Optimizing Remote Accesses for Offloaded Kernels

Application to High-Level Synthesis for FPGA

Christophe Alias, **Alain Darte**, Alexandru Plesco

Compsys Team
Laboratoire de l'Informatique du Parallélisme (LIP)
École normale supérieure de Lyon

Workshop on Polyhedral Compilation Techniques (IMPACT'12)
Jan. 23, 2012, Paris, France

Outline

1. Context and motivations (see ASAP'10 paper)
   - HLS tools, interfaces, and communications
   - Optimizing DDR accesses
2. Communicating processes and "double buffering"
3. Kernel off-loading with polyhedral techniques

High-level synthesis (HLS) tools

Many industrial and academic tools

- Spark, Gaut, Ugh, MMalpha, Catapult-C, Pico-Express, Impulse-C, etc.

Quite good at optimizing computation kernel

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

But most designers prefer to ignore HLS tools and code in VHDL.

High-level synthesis (HLS) tools

Many industrial and academic tools

- Spark, Gaut, Ugh, MMalpha, Catapult-C, Pico-Express, Impulse-C, etc.

Quite good at optimizing computation kernel

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

But most designers prefer to ignore HLS tools and code in VHDL.

Still a huge problem for feeding the accelerators with data

- Lack of good interface support ☛ write (expert) VHDL glue.
- Lack of communication opt. ☛ redesign the algorithm.
- Lack of powerful code analyzers ☛ rename or find tricks.

Our goal: use HLS tools as back-end compilers

Focus on accelerators limited by bandwidth

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

Our goal: use HLS tools as back-end compilers

Focus on accelerators limited by bandwidth

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

Apply source-to-source transformations

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

Our goal: use HLS tools as back-end compilers

Focus on accelerators limited by bandwidth

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

Apply source-to-source transformations

- Push 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:

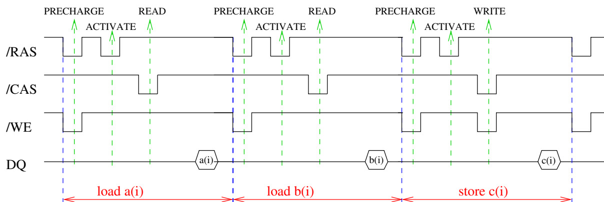
- Syntax-directed translation to hardware.
- Basic DDR-latency-aware software pipelining with internal FIFOs.
- Full interface within the complete system.
- A few compilation pragmas.

Asymmetric DDR accesses: need burst communications

Ex: DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz. Successive reads to the same row every 10 ns, to different rows every 80 ns.

➠ 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];
}
```



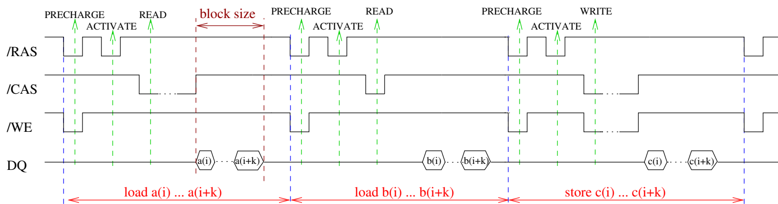
Non-optimized version: time gaps + data thrown away.

Asymmetric DDR accesses: need burst communications

Ex: DDR-400 128Mbx8, size 16MB, CAS 3, 200MHz. Successive reads to the same row every 10 ns, to different rows every 80 ns.

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

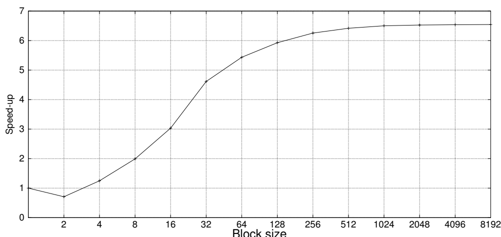
```
void vector_sum (int* __restrict__ a, b, c, int n) {
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}
```



Optimized block version: reduces gaps, exploits burst.

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.

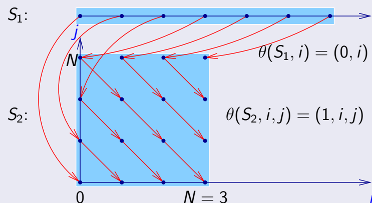
Outline

1. Context and motivations (see ASAP'10 paper)
2. Communicating processes and "double buffering"
   - Loop tiling and the polytope model
   - Overview of the compilation scheme
   - Communication coalescing: related work
3. Kernel off-loading with polyhedral techniques

Polyhedral model in a nutshell

Ex: product of polynomials

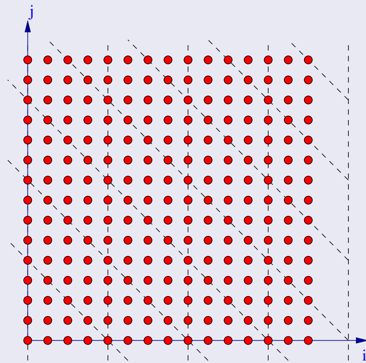
```
  for (i=0; i<= 2*N; i++)
S1: c[i] = 0;

