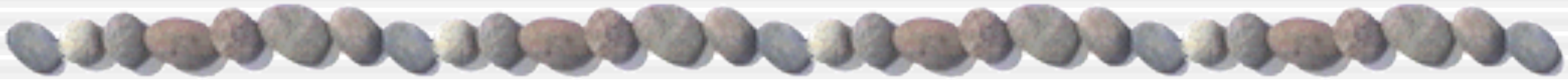
UNIVERSITY of CALIFORNIA Riverside

COMPUTER
SCIENCE &ENGINEERING

Compiled Acceleration of C Codes for FPGAs



Walid Najjar

Professor, Computer Science & Engineering

University of California Riverside

ROCCC

- Riverside Optimizing Compiler for Configurable Computing
    - A C/C++ to VHDL compiler
    - Built on SUIF 2 and MachSUIF
- Objective
    - Code acceleration via mapping of circuits to FPGA
    - Same speed as hand-written VHDL codes
    - Improved productivity
        - Allows design and algorithm space exploration
    - Keeps the user fully in control
        - We automate only what is very well understood

Motivation

- Bridge the *semantic* gap
  - Between algorithms and circuits
- Large scale parallelism on FPGAs
  - Exploiting it with HDLs can be labor intensive
- Bridge the *productivity* gap
  - Translating concise C codes to large scale circuits

Focus

- Extensive compile time optimizations
  - Maximize parallelism, speed and throughput
  - Minimize area and memory accesses
- Optimizations
  - Loop level: fine grained parallelism
  - Storage level: compiler configured storage for data reuse
  - Circuit level: expression simplification, pipelining

Target Applications

Any application that can be accelerated on an FPGA

- Embedded domain
  - signal, image, video processing, communication, cryptography, pattern matching
- Biological sciences
  - Protein folding, DNA and RNA string matching
- Network processing
  - Virus signature detection, payload parsing
- Data mining

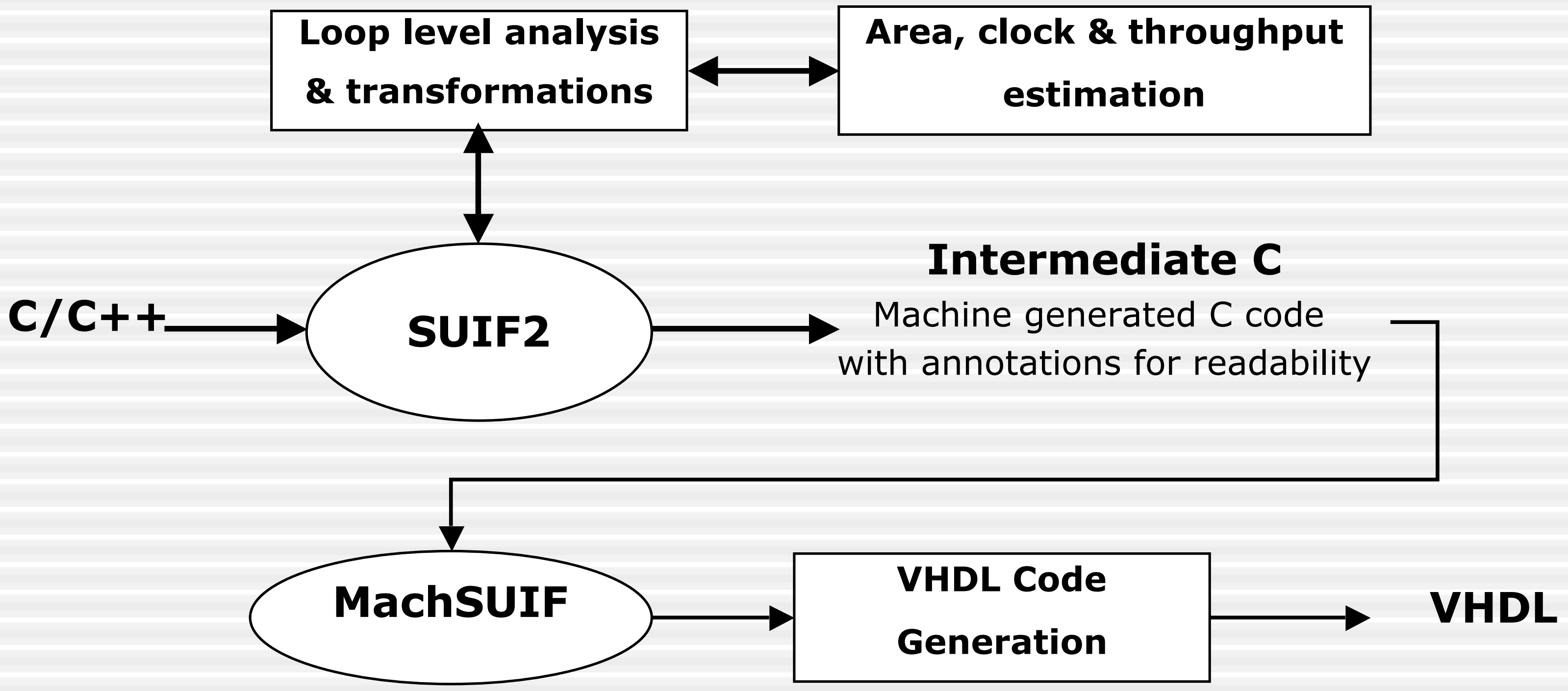
Features

- Smart compiling, simple control
  - Extensive compile time transformations and optimizations
  - All under user control
- Importing existing IP into C code
  - Leverage the huge wealth of IP codes when possible
- Not only a compiler
  - A design space exploration tool

