P6.Dataflow Graph Analysis

Tiago Santos
CTM / HumanISE
INESC TEC

Mapping loop-based basic blocks onto accelerators

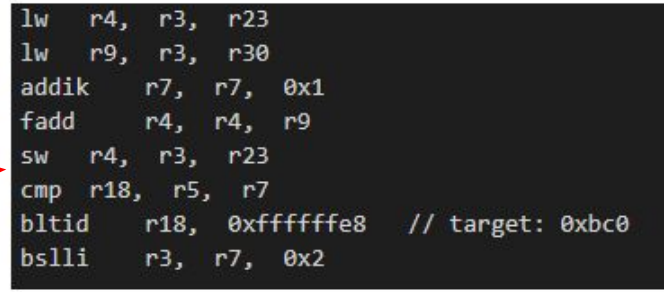
- Detection on the Binary Translation Framework (BTF)
  - Capable of detecting loop-based basic blocks

```
for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    x[i] = x[i] + beta * A[j][i] * y[j];

for (i = 0; i < _PB_N; i++)
  x[i] = x[i] + z[i];

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    w[i] = w[i] +  alpha * A[i][j] * x[j];
```



Mapping loop-based basic blocks onto accelerators

- How can we efficiently map loop-based BBs onto accelerators?

- Two possible optimizations:
  - Map multiple iterations instead of just one (published in FPT' 21)
  - Detect memory accesses that can be implemented as streams (work in progress)

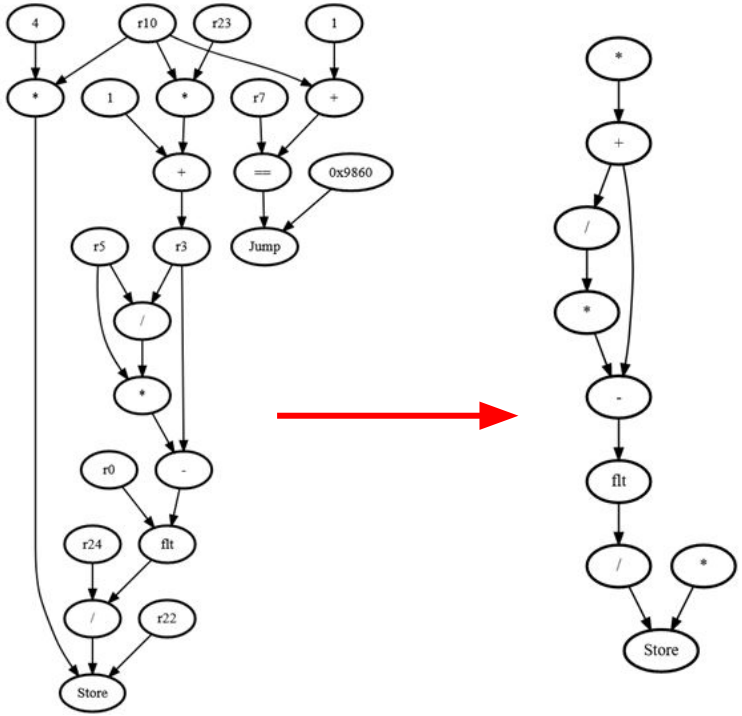
Mapping multiple iterations of a basic block

- Loop-based basic blocks are repeated, sequentially, on the trace
- Can we capture N iterations at once, instead of just one?
- Parallel with the loop unrolling performed over source code
- Potential to parallelize iteration-independent operations, and improve ILP of inter and intra-iteration dependent operations

From instructions to a dataflow graph

- Segments converted onto a data flow graph (DFG)
  - Data stored in register nodes and modified by operation nodes

