Le modèle polyédrique
“avec les mains”

Steven Derrien, Université de Rennes 1
(with contributions from A. Morvan and P. Quinton)
CAIRN research Group, IRISA/INRIA

Part I : Introduction

Outline

1. Overall context
   1. Compiling for multi-core machines
   2. Compiling for power-efficient embedded systems
2. Loop and data-layout transformations
   1. Shift, Interchange, Fusion/Fission, Skewing, Tiling, etc.
   2. Array expansion, contraction, slicing, etc .
3. Wrapping up example
   1. Image processing kernel example

Goal of this talk

- What you will find in this talk
  - A brief explanation of why loop transformations are useful
  - An overview of most common loop & layout transformations
  - A presentation of the key ideas used in polyhedral compilation
  - Probably some typos ;)

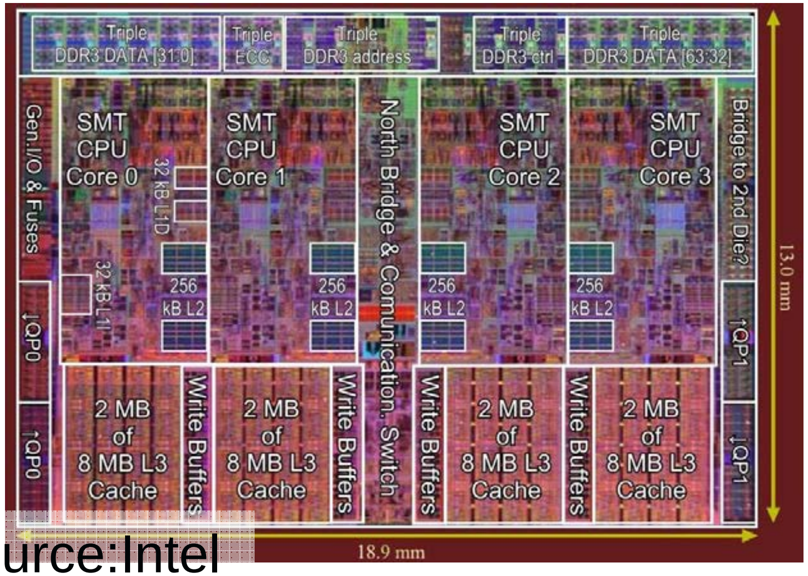
- What you will NOT find in this talk
  - An in-depth tutorial on the polyhedral model

  Got to http://labexcompilation.ens-lyon.fr/polyhedral-school/

- What you MAY find in this talk
  - Some inspiration to try by yourself what state-of-the art polyhedral compilation are **now** capable of ...

Multi-core processor architectures

- Nehalem : Intel Core i7
  - Four processor core + shared L3 cache with coherency



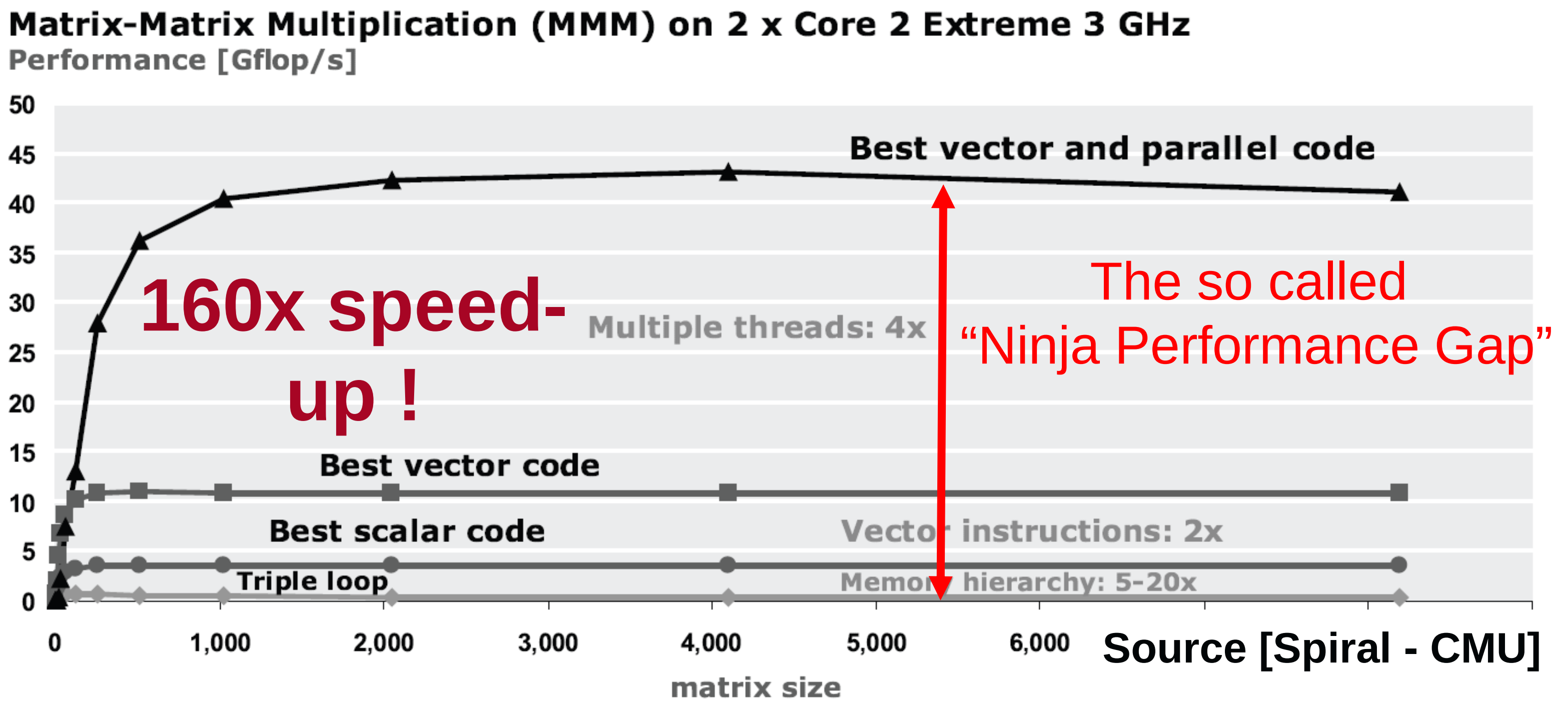
Source:Intel

- Simultaneous Multithreading (2 threads/core)
- SIMD instruction set with 128 bits registers (SSE4)

- Main programming model is thread level parallelism
  - Using openMP, pthreads, …
  - SIMD is handled by the compiler back-end

Program optimizations & performance

- Impact of optimizations on performance



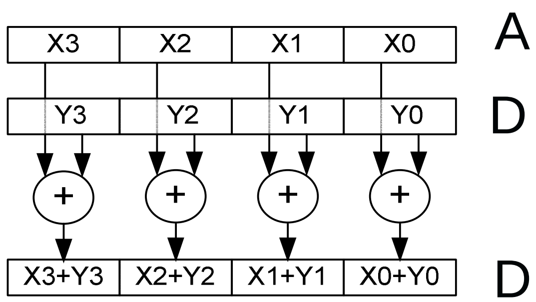
Source [Spiral - CMU]

- Origin of improvements
  – Parallelism (thread x SIMD): 8x - Memory optimization: 5x-20x !

SIMD short width vector instructions

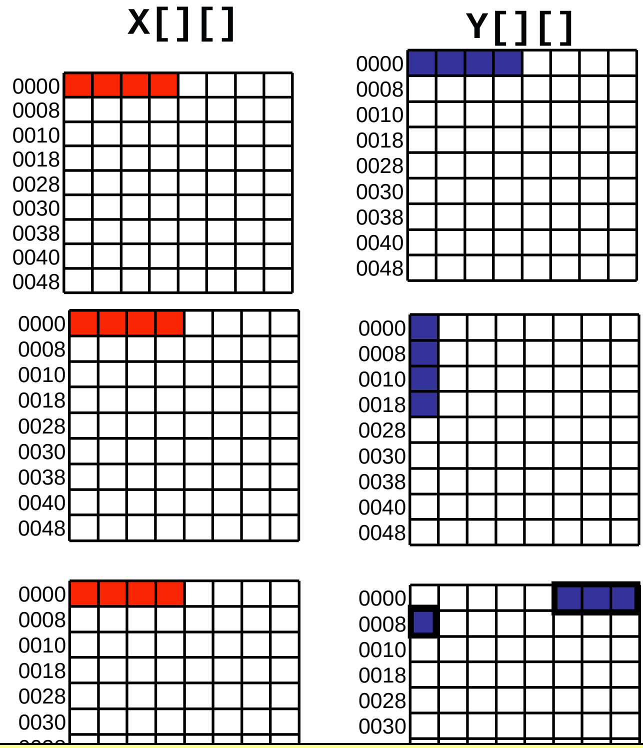
- Expose vector level parallelism in the ISA
  - Initially for regular (8bits, 16bits data) multimedia kernels
  - Extended to support floating point (Intel SSE, AVX)
  - Very challenging for compilers !
- Example from SSE : **`ADDPS`** ***`xmm1, xmm2/m128`***
  - **`m128`** : 16 bytes aligned memory location,
  - **`xmm0-7`** : 128 bit SSE registers
- Operation

```
D[31-0]  :=D[31-0]  +A[31-0];
D[63-32] :=D[63-32] +A[63-32];
D[95-64] :=D[95-64] +A[95-64];
D[127-96]:=D[127-96]+A[127-96];
```



SIMD instructions : layout constraints

- SIMD memory access = only contiguous data in memory
  - Unaligned accesses (64/128 bits) are not supported or cause performance penalties




```
for(i=0;i<8;i++) {
  for(j=0;j<8;j++) {
    X[8*i+j]+=A[j]*Y[8*i+j];
  }
}
```

Efficient SIMD vectorization

```
for(i=0;i<8;i++) {
  for(j=0;j<8;j++) {
    X[8*i+j]+=A[j]*Y[8*j+i];
  }
}
```

No SIMD because of the Y[j][i] **non contiguous** access pattern

```
for(i=0;i<8;i++) {
  for(j=0;j<8;j++) {
    X[8*i+j]+=A[j]*Y[8*i+j+5];
  }
}
```

Inefficient vectorization (unaligned access)

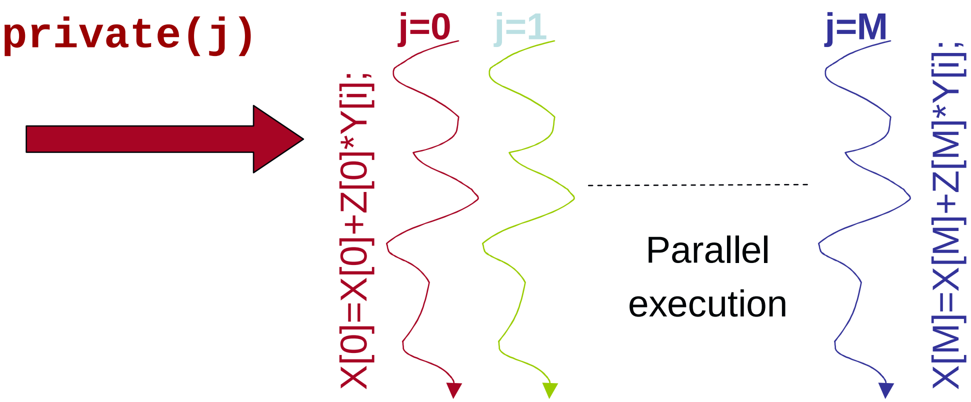
How to transform loops (and possibly data organization) to enable efficient SIMD vectorization ?

Thread level parallelism (OpenMP)

- OpenMP = simple way to expose thread level parallelism
  - Through coMPIler directives in the user source code (#pragma)
  - Targeted toward shared memory machine models
- Example : **`#pragma omp parallel for`**
  - Every j iteration can be executed by its own thread.
  - Threads synchronize at the end of the loop.

```
for(i=0;i<N;i++) {
#pragma omp parallel for private(j)
  for(j=0;j<M;j++) {
    X[j]=X[j]+Z[j]*Y[i];
  }
}
```

```
for(i=0;i<N;i++) {
```




```
  « sync »
}
```

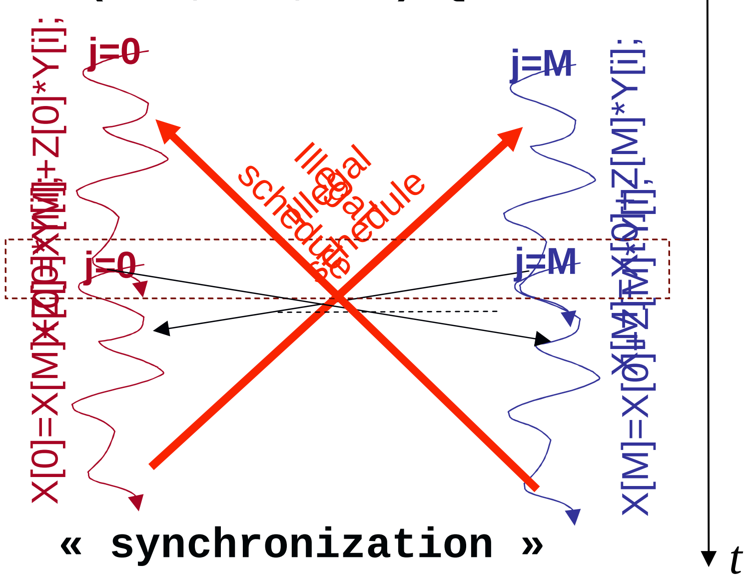
Data race issues in thread level parallelism

- The relative execution order of threads is not known
  - Dynamically determined by the OS scheduler
- The program execution may exhibit “data races”
  - When thread *x* reads a memory cell written by thread *y*
  - *Read can happen before write (or the other way round)*

```
for(i=0;i<N;i++) {
#pragma omp parallel for private(j)
  for(j=0;j<M;j++) {
    X[j]=X[M-j]+Z[j]*Y[i];
  }
}
```

How to guarantee the absence of data race in a OpenMP program ?

```
for(i=0;i<N;i++) {
```




```
}
```

Synchronization cost in Thread level parallelism

- The runtime forks threads and wait till their completion
  - This has obvious performance overhead.

```
for(i=0;i<N;i++) {
#pragma omp parallel for private(j)
  for(j=0;j<4;j++) {
    X[j]=X[j]+Z[j]*Y[i];
  }
}
```

The thread parallel version is very likely to be slower than the sequential one

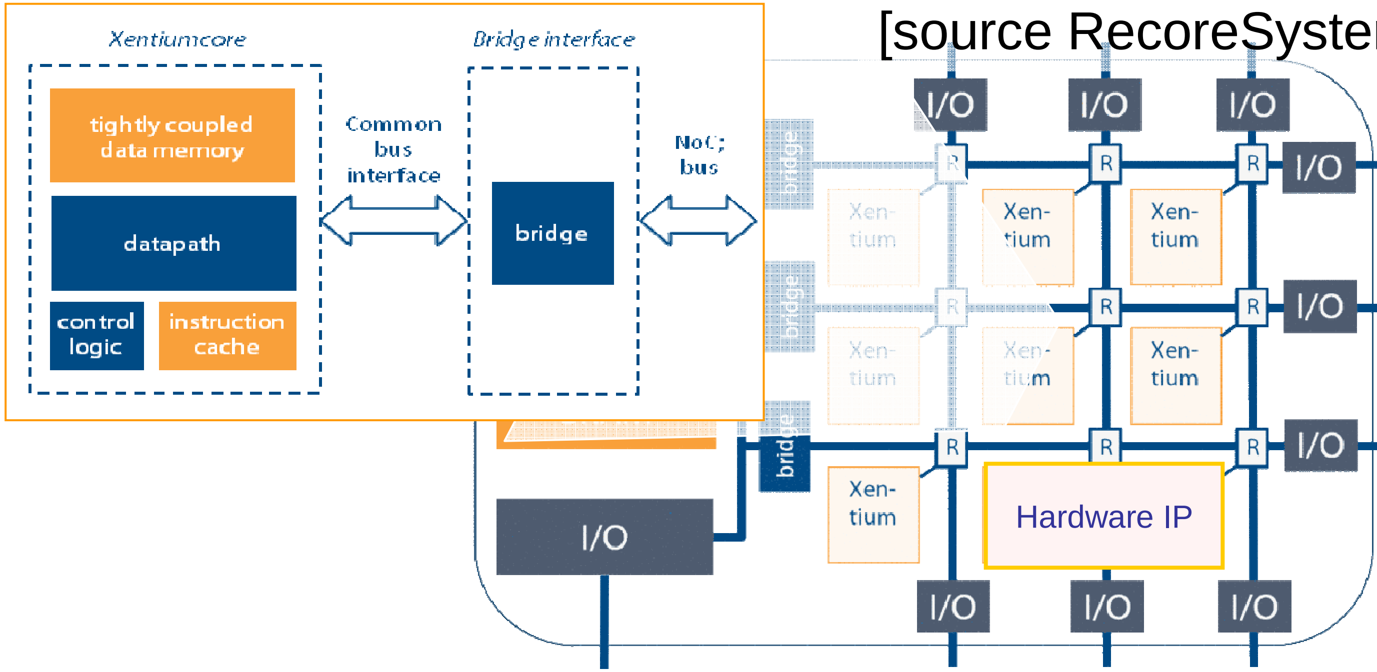
- Need to expose « coarser grain » parallelism.
  - Minimize the frequency of synchronization operations
  - Partition the computations in ***large*** independent “chunks”.
  - Pay attention to memory hierarchy (spatial/temporal locality)

How to perform (efficient) automatic parallelization ?