What ROCCC would not do

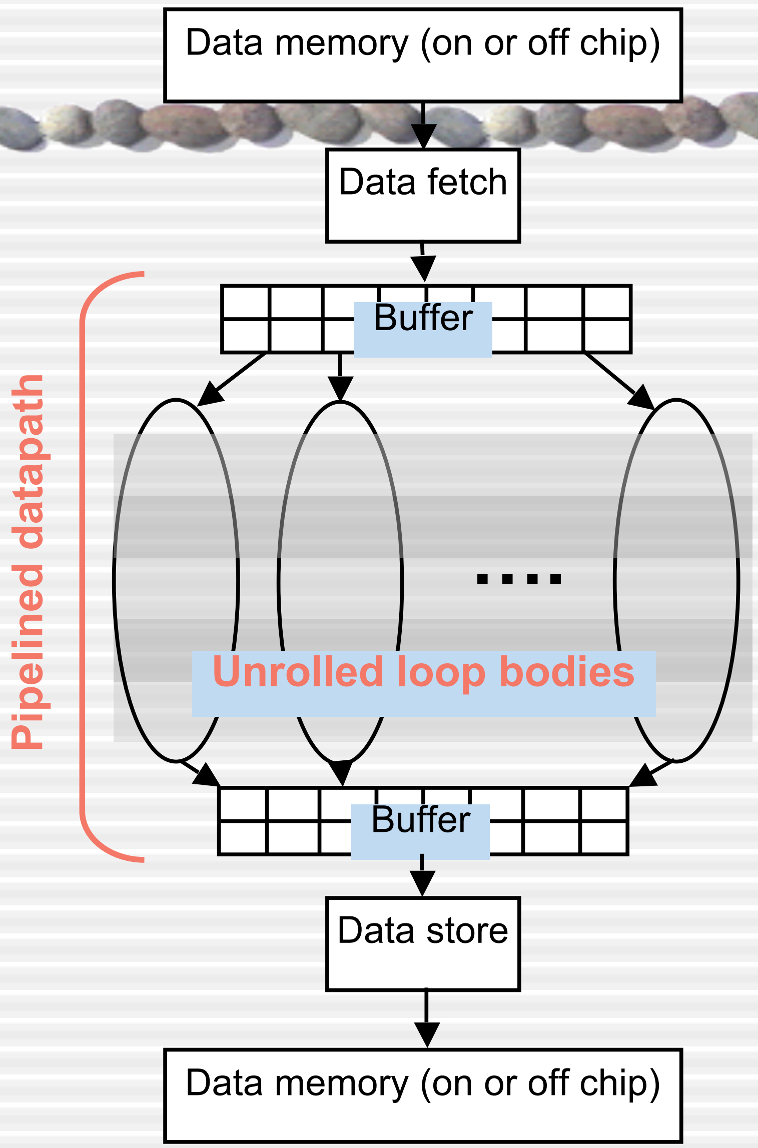
- Compile arbitrary code
  - Application codes optimized for sequential execution
  - FPGA implementation requires other algorithms
  - Code generation for FPGAs is hard enough, we cannot also solve the “dusty deck” problem too!
- FPGA as an accelerator
  - ROCCC is <u>not</u> intended to compile the whole code to FPGA
  - Only compute intensive code segments, typically parallel loops
- Automation: User stays in the loop
  - We can automate what we understand very well
  - So much that we do not yet know or understand, too early for full automation

ROCCC Overview – Current


Loop level analysis & transformations
Area, clock & throughput estimation
C/C++
SUIF2
Intermediate C
Machine generated C code with annotations for readability
MachSUIF
VHDL Code Generation
VHDL

Execution Model

- A simplified model
  - Decoupled memory access from datapath
  - Parallel loop iterations
  - Pipelined datapath



Outline

- Circuit Optimization
  - Same clock speed as hand written HDL code
  - Throughput of one always
- Storage Optimization
  - Minimize number of re-fetch from memory
- Loop Transformations
  - Maximize parallelism
  - Understand impact on area, clock and throughput

Compiled and Hand-written

- Prior results
  - A factor of 2x in speed between hand-coded HDL and compiler generated.
  - Results from SA-C and StreamsC
- Comparison
  - Xilinx IP codes from the web site.
  - Same codes, written in C and compiled.
  - Criteria: Clock rate and Area

Comparison – Clock Rates

Code	Xilinx	ROCCC	%Clock
bit_correlator	212	144	0.679
mul_acc	238	238	1.000
udiv	216	272	1.259
square root	167	220	1.317
cos	170	170	1.000
Arbitrary LUT	170	170	1.000
FIR	185	194	1.049
DCT	181	133	0.735
Wavelet*	104	101	0.971

Comparable

clock rates

(* hand written by us in VHDL)

Performance – Area

Code	Xilinx IP	ROCCC	%Area (slice)
bit_correlator	9	19	2.11
mul_acc	18	59	3.28
udiv	144	495	3.44
square root	585	1199	2.05
cos	150	150	1.00
Arbitrary LUT	549	549	1.00
FIR	270	293	1.09
DCT	412	724	1.76
Wavelet*	1464	2415	1.65

Average area factor: 2.5

Efficacy of Pipelining Scheme

- Compared three schemes
  - ROCCC (us)
  - ImpulseC (LANL)
  - Constraints solver (IRISA, France)

- Benchmarks
  - “Datapath” – a simple compute intensive datapath with feedback within the loop.
  - “Control”– a CORDIC algorithm, a doubly nested control–flow–dominated loop body, with data–dependent branching within the loop.

Pipelining – Results

	Stages	Rate	Memory	Slices	Freq.(MHz)	Samples/s
	DATAPATH – 8 bits					
Impulse	3	2	NA	336	59	29 M
ROCCC	1	1	NA	46	46	46 M
Solver	3	3	4 (2%)	110	161	36 M
	DATAPATH – 32 bits					
Impulse	4	2	NA	901	51	25 M
ROCCC	2	1	NA	125	27	27 M
Solver	3	3	4	304	80	26 M
	CONTROL – 32 bits					
impulse	3	2	NA	157	117	58 M
ROCCC	37	1	NA	2234	79.5	79 M
Solver	2	2	1	196	147	73 M

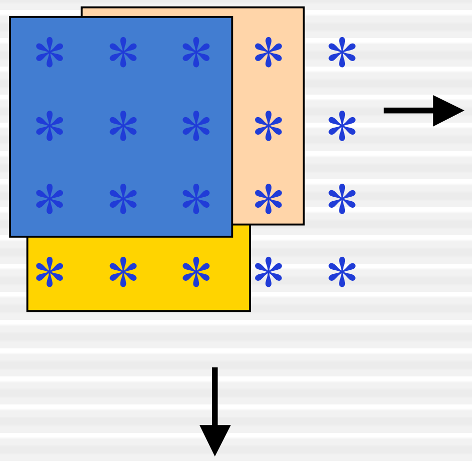
Comments on the Pipeline

- Clock
  - ROCCC has the lowest clock cycle but the highest throughput.
  - Both Datapath and Control
- Area
  - ROCCC has the smallest area on Datapath.
  - The largest on Control.
- Approach
  - No separate controller.
  - Control of the pipeline is integrated with the datapath.

