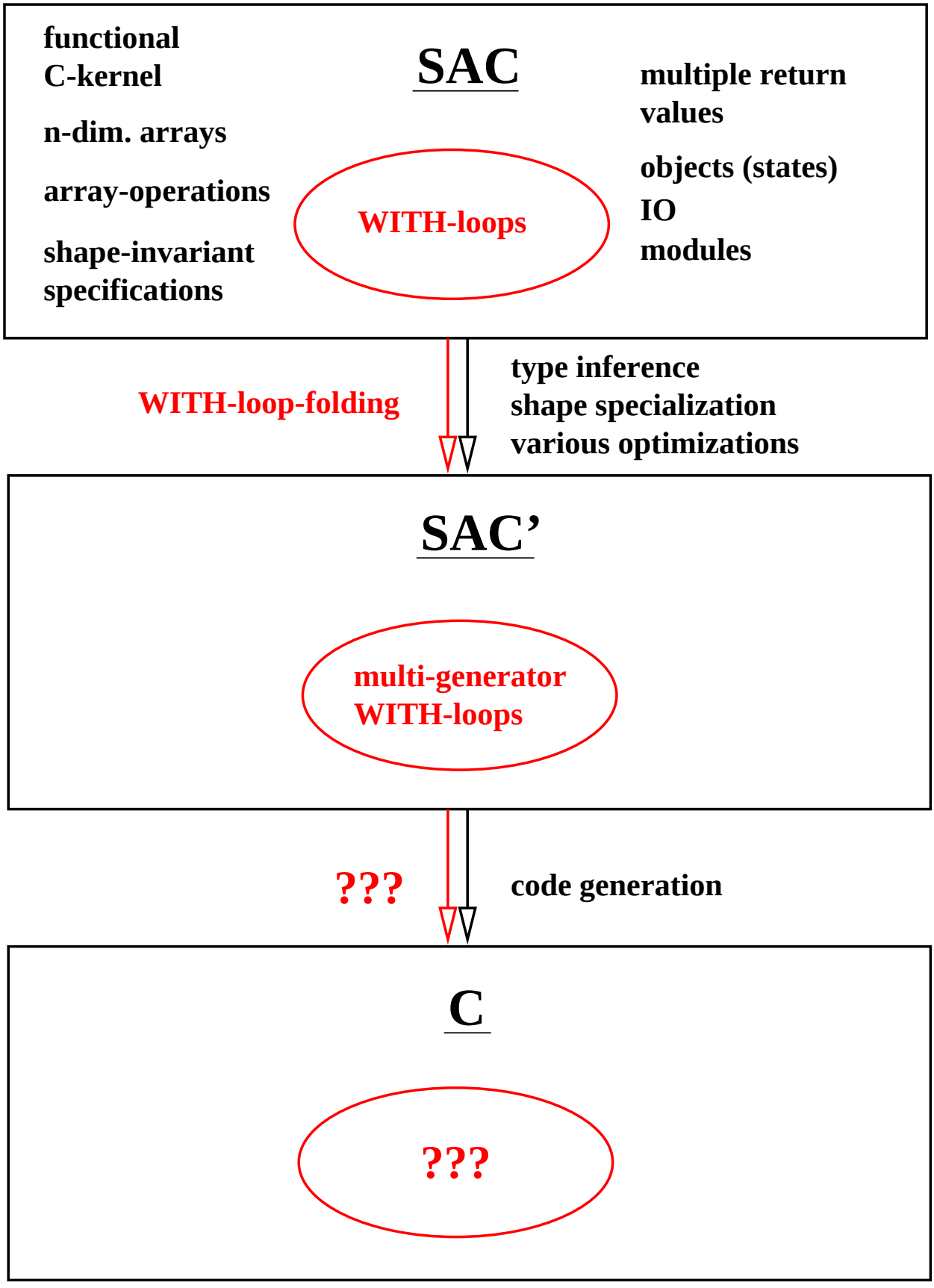
# On Code Generation for Multi-Generator `With`-Loops in SAC

Dietmar Kreye
University of Kiel, Germany

---

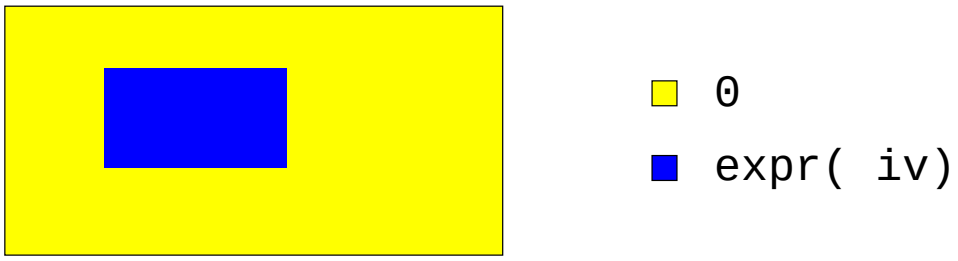
11th International Workshop on the
Implementation of Functional Languages

# Compilation of SAC

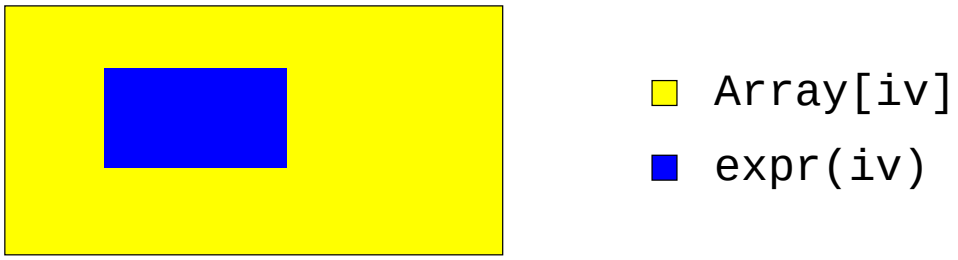


# The With-Loop Construct

1) res = with (iv ∈ IndexVectorSet)
genarray( Shape, expr(iv));




2) res = with (iv ∈ IndexVectorSet)
modarray( Array, iv, expr(iv));




3) res = with (iv ∈ IndexVectorSet)
fold( F, Neutr, expr(iv));

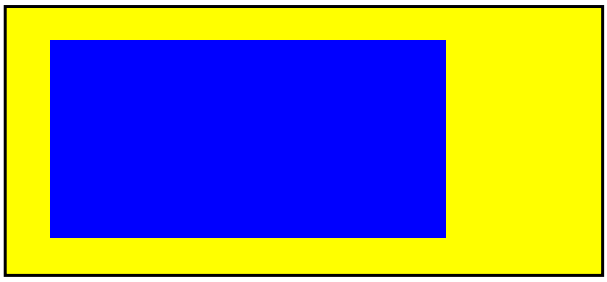
Neutr F expr(iv) F ... F expr(iv)

IndexVectorSet

# The With-Loop Generators

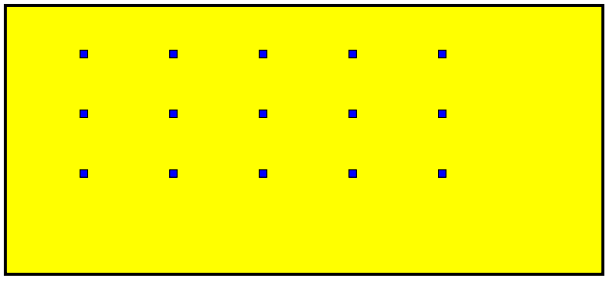
## Rectangular Subsets:

```
with (lbv <= iv < ubv)
```