Embedded many-core/MPSoC

- Power efficient heterogeneous parallel architecture
  - Various type of PEs interconnected through a network-on-a-Chip

[source RecoreSystems]

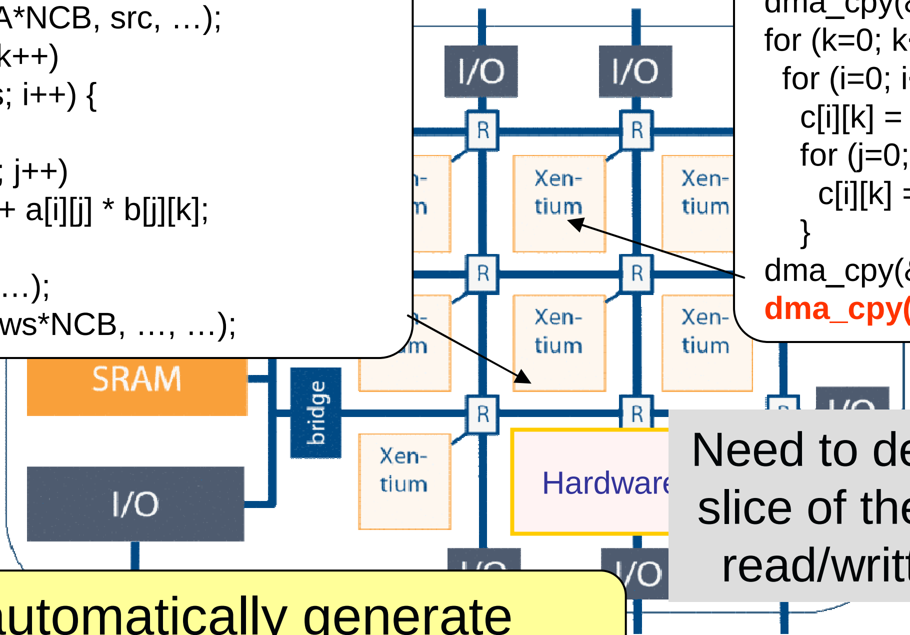


- Distributed Scratchpad Memory programming model
  - Global shared memory with software managed local memories

Distributed Scratchpad memory model

- Processors only work on local scratchpad memory
  - Global memory used to synchronize and exchange data
  - Scratchpad content is managed by the programmer (DMA)

```
dma_cpy (&p, 1, src, …);
dma_cpy(&a, p.rows*NCA, src, …);
dma_cpy(&b, NCA*NCB, src, …);
for (k=0; k<NCB; k++)
  for (i=0; i<p.rows; i++) {
    c[i][k] = 0.0;
    for (j=0; j<NCA; j++)
      c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }
dma_cpy (&p, ..,, …);
dma_cpy(&c, p.rows*NCB, …, …);
```



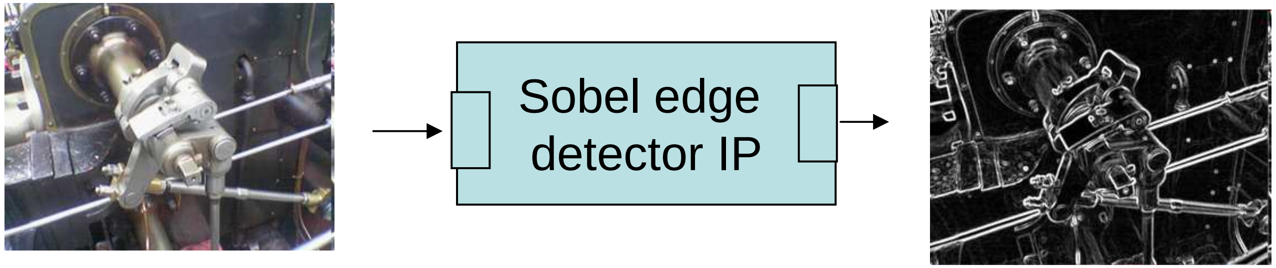

```
dma_cpy(&p, 1, src, …);
dma_cpy(&a, p.rows*NCA, src, …);
dma_cpy(&b, NCA*NCB, src, …);
for (k=0; k<NCB; k++)
  for (i=0; i<p.rows; i++) {
    c[i][k] = 0.0;
    for (j=0; j<NCA; j++)
      c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }
dma_cpy(&p, 1,, …);
dma_cpy(&c, p.rows*NCB,, …);
```

Need to determine accurately the slice of the data that needs to be read/written to global memory.

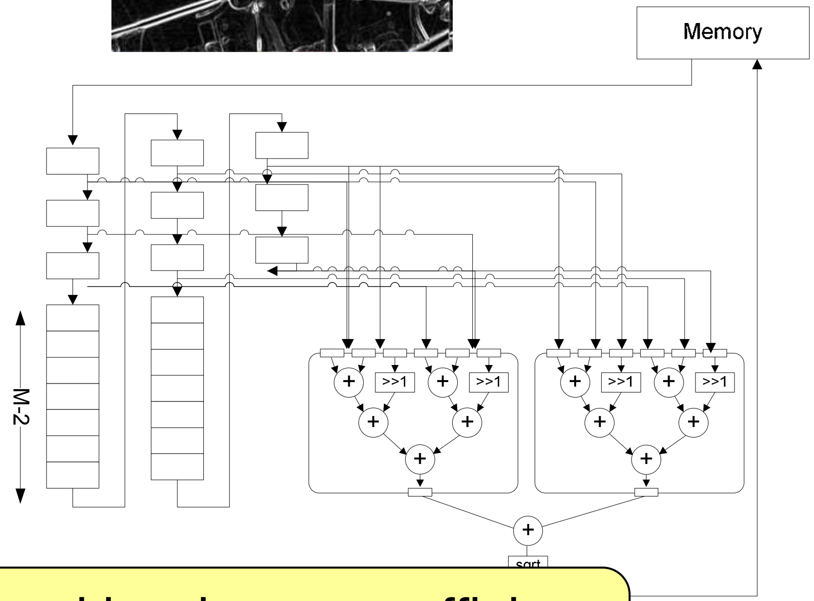
How to automatically generate (efficient) DMA code?

High Level Synthesis

- Generating custom hardware from C/C++
  - HLS tools help boosting designers productivity by up to 5x-10x !




```
void image(char in[M][N], char out[M][N]) {
  for(int i=1;i<N-1;i++) {
    for(int j=0;j<M;j++) {
S0:     Gx=in[i][j+1]+2*in[i-1][j+1]+in[i+1][j+1]+
           in[i][j-1]+2*in[i-1][j-1]+in[i+1][j-1];
S1:     Gy=in[i+1][j-1]+2*in[i+1][j]+in[i+1][j-1]+
           in[i-1][j-1]+2*in[i-1][j]+in[i-1][j-1];
S2:     out[i][j]= sqrt(Gx*Gy);
    }
  }
}
```



How to make automatically synthesized hardware as efficient as manually designed circuits ?

Outline

1. Overall context
   1. Compiling for multi-core machines
   2. Compiling for power-efficient embedded systems
2. Loop and data-layout transformations
   1. Shift, Interchange, Fusion/Fission, Skewing, Tiling, etc.
   2. Array expansion, contraction, slicing, etc .
3. Wrapping up example
   1. Image processing kernel example

Loop transformations, what for ?

- Improve performance and/or energy efficiency by …

- Exposing additional parallelism !
  - Thread level, SIMD, task level, etc …
- Improving the efficiency of the memory hierarchy
  - Spatial & temporal locality for registers, caches, TLB, disks, …

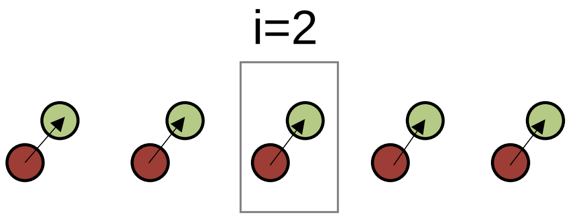
Loop shifting

- Delay an statement by a constant number of iterations
  - Increase instruction level parallelism by allowing pipelining.
  - Not always legal (must enforce data dependencies)

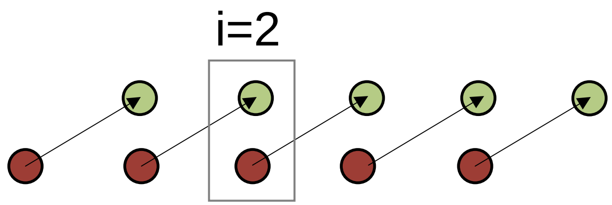
```
for(j=0;j<N;j++) {
S0:  Y[j]=foo(X[j]);
S1:  Z[j]=bar(Y[j]);
}
```

shifting **S0** by 1

```
Y[0]=Y[0]+X[0];
for(j=1;j<N;j++) {
S0: Y[j]=Y[j]+X[j];
S1: Z[j-1]=a*Y[j-1];
}
Z[N-1]=a*Y[N-1];
```



There is a RAW dependency on Y[j] between S0 and S1.



The dependency was removed : S0 and S1 can run in parallel

Loop fusion

- Merge several loops into a single one
  - Improve temporal locality of memory accesses
  - The transformation is not always possible

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
S0: Y[i,j]=foo(X[i,j]);
  }
}
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
S1: Z[i,j]=bar(Y[i,j]);
  }
}
```

fusion(S0,S1) →

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    Y[i,j]=foo(X[i,j]);
    Z[i,j]=bar(Y[i,j]);
  }
}
```

If Y[,] does not entirely fit in the cache, the second loop will suffer a ~100% cache miss rate.

Y[i,j] is reused immediately after its production, we have very good temporal locality

Loop distribution

- Split a single loop into several loops
  - Can expose parallelism in one of the loop

```
for(i=1;i<1000;i++){
S0: A[i]=A[i-1]+3
S1: C[i]=A[i]+5
}
```

distribute(S0,S1) →

```
for(i=1;i<1000;i++) {
S0: A[i]=A[i-1]+3;
}
#pragma omp parallel for
for(i=1;i<1000;i++) {
S1: C[i]=A[i]+5;
}
```

The iterations of the loop are not fully parallel (RAW on A[i-1]→A[i])

The iterations of the second loop are now fully parallel (but we degraded locality)

- Remark :
  - In general, there is a trade-off between parallelism and locality

Loop interchange

- Interchange two loop indices in a loop nest
  - May be used to expose parallelism or to improve locality
  - The transformation is not always possible

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    X[i]=X[i]+Y[i]*Y[j];
  }
}
```

Interchange(i,j) →

```
for(j=0;j<N;j++) {
  for(i=0;i<N;i++) {
    X[i]=X[i]+Y[i]*Y[j];
  }
}
```

The new inner loop is parallel

```
for(p=0;p<N;p++) {
  for(q=0;q<N;q++) {
    X[q][p]=a*X[q][p];
  }
}
```

Interchange(i,j) →

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    X[i][j]=a*X[i][j];
  }
}
```

X[i][j] has better ***spatial*** locality

Loop strip-mining

- Breaks an innermost loop into *chunks* of constant size

```
#define N=128
float **A,**B,**C;
for(i=0;i<N;i++){
  for(k=0;k<N;k++)
    for(j=0;j<N;j++){
S0:   C[i,j]+=A[i,k]*B[k,j];
  }
}
```

```
for(i=0;i<N;i++) {
  for(k=0;k<N;k++) {
    for(jj=0;jj<N;jj+=8)
      for(j=0;j<8;j++)
S0:     C[i,j+jj]+=A[i,k]*B[k,j+jj];
  }
}
```

Unrolling the innermost will help vectorizing the code

```
for(i=0;i<N;i++) {
  for(k=0;k<N;k++) {
    for(j=0;j<8;j++)
      for(jj=0;jj<N;jj+=8)
S0:     C[i,j+jj]+=A[i,k]*B[k,j+jj];
  }
}
```

The loop iterating over index j can be nicely distributed to 8 threads

Loop tiling

- Break the loops into « tiles » or blocks
  - Expose coarse grain parallelism & improve temporal data reuse
  - Legal only if all loop are permutable (i.e. can be interchanged)
  - Very effective parallelizing program transformation
- Classical example : the matrix product

```
float **A,**B,**C;
for(i=0;i<16;i++) {
  for(j=0;j<16;j++) {
    for(k=0;k<16;k++)
S0:   C[i,j]+=B[i,k]*A[k,j];
  }
}
```

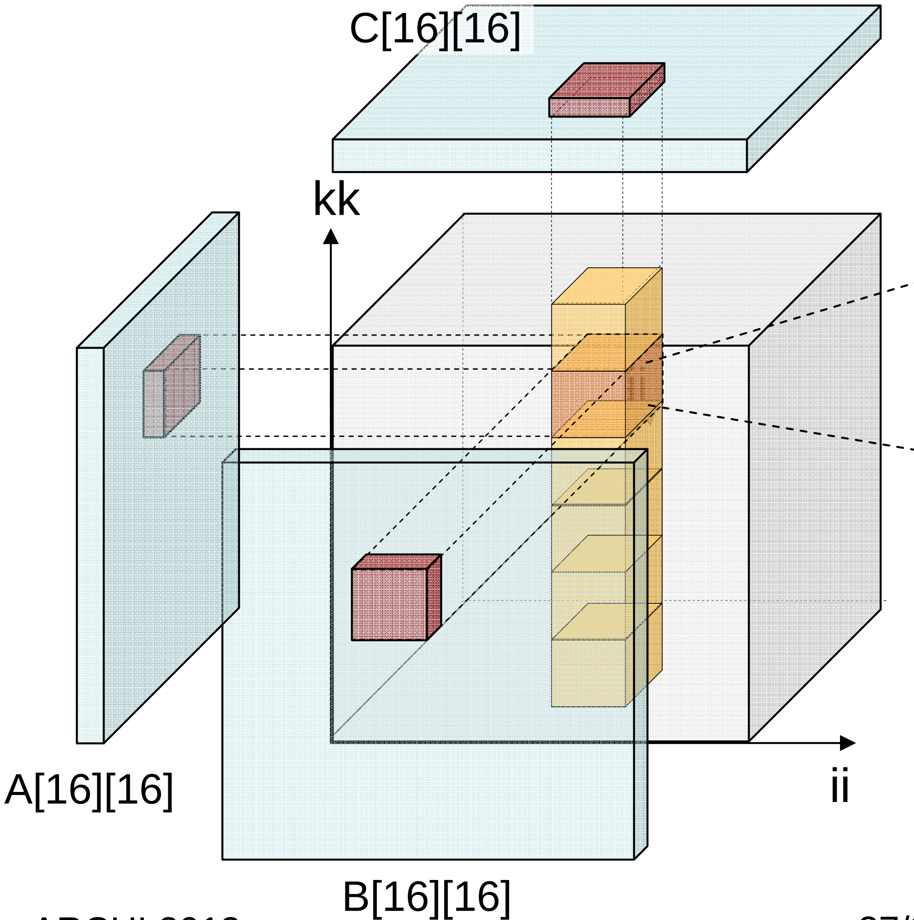
```
float **A,**B,**C;
for(ii=0;ii<16;ii+=4)
 for(jj=0;jj<16;jj+=4)
  for(kk=0;kk<16;kk+=4)
   for(i=0;i<4);i++)
    for(j=0;j<4;j++)
      for(k=0;k<4;k++)
S1:     C[i+ii,j+jj]+=
           B[i+ii,k+kk]*A[k+kk,j+jj];
```

