# Introduction to Automated Polyhedral Code Optimizations and Tiling

Alain Darte

CNRS, Compsys team
Laboratoire de l'Informatique du Parallélisme
École normale supérieure de Lyon

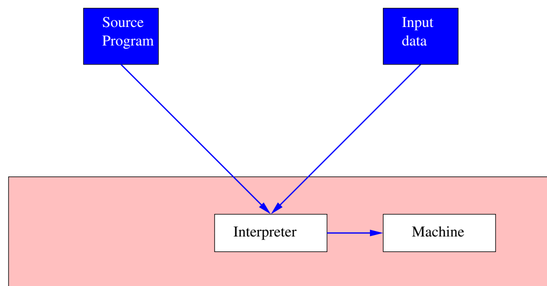
Spring School on Numerical Simulation
and Polyhedral Code Optimizations
Saint Germain au Mont d'Or, May 10, 2016

# Outline

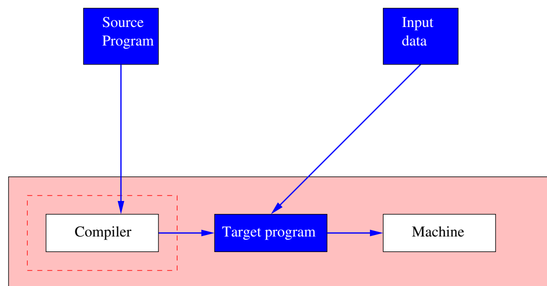

1. Generalities
2. Polyhedral compilation
3. Classic loop transformations
4. Systems of uniform recurrence equations
5. Detection of loop parallelism
6. Kernel offloading and loop tiling
7. Inter-tile data reuse and local storage

# GENERALITIES

# Compiler: generalities

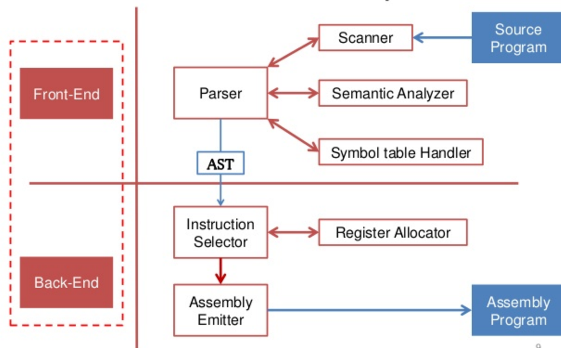


# Compiler: generalities




- AoT vs JIT
- Static vs dynamic
- Cross compilation
- Parametric codes
- Worst-case opt. vs average-case opt.
- Language-specific or multi-language
- Intermediate representations (IR)
- Runtime & libraries

# Front-end and back-end




Figures from http://fr.slideshare.net/ssuser2cbb78/gcc-37841549.

## Front-end

- Target independent
- Source-to-source
- High-level transf.
- Task & loop par.
- Memory opt.

## Back-end

- Target dependent
- Instr-level par. (ILP)
- Pipelining
- AVX, SIMD instr.
- SSA and registers

# Moore's law and Dennard scaling

Moore's law (1965)

Number of transistors 2x every 2 years, performance x2 every 18 months.

Dennard scaling (1974)

Scaling size, voltage, frequency, with constant power density: $P \sim CV^2 f$.

# Moore's law and Dennard scaling

## Moore's law (1965)

Number of transistors 2x every 2 years, performance x2 every 18 months.

## Dennard scaling (1974)

Scaling size, voltage, frequency, with constant power density: $P \sim CV^2 f$.

If reduction of width gates by factor $\lambda$:

- Reduction by $\lambda$ of capacitance ($C \sim A/\Delta$).
- Reduction by $\lambda$ of voltage, intensity, and delay time ($1/f \sim CV/I$).
- $P' = (C/\lambda).(V^2/\lambda^2).(\lambda f) = P/\lambda^2$ thus power density constant.
- Or can reduce $V$ by less with better $f$, but power increase.

# Moore's law and Dennard scaling

## Moore's law (1965)

Number of transistors 2x every 2 years, performance x2 every 18 months.

## Dennard scaling (1974)

Scaling size, voltage, frequency, with constant power density: $P \sim CV^2 f$.

If reduction of width gates by factor $\lambda$:

- Reduction by $\lambda$ of capacitance ($C \sim A/\Delta$).
- Reduction by $\lambda$ of voltage, intensity, and delay time ($1/f \sim CV/I$).
- $P' = (C/\lambda).(V^2/\lambda^2).(\lambda f) = P/\lambda^2$ thus power density constant.
- Or can reduce $V$ by less with better $f$, but power increase.

## End of Dennard scaling $f \sim V - V_t$ (supply minus threshold voltage).

But leakage power not negligible if $V_t$ too small.

- Frequency max around 2006 ($\sim 4GHz$).
- ILP, low-power CPU designs, multicore designs, GPUs.

☛ More burden on the programmers & compilers.

# The almost useless Amdahl's law

See "A few bad ideas on the way to the triumph of parallel computing", R. Schreiber, JPDC'14.

## Amdahl's law (1967)

- $T_{\text{seq}} = sT_{\text{seq}} + (1-s)T_{\text{seq}}$, with $s$ fraction not parallelizable.
- With $p$ proc., $T_{\text{par}}/T_{\text{seq}} \geq s + (1-s)/p \geq s$. Speed-up $\leq 1/s$.

# The almost useless Amdahl's law

See "A few bad ideas on the way to the triumph of parallel computing", R. Schreiber, JPDC'14.

## Amdahl's law (1967)

- $T_{\text{seq}} = sT_{\text{seq}} + (1-s)T_{\text{seq}}$, with $s$ fraction not parallelizable.
- With $p$ proc., $T_{\text{par}}/T_{\text{seq}} \geq s + (1-s)/p \geq s$. Speed-up $\leq 1/s$.

Simplistic view: correct formula in wrong model ($\sim$ *PRAM*).

- Memory, comm., dedicated cores, program scaling, algorithmics!
- No insight even in favorable case, e.g., 2 independent purely sequential threads. Better: degree of parallelism, critical path, ratio comm./computation, data locality (spatial & temporal), bandwidth.
- But: make sure single core perf. remains good in parallel implem.

# The almost useless Amdahl's law

See "A few bad ideas on the way to the triumph of parallel computing", R. Schreiber, JPDC'14.

Amdahl's law (1967) Useless except to show what it does not consider

- $T_{\text{seq}} = sT_{\text{seq}} + (1-s)T_{\text{seq}}$, with $s$ fraction not parallelizable.
- With $p$ proc., $T_{\text{par}}/T_{\text{seq}} \geq s + (1-s)/p \geq s$. Speed-up $\leq 1/s$.

Simplistic view: correct formula in wrong model ($\sim$ *PRAM*).

- Memory, comm., dedicated cores, program scaling, algorithmics!
- No insight even in favorable case, e.g., 2 independent purely sequential threads. Better: degree of parallelism, critical path, ratio comm./computation, data locality (spatial & temporal), bandwidth.
- But: make sure single core perf. remains good in parallel implem.

# The almost useless Amdahl's law

See "A few bad ideas on the way to the triumph of parallel computing", R. Schreiber, JPDC'14.

Amdahl's law (1967) Useless except to show what it does not consider

- $T_{\text{seq}} = sT_{\text{seq}} + (1 - s)T_{\text{seq}}$, with $s$ fraction not parallelizable.
- With $p$ proc., $T_{\text{par}}/T_{\text{seq}} \geq s + (1 - s)/p \geq s$. Speed-up $\leq 1/s$.

Simplistic view: correct formula in wrong model ($\sim$ *PRAM*).

- Memory, comm., dedicated cores, program scaling, algorithmics!
- No insight even in favorable case, e.g., 2 independent purely sequential threads. Better: degree of parallelism, critical path, ratio comm./computation, data locality (spatial & temporal), bandwidth.
- But: make sure single core perf. remains good in parallel implem.

Gustafson's law Different reasoning, "scaling" programs.

- $T_{\text{par}} = sT_{\text{par}} + (1 - s)T_{\text{par}}$, with $s$ fraction of time in seq. core.
- With $p$ proc., $T_{\text{seq}}/T_{\text{par}} \geq s + (1 - s)p$. Speed-up $\geq (1 - s)p$.

# Where are we? HPC, embedded computing, accelerators

**Performance increase** how much is due to frequency increase, architecture improvements, compilers, algorithm design?

**Today's complications** Multi-level parallelism, memory hierarchy, bandwidth issues, attached accelerators, evolution in programming models.

# Where are we? HPC, embedded computing, accelerators

Performance increase how much is due to frequency increase, architecture improvements, compilers, algorithm design?

Today's complications Multi-level parallelism, memory hierarchy, bandwidth issues, attached accelerators, evolution in programming models.

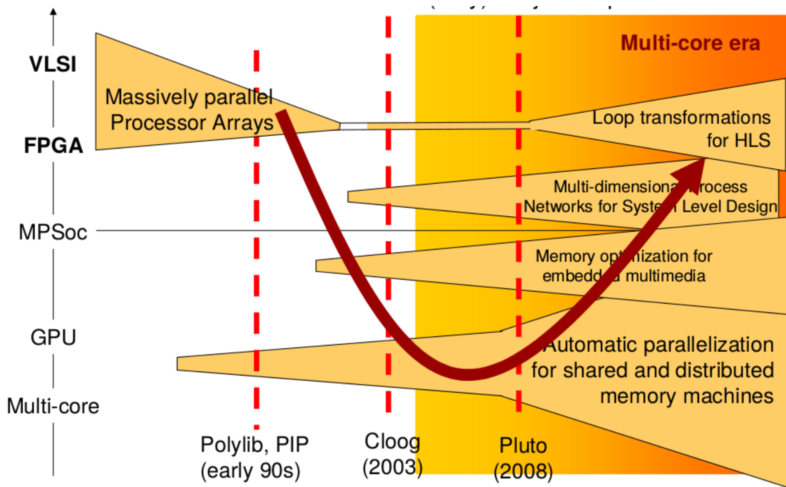
Compiler challenges

- PPPP Programmability, performance, portability, productivity.
- Virtualization New IRs, Just-in-time (JIT) compilation, runtimes.
- Maintenance Retargetable compilers, cleaner designs.
- Predictability Architecture design, worst-case execution time (WCET).
- Certification Correct-by-construction, verification, certified compilers.
- Languages Domain-specific languages, parallel languages.

☛ New expectations, new compiler issues, but we can solve more.
Automatic parallelization is irrealistic, but semi-automatic tools can help.

# POLYHEDRAL COMPILATION

# One view of history (borrowed from Steven Derrien)



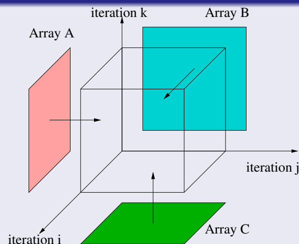

Some actors of this evolution are here:

P. Feautrier, P. Quinton, S. Rajopadhye, F. Irigoin, J. Ramanujam, A. Darte, A. Cohen, U. Bondhugula, S. Verdoolaege, T. Yuki, T. Grosser.

# Multi-dimensional affine representation of loops and arrays

## Matrix Multiply

```
int i,j,k;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
S:    C[i][j] = 0;
      for(k = 0; k < n; k++) {
T:       C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```




Instance-wise, element-wise, symbolic, parametric

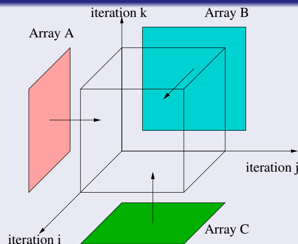
## Polyhedral Description        Omega/ISCC syntax

```
Domain := [n]->{S[i,j]: 0<=i,j<n; T[i,j,k]: 0<=i,j,k<n};

Read := [n]->{T[i,j,k]->A[i,k]; T[i,j,k]->B[k,j]; T[i,j,k]->C[i,j]};

Write := [n]->{S[i,j]->C[i,j]; T[i,j,k]->C[i,j]};

Order := [n]->{S[i,j]->[i,j,0]; T[i,j,k]->[i,j,1,k]};
```

# Multi-dimensional affine representation of loops and arrays

## Matrix Multiply

```
int i,j,k;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
S:     C[i][j] = 0;
       for(k = 0; k < n; k++) {
T:         C[i][j] += A[i][k] * B[k][j];
       }
   }
}
```




Instance-wise, element-wise, symbolic, parametric

## Polyhedral Description    Omega/ISCC syntax    So, that's it?

```
Domain := [n]->{S[i,j]: 0<=i,j<n; T[i,j,k]: 0<=i,j,k<n};

Read := [n]->{T[i,j,k]->A[i,k]; T[i,j,k]->B[k,j]; T[i,j,k]->C[i,j]};

Write := [n]->{S[i,j]->C[i,j]; T[i,j,k]->C[i,j]};

Order := [n]->{S[i,j]->[i,j,0]; T[i,j,k]->[i,j,1,k]};
```

# Ex: PPCG code for CPU+GPU, GPU part

```
__global__ void kernel0(float *A, float *B, float *C, int n) /* n=12288 */
{
    int b0 = blockIdx.y, b1 = blockIdx.x; /* Grid: 192x192 blocks, each with 32x32 threads */
    int t0 = threadIdx.y, t1 = threadIdx.x; /* Loops: 384x384x768 tiles, each with 32x32x16 points */
    __shared__ float shared_A[32][16]; /* Thus 1 block = 2x2x768 tiles, 1 thread = 1x1x16 points */
    __shared__ float shared_B[16][32];
    float private_C[1][1];

    for (int g1 = 32 * b0; g1 <= 12256; g1 += 6144) /* 6144 = 32 (tile size) x 192 (number of blocks) */
      for (int g3 = 32 * b1; g3 <= 12256; g3 += 6144) {
        private_C[0][0] = C[(t0 + g1) * 12288 + (t1 + g3)];
        for (int g9 = 0; g9 <= 12272; g9 += 16) { /* 16 consecutive points along k in a thread */
          if (t0 <= 15) /* 32x32 threads, only 16x32 do the transfer */
            shared_B[t0][t1] = B[(t0 + g9) * 12288 + (t1 + g3)];
          if (t1 <= 15) /* 32x32 threads, only 32x16 do the transfer */
            shared_A[t0][t1] = A[(t0 + g1) * 12288 + (t1 + g9)];
          __syncthreads();
          for (int c4 = 0; c4 <= 15; c4 += 1) /* compute the 16 consecutive points along k */
            private_C[0][0] += (shared_A[t0][c4] * shared_B[c4][t1]);
          __syncthreads();
        }
        C[(t0 + g1) * 12288 + (t1 + g3)] = private_C[0][0];
        __syncthreads();
      }
}
```

PPCG compiler (Parkas)
Verdoolaege, Cohen, etc.

# Ex: PPCG code for CPU+GPU, GPU part (Volkov-like)

```
__global__ void kernel0(float *A, float *B, float *C, int n) /* n=12288 */
{
    int b0 = blockIdx.y, b1 = blockIdx.x; /* Grid: 192x192 blocks, each with 16x16 threads */
    int t0 = threadIdx.y, t1 = threadIdx.x; /* Loops: 384x384x768 tiles, each with 32x32x16 points */
    __shared__ float shared_A[32][16]; /* Thus 1 block = 2x2x768 tiles, 1 thread = 2x2x16 points */
    __shared__ float shared_B[16][32];
    float private_C[2][2];

    for (int g1 = 32 * b0; g1 <= 12256; g1 += 6144) /* 6144 = 32 (tile size) x 192 (number of blocks) */
      for (int g3 = 32 * b1; g3 <= 12256; g3 += 6144) {
        private_C[0][0] = C[(t0 + g1) * 12288 + (t1 + g3)]; /* 2x2 points unrolled for register usage */
        private_C[0][1] = C[(t0 + g1) * 12288 + (t1 + g3 + 16)];
        private_C[1][0] = C[(t0 + g1 + 16) * 12288 + (t1 + g3)];
        private_C[1][1] = C[(t0 + g1 + 16) * 12288 + (t1 + g3 + 16)];
        for (int g9 = 0; g9 <= 12272; g9 += 16) { /* 16 consecutive points along k in a thread */
          for (int c1 = t1; c1 <= 31; c1 += 16) /* 16x32 to bring with 16x16 threads */
            shared_B[t0][c1] = B[(t0 + g9) * 12288 + (g3 + c1)];
          for (int c0 = t0; c0 <= 31; c0 += 16) /* 32x16 to bring with 16x16 threads */
            shared_A[c0][t1] = A[(g1 + c0) * 12288 + (t1 + g9)];
          __syncthreads();
          for (int c2 = 0; c2 <= 15; c2 += 1) { /* unrolled for register usage */
            private_C[0][0] += (shared_A[t0][c2] * shared_B[c2][t1]);
            private_C[0][1] += (shared_A[t0][c2] * shared_B[c2][t1 + 16]);
            private_C[1][0] += (shared_A[t0 + 16][c2] * shared_B[c2][t1]);
            private_C[1][1] += (shared_A[t0 + 16][c2] * shared_B[c2][t1 + 16]);
          }
          __syncthreads();
        }
        C[(t0 + g1) * 12288 + (t1 + g3)] = private_C[0][0];
        C[(t0 + g1) * 12288 + (t1 + g3 + 16)] = private_C[0][1];
        C[(t0 + g1 + 16) * 12288 + (t1 + g3)] = private_C[1][0];
        C[(t0 + g1 + 16) * 12288 + (t1 + g3 + 16)] = private_C[1][1];
        __syncthreads();
      }
```

PPCG compiler (Parkas)
Verdoolaege, Cohen, etc.

# Typical criticism against polyhedral techniques

Too hard to understand

Heavy formalism

No such codes

Not worth the effort

# Typical criticism against polyhedral techniques

Too hard to understand

➢ Easier now with demonstrators, compilers (e.g., PIPS, Pluto, PPCG, LLVM support), tutorials, and schools.

Heavy formalism

No such codes

Not worth the effort

# Typical criticism against polyhedral techniques

Too hard to understand

➢ Easier now with demonstrators, compilers (e.g., PIPS, Pluto, PPCG, LLVM support), tutorials, and schools.

Heavy formalism

➢ Easier now with tools for manipulating integer set relations (e.g., Cloog, ISCC, ISL, Barvinok). No need to "understand".

No such codes

Not worth the effort

# Typical criticism against polyhedral techniques

Too hard to understand

➢ Easier now with demonstrators, compilers (e.g., PIPS, Pluto, PPCG, LLVM support), tutorials, and schools.

Heavy formalism

➢ Easier now with tools for manipulating integer set relations (e.g., Cloog, ISCC, ISL, Barvinok). No need to “understand”.

No such codes

➢ Not true for specific domains/architectures. But yes, extensions needed (e.g., approx., reconstruction, summaries, while loops).

Not worth the effort

# Typical criticism against polyhedral techniques

Too hard to understand

➢ Easier now with demonstrators, compilers (e.g., PIPS, Pluto, PPCG, LLVM support), tutorials, and schools.

Heavy formalism

➢ Easier now with tools for manipulating integer set relations (e.g., Cloog, ISCC, ISL, Barvinok). No need to "understand".

No such codes

➢ Not true for specific domains/architectures. But yes, extensions needed (e.g., approx., reconstruction, summaries, while loops).

Not worth the effort

➢ Often the right limit for automation. Key for many analyses & optimizations, not just loop transformations. Growing industrial interest.

# Typical criticism against polyhedral techniques

Too hard to understand

➢ Easier now with demonstrators, compilers (e.g., PIPS, Pluto, PPCG, LLVM support), tutorials, and schools.

Heavy formalism

➢ Easier now with tools for manipulating integer set relations (e.g., Cloog, ISCC, ISL, Barvinok). No need to "understand".

No such codes

➢ Not true for specific domains/architectures. But yes, extensions needed (e.g., approx., reconstruction, summaries, while loops).

Not worth the effort

➢ Often the right limit for automation. Key for many analyses & optimizations, not just loop transformations. Growing industrial interest.

☛ Key to reason on multi-dimensional computations and data structures, and avoid explicit unrolling. More applications to come (e.g., loop termination, parallel languages, verification/certification, WCET).

# (Parametric) analysis, transformations, optimizations

## Loop transformations

- Automatic parallelization.
- Transformations framework.
- Scanning & code generation.
- Dynamic & speculative opt.

## Mapping computations & data

- Systolic arrays design.
- Data distribution.
- Communication opt.
- Streams optimizations.

## Instance/element-wise analysis

- Single assignment.
- Liveness, array expansion/reuse.
- Analysis of parallel programs.
- Data races & deadlocks detection.

## Counting, (de-)linearizing

- Cache misses.
- FIFOs, array linearizations.
- Memory size computations.
- Loop termination (e.g., WCET).

and many more. . .

# Related polyhedral prototypes, libraries, compilers

**Tools**

- ✓ PIP: parametric (I)LP.
- ✓ Polylib: polyhedra, generators.
- ✓ Omega, isl/iscc: integer sets, Presburger.
- ✓ Ehrhart & Barvinok: counting.
- ✓ Cloog: code generation.
- ✓ Fada/Candl: dataflow analysis.
- ✓ Cl@k: critical lattices and array contraction.
- ✓ Clan: polyhedral "extractor".
- ✓ Clint: visualization.

. . .

**Compiler or infrastructures**

- ✓ Alpha: SAREs to HLS.
- ✓ Compaan: polyhedral streams.
- ✓ Pips, Par4All, dHPF, Pluto, & R-Stream: parallelizing compilers.
- ✓ Graphite: library for GCC.
- ✓ Polly: library for LLVM.
- ✓ Gecos: user-friendly tool for HLS.
- ✓ PiCo, Chuba, PolyOpt: HLS compilers/prototypes.
- ✓ PPCG: code generator for GPU.
- ✓ Apollo: static/dynamic opt.

. . .

# Basic form: affine bounds and array access functions

Fortran DO loops:

Affine bounds of surrounding counters & parameters.

