Opportunities and Challenge of the Spatial Computing Paradigm - The Programmability Issue

Walid Najjar
Computer Science & Engineering
University of California Riverside

UC RIVERSIDE UNIVERSITY OF CALIFORNIA CSE

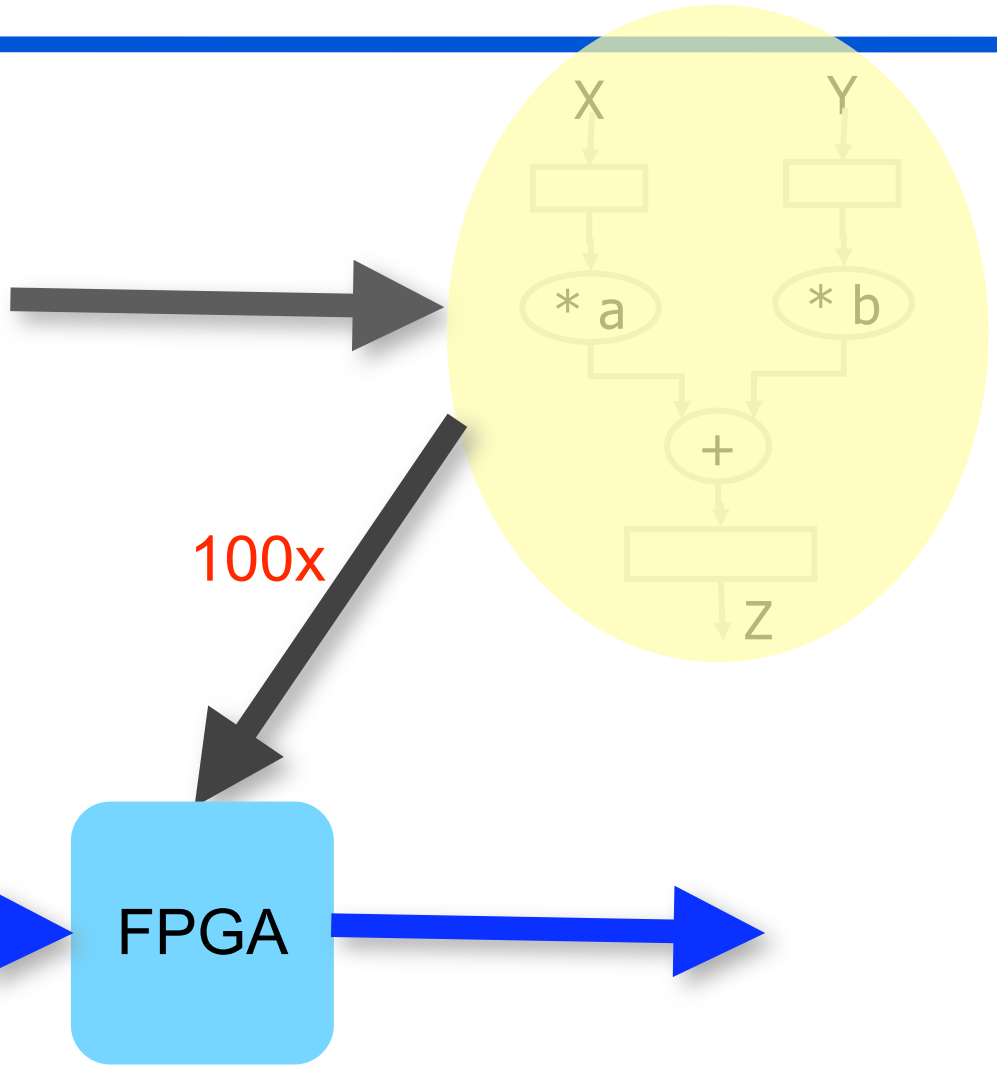
Outline of Talk

- **INTRODUCTION**
  - The spatial computing model
  - An FPGA Primer
  - Potentials of FPGAs as computing platforms
  - The programmability problem
- **EXPERIENCE WITH ROCCC 1.0**
  - Overview
  - Compiler optimizations
  - Applications
- **DESIGN OF ROCCC 2.0**
  - The “why” and “how”

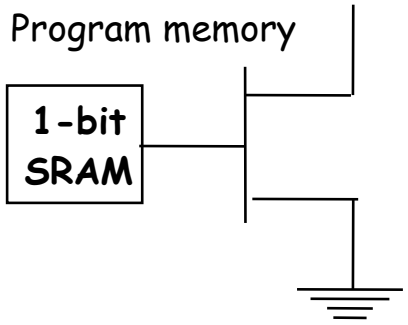
Spatial Computing

Loop body

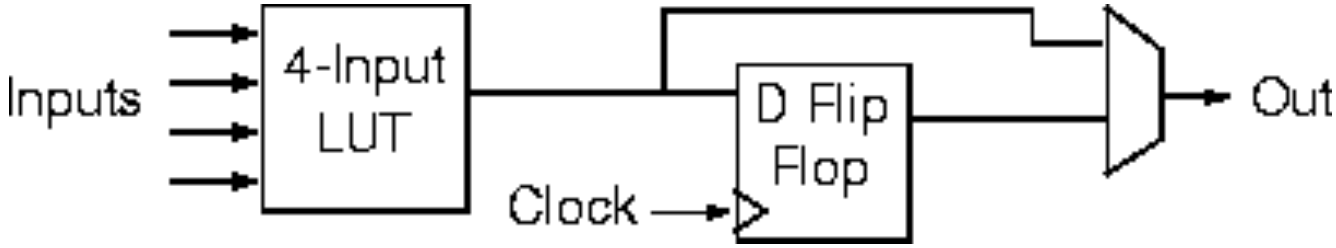
```
LW   rx, X
MPY  r1, rx, ra
LW   ry, Y
MPY  r2, ry, rb
ADD  r3, r1, r2
SW   r3, Z
```



