New Insights on Scalar Promotion with the Polyhedral Model

Alec Sadler, **Nathan Chandanson**, Hugo Thievenaz, Christophe Alias

Inria & ENS Lyon

IMPACT'26, Kraków

Scalar Promotion

Scalar Promotion turn arrays into a collection of scalar variables

Benefits:

- Enable back-end optimizations (vectorization, register alloc.)
- Reduce memory traffic

```
int A[3];
int res = 0;
for (int i=0; i < 3; i++)
    A[i] = rand();
for (int i=0; i < 3; i++)
    res+= A[i];
```

SRA →

```
int a = rand();
int b = rand();
int res = rand();
res += a;
res += b;
```

Scalar Promotion

Scalar Promotion turn arrays into a collection of scalar variables

Benefits:

- Enable back-end optimizations (vectorization, register alloc.)
- Reduce memory traffic

Challenges:

- Few registers, large and multiple arrays:
  ☞ memory aware scheduling + array liveness analysis
- Mitigate control complexity

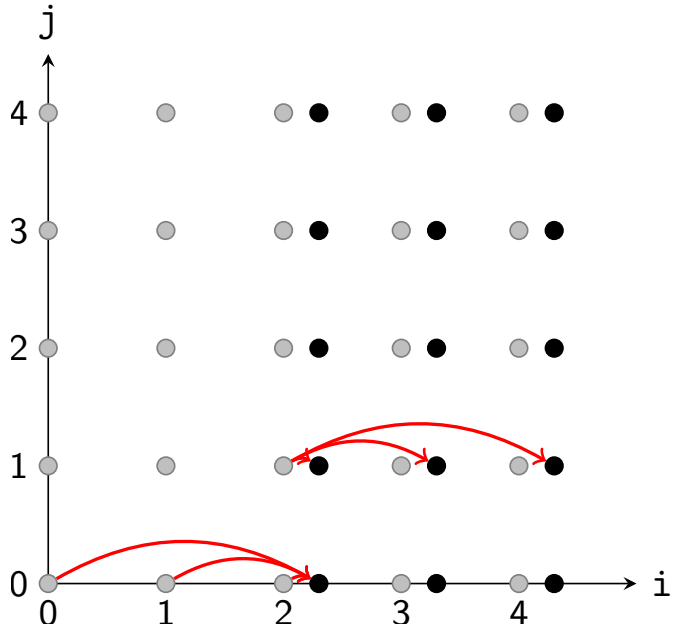
```
int A[3];
int res = 0;
for (int i=0; i < 3; i++)
    A[i] = rand();
for (int i=0; i < 3; i++)
    res+= A[i];
```

SRA →

```
int a = rand();
int b = rand();
int res = rand();
res += a;
res += b;
```

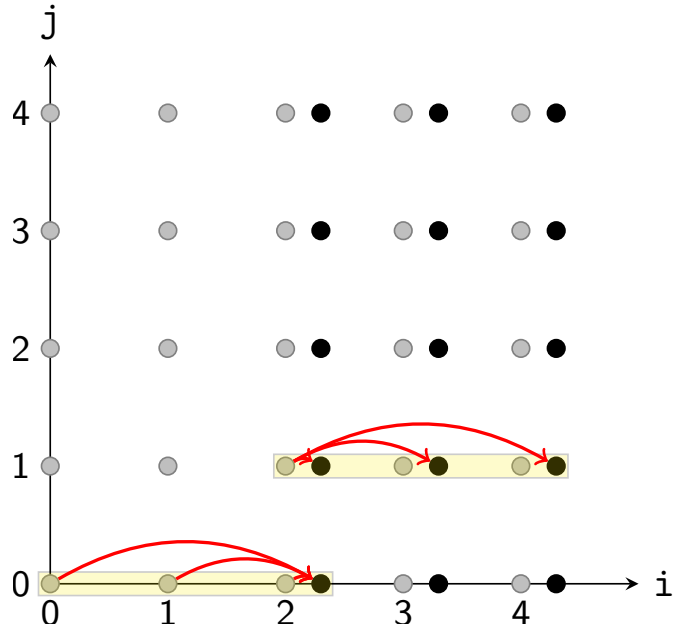
Case Study 1: 2D Blur Filter

```
for(j=0; j<N; j++)
  for(i=0; i<N; i++) {
P:  temp[i,j] = f(input,i);
    if(i>=2)
C:    out[i,j] =
        temp[i,j] +
        temp[i-1,j] +
        temp[i-2,j];
  }
```



