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

```c
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!
#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;
}

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

- map
- zip
- reduce
- groupBy

- key1
- key2
- key3
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

```scala
val clusters = samples groupBy { sample =>
  val dists = kMeans map { mean =>
    mean.zip(sample){ (a,b) => sq(a - b) } reduce { (a,b) => a + b }
  }
  // Compute closest mean for each 'sample'
  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 { a => a / count }
  // 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

Parallel Patterns

Pattern Transformations
  Fusion
  Pattern Tiling
  Code Motion

Tiled Parallel Pattern IR

Hardware Generation
  Memory Allocation
  Template Selection
  Metapipeline Analysis

MaxJ HGL

Bitstream Generation

FPGA Configuration
Our Approach

High-level Parallel Patterns helps productivity

Data Locality improved with parallel pattern tiling transformations

Nested Parallelism exploited with hierarchical pipelines and double buffers

Generate MaxJ to generate VHDL

Delite

Parallel Patterns

Pattern Transformations
- Fusion
- Pattern Tiling
- Code Motion

Tiled Parallel Pattern IR

Hardware Generation
- Memory Allocation
- Template Selection
- Metapipeline Analysis

MaxJ HGL

Bitstream Generation

FPGA Configuration
Our Approach

Parallel Patterns

Pattern Transformations
- Fusion
- Pattern Tiling
- Code Motion

Tiled Parallel Pattern IR

Hardware Generation
- Memory Allocation
- Template Selection
- Metapipeline Analysis

MaxJ HGL

Bitstream Generation

FPGA Configuration
Our Approach

- Parallel Patterns
  - Pattern Transformations
    - Fusion
    - Pattern Tiling
    - Code Motion
  - Tiled Parallel Pattern IR
    - Hardware Generation
      - Memory Allocation
      - Template Selection
      - Metapipeline Analysis
  - MaxJ HGL
  - Bitstream Generation
  - FPGA Configuration
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

map

groupBy key

reduce  reduce

out_tile0  out_tile1
**kMeans: Untiled**

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

**kMeans #reads:** \( n \times k \times d \)
Parallel Pattern Strip Mining

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

<table>
<thead>
<tr>
<th>Parallel Patterns</th>
<th>Strip Mined Patterns</th>
</tr>
</thead>
</table>
| `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

### Variables
- `n`: number of samples
- `k`: number of clusters
- `d`: dimensionality of samples

### Data Structures
- **samples**
  - `n` samples
  - `kMeans` with `k` clusters
- **samplesBlock**
  - `bs` blocks of `n` samples
- **sum**
  - `bs` blocks of `m` dimensions
- **count**
  - `bs` blocks of `k` counts
- **mindist**
  - `bs` blocks of `k` minimum distances
  - `minDistIdx`

### Computational Costs
- **kMeans #reads:** \( n \times k \times d \)
Reorder nested patterns
- Move ‘copy’ operations out toward outer pattern(s)
- Improves locality and reuse of on-chip memory

<table>
<thead>
<tr>
<th>Strip Mined Patterns</th>
</tr>
</thead>
</table>
| `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}
  })
} |

<table>
<thead>
<tr>
<th>Interchanged Patterns</th>
</tr>
</thead>
</table>
| `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) => a + b}
  })
} |

map(b0,b1) {i,j =>
  a(i,j) + b(i,j) }
}
Pattern Interchange: kMeans

\[ \text{kMeans \#reads: } \left(\frac{n}{bs}\right) \times k \times d \]
Our Approach

- Parallel Patterns
  - Pattern Transformations
    - Fusion
    - Pattern Tiling
    - Code Motion
  - Tiled Parallel Pattern IR
    - Hardware Generation
      - Memory Allocation
      - Template Selection
      - Metapipeline Analysis
  - MaxJ HGL
  - Bitstream Generation
  - FPGA Configuration
## Template Selection

<table>
<thead>
<tr>
<th>Pipe. Exec. Units</th>
<th>Description</th>
<th>IR Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>SIMD parallelism</td>
<td>Map over scalars</td>
</tr>
<tr>
<td>Reduction tree</td>
<td>Parallel reduction of associative operations</td>
<td>MultiFold over scalars</td>
</tr>
<tr>
<td>Parallel FIFO</td>
<td>Buffer ordered outputs of dynamic size</td>
<td>FlatMap over scalars</td>
</tr>
<tr>
<td>CAM</td>
<td>Fully associative key-value store</td>
<td>GroupByFold over scalars</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memories</th>
<th>Description</th>
<th>IR Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Scratchpad memory</td>
<td>Statically sized array</td>
</tr>
<tr>
<td>Double buffer</td>
<td>Buffer coupling two stages in a metapipeline</td>
<td>Metapipeline</td>
</tr>
<tr>
<td>Cache</td>
<td>Tagged memory exploits locality in random accesses</td>
<td>Non-affine accesses</td>
</tr>
</tbody>
</table>

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

map(N) { r =>
    row = matrix.slice(r)
}

diff = map(D) { i =>
    row(i) - sub(i)
}

vprod = map(D,D) {(i,j)=>
    diff(i) * diff(j)
}

vprod

TileMemController
Pipe1

12 1234

row = matrix.slice(r)
diff = map(D) { i =>
    row(i) - sub(i)
}
vprod = map(D,D) {(i,j)=>
    diff(i) * diff(j)
}

Metapipeline – 4 stages

TileMemController
Pipe1

TileMemController
Pipe2

TileMemController
Pipe3

TileMemController
Pipe4

row = matrix.slice(r)
diff = map(D) { i =>
    row(i) - sub(i)
}
vprod = map(D,D) {(i,j)=>
    diff(i) * diff(j)
}

Metapipeline – 4 stages

TileMemController
Pipe1

TileMemController
Pipe2

TileMemController
Pipe3

TileMemController
Pipe4

row = matrix.slice(r)
diff = map(D) { i =>
    row(i) - sub(i)
}
vprod = map(D,D) {(i,j)=>
    diff(i) * diff(j)
}

Metapipeline – 4 stages

TileMemController
Pipe1

TileMemController
Pipe2

TileMemController
Pipe3

TileMemController
Pipe4
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 $\leftarrow \rightarrow$ 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

![Graph showing speedup and resource use for gemm, gda, and kmeans benchmarks. The graph compares the performance with and without tiling and metapielining. The bars for gemm show a 4.1% speedup with tiling and 6.3% with tiling and metapielining. For gda, the speedup is 13.4% with tiling and 39.4% with tiling and metapielining. For kmeans, the speedup is 15.5% with tiling and 19.7% with tiling and metapielining.](image-url)
Evaluation

![Evaluation Graph](image-url)
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 **metapiipelining** – to generate efficient FPGA designs from parallel patterns
- Automatic tiling transformations placing fewer restrictions on memory access patterns
- Analysis to automatically infer designs with metapiipelines and double buffers
- Significant speedups of up to 39.4x with minimal impact on FPGA resource utilization