  for (i=0; i<=N; i++)
    for (j=0; j<=N; j++)
S2: c[i+j] = c[i+j] + a[i]*b[j]
```



- Affine (parameterized) loop bounds and accesses
- Iteration domain, iteration vector
- Instance-wise analysis, affine transformations
- PIP: lexico-min in a polytope, given as a Quast (tree, internal node = affine inequality of parameters, leaf = affine function).

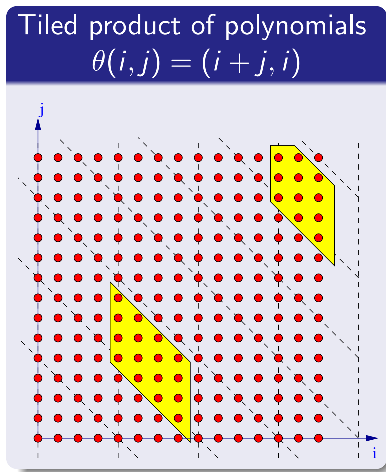
Polyhedral model: tiling

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 $\theta : (i, j) \mapsto (i + j, i)$.

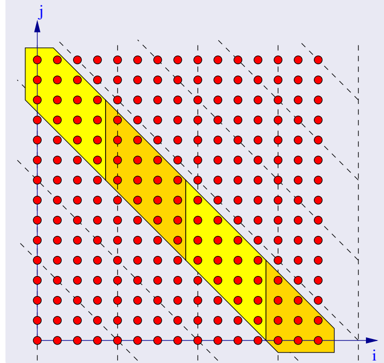
Polyhedral model: tiling



- 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 $\theta(i,j) = (i+j, i)$



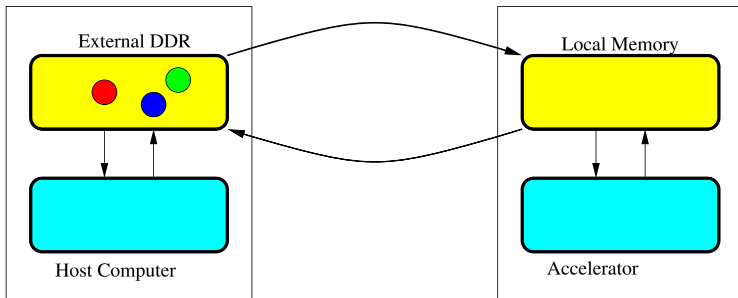
- n loops transformed into n tile loops + n intra-tile loops.
- Expressed from permutable loops: affine function θ, here $\theta : (i,j) \mapsto (i+j, i)$.
- Tile: atomic block operation.
- Increases granularity of computations.
- Enables communication coalescing (hoisting).

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

Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) $\rightarrow$ ● followed by (●, ●) $\rightarrow$ ●.

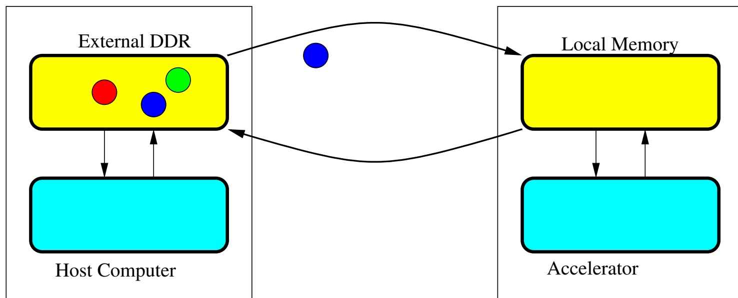
Approach 1: compute all tiles in sequence, with no overlap.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

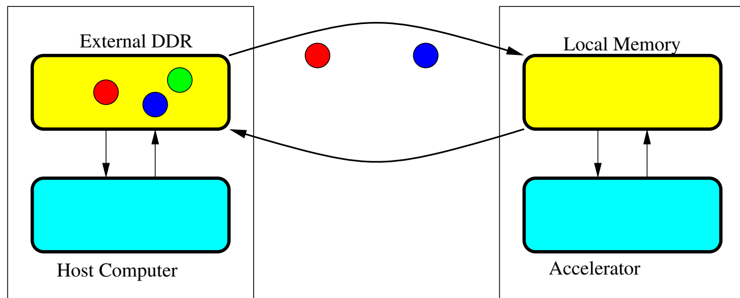
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

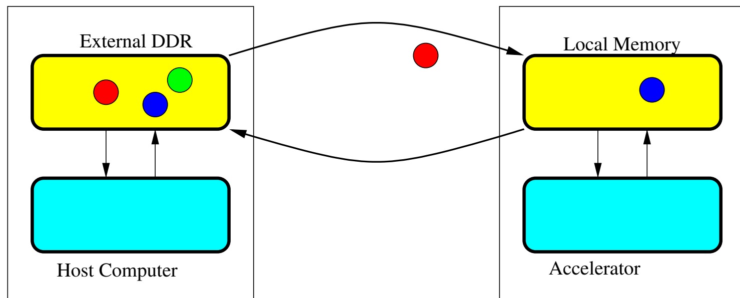
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) $\rightarrow$ ● followed by (●, ●) $\rightarrow$ ●.

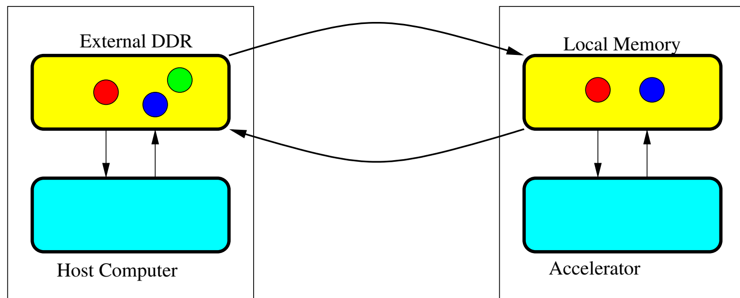
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

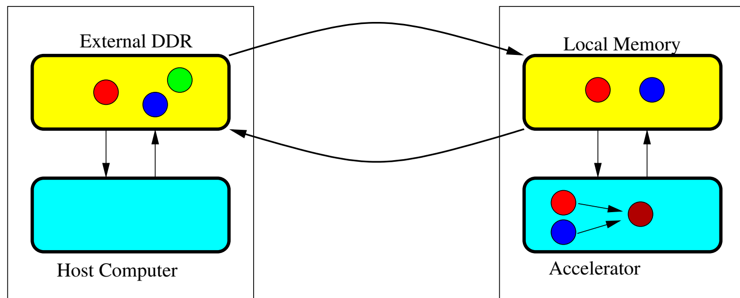
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

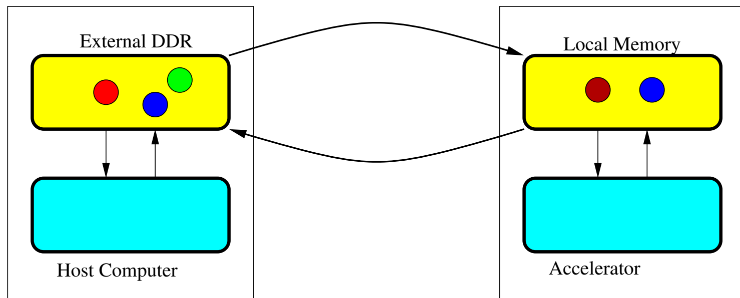
Approach 1: compute all tiles in sequence, with no overlap.
Compute Tile 1 locally.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