Case Study 1: 2D Blur Filter

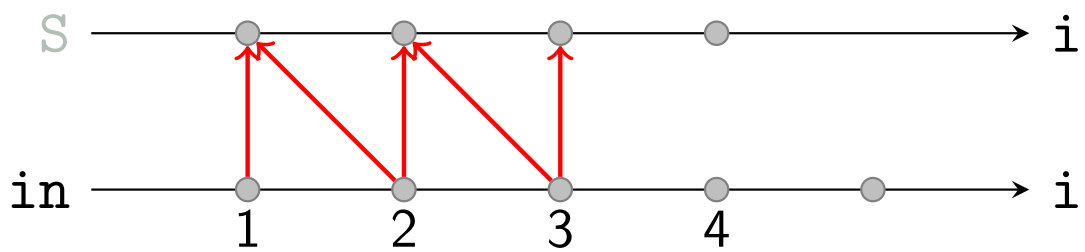
```
for(j=0; j<N; j++)
  for(i=0; i<N; i++) {
P:   temp[i,j] = f(input,i);
     if(i>=2)
C:     out[i,j] =
         temp[i,j] +
         temp[i-1,j] +
         temp[i-2,j];
  }
```



- Array liveness exposes **3-register scalarization**
- How to distribute the data among the registers?

Case Study 2: 1D Convolution

```
for(i=1; i<N-1; i++) {
   out[i] = in[i] + in[i+1]; //S
}
```

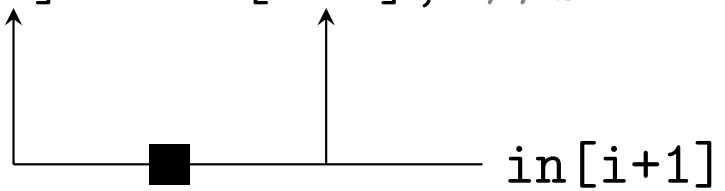


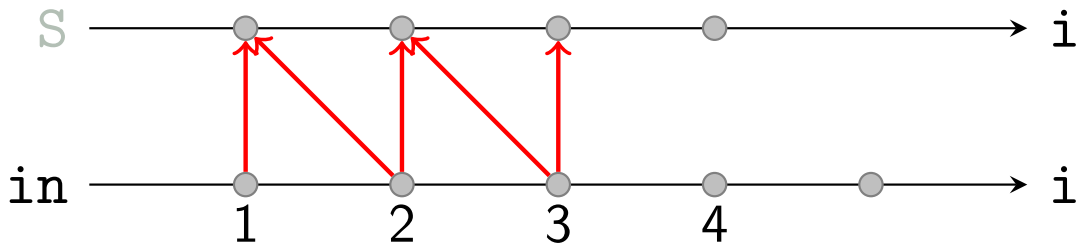
- No direct scalar promotion

Case Study 2: 1D Convolution

```
for(i=1; i<N-1; i++) {
   out[i] = in[i] + in[i+1]; //S
}
```

Data systolization





- No direct scalar promotion
- How to expose data systolization?

Related Work

Register tiling: scalar promotion bound to a loop tiling

- Perfect loop nest, rectangular tiling: [Jimenez'02], with different tile sizes [Domagala'16]
- Non-perfect loop nests, affine tiling [Renganarayana'09]

Related Work

Register tiling: scalar promotion bound to a loop tiling

- Perfect loop nest, rectangular tiling: [Jimenez'02], with different tile sizes [Domagala'16]
- Non-perfect loop nests, affine tiling [Renganarayana'09]

Data systolization:

- Common hand optimization in hardware design
- Automation on PPN [Verdoolaege'10] and DPN [Alias'21]

Contributions

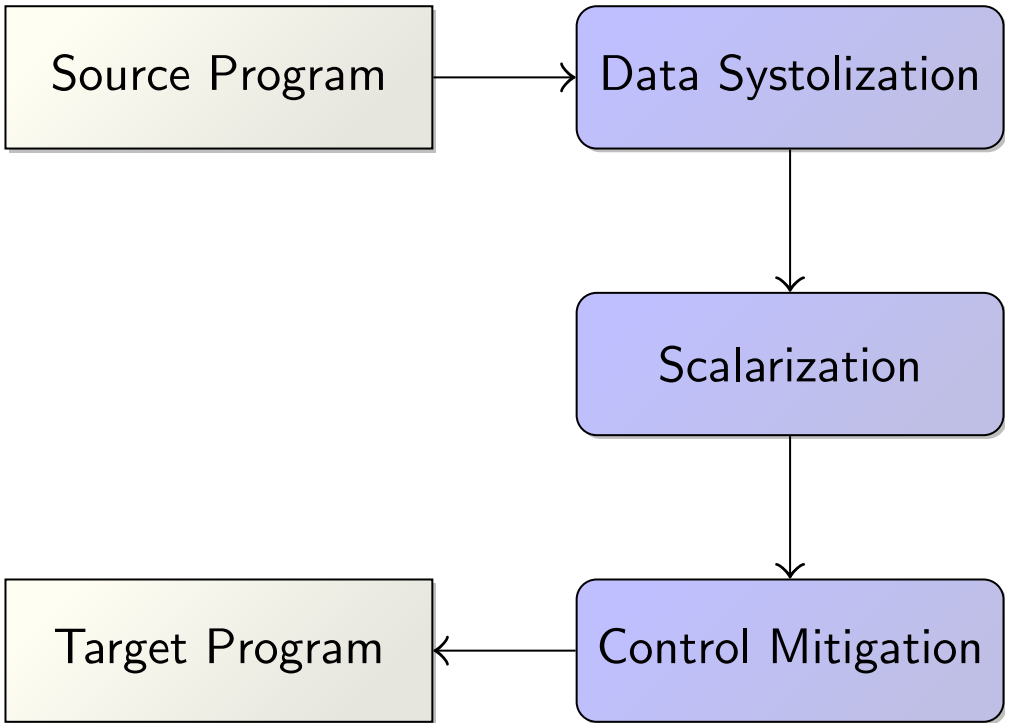
Unified approach for scalar promotion and data systolization

