Extending a Stream Programming Paradigm to Hardware Accelerator Platforms

Sek Chai
Motorola
Schaumburg, IL
sek.chai@motorola.com

Abelardo López-Lagunas
Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Toluca, México
abelardo.lopez@itesm.mx

Nikos Bellas
University of Thessaly, Greece
nbellas@inf.uth.gr

Motorola General Business Information, saahpc2009, v1.0
A patent is pending that claims aspects of items and methods described in this paper and presentation.
MOTOROLA and the Stylized M Logo are registered in the US Patent & Trademark Office.
All other product or service names are the property of their respective owners. © Motorola, Inc. 2008

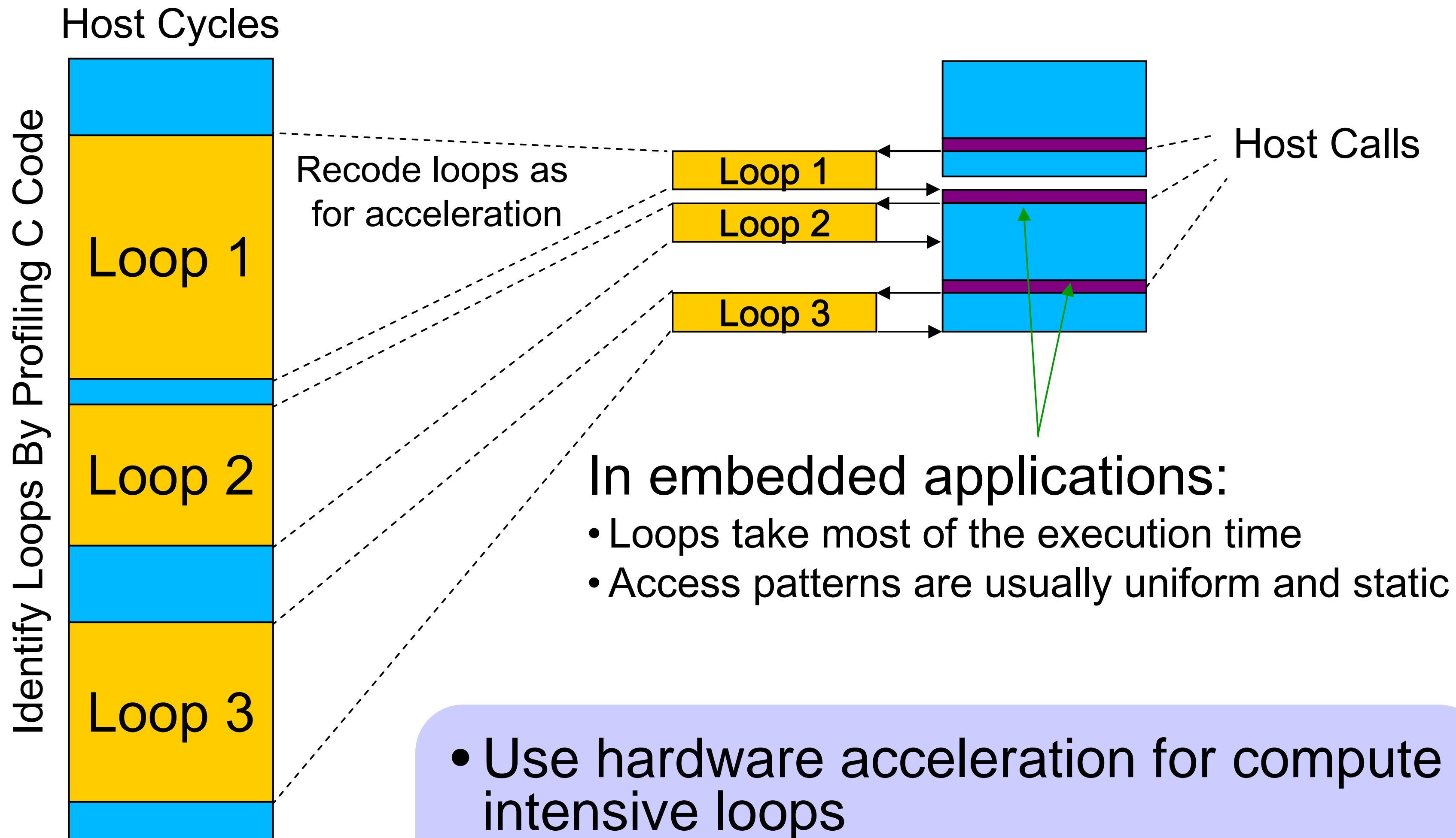
Contributors

Nikos Bellas
Sek Chai
Silviu Chiricescu
Malcolm Dwyer
Ray Essick
Dan Linzmeier
Abelardo Lopez-Lagunas
Brian Lucas
Phil May
Kent Moat
Jim Norris
Michael Schuette
Ali Saidi

Agenda

- Separation of concerns between computation and memory access
- RSVP and Proteus streaming accelerators
- Results and summary

Hardware Acceleration



In embedded applications:

- Loops take most of the execution time
- Access patterns are usually uniform and static

- Use hardware acceleration for compute intensive loops
- Keep single processor programming flow

Stream Processing

Example processing chain as stream kernels

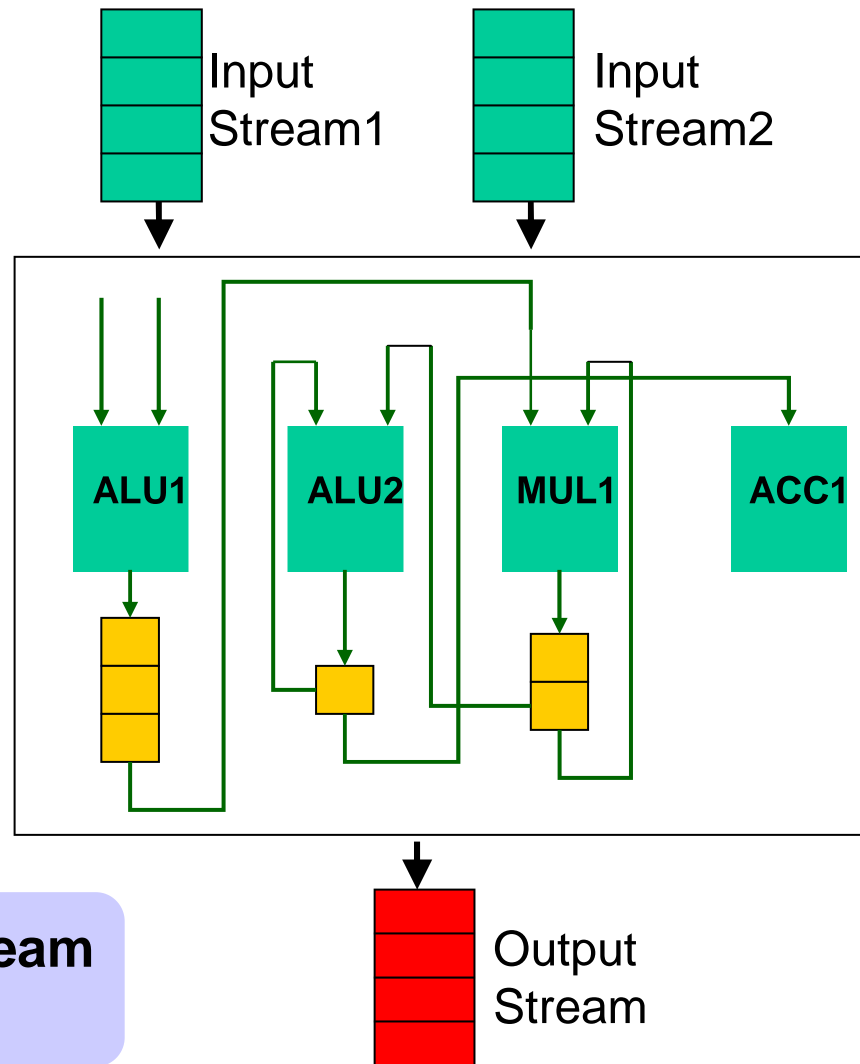
Characteristics	Architecture implications
Computation is repetitive on localized data regions	Explicit parallelism. Overlap data movements to accelerator.
Kernels are independent and self contained	
Large (possibly infinite) amount of data	Low temporal locality for data. Traditional caches are not effective.
Limited lifetime of datum	
Compute graph is mostly constant; Static computation patterns	Memory access patterns are deterministic.

