Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators

Alexandru PLESCO

Ph.D. Defense

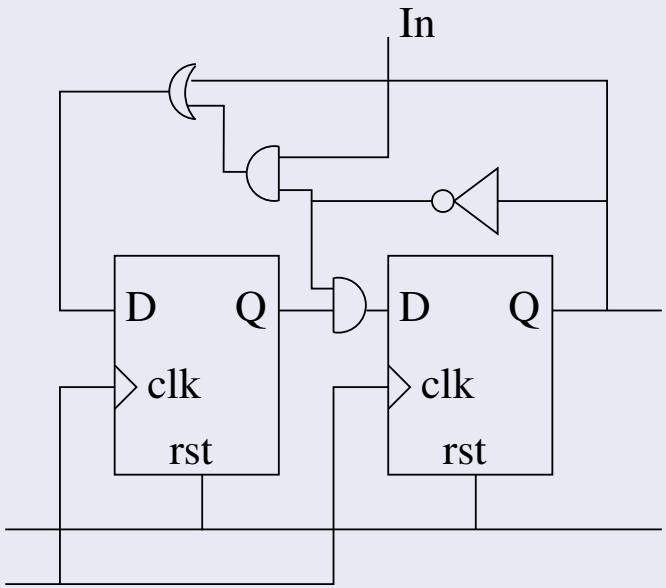
Supervisors: Christophe Alias, Alain Darte, Tanguy Risset
Compsys Team
Laboratoire de l'Informatique du Parallélisme (LIP)
École Normale Supérieure de Lyon
France

September 27, 2010

Digital circuits design methodologies

Description level

- Schematics, RTL
- Structural VHDL
- Behavioral VHDL
- Verilog

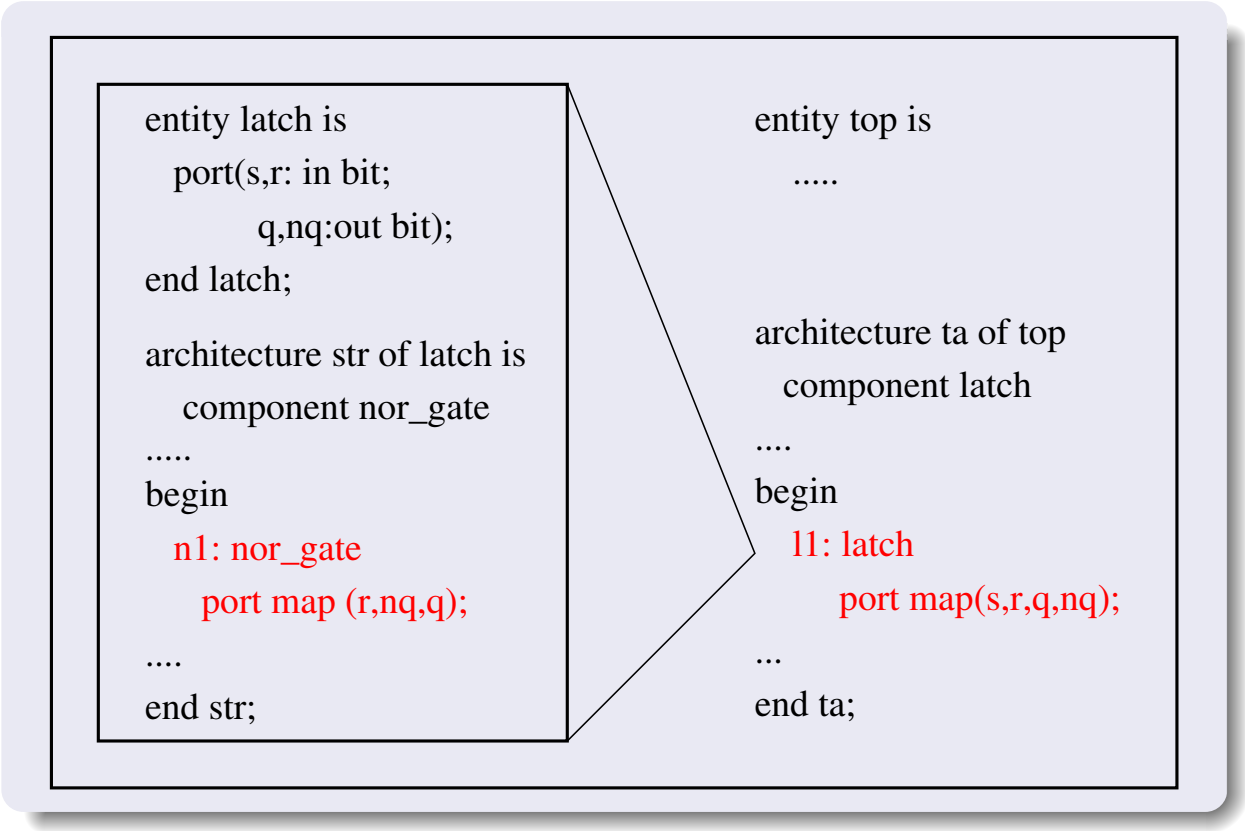


Higher abstraction level $\Rightarrow$ Easier for designer, harder for compiler.

Digital circuits design methodologies

Description level

- Schematics, RTL
- Structural VHDL
- Behavioral VHDL
- Verilog

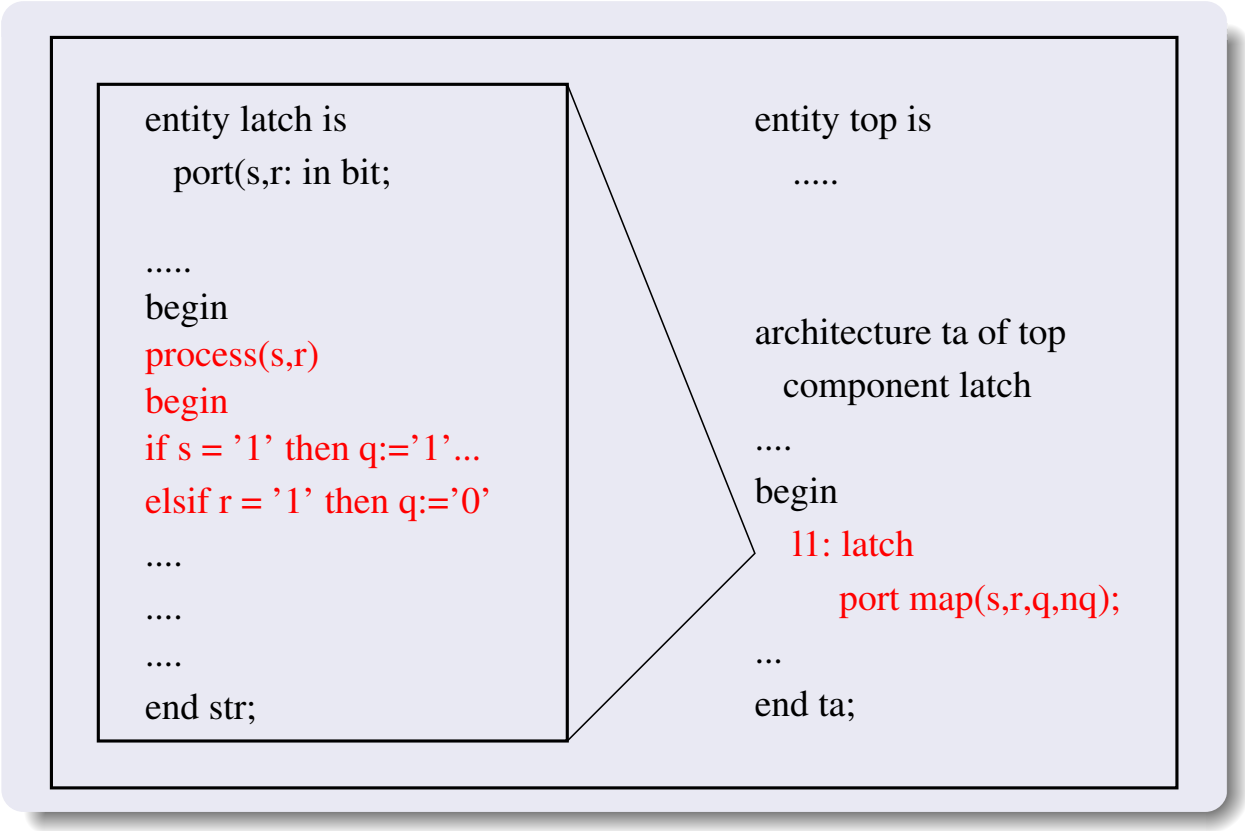


Higher abstraction level $\Rightarrow$ Easier for designer, harder for compiler.

Digital circuits design methodologies

Description level

- Schematics, RTL
- Structural VHDL
- Behavioral VHDL
- Verilog

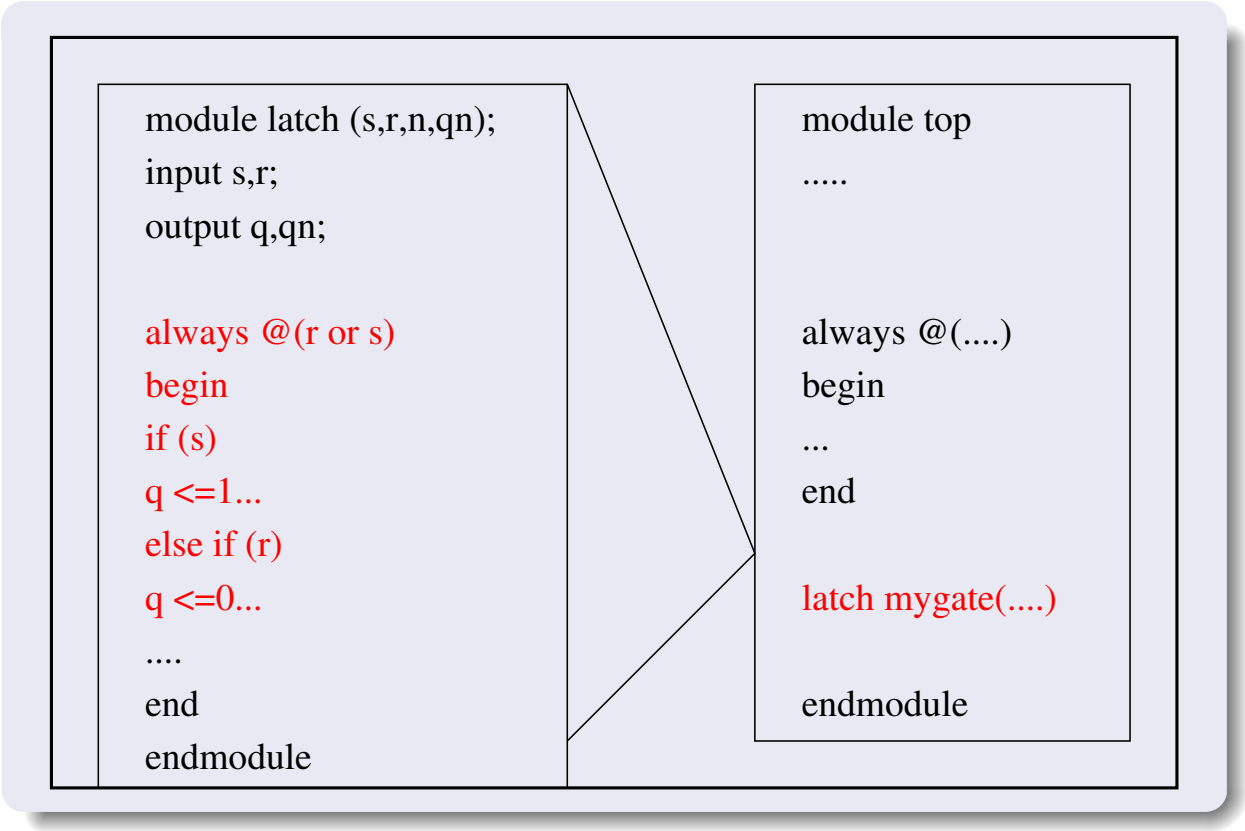


Higher abstraction level $\Rightarrow$ Easier for designer, harder for compiler.

Digital circuits design methodologies

Description level

- Schematics, RTL
- Structural VHDL
- Behavioral VHDL
- Verilog



Higher abstraction level $\Rightarrow$ Easier for designer, harder for compiler.

High-Level Synthesis (HLS)

Increasing complexities of accelerated algorithms:

- Telecommunication equipment (3G, 4G, ...)
- Multimedia devices (Video/audio CODECS, ...)
- High performance computing (encryption, simulation, ...)

➽ Increasing demands in processing power

High-Level Synthesis (HLS)

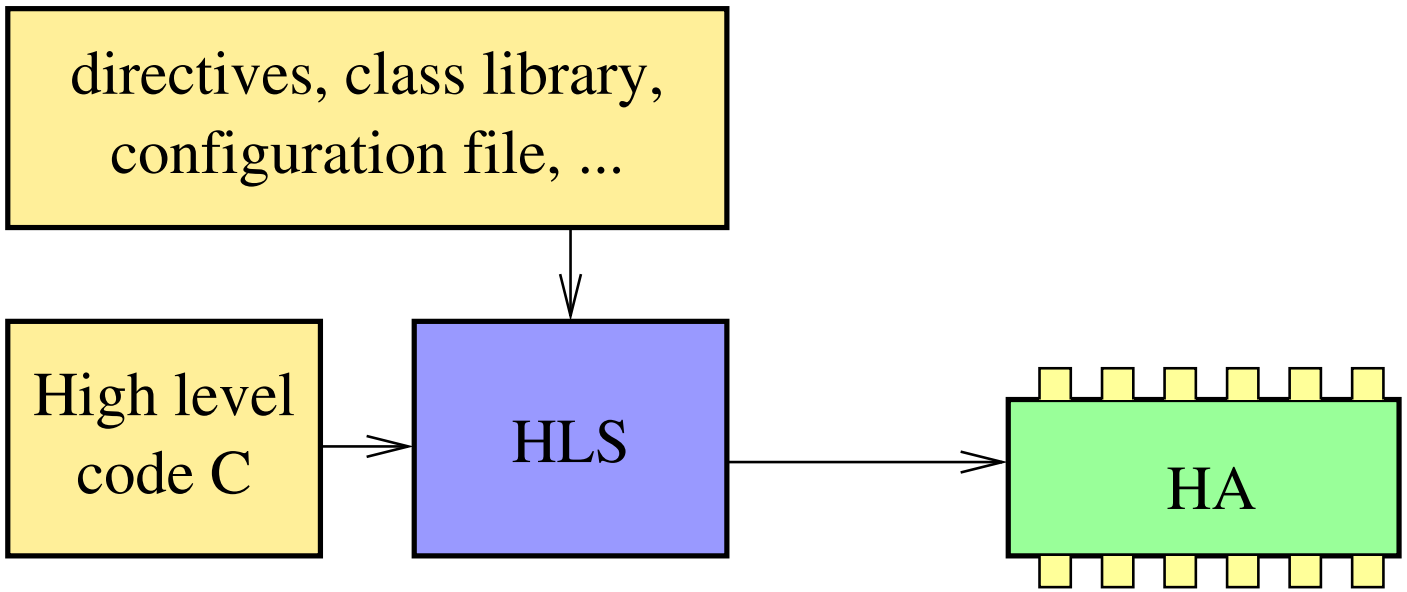
Increasing complexities of accelerated algorithms:

- Telecommunication equipment (3G, 4G, ...)
- Multimedia devices (Video/audio CODECS, ...)
- High performance computing (encryption, simulation, ...)