Mitigation of code complexity

Composable with other polyhedral passes

Preliminary validation on Polybenchs

Our Approach



- Generic polyhedral scalarization algorithm
- Preprocessing enabling data systolization
- Postprocessing of polyhedral IR to mitigate the control

Contraction-Driven Scalarisation

```
f[0]=0; f[1]=1;
for(i=2; i<=N; i++)
  f[i] = f[i-1] + f[i-2];
result = f[N];
```

Step 1: Array contraction $\sigma_f(i) = i \bmod 2$ [Thievenaz'24]

Contraction-Driven Scalarisation

```
f[0%2]=0; f[1%2]=1;
for(i=2; i<=N; i++)
  f[i%2] = f[(i-1)%2] + f[(i-2)%2];
result = f[N%2];
```

Step 1: Array contraction $\sigma_f(i) = i \bmod 2$ [Thievenaz'24]

Contraction-Driven Scalarisation

```
f[0%2]=0; f[1%2]=1;
for(i=2; i<=N; i++)
  f[i%2] = f[(i-1)%2] + f[(i-2)%2];
result = f[N%2];
```

Step 1: Array contraction $\sigma_f(i) = i \bmod 2$ [Thievenaz'24]

Step 2: Loop unroll, factor 2

Contraction-Driven Scalarisation

```
f[0%2]=0; f[1%2]=1;
for(i=2; i<=N; i+=2) {
  f[i%2] = f[(i-1)%2] + f[(i-2)%2];
  if(i+1 <= N)
    f[(i+1)%2] = f[i%2] + f[(i-1)%2]; }
result = f[N%2];
```

Step 1: Array contraction $\sigma_f(i) = i \bmod 2$ [Thievenaz'24]

Step 2: Loop unroll, factor 2

Contraction-Driven Scalarisation

```
f[0%2]=0; f[1%2]=1;
for(i=2; i<=N; i+=2) {
  f[i%2] = f[(i-1)%2] + f[(i-2)%2];
  if(i+1 <= N)
    f[(i+1)%2] = f[i%2] + f[(i-1)%2]; }
result = f[N%2];
```

Step 1: Array contraction $\sigma_f(i) = i \bmod 2$ [Thievenaz'24]

Step 2: Loop unroll, factor 2

Step 3: Scalarize using $i \bmod 2 = 0$

Contraction-Driven Scalarisation

```
R0=0; R1=1;
for(i=2; i<=N; i+=2) {
  R0 = R1 + R0;
  if(i+1 <= N)
    R1 = R0 + R1; }
result = (N%2==1?R1:R0);
```

Step 1: Array contraction $\sigma_f(i) = i \bmod 2$ [Thievenaz'24]

Step 2: Loop unroll, factor 2

Step 3: Scalarize using $i \bmod 2 = 0$

Challenges

- **Multiple arrays** with **differents mappings**. **How to unroll?**

```
for(i=1; i<=N; i++) {
  for(j=1; j<=N; j++) {
    A[i%2,j%3] = B[2i+j%3]
```

Challenges

- **Multiple arrays** with **differents mappings**. **How to unroll?**

```
for(i=1; i<=N; i++) {
  for(j=1; j<=N; j++) {
    A[i%2,j%3] = B[2i+j%3]
```

For each loop, collect the modulos and take the lcm:

☞ Loop `i`: 2,3 $\rightarrow$ lcm$(2,3)$ = factor 6

Challenges

- **Multiple arrays** with **differents mappings**. **How to unroll?**

```
for(i=1; i<=N; i++) {
  for(j=1; j<=N; j++) {
    A[i%2,j%3] = B[2i+j%3]
```

For each loop, collect the modulos and take the lcm:
☞ Loop `i`: 2,3 $\rightarrow$ lcm$(2,3)$ = factor 6

- Approach **parametrized by an affine schedule**
  ☞ consider schedule dimensions instead of loops

Preprocessing: Data Systolization

```
for(i=1; i<=N; i++) {
  out[i] = in[i] + in[i+1]; //S
}
```

Step 1: Expose the data pipelining

Preprocessing: Data Systolization

```
for(i=1-δ_{i,ℓ}; i<=N+δ_{i,u}; i++) {
  pipeline[i+1] = in[i+1]; //P
  if(1 ≤ i ≤ N)
    out[i] = pipeline[i] + pipeline[i+1]; //S
}
```