Storage Optimizations

- Objective
  - Detect the reuse of data
  - Structure on chip storage for that data
  - Schedule the access for reuse
  - De-allocate storage when data is not needed
- All at compile time
- Storage optimization reduces bandwidth pressure

Window Operation



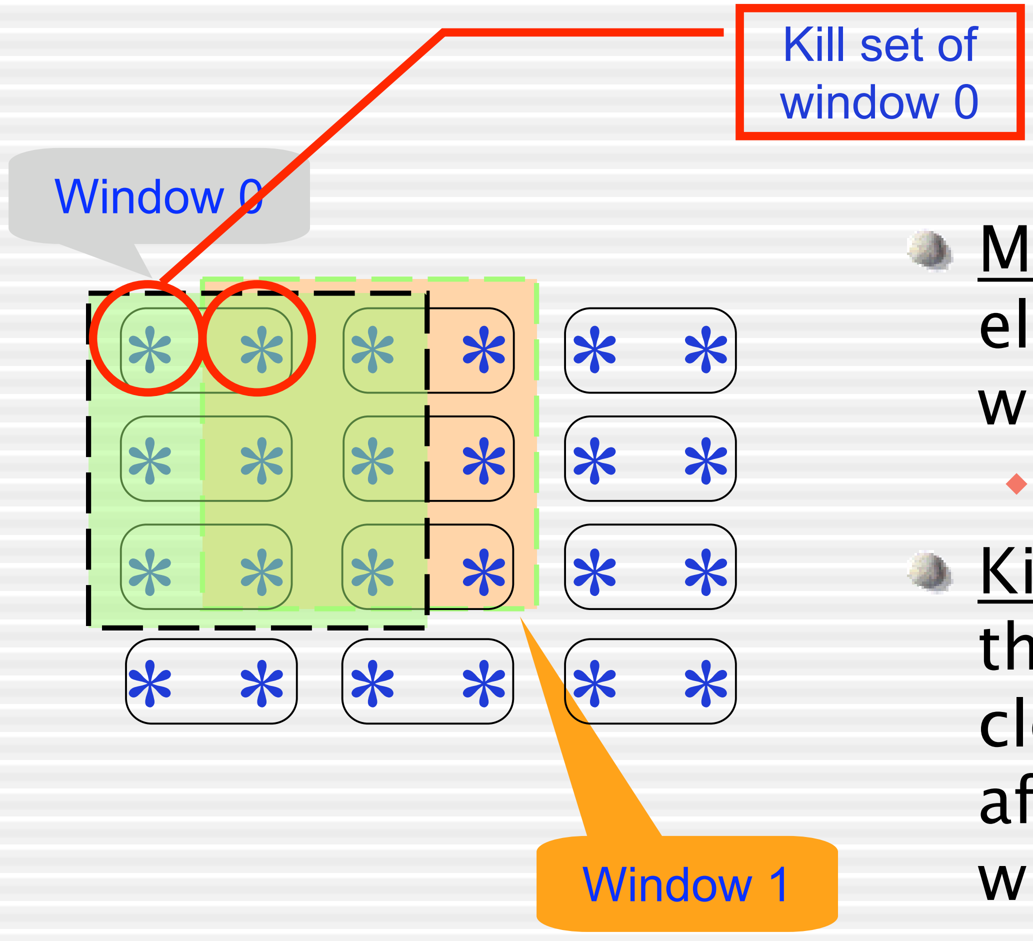
- Window operation: common in signal and image processing
- A window operation: one iteration of a loop or loop nest.
- Window sliding: movement in the iteration space.
- High memory bandwidth pressure.
  - Data reuse
- Separate reading/writing memories.
  - Parallelism

Ref: Guo, Buyukkurt and Najjar, LCTES 2004

Smart Buffer

- Definition
  - In data-path storage (registers)
  - Configured and scheduled by the compiler
  - No register addressing: data is pushed by controller into the data path every cycle
- Parameters
  - Determined by the compiler based on
    - Window sizes in x and y, stride in x and y
    - Data bit width
    - Bus width to memory

Smart Buffer Components

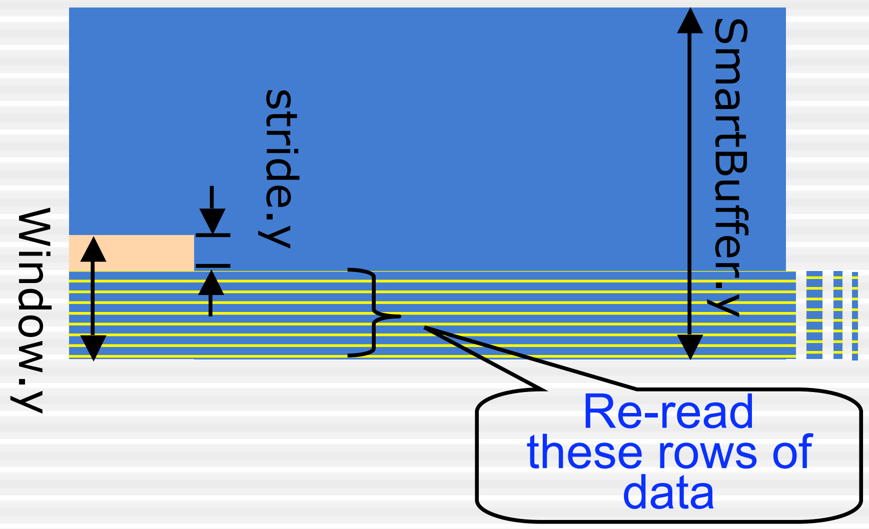


- Managed set: the set of elements covered by a window.
  - All *live:* window available
- Kill set: a set consists of the elements needed to clear their *live* signals after exporting this window

Smart Buffer Code Generation

- Compile time analysis
  - Relies on window size, strides, data width and bus width.
  - Generates *windows* and *sets* in the IR.
- Resulting VHDL code
  - Is not aware of the concepts of *sets* and *windows.*
  - Only describes the logical and sequential relationship between signals/registers
  - Automatic code generation
- We shift run-time control burden to compiler

Smart Buffer Re-Read Factor



- Before: each pixel needs to be read nine times except the image's border.
- After: only a small portion needs to be read twice:

$$\frac{Window.y - stride.y}{SmartBuffer.y} * 100\%$$

$$\frac{3-1}{32} * 100\% = 6.25\%$$

Re-read factor on MIPS: 9 times!

