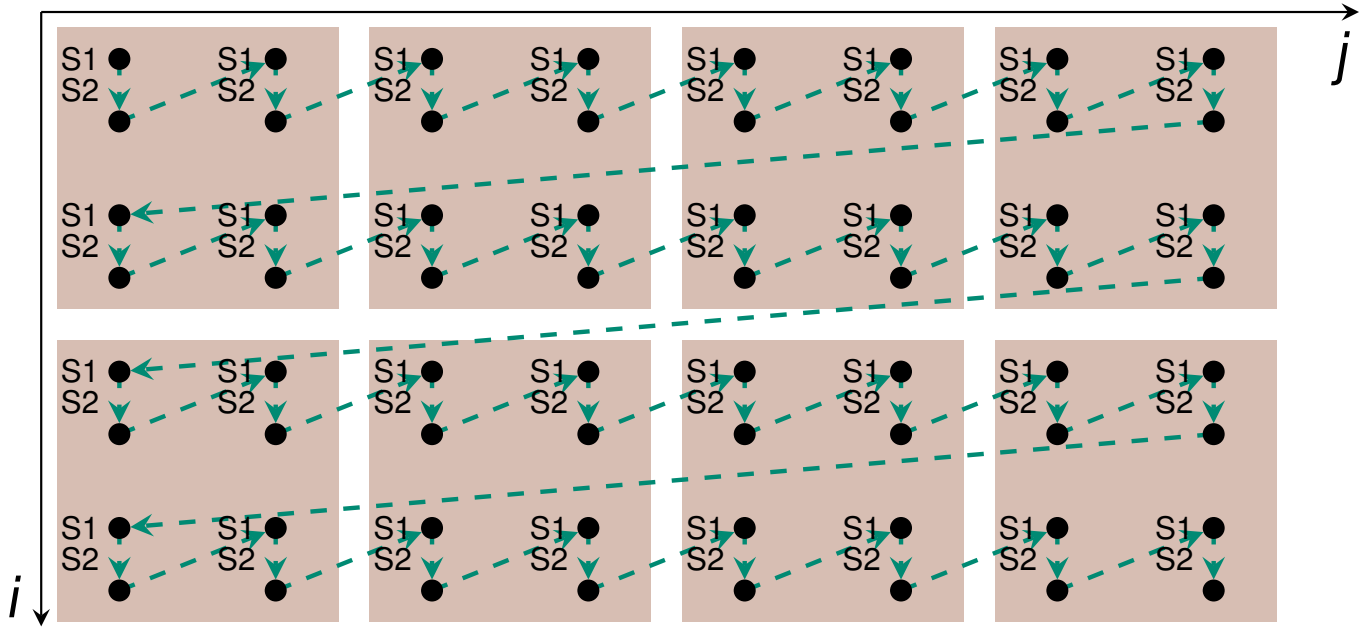
```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
S1:   t = A[i];
S2:   B[i][j] = t;
    }
```