Step 1: Expose the data pipelining

Preprocessing: Data Systolization

```
for(i=1-δ_{i,ℓ}; i<=N+δ_{i,u}; i++) {
  pipeline[i+1] = in[i+1]; //P
  if(1 ≤ i ≤ N)
    out[i] = pipeline[i] + pipeline[i+1]; //S
}
```

Step 1: Expose the data pipelining

Step 2: Set initialization iterations from array dataflow analysis

Preprocessing: Data Systolization

```
for(i=0; i<=N; i++) {
  pipeline[i+1] = in[i+1]; //P
  if(1 ≤ i ≤ N)
    out[i] = pipeline[i] + pipeline[i+1]; //S
}
```

Step 1: Expose the data pipelining

Step 2: Set initialization iterations from array dataflow analysis

Preprocessing: Data Systolization

```
for(i=0; i<=N; i++) {
  pipeline[i+1] = in[i+1]; //P
  if(1 ≤ i ≤ N)
    out[i] = pipeline[i] + pipeline[i+1]; //S
}
```

Step 1: Expose the data pipelining

Step 2: Set initialization iterations from array dataflow analysis

Step 3: Contract pipeline array: $\sigma_{pipeline}(i) = i \bmod 2$

Step 4: Apply our scalarization procedure

Preprocessing: Data Systolization

```
for(i=0; i<=N; i+=2) {
  R1 = in[i+1];
  if(i >= 1)
    out[i] = R0 + R1;
  R0 = in[i+2];
  if(i+1 >= 1)
    out[i+1] = R1 + R0;
}
```

Step 1: Expose the data pipelining

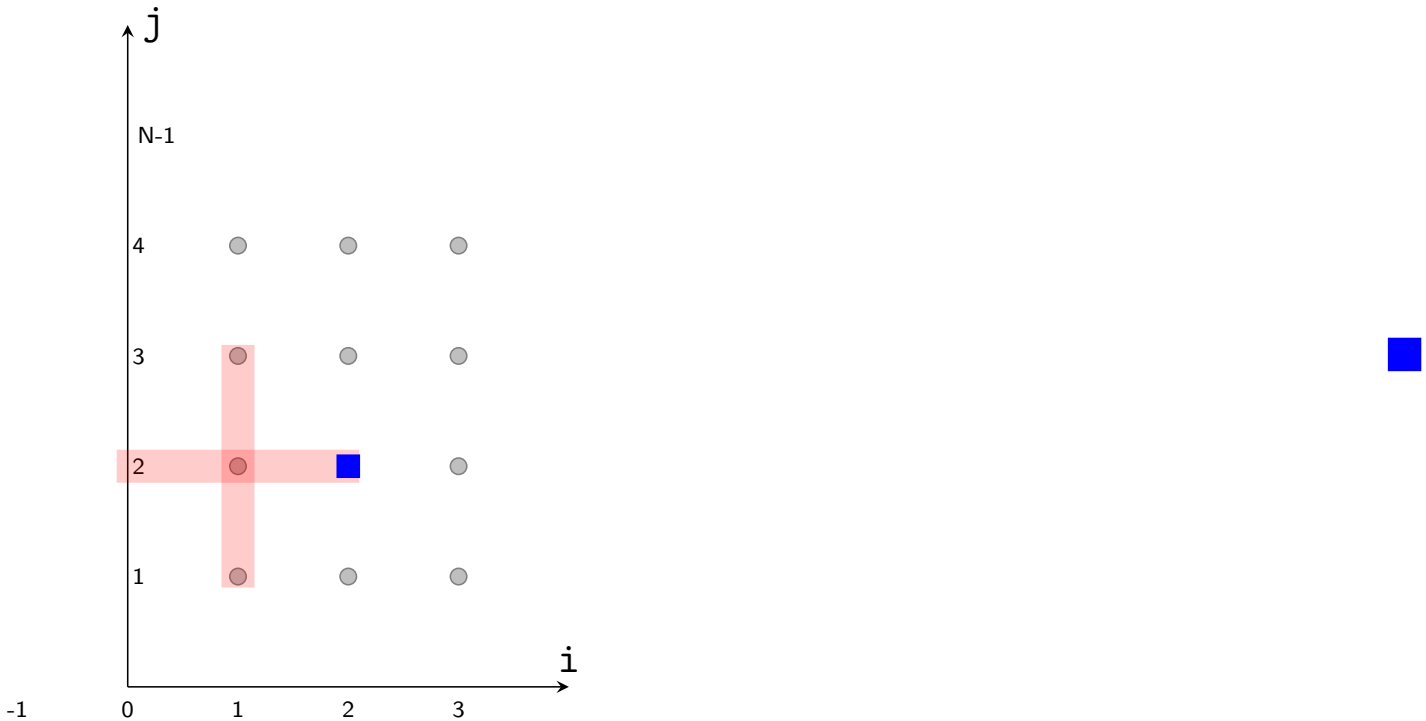
Step 2: Set initialization iterations from array dataflow analysis