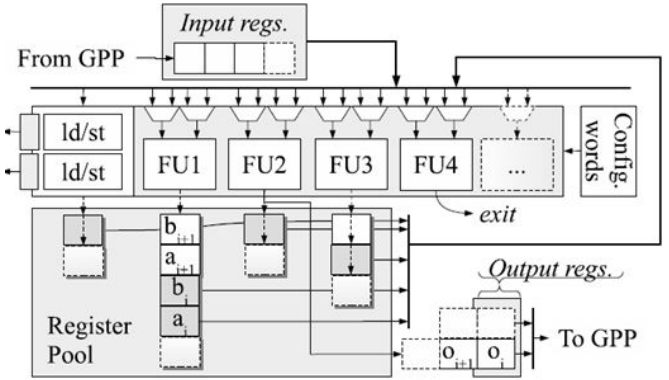
- Further simplified into a dependence graph
  - Branching operations removed
  - Intermediate registers are removed
  - In/outs are removed



Scheduling onto accelerators

- Our accelerator model for this optimization is a Coarse-Grained Loop Accelerator
  - A 1D CGRA with a fully connected crossbar
  - Two types of functional units: ALUs and memory ports
  - Fully access to the system memory
  - Previously validated on-chip

- We use list scheduling to map the dependency graph onto the accelerator:
  - Priority is defined by the latency-adjusted longest path from a node to a graph sink
  - Number of successors used as a tie-breaking heuristic



N. M. C. Paulino, J. C. Ferreira and J. M. P. Cardoso, "Generation of Customized Accelerators for Loop Pipelining of Binary Instruction Traces," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 21-34, Jan. 2017, doi: 10.1109/TVLSI.2016.2573640.

Dealing with constraints

- How to deal with any possible memory dependencies between memory accesses? (e.g. Read-after-Write, Write-after-Read, Write-after-Write)
  - We can assume that memory accesses alias by adding extra edges to the dependency graph to enforce a worst-case scenario

- How do we know if the iterations we're repeating should actually execute?
  - E.g., if we choose to offload 5 repeated iterations on a loop with 1004 iterations, we'd call the accelerator 200 times and execute the remaining 4 iterations on the CPU
  - Easy to establish guards for loops with a fixed number of iterations