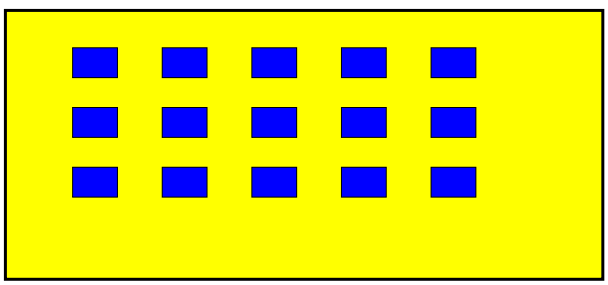
## Rectangular Grids:

```
with (lbv <= iv < ubv step sv)
```



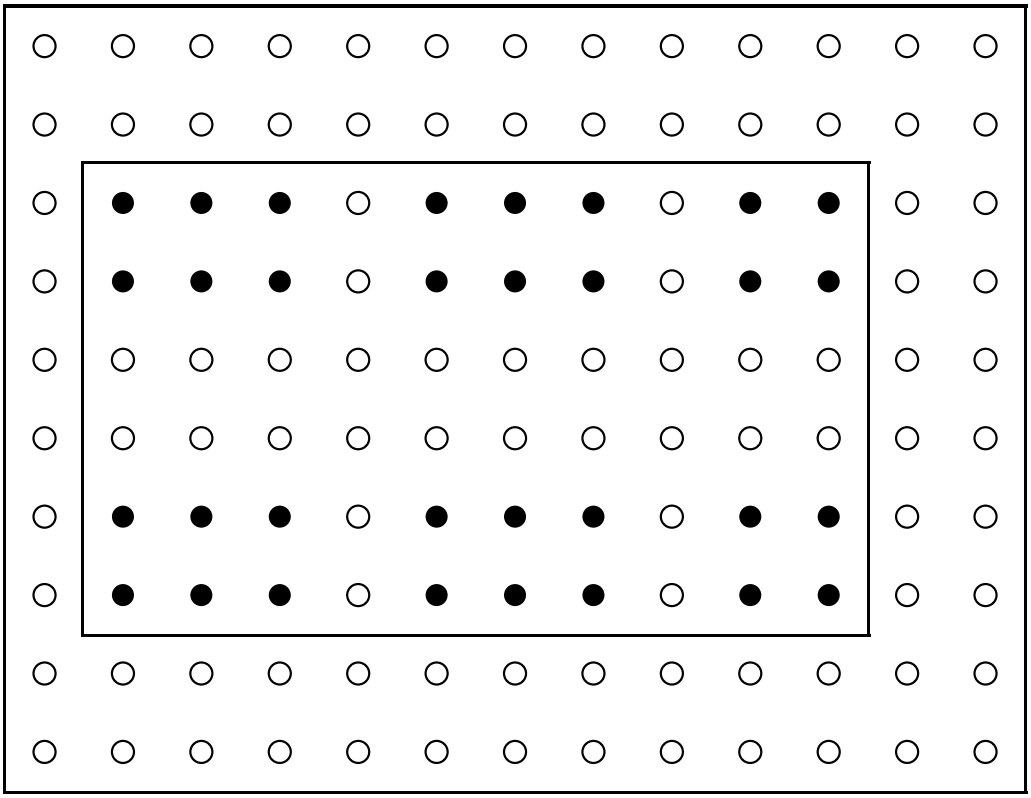
## Rectangular Wide Grids:

```
with (lbv <= iv < ubv step sv width wv)
```



# With-Loop Example

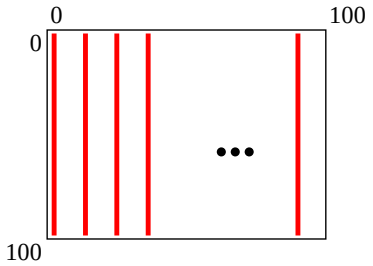
```
A = with ([2,1] <= iv < [8,11]
          step [4,4] width [2,3])
    genarray( [10,13], expr(iv));
```



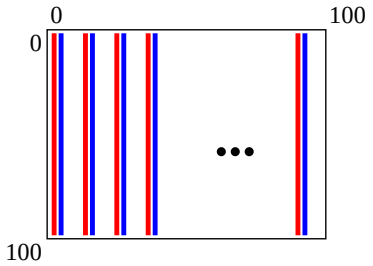
$$\texttt{A[iv]} := \begin{cases} [\![expr(\texttt{iv})]\!] & : \quad \texttt{iv} \in \bullet \\ 0 & : \quad \texttt{iv} \in \circ \end{cases}$$

# Consecutive Usage of `With`-Loops

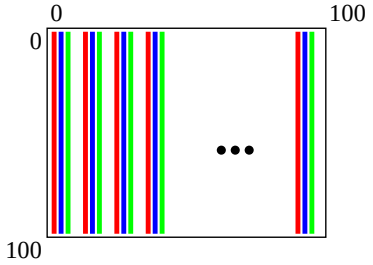
```
A = with ([0,0] <= iv < [100,100] step [1,4])
    genarray( [100,100], op1(iv));
```



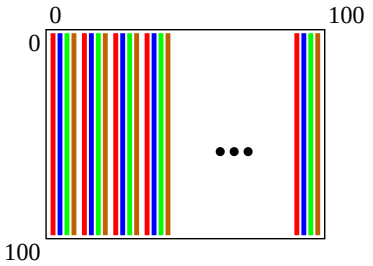

```
B = with ([0,1] <= iv < [100,100] step [1,4])
    modarray( A, iv, op2(iv));
```




```
C = with ([0,2] <= iv < [100,100] step [1,4])
    modarray( B, iv, op3(iv));
```




```
D = with ([0,3] <= iv < [100,100] step [1,4])
    modarray( C, iv, op4(iv));
```



# Avoiding Temporaries through With-Loop Folding

## Basic Idea:

Condense subsequent operations on arrays into a single one:

$$\texttt{(map f)} \circ \texttt{(map g)} \quad \Longleftrightarrow \quad \texttt{map (f} \circ \texttt{g)}$$