Step 3: Contract pipeline array: $\sigma_{pipeline}(i) = i \bmod 2$

Step 4: Apply our scalarization procedure

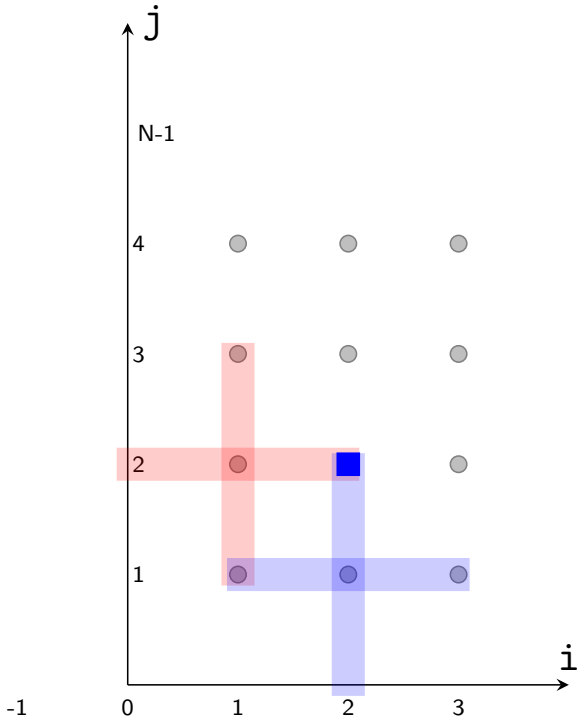
Multidimensional Case: 2D Convolution

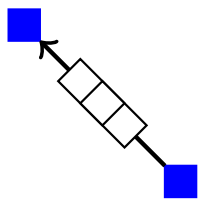
```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```



Multidimensional Case: 2D Convolution

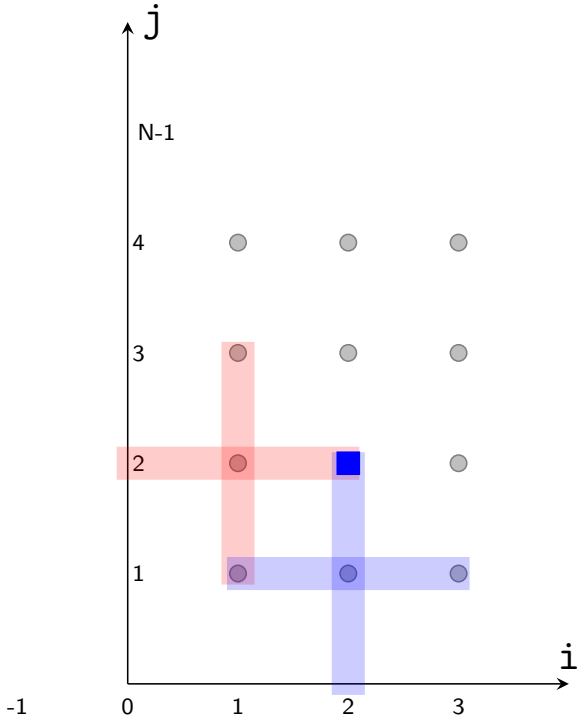
```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```

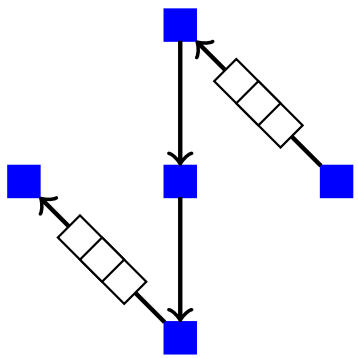




Multidimensional Case: 2D Convolution

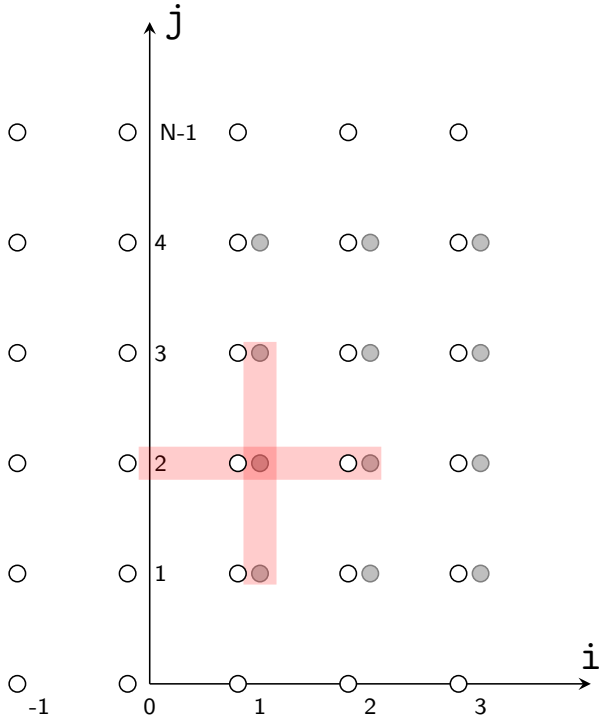
```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```





Multidimensional Case: 2D Convolution