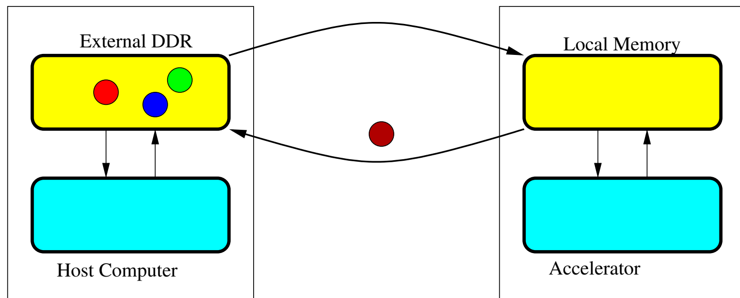
Approach 1: compute all tiles in sequence, with no overlap.
Compute Tile 1 locally.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) $\rightarrow$ ● followed by (●, ●) $\rightarrow$ ●.

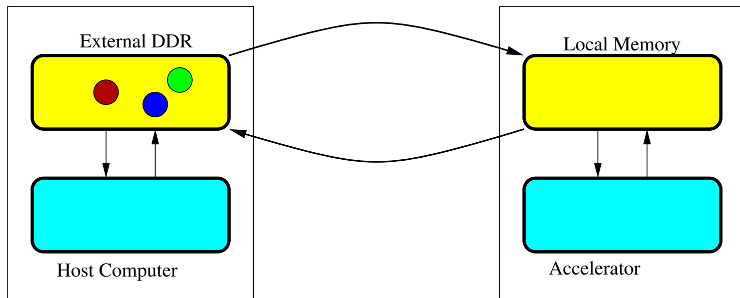
Approach 1: compute all tiles in sequence, with no overlap.
Bring results of Tile 1 to external DDR.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

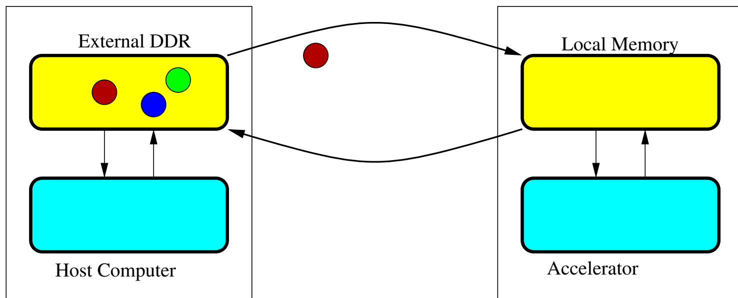
Approach 1: compute all tiles in sequence, with no overlap.
Bring results of Tile 1 to external DDR.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

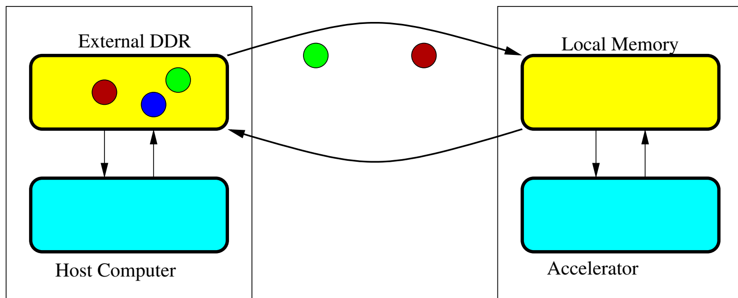
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 2 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

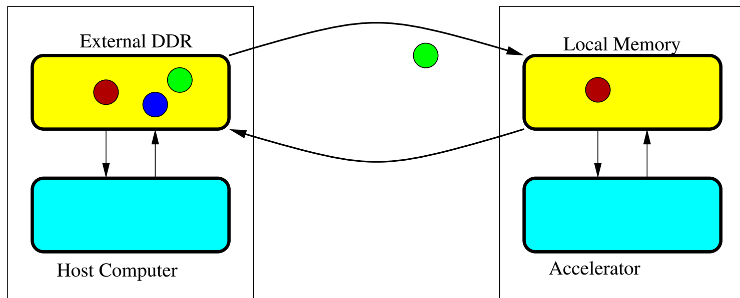
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 2 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

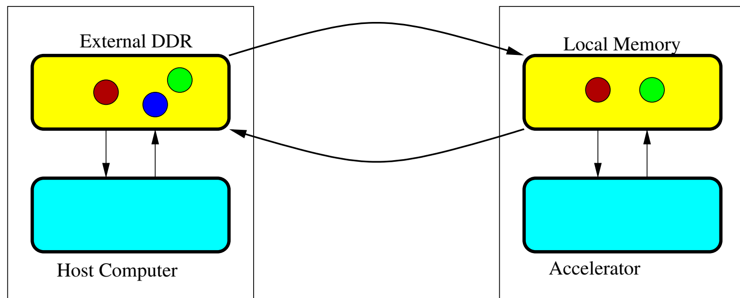
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 2 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

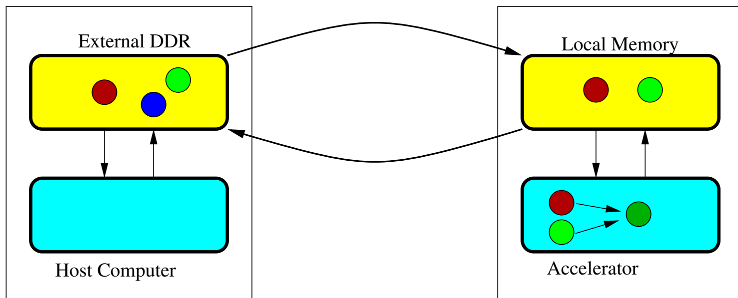
Approach 1: compute all tiles in sequence, with no overlap.
Bring data for Tile 2 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

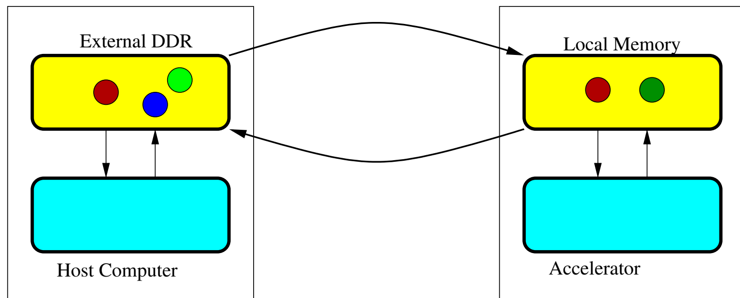
Approach 1: compute all tiles in sequence, with no overlap.
Compute Tile 2 locally.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) $\rightarrow$ ● followed by (●, ●) $\rightarrow$ ●.

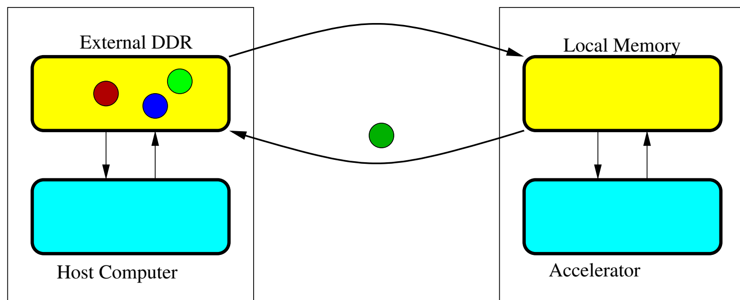
Approach 1: compute all tiles in sequence, with no overlap.
Compute Tile 2 locally.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

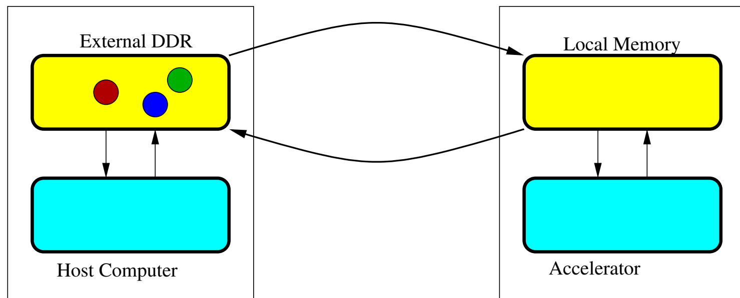
Approach 1: compute all tiles in sequence, with no overlap.
Bring results of Tile 2 to external DDR.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) $\rightarrow$ ● followed by (●, ●) $\rightarrow$ ●.