## Example:

```
scal_prod = sum( A*B);
```

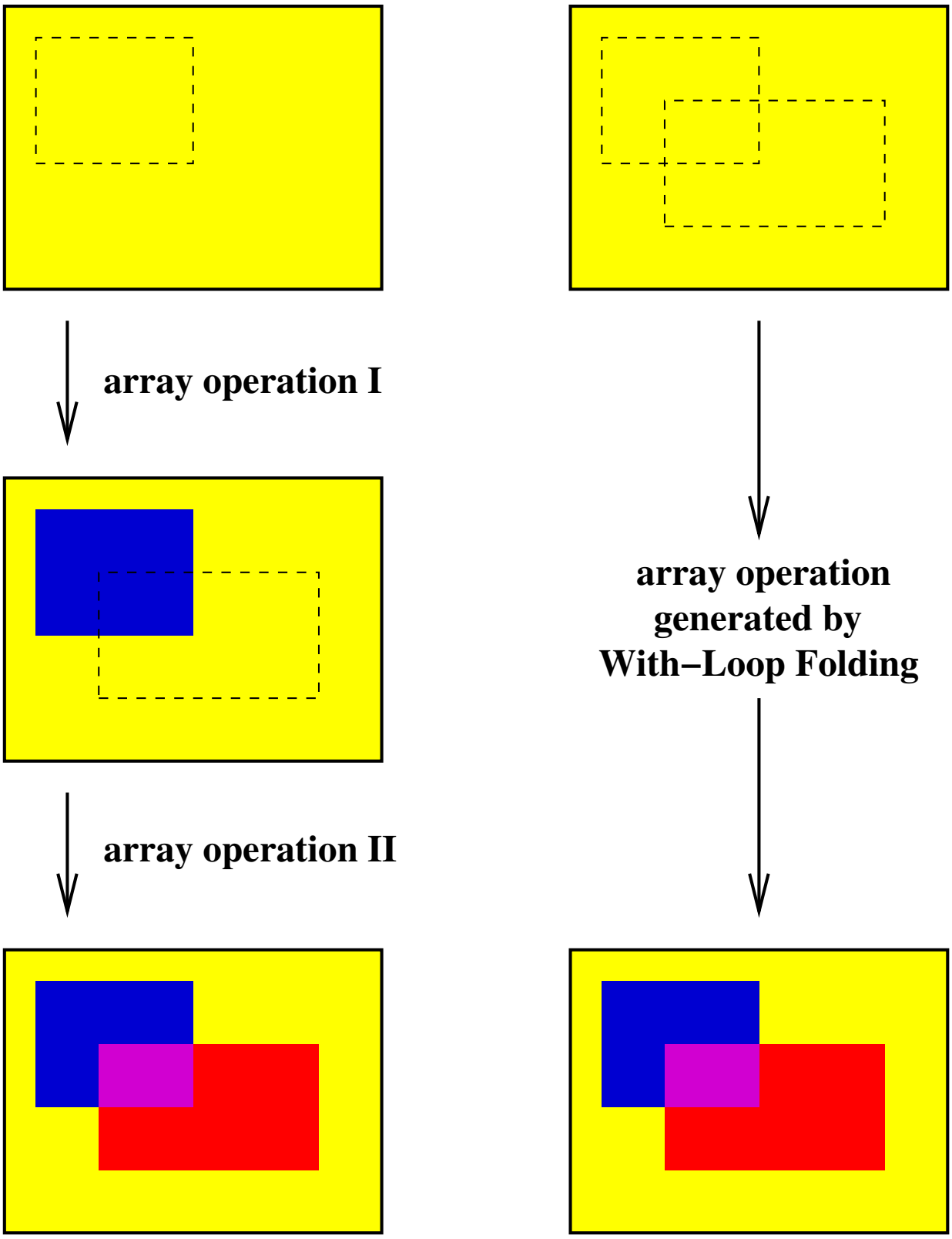
$\Downarrow$

```
tmp_array = with (0*shape(A) <= iv < shape(A))
            genarray( shape(A), A[iv] * B[iv]);

scal_prod = with (0*shape(A) <= iv < shape(A))
            fold( +, 0, tmp_array[iv]);
```

$\Downarrow$

```
scal_prod = with (0*shape(A) <= iv < shape(A))
            fold( +, 0, A[iv] * B[iv]);
```

# With-Loop Folding in General

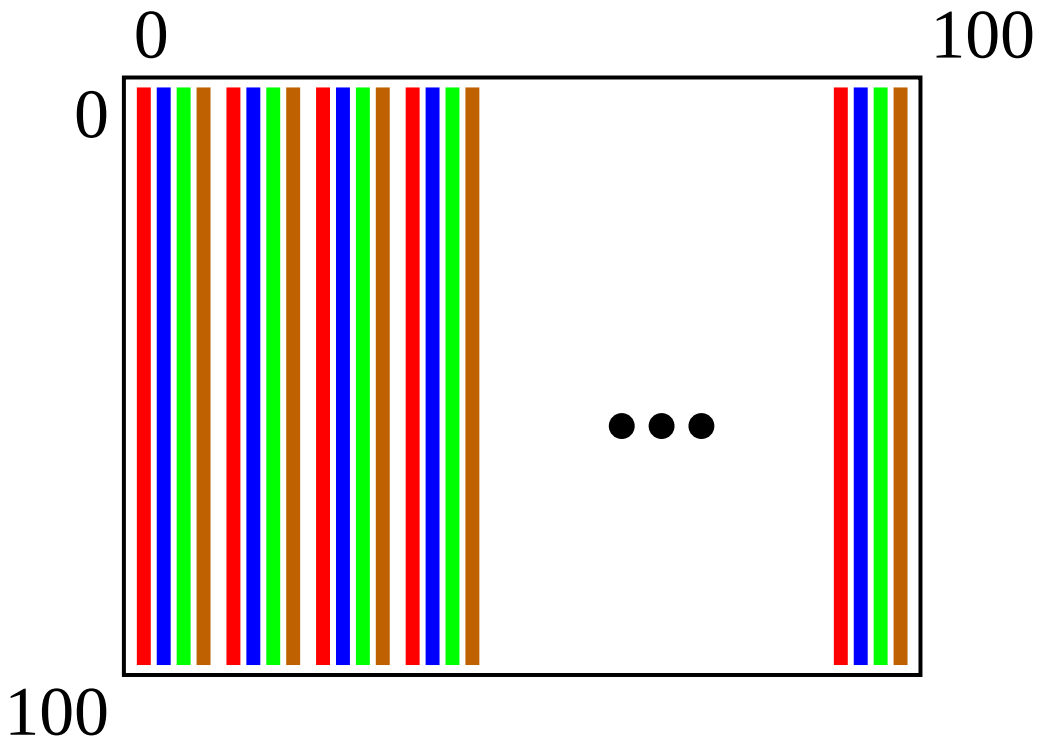


# Multi-Generator `With`-Loops

## Syntactical Representation:

```
res = with ([0,0] <= iv < [100,100] step [1,4]): op1(iv)
           ([0,1] <= iv < [100,100] step [1,4]): op2(iv)
           ([0,2] <= iv < [100,100] step [1,4]): op3(iv)
           ([0,3] <= iv < [100,100] step [1,4]): op4(iv)
      genarray( [100,100]);
```

## Graphical Representation:



# “Naïve” Compilation

```
for (iv[0] = 0; iv[0] < 100; iv[0]++) {
   for (iv[1] = 0; iv[1] < 100; iv[1] += 4) {
      res[iv] = op1(iv);
   }
}
```

first generator ($G_1$)

```
for (iv[0] = 0; iv[0] < 100; iv[0]++) {
   for (iv[1] = 1; iv[1] < 100; iv[1] += 4) {
      res[iv] = op2(iv);
   }
}
```

second generator ($G_2$)