```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```

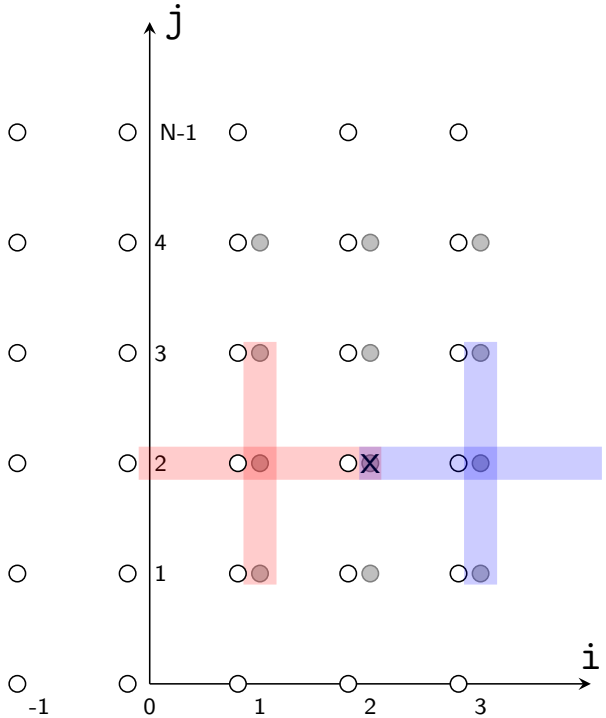


Same process with initialization

```
pipeline[i,j] = in[i+1,j];
```

Multidimensional Case: 2D Convolution

```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```

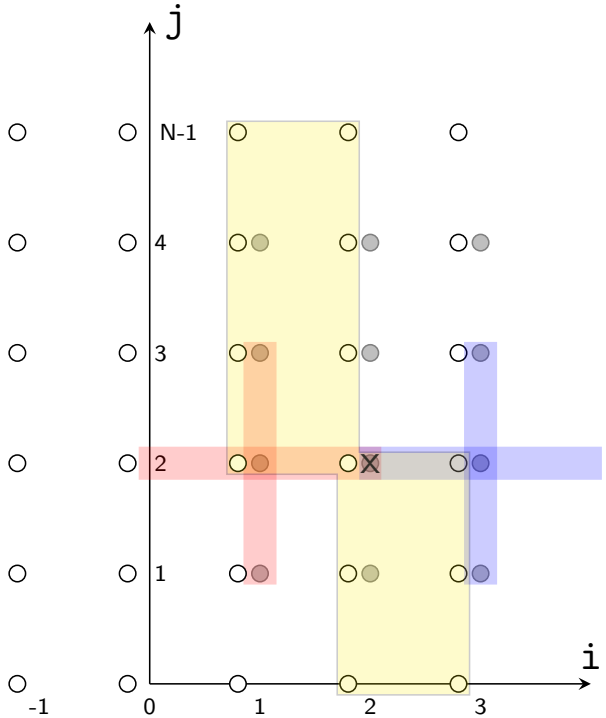


Same process with initialization

```
pipeline[i,j] = in[i+1,j];
```

Multidimensional Case: 2D Convolution

```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```



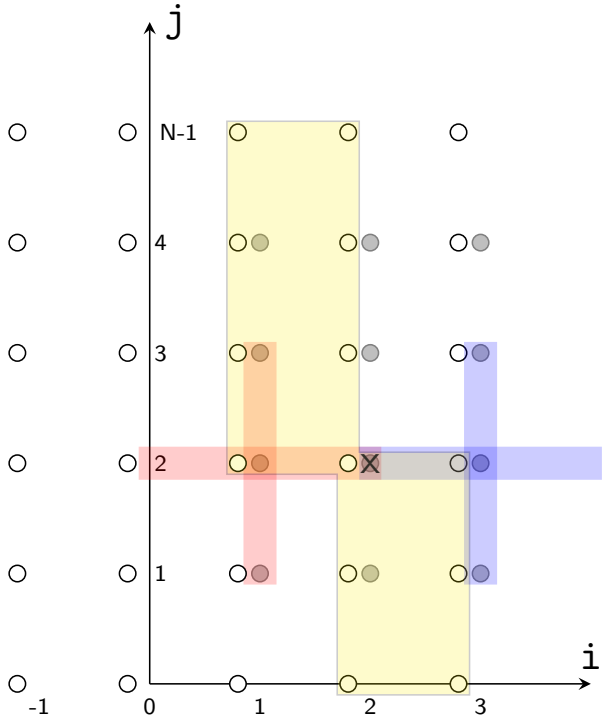
Same process with initialization

`pipeline[i,j] = in[i+1,j];`

Parametrized liveness!