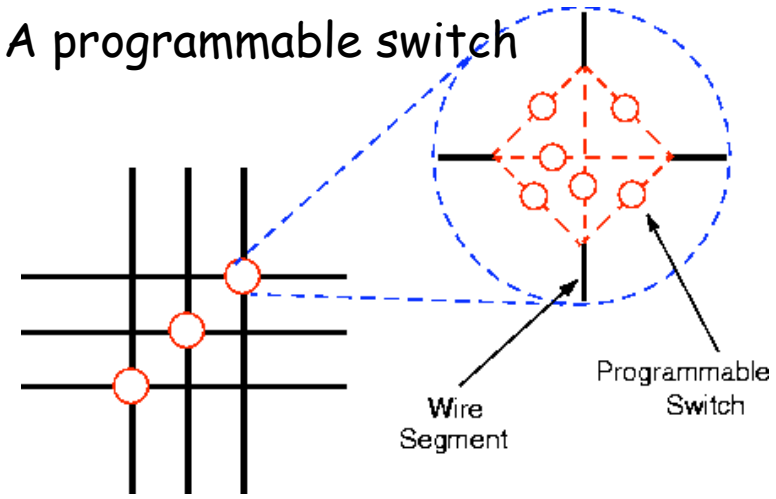
An FPGA Primer



Basic building block



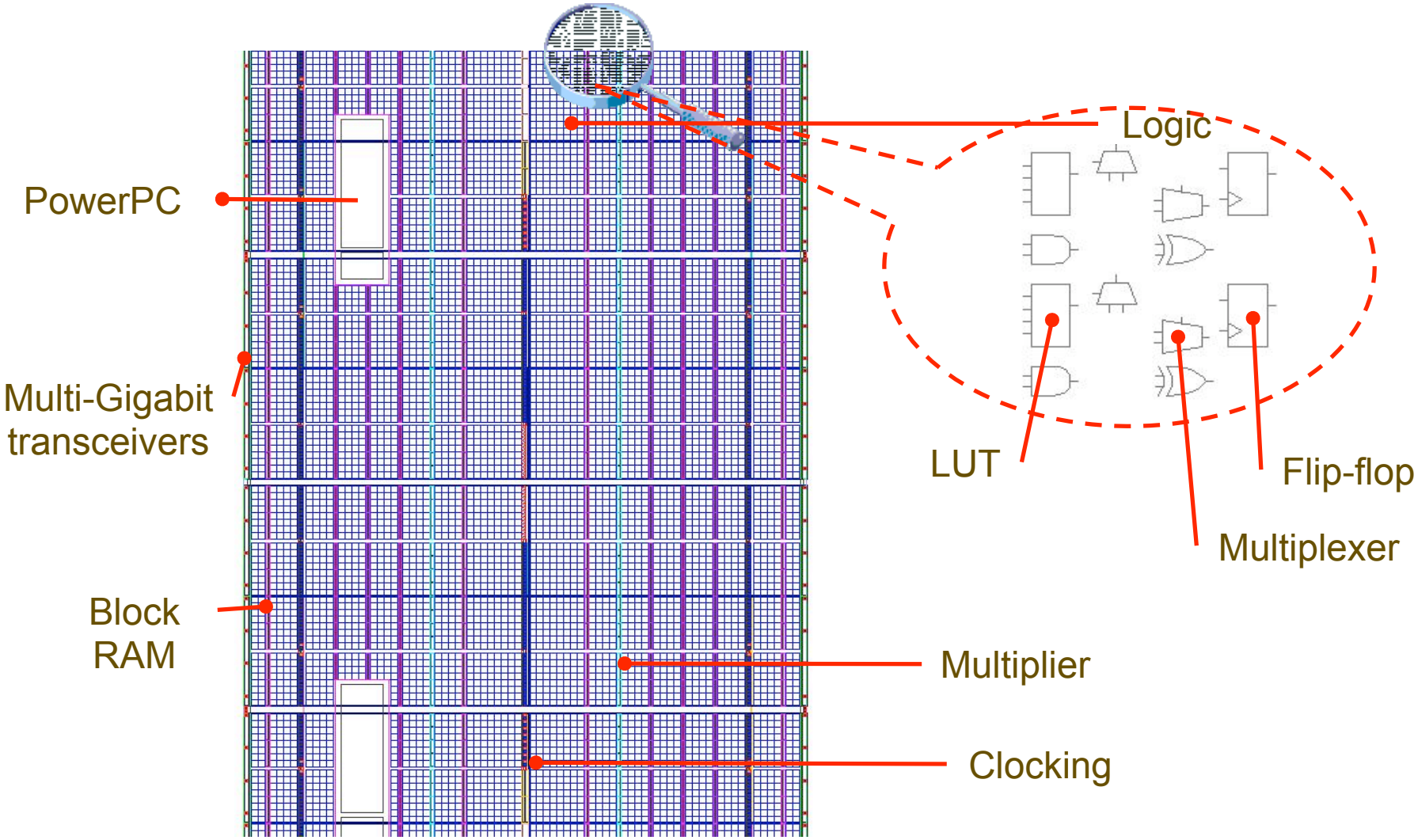
A logic cell, LUT: look-up table



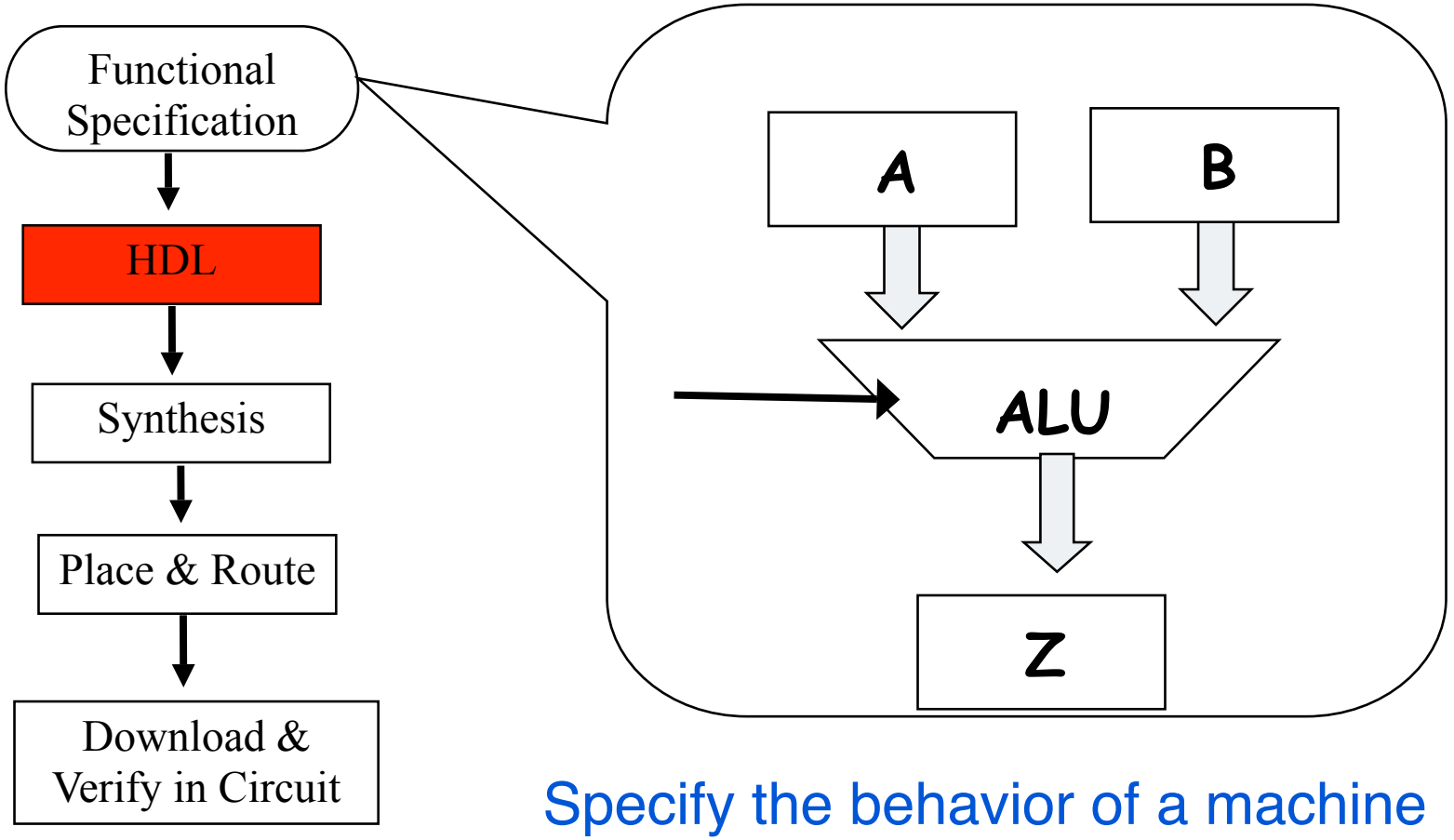
Xilinx Virtex 4 LX200:
> 200,000 logic cells
In 89,000 slices
96 DSP cores
500 MHz rated
0.7 MB on-chip BRAM

UC RIVERSIDE UNIVERSITY OF CALIFORNIA CSE

Field Programmable Gate Array

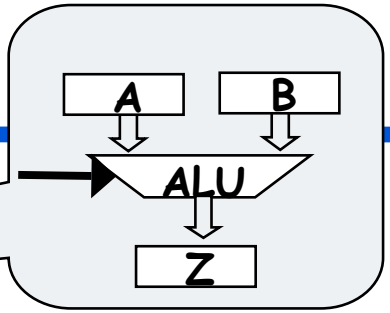


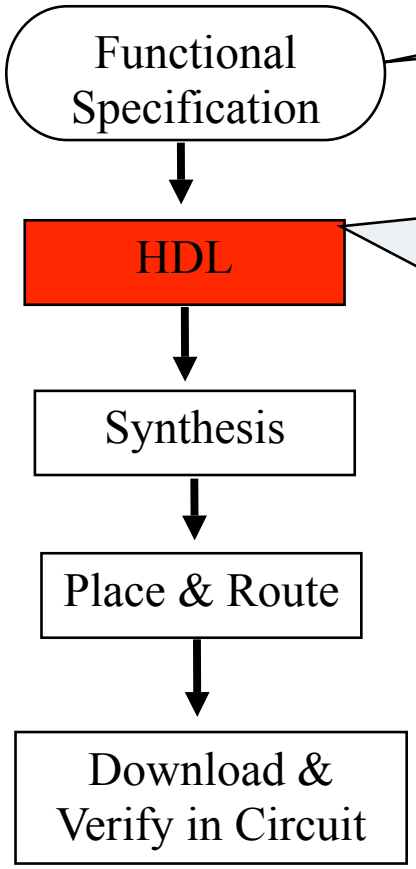
Programming FPGAs



Specify the behavior of a machine

Programming FPGAs

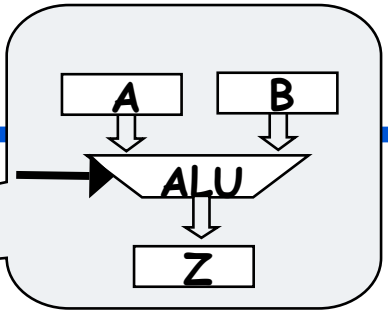





```
-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;
-- this is the entity
entity 2input_with_control is
  port (
    A : in std_logic;
    B : in std_logic;
    Z : out std_logic;
    Control: in std_logic);
end entity 2input_with_control;
-- here comes the architecture
architecture ALU of 2input_with_control is
-- Internal signals and components defined here
begin
     case Control is
          when '1' => Z <= A - B;
          when others => Z <= A + B;
end architecture ALU;
```

Hardware description language:
VHDL, Verilog, SystemC

Programming FPGAs



Functional Specification

HDL

Synthesis

Place & Route

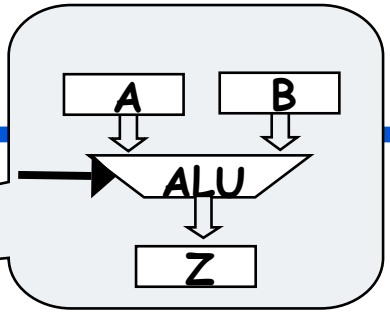
Download & Verify in Circuit

```
(cell yyy (cellType generic)
 (view schematic_ (viewType netlist)
  (interface
   (port CLEAR (direction INPUT))
   (port CLOCK (direction INPUT)) ... )
  (contents
   (instance I_36_1 (viewRef view1 (cellRef dff_4)))
   (instance (rename I_36_3 "I$3") (viewRef view1
              (cellRef addsub_4)))
   ...
   (net CLEAR
    (joined
     (portRef CLEAR)
     (portRef aset (instanceRef I_36_1))
     (portRef aset (instanceRef I_36_3))))
```

Synthesis tool: HDL to netlist format

(op (input list) (output list))

Programming FPGAs



Functional Specification

HDL

Synthesis

Place & Route

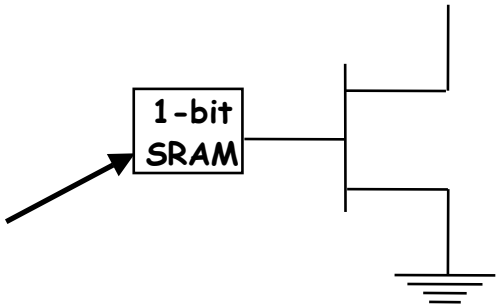
Download & Verify in Circuit

Which op on which logic cell in which slice?
Which switch should be open?
Which should be closed?

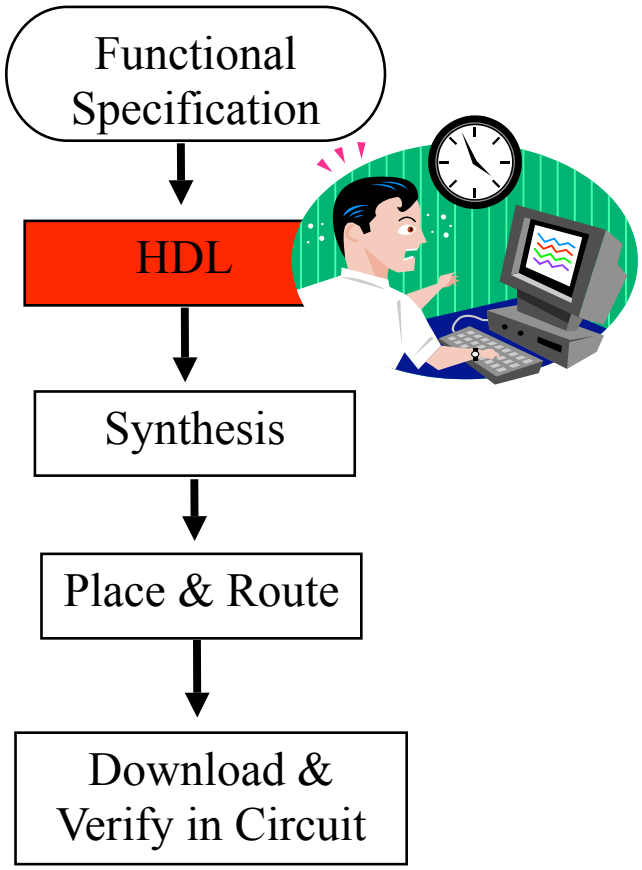
While (Minimize area and clock cycle time!)

NP-hard -- Simulated annealing,
large jobs take hours and days.

Place and route tool generates the bits that go here



Low-level Abstraction



- Clock-cycle level accuracy
- Tedious
- Error-prone
- Acquisition of the skill
  - Digital design background
  - Syntax
- Low-level design
- Low productivity

FPGA: A New HPC Platform?

David Strensky, *FPGAs Floating-Point Performance -- a pencil and paper evaluation*, in HPCwire.com

Comparing a dual core Opteron to FPGA on fp performance:

- Opteron: 2.5 GHz, 1 add and 1 mult per cycle. 2.5 x 2 x 2 = 10 Gflops
- FPGAs Xilinx V4 and V5 with DSP cores
  - Balanced allocation of dp fp adders, multipliers and registers
  - Use both DSP and logic for multipliers, run at lower speed
  - Logic for I/O interfaces

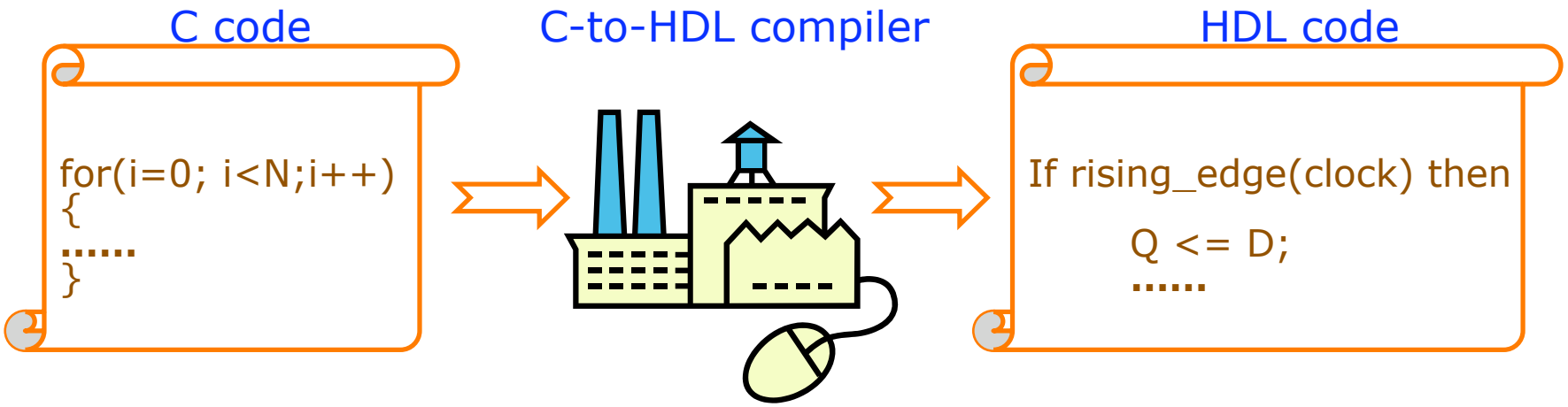
	Double Gflop/s		
	Opt	V-4	V-5
Macc	10	15.9	28.0
Mult	5	12.0	19.9
Add	5	23.9	55.3