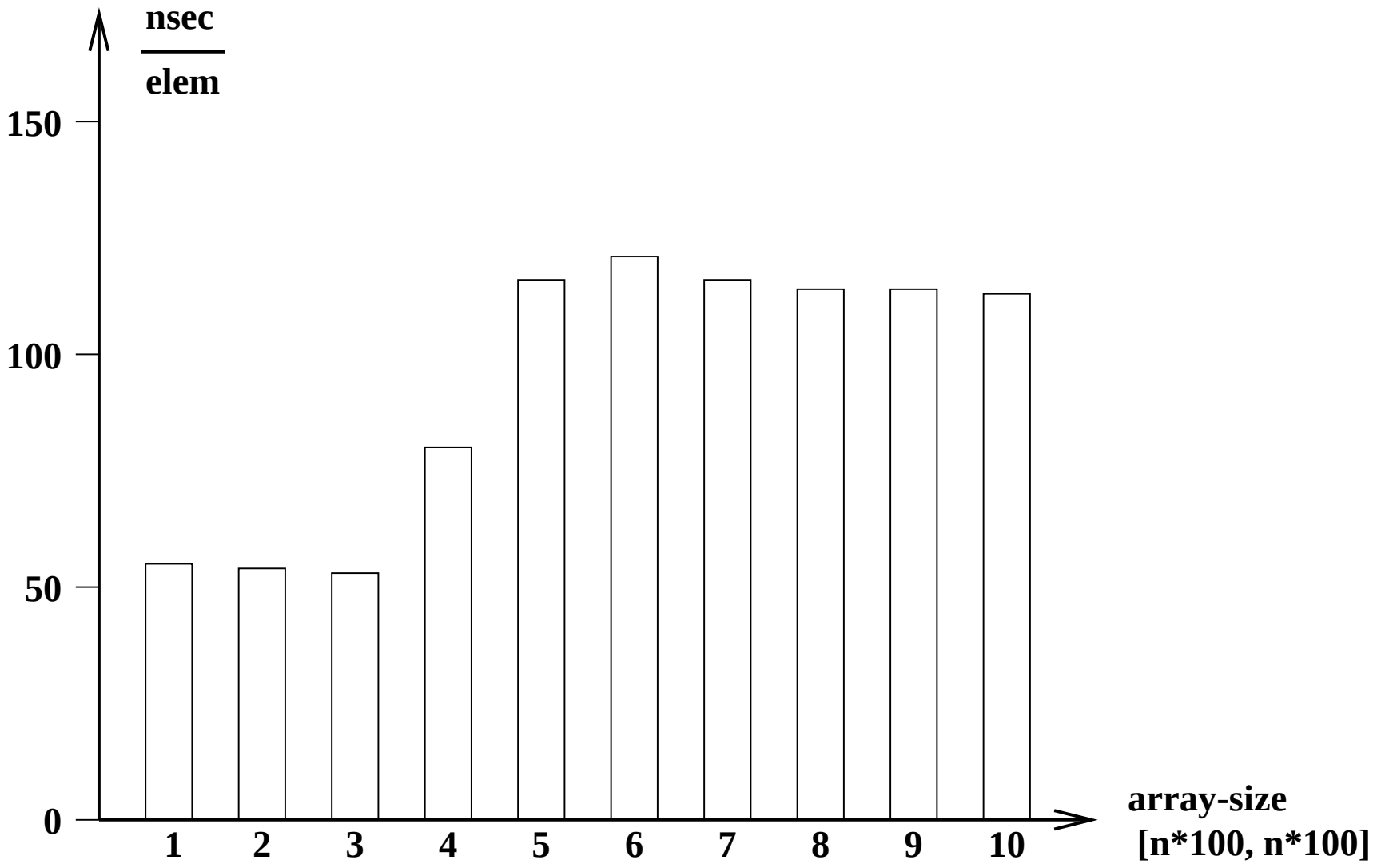
```
for (iv[0] = 0; iv[0] < 100; iv[0]++) {
   for (iv[1] = 2; iv[1] < 100; iv[1] += 4) {
      res[iv] = op3(iv);
   }
}
```

third generator ($G_3$)

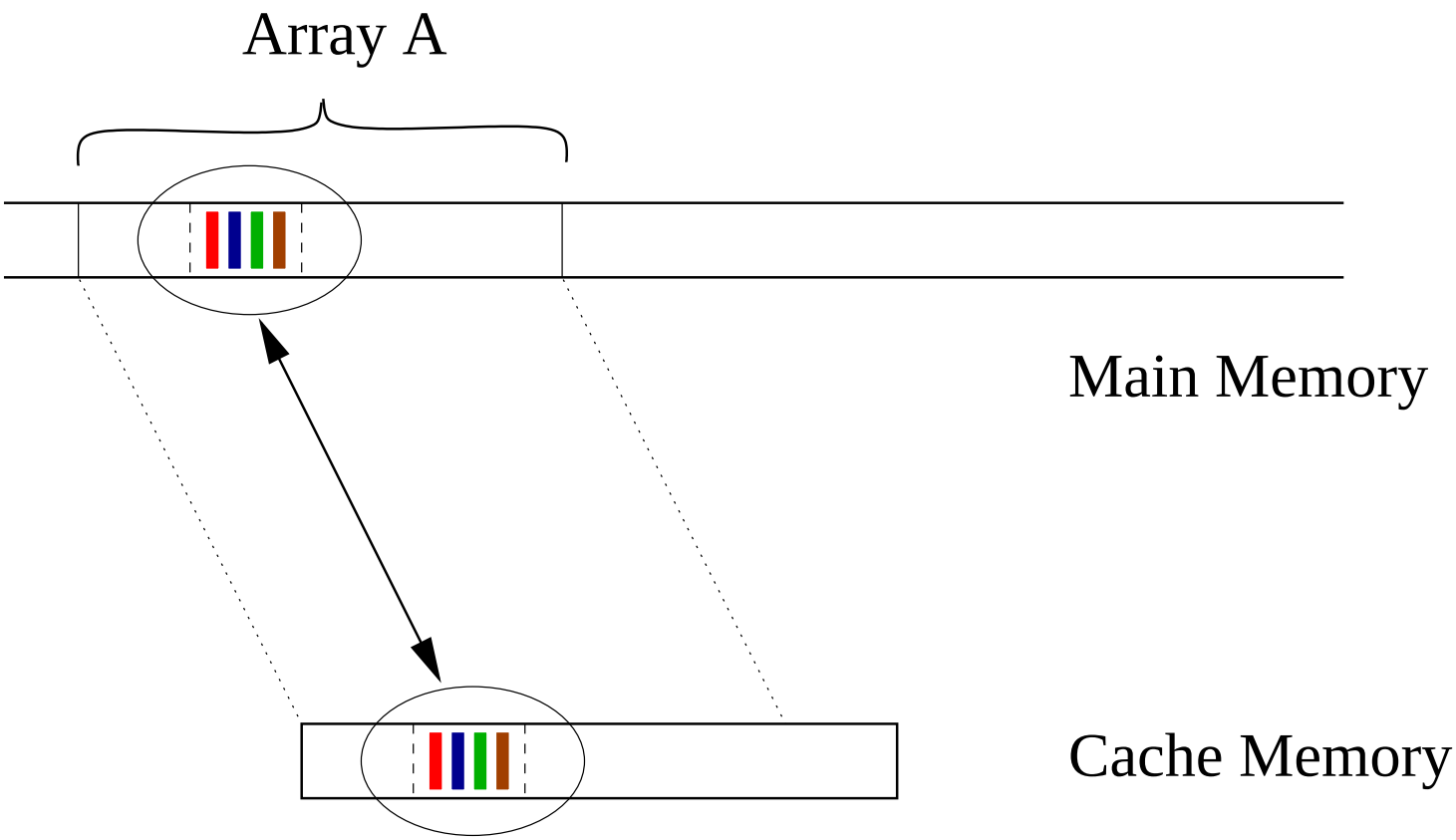
```
for (iv[0] = 0; iv[0] < 100; iv[0]++) {
   for (iv[1] = 3; iv[1] < 100; iv[1] += 4) {
      res[iv] = op4(iv);
   }
}
```

fourth generator ($G_4$)

