Advanced Computer Architecture

—

Part II: Embedded Computing Challenges of HLS

Paolo Ienne
<paolo.ienne@epfl.ch>

(with Lana Josipović)

Difficult Cases for HLS

- Which schedules can be achieved with HLS tools?
  - What are desirable schedules?
- How software programmers handle such cases?
  - What is the situation with traditional multicore processors?
- Has the research community tried the same for HLS?

1

Variable Latency

How can one cope with variable latency operations

Variable Latency

- Variable latencies in computations, memory accesses, or loop execution time
  - Floating-point units, cache hit/miss, variable loop bounds, early-exit condition,…
  - Prevent good pipelining using standard HLS techniques

Conservative Pipelines

- Static HLS: assume the **worst-case latency**
  - Reserve additional pipeline stages for variable-latency operations
  - **Hardware overhead** in area (power, timing)→ may not be feasible for larger latencies
  - High throughput in particular cases (e.g., no loop-carried dependencies on variable-latency op)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Focus on the inner loop, initially

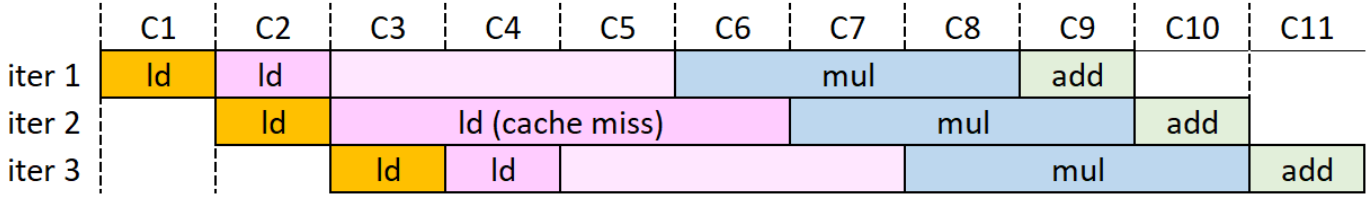
Sparse-matrix dense-vector multiplication (SpMV)

Conservative Pipelines

- Static HLS: assume the **worst-case latency**
  - Reserve additional pipeline stages for variable-latency operations
  - **Hardware overhead** in area (power, timing)→ may not be feasible for larger latencies
  - High throughput in particular cases (e.g., no loop-carried dependencies on variable-latency op)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable-latency memory access



Assuming worst-case latency: regardless of whether a cache hit or miss occurs, create schedule assuming it is a miss

Cache hit latency = 1
Cache miss latency = 4
N (number of loop iterations) = 3
L (iteration latency) = 9
II (initiation interval) = 1
Total latency = (N-1)*II + L = 17

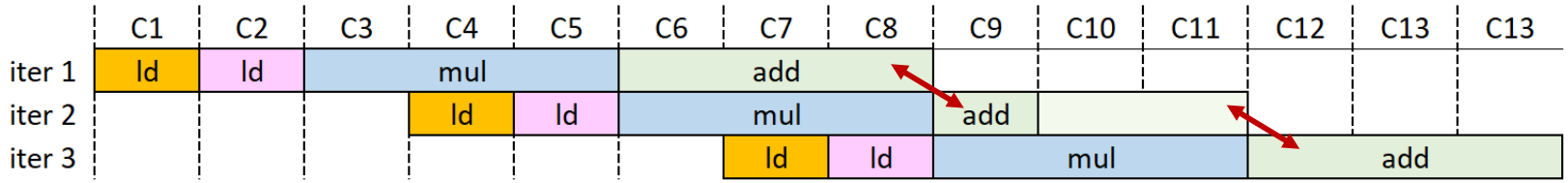
High throughput at a latency & area cost