Compiler Transformations

- Pre-Optimization Passes
  - Control Flow Analysis (√)
  - Data Flow Analysis (√)
  - Dependence Analysis in Loops
  - Alias Analysis
- General Transforms
  - Constant Propagation (√)
  - Constant Folding & Identities (√)
  - Copy Propagation (√)
  - Dead Store Elimination (√)
  - Common Sub Expression Elimination (√)
  - Partial Redundancy Elimination (√)
  - Unreachable Code Elimination (√)
- Memory Transformations
  - Scalar Replacement (√)
- Loop Level Transformations
  - Loop Independent Conditional Removal (√)
  - Loop Peeling (√)
  - Index Set Splitting
  - Loop Unrolling – Full (√)
  - Loop Unrolling – Partial (√)
  - Loop Fusion (√)
  - Loop Tiling
  - Invariant Code Motion (√)
  - Strength Reduction

Examples

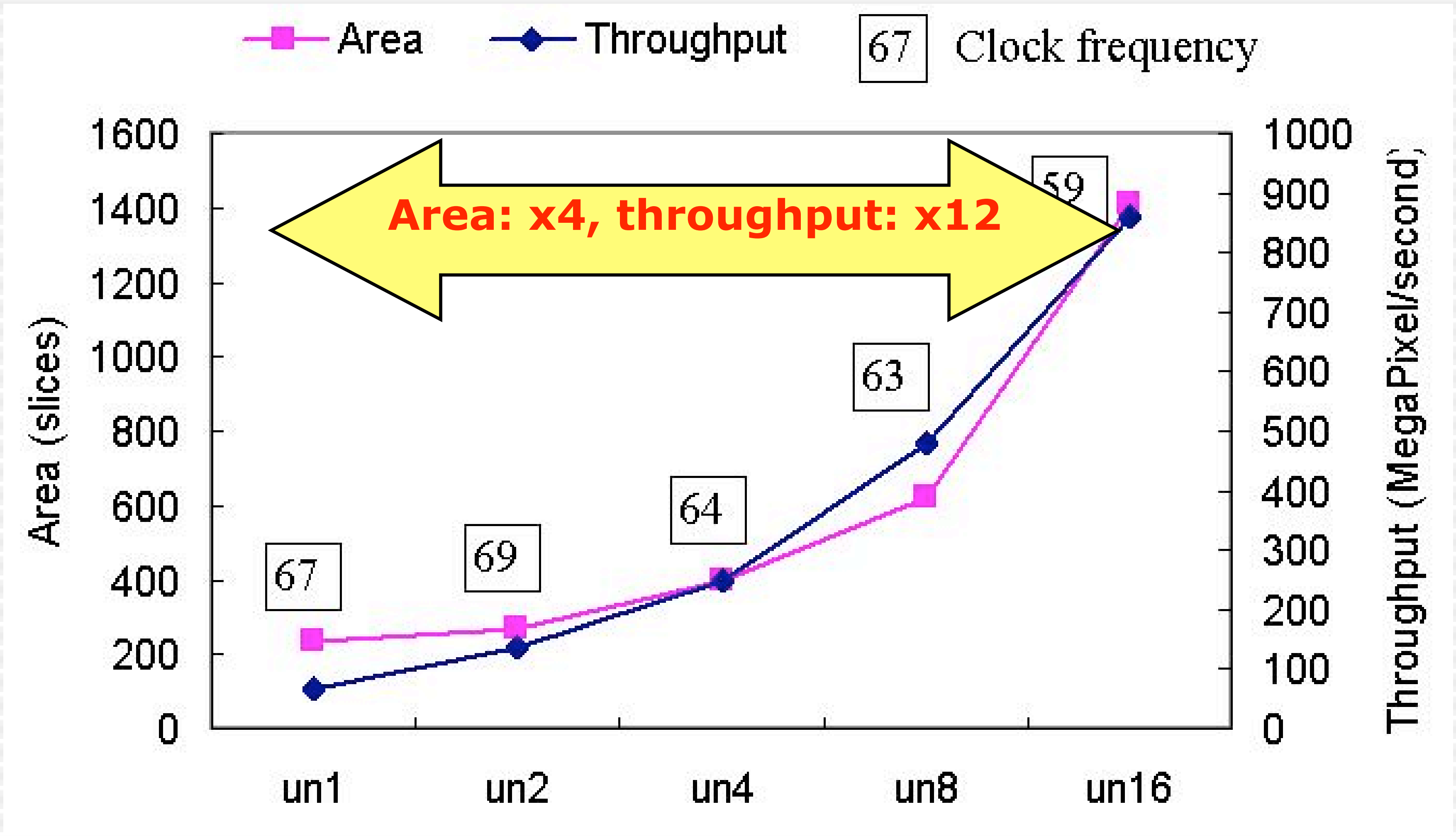
- FIR
  - 5 tap, 8 bits
- Discrete Wavelet Transform
  - 5x3 (lossy) 8 bits
- Smith–Waterman
  - 2 bit data path: DNA
  - 5 bit data path: protein folding
- Bloom Filter
  - Probabilistic exact string matching

FIR C Code

FIR 5-tap

```
for (i=0; i<N; i=i+1) {
   C[i] = 3*A[i] + 5*A[i+1] + 7*A[i+2] +
   9*A[i+3] - A[i+4];}
```