➽ Increasing demands in processing power
+ Short time to market
+ Development effort

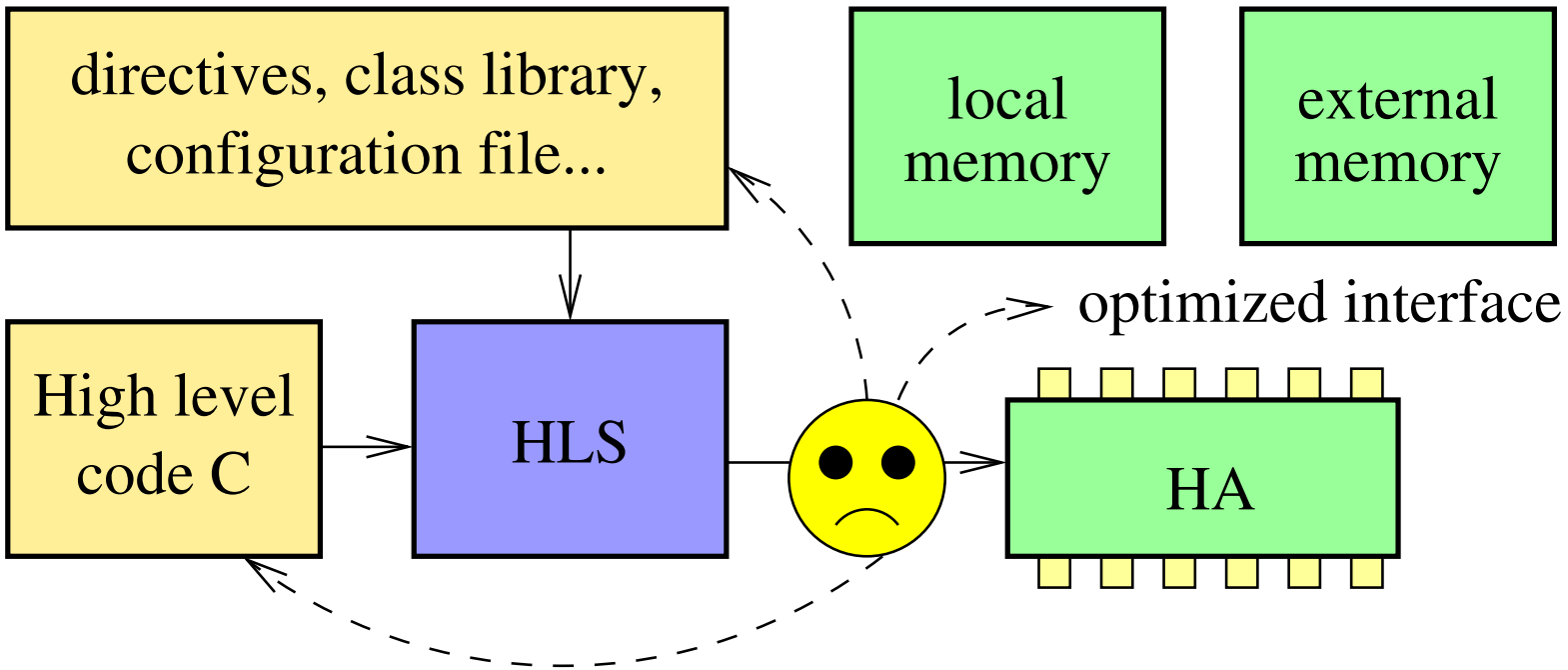
➽ HLS tools become mandatory to meet design constraints

What is HLS?

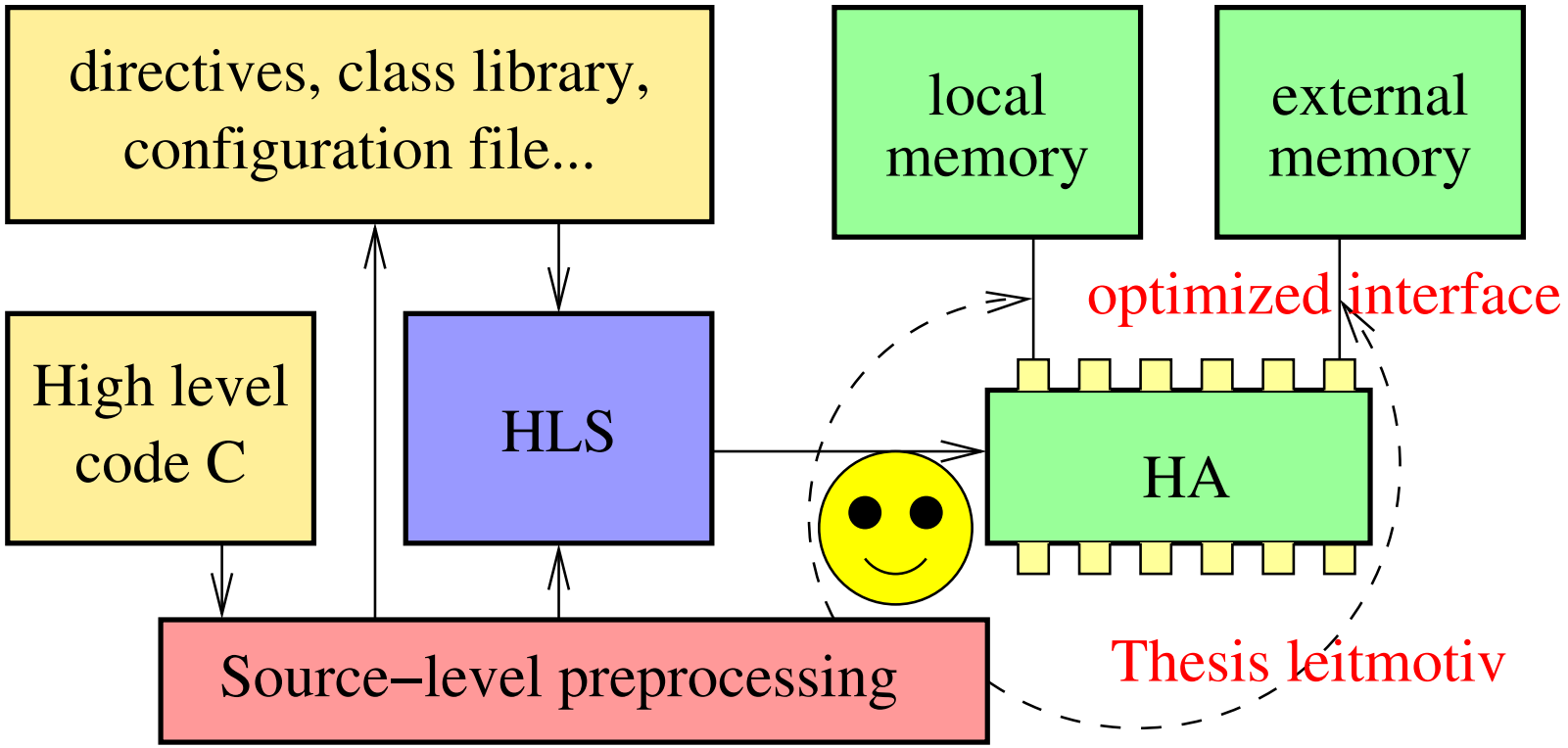


FSM, instruction–level parallelism,
resource sharing, software pipelining, ...

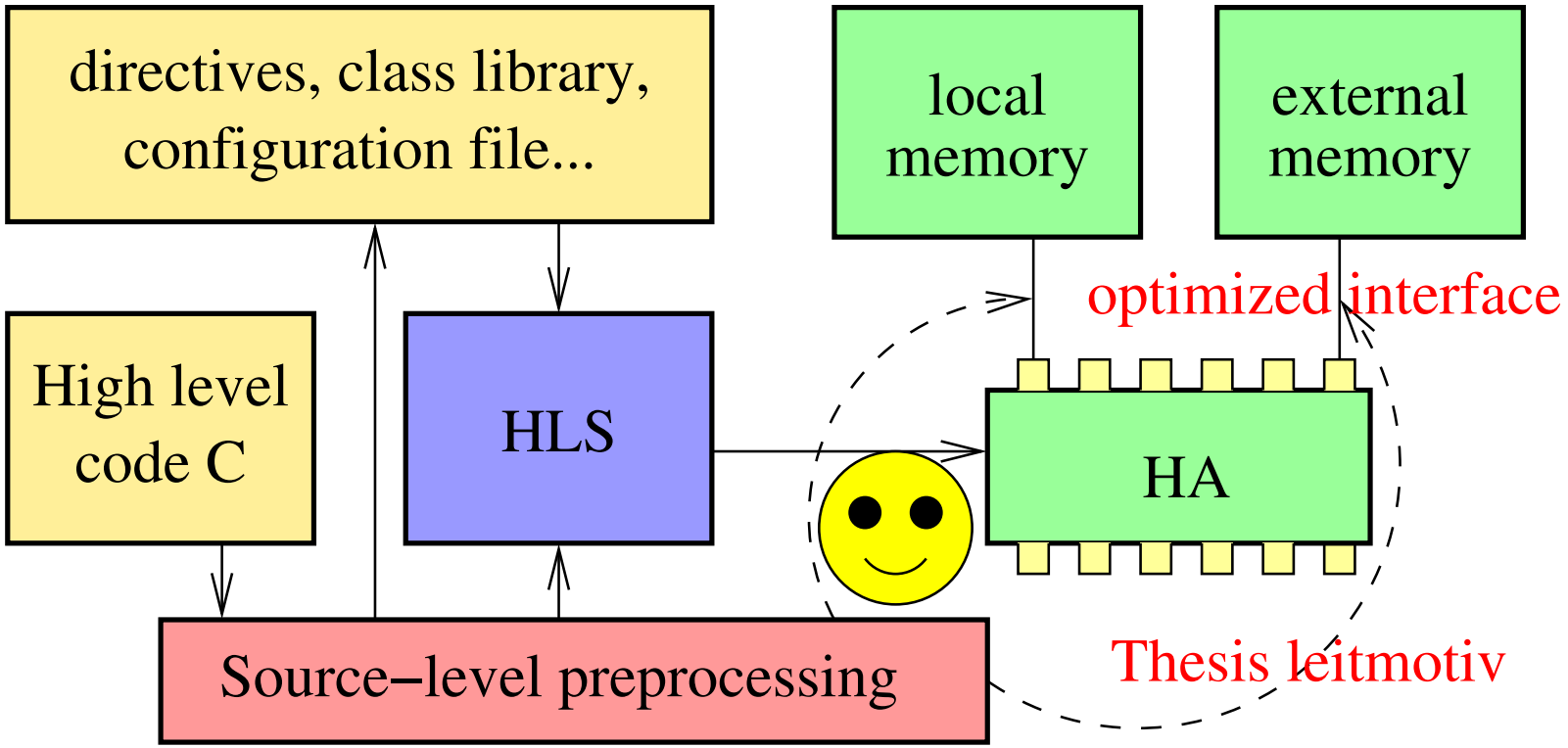
What is HLS?


directives, class library,
configuration file...
local
memory
external
memory
optimized interface
High level
code C
HLS
HA

What is HLS?


directives, class library, configuration file...
local memory
external memory
optimized interface
High level code C
HLS
HA
Source–level preprocessing
Thesis leitmotiv

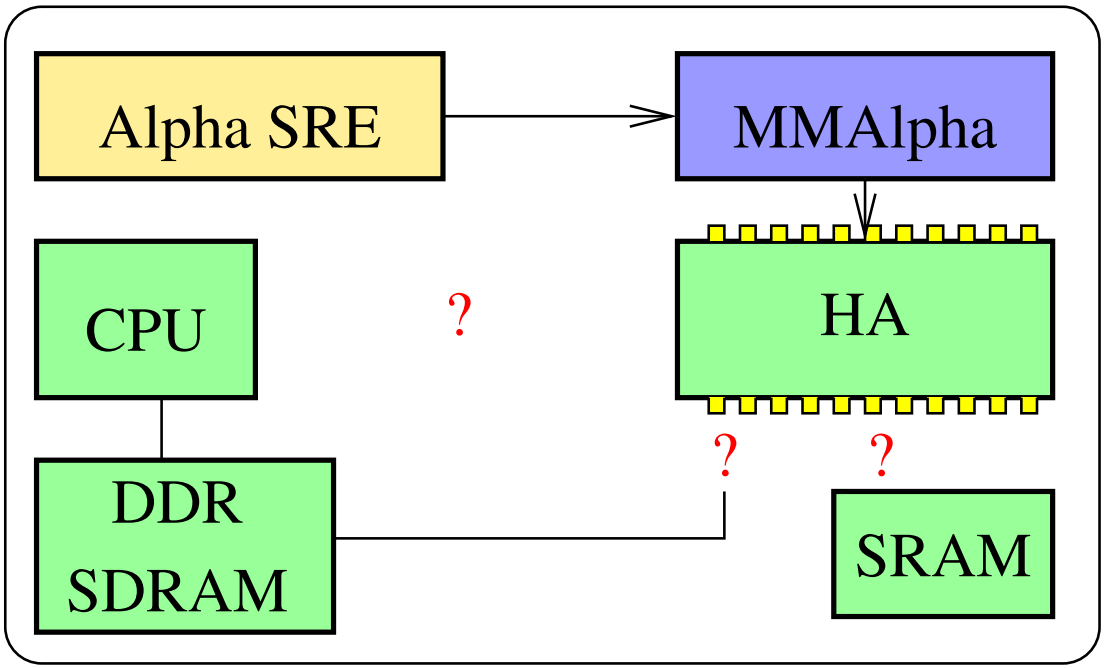
What is HLS?

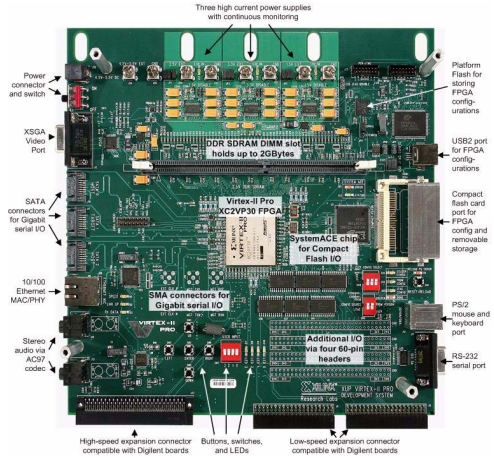


- Optimize memory accesses, data reuse, and interconnections

Outline

Design with MMAlpha HLS tool

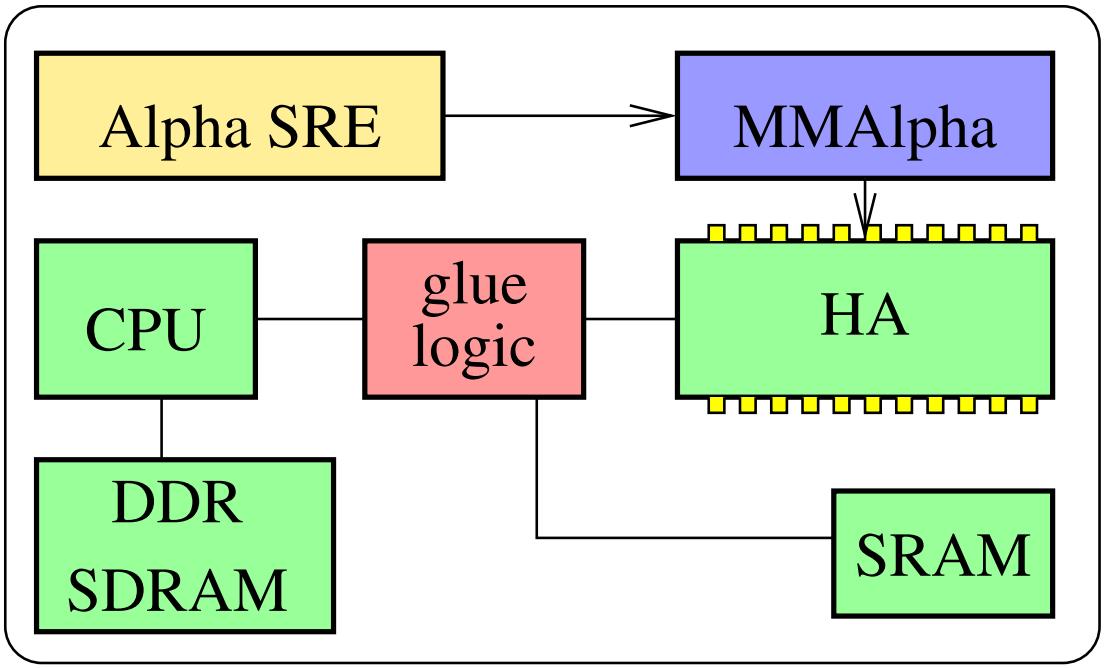




MMAlpha

- Slave HA
- Synchronous interface
- Regular control inside HA (dataflow)