Conservative Pipelines

- Static HLS: assume the **worst-case latency**
  - Reserve additional pipeline stages for variable-latency operations
  - **Hardware overhead** in area (power, timing)→ may not be feasible for larger latencies
  - High throughput in particular cases (e.g., no loop-carried dependencies on variable-latency op)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }
  out[i] = tmp;
}
```

Variable-latency addition



Assuming worst-case latency: regardless of actual add latency, create schedule assuming max. latency

Add latency = 1-3
L = 8
II = 3 always!
Total latency = 14

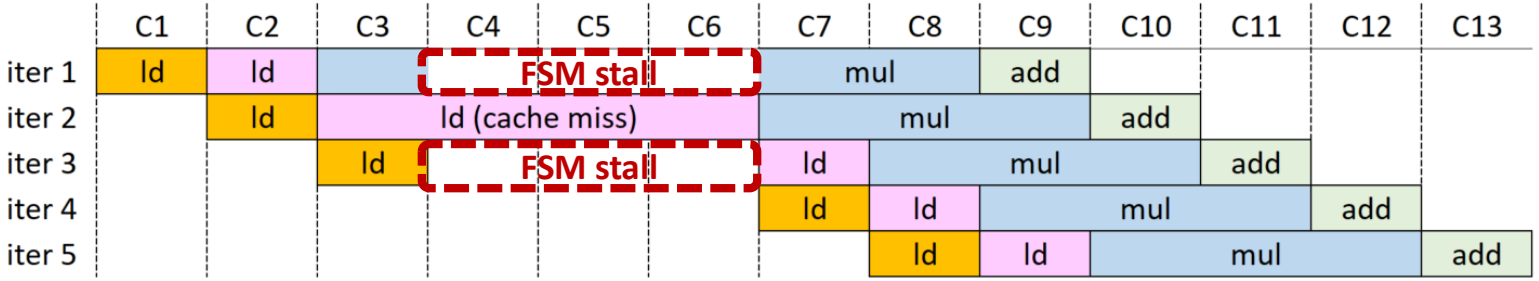
Worst-case throughput in case of loop-carried dependency!
(plus latency & area cost)

Pipeline Stalling

- Static HLS: stall **entire pipeline** in case of a variable-latency event
  - Schedule each operation based on its **minimum latency**
  - If an operation does not complete within min. latency, block operation and **stall pipeline**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable-latency memory access



Stalling entire pipeline in case the operation does not complete in min. latency

Cache hit latency = 1
Cache miss latency = 4
II = 1-4
Best-case total latency = 11
Worst-case total latency = 17

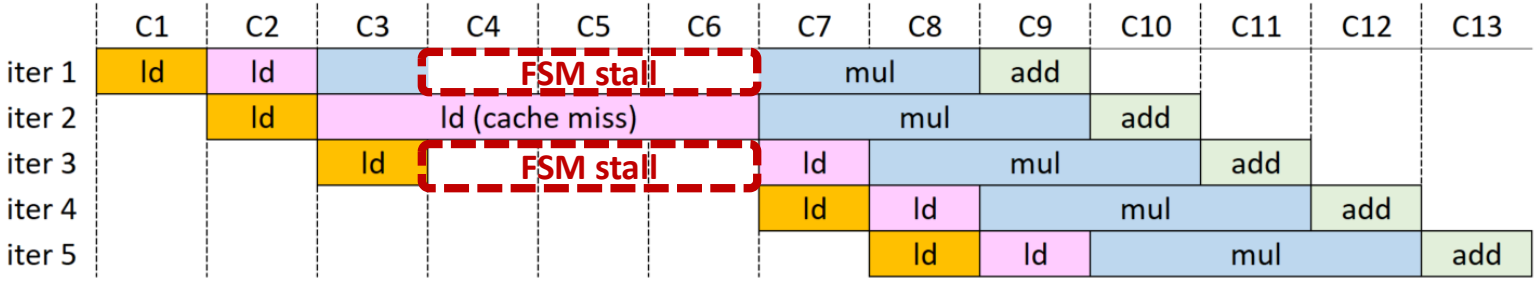
Performance significantly varies with the number and distribution of cache misses

Pipeline Stalling

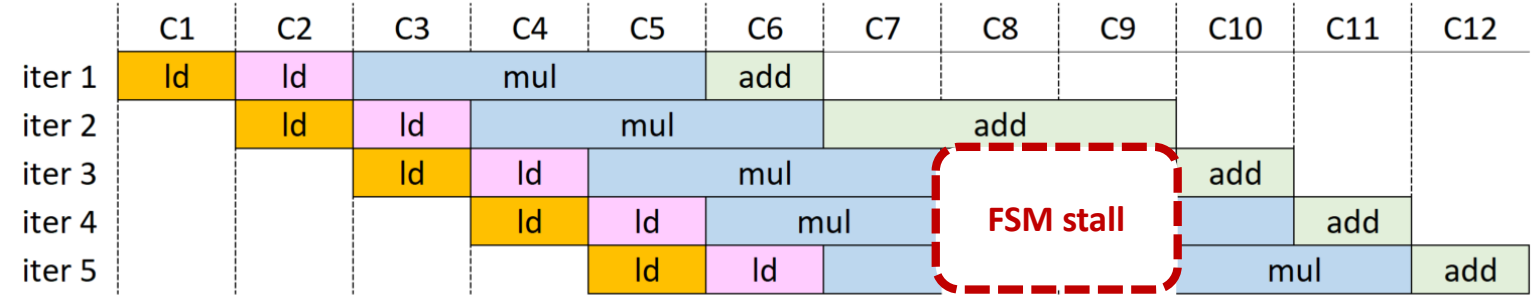
- Static HLS: stall **entire pipeline** in case of a variable-latency event
  - Schedule each operation based on its **minimum latency**
  - If an operation does not complete within min. latency, block operation and **stall pipeline**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable-latency addition



Stalling entire pipeline in case the operation does not complete in min. latency

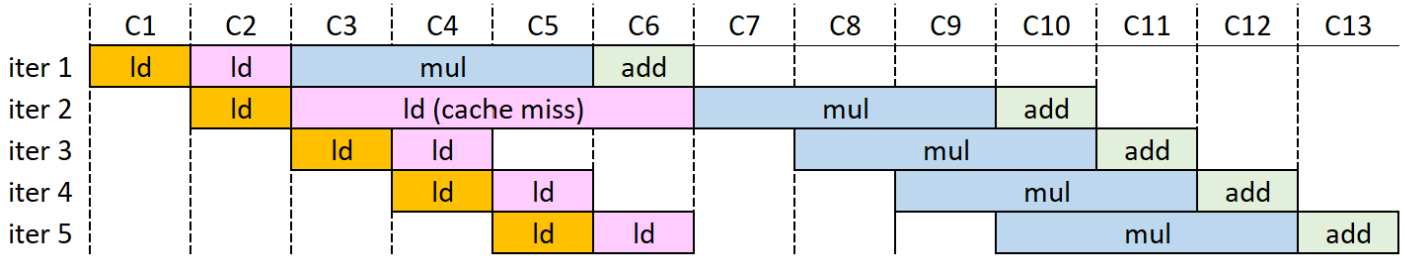


Dataflow

- **Dynamic HLS**: naturally handles variable latencies
  - Handshaking mechanism stalls the successors of long-latency operation
  - Other computations can advance during stall
  - Yet, computations in the same unit happen **strictly in order**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable-latency memory access



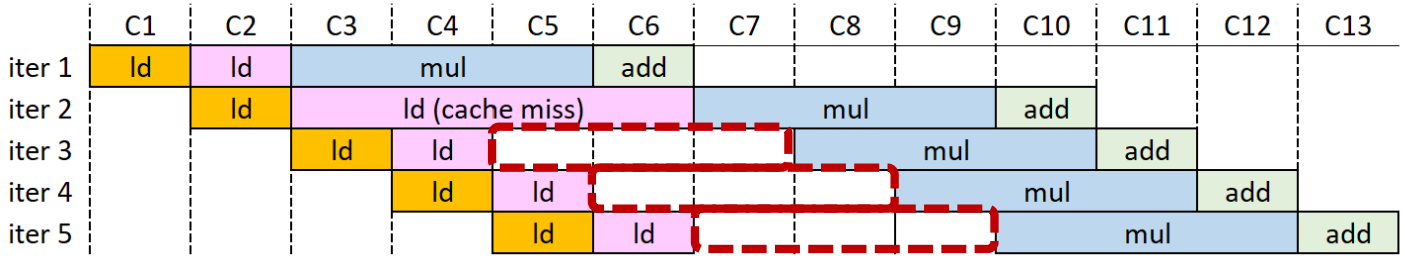
Dynamically scheduled pipeline: some computations can advance during long-latency stall

Dataflow

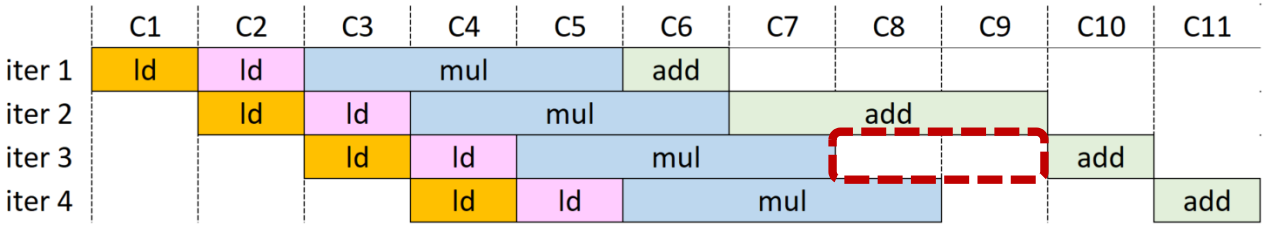
- **Dynamic HLS**: naturally handles variable latencies
  - Handshaking mechanism stalls the successors of long-latency operation
  - Other computations can advance during stall
  - Yet, computations in the same unit happen **strictly in order**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }
  out[i] = tmp;
}
```

Variable-latency addition



Dynamically scheduled pipeline: some computations can advance during long-latency stall

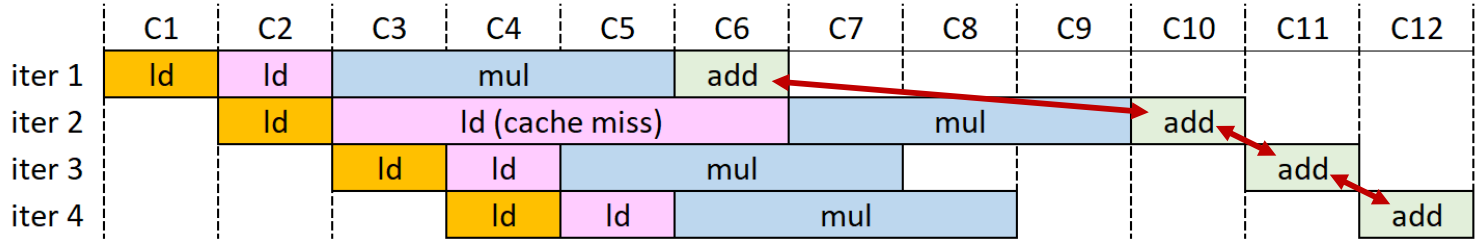


Schedule also adapts well in the presence of loop-carried variably-latency dependencies