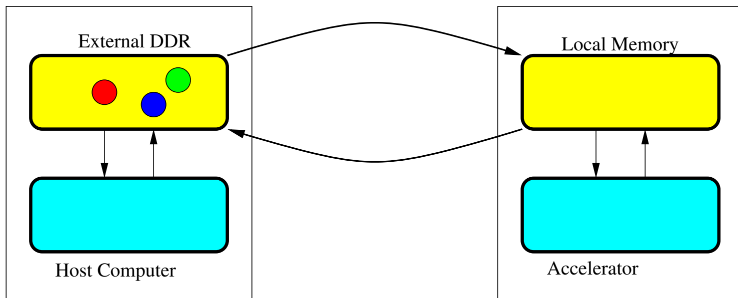
Approach 1: compute all tiles in sequence, with no overlap.
Bring results of Tile 2 to external DDR.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

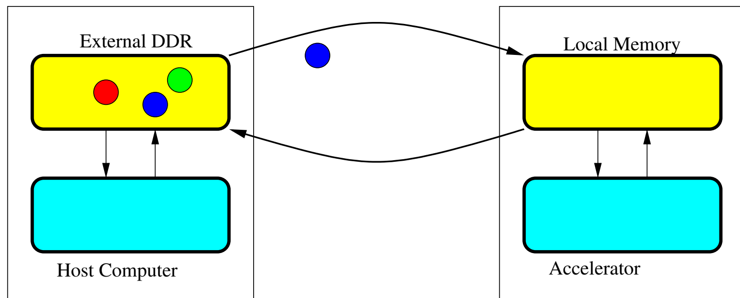
Approach 2: pipeline transfers & computations, no inter-tile reuse.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

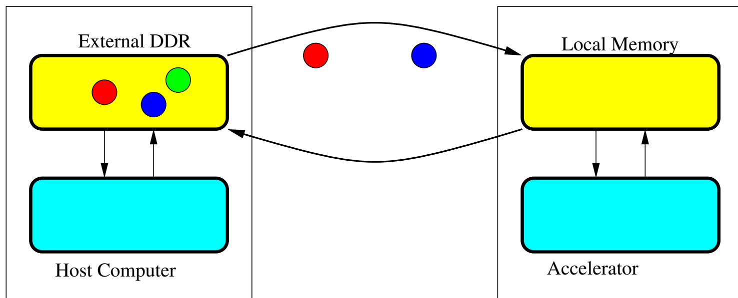
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

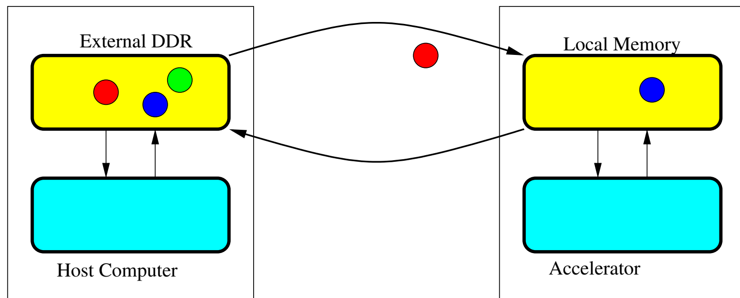
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

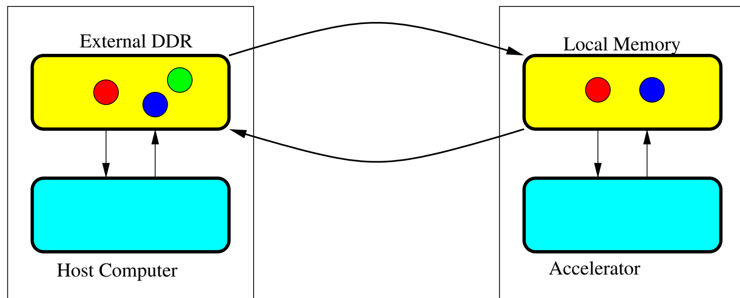
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

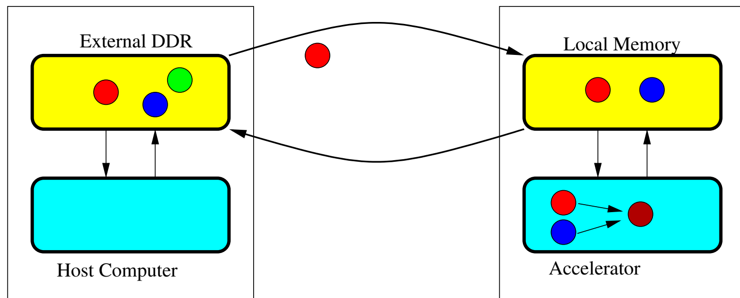
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

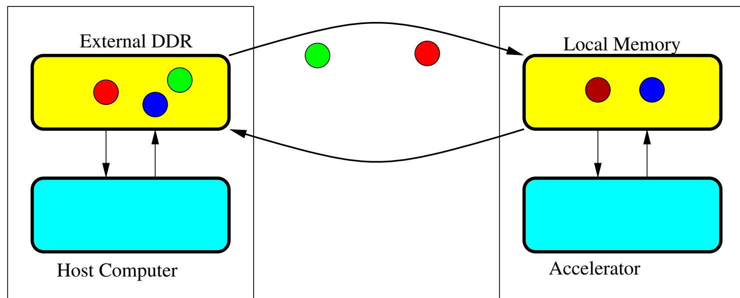
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Compute Tile 1 locally and start data transfer for Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

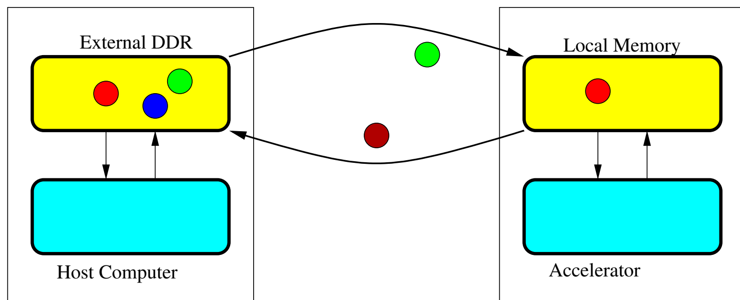
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Compute Tile 1 locally and start data transfer for Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

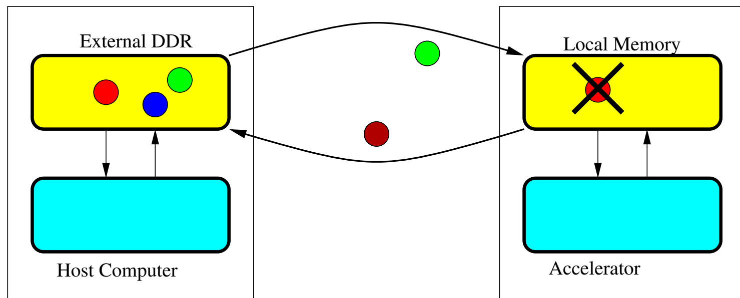
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Bring back results of Tile 1 and receive data for Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

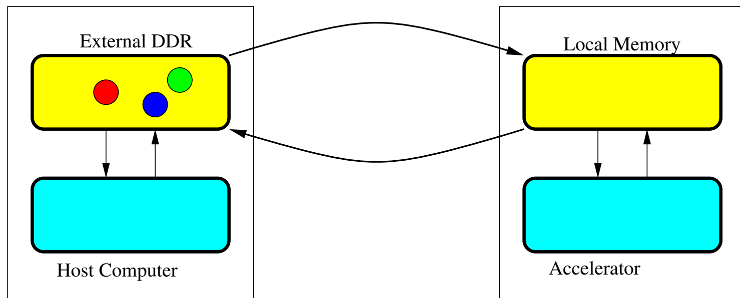
Approach 2: pipeline transfers & computations, no inter-tile reuse.
Wrong for Tile 2: need inter-tile analysis + inter-tile reuse.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

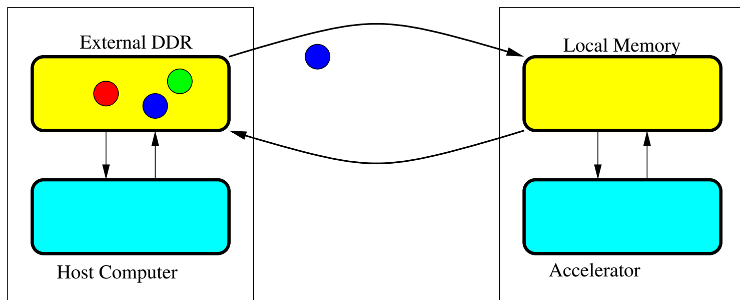
Approach 3: pipeline transfers/computations, use inter-tile reuse.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

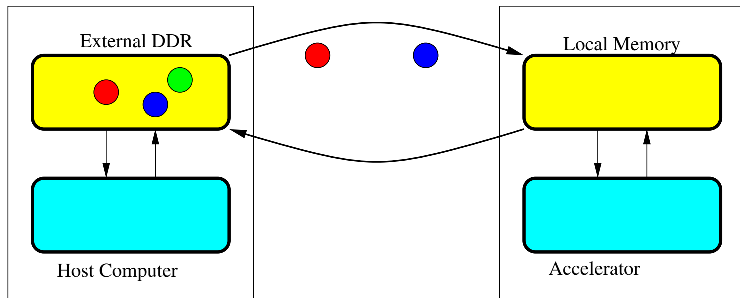
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

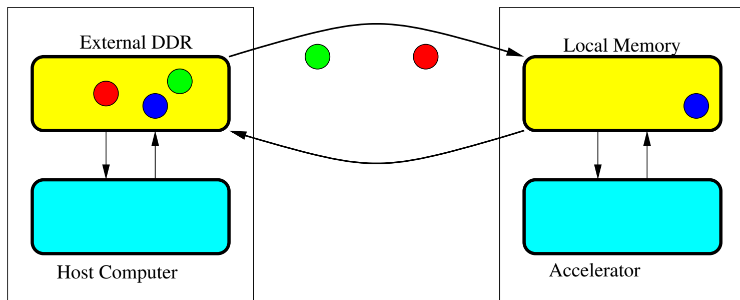