A 4x4x4 Tile

Best understood with an visual representation …

Loop Tiling

- Tiling helps improving spatial and temporal locality
  - One chooses tile size such that all data fits into the cache

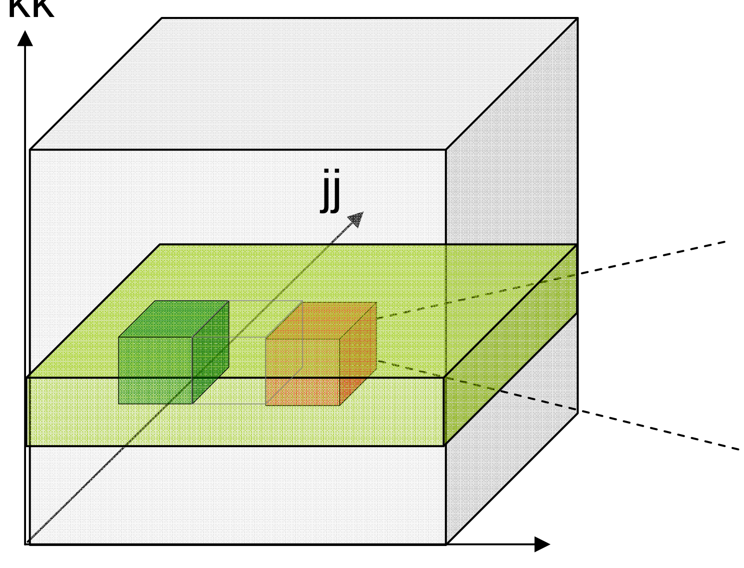



```
float **A,**B,**C;
for(ii=0;ii<16;ii+=4)
 for(jj=0;jj<16;jj+=4)
  for(kk=0;kk<16;kk+=4)
   for(i=0;i<4);i++)
    for(j=0;j<4;j++)
      for(k=0;k<4;k++)
S1:     C[i+ii,j+jj]+=
           B[i+ii,k+kk]*A[k+kk,j+jj];
```

A 4x4x4 Tile

Loop Tiling

- Simple way of exposing coarse grain parallelism
  - Tiles are executed as *atomic* execution units, there is no synchronization during a tile execution.
- Tiling enables efficient parallelization
  - It improves locality and reduces synchronization overhead
  - Finding the “right” tile size and shape is difficult (open problem)



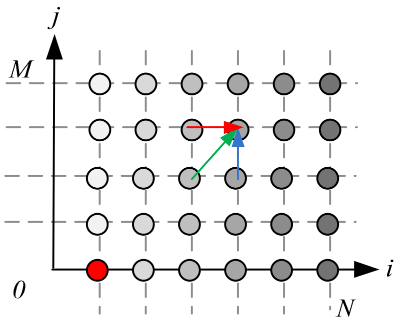

```
float **A,**B,**C;
#pragma omp parallel for private(ii)
for(ii=0;ii<16;ii+=4)
#pragma omp parallel for private(jj)
for(jj=0;jj<16;jj+=4)
  for(kk=0;kk<16;kk+=4)
     for(i=0;i<4;i++)
      for(j=0;j<4;j++)
       for(k=0;k<4;k++)
S1:      C[i+ii,j+jj]+=
            B[i+ii,k+kk]*A[k+kk,j+jj];
```

A 4x4x4 Tile

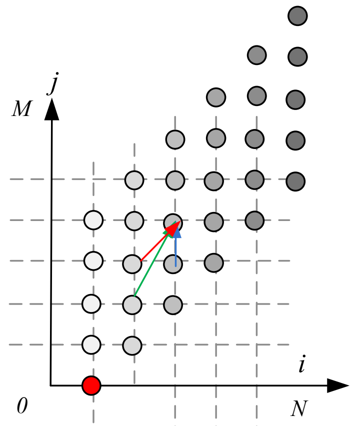
Loop skewing

- Shift the innermost loop *j* by the outermost loop index *I*
  - Changes array index expressions but not execution order

```
for(i=0;i<N;i++) {
 for(j=0;j<N;j++)
   S[i,j]=max(S[i-1,j-1]+A,
              S[i,j-1]+B
              S[i-1,j]+C);
}
```




```
for(i=0;i<N;i++) {
  for(j=i;j<N+i;j++)
     S[i,j]= max(S[i-1,j-i-1]+A,
                 S[i,j-i-1]+B
                 S[i-1,j-i]+C);
}
```



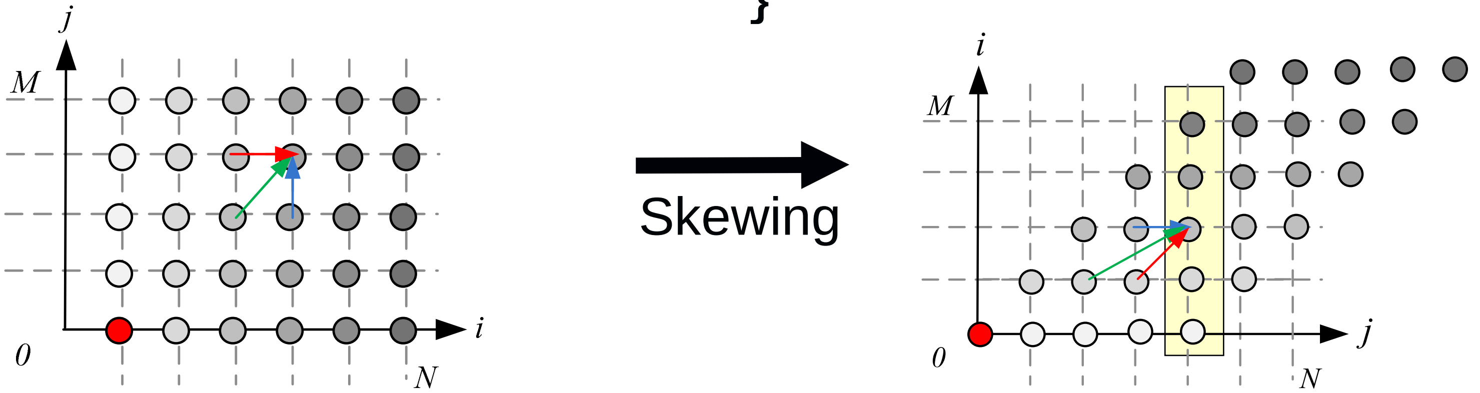
- Wait a minute, what's the use of this transformation?
  - None, unless used jointly with a loop interchange

Loop skewing + interchange

- Shift the innermost loop *j* by the outermost loop index *i*
- Then, interchange the innermost and outermost loops

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++)
    S[i,j]=max(S[i-1,j-1]+A,
               S[i,j-1]+B
               S[i-1,j]+C);
}
```

```
for(j=0;j<2*N-1;j++)
  for(i=max(0,N-j);i<min(N,j);i++)
    S[i,j]=max(S[i-1,j-i-1]+A,
               S[i,j-i-1]+B
               S[i-1,j-i]+C);
}
```



No dependencies between different iterations of a given j loop !

Data layout transformation, what for ?

- Optimizing memory size
  - Reducing statically allocated array sizes whenever possible
- Enabling parallel execution
  - Allocate extra memory space to enable parallel execution
- Improving the efficiency of software caches
  - Find which data set to move in a software controlled cache
- Communication synthesis in distrib. memory machines
  - Derive the set of data that needs to be transmitted from one processor to another.

Array privatization/expansion

- Motivating example

```
   for(i=0;i<N;i++) {
S0:  tmp = …
     for(j=0;j<=i;j++) {
S1:    tmp=tmp+X[j]*C[i][j];
     }
S3:  Y[i] = tmp;
   }
```

Parallel execution of the i loop lead to a data race on shared variable tmp.

The parallel execution becomes legal if each iteration j **owns its value of tmp** !

- Privatization = each parallel task owns a copy of the var.
  - **Remark** : openMP supports privatization (**private** directive)

```
   // expansion of tmp as tmp[N]
   for(i=0;i<N;i++) {
S0:  tmp[i] = …
     for(j=0;j<=i;j++) {
S1:    tmp[i]=tmp[i]+X[j]*C[i][j];
S3:  Y[i] = tmp[i];
   }
```

```
   #omp parallel for private i,j,tmp
   for(i=0;i<N;i++) {
S0:  tmp = …
     for(j=0;j<=i;j++)
S1:    tmp=tmp+X[j]*C[i][j];
S3:  Y[i] = tmp;
   }
```

Which variables/array to privatize ? How much expansion is needed ?

Array contraction

- For embedded systems with scarce memory resources
  - Replace a temporary array by a smaller one
  - We must find a new legal array size and addressing scheme

```
for(i=0;i<N;i++) {
  tmp[i,0]=foo(X[i]);
  for(j=1;j<N;j++) {
    tmp[i,j]=foo(X[i,j]);
    Z[i-1,j]=bar(tmp[i,j-1]);
  }
  Z[N-1,j]=bar(Y[N-1,j]);
}
```

tmp is a NxN array

```
for(i=0;i<N;i++) {
  tmp[0]=foo(X[i]);
  for(j=1;j<N;j++) {
    tmp[j%2]=foo(X[i,j]);
    Z[i-1,j]=bar(tmp[(j-1)%2]);
}
  Z[N-1,j]=bar(Y[N-1,j]);
}
```

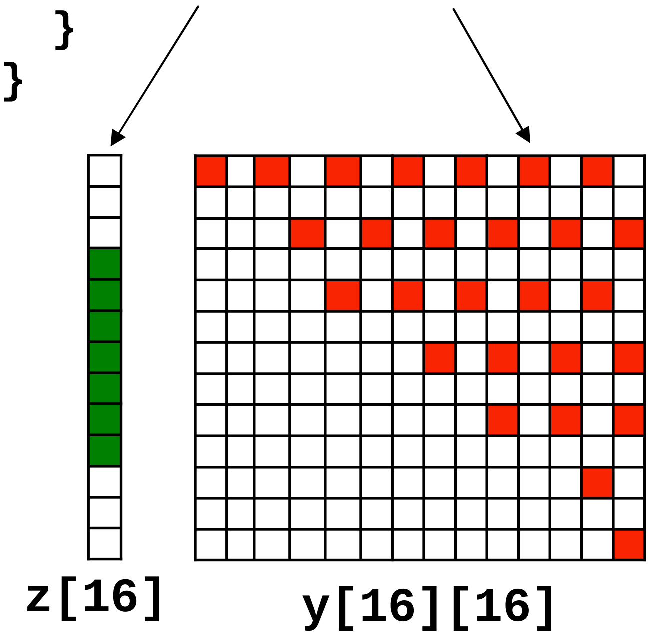
tmp is now a 2x1 array

- Very effective if combined with loop fusion !

Array slicing for scratchpad memory

- Scratchpad management require explicit copy operations
  - The programmer/compiler must figure out which data to load/save to/from the scratchpad memory.

```
for(i=0;i<8;i++) {
  for(j=0;j<i;i++) {
     Z[i-j+3]= Y[2*i][2*j]+…;
  }
}
```



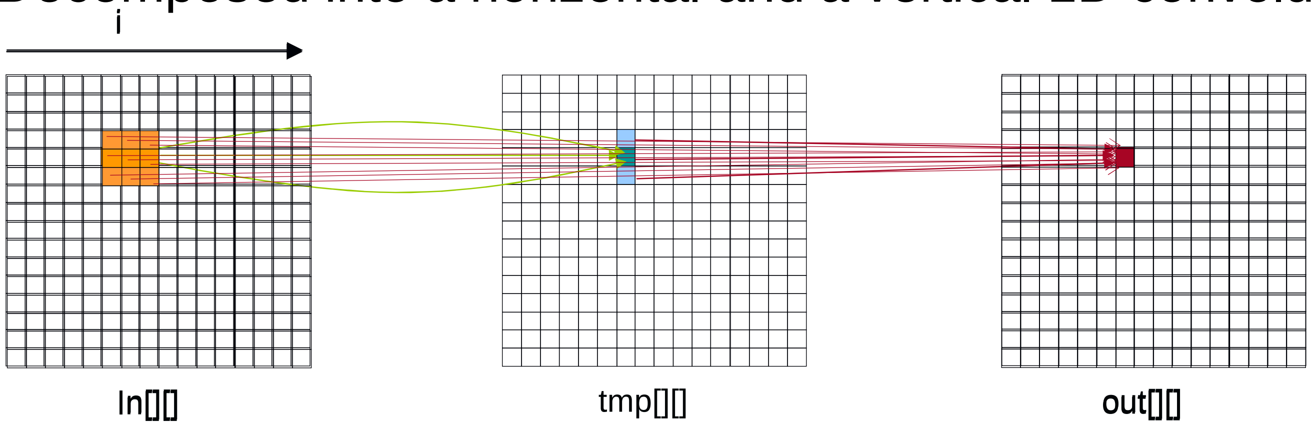

```
// copy to scratchpad
for(i=0;i<8;i++) {
  _z[i]=z[3+i];
  for(j=0;j<i;i++)
    _y[i][j]=Y[2*i][2*j];
}
// run the computations
for(i=0;i<8;i++)
  for(j=0;j<i;j++)
    _z[i-j]=_y[i][j]+…;
// writeback to main memory
for(i=0;i<8;i++) {
  z[3+i]=z[i];
  for(j=0;j<i;i++)
    y[2*i][2*j]=_Y[i][j];
}
```

Outline

1. Overall context
    1. Compiling for multi-core machines
    2. Compiling for power-efficient embedded systems
2. Loop and data-layout transformations
    1. Shift, Interchange, Fusion/Fission, Skewing, Tiling, etc.
    2. Array expansion, contraction, slicing, etc .
3. Wrapping up example
    1. Image processing kernel example

Image processing pipeline example

- Image filtering with separable 2D convolution kernel
  - Decomposed into a horizontal and a vertical 1D convolution



- A naïve implementation

```
void image(int M, int N, char in[M][N], char out[M][N]) {
  int tmp[M][N];
  for(i=1;i<N-1;i++)
    for(j=0;j<M;j++)
S0:      tmp[i][j]=f1(in[i][j],in[i-1][j],in[i+1][j]);

  for(i=1;i<N-1;i++)
    for(j=1;j<M-1;j++)
S1:      out[i][j]=f2(tmp[i][j],tmp[i][j-1],tmp[i][j+1]);
}
```

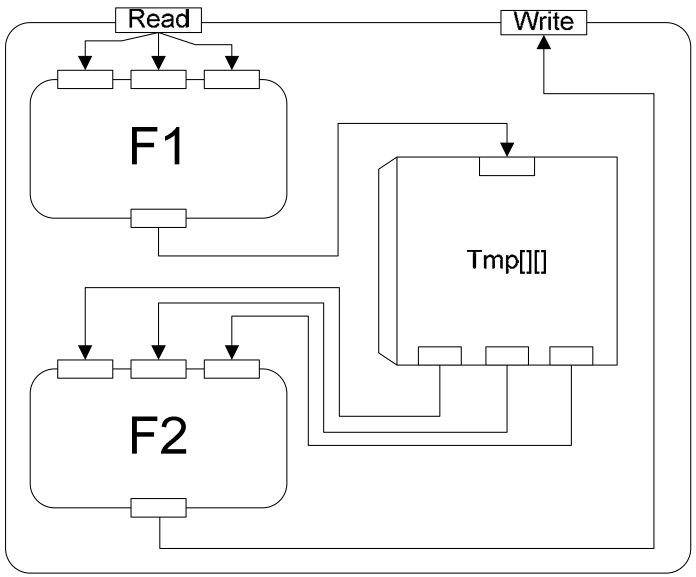
Horizontal filter

Vertical filter

Image processing pipeline example

- Why not synthesizing the kernel as custom hardware ?
  - By using a state of the art High Level Synthesis tool

```
void image(int M,N, char **in,char **out){
  int tmp[M][N];
  for(i=1;i<N-1;i++)
    for(j=0;j<M;j++)
S0:    tmp[i][j]=f1(in[i][j],
                    in[i-1][j],
                    in[i+1][j]);
  for(i=1;i<N-1;i++)
    for(j=1;j<M-1;j++)
S1:    out[i][j]=f2(tmp[i][j],
                    tmp[i][j-1],
                    tmp[i][j+1]);
}
```



- Results
  - 2.M.N clock cycles, O(MN) memory cost, 4.MN byte I/O mem access
  - Considering external I/O with 6 cycle access latency $\Rightarrow$24M.N cycles

Image processing pipeline example

- Loop fusion (with shifting)
  - Reduce clock cycle count from 2.*M.N*+ε to *M*.(*N*+*1*) +ε
- Array contraction
  - Reduces local buffer size form *M.N* to 3 !