Kernel Parallelism

Kernels read and write streams (no global variables)

Optimal exploitation of Instruction & Data Level parallelism via loop unrolling modulo scheduling, etc

Lack of dynamic memory schedule allows the compiler to produce optimal code for the given hardware resources



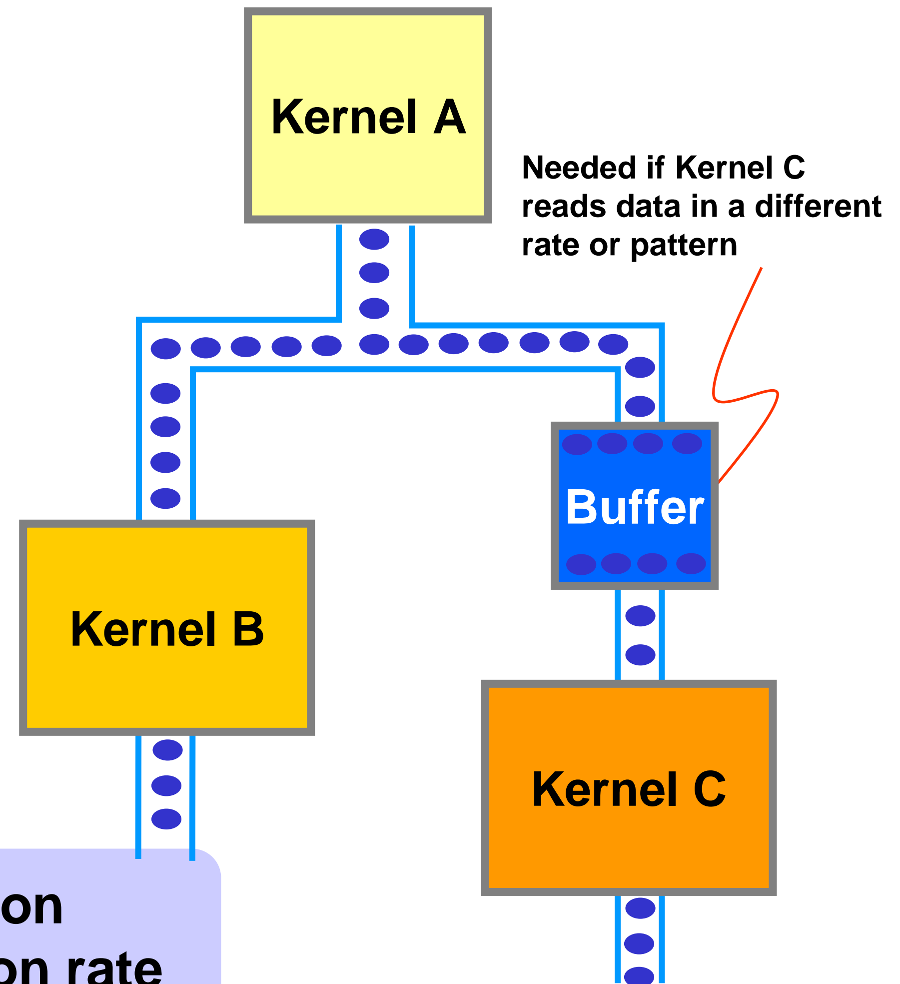
Chain *functional units* based on stream consumption and production rate

Communication Locality

Inter-kernel communication through a producer-consumer model.

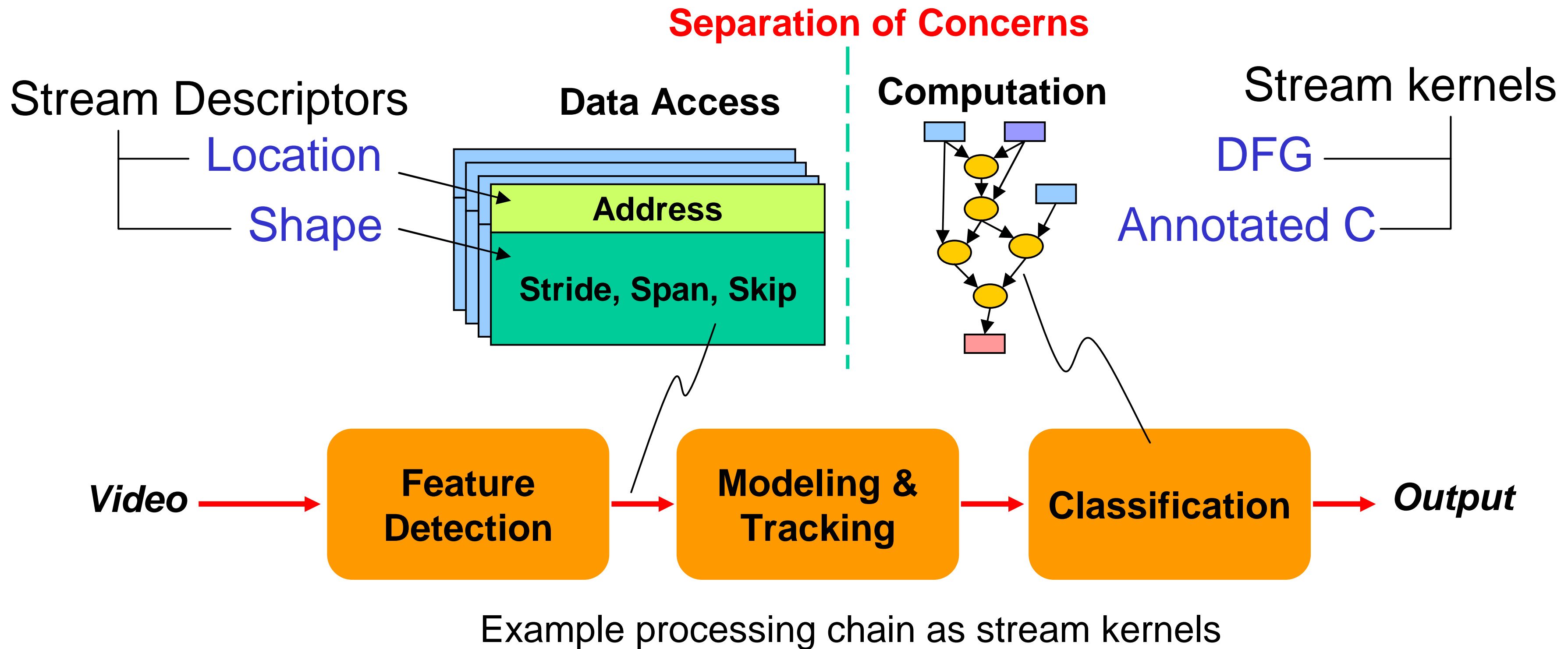
System-level task scheduling is easier because the communications are explicit in the program