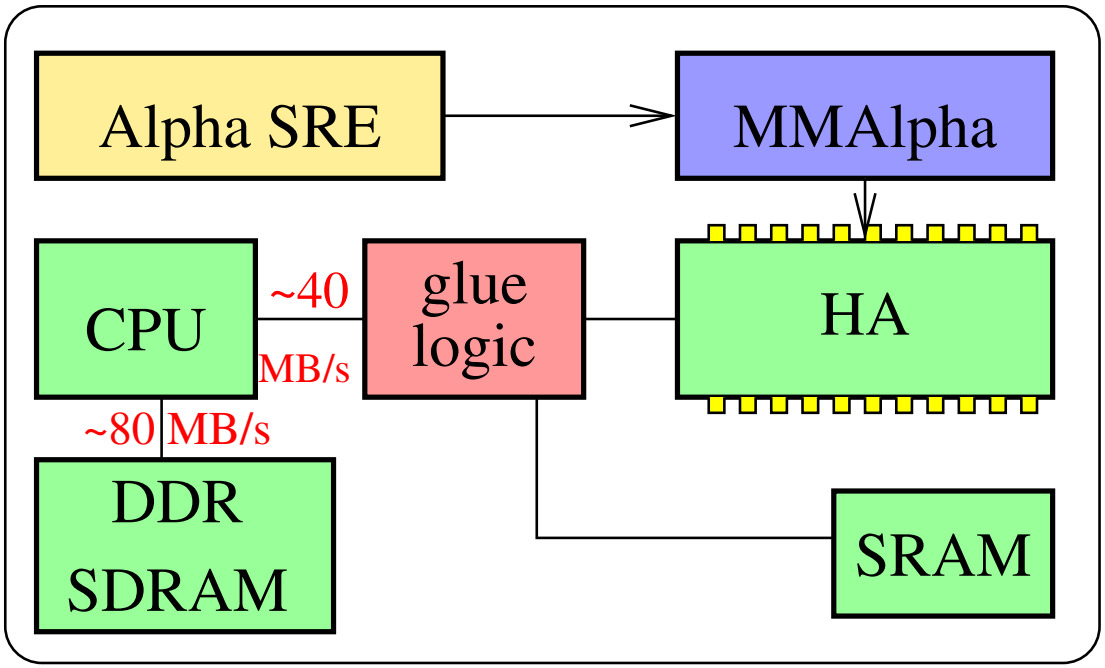
Design with MMAlpha HLS tool



Glue logic: interface + memory manager for

- data reuse
- burst transfers

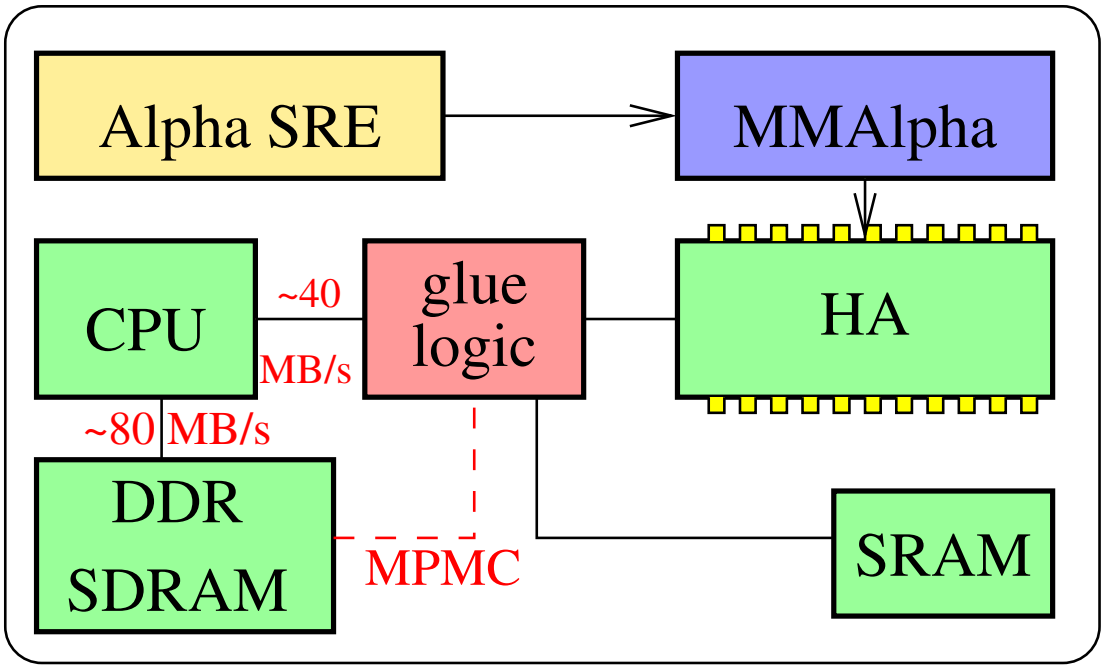
Design with MMAlpha HLS tool



Problems

- Small bandwidth
- Multi-port memory controller?
- Important design and programming effort

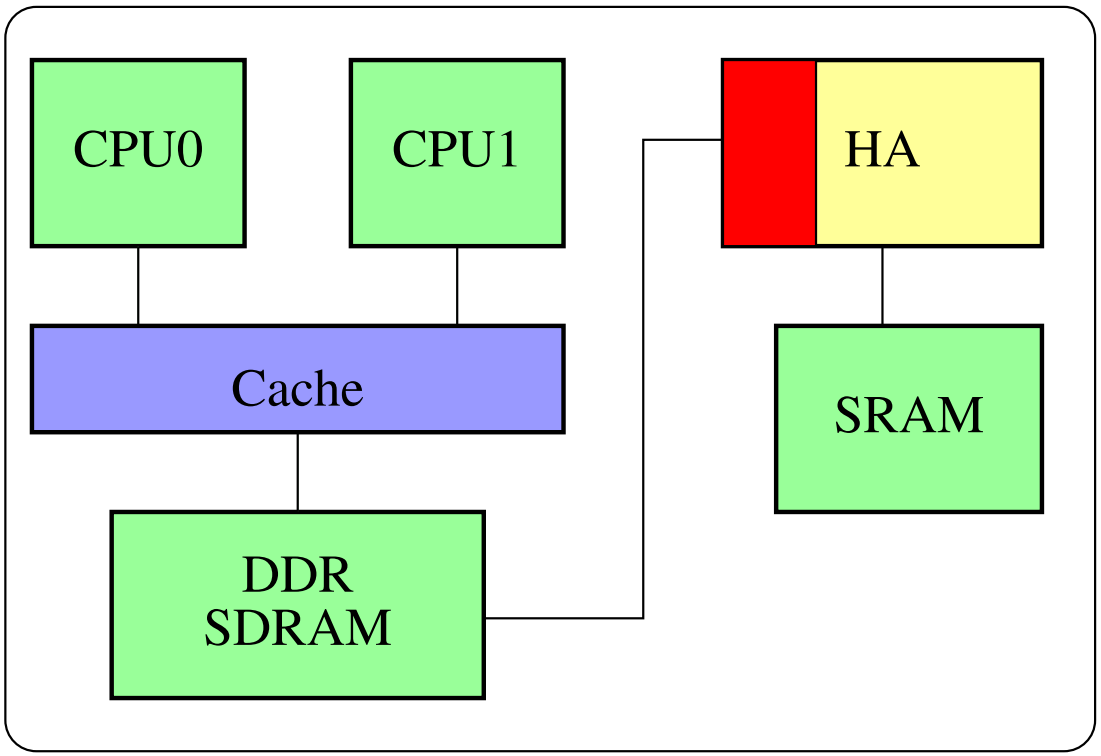
Design with MMAlpha HLS tool



Problems

- Small bandwidth
- Multi-port memory controller?
- Important design and programming effort

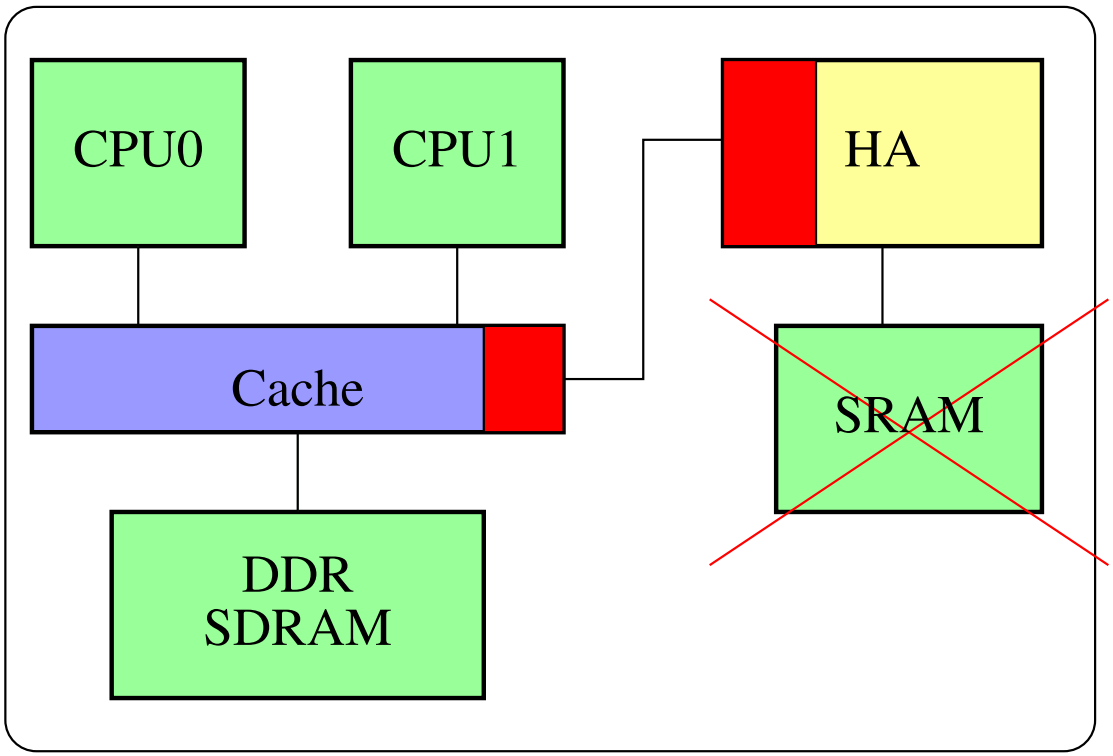
Moving the glue logic inside the HA



Control logic: inside HA.

Which HLS tool?

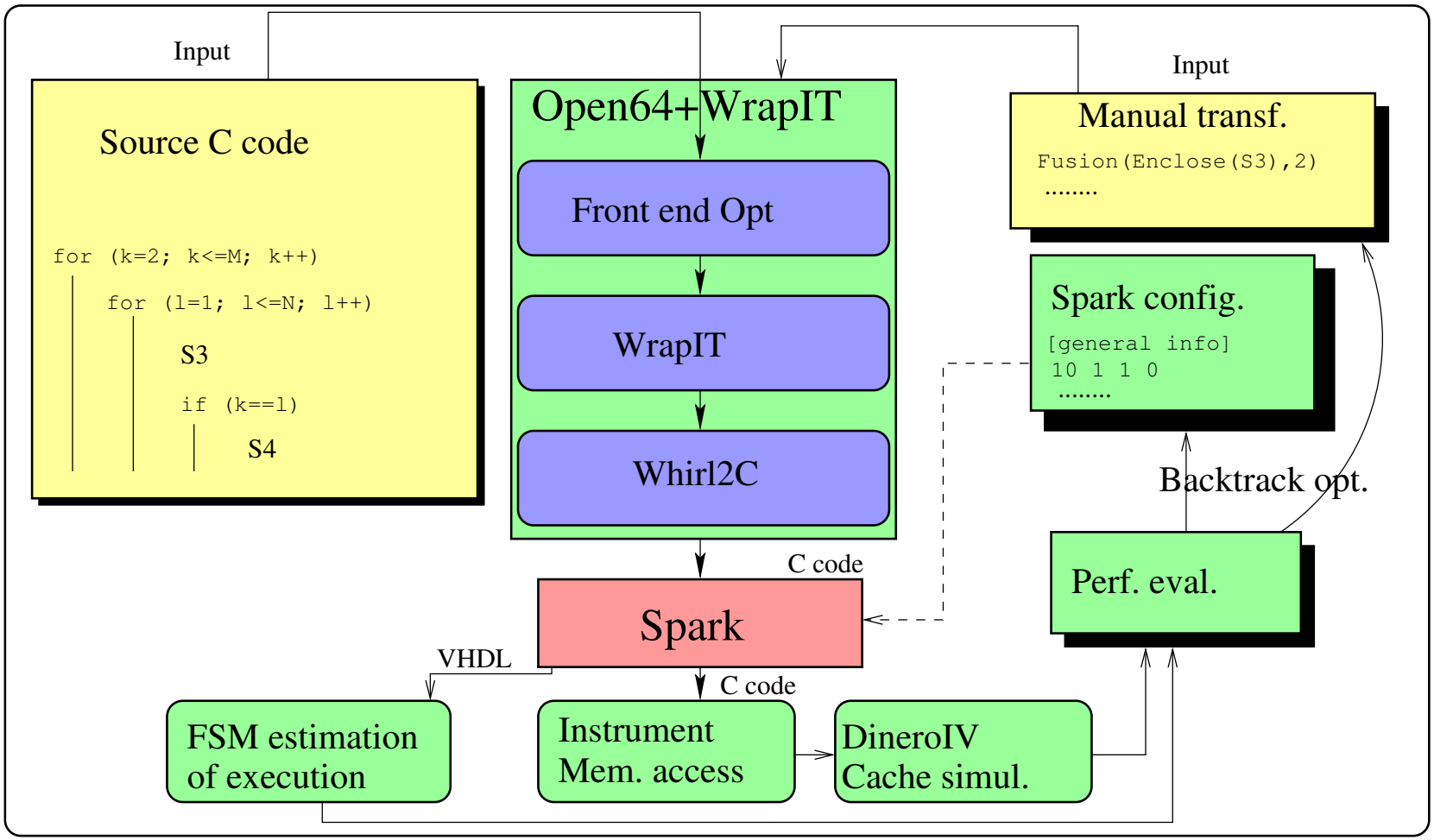
Moving the glue logic inside the HA



Control logic: inside HA.

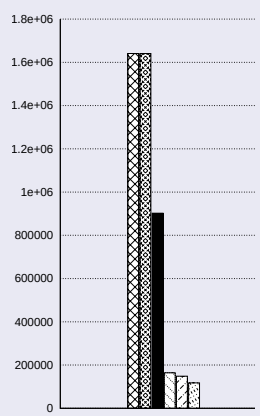
Which HLS tool? Spark

Design with Spark and WRaP-IT

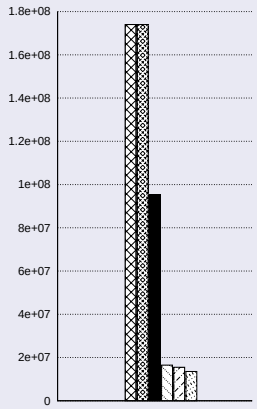


Experimental results

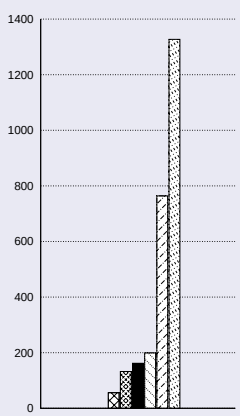
h264/h263 YUV420 to RGB444 converter



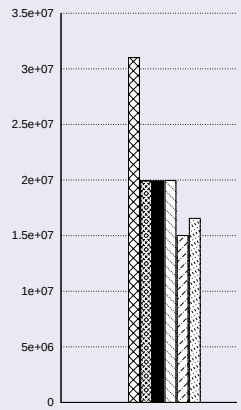
Cache misses



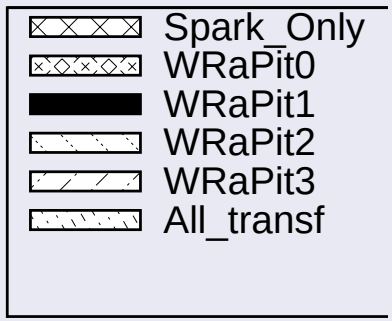
Memory R/W



FSM states



Execution cycles



Manual transformation sequences

- increases the cache hit ratio
- decreases the memory bandwidth requirements
- increases the number of FSM states (may degrade frequency)
- decreases the total execution time

Try to improve HLS tools with high-level transformations

Experiments with MMAlpha

- Parallel (systolic) architecture.
- Data should be available at MMAlpha ports, fed by FIFOs.
- Need to implement (in VHDL) a complex memory controller.

➽ Improved performances but still (very) limited by bandwidth.

Try to improve HLS tools with high-level transformations

Experiments with MMAlpha

- Parallel (systolic) architecture.
- Data should be available at MMAlpha ports, fed by FIFOs.
- Need to implement (in VHDL) a complex memory controller.