Multidimensional Case: 2D Convolution

```
for(i=1; i<N-1; i++) {
  for(j=1; j<N-1; j++) {
    out[i,j] = in[i-1,j] + ... + in[i+1,j];
}}
```



Same process with initialization

`pipeline[i,j] = in[i+1,j];`

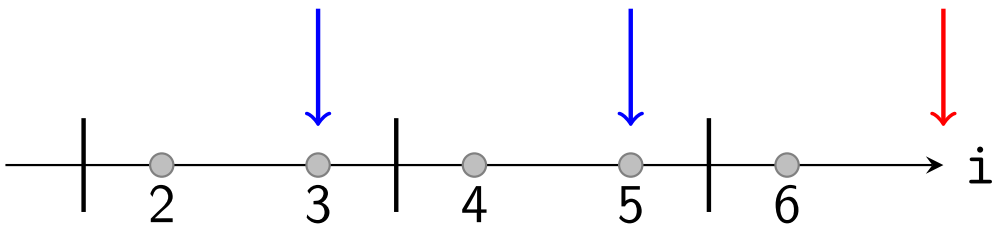
Parametrized liveness!

Solution:

- Cut with hyperplane $\phi(i, j) = j$
- Same initialization iterations for each tile
- Contract and scalarize

Postprocessing: Optimized Code Generation

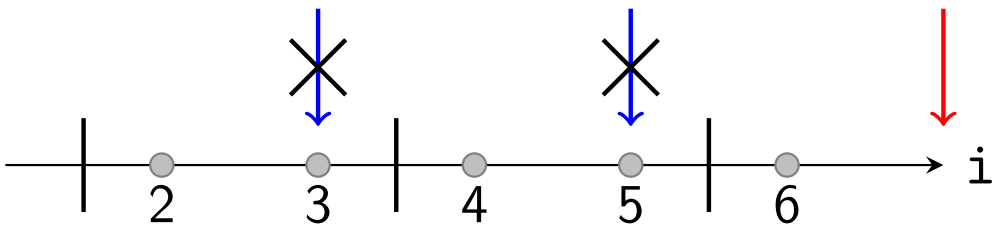
```
R0=0; R1=1;
for(i=2; i<=N; i+=2) {
   R0 = R1 + R0;
   if(i+1 <= N)
      R1 = R0 + R1; }
result = (N%2==1?R1:R0);
```



Corner case control executed each iteration!

Postprocessing: Optimized Code Generation

```
R0=0; R1=1;
for(i=2; i<=N; i+=2) {
   R0 = R1 + R0;
   if(i+1 <= N)
      R1 = R0 + R1; }
result = (N%2==1?R1:R0);
```



Corner case control executed each iteration!

Solution: separate steady iterations `i` from corner cases

Experimental Setup

Implemented: Scalarization + mitigation, **by hand:** data systolization preprocessing, slightly different scheme.

Benchmarks:

- Polybench kernels: *gemm*, *jacobi-1d*, *jacobi-2d*, *2mm*, *gesummv*, *symm*, *correlation*, *covariance*.
- *2D blur filter* and *fibonnacci*

Baseline: `g++ -O3`, v15.1.1

Machine: x86-64 AMD processor 8 cores, 16 `%xmmi` vector/floating point registers.

Measures with: `perforator` profiling utility based on `perf`.

Experimental Results

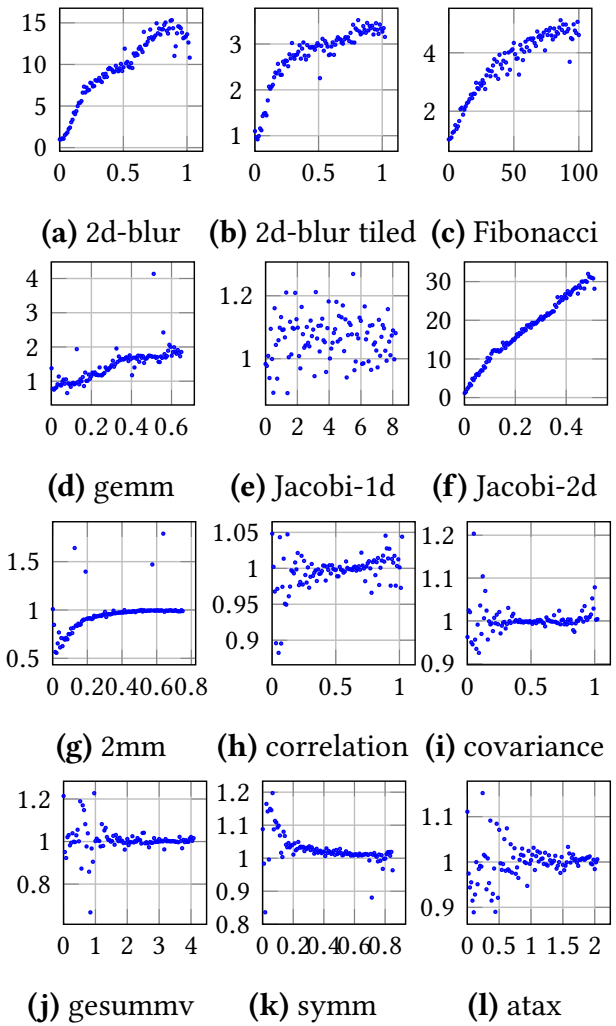


Figure 1. Speedup of each kernels. Speedups on the Y-axis are plotted over the input size N on the X-axis, scaled at 10^3.