Classical tiling criterion: not allowed

# False Dependences prevent Tiling

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
S1:   t = A[i];
S2:   B[i][j] = t;
    }
```




Classical tiling criterion: not allowed

but tiling is possible

# Relaxed Permutability Criterion

Main idea

- a loop transformation is correct if it does not lead to live range interference

# Relaxed Permutability Criterion

Main idea

- a loop transformation is correct if it does not lead to live range interference
- tiling only changes the order of execution of iterations

# Relaxed Permutability Criterion

Main idea

- a loop transformation is correct if it does not lead to live range interference
- tiling only changes the order of execution of iterations
- if live ranges are local to an iteration then they are guaranteed not to interfer due to tiling

# Relaxed Permutability Criterion

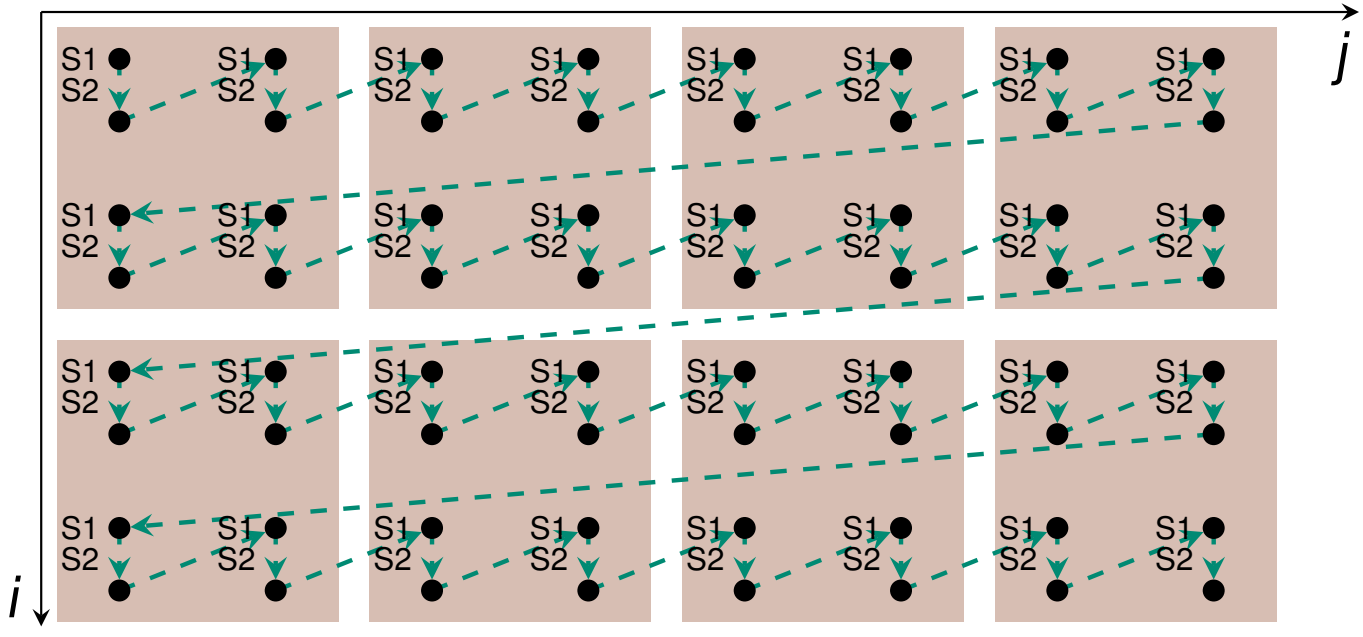
- Classical Permutability Criterion
  - Each dependence is forward in each loop of the band

# Relaxed Permutability Criterion

- Classical Permutability Criterion
  - Each dependence is forward in each loop of the band
- Relaxed Permutability Criterion
  - The same classical criterion except that we ignore anti-dependences that are adjacent to only *local* live ranges

# False Dependences prevent Tiling

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
S1:   t = A[i];
S2:   B[i][j] = t;
    }
```