Memory wall problem can be significantly reduced



Chain *hardware accelerators* based on stream consumption and production rate

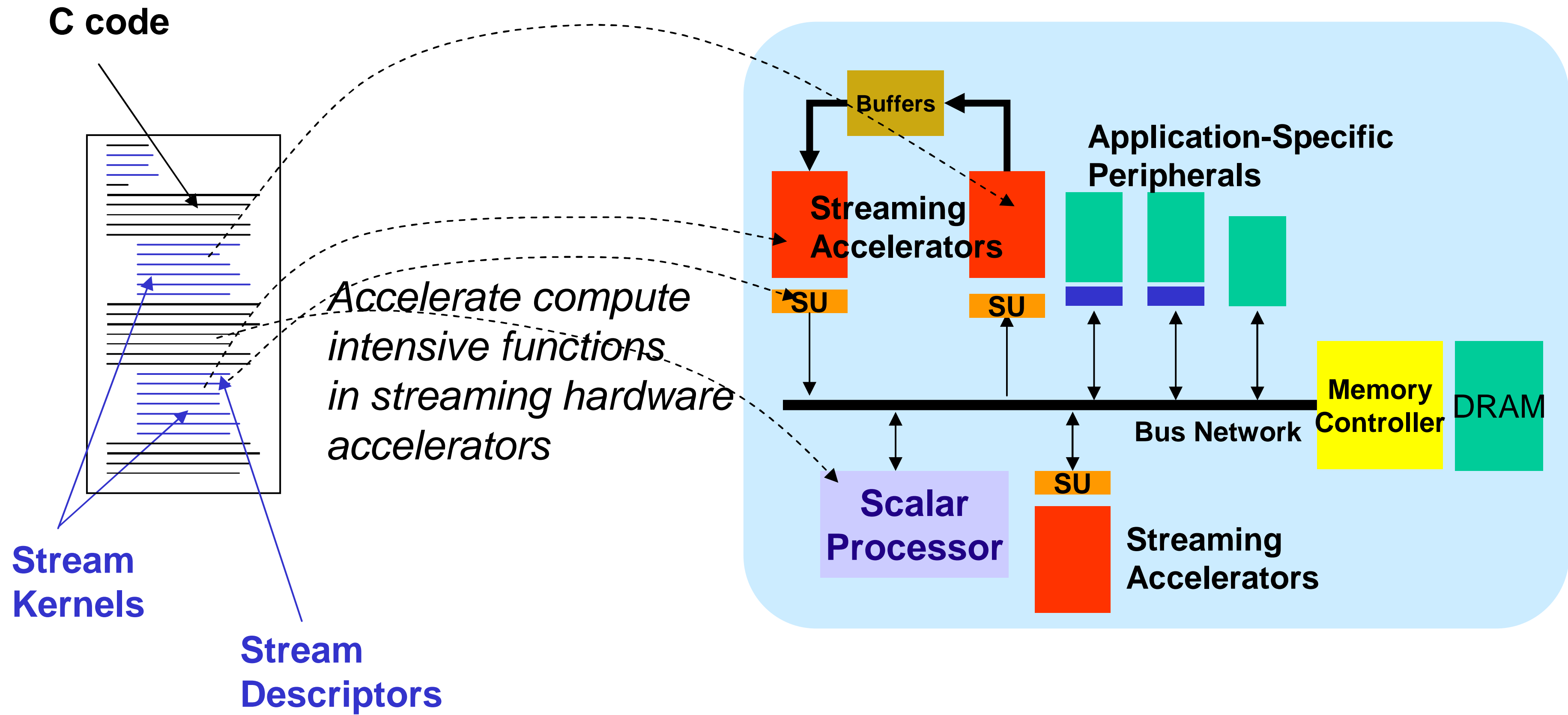
Separation of Concerns



Example processing chain as stream kernels

Decoupled memory accesses and computations:
- enable better optimization of hardware
- simplify compiler tasks