➽ Improved performances but still (very) limited by bandwidth.

Experiments with Spark

- Accelerator to be connected to a CPU, possibly with a cache.
- I/O pins: 1 dedicated pin per data bit
- Apply loop transformations: optimize transfers and reuse.

➽ Loop transformations appeared to be useful.

Try to improve HLS tools with high-level transformations

Experiments with MMAlpha

- Parallel (systolic) architecture.
- Data should be available at MMAlpha ports, fed by FIFOs.
- Need to implement (in VHDL) a complex memory controller.

➽ Improved performances but still (very) limited by bandwidth.

Experiments with Spark

- Accelerator to be connected to a CPU, possibly with a cache.
- I/O pins: 1 dedicated pin per data bit
- Apply loop transformations: optimize transfers and reuse.

➽ Loop transformations appeared to be useful. **But:**

- How to interface? No way to reuse pins.
- Thus performances improvements could only be simulated.
- Frequency sometimes lower.

Outline

Goal of this study: use HLS tools as a back-end compiler

- Show that high-level transformations are useful and needed.
- Focus on accelerators limited by bandwidth: optimize throughput and put necessary hardware for computations.
- Optimize transfers at C level.
- Compile any new functions with the same HLS tool.

➽ Try to consider HLS tools the same way back-end compilers are used, in standard compilation, by front-end optimizers.

C2H (Altera HLS tool) main features

Syntax-directed translation to hardware

- Hierarchical FSMs: stalls to wait for the longest inner FSM.
- Access to external memory through arrays and pointers.
- One local memory for each local array.

C2H (Altera HLS tool) main features

Syntax-directed translation to hardware

- Hierarchical FSMs: stalls to wait for the longest inner FSM.
- Access to external memory through arrays and pointers.
- One local memory for each local array.

Software pipelined loops ➽ Optimize CPLI (initiation interval)

- Basic software pipelining with rough data dependence analysis.
- Latency-aware pipelined DDR accesses (with internal FIFOs).

C2H (Altera HLS tool) main features

Syntax-directed translation to hardware

- Hierarchical FSMs: stalls to wait for the longest inner FSM.
- Access to external memory through arrays and pointers.
- One local memory for each local array.

Software pipelined loops ➽ Optimize CPLI (initiation interval)

- Basic software pipelining with rough data dependence analysis.
- Latency-aware pipelined DDR accesses (with internal FIFOs).

Full interface within the complete system

- Accelerator initiated as a (blocking or not) function call.
- Memory mapped connection ports with Avalon interconnect.

C2H (Altera HLS tool) main features

Syntax-directed translation to hardware

- Hierarchical FSMs: stalls to wait for the longest inner FSM.
- Access to external memory through arrays and pointers.
- One local memory for each local array.

Software pipelined loops ➽ Optimize CPLI (initiation interval)

- Basic software pipelining with rough data dependence analysis.
- Latency-aware pipelined DDR accesses (with internal FIFOs).

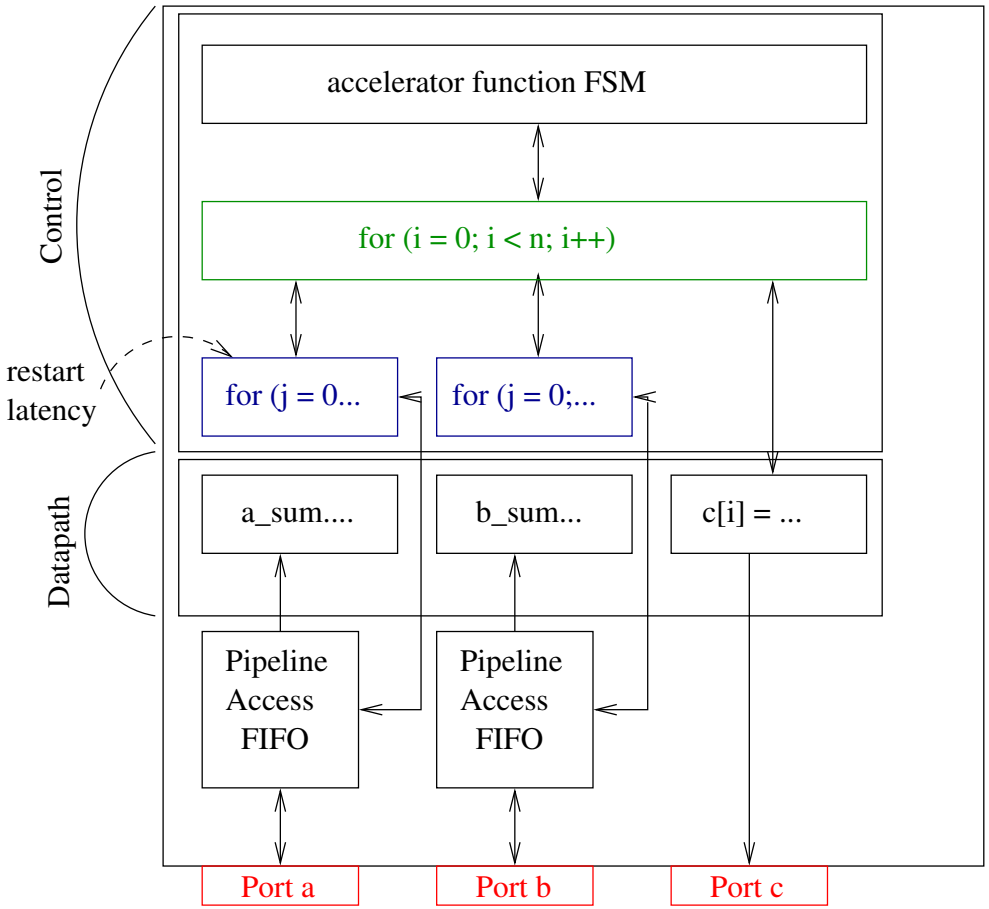
Full interface within the complete system

- Accelerator initiated as a (blocking or not) function call.
- Memory mapped connection ports with Avalon interconnect.

A few compilation pragmas

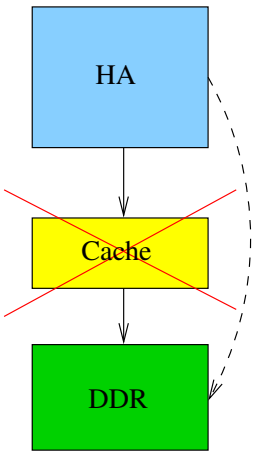
- *restrict*: pointer does not alias with any other pointer.
- *arbitration share*: how many accesses without re-arbitration.

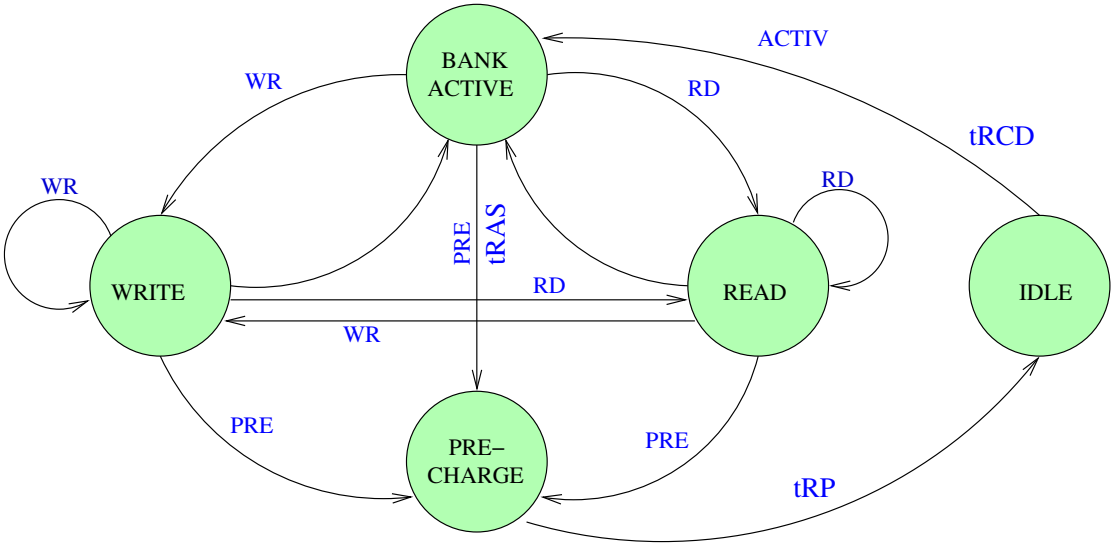
Nested finite state machines and pipelined accesses




```
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;
  }
}
```

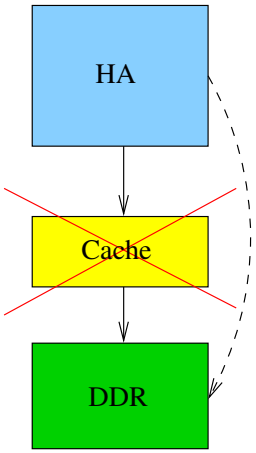
DDR SDRAM asymmetric accesses

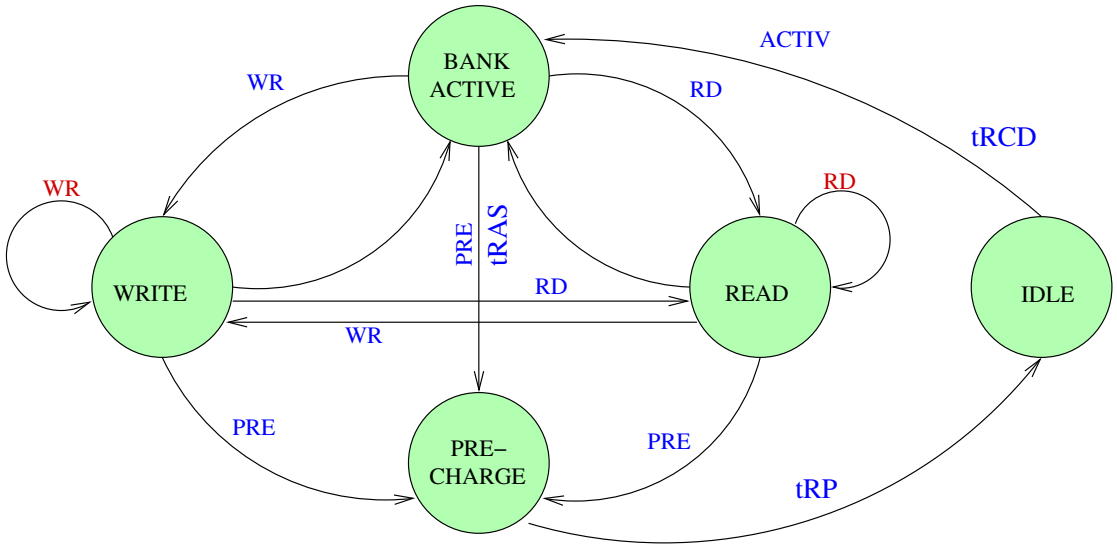




DDR-400 128Mbx8, size of 16MB, CAS 3, at 200MHz.

DDR SDRAM asymmetric accesses

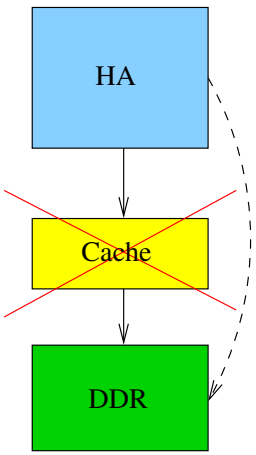


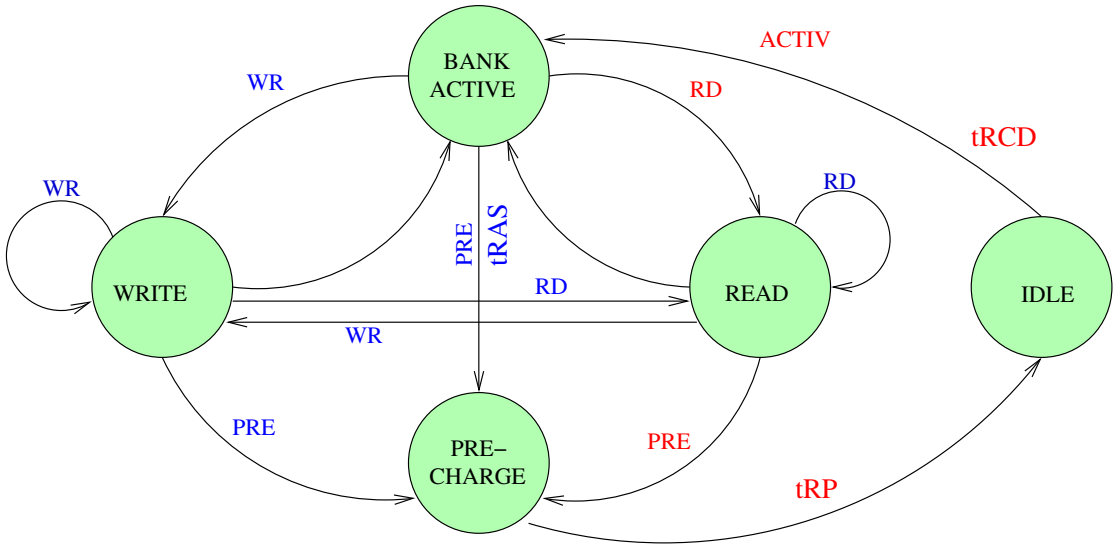


DDR-400 128Mbx8, size of 16MB, CAS 3, at 200MHz.

- Successive reads to the same row: 1 data every 10ns.

DDR SDRAM asymmetric accesses

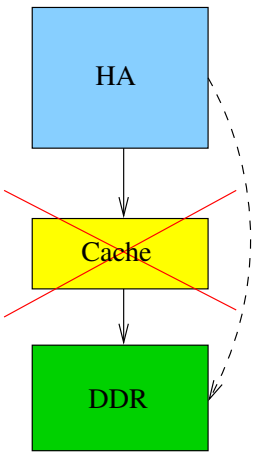


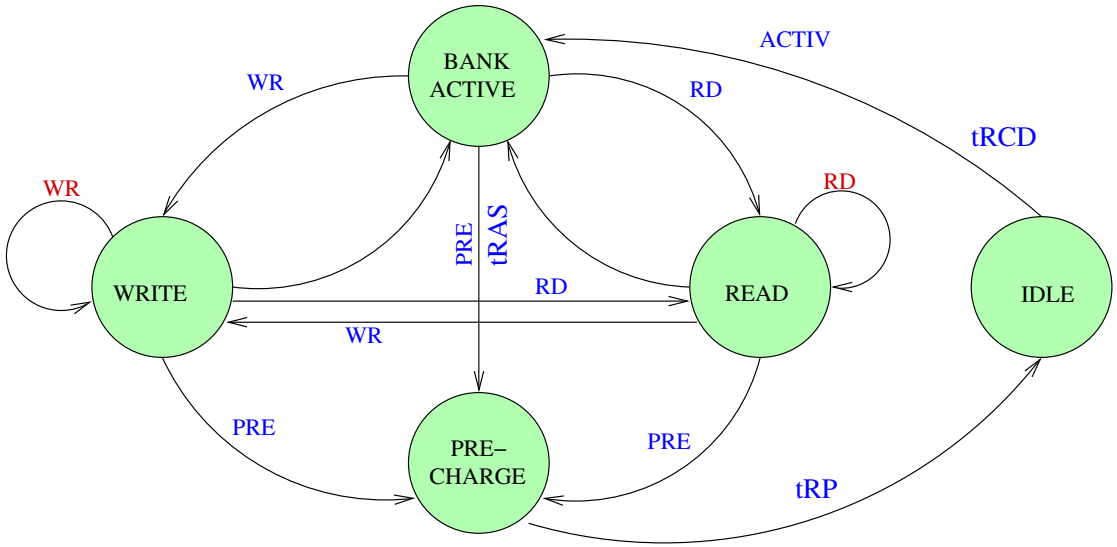


DDR-400 128Mbx8, size of 16MB, CAS 3, at 200MHz.

- Successive reads to the same row: 1 data every 10ns.
- Successive reads with a row change: 1 data every 80ns.

