Generating Configurable Hardware From Parallel Patterns

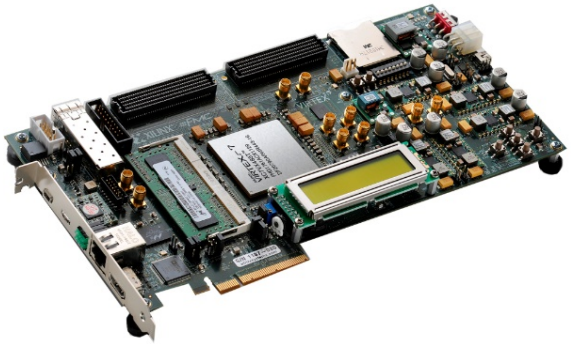
Raghu Prabhakar, David Koeplinger, Kevin J. Brown,
HyoukJoong Lee, Chris De Sa, Christos Kozyrakis, Kunle Olukotun

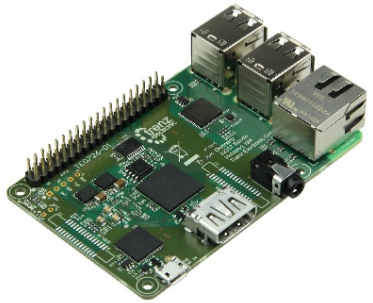
Stanford University

ASPLOS 2016

Motivation

- Increasing interest to use FPGAs as accelerators
  - Key advantage: Performance/Watt



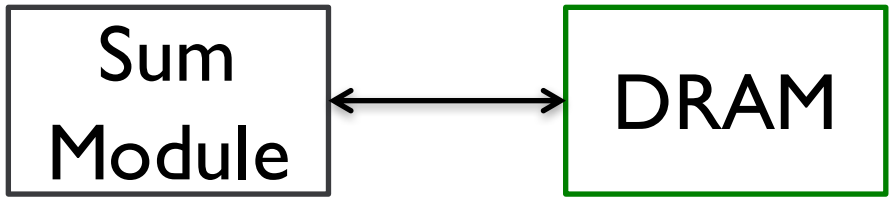


- Key domains:
  - Big data analytics, image processing, financial analytics, scientific computing, search

Problem: Programmability

- Verilog and VHDL too low level for software developers
- High level synthesis (HLS) tools need user pragmas to help discover parallelism
  - C-based input, pragmas requiring hardware knowledge
  - Limited in exploiting data locality
  - Difficult to synthesize complex data paths with nested parallelism

Hardware Design - HLS



Add 512 integers stored in external DRAM

```
void(int*  mem) {

    mem[512]  = 0;

    for(int  i=0;  i<512;  i++) {
        mem[512]  +=  mem[i];
    }
}
```

27,236 clock cycles for computation

Two-orders of magnitude too long!

Optimized Design - HLS

```
#define CHUNKSIZE (sizeof(MPort)/sizeof(int))
#define LOOPCOUNT (512/CHUNKSIZE)

void(MPort* mem) {
    MPort buff[LOOPCOUNT];
    memcpy(buff, mem, LOOPCOUNT);

    int sum = 0;
    for(int i=1; i<LOOPCOUNT; i++) {
        #pragma PIPELINE
        for(int j=0; j<CHUNKSIZE; j++) {
            #pragma UNROLL
            sum += (int)(buff[i]>>j*sizeof(int)*8);
        }
    }
    mem[512] = sum;
}
```

Width of DRAM controller interface

Burst Access

Use local variable

Loop Restructuring

Special compiler directives

Bit shifting to extract individual elements

302 clock cycles for computation

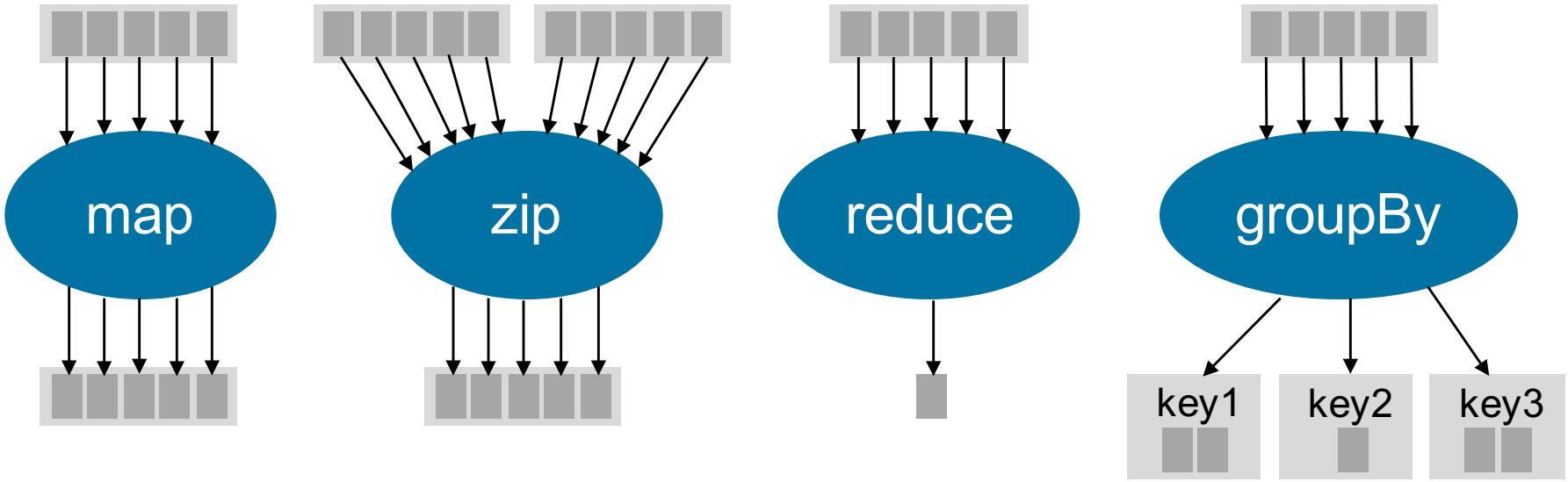
So, we need to ...

- Use ***Higher-level Abstractions***
  - Productivity: Developer focuses on application
  - Performance:
    - ***Capture Locality*** to reduce off-chip memory traffic
    - ***Exploit Parallelism*** at multiple nesting levels

- Smart compiler generates efficient hardware

Parallel Patterns

- Constructs with special properties with respect to parallelism and memory access



Why Parallel Patterns?