Watts		
Opt	V-4	V-5
95	25	~35

- **Higher percentage of peak on FPGA (streaming)**
- **1/3 of the power!**

UC RIVERSIDE UNIVERSITY OF CALIFORNIA CSE

High-level Design Abstraction Desirable



- Higher design productivity
- Ease to learn
- System co-design

Challenge

- **FPGA is an amorphous mass of logic**

von Neumann machines	FPGAs
Temporal computing	Spatial computing
Sequential	Parallel
Centralized storage	Distributed storage
Control flow driven	Data flow driven

Outline of Talk

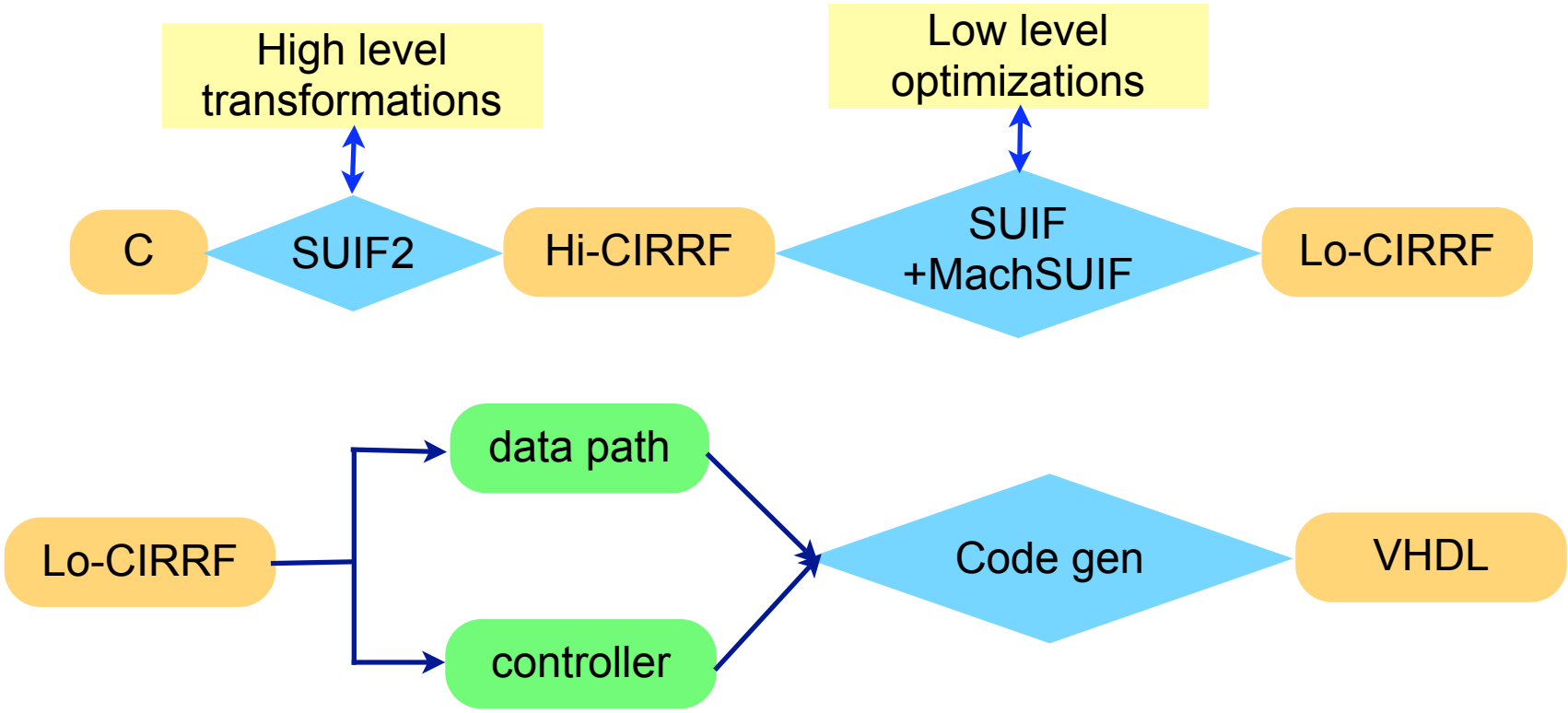
- INTRODUCTION
  - An FPGA Primer
  - Potentials of FPGAs as computing platforms
  - The programmability problem
- **EXPERIENCE WITH ROCCC 1.0**
  - Overview
  - Compiler optimizations
  - Applications
- DESIGN OF ROCCC 2.0
  - The “why” and “how”

ROCCC

Riverside Optimizing Compiler for Configurable Computing

- **Code acceleration**
  - By mapping of circuits to FPGA
  - Achieve 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

Overview of ROCCC



CIRRF
Compiler Intermediate Representation for Reconfigurable Fabrics

Limitations on the C code:
- No recursion
- No pointers
- Must have a loop

UC RIVERSIDE UNIVERSITY OF CALIFORNIA CSE

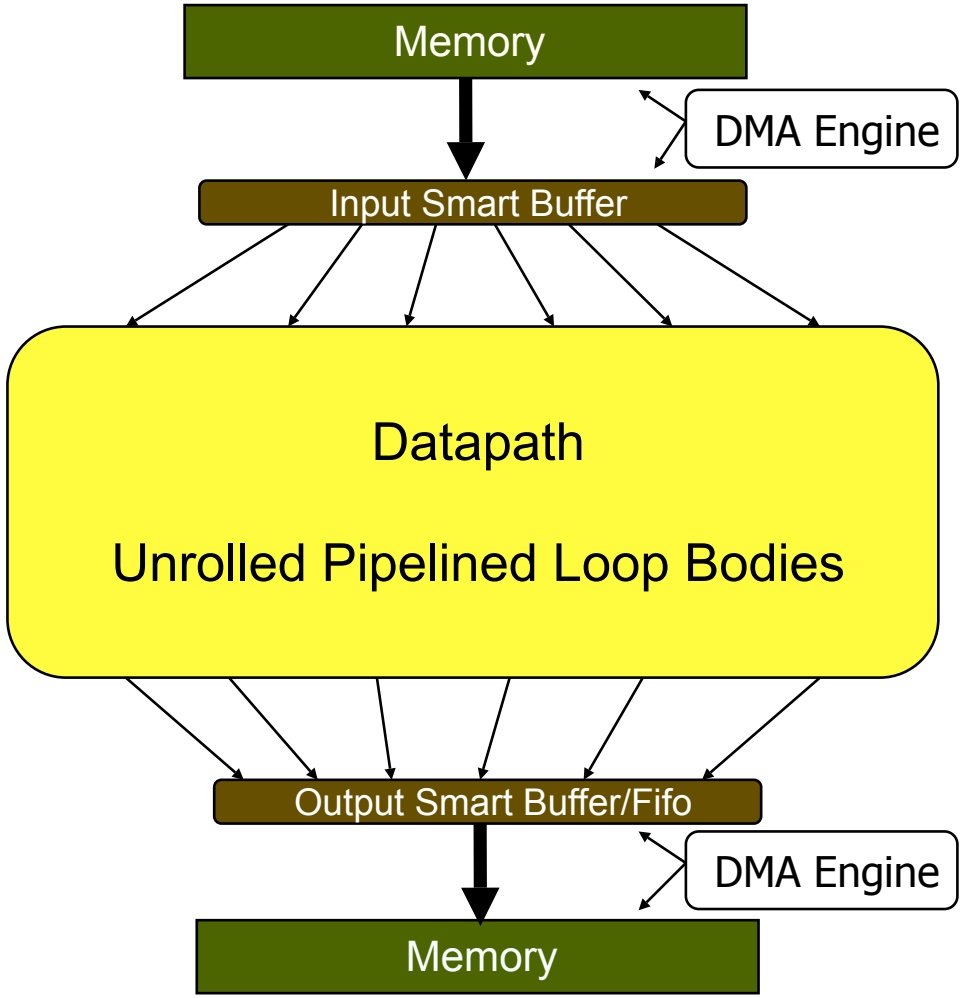
Focus

- **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

Generated Circuit Structure

- **Decoupled architecture**
  - Memory accesses separate from datapath instructions
  - Memory accesses configured by the compiler
  - Parallel loop bodies
  - Smart input buffer handles data reuse



Simple example

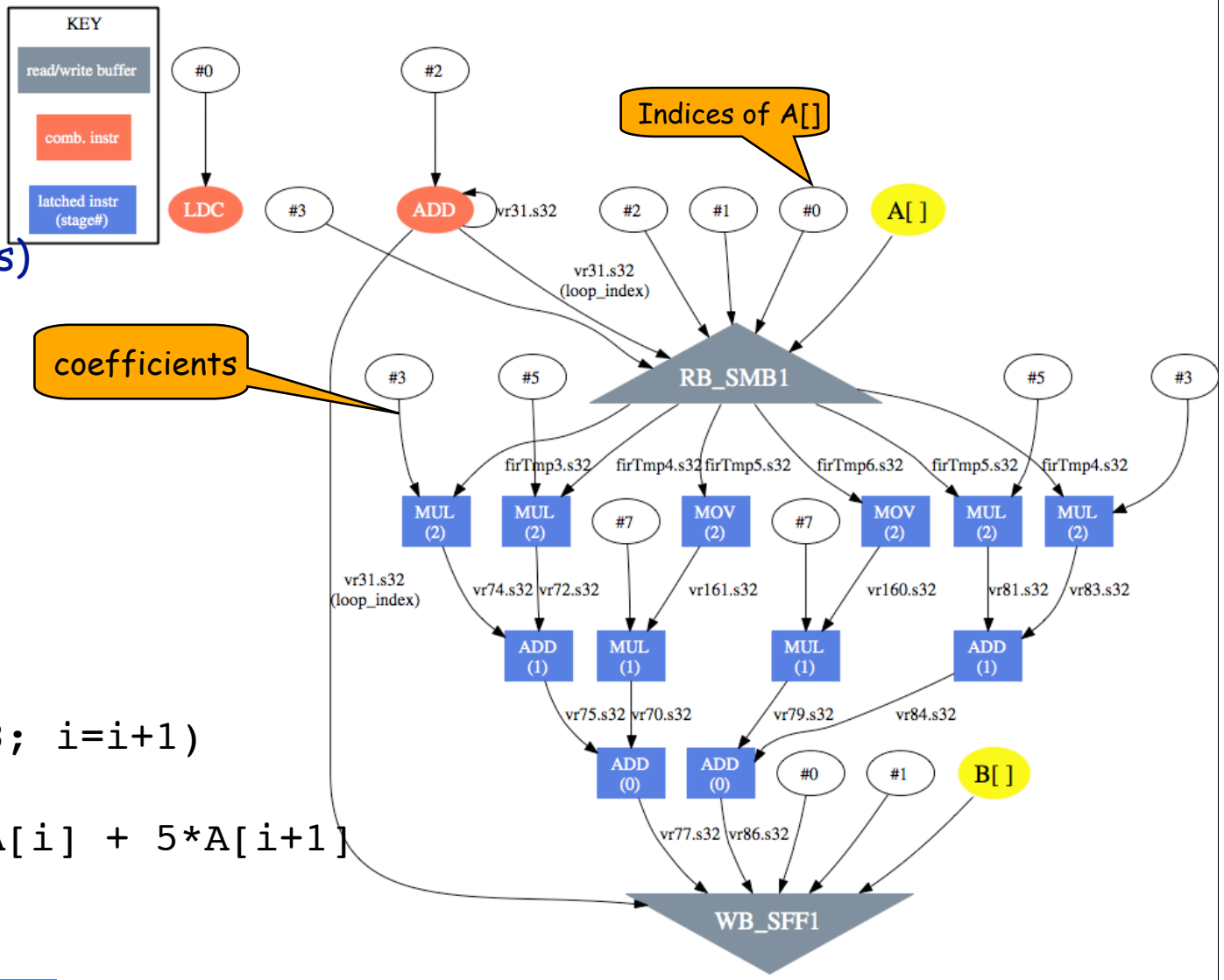
5-tap FIR

```
B[i] = 3*A[i] + 5*A[i+1] + 7*A[i+2]
```

```
#define N 516
void begin_hw();
void end_hw();
int main()
{
    int i;
    const int T[5] = {3,5,7,9,11};
    int A[N], B[N];
begin_hw();
L1: for (i=0; i<=(N-5); i=i+1)
   {
    B[i] = T[0]*A[i] + T[1]*A[i+1] + T[2]*A[i+2] + T[3]*A[i+3] +
    T[4]*A[i+4];
     }
end_hw();  }
```