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring data for Tile 1 to local memory.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

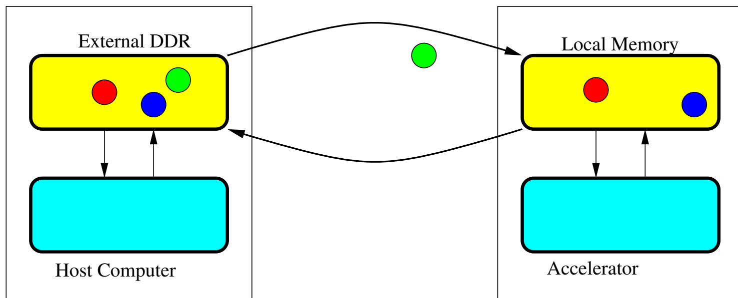
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring data for Tile 1 to local memory, start transfer for Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

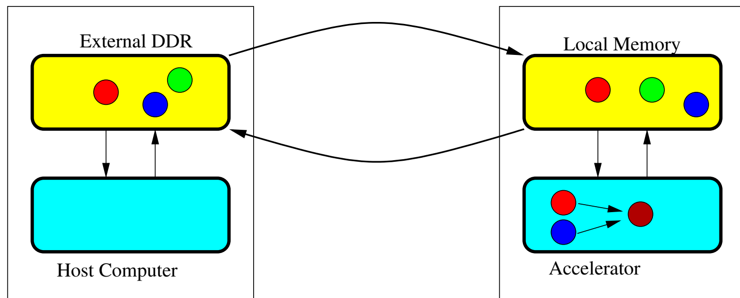
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring data for Tile 1 to local memory, start transfer for Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) $\rightarrow$ ● followed by (●, ●) $\rightarrow$ ●.

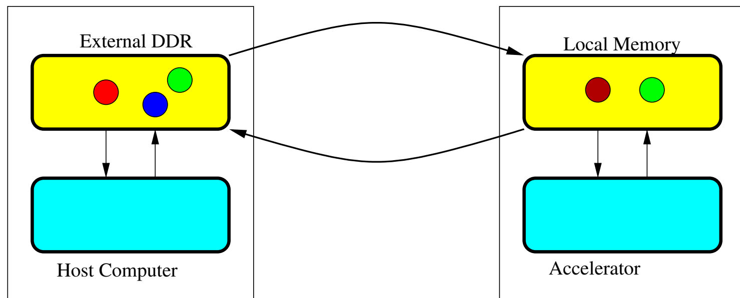
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Compute Tile 1 locally and finish transfer for Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

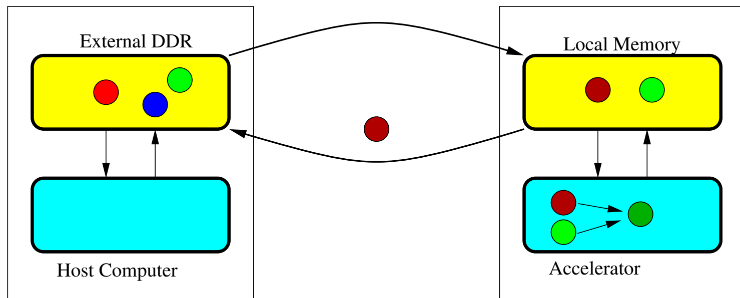
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Finish to compute Tile 1 locally.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

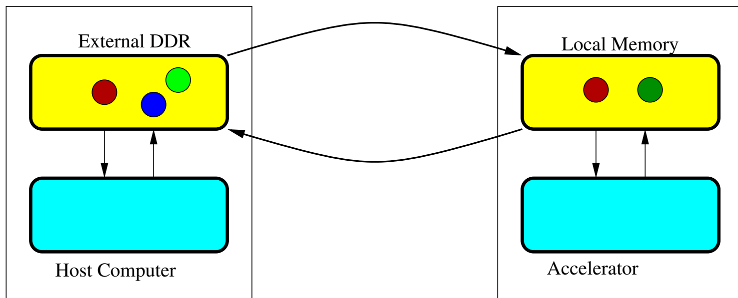
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring back results of Tile 1 and keep data to compute Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

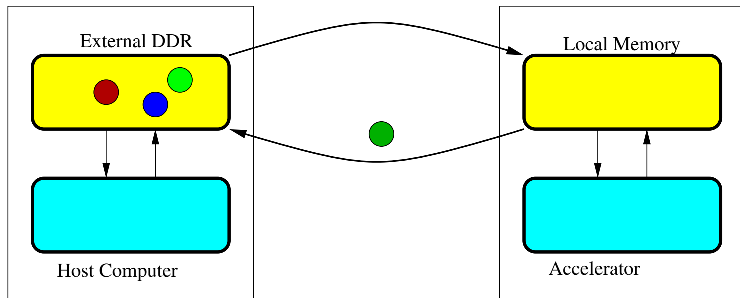
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring back results of Tile 1 and keep data to compute Tile 2.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

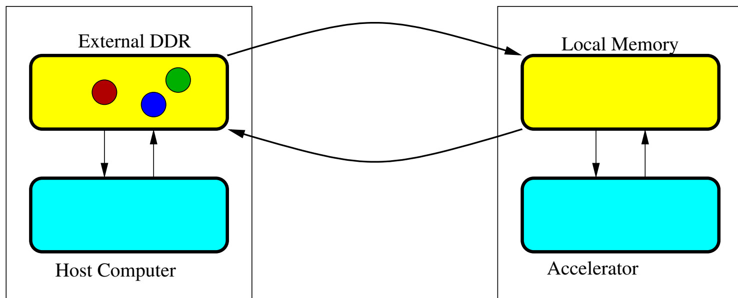
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring results of Tile 2 to external DDR.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

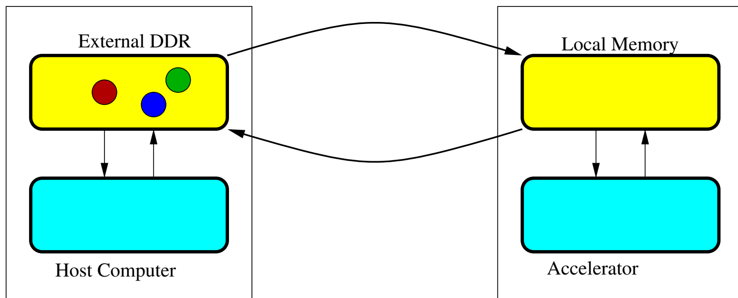
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring results of Tile 2 to external DDR.



Goals and principles: illustrating example

We use tiling to increase spatial locality in the DDR accesses.
Here ● represents all elements of a given array for a given tile.
Example: compute (●, ●) → ● followed by (●, ●) → ●.