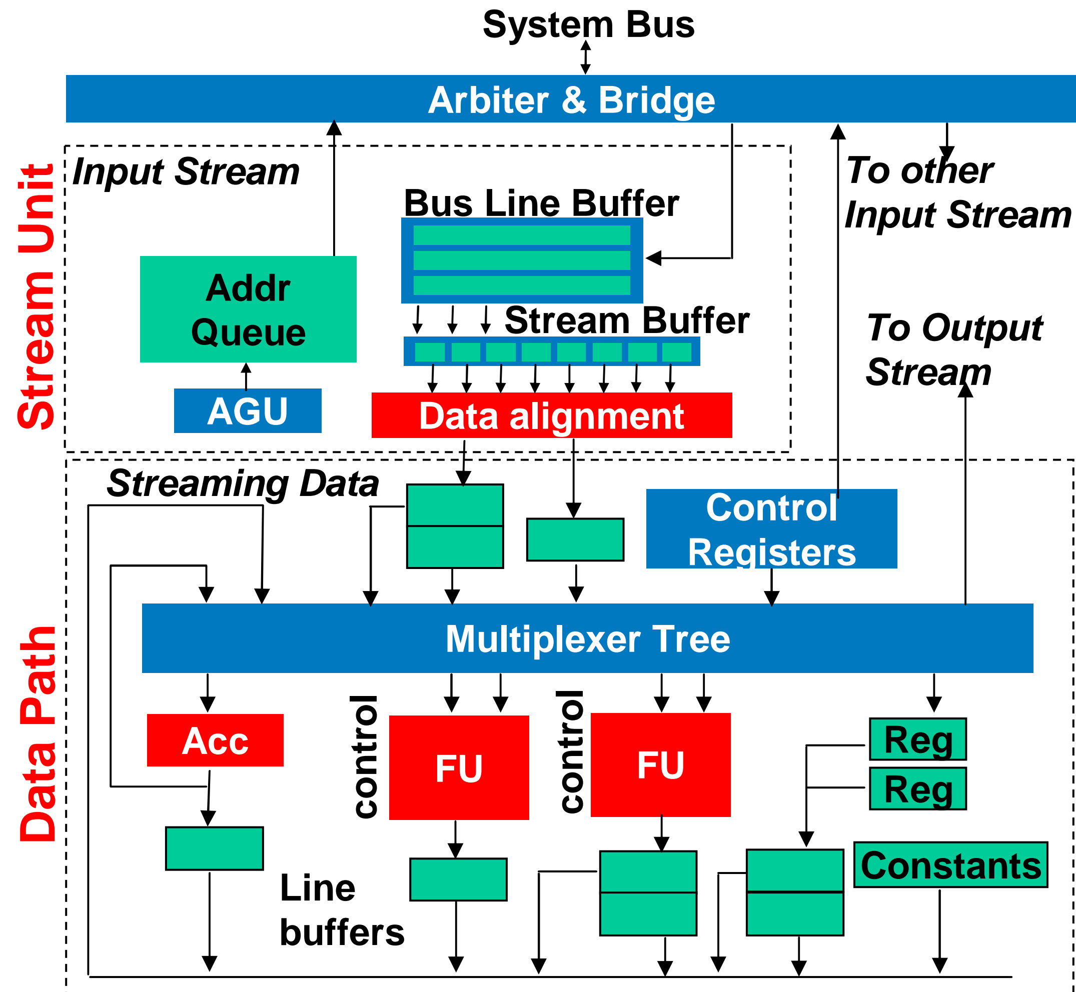
Hardware Acceleration of Stream Kernels



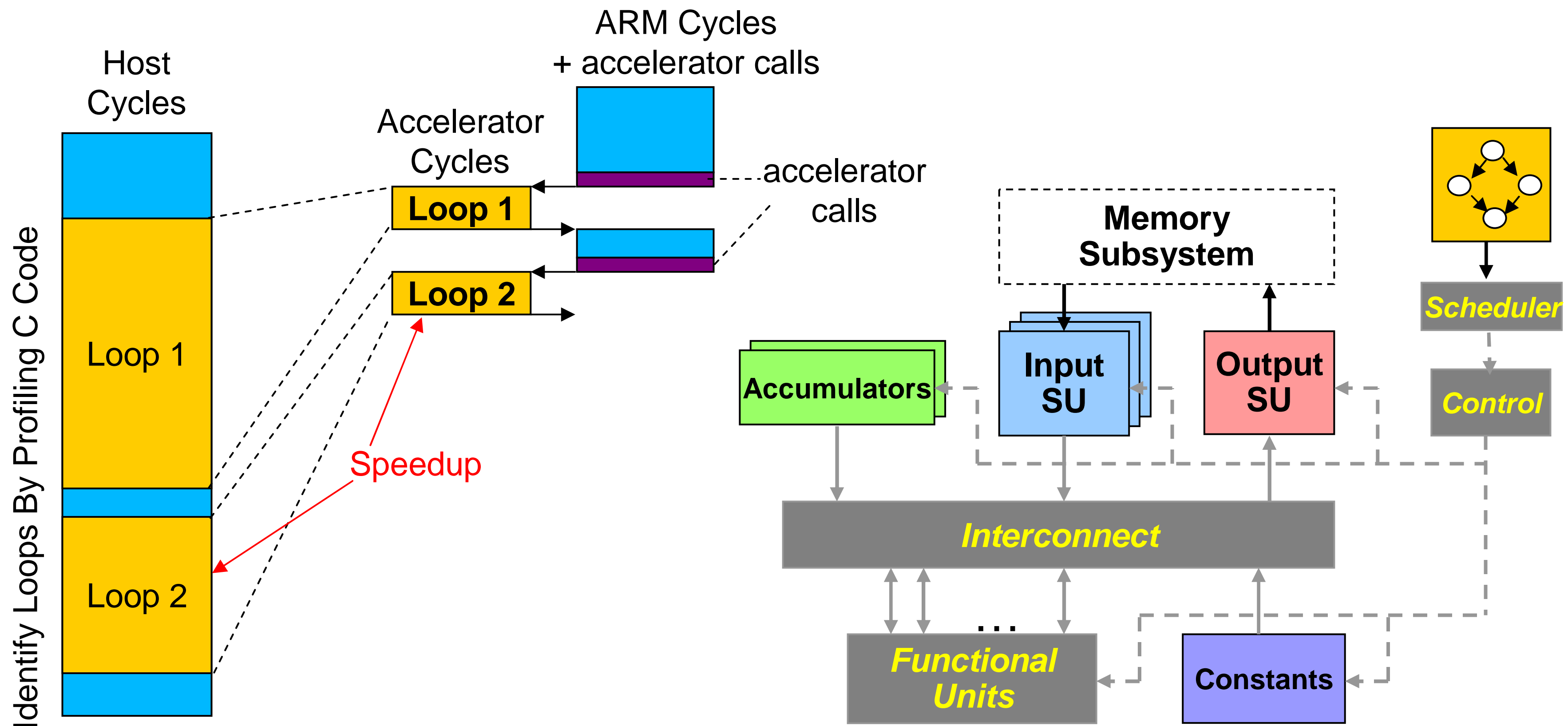
Stream Descriptors help define memory subsystem structure.

Stream Kernels define hardware accelerators.

Streaming Accelerator Template



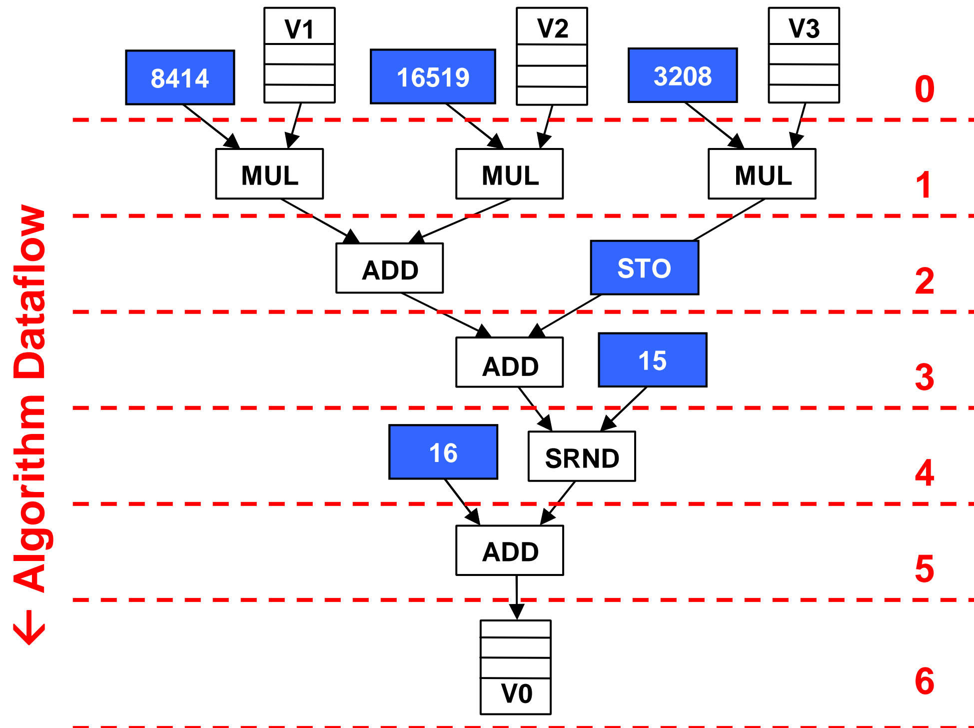
Programmer Visible Architectural Elements