Experimental evaluation - setup

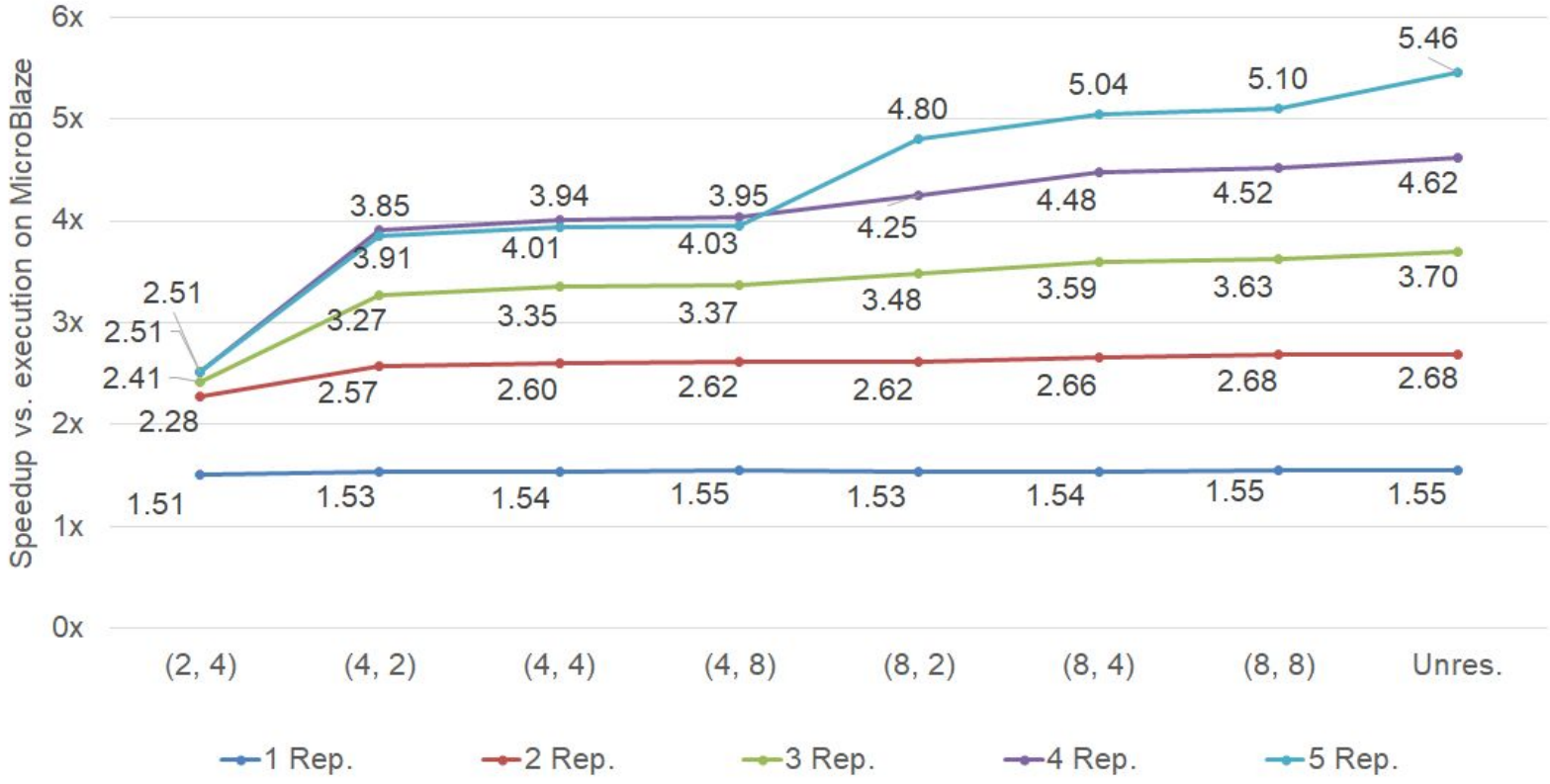
- 19 benchmarks from the PolyBench benchmark suite
- Compiled for the MicroBlaze architecture using floating-point datatypes
- Three-fold analysis:
  - Repetition values up to 5 for all benchmarks
  - Different CGRA functional units configurations (number of memory ports and ALUs)
  - Scheduling the basic blocks with and without memory overlaps
- Results measured in terms of average speedup achieved by an iteration vs. the CPU version

Experimental evaluation - detected basic blocks

- 51 basic blocks found
- Average of ~12 instructions/block
- 3:1 ratio between arithmetic and memory operations

Benchmark	#Basic Blocks	Avg. Instructions	Avg. Arithmetic Ops.	Avg. Memory Ops
2mm	5	12.20	9.80	1.40
3mm	5	12.20	9.80	1.40
adi	5	16.20	10.80	4.40
atax	3	8.33	5.67	1.67
bicg	3	11.33	7.00	3.33
covariance	3	8.33	5.33	2.00
doitgen	1	12.00	10.00	1.00
fdtd2d	1	19.00	15.00	3.00
gemm	2	10.50	6.50	3.00
gemver	4	11.25	6.50	3.75
gesummv	1	16.00	7.00	8.00
mvt	4	10.75	7.75	2.00
nussinov	3	6.67	4.33	1.00
symm	1	18.00	11.00	6.00
syr2k	3	15.67	10.00	4.67
syrk	3	11.67	7.33	3.33
trisolv	3	10.33	7.67	1.67
trmm	1	12.00	8.00	3.00
Total (avg.)	51	11.88	8.14	2.73

Experimental evaluation – average speedups



Concluding Remarks

- The developed optimization using repeated basic blocks produced interesting results:
  - Even the most pessimistic version had clear advantages over a CPU execution
  - Not considering memory overlap increases speedups up to 40% (5.1x vs 2.9x)
  - CGRA adaptability and configurability are of paramount importance
  - Speedups generally increase with the number of repetitions, as long as the CGRA has enough resources

- Memory overlapping impose the biggest barriers towards better and more robust optimizations

- Future work: employ memory disambiguation techniques to find solutions within the upper and lower bounds defined by this study, and explore the interplay with loop pipelining