However, pipeline is sometimes unused due to in-order computation execution

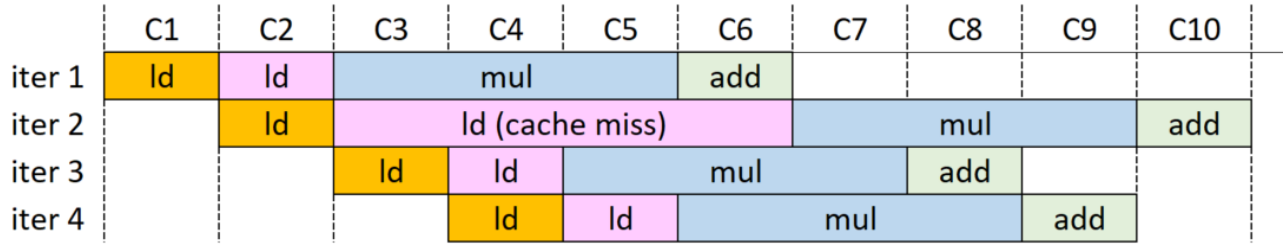
Superscalar Processors

- Fully **out-of-order pipelines**
  - Loop iterations are started speculatively
  - All operations execute **completely out-of-order**, only respecting dependencies



Out-of-order processors: perfect throughput if **perfect instruction supply and branch prediction**

- In HLS, associativity analysis **could even give a better result**
  - Yet, nobody went there in HLS in a general way, not even respecting dependence ordering



Out-of-order pipelines: maximal throughput exploiting further reordering opportunities

2

Memory and Caches

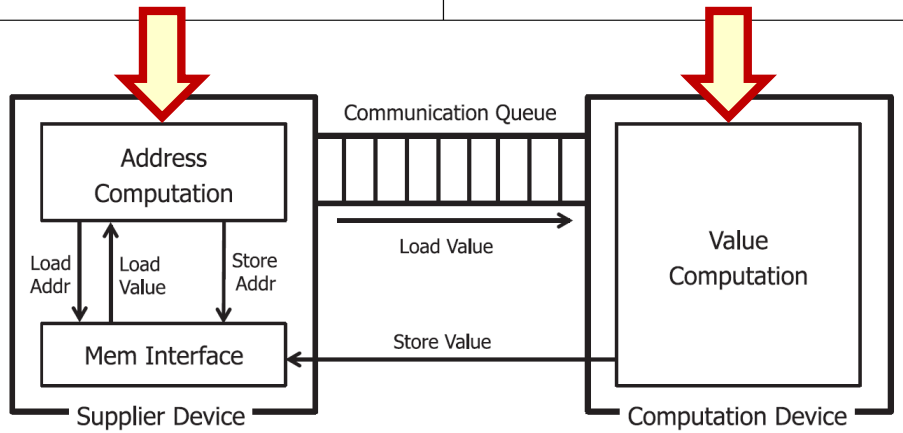
A key source of variable latency

Access/Execute Decoupling

- Separate variable-latency memory accesses and computation
  - Data is loaded from memory, stored in FIFOs, and sent to execution datapath as soon as ready
  - Requires nontrivial **code restructuring** (by user or compiler)

Original	AP slice	EP slice

`for (i=0; i<100; i++) {`
`  v1 = LOAD(&a[i]);`
`  v2 = LOAD(&b[i]);`
`  val = v1 + v2 * k;`
`  STORE(&c[i], val);`
`}`
	`for (i=0; i<100; i++) {`
	`  v1 = LOAD(&a[i]);`
	`  PRODUCE(v1);`
	`  v2 = LOAD(&b[i]);`
	`  PRODUCE(v2);`
	`  STORE_ADDR(& c[i]);`
	`}`
		`for (i=0; i<100; i++) {`
		`  v1 = CONSUME();`
		`  v2 = CONSUME();`
		`  val = v1 + v2 * k;`
		`  STORE_VAL(val);`
		`}`



Source: T. Ham, J. L. Aragón, and M. Martonosi, ACM TACO, June 2017

Access/Execute Decoupling

- Separate variable-latency memory accesses and computation
  - Data is loaded from memory, stored in FIFOs, and sent to execution datapath as soon as ready
  - Requires nontrivial **code restructuring** (by user or compiler)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  #pragma HLS dataflow
  for (c = s; c < e; c++) {
    cid = col[c];
    fifo_vec.write(vec[cid]);