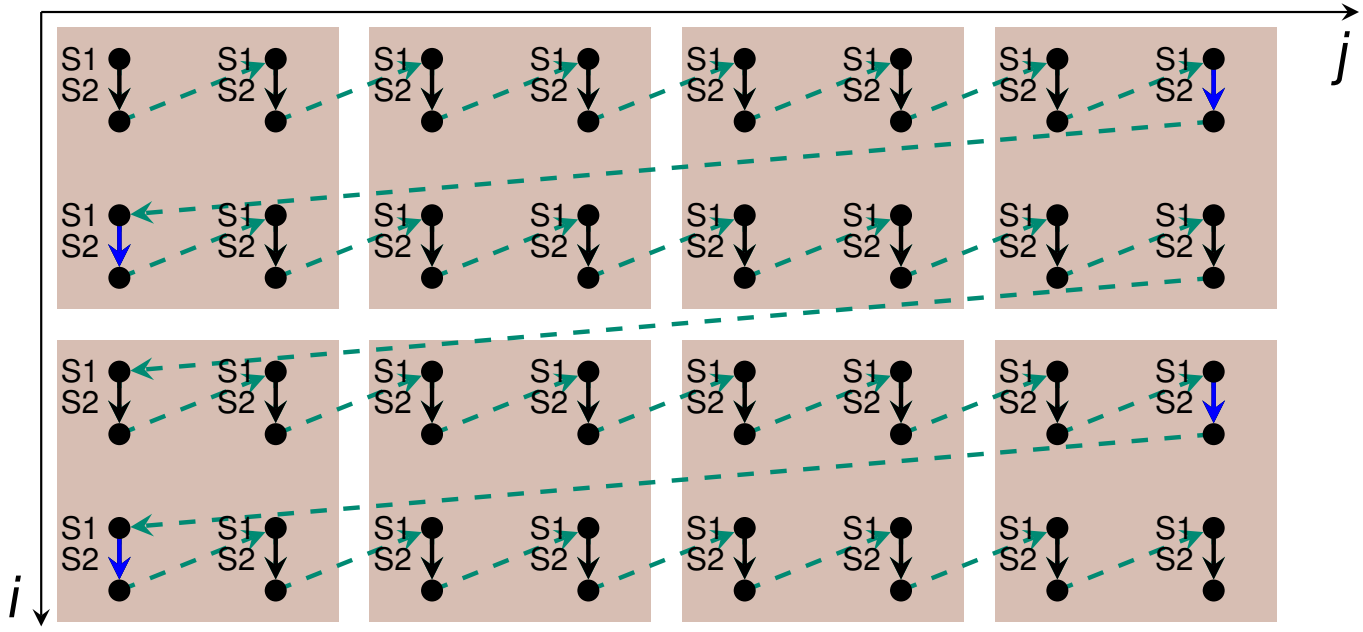



Classical tiling criterion: not allowed

but tiling is possible

# False Dependences prevent Tiling

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
S1:   t = A[i];
S2:   B[i][j] = t;
    }
```

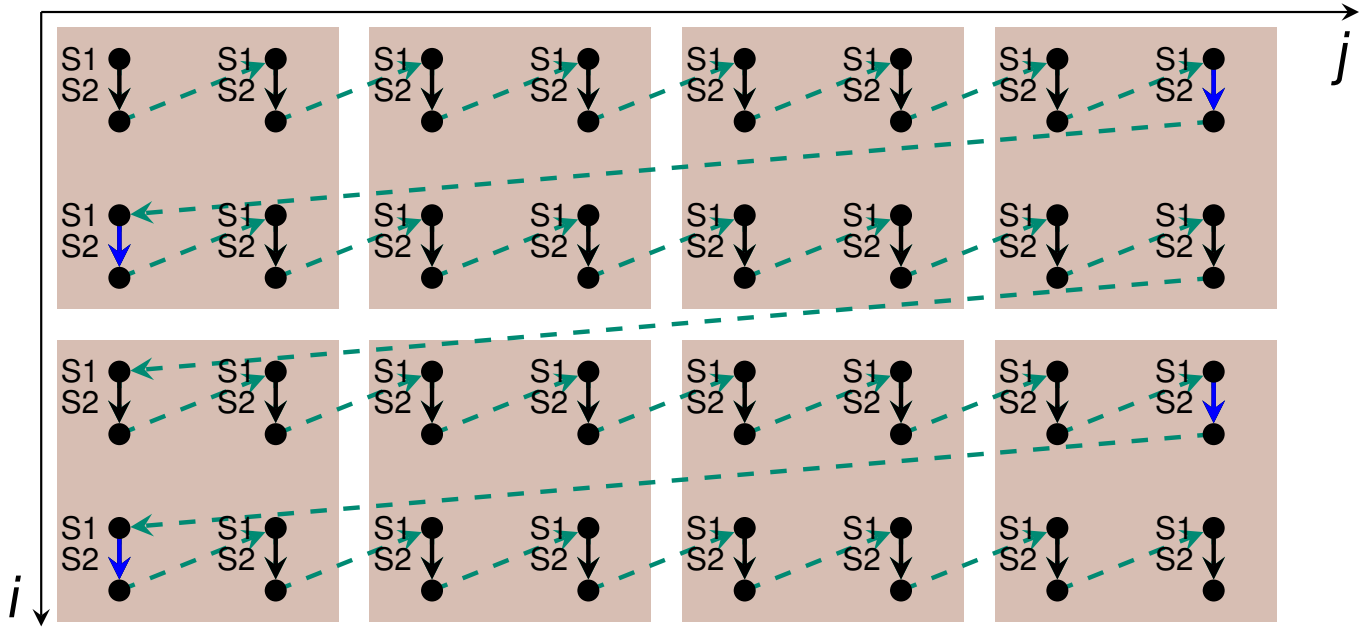



Classical tiling criterion: not allowed

but tiling is possible

# False Dependences prevent Tiling

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
S1:   t = A[i];
S2:   B[i][j] = t;
    }
```