S. Chai, et. al., “Streaming Processors for Next Generation Mobile Imaging Applications”, in IEEE Communications Magazine, Circuits for Communication Series, vol 43, no 12, Dec 2005, pp. 81-89

S. Chiricescu, et. al., "The Reconfigurable Streaming Vector Processor (RSVP™),“ Micro, December 2003.

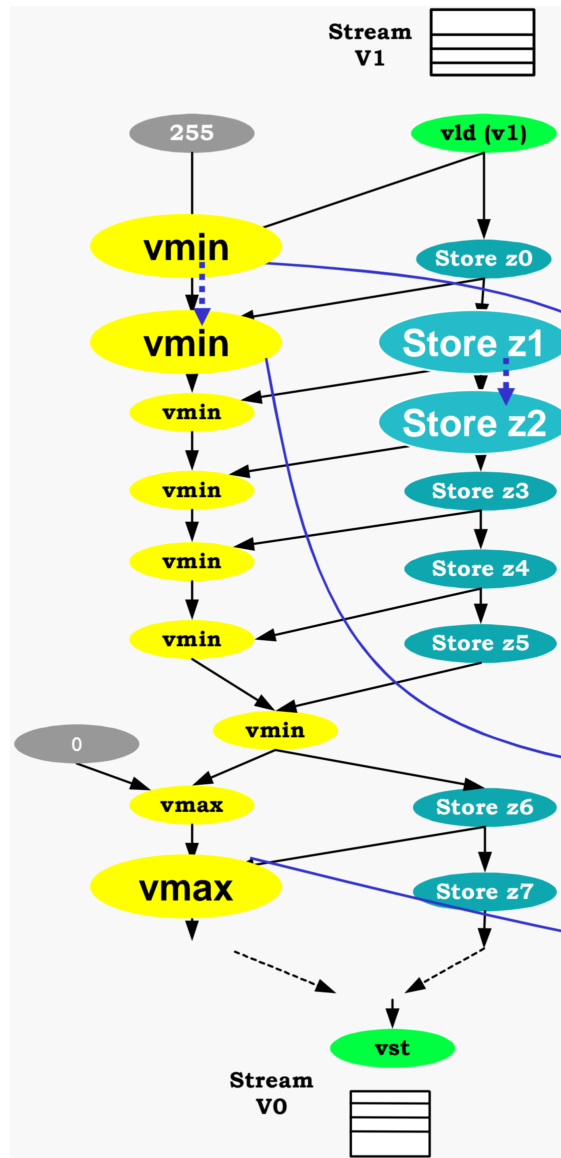
Describing Computation



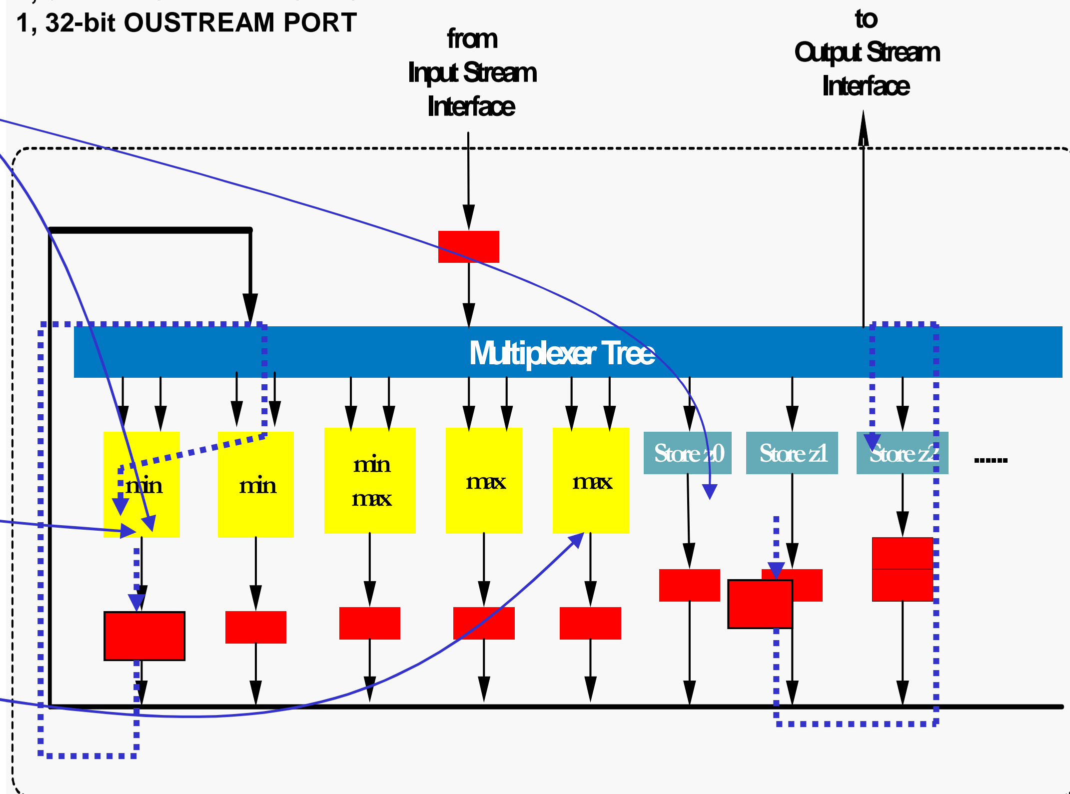
Components of sDFG (stream data flow graph)

- Nodes in graph represent computation (performed by functional units)
- Edges in graph represent data movement between functional units
- Multiple computation elements arranged across the data streams