```
DO i=1, N
  DO j=1, N
    S: a(i,j) = c(i,j-1)
    T: c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

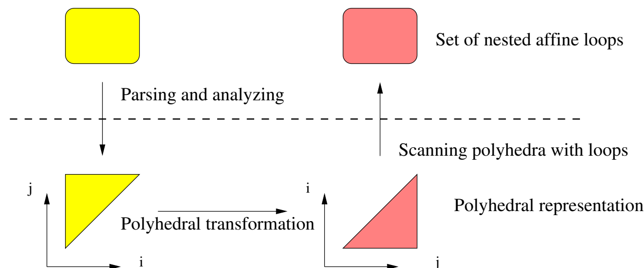
- Multi-dimensional arrays, same restriction for access functions.
- Loop increment $= 1$.
- Iteration domain: polyhedron.
- Iteration vector $(i, j)$.
- Lexico. order: $S(i, j) \to (i, j, 0)$, $T(i, j) \to (i, j, 1)$.

# Basic form: affine bounds and array access functions

**Fortran DO loops:**
Affine bounds of surrounding counters & parameters.

```
DO i=1, N
  DO j=1, N
    S: a(i,j) = c(i,j-1)
    T: c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

- Multi-dimensional arrays, same restriction for access functions.
- Loop increment = 1.
- Iteration domain: polyhedron.
- Iteration vector $(i, j)$.
- Lexico. order: $S(i,j) \to (i,j,0)$, $T(i,j) \to (i,j,1)$.




Same as reordering summations:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} u_{i,j} = \sum_{j=1}^{n}\sum_{i=1}^{j} u_{i,j}$$

# Why loops? Repetitive structures, hot spots

**DO loops**: few lines for a (possibly) large set of regular computations.

- Smaller code size.
- Repetitive structure: regularity that can be exploited.
- Hot parts of codes where optimizations are needed.
- Large potential for optimizations (e.g., parallelism, memory usage).
- Algorithms complexity depends on code size, not computations size.
- Parametric loops, needed for "optimality" w.r.t. unroll version.
- Abstraction between low-level (hardware) & high-level (program).

**Arrays**: similar properties for multi-dimensional storage.

☛ To be exploited: structure, parameters, symbolic unrolling.

# Be careful: different types of codes with loops

## Fortran DO loops:

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

## C for loops:

```
for (i=1; i<=N; i++) {
  for (j=1; j<=N; j++) {
    a[i][j] = c[i][j-1];
    c[i][j] = a[i][j] + a[i-1][N];
  }
}
```

## C for and while loops:

```
y = 0; x = 0;
while (x <= N && y <= N) {
  if (?) {
    x=x+1;
    if (y >= 0 && ?) y=y-1;
  }
  y=y+1;
}
```

## Uniform recurrence equations

$\forall(i,j)$ such that $1 \leq i,j \leq N$

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$

# More types of codes with loops

## FAUST: real-time for music

```
random = +(12345) ~ *(1103515245);
noise = random/2147483647.0;
process = random/2 : @(10);
```

$$\Leftrightarrow \begin{cases} R(t) = 12345 + 1103515245 \times R(t-1) \\ N(t) = R(t)/2147483647.0; \\ P(t) = 0.5 \times R(t-10) \end{cases}$$

## Array languages

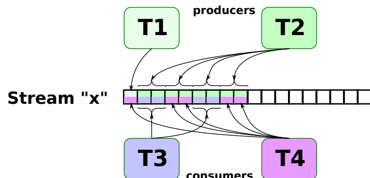
```
A = B + C
A[1:n] = A[0:n-1] + A[2:n+1]
```

## X10 parallel language

```
finish
  for (i in 0..n-1){
    S1;
    async S2;
  }
```

## OpenStream

```
#pragma omp task output (x) // Task T1
x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x « window_a[2]) // Task T2
  window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x » window_b[2]) // Task T3
    use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x) // Task T4
  use (x);
}
```



# Some fundamental mathematical tools

Linear programming, integer linear programming, parametric optimization

$\text{minlex}\{x \mid Ax \geq Bn + c,\, x \geq 0\}$

# Some fundamental mathematical tools

Linear programming, integer linear programming, parametric optimization

$\text{minlex}\{x \mid Ax \geq Bn + c,\ x \geq 0\}$

If non-empty sets, duality theorem

$\min\{c.x \mid Ax \geq b,\ x \geq 0\} = \max\{y.b \mid yA \leq c,\ y \geq 0\}$

# Some fundamental mathematical tools

Linear programming, integer linear programming, parametric optimization

$\text{minlex}\{x \mid Ax \geq Bn + c,\ x \geq 0\}$

If non-empty sets, duality theorem and affine form of Farkas lemma

$\min\{c.x \mid Ax \geq b,\ x \geq 0\} = \max\{y.b \mid yA \leq c,\ y \geq 0\}$

$c.x \leq d\ \forall x \text{ s.t. } Ax \leq b \Leftrightarrow \exists y \geq 0 \text{ s.t. } c = yA \text{ and } y.b \leq d$

# Some fundamental mathematical tools

Linear programming, integer linear programming, parametric optimization

$\text{minlex}\{x \mid Ax \geq Bn + c,\ x \geq 0\}$

If non-empty sets, duality theorem and affine form of Farkas lemma

$\min\{c.x \mid Ax \geq b,\ x \geq 0\} = \max\{y.b \mid yA \leq c,\ y \geq 0\}$

$c.x \leq d\ \forall x \text{ s.t. } Ax \leq b \Leftrightarrow \exists y \geq 0 \text{ s.t. } c = yA \text{ and } y.b \leq d$

Lattices, systems of Diophantine equations, Hermite and Smith forms

Lattice: $\mathcal{L} = \{x \mid \exists y \in \mathbb{Z}^n \text{ s.t. } x = Ay\}$

Hermite: $A = QH$, $Q$ unimodular, $H$ triangular.

Smith: $A = Q_1 S Q_2$, $Q_1$ and $Q_2$ unimodular, and $S$ diagonal.

# Some fundamental mathematical tools

Linear programming, integer linear programming, parametric optimization

$\text{minlex}\{x \mid Ax \geq Bn + c, x \geq 0\}$

If non-empty sets, duality theorem and affine form of Farkas lemma

$\min\{c.x \mid Ax \geq b, x \geq 0\} = \max\{y.b \mid yA \leq c, y \geq 0\}$

$c.x \leq d \ \forall x \text{ s.t. } Ax \leq b \Leftrightarrow \exists y \geq 0 \text{ s.t. } c = yA \text{ and } y.b \leq d$

Lattices, systems of Diophantine equations, Hermite and Smith forms

Lattice: $\mathcal{L} = \{x \mid \exists y \in \mathbb{Z}^n \text{ s.t. } x = Ay\}$

Hermite: $A = QH$, $Q$ unimodular, $H$ triangular.

Smith: $A = Q_1 S Q_2$, $Q_1$ and $Q_2$ unimodular, and $S$ diagonal.

Manipulation of integer sets, Presburger arithmetic, counting

☛ Chernikova, Ehrhart, Barvinok, quasi-polynomials

# Some fundamental mathematical tools

Linear programming, integer linear programming, parametric optimization

$\text{minlex}\{x \mid Ax \geq Bn + c,\ x \geq 0\}$

If non-empty sets, duality theorem and affine form of Farkas lemma

$\min\{c.x \mid Ax \geq b,\ x \geq 0\} = \max\{y.b \mid yA \leq c,\ y \geq 0\}$

$c.x \leq d\ \forall x \text{ s.t. } Ax \leq b \Leftrightarrow \exists y \geq 0 \text{ s.t. } c = yA \text{ and } y.b \leq d$

Lattices, systems of Diophantine equations, Hermite and Smith forms

Lattice: $\mathcal{L} = \{x \mid \exists y \in \mathbb{Z}^n \text{ s.t. } x = Ay\}$

Hermite: $A = QH$, $Q$ unimodular, $H$ triangular.

Smith: $A = Q_1 S Q_2$, $Q_1$ and $Q_2$ unimodular, and $S$ diagonal.

Manipulation of integer sets, Presburger arithmetic, counting

☛ Chernikova, Ehrhart, Barvinok, quasi-polynomials

In many cases, no need to really understand the theory anymore ☛ ISL

# $\min\{c.x \mid Ax \geq b,\ x \geq 0\} = \max\{y.b \mid yA \leq c,\ y \geq 0\}$

$$\min \left\{ \begin{array}{l} 11t + 10u \mid \\ \quad 2t + 3u \geq 5 \\ \quad 3t + 2u \geq 4 \\ \quad 5t + u \geq 12 \\ \quad t \geq 0,\ u \geq 0 \end{array} \right. = \max \left\{ \begin{array}{l} 5x + 4y + 12z \mid \\ \quad 2x + 3y + 5z \leq 11 \\ \quad 3x + 2y + z \leq 10 \\ \quad x \geq 0,\ y \geq 0,\ z \geq 0 \end{array} \right.$$

Primal problem: doped athlete buy the right numbers $t$ and $u$ of doping pills (with unit price 11 and 10) to get a sufficient intake (5, 4, and 12) of 3 elementary products, knowing the content of each mixed product.

Dual problem: dealer sell the 3 elementary products at maximal price, while being cheaper than doping pills.

# $\min\{c.x \mid Ax \geq b,\ x \geq 0\} = \max\{y.b \mid yA \leq c,\ y \geq 0\}$

$$
\min \left\{ \begin{array}{l} 11t + 10u \mid \\ \quad 2t + 3u \geq 5 \\ \quad 3t + 2u \geq 4 \\ \quad 5t + u \geq 12 \\ \quad t \geq 0,\ u \geq 0 \end{array} \right. \quad = \quad \max \left\{ \begin{array}{l} 5x + 4y + 12z \mid \\ \quad 2x + 3y + 5z \leq 11 \\ \quad 3x + 2y + z \leq 10 \\ \quad x \geq 0,\ y \geq 0,\ z \geq 0 \end{array} \right.
$$

**Primal problem: doped athlete** buy the right numbers $t$ and $u$ of doping pills (with unit price 11 and 10) to get a sufficient intake (5, 4, and 12) of 3 elementary products, knowing the content of each mixed product.

**Dual problem: dealer** sell the 3 elementary products at maximal price, while being cheaper than doping pills.

**Complexity**

- Optimal rational solution: polynomial (L. Khachiyan).
- Optimal integer solution: NP-complete and inequality only.
- Simplex algorithm (fast in practice), basis reduction (Lenstra et al.), parametric linear programming (P. Feautrier).

# Example of Sven Verdoolaege's iscc script

See tutorial http://barvinok.gforge.inria.fr/tutorial.pdf. Example borrowed from http://compsys-tools.ens-lyon.fr/iscc/ (thanks to A. Isoard).

```
# void polynomial_product(int n, int *A, int *B, int *C) {
#     for(int k = 0; k < 2*n-1; k++)
# S:      C[k] = 0;
#     for(int i = 0; i < n; i++)
#         for(int j = 0; j < n; j++)
# T:          C[i+j] += A[i] * B[j];
# }

Domain := [n] -> {
    S[k] : k <= -2 + 2n and k >= 0;
    T[i, j] : i >= 0 and i <= -1 + n and j <= -1 + n and j >= 0;
};

Read := [n] -> {
    T[i, j] -> C[i + j]; T[i, j] -> B[j]; T[i, j] -> A[i];
} * Domain;

Write := [n] -> {
    S[k] -> C[k]; T[i, j] -> C[i + j];
} * Domain;

Schedule := [n] -> {
    T[i, j] -> [1, i, j]; S[k] -> [0, k, 0];
};
```

# Example of Sven Verdoolaege's iscc script (Cont'd)

```
Schedule := [n] -> {
    T[i, j] -> [1, i, j]; S[k] -> [0, k, 0];
};

### Happens-Before relation without syntactic sugar equivalent to:
# Before := Schedule << Schedule;

Lexico := { [i0,i1,i2] -> [j0,j1,j2] : i0 < j0 or (i0 = j0 and i1 < j1)
                                  or (i0 = j0 and i1 = j1 and i2 < j2) };

Before := Schedule . Lexico . (Schedule^-1)
print Before;
```

We get the strict sequential order of operations in the program:

```
[n] -> { S[k] -> T[i, j]; S[k] -> S[k'] : k' > k;
         T[i, j] -> T[i', j'] : i' > i; T[i, j] -> T[i, j'] : j' > j }
```

```
RaW := (Write . (Read^-1)) * Before;
print RaW;
```

We get the read-after-write memory-based data dependences:

```
[n] -> { S[k] -> T[i, k - i] : 0 <= k <= -2 + 2n and i >= 0 and
                               -n + k < i <= k and i < n;
         T[i, j] -> T[i', i + j - i'] : 0 <= i < n and 0 <= j < n and i' > i
                            and i' >= 0 and -n + i + j < i' <= i + j and i' < n }
```

# Some iscc features and syntax with sets and relations/maps

+, -, * union, difference, intersection. domain m, range m set from map.

m(s) apply map to set. m1.m2 join of maps.

s1 cross s2, m1 cross m2 cartesian product. deltas m set of differences.

coefficients s constraints for Farkas lemma.

lexmin s, lexmin m lexicographic minimum, same for max.

codegen s, codegen m scanning domain (following map m). Example:

```
codegen RaW;
```

```
for (int c0 = 0; c0 < n; c0 += 1)
  for (int c1 = 0; c1 < n; c1 += 1) {
    for (int c2 = max(0, -n + c0 + c1 + 1); c2 < c0; c2 += 1)
      T(c2, c0 + c1 - c2);
    S(c0 + c1);
  }
```

card s, card m Number of integer points in set or image. Example:

```
print card RaW;
[n] -> { S[k] -> ((-1 + 2 * n) - k) : n <= k <= -2 + 2n;
         S[k] -> (1 + k) : 0 <= k < n;
         T[i, j] -> ((-1 + n) - i) : i <= -2 + n and n - i <= j < n;
         T[i, j] -> j : i >= 0 and 0 < j < n - i }
```

# Courses in 2013 polyhedral spring school

See the thematic quarter on compilation http://labexcompilation.ens-lyon.fr/ for polyhedral spring school and keynotes on HPC languages.

S. Rajopadhye A view on history.

P. Feautrier "Basics" in terms of mathematical concepts.

L.-N. Pouchet Polyhedral loop transformations, scheduling.

S. Verdoolaege Integer sets & relations: high-level modeling and implem.

A. Miné Program invariants and abstract interpretation.

B. Creusillet Array region analysis and applications.

U. Bondhugula (+ A. Darte) Compil. for distr. memory, memory mapping.

Sadayappan (+ N. Vasilache) Polyhedral transf. for SIMD architectures.

# Courses in 2013 polyhedral spring school

See the thematic quarter on compilation `http://labexcompilation.ens-lyon.fr/` for polyhedral spring school and keynotes on HPC languages.

S. Rajopadhye A view on history.

P. Feautrier "Basics" in terms of mathematical concepts.

L.-N. Pouchet Polyhedral loop transformations, scheduling.

S. Verdoolaege Integer sets & relations: high-level modeling and implem.

A. Miné Program invariants and abstract interpretation.

B. Creusillet Array region analysis and applications.

U. Bondhugula (+ A. Darte) Compil. for distr. memory, memory mapping.

Sadayappan (+ N. Vasilache) Polyhedral transf. for SIMD architectures.

Missing topics: tiling, dependence analysis, induction variable recognition, array privatization, loop fusion, code generation, Ehrhart theory, locality optimizations, benchmarks, high-level synthesis, trace analysis, program equivalence, termination, extensions, pipelining, streams/offloading, . . .

# CLASSIC LOOP TRANSFORMATIONS

# Catalog of loop and array transformations

- Loop unrolling (by constant factor)
- Software pipelining
- Loop fusion/distribution
- Loop peeling/statement sinking
- loop shifting (retiming)
- Loop interchange
- Loop skewing (by constant factor)
- Loop reversal
- Unimodular transformation
- Affine transformation

- Strip mining (by parametric factor)
- Unroll-and-jam
- Loop tiling
- Scalar privatization/expansion
- Single assignment expansion
- Array unrolling
- Array padding
- Array linearization
- Array contraction
- Affine modulo allocation

# Partial and total loop unrolling

```
DO i=1, 10
  a(i) = b(i)
  d(i) = a(i-1)
ENDDO
```

Unrolling by 2 $\longrightarrow$

```
DO i=1, 10, 2
  a(i) = b(i)
  d(i) = a(i-1)
  a(i+1) = b(i+1)
  d(i+1) = a(i)
ENDDO
```

## Partial unrolling

- Replicates instructions to improve schedule & resource usage.
- Can be used for array scalarization.
- Increases code size and (indirectly) register usage.

## Total loop unrolling

- Flattens the loops and changes structure.

# Strip mining, loop coalescing

```
DO i=1, N
  a(i) = b(i) + c(i)
ENDDO
```

Strip mining
⟶
⟵
Loop linearization

```
DO I_s=1, N, s
  DO i=I_s, min(N, I_s+s-1)
    a(i) = b(i) + c(i)
  ENDDO
ENDDO
```

## Strip mining

- Performs parametric loop unrolling.
- Changes the structure (2D space).
- Creates blocks of computations.
- Can be used as a preliminary step for tiling.

## Loop linearization

- Can reduce the control of loops.
- Reduces the problem dimension.

# Software pipelining and modulo scheduling

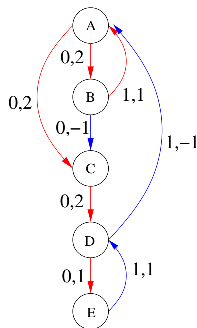
Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

Sequential code

```
L400:
   ld[r26] → r27
      nop
   add r27, 6740 → r26
   ld 0x1A54[r27] → r27
      nop
   sub.f r27, r25 → r0
   bne L400
      nop
L399:
```

$8n$ cycles.



# Software pipelining and modulo scheduling

Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

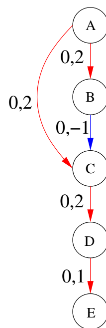
**Sequential code**

```
L400:
  ld[r26] → r27
    nop
  add r27, 6740 → r26
  ld 0x1A54[r27] → r27
    nop
  sub.f r27, r25 → r0
  bne L400
    nop
L399:
```

$8n$ cycles.

**Code compaction**

```
L400:
  ld[r26]→ r27
    nop
  add r27, 6740→ r26
  ld 0x1A54[r27]→ r27
    nop
  sub.f r27, r25→ r0
  bne L400
    nop
L399:
```



# Software pipelining and modulo scheduling

Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

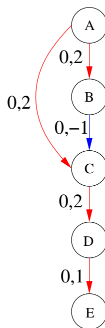
**Sequential code**

```
L400:
  ld[r26] → r27
    nop
  add r27, 6740 → r26
  ld 0x1A54[r27] → r27
    nop
  sub.f r27, r25 → r0
  bne L400
    nop
L399:
```

$8n$ cycles.

**Code compaction**

```
L400:
  ld[r26]→ r27
    nop
  ld 0x1A54[r27]→ r27
  add r27, 6740→ r26
    nop
  sub.f r27, r25→ r0
  bne L400
    nop
L399:
```



# Software pipelining and modulo scheduling

Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

**Sequential code**

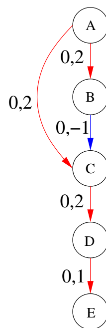
```
L400:
  ld[r26] → r27
    nop
  add r27, 6740 → r26
  ld 0x1A54[r27] → r27
    nop
  sub.f r27, r25 → r0
  bne L400
    nop
L399:
```

8$n$ cycles.

**Code compaction**

```
L400:
  ld[r26]→ r27
    nop
  ld 0x1A54[r27]→ r27
  add r27, 6740→ r26
  sub.f r27, r25→ r0
  bne L400
    nop
L399:
```

7$n$ cycles.



# Software pipelining and modulo scheduling

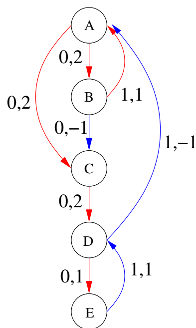
Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

**Sequential code**