- Concise
- Can express large class of workloads in the machine learning and data analytics domain
- Captures rich semantic information about parallelism and memory access patterns
- Enables powerful transformations using pattern matching and re-write rules
- Enables generating efficient code for different architectures

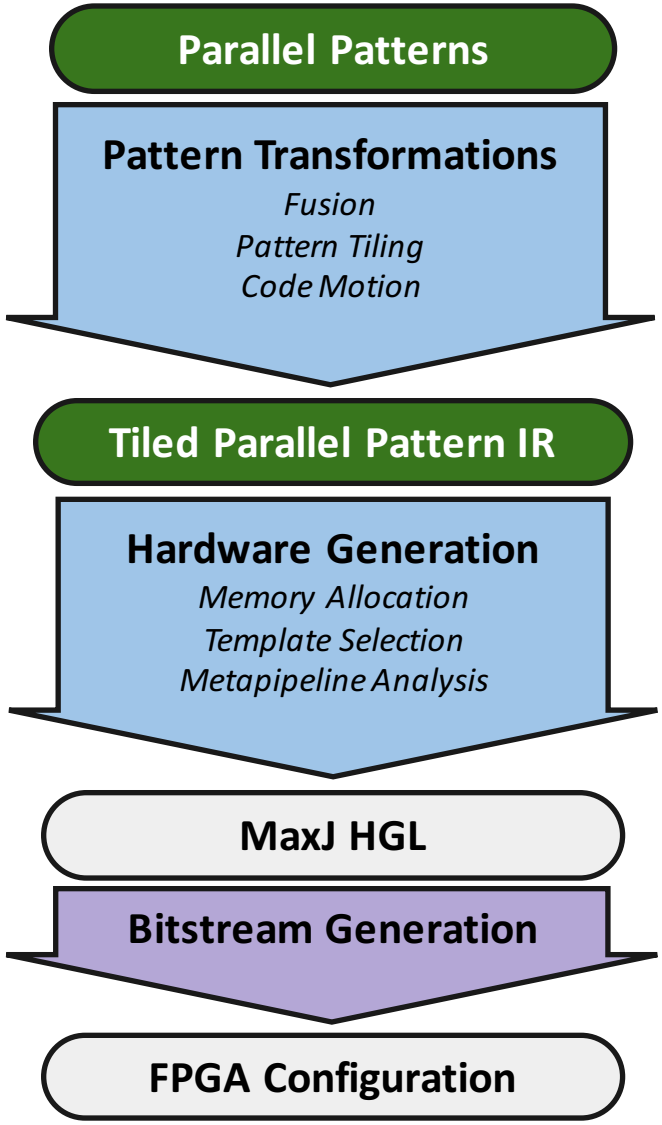
Parallel Pattern Language

- A data-parallel language that supports parallel patterns
- Example application: *k*-means

```
val clusters = samples groupBy { sample =>
  val dists = kMeans map { mean =>
    mean.zip(sample){ (a,b) => sq(a - b) } reduce { (a,b) => a + b }
  }
  Range(0, dists.length) reduce { (i,j) =>
    if (dists(i) < dists(j)) i else j
  }
}
val newKmeans = clusters map { e =>
  val sum = e.reduce { (v1,v2) => v1.zip(v2){ (a,b) => a + b } }
  val count = e.map { v => 1 } reduce { (a,b) => a + b }
  sum.map { d => d / count }
}
```

```
// Compute closest mean for each 'sample'
// 1. Compute distance with each mean
// 2. Select the mean with shortest distance
// Compute average of each cluster
// 1. Compute sum of all assigned points
// 2. Compute number of assigned points
// 3. Divide each dimension of sum by count
```

Our Approach



Our Approach

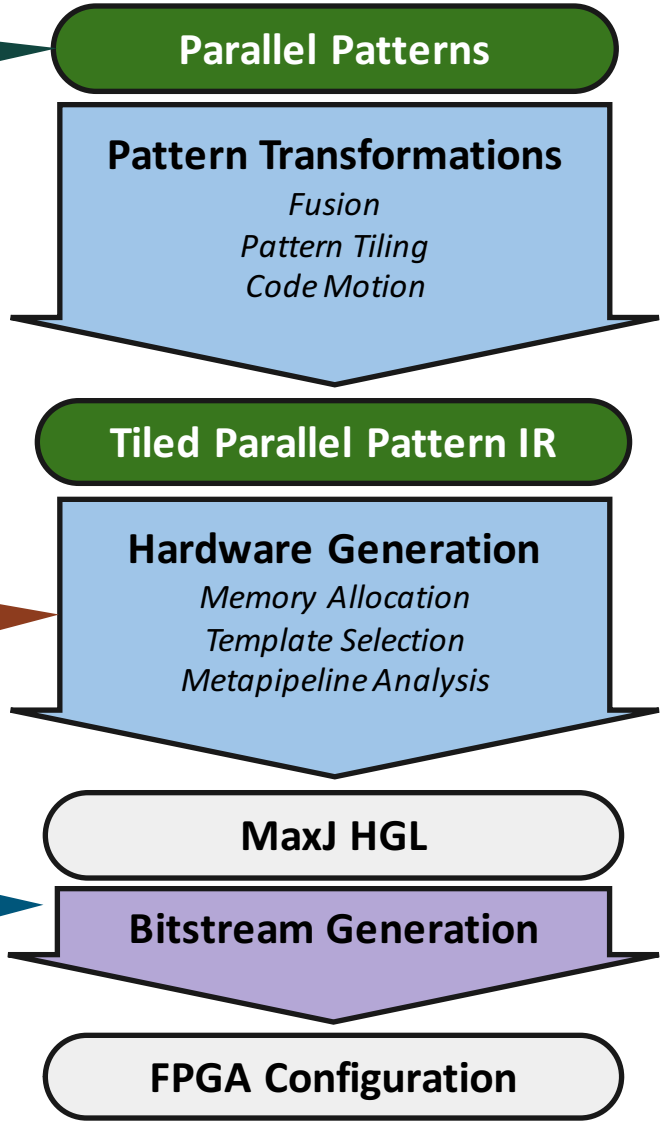
High-level Parallel Patterns helps productivity

Delite

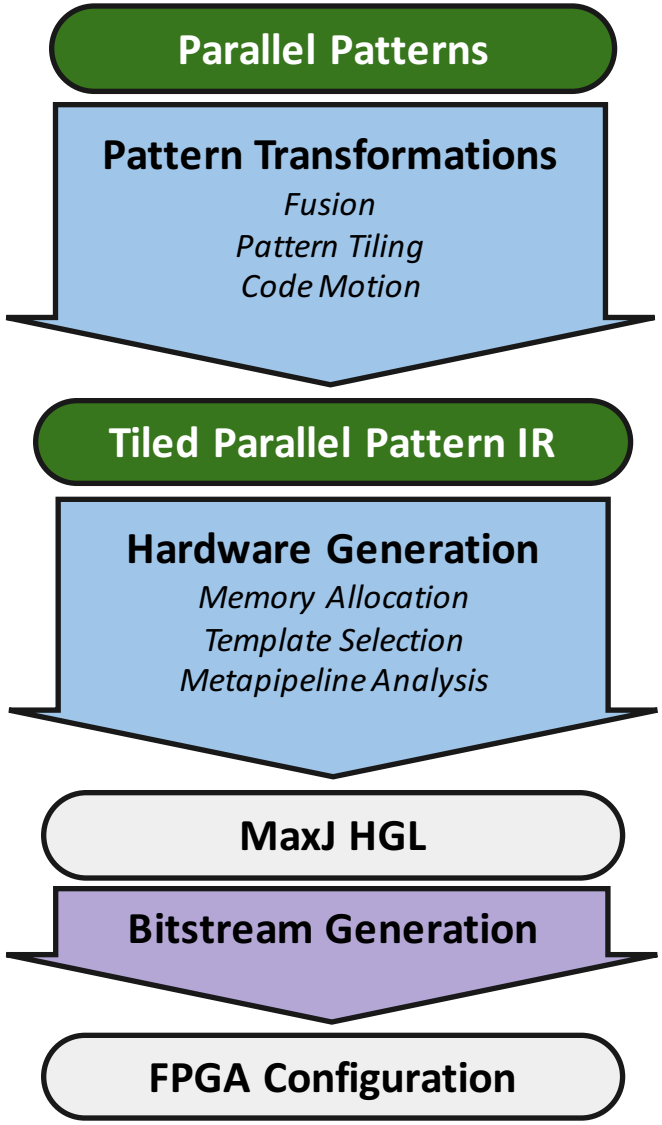
Data Locality improved with **parallel pattern tiling transformations**