Lo-CIRRF Viewer

Example: 3-tap FIR unrolled once (two concurrent iterations)




```
int main()
{
  int i;
  int A[32];
  int B[32];
  for (i=0; i<28; i=i+1)
    {
      B[i] = 3*A[i] + 5*A[i+1]
+ 7*A[i+2];
    }
}
```

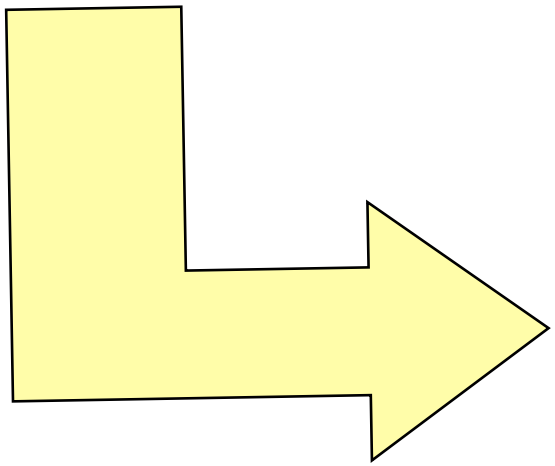
Example Of Additional Features

```
typedef ROCCC_int23 int ;
ROCCC_int23  x[100][50], y[50], z[50] ;
Loop_k: for (k=0 ; k<100; ++k)
{
  Loop_m: for (m=0; m < 50;++m)
  {
     x[k][m] = (y[m] + z[m])* k ;
  }
}
```

Compiler directives:

LoopInterchange Loop_m Loop_k

PartiallyUnroll Loop_k 4



```
typedef ROCCC_int23 int ;
ROCCC_int23  x[100][50], y[50], z[50] ;
Loop_m: for (m=0; m < 50;++m)
{
  Loop_k: for (k=0 ; k<100; ++k)
  {
     x[k][m] = (y[m] + z[m])*k ;
     x[k+1][m] = (y[m] + z[m])*(k+1) ;
     x[k+2][m] = (y[m] + z[m])*(k+2) ;
     x[k+3][m] = (y[m] + z[m])*(k+3) ;
  }
}
```

So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- Efficient code generation and pipelining
- Import of existing IP cores

So far, working compiler with …

- **Extensive compiler optimizations and transformations**
- **Analysis and hardware support for data reuse**
- **Efficient code ge**
- **Import of existin**
- **Experience with (DPR)**

•Loop, array & procedure transformations.
•Maximize clock speed & parallelism, within resources.
•Under user control.

B. A. Buyukkurt, et al. Impact of Loop Unrolling on Throughput, Area and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs, ARC 2006, Delft, The Netherlands.

UC RIVERSIDE UNIVERSITY OF CALIFORNIA CSE

High Level Transformations

Loop	Procedure	Array
• *Normalization* • *Invariant code motion* • *Peeling* • *Unrolling* • *Fusion* • *Tiling (blocking)* • *Strip mining* • *Interchange* • *Un-switching* • *Skewing* • *Induction variable substitution* • *Forward substitution*	• *Code hoisting* • *Code sinking* • *Constant propagation* • *Algebraic identities simplification* • *Constant folding* • *Copy propagation* • *Dead code elimination* • *Unreachable code elimination* • *Scalar renaming* • *Reduction parallelization* • *Division/multiplication by constant approximation* • *If conversion*	• *Scalar replacement* • *Array RAW/WAW elimination* • *Array renaming* • *Constant array value propagation* • *Feedback reference elimination*

So far, working compiler with …

- Extensive compiler optimizations and transformations
- **Analysis and hardware support for data reuse**
- Efficient code generation and pipelining
- Import of existir
- Experience with (DPR)

•Smart buffer technique reduces off chip memory accesses by > 90%

Z. Guo et al. Input Data Reuse In Compiling Window Operations Onto Reconfigurable Hardware, LCTES 2004.

So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hardware support for data reuse
- **Efficient code generation and pipelining**
- Import of existing IP cores
- Experience wit
  (DPR)

•Clock speed comparable to hand written HDL codes

Z. Guo et al. Optimized Generation of Data-Path from C Codes, DATE 2005.

So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and hard
- Efficient code ge
- **Import of existing IP cores**
- Experience with Dynamic Partial Reconfiguration (DPR)

- Huge wealth of existing IP cores.
- Wrapper makes core look like a function call in C code.

Z. Guo et al. Automation of IP Core Interface Generation for Reconfigurable Computing, FPL 2006.

So far, working compiler with …

- Extensive compiler optimizations and transformations
- Analysis and h
- Efficient code g
- Import of existing IP cores
- **Experience with Dynamic Partial Reconfiguration (DPR)**

•DPR allows reconfiguration of a subset of the FPGA, dynamically, under software control.
•Reduces configuration overhead.

A. Mitra et al. Dynamic Co-Processor Architecture for Software Acceleration on CSoCs, ICCD 2006.

Summary of Applications in ROCCC

- **Bioninformatics and Data Mining**
  - Dynamic programming code
  - Generate systolic array
- **Molecular Dynamics**
  - Heavy floating-point use, long pipeline, massive data
- **Image and video processing**
- **Encryption/decryption, CRC calculation**

- **Speedup:**
  - One to four orders of magnitude

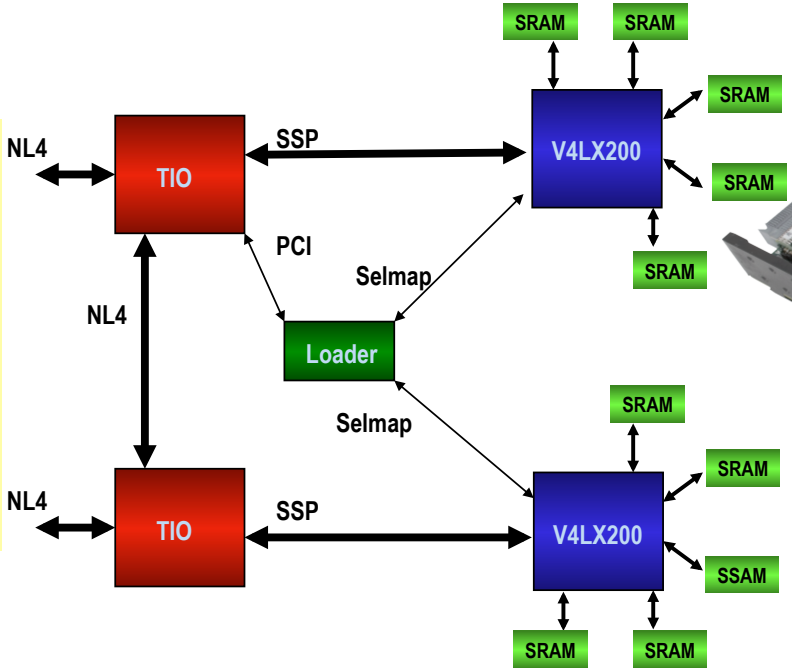
- **Experience**
  - ***Must redesign algorithms for spatial computing***

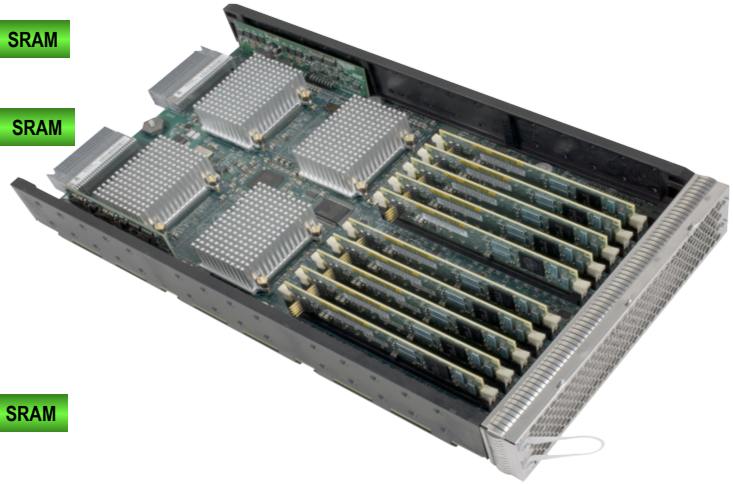
Platforms - SGI RASC RC100

10 TB single core shared memory

12.8GB/s

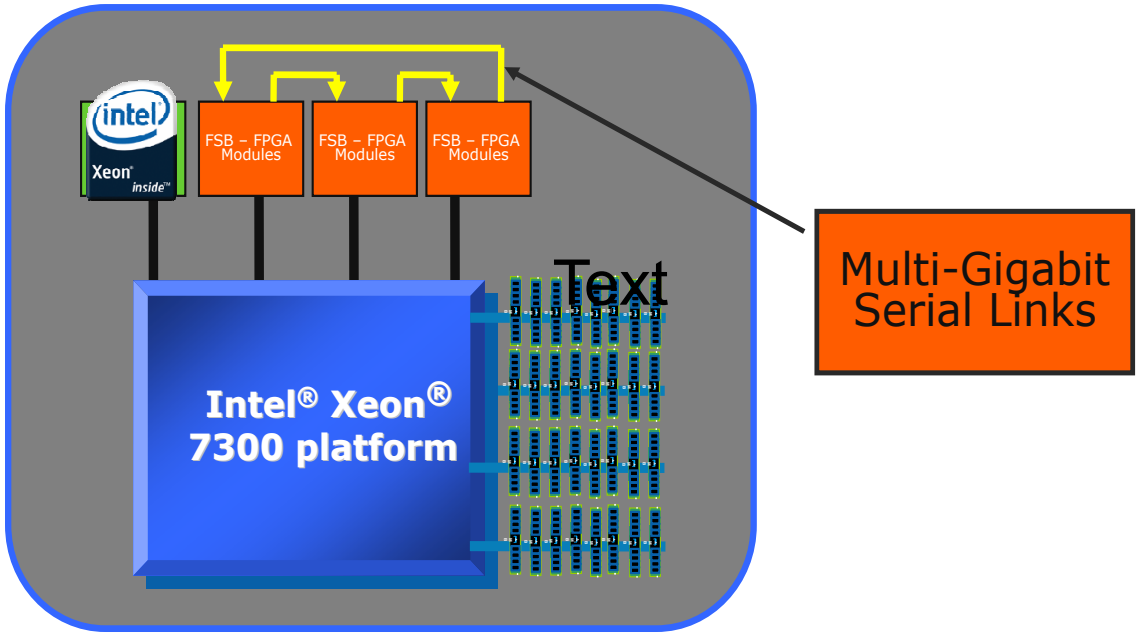
NUMA Link





Platforms - Intel FSB-FPGA

- Multiple FPGA modules can be connected in a ring topology
  - Partition complex algorithms across multiple accelerator modules
  - Higher degrees of parallelization for even higher performance
  - Connect to external I/O sources