    vec = fifo_vec.read();
    tmp += val[c] * vec;
  }

  out[i] = tmp;
}
```

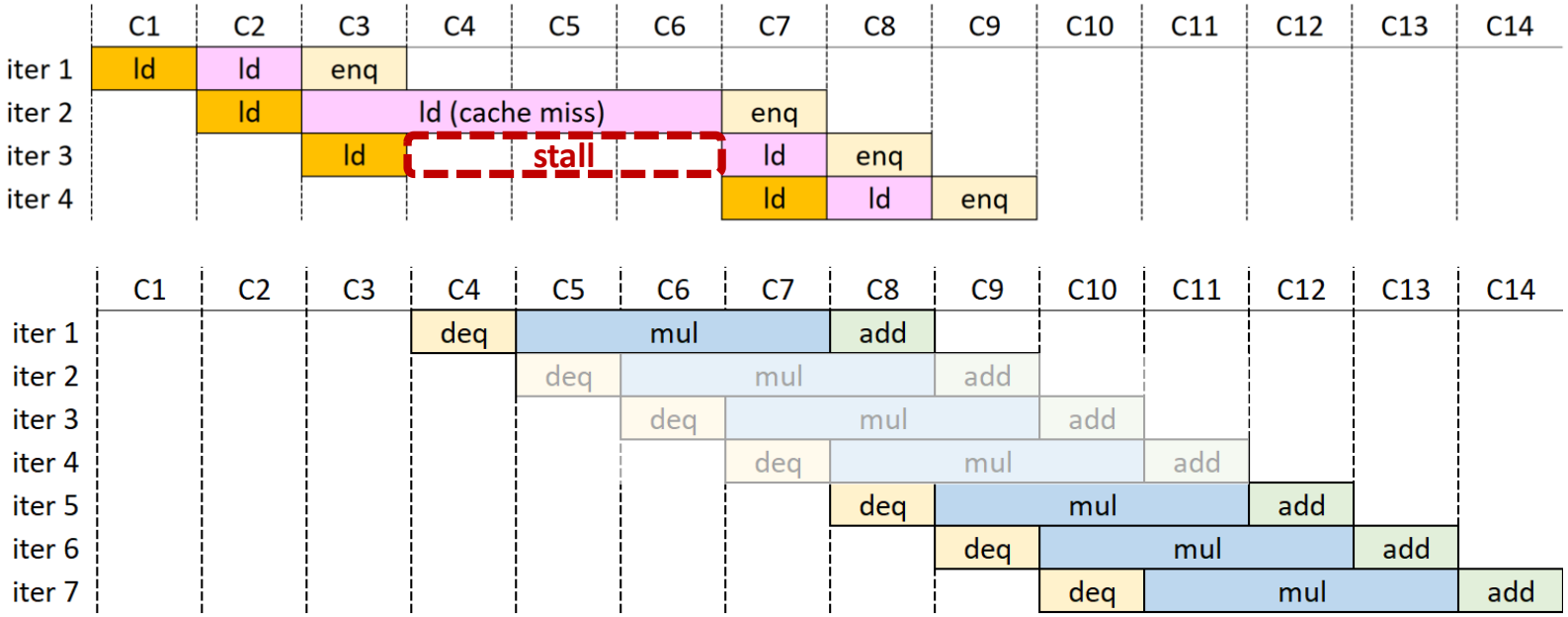
Using the **dataflow pragma** and A/E decoupling in VivadoHLS

Access/Execute Decoupling

- Separate variable-latency memory accesses and computation
  - Data is loaded from memory, stored in FIFOs, and sent to execution datapath as soon as ready
  - Requires nontrivial **code restructuring** (by user or compiler)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable-latency memory access



In-order access/execute decoupling

Execution similar to that of a **dynamic schedule** but composing **static building blocks**

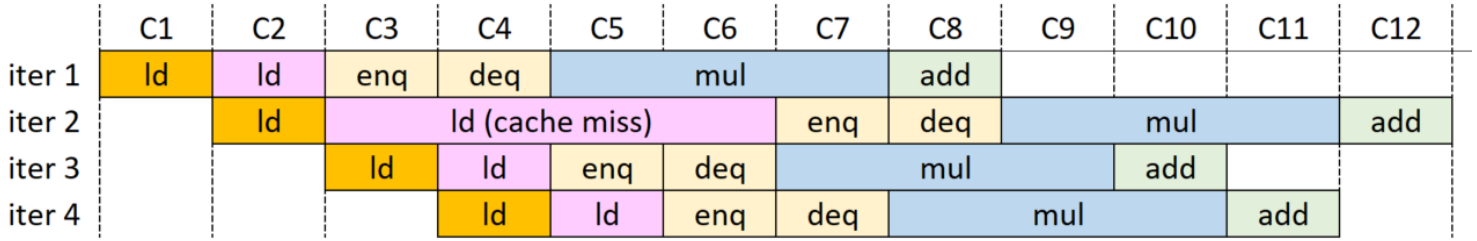
But in-order execution still prevents full datapath utilization

Out-of-Order Access/Execute Decoupling

- Separate variable-latency memory accesses and computation
  - Allow **out-of-order dequeuing**
  - Requires **even less trivial code restructuring** (by user or compiler)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable-latency memory access



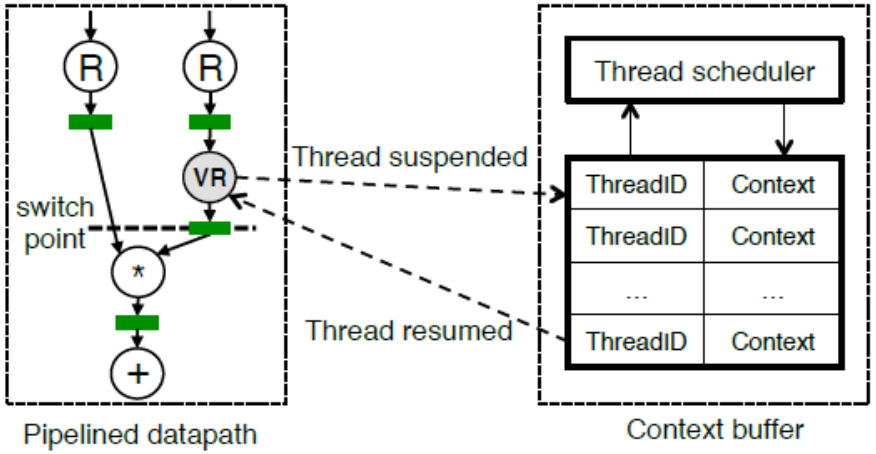
Out-of-order access/execute decoupling: maximal throughput

Latency of some iterations can be **completely hidden**
→ depends on distribution of short/long iterations

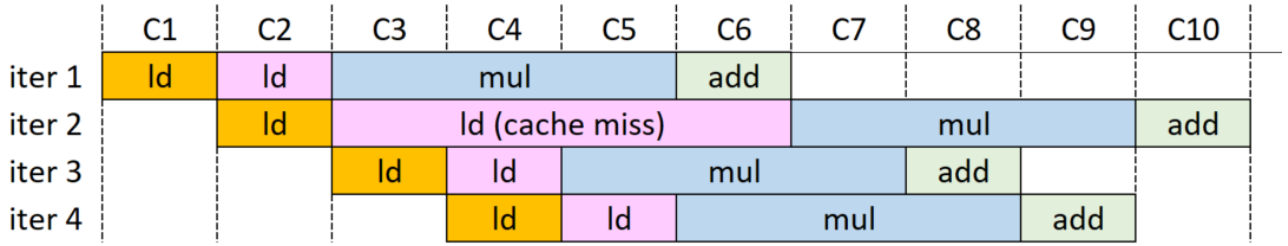
Effective II can be 1 even in the presence of cache misses
→ unlike previous solutions, a cache miss does not necessarily lower the II
(other iterations can proceed out-of-order)

Locally Out-of-Order Pipelines

- "Multithreaded" Pipeline Synthesis
  - Only **locally dynamically scheduled** and **locally out-of-order**,
  - Specific operations are suspended (i.e., context saved) and can execute out-of-order
  - Suspended operations releases resources, subsequent ones can continue without stalling
  - No general tagging and no reordering thanks to associativity and commutativity



Iterations can reorder inside the pipeline and a context buffer keeps track of the order and the context (i.e., live values of the thread)



Locally out-of-order pipelines: maximal throughput

Source: M. Tan, B. Liu, S. Dai, and Z. Zhang, ICCAD, 2014

3

Nested Loops

When variable latency becomes truly painful

Variable Loop Bounds

- In addition to operations, the **number of loop iterations** can also be variable
  - E.g., loop bounds computed at runtime, early exit condition,…
  - As in the case of variable-latency operations, not trivial to handle with standard HLS

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable loop bounds

Now considering the whole code

Variable Loop Bounds

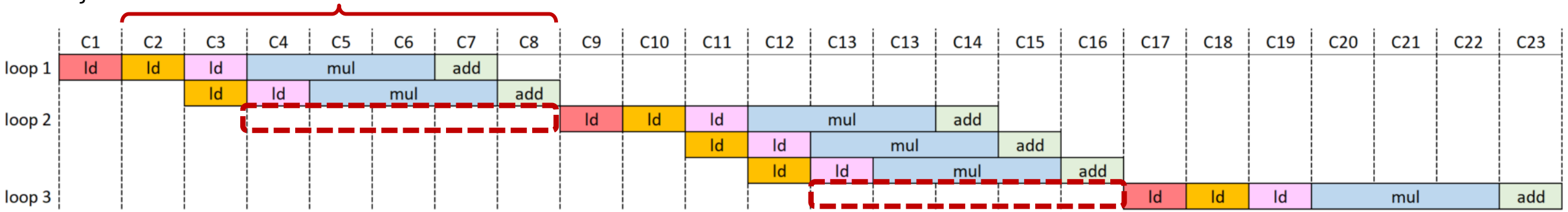
- Static HLS: start a new inner loop **only when previous one completes**
  - When all loop bounds are known, loops can be analysed statically and flattened
  - In general case: exit inner loop, compute new loop bounds, enter inner loop

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable loop bounds