```
void image(int M, int N, int in[M][N],int out[M][N]) {
  int tmp[3]; // local memory

  for (i = 1; i < N-1; i++)
  for (j = 0; j < 2; j++)
     if (j%8) buf =
S0:  tmp[j%3] = f(in[i][j], in[i-1][j], in[1+i][j]);
  for (j = 2; j < M; j++)
S0:  tmp[j%3] = f(in[i][j], in[i-1][j], in[1+i][j]);
S1:  out[i][j-1] = f(tmp[(j-1)%3],tmp[(j-2)%3],tmp[j%3]);
```

All these transformations can now be fully automated thanks to steady improvements in polyhedral coMPIlation

Part II : Hands-on !

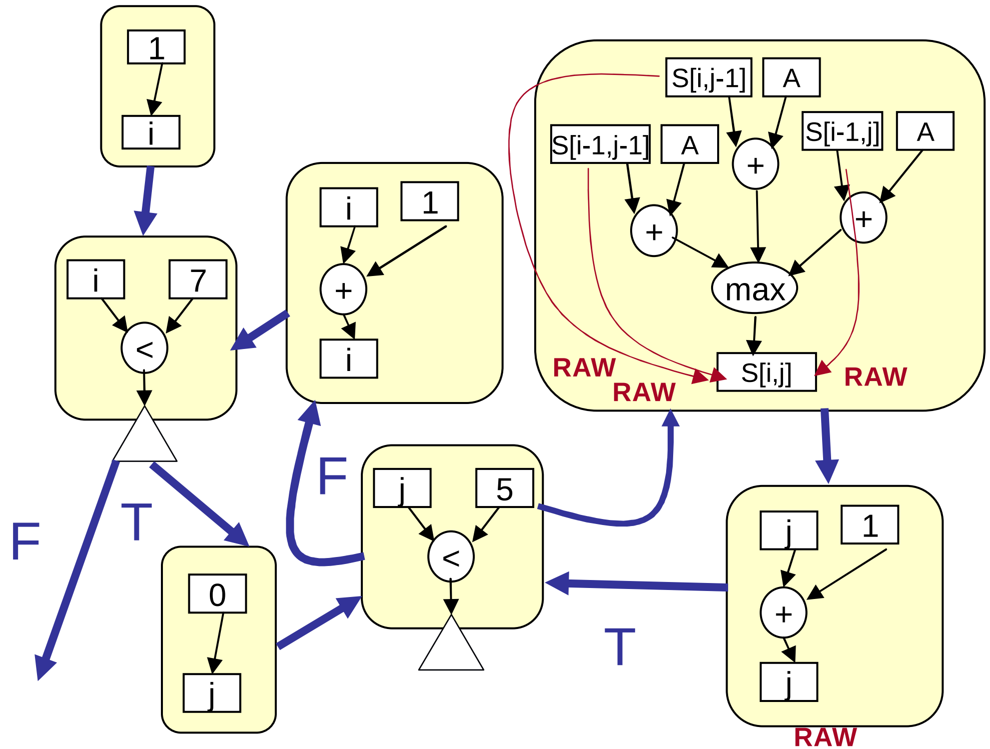
Outline

1. Representing & reasoning about loops in compilers
    1. CDFG & Expanded Dependence Graph (loops)
    2. The case for a compact instance wide representation
2. Polyhedral representation of Affine Control Loops
    1. Statement Iteration domains as polyhedral sets
    2. Lexicographic ordering (aka multi-dimensional time)
3. Polyhedral program transformations
    1. Loop transformations as affine transformations
    2. Composability of loop transformations
4. Semantic preserving schedules
    1. Dependence Analysis (memory vs value based) & PRDGs
    2. Checking the legality of a schedule

How to model loop nests in a coMPIler ?

- Control & Data Flow Dependence Graph
  - Does not capture the “regularity“ present in most loop nests.
  - Coarse dependency information between statements
  - Inter-iteration analysis is quite difficult (we don’t “*see”* for loops)

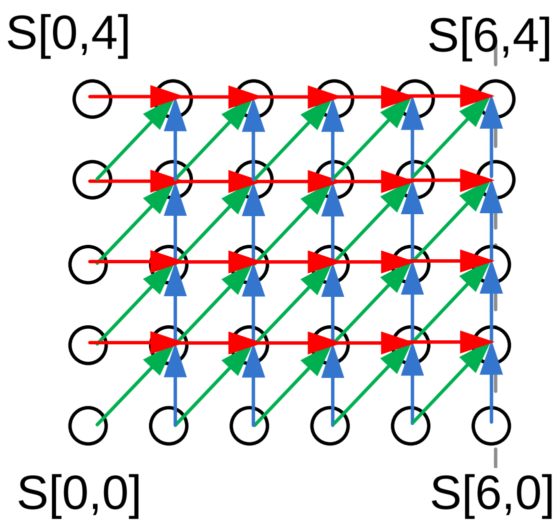
```
for(i=1;i<7;i++) {
  for(j=1;j<5;j++)
     S[i,j]= max(
           S[i-1,j-1]+A,
           S[i,j-1]  +B,
           S[i-1,j]  +C
         );
}
```



How to model loop nests in a compiler ?

- Use a dependence graph as in previous slides ?
  - Every iteration is represented as a vertex of the graph
  - Data dependencies are modeled as edges in the graph

```
for(i=1;i<7;i++) {
  for(j=1;j<5;j++)
    S[i,j]=max(
               S[i-1,j-1]+A,
               S[i,j-1]+B,
               S[i-1,j]+C
            );
}
```



- Limitations
  - Only for loop bounds known at compile time and not scalable

What do we need ?

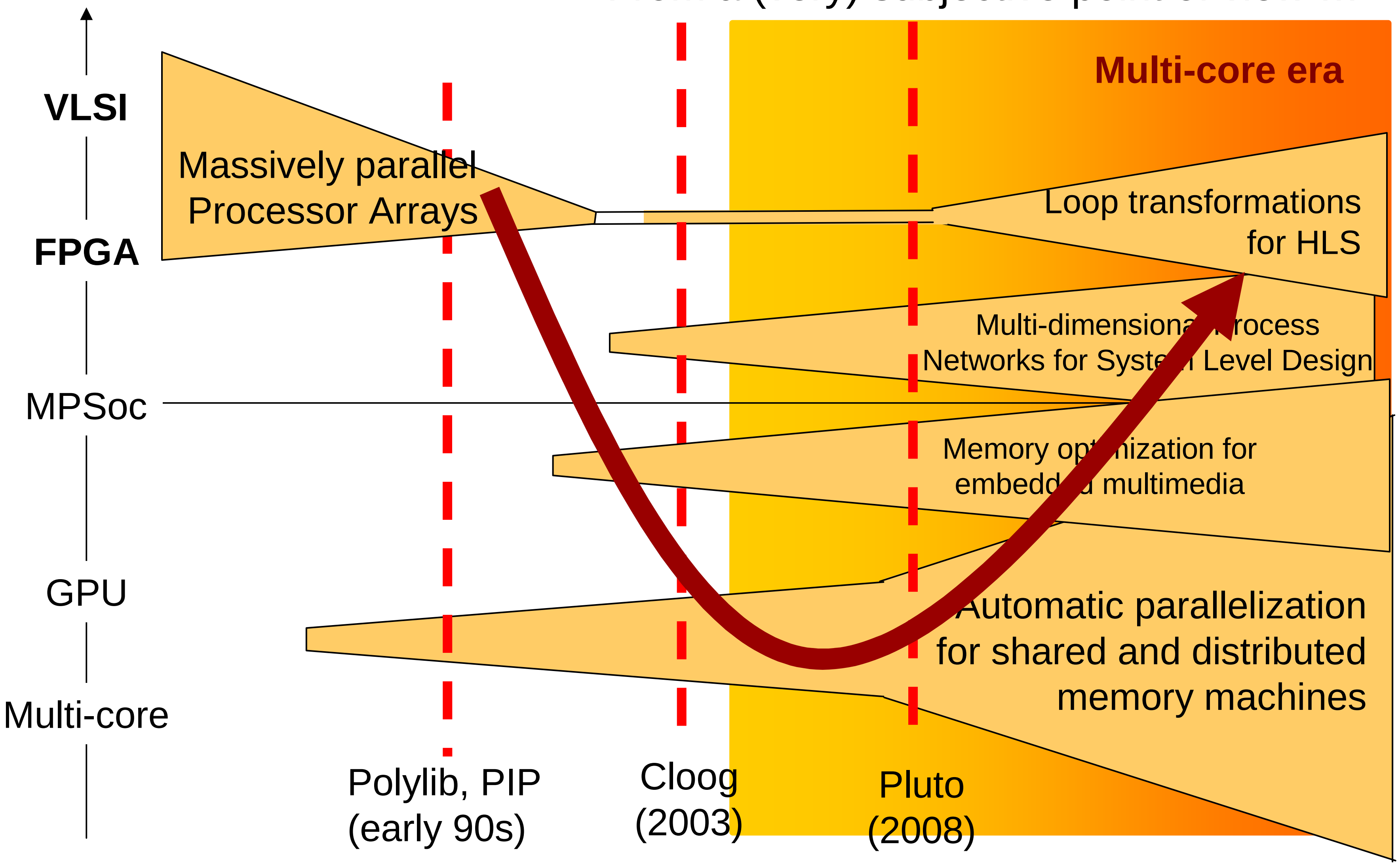
- We need a ***compact*** model which captures ***regularity***
  - Model *size* should be independent of loop iteration count
  - But it should not be restricted to simple/toy perfect loops

- We need ***instance wise*** dependency information
  - Dependency information for each execution of a loop statement

All of these requirements are fulfilled by polyhedral representations of programs

A short story of the polyhedral model

From a (very) subjective point of view …



Loop iterators and parameters

- **Loop iterator** = indices of loops surrounding the statement
- **Parameter** = variable whose value does not change during the whole loop nest execution (example : size of an image).

```
L1:
for(i=1;i<=10;i+=1){
  x[P+i-1]=i;
  for(j=1;j<=P;j+=1){
    z[j] = x[j] + x[j];
  }
  …
}
```

P is a parameter for the loop nest above

```
L2:
  for(i=1;i<=10;i+=1){
    x[size_x-1]=i;
L3: for(j=1;j<=Z;j+=1){
      z[j] = x[j] + x[j];
    }
    Z = … ;
  }
```

Z is not a parameter for the loop nest L2 above

… but **Z** is a parameter in the context of the single loop L3 !

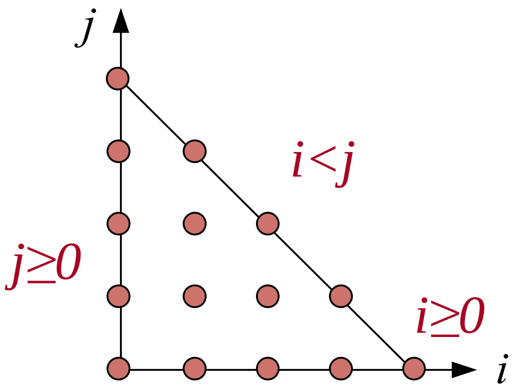
Notion of polyhedral iteration domain

- Iteration domain : model of all iterations of a loop nest
  - Modeled as a union of parameterized integer polyhedron

> Integer polyhedron = convex domain defined by a conjunction of affine constraints

  - Contraints bind together loop iterators and parameters

```
for(i=0;i<N;i++) {
  for(j=0;j<N-i;j++) {
    …
  }
}
```



$\{\ i,j \mid 0 \leq i < N \wedge 0 \leq j < i\ \}$

We can benefit from linear programming techniques !

A few important definitions

- Statement
  - Instruction/operation in the program source code. In a loop, a statement is executed several times.
- Statement Iteration vector
  - Vector made of the values of loop indices and parameters surrounding a statement execution (starting with outer loop).
- Statement instance
  - A particular execution of a statement. A statement instance can be identified by its corresponding iteration vector.
- Statement domain
  - A union of polyhedron representing all instances of a given statement S. We write it D_S.

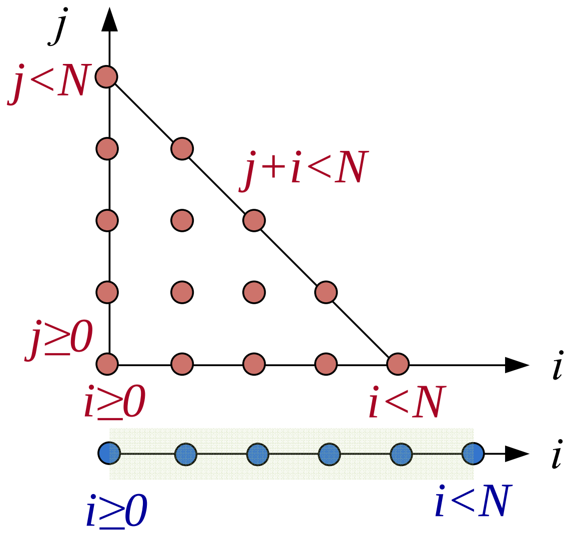
Statement Iteration domain

- Iteration set where a given statement S_i is executed.
  - Again modeled as a parameterized polyhedron using enclosing loop iterators and parameters as dimension indices.
  - The polyhedron constraints are constructed out of enclosing loop bounds and guards.
- Example

Loop iterators **i,j**

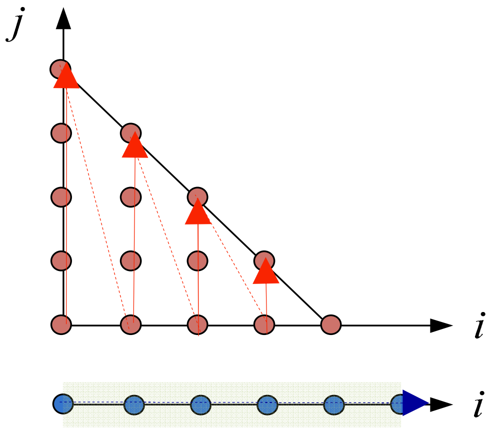
Parameters : **N**

```
  for(i=0;i<N;i++) {
S0:  Y[i] = …
     for(j=0;j<N;j++) {
       if(j<N-i)
S1:          Y[i]=Y[i]*X[i][j];
     }
  }
```



Notion of Lexicographical ordering

- Representing the set of iterations is not enough
  - We must model in which order computations are performed



S1(i,j) is executed after *S1(i,j-1)*

S1(i+1,j) is executed after *S1(i,j')* for all *j, j'*

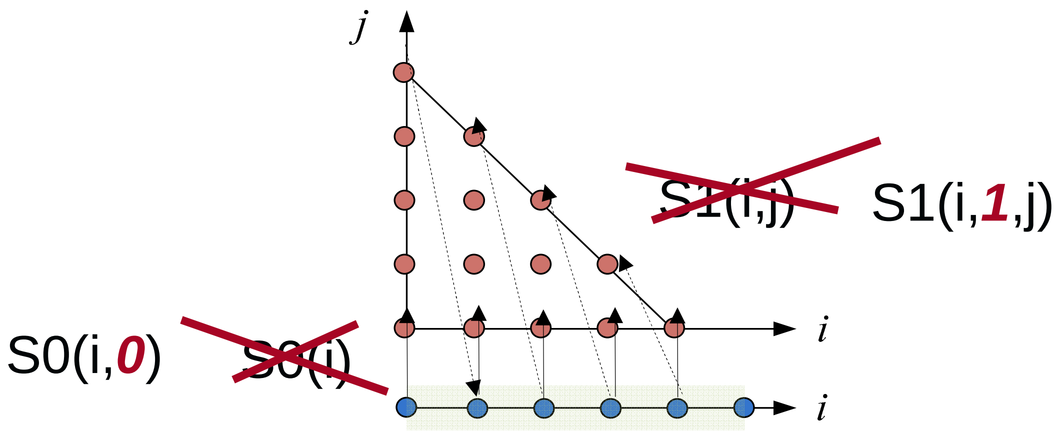
- We use lexicographic ordering ($\prec$) over the index set
  - Intuition : think of it as *hours, minutes, seconds*

$$\vec{x} = (x_1, \ldots, x_n) \qquad \vec{x} \prec \vec{y} \Leftrightarrow \bigvee_{i=1}^{N} \left(\left(\bigwedge_{k=i+1}^{N} x_k = y_k \right) \wedge x_i < y_i \right)$$
$$\vec{y} = (y_1, \ldots, y_n)$$

- In the following, we will write $S(i,j)$ as $S(\vec{x})$ with $\vec{x} = (i, j, \ldots)$

Notion of Lexicographical ordering

- Below, S1(i,0) is always executed after S0(i)
  - However we don't have $(i,0) \succ (i)$



- To model such textual order, we add ***scalar*** dimensions
  - They are artifact indices whose value are constants for a stmt

D_{S0}: { i,pos | $0 \le i < N \wedge pos=0$ }

D_{S1}: { i,pos,j | $0 \le i < N \wedge 0 \le j < N-i \wedge pos=1$ }

  - Now we have $(i,1,0) \succ (i,0)$ enforced
    - means a total order for all statement instances in the loop nest.

Notion of statement schedule

- The expression used for lex. ordering is a ***schedule***
  - It gives a time instant for each instance of the statement

