Mapping Brook Stencils to Imagine

Jacob Chang
Nathan Hill
Jae-Wook Lee
Alex Solomatnikov

Motivation

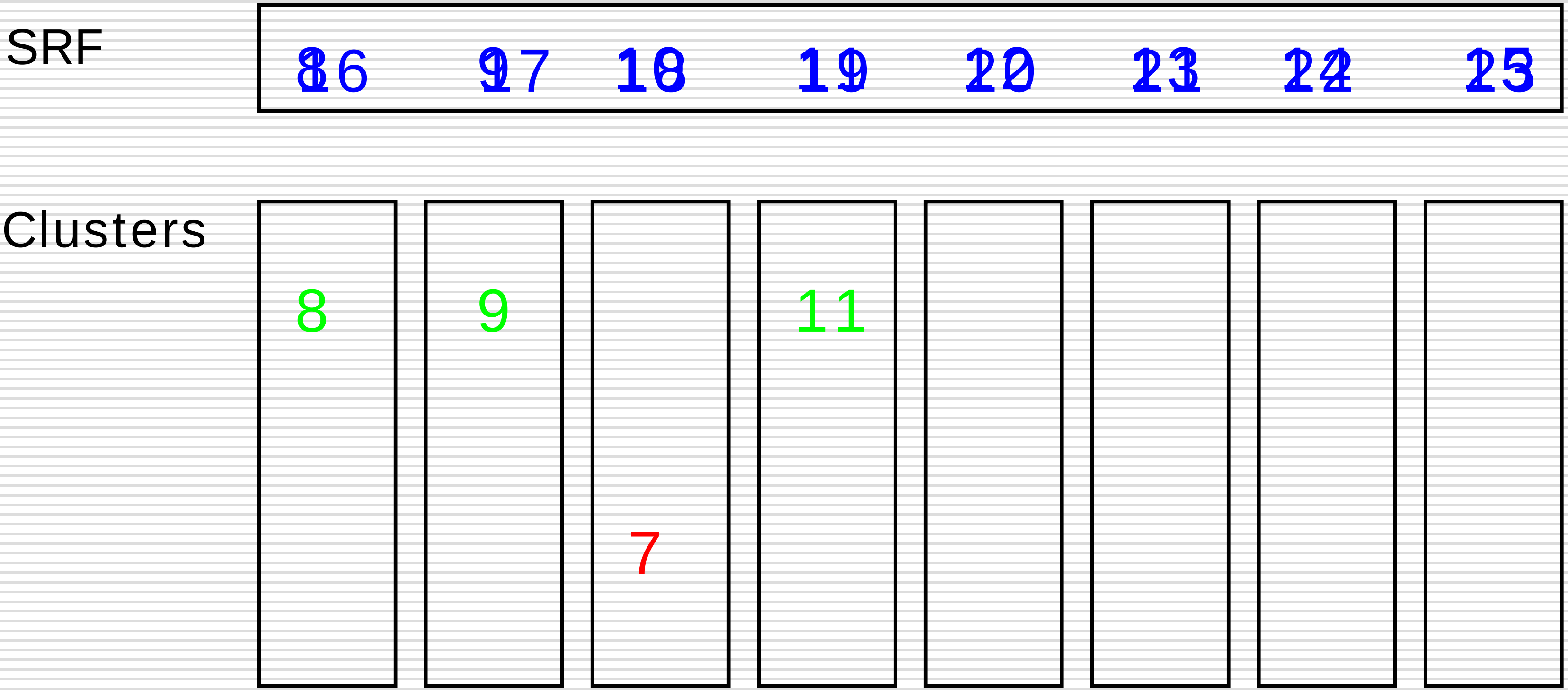
- KernelC programming is hard:
  - Cluster communication is limited and depends on particular access pattern
  - No conditionals, only predicates
  - Bookkeeping of state/data is needed
- High-level streaming language (Brook):
  - Easy to use
  - No architecture specific details
  - But no compiler yet!