Inner-loop pipeline has to empty before another inner loop can start: pipeline empty for inner loop iteration latency + loop bound computation latency

Average II lowered during loop transitions

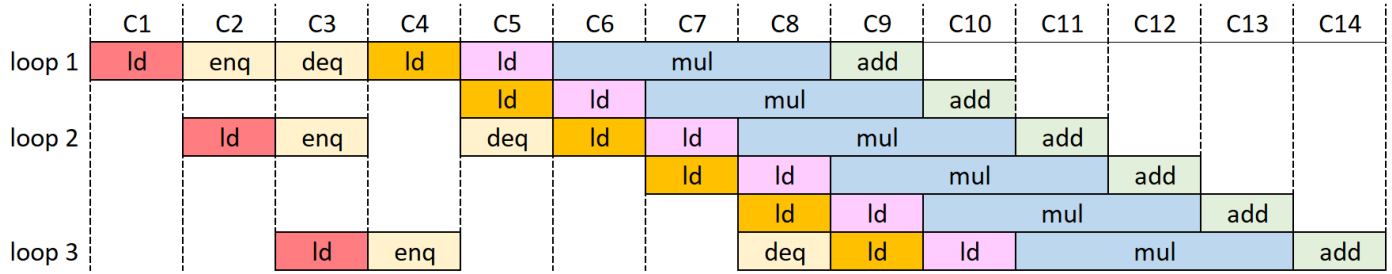


Access/Execute Decoupling

- A/E decoupling works here, but the transformation is even more complex:
  - Compute loop bounds and enqueue into FIFO
  - Dequeue bounds and execute inner loop

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable loop bounds



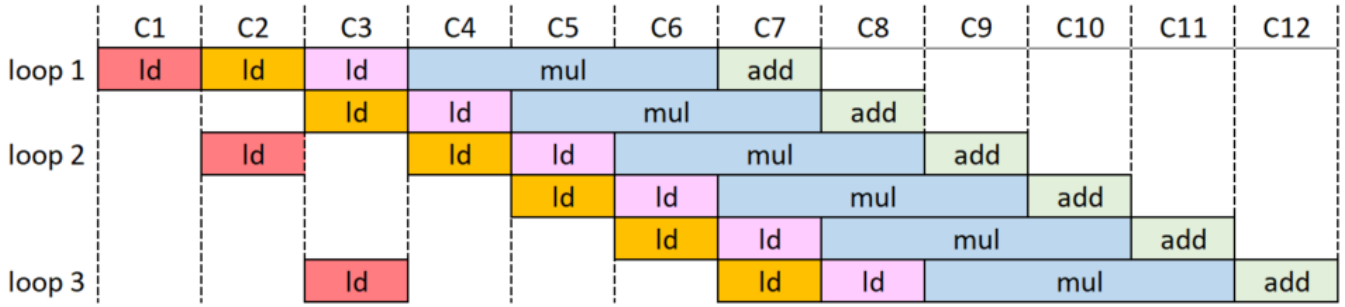
II = 1 (assuming FIFOs are appropriately sized)

Dataflow

- **Dynamic HLS**: naturally starts a new loop **as soon as pipeline is ready**
  - No special mechanism or transformation required

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Variable loop bounds



II = 1 (no transformations needed, FIFOs placement automatic)

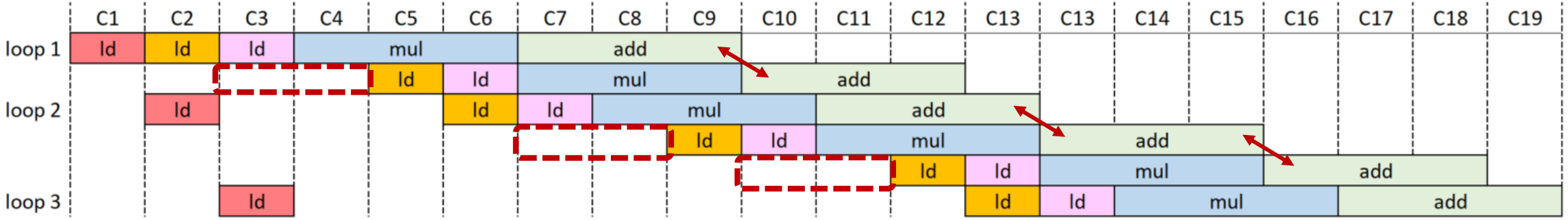
Loop-Carried Dependencies

- **Dynamic HLS**: in-order pipelines
  - Different operations execute **out of order** with respect to each other, but each operation processes its own data **in order**
  - Limited throughput in case of **long-latency loop-carried dependencies**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

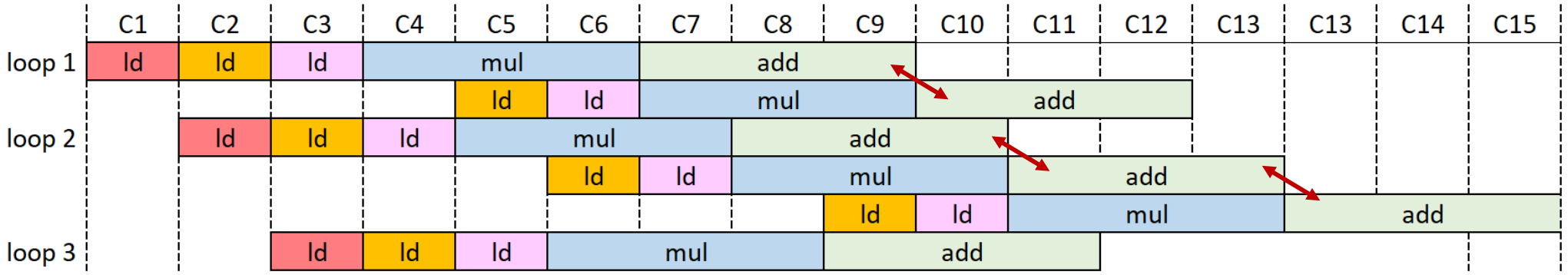
Long latency adder

In-order dataflow execution: the long-latency addition limits the II of each inner loop to II = 3. A new inner loop can start at best one cycle after the last iteration of the previous loop



Superscalar Processors

- Fully **out-of-order pipelines**
  - All loop iterations (outer and inner) are started speculatively
  - All operations execute **completely out-of-order**, only respecting dependencies



Out-of-order processors: again perfect throughput if **perfect instruction supply and branch prediction**

4

Multiple Pipelines

Exploiting parallelism beyond ILP

Spatial Parallelism

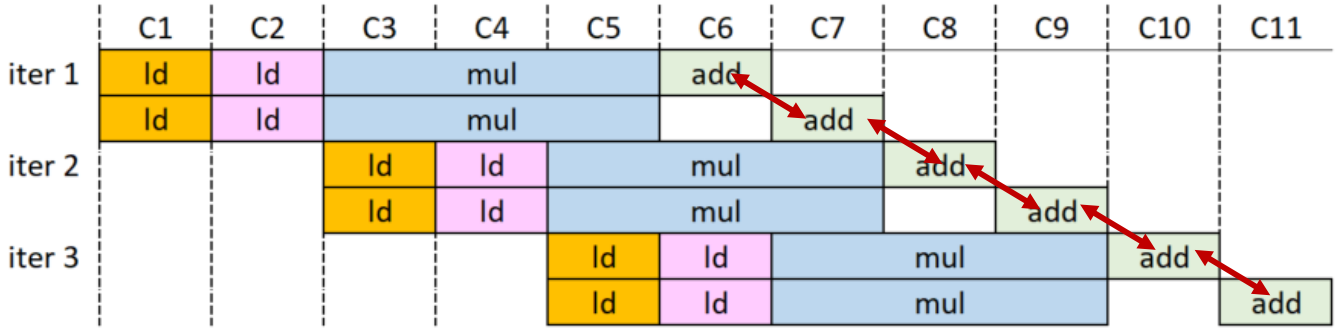
- So far, instruction-level parallelism within a **single datapath**
- **Replicate datapath/kernel** to increase parallelism
- Challenges:
  - How to express the parallelism to the compiler?
  - How to maximize utilization of each datapath?

Loop Unrolling

- **Standard unrolling:** replicate computations within a loop
  - Achieves spatial parallelism in regular loops
  - Not suitable for irregular code (e.g., unknown loop bounds, memory/data dependencies)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
  #pragma HLS unroll factor=2
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

Unrolling with an **HLS pragma**



Does not increase performance due to loop-carried dependency!
Exactly the same performance as a perfect pipeline (effective II = 1)

Task Parallelism (Loop Replication)

- Replicate loops and execute **multiple loop instances** in parallel
  - Difficult to express using a C-based HLS tool (code restructuring and pragmas)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