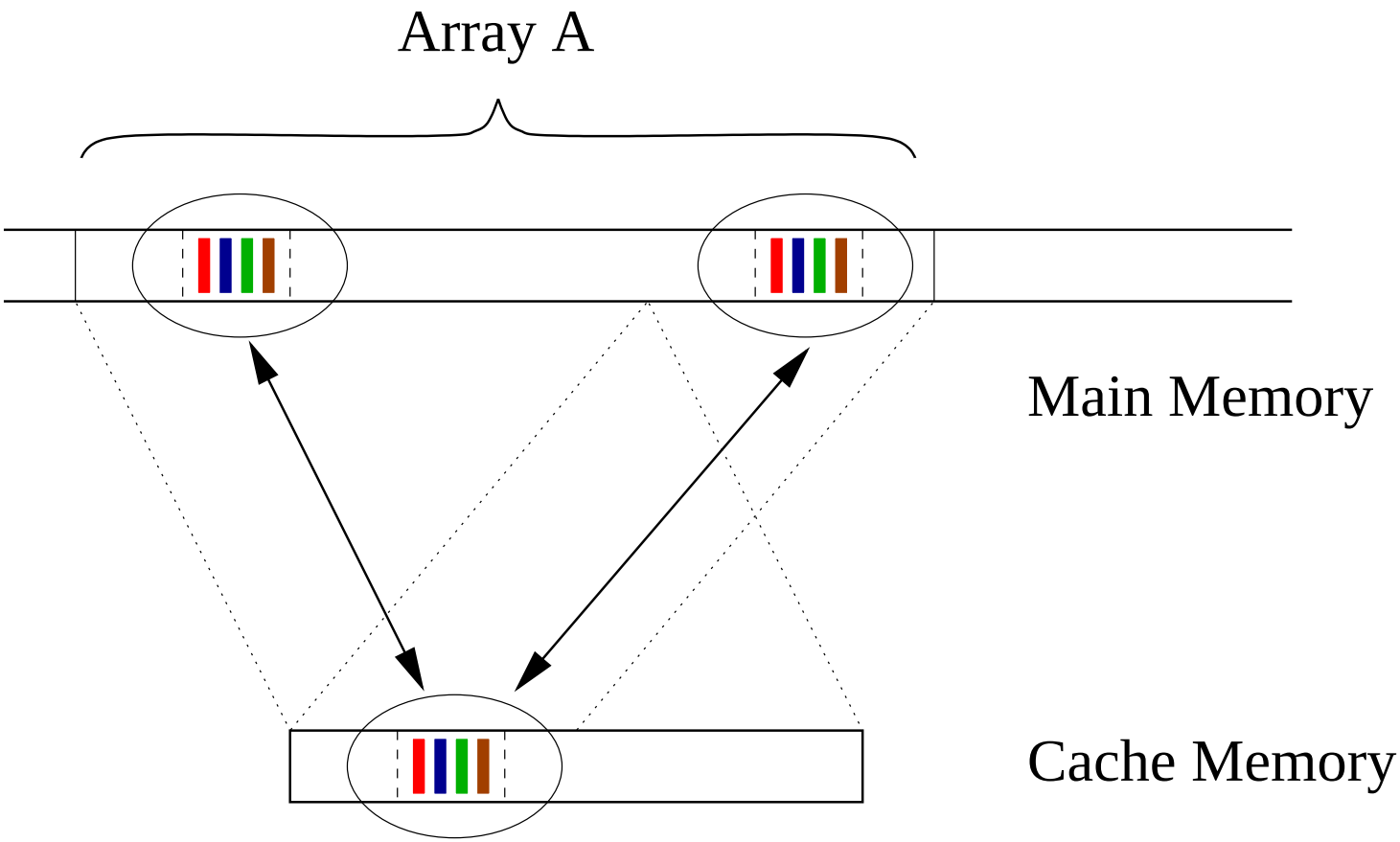
# “Naïve” Compilation: Runtimes



# The Impact of Caches (1)

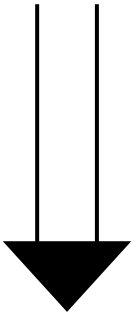


# The Impact of Caches (2)



# The Canonical Order

```
res = with ([0,0] <= iv < [100,100] step [1,4]): op1(iv)
           ([0,1] <= iv < [100,100] step [1,4]): op2(iv)
           ([0,2] <= iv < [100,100] step [1,4]): op3(iv)
           ([0,3] <= iv < [100,100] step [1,4]): op4(iv)
      genarray( [100,100]);
```



