Cours M2: Compilation avancée et optimisation de programmes

Alain Darte

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

Kernel offloading optimizations and double buffering

Outline

Kernel acceleration: portability problem

Hardware accelerators FPGA, GPU, dedicated board, multicore

- Better energetic profitability.
- Huge portability issue.
- Costly compiler development.

How to automate application porting?

- High-productivity and high-performance languages.
- Library/directives-type support, e.g., OpenAcc.
- Application-aware, compilation-aware, OS-aware, and architecture-aware languages.
- Source-to-source compilation, adaptable to back-ends.

Targeting C dialects. Example of high-level synthesis

Often a C dialect with good back-end optimizations.
Ex: C-to-VHDL high-level synthesis (HLS).

Many academic and industrial tools

- Spark, Gaut, Ugh, Paro, Compaan, Catapult-C, Pico-Express, Impulse-C, C2H, . . .

HLS tools quite good at optimizing computation kernel

- Optimizes finite state machine (FSM).
- Exploits instruction-level parallelism (ILP).
- Performs operator selection, resource sharing, scheduling, etc.

Targeting C dialects. Example of high-level synthesis

Often a C dialect with good back-end optimizations.
Ex: C-to-VHDL high-level synthesis (HLS).

Many academic and industrial tools

- Spark, Gaut, Ugh, Paro, Compaan, Catapult-C, Pico-Express, Impulse-C, C2H, . . .

HLS tools quite good at optimizing computation kernel

- Optimizes finite state machine (FSM).
- Exploits instruction-level parallelism (ILP).
- Performs operator selection, resource sharing, scheduling, etc.

But still a huge problem for feeding the accelerators with data

- Sometimes, no synchronization support with memory ☛ unusable.
- In general, lack of interface support ☛ write (expert) VHDL glue.
- Lack of communication opt. ☛ redesign the algorithm.
- Lack of powerful code analyzers ☛ find tricks.

☛ How to do this automatically at C-level?

Current trend: kernel offloading at C level

C-to-C layer for application *outlining* or *offloading* consisting in

- Function isolation: analyze function footprint and rewrite.
- Optimization: reduce communications, express parallelism.
- Specialization: adapt the code to the specific C compiler.

☛ Ex: work of D. Quinlan (Livermore), S. Guelton, M. Amini (Mines Paris)

Elementary approach

- Analyze the data read and written by the function to offload.
- Perform the transfer from distant memory.
- Do accelerated computation on local memory.
- Transfer back for updating the distant memory.

☛ No pipeline, no double-buffering, no data reuse, no local memory size optimization, etc.

Optimized offloading: pipelining, reuse, local memories

Optimized approach:

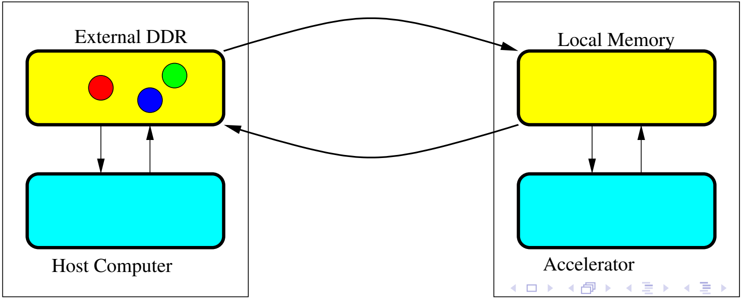
- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.

Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).



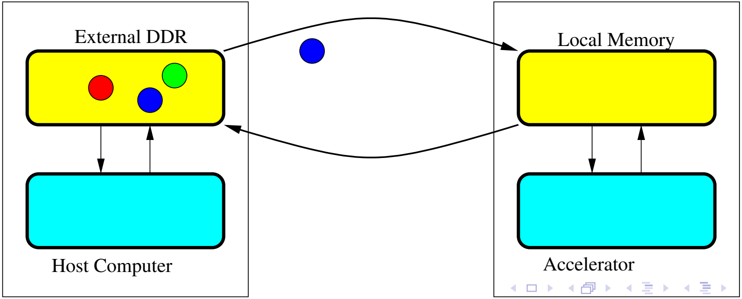
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



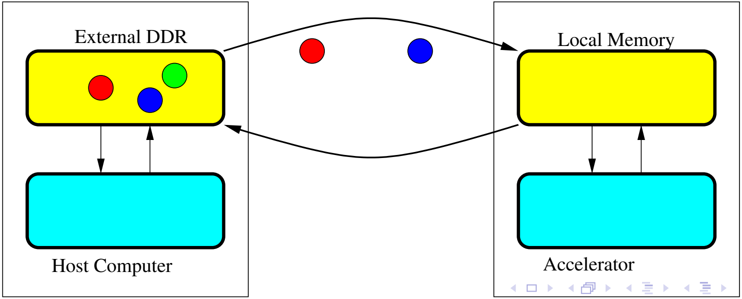
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



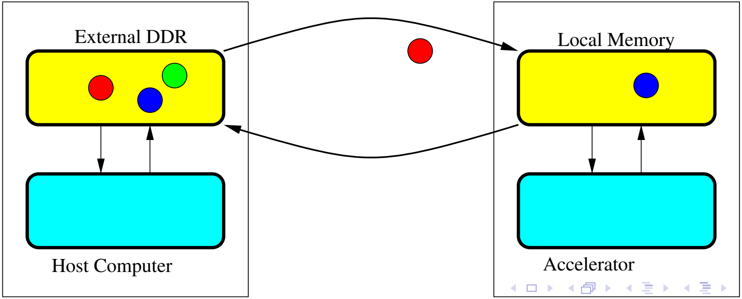
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) $\rightarrow$ ● (block 1) then (●, ●) $\rightarrow$ ● (block 2).

Load data for block 1 in local memory.



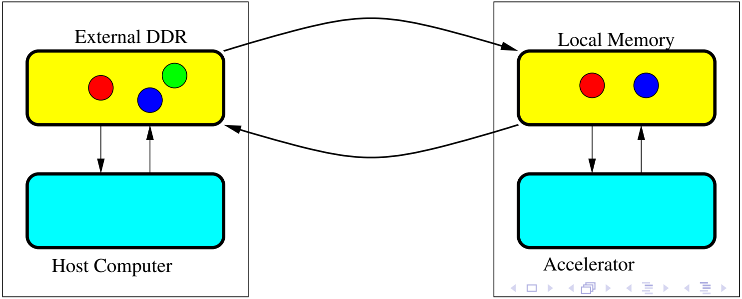
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



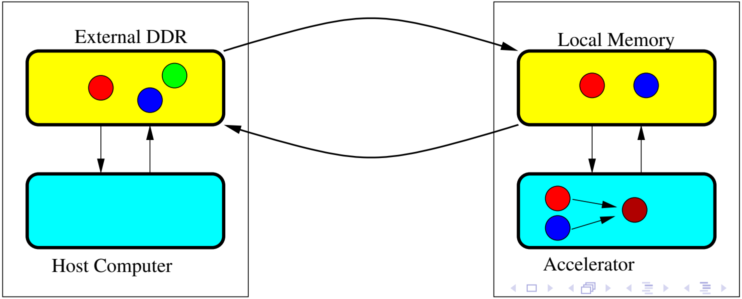
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 1 locally.



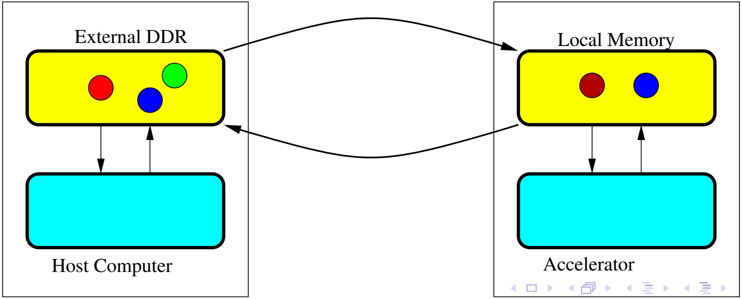
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 1 locally.



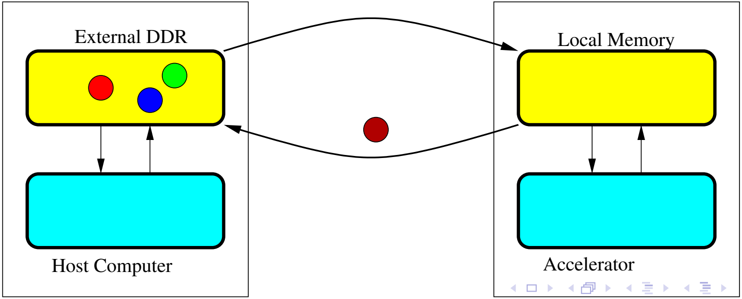
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 1 to distant DDR memory.



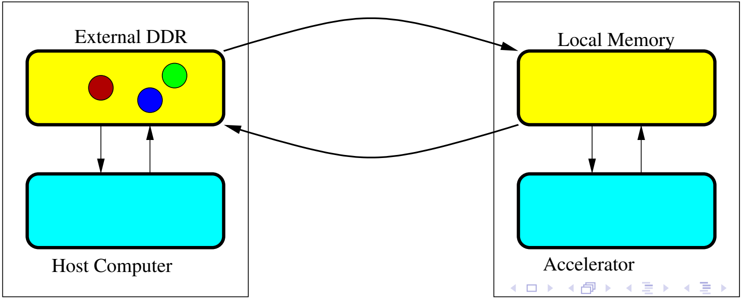
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 1 to distant DDR memory.



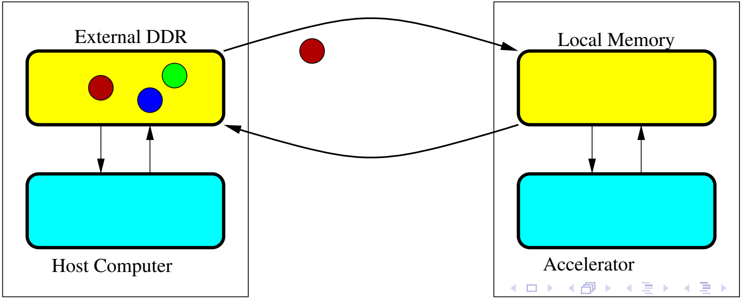
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 2 in local memory.



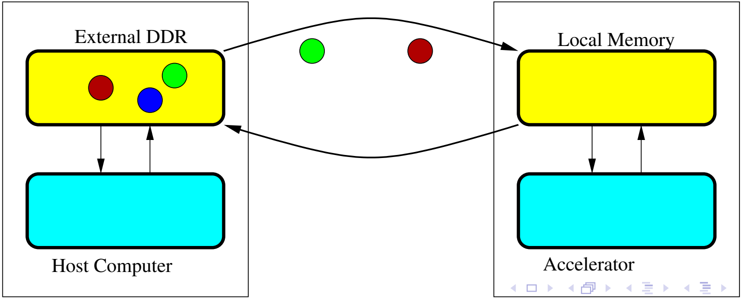
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 2 in local memory.



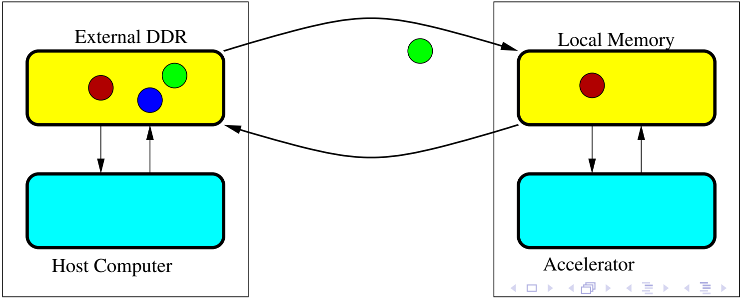
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 2 in local memory.



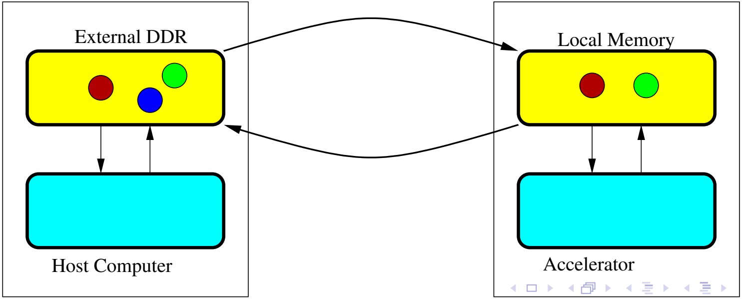
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 2 in local memory.



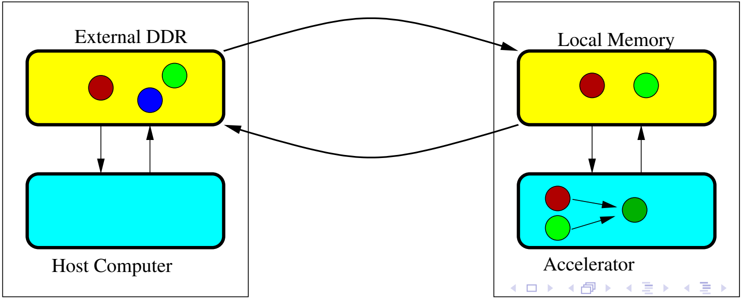
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 2 locally.