DDR SDRAM asymmetric accesses





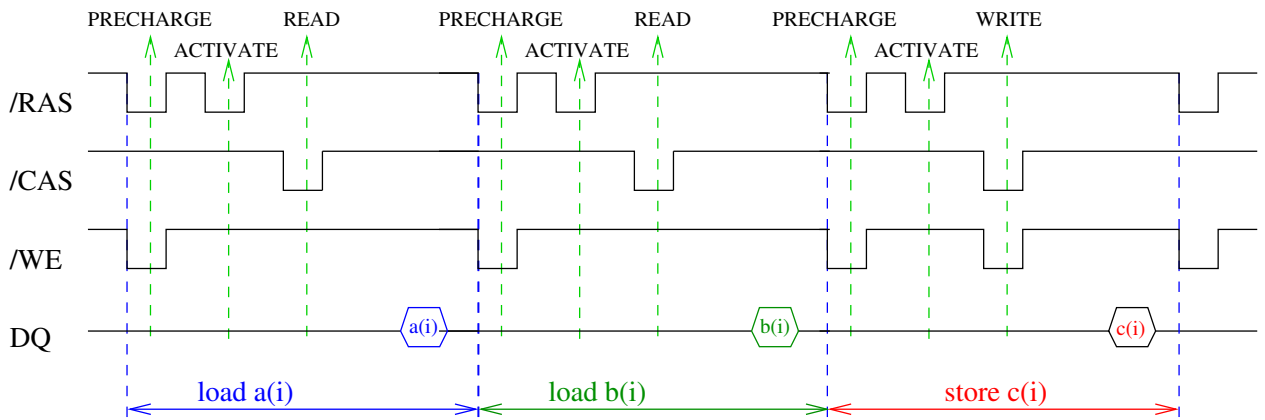
DDR-400 128Mbx8, size of 16MB, CAS 3, at 200MHz.

- Successive reads to the same row: 1 data every 10ns.
- Successive reads with a row change: 1 data every 80ns.

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

Simple example: vector sum

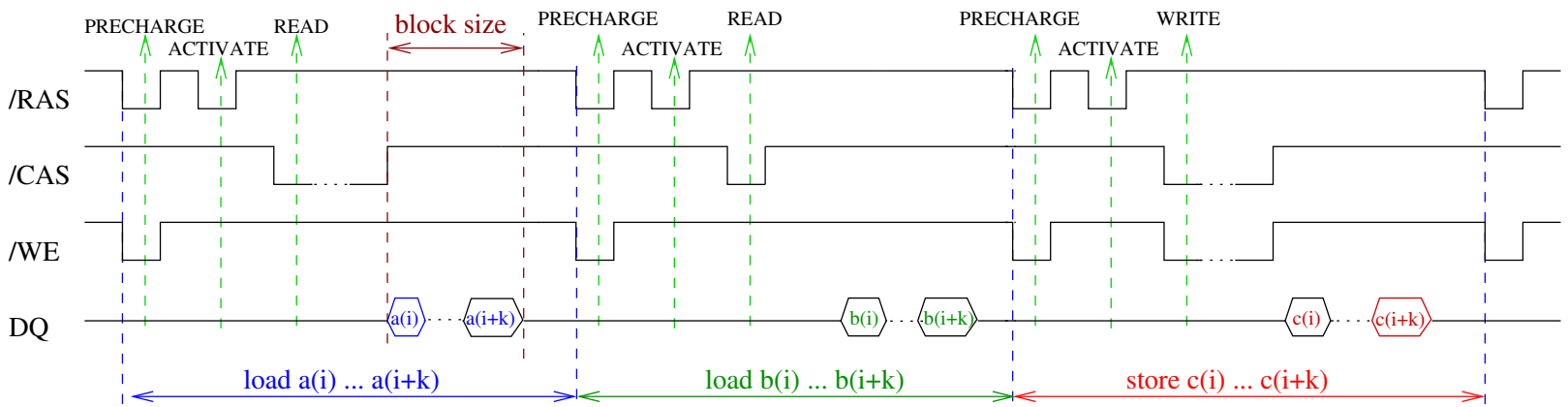
```
int vector_sum (int* __restrict__ a, int* __restrict__ b,
                int* __restrict__ c, int n) {
    int i;
    for (i = 0; i < n; i++) c[i] = a[i] + b[i];
    return 0;
}
```



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

Simple example: vector sum

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



Optimized block version: reduces gaps, exploits burst.

Strip-mining and loop distribution

Loop distribution: too large local memory.
Unrolling: too many registers.
} ➠ strip-mining + loop distribution.

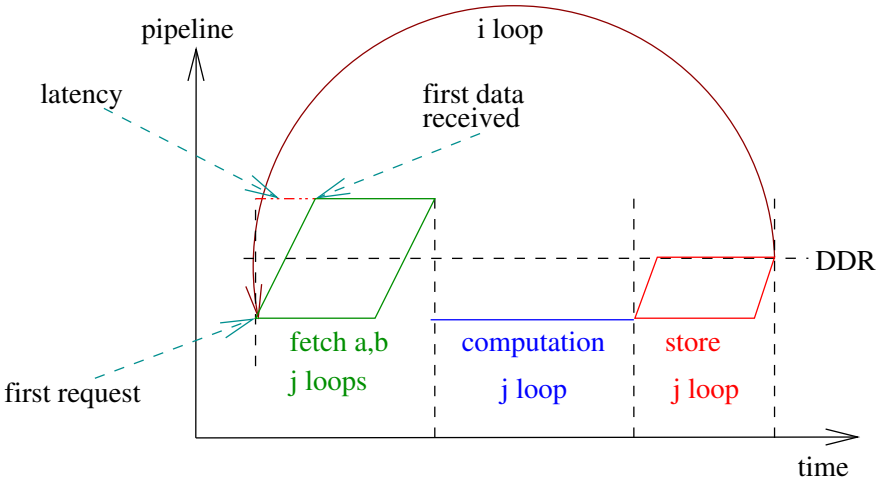
```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b[i+j]; //prefetch
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j]; //store
}
```

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
}
```

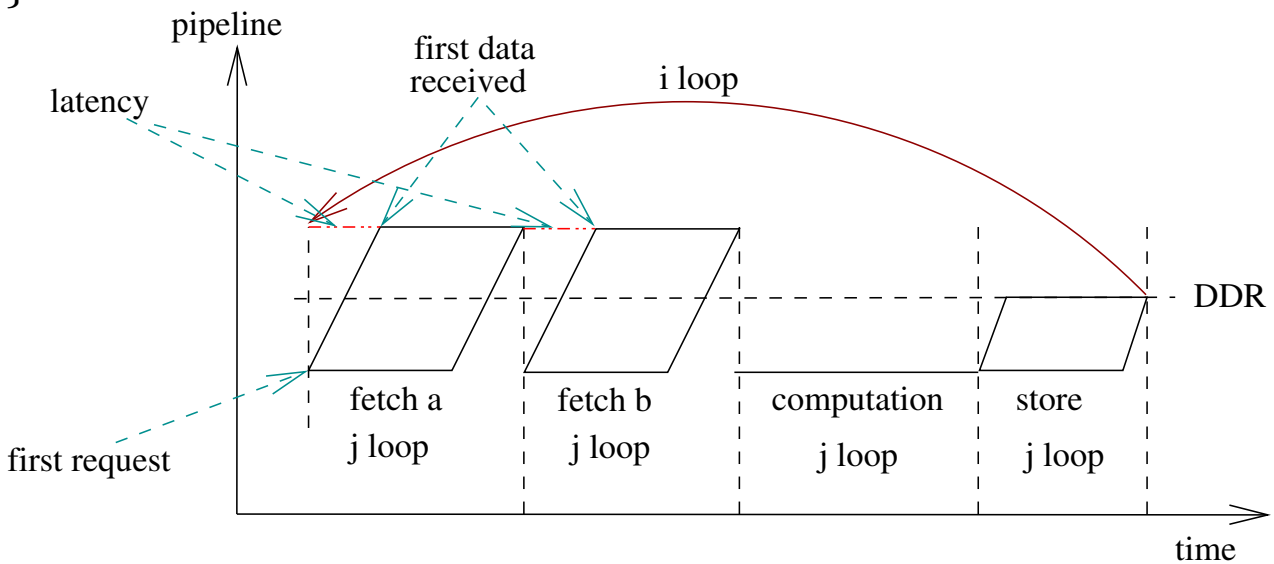
➠ Does not work!



- 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 with a single loop & an automaton

```
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;}
  }
}
```

- CPLI = 21! Problem with dependence analyzer and software pipeliner.
- Better behavior (CPLI=3) with *case* statement: by luck.
- Further loop unrolling to get CPLI = 1: too complex.

Emulating nested loops with a single loop & an automaton

```
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;}
  }
}
```

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

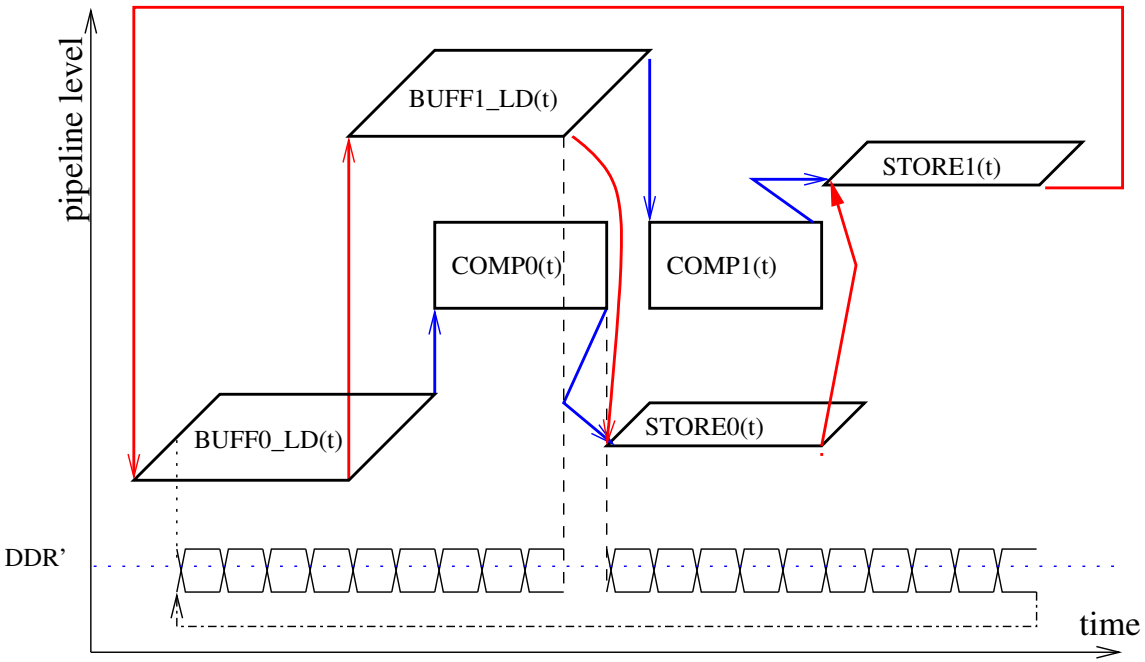
Emulating nested loops, regrouping transfers

```
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.

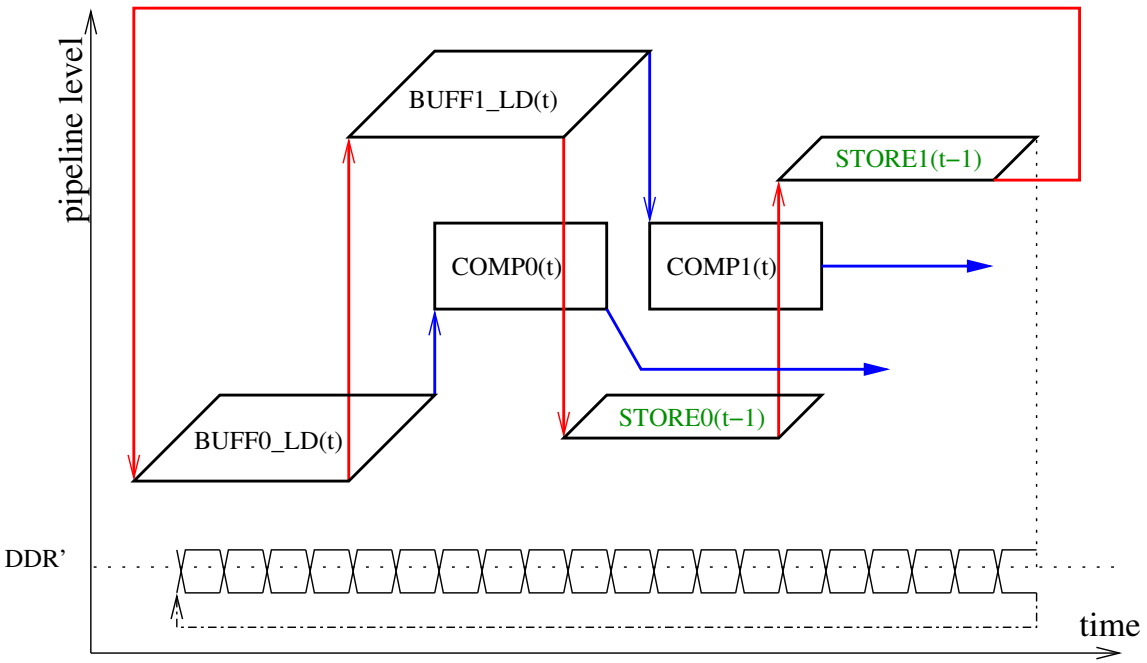
How to decrease CPLI and generalize to more complex codes?

Decompose into communication & computation processes



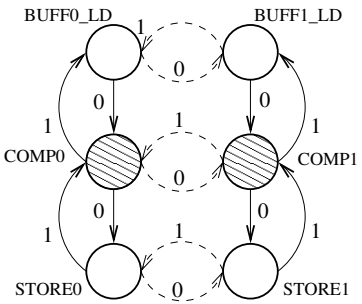
- Pipeline computation and communication.
- Force suitable order of DDR requests.
- Overlap computation and communication.
- Play with flow/anti dependences.

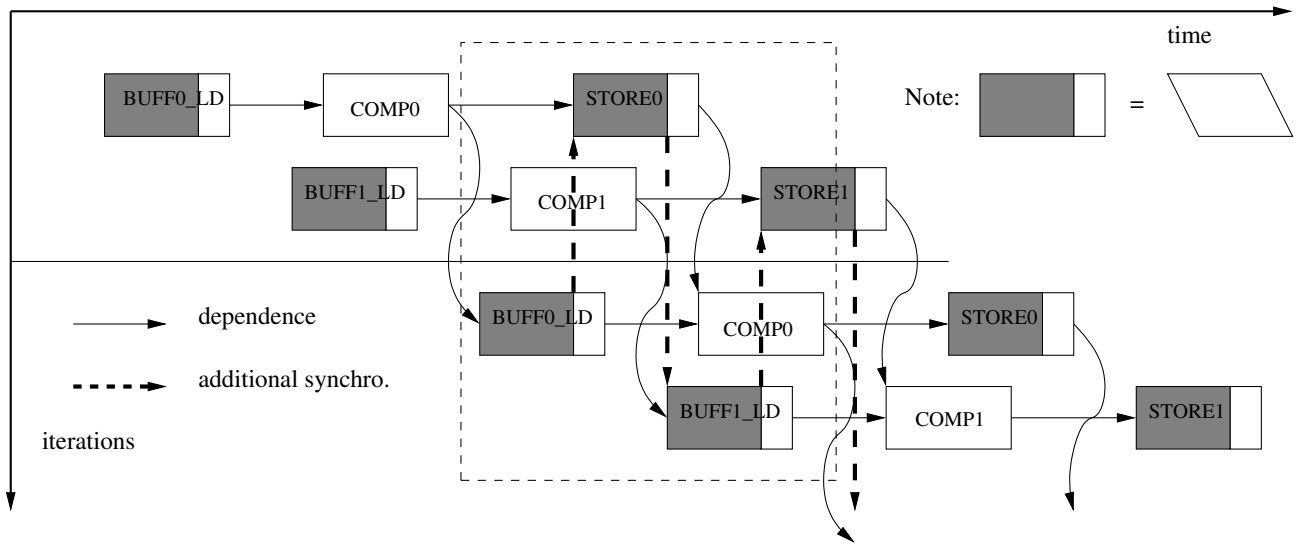
Decompose into communication & computation processes



- Pipeline computation and communication.
- Force suitable order of DDR requests.
- Overlap computation and communication.
- Play with flow/anti dependences.