```
L400:
   ld[r26] → r27
      nop
   add r27, 6740 → r26
   ld 0x1A54[r27] → r27
      nop
   sub.f r27, r25 → r0
   bne L400
      nop
L399:
```

8$n$ cycles.




**Software pipelining**

```
L400:
   ld[r26]→ r27
      nop
   add r27, 6740→ r26
   ld 0x1A54[r27], r27
      nop
   sub.f r27, r25→ r0
   bne L400
      nop
L399:
```

# Software pipelining and modulo scheduling

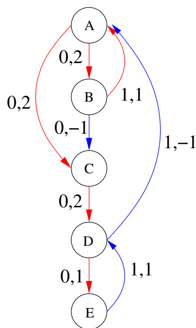
Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

**Sequential code**

```
L400:
   ld[r26] → r27
      nop
   add r27, 6740 → r26
   ld 0x1A54[r27] → r27
      nop
   sub.f r27, r25 → r0
   bne L400
      nop
L399:
```

$8n$ cycles.




**Software pipelining**

```
   ld[r26]→ r27
      nop
   add r27, 6740→ r26
L400:
   ld 0x1A54[r27]→ r27
      nop
   sub.f r27, r25→ r0
   bne L400
      nop
   ld[r26]→ r27
      nop
   add r27, 6740→ r26
L399:
```

# Software pipelining and modulo scheduling

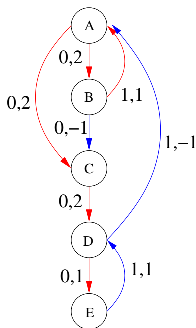
Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

**Sequential code**

```
L400:
  ld[r26] → r27
    nop
  add r27, 6740 → r26
  ld 0x1A54[r27] → r27
    nop
  sub.f r27, r25 → r0
  bne L400
    nop
L399:
```

$8n$ cycles.




**Software pipelining** + speculation

```
  ld[r26]→ r27
    nop
  add r27, 6740→ r26
L400:
  ld 0x1A54[r27]→ r27
  ld[r26]→ r27
  sub.f r27, r25→ r0
  bne L400
  add r27, 6740→ r26
L399:
```

$3 + 5n$ cycles.

# Software pipelining and modulo scheduling

Optimize throughput, with dependence & resource constraints.
Cyclic dependence graph with iteration distance (+ latency information).

Ex: sequential, pipelined LANai3.0, load & branch = one "shadow".

| Sequential code | Code compaction | Software pipelining + speculation |
|---|---|---|
| L400:<br>**ld**[r26] → r27<br>nop<br>**add** r27, 6740 → r26<br>**ld** 0x1A54[r27] → r27<br>nop<br>**sub.f** r27, r25 → r0<br>**bne** L400<br>nop<br>L399: | L400:<br>**ld**[r26]→ r27<br>nop<br>**ld** 0x1A54[r27]→ r27<br>**add** r27, 6740→ r26<br>**sub.f** r27, r25→ r0<br>**bne** L400<br>nop<br>L399: | **ld**[r26]→ r27<br>nop<br>**add** r27, 6740→ r26<br>L400:<br>**ld** 0x1A54[r27]→ r27<br>**ld**[r26]→ r27<br>**sub.f** r27, r25→ r0<br>**bne** L400<br>**add** r27, 6740→ r26<br>L399: |
| $8n$ cycles. | $7n$ cycles. | $3 + 5n$ cycles. |

$\sigma(S, i) = \lambda.i + \rho_S = \lambda.(i + q_S) + r_S$ with $0 \leq r_S < \lambda$ i.e., $r_S = \rho_S \bmod \lambda$

➠ Modulo scheduling: initiation interval (1/throughput). Here $\lambda = 5$.

# Loop shifting or retiming

```
DO i=1, N
  a(i) = b(i)
  d(i) = a(i-1)
ENDDO
```

Loop shifting $\longleftrightarrow$

```
DO i=0, N
  IF (i > 0) THEN
    a(i) = b(i)
  IF (i < N) THEN
    d(i+1) = a(i)
ENDDO
```

Here: dependence at distance 1 (loop-carried or inter-iteration) transformed into dependence at distance 0 (loop-independent or intra-iteration).

## Main features

- Similar to software pipelining.
- Creates prelude/postlude or introduces `if` statements.
- Can be used to align accesses and enable loop fusion.
- Particularly suitable to handle constant dependence distances.

# Loop peeling and statement sinking

```
DO i=0, N
  IF (i > 0) THEN
    a(i) = b(i)
  IF (i < N) THEN
    d(i+1) = a(i)
ENDDO
```

Loop peeling

Statement sinking

```
d(1) = a(0)
DO i=1, N-1
  a(i) = b(i)
  d(i+1) = a(i)
ENDDO
a(N) = b(N)
```

## Loop peeling

- Removes a few iterations to make code simpler.
- May enable more transformations.
- Reduces the iteration domain (range of loop counter).

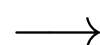
## Statement sinking

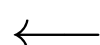
- Used to make loops perfectly nested.

# Loop distribution and loop fusion

```
DO i=1, N
  a(i) = b(i)
  d(i) = a(i-1)
ENDDO
```

Loop distribution





Loop fusion

```
DO i=1, N
  a(i) = b(i)
ENDDO
DO i=1, N
  d(i) = a(i-1)
ENDDO
```

Here: intra-loop dependence transformed into inter-loop dependence.

Loop distribution

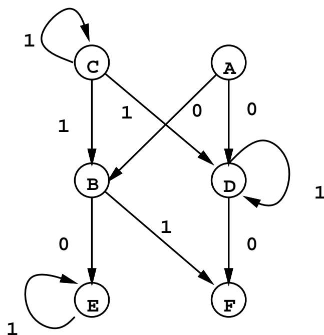
- Used to parallelize/vectorize loops (no inter-iteration dependence).
- Valid if statements not involved in a circuit of dependences.
- Parallelization: separate strongly connected components.

Loop fusion

- Increases the granularity of computations.
- Reduces loop overhead.
- Usually improves spatial & temporal data locality.
- May enable array scalarization.

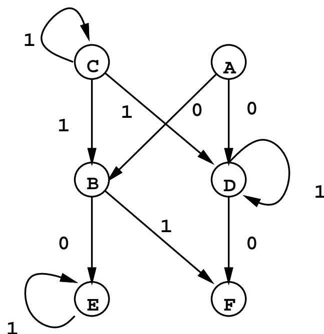
# Loop parallelization with loop distribution

```
DO i=1,N
   A(i) = 2*A(i) + 1
   B(i) = C(i-1) + A(i)
   C(i) = C(i-1) + G(i)
   D(i) = D(i-1) + A(i) + C(i-1)
   E(i) = E(i-1) + B(i)
   F(i) = D(i) + B(i-1)
ENDDO
```



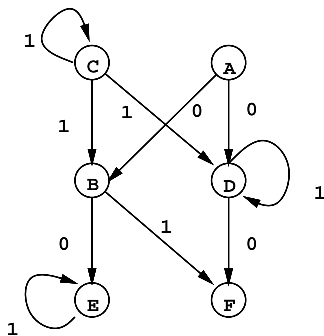
# Loop parallelization with loop distribution

```
DOPAR i=1,N
  A(i) = 2*A(i) + 1
ENDDOPAR
DOSEQ i=1,N
  C(i) = C(i-1) + G(i)
ENDDOSEQ
DOPAR i=1,N
  B(i) = C(i-1) + A(i)
ENDDOPAR
DOSEQ i=1,N
  E(i) = E(i-1) + B(i)
ENDDOSEQ
DOSEQ i=1,N
  D(i) = D(i-1) + A(i) + C(i-1)
ENDDOSEQ
DOPAR i=1,N
  F(i) = D(i) + B(i-1)
ENDDOPAR
```



# Loop parallelization with partial loop distribution

```
DOSEQ i=1,N
   C(i) = C(i-1) + G(i)
ENDDOSEQ
DOPAR i=1,N
   A(i) = 2*A(i) + 1
   B(i) = C(i-1) + A(i)
ENDDOPAR
DOSEQ i=1,N
   D(i) = D(i-1) + A(i) + C(i-1)
   E(i) = E(i-1) + B(i)
ENDDOSEQ
DOPAR i=1,N
   F(i) = D(i) + B(i-1)
ENDDOPAR
```



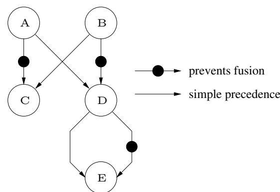

Instance of typed loop fusion, with 2 types (par. & seq.), and possibly fusion-preventing edges for 1 type (par.). ☛ NP-complete.

# Loop shifting and loop parallelization

## Maximal typed fusion

Easy for 1 type, NP-complete for $\geq 2$ types.

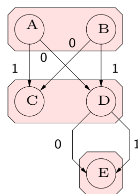
```
DO i=2, n
  a(i) = f(i)
  b(i) = g(i)
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
  e(i) = d(i-1) + d(i)
ENDDO
```




```
DOPAR i=2, n
  a(i) = f(i)
  b(i) = g(i)
ENDDOPAR
DOPAR i=2, n
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
ENDDOPAR
DOPAR i=2, n
  e(i) = d(i-1) + d(i)
ENDDO
```

## Same problem with loop shifting?

```
DO i=2, n
  a(i) = f(i)
  b(i) = g(i)
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
  e(i) = d(i-1) + d(i)
ENDDO
```

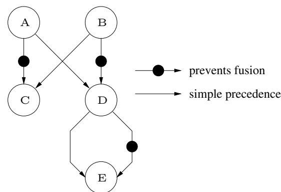



```
DOPAR i=2, n
  a(i) = f(i)
  b(i) = g(i)
ENDDOPAR
DOPAR i=2, n
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
ENDDOPAR
DOPAR i=2, n
  e(i) = d(i-1) + d(i)
ENDDO
```

# Loop shifting and loop parallelization

## Maximal typed fusion Easy for 1 type, NP-complete for $\geq$ 2 types.

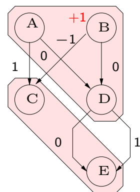
```
DO i=2, n
  a(i) = f(i)
  b(i) = g(i)
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
  e(i) = d(i-1) + d(i)
ENDDO
```




```
DOPAR i=2, n
  a(i) = f(i)
  b(i) = g(i)
ENDDOPAR
DOPAR i=2, n
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
ENDDOPAR
DOPAR i=2, n
  e(i) = d(i-1) + d(i)
ENDDO
```

## Same problem with loop shifting? NP-complete even for 1 type.

```
DO i=2, n
  a(i) = f(i)
  b(i) = g(i)
  c(i) = a(i-1) + b(i)
  d(i) = a(i) + b(i-1)
  e(i) = d(i-1) + d(i)
ENDDO
```




```
a(2) = f(2)
d(2) = a(2) + b(1)
DOPAR i=3, n
  a(i) = f(i)
  b(i-1) = g(i-1)
  d(i) = a(i) + b(i-1)
ENDDOPAR
b(n) = g(n)
DOPAR i=2, n
  c(i) = a(i-1) + b(i)
  e(i) = d(i-1) + d(i)
ENDDOPAR
```

# A “true” multi-dimensional shifting problem

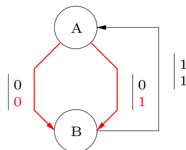
## In dimension one

- Maximal fusion with shifting: NP-complete (previous slide).
- Complete fusion? Easy. Iff all (undirected) cycles have weight 0.

## In dimension two

- Find an outer shift to enable the complete inner fusion.

```
DO i=1, n-1
  DO j=1, n-1
    a(i,j) = b(i-1,j-1)
    b(i,j) = a(i,j) + a(i,j-1)
  ENDDO
ENDDO
```




```
DO i=1, n-1
  DOPAR j=1, n-1
    a(i,j) = b(i-1,j-1)
  ENDDOPAR
  DOPAR j=1, n-1
    b(i,j) = a(i,j) + a(i,j-1)
  ENDDOPAR
ENDDO
```

# A "true" multi-dimensional shifting problem

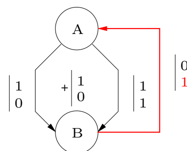
## In dimension one

- Maximal fusion with shifting: NP-complete (previous slide).
- Complete fusion? Easy. Iff all (undirected) cycles have weight 0.

## In dimension two

- Find an outer shift to enable the complete inner fusion.

```
DO i=1, n-1
  DO j=1, n-1
    a(i,j) = b(i-1,j-1)
    b(i,j) = a(i,j) + a(i,j-1)
  ENDDO
ENDDO
```




```
DO i=1, n
  DO j=1, n-1
    IF (i<n) THEN
      a(i,j) = b(i-1,j-1)
    IF (i>1) THEN
      b(i-1,j) = a(i-1,j) + a(i-1,j-1)
  ENDDO
ENDDO
```

# A "true" multi-dimensional shifting problem

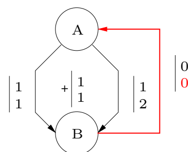
## In dimension one

- Maximal fusion with shifting: NP-complete (previous slide).
- Complete fusion? Easy. Iff all (undirected) cycles have weight 0.

## In dimension two

- Find an outer shift to enable the complete inner fusion.
- NP-complete (as many other retiming problems).

```
DO i=1, n-1
  DO j=1, n-1
    a(i,j) = b(i-1,j-1)
    b(i,j) = a(i,j) + a(i,j-1)
  ENDDO
ENDDO
```




```
DO i=1, n
  DOPAR j=1, n
    IF (i>1) and (j>1) THEN
      b(i-1,j-1) = a(i-1,j-1) + a(i-1,j-2)
    IF (i<n) and (j<n) THEN
      a(i,j) = b(i-1,j-1)
  ENDDOPAR
ENDDO
```

☛ These are very particular problems & instances, but still, this shows the difficulty to "push" dependences inside, i.e., to optimize locality.

# Parallelism detection: more loop transformations needed

Is there some loop parallelism (i.e., parallel loop iterations) in the following two codes? What is their degree of parallelism?

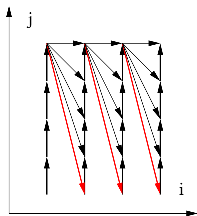
```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,j)
  ENDDO
ENDDO
```

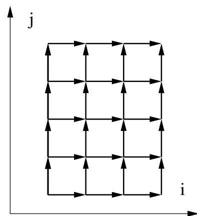
# Parallelism detection: more loop transformations needed

Is there some loop parallelism (i.e., parallel loop iterations) in the following two codes? What is their degree of parallelism?

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```




```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,j)
  ENDDO
ENDDO
```



# Loop interchange

Loop interchange: $(i, j) \mapsto (j, i)$.

```
DO i=1, N
  DO j=1, i
    a(i,j+1) = a(i,j) + 1
  ENDDO
ENDDO
```

Loop interchange $\longleftrightarrow$

```
DO j=1, N
  DO i=j, N
    a(i,j+1) = a(i,j) + 1
  ENDDO
ENDDO
```

Here: dependence distance $(0, 1)$ transformed into distance $(1, 0)$.

Main features

- May involve bounds computations as in $\sum_{i=1}^{n}\sum_{j=1}^{i} S_{i,j} = \sum_{j=1}^{n}\sum_{i=j}^{n} S_{i,j}$.
- Can impact loop parallelism.
- Basis of loop tiling.
- Changes order of memory accesses and thus data locality.

# Unroll-and-jam

```
DO i=1, 2N
   DO j=1, M
      a(i,j) = ...
   ENDDO
ENDDO
```

Unroll-and-jam

$\longleftrightarrow$

```
DO i=0, 2N, 2
   DO j=0, M
      a(i,j) = ...
      a(i+1,j) = ...
   ENDDO
ENDDO
```

Interests:

- Combines outer loop unrolling and loop fusion.
- Changes order of iterations and locality, keeping same loop nesting.
- Can be viewed as a restricted form of tiling $s \times 1$.

# Loop reversal and loop skewing

Loop reversal: $i \mapsto -i$, loop executed in opposite order.

Loop skewing: $(i, j) \mapsto (i, j + i)$, loop iterations in the same order.

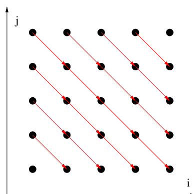
```
DO i=1, N
  DO j=1, N
    a(i,j-1) = a(i-1,j) + 1
  ENDDO
ENDDO
```

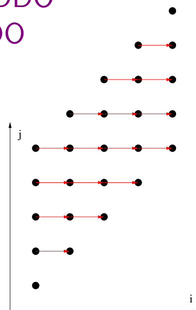
Skewing by 1 $\longrightarrow$

$\longleftarrow$ Skewing by $-1$

```
DO i=1, N
  DO j=1+i, N+i
    a(i,j-i-1) = a(i-1,j-i) + 1
  ENDDO
ENDDO
```

Dependence distance $(1, -1)$ transformed into distance $(1, 0)$.



# Unimodular transf.: reversal + skewing + interchange

```
DO i=1, N
  DO j=1, N
    a(i,j) = ...
  ENDDO
ENDDO
```

Unimodular $U$

$\longrightarrow$

$\longleftarrow$

Unimodular $U^{-1}$

```
DO t=2, 2N
  DO p=max(1,t-N), min(N,t-1)
    a(p,t-p) = ...
  ENDDO
ENDDO
```

Here, $(i,j) \mapsto (t,p) = (i+j,i)$. Loop bounds with Fourier-Motzkin elim.:

$$1 \leq i,j \leq N \Leftrightarrow 1 \leq p, t-p \leq N \Leftrightarrow 1 \leq p \leq N,\ t-N \leq p \leq t-1$$

Elimination of $p \Rightarrow 2 \leq t \leq 2N$, $1 \leq N$, Elimination of $t \Rightarrow 1 \leq N$

# Unimodular transf.: reversal + skewing + interchange

```
DO i=1, N
  DO j=1, N
    a(i,j) = ...
  ENDDO
ENDDO
```

Unimodular $U$ $\longrightarrow$

$\longleftarrow$ Unimodular $U^{-1}$

```
DO t=2, 2N
  DO p=max(1,t-N), min(N,t-1)
    a(p,t-p) = ...
  ENDDO
ENDDO
```

Here, $(i,j) \mapsto (t,p) = (i+j, i)$. Loop bounds with Fourier-Motzkin elim.:

$$1 \le i, j \le N \Leftrightarrow 1 \le p, t-p \le N \Leftrightarrow 1 \le p \le N,\ t-N \le p \le t-1$$

Elimination of $p \Rightarrow 2 \le t \le 2N$, $1 \le N$, Elimination of $t \Rightarrow 1 \le N$

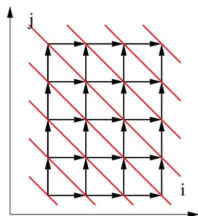
In general: “For all $\vec{i} \in \mathcal{P}$ do $S(\vec{i})$” $\longrightarrow$ “For all $\vec{p} \in U\mathcal{P}$ do $S(U^{-1}\vec{p})$”.

$$\begin{pmatrix} t \\ p \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix} \quad \begin{pmatrix} i \\ j \end{pmatrix} = U^{-1} \begin{pmatrix} t \\ p \end{pmatrix} \quad \text{Here} \quad U = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad U^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

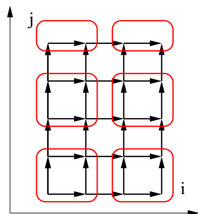
New implicit execution order: iterate lexicographically on $(t,p)$.
If $S(\vec{i})$ depends on $T(\vec{j})$, dep. distance $d = \vec{i} - \vec{j}$ lexico-positive: $\vec{d} \succeq_{\text{lex}} \vec{0}$.
New distance $\vec{d'} = U(\vec{i} - \vec{j}) = U\vec{d}$. Validity condition: $\vec{d'} = U\vec{d} \succeq_{\text{lex}} \vec{0}$.

# Loop tiling, blocked algorithms

```
DO t=2, 2N
  DOPAR j=max(1,t-N), min(N,t-1)
    a(t-j,j) = c(t-j,j-1)
    c(t-j,j) = a(t-j,j) + a(t-j-1,j)
  ENDDO
ENDDO
```




```
DO I=1, N, B
  DO J=1, N, B
    DO i=I, min(I+B-1,N)
      DO j=J, min(J+B-1,N)
        a(i,j) = c(i,j-1)
        c(i,j) = a(i,j) + a(i-1,j)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

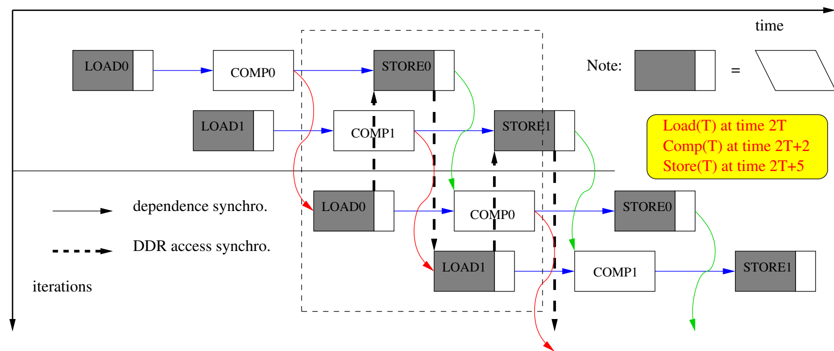



➽ Tiling and parallelism detection: similar problem.