```
    for(i=0;i<N;i++) {
S0:   Y[i] = …
      for(j=0;j<=i;j++) {
S1:     tmp=Y[i]*X[i][j];
S2:     Y[i]=…
      }
S3:   res = Y[N-i] + Y[i]
    }
```

- Statement S0 schedule is (i,0)
- Statement S1 schedule is (i,1,j,0)
- Statement S2 schedule is (i,1,j,1)
- Statement S3 schedule is (i,2)

- Loop transformations can be seen as a change of schedule
  - We will restrict to quasi-affine schedule transformations
  - Quasi-affine schedule can express most loop transformations
    - Also enables complex compositions of loop transformations

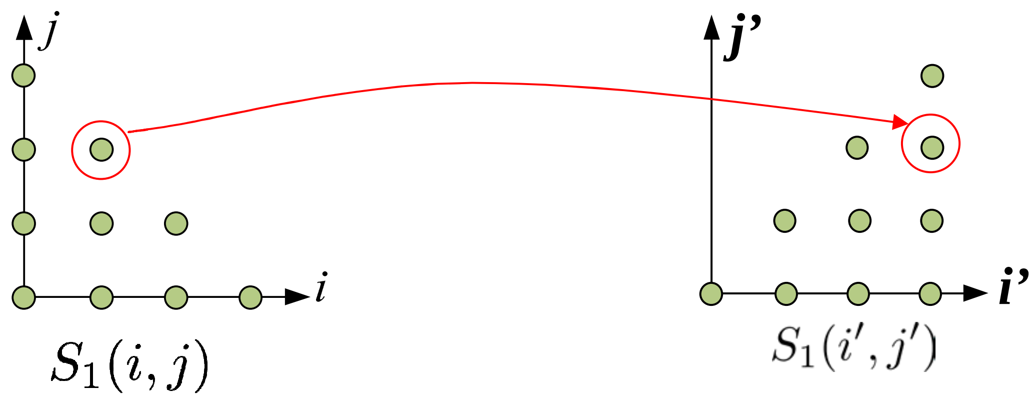
Notion of statement scheduling

- Transforming a loop = rescheduling its stmt instances
  - We map every instance $S_i(\vec{x})$ to a new index space $S_i(\vec{x}' = \Theta(\vec{x}))$
  - The mapping is expressed using affine functions

$$S_0(\vec{x}) \rightarrow S_i(\Theta_{S_i}(\vec{x})) \qquad \Theta_{S_i}(\vec{x}) = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & & \vdots \\ a_{p,1} & \dots & a_{p,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

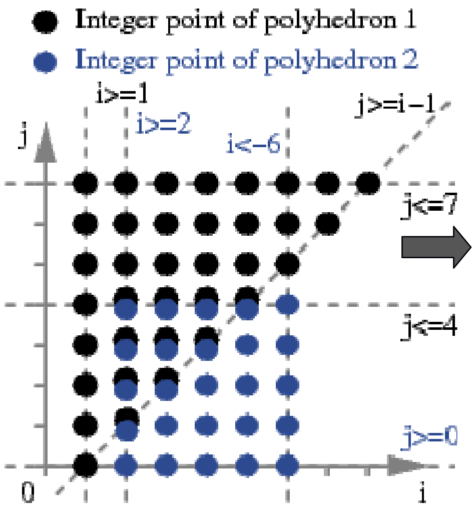
> $\Theta_{S_i}(\vec{x})$ is the scheduling (or scattering) function for $S_i(\vec{x})$

- Scheduling = "affine" transformation of the domain



Scheduling and code generation

- We must regenerate code for the transformed index set !
  - Sequence of loops which scans the transformed domains
  - Known as the polyhedron scanning problem
- Example : ClooG code generator [1]




```
for (i=1;i<=8;i++) {
  for (j=i-1;j<=7;j++)
    S1(i,j);
  if ((i>=2)&&(i<=6)) {
    for (j=0;j<=4;j++)
      S2(i,j);
  }
}
```

optimized for code size

optimized for control

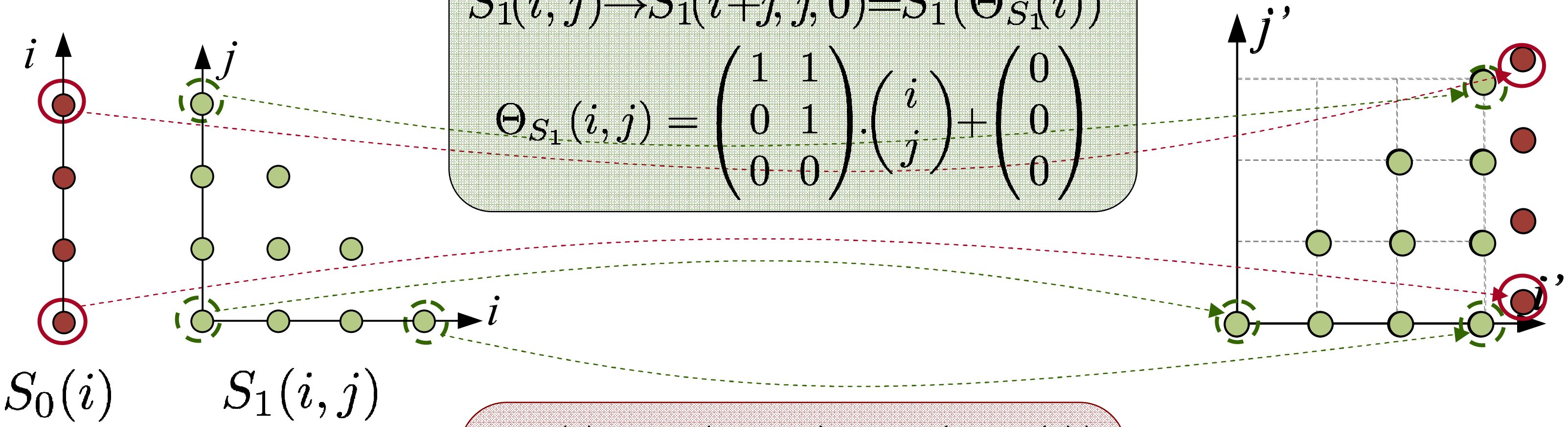
```
  for (j=0;j<=7;j++) S1(1,j);
  for (i=2;i<=5;i++) {
    for (j=0;j<=i-2;j++) S2(i,j)
    for (j=i-1;j<=4;j++) {
      S1(i,j);
      S2(i,j) ;
    }
    for (j=5;j<=7;j++) S1(i,j);
  }
  for (j=0;j<=4;j++) S2(6,j) ;
  for (j=5;j<=7;j++) S1(6,j);
  for (i=7;i<=8;i++)
    for (j=i-1;j<=7;j++)
S1(i,j);
```


[1] Cedric Bastoul, Code Generation in the Polyhedral Model Is Easier Than You Think. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, 2004

Polyhedral loop transformation in a nutshell

$$S_1(i,j) \rightarrow S_1(i+j, j, 0) = S_1(\Theta_{S_1}(i))$$

$$\Theta_{S_1}(i,j) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



$$S_0(i) \rightarrow S_0(3, i, 1) = S_0(\Theta_{S_0}(i))$$

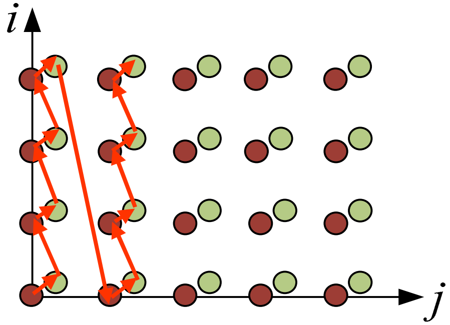
$$\Theta_{S_0}(i) = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} .i + \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}$$

```
    for(i=0;i<4;i++) {
S0:   X[i]=…;
      for(j=0;j<4-i;j++) {
S1:     X[i]=…;
      }
    }
```

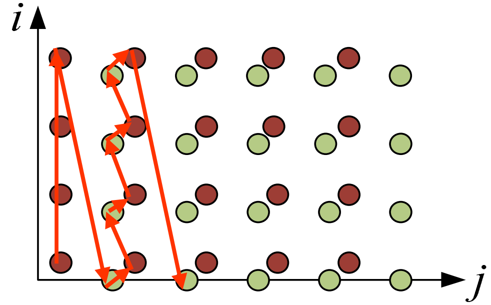
```
    for(i=0;i<4;i++) {
      for(j=0;j<i+1;j++)
S1:     X[i-j]=…;
    }
    for(j=0;j<i+1;j++) {
S1:     X[3-j]=…;
S0:     X[j]=…;
    }
```

Scheduling & loop transformations

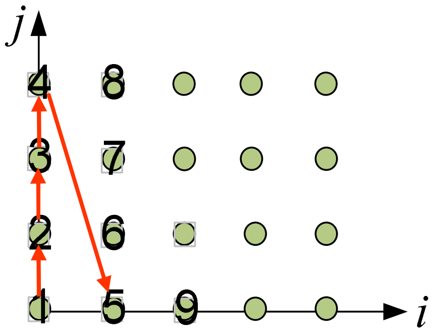
- Loop shifting
  - Shift a statement by some constant along a domain dimension



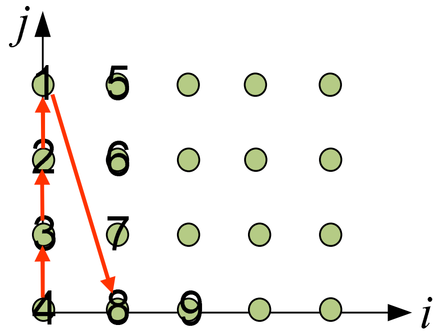
$$S_0(i,j,0) \rightarrow S_0(i,j,1)$$
$$S_1(i,j,1) \rightarrow S_1(i+1,j,0)$$



- Loop reversal
  - Negates a loop index expression in the schedule

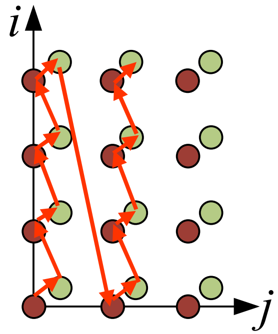


$$S_0(i,j) \rightarrow S_0(i,-j)$$

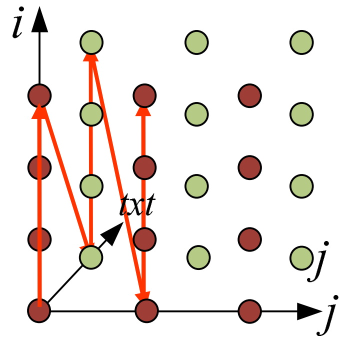


Scheduling & loop transformations

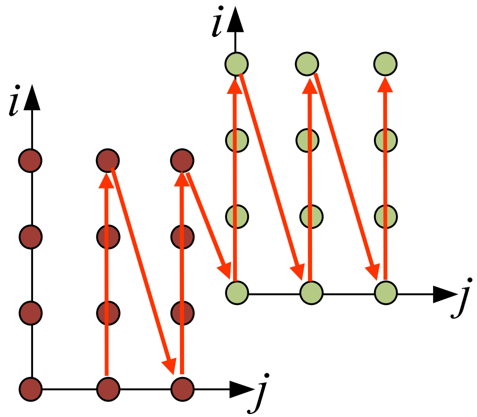
- Loop distribution
  - Distributes statements in distinct loops using a scalar dimension



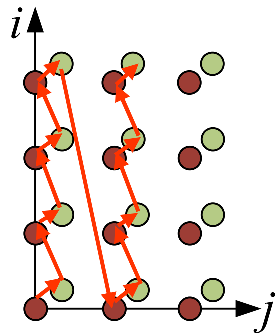
$$S_0(i,j,0) \rightarrow S_0(i,0,j)$$
$$S_1(i,j,1) \rightarrow S_1(i,1,j)$$



- Loop fusion
  - merge scalar dimensions to fuse/merge successive loops

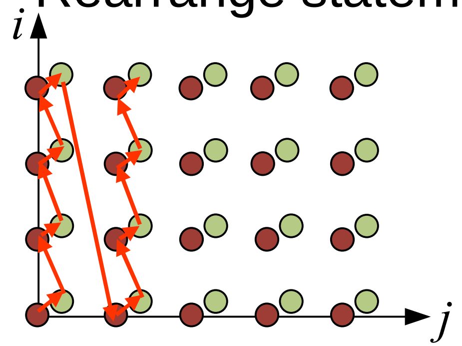


$$S_0(0,i,j) \rightarrow S_0(i,j,0)$$
$$S_1(1,i,j) \rightarrow S_1(i,j,1)$$

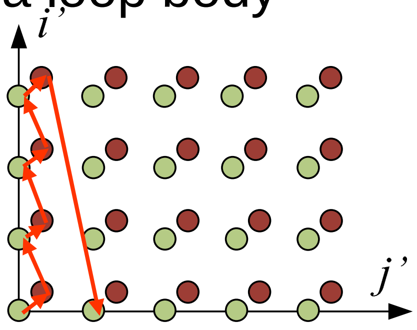


Scheduling & loop transformations

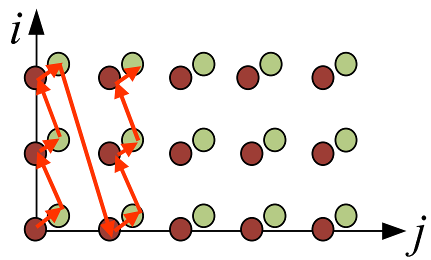
- Statement interchange
  - Rearrange statements textual ordering in a loop body



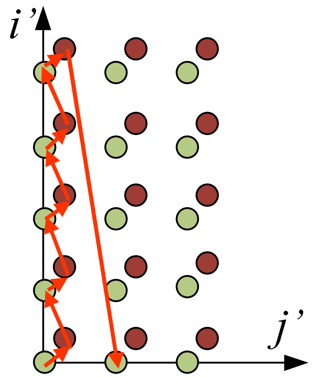
$$S_0(i,j,0) \rightarrow S_0(i,j,1)$$
$$S_1(i,j,1) \rightarrow S_1(i,j,0)$$



- Loop interchange
  - Index swapping in the schedule to change loop indices depths



$$S_0(i,j,0) \rightarrow S_0(j,i,1)$$
$$S_1(i,j,1) \rightarrow S_1(j,i,0)$$



Composing transformations

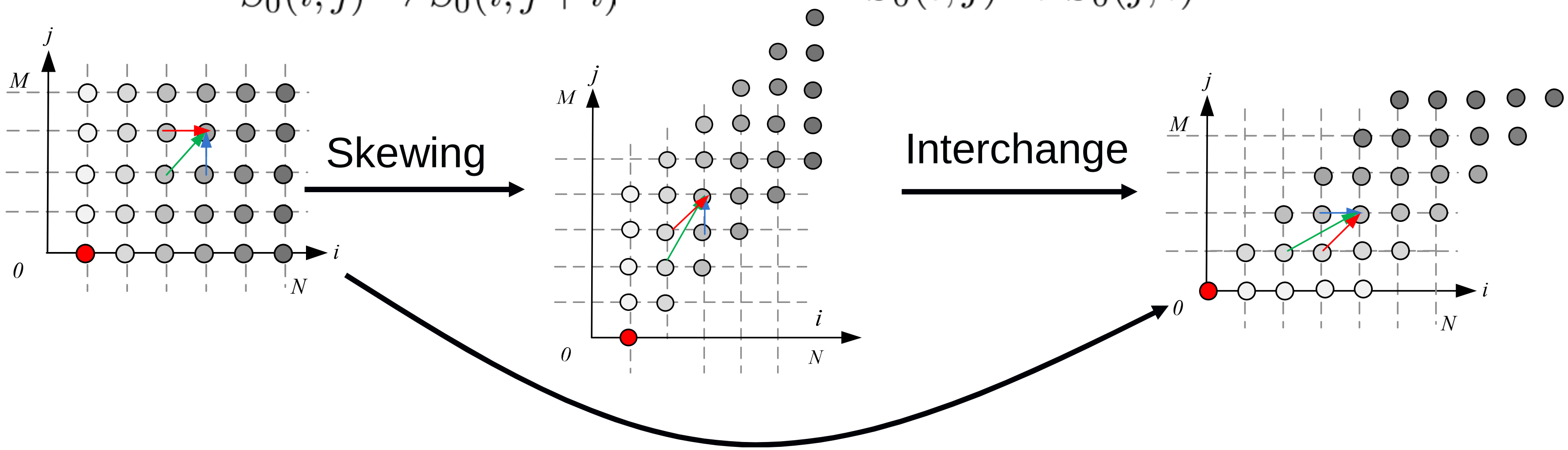
- With this formalism we can compose transformations
  - Simply by composing the statement scheduling functions

$$\Theta_{S_0}(i,j) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix}$$

$$S_0(i,j) \rightarrow S_0(i, j+i)$$

$$\Theta_{S_0}(i,j) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix}$$

$$S_0(i,j) \rightarrow S_0(j,i)$$



Skewing + Interchange

$$\Theta_{S_0}(i,j) = \left(\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} . \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right) . \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix}$$

$$S_0(i,j) \rightarrow S_0(i+j, j)$$

How to model parallel execution ?

- By scheduling statement instances at a same timestamp

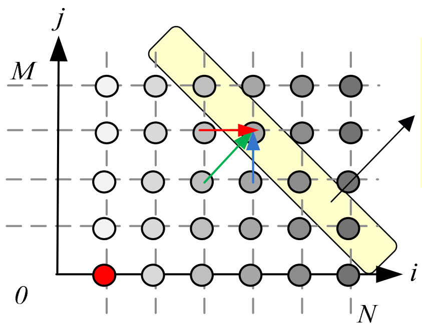
```
   for(i=1;i<7;i++) {
S0:     X[i]= …;
S1:     Y[i]= …;
   }
```

S0 $(i,0) \rightarrow (i)$

S1 $(i,1) \rightarrow (i)$