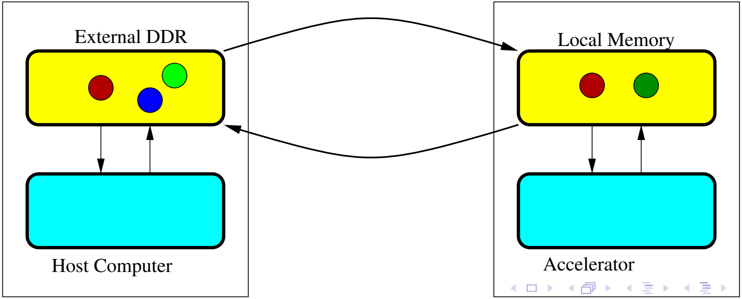
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 2 locally.



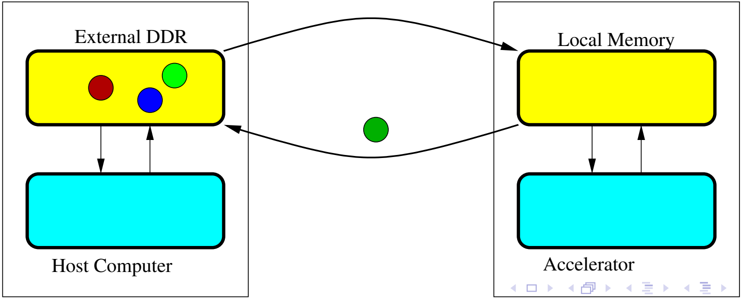
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 2 to distant DDR memory.



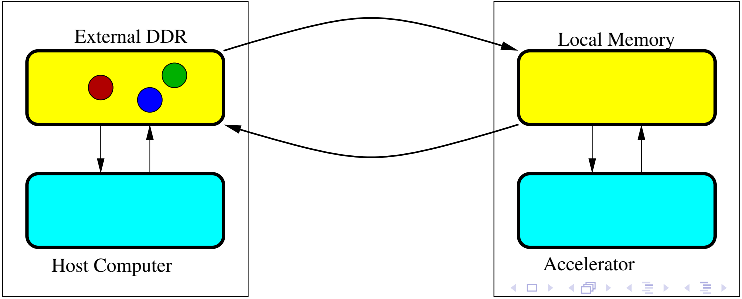
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Computation of blocks in sequence, with no overlap.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 2 to distant DDR memory.

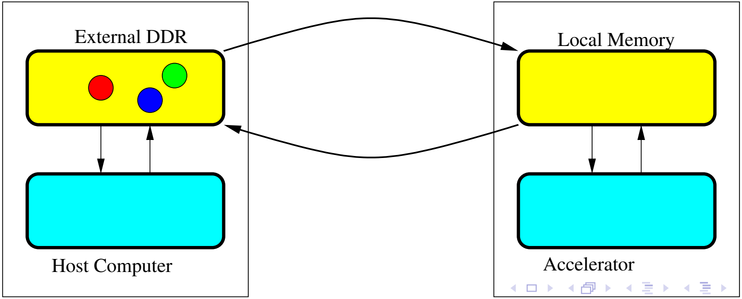


Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).



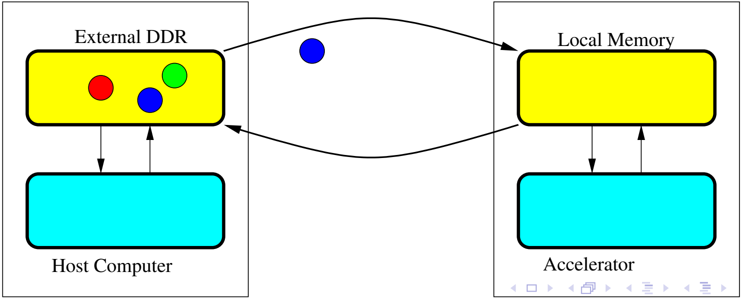
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



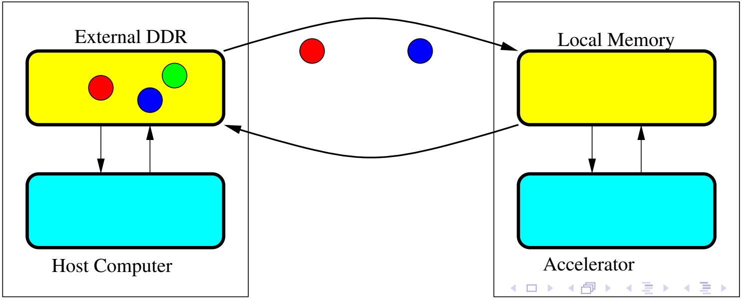
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



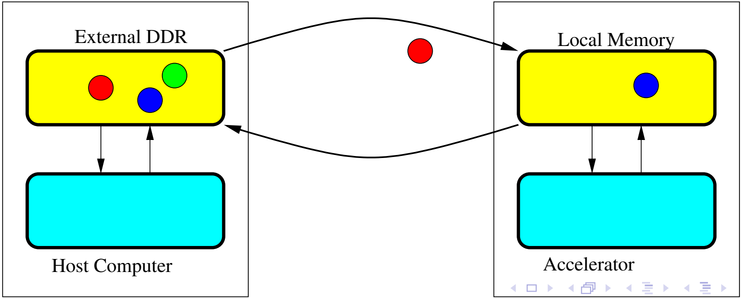
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



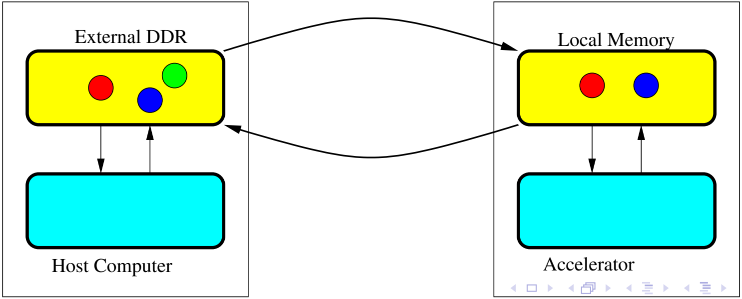
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Load data for block 1 in local memory.



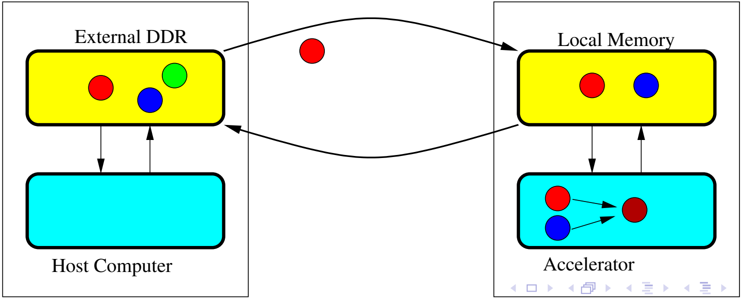
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 1 locally and start loading for block 2.



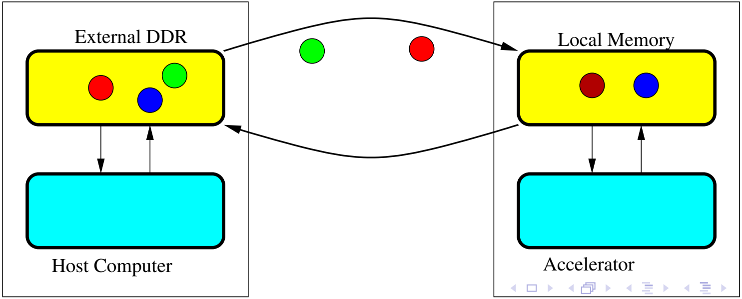
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 1 locally and start loading for block 2.



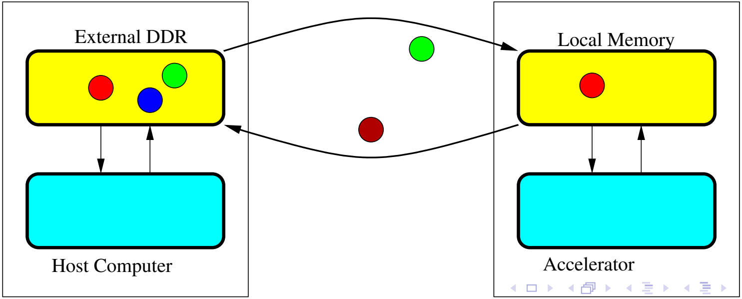
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 1 and finish loading for block 2.



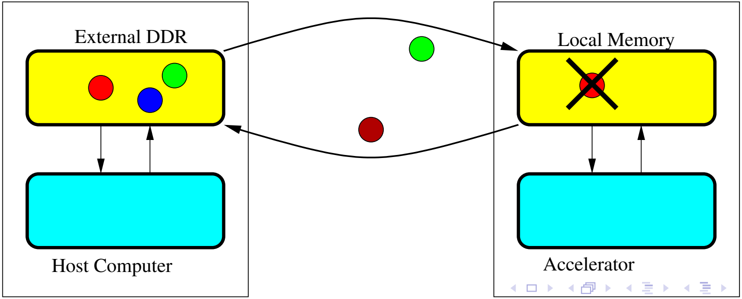
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Overlapping of communications and computations (pipeline).

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Wrong! Analysis for inter-block reuse is necessary.

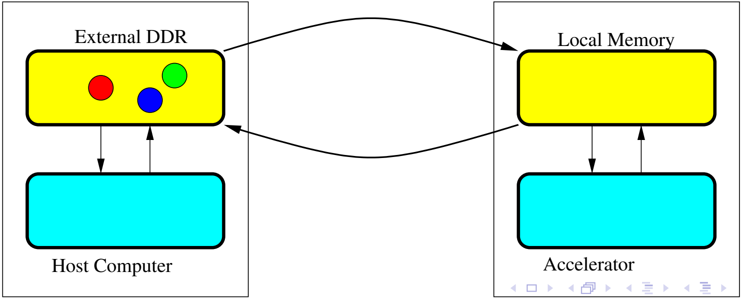


Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).



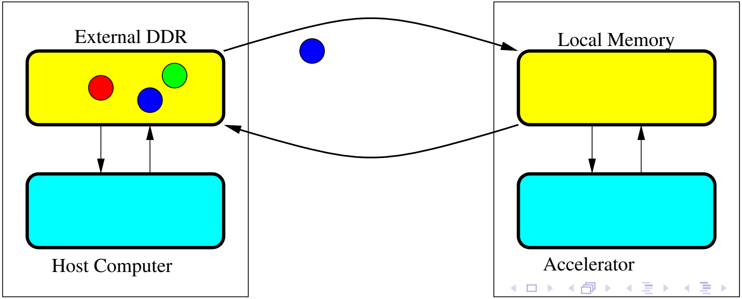
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Loading for block 1.



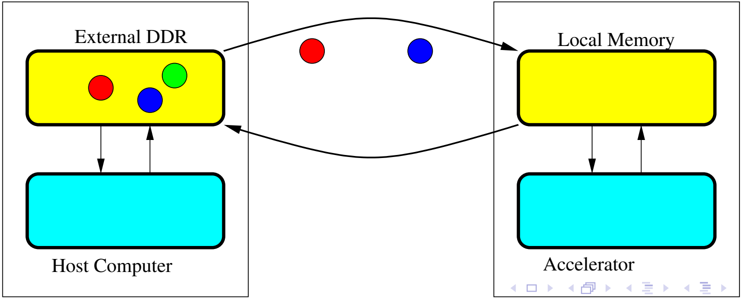
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Loading for block 1.



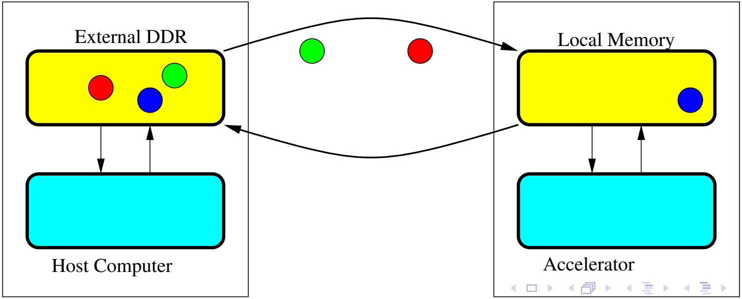
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Loading for block 1, start loading for block 2.



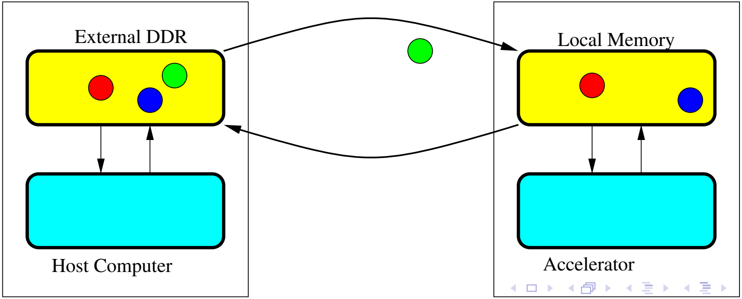
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Loading for block 1, start loading for block 2.



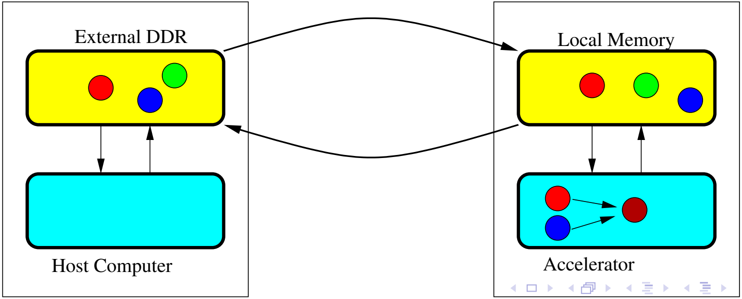
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Compute block 1 locally and finish loading for block 2.