```
#pragma HLS dataflow
for (i = 0; i < num_rows, i+=2) {
  tmp1 = 0; tmp2 = 0;
  fifo_s1.write(row[i]); fifo_e1.write(row[i+1]);
  fifo_s2.write(row[i+1]); fifo_e2.write(row[i+2]);

  s1=fifo_s1.read(); e1=fifo_e1.read();

  for (c = s1; c < e1; c++) {
    cid1 = col[c];
    tmp1 += val[c] * vec[cid];
  }
  out[i] = tmp1;

  s2=fifo_s2.read(); e2=fifo_e2.read();

  for (c = s2; c < e2; c++) {
    cid2 = col[c];
    tmp2 += val[c] * vec[cid];
  }
  out[i+1] = tmp2;
}
```

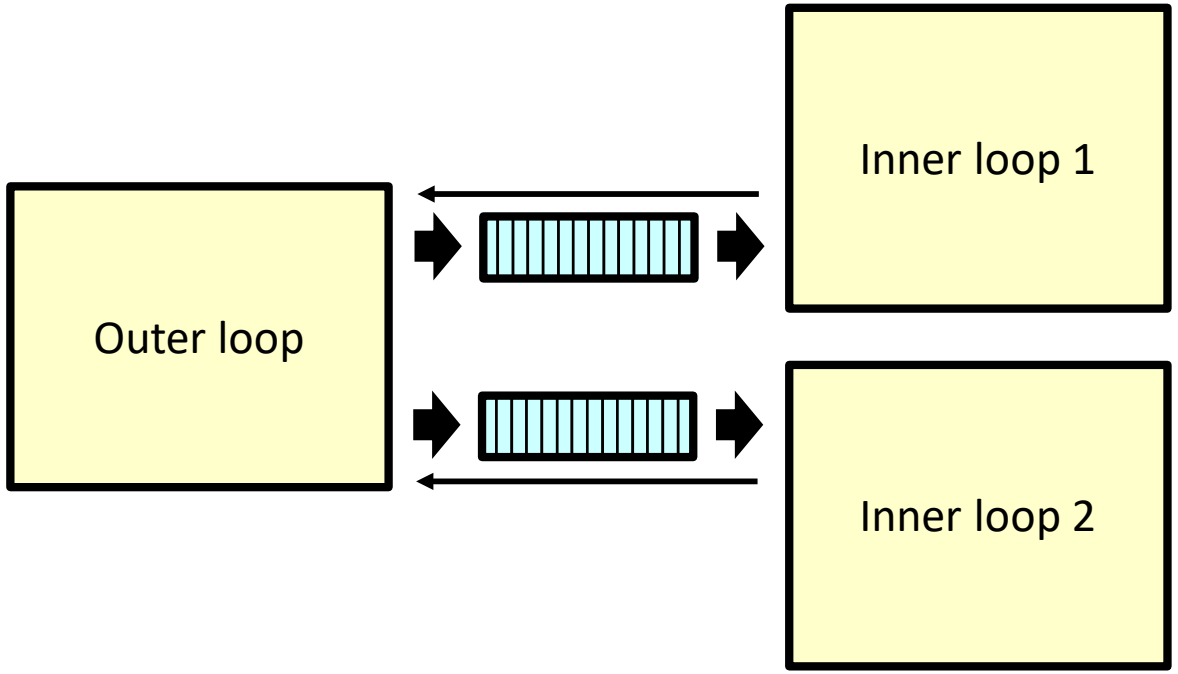
Looks and feels like VivadoHLS code but almost certainly does not work!

Using the dataflow pragma and A/E decoupling to parallelize loops

Task Parallelism (Loop Replication)

- Replicate loops and execute **multiple loop instances** in parallel
  - Difficult to express using a C-based HLS tool (code restructuring and pragmas)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```



Multithreading

- C-based HLS tools require a **structural description** of the pipeline connectivity
- Yet, **fork-join** schemes express the same in **software implementations**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

```
for (i = 0; i < num_rows, i+=2) {
  s = row[i]; e = row[i+1];
  pthread_create(&th1, NULL, colvec_prod, s, e);

  s = row[i+1]; e = row[i+2];
  pthread_create(&th2, NULL, colvec_prod, s, e);

  pthread_join(th1, &tmp1);
  pthread_join(th2, &tmp2);