- correlation, covariance: scalarization of non-significant part
- gesumv, symm, atax: scalarized with -O3
- 2mm: code complexity

Post-processing evaluation

	2d-blur-filter		gemm		2mm	
	Scalarized	Post-processed	Scalarized	Post-processed	Scalarized	Post-processed
cpu-cycles	712k	608k	32,7M	32,5M	87,5M	87,5M
instructions	3,2M	1,7M	117M	124M	424M	424M
branch-instructions	208k	22k	2,6M	2,6M	52,8M	53,4M
branch-misses	136	145	5,6k	5,6k	6,5k	7k
cache-references	200k	202k	2,3M	2,3M	5,9M	5,6M
cache-misses	26k	25k	360k	371k	1244k	948k
time-elapsed	758μs	507μs	24,8ms	24,4ms	63,5m	63,6ms

Post-processing evaluation

	2d-blur-filter		gemm		2mm	
	Scalarized	Post-processed	Scalarized	Post-processed	Scalarized	Post-processed
cpu-cycles	712k	608k	32,7M	32,5M	87,5M	87,5M
instructions	3,2M	1,7M	117M	124M	424M	424M
branch-instructions	208k	22k	2,6M	2,6M	52,8M	53,4M
branch-misses	136	145	5,6k	5,6k	6,5k	7k
cache-references	200k	202k	2,3M	2,3M	5,9M	5,6M
cache-misses	26k	25k	360k	371k	1244k	948k
time-elapsed	758μs	507μs	24,8ms	24,4ms	63,5m	63,6ms

- 2d-blur-filter: effective

Post-processing evaluation

	2d-blur-filter		gemm		2mm	
	Scalarized	Post-processed	Scalarized	Post-processed	Scalarized	Post-processed
cpu-cycles	712k	608k	32,7M	32,5M	87,5M	87,5M
instructions	3,2M	1,7M	117M	124M	424M	424M
branch-instructions	208k	22k	2,6M	2,6M	52,8M	53,4M
branch-misses	136	145	5,6k	5,6k	6,5k	7k
cache-references	200k	202k	2,3M	2,3M	5,9M	5,6M
cache-misses	26k	25k	360k	371k	1244k	948k
time-elapsed	758μs	507μs	24,8ms	24,4ms	63,5m	63,6ms

- 2d-blur-filter: effective
- gemm, 2mm: same

Post-processing evaluation

	2d-blur-filter		gemm		2mm	
	Scalarized	Post-processed	Scalarized	Post-processed	Scalarized	Post-processed
cpu-cycles	712k	608k	32,7M	32,5M	87,5M	87,5M
instructions	3,2M	1,7M	117M	124M	424M	424M
branch-instructions	208k	22k	2,6M	2,6M	52,8M	53,4M
branch-misses	136	145	5,6k	5,6k	6,5k	7k
cache-references	200k	202k	2,3M	2,3M	5,9M	5,6M
cache-misses	26k	25k	360k	371k	1244k	948k
time-elapsed	758μs	507μs	24,8ms	24,4ms	63,5m	63,6ms

- 2d-blur-filter: effective
- gemm, 2mm: same
- Never worse when separating

Conclusion

Contributions:

- Unified approach for scalarization and data systolization
- Parametrized by an affine schedule and an array contraction mapping
- Composable with other polyhedral compilation passes

Perspectives:

- Full automation of data systolization
- Reduce further the code complexity
- Interplay with other polyhedral compilation passes

Questions ?

```
for (i = 0; i < N - 1; i++) {
  systo[2] = A[i+1];
  if ( i >= 1 && i <= N-2)
    B[i] = 0.33333 * (systo[0] + systo[1] +
        systo[2]);
  for(offset=0; offset<=1; offset++)
    systo[offset] = systo[offset+1];
}
```

```
for (i = -1; i < N-1; i++) {
  for (j = 0; j < N; j++) {
    systo[12] = A[i+1][j];
    if ( i>= 1 && i <= N-1 && j >= 1 && j <=
        N-1)
      B[i][j] = 0.2 *
          (systo[0]+systo[5]+systo[6]+systo[7]
          +systo[12]);
    for(offset=0; offset<=11; offset++)
      systo[offset] = systo[offset+1];
    }
  }
```