Coarse-grain software pipelining





Semantically, "as if":

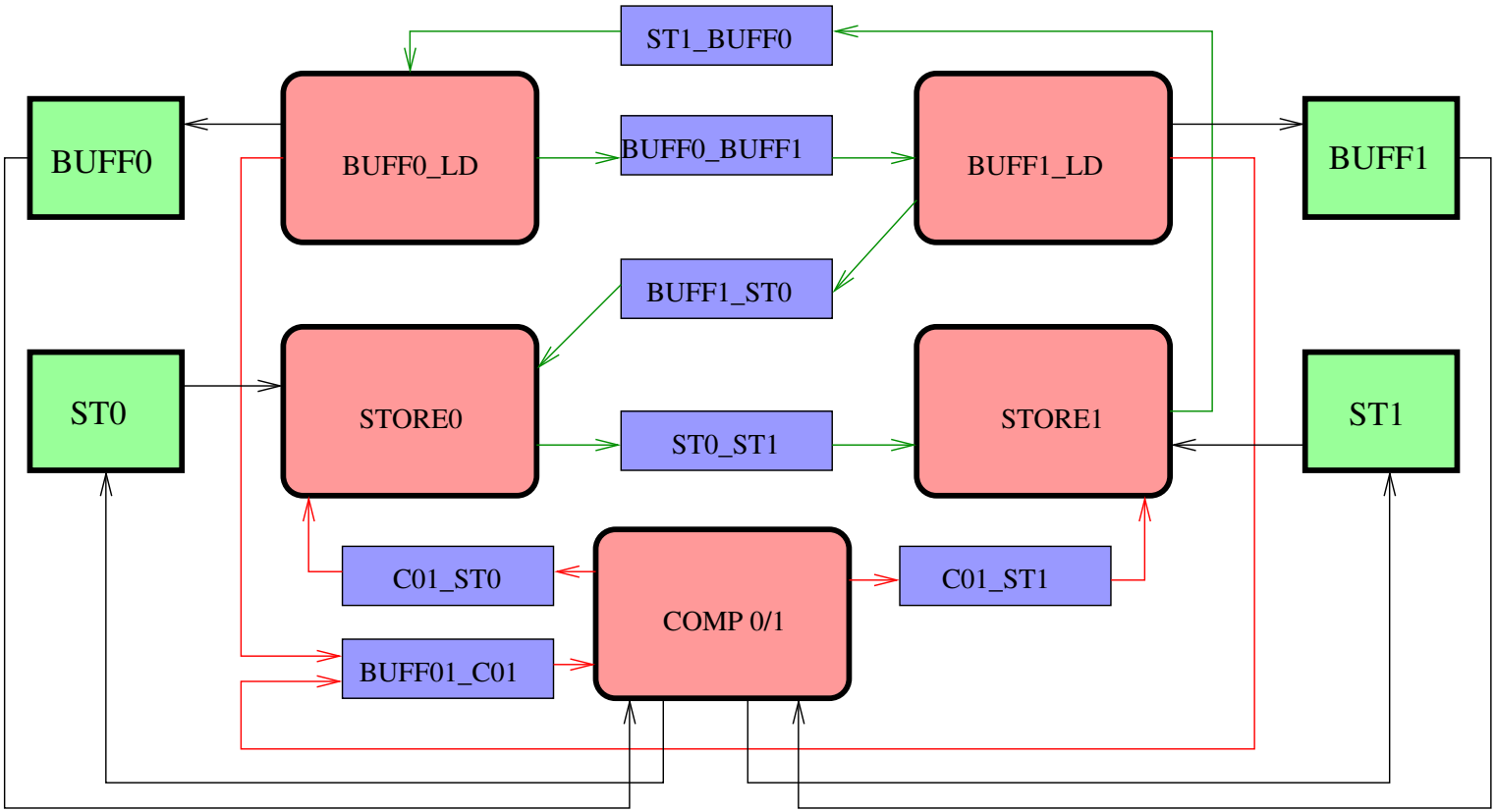
- BUFF0_LD(t) at $4t$, BUFF1_LD(t) at $4t + 2$.
- COMP0(t) at $4t + 2$, COMP1(t) at $4t + 4$.
- STORE0(t) at $4t + 5$, STORE1(t) at $4t + 7$.

Here, STORE0($t - 1$) finished before COMP0(t).

➽ Similar to "double buffering"

General architecture organization: typical example

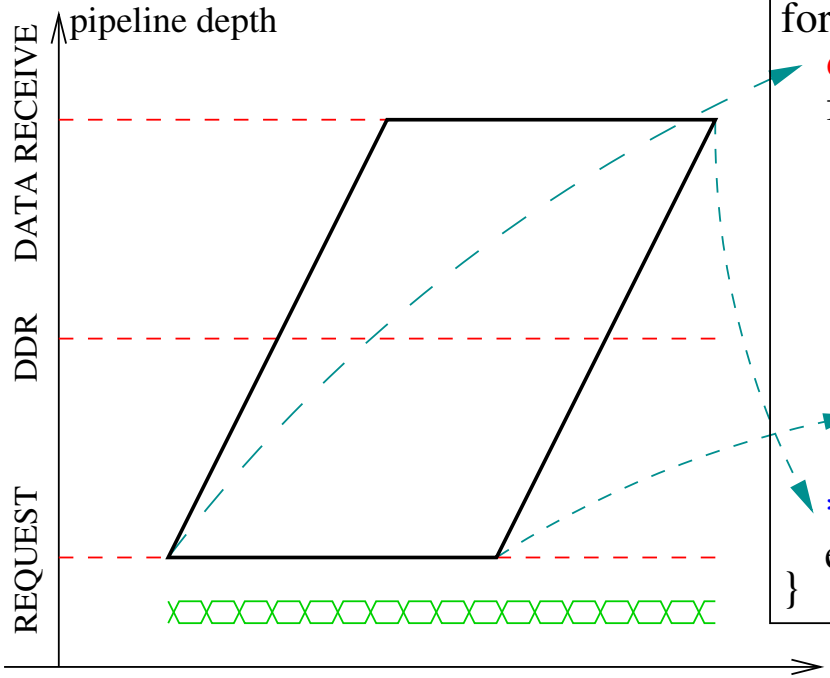
Use dedicated FIFOs of size 1 for synchronizations.
Data transfers done through explicit memory accesses.



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.

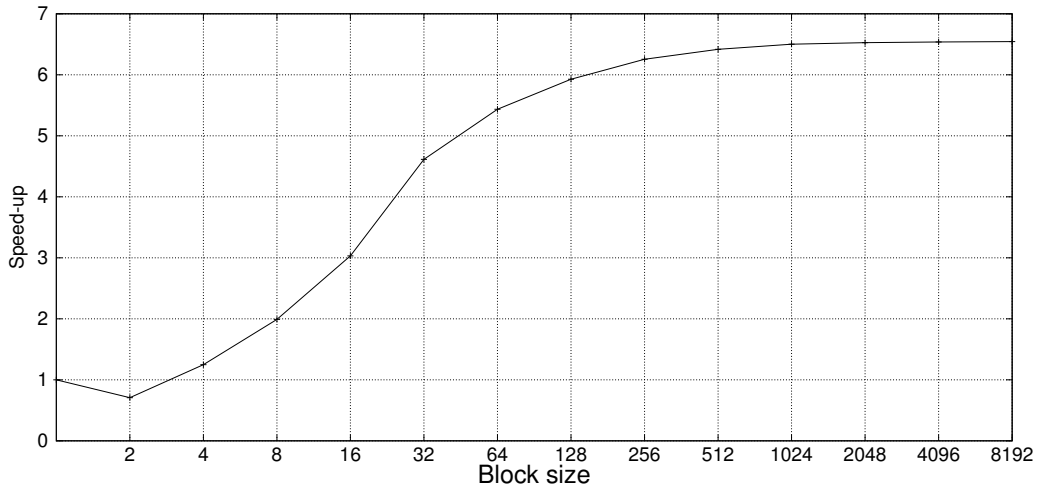



```
for (t=0; t<iter_space; t+=db_iter) {
    dummy_read +=*st1_buff0_read;
    for(r=0,tmp=dummy_read; r<r_sup; r++) {
        if (s==0) {
            compute local and global addresses for array a
            and scan the iteration space of array a;
            if end of iteration space: s++;
        } else if (s==1) { same as s==0 for array b; }
        transfer data from DDR to local memory;
        if (r == r_sup −1) {*buff0_buff1_write = 0; tmp = 0; }
    }
    *buff0_c01_write = tmp;
    external linearized loop control;
}
```

Experimental results: typical examples

Typical speed-up vs block size figure

(pS accurate simulation).



- SA: system alone.
- VS0 & VS1: vector sum direct & optimized version.
- MM0 & MM1: matrix-matrix multiply direct & optimized.

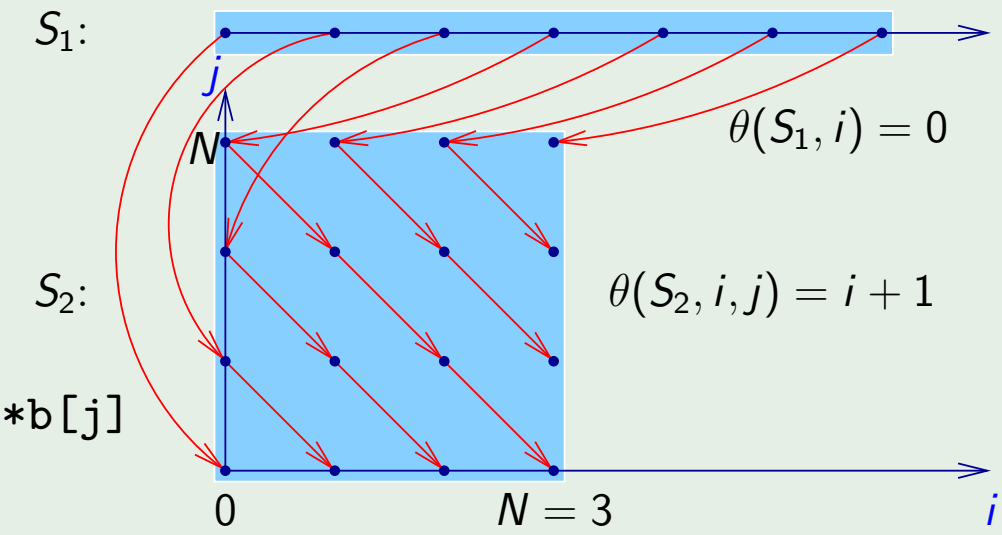
Kernel	Speed-up	ALUT	Dedicated registers	Total registers	Total block memory bits	DSP block 9-bit elements	Max Frequency (MHz > 100)
SA	1	5105	3606	3738	66908	8	205.85
VS0	1	5333	4607	4739	68956	8	189.04
VS1	6.54	10345	10346	11478	269148	8	175.93
MM0	1	6452	4557	4709	68956	40	191.09
MM1	7.37	15255	15630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