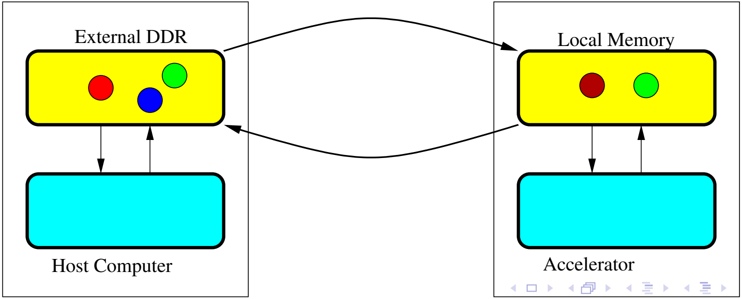
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Finish computing for block 1.



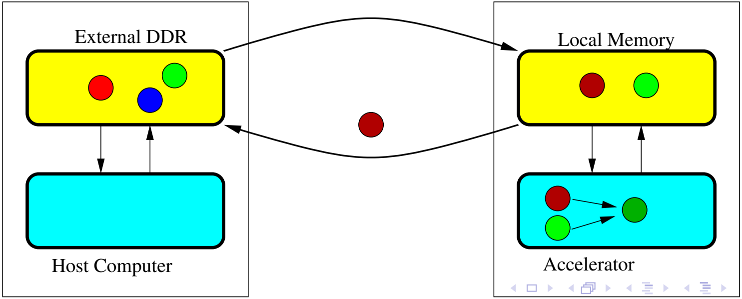
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 1, keep some data and compute block 2.



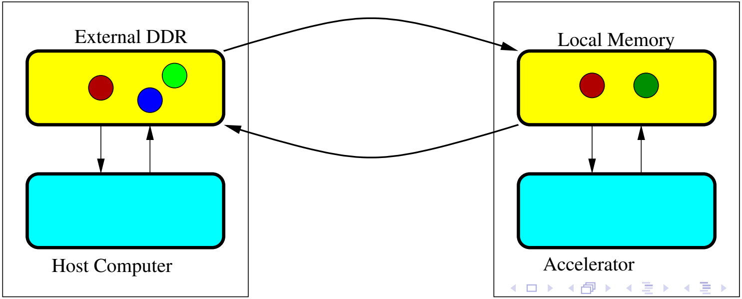
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 1, keep some data and compute block 2.



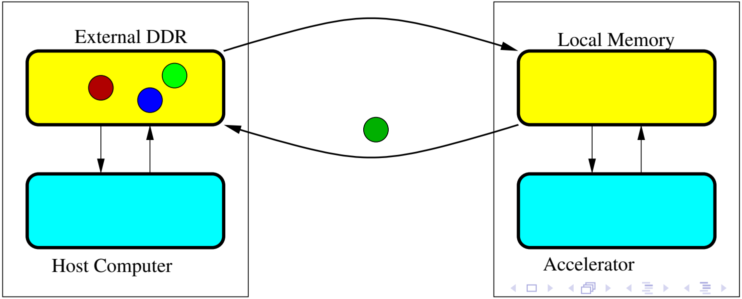
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 2 in distant DDR memory.



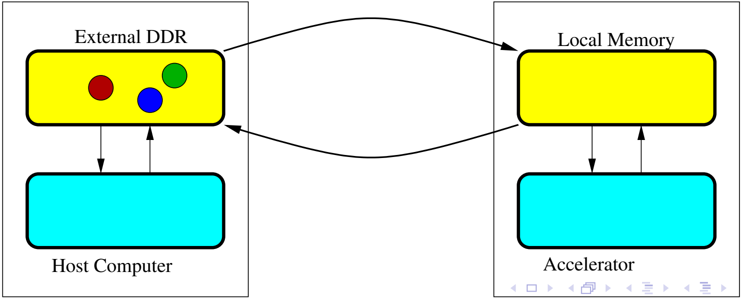
Optimized offloading: pipelining, reuse, local memories

Optimized approach:

- Defines a notion of block (tile).
- Impacts the size of the local memory and the spatial locality.
- Pipeline with local data reuse.

Ex: compute (●, ●) → ● (block 1) then (●, ●) → ● (block 2).

Store results of block 2 in distant DDR memory.



C-to-C-to-VHDL optimizations using Altera C2H

Focus on accelerators limited by bandwidth

- Use the adequate FPGA resources for computation throughput.
- Optimize bandwidth throughput.

C-to-C-to-VHDL optimizations using Altera C2H

Focus on accelerators limited by bandwidth

- Use the adequate FPGA resources for computation throughput.
- Optimize bandwidth throughput.

Apply source-to-source transformations

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

C-to-C-to-VHDL optimizations using Altera C2H

Focus on accelerators limited by bandwidth

- Use the adequate FPGA resources for computation throughput.
- Optimize bandwidth throughput.

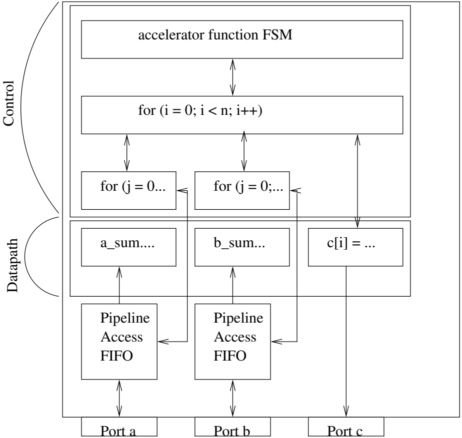
Apply source-to-source transformations

- 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.

Nested finite state machines and pipelined accesses

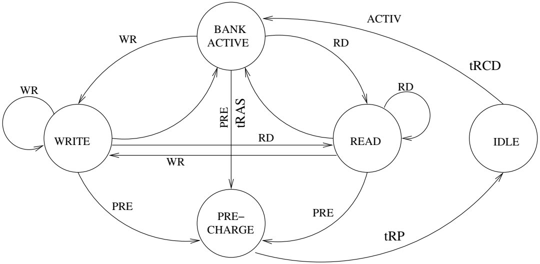



```
void acc(int *a, int *b, int *c) {
  int i, j, k, a_sum, b_sum;
  for(i=0; i<n; i++) {
    for(j=0; j<m; j++)
      a_sum += a[j];
    for(j=0; j<p; j++)
      b_sum += b[j];
    c[i] = a_sum + b_sum;
  }
}
```

DDR SDRAM asymmetric accesses

DDR specifications:

- DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz.
- Successive reads to the same row: $\boxed{\text{10ns}}$.
- Successive reads with a row change: $\boxed{\text{80ns}}$.



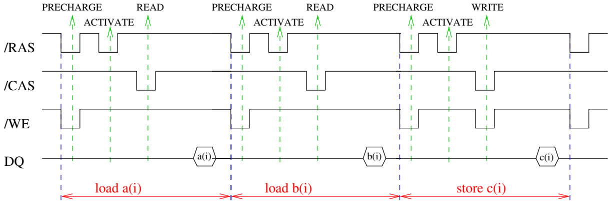
➽ For accelerators exploiting full bandwidth, frequent changes of rows kill performances. Need to use "burst" communications.

Throughput when accessing (asymmetric) DDR memory

Here, with DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz, successive reads to the same row every $\boxed{10\text{ ns}}$, to different rows every $\boxed{80\text{ ns}}$.

☛ A bad spatial DDR locality can kill performances by a factor 8!

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



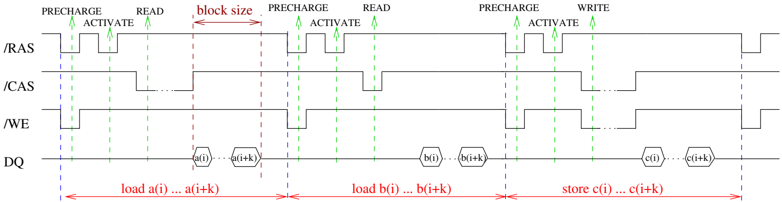
C2H-compiled code: pipelined but time gaps & data thrown away.

Throughput when accessing (asymmetric) DDR memory

Here, with DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz, successive reads to the same row every $\boxed{\text{10 ns}}$, to different rows every $\boxed{\text{80 ns}}$.

☛ A bad spatial DDR locality can kill performances by a factor 8!

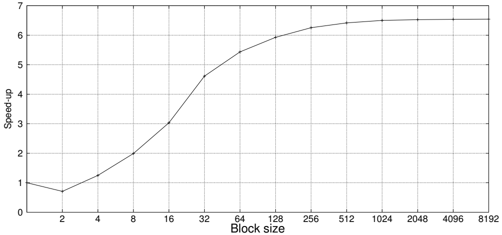
```
void vector_sum (int* __restrict__ a, b, c, int n) {
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}
```



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

Experimental results: typical examples

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



Kernel	Speed-up	ALUT	Dedicated registers	Total registers	Total block memory bits	DSP block 9-bit elements	Max Frequency (MHz > 100)
SA	1	5105	3606	3738	66908	8	205.85
VS0	1	5333	4607	4739	68956	8	189.04
VS1	6.54	10345	10346	11478	269148	8	175.93
MM0	1	6452	4557	4709	68956	40	191.09
MM1	7.37	15255	15630	15762	335196	188	162.02

- SA: system alone.
- VS0 & VS1: vector sum direct & optimized version.
- MM0 & MM1: matrix-matrix multiply direct & optimized ($\sim$ 500 lines!)

Strip-mining and loop distribution

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

➽ strip-mining + loop distribution.

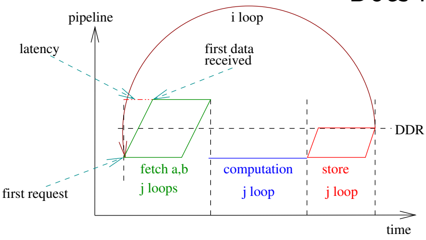
```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j]; //store
}
```

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
}
```

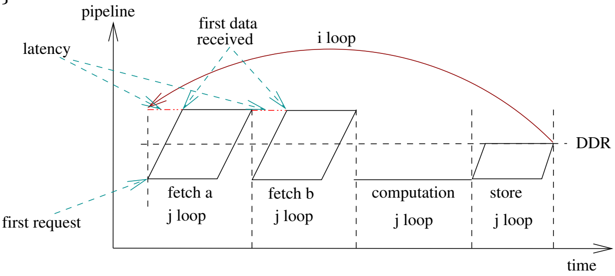
➽ Does not work!



- 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 nested loops: similar to juggling

```
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.
- CPLI (II) = 21! Problem with dependence analyzer and software pipeliner.
- Better behavior (CLPI=3) with *case* statement: by luck.
- Further loop unrolling to get CPLI 1: too complex.
- But still a problem with interleaved DDR accesses!

Emulating nested loops, regrouping transfers

```
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;
- CPLI not equal to 1, unless *restrict* pragma added: but leads to potentially wrong codes.

How to decrease CPLI and generalize to more complex codes?

Outline

1. Context and motivations
2. "Double buffering" execution style
   - Loop tiling and the polyhedral model
   - Overview of the compilation scheme
   - Implementation details: synchronization and memory mapping
3. Communication coalescing

Reminder: all-affine fully-analyzable polyhedral model

Fortran-like C `for` loops:

```
for (i=0, i<=2N; i++)
 c[i] = 0;
for (i=0; i<=N; i++)
 for (j=0; j<=N; j++)
  c[i+j] = c[i+j] + p[i]*q[j];
```

- Affine nested loops: polytopes.
- Multi-dimensional arrays with affine access functions.
- Orders: affine transformations.
- Static control, exact analysis.

Reminder: all-affine fully-analyzable polyhedral model

Fortran-like C `for` loops:

```
for (i=0, i<=2N; i++)
 c[i] = 0;
for (i=0; i<=N; i++)
 for (j=0; j<=N; j++)
  c[i+j] = c[i+j] + p[i]*q[j];
```

- Affine nested loops: polytopes.
- Multi-dimensional arrays with affine access functions.
- Orders: affine transformations.
- Static control, exact analysis.

Typical criticism: such codes do not exist.

Reminder: all-affine fully-analyzable polyhedral model

Fortran-like C for loops:

```
for (i=0, i<=2N; i++)
 c[i] = 0;
for (i=0; i<=N; i++)
 for (j=0; j<=N; j++)
  c[i+j] = c[i+j] + p[i]*q[j];
```

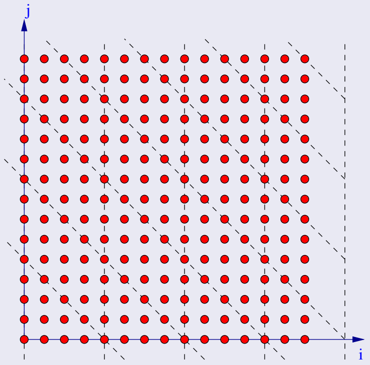
- Affine nested loops: polytopes.
- Multi-dimensional arrays with affine access functions.
- Orders: affine transformations.
- Static control, exact analysis.

Typical criticism: such codes do not exist. But:

- Applicable to specific domains: e.g., signal/video processing.
- Required for static automation, very suitable for HLS.
- Can be limited to the part to analyze: here non-local accesses.
- Central model & source of inspiration for more general cases.
- Recent revival: ISL, PIPS4ALL, PLUTO, GRAPHITE, R-STREAM, COMPAAN, CHUBA, GECOS, ...

Polyhedral model: tiling

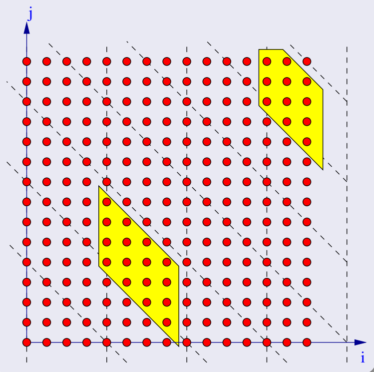
Tiled product of polynomials
$\theta(i,j) = (i+j, i)$



- n loops transformed into n tile loops + n intra-tile loops.
- Expressed from permutable loops: affine function θ, here $\theta : (i,j) \mapsto (i+j, i)$.

Polyhedral model: tiling

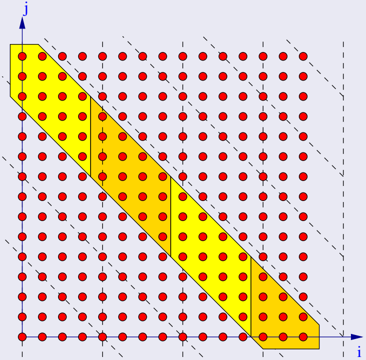
Tiled product of polynomials
$\theta(i,j) = (i+j, i)$



- n loops transformed into n tile loops + n intra-tile loops.
- Expressed from permutable loops: affine function θ, here $\theta : (i,j) \mapsto (i+j, i)$.
- Tile: atomic block operation.
- Increases granularity of computations.
- Enables communication coalescing (hoisting).

Polyhedral model: tiling

Tiled product of polynomials
$\theta(i, j) = (i + j, i)$



- n loops transformed into n tile loops + n intra-tile loops.
- Expressed from permutable loops: affine function θ, here $\theta : (i, j) \mapsto (i + j, i)$.
- Tile: atomic block operation.
- Increases granularity of computations.
- Enables communication coalescing (hoisting).

☛ We focus on a tile strip: double buffering $\simeq$ loop unrolling by 2.

Optimized transfers with maximal intra- & inter-tile reuse

Double buffering style for optimized communications.

- Communication coalescing: each tile T has a $\mathrm{Load}(T)$ and a $\mathrm{Store}(T)$.
- Five pipelined communicating processes for loading, computing, storing.
- Tiling $+$ coarse-grain software pipelining $=$ affine function θ'.
- Transfers are done according to rows: spatial locality for DDR accesses.
- Exploits data reuse: temporal locality $+$ fewer communications.

Optimized transfers with maximal intra- & inter-tile reuse

Double buffering style for optimized communications.

- Communication coalescing: each tile T has a $\mathrm{Load}(T)$ and a $\mathrm{Store}(T)$.
- Five pipelined communicating processes for loading, computing, storing.
- Tiling + coarse-grain software pipelining = affine function θ'.
- Transfers are done according to rows: spatial locality for DDR accesses.
- Exploits data reuse: temporal locality + fewer communications.

Local memory management defines local buffers with reuse.

- Requires lifetime analysis with respect to θ'.
- Lattice-based memory reduction: mix bounding box & sliding window.
- Reduces memory size and provides access functions: $A\vec{i} \bmod \vec{b}$.

Optimized transfers with maximal intra- & inter-tile reuse

Double buffering style for optimized communications.

- Communication coalescing: each tile T has a $\mathrm{Load}(T)$ and a $\mathrm{Store}(T)$.
- Five pipelined communicating processes for loading, computing, storing.
- Tiling + coarse-grain software pipelining = affine function θ'.
- Transfers are done according to rows: spatial locality for DDR accesses.
- Exploits data reuse: temporal locality + fewer communications.

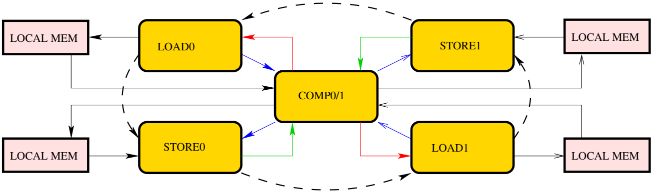
Local memory management defines local buffers with reuse.

- Requires lifetime analysis with respect to θ'.
- Lattice-based memory reduction: mix bounding box & sliding window.
- Reduces memory size and provides access functions: $A\vec{i} \bmod \vec{b}$.

Code generation generates final C code in a linearized form

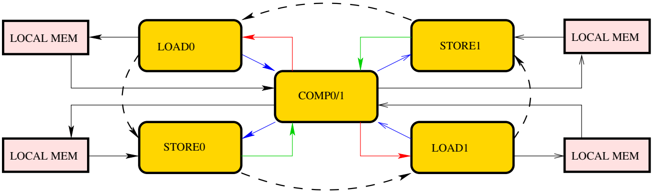
- Placement of FIFO synchronizations.
- Boulet-Feautrier's method for polytope scanning.

Possible organization of load/store and compute processes

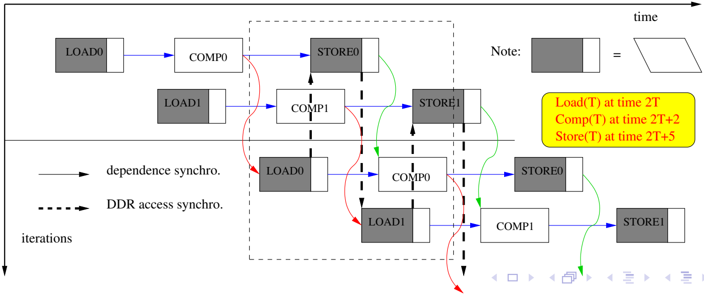


- One function for each communicating process, one memory for each array.
- Dedicated FIFOs of size 1 for synchronizations.
- Transfers through explicit memory accesses.

Possible organization of load/store and compute processes



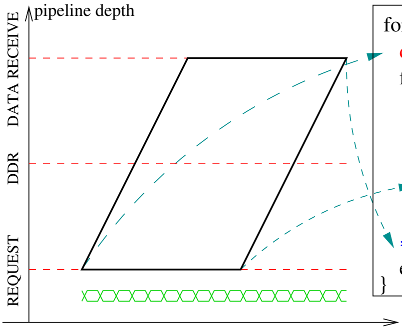
- One function for each communicating process, one memory for each array.
- Dedicated FIFOs of size 1 for synchronizations.
- Transfers through explicit memory accesses.



How to synchronize at C-level?

Need two kinds of synchronizations

- Sequential access to shared resource (computation or DDR).
- Data-flow: wait for data to arrive.




```
for (t=0; t<max_tile_domain; t+=tile_size) {
  dummy_read += *st1_ld0_read;
  for(r=true, tmp=dummy_read; r==true; ) {
     compute next external and local addresses:    Quast
     transfer data from DDR to local memory: *p1=*p2
     if (last iteration) {
        r=false; *ld0_st0_write = 0; tmp = 0;
     }
  }
  *ld0_comp_write = tmp;
  external linearized loop control;
}
```

Generate local memory accesses

Tiling and inter-tile reuse requires local storage: need to define access function to local memory, avoiding "fragmentation".

- Define software pipelining: new schedule dim., function of T.
- Compute liveness and conflicting differences (see hereafter), given transfer sets $\mathrm{Load}(T)$ & $\mathrm{Store}(T)$.
- Fold memory thanks to lattice-based memory allocation (affine function + modulo): existing software Bee+Cl@k.
- Replace in computation function all external accesses by local accesses and generate code for scanning transfer sets.

Memory reuse for scheduled programs

Given an array A with multiple reads/writes and a scheduled program (communicating processes + schedule θ'), target:

- Reduction of the allocation size (size of buffer).
- Simplicity of the addressing functions.

Alternative solutions

- Optimal size with Ehrhart counting ☛ approximations?
- Approximation of maximal number of live values ☛ mapping?
- Bounding box ☛ too inefficient for general live-ranges.

Memory reuse for scheduled programs

Given an array A with multiple reads/writes and a scheduled program (communicating processes + schedule θ'), target:

- Reduction of the allocation size (size of buffer).
- Simplicity of the addressing functions.

Alternative solutions

- Optimal size with Ehrhart counting ☛ approximations?
- Approximation of maximal number of live values ☛ mapping?
- Bounding box ☛ too inefficient for general live-ranges.
- Modular mapping $\vec{i} \mapsto A\vec{i} \bmod b$ ☛ simple and quite efficient.

☛ Not a perfect scheme, does not reach minimal size, but: robust, expressed in terms of θ', usable with approximations.

Example of intermediate buffer: DCT-like example

Two synchronized, pipelined (ASAP) processes, communicating through a shared buffer A.

```
DO b_r = 0, 63
  DO b_c = 0, 63
    DO r = 0, 7
      S: A(b_r, b_c, r, *) = ...
    ENDDO
  ENDDO
ENDDO
```