```
for(i=1;i<7;i++) {
   // in parallel
   Y[i]= … ; X[i]= …;
}
```

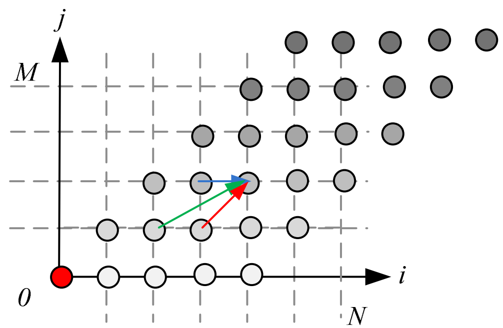
- Parallel loop = all its iterations have the same schedule
  - We ignore some dimension of the schedule when checking for legality, but keep them for code generation.



All these iterations have the same schedule

$$\Theta_{S_0}(i,j) = \begin{pmatrix} 1 & 1 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix} = i + j$$

Parallel schedule for legality check



$$\Theta_{S_0}(i,j) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix} = (i + j, j)$$

Full schedule for code generation

Statement vs instance level dependencies

- Name based dependency analysis
  - Performed at the statement and array object level, not at the array cell level (modifying one cell ⇔ modifying the whole array)

```
  for(i=0;i<N;i++) {
S0: x[i] = … ;
S1: …    = x[i+1]
  }
```

We find a RAW dependency although S_0 and S_1 never write/read to the same cell of the array **`x[…]`**.

- Array based dependency
  - Performed the statement and array cell level (S_0 and S_1 are dependant if one execution of S_0,S_1 writes/reads to a same cell)

```
  for(i=0;i<N;i++) {
S0: x[0] = … ;
S1: …    = x[i]
  }
```

We find a RAW dependency although S_0 and S_1 write/read to the same array cell only once in the loop (for i=0)

Can't we really do better than this ?

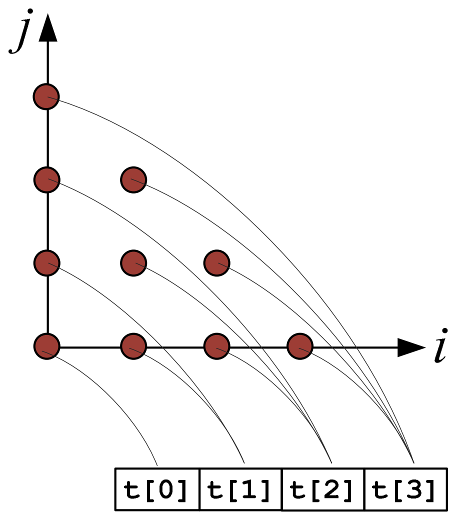
Notion of memory access functions

- How to model memory access more accurately ?
  - We know that every access has an enclosing iteration domain
    - We know the set of iterations where this access occurs
  - We can also model the set of array cells accessed in a statement
- We handle only certain type of memory accesses
  - If index expressions = affine expressions of iterators+parameters
  - This set of index expression defines an ***access function***

```
  for(i=0;i<N;i++) {
    for(j=0;j<=i;j++) {
S0:   tmp[i-j]=…;
    }
  }
```

Access function for **tmp[i-j]** in S_0:

$$(i, j) \rightarrow tmp(i - j) \mid (i, j) \in \mathcal{D}_{S0}$$



Instancewise dependency information

- We propose to reason at the statement instance-level

We want to relate statement instances $S(\vec{x})$ and $S'(\vec{y})$ rather than simply relating S and S'

We will hence write $S(\vec{x})\,\delta\,S'(\vec{y})$ when $S(\vec{x})$ depends on $S'(\vec{y})$

- We will consider two different type of dependency analysis
  - Memory based dependency analysis,
    - Looks for constraints enabling RAW, WAR and WAW dependencies enforcement at the memory cell level.
    - Does not question original program memory allocation choice
  - Value based dependency analysis
    - Looks for the underlying value producer/consumer relations
    - More accurate, but may involve a memory expansion step

Example : RAW memory dependency

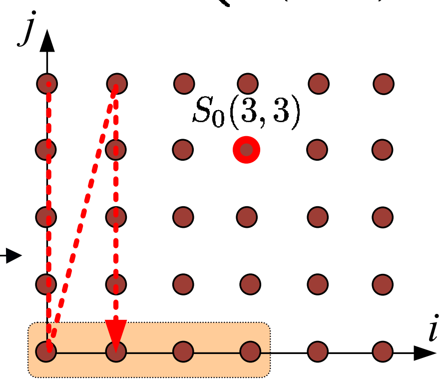
- There is a RAW dependency between $S(\vec{x})$ and $S'(\vec{y})$ if
  - $S(\vec{x})$ is executed after $S'(\vec{y})$ in the original program ($S'(\vec{y}) \prec S(\vec{y})$)
  - $S(\vec{x})$ contains a read operation to a memory cell written by $S'(\vec{y})$
- Example

```
for(i=0;i<N;i++) {
  for(j=0;j<=5;j++) {
S0: tmp[j]=tmp[i-j]+x[i];
  }
}
```

$$S_0(i,j)\,\delta\,S_0(i',j') \text{ iff } \begin{cases} (i,j) \in \mathcal{D}_{S_0} \\ (i',j') \in \mathcal{D}_{S_0} \\ j' = i - j \\ (i',j') \prec (i,j) \end{cases}$$

For $S_0(i=3, j=3)$ we have

$$S_0(3,3)\,\delta\,S_0(i',j') \text{ iff } \begin{cases} 0 \leq i' < N \\ 0 \leq j' < M \\ j' = 0 \wedge i' \leq 3 \end{cases}$$

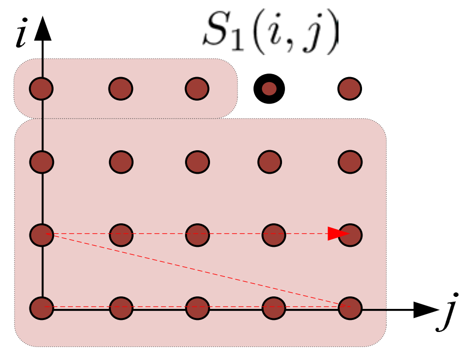


- Same approach for WAR and WAW dependencies

Value based dependency analysis

- Memory based dependency analysis is conservative
  - It can hide some obvious parallelization opportunities
- Example

```
   for(i=0;i<N;i++) {
S0:  tmp = 0;
     for(j=0;j<=M;j++) {
S1:    tmp=tmp+X[j]*C[i][j];
S3:  Y[i] = tmp;
   }
```



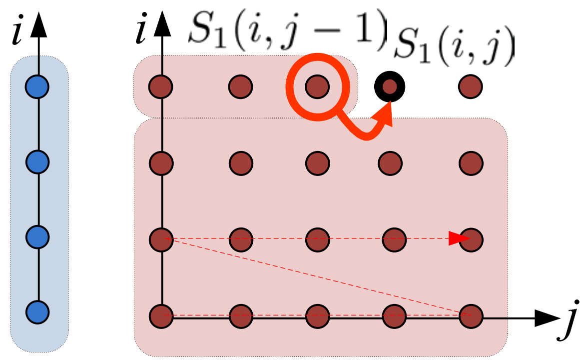
RAW dependency $S_1(i,j)\,\delta\,S_1(i',j')$ iff $\begin{cases} 0 \le i < N \wedge 0 \le j < M \\ 0 \le i' < N \wedge 0 \le j' < M \\ (i' < i) \vee (i' = i \wedge j' < j) \end{cases}$

S(i,j) depends on all previous iterations of the loop, no parallelization seems possible. But, when looking at the algorithm, it is obvious that each Y[i] can be computed on a different thread (with tmp privatized)

Value based dependency analysis

- To see this we must look at the value flow in the program
  - Focus on values production/consumption relations
  - These relations are a subset of RAW dependencies.
- How to obtain this value flow relation ?
  - Given a RAW dep. $S_1(\vec{x})\,\delta\,S_2(\vec{y})$, we look for the statement instance $S_2(\vec{y})$ which produced the value used at $S_1(\vec{x})$

```
   for(i=0;i<N;i++) {
S0:  tmp = …
     for(j=0;j<=i;j++) {
S1:    tmp=tmp+X[j]*C[i][j];
S3:  Y[i] = tmp;
   }
```



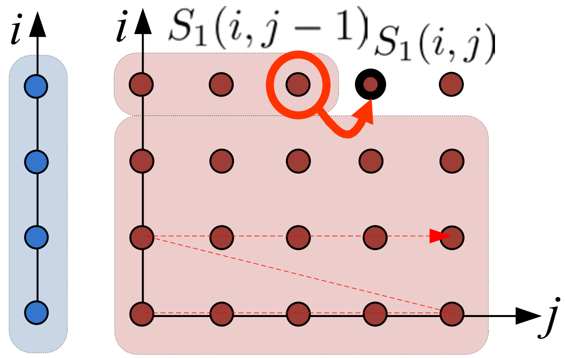
  - This statement instance is the last one (i.e. the lexicographical maximum of all $S_2(\vec{y})$ preceding $S_1(\vec{x})$)

Value based dependency analysis

- Finding the last preceding write to a given array cell ?
  - This write instance is the lexicographical maximum of all preceding producers candidates [2].
  - Found through *Parametric Integer Programming* [1], the solution is in the form of a piecewise affine function.

```
    for(i=0;i<N;i++) {
S0:   tmp = …
      for(j=0;j<=i;j++) {
S1:     tmp=tmp+X[j]*C[i][j];
S3:   Y[i] = tmp;
    }
```

$$S_1(i,j)\,\delta \begin{cases} S_1(i,j-1) & j >= 1 \\ S_0(i) & j < 1 \end{cases}$$

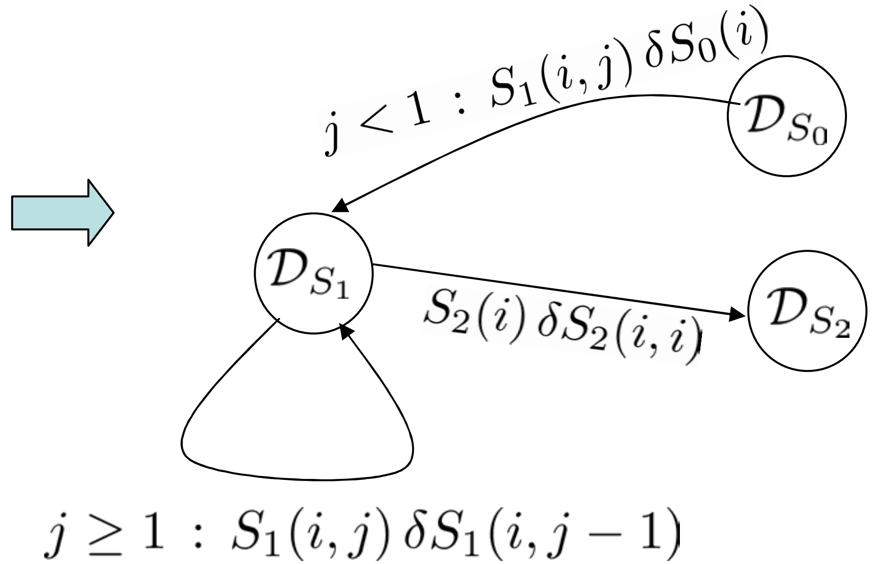


1. P. Feautrier. *Parametric Integer Programming*. *RAIRO Recherche Opérationnelle*, 1988.
2. P. Feautrier. *Dataflow Analysis of Scalar and Array References*, IJPD, 1991

Polyhedral Reduced Dependence Graph

- We use a PRDG to model dependencies in a loop nest
  - Statements domains form the vertices of the graph
  - Dependency information form the edges of the graph
- Example
  - We assume that dependency information is obtained through a value based dependency instead of a memory based analysis