Idea

- Main Brook abstractions are streams and stencils:
  - Streams specify the shape of data
  - Stencils specify access patterns
- Mapping arbitrary stream/stencil pair to KernelC code:
  - Does not depend on the computation itself
  - The rest is just a substitution of stencil elements into user formula
  - Instruction scheduling/register allocation can be handled by KernelC compiler

1D Example

- Stencil [-3, 1]



Objectives

- Minimize cluster communication
  - May be a limiting factor
- Minimize storage requirements
  - If possible fit everything into LRFs
  - If not, use scratchpad for storage
- Handle as large stencil space as possible, i.e. 1D, 2D, ...
- Try to utilize property of computation (i.e. associativity) for optimization

Approach

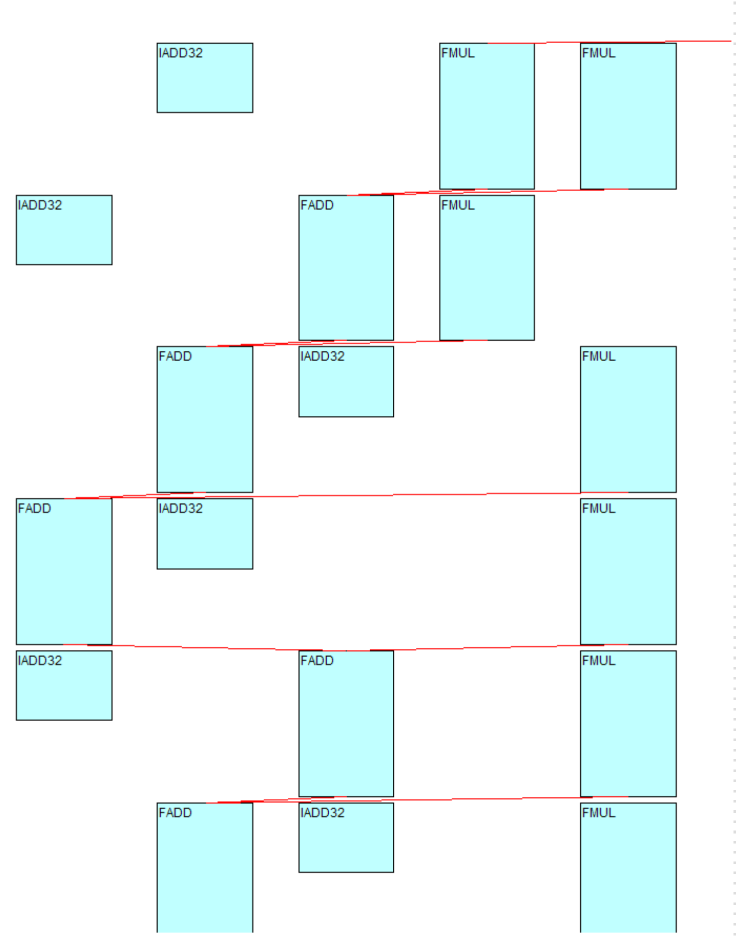
- Develop samples of KernelC code for various types of streams/stencils
- Write a Perl script to generate code for loop/communication
- Use simple kernel for evaluation of results, i.e. convolution
- Analyze limiting factors:
  - Arithmetic unit utilization
  - Inter-cluster communication
  - Storage

Issues

- KernelC scheduler cannot restructure computation, e.g. if you write:

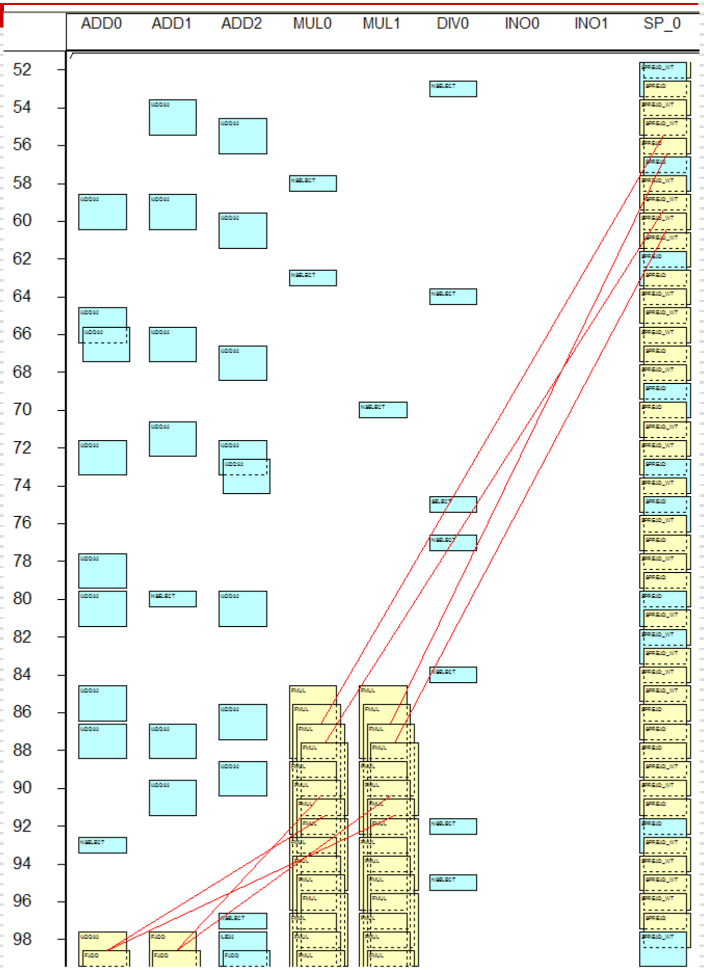
  o = a+b+c+d+e+f =>

- Dependency is the limiting factor!
- Our solution: binary tree generated by Perl



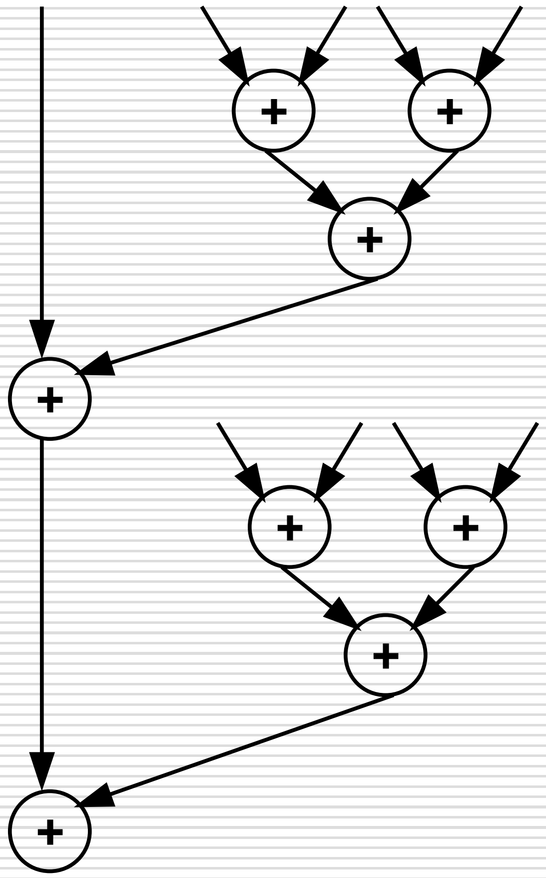
But ...

- KernelC uses "lazy" scheduling, i.e. schedules ops as late as possible:
  - Lifetime of temporaries increased
  - Register file pressure is increased
  - Scheduler fails even for relatively small stencils
- Our solution: asymmetric tree generated by Perl