Datapath Construction

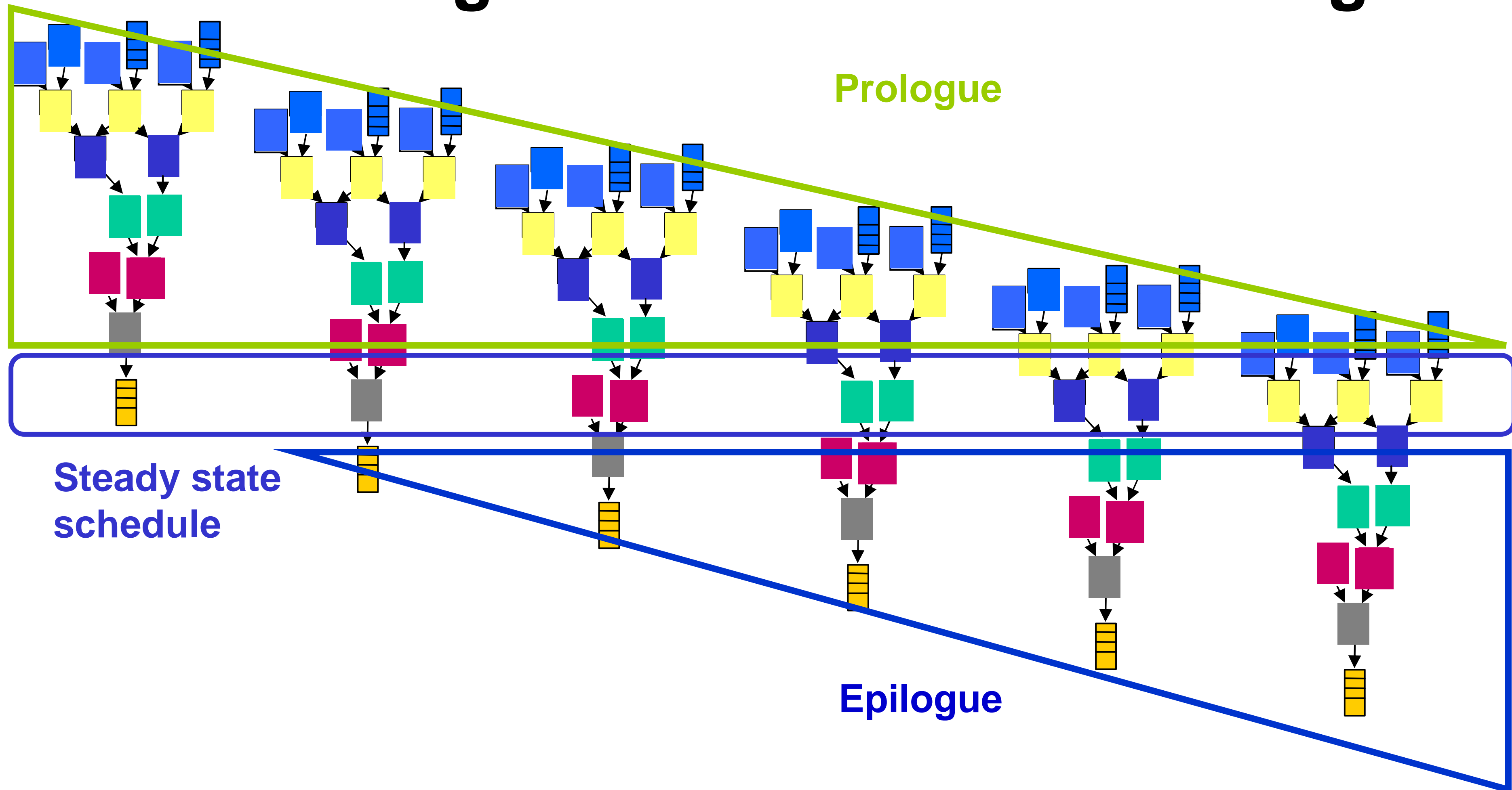


Resource constraints:
192 ALU bits,
128 MUL bits,
64 SHIFTER bits
128 NAMED REG bits
4, 32-bit INSTREAM PORTS
1, 32-bit OUSTREAM PORT



Map sDFG to functional units

Streaming DFG – Modulo Scheduling



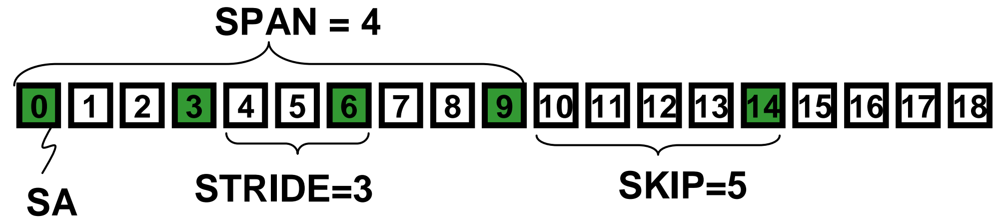
Hardware mechanisms ensure that the prologue and the epilogue are executed correctly

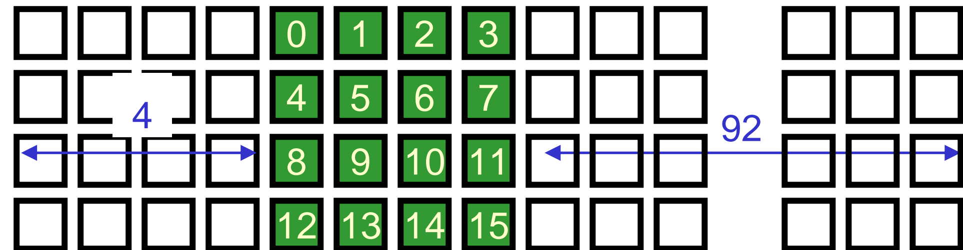
Describing Memory Access

A method to move data efficiently using known shape of data

For data prefetch, staging, and reuse

- Efficient data movement
- Utilize unused bandwidth
- Less sensitive to peak bandwidth

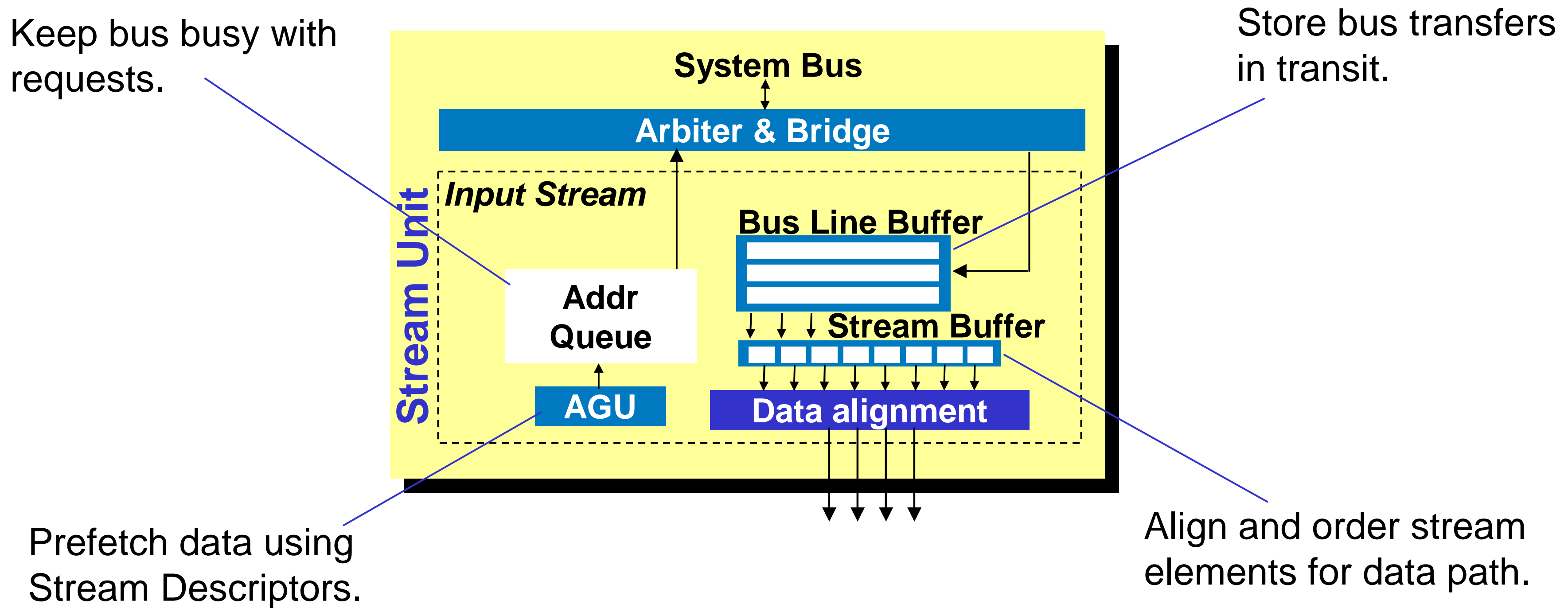




2-D Subarrays (row)
(SA, Stride, span, skip)
(4, 1, 4, 97)