Classical tiling criterion: not allowed

but tiling is possible

Relaxed tiling criterion: allowed

# Tilability of Selected PolyBench Benchmarks

| | Original | 3AC | 3AC + relaxed criterion | Expanded |
|---|---|---|---|---|
| 3mm | yes | no | yes | yes |
| dynprog | yes | no | yes | yes |
| fdtd-2d | yes | no | yes | yes |
| syr2k | yes | no | yes | yes |
| fdtd-apml | yes | no | yes | yes |
| bicg | yes | no | yes | yes |
| symm | no | no | yes | yes |
| cholesky | no | no | yes | yes |

# Conclusion

Relaxed permutability criterion

- Allows tiling in presence of false dependences
- No expansion or privatization required

Future directions

- Combination with on-demand array expansion

# Outline

Relaxed Permutability Criterion

Conclusion

PENCIL

# Outline

Relaxed Permutability Criterion

Conclusion

**PENCIL**

# Motivation

- Programming accelerators: low level APIs (OpenCL, CUDA, ...)

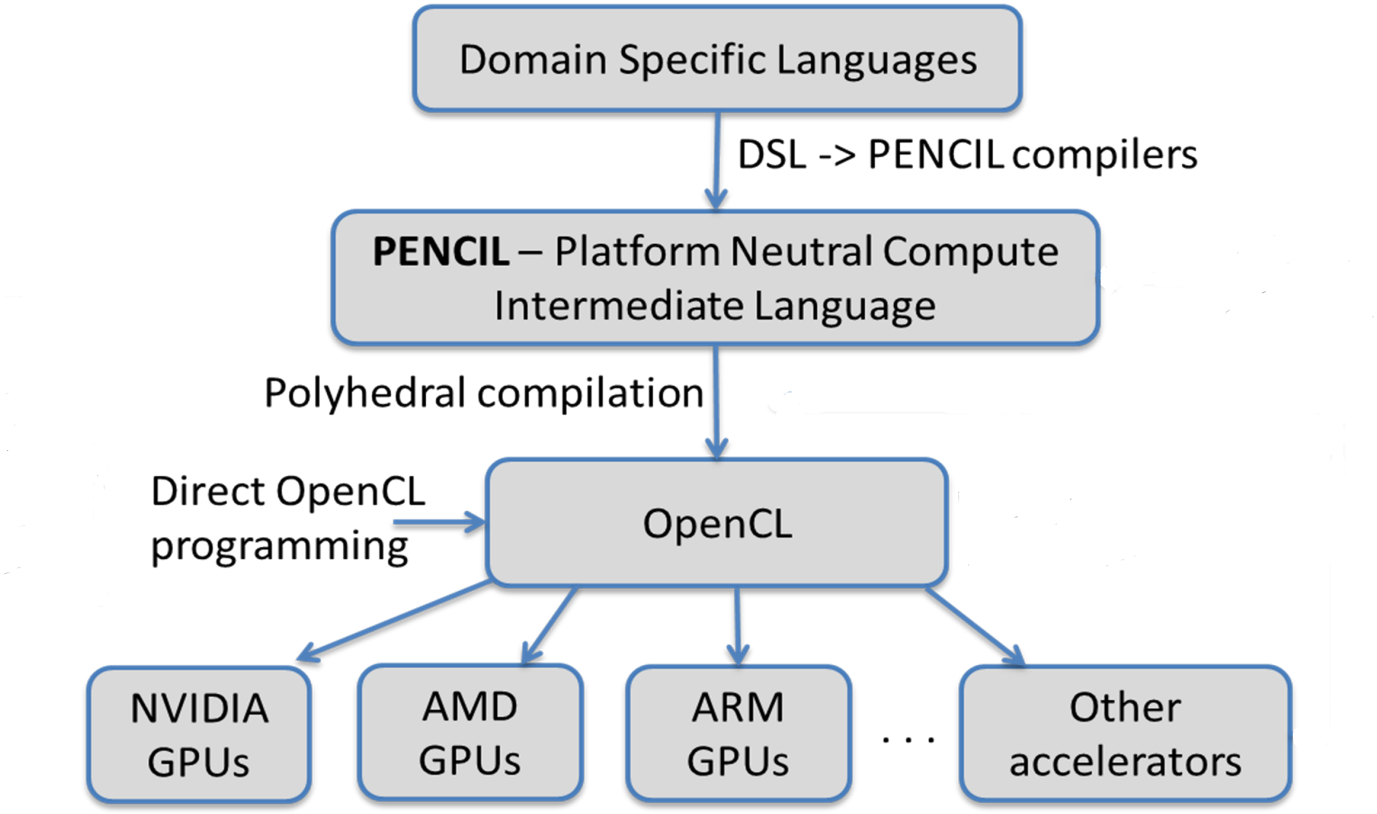
# Motivation

- Programming accelerators: low level APIs (OpenCL, CUDA, ...)
- Problems of low level APIs: difficult to use, non portable code, ...

# Motivation

- Programming accelerators: low level APIs (OpenCL, CUDA, ...)
- Problems of low level APIs: difficult to use, non portable code, ...
- Solution
  - Write code in a high level language
  - Use a compiler for parallelization/optimization

# PENCIL


Domain Specific Languages
DSL -> PENCIL compilers
PENCIL – Platform Neutral Compute Intermediate Language
Polyhedral compilation
Direct OpenCL programming
OpenCL
NVIDIA GPUs
AMD GPUs
ARM GPUs
. . .
Other accelerators

# PENCIL Intermediate Language

Subset of C99

- restrictions on pointer use
  - goals: no write aliasing, constant array references, ...

# PENCIL Intermediate Language

Subset of C99

- restrictions on pointer use
  - goals: no write aliasing, constant array references, ...
  - restrictions:
    - C99 VLA syntax for array declaration (`int A[m]` instead of `int *A`)