Intel Developer FORUM

Flexible Hardware Configuration

7

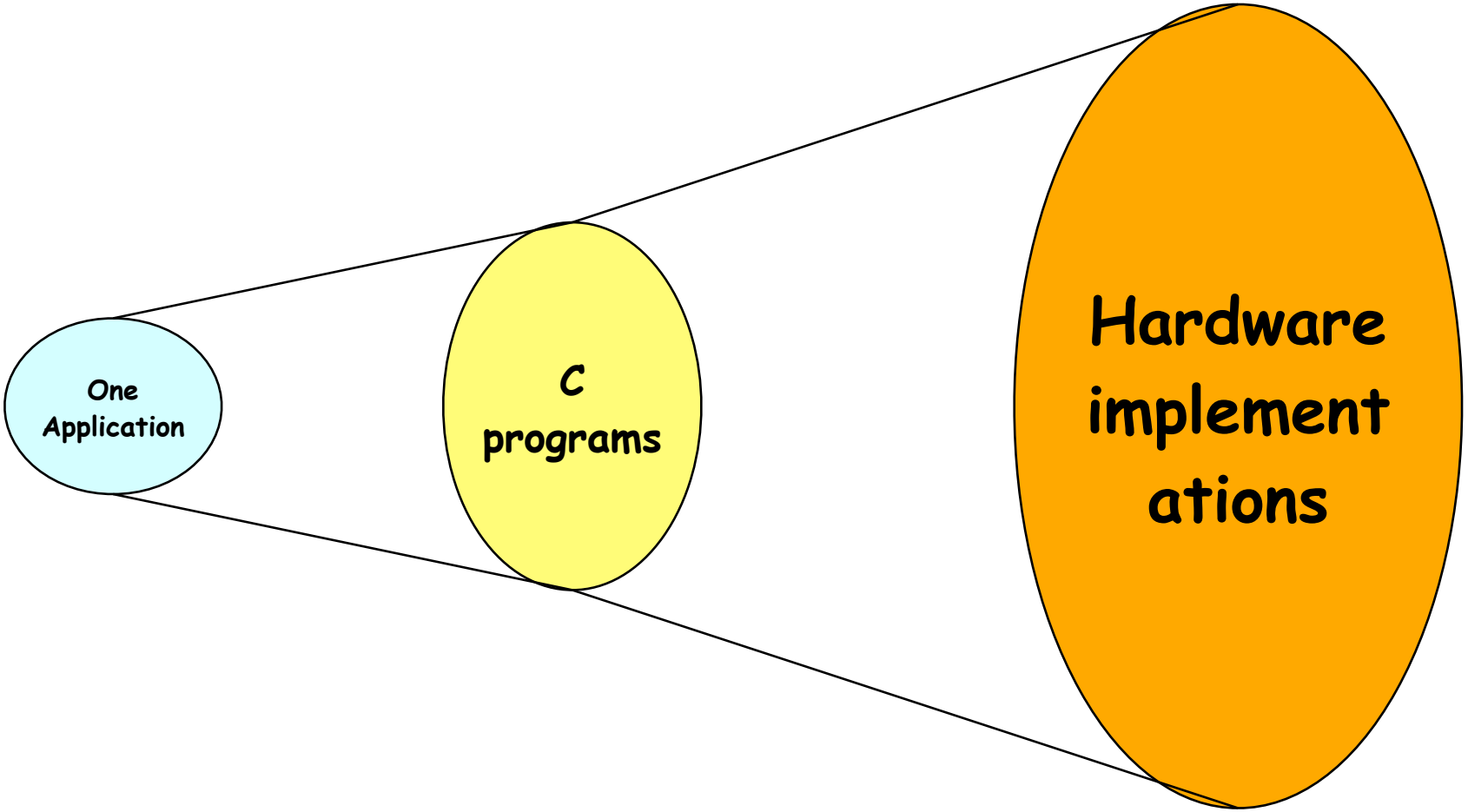
Outline of Talk

- INTRODUCTION
  - An FPGA Primer
  - Potentials of FPGAs as computing platforms
  - The programmability problem
- EXPERIENCE WITH ROCCC 1.0
  - Overview
  - Compiler optimizations
  - Applications
- **DESIGN OF ROCCC 2.0**
  - The “why” and “how”

Problems with ROCCC 1.0

- **Top down compilation approach**
  - Isolate the user from the details of the target platform
  - Works with CPUs: one underlying fundamental model, von Neumann
- **Complexity of platforms**
  - A plethora of platforms with varying capabilities
    - On board memories, I/O interfaces, firmware support etc.
    - Evolving FPGA architectures, a moving target
  - User is unaware of complexities of target platform ⇒ complexities are reflected in the compiler
- **We have a hardware efficient algorithm**
  - How do we express it in C?
  - User must navigate hardware design space using compiler transformations: compiler technology not suitable for this

From Applications to Hardware



Why?

Example: two nested loops accessing two arrays

Design space explosion

Merge arrays?

Fuse the loops?

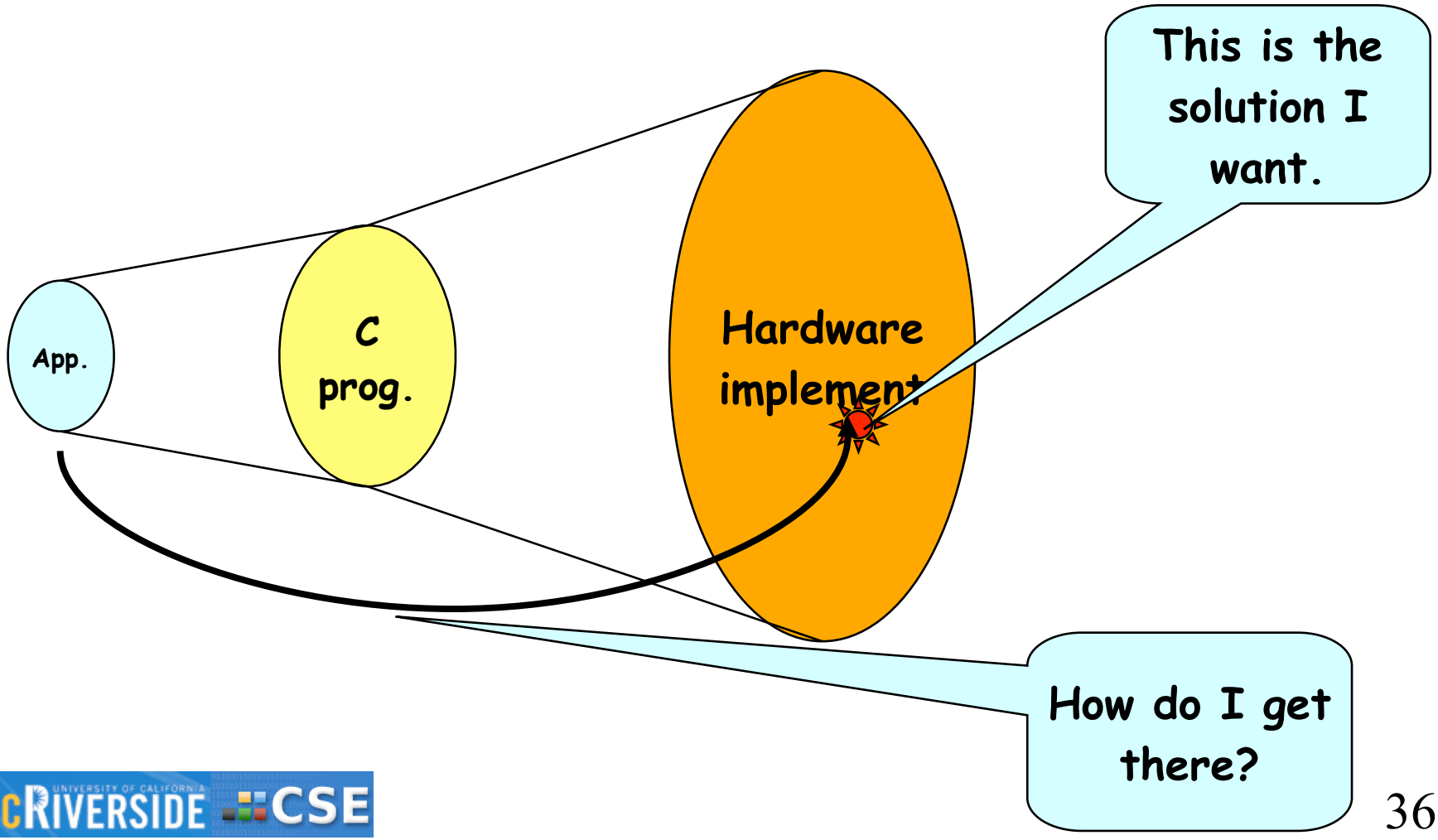
Partial or full unroll?

Multiple clock domains?

Unroll loop 1 or loop 2?

Access arrays in chunks or as streams?

Hardware Specification



Next Generation: ROCCC 2.0

- **Goals:**
  - Give more control of generated structure to designers
    - Build hardware systems in C from the bottom up
    - Description of components and interconnections using a C subset
  - Still maintain optimizations for hardware from ROCCC 1.0
- **Two objectives:**
  - *Modularity* and *composability*
  - Keeping the positives of ROCCC 1.0
- **How**
  - Compile standalone C functions to HDL modules
  - Import pre-existing cores
    - IP or pre-compiled
  - Separate platform specific interfaces from algorithm codes
    - These can be other modules too
    - Multiple interfaces possible in each platform

ROCCC 2.0 Design Flow

- **Conceive hardware algorithm**
- **Build circuit *bottom-up***
  - Until design is done, do:
    - Write a function in C
      - May import other modules
    - Generate a new HDL module
    - Test, evaluate, verify
    - Repeat
  - Add platform interfaces
- **Final evaluation on platform**

Example: Matrix multiplication

- Many, many hardware algorithms
- All use *macc* operators (multiply accumulate)

❑ **Design flow**

- Build a *macc* module
  - Use or augment an existing IP core
  - Build from scratch using + and *
- Build *Vvmult,* using macc
- Build *Vmmult,* using *Vvmult*
- Build *Mmmult,* using *Vmmult*

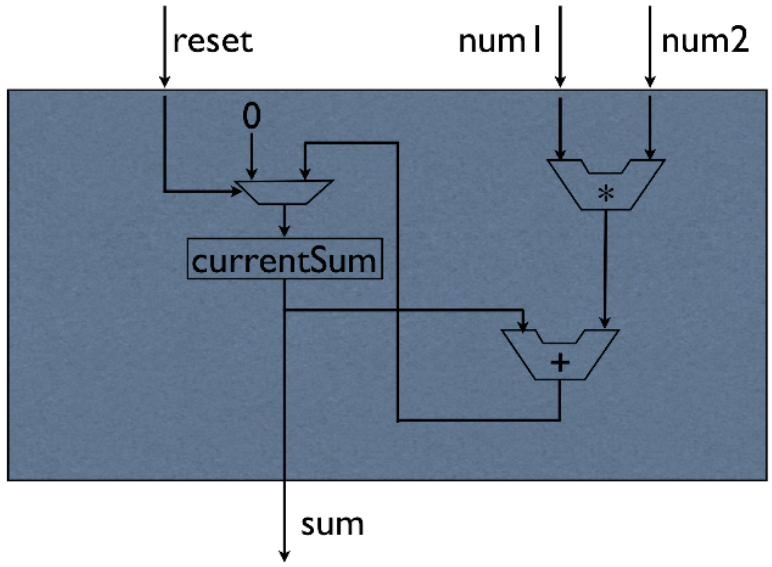
Multiple opportunities for parallelism

Approach & Plan

- **Open source software**
  - International collaborative effort
- **Initial version -- only C**
  - Based on existing ROCCC tool set
  - Can be completed in about 6 months
- **Second version -- support C++**
  - C functions: no state
  - C++ classes have state
- **Ultimate goal**
  - Understand what a language would look like

ROCCC 2.0 Example

```
typedef struct
{
  // Inputs
  int reset_in ;
  float num1_in ;
  float num2_in ;

  // Outputs
  float sum_out ;

  // State
  float currentSum_state ;
} MACC_t ;

MACC_t MACC(MACC_t m)
{
  if (m.reset_in == 1)
  {
    m.currentSum_state = 0 ;
  }
  else
  {
    m.currentSum_state += m.num1_in * m.num2_in ;
  }
  m.sum_out = m.currentSum_state ;
  return m ;
}
```