Outline

Polyhedral model

Ex: product of polynomials

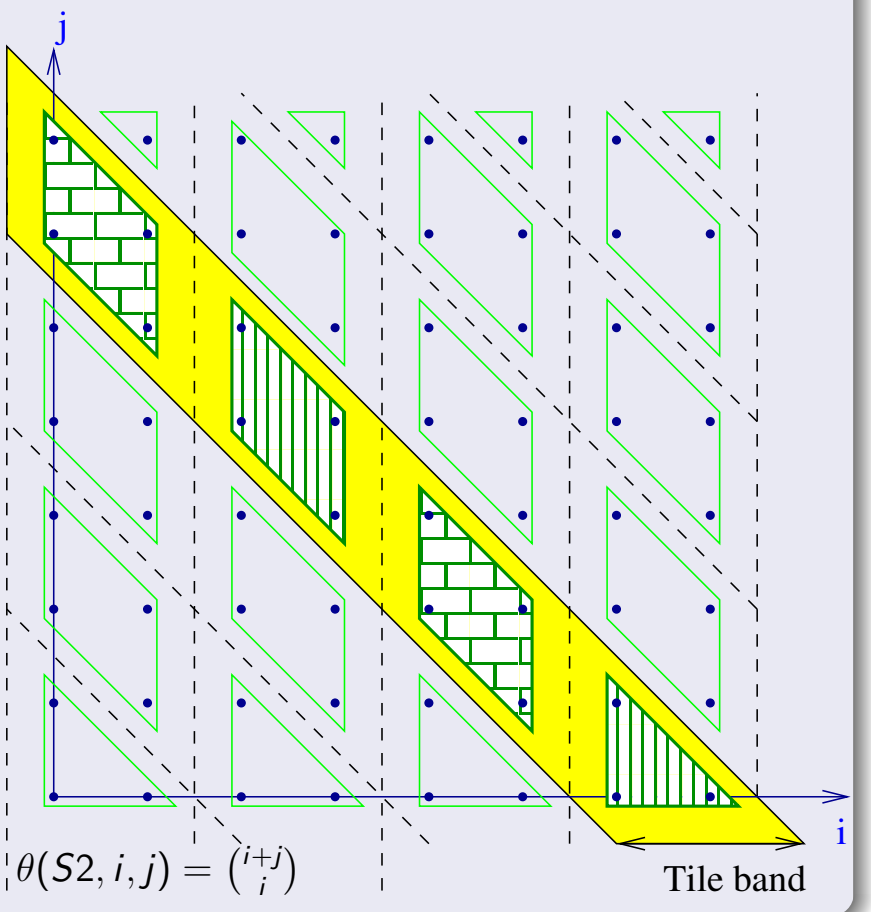
```
  for (i=0; i<= 2*N; i++)
S1: c[i] = 0;

  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

Polyhedral model: tiling

Tiled product of polynomials



- Tile: atomic block operation
- Increases granularity of computations
- Tile band: double buffering
- n loops transformed into n tile loops + n block loops
- Expressed from permutable loops (function θ)

Overview of the method

Derive automatically the C2H-compliant C functions for the pipelined accelerators: load, store and compute. Blocks obtained by loop tiling, pipelined in a "double-buffering" scheme.

1. **Communication coalescing** - prefetches data out of tile, following rows, and exploits data reuse
2. **Local memory management** - defines memory elements, reduces size, and computes access functions
3. **Code generation** - generates final C code in a linearized form while optimizing accesses to the DDR

Overview of the method

Derive automatically the C2H-compliant C functions for the pipelined accelerators: load, store and compute. Blocks obtained by loop tiling, pipelined in a "double-buffering" scheme.

1. **Communication coalescing** - prefetches data out of tile, following rows, and exploits data reuse
2. **Local memory management** - defines memory elements, reduces size, and computes access functions
3. **Code generation** - generates final C code in a linearized form while optimizing accesses to the DDR

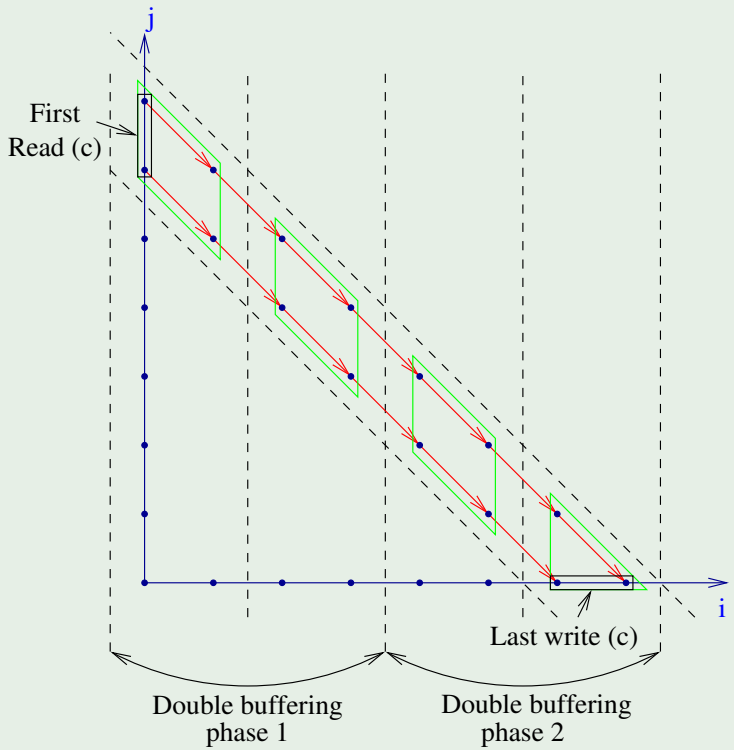
Overview of the method

Derive automatically the C2H-compliant C functions for the pipelined accelerators: load, store and compute. Blocks obtained by loop tiling, pipelined in a "double-buffering" scheme.

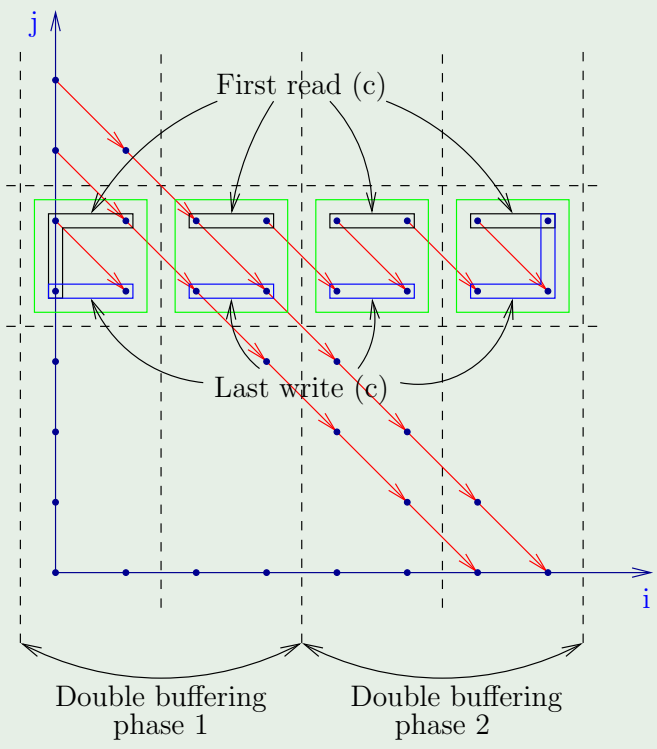
1. **Communication coalescing** - prefetches data out of tile, following rows, and exploits data reuse
2. **Local memory management** - defines memory elements, reduces size, and computes access functions
3. **Code generation** - generates final C code in a linearized form while optimizing accesses to the DDR

Loop tiling: impact on communication

Version 1



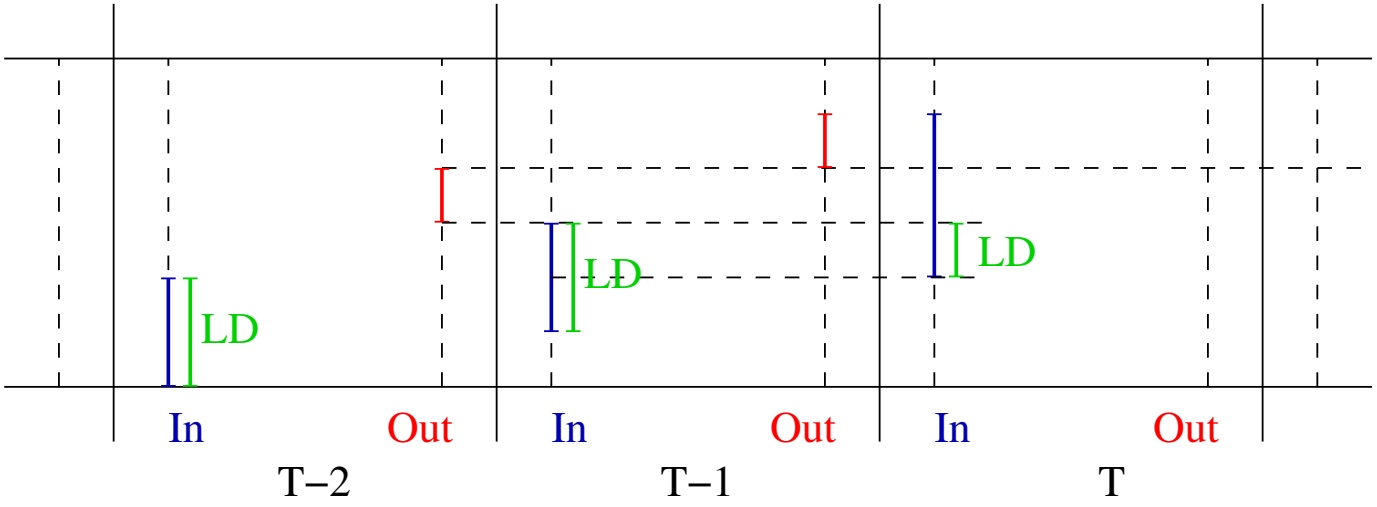
Version 2



Load $\sim$ FirstRead $\cap$ tile domain **Store** $\sim$ LastWrite $\cap$ tile domain

FirstRead/LastWrite $\sim$ Array dataflow analysis

Formalization of valid, exact, and approximated load



Valid load

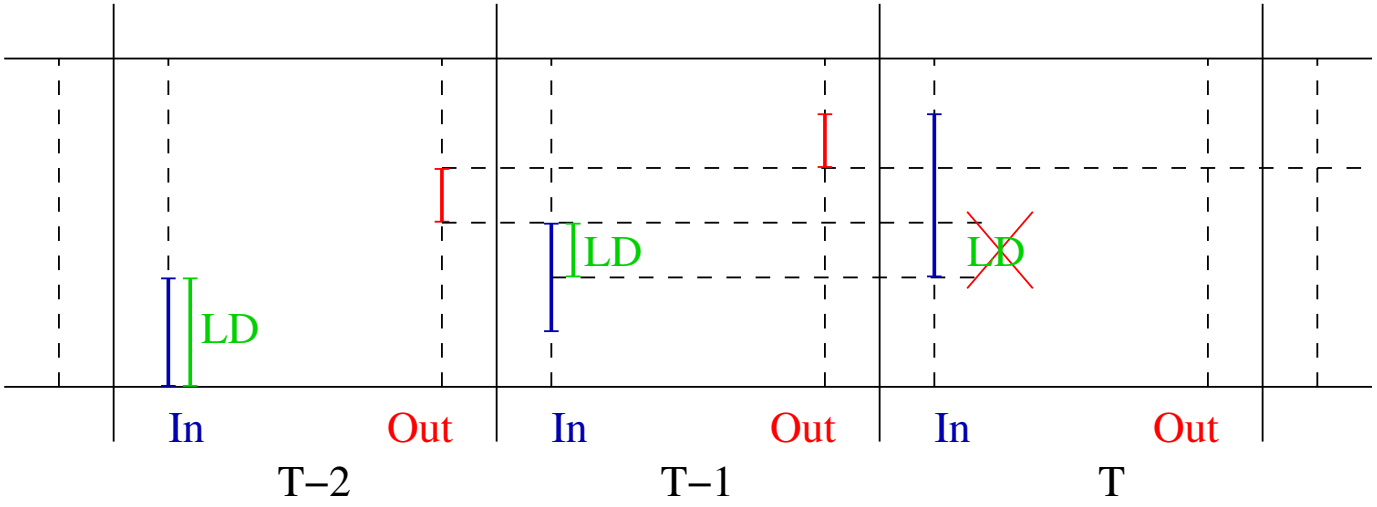
(i) Load at least what is needed but not previously produced:

$$\cup_{t \leq T} \left\{ \mathrm{In}(t) \setminus \mathrm{Out}(t' < t) \right\} \subseteq \mathrm{Load}(t \leq T)$$

(ii) Do not overwrite live data:

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

Formalization of valid, exact, and approximated load



Exact load

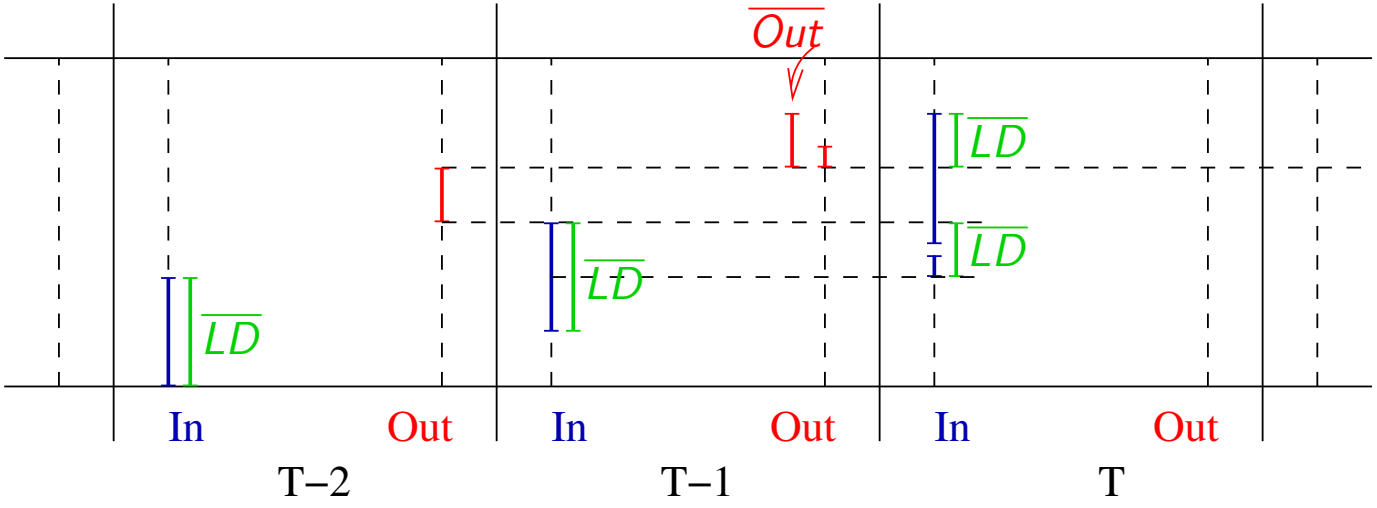
(i) Load exactly what is needed but not previously produced:

$$\forall T, \cup_{t \leq T} \left\{ \mathrm{In}(t) \setminus \mathrm{Out}(t' < t) \right\} = \mathrm{Load}(t \leq T)$$

(ii) All loads must be disjoint:

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

Formalization of valid, exact, and approximated load



Valid approximated load

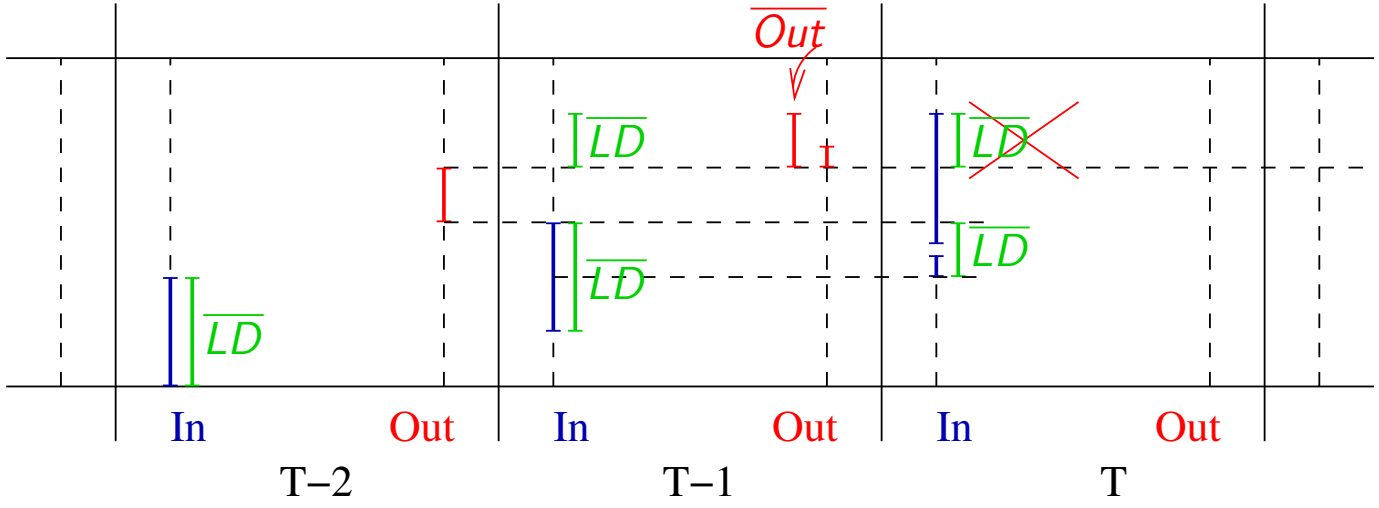
(i) Load at least the exact amount of data:

$$\cup_{t \leq T} \left\{ \overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t' < t) \right\} \subseteq \mathrm{Load}(t \leq T)$$

(ii) Do not overwrite possible live 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:

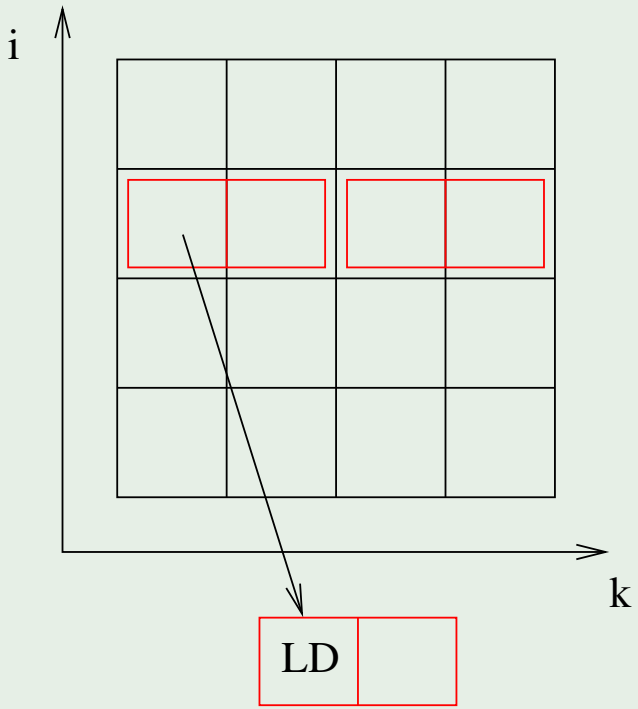
$$\cup_{t \leq T} \left\{ \overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t' < t) \right\} \subseteq \mathrm{Load}(t \leq T)$$

(ii) Do not overwrite possible live data:

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

Local memory management

Simplest case: immediate data consumption with no reuse



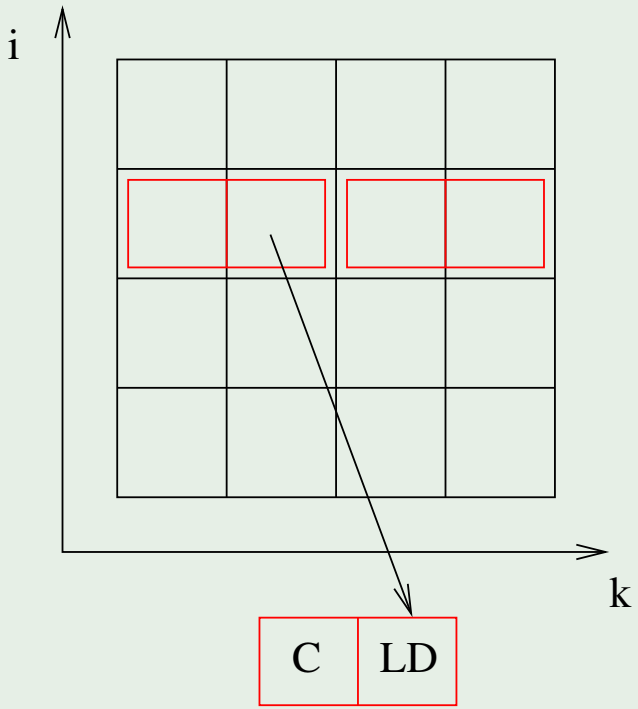
$a(i)$: external memory

$a_{tmp}(\sigma(i))$: local memory

- Array contraction with a double buffering scheduling (loop unrolling by 2 plus software pipelining of tiles)

Local memory management

Simplest case: immediate data consumption with no reuse



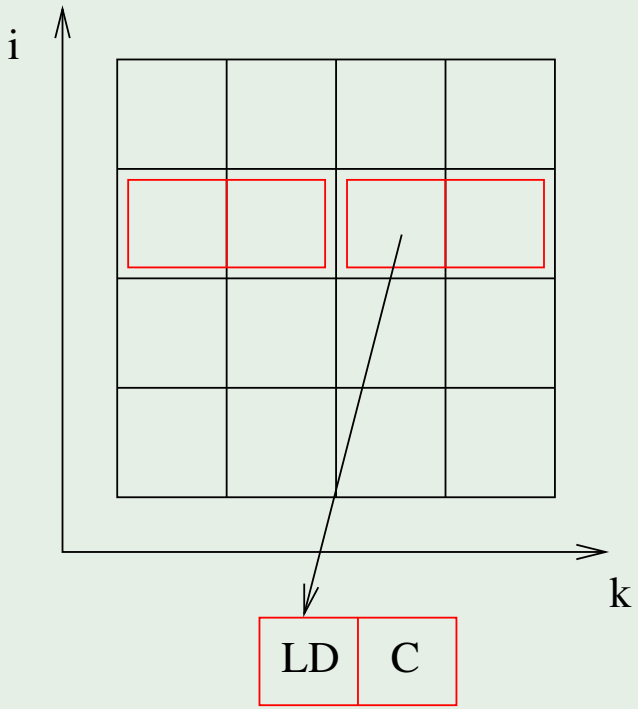
$a(i)$: external memory

$a_{tmp}(\sigma(i))$: local memory

- Array contraction with a double buffering scheduling (loop unrolling by 2 plus software pipelining of tiles)

Local memory management

Simplest case: immediate data consumption with no reuse



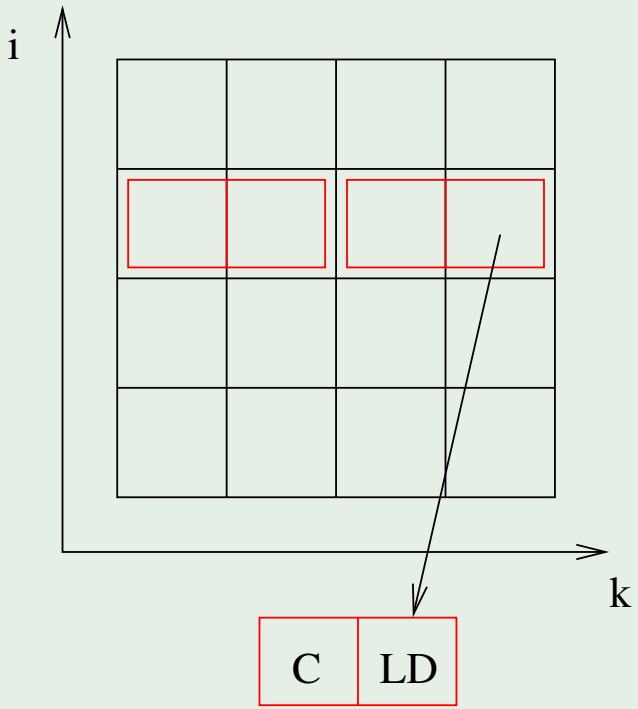
$a(i)$: external memory

$a_{tmp}(\sigma(i))$: local memory

- Array contraction with a double buffering scheduling (loop unrolling by 2 plus software pipelining of tiles)

Local memory management

Simplest case: immediate data consumption with no reuse



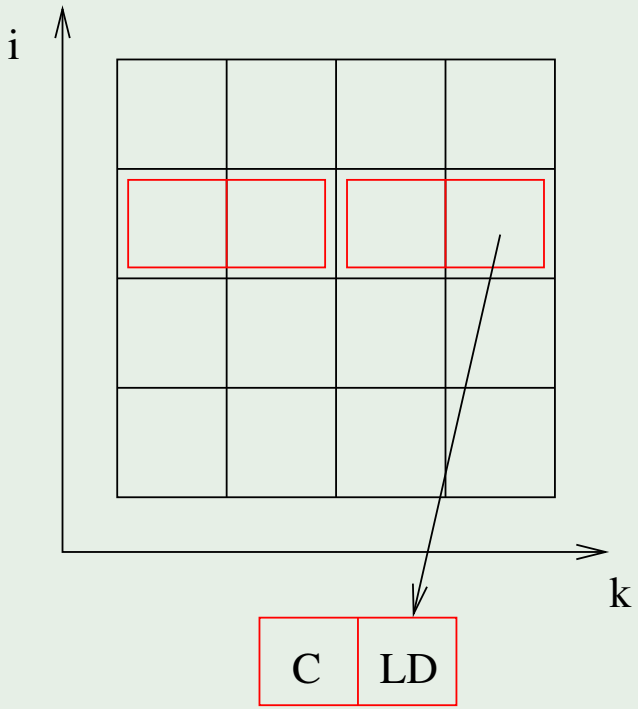
$a(i)$: external memory

$a_{tmp}(\sigma(i))$: local memory

- Array contraction with a double buffering scheduling (loop unrolling by 2 plus software pipelining of tiles)

Local memory management

Simplest case: immediate data consumption with no reuse



$a(i)$: external memory

$a_{tmp}(\sigma(i))$: local memory

- Array contraction with a double buffering scheduling (loop unrolling by 2 plus software pipelining of tiles)

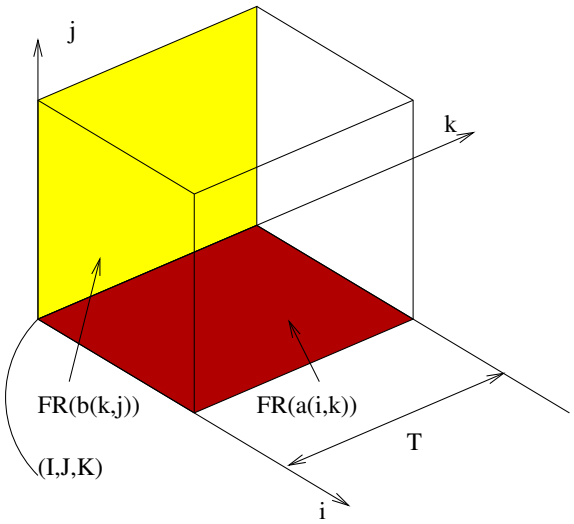
General organization

Iteration over tiles