```
DO b_r = 0, 63
  DO b_c = 0, 63
    DO c = 0, 7
      T: ... = A(b_r, b_c, *, c)
    ENDDO
  ENDDO
ENDDO
```

Example of intermediate buffer: DCT-like example

Two synchronized, pipelined (ASAP) processes, communicating through a shared buffer A.

DO $b_r = 0,\ 63$
  DO $b_c = 0,\ 63$
    DO $r = 0,\ 7$
      S: $A(b_r, b_c, r, *) = \ldots$
    ENDDO
  ENDDO
ENDDO

DO $b_r = 0,\ 63$
  DO $b_c = 0,\ 63$
    DO $c = 0,\ 7$
      T: $\ldots = A(b_r, b_c, *, c)$
    ENDDO
  ENDDO
ENDDO

Full array (no reuse) $64 \times 64 \times 8 \times 8 = 2^{18} = 256K$.

Intuitive solution write in $A(b_r \bmod 2, b_c \bmod 2, r, c)$ (4 blocks)

Best linear allocation 112 with $\sigma = \begin{cases} r \bmod 4 \\ 16(b_r + b_c) + 2r + c \bmod 28 \end{cases}$

Memory reuse for scheduled programs

Given

- An array A with multiple reads and writes.
- Scheduled program or communicating processes, thanks to θ.

Goal

- Reduction of the allocation size (size of buffer).
- Simplicity of the addressing functions.

Solutions

- Optimal size with Ehrhart counting ☛ approximations?
- Approximation of maximal number of live values ☛ mapping?
- Modular mapping $\vec{i} \mapsto A\vec{i} \bmod b$ ☛ simple and quite efficient.

Modular mapping and admissible lattice

Definition (Modular mapping)

A modular mapping $(M, \vec{b})$, with $M \in \mathcal{M}_{p,n}(\mathbb{Z})$ and $\vec{b} \in \mathbb{N}^p$, maps index $\vec{i}$ to $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ in p-dimensional array with shape $\vec{b}$.

Definition (Lifetime analysis)

Two indices $\vec{i}$ and $\vec{j}$ of $\mathbb{Z}^n$ are conflicting $(\vec{i} \bowtie \vec{j})$ if they correspond to two simultaneously live values in the schedule θ.

Define $DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$. ☛ Can be over-approximated.

Modular mapping and admissible lattice

Definition (Modular mapping)

A modular mapping $(M, \vec{b})$, with $M \in \mathcal{M}_{p,n}(\mathbb{Z})$ and $\vec{b} \in \mathbb{N}^p$, maps index $\vec{i}$ to $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ in p-dimensional array with shape $\vec{b}$.

Definition (Lifetime analysis)

Two indices $\vec{i}$ and $\vec{j}$ of $\mathbb{Z}^n$ are conflicting ($\vec{i} \bowtie \vec{j}$) if they correspond to two simultaneously live values in the schedule θ.

Define $DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$. ☛ Can be over-approximated.

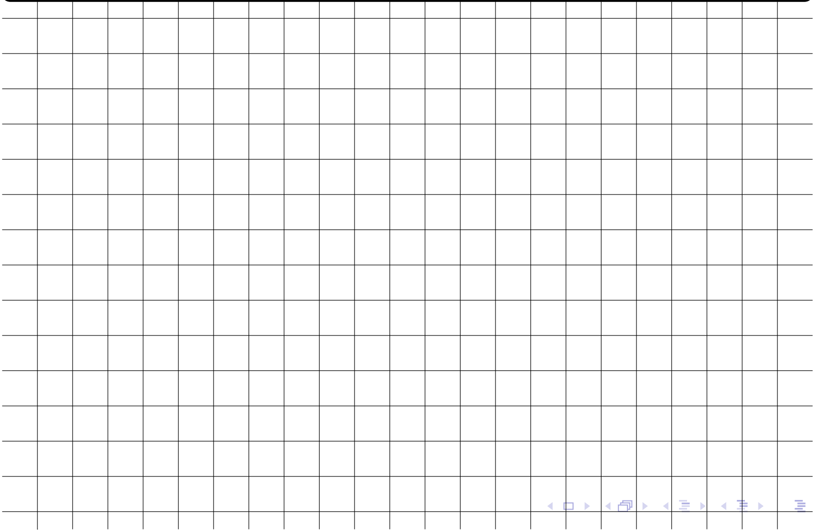
Lemma

The modular mapping $\sigma = (M, \vec{b})$ is valid iff $DS \cap \ker \sigma = \{\vec{0}\}$

☛ $\ker \sigma$ admissible lattice for DS.

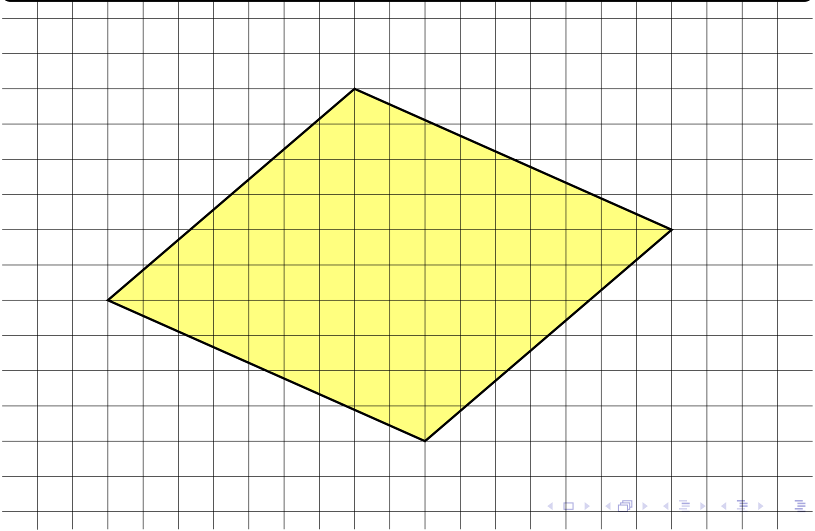
Critical and admissible lattices

Integer points

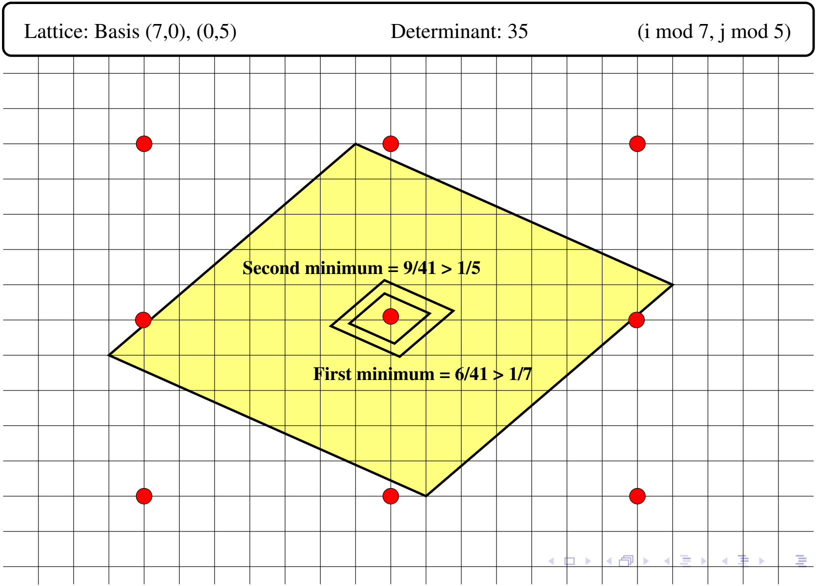


Critical and admissible lattices

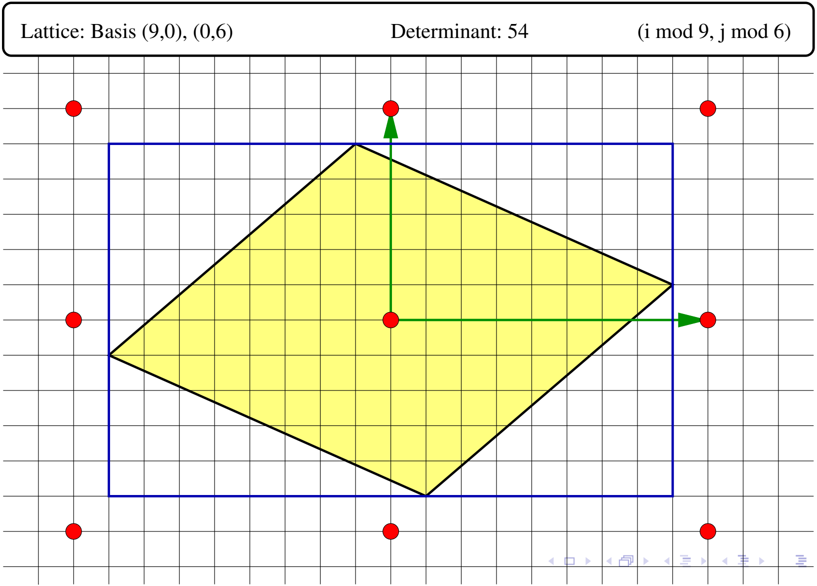
0–Symmetric Polytope: vertices (8,1), (–8,–1), (–1,5), and (1,–5)



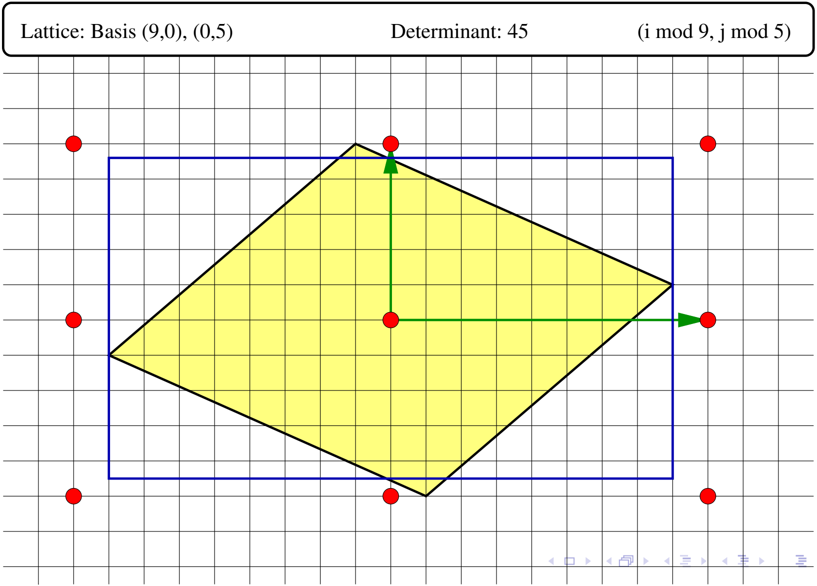
Critical and admissible lattices



Critical and admissible lattices

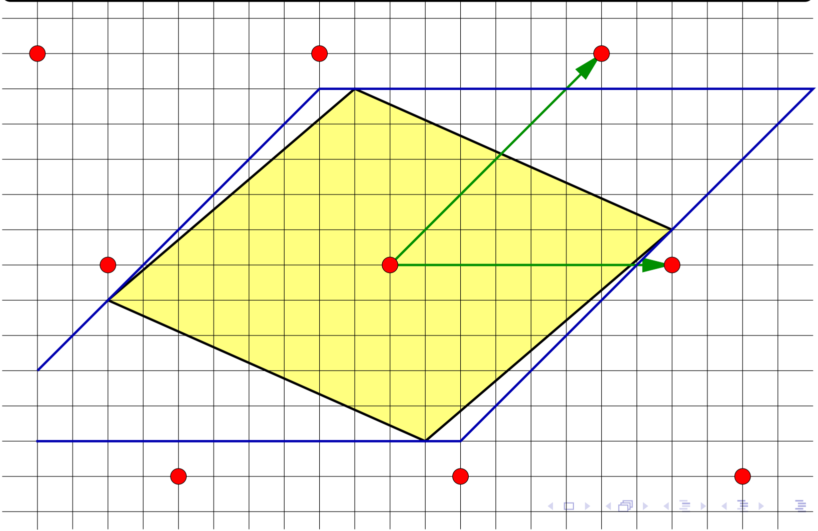


Critical and admissible lattices



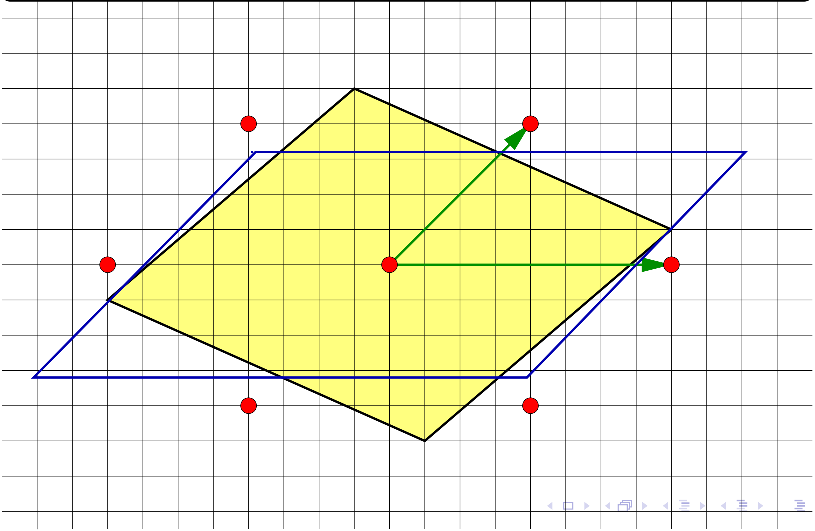
Critical and admissible lattices

Lattice: Basis (8,0), (6,6) Determinant: 48 (i–j mod 8, j mod 6)



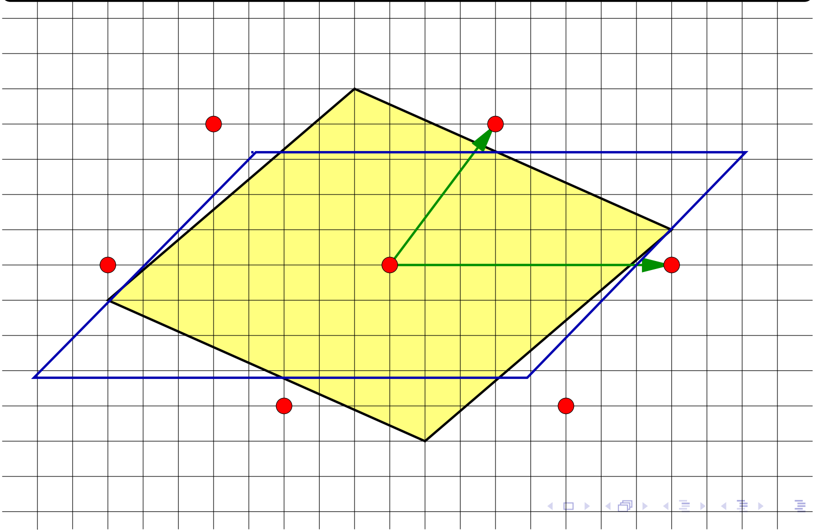
Critical and admissible lattices

Lattice: Basis (8,0), (4,4) Determinant: 32 (i−j mod 8, j mod 4)



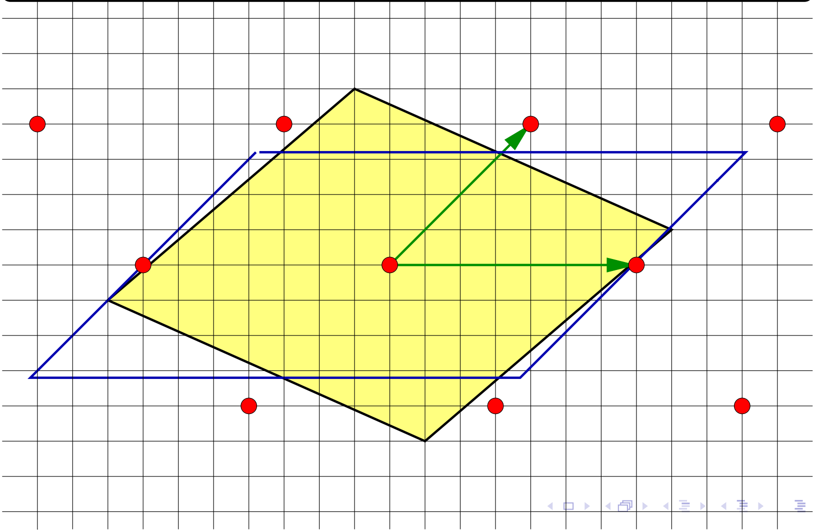
Critical and admissible lattices

Lattice: Basis (8,0), (3,4) Determinant: 32 4i–3j mod 32



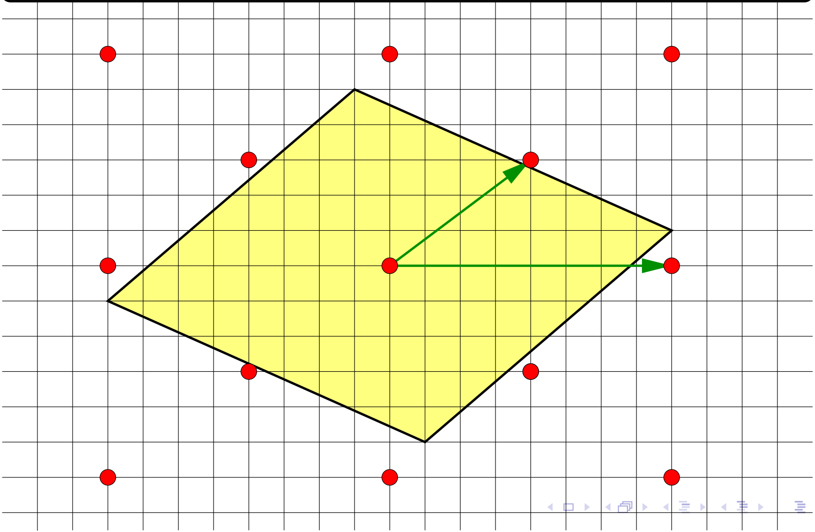
Critical and admissible lattices

Lattice: Basis (7,0), (4,4) Determinant: 28 (i−j mod 7, j mod 4)



Critical and admissible lattices

Critical Lattice: Basis (4,3), (8,0) Determinant: 24 3i–4j mod 24



Lattice-based memory allocation: process

1. Lifetime analysis of the array elements of A, w.r.t. θ.
2. Relation $\bowtie$: Build the polytope of conflicting differences.
3. Admissible lattice: Build an admissible Λ of small determinant.
4. Modulo function: Compute $\sigma = (M, \vec{b})$ such that $\ker \sigma = \Lambda$.
5. Code generation: Define new array A' and replace each occurrence of $A(\vec{i})$ with $A'(M\vec{i} \bmod \vec{b})$.

☛ Not a perfect scheme, does not reach minimal size, but: robust, expressed in terms of θ, usable with approximations.

Remove nested-loop latency by linearization

- Generate two functions for input data transfers.
- Generate (one or) two functions for output data transfers.
- Generate one function for computations.
- Use Boulet-Feautrier to iterate on input/output data sets and computation sets. (Other solutions possible in simple cases.)
- Insert synchronizations according to software pipeline.
- Compile and run! (Actually, with some rewriting for C2H.)

☛ Correct (in theory) code generation. Still need to be validated and improved in terms of code complexity.

Outline

1. Context and motivations
2. "Double buffering" execution style
3. Communication coalescing
   - Communication coalescing: related work
   - Exact inter-tile data reuse in a tile strip
   - Extensions to more general situations

Related work: parallel languages & scratchpad memories

- Compiler-directed scratchpad memory hierarchy design & management: Kandemir, Choudhary, DAC'02.
- Effective communication coalescing for data-parallel applications: Chavarría-Miranda, Mellor-Crummey, PPoPP'05.
- Communication optimizations for fine-grained UPC applications: Chen, Iancu, Yelick, PACT'05.
- DRDU: A data reuse analysis technique for efficient scratchpad memory management: Issenin, Borckmeyer, Miranda, Dutt. ACM TODAES 2007.
- Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories: Baskaran, Bondhugula, Krishnam., Ramanujam, Rountev, Sadayappan, PPoPP'08.
- A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction: Leung, Vasilache, Meister, Baskaran, Wohlford, Bastoul, Lethin, GPGPU'10.
- A reuse-aware prefetching scheme for scratchpad memory: Cong, Huang, Liu, Zou, DAC'11.
- PIPS is not (just) polyhedral software: Amini, Ancourt, Coelho, Creusillet, Guelton, Irigoin, Jouvelot, Keryell, Villalon, IMPACT'11.

Communication coalescing: main principles

Hoist communications out of loops (out of tile or out of tile strip).

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S(i,j)
  endfor
endfor
```

```
for (I=0; I<N; I+=b)
  for (J=0; J<N; J+=b)
    Transfer(I,J)
    for (i=I; i<min(I+b,N); i++)
      for (j=J; j<min(J+b,N); j++)
        S(i,j)
      endfor
    endfor
  endfor
endfor
```

```
for (I=0; I<N; I+=b)
  Transfer(I)
  for (J=0; J<N; J+=b)
    for (i=I; i<min(I+b,N); i++)
      for (j=J; j<min(J+b,N); j++)
        S(i,j)
      endfor
    endfor
  endfor
endfor
```