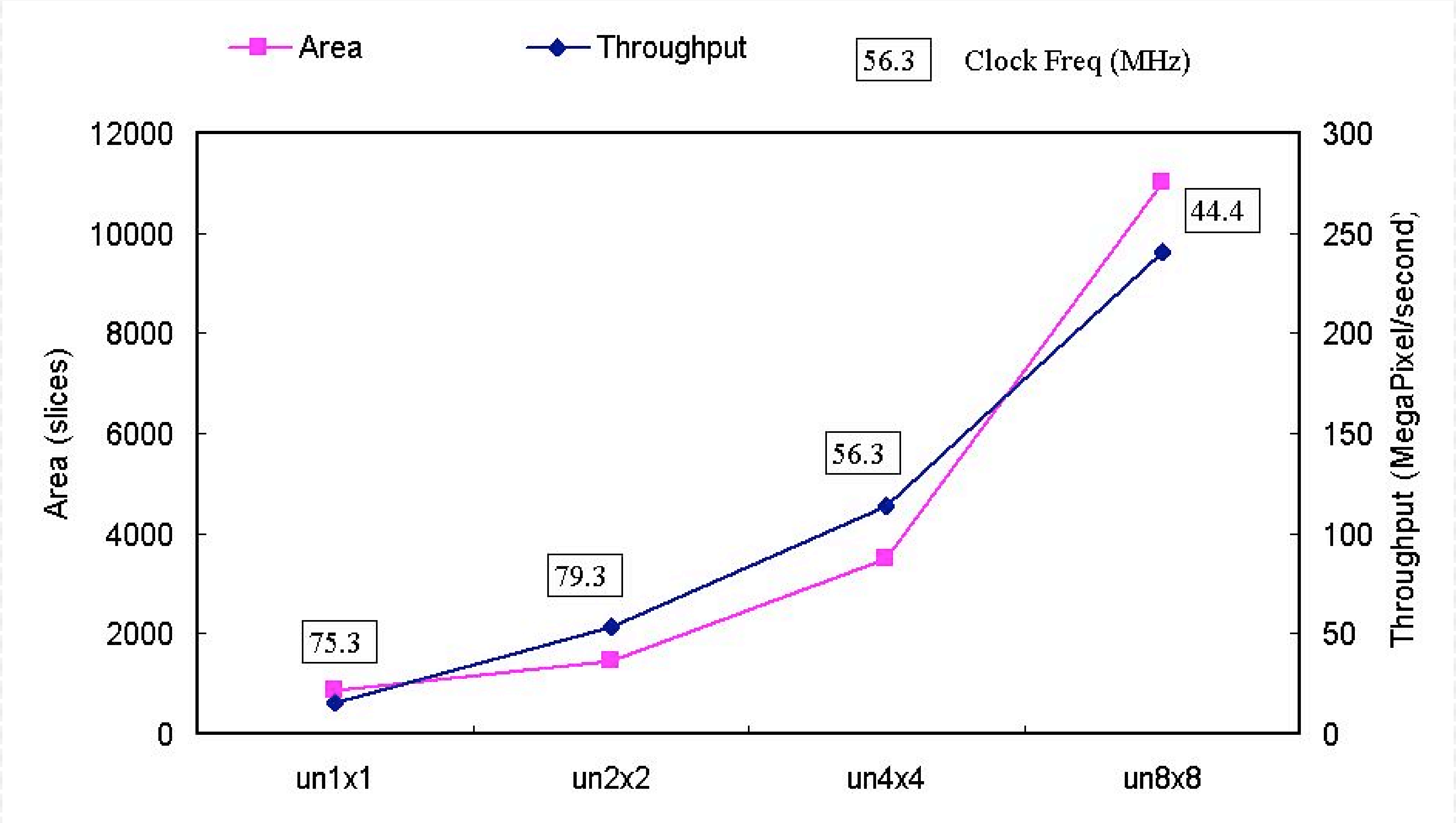
FIR 5-tap


Area
Throughput
67
Clock frequency
Area: x4, throughput: x12
Area (slices)
Throughput (MegaPixel/second)
1600
1400
1200
1000
800
600
400
200
0
1000
900
800
700
600
500
400
300
200
100
0
67
69
64
63
59
un1
un2
un4
un8
un16

DWT C Code

```
for(i = 0; i<508; i = 1+i) {
        for(j = 0; j<510; j = 1+j {
           sum = (6*image[i][j])>> 3;
           sum = sum+(6* image[i][1+j])>> 3;
           sum = sum+(6* image[i][2+j])>> 3;
           sum = sum+(2* image[1+i][j])>> 3;
           sum = sum+(2* image[1+i][1+j])>> 3;
           sum = sum+(2* image[1+i][2+j])>> 3;
           sum = sum+(-1* image[2+i][j])>> 3;
           sum = sum+(-1* image[2+i][1+j])>> 3;
           sum = sum+(-1* image[2+i][2+j])>> 3;
           sum = sum+(8* image[3+i][j])>> 3;
           sum = sum+(8* image[3+i][1+j])>> 3;
           sum = sum+(8* image[3+i][2+j])>> 3;
           sum = sum+(-4* image[4+i][j])>> 3;
           sum = sum+(-4* image[4+i][1+j])>> 3;
           sum = sum+(-4* image[4+i][2+j])>> 3;
           output[i][j] = sum; } }
```

DWT


Area
Throughput
56.3 Clock Freq (MHz)
12000
10000
8000
6000
4000
2000
0
Area (slices)
300
250
200
150
100
50
0
Throughput (MegaPixel/second)
75.3
79.3
56.3
44.4
un1x1
un2x2
un4x4
un8x8