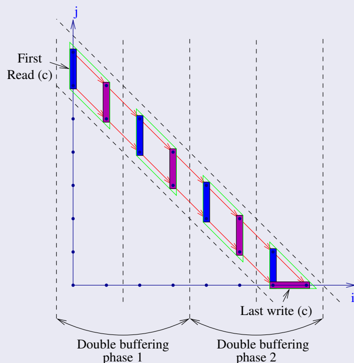
Approach 3: pipeline transfers/computations, use inter-tile reuse.
Bring results of Tile 2 to external DDR.



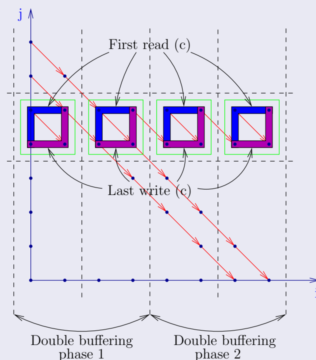
pipelining + data reuse ☛ need for intra & inter-tile analysis + tile scheduling (software pipelining) + local memory management

Loop tiling: impact on reuse and communication

Version 1



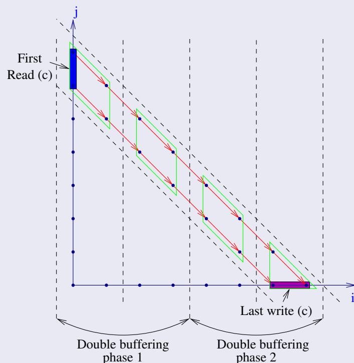
Version 2



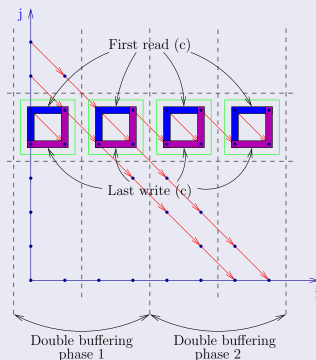
Load $\simeq$ first reads $\cap$ tile domain **Store** $\simeq$ last writes $\cap$ tile domain.

Loop tiling: impact on reuse and communication

Version 1



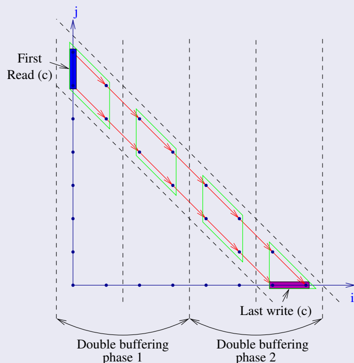
Version 2



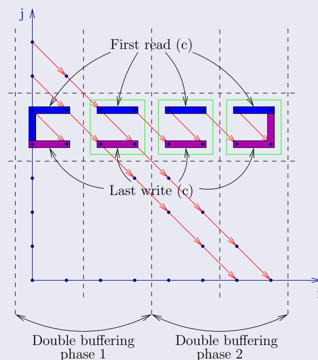
Load $\simeq$ first reads $\cap$ tile domain **Store** $\simeq$ last writes $\cap$ tile domain.

Loop tiling: impact on reuse and communication

Version 1



Version 2



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

Optimized transfers with maximal intra & inter-tile reuse

Double buffering style for optimized communications.

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

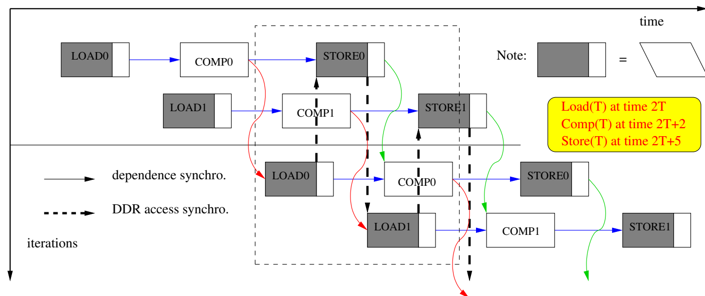
Local memory management defines local buffers with reuse.

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

Code generation generates final C code in a linearized form

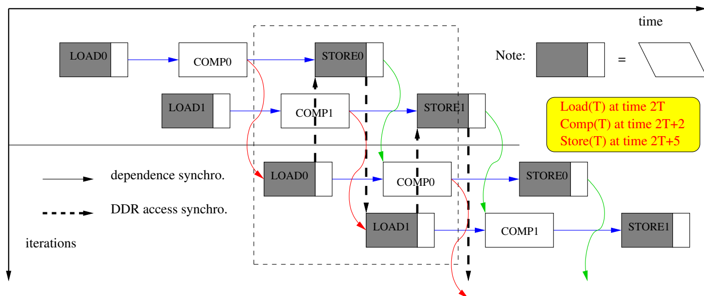
- Placement of FIFO synchronizations.
- Boulet-Feautrier's method for polytope scanning.

Organization of communication & computation processes

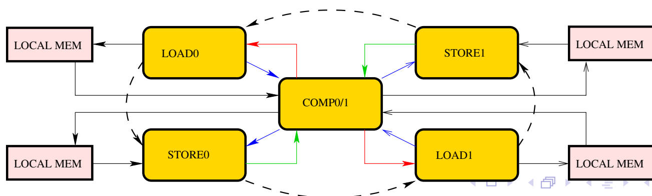


- One function for each communicating process, one memory for each array.
- Dedicated FIFOs of size 1 for synchronizations.
- Transfers through explicit memory accesses.

Organization of communication & computation processes



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



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.

Main principles