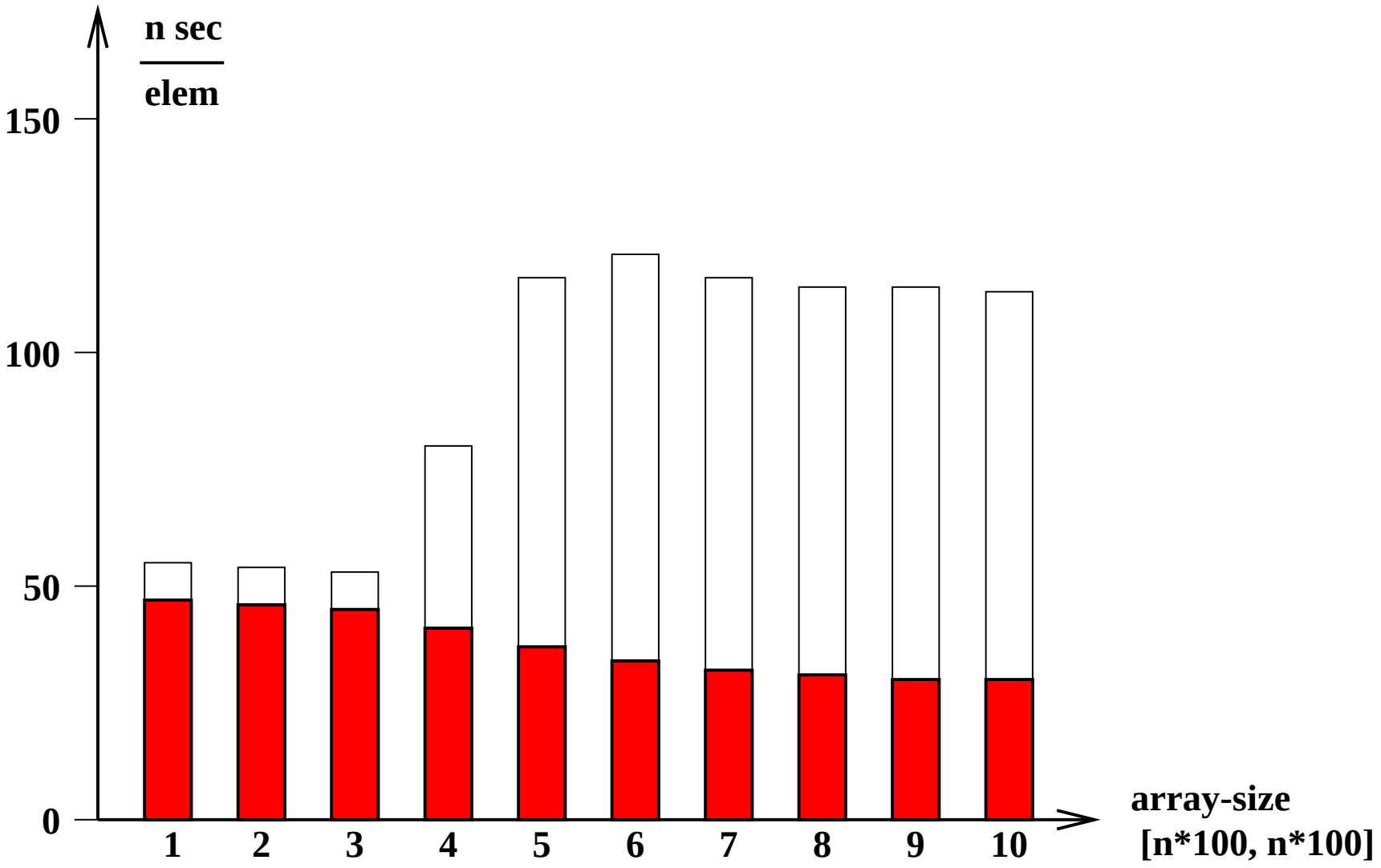
**Idea:**

**Compute array elements in the same order as they will be stored in memory.**

```
for (iv[0] = 0; iv[0] < 100; iv[0]++) {
   for (iv[1] = 0; iv[1] < 100; ) {
      res[iv] = op1(iv);
      iv[1] += 1;
      res[iv] = op2(iv);
      iv[1] += 1;
      res[iv] = op3(iv);
      iv[1] += 1;
      res[iv] = op4(iv);
      iv[1] += 1;
   }
}
```

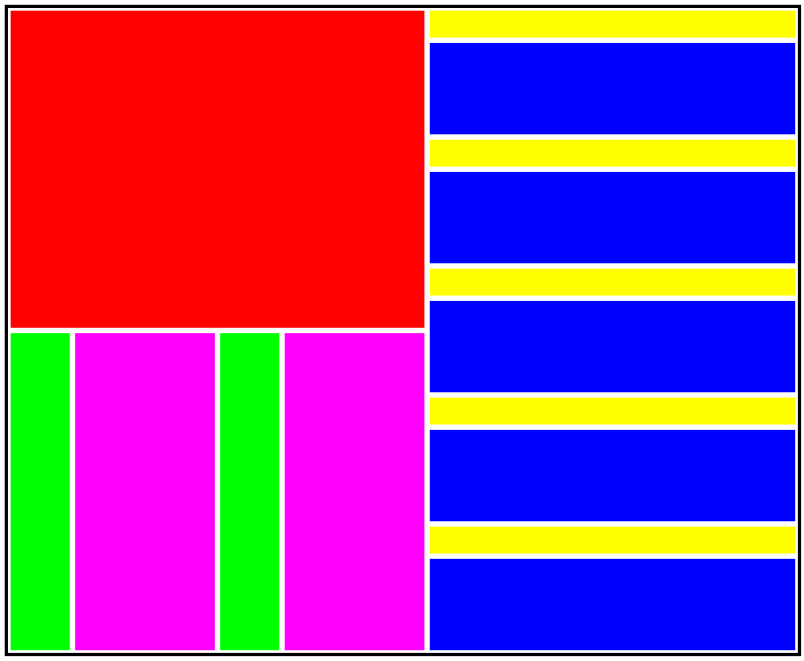
all generators $(G_1 \ldots G_4)$

# Runtimes: Naïve vs. Canonical

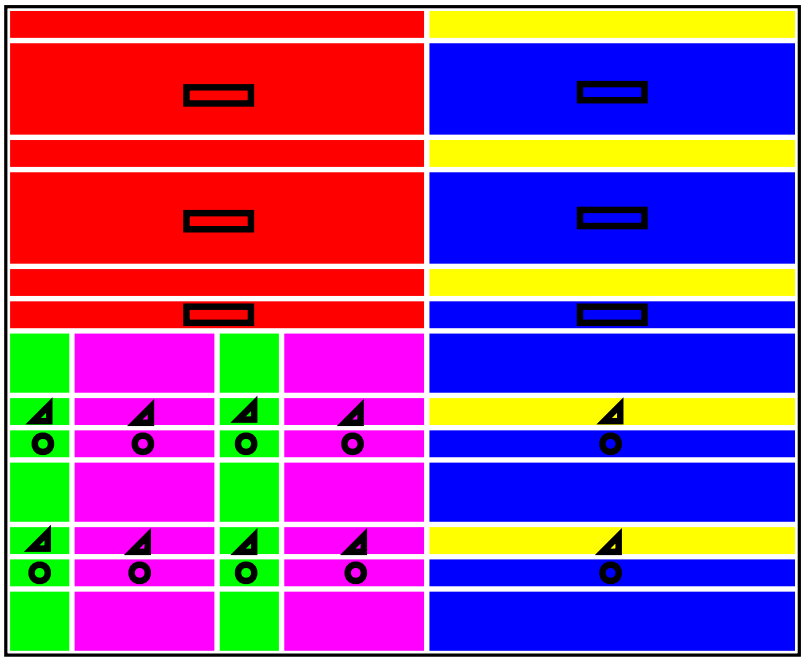


# Compilation for Canonical Order

## Example — Initial Layout:

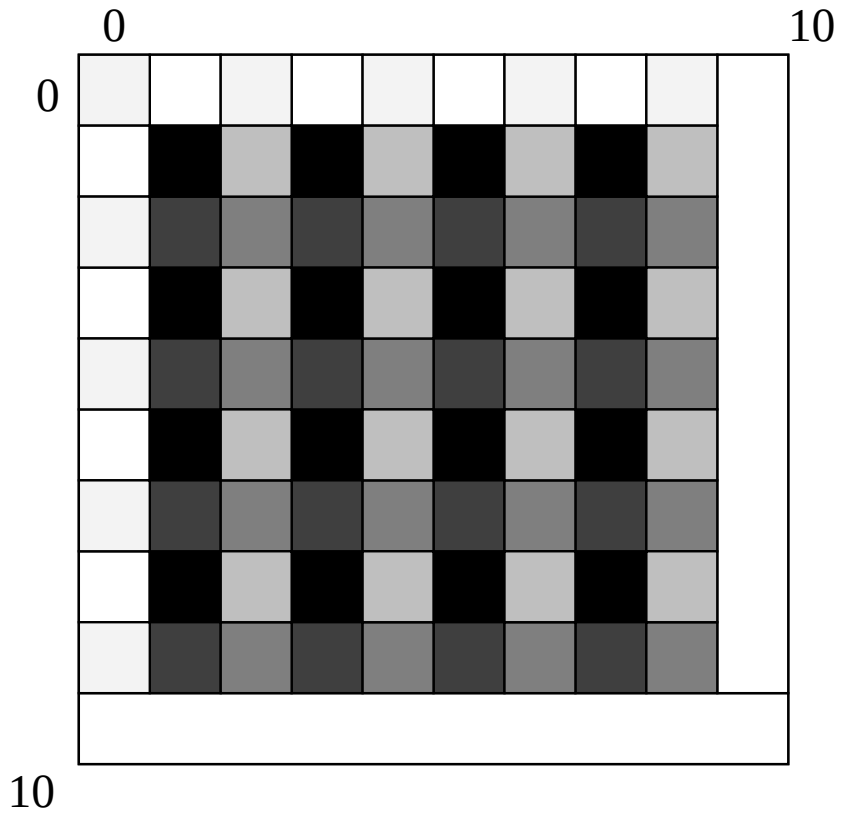


## Example — Final Layout:



# Multigrid Relaxation: Coarse to Fine Mapping

## The Algorithm:




$$\text{weights} = \begin{pmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & 1.0 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{pmatrix}$$