Nested Parallelism exploited with **hierarchical pipelines** and **double buffers**

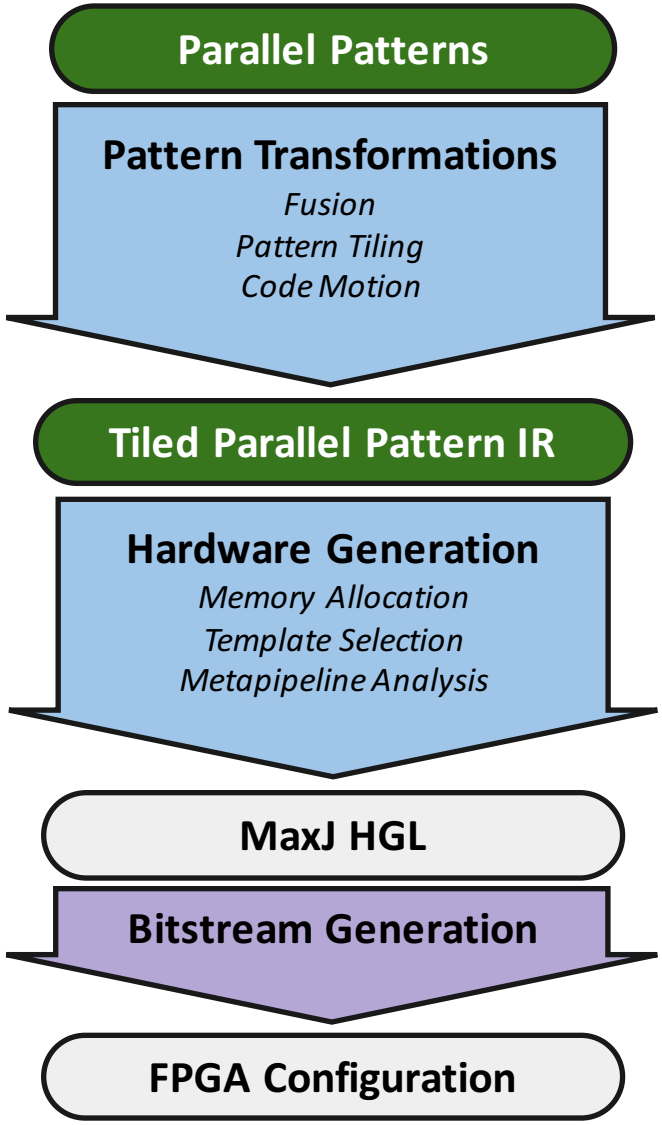
Generate MaxJ to **generate VHDL**



Our Approach

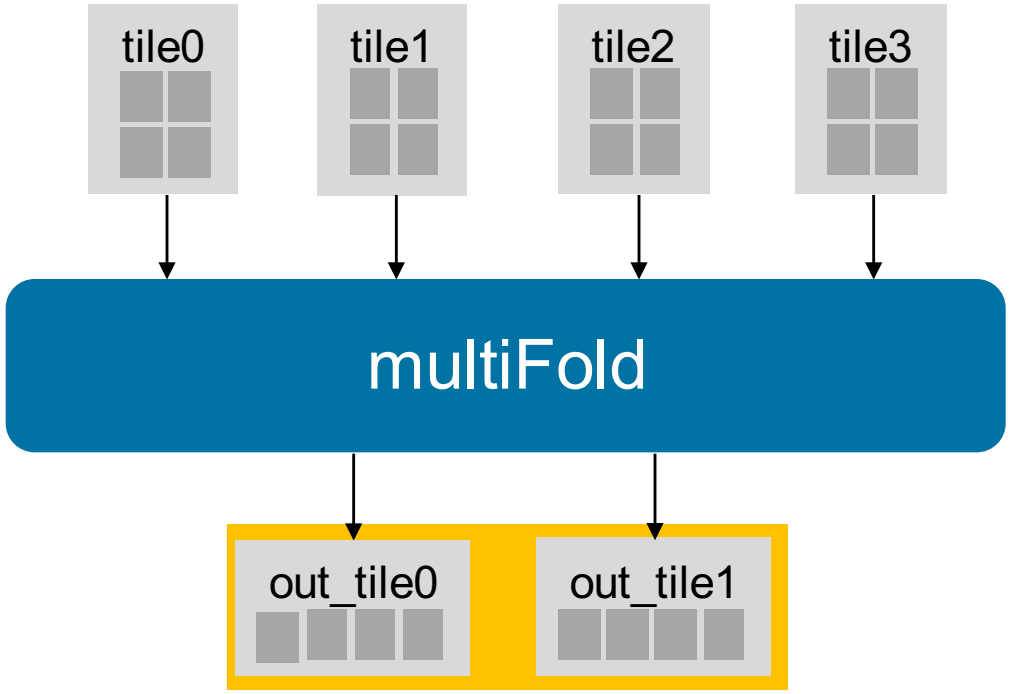


Our Approach

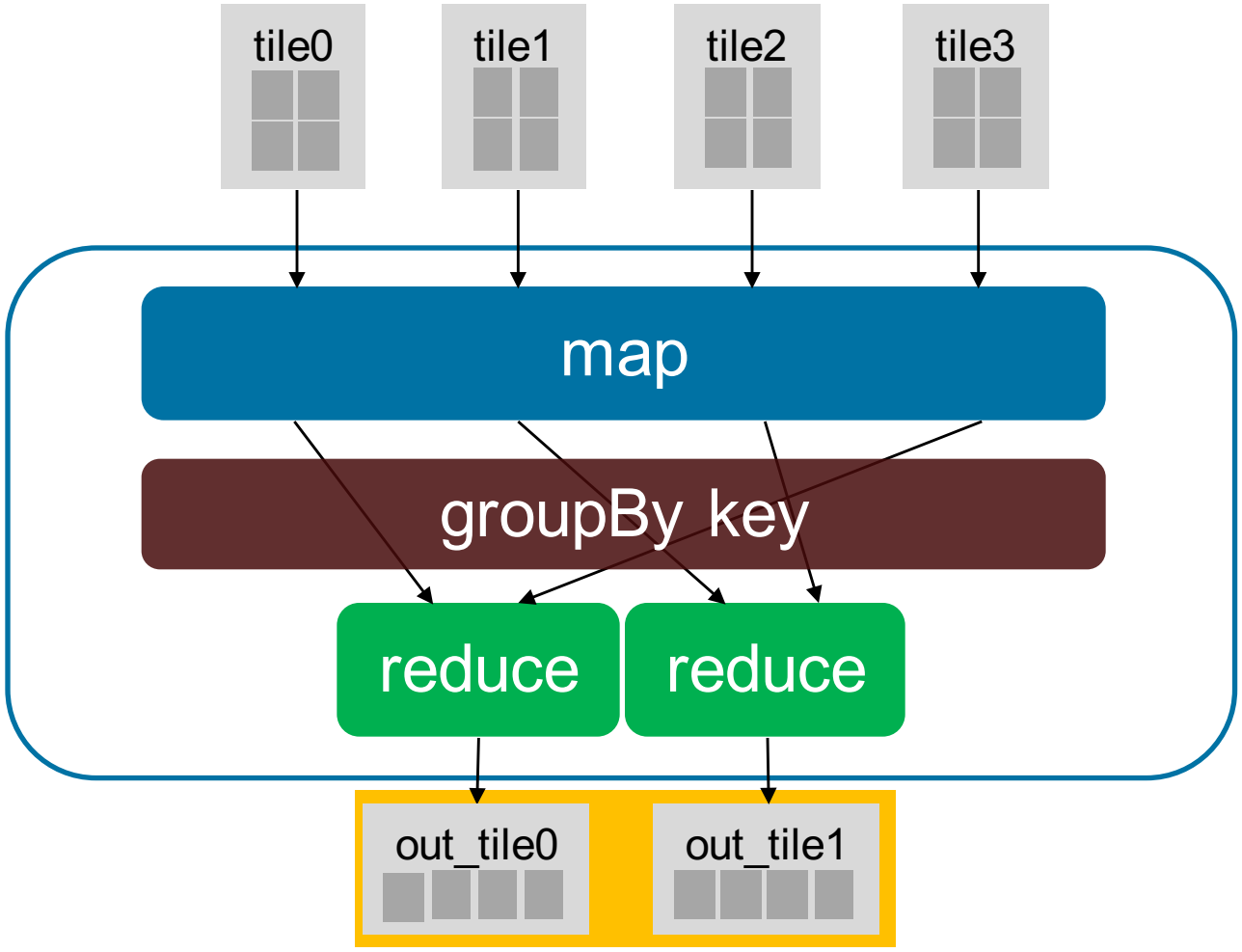


Parallel Pattern Tiling: MultiFold

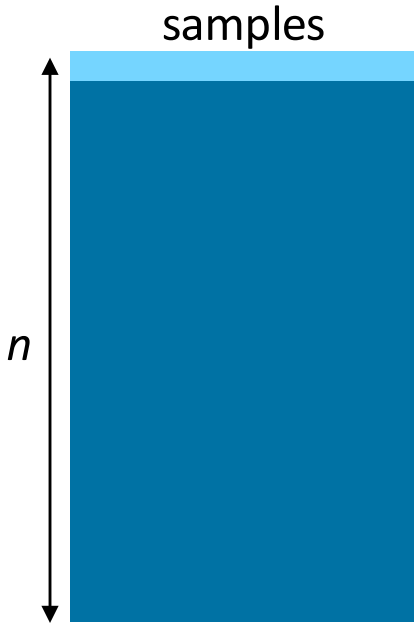
- Tiling using polyhedral analysis limits data access patterns to affine functions of loop indices