```
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
```

Communication coalescing

- Hoist communications out of loops.
- Coalesce out of a tile or out of a tile strip.

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).

Main principles

```
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
```

Communication coalescing

- Hoist communications out of loops.
- Coalesce out of a tile or out of a tile strip.

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).

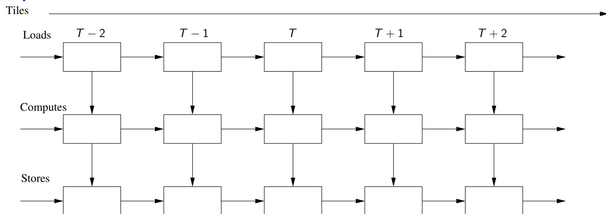
➽ But hypotheses and how "writes" are handled not clear.

Outline

1. Context and motivations (see ASAP'10 paper)
2. Communicating processes and "double buffering"
3. Kernel off-loading with polyhedral techniques
   - Optimizing reuse of remote accesses
   - Algorithmic solution based on parametric linear programming
   - Illustrating example

What do we put in Load(T) and Store(T)?

Minimal dependence structure:



Goal: make computations as local as possible.

- 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.

Two important consequences:

- Live-ranges can be all different: bounding box not enough.
- External memory not up-to-date: over-loading unsafe.

General specification

Define

- $\mathrm{Load}(T)$: data loaded from DDR just before executing tile T.
- $\mathrm{Store}(T)$: data stored to DDR just after T.

- $\mathrm{In}(T)$: 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 $\overline{\mathrm{Out}}(T)$.
- $\underline{\mathrm{Out}}(T)$: data provably written, under-approximation of $\underline{\mathrm{Out}}(T)$.

Can we give conditions for $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$ to be valid?
How to compute then? Can they be over-approximated too?

Extreme solutions

- For all T, $\mathrm{Load}(T) = \mathrm{In}(T)$, $\mathrm{Store}(T) = \mathrm{Out}(T)$ ➽ no inter-tile reuse.
- All $\mathrm{Load}(T)$ empty except first one ➽ no pipelining and overlapping.

Formalization of valid, exact, and approximated load

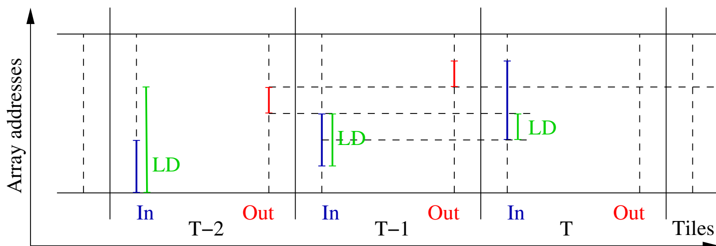
Valid load

(i) Load at least what is needed but not previously produced:

$$\mathrm{In}(T) \setminus \mathrm{Out}(t < T) \subseteq \mathrm{Load}(t \leq T)$$

(ii) Do not overwrite locally-defined data:

$$\mathrm{Out}(t < T) \cap \mathrm{Load}(T) = \emptyset$$



Formalization of valid, exact, and approximated load

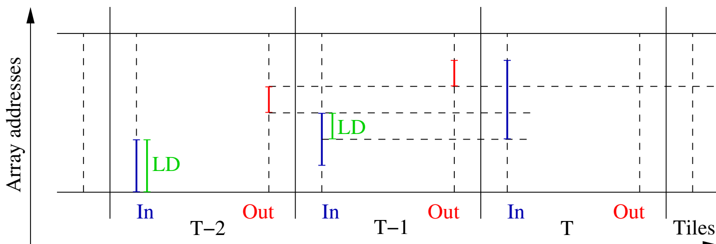
Exact load

(i) Load exactly what is needed but not previously produced:

$$\cup_{t \leq T_{\max}} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\} = \mathrm{Load}(t \leq T_{\max})$$

(ii) All loads should be disjoint (no redundant transfers):

$$\mathrm{Load}(T) \cap \mathrm{Load}(T') = \emptyset, \forall T \neq T'$$



Formalization of valid, exact, and approximated load

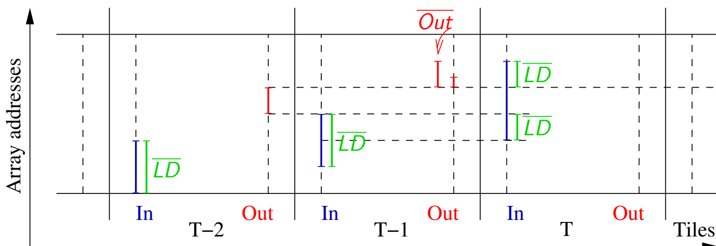
Valid approximated load

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

$$\overline{\mathrm{In}}(T) \setminus \underline{\mathrm{Out}}(t < T) \subseteq \mathrm{Load}(t \leq T)$$

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

$$\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$$



Formalization of valid, exact, and approximated load

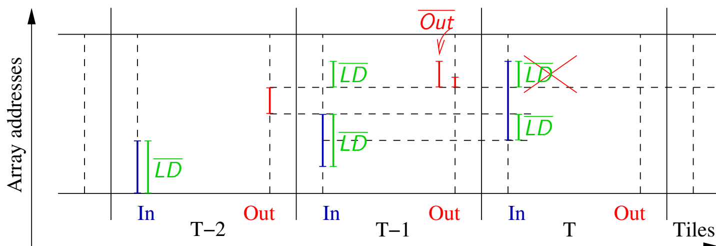
Valid approximated load

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

$$\overline{\mathrm{In}}(T) \setminus \underline{\mathrm{Out}}(t < T) \subseteq \mathrm{Load}(t \leq T)$$

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

$$\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$$



Subtleties due to writes and live-ranges

Main conclusions:

- If a data is locally written, be careful with data over-loading.
- If a data may be locally written, be careful when over-loading and when over-writing back to the DDR.
- Many schemes are possible: to minimize live-ranges, load as late as possible and store back as soon as possible.
- To avoid the problems due to over-loading and over-writing, two solutions:
  - Design an exact scheme.
  - Deal with approximations thanks to pre-loading.
- Live-range splitting (i.e., re-loads) may be useful. This has still to be explored.

Handling approximations of data accesses

Exact situation

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

Handling approximations of data accesses

Exact situation

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

Approximated situation

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

Handling approximations of data accesses

Exact situation

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

Approximated situation NO!

$$\mathrm{Store}(T) = \overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t > T)$$ ☛ may write wrong values in DDR

$$\mathrm{Load}(T) = \overline{\mathrm{In}}(T) \setminus \{\overline{\mathrm{In}}(t < T) \cup \overline{\mathrm{Out}}(t < T)\}$$ ☛ may forget to load from DDR

Handling approximations of data accesses

Exact situation

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

Possible solution with $\overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t > T) \subseteq \mathrm{Store}(T)$

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

Handling approximations of data accesses

Exact situation

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

Possible solution with $\overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t > T) \subseteq \mathrm{Store}(T)$

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

Intuitively, to reduce live-ranges, load ALAP and store 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 x may be written, i.e., $x \in \overline{\mathrm{Out}}(t \le T_{\max}) \setminus \underline{\mathrm{Out}}(t \le T_{\max})$.
- Load a value x always before it may be written, i.e., $x \notin \overline{\mathrm{Out}}(t < T)$.

Quast manipulations, simplifications, and inversions

For each array c, consider an array element $c(\vec{m})$.

- Compute 3 quasts, parameterized by $\vec{m}$ and outer tile indices:
  - $\overline{\mathrm{In}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{In}}(T)\}$ (Note: $= +\infty$ if set empty).
  - $\overline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{Out}}(T)\}$.
  - $\underline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \underline{\mathrm{Out}}(T)\}$.

Quast manipulations, simplifications, and inversions

For each array c, consider an array element $c(\vec{m})$.

- Compute 3 quasts, parameterized by $\vec{m}$ and outer tile indices:
  - $\overline{\mathrm{In}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{In}}(T)\}$ (Note: $= +\infty$ if set empty).