0

Coarse[iv/2]

0.5*(coarse[(iv+[0,1]-1)/2]+
coarse[(iv+[2,1]-1)/2])

coarse[iv/2]

0.5*(coarse[(iv+[1,0]-1)/2]+
coarse[(iv+[1,2]-1)/2])

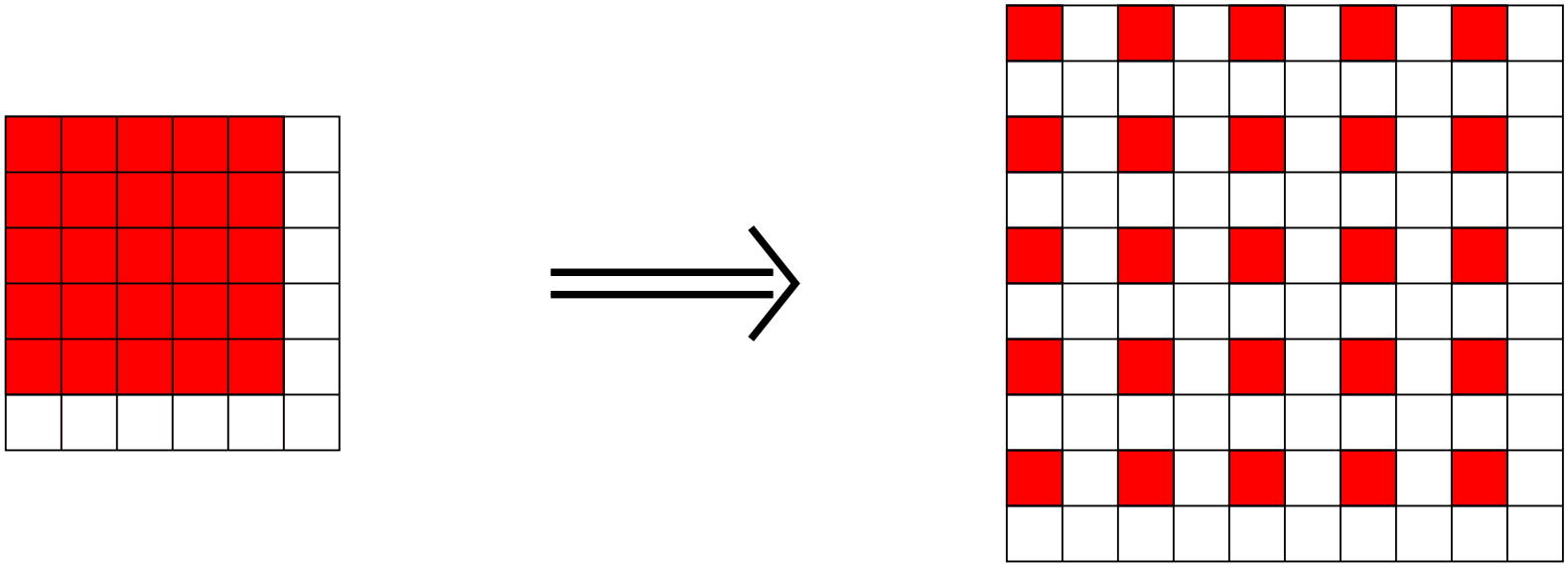
0.25*(coarse[(iv+[0,0]-1)/2]+coarse[(iv+[2,0]-1)/2]+
coarse[(iv+[0,2]-1)/2]+coarse[(iv+[2,2]-1)/2])

# Coarse to Fine Mapping

## SAC Code:

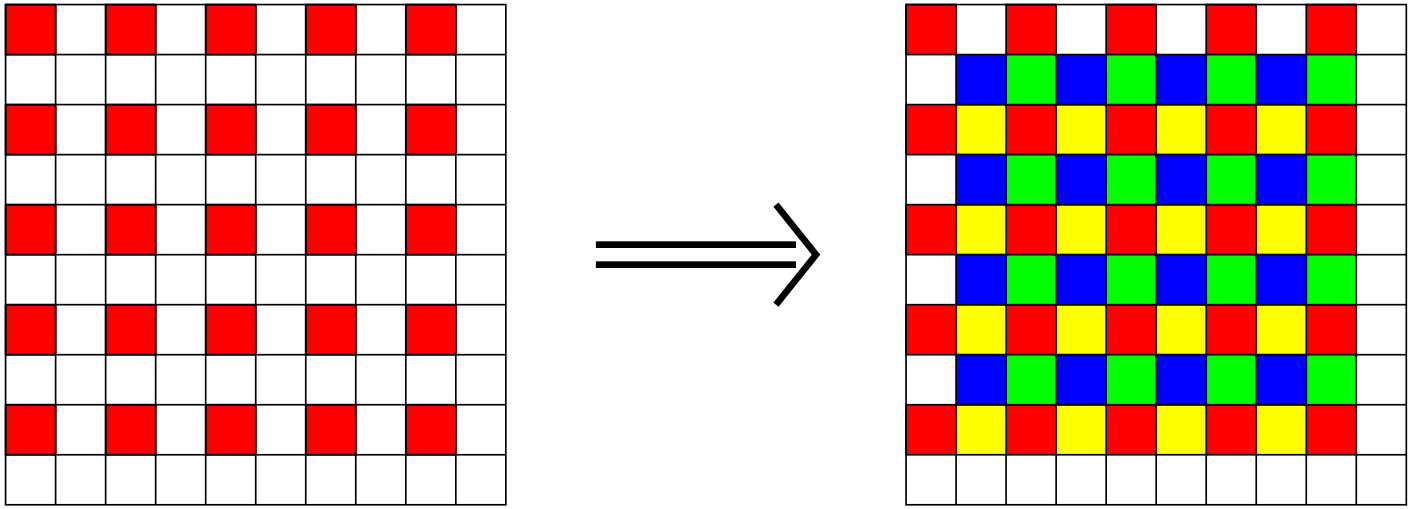
```
double[] Coarse2Fine( double[] coarse, double[] weights)
{
  fine = with (0*shape(coarse) <= iv < shape(coarse)
               step 0*shape(coarse)+2)
         genarray( 2*shape(coarse)-2, coarse[iv/2]);

  fine = with (0*shape(fine) < iv < shape(fine)-1) {
           val = with (0*shape(weights) <= delta < shape(weights))
                 fold( +, 0, weights[delta] * fine[iv+delta-1]);
         } modarray( fine, iv, val);
  return (fine);
}
```

# Multigrid Relaxation: Coarse to Fine Mapping (1)



```
fine = with ( . <= iv <= . step [2,2] )
       genarray( 2*shape(coarse)-2, coarse[iv/2]);
```