Static scratch-pad optimizations

- Decides statically which array portions will remain in SPM.
- Granularity of arrays and function calls.

Dynamic scratch-pad optimizations

- Make a copy of distant memory before a tile or before a tile strip.
- Work at the granularity of array sections = approximation.
- Only "regular" inter-tile reuse (null space of affine functions or shifts).
- Apparently, no pipelining/overlapping (except in RStream).

Communication coalescing: main principles

Hoist communications out of loops (out of tile or out of tile strip).

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S(i,j)
  endfor
endfor
```

```
for (I=0; I<N; I+=b)
  for (J=0; J<N; J+=b)
    Transfer(I,J)
    for (i=I; i<min(I+b,N); i++)
      for (j=J; j<min(J+b,N); j++)
        S(i,j)
      endfor
    endfor
  endfor
endfor
```

```
for (I=0; I<N; I+=b)
  Transfer(I)
  for (J=0; J<N; J+=b)
    for (i=I; i<min(I+b,N); i++)
      for (j=J; j<min(J+b,N); j++)
        S(i,j)
      endfor
    endfor
  endfor
endfor
```

Static scratch-pad optimizations

- Decides statically which array portions will remain in SPM.
- Granularity of arrays and function calls.

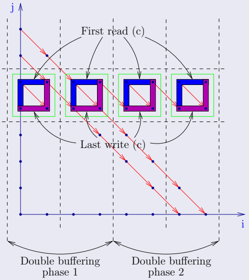
Dynamic scratch-pad optimizations ☛ but unclear & incomplete

- Make a copy of distant memory before a tile or before a tile strip.
- Work at the granularity of array sections = approximation.
- Only "regular" inter-tile reuse (null space of affine functions or shifts).
- Apparently, no pipelining/overlapping (except in RStream).

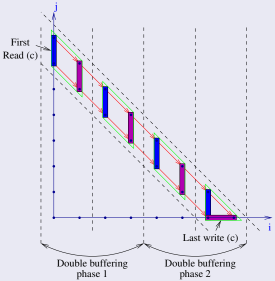
Loop tiling: impact on reuse and communication

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        c[i+j] = c[i+j] + p[i]*q[j];
```

$(i, j) \mapsto (n - j - 1, i)$



$(i, j) \mapsto (i + j, i)$

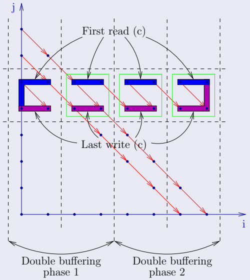


Load $\simeq$ first reads $\cap$ tile domain. **Store** $\simeq$ last writes $\cap$ tile domain.

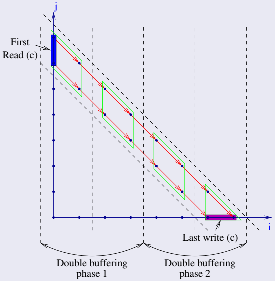
Loop tiling: impact on reuse and communication