Smith-Waterman Code

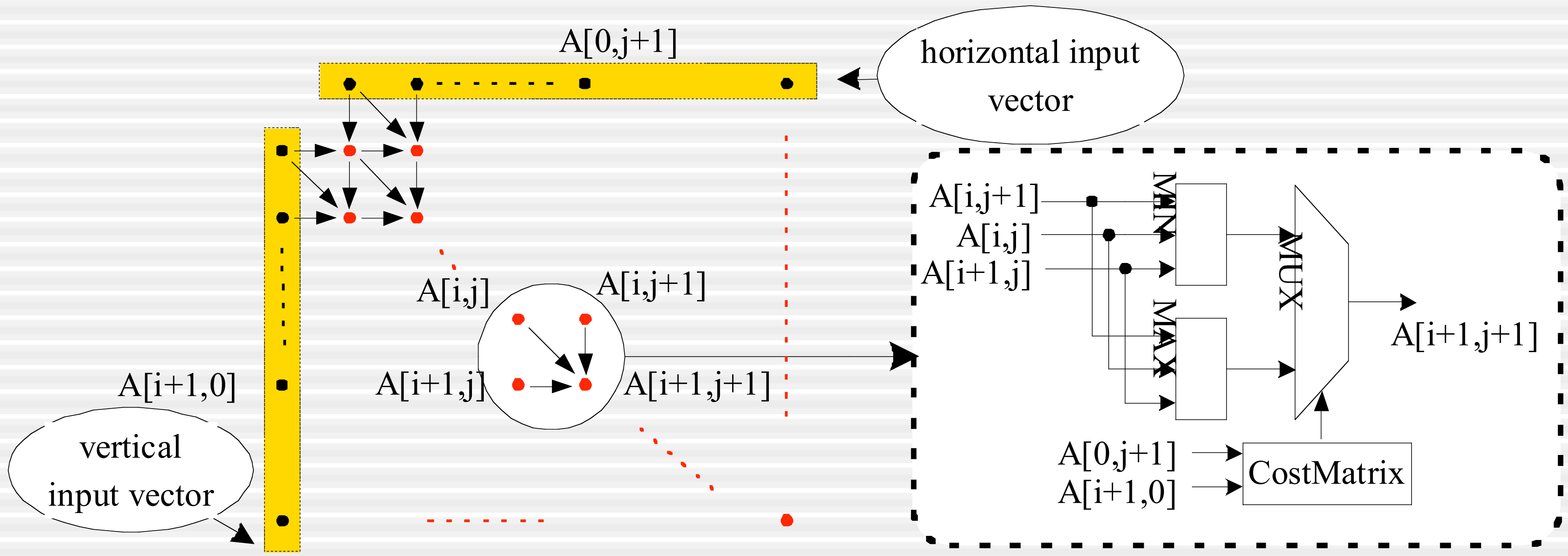
- Dynamic Programming
  - Used in protein modeling, bio-informatics, data mining …
  - A wave-front algorithm with two input strings

$$A[i,j] = F(A[i,j-1], A[i-1, j-1], A[i-1, j])$$

$$F\ = CostMatrix(A[i,0],A[0,j])$$

- Our Approach
  - “Chunk” the input strings in fixed sizes *k*
  - Build a *k x k* template hardware by compiling two nested loops (k each) and fully unrolling both.
  - Host strip mines the two outer loops over this template.

S-W View


A[0,j+1]
horizontal input vector
vertical input vector
A[i+1,0]
A[i,j]
A[i,j+1]
A[i+1,j]
A[i+1,j+1]
A[i,j+1]
A[i,j]
A[i+1,j]
MIN
MAX
MUX
A[i+1,j+1]
A[0,j+1]
A[i+1,0]
CostMatrix

S-W C Code

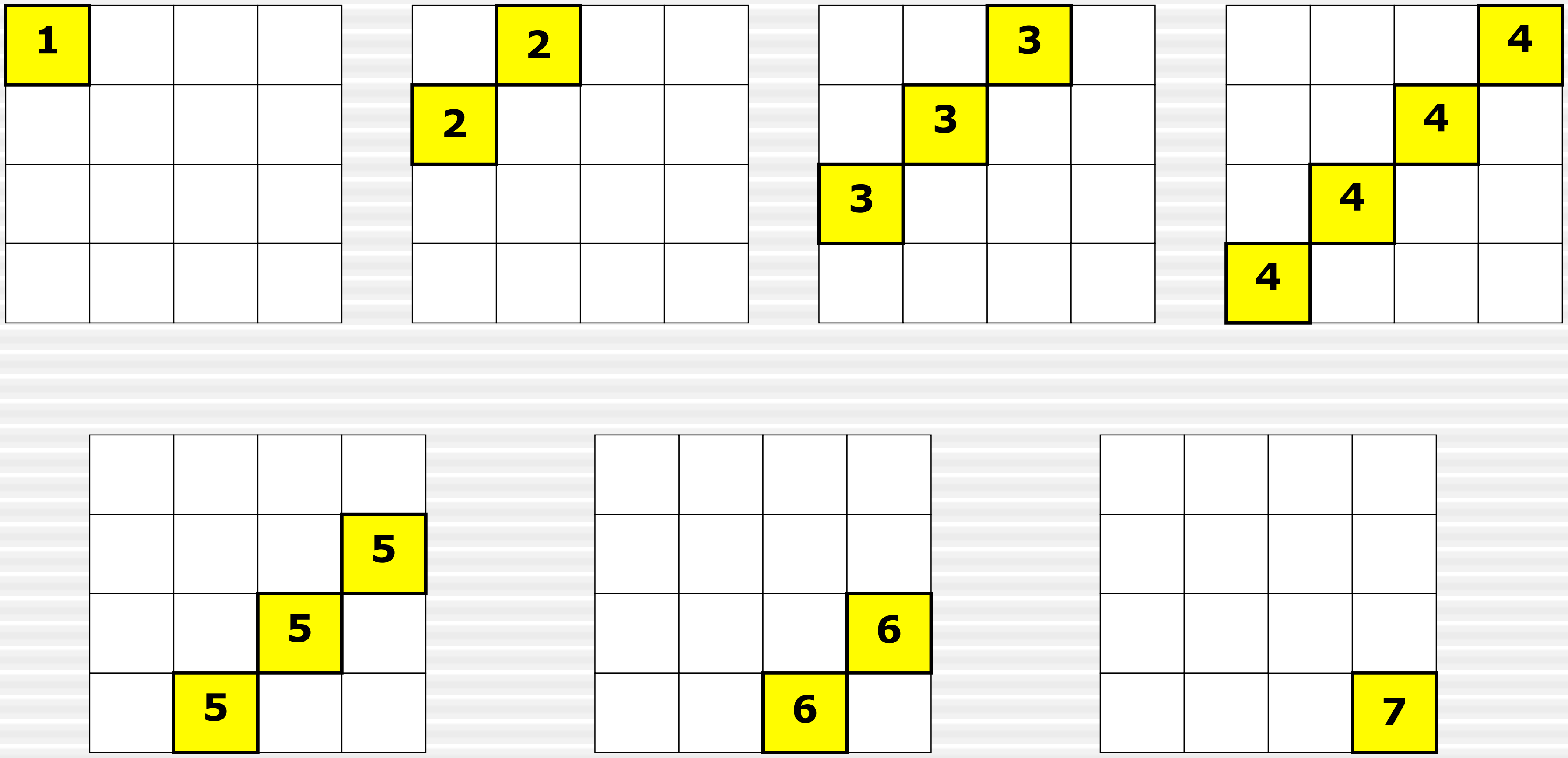
```
int  One_Cell(int a, int b, int c, int d, int e){
       int t1, t2, xy, sel;
       t1 = min3(a, b, c);
       t2 = max3(a, b, c);
       xy = bitcmb(d, e);
       sel = boollut(xy);
       return boolsel(t1, t2, sel);      }
int main(){
     int i, j, N =1024;
     int A[1024][1024];
     for(i=1; i<N; i=i+1)
         for(j=1; j<N; j=j+1)
             A[i][j] = One_Cell(A[i-1][j], A[i][j-1],
                     A[i-1][j-1], BH[i-1], BV[j-1]);
}
```