# Multigrid Relaxation: Coarse to Fine Mapping (2)

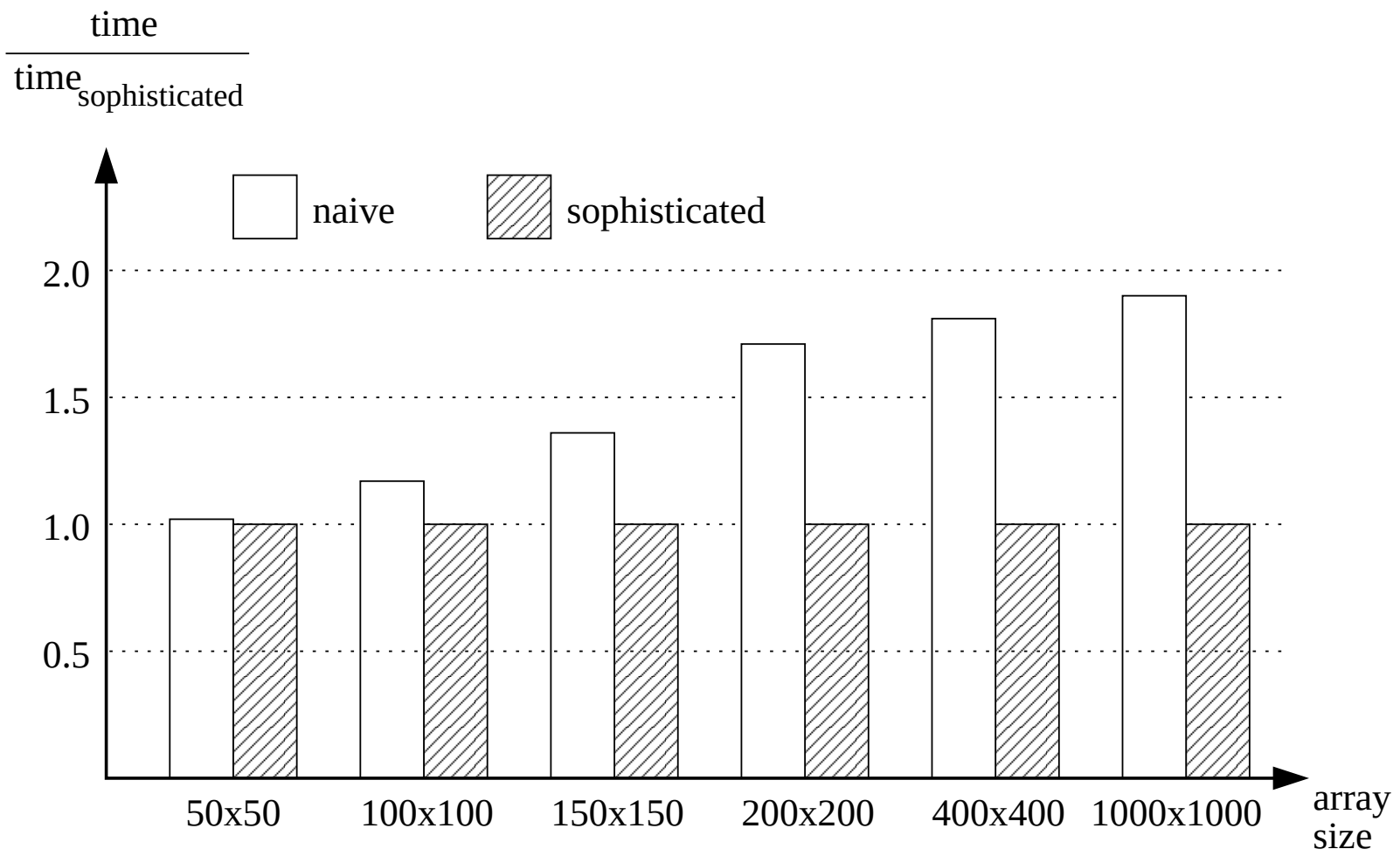


weights = $\begin{pmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & 1.0 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{pmatrix}$ ;

```
fine = with ( . < iv < . ) {
          val = sum( weights * tile( shape(weights), iv, fine));
       }
       modarray( fine, iv, val);
```

# Coarse to Fine Mapping

## Runtime Performance:



# Beyond the Canonical Order (1)

## Segmentation:

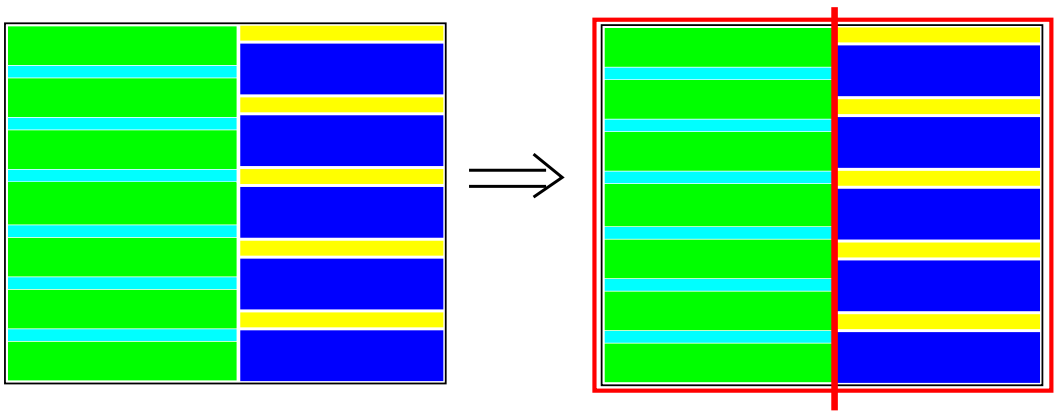
- Problem:

  Incompatible strides in adjacent generators.

  $\Longrightarrow$ Size of compiled code might explode.

- Solution:

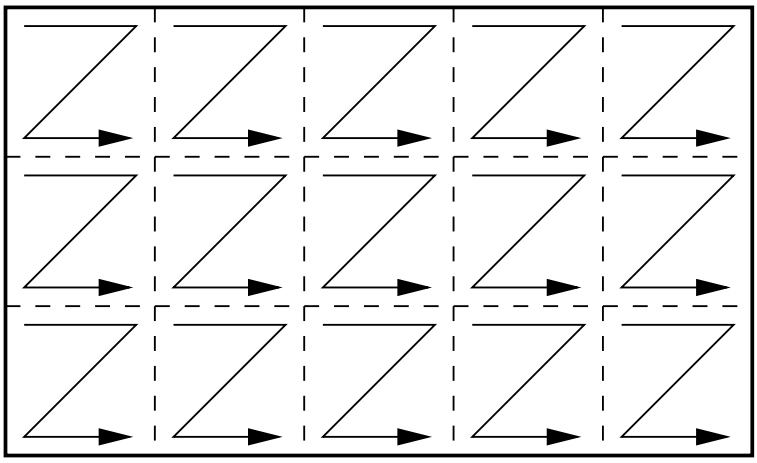
  Canonical order on selected groups of generators only.



# Beyond the Canonical Order (2)

## Tiling:

- Well-known optimization technique in high-performance computing.
- Rearrange iteration order.
  - ➜ Reduce distance between repeated accesses to one memory location.
  - ➜ Keep data in cache until it is reused.
- **Canonical order on small rectangular subarrays (tiles).**
  - ➜ Improve spatial and temporal locality.



# Conclusions

## Execution Order of `With`-Loops IMPORTANT!

- **Major performance impact**
  - ➢ loop overhead
  - ➢ cache utilization
- **Naïve compilation: generator by generator**
  - ➢ performance penalties on large arrays
- **Canonical order**
  - ➢ much better performance through spatial reuse
- **Segmentation**
  - ➢ pragma-annotated segmentation scheme
  - ➢ avoid explosion of compiled code in weird situations
- **Tiling**
  - ➢ pragma-annotated tiling scheme
  - ➢ exploit temporal reuse to further improve performance

## Future Work:

- Compiler-inferred segmentation scheme
- Compiler-inferred tiling scheme