- Current parallel patterns cannot represent tiling
- New parallel pattern describes tiled computation

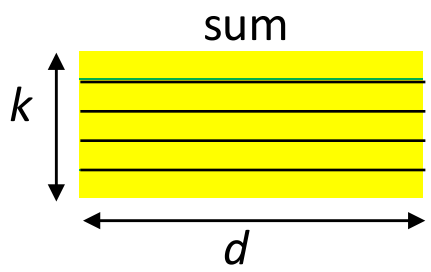


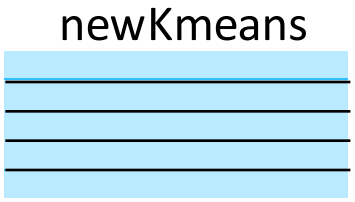
Parallel Pattern Tiling: MultiFold



kMeans: Untiled

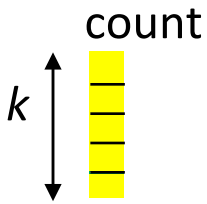




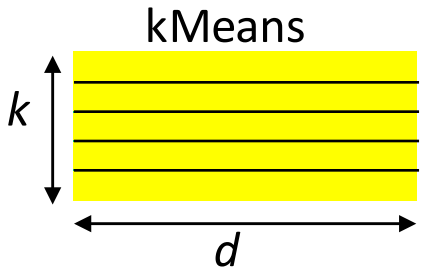


mindist

minDistIdx



- Data dependent (non-affine) access to 'sum' and 'count'
- Lots of data locality
- Typically, *n* >> *k*