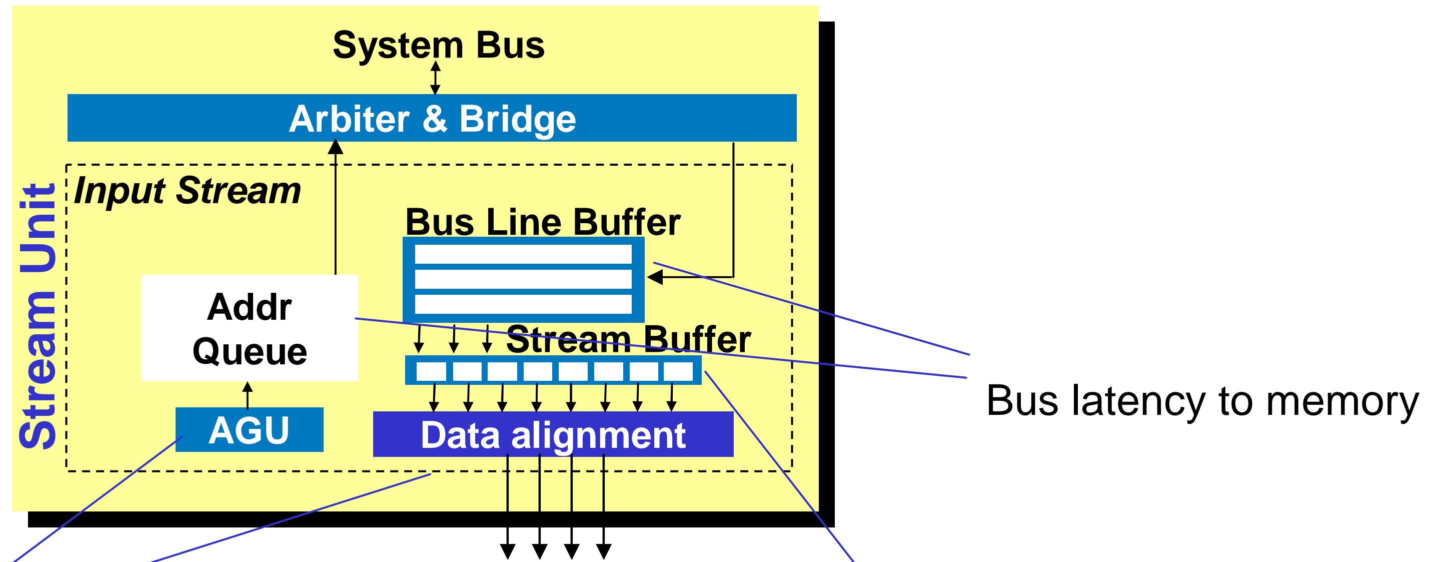
Mapped to "stream unit" or smart DMA in hardware accelerators

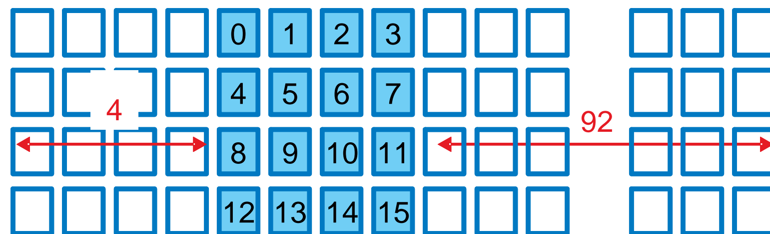
Stream Unit



Larger queues and buffers allow more aggressive prefetching of stream elements

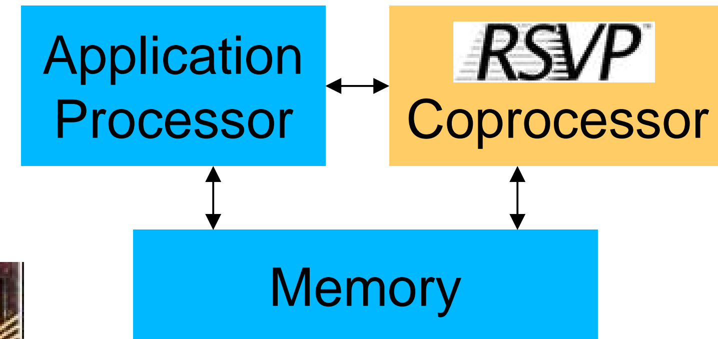
Stream Unit



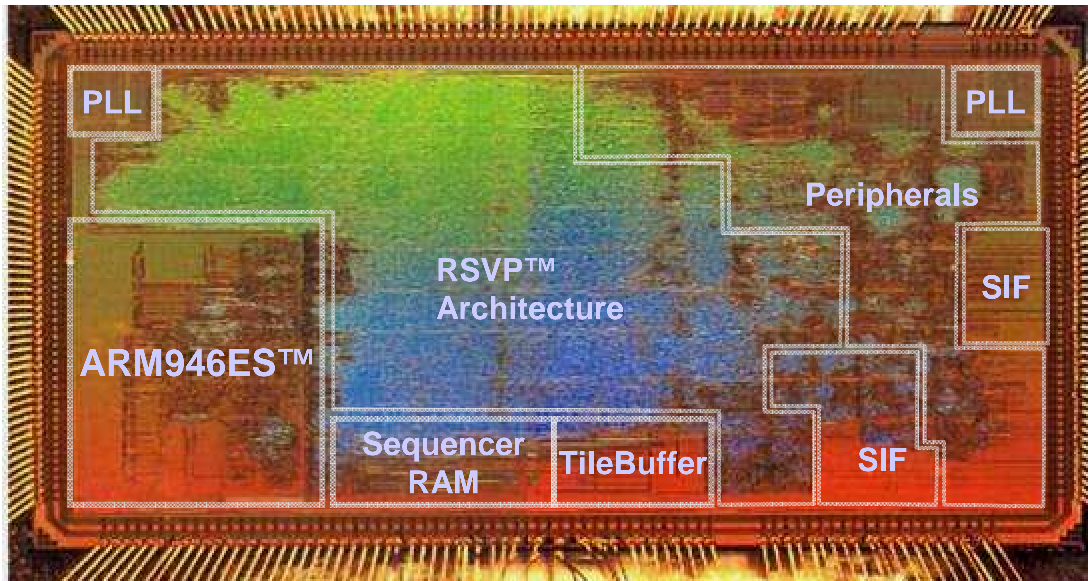


Reconfigurable Streaming Vector Processor

A software programmable vector accelerator based on a "streaming dataflow" programming model