    - cannot read or write to a pointer except passing an array reference to a function: `foo(A);`

# PENCIL Intermediate Language

Subset of C99

- restrictions on pointer use
  - goals: no write aliasing, constant array references, ...
  - restrictions:
    - C99 VLA syntax for array declaration (`int A[m]` instead of `int *A`)
    - cannot read or write to a pointer except passing an array reference to a function: `foo(A);`
- no gotos

Extensions (builtins and directives)

- `__pencil_assume(expression)`
- `__pencil_kill(T)`
- `__pencil_reduce(...)`
- `#pragma pencil independent`
- Summary functions

# Compiling PENCIL

- We use the PPCG polyhedral compiler
- Polyhedral moldel: an algebraic representation of programs (focus on loop nests).
- Static-affine control
  - Static control: not data dependent (`if (A[i])`).
  - loop bounds, conditionals and array subscripts should be affine with respect to the loop iterators and a set of symbolic constants. Affine: $i + j \geq 0$. Non-affine: $i * i \geq 0$

# Compiling PENCIL

- We use the PPCG polyhedral compiler
- Polyhedral moldel: an algebraic representation of programs (focus on loop nests).
- Static-affine control
  - Static control: not data dependent (`if (A[i])`).
  - loop bounds, conditionals and array subscripts should be affine with respect to the loop iterators and a set of symbolic constants. Affine: $i + j \geq 0$. Non-affine: $i * i \geq 0$

# Compiling PENCIL Extensions

- Non static-affine array accesses (read/write)
  - Treated as a *may* access to the whole array dimension
  - Example:

    ```
    A[i] = B[foo(i)];
    ```

    is treated as

    ```
    A[i] = B[*];
    ```

# Compiling PENCIL Extensions

- Non static-affine array accesses (read/write)
  - Treated as a *may* access to the whole array dimension
  - Example:

    ```
    A[i] = B[foo(i)];
    ```

    is treated as

    ```
    A[i] = B[*];
    ```

- Non static-affine conditionals

```
if (A[i])
{
. .  B[i] = 0;
. .  C[i] = 0;
}
```