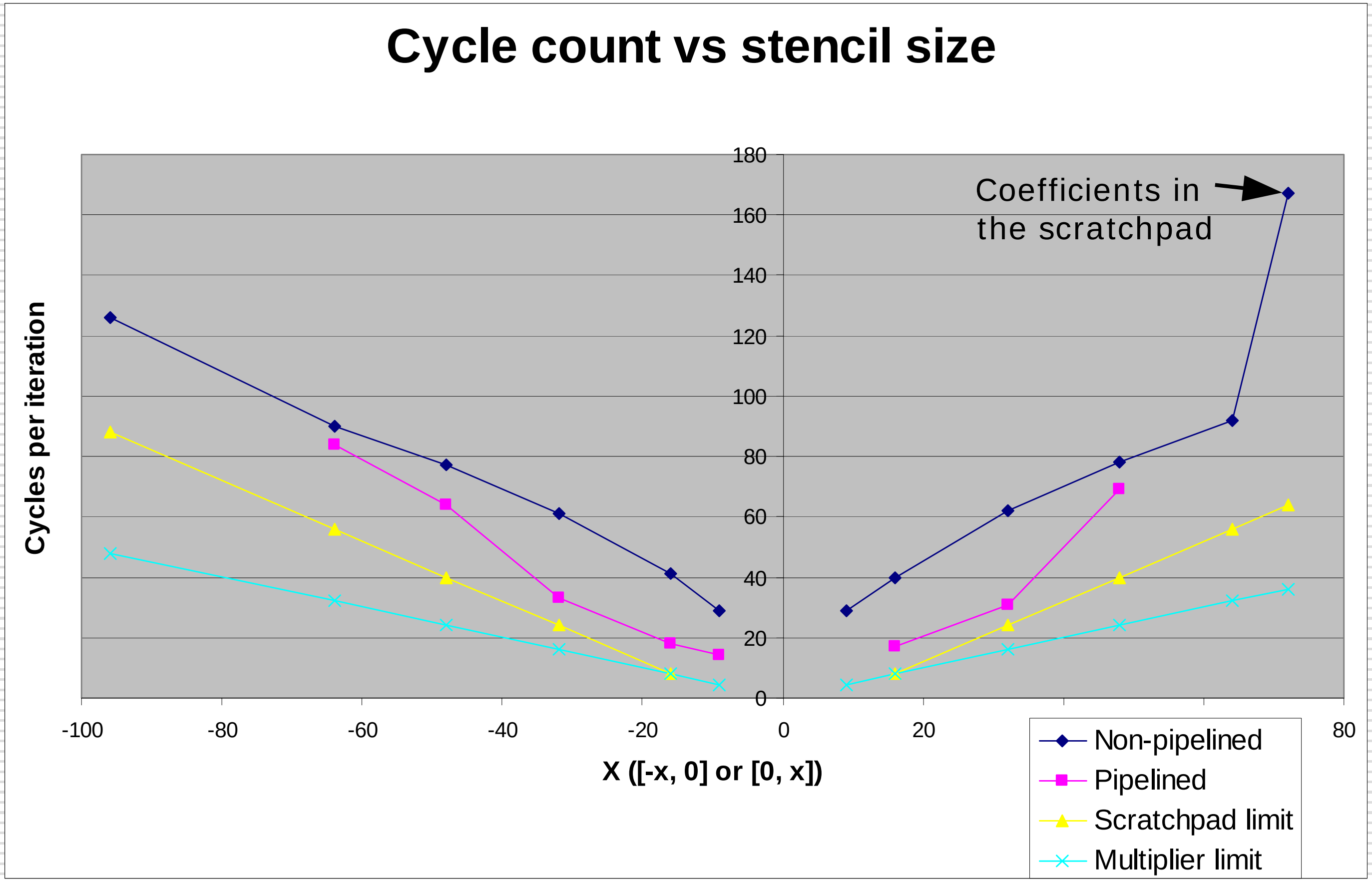


Asymmetric Tree

- The length of critical path can be adjusted:
  - Forces the scheduler to schedule ops earlier
  - Reduces register file pressure



Results: 1D stencil, convolution



Associative Computations

- Convolution computation can be broken into several parts, i.e.:

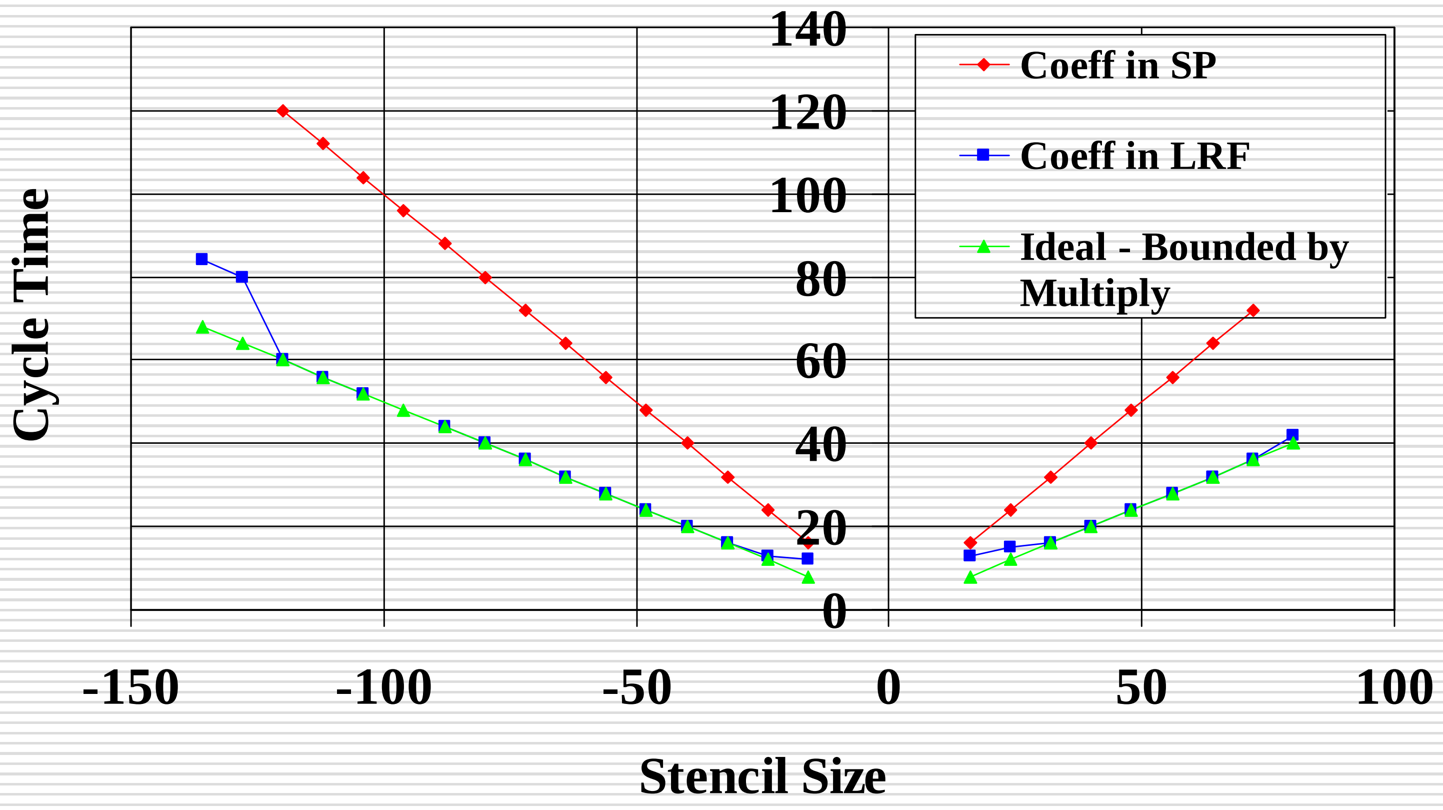
  $a = c_0*x_0+c_1*x_1+c_2*x_2+c_3*x_3$

  $b = c_4*x_4+c_5*x_5+c_6*x_6+c_7*x_7$

  $out = a + b$

- Instead of storing all input data for next iteration we can store computed partial convolutions:
  - Storage and bandwidth requirements reduced by a factor of 8
  - Can handle larger stencils more efficiently