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        c[i+j] = c[i+j] + p[i]*q[j];
```

$(i, j) \mapsto (n - j - 1, i)$



$(i, j) \mapsto (i + j, i)$



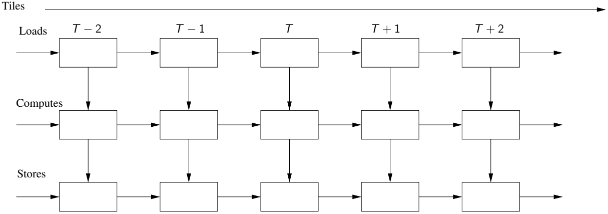
Load $\simeq$ first reads $\cap$ tile domain. **Store** $\simeq$ last writes $\cap$ tile domain.

General specification of data transfers

Definition

- Load(T): data loaded from DDR just before executing tile T.
- Store(T): data stored to DDR just after T.
- In(T): data read before being written in the tile T.
- Out(T): data written by the tile T.

Minimal dependence structure



Goals

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

What do we put in $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$?

Extreme solutions

- $\forall T$, $\mathrm{Load}(T) = \emptyset$ except $\mathrm{Load}(T_0) =$ copy of all the memory involved in the tile strip ☛ no pipelining and no overlapping.
- $\forall T$, $\mathrm{Load}(T) = \mathrm{In}(T)$, $\mathrm{Store}(T) = \mathrm{Out}(T)$ where $\mathrm{In}(T) =$ data read before written in T, $\mathrm{Out}(T) =$ data written in T ☛ no inter-tile reuse.

What do we put in $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$?

Extreme solutions

- $\forall T$, $\mathrm{Load}(T) = \emptyset$ except $\mathrm{Load}(T_0) =$ copy of all the memory involved in the tile strip ☛ no pipelining and no overlapping.
- $\forall T$, $\mathrm{Load}(T) = \mathrm{In}(T)$, $\mathrm{Store}(T) = \mathrm{Out}(T)$ where $\mathrm{In}(T) =$ data read before written in T, $\mathrm{Out}(T) =$ data written in T ☛ no inter-tile reuse.

Exact situation with ALAP loads and ASAP stores

- Always reuse local data: intra- and inter-tile reuse in a tile strip.
- Remote store only after last write ☛ external memory not up-to-date.
- Minimize each local live-range ☛ bounding box not enough.

What do we put in $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$?

Extreme solutions

- $\forall T$, $\mathrm{Load}(T) = \emptyset$ except $\mathrm{Load}(T_0) =$ copy of all the memory involved in the tile strip ☛ no pipelining and no overlapping.
- $\forall T$, $\mathrm{Load}(T) = \mathrm{In}(T)$, $\mathrm{Store}(T) = \mathrm{Out}(T)$ where $\mathrm{In}(T) =$ data read before written in T, $\mathrm{Out}(T) =$ data written in T ☛ no inter-tile reuse.

Exact situation with ALAP loads and ASAP stores

- Always reuse local data: intra- and inter-tile reuse in a tile strip.
- Remote store only after last write ☛ external memory not up-to-date.
- Minimize each local live-range ☛ bounding box not enough.

To avoid useless transfers and reduce local lifetimes

- $\mathrm{Load}(T) = \mathrm{In}(T) \setminus \{\mathrm{In}(t < T) \cup \mathrm{Out}(t < T)\}$
- $\mathrm{Store}(T) = \mathrm{Out}(T) \setminus \mathrm{Out}(t > T)$

What do we put in $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$?

Extreme solutions

- $\forall T$, $\mathrm{Load}(T) = \emptyset$ except $\mathrm{Load}(T_0) =$ copy of all the memory involved in the tile strip ☛ no pipelining and no overlapping.
- $\forall T$, $\mathrm{Load}(T) = \mathrm{In}(T)$, $\mathrm{Store}(T) = \mathrm{Out}(T)$ where $\mathrm{In}(T) =$ data read before written in T, $\mathrm{Out}(T) =$ data written in T ☛ no inter-tile reuse.

Exact situation with ALAP loads and ASAP stores

- Always reuse local data: intra- and inter-tile reuse in a tile strip.
- Remote store only after last write ☛ external memory not up-to-date.
- Minimize each local live-range ☛ bounding box not enough.

To avoid useless transfers and reduce local lifetimes

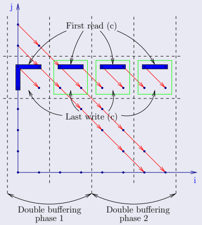
- $\mathrm{Load}(T) = \mathrm{In}(T) \setminus \{\mathrm{In}(t < T) \cup \mathrm{Out}(t < T)\}$
- $\mathrm{Store}(T) = \mathrm{Out}(T) \setminus \mathrm{Out}(t > T)$

or, equivalently, defined by optimization

- $\mathrm{Load}(T) = \{\vec{m} \mid \mathrm{FirstOpReadBeforeWrite}(\vec{m}) \in T\}$
- $\mathrm{Store}(T) = \{\vec{m} \mid \mathrm{LastOpWrite}(\vec{m}) \in T\}$

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i, j) \mapsto (n - j - 1, i)$



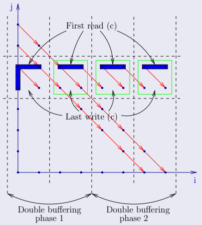
Reads of $c[m]$ as a function of (i, j):

$$\begin{cases} i + j = {\color{red} m} \\ 0 \leq i \leq {\color{red} n} - 1,\ 0 \leq j \leq {\color{red} n} - 1 \end{cases}$$

blue=constant, red=parameter

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i,j) \mapsto (n-j-1, i)$



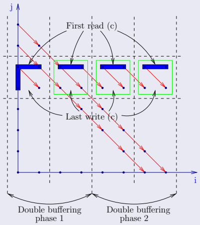
Introduction of the change of basis $(i,j) \mapsto (i' = n-1-j, j' = i)$:

$$\begin{cases} i + j = m,\ i' = n - 1 - j,\ j' = i \\ 0 \le i \le n-1,\ 0 \le j \le n-1 \end{cases}$$

blue=constant, red=parameter

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i,j) \mapsto (n-j-1, i)$



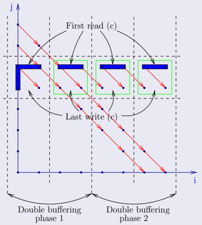
Tiling $(I,J) = (\lfloor \frac{i'}{b} \rfloor, \lfloor \frac{j'}{b} \rfloor)$, I parameter:

$$\begin{cases} i + j = m,\ i' = n - 1 - j,\ j' = i \\ 0 \le i \le n-1,\ 0 \le j \le n-1 \\ bI \le i' \le b(I+1) - 1 \\ bJ \le j' \le b(J+1) - 1 \end{cases}$$

blue=constant, red=parameter

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i, j) \mapsto (n - j - 1, i)$



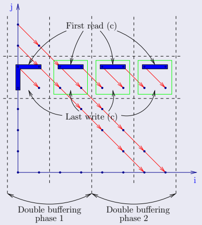
Use PIP to find the first read in the tile strip, i.e., lexicographic minimum of (I, J, i', j'):

$$\min_{\prec_{lex}} \begin{cases} i + j = m,\ i' = n - 1 - j,\ j' = i \\ 0 \leq i \leq n - 1,\ 0 \leq j \leq n - 1 \\ bI \leq i' \leq b(I + 1) - 1 \\ bJ \leq j' \leq b(J + 1) - 1 \end{cases}$$

blue=constant, red=parameter

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i, j) \mapsto (n - j - 1, i)$



Use PIP to find the first read in the tile strip, i.e., lexicographic minimum of (I, J, i', j'):

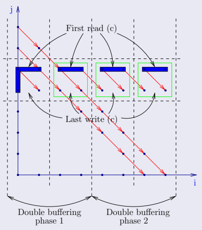
$$\min_{\prec_{lex}} \begin{cases} i + j = {\color{red}m},\ i' = {\color{red}n} - 1 - j,\ j' = i \\ 0 \le i \le {\color{red}n} - 1,\ 0 \le j \le {\color{red}n} - 1 \\ {\color{blue}b}{\color{red}I} \le i' \le {\color{blue}b}({\color{red}I} + 1) - 1 \\ {\color{blue}b}J \le j' \le {\color{blue}b}(J + 1) - 1 \end{cases}$$

blue=10, red=parameter

if $(-10I + N - m \ge 0)$
  if $(10I - N + m + 9 \ge 0)$ /* vertical band of elements, first tile */
    $(J, ii, jj, i, j) = (0, N - m, 0, 0, m)$
  else $\perp$ /* means undefined */
else
  if $(-10I + 2N - m \ge 0)$
    if $(-10I + N - m + 9 \ge 0)$ /* horizontal band, first tile */
      $(J, ii, jj, i, j) = (0, 10I, 10I - N + m, 10I - N + m, N - 10I)$
    else with $k = \lfloor \frac{N+9m+9}{10} \rfloor$ /* generic horizontal case */
      $(J, ii, jj, i, j) = (I + m - k, 10I, 10I - N + m, 10I - N + m, N - 10I)$
  else $\perp$ /* undefined */

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i, j) \mapsto (n - j - 1, i)$



Use PIP to find the first read in the tile strip, i.e., lexicographic minimum of (I, J, i', j'):

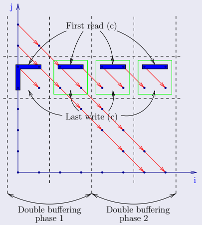
$$\min_{\prec_{lex}} \begin{cases} i + j = {\color{red}m},\ i' = {\color{red}n} - 1 - j,\ j' = i \\ 0 \le i \le {\color{red}n} - 1,\ 0 \le j \le {\color{red}n} - 1 \\ {\color{blue}b}{\color{red}I} \le i' \le {\color{blue}b}({\color{red}I} + 1) - 1 \\ {\color{blue}b}J \le j' \le {\color{blue}b}(J + 1) - 1 \end{cases}$$

blue=10, red=parameter

if $(-10I + N - m \ge 0)$
   if $(10I - N + m + 9 \ge 0)$ /* vertical band of elements, first tile */
      $(i, j) = (0, m)$
   else $\perp$
else
   if $(-10I + 2N - m \ge 0)$
      if $(-10I + N - m + 9 \ge 0)$ /* horizontal band, first tile */
         $(i, j) = (10I - N + m, N - 10I)$
      else with $k = \lfloor \frac{N+9m+9}{10} \rfloor$ /* generic horizontal case */
         $(i, j) = (10I - N + m, N - 10I)$
   else $\perp$ /* means undefined */

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i,j) \mapsto (n-j-1, i)$



Use PIP to find the first read in the tile strip, i.e., lexicographic minimum of (I, J, i', j'):

$$\min_{\prec_{lex}} \begin{cases} i + j = m,\ i' = n - 1 - j,\ j' = i \\ 0 \le i \le n-1,\ 0 \le j \le n-1 \\ bI \le i' \le b(I+1) - 1 \\ bJ \le j' \le b(J+1) - 1 \end{cases}$$

blue=10, red=parameter

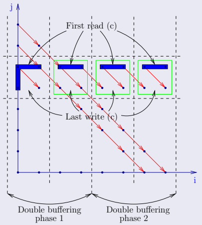
if $(-10I + N - m \ge 0)$
  if $(10I - N + m + 9 \ge 0)$
    $(i,j) = (0, m)$ /* vertical portion of c */
  else $\perp$
else
  if $(-10I + 2N - m \ge 0)$
    $(i,j) = (10I - N + m, N - 10I)$ /* horizontal portion of c */
  else $\perp$ /* means undefined */

This gives the array elements whose first access is a read:

$\{m \mid \max(0, N - 10I - 9) \le m \le N - 10I\} \cup \{m \mid N - 10I + 1 \le m \le 2N - 10I\}$

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i,j) \mapsto (n-j-1, i)$



Use PIP to find the first read in the tile strip, i.e., lexicographic minimum of (I, J, i', j'):

$$\min_{\prec_{lex}} \begin{cases} i+j = m,\ i' = n-1-j,\ j' = i \\ 0 \le i \le n-1,\ 0 \le j \le n-1 \\ bI \le i' \le b(I+1)-1 \\ bJ \le j' \le b(J+1)-1 \end{cases}$$

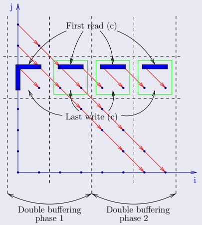
blue=10, red=parameter

After simplification:

$$\mathrm{FirstOpRead}(m) = \{(i,j) \mid (i,j) = (0,m),\ 0 \le m,\ n-10-10I \le m \le n-1-10I\}$$
$$\cup \{(i,j) \mid (i,j) = (10I - n + 1 + m, n - 1 - 10I),\ n - 10I \le m \le 2n-2-10I\}$$

Example: $\mathrm{Load}(J)$ for a $b \times b$ tile indexed by J

$(i, j) \mapsto (n - j - 1, i)$



Use PIP to find the first read in the tile strip, i.e., lexicographic minimum of (I, J, i', j'):

$$\min_{\prec_{lex}} \begin{cases} i + j = \textcolor{red}{m},\ i' = \textcolor{red}{n} - 1 - j,\ j' = i \\ 0 \leq i \leq \textcolor{red}{n} - 1,\ 0 \leq j \leq \textcolor{red}{n} - 1 \\ \textcolor{blue}{b}\textcolor{red}{I} \leq i' \leq \textcolor{blue}{b}(\textcolor{red}{I} + 1) - 1 \\ \textcolor{blue}{b}J \leq j' \leq \textcolor{blue}{b}(J + 1) - 1 \end{cases}$$

blue=10, red=parameter

After simplification:

$$\mathrm{FirstOpRead}(m) = \{(i, j) \mid (i, j) = (0, m),\ 0 \leq m,\ n - 10 - 10I \leq m \leq n - 1 - 10I\}$$
$$\cup \{(i, j) \mid (i, j) = (10I - n + 1 + m, n - 1 - 10I),\ n - 10I \leq m \leq 2n - 2 - 10I\}$$

Introduction of tile constraints and expression of m as a function of J:

$$\mathrm{FirstReadInTile}(J) = \{m \mid \max(0, n - 10I - 10) \leq m \leq n - 1 - 10I,\ J = 0\}$$
$$\cup \{m \mid \max(1, 10J) \leq m + 10I - n + 1 \leq \min(n - 1, 10J + 9)\}$$

Weaknesses and potential for improvements

Note

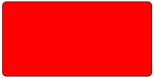
- This is the first process to automate double-buffering with intra- and inter-tile reuse, and entirely at C level.
- Combination of several polyhedral techniques: tiling, code analysis, memory reuse with modulo (not explained here), polyhedral code generation.

Weaknesses

- Needs "tilable" portion of code.
- Needs exact analysis of data usage ☛ approximations?
- Needs constant tile size ☛ parameterization?
  - Recompile (analysis & code generation) for each tile size.
  - Painful (hand-made) code specialization for each tile size.
  - Local memory size known only at the end of the process.

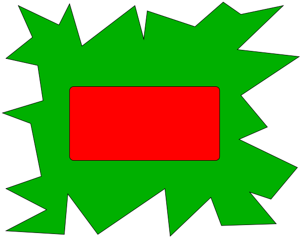
Reminder: beyond the polyhedral model

Polyhedral model.



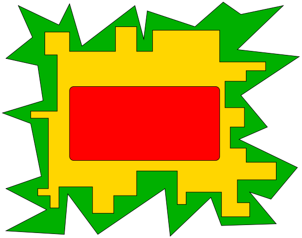
Reminder: beyond the polyhedral model

Polyhedral model.
Real life.



Reminder: beyond the polyhedral model

Polyhedral model.
Real life.

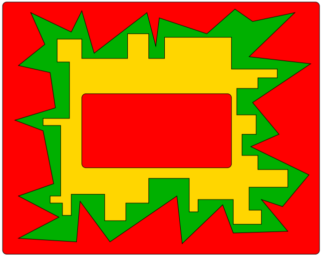


Extensions.

- Non-affine constraints.
- Non-static control, while loops.
- Beyond induction variables.

Reminder: beyond the polyhedral model

Polyhedral model.
Real life.



Extensions.

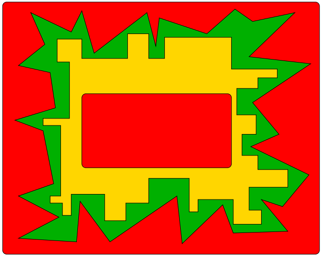
- Non-affine constraints.
- Non-static control, while loops.
- Beyond induction variables.

Approximations.

- Dependences, lifetime, data & iteration domains, etc.
- Array region analysis (Creusillet).
- Runtime info., trace analysis.

Reminder: beyond the polyhedral model

Polyhedral model.
Real life.



Extensions.

- Non-affine constraints.
- Non-static control, while loops.
- Beyond induction variables.

Approximations.

- Dependences, lifetime, data & iteration domains, etc.
- Array region analysis (Creusillet).
- Runtime info., trace analysis.

- $\mathrm{In}(T)$: data read before being written in the tile T.
- $\mathrm{Out}(T)$: data written by the tile T.
- $\overline{\mathrm{In}}(T)$: possibly read before being written, over-approximation of $\mathrm{In}(T)$.
- $\overline{\mathrm{Out}}(T)$: data possibly written, over-approximation of $\mathrm{Out}(T)$.
- $\underline{\mathrm{Out}}(T)$: data provably written, under-approximation of $\mathrm{Out}(T)$.

Approximation scheme for $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$

Valid approximated loads and stores

(i) Load at least the exact amount of data:

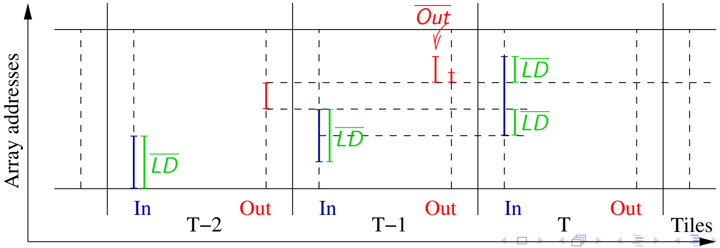
$\overline{\mathrm{In}}(T) \setminus \underline{\mathrm{Out}}(t < T) \subseteq \mathrm{Load}(t \leq T)$ ☛ need to over-approximate

(ii) Do not overwrite possibly locally-defined data:

$\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$ ☛ be careful with over-loading

(iii) Preload any data that may be written but not for sure:

$\mathrm{Store}(T) \setminus \underline{\mathrm{Out}}(t \leq T) \subseteq \mathrm{Load}(t \leq T)$ ☛ risk of storing garbage



Approximation scheme for $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$

Valid approximated loads and stores

(i) Load at least the exact amount of data:

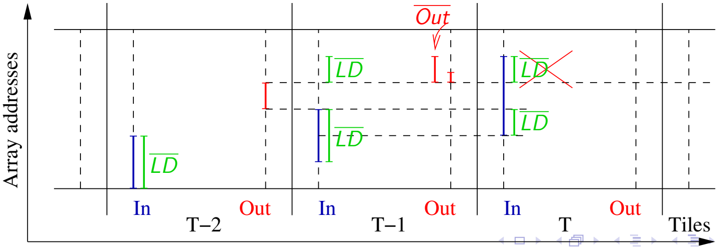
$\overline{\mathrm{In}}(T) \setminus \underline{\mathrm{Out}}(t < T) \subseteq \mathrm{Load}(t \leq T)$ ☛ need to over-approximate

(ii) Do not overwrite possibly locally-defined data:

$\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$ ☛ be careful with over-loading

(iii) Preload any data that may be written but not for sure:

$\mathrm{Store}(T) \setminus \underline{\mathrm{Out}}(t \leq T) \subseteq \mathrm{Load}(t \leq T)$ ☛ risk of storing garbage



Approximation is (unexpectedly) feasible!

Intuition for loading ALAP and storing ASAP

- Store x just after T if x is never written after T, i.e., $x \notin \overline{\mathrm{Out}}(t > T)$.
- Preload x if written, not for sure: $x \in \overline{\mathrm{Out}}(t \leq T_{\mathrm{max}}) \setminus \underline{\mathrm{Out}}(t \leq T_{\mathrm{max}})$.
- Load a value x always before it may be written, i.e., $x \notin \overline{\mathrm{Out}}(t < T)$.

Approximation is (unexpectedly) feasible!

Intuition for loading ALAP and storing ASAP

- Store x just after T if x is never written after T, i.e., $x \notin \overline{\mathrm{Out}}(t > T)$.
- Preload x if written, not for sure: $x \in \overline{\mathrm{Out}}(t \leq T_{\max}) \setminus \underline{\mathrm{Out}}(t \leq T_{\max})$.
- Load a value x always before it may be written, i.e., $x \notin \overline{\mathrm{Out}}(t < T)$.

Solution with set equations Don't read! ☺

$$\begin{cases} \overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t > T) \subseteq \mathrm{Store}(T) & \text{(data possibly written)} \\ \overline{\mathrm{In}}'(T) = \overline{\mathrm{In}}(T) \cup (\mathrm{Store}(T) \setminus \underline{\mathrm{Out}}(T)) & \text{(all data that are ``read'')} \\ \overline{\mathrm{Ra}}(T) = \overline{\mathrm{In}}'(T) \setminus \underline{\mathrm{Out}}(t < T) & \text{(all data that need a remote access)} \\ \mathrm{Load}(T) = \left(\overline{\mathrm{In}}'(T) \cup (\overline{\mathrm{Out}}(T) \cap \overline{\mathrm{Ra}}(t > T))\right) \setminus \left(\overline{\mathrm{In}}'(t < T) \cup \overline{\mathrm{Out}}(t < T)\right) & \end{cases}$$

Approximation is (unexpectedly) feasible!

Intuition for loading ALAP and storing ASAP

- Store x just after T if x is never written after T, i.e., $x \notin \overline{\mathrm{Out}}(t > T)$.
- Preload x if written, not for sure: $x \in \overline{\mathrm{Out}}(t \leq T_{\max}) \setminus \underline{\mathrm{Out}}(t \leq T_{\max})$.
- Load a value x always before it may be written, i.e., $x \notin \overline{\mathrm{Out}}(t < T)$.

Solution with set equations Don't read! ☺

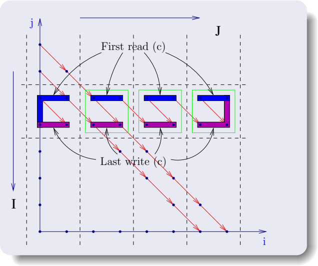
$$\begin{cases} \overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t > T) \subseteq \mathrm{Store}(T) & \text{(data possibly written)} \\ \overline{\mathrm{In}}'(T) = \overline{\mathrm{In}}(T) \cup (\mathrm{Store}(T) \setminus \underline{\mathrm{Out}}(T)) & \text{(all data that are "read")} \\ \overline{\mathrm{Ra}}(T) = \overline{\mathrm{In}}'(T) \setminus \underline{\mathrm{Out}}(t < T) & \text{(all data that need a remote access)} \\ \mathrm{Load}(T) = \left(\overline{\mathrm{In}}'(T) \cup (\overline{\mathrm{Out}}(T) \cap \overline{\mathrm{Ra}}(t > T))\right) \setminus \left(\overline{\mathrm{In}}'(t < T) \cup \overline{\mathrm{Out}}(t < T)\right) & \end{cases}$$

Solution by optimization Don't read! ☺

- $\overline{\mathrm{In}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{In}}(T)\}$ (first time it is read).
- $\overline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{Out}}(T)\}$ (first time it may be written).
- $\underline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \underline{\mathrm{Out}}(T)\}$ (first time it is written for sure).

then combine to get $T(\vec{m}) = \min(\overline{\mathrm{Out}}(\vec{m}), \underline{\mathrm{Out}}(\vec{m}), \overline{\mathrm{In}}(\vec{m}))$, unless $\underline{\mathrm{Out}}(\vec{m}) \leq_{lex} \overline{\mathrm{In}}(\vec{m})$ in which case $T(\vec{m}) = -\infty$ (no need to load).

Parameterization is (unexpectedly) feasible!



Tiling:

- $(i, j) \mapsto (i', j') = (n - j - 1, i)$

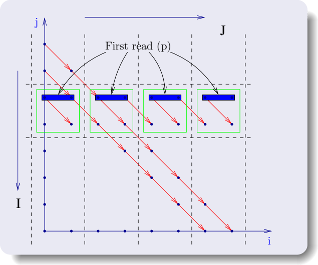
Parameters:

- (I,J): first index in tile.
- n: loop bound, b: tile size.

Transfers $(m = i + j = j' + n - i' - 1)$:

- Load_p, Load_q, Load_c, Store_c.

Parameterization is (unexpectedly) feasible!



Tiling:

- $(i, j) \mapsto (i', j') = (n - j - 1, i)$

Parameters:

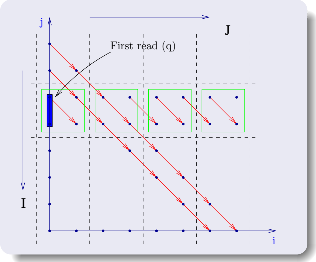
- (I,J): first index in tile.
- n: loop bound, b: tile size.

Transfers $(m = i + j = j' + n - i' - 1)$:

- Load_p, Load_q, Load_c, Store_c.

$$\mathrm{Load}_p = \{m \mid 1 - b \leq I \leq n - 1,\ 0 \leq m \leq n - 1,\ J \leq m \leq J + b - 1\}$$

Parameterization is (unexpectedly) feasible!



Tiling:

- $(i,j) \mapsto (i',j') = (n-j-1, i)$

Parameters:

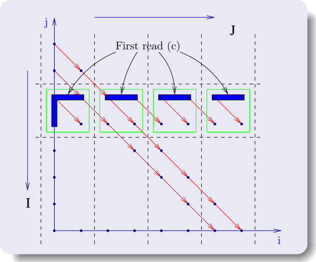
- (I,J): first index in tile.
- n: loop bound, b: tile size.

Transfers $(m = i + j = j' + n - i' - 1)$:

- Load_p, Load_q, Load_c, Store_c.

$$\mathrm{Load}_p = \{m \mid 1-b \leq I \leq n-1,\ 0 \leq m \leq n-1,\ J \leq m \leq J+b-1\}$$
$$\mathrm{Load}_q = \{m \mid 1-b \leq J \leq n-1,\ J \leq 0,\ 0 \leq m \leq n-1,\ 1 \leq n-I-m \leq b\}$$

Parameterization is (unexpectedly) feasible!



Tiling:

- $(i,j) \mapsto (i',j') = (n-j-1, i)$

Parameters:

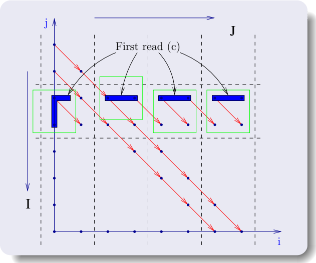
- (I,J): first index in tile.
- n: loop bound, b: tile size.

Transfers ($m = i + j = j' + n - i' - 1$):

- Load_p, Load_q, Load_c, Store_c.

$$\mathrm{Load}_p = \{m \mid 1-b \le I \le n-1,\ 0 \le m \le n-1,\ J \le m \le J+b-1\}$$
$$\mathrm{Load}_q = \{m \mid 1-b \le J \le n-1,\ J \le 0,\ 0 \le m \le n-1,\ 1 \le n-I-m \le b\}$$
$$\mathrm{Load}_c = \{m \mid 1-b \le J \le 0,\ 0 \le m \le n-1,\ 2 \le n-I-m \le b\}$$
$$\cup \{m \mid 1-b \le I \le -1,\ n \le m \le 2n-2,\ n+J-1 \le m \le n+J+b-2\}$$
$$\cup \{m \mid 0 \le I \le n-1,\ \max(0,J) \le m-(n-I-1) \le \min(J+b-1, n-1)\}$$

Parameterization is (unexpectedly) feasible!



Tiling:

- $(i, j) \mapsto (i', j') = (n - j - 1, i)$

Parameters:

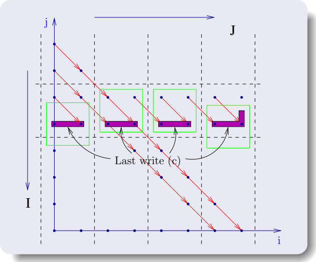
- (I,J): first index in tile.
- n: loop bound, b: tile size.

Transfers $(m = i + j = j' + n - i' - 1)$:

- Load_p, Load_q, Load_c, Store_c.

$$\mathrm{Load}_p = \{m \mid 1 - b \le I \le n - 1,\ 0 \le m \le n - 1,\ J \le m \le J + b - 1\}$$
$$\mathrm{Load}_q = \{m \mid 1 - b \le J \le n - 1,\ J \le 0,\ 0 \le m \le n - 1,\ 1 \le n - I - m \le b\}$$
$$\begin{aligned}\mathrm{Load}_c = &\{m \mid 1 - b \le J \le 0,\ 0 \le m \le n - 1,\ 2 \le n - I - m \le b\} \\ \cup &\{m \mid 1 - b \le I \le -1,\ n \le m \le 2n - 2,\ n + J - 1 \le m \le n + J + b - 2\} \\ \cup &\{m \mid 0 \le I \le n - 1,\ \max(0, J) \le m - (n - I - 1) \le \min(J + b - 1, n - 1)\}\end{aligned}$$

Parameterization is (unexpectedly) feasible!



Tiling:

- $(i, j) \mapsto (i', j') = (n - j - 1, i)$

Parameters:

- (I,J): first index in tile.
- n: loop bound, b: tile size.

Transfers ($m = i + j = j' + n - i' - 1$):

- Load_p, Load_q, Load_c, Store_c.

$$\mathrm{Load}_p = \{m \mid 1 - b \le I \le n - 1,\ 0 \le m \le n - 1,\ J \le m \le J + b - 1\}$$
$$\mathrm{Load}_q = \{m \mid 1 - b \le J \le n - 1,\ J \le 0,\ 0 \le m \le n - 1,\ 1 \le n - I - m \le b\}$$
$$\begin{aligned}\mathrm{Load}_c = &\{m \mid 1 - b \le J \le 0,\ 0 \le m \le n - 1,\ 2 \le n - I - m \le b\}\\ \cup &\{m \mid 1 - b \le I \le -1,\ n \le m \le 2n - 2,\ n + J - 1 \le m \le n + J + b - 2\}\\ \cup &\{m \mid 0 \le I \le n - 1,\ \max(0, J) \le m - (n - I - 1) \le \min(J + b - 1, n - 1)\}\end{aligned}$$
$$\begin{aligned}\mathrm{Store}_c = &\{m \mid I \le n - 1,\ J \le n - b - 1,\ 0 \le m,\ n - I + J \le m \le J + b - 1\}\\ \cup &\{m \mid I \le n - 1,\ n - b \le J,\ 0 \le m \le 2n - 2,\ n - I + J \le m \le 2n - I - 2\}\\ \cup &\{m \mid 1 - b \le I,\ J \le n - 1,\ 0 \le m \le 2n - 2,\ J \le m,\ n - I - b \le m,\\ &\quad n - I + J - b \le m \le n - I + J - 1\}\end{aligned}$$

Size of local buffers, with "double-buffering" execution

ISL/OMEGA-like input (with $b > 0$ and $n > 0$)