kMeans #reads: ***n * k * d***

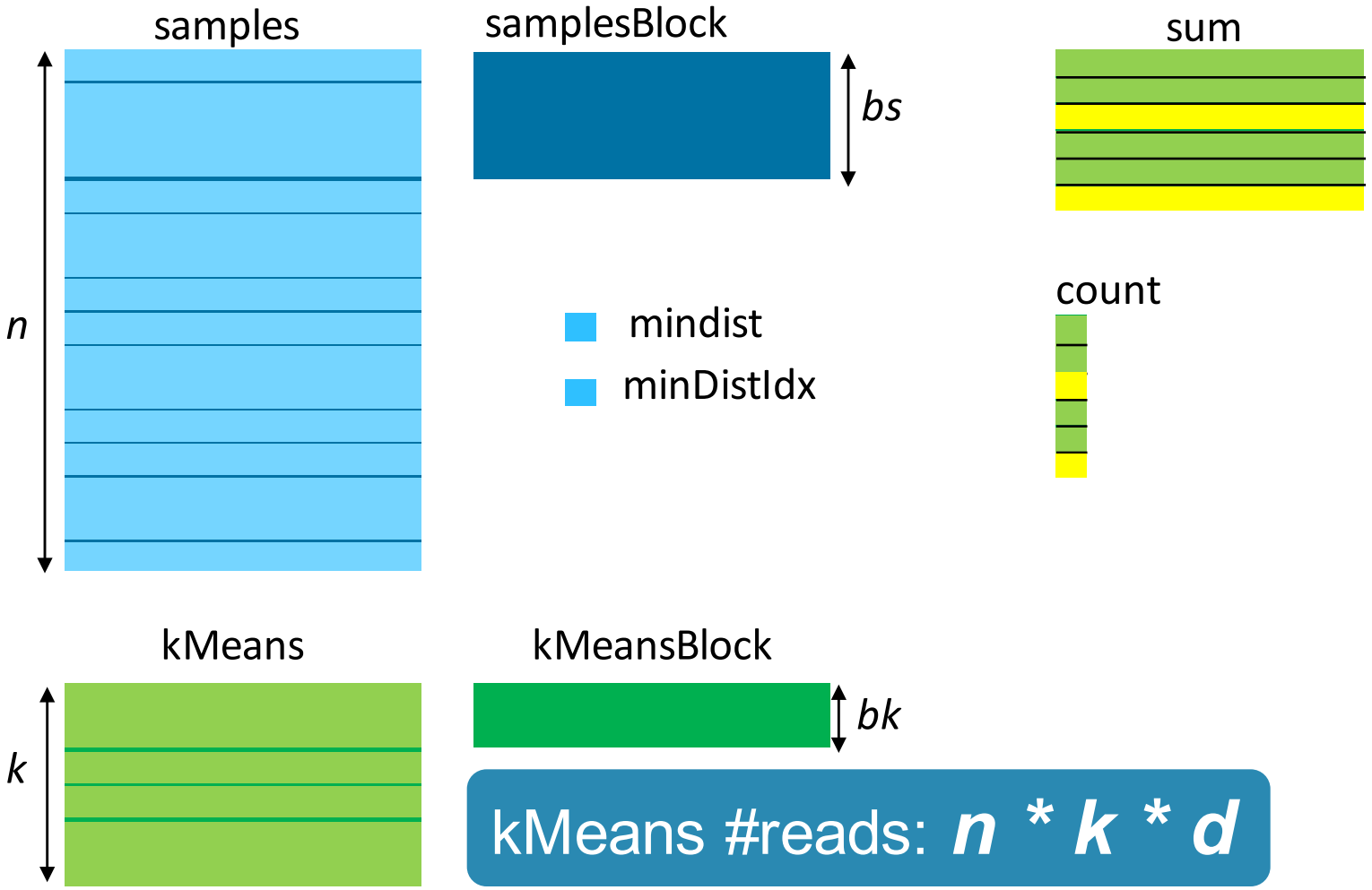
Parallel Pattern Strip Mining

- Transform parallel pattern → nested patterns
  - Strip mined patterns enable computation reordering
- Insert copies to enhance locality
  - Copies guide creation of on-chip buffers

Parallel Patterns	Strip Mined Patterns

`map(d){i => 2*x(i)}`	`multiFold(d/b){ii =>`
	`  xTile = x.copy(b + ii)`
	`  (i, map(b){i => 2*xTile(i)`
	`}) }`

Strip Mining: kMeans



kMeans #reads: ***n * k * d***

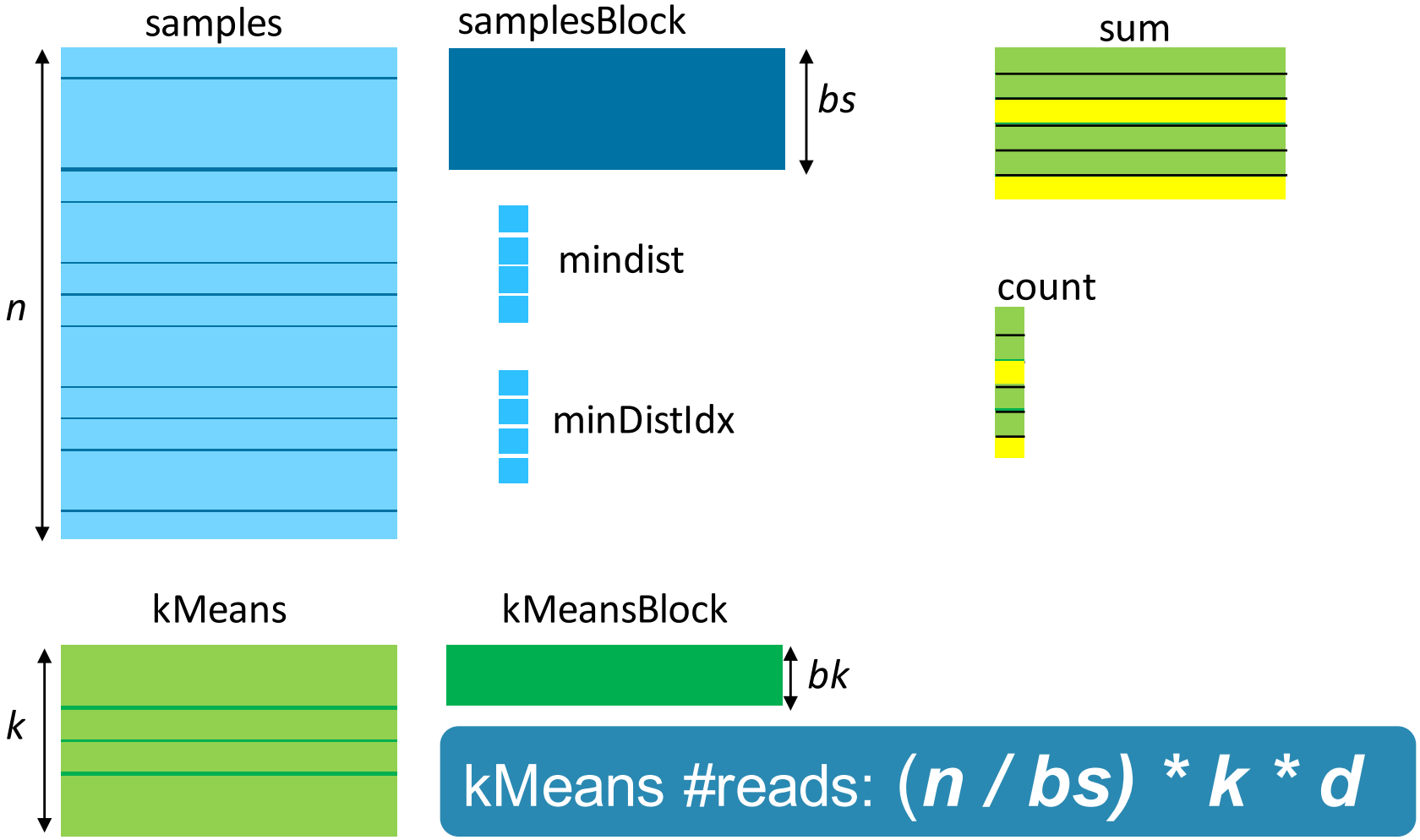
Parallel Pattern Interchange

- Reorder nested patterns
  - Move 'copy' operations out toward outer pattern(s)
  - Improves locality and reuse of on-chip memory