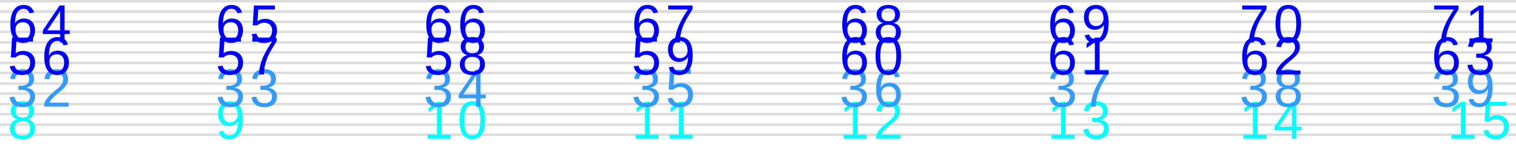
Results: 1D, associative case



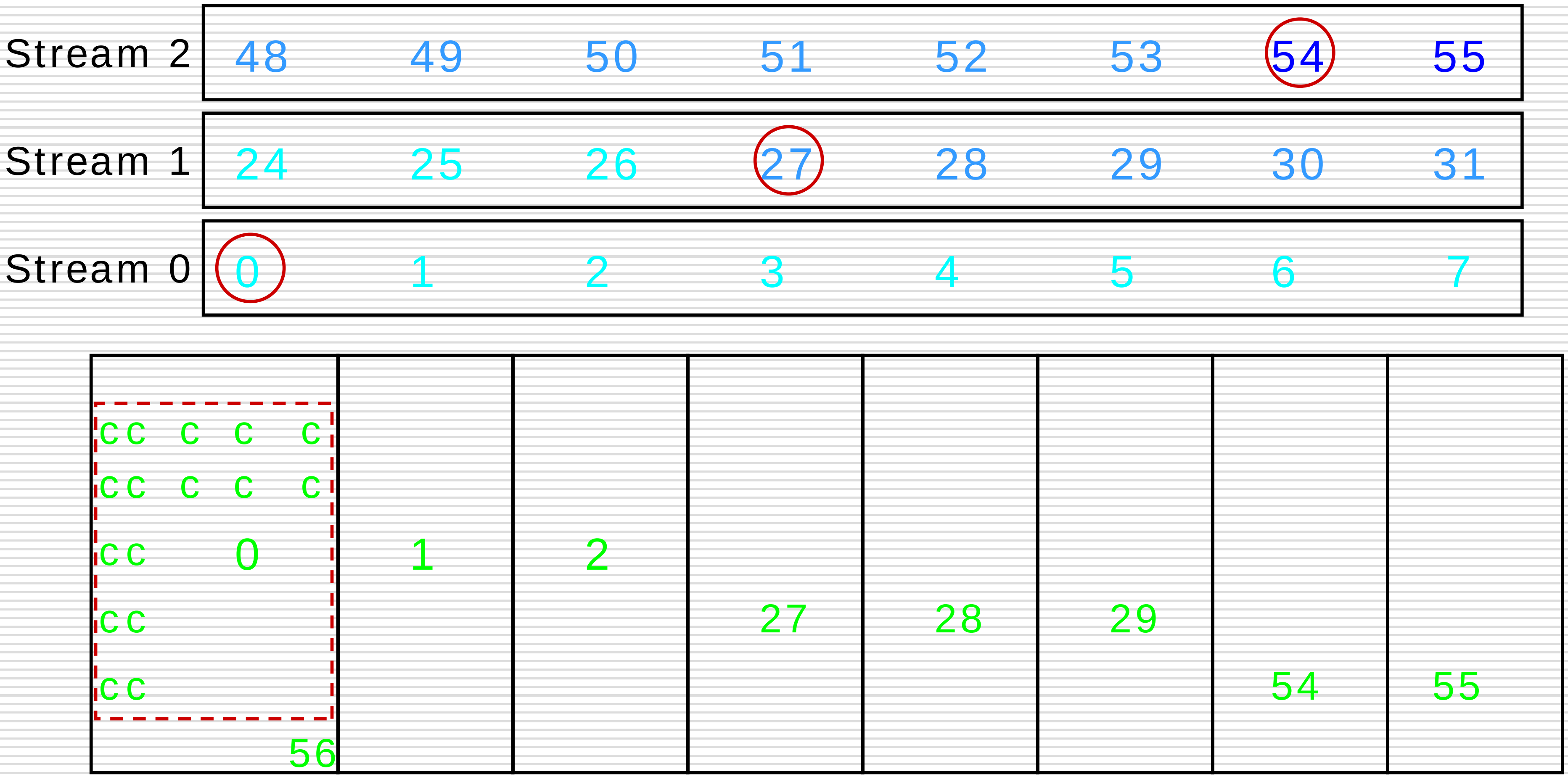
2D Streams/Stencils

- Example:
  - 2D stream 27x8 elements
  - Stencil 5x5: -2≤x≤2, -2≤y≤2

C	C	C	C	C			
C	C	C	C	C			
C	C	0	1	2	3	...	26
C	C	27	28	29	30	...	53
C	C	54	55	56	57	...	80
		81	82	83	84	...	107
		...					



2D Streams/Stencils



2D Stencil Issues

- Storage Requirement
  - $$(\#\text{ of registers}) = \left(\left\lceil \frac{W_{sten} + d \cdot (H_{sten} - 1)}{n_c} \right\rceil + 1\right) \times n_c \times H_{sten} + W_{sten} \times H_{sten} + C$$

    $$(where\ n_c = 8, d = W_{2Dstream} \% n_c, \text{ and C} = \text{constant overhead})$$
  - We also exploited the property of operation (e.g. associativity) aggressively to mitigate storage requirement in 2D case.
    (16*9 was schedulable for 2D convolution with partial sum.)
- Stream Requirement
  - In case of H_{sten}>8, put a preprocessing kernel for consolidation
- Scheduler Issue
  - Currently, register allocation failure for 7*7 stencil
    (4*3, 5*5, and 6*6 are okay. 25*4 takes forever..)

Conclusions

- Efficient mapping of Brook stencils to Imagine was demonstrated:
  - General case
  - Associative computation
- Scheduler issues:
  - No computation restructuring
  - "Lazy" scheduling greatly increases register file pressure
  - Second port of scratchpad is not used since scheduler can't resolve dependency