# In practice, need to combine all. Ex: HLS with C2H Altera

Optimize DDR accesses for bandwidth-bound accelerators.

- Use tiling for data reuse and to enable burst communication.
- Use fine-grain software pipelining to pipeline DDR requests.
- Use double buffering to hide DDR latencies.
- Use coarse-grain software pipelining to hide computations.



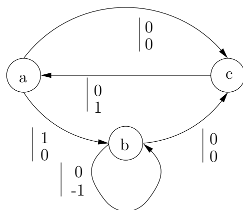
# SYSTEMS OF UNIFORM RECURRENCE EQUATIONS

# SURE: system of uniform recurrence equations (1967)

Karp, Miller, Winograd: "The organization of computations for uniform recurrence equations" (J. ACM, 14(3), pp. 563-590).

$$\forall \vec{p} \in \{\vec{p} = (i,j) \mid 1 \leq i,j \leq N\}$$

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$




- RDG (reduced dependence graph) $G = (V, E, \vec{w})$.
- EDG (expanded dep. graph): vertices $V \times \mathcal{P}$ = unrolled RDG.

Semantics:

- Explicit dependences, implicit schedule.
- Compute left-hand side first, unless not in $\mathcal{P}$ (input data).

# Two main problems: computability & scheduling

Computability (KMW view) A SURE is computable for all bounded domains P if and only if the RDG has no cycle $C$ with $\vec{w}(C) = \vec{0}$.

Scheduling (dual view): Lamport, Feautrier, etc. How to find an explicit schedule? With guaranteed "latency"?

Looking for an elementary cycle of zero weight: NP-complete.

Looking for a multi-cycle of zero weight: polynomial.

Looking for a cycle of zero weight: polynomial.

# Two main problems: computability & scheduling

Computability (KMW view) A SURE is computable for all bounded domains P if and only if the RDG has no cycle $C$ with $\vec{w}(C) = \vec{0}$.

Scheduling (dual view): Lamport, Feautrier, etc. How to find an explicit schedule? With guaranteed "latency"?

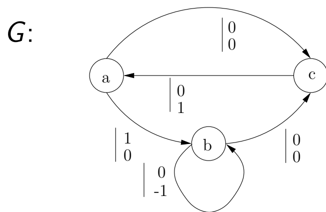
Looking for an elementary cycle of zero weight: NP-complete.

Looking for a multi-cycle of zero weight: polynomial.

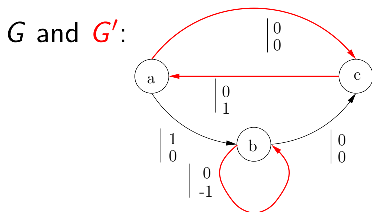
Looking for a cycle of zero weight: polynomial.

Key structure: the subgraph $G'$ induced by all edges that belong to a multi-cycle (i.e., union of cycles) of zero weight.

$G$:




$G$ and $G'$:



# Key properties for multi-dimensional decomposition

**Lemma (Look in $G'$)**

*A zero-weight cycle is a zero-weight multi-cycle.*

➽ *Look in the subgraph $G'$ only.*

Note: clear, but how to compute $G'$?

**Lemma (Look in each SCC)**

*A zero-weight cycle belongs to a strongly connected component.*

➽ *Look in each strongly connected component (SCC) separately.*

Note: this is the recursive step.

**Lemma (End of recursion)**

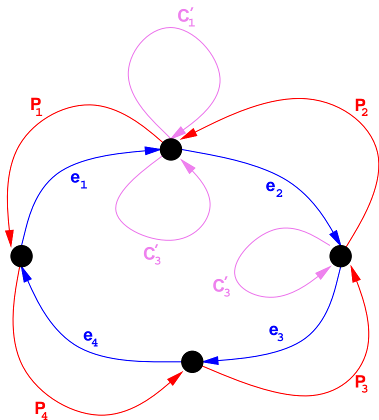
*If $G'$ is strongly connected, there is a zero-weight cycle.*

➽ *This stops the recursion.*

# Key properties for multi-dimensional decomposition

## Lemma (End of recursion)

*If $G'$ is strongly connected, there is a zero-weight cycle.*




- $\sum_i e_i$ cycle that visits all vertices.
- $e_i$ in multi-cycle $C_i$, with $\vec{w}(C_i) = \vec{0}$.
- $C_i = e_i + P_i + C'_i$.
- Follow the $e_i$, then the $P_i$ and, on the way, plug the $C'_i$.

# Karp, Miller, and Winograd's decomposition

Boolean KMW($G$):

- Build $G'$ the subgraph of zero-weight multicycles of $G$.
- Compute $G'_1, \ldots, G'_s$, the $s$ SCCs of $G'$.
  - If $s = 0$, $G'$ is empty, return TRUE.
  - If $s = 1$, $G'$ is strongly connected, return FALSE.
  - Otherwise return $\wedge_i$KMW($G'_i$) (logical AND).

Then, $G$ is computable iff KMW($G$) returns TRUE.

# Karp, Miller, and Winograd's decomposition

Boolean KMW($G$):

- Build $G'$ the subgraph of zero-weight multicycles of $G$.
- Compute $G'_1, \ldots, G'_s$, the $s$ SCCs of $G'$.
  - If $s = 0$, $G'$ is empty, return TRUE.
  - If $s = 1$, $G'$ is strongly connected, return FALSE.
  - Otherwise return $\wedge_i$KMW($G'_i$) (logical AND).

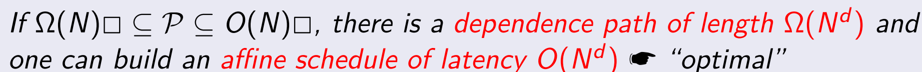
Then, $G$ is computable iff KMW($G$) returns TRUE.

Depth $d = 0$ if $G$ acyclic, $d = 1$ if all SCCs have an empty $G'$, etc.

**Theorem (Depth of the decomposition)**

*If $G$ is computable, $d \leq n$ (dimension of $\mathcal{P}$). Otherwise, $d \leq n + 1$.*

**Theorem (Optimal number of parallel loops)**



# But how to compute $G'$? Primal and dual programs.

$e \in G'$ iff $v_e = 0$ in any optimal solution of the linear program:

$$\min \left\{ \textstyle\sum_e v_e \;\middle|\; \vec{q} \geq \vec{0},\ \vec{v} \geq \vec{0},\ \vec{q} + \vec{v} \geq \vec{1},\ C\vec{q} = \vec{0},\ W\vec{q} = \vec{0} \right\}$$

Always interesting to take a look at the dual program:

$$\max \left\{ \textstyle\sum_e z_e \;\middle|\; \vec{0} \leq \vec{z} \leq \vec{1},\ \vec{X}.\vec{w}(e) + \rho_v - \rho_u \geq z_e,\ \forall e = (u, v) \in E \right\}$$

# But how to compute $G'$? Primal and dual programs.

$e \in G'$ iff $v_e = 0$ in any optimal solution of the linear program:

$$\min \left\{ \sum_e v_e \;\middle|\; \vec{q} \geq \vec{0},\ \vec{v} \geq \vec{0},\ \vec{q} + \vec{v} \geq \vec{1},\ C\vec{q} = \vec{0},\ W\vec{q} = \vec{0} \right\}$$

Always interesting to take a look at the dual program:

$$\max \left\{ \sum_e z_e \;\middle|\; \vec{0} \leq \vec{z} \leq \vec{1},\ \vec{X}.\vec{w}(e) + \rho_v - \rho_u \geq z_e,\ \forall e = (u, v) \in E \right\}$$

☛ Generalizes modulo scheduling: $\sigma(u, \vec{p}) = \vec{X}.\vec{p} + \rho_u$ (in 1D: $\lambda.p + \rho_u$).

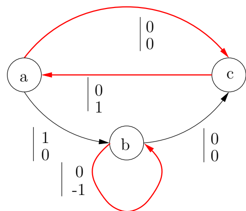
For any optimal solution:

- $e \notin G' \Leftrightarrow \vec{X}.\vec{w}(e) + \rho_v - \rho_u \geq 1$ ☛ loop carried.
- $e \in G' \Leftrightarrow \vec{X}.\vec{w}(e) + \rho_v - \rho_u = 0$ ☛ loop independent.

and keep going until all dependences become carried.

☛ Multi-dimensional scheduling and loop transformations.

# Scheduling/parallelization of illustrating example




$$\forall \vec{p} \in \{\vec{p} = (i,j) \mid 1 \le i,j \le N\}$$

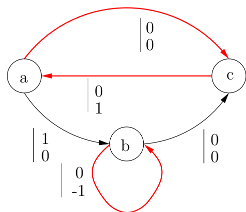
$$\left\{ \begin{array}{l} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{array} \right.$$

$$\left. \begin{array}{l} \vec{X_1}.(0,1) = 0 \\ \vec{X_1}.(1,0) \ge 2 \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \vec{X_1} = (2,0),\ \rho_a = 1 \\ \rho_b = 0,\ \rho_c = 1 \end{array} \right.$$

$$\text{Final schedule} \left\{ \begin{array}{l} \sigma_a(i,j) = (2i+1, 2j) \\ \sigma_b(i,j) = (2i, -j) \\ \sigma_c(i,j) = (2i+1, 2j+1) \end{array} \right.$$

```
DO i=1, N
   DO j=N, 1, -1
      b(i,j) = a(i-1,j) + b(i,j+1)
   ENDDO
   DO j=1, N
      a(i,j) = c(i,j-1)
      c(i,j) = a(i,j) + b(i,j)
   ENDDO
ENDDO
```

# Scheduling/parallelization of illustrating example




$$\forall \vec{p} \in \{\vec{p} = (i,j) \mid 1 \leq i,j \leq N\}$$

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$

$$\left.\begin{array}{l} \vec{X_1}.(0,1) = 0 \\ \vec{X_1}.(1,0) \geq 2 \end{array}\right\} \Rightarrow \begin{cases} \vec{X_1} = (2,0),\ \rho_a = 1 \\ \rho_b = 0,\ \rho_c = 1 \end{cases}$$

$$\text{Final schedule} \begin{cases} \sigma_a(i,j) = (2i+1, 2j) \\ \sigma_b(i,j) = (2i, -j) \\ \sigma_c(i,j) = (2i+1, 2j+1) \end{cases}$$

```
DO i=1, N
   DO j=N, 1, -1
      b(i,j) = a(i-1,j) + b(i,j+1)
   ENDDO
   DO j=1, N
      a(i,j) = c(i,j-1)
      c(i,j) = a(i,j) + b(i,j)
   ENDDO
ENDDO
```

☛ Today: most work based on Farkas lemma (affine form) following Feautrier (1992) & generalized tiling as Pluto (2008).

# DETECTION OF LOOP PARALLELISM

# Back to DO loops: dependence distances

Fortran DO loops:

- Explicit schedule
- Implicit dependences
- EDG: $S(\vec{i}) \Rightarrow T(\vec{j})$.
- RDG: $S \rightarrow T$.

```
DO i=1, N
  DO j=1, N
    a(i,j) = ...
    b(i,j) = a(j,i) + 1
  ENDDO
ENDDO
```

Pair set $R_{S,T} = \{(\vec{i},\vec{j}) \mid S(\vec{i}) \Rightarrow T(\vec{j})\}$

Affine dep. $\vec{i} = f(\vec{j})$ when possible.

Distances $E_{S,T} = \{(\vec{j} - \vec{i}) \mid S(\vec{i}) \Rightarrow T(\vec{j})\}$.

Over-approx. $\overline{E}_{S,T}$ s.t. $E_{S,T} \subseteq \overline{E}_{S,T}$:

- affine relation
- polyhedron: vertices, rays, lines
- direction vector: $\mathbb{Z}$, $+$, $-$, $*$
- dependence level: 1, 2, ..., $\infty$

# Back to DO loops: dependence distances

**Fortran DO loops:**

- Explicit schedule
- Implicit dependences
- EDG: $S(\vec{i}) \Rightarrow T(\vec{j})$.
- RDG: $S \to T$.

```
DO i=1, N
  DO j=1, N
    a(i,j) = ...
    b(i,j) = a(j,i) + 1
  ENDDO
ENDDO
```

**Pair set** $R_{S,T} = \{(\vec{i},\vec{j}) \mid S(\vec{i}) \Rightarrow T(\vec{j})\}$

**Affine dep.** $\vec{i} = f(\vec{j})$ when possible.

**Distances** $E_{S,T} = \{(\vec{j} - \vec{i}) \mid S(\vec{i}) \Rightarrow T(\vec{j})\}$.

Here: $E = \left\{ \begin{pmatrix} i-j \\ j-i \end{pmatrix} \middle| \; i-j \geq 1,\ 1 \leq i,\ j \leq N \right\}$

**Over-approx.** $\overline{E}_{S,T}$ s.t. $E_{S,T} \subseteq \overline{E}_{S,T}$:

- affine relation
- polyhedron: vertices, rays, lines
- direction vector: $\mathbb{Z}$, $+$, $-$, $*$
- dependence level: 1, 2, ..., $\infty$

**Polyhedral approximation:** $E' = \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ -1 \end{pmatrix} \middle| \; \lambda \geq 0 \right\}$

**Direction vector:** $E' = \begin{pmatrix} + \\ - \end{pmatrix} = \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ -1 \end{pmatrix} \middle| \; \lambda,\ \mu \geq 0 \right\}$

**Level:** $E' = ① = \begin{pmatrix} + \\ * \end{pmatrix} = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ 1 \end{pmatrix} \middle| \; \lambda \geq 0 \right\}$

# Uniformization of dependences: example

```
DO i=1, N
  DO j=1, N
    S: a(i,j) = c(i,j-1)
    T: c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

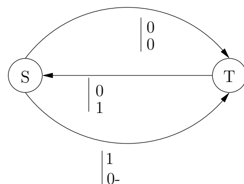
`S(i-1,N)` $\Rightarrow$ `T(i,j)`
Dep. distance $(1, j - N)$.

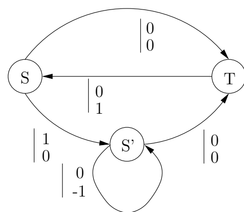
# Uniformization of dependences: example

```
DO i=1, N
  DO j=1, N
    S: a(i,j) = c(i,j-1)
    T: c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

S(i-1,N) $\Rightarrow$ T(i,j)
Dep. distance $(1, j-N)$.

Direction vector $(1, 0-) = (1, 0) + k(0, -1)$, $k \geq 0$ ☛ SURE! KMW!

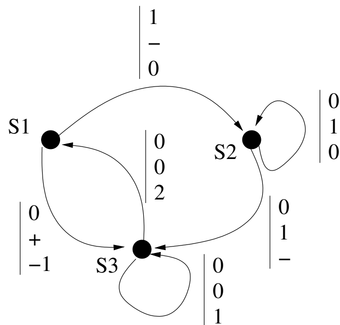


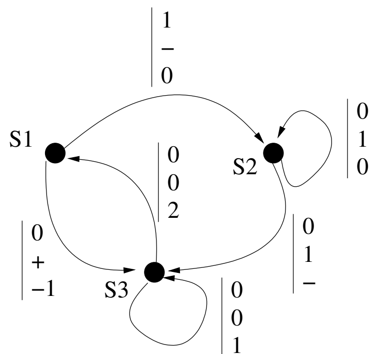

No parallelism ($d = 2$). Code appears purely sequential (and here it is).

# Second example with direction vectors

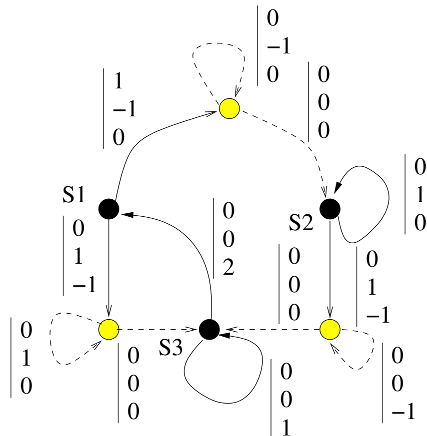
```
DO i= 1, N
  DO j = 1, N
    DO k = 1, j
      S1: a(i,j,k) = c(i,j,k-1) + 1
      S2: b(i,j,k) = a(i-1,j+i,k) + b(i,j-1,k)
      S3: c(i,j,k+1) = c(i,j,k) + b(i,j-1,k+i) + a(i,j-k,k+1)
    ENDDO
  ENDDO
ENDDO
```



# Second example: dependence graphs

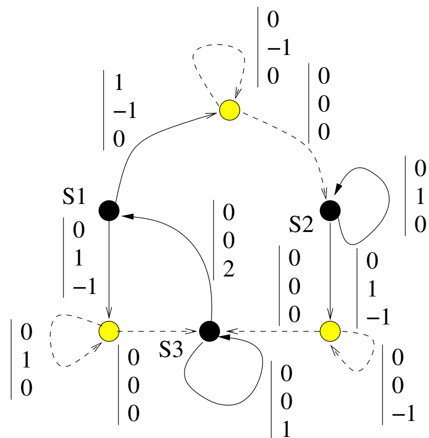



Initial RDG.

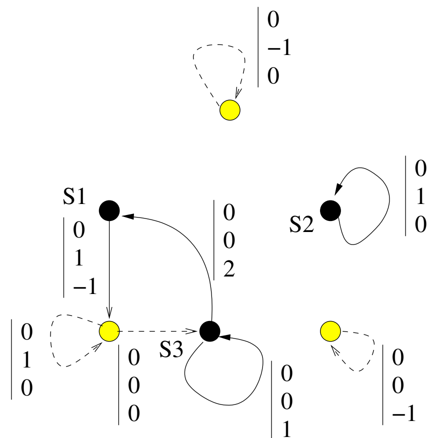



Uniformized RDG.

# Second example: $G$ and $G'$




Uniformized RDG.




$G'$: zero-weight multi-cycles.

$(2i, j)$ for $S_2$, $(2i + 1, 2k)$ for $S_1$, and $(2i + 1, 2k + 3)$ for $S_3$.

# Second exemple: parallel code generation

```
DOSEQ i=1, n
  DOSEQ j=1, n /* scheduling (2i, j) for S2*/
    DOPAR k=1, j
      S2: b(i,j,k) = a(i-1,j+i,k) + b(i,j-1,k)
    ENDDOPAR
  ENDDOSEQ
  DOSEQ k = 1, n+1
    IF (k ≤ n) THEN /* scheduling (2i+1, 2k) for S1*/
      DOPAR j=k, n
        S1: a(i,j,k) = c(i,j,k-1) + 1
      ENDDOPAR
    IF (k ≥ 2) THEN /* scheduling (2i+1, 2k+3) for S3*/
      DOPAR j=k-1, n
        S3: c(i,j,k) = c(i,j,k-1) + b(i,j-1,k+i-1) + a(i,j-k+1,k)
      ENDDOPAR
  ENDDOSEQ
ENDDOSEQ
```

- ➢ Loop distribution of $j$, $k$ loops: $S_2$ then $S_1 + S_3$.
- ➢ Loop interchange of $j$ and $k$ loops for $S_1$ and $S_3$.
- ➢ Loop shifting in $k$, then loop distribution of $j$ loop.

# Allen-(Callahan)-Kennedy (1987): loop distribution

AK($G$, $k$):

- Remove from $G$ all edges of level $< k$.
- Compute $G_1, \ldots, G_s$ the s SCCs of $G$ in topological order.
  - If $G_i$ has a single statement $S$, with no edge, generate DOPAR loops in all remaining dimensions, and generate code for $S$.
  - Otherwise:
    - Generate DOPAR loops from level $k$ to level $l - 1$, and a DOSEQ loop for level $l$, where $l$ is the minimal level in $G_i$.
    - call AK($G_i$, $l + 1$). /* $d_S$ sequential loops for statement $S$ */

➠ Variant of (dual of) KMW with DOPAR as high as possible.

# Allen-(Callahan)-Kennedy (1987): loop distribution

AK($G$, $k$):

- Remove from $G$ all edges of level $< k$.
- Compute $G_1, \ldots, G_s$ the s SCCs of $G$ in topological order.
  - If $G_i$ has a single statement $S$, with no edge, generate DOPAR loops in all remaining dimensions, and generate code for $S$.
  - Otherwise:
    - Generate DOPAR loops from level $k$ to level $l-1$, and a DOSEQ loop for level $l$, where $l$ is the minimal level in $G_i$.
    - call AK($G_i$, $l+1$). /* $d_S$ sequential loops for statement $S$ */

➽ Variant of (dual of) KMW with DOPAR as high as possible.