```
1  void Load0() {
2   for (T_1 = ...) {
3    ...
4     for (T_{n-1} = ...) {
5      for (T_n = L(T_1, ..., T_{n-1});
6            T_n ≤ U(T_1, ..., T_{n-1});
7            T_n += 2) {
8       //Synchronize from Store1()
9       //Load(T_1, ..., T_n) + sync. to Load1()
10      //Synchronize to Compute()
11    }}...}}
12 void Load1() {...}
13 void Store0() {...}
14 void Store1() {...}
15 void Compute() {...}
```

- Similar for functions Load1, Store0, Store1, Compute
- Loop nests: linearized
- Synchronizations
- Code generation ensures spatial data locality for optimized DDR access

Kernel code generation



$$D_a : \begin{cases} T \cdot I \leq i \leq T \cdot I + (T-1) \\ j = T \cdot J \\ T \cdot K \leq k \leq T \cdot K + (T-1) \end{cases} : LD(a(i,k))$$

$$D_b : \begin{cases} i = T \cdot I \\ T \cdot J \leq j \leq T \cdot J + (T-1) \\ T \cdot K \leq k \leq T \cdot K + (T-1) \end{cases} : LD(b(k,j))$$

Linearized loops ➽ use Boulet-Feautrier with $(D_a, \theta_a), (D_b, \theta_b)$

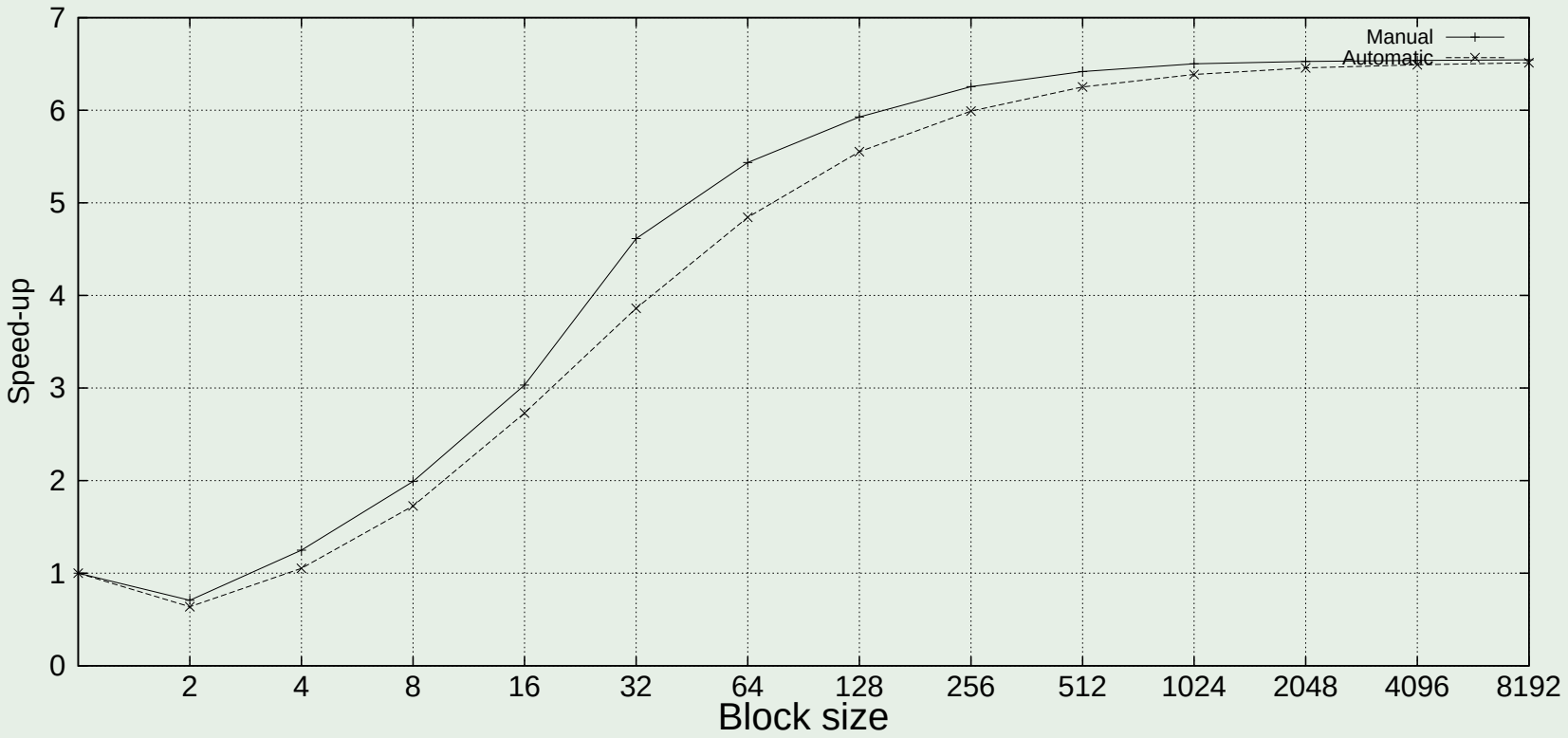
$$\left.\begin{array}{l} \theta_a(i,j,k) = (0, i, k) \\ \theta_b(i,j,k) = (1, k, j) \end{array}\right\}$$

Arrays are read one after the other
Scan arrays row by row

Manually- vs. automatically-transformed

Method implemented in CHUBA (array contraction: BEE)

Ex: Vector sum speed-ups



Synthesis results

Kernel	Speed-up	ALUT	Dedicated registers	Total registers
System alone		4406	3474	3606
DMA direct implementation	1	4598	3612	3744
DMA transformed manually (1K tile)	5.95	9665	10244	10376
DMA transformed manually (8K tile)	6.01	9853	10517	10649
DMA automatic (8K tile)	5.98	11052	12133	12265
Vector sum direct implementation	1	5333	4607	4739
Vector sum transformed manually (1K tile)	6.50	10345	10346	11478
Vector sum transformed manually (8K tile)	6.54	10881	11361	11493
Vector sum automatic (8K tile)	6.51	11632	13127	13259

Kernel	Total block memory bits	DSP block 9-bit elements	Max Freq. (MHz)
System alone	66908	8	205.85
DMA direct implementation	66908	8	200.52
DMA transformed manually (1k tile)	203100	8	167.25
DMA transformed manually (8k tile)	1120604	8	162.55
DMA automatic (8k tile)	1120348	48	167.87
Vector sum direct implementation	68956	8	189.04
Vector sum transformed manually (1k tile)	269148	8	175.93
Vector sum transformed manually (8k tile)	1645404	8	164
Vector sum automatic (8k tile)	1644892	48	159.8

Synthesis results

Kernel	Speed-up	ALUT	Dedicated registers	Total registers
System alone		4406	3474	3606
DMA direct implementation	1	4598	3612	3744
DMA transformed manually (1K tile)	5.95	9665	10244	10376
DMA transformed manually (8K tile)	6.01	9853	10517	10649
DMA automatic (8K tile)	5.98	11052	12133	12265
Vector sum direct implementation	1	5333	4607	4739
Vector sum transformed manually (1K tile)	6.50	10345	10346	11478
Vector sum transformed manually (8K tile)	6.54	10881	11361	11493
Vector sum automatic (8K tile)	6.51	11632	13127	13259

Kernel	Total block memory bits	DSP block 9-bit elements	Max Freq. (MHz)
System alone	66908	8	205.85
DMA direct implementation	66908	8	200.52
DMA transformed manually (1k tile)	203100	8	167.25
DMA transformed manually (8k tile)	1120604	8	162.55
DMA automatic (8k tile)	1120348	48	167.87
Vector sum direct implementation	68956	8	189.04
Vector sum transformed manually (1k tile)	269148	8	175.93
Vector sum transformed manually (8k tile)	1645404	8	164
Vector sum automatic (8k tile)	1644892	48	159.8

Synthesis results

Kernel	Speed-up	ALUT	Dedicated registers	Total registers
System alone		4406	3474	3606
DMA direct implementation	1	4598	3612	3744
DMA transformed manually (1K tile)	5.95	9665	10244	10376
DMA transformed manually (8K tile)	6.01	9853	10517	10649
DMA automatic (8K tile)	5.98	11052	12133	12265
Vector sum direct implementation	1	5333	4607	4739
Vector sum transformed manually (1K tile)	6.50	10345	10346	11478
Vector sum transformed manually (8K tile)	6.54	10881	11361	11493
Vector sum automatic (8K tile)	6.51	11632	13127	13259

Kernel	Total block memory bits	DSP block 9-bit elements	Max Freq. (MHz)
System alone	66908	8	205.85
DMA direct implementation	66908	8	200.52
DMA transformed manually (1k tile)	203100	8	167.25
DMA transformed manually (8k tile)	1120604	8	162.55
DMA automatic (8k tile)	1120348	48	167.87
Vector sum direct implementation	68956	8	189.04
Vector sum transformed manually (1k tile)	269148	8	175.93
Vector sum transformed manually (8k tile)	1645404	8	164
Vector sum automatic (8k tile)	1644892	48	159.8

Synthesis results

Kernel	Speed-up	ALUT	Dedicated registers	Total registers
System alone		4406	3474	3606
DMA direct implementation	1	4598	3612	3744
DMA transformed manually (1K tile)	5.95	9665	10244	10376
DMA transformed manually (8K tile)	6.01	9853	10517	10649
DMA automatic (8K tile)	5.98	11052	12133	12265
Vector sum direct implementation	1	5333	4607	4739
Vector sum transformed manually (1K tile)	6.50	10345	10346	11478
Vector sum transformed manually (8K tile)	6.54	10881	11361	11493
Vector sum automatic (8K tile)	6.51	11632	13127	13259

Kernel	Total block memory bits	DSP block 9-bit elements	Max Freq. (MHz)
System alone	66908	8	205.85
DMA direct implementation	66908	8	200.52
DMA transformed manually (1k tile)	203100	8	167.25
DMA transformed manually (8k tile)	1120604	8	162.55
DMA automatic (8k tile)	1120348	48	167.87
Vector sum direct implementation	68956	8	189.04
Vector sum transformed manually (1k tile)	269148	8	175.93
Vector sum transformed manually (8k tile)	1645404	8	164
Vector sum automatic (8k tile)	1644892	48	159.8

Contributions

- Focus on memory optimizations and interface generation.
- Demonstrates importance of source-to-source optimizations (script + use of transformation tool) in front of HLS tools.
- To our knowledge, first process to automate communications and integrate hardware accelerators, entirely at C level.
- Identifies important needs for synchronization mechanisms at source level and for better pragmas (e.g., *restrict* for pairs).
- Analysis and transformations appear to be very similar to optimizations needed for GPUs.
- Starting point for using HLS tools as back-end compilers?

Perspectives

Many opportunities for improvements.

- Design more domain-specific code generation.
- Define compilation directives at C level for hardware synthesis.
- Design customized memories and inter-processes buffers.
- Exploit schedule with slacks for GALS pipelined designs.
- ...