Modules in ROCCC 2.0

- **Module: hardware equivalent of a procedure**
  - Can exist as:
    - C code,
    - VHDL/Verilog code
    - Hardware macro (FPGA specific circuit)
  - Can be imported into other C modules or C code in any of these forms
- **Importing of Cores in ROCCC 2.0**
  - Core can be user generated or imported
  - Once synthesized, C stub functions can be called from other functions
  - Cores can be integrated in the same way and treated as black boxes
  - Compiler can automatically replicate black boxes and connect I/O

MACC Example

```
/**** MACC.c *****/
typedef struct{
// Inputs
      int num1_in, num2_in,
      int num3_in;
// Outputs
int sum_out;
// State variables
} MACC_t;

MACC_t  MACC(MACC_t m)
{
m.sum_out = (m.num1_in*m.num2_in) +
      m.num3_in;
return m;
}
/***** End MACC.c *****/
```

```
/******* Calling.c *******/
#include "roccc-library.h"
typedef struct{
int A_in[100]; int B_in[100];
int final_out;
} CalltoMACC;

CalltoMACC MACC_Call(Calling_t x){
MACC_t aMACC; int i;

   for(i=0; i<100;++i) {
   aMACC.num1_in = x.A_in[i];
   aMACC.num2_in = x.B_in[i];
   aMACC.num3_in =

         aMACC.sum_out;
MACC(aMACC);
}

x.final_out = aMACC.sum_out;
return x;
}
```

FIR Example

```
/******* Fir.c  ********/
typedef struct{
    int A_in[516];
    int B_out[516];
} FIR_t;

FIR_t FIR(FIR_t f){
    int i;
    const int T[5] = {3, 5, 7, 9, 11};
    for(i = 0 ; i < 512; ++i) {
      f.B_out[i] = T[0]*f.A_in[i] + T[1]*f.A_in[i+1] +
    T[2]*f.A_in[i+2] + T[3]*f.A_in[i+3] + T[4]*f.A_in[i+4];
    }
return f;
}
/******* End Fir.c ******/
```

FLOAT ALU Example

```
/******* FloatALU  *******/
typedef struct{
    float num1_in;
    float num2_in;
    int select_in;
    float result_out;
} FloatALU_t;

FloatALU_t FloatALU(FloatALU_t f)
{
if (f.select_in == 1)
    {f.result_out = f.num1_in*f.num2_in;}
else
    {f.result_out = f.num1_in + f.num2_in;}

return f;
}
/******* End FloatALU ******/
```

Status of ROCCC 2.0

- **Open source project**
- **Support:**
  - National Science Foundation
  - Matrixware Inc. (Vienna, Austria)
  - Cisco
- **Currently working on v0.1**
  - C based, using the same framework as ROCCC 1.0
  - Limited release October 2008
- **Version (0.2)**
  - Move to better and newer compiler framework (December 2008)
- **Future**
  - Move to C++

Conclusion

- **C to spatial code compilation is possible**
  - without recourse to “parallelizing” constructs
- **Top down approach is impractical**
  - combine bottom up module and top down compiler transformations
- **No “standard” or commonly used platform yet**
  - until then we need to adapt: execution model, interfaces etc.

- **A new language for spatial computing?**
  - YES! but
  - design based on actual experience with applications, platforms, compiler transformmations
  - not idle cogitation

Automatic Systolic Array Generation from C Descriptions of Dynamic Programming Algorithms in ROCCC

Betul Buyukkurt

Walid A. Najjar

Department of Computer Science & Engineering

University of California Riverside

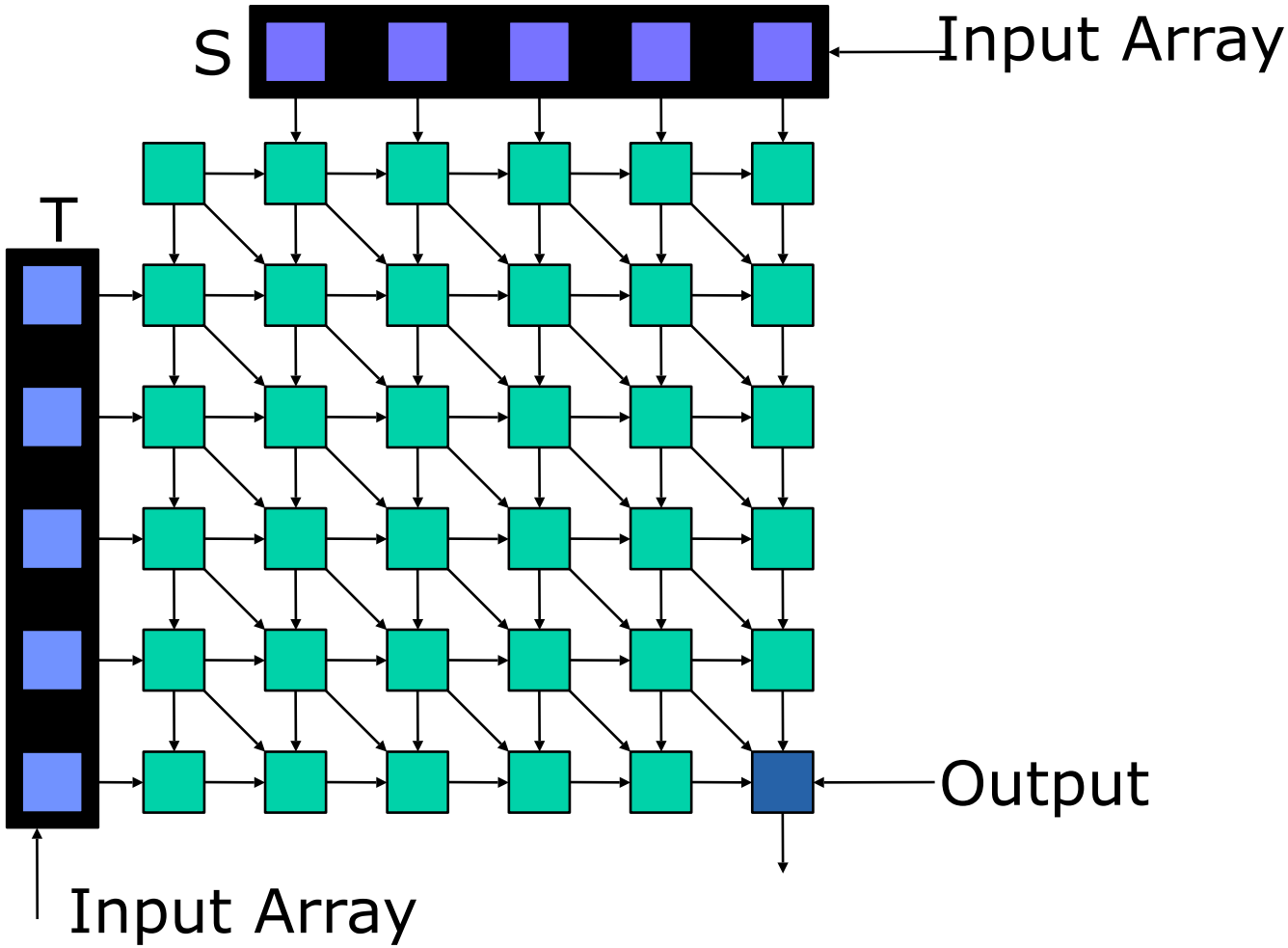
Smith Waterman Algorithm

- A dynamic programming string matching algorithm
  - used widely in sequence matching applications
- Computes a matching score of two input strings S and T using a 2D matrix
  - Computation of each cell depends on the computed values of three neighboring cells: north, west and northwest

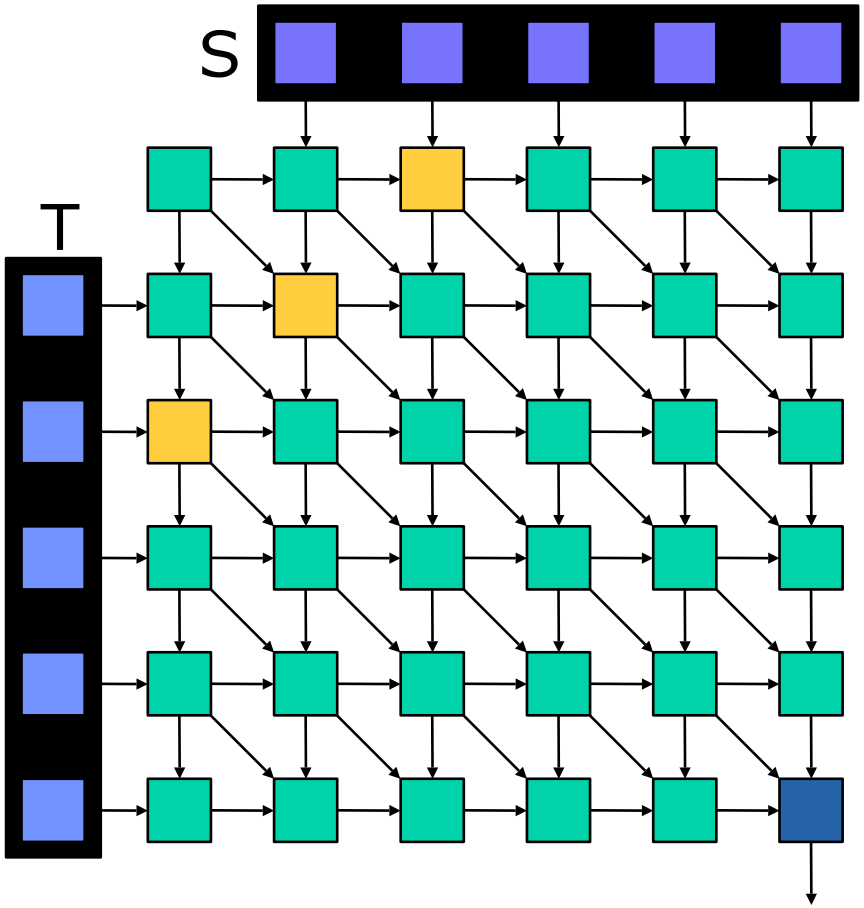
	S_i		
T_j	a	b	...
	c	d	...
	...	...	...

$$d = \min \begin{cases} a & \text{if } (S_i == T_j) \\ a + \text{substitution cost} & \text{if } (S_i != T_j) \\ b + \text{insertion cost} & \\ c + \text{deletion cost} & \end{cases}$$