**Theorem (Optimality for AK w.r.t. dependence levels)**

*Nested loops $\mathcal{L}$, RDG $G$ with levels. One can build some nested loops $\mathcal{L}'$, with same structure as $\mathcal{L}$ and same RDG as $G$, with bounds parameterized by $N$ such that, for each SCC $G_i$ of $G$, there is a path in the EDG of $\mathcal{L}'$ that visits $\Omega(N^{d_S})$ times each statement $S$ of $G_i$ ($d_S$: depth w.r.t. $S$).*

# Darte-Vivien (1997): unimodular + shift + distribution

Boolean DV($G$, $k$) /* $G$ uniformized graph, with virtual and actual nodes */

- Build $G'$ generated by the zero-weight multi-cycles of $G$.
- Modify slightly $G'$ (technical detail not explained here).
- Choose $\vec{X}$ (vector) and, for each $S$ in $G'$, $\rho_S$ (scalar) s.t.:

$$\begin{cases} \text{if } e = (u, v) \in G' \text{ or } u \text{ is virtual, } \vec{X}.\vec{w}(e) + \rho_v - \rho_u \geq 0 \\ \text{if } e \notin G' \text{ and } u \text{ is actual, } \vec{X}.\vec{w}(e) + \rho_v - \rho_u \geq 1 \end{cases}$$

  For each actual node $S$ of $G$ let $\rho_S^k = \rho_S$ and $\vec{X}_S^k = \vec{X}$.
- Compute $G'_1, \ldots, G'_s$ the SCC of $G'$ with $\geq 1$ actual node:
  - If $G'$ is empty or has only virtual nodes, return TRUE.
  - If $G'$ is strongly connected with $\geq 1$ actual node, return FALSE.
  - Otherwise, return $\bigwedge_{i=1}^{s} \text{DV}(G'_i, k+1)$ ($\bigwedge$ = logical AND).

➽ Dual of KMW after dependence uniformization. Analyzing the cycle weights in $G'$ leads to a variant to get a max. number of permutable loops.

# General affine multi-dimensional schedules (Feautrier)

Affine dependences (or even relations): $T(\vec{j})$ depends on $S(\vec{i})$ if $(\vec{i}, \vec{j}) \in \mathcal{D}_e$ where $e = (S, T)$ and $\mathcal{D}_e$ is a polyhedron.

- Look for affine schedule $\sigma$ such that $\sigma(S, \vec{i}) <_{lex} \sigma(T, \vec{j})$ for all $(\vec{i}, \vec{j}) \in \mathcal{D}_e$. Use affine form of Farkas lemma (mechanical operation).
- Write $\sigma(S, \vec{i}) + \epsilon_e \leq \sigma(T, \vec{j})$ with $\vec{\epsilon} \geq \vec{0}$ and maximize the number of dependence edges $e$ such that $\epsilon_e \geq 1$.
- Remove edges $e$ such that $\epsilon_e \geq 1$ and continue to get remaining dimensions ☛ multi-dimensional affine schedule.

➽ Generalization of the constraints used in the dual of KMW.

# General affine multi-dimensional schedules (Feautrier)

Affine dependences (or even relations): $T(\vec{j})$ depends on $S(\vec{i})$ if $(\vec{i}, \vec{j}) \in \mathcal{D}_e$ where $e = (S, T)$ and $\mathcal{D}_e$ is a polyhedron.

- Look for affine schedule $\sigma$ such that $\sigma(S, \vec{i}) <_{lex} \sigma(T, \vec{j})$ for all $(\vec{i}, \vec{j}) \in \mathcal{D}_e$. Use affine form of Farkas lemma (mechanical operation).
- Write $\sigma(S, \vec{i}) + \epsilon_e \leq \sigma(T, \vec{j})$ with $\vec{\epsilon} \geq \vec{0}$ and maximize the number of dependence edges $e$ such that $\epsilon_e \geq 1$.
- Remove edges $e$ such that $\epsilon_e \geq 1$ and continue to get remaining dimensions ☛ multi-dimensional affine schedule.

➠ Generalization of the constraints used in the dual of KMW.

To perform tiling, look for several dimensions (permutable loops) such that $\sigma(T, \vec{j}) - \sigma(S, \vec{i}) \geq 0$ instead of $\sigma(T, \vec{j}) - \sigma(S, \vec{i}) \geq 1$. But more complicated to avoid the $\vec{0}$ solution and guarantee linear independence. Key idea in Pluto: minimize dependence distance $\sigma(T, \vec{j}) - \sigma(S, \vec{i})$.

# Loop parallelization: optimality w.r.t. dep. abstraction

Lamport (1974) hyperplane scheduling = skewing, interchange.

Allen-Kennedy (1987) loop distribution, optimal for levels.

Wolf-Lam (1991) unimodular, optimal for direction vectors and one statement. Based on finding permutable loops.

Feautrier (1992) general affine scheduling, complete for affine dependences and affine transformations, but not optimal. Relies on Farkas lemma.

Darte-Vivien (1997) unimodular, shift, distribution, optimal for polyhedral abstraction of distances. Finds permutable loops, too.

Lim-Lam (1998) extension to coarse-grain parallelism, vague.

Bondhugula-Ramanujam-Sadayappan (2008) improved extension, permutable loops (tiling), locality optimization. ☛ Pluto compiler. Principle: look for maximum number of linearly-independent solutions with nonnegative dependence distance. Tile bands.

# Yet another application of SUREs: understand "iterations"

## Fortran DO loops:

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

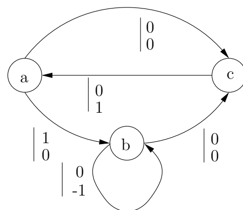
## Uniform recurrence equations:

$$\forall p \in \{p = (i,j) \mid 1 \le i,j \le N\}$$

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$

## C for and while loops:

```
y = 0; x = 0;
while (x <= N && y <= N) {
  if (?) {
    x=x+1;
    if (y >= 0 && ?) y=y-1;
  }
  y=y+1;
}
```



# KERNEL OFFLOADING AND LOOP TILING

# High-level synthesis (HLS) tools for FPGA

Industrial tools: pretty good for optimizing computation kernel
But still a huge problem for feeding the accelerators with data.

Our idea (∼2009): use HLS tools as back-end compilers, assuming it puts the necessary computing resources to be limited by bandwidth.

- Push all the dirty work in the back-end compiler.
- Optimize transfers at C level.
- Compile any new functions with the same HLS tool.

# High-level synthesis (HLS) tools for FPGA

Industrial tools: pretty good for optimizing computation kernel
But still a huge problem for feeding the accelerators with data.

Our idea (~2009): use HLS tools as back-end compilers, assuming it puts the necessary computing resources to be limited by bandwidth.

- Push all the dirty work in the back-end compiler.
- Optimize transfers at C level.
- Compile any new functions with the same HLS tool.

Use Altera C2H as a back-end compiler. Main features:

- Syntax-directed translation to hardware:
  - Local array = local memory, other arrays/pointers = external memory.
  - Hierarchical FSMs: outer FSM stalls to wait for the latest inner FSM.
- Software pipelined loops:
  - Basic software pipelining with rough data dependence analysis.
  - Latency-aware pipelined DDR accesses (with internal FIFOs).
- Full interface within the complete system:
  - Accelerator(s) initiated as (blocking or not) function call(s).
  - Possibility to define FIFOs between accelerators.

# High-level synthesis (HLS) tools for FPGA

Industrial tools: pretty good for optimizing computation kernel
But still a huge problem for feeding the accelerators with data.

Our idea (∼2009): use HLS tools as back-end compilers, assuming it puts the necessary computing resources to be limited by bandwidth.

- Push all the dirty work in the back-end compiler.
- Optimize transfers at C level.
- Compile any new functions with the same HLS tool.

Use Altera C2H as a back-end compiler. Main features:

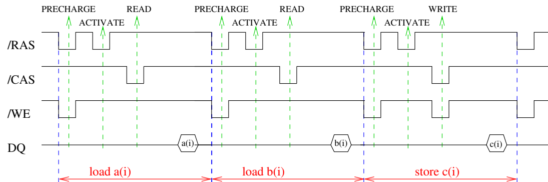
- Syntax-directed translation to hardware:
  - Local array = local memory, other arrays/pointers = external memory.
  - Hierarchical FSMs: outer FSM stalls to wait for the latest inner FSM.
- Software pipelined loops:
  - Basic software pipelining with rough data dependence analysis.
  - Latency-aware pipelined DDR accesses (with internal FIFOs).
- Full interface within the complete system:
  - Accelerator(s) initiated as (blocking or not) function call(s).
  - Possibility to define FIFOs between accelerators.

Similar study Pouchet et al. FPGA'13 for Xilinx AutoESL

# Throughput when accessing (asymmetric) DDR memory

Ex: DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz. Successive reads, same row = 10 ns, different rows = 80 ns. Even if fully pipelined ($\lambda$=1), a bad spatial DDR locality can kill performances by a factor 8! Example:

```
void vector_sum (int* __restrict__ a, b, c, int n) {
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}
```

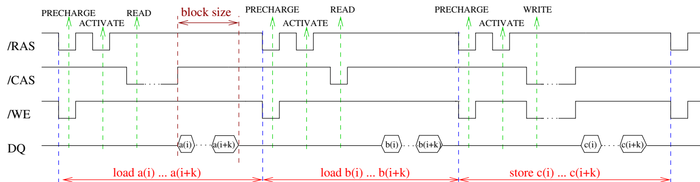



C2H-compiled code: pipelined but time gaps & data thrown away.

# Throughput when accessing (asymmetric) DDR memory

Ex: DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz. Successive reads, same row = 10 ns, different rows = 80 ns. Even if fully pipelined ($\lambda$=1), a bad spatial DDR locality can kill performances by a factor 8! Example:

```
void vector_sum (int* __restrict__ a, b, c, int n) {
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}
```

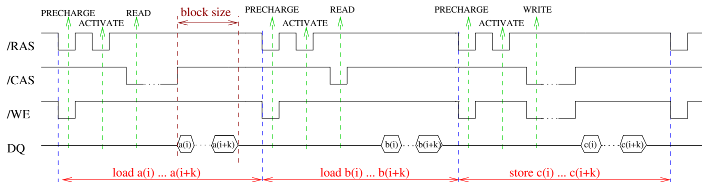



Block version: reduces gaps, exploits bursts and temporal reuse.

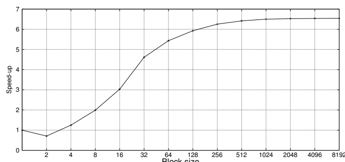
# Throughput when accessing (asymmetric) DDR memory

Ex: DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz. Successive reads, same row = 10 ns, different rows = 80 ns. Even if fully pipelined ($\lambda$=1), a bad spatial DDR locality can kill performances by a factor 8! Example:

```
void vector_sum (int* __restrict__ a, b, c, int n) {
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}
```



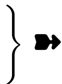

Block version: reduces gaps, exploits bursts and temporal reuse.




Typical figure with speed-up vs block size (here vector sum).

# Strip-mining and loop distribution?

Loop distribution: too large local memory.
Unrolling: too many registers.



strip-mining +
loop distribution.

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j]; //store
}
```

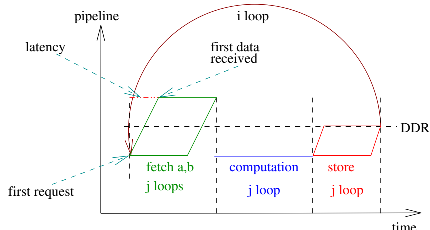
# Strip-mining and loop distribution?

Loop distribution: too large local memory.
Unrolling: too many registers.

} ➽ strip-mining + loop distribution.

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j]; //store
}
```

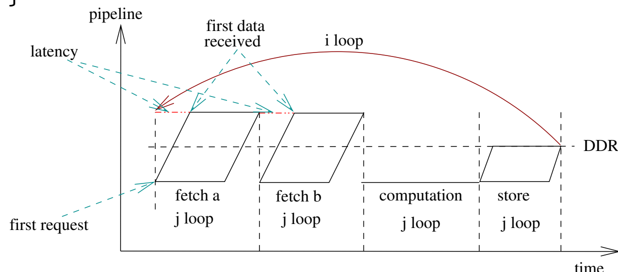
➽ Does not work!




- Accesses to arrays a and b still interleaved!
- Loop latency penalty.
- Outer loop not pipelined.

# Introduce false dependences?

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) tmp = BLOCK; a_tmp[j] = a[i+j];
  for(j=0; j<tmp; j++) b_tmp[j] = b[i+j];
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j];
}
```




➽ Still pay loop latency penalty and poor outermost loop pipeline.

# Emulating (linearizing) nested loops?

```
i=0; j=0; bi=0;
for (k=0; k<4*MAX; k++) {
  if (j==0) a_tmp[i] = a[bi+i];
  else if (j==1)
    b_tmp[i] = b[bi+i];
  else if (j==2)
    c_tmp[i] = a_tmp[i] + b_tmp[i];
  else c[bi+i] = c_tmp[i];

  if (i<BLOCK-1) i++;
  else {
    i=0;
    if (j<3) j++;
    else {j=0; bi = bi + BLOCK;}
  }
}
```

- Need to use *restrict* pragma for all arrays.
- $\lambda = 21$! Problem with dependence analyzer and software pipeliner.
- Better behavior ($\lambda = 3$) with *case* statement: by luck.
- Further loop unrolling to get $\lambda = 1$: too complex.
- But still a problem with interleaved DDR accesses due to speculative prefetching!

# Emulating nested loops with a single transfer instruction?

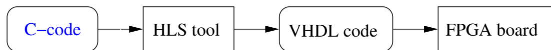
```
i=0; j=0; bi=0;
for (k=0; k<3*MAX; k++) {
  if (j==0) { ptr_1 = &a[bi+i]; ptr_2 = &a_tmp[i]; }
  else if (j==1) { ptr_1 = &b[bi+i]; ptr_2 = &b_tmp[i]; }
  else if (j==2) { ptr_1 = &c_tmp[i]; ptr_2 = &c[bi+i];
                   c_tmp[i] = a_tmp[i] + b_tmp[i]; }
  *ptr_2 = *ptr_1;

  if (i<BLOCK-1) i++;
  else { i=0; if (j<2) j++; else {j=0; bi = bi + BLOCK;}}
}
```

- No more interleaving between arrays a and b;
- $\lambda$ not equal to 1, unless *restrict* pragma added: but leads to potentially wrong codes (data dependences are lost).

How to decrease $\lambda$ and generalize to more complex codes?
☛ Communicating C processes, with suitable synchronizations.

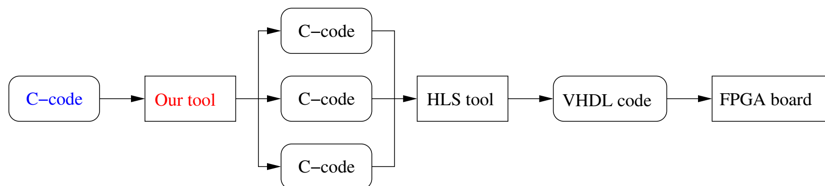
# Source-to-source transformations and kernel acceleration

How to optimize transfers automatically?



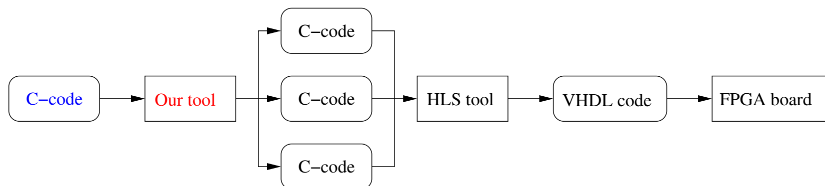
# Source-to-source transformations and kernel acceleration

How to optimize transfers automatically? With C-level processes.



# Source-to-source transformations and kernel acceleration

How to optimize transfers automatically? With C-level processes.

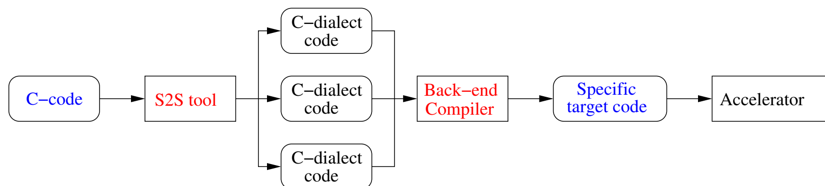



General problem: porting applications on hardware accelerators

- Ex: FPGA, GPU, dedicated board, multicore.
- Huge portability issue, costly compiler development.

# Source-to-source transformations and kernel acceleration

How to optimize transfers automatically? With C-level processes.




General problem: porting applications on hardware accelerators

- Ex: FPGA, GPU, dedicated board, multicore.
- Huge portability issue, costly compiler development.

Kernel offloading/function outlining

- High-productivity and high-performance languages.
- Library/directives-type support, e.g., OpenAcc.
- Source-to-source compilation, targeting back-end C dialects.

# Data Movement between Memory Levels

CPU Multi-level cache optimization

GPU Host to Global / Global to Shared / Shared to registers

MPPA External DDR to multi scratch-pads

FPGA External DDR to local memory

# Data Movement between Memory Levels

CPU Multi-level cache optimization

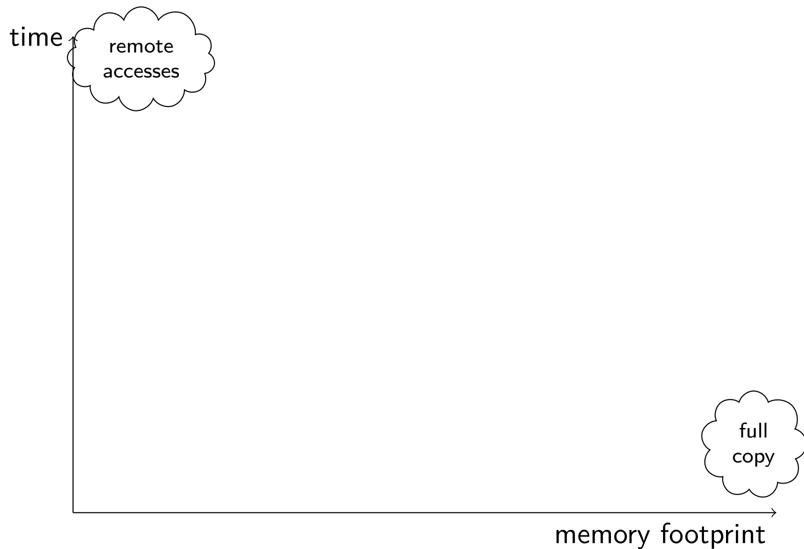
GPU Host to Global / Global to Shared / Shared to registers

MPPA External DDR to multi scratch-pads

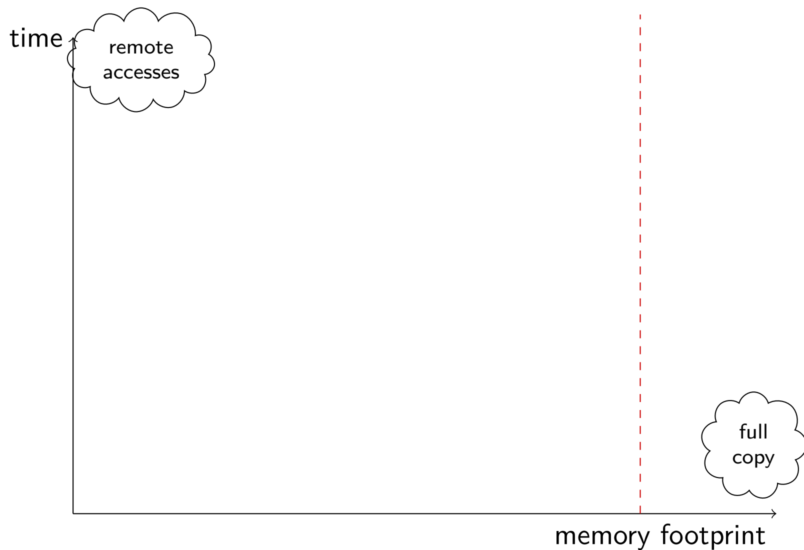
FPGA External DDR to local memory

- Computation by blocks ☛ loop tiling
- Transfer optimization ☛ intra-tile & inter-tile data reuse
- Communication/computation overlap ☛ pipelining
- Cost model ☛ parametric tile sizes

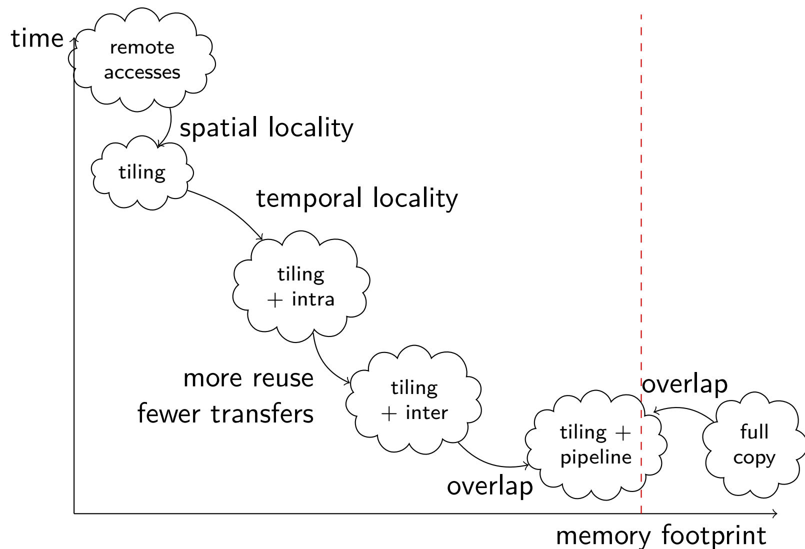
# Tiling, Data Reuse, and Pipelining



# Tiling, Data Reuse, and Pipelining

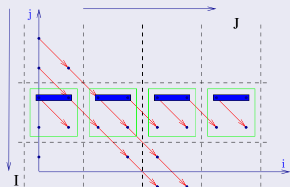


# Tiling, Data Reuse, and Pipelining



# Kernel offloading with parametric tiling & double buffering

Parameter $n$, tiles of size $b \times b$.

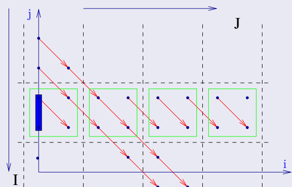



```
int i,j;
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) {
    C[i+j] = C[i+j] + A[i]*B[j];
  }
}
```

- Pipelining of tiles.
- Computation of loads/stores.
- Intra- and inter-tile data reuse.

# Kernel offloading with parametric tiling & double buffering

Parameter $n$, tiles of size $b \times b$.

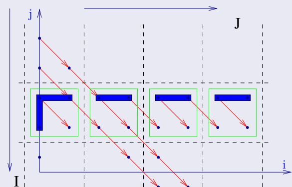



```
int i,j;
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) {
    C[i+j] = C[i+j] + A[i]*B[j];
  }
}
```

- Pipelining of tiles.
- Computation of loads/stores.
- Intra- and inter-tile data reuse.

# Kernel offloading with parametric tiling & double buffering

Parameter $n$, tiles of size $b \times b$.