  out[i] = tmp1;
  out[i+1] = tmp2;
}
```

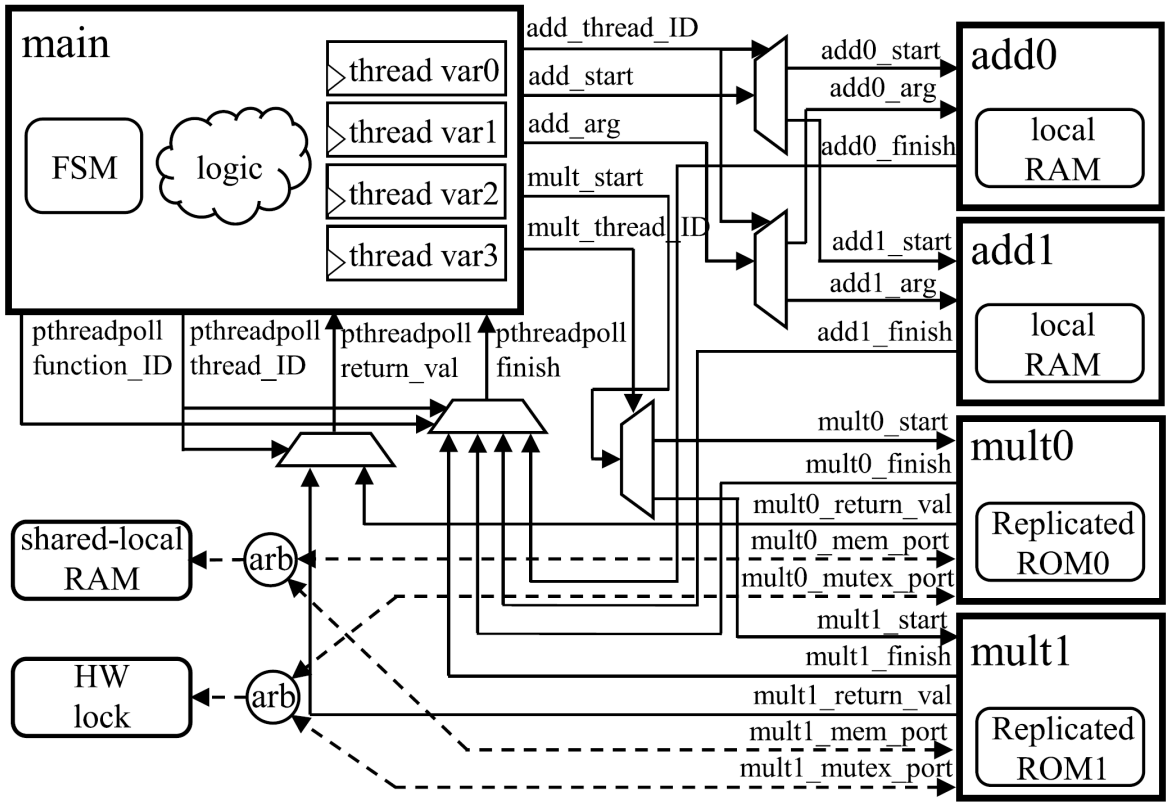
Inner loop in a function

Using a common **fork-join** scheme to parallelize loops

Multithreading

- Research work has shown **support for pThreads in HLS**
- Some (important) limitations: **number of tasks/threads known at compile time** (unadapted to recursive algorithms, etc.)

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

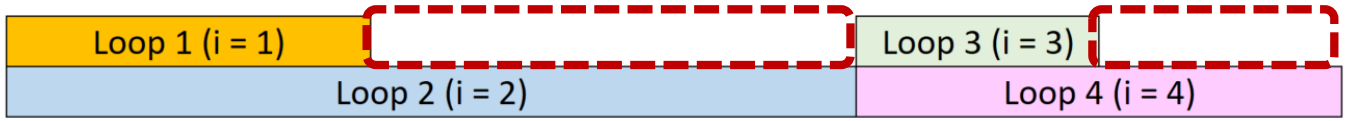


Source: J. Choi, S. D. Brown, J. H. Anderson, IEEE TVLSI, October 2017

Simple Multithreading

- Replicate loops and execute **multiple loop instances** in parallel
- Increases parallelism, but datapaths **not fully used if inner loop latencies differ**

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```



Datapath unused due to different loop latencies

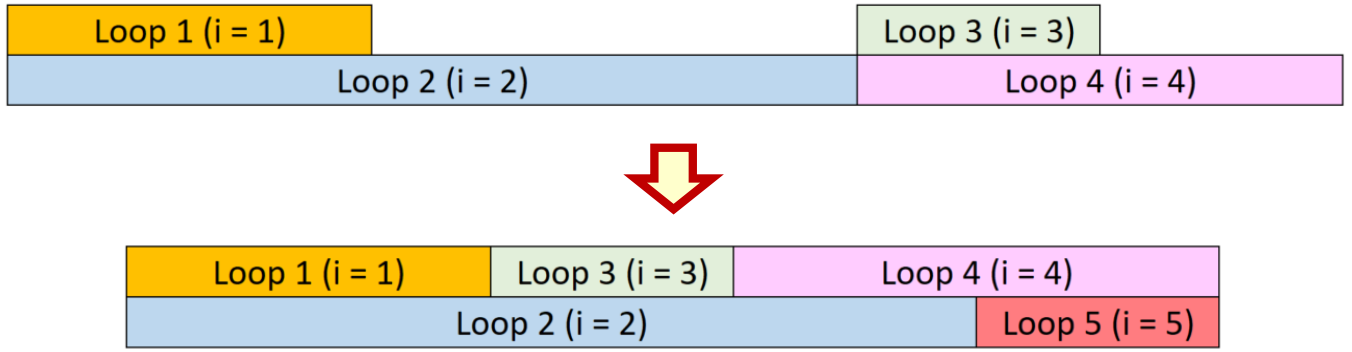
5

Advanced Fork-Join Schemes

Cilk, CilkPlus, Intel TBB, etc.

Multithreaded Runtime Systems

- **Old idea in HPC**: Cilk (MIT, 1994) and Cilk Plus (Intel, 2009), then Threading Building Blocks or TBB (Intel, 2000's)
- **Lightweight tasks** running over a fixed number of threads
- **Work-stealing schedulers** to load balance threads
- State of the art software paradigm for algorithms with **strongly dynamic behavior** (e.g., recursive algorithms)



Maximal datapath usage and throughput

Simple Automatic Partitioning

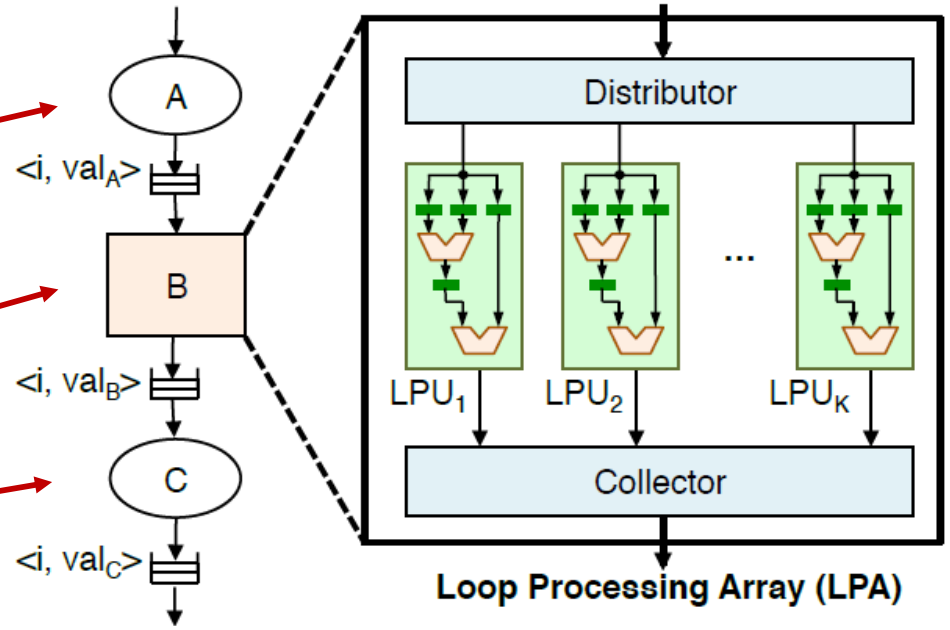
- Similar to A/E decoupling in **connecting static FSM through buffers**
- **Load balancing** to maximize parallelism and kernel utilization
  - Dispatch computation to processing units based on availability

```
for (i = 0; i < num_rows, i++) {
  tmp = 0;
  s = row[i]; e = row[i+1];