# Compiling PENCIL Extensions

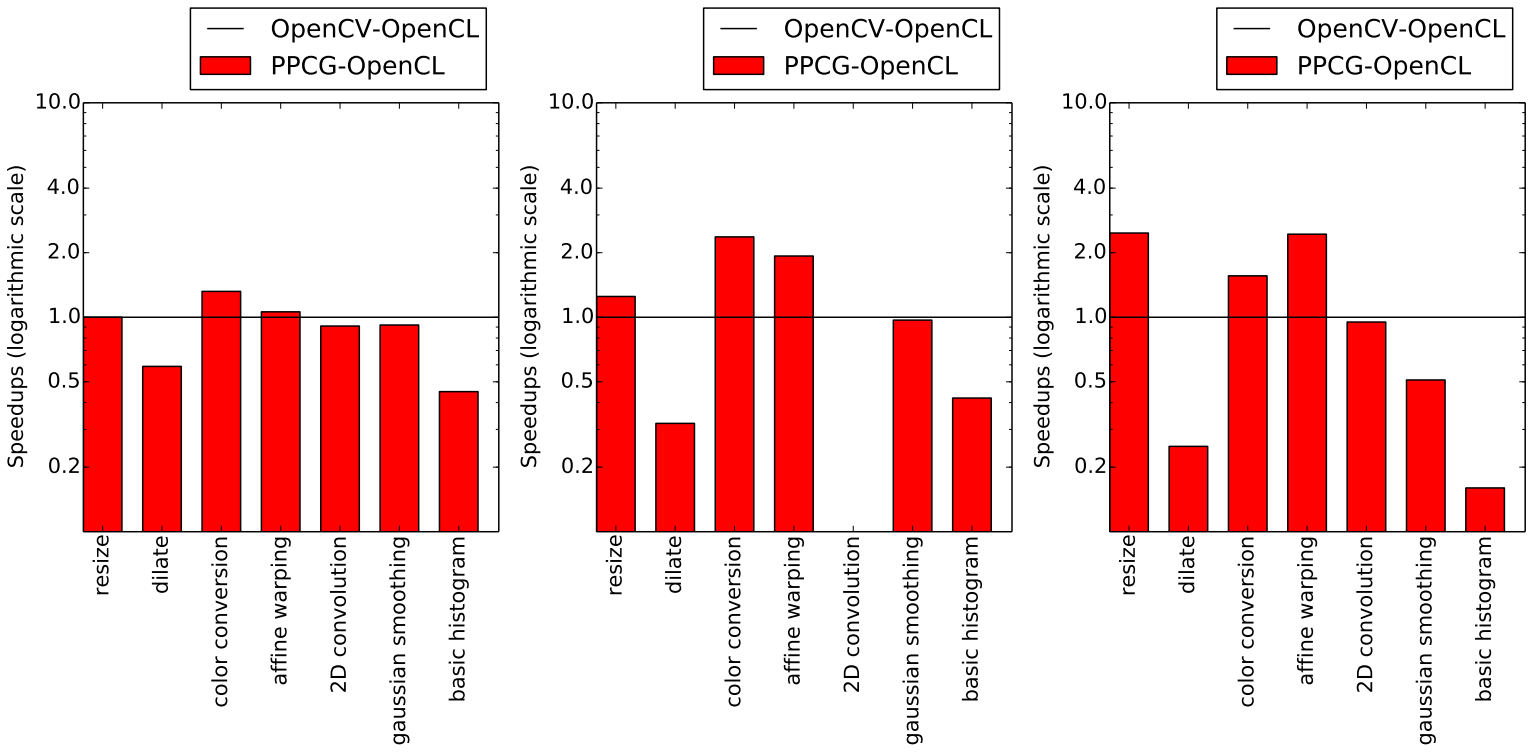
- **__pencil_assume**(expression)
  - expression is an affine constraint on loop parameters
  - expression is added to the context (set of affine constraints on loop parameters)
  - This information (context) is used whenever needed

# Compiling PENCIL Extensions

- **__pencil_assume**(expression)
  - expression is an affine constraint on loop parameters
  - expression is added to the context (set of affine constraints on loop parameters)
  - This information (context) is used whenever needed
- **independent** directive
  - Used currently during parallelism detection (remove loop carried dependences)

# Image Processing: Experiments

Speedup of code generated from PENCIL over OpenCV library




Nvidia Fermi ARM Mali AMD Radeon