```
int i,j;
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) {
    C[i+j] = C[i+j] + A[i]*B[j];
  }
}
```

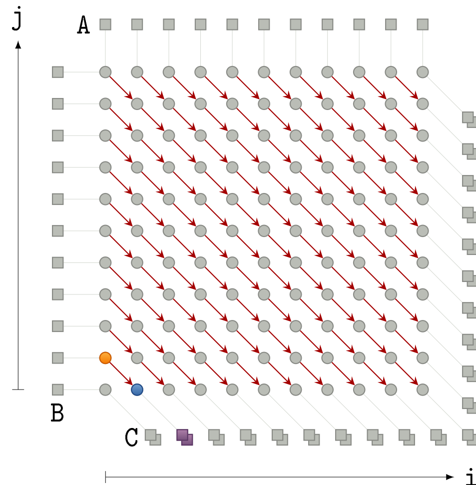
- Pipelining of tiles.
- Computation of loads/stores.
- Intra- and inter-tile data reuse.

$$\mathrm{Load}_C = \{m \mid 0 \le m,\ n - I - b \le m \le n - 1 - I,\ J = 0\} \\ \cup \{m \mid \max(1, J) \le m + I - n + 1 \le \min(n - 1, J + b - 1)\}$$

- size $3b - 1 = (2b - 1) + b$ when $n \ge 2b + 1$: 2 full tiles.
- size $b + n - 1 = (2b - 1) + (n - b)$ when $b \le n \le 2b$: 1 full tile, 1 partial tile.
- size $2n - 1$ when $n \le b - 1$: 1 partial tile.

☛ Parametric (not quadratic, but piece-wise affine), with possible approximations, corresponding liveness analysis, and memory mapping.

# Loop tiling and permutable loops



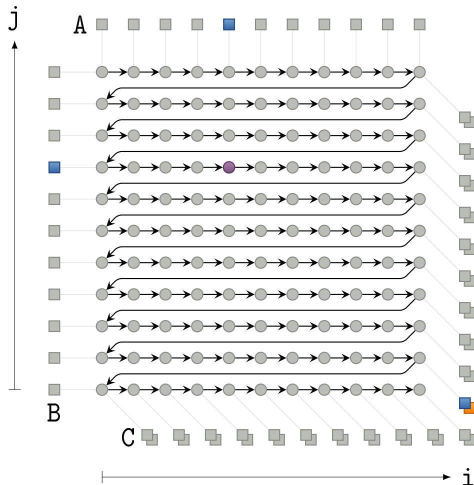

Product of two polynomials:

- arguments in $A$ and $B$;
- result in $C$.
- dep. distance $(1, -1)$.
  prevents permutation.

```
for (i=0, i<=2n-1; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i+j] = c[i+j] + A[i]*B[j];
```

Order: lexico along $(i + j, j)$.

# Loop tiling and permutable loops




Product of two polynomials:

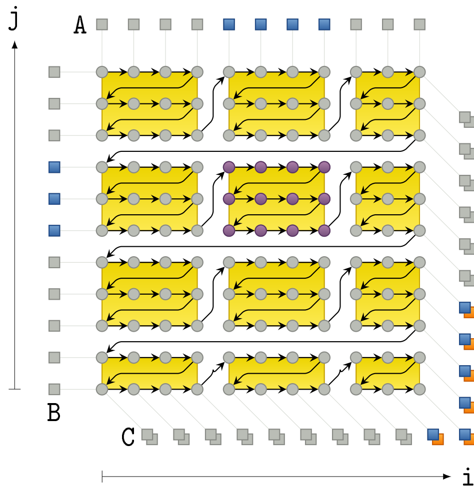
- arguments in $A$ and $B$;
- result in $C$.
- dep. distance $(1, -1)$.

  ☛ now $(1, 1)$.

```
for (i=0, i<=2n-1; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i+j] = c[i+j] + A[i]*B[j];
```

Order: lexico along $(n - j - 1, i)$.

# Loop tiling and permutable loops




Product of two polynomials:

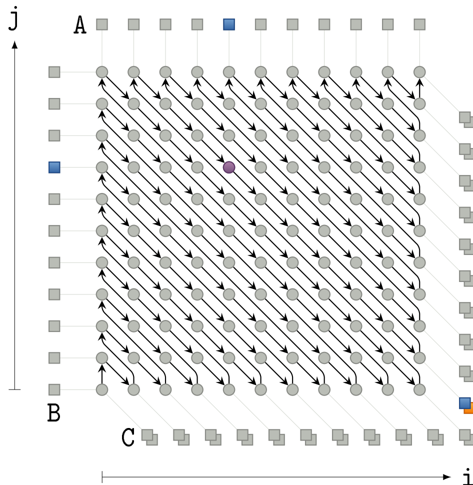
- arguments in $A$ and $B$;
- result in $C$.
- dep. distance $(1, -1)$.
  ☛ now $(1, 1)$, $(1, 0)$, $(0, 1)$.

```
for (i=0, i<=2n-1; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i+j] = c[i+j] + A[i]*B[j];
```

Order: lexico along $(n - j - 1, i)$.

Tiling: $(\lfloor \frac{n-j-1}{s_1} \rfloor, \lfloor \frac{i}{s_2} \rfloor, n - j - 1, i)$.

# Loop tiling and permutable loops



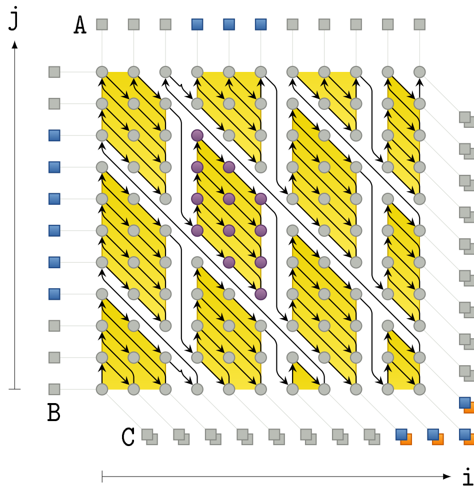

Product of two polynomials:

- arguments in $A$ and $B$;
- result in $C$.
- dep. distance $(1, -1)$.
  ☛ now $(0, 1)$.

```
for (i=0, i<=2n-1; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i+j] = c[i+j] + A[i]*B[j];
```

Order: lexico along $(i+j, j)$.

# Loop tiling and permutable loops




Product of two polynomials:

- arguments in $A$ and $B$;
- result in $C$.
- dep. distance $(1, -1)$.
  ☛ now $(0, 1)$.

```
for (i=0, i<=2n-1; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    c[i+j] = c[i+j] + A[i]*B[j];
```

Order: lexico along $(i + j, j)$.

Tiling: $(\lfloor \frac{i+j}{s_1} \rfloor, \lfloor \frac{i}{s_2} \rfloor, i + j, i)$.

# Validity of loop tiling through permutability

Initial paper: "Supernote partitioning", F. Irigoin, R. Triolet, ICS'88.

Permutability condition:

- Dependences not loop-carried are not relevant. Indeed, $(\vec{0}, \vec{d})$ remains unchanged, thus lexicopositive.

# Validity of loop tiling through permutability

Initial paper: "Supernote partitioning", F. Irigoin, R. Triolet, ICS'88.

Permutability condition:

- Dependences not loop-carried are not relevant. Indeed, $(\vec{0}, \vec{d})$ remains unchanged, thus lexicopositive.
- $\vec{d} \geq \vec{0}$ for all distance vector $\vec{d}$: $\vec{d} \succ \vec{0} \Rightarrow P\vec{d} \succ \vec{0}$ ($P$ permutation).

# Validity of loop tiling through permutability

Initial paper: "Supernote partitioning", F. Irigoin, R. Triolet, ICS'88.

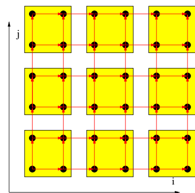
Permutability condition:

- Dependences not loop-carried are not relevant. Indeed, $(\vec{0}, \vec{d})$ remains unchanged, thus lexicopositive.
- $\vec{d} \geq \vec{0}$ for all distance vector $\vec{d}$: $\vec{d} \succ \vec{0} \Rightarrow P\vec{d} \succ \vec{0}$ ($P$ permutation).

Tiling validity: After tiling, $\vec{i} \mapsto (\vec{I}, \vec{ii}) = (\lfloor \vec{i}/\vec{s} \rfloor, \vec{i} \bmod \vec{s})$. Furthermore:

$$\vec{j} \geq \vec{i} \Rightarrow \vec{j} - \vec{i} \geq \vec{0} \Rightarrow (\vec{j} - \vec{i})/\vec{s} \geq \vec{0} \Rightarrow \vec{j}/\vec{s} \geq \vec{i}/\vec{s} \Rightarrow \lfloor \vec{j}/\vec{s} \rfloor \geq \lfloor \vec{i}/\vec{s} \rfloor$$

☛ Dep. $\vec{d} \succ \vec{0}$ becomes $\vec{d}' \succeq \vec{0}$. Tiles can still be scheduled.




Valid: yes

# Validity of loop tiling through permutability

Initial paper: "Supernote partitioning", F. Irigoin, R. Triolet, ICS'88.

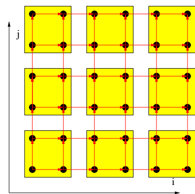
Permutability condition:

- Dependences not loop-carried are not relevant. Indeed, $(\vec{0}, \vec{d})$ remains unchanged, thus lexicopositive.
- $\vec{d} \geq \vec{0}$ for all distance vector $\vec{d}$: $\vec{d} \succ \vec{0} \Rightarrow P\vec{d} \succ \vec{0}$ ($P$ permutation).

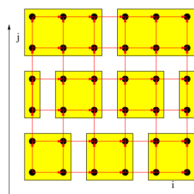
Tiling validity: After tiling, $\vec{i} \mapsto (\vec{I}, \vec{ii}) = (\lfloor \vec{i}/\vec{s} \rfloor, \vec{i} \bmod \vec{s})$. Furthermore:

$$\vec{j} \geq \vec{i} \Rightarrow \vec{j} - \vec{i} \geq \vec{0} \Rightarrow (\vec{j} - \vec{i})/\vec{s} \geq \vec{0} \Rightarrow \vec{j}/\vec{s} \geq \vec{i}/\vec{s} \Rightarrow \lfloor \vec{j}/\vec{s} \rfloor \geq \lfloor \vec{i}/\vec{s} \rfloor$$

☛ Dep. $\vec{d} \succ \vec{0}$ becomes $\vec{d}' \succeq \vec{0}$. Tiles can still be scheduled.




Valid: yes




Still valid?

# Validity of loop tiling through permutability

Initial paper: "Supernote partitioning", F. Irigoin, R. Triolet, ICS'88.

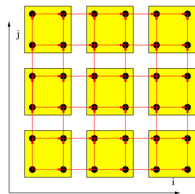
Permutability condition:

- Dependences not loop-carried are not relevant. Indeed, $(\vec{0}, \vec{d})$ remains unchanged, thus lexicopositive.
- $\vec{d} \geq \vec{0}$ for all distance vector $\vec{d}$: $\vec{d} \succ \vec{0} \Rightarrow P\vec{d} \succ \vec{0}$ ($P$ permutation).

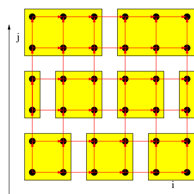
Tiling validity: After tiling, $\vec{i} \mapsto (\vec{I}, \vec{ii}) = (\lfloor \vec{i}/\vec{s} \rfloor, \vec{i} \bmod \vec{s})$. Furthermore:

$$\vec{j} \geq \vec{i} \Rightarrow \vec{j} - \vec{i} \geq \vec{0} \Rightarrow (\vec{j} - \vec{i})/\vec{s} \geq \vec{0} \Rightarrow \vec{j}/\vec{s} \geq \vec{i}/\vec{s} \Rightarrow \lfloor \vec{j}/\vec{s} \rfloor \geq \lfloor \vec{i}/\vec{s} \rfloor$$

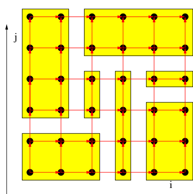
☛ Dep. $\vec{d} \succ \vec{0}$ becomes $\vec{d}' \succeq \vec{0}$. Tiles can still be scheduled.



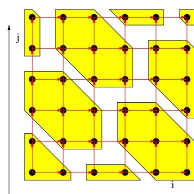
Valid: yes



Still valid?



Still valid?



Still valid?

# "Ratio" of computation and communication volume

"Pen-ultimate Tiling?", P. Boulet, A. Darte, T. Risset, Y. Robert, Integration J., 1994.

Tile shape given by tile edges. Ex: $P = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$. $H = P^{-1}$

$H = \frac{1}{6}\begin{pmatrix} 3 & -1 \\ 0 & 2 \end{pmatrix}$. With $D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $HD = \frac{1}{6}\begin{pmatrix} 3 & 2 \\ 0 & 2 \end{pmatrix} \geq \vec{0}$.

Tile computation volume equal to $|\det P| = 1/|\det H|$.

# "Ratio" of computation and communication volume

"Pen-ultimate Tiling?", P. Boulet, A. Darte, T. Risset, Y. Robert, Integration J., 1994.

Tile shape given by tile edges. Ex: $P = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$. $H = P^{-1}$

$H = \frac{1}{6}\begin{pmatrix} 3 & -1 \\ 0 & 2 \end{pmatrix}$. With $D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $HD = \frac{1}{6}\begin{pmatrix} 3 & 2 \\ 0 & 2 \end{pmatrix} \geq \vec{0}$.

Tile computation volume equal to $|\det P| = 1/|\det H|$.

Tile communication volume roughly $\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|}$. Indeed:

$$|\det(d, p_2, \ldots, p_n)| = |(p_2 \wedge \cdots \wedge p_n).d| = |(\det P)h_1.d| = |\det P|(h_1.d)$$

# "Ratio" of computation and communication volume

"Pen-ultimate Tiling?", P. Boulet, A. Darte, T. Risset, Y. Robert, Integration J., 1994.

Tile shape given by tile edges. Ex: $P = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$. $H = P^{-1}$

$$H = \frac{1}{6}\begin{pmatrix} 3 & -1 \\ 0 & 2 \end{pmatrix}. \text{ With } D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, HD = \frac{1}{6}\begin{pmatrix} 3 & 2 \\ 0 & 2 \end{pmatrix} \geq \vec{0}.$$

Tile computation volume equal to $|\det P| = 1/|\det H|$.

Tile communication volume roughly $\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|}$. Indeed:

$$|\det(d, p_2, \ldots, p_n)| = |(p_2 \wedge \cdots \wedge p_n).d| = |(\det P)h_1.d| = |\det P|(h_1.d)$$

Aspect ratio Minimize communication for a given computation volume.

Minimize $\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|^{\frac{1}{n}}}$ subject to $\det H \neq 0$, $HD \geq \vec{0}$.

# "Ratio" of computation and communication volume

"Pen-ultimate Tiling?", P. Boulet, A. Darte, T. Risset, Y. Robert, Integration J., 1994.

Tile shape given by tile edges. Ex: $P = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$. $H = P^{-1}$

$H = \frac{1}{6}\begin{pmatrix} 3 & -1 \\ 0 & 2 \end{pmatrix}$. With $D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $HD = \frac{1}{6}\begin{pmatrix} 3 & 2 \\ 0 & 2 \end{pmatrix} \geq \vec{0}$.

Tile computation volume equal to $|\det P| = 1/|\det H|$.

Tile communication volume roughly $\dfrac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|}$. Indeed:

$$|\det(d, p_2, \ldots, p_n)| = |(p_2 \wedge \cdots \wedge p_n).d| = |(\det P)h_1.d| = |\det P|(h_1.d)$$

Aspect ratio Minimize communication for a given computation volume.

Minimize $\dfrac{\sum_{i=1}^{n}\sum_{j=1}^{m}(HD)_{i,j}}{|\det H|^{\frac{1}{n}}}$ subject to $\det H \neq 0$, $HD \geq \vec{0}$. Solvable!

- If $D$ is square, non singular, opt. for $H = D^{-1}$ and ratio $n|\det D|^{\frac{1}{n}}$.
- General case: cone $C(D)$, opt. for $n$ generators of $C^*(D)$, scaled to induce the same communication on all faces.

# Generating loop bounds for tiled codes

- "Scanning Polyhedra with DO Loops", Ancourt, Irigoin, PPoPP'91.
- "An Efficient Code Generation Technique for Tiled Iteration Spaces", Goumas, Athanasaki, Koziris, IEEE TPDS, 2003.
- "Parameterized Tiled Loops for Free", Renga., Kim, Rajopadhye, Strout, PLDI'07.

Tile of size $\vec{s}$ with origin $\vec{i_0}$: $\{\vec{i} \mid \vec{i_0} \leq \vec{i} \leq \vec{i_0} + \vec{s} - \vec{1}\}$.

Elementary bounding box method: many empty tiles.

```
//Original code
for(t=1; t<=M; t++)
  for(i=2; i<=N; i++)
    S(t,i);
```

```
//Skewed code
for(t=1; t<=M; t++)
  for(k=2+t; k<=N+t; k++)
    S(t,k-t);
```

```
//Tiled code, many empty iterations
for(T=1; T<=M; T+=st)
  for(K=3; K<=N+M; K+=sk}
    for(t=max(T,1); t<=min(T+st-1,M); t++)
      for(k=max(K,2+t); k<=min(K+sk-1,N+t); k++)
        S(t,k-t);
```

# Generating loop bounds for tiled codes

- "Scanning Polyhedra with DO Loops", Ancourt, Irigoin, PPoPP'91.
- "An Efficient Code Generation Technique for Tiled Iteration Spaces", Goumas, Athanasaki, Koziris, IEEE TPDS, 2003.
- "Parameterized Tiled Loops for Free", Renga., Kim, Rajopadhye, Strout, PLDI'07.

Tile of size $\vec{s}$ with origin $\vec{i_0}$: $\{\vec{i} \mid \vec{i_0} \leq \vec{i} \leq \vec{i_0} + \vec{s} - \vec{1}\}$.

Exact computation of non-empty tiles for fixed tile sizes.

```
//Original code
for(t=1; t<=M; t++)
  for(i=2; i<=N; i++)
    S(t,i);
```

```
//Skewed code
for(t=1; t<=M; t++)
  for(k=2+t; k<=N+t; k++)
    S(t,k-t);
```

Code not depicted. Only for fixed tile sizes. Express the lattice $\mathcal{L}(\vec{s})$ of all tile origins $\vec{i_0} = \vec{0} \bmod \vec{s}$, and project existential variables.

# Generating loop bounds for tiled codes

- "Scanning Polyhedra with DO Loops", Ancourt, Irigoin, PPoPP'91.
- "An Efficient Code Generation Technique for Tiled Iteration Spaces", Goumas, Athanasaki, Koziris, IEEE TPDS, 2003.
- "Parameterized Tiled Loops for Free", Renga., Kim, Rajopadhye, Strout, PLDI'07.

Tile of size $\vec{s}$ with origin $\vec{i_0}$: $\{\vec{i} \mid \vec{i_0} \leq \vec{i} \leq \vec{i_0} + \vec{s} - \vec{1}\}$.

Approx. iterations $\{\vec{z} \mid Q\vec{z} \geq \vec{b}\}$ by the "outer set" $\{\vec{z} \mid Q\vec{z} + Q^{+}\vec{s} \geq \vec{b}\}$.