```
   for(i=0;i<N;i++) {
S0:  tmp = …
     for(j=0;j<=i;j++) {
S1:    tmp=tmp+X[j]*C[i][j];
S2:  Y[i] = tmp;
   }
```



Loop transformation legality

- The schedule must enforce dependency constraints
  - If statement instance $S_1(\vec{y})$ depends on $S_0(\vec{x})$, the schedule must be such that $S_1(\vec{y})$ is scheduled after $S_0(\vec{x})$, or more formally

$$\forall\, \vec{x}, \vec{y} \;\; s.t.\;\; S_0(\vec{x})\, \delta\, S_1(\vec{y}) \implies \Theta_{S_0}(\vec{x}) \succ \Theta_{S_1}(\vec{y})$$

- We can deduce the set of violated dependencies
  - All pair of point not enforcing the dependency constraints

$$\forall\, \vec{x}, \vec{y} \;\; : \;\; S_0(\vec{x})\, \delta\, S_1(\vec{y}) \wedge \Theta_{S_0}(\vec{x}) \preceq \Theta_{S_1}(\vec{y})$$

  - With

$$\Theta_{S_0}(\vec{x}) \preceq \Theta_{S_1}(\vec{y}) \Leftrightarrow \bigvee_{i=1}^{N} \left(\left(\bigwedge_{k=i+1}^{N} \Theta^k_{S_0}(\vec{x}) = \Theta^k_{S_1}(\vec{y}) \right) \wedge \Theta^i_{S_0}(\vec{x}) < \Theta^i_{S_1}(\vec{y}) \right)$$

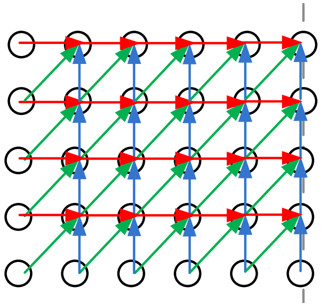
Verifying loop transformation legality amounts to check the emptiness of a union of integer polyhedron.

Loop transformation legality

- Example
  - Kernel

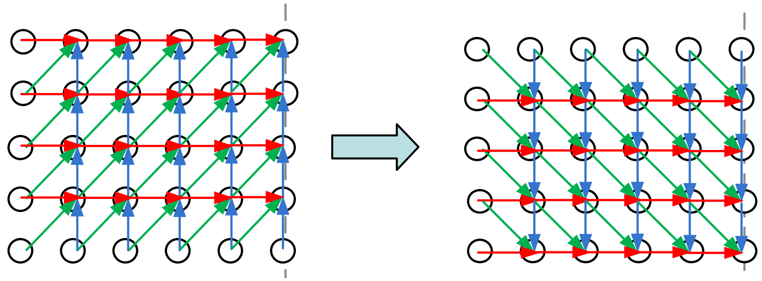
```
 for(i=1;i<7;i++) {
   for(j=0;j<5;j++)
S0:  X[i,j]=max(X[i-1,j-1]+A,
                X[i,j-1]+B
                X[i-1,j]+C);
 }
```

$$S_0(i,j)\,\delta\,S_0(i,j-1)\ \forall\, j>0$$
$$S_0(i,j)\,\delta\,S_0(i-1,j)\ \forall\, i>0$$
$$S_0(i,j)\,\delta\,S_0(i-1,j-1)\ \forall\, i,j>0$$



  - Transformation $\Theta_{S_0}(i,j) = (i, 4-j)$

$$S_0(i,j)\,\delta\,S_0(i,j-1) \Rightarrow \Theta_{S_0}(i,j) \succ \Theta_{S_0}(i,j-1)$$
$$S_0(i,j)\,\delta\,S_0(i-1,j) \Rightarrow \Theta_{S_0}(i,j) \succ \Theta_{S_0}(i,j-1)$$
$$S_0(i,j)\,\delta\,S_0(i-1,j-1) \Rightarrow \Theta_{S_0}(i,j) \succ \Theta_{S_0}(i,j-1)$$



$\Longrightarrow$ ~~$(i,5-j) \succ (i,5-j+1)\ \forall\,(i,j)$~~ $\in \mathcal{D}_{S0}$

This clause does obviously not hold, and there is a dependency violation for all *(i,j)* !!!

In general, one have to use an ILP/SMT solver to prove a schedule

Part III : scheduling & parallelization

Outline

1. Finding one dimensional schedules
    1. For a simple case (uniform dependencies)
    2. For affine dependencies by quantifier elimination
    3. The vertex and Farkas approaches
2. Finding multi-dimensional schedules
    1. Feautrier Greedy heuristic
    2. Iterative polyhedral compilation
    3. Locality aware parallelization

But how to find a good/legal schedule ?

- Pick schedules randomly and see if they are correct ?
  - Very low chance to find a legal schedule for a given try
  - Legality checks are costly (polyhedron of a pressburger formula)
- Find some constraints over schedule coefficients
  - s.t. when enforced, the schedule is guaranteed to be legal.
  - How to derive these constraints ?
- In the following, we will start by studying 1D schedules
  - 1D schedules map every statement instance to a simple timestamp.
  - The timestamp is an affine function of the statement index

$$\Theta(\vec{x}) = \tau_0 x_0 + \tau_1 x_1 + \ldots + \tau_{N_{dim}-1} x_{N_{dim}-1} + \tau_{N_{dim}}$$

A (too) simple example

- Searching for a 1D schedule for our example

```
   for(i=1;i<6;i++) {
     for(j=0;j<4;j++)
S0:      X[i,j]=max(X[i-1,j-1]+A,
                    X[i,j-1]+B
                    X[i-1,j]+C);
   }
```

The RAW dependencies are

$$\begin{array}{lcl} S_1(i,j)\ \delta\ S_1(i-1,j) & : & i \geq 1 \\ S_1(i,j)\ \delta\ S_1(i,j-1) & : & j \geq 1 \\ S_1(i,j)\ \delta\ S_1(i-1,j-1) & : & i \geq 1 \wedge j \geq 1 \end{array}$$

- The scheduling function is written as

$$\Theta_{S_1}(i,j) = \tau_0.i + \tau_1.j + \tau_2$$

- To be legal, it must enforce all dependencies

$$\begin{array}{rl} S_1(i,j)\ \delta\ S_1(i-1,j) & \Rightarrow \Theta_{S_1}(i,j) > \Theta_{S_1}(i-1,j) \quad \forall (i,j) \in \mathcal{D}_{S_1} \wedge i \geq 1 \\ S_1(i,j)\ \delta\ S_1(i,j-1) & \Rightarrow \Theta_{S_1}(i,j) > \Theta_{S_1}(i,j-1) \quad \forall (i,j) \in \mathcal{D}_{S_1} \wedge j \geq 1 \\ S_1(i,j)\ \delta\ S_1(i-1,j-1) & \Rightarrow \Theta_{S_1}(i,j) > \Theta_{S_1}(i-1,j-1) \quad \forall (i,j) \in \mathcal{D}_{S_1} \wedge i,j \geq 1 \end{array}$$

A (too) simple example

- We can now inject Θ_{S1} definition in the constraints

$$\Theta_{S_1}(i,j) = \tau_0.i + \tau_1.j + \tau_2$$

$$\begin{array}{ll} \Theta_{S_1}(i,j) > \Theta_{S_1}(i-1,j) & \forall (i,j) \in \mathcal{D}_{S_1} \wedge i \geq 1 \\ \Theta_{S_1}(i,j) > \Theta_{S_1}(i,j-1) & \forall (i,j) \in \mathcal{D}_{S_1} \wedge j \geq 1 \\ \Theta_{S_1}(i,j) > \Theta_{S_1}(i-1,j-1) & \forall (i,j) \in \mathcal{D}_{S_1} \wedge i,j \geq 1 \end{array}$$

- And derive constraints over the coefficients τ_i

$$\begin{array}{lll} S_1(i,j)\ \delta\ S_1(i-1,j) & \Rightarrow & \tau_0(i-1) + \tau_1 j + \tau_2 < \tau_0 i + \tau_1 j + \tau_2 \\ & \Rightarrow & \tau_0(i-1) - \tau_0 i) > 0 \\ & \Rightarrow & \tau_0 > 0 \\ S_1(i,j)\ \delta\ S_1(i,j-1) & \Rightarrow & \tau_1 > 0 \\ S_1(i,j)\ \delta\ S_1(i-1,j-1) & \Rightarrow & \tau_1 + \tau_2 > 0 \end{array}$$

One legal schedule

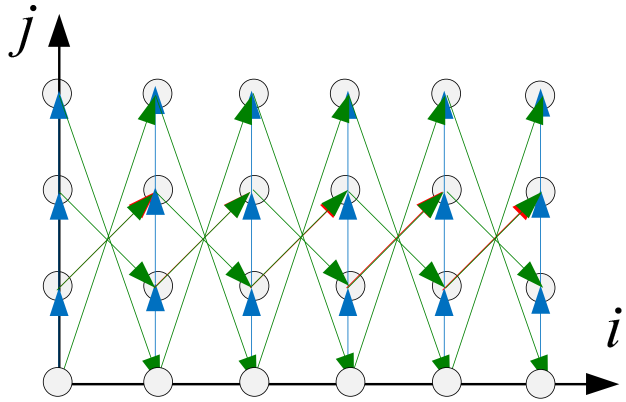
$$\Theta_{S_1}(i,j) = i + j$$

A (less) simple example

- With *non uniform* dependencies

```
    for(i=1;i<6;i++) {
      for(j=0;j<4;j++)
S0:     X[i,j]=max(X[i,j-1]+A,
                   X[i-1,3-j]+C);
    }
```

$$S_1(i,j)\ \delta\ S_1(i,j-1) \quad : \quad j \geq 1$$
$$S_1(i,j)\ \delta\ S_1(i-1,3-j) \quad : \quad i \geq 1$$



- The constraints over the τ_i become

$$S_1(i,j)\ \delta\ S_1(i,j-1) \quad \Rightarrow \quad \tau_1 > 0$$

$$S_1(i,j)\ \delta\ S_1(i-1,3-j) \quad \Rightarrow \quad \tau_1(3-j) + \tau_0.(i-1) + \tau_2 < \tau_0 i + \tau_1 j + \tau_2$$
$$\Rightarrow \quad \boxed{\tau_1.(2j-3) + \tau_0 > 0} \quad \forall (i,j) \in \mathcal{D}_{S_0}$$

- The constraints now involve iteration domain indices …
  - The scheduling legality depends on the iteration domain shape !!

Quantifier elimination

- How to get rid of iteration indices in the constraints ?
  - Obtain an equivalent quantifier free expression (i.e. involving only scheduling coefficients) for constraints such as

$$\forall (i,j) \in \mathcal{D}_{\mathcal{S}} \Rightarrow \tau_1.(2j-3)+\tau_1 > 0$$

- Two approach can be used
  - The first one by Quinton et al. is known as the vertex method [1]
  - The second one by Feautrier leverages the Farkas lemma [2].

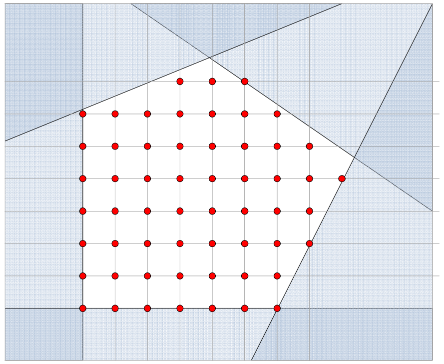
[1] Patrice Quinton, Vincent Van Dongen: The mapping of linear recurrence equations on regular arrays. VLSI Signal Processing 1(2): 95-113 (1989)

[2] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, I, One Dimensional Time. Int. Journal of Parallel Programming, 21(5):313--348, October 1992

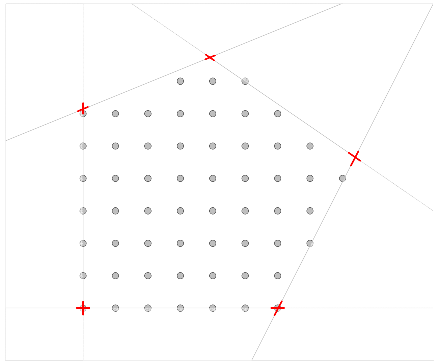
The vertex method (oversimplified)

UNIVERSITÉ DE RENNES 1

- Background : a polyhedron has two representations
  - The Chernikova algorithm permit to change from one representation to the other (very costly)



Using constraints

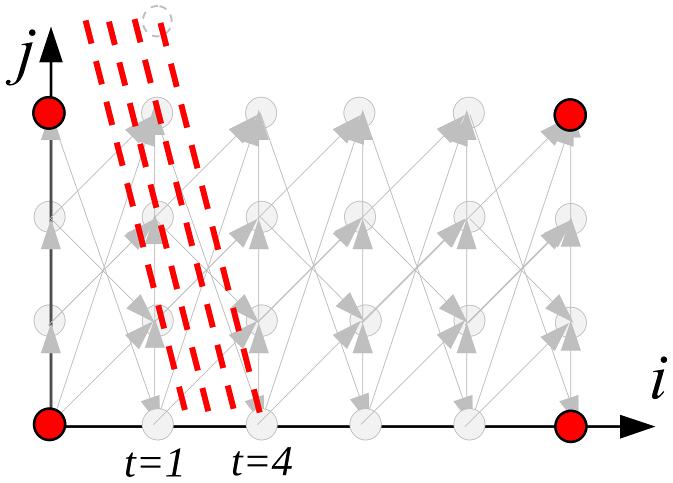


Using vertices (and rays)

- Main trick
  - A scheduling legal for all vertices of $\mathcal{D}$ is legal for all points inside the domain $\mathcal{D}$.
  - Let's use the vertex position to derive quantifier free constraints !

The vertex method (oversimplified)

- Back to the example



$\tau_1 > 0$ (from slide 68)

$$\forall(i,j) \in \mathcal{D}_{\mathcal{S}} \Rightarrow \tau_1.(2j-3) + \tau_0 > 0$$

+

Vertices : (*i=0, j=0*), (*i=5, j=0*)
(*i=0, j=3*), (*i=5, j=3*)

⇩

$$\begin{array}{rcl} \tau_1 & > & 0 \\ 3\tau_1 + \tau_0 & \geq & 0 \\ -3\tau_1 + \tau_0 & \geq & 0 \end{array}$$

$$\Phi_{S_1}(i,j) = i + 4j$$

- In practice things may be slightly more complicated
  – For more details, read the paper !

The Farkas algorithm (oversimplified)

- Farkas lemma
  - Given a polyhedron $\mathcal{D}$ defined by affine constraints $C.\vec{x} + \vec{b} \geq 0$
  - An affine function is positive for all points in $\mathcal{D}$ iff it can be written as a (positive) combination of constraints $\vec{c_i}.\vec{x} + \vec{b_i} \geq 0$

$$f(\vec{x}) \geq 0 \;\; \forall \vec{x} \in \mathcal{D} \Leftrightarrow f(\vec{x}) = \sum_i \mu_i(\vec{c_i}.\vec{x} + \vec{b_i}) \text{ with } \mu_i \geq 0$$

  - The (positive) coefficients of this linear combination are called Farkas multipliers (μ_i)
- How to use this ?
  - Write the schedule constraint as a (positive) linear combination of the statement domain constraints
  - We obtain a new system of constraints involving only Farkas multipliers (μ_i) and scheduling coefficient (τ_i).

The Farkas approach (example)

- Scheduling constraint from slide 68

$$\forall (i,j) \in \mathcal{D}_S \Rightarrow \tau_0 + \tau_1.(2j-3) >= 0 \qquad \mathcal{D}_S = \begin{cases} i & >= 0 \\ j & >= 0 \\ 7-i & >= 0 \\ 3-j & >= 0 \end{cases}$$

$$\tau_0 + \tau_1.(2j-3) = \mu_0.i + \mu_1.j + \mu_2.(3-j) + \mu_3.(7-i)$$

$$\boxed{\tau_0 - 3\tau_1} + \boxed{2\tau_1}j = \boxed{(\mu_0 - \mu_3)}i + \boxed{(\mu_1 - \mu_2)}.j + \boxed{3\mu_2 + 7\mu_3}$$

Identification

$$\boxed{2\tau_1 = \mu_1 - \mu_2} \quad \boxed{\tau_0 - 3\tau 1 = 3\mu_2 + 7\mu_3} \quad \boxed{0 = \mu_0 - \mu_3}$$

Gauss elimination →

$$\begin{array}{l} \tau_0 + 3\tau 1 = 3\mu_1 + 7\mu_3 \\ \tau_0 - 3\tau 1 = 3\mu_2 + 7\mu_3 \\ \mu_1 \geq 0 \\ \mu_3 \geq 0 \end{array}$$

projection →

$$\begin{array}{rl} 3\tau_1 + \tau_0 & \geq 0 \\ -3\tau_1 + \tau_0 & \geq 0 \end{array}$$

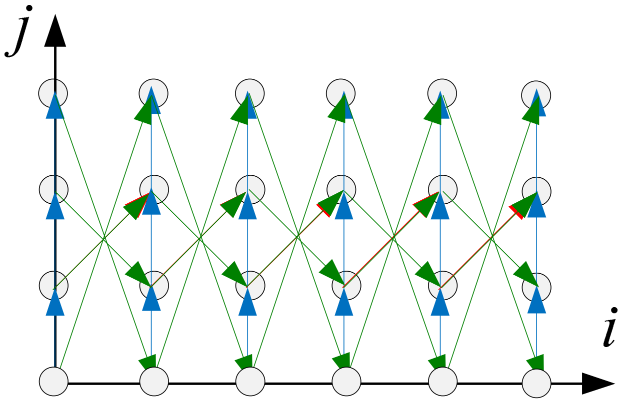
$$\Phi_{S_1}(i,j) = i + 4j$$

Limitations of 1D scheduling functions

- Consider a parameterized version of our example loop