  - $\overline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{Out}}(T)\}$.
  - $\underline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \underline{\mathrm{Out}}(T)\}$.

- Combine them to get $T(\vec{m}) = \min(\overline{\mathrm{Out}}(\vec{m}), \min(\underline{\mathrm{Out}}(\vec{m}), \overline{\mathrm{In}}(\vec{m})))$, with just a slight change: If $\min(\underline{\mathrm{Out}}(\vec{m}), \overline{\mathrm{In}}(\vec{m})) = \underline{\mathrm{Out}}(\vec{m})$, replace by the leaf by $-\infty$, i.e., no need to load. Then:

  if $T(\vec{m}) \neq \pm\infty$, load $\vec{m}$ just before tile $T(\vec{m})$.

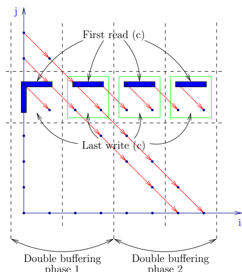
Quast manipulations, simplifications, and inversions

For each array c, consider an array element $c(\vec{m})$.

- Compute 3 quasts, parameterized by $\vec{m}$ and outer tile indices:
  - $\overline{\mathrm{In}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{In}}(T)\}$ (Note: $= +\infty$ if set empty).
  - $\overline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \overline{\mathrm{Out}}(T)\}$.
  - $\underline{\mathrm{Out}}(\vec{m}) = \min\{T \mid \vec{m} \in \underline{\mathrm{Out}}(T)\}$.
- Combine them to get $T(\vec{m}) = \min(\overline{\mathrm{Out}}(\vec{m}), \min(\underline{\mathrm{Out}}(\vec{m}), \overline{\mathrm{In}}(\vec{m})))$, with just a slight change: If $\min(\underline{\mathrm{Out}}(\vec{m}), \overline{\mathrm{In}}(\vec{m})) = \underline{\mathrm{Out}}(\vec{m})$, replace by the leaf by $-\infty$, i.e., no need to load. Then:

  if $T(\vec{m}) \neq \pm\infty$, load $\vec{m}$ just before tile $T(\vec{m})$.
- Invert $T(\vec{m})$ into $\vec{m}(T)$ ($\vec{m}$ is now a variable, T a parameter), add the constraints for tile T, this gives $\mathrm{Load}(T)$ as a union of polytopes (or possibly LBLs) parameterized by tile indices.

Back to polynomial example

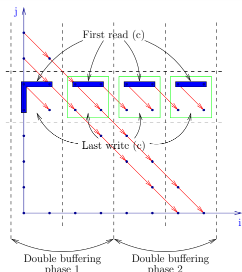


First reads of c (horizontal tiling). System to be solved by PIP:

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

blue=constant (10), red=parameter

Back to polynomial example



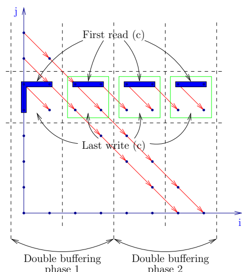
First reads of c (horizontal tiling). System to be solved by PIP:

$$\begin{cases} ii = N - j,\ jj = i,\ i + j = \color{red}{m} \\ 0 \le i \le N,\ 0 \le j \le N \\ \color{blue}{b}\color{red}{I} \le ii \le b(\color{red}{I} + 1) - 1 \\ \color{blue}{b}J \le jj \le \color{blue}{b}(J + 1) - 1 \end{cases}$$

blue=constant (10), red=parameter

if $(-10I + N - m \geq 0)$
  if $(10I - N + m + 9 \geq 0)$ /* vertical band of elements, first tile */
    $(J, ii, jj, i, j) = (0, N - m, 0, 0, m)$
  else $\perp$ /* means undefined */
else
  if $(-10I + 2N - m \geq 0)$
    if $(-10I + N - m + 9 \geq 0)$ /* 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 */

Back to polynomial example



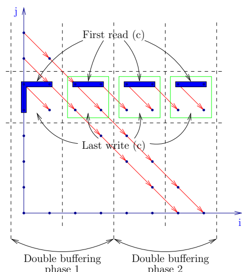
First reads of c (horizontal tiling).
System to be solved by PIP:

$$\begin{cases} ii = N - j,\ jj = i,\ i + j = m \\ 0 \leq i \leq N,\ 0 \leq j \leq N \\ bI \leq ii \leq b(I+1) - 1 \\ bJ \leq jj \leq b(J+1) - 1 \end{cases}$$

blue=constant (10), red=parameter

if $(-10I + N - m \geq 0)$
  if $(10I - N + m + 9 \geq 0)$ /* vertical band of elements, first tile */
    $(i, j) = (0, m)$
  else $\perp$
else
  if $(-10I + 2N - m \geq 0)$
    if $(-10I + N - m + 9 \geq 0)$ /* 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 */

Back to polynomial example



First reads of c (horizontal tiling).
System to be solved by PIP:

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

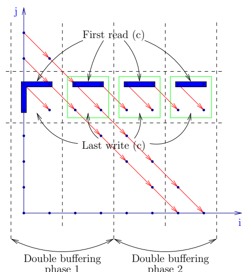
blue=constant (10), red=parameter

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

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

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

Back to polynomial example



First reads of c (horizontal tiling). System to be solved by PIP:

$$\begin{cases} ii = N - j,\ jj = i,\ i + j = m \\ 0 \leq i \leq N,\ 0 \leq j \leq N \\ bI \leq ii \leq b(I+1) - 1 \\ bJ \leq jj \leq b(J+1) - 1 \end{cases}$$

blue=constant (10), red=parameter

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

First operation that accesses m:

$$\begin{aligned} \mathrm{FirstOpRead}(m) = \{(i,j) \mid (i,j) = (0,m),\ \max(0, N-10I-9) \leq m \leq N-10I\} \\ \cup \{(i,j) \mid (i,j) = (10I - N + m, N - 10I),\ N - 10I + 1 \leq m \leq 2N - 10I\} \end{aligned}$$

Introduce tile T and invert to get the data to be loaded at T:

$$\begin{aligned} \mathrm{FirstReadInTile}(T) = \{m \mid \max(0, N-10I-9) \leq m \leq N-10I,\ T = 0\} \\ \cup \{m \mid \max(1, 10T) \leq m + 10I - N \leq \min(N, 10T+9)\} \end{aligned}$$

Conclusion: contributions

- Bring HPC compilation tools to HLS of hardware accelerators.
- To our knowledge, first process to automate communications and integrate FPGA hardware accelerators, entirely at C level.
- Identifies important needs for synchronization mechanisms at source level and for better pragmas (e.g., *restrict* for pairs).
- Quite general analysis and transformations to pipeline kernel off-loading and optimize remote accesses (GPGPUs? Other?).
- Starting point for using HLS tools as back-end compilers.

Conclusion: perspectives

Many many opportunities for improvements.

- Design more efficient Quast simplifications, compare with ISL.
- Extend to parametric tile sizes.
- Implement approximations and live-range splitting.
- Explore link between coarse-grain schedule and memory size.
- Design more domain-specific code generation.
- Define compilation directives at C level for hardware synthesis.
- Include parallelism and multi-process accelerators
- Design customized memories and inter-processes buffers.
- Exploit schedule with slacks for GALS pipelined designs.
- Design a streaming language with shared memory for inter-process communication.
- ...

Conclusion: perspectives

Many many opportunities for improvements.

- Design more efficient Quast simplifications, compare with ISL.
- Extend to parametric tile sizes.
- Implement approximations and live-range splitting.
- Explore link between coarse-grain schedule and memory size.
- Design more domain-specific code generation.
- Define compilation directives at C level for hardware synthesis.
- Include parallelism and multi-process accelerators
- Design customized memories and inter-processes buffers.
- Exploit schedule with slacks for GALS pipelined designs.
- Design a streaming language with shared memory for inter-process communication.
- . . .

Thank you for your attention!