```
//Skewed code
for(t=1; t<=M; t++)
  for(k=2+t; k<=N+t; k++)
    S(t,k-t);
```

Outer set shift:

- $1 \leq t \leq M \Rightarrow 2 - st \leq t \leq M$
- $2 \leq k-t \leq N \Rightarrow 3-sk \leq k-t \leq N+st-1$

```
//Tiled code, fewer empty iterations
LBT = 2-st; LBT = ceil(LBT/st)*st;
for(T=LBT; T<=M; T+=st) {
  LBK = 3+t-sk; LBK = ceil(LBK/sk)*sk;
  for(K=LBK; K<=N+t+st-1; K+=sk}
    for(t=max(T,1); t<=min(T+st-1,M); t++)
      for(k=max(K,2+t); k<=min(K+sk-1,N+t); k++)
        S(t,k-t);
}
```

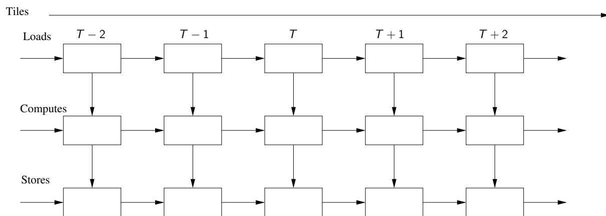
# INTER-TILE DATA REUSE AND LOCAL STORAGE

# General specification of data transfers

## Definition

- $\mathrm{Load}(T)$: data loaded from DDR just before executing tile $T$.
- $\mathrm{Store}(T)$: data stored to DDR just after $T$.
- $\mathrm{In}(T)$: input data for $T$, i.e., read before being possibly written in $T$.
- $\mathrm{Out}(T)$: output data for $T$, i.e., data written in $T$.

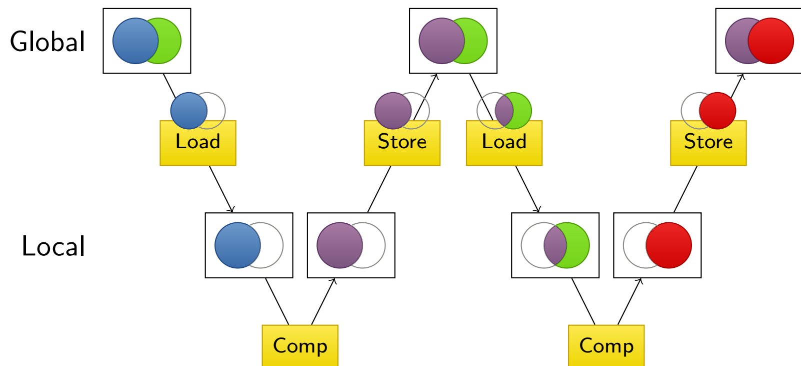
## Minimal dependence structure



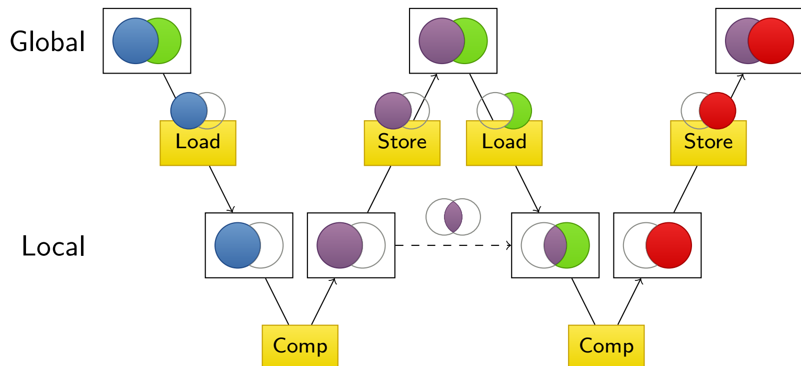

## Goals

- Reuse local data: intra- and inter-tile reuse in a tile strip.
- Do not store back after each write: external memory not up-to-date.
- Minimize live-ranges in local memory: need local memory allocation.

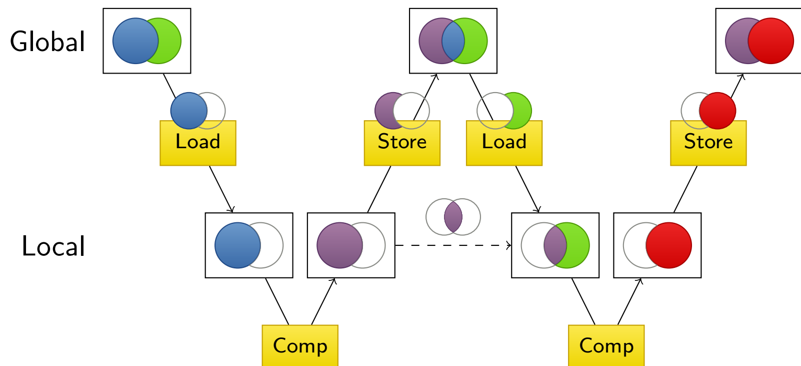
# Inter-tile data reuse



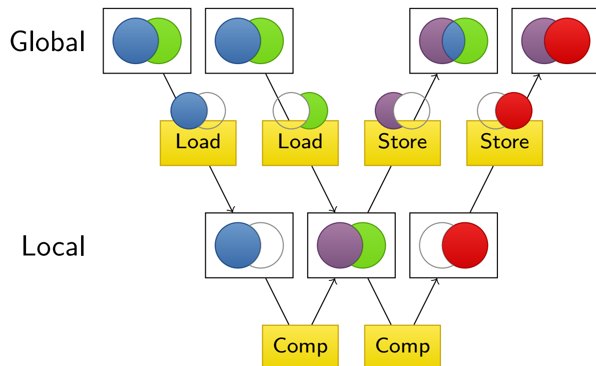
# Inter-tile data reuse



# Inter-tile data reuse

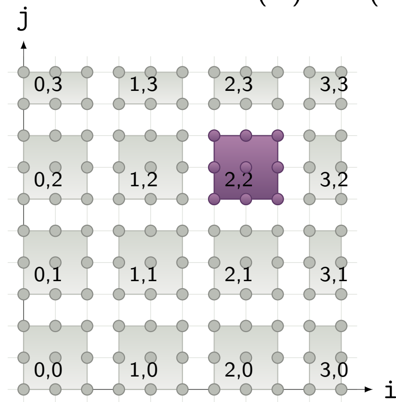


# Inter-tile data reuse enables pipelining



# Inter-tile reuse formula w.r.t. a generic tile $T$

$$\mathrm{Load}(T) = \mathrm{In}(T) \setminus \bigcup_{T' \sqsubset_s T} \mathrm{In}(T') \cup \mathrm{Out}(T')$$

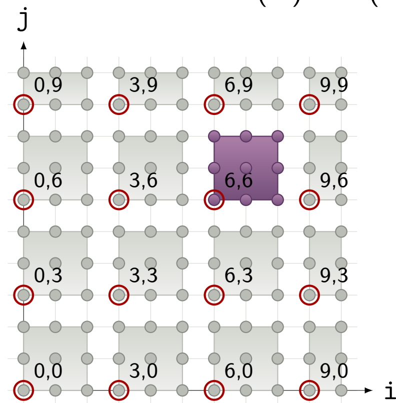



$$\begin{cases} s_i I \leq i < s_i(I+1) \\ s_j J \leq j < s_j(J+1) \\ i, j \in \mathrm{Domain} \end{cases}$$

Quadratic if $T$ indexed by $\lfloor \vec{i} / \vec{s} \rfloor$.

# Inter-tile reuse formula w.r.t. a generic tile $T$

$$\mathrm{Load}(T) = \mathrm{In}(T) \setminus \bigcup_{T' \sqsubset_s T} \mathrm{In}(T') \cup \mathrm{Out}(T')$$



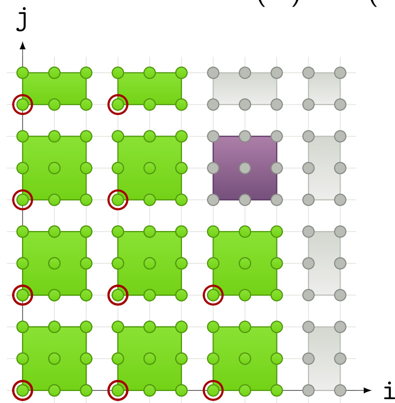

$$\begin{cases} I \leq i < I + s_i \\ J \leq j < J + s_j \\ i, j \in \mathrm{Domain} \end{cases}$$

Affine with tile origin? Not yet.

$$\begin{cases} I \equiv 0 \pmod{s_i} \\ J \equiv 0 \pmod{s_j} \end{cases}$$

# Inter-tile reuse formula w.r.t. a generic tile $T$

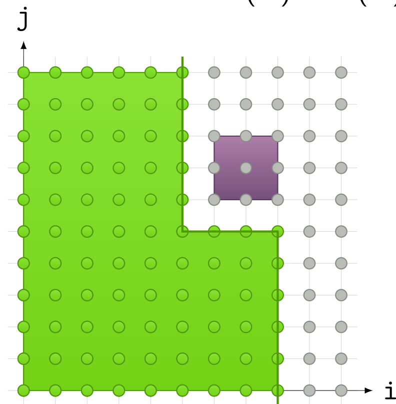
$$\mathrm{Load}(T) = \mathrm{In}(T) \setminus \bigcup_{T' \sqsubset_s T} \mathrm{In}(T') \cup \mathrm{Out}(T')$$




Still not affine

# Inter-tile reuse formula w.r.t. a generic tile $T$

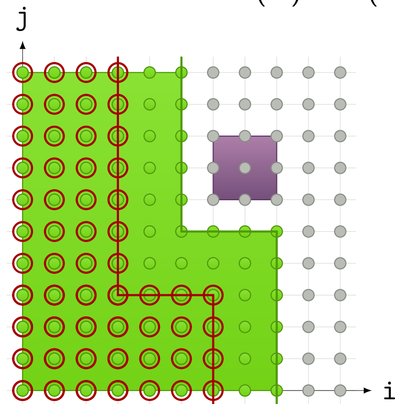
$$\mathrm{Load}(T) = \mathrm{In}(T) \setminus \bigcup_{T' \sqsubset_s T} \bigcup_{x \in T'} \mathrm{in}(x) \cup \mathrm{out}(x)$$




Piece-wise affine!

# Inter-tile reuse formula w.r.t. a generic tile $T$

$$\mathrm{Load}(T) = \mathrm{In}(T) \setminus \bigcup_{T' \prec_s T} \mathrm{In}(T') \cup \mathrm{Out}(T')$$




Still piece-wise affine.
Computations can be done with iscc.

# Approximations: why?

Some operations *may* execute

- `if` conditions that are not analyzable.

Some data *may* be accessed

- access functions that are not fully analyzable.

Approximated In/Out sets for tiles ☛ $\overline{\mathrm{In}}$, $\overline{\mathrm{Out}}$, $\underline{\mathrm{Out}}$.

- due to the analysis (e.g., array regions);
- by choice to represent simpler sets (e.g., hyper-rectangles);
- to simplify the analysis (e.g., Fourier-Motzkin).

Approximated Load/Store sets ☛ $\overline{\mathrm{Store}}$, $\overline{\mathrm{Load}}$.

- to simplify code generation;
- to perform communications by blocks;
- to simplify memory allocation;
- ...

# May-write approximation hazard

Problem:

- Tile T may write X, i.e., $X \in \overline{\mathrm{Out}}(T)$
- Tile T' may read X, i.e., $X \in \overline{\mathrm{In}}(T')$
- Only T and T' may access X

Should we load X? When?

# May-write approximation hazard

Problem:

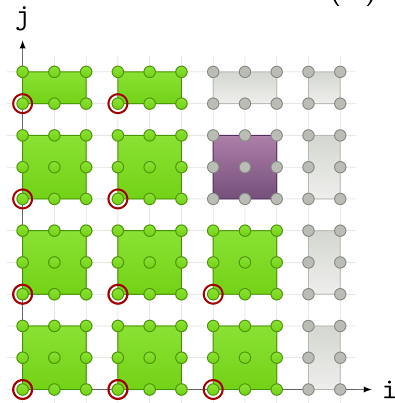
- Tile T may write X, i.e., $X \in \overline{\mathrm{Out}}(T)$
- Tile T' may read X, i.e., $X \in \overline{\mathrm{In}}(T')$
- Only T and T' may access X

Should we load X? When?

- T' happens before T ☛ load before T'
- T' happens after T and $X \in \underline{\mathrm{Out}}(T)$ ☛ do not load
- T' happens after T and $X \notin \underline{\mathrm{Out}}(T)$ ☛ load before T

# Reuse formula with approximation and *point-wise* functions

$$\text{Load}(T) = F(T) \setminus \bigcup_{T' \sqsubset_s T} F(T')$$

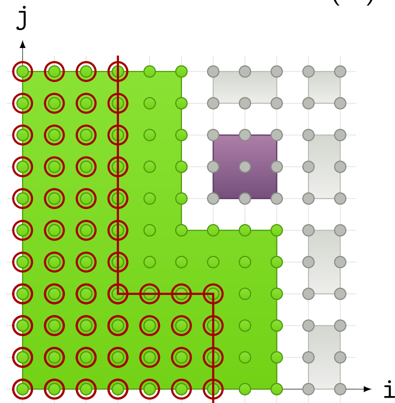



$F$ defined at tile level.

Can we use the same trick as for the exact case?

# Reuse formula with approximation and *point-wise* functions

$$\mathrm{Load}(T) = F(T) \setminus \bigcup_{T' \prec_s T} F(T')$$

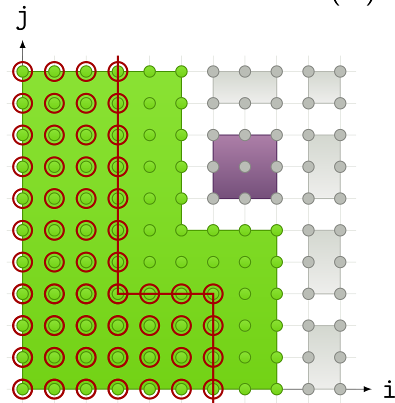



Yes if

$$\bigcup_{T' \sqsubset_s T} F(T') = \bigcup_{T' \prec_s T} F(T')$$

# Reuse formula with approximation and *point-wise* functions

$$\mathrm{Load}(T) = F(T) \setminus \bigcup_{T' \prec_s T} F(T')$$

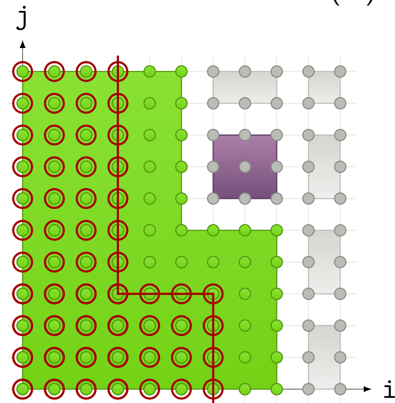



$\mathcal{C}$ set of all tiles, $\mathcal{C}_1, \mathcal{C}_2 \subseteq \mathcal{C}$
such that $\bigcup_{T \in \mathcal{C}_1} T = \bigcup_{T \in \mathcal{C}_2} T$

$$\bigcup_{T \in \mathcal{C}_1} F(T) \stackrel{?}{=} \bigcup_{T \in \mathcal{C}_2} F(T)$$

# Reuse formula with approximation and *point-wise* functions

$$\mathrm{Load}(T) = F(T) \setminus \bigcup_{T' \prec_s T} F(T')$$




$\mathcal{C}$ set of all tiles, $\mathcal{C}_1, \mathcal{C}_2 \subseteq \mathcal{C}$
such that $\bigcup_{T \in \mathcal{C}_1} T = \bigcup_{T \in \mathcal{C}_2} T$

$$\bigcup_{T \in \mathcal{C}_1} F(T) \stackrel{?}{=} \bigcup_{T \in \mathcal{C}_2} F(T)$$

Iff $F$ is point-wise:

$$\exists f, \forall T \in \mathcal{C}, F(T) = \bigcup_{x \in T} f(x)$$

☛ similar to the exact case.

# Simple script with iscc

```
# Inputs
Params := [M, N, s_1, s_2] -> { : s_1 >= 0 and s_2 >= 0 };
Domain := [M, N] -> { # Iteration domains
  S_1[i_1, i_2] : 1 <= i_2 <= N-2 and 0 <= i_1 <= M-1;
  S_2[i_1, i_2] : 1 <= i_2 <= N-2 and 0 <= i_1 <= M-1;
} * Params;
Read := [M, N] -> { # Read access functions
  S_1[i_1, i_2] -> A[m] : -1 + i_2 <= m <= 1 + i_2;
  S_2[i_1, i_2] -> B[i_2]; } * Domain;
Write := [M, N] -> { # Write access functions
  S_1[i_1, i_2] -> B[i_2];
  S_2[i_1, i_2] -> A[i_2]; } * Domain;
Theta := [M, N] -> { # Preliminary mapping
  S_1[i_1, i_2] -> [i_1, 2 i_1 + i_2, 0];
  S_2[i_1, i_2] -> [i_1, 1 + 2 i_1 + i_2, 1]; };

# Tools for set manipulations
Tiling := [s_1, s_2] -> { # Two dimensional tiling
  [[I_1, I_2] -> [i_1, i_2, k]] -> [i_1, i_2, k] :
     I_1 <= i_1 < I_1 + s_1 and I_2 <= i_2 < I_2 + s_2 };
Coalesce := { [I_1, I_2] -> [[I_1, I_2] -> [i_1, i_2, k]] };
Strip := { [I_1, I_2] -> [I_1, I_2'] };
Prev := { # Lexicographic order
  [[I_1, I_2] -> [i_1, i_2, k]] -> [[I_1, I_2] -> [i_1', i_2', k']] :
     i_1' <= i_1 - 1 or (i_1' <= i_1 and i_2' <= i_2 - 1)
     or (i_1' <= i_1 and i_2' <= i_2 and k' <= k - 1) };
TiledPrev := [s_1, s_2] -> { # Special ``lexicographic'' order
  [I_1, I_2] -> [I_1', I_2'] : I_1' <= I_1 - s_1 or
     (I_1' <= I_1 and I_2' <= I_2 - s_2) } * Strip;
TiledNext := TiledPrev^-1;
```

```
# Set/relation computations
TiledRead  := Tiling.(Theta^-1).Read;
TiledWrite := Tiling.(Theta^-1).Write;
In  := Coalesce.(TiledRead - (Prev.TiledWrite));
Out := Coalesce.TiledWrite;
Load  := In - ((TiledPrev.In) + (TiledPrev.Out));
Store := Out - (TiledNext.Out);
print coalesce (Load % Params);
print coalesce (Store % Params);
```

# Load/store sets for tiled Jacobi 1D

$$
\begin{aligned}
\mathrm{Load}(\vec{I}) = & \{\mathrm{A}(m) \mid 1 \le m + 2I_1 - I_2 \le s_2,\ s_1 \ge 1,\ I_1 \ge 0,\ m \ge 1,\ I_1 \le -1 + M, \\
& \quad I_2 \ge 2 - s_2 + 2I_1,\ m \le -1 + N,\ N \ge 3\} \\
& \cup \{\mathrm{A}(m) \mid m \ge 1 + I_2,\ m \ge 1,\ M \ge 1,\ m \le -1 + N,\ I_1 \le -1, \\
& \quad I_1 \ge 1 - s_1,\ I_2 \ge 2 - s_2,\ N \ge 3,\ m \le s_2 + I_2\} \\
& \cup \{\mathrm{A}(1) \mid I_2 = 1 + 2I_1 \wedge 0 \le I_1 \le -1 + M,\ N \ge 3,\ s_1 \ge 1,\ s_2 \ge 1\} \\
& \cup \{\mathrm{A}(m) \mid 0 \le m \le 1,\ I_2 = 1 \le s_2,\ 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3\} \\
& \cup \{\mathrm{A}(0) \mid 0 \le I_1 \le M - 1,\ N \ge 3,\ s_1 \ge 1,\ 1 \le I_2 - 2I_1 \ge 2 - s_2\} \\
& \cup \{\mathrm{A}(0) \mid 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3,\ I_2 \ge 2 - s_2,\ I_2 \le 0\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Store}(\vec{I}) = & \{\mathrm{B}(m) \mid m \ge 1,\ m \ge 2 - 2M + s_2 + I_2,\ m \le -2 + N, \\
& \quad I_1 \ge 1 - s_1,\ 2 \le m + 2s_1 + 2I_1 - I_2 \le 1 + s_2,\ s_1 \ge 1\} \\
& \cup \{\mathrm{B}(m) \mid m \ge 1,\ s_1 \ge 1,\ m \le -2 + N,\ I_1 \le -1 + M,\ m \le 1 - 2M + s_2 + I_2, \\
& \quad m \ge 2 - 2s_1 - 2I_1 + I_2,\ I_1 \ge 1 - s_1,\ M \ge 1,\ m \ge 2 - 2M + I_2\} \\
& \cup \{\mathrm{A}(m) \mid m \ge 1,\ m \ge 1 - 2M + s_2 + I_2,\ m \le -2 + N, \\
& \quad I_1 \ge 1 - s_1,\ 1 \le m + 2s_1 + 2I_1 - I_2 \le s_2,\ s_1 \ge 1\} \\
& \cup \{\mathrm{A}(m) \mid m \ge 1,\ s_1 \ge 1,\ m \le -2 + N,\ I_1 \le -1 + M,\ m \le -2M + s_2 + I_2, \\
& \quad m \ge 1 - 2s_1 - 2I_1 + I_2,\ I_1 \ge 1 - s_1,\ M \ge 1,\ m \ge 1 - 2M + I_2\}
\end{aligned}
$$

# Local buffer sizes

| Sequential Memory Size | Pipelined Memory Size |
|---|---|
| **jacobi-1d-imper** | |
| A$[2s_1 + s_2]$<br>B$[2s_1 + s_2 - 1]$ | A$[2s_1 + 2s_2]$<br>B$[2s_1 + 2s_2 - 2]$ |
| **jacobi-2d-imper** | |
| A$[2s_1 + s_2, \min(2s_1, s_2 + 1) + s_3]$<br>B$[2s_1 + s_2 - 1, \min(2s_1, s_2 + 1) + s_3 - 1]$ | A$[2s_1 + s_2, \min(2s_1, s_2 + 1) + 2s_3]$<br>B$[2s_1 + s_2 - 1, \min(2s_1, s_2 + 1) + 2s_3 - 2]$ |
| **seidel-2d** | |
| A$\begin{bmatrix} s_1 + s_2 + 1, \\ \min(2s_1 + 2, s_1 + s_2, 2s_2 + 2) + s_3 \end{bmatrix}$ | A$\begin{bmatrix} s_1 + s_2 + 1, \\ \min(2s_1 + 2, s_1 + s_2, 2s_2 + 2) + 2s_3 \end{bmatrix}$ |
| **gemm** | |
| A$[s_1, s_3]$<br>B$[s_3, s_2]$<br>C$[s_1, s_2]$ | A$[s_1, 2s_3]$<br>B$[2s_3, s_2]$<br>C$[s_1, s_2]$ |
| **floyd-warshall** | |
| path$\begin{bmatrix} \max(k+1, n-k), \\ \max(k+1, n-k) \end{bmatrix}$ | path$\begin{bmatrix} \max(k+1, n-k), \\ \max(k+1, n-k, 2s_2) \end{bmatrix}$ |

# Instance-wise order for sequential and parallel loops