Strip Mined Patterns	Interchanged Patterns

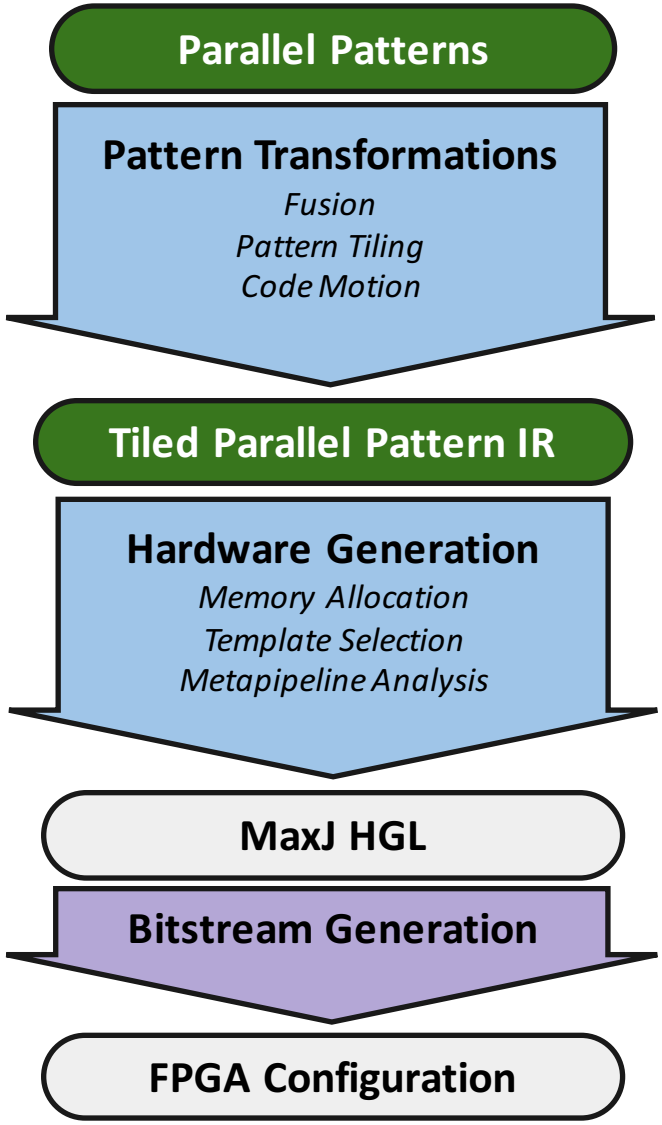
`multiFold(m/b0,n/b1){ii,jj =>`
`  xTl = x.copy(b0+ii, b1+jj)`
`  ((ii,jj), map(b0,b1){i,j =>`
`  multiFold(p/b2){kk =>`
`  yTl = y.copy(b1+jj, b2+kk)`
`  (0, multiFold(b2){ k =>`
`  (0, xTl(i,j)* yTl(j,k))`
`  }{(a,b) => a + b})`
`  }{(a,b) => a + b}`
`  })`
`}`
	`multiFold(m/b0,n/b1){ii,jj =>`
	`  xTl = x.copy(b0+ii, b1+jj)`
	`  ((ii,jj), multiFold(p/b2){kk =>`
	`  yTl = y.copy(b1+jj, b2+kk)`
	`  (0, map(b0,b1){i,j =>`
	`  (0, multiFold(b2){ k =>`
	`  (0, xTl(i,j)* yTl(j,k))`
	`  }{(a,b) => a + b})`
	`  })`
	`  }{(a,b) =>`
	`  map(b0,b1){i,j =>`
	`  a(i,j) + b(i,j) }`
	`  })`
	`}`

Pattern Interchange: kMeans



kMeans #reads: ***(n / bs) * k * d***

Our Approach



Template Selection

Pipe. Exec. Units	Description	IR Construct
Vector	SIMD parallelism	Map over scalars
Reduction tree	Parallel reduction of associative operations	MultiFold over scalars
Parallel FIFO	Buffer ordered outputs of dynamic size	FlatMap over scalars
CAM	Fully associative key-value store	GroupByFold over scalars

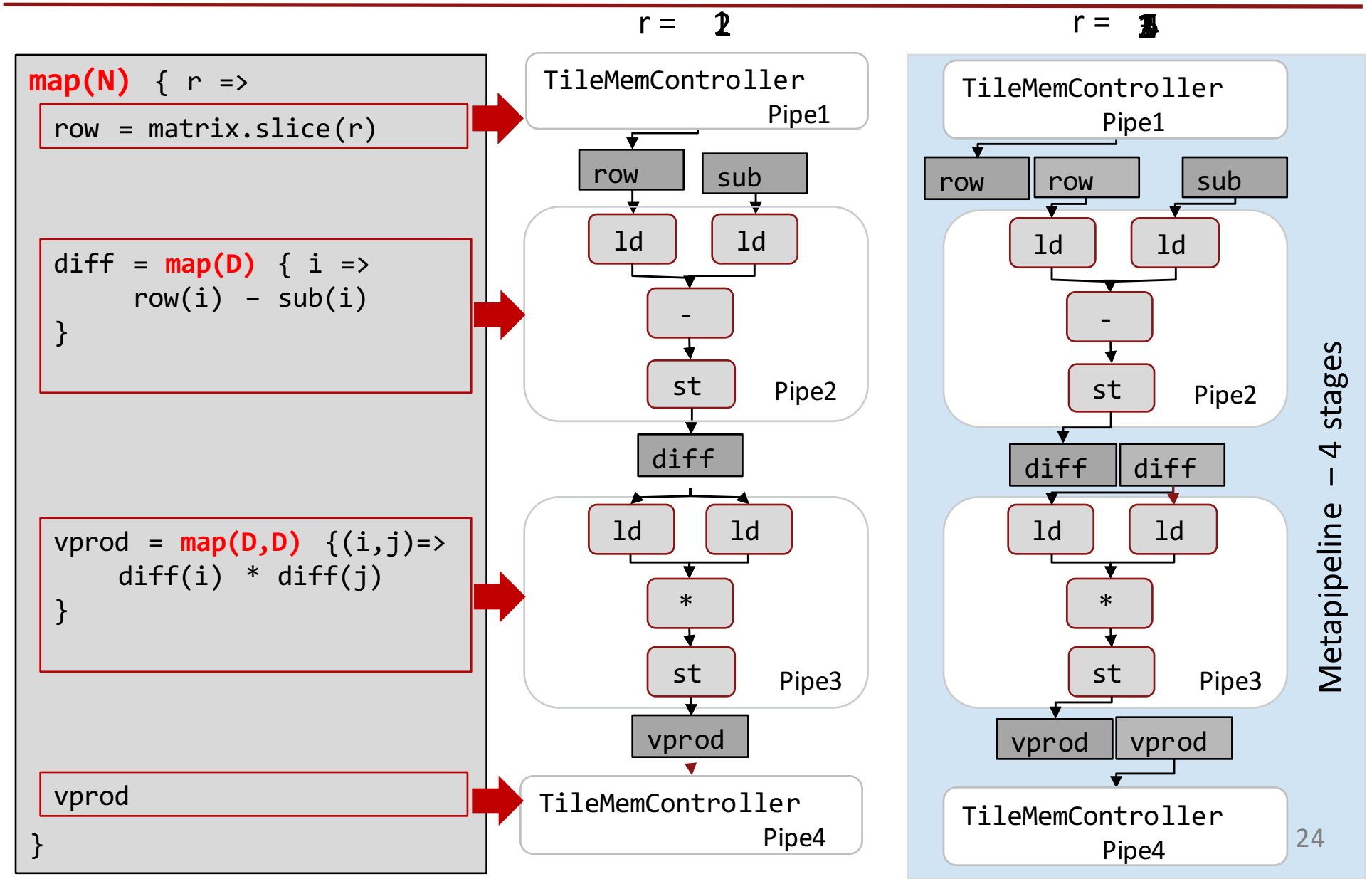
Memories	Description	IR Construct
Buffer	Scratchpad memory	Statically sized array
Double buffer	Buffer coupling two stages in a metapipeline	Metapipeline
Cache	Tagged memory exploits locality in random accesses	Non-affine accesses