S-W 2x2 Template

COMPILER GENERATED

```
for(i = 1;(i< N);i = i+2)
   for(j = 1;(j< N);j = j+2)
      for(tmp0 = 0;(tmp0< 2);tmp0 = tmp0+1)
         for(tmp1 = 0;(tmp1< 2);tmp1 = tmp1+1) {
            int  tmp00;
            t1 =  min3(A[i+tmp0- 1][j+tmp1],
                                 A[i+tmp0][j+tmp1- 1],
                                 A[i+tmp0- 1][j+tmp1- 1]);
            t2 =  max3(A[i+tmp0- 1][j+tmp1],
                                  A[i+tmp0][j+tmp1- 1],
                                  A[i+tmp0- 1][j+tmp1- 1]);
            xy =  bitcmb(BH[i+tmp0- 1], BV[j+tmp1- 1]);
            sel =  boollut(xy);
            tmp00 =  boolsel(t1, t2, sel);
            A[i+tmp0][j+tmp1] =tmp00;
         }
```

S-W 4x4 Tile Execution



S-W Results

Tile	Area (slices)	Area (%)	Clock (MHz)	Pipeline stages	GCUPS			Speedup	
					Tile	Chip	P4	Tile	Chip
2-bit data path									
4x4	63	1%	126	3	0.672	189	0.012	**56**	15,750
8x8	286	1%	90	5	1.15	71.2	0.012	**96**	5,934
12x12	755	3%	97	8	1.75	41.1	0.012	**146**	3,425
16x16	1394	5%	108	11	2.51	31.9	0.012	**209**	2,658
5-bit data path									
4x4	817	3%	55	3	0.293	6.35	0.012	**24**	529
8x8	3604	15%	53	5	0.678	3.33	0.012	**56**	277
12x12	8344	35%	58	8	1.04	2.08	0.012	**87**	174
16x16	14883	63%	52	11	1.21	1.21	0.012	**101**	101

Bloom Filter

- Work in progress
  - A bloom filter is a space-efficient data structure used to test the set membership of an element.
  - Adapted to detect virus signature bit patterns in packets.
- Preliminary results
  - 584 MB/sec on 1173 slices out of 46592 (2%)

Bloom Filter C Code

```
for(i=0;i<248;i++)
{   for(j=0;j<7;j++)
  { value = input_stream[i+j];
    temp = value & 0x1;
    for(k=0; k<7; k++)
     {
     result_location1 = result_location1 ^ (hash_function1[k]& temp);
     result_location2 = result_location2 ^ (hash_function1[k]& temp);
     result_location3 = result_location3 ^ (hash_function1[k]& temp);
     result_location4 = result_location4 ^ (hash_function1[k]& temp);
        value = value >> 1;
     }
found = bit_array[result_location1] & bit_array[result_location2] &
     bit_array[result_location3] &  bit_array[result_location4];
   }
}
```

Compile time constant, folded

In data-path Table lookup

Productivity “Speedup”

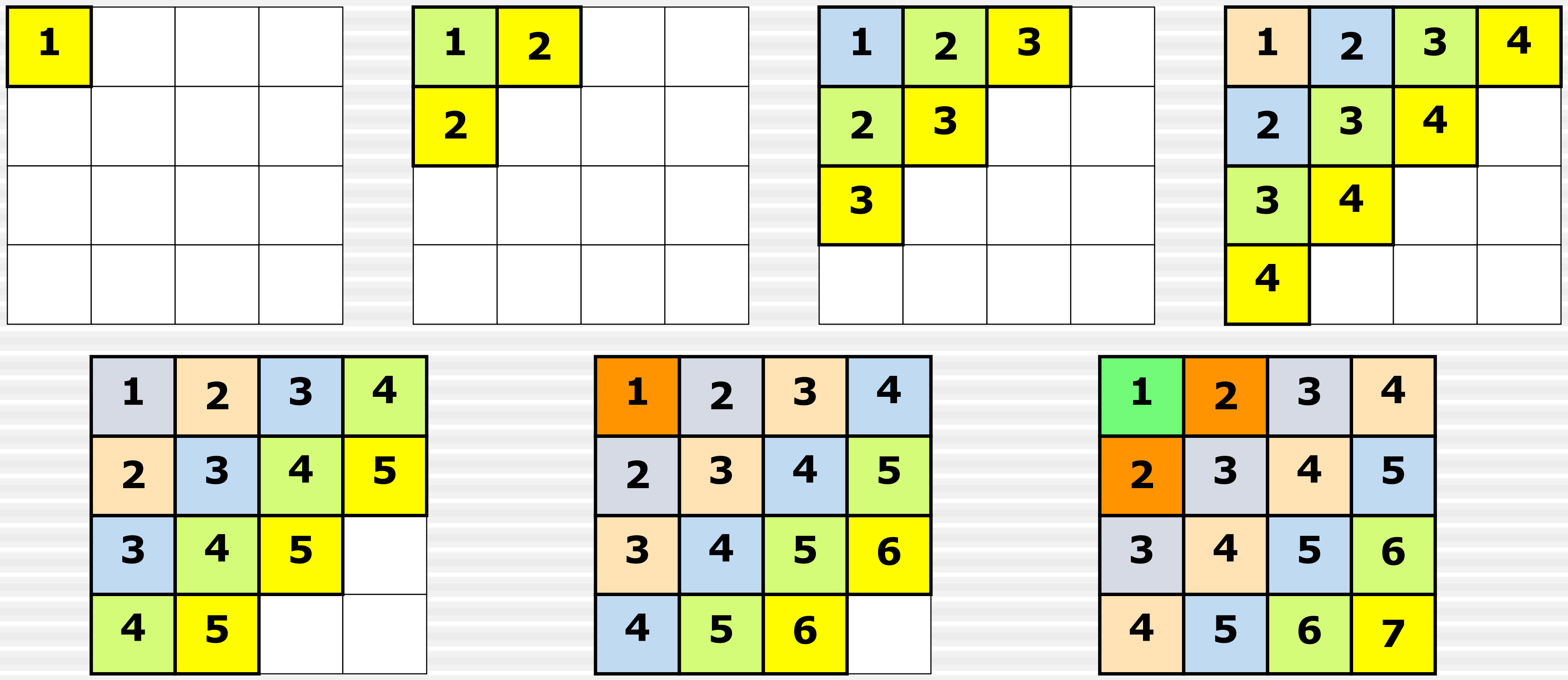
Code	C	VHDL	Transformations
FIR	2	1,100	8x unrolled
DWT	18	16,500	8x8 unrolled
S–W	13	12,000	16x16 tile
B–F	11	3,400	8 bytes