```
Domain := [b,n] -> { [i,j] : 0 <= i,j < n };
Read   := [b,n] -> { [i,j] -> c[m] : m = i+j } * Domain
        + [b,n] -> { [i,j] -> p[m] : m = i } * Domain
        + [b,n] -> { [i,j] -> q[m] : m = j } * Domain;
Write := [b,n] -> { [i,j] -> c[m] : m = i+j } * Domain;
Schedule := [b,n] -> { [i,j] -> [n-j-1,i] } * Domain;
```

Output for memory size

Array p

- size $2b$, if $n \geq 2b+1$: 2 overlapping tiles.
- size n if $n \leq 2b$: less than 2 tiles.

Array q

- size b if $n \geq b$: 1 full tile.
- size n if $n \leq b-1$: 1 incomplete tile.

Array c

- size $(2b-1)+b = 3b-1$ if $n \geq 2b+1$: 2 full overlapping tiles.
- size $(2b-1)+(n-b) = b+n-1$ if $b \leq n \leq 2b$: 1 full, one incomplete
- size $2n-1$ if $n \leq b-1$: only one tile.

☛ Distinguishes incomplete tiles and tiles starting out of domain.

Conclusions

Contributions

- Automate double-buffering with inter-tile reuse, at C level.
- Starting point for using HLS tools as back-end compilers?
- Quite general mechanisms: GPUs, other?

Perspectives

- More approximations & parameters in polyhedral model.
- More than parallelism, pipelining.
- Synthesis of communicating processes + customized buffers.
- Compilation of streaming languages with multi-dimensional shared buffers (i.e., not FIFOs)?