Controllers	Description	IR Construct
Sequential	Coordinates sequential execution	Sequential IR node
Parallel	Coordinates parallel execution	Independent IR nodes
Metapipeline	Execute nested parallel patterns in a pipelined fashion	Outer parallel pattern with multiple inner patterns
Tile memory	Fetch tiles of data from off-chip memory	Transformer-inserted array copy

Metapipelining

- Hierarchical pipeline: A “pipeline of pipelines”
  - Exploits nested parallelism
- Inner stages could be other nested patterns or combinational logic
  - Does not require iteration space to be known statically
  - Does not require complete unrolling of inner patterns
- Intermediate data from each stage automatically stored in double buffers
  - Allows stages to have variable execution times
- No need to calculate initiation interval (II)
  - Use asynchronous control signals to begin next iteration

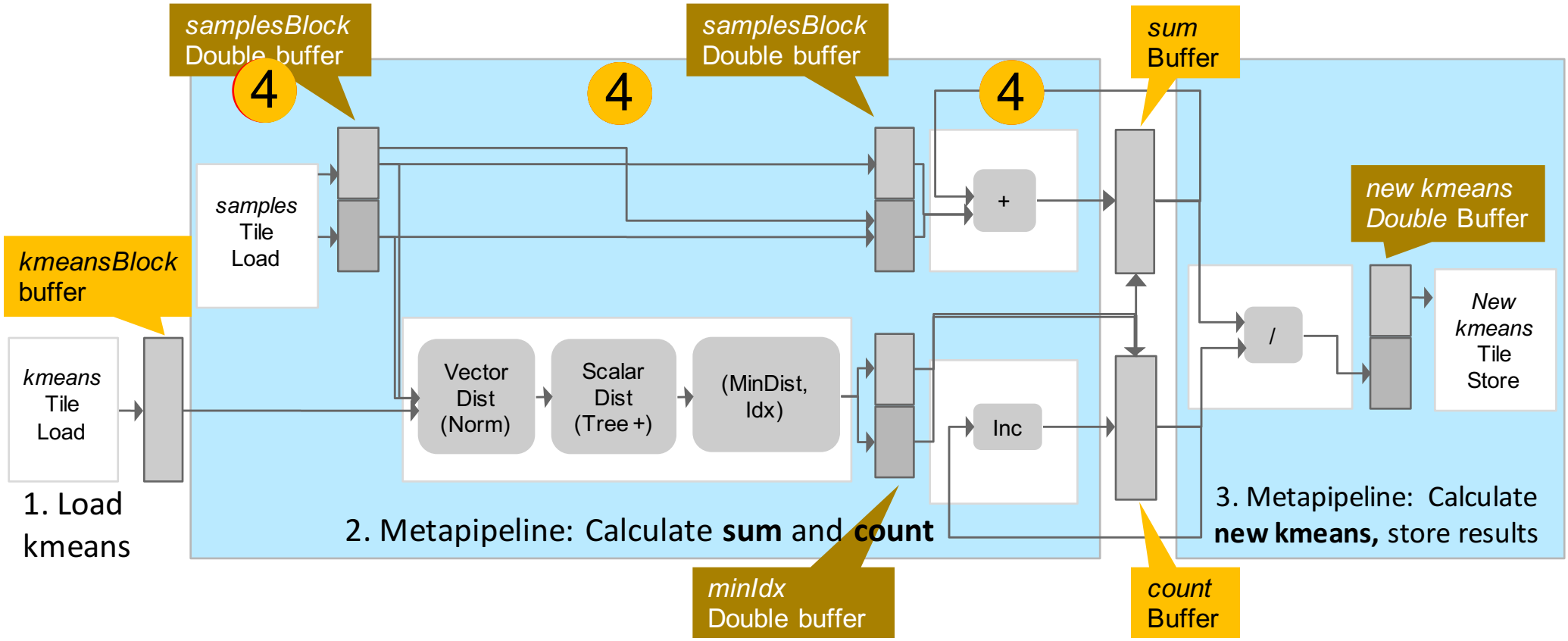
Metapipeline – Intuition



Metapipeline Analysis

- Detects Metapipelines in the tiled parallel pattern IR
- Detection
  - Chain of producer-consumer parallel patterns within the body of another parallel pattern
- Scheduling
  - Topological sort of IR of parallel pattern body
  - List of stages, where each stage consists of one or more independent parallel patterns
  - Promote intermediate buffers to double buffers

Putting It All Together: kMeans



Similar to (and more general than) hand-written designs[1]

[1] Hussain et al, "Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data", AHS 2011