A ratio of ~ 1,000

Current and Future Work

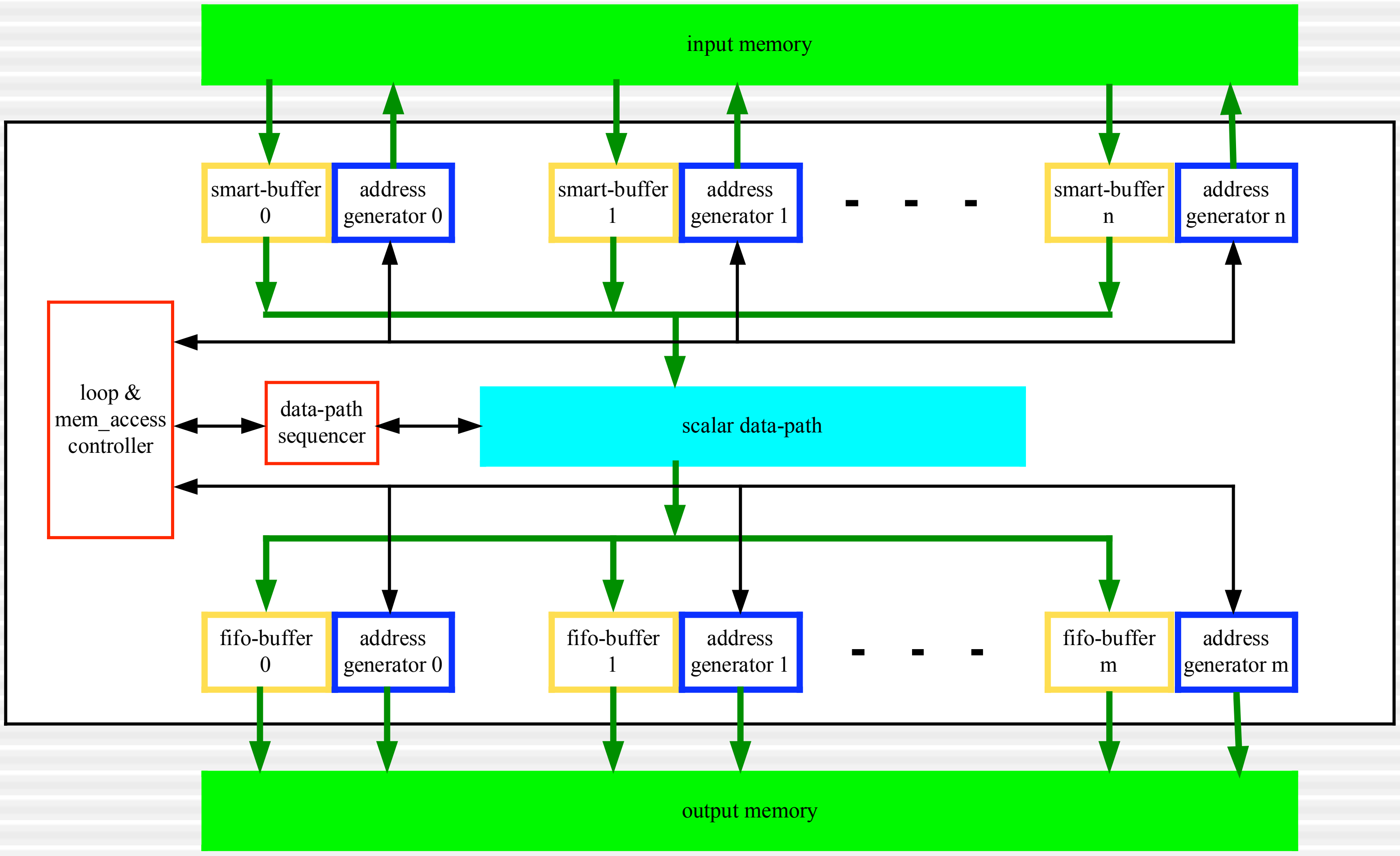
- (More) Compiler transformations
  - Multi-Loop fusion
  - Pipelining of tiled code
  - Smarter smart buffer
- Backend IR for configurable computing
  - Supports circuit optimization and generation
  - Allow multiple front-ends and multiple targets

S-W Pipelined Tile



Increase throughput & speedup by (2k -1)

Smarter Buffer



Conclusion

- ROCCC can
  - Extract and deliver large scale parallelism
    - Instruction and loop levels
  - Optimize on-chip storage
  - High throughput and speedup

www.cs.ucr.edu/roccc

Thank you