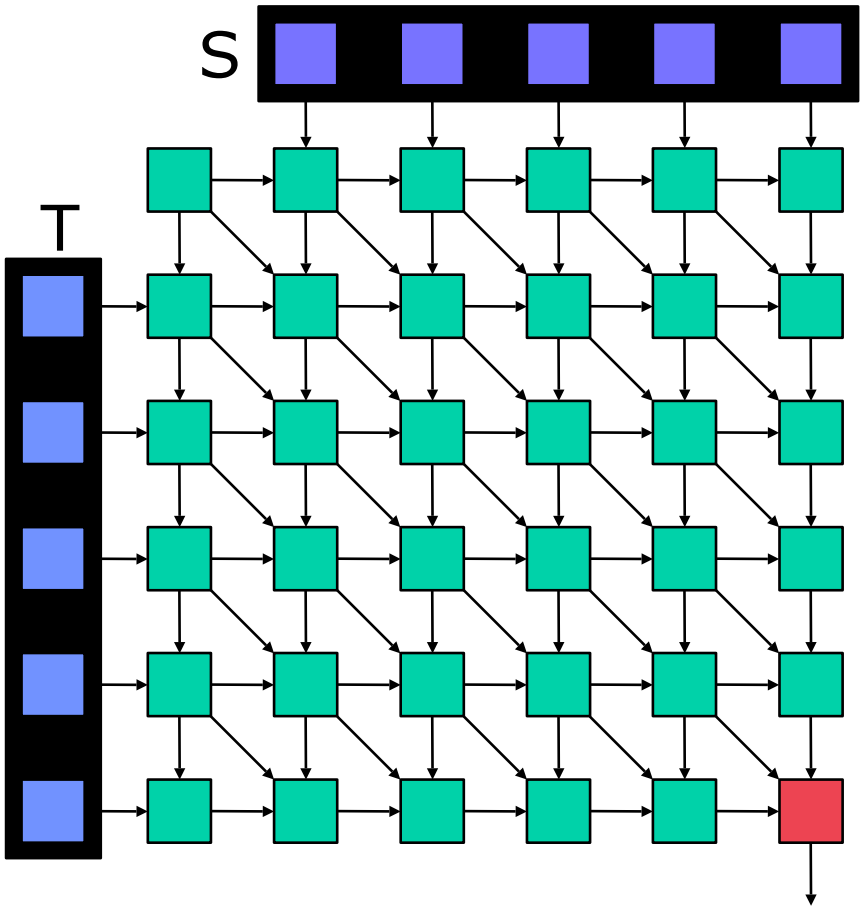
Systolic Array Generation



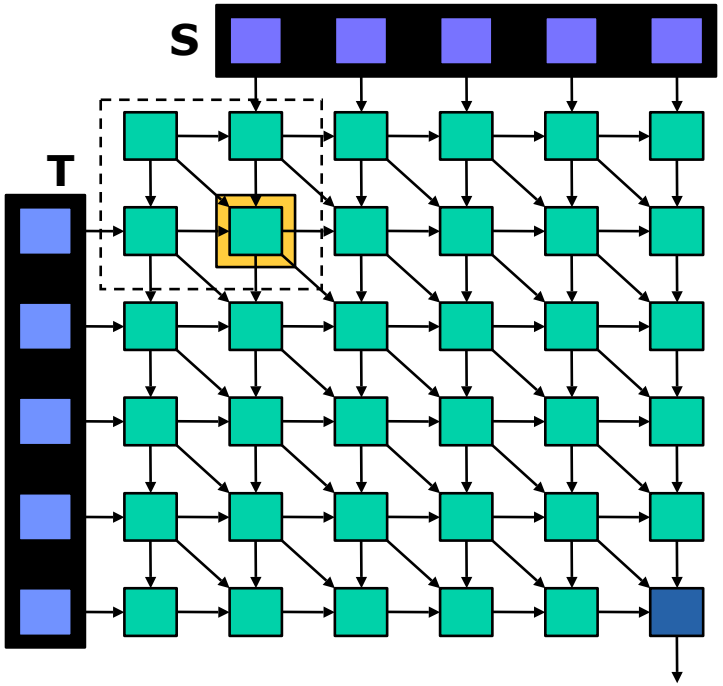
Systolic Array Generation



Systolic Array Generation

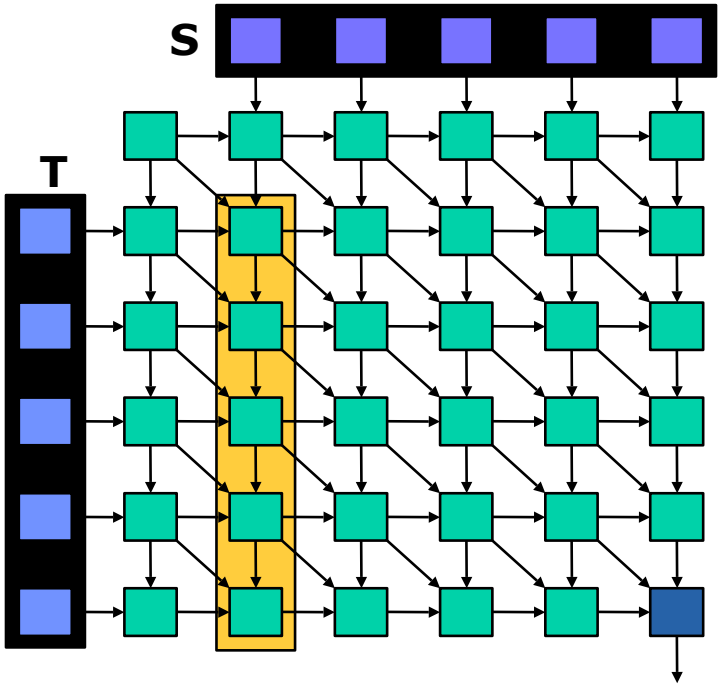


ROCCC Implementation



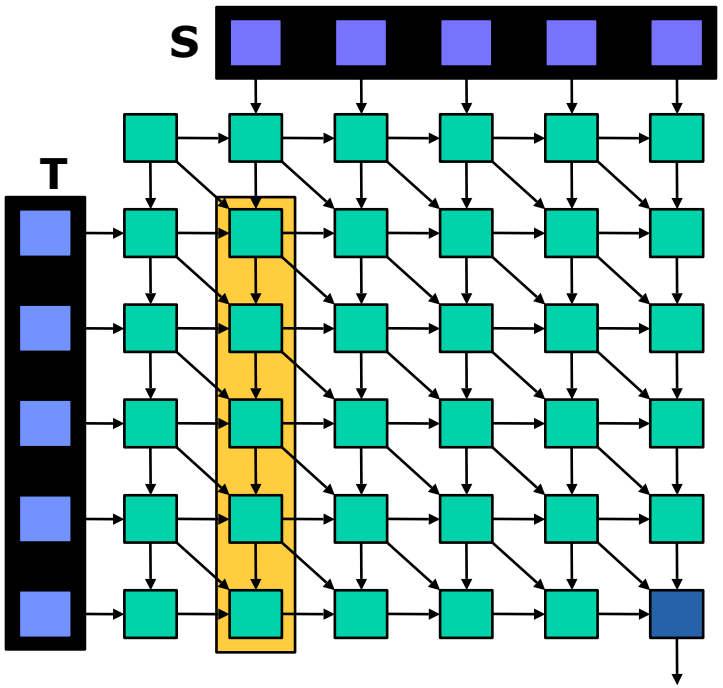

```
begin_hw();

for(i=1; i<N; i=i+1)
   for(j=1; j<N; j=j+1){
       A[i][j] = F(A[i-1][j],
                   A[i][j-1],
                   A[i-1][j-1],
                   T[i-1],
                   S[j-1]);
   }

end_hw();
```

Loop Unrolling



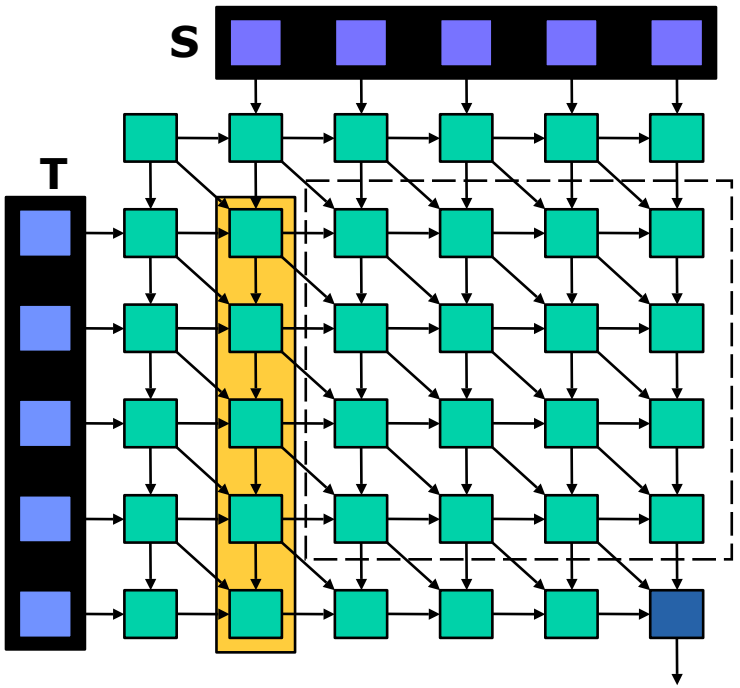

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
        A[i][j] = F(…);
        A[i+1][j] = F(…);
        A[i+2][j] = F(…);
        …
        A[i+k-1][j] = F(…);
   }
```

Scalar Replacement



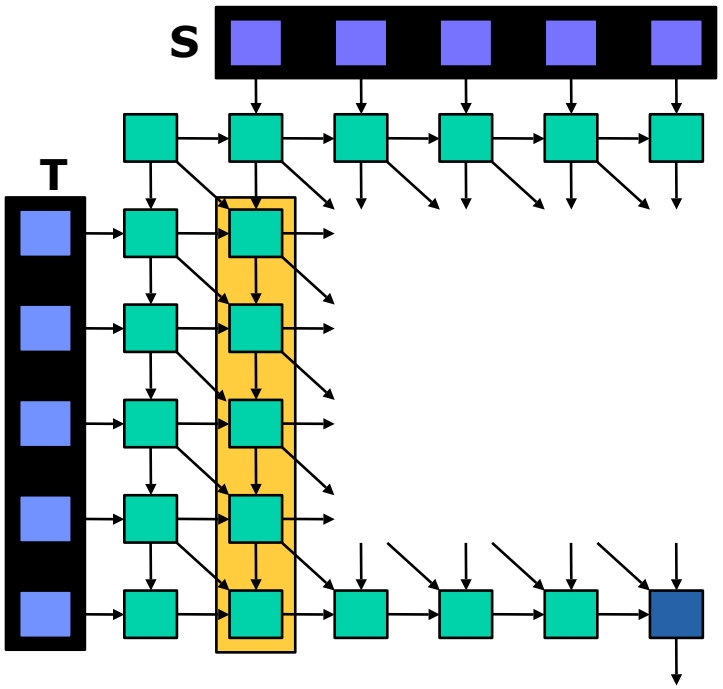

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
        a00 = A[i-1][j-1];
        a01 = A[i-1][j];
        a10 = A[i][j-1];
        a20 = A[i+1][j-1];
        t0 = T[i-1]; s0 = S[j-1];

        …
        a11 = F(a00,a01,a10,t0,s0);
        a21 = F(a10,a11,a20,t1,s0);

        …
        ak1 = F(am0,am1,ak0,tm,sm);

        A[i][j] = a11;
        A[i+1][j] = a21;

        …
        A[i+k-1][j] = ak1;
   }
```

Feedback Store Elimination



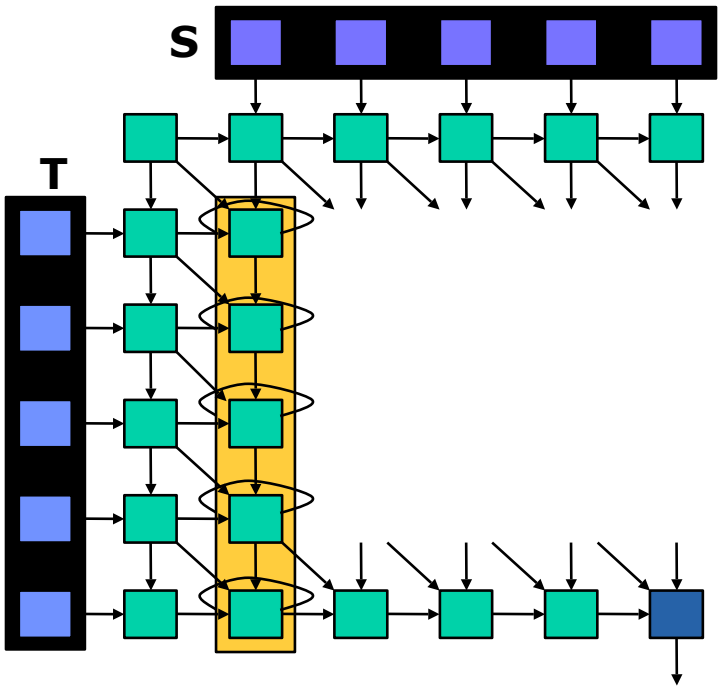

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
        a00 = A[i-1][j-1];
        a01 = A[i-1][j];
        a10 = A[i][j-1];
        a20 = A[i+1][j-1];
        t0 = T[i-1]; s0 = S[j-1];

        …
        a11 = F(a00,a01,a10,t0,s0);
        a21 = F(a10,a11,a20,t1,s0);

        …
        ak1 = F(am0,am1,ak0,tm,sm);

        A[i][j] = a11;
        A[i+1][j] = a21;