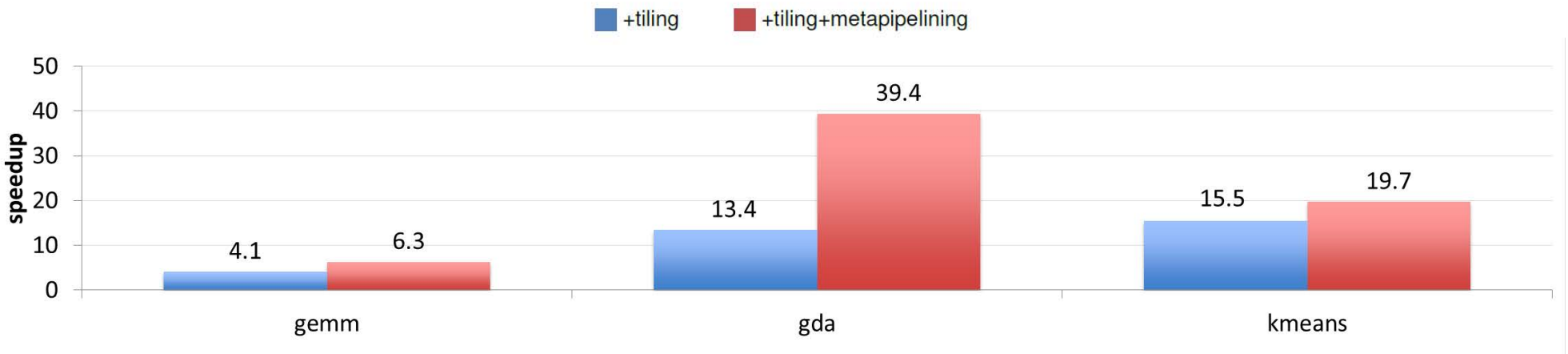
Experimental Setup

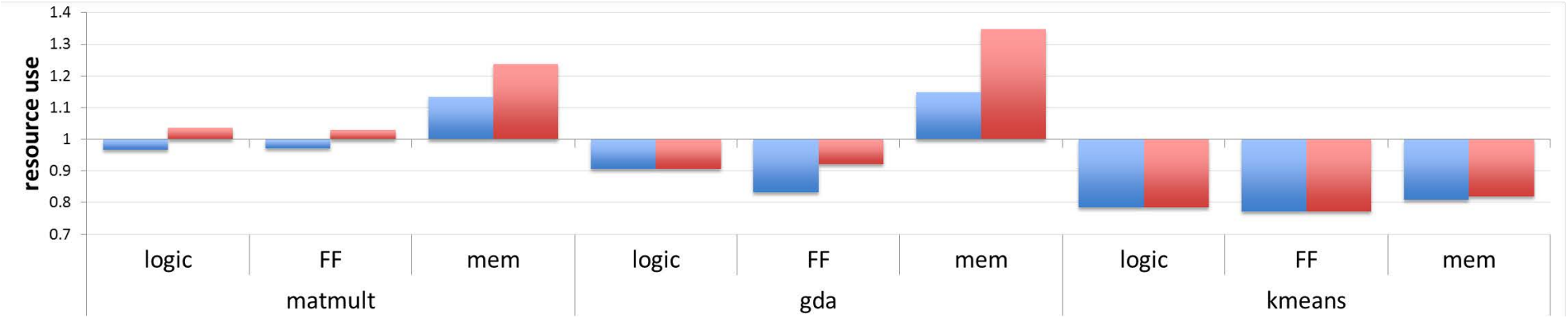
- Board:
  - Altera Stratix V
  - 48 GB DDR3 off-chip DRAM, 6 memory channels
  - Board connected to host via PCI-e
- Execution time reported = ***FPGA execution time***
  - CPU ← → FPGA communication, FPGA configuration time not included
- Goal: How beneficial is ***tiling*** and ***metapipelining***?

Experimental Setup

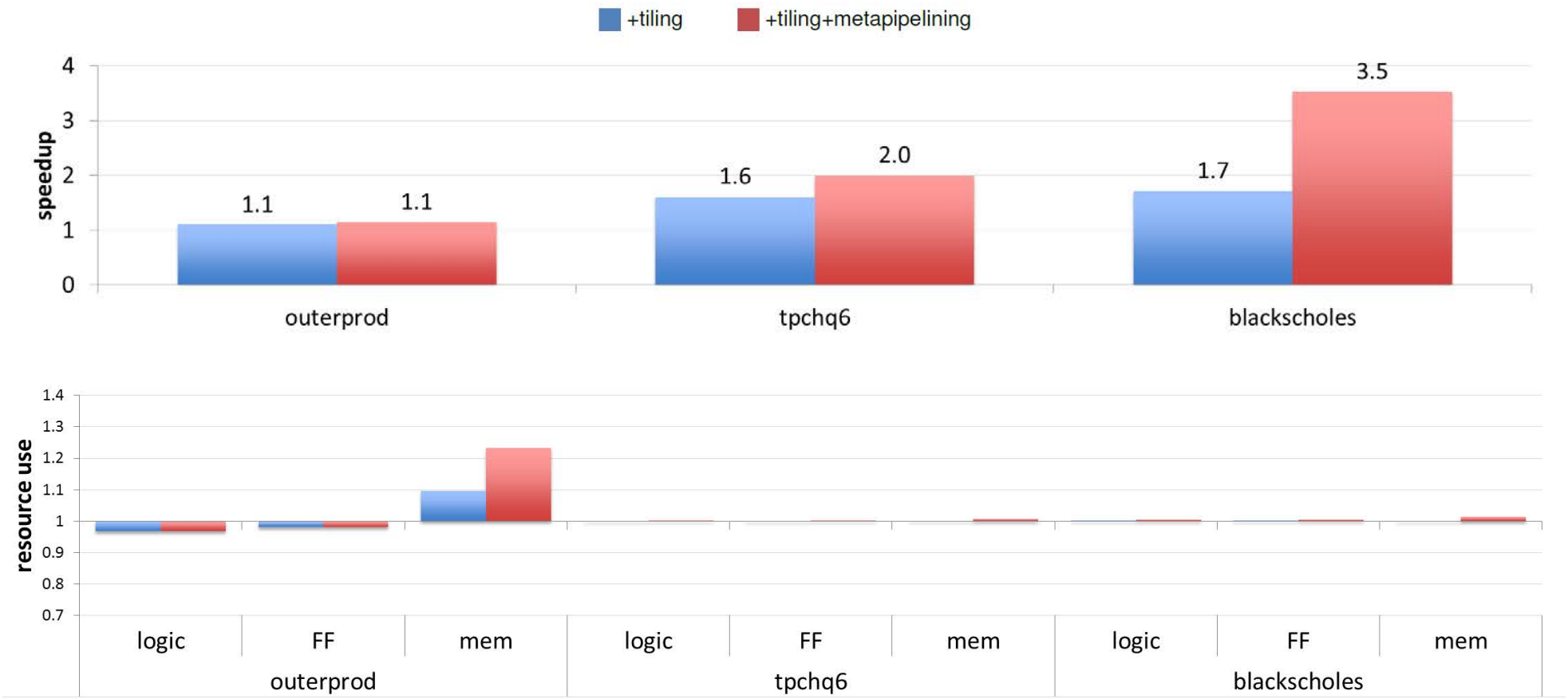
- Baseline
  - Auto generated MaxJ
  - Representative of state-of-the-art HLS tools
- Baseline Optimizations
  - Pipelined execution of innermost loops
  - Parallelized (unrolled) inner loops
    - Parallelism factor chosen by hand
  - Data locality captured at the level of a DRAM burst (384 bytes)
- Parallelism factors are kept consistent across baseline and optimized versions from our flow

Evaluation


+tiling
+tiling+metapipelining
speedup
50
40
30
20
10
0
4.1
6.3
13.4
39.4
15.5
19.7
gemm
gda
kmeans


resource use
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
logic
FF
mem
matmult
logic
FF
mem
gda
logic
FF
mem
kmeans

Evaluation


+tiling
+tiling+metapipelining
speedup
4
3
2
1
0
1.1
1.1
1.6
2.0
1.7
3.5
outerprod
tpchq6
blackscholes
resource use
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
logic
FF
mem
logic
FF
mem
logic
FF
mem
outerprod
tpchq6
blackscholes

Results Summary

- Speedup with tiling: up to **15.5x**
- Speedup with tiling + metapipelining: up to **39.4x**
- Minimal (often positive!) impact on resource usage
  - Tiled designs have fewer off-chip data loaders and storers

Summary

- Two key optimizations:
  tiling and **metapipelining** – to generate efficient FPGA designs from parallel patterns
- Automatic tiling transformations placing fewer restrictions on memory access patterns
- Analysis to automatically infer designs with metapipelines and double buffers
- Significant speedups of up to 39.4x with minimal impact on FPGA resource utilization