```
   for(i=1;i<6;i++) {
     for(j=0;j<M;j++)
S0:    X[i,j]=max(X[i,j-1]+A,
                  X[i-1,M-j-1]+C);
   }
```

$$S_1(i,j)\ \delta\ S_1(i,j-1) \quad : \quad j \geq 1$$
$$S_1(i,j)\ \delta\ S_1(i-1,M-j-1) \quad : \quad i \geq 1$$



- The scheduling now follows $\Theta_{S_1}(i,j) = \tau_0.i + \tau_1.j + \tau_2.M + \tau_3$
  - This leads to the following constraint system

$$S_1(i,j)\ \delta\ S_1(i,j-1) \Rightarrow \tau_1 > 0$$

$$\begin{aligned} S_1(i,j)\ \delta\ S_1(i-1,M-j+1) &\Rightarrow \tau_0 i + \tau_1 j + \tau_2 M + \tau_3 > \tau_1(M-j+1) + \tau_0.(i-1) + \tau_2 M + \tau_3 \\ &\Rightarrow 2\tau_1 j - \tau_1 M - \tau_1 + \tau_0 > 0 \ \ \forall (i,j) \\ &\Rightarrow 2\tau_1 j - \tau_1 M - \tau_1 + \tau_0 - 1 \geq 0 \ \ \forall (i,j) \ \in \ \mathcal{D}_{S_0} \end{aligned}$$

The Farkas approach (example)

- Scheduling constraint from previous slide

$$\forall (i,j) \in \mathcal{D}_S \Rightarrow \tau_0 - 1 + \tau_1.(2j - M - 1) \geq 0$$

$$\mathcal{D}_S = \begin{cases} i & >= 0 \\ j & >= 0 \\ 7 - i & >= 0 \\ M - 1 - j & >= 0 \end{cases}$$

$$\tau_0 - 1 + \tau_1.(2j - 1) - \tau_1 M = \mu_0.i + \mu_1.j + \mu_2.(M - j - 1) + \mu_3.(7 - i)$$

$$(\tau_0 - \tau_1 - 1) + (2\tau_1)j - (\tau_1)M = (\mu_0 - \mu_3)i + (\mu_1 - \mu_2).j + \mu_2 M + 7\mu_3 - \mu_2$$

Identification

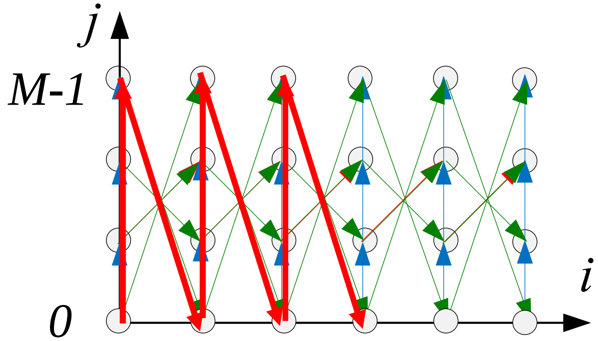
$-\tau_1 = \mu_2 \longrightarrow \tau_1 \leq 0$ which contradicts $\tau_1 > 0$

There is no scheduling solution able to satisfy the constraints for both dependencies !

Limitations of 1D scheduling functions

- But, there must be a legal schedule for the loop nest
  - Indeed, we can write the initial program schedule as

$$\Phi_{S_1}(i,j) = i + Mj$$



- This schedule is however not an *affine* schedule
  - The product $M;j$ is not affine as M is not a constant

Multidimensional schedules

- Not all loop nests admit one-dimensional schedules
  - Even when they do, this might not be the best schedule
- We can instead use multidimensional schedules
  - But how to derive legal schedules ?
- Several approaches have been proposed
  - A greedy algorithm by Feautrier (1992) [1]
  - A framework for polyhedral iterative compilation (2008) [2]
  - A locality aware parallelization algorithm (2008) [3]

[1] Paul Feautrier: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. International Journal of Parallel Programming, 1992

[2] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, John Cavazos: Iterative optimization in the polyhedral model: part ii, multidimensional time. PLDI 2008

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayappan: A practical automatic polyhedral parallelizer and locality optimizer. PLDI 2008

Feautrier's greedy algorithm

- Based on the idea of weakly satisfied dependency
  - A dependency $S_i(\vec{x}) \; \delta \; S_j(\vec{x'})$ is weakly satisfied at a depth *d*, for a schedule Θ_{S_i}, when, given

$$\Theta_{S_i}(\vec{x}) = \begin{pmatrix} \Theta^1_{S_i}(\vec{x}) \\ \Theta^2_{S_i}(\vec{x}) \\ \dots \\ \Theta^n_{S_i}(\vec{x}) \end{pmatrix}$$

We have $\Theta^k_{S_i}(\vec{x}) \; = \; \Theta^k_{S_j}(\vec{x'}) \quad \forall k \leq d$

  - A weakly satisfied dependency at a depth *d* can still be strongly satisfied at dimensions *k>d*.
- Intuition
  - By allowing weakly satisfied dependencies we "leave slack" to the scheduler and postpone the problem to later stage.

[1] Paul Feautrier: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. International Journal of Parallel Programming, 1992.

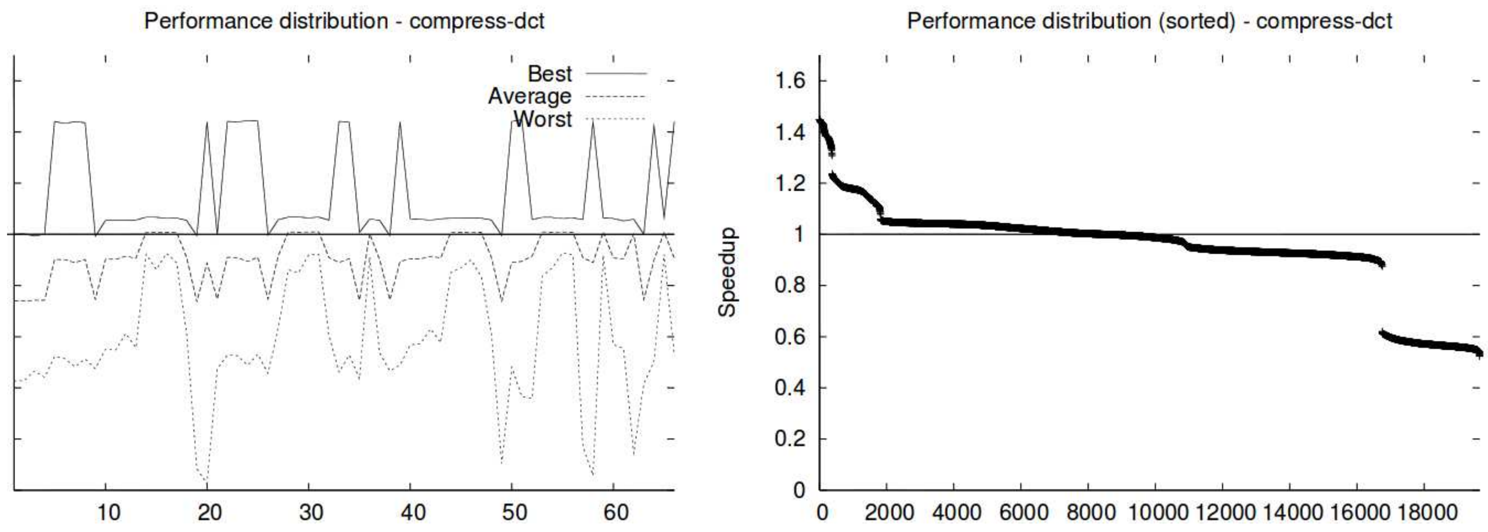
Feautrier’s greedy algorithm

- Uses a greedy algorithm
  - Focus on strongly connected components in the PRDG
  - Starts by the outermost dimension, proceeds to the innermost
  - At every dimension *d*, find a partial schedule that :
    - makes sure all dependencies at weakly satisfied at depth *d*
    - maximizes the number of fully satisfied dependencies
  - The algorithm stops when all dependencies are satisfied
- The algorithm maximizes parallelism
  - Here parallelism means the number of inner parallel loop
  - Does not consider memory access locality
  - Little practical use “as is”

[1] Paul Feautrier: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. International Journal of Parallel Programming, 1992.

Iterative polyhedral compilation

- Enable fast exploration of **many** legal programs
  - Build a convex set of multidimensional legal schedules for bounded [-1,1] schedule coefficients.
  - Explore this set to find the most profitable transformation.

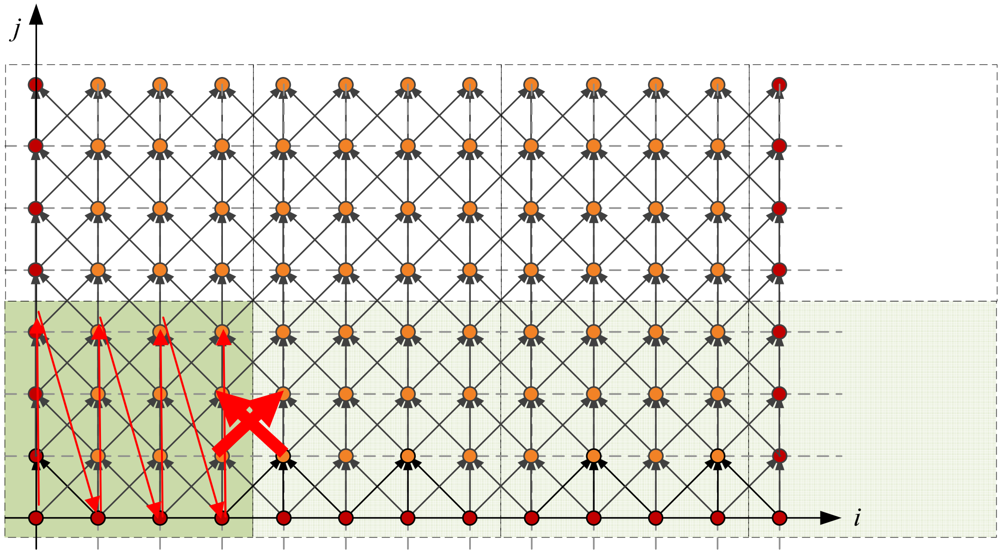


[2] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, John Cavazos: Iterative optimization in the polyhedral model: part ii, multidimensional time. PLDI 2008

A locality aware parallelization algorithm

- Tiling is a widely used parallelizing transformation
  - It is usually applied as a post-scheduling optimization
  - We need to make sure the transformed program can be tiled
  - Reminder : in a tiled program, tiles are executed atomically

This tiling is not possible as tiles have cyclic dependencies

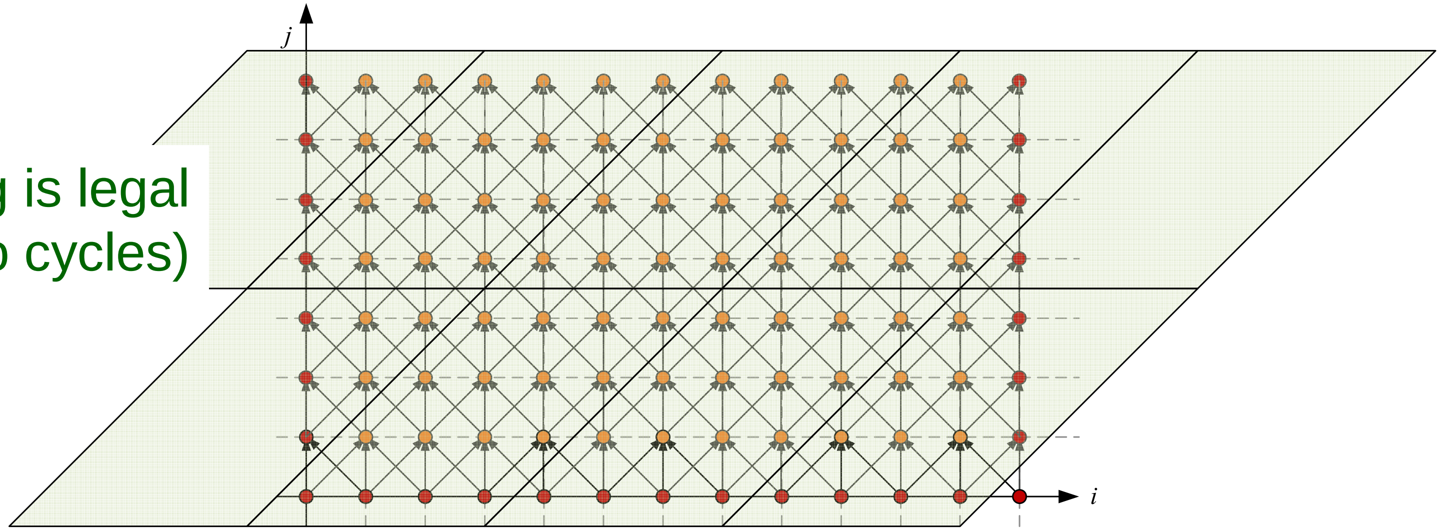


[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayappan: A practical automatic polyhedral parallelizer and locality optimizer. PLDI 2008

Scheduling for Tilability

- Must ensure an unidirectional flow of data after transfo.
  - This constraint can be applied to some innermost loop index
    - Then only this set of innermost can be tiled.
  - Tilability often prevents loop fusion (parallelism/locality trade-off)

This tiling is legal (no cycles)

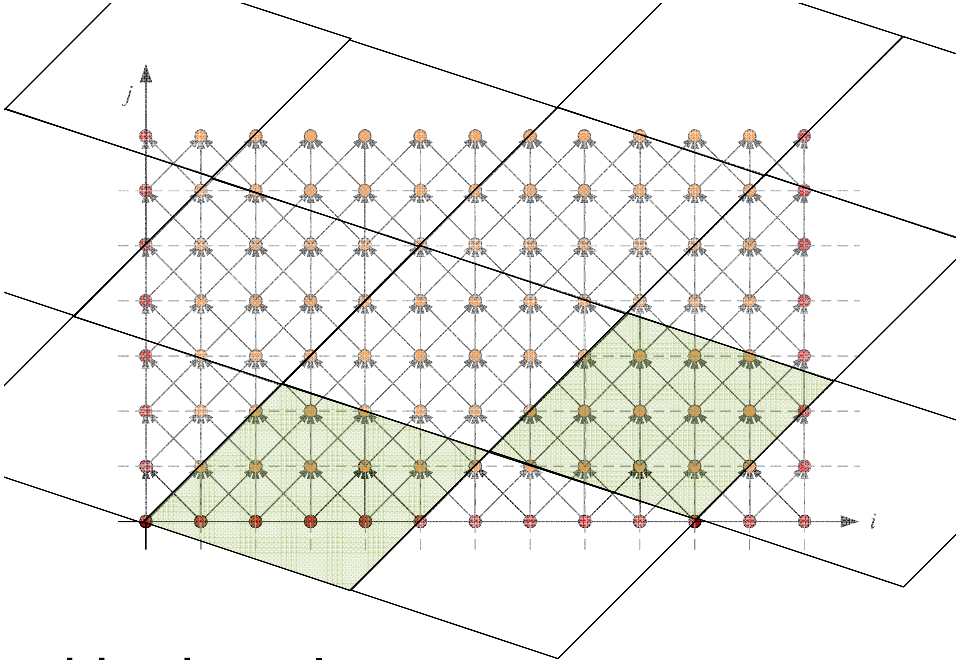


- The constraint is formalized as follow

$$\forall\, \vec{x}, \vec{y} \;\; s.t. \;\; S_0(\vec{x})\, \delta\, S_1(\vec{y}) \implies \Theta_{S_0}(\vec{x}) \succ \Theta_{S_1}(\vec{y}) \;\wedge\; \forall k \;\; \Theta^k_{S_0}(\vec{x}) \geq \Theta^k_{S_1}(\vec{y})$$

The Pluto algorithm

- Searches multi-dimensional schedules retaining tilability
  - Heuristic to find the maximum number of tilable loops
  - Try to minimize reuse distance to improve temporal locality



- Implemented in the Pluto source-to-source compiler
  - http://pluto-compiler.sourceforge.net/ with openMP and Cuda back-end

Part IV

Current/open research topics

Current/open research topics

- Improving it efficiency
  - Taking advantage of hardware specificities (GPU, Many-Core)
- Making it mainstream !
  - Polly in LLVM, Graphite/Gcc, Pluto, PolyRose, etc.
  - Putting it to work in *real* production compilers
- Go beyond affine control loop and affine array accesses
  - How to deal with data-dependant behavior ?
  - How to use speculative polyhedral parallelization ?
- Make it more scalable
  - The full polyhedral hammer is often overkill, one may use simpler abstractions while retaining efficiency.

Questions ?