        …
        A[i+k-1][j] = ak1;
   }
```

Feedback Store Elimination



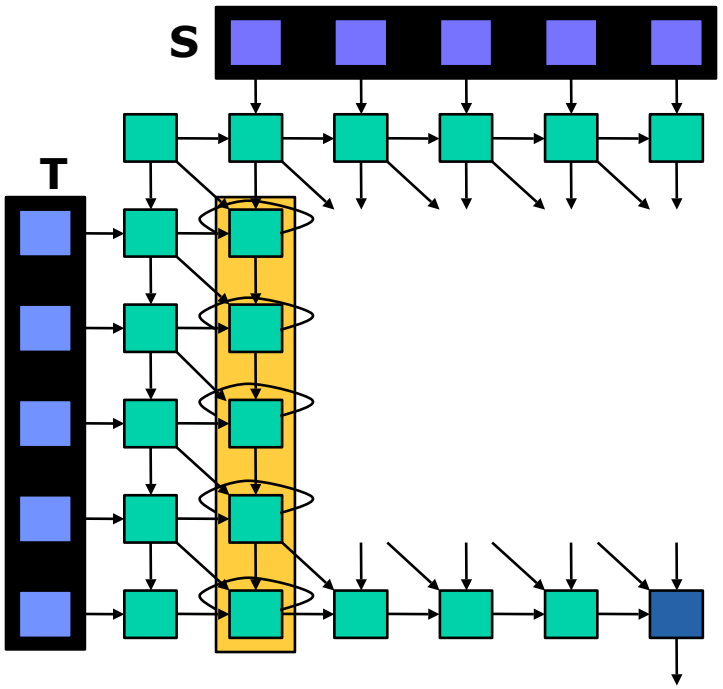

```
for(i=1; i<N; i=i+k)
   for(j=1; j<N; j=j+1){
        a00 = A[i-1][j-1];
        a01 = A[i-1][j];
        a10 = ;
        a20 = ;
        t0 = T[i-1]; s0 = S[j-1];

        …
        a11 = F(a00,a01,a10,t0,s0);
        a12 = F(a01,a11,a20,t1,s0);

        …
        a1k = F(a0k,amk,ak0,tm,sm);

         = a11;
         = a12;

        …
        A[i+k-1][j] = ak1;
   }
```

Feedback Store Elimination



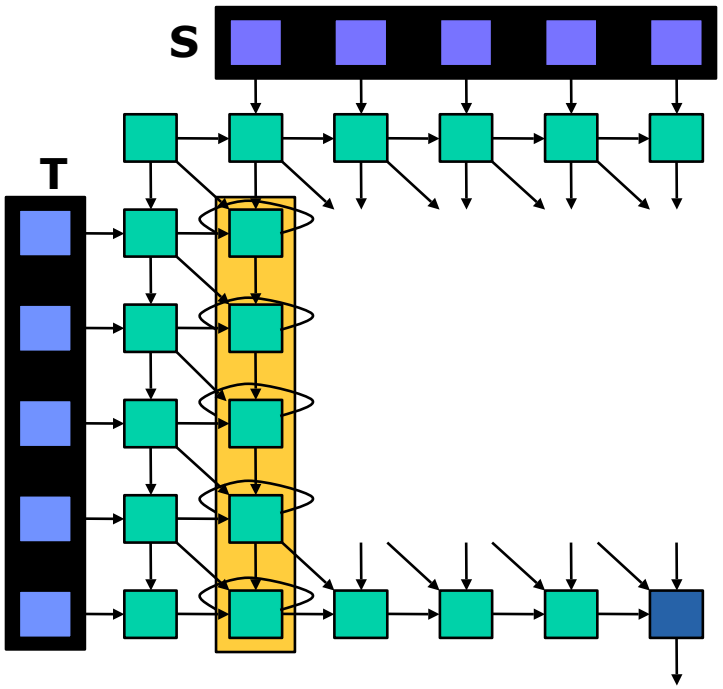

```
for(i=1; i<N; i=i+k){
    x11 = A[i][0];
    x12 = A[i+1][0];

    …
    for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = x11;
       a20 = x12;
       t0 = T[i-1]; s0 = S[j-1];

       …
       x11 = a11;
       x12 = a12;

       …
       A[i+k-1][j] = ak1;
    }
}
```

Loop Invariant Code Motion




```
for(i=1; i<N; i=i+k){
    x11 = A[i][0];
    x12 = A[i+1][0]; …
    t0 = T[i-1]; …
    for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = x11;
       a20 = x12;
       s0 = S[j-1];…
       …
       x11 = a11;
       x12 = a12;
       …
       A[i+k-1][j] = ak1;
    }
}
```

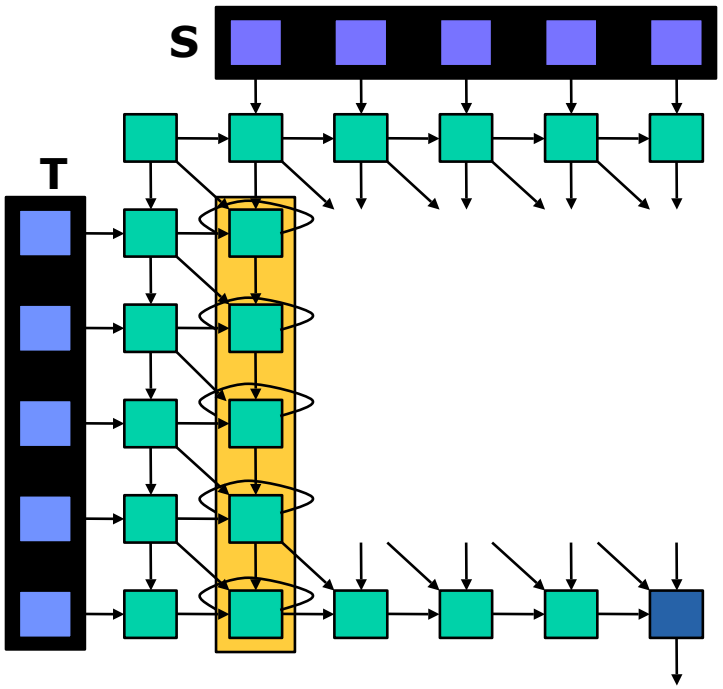
Output Generation




```
for(i=1; i<N; i=i+k){
    x11 = A[i][0];
    x12 = A[i+1][0];
     …
    t0 = T[i-1]; …
    for(j=1; j<N; j=j+1){
       a00 = A[i-1][j-1];
       a01 = A[i-1][j];
       a10 = x11;
       a20 = x12;
       s0 = S[j-1];…
        …
       x11 = a11;
       x12 = a12;
        …
       A[i+k-1][j] = ak1;
    }
}
```

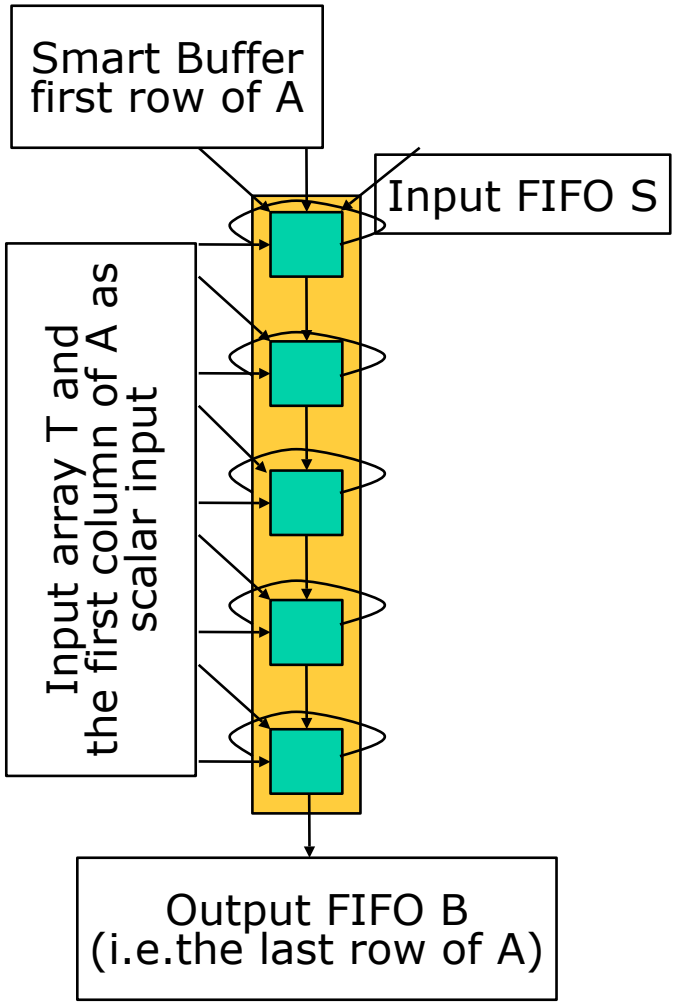
Hardware process

Host Process in HI-CIRRF




```
for(j=1; j<N; j=j+1){
    ROCCC_init_inputscalar(x11,x12, …
                           t0, …);
    ROCCC_smartbuffer1(A, j,-1, a00,
                              0, a01);
    ROCCC_input_fifo1(S, j, -1, s0);
    a10 = ROCCC_load_prev(x11);
    a20 = ROCCC_load_prev(x12);

    …
    a11 = F(a00,a01,a10,t0,s0);
    a21 = F(a10,a11,a20,t1,s0);

    …
    ak1 = F(am0,am1,ak0,tm,sm);

    ROCCC_store2next(x11, a11);
    ROCCC_store2next(x12, a12);

    …
    ROCCC_output_fifo(B, j, 1, ak1);
}
```

Host Process in HI-CIRRF




```
for(j=1; j<N; j=j+1){
    ROCCC_init_inputscalar(x11,x12, …
                           t0, …);
    ROCCC_smartbuffer1(A, j,-1, a00,
                       0, a01);
    ROCCC_input_fifo1(S, j, -1, s0);
    a10 = ROCCC_load_prev(x11);
    a20 = ROCCC_load_prev(x12);

    …
    a11 = F(a00,a01,a10,t0,s0);
    a21 = F(a10,a11,a20,t1,s0);

    …
    ak1 = F(am0,am1,ak0,tm,sm);

    ROCCC_store2next(x11, a11);
    ROCCC_store2next(x12, a12);

    …
    ROCCC_output_fifo(B, j, 1, ak1);
}
```

Systolic Array Generation Results

SW Results

	Intel Xeon	Intel Itanium-2	200-cells, 2.8% area	512-cells, 6.9% area	1024-cells, 17% area
Speed	2.8 GHz	1.5 GHz	191 MHz	188 MHz	174 MHz
GCUPS	0.049	0.084	38.2	96.25	178.65
Speedup over Xeon	1	1.7	780	1964	3646

DTW Results

	Intel Xeon	Intel Itanium-2	256-cells datasize, 24-bit 73% area	512-cells datasize, 12-bit 61% area
Speed	2.8 GHz	1.5 GHz	42 MHz	52 MHz
GCUPS	0.049	0.084	10.8	26.6
Speedup over Xeon	1	1.7	385	950

Systolic Array Generation Results

System	Number of Chips	PEs per chip	System Performance (CUPS)	Device Performance (CUPS)	Run-time reconfiguration requirement
Splash(XC3090)	32	8	370 M	11 M	No
Splash 2(XC4010)	16	14	43 B	2,687 M	No
SAMBA(XC3090)	32	4	1,280 M	80 M	No
Paracel(ASIC)	144	192	276 B	1,900 M	N/A
Celera (software implementation)	800	1	250 B	312 M	N/A
JBits (XCV1000-6)	1	4,000	757 B	757 B	Yes
JBits (XC2V6000-5)	1	11,000	3,225 B	3,225 B	Yes
HokieGene (XC2V6000-4)	1	7000	1,260 B	1,260 B	Yes
This implementation (XCV1000-6)	1	4,032	742 B	742 B	No
This implementation (XCV1000E-6)	1	4,032	814 B	814 B	No
ROCCC	**1**	**1024**	**178.65B**	**178.65B**	**No**
ROCCC Extp.	**1**	**4096**	**714.6B**	**714.6B**	**No**