  for (c = s; c < e; c++) {
    cid = col[c];
    tmp += val[c] * vec[cid];
  }

  out[i] = tmp;
}
```

A

B

C



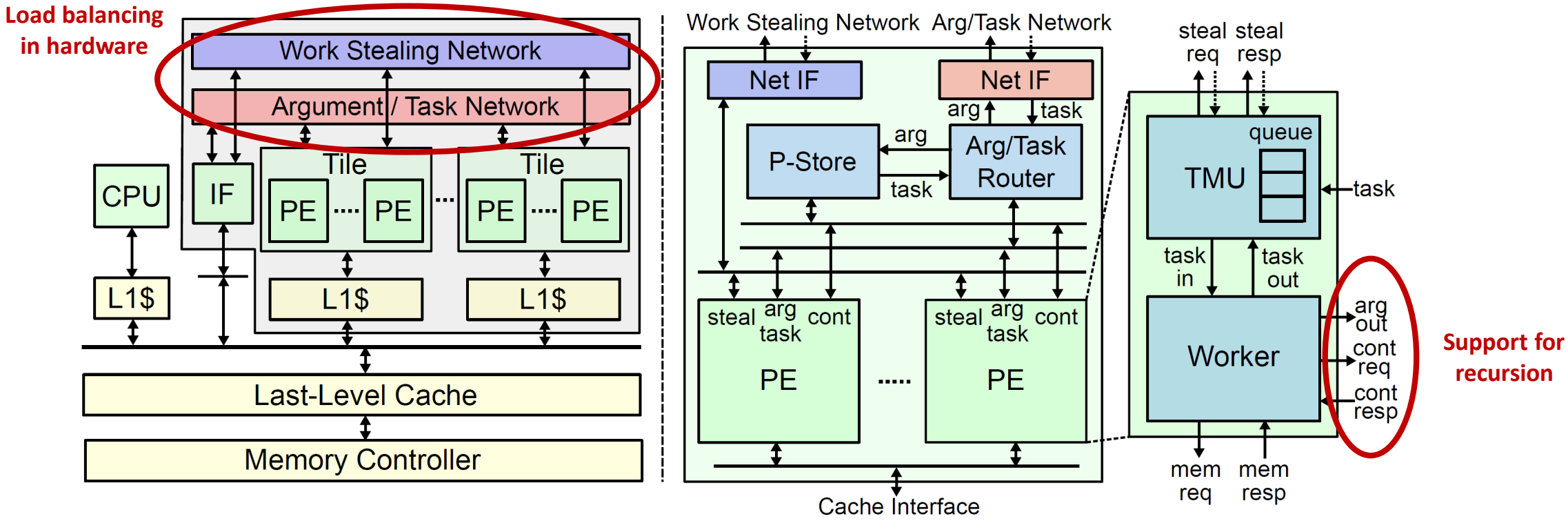
Simple case: no recursion possible

Each dynamic-bound inner loop is mapped to a loop processing array (LPA), which consists of multiple loop processing units (LPUs). The distributor contains a scheduler with a dynamic work distribution policy.

Source: M. Tan, B. Liu, R. Zhao, S. Dai, and Z. Zhang, ICCAD, 2015

Multithreaded Runtime Systems in Hardware

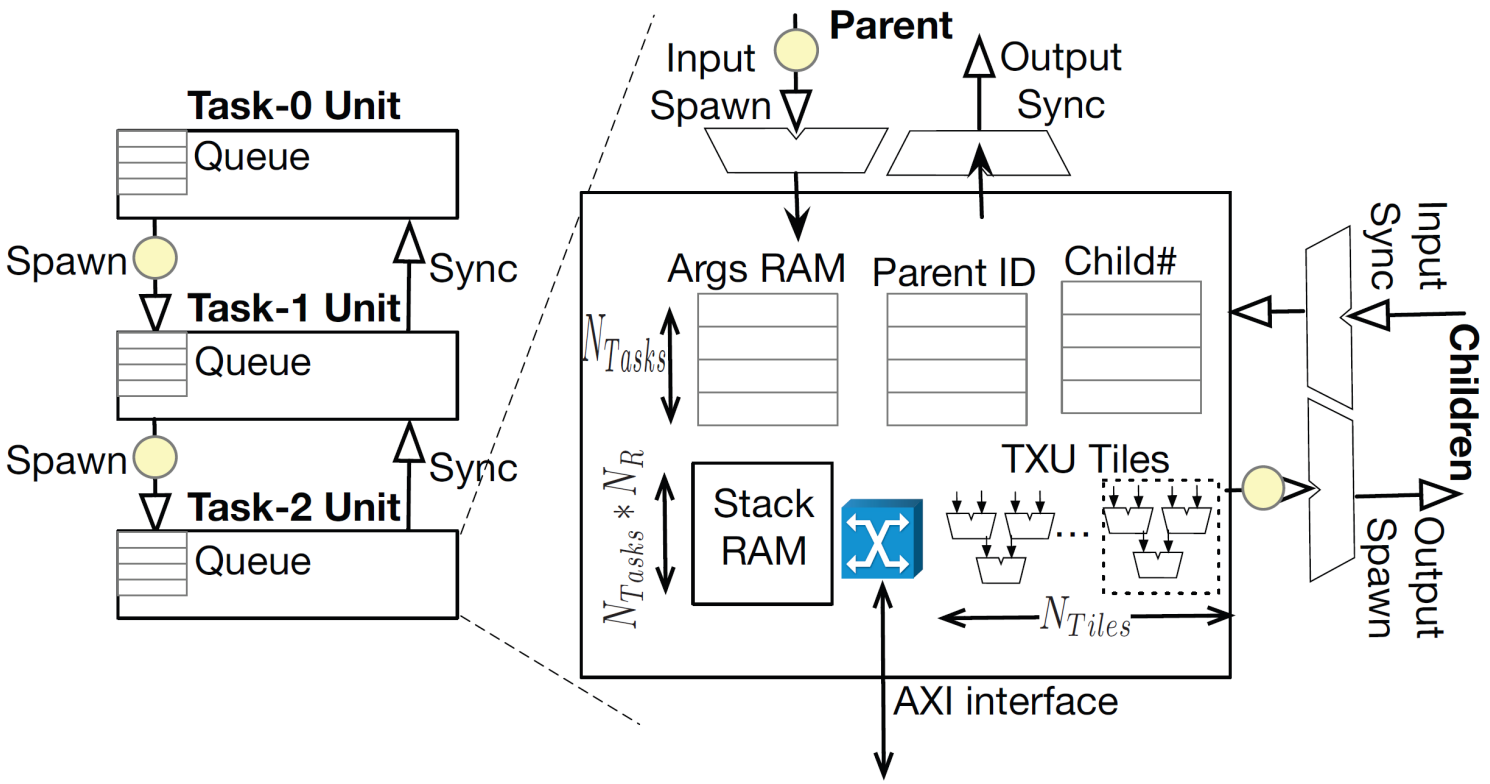
- Very few efforts to “imitate” Cilk and TBB in HLS (1/2): **ParallelXP**
  - Not quite Cilk/TBB nor really HLS
  - Exploits the continuation passing idea to create a **customizable runtime system** where users can plug in their own processing elements



Source: T. Chen, S. Srinath, C. Batten, and G. E. Suh, MICRO-51, 2018

Multithreaded Runtime Systems in Hardware

- Very few efforts to “imitate” Cilk and TBB in HLS (2/2): **TAPAS**
  - More of an HLS/compiler effort, based on a compiler front end supporting fork-join
  - Only **direct connection between task units** based on parent-child relations (multiple units for the same task? load balancing among them?)



Source: S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam, MICRO-51, 2018

6

Conclusions

HLS, software programmers, and accelerators

Takeaway

- High-Level Synthesis is a **fairly mature field** with a number of options available
- Of commercial interest **virtually only for FPGAs**; ASIC designers seem to stick almost exclusively to RTL because they want to get everything they can from technology
- Still, it is **not really meant to** (nor in fact does) **give access to FPGAs for software programmers**; most of the available options speed-up development for hardware engineers who more or less know what they want
- Only **very slow movement towards** software support for **irregular parallelism** (the one which might interest software programmers attracted by FPGAs).

Caveat #1

The accelerator design must compensate for the FPGA reconfigurability

- Processors exploit amazingly well transistors in a given technology
- GPUs do that too and also extract a huge lot of parallelism—when the application fits the architecture
- ASICs are for the lucky few, but if an application is sufficiently important (e.g., Google's TPUs), they are unbeatable, almost by definition
- On FPGAs, one must fight the overhead of reconfigurability (one order of magnitude slower and bigger?) before one can gain!
- Maybe FPGAs are not the right reconfigurable platform for accelerators…

Caveat #2

"It's the memory, stupid!"

Dick Sites, 1996

- HLS is getting better at designing the computational part of accelerators
- Most of the performance depends on moving data efficiently
- Even the simplest aspects of this are hard or hopeless for HLS compilers (memory disambiguation, etc.)
- Few tools to help designing application-specific memory systems
- Expect to plan data movement by hand—and to code it in RTL…

Caveat #3

HLS may give you great kernels but does not give you full accelerators

- HLS tools are ok with fine grain parallelism (akin to ILP) but not more
- HLS tools have embraced some languages for specific computational patterns (CUDA, OpenCL,…)—but are GPUs not better when you can use these languages proficiently?
- Very limited efforts and progress in adopting programming models where FPGA acceleration might truly excel
- Manual design in RTL (with ad-hoc use of HLS, perhaps) seems the only way of achieving truly competitive accelerators today

References

- M. Tan, B. Liu, S. Dai, and Z. Zhang, *Multithreaded pipeline synthesis for data-parallel kernels*, ICCAD, November 2014
- T. Ham, J. L. Aragón, and M. Martonosi, *Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures*, ACM TACO, June 2017
- T. Chen and G. E. Suh, *Efficient data supply for hardware accelerators with prefetching and access/execute decoupling*, MICRO-49, October 2016
- J. Choi, S. D. Brown, J. H. Anderson, *From Pthreads to Multicore Hardware Systems in LegUp High-Level Synthesis for FPGAs*, IEEE TVLSI, October 2017
- M. Tan, B. Liu, R. Zhao, S. Dai, and Z. Zhang, *ElasticFlow: A complexity-effective approach for pipelining irregular loop nests*, ICCAD, November 2015
- R. Halstead and W. Najjar, *Compiled multithreaded data paths on FPGAs for dynamic workloads*, CASES, September 2013
- T. Chen, S. Srinath, C. Batten, and G. E. Suh, *An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware*, MICRO-51, October 2018
- S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam, *TAPAS: generating parallel accelerators from parallel programs*, MICRO-51, October 2018