```
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
S:      ...
T:      ...
   }
}
```

- Total order $\prec$ defined by a sequential schedule $\sigma$ and lexicographic order.
- $\sigma(S(i,j)) = (i,j,0)$, $\sigma(T(i,j)) = (i,j,1)$.
- $O \prec O'$ iff $\sigma(O) <_{\text{lex}} \sigma(O')$.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.

# Instance-wise order for sequential and parallel loops

```
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
S:     ...
T:     ...
   }
}
```

- Total order $\prec$ defined by a sequential schedule $\sigma$ and lexicographic order.
- $\sigma(S(i,j)) = (i,j,0)$, $\sigma(T(i,j)) = (i,j,1)$.
- $O \prec O'$ iff $\sigma(O) <_{\text{lex}} \sigma(O')$.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.

```
for(i=0; i<n; i++) {
  forpar(j=0; j<n; j++) {
S:     ...
T:     ...
   }
}
```

- Partial order $\prec$, some form of lexicographic order.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j = j')$.

# Instance-wise order for sequential and parallel loops

```
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
S:      ...
T:      ...
   }
}
```

- Total order $\prec$ defined by a sequential schedule $\sigma$ and lexicographic order.
- $\sigma(S(i,j)) = (i,j,0)$, $\sigma(T(i,j)) = (i,j,1)$.
- $O \prec O'$ iff $\sigma(O) <_{\text{lex}} \sigma(O')$.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.

```
for(i=0; i<n; i++) {
  forpar(j=0; j<n; j++) {
S:      ...
T:      ...
   }
}
```

- Partial order $\prec$, some form of lexicographic order.
- $S(i,j) \prec T(i',j')$ iff $i < i'$ or $(i = i'$ and $j = j')$.

```
forpar(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
S:      ...
T:      ...
   }
```

# More general orders: polyhedral fragment of X10

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

# More general orders: polyhedral fragment of X10

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Yes. Similar to data-flow analysis. Partial order $\prec$: incomplete lexicographic order.

# More general orders: polyhedral fragment of X10

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

Yes. Similar to data-flow analysis. Partial order $\prec$: incomplete lexicographic order.

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Undecidable. Partial order $\prec_c$ defined by $\vec{x} \prec_c \vec{y}$ iff $\vec{x} \prec \vec{y}$ or $\phi(\vec{x}) < \phi(\vec{y})$. $\phi(\vec{x}) = \#$ advances before (for $\prec$) $\vec{x}$.

# Liveness analysis

Uses of liveness analysis:

- Necessary for memory reuse:
  - Register allocation: interference graph.
  - Array contraction: conflicting relations.
  - Even wire usage: bitwidth analysis.
- Important information for:
  - Communication: live-in/live-out sets (inlining, offloading)
  - Memory footprint (e.g., for cache prediction)
  - Lower/upper bounds on memory usage.

# Liveness analysis

Uses of liveness analysis:

- Necessary for memory reuse:
  - Register allocation: interference graph.
  - Array contraction: conflicting relations.
  - Even wire usage: bitwidth analysis.
- Important information for:
  - Communication: live-in/live-out sets (inlining, offloading)
  - Memory footprint (e.g., for cache prediction)
  - Lower/upper bounds on memory usage.

Several variants:

- Value-based or memory-based analysis.
- Liveness sets or interference graphs.
- Control flow graphs (CFG): basic blocks, SSA, SSI, etc.
- Task graphs, parallel specifications: not really explored so far.

# Array contraction: symbolic unrolling, analysis, mapping

```
x = ...;
y = x + ...;
... = y;
```

$\Rightarrow$

```
x = ...;
x = x + ...;
... = x;
```

# Array contraction: symbolic unrolling, analysis, mapping

```
c[0] = 0;
for(i=0; i<n; i++) {
  c[i+1] = c[i] + ...;
}
```

$\Rightarrow$

```
c = 0;
for(i=0; i<n; i++) {
  c = c + ...;
}
```

# Array contraction: symbolic unrolling, analysis, mapping

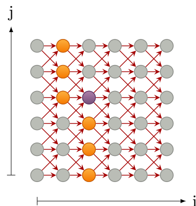
```
c[0] = 0;
for(i=0; i<n; i++) {
  c[i+1] = c[i] + ...;
}
```

⇒

```
c = 0;
for(i=0; i<n; i++) {
  c = c + ...;
}
```

```
for(i=0; i<n; ++i) {
  for(j=0; j<n; ++j) {
     A[i][j] = A[i-1][j-1] +
        A[i-1,j] + A[i-1,j+1];
  }
}
```

Mapping: `a[i][j]` $\mapsto$ `a[(j-i)%(n+1)]`



# Array contraction: symbolic unrolling, analysis, mapping

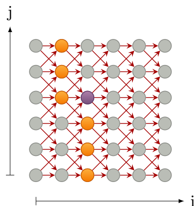
```
c[0] = 0;
for(i=0; i<n; i++) {
  c[i+1] = c[i] + ...;
}
```

⇒

```
c = 0;
for(i=0; i<n; i++) {
  c = c + ...;
}
```

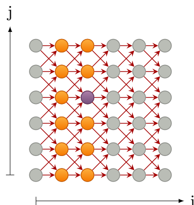
```
for(i=0; i<n; ++i) {
  for(j=0; j<n; ++j) {
     A[i][j] = A[i-1][j-1] +
        A[i-1,j] + A[i-1,j+1];
  }
}
```

Mapping: `a[i][j]` ↦ `a[(j-i)%(n+1)]`




```
for(i=0; i<n; ++i) {
  forpar(j=0; j<n; ++j) {
     A[i][j] = A[i-1][j-1] +
        A[i-1,j] + A[i-1,j+1];
  }
}
```

Mapping: `a[i][j]` ↦ `a[i%2][j]`



# Liveness at a given "step" with iscc

```
# Inputs
Params := [n] -> { : n >= 0 };
Domain := [n] -> { S[i,j] : 0 <= i, j < n };
Read := [n] -> { S[i,j] -> A[i-1,j-1]; S[i,j] -> A[i-1,j];
                 S[i,j] -> A[i-1,j+1] } * Domain;
Write := [n] -> { S[i,j] -> A[i,j] } * Domain;
Sched := [n] -> { S[i,j] -> [i,j] };

# Operators
Prev := { [i,j]->[k,l]: i<k or (i=k and j<l) };
Preveq := { [i,j]->[k,l]: i<k or (i=k and j<=l) };
WriteBeforeTStep := (Prev^-1).(Sched^-1).Write;
ReadAfterTStep := Preveq.(Sched^-1).Read;

# Liveness and conflicts
Live := WriteBeforeTStep * ReadAfterTStep;
Conflict := (Live^-1).Live;
Delta := deltas Conflict;
```

$$\begin{aligned}\text{Delta}(n) = \quad & \{(1, i_1) \mid i_1 \leq 0,\ n \geq 3,\ i_1 \geq 1 - n\} \cup \\ & \{(0, i_1) \mid i_1 \geq 1 - n,\ n \geq 2,\ i_1 \leq -1 + n\} \cup \\ & \{(-1, i_1) \mid i_1 \geq 0,\ n \geq 3,\ i_1 \leq -1 + n\}\end{aligned}$$

# Generalizations? Liveness sets not the right concept

Inner parallelism Almost the same.

# Generalizations? Liveness sets not the right concept

Inner parallelism Almost the same.

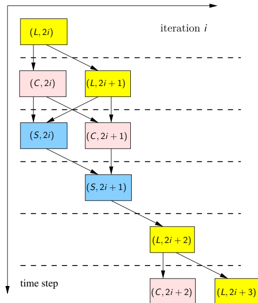
Seq/Par nested loops Can use a careful hierarchical approach.

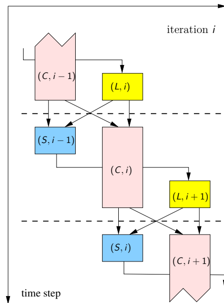
# Generalizations? Liveness sets not the right concept

Inner parallelism Almost the same.

Seq/Par nested loops Can use a careful hierarchical approach.

Software pipelining Harder to get a concept of "time".




On the right, values computed in $S(i-1)$ and $L(i+1)$ both conflict with those in $(C, i)$, but not with each other. Not a clique.

# Reasoning at the level of traces

Define:

- $a \in t$ iff $a$ is executed in a trace $t$;
- $a \prec_t b$ iff $a \in t$, $b \in t$ and $a$ is executed before $b$ in $t$;
- $S_\exists(a, b)$ iff there is a trace $t$ such that $a \prec_t b$.
- $R_\forall(a, b) = \neg S_\exists(b, a)$ iff, for all traces $t$, $a, b \in t$ implies $a \prec_t b$.

Then, $a$ and $b$ conflict ($a \bowtie b$) if, for some trace $t$, $W_a \prec_t W_b \prec_t R_a$.

# Reasoning at the level of traces

Define:

- $a \in t$ iff $a$ is executed in a trace $t$;
- $a \prec_t b$ iff $a \in t$, $b \in t$ and $a$ is executed before $b$ in $t$;
- $S_\exists(a, b)$ iff there is a trace $t$ such that $a \prec_t b$.
- $R_\forall(a, b) = \neg S_\exists(b, a)$ iff, for all traces $t$, $a, b \in t$ implies $a \prec_t b$.

Then, $a$ and $b$ conflict ($a \bowtie b$) if, for some trace $t$, $W_a \prec_t W_b \prec_t R_a$.

Conservative approximations for $a \bowtie b$:

- iff $S_\exists(W_a, R_a)$, $S_\exists(W_a, W_b)$, $S_\exists(W_b, R_a)$ iff $\neg R_\forall(R_a, W_a)$, $\neg R_\forall(W_b, W_a)$, $\neg R_\forall(R_a, W_b)$.
- with an under-approximation $\underline{R}_\forall \subseteq R_\forall$.

# Reasoning at the level of traces

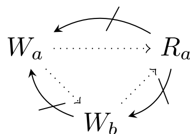
Define:

- $a \in t$ iff $a$ is executed in a trace $t$;
- $a \prec_t b$ iff $a \in t$, $b \in t$ and $a$ is executed before $b$ in $t$;
- $S_\exists(a, b)$ iff there is a trace $t$ such that $a \prec_t b$.
- $R_\forall(a, b) = \neg S_\exists(b, a)$ iff, for all traces $t$, $a, b \in t$ implies $a \prec_t b$.

Then, $a$ and $b$ conflict ($a \bowtie b$) if, for some trace $t$, $W_a \prec_t W_b \prec_t R_a$.

Conservative approximations for $a \bowtie b$:

- iff $S_\exists(W_a, R_a)$, $S_\exists(W_a, W_b)$, $S_\exists(W_b, R_a)$ iff $\neg R_\forall(R_a, W_a)$, $\neg R_\forall(W_b, W_a)$, $\neg R_\forall(R_a, W_b)$.
- with an under-approximation $\underline{R}_\forall \subseteq R_\forall$.




When $\underline{R}_\forall$ is a partial order $\preceq$, $a \bowtie b$ iff $\boxed{R_a \not\preceq W_a,\ W_b \not\preceq W_a,\ R_a \not\preceq W_b}$.

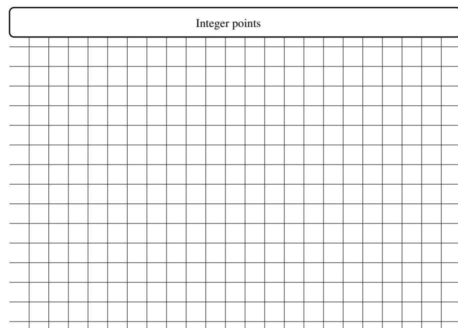
☛ Covers sequential code, OpenMP-like loop parallelism, OpenMP-4.0 task parallelism, X10, OpenStream, even some form of if conditions, etc.

# Modular mappings and lattices

**Modulo mapping** $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

**Validity** iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

**Lattice** An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



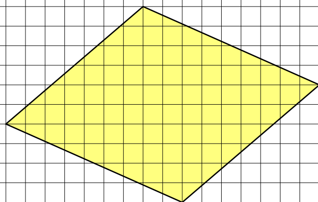

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



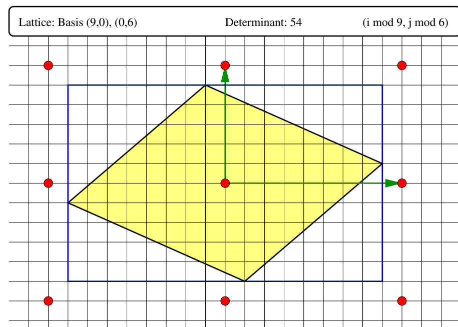

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



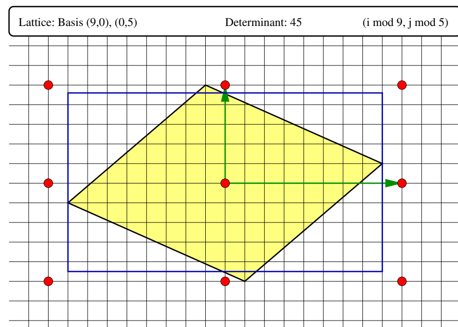

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



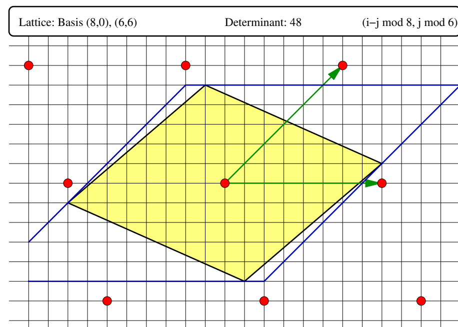

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



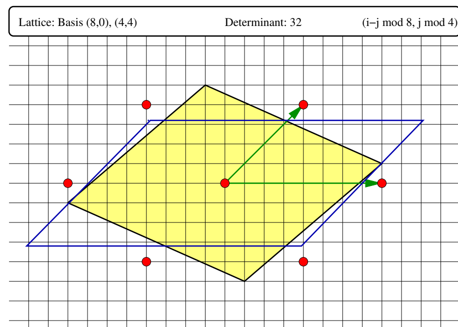

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



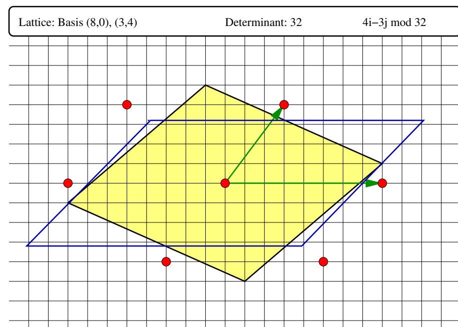

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



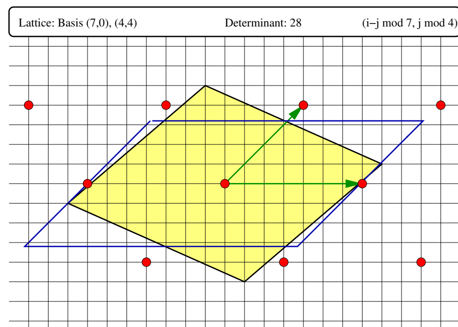

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).



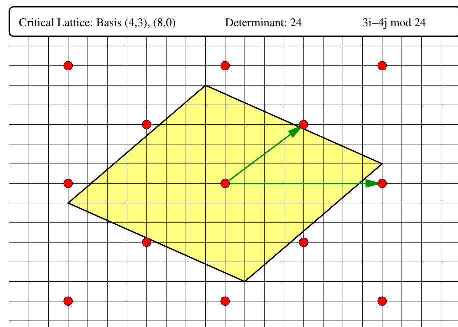

- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Modular mappings and lattices

Modulo mapping $\vec{i} \mapsto \sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (modulo componentwise).

Validity iff $\vec{i} \bowtie \vec{j}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$ iff, with $\mathrm{DS} = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, $\mathrm{DS} \cap \ker \sigma = \{\vec{0}\}$.

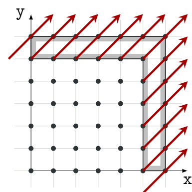
Lattice An allocation is optimal iff its kernel is a strictly admissible (integer) lattice for DS of minimal determinant (critical lattice).




- Successive modulo approach.
- Exhaustive search possible.
- Upper/lower bounds linked to Minkowski's theorems, basis reduction, gauge functions.
  ☛ good order of magnitude if *DS* is a polyhedron.

# Dealing with union of polyhedra: new theory

Live-out set of a tiled code:

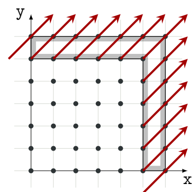



- Successive modulo:
  $(x, y) \mapsto (x \bmod N, y \bmod N)$.
- Skewed mapping:
  $(x, y) \mapsto (x - y \bmod (2N - 1), y \bmod 2)$.

☛ How to find the second one?

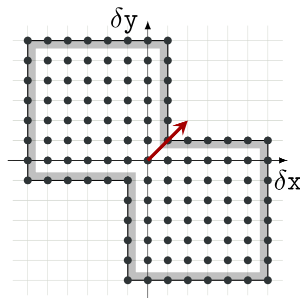
# Dealing with union of polyhedra: new theory

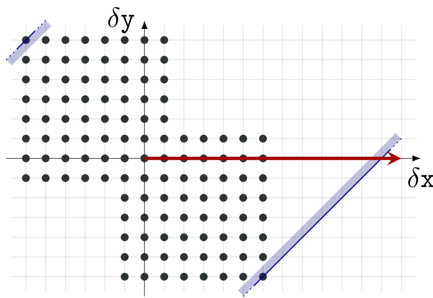
Live-out set of a tiled code:




- Successive modulo:
  $(x, y) \mapsto (x \bmod N, y \bmod N)$.
- Skewed mapping:
  $(x, y) \mapsto (x - y \bmod (2N - 1), y \bmod 2)$.

☛ How to find the second one?



# Many pieces of the puzzle get together

New generalizations and links with previous approaches.

- Liveness analysis for parallel specifications.
- Interference graph structure analysis and exploitation.
- Lattice-based memory allocation extensions.

☛ Towards a better understanding of parallel languages: semantics, static analysis, and links with the runtime.

And to conclude on tiling:

- Many other extensions exist: tile shapes, parallelism, cost models, etc. See talks by Uday, Sven, Ram.
- Still a need for performance models to guide transformations, parametric tiling is one step. See talk by Markus on perf. models.
- Need to integrate user/algorithm freedom. Ex: domain decomposition to define parallel tiles, multipartitioning for ADI codes, etc.