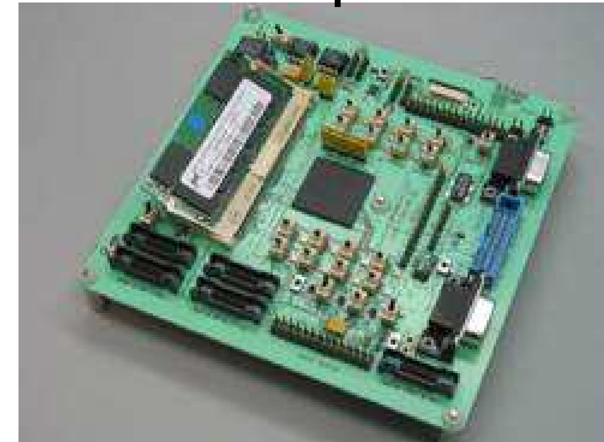


ARM946+RSVP™-I SoC



SoC (ARM946+RSVP) in 0.18µm which contains 9.5M transistors in a 5.04 x 9.03 mm^2 die. Power consumption is 587mW (1.8V, 120MHz core, 60MHz bus).

RSVP-I Development Board

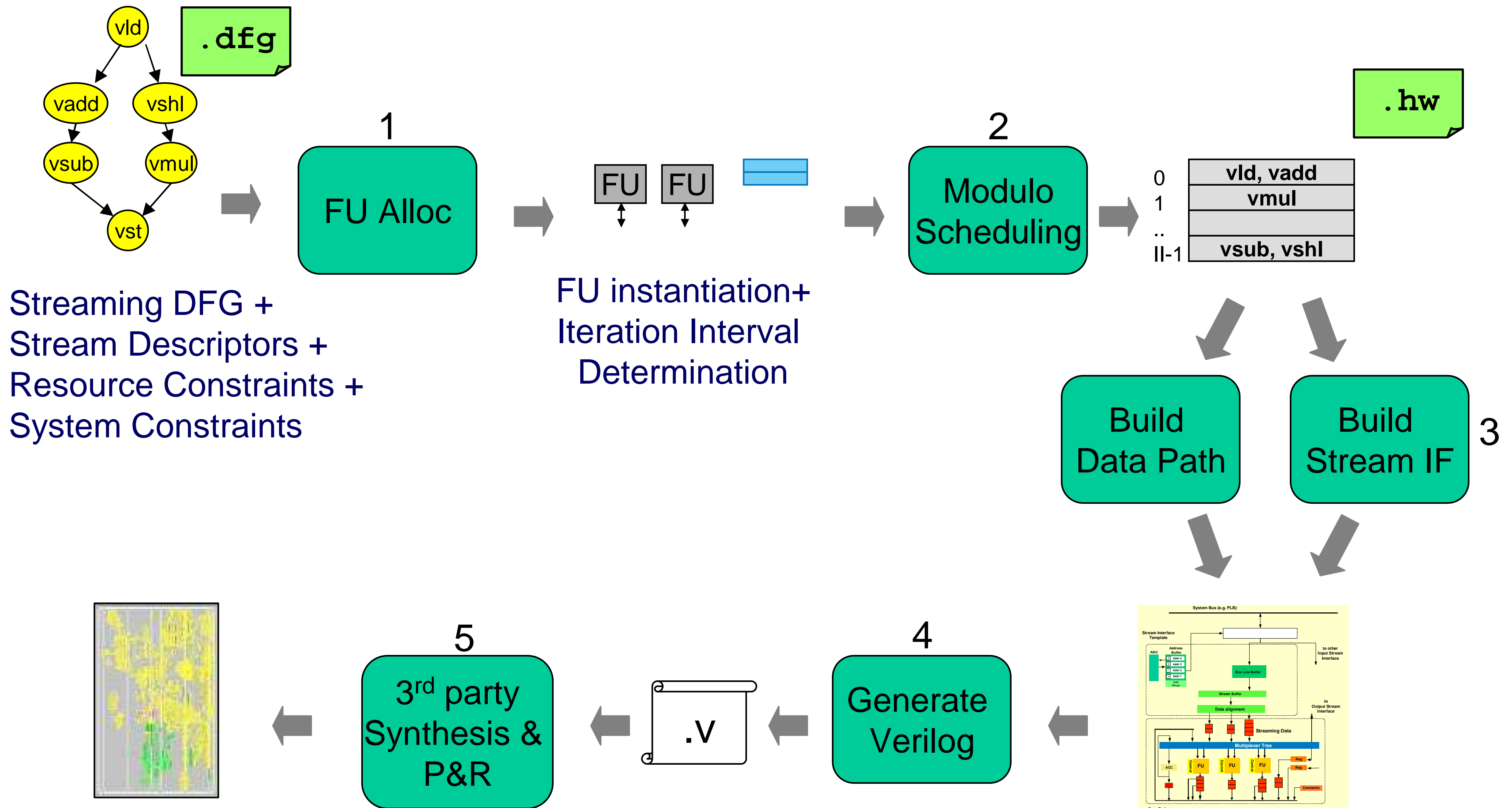


S. Chai, et. al., "Streaming Processors for Next Generation Mobile Imaging Applications", in IEEE Communications Magazine, Circuits for Communication Series, vol 43, no 12, Dec 2005, pp. 81-89

S. Chiricescu, et. al., "The Reconfigurable Streaming Vector Processor (RSVP™)," Micro, December 2003.

Proteus Streaming Accelerator Design Flow

(FPGA)



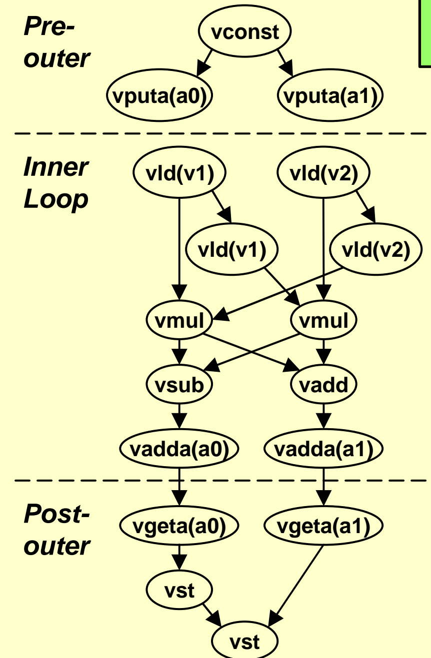
DFG and Scheduler example

```
// In-lined C code
cfir32(const short *x, const short *h, short *r,
const short nh, const short nr) {

extern void d_cfir32;   // entry point to the DFG
_vload(&d_cfir32);       // Load DFG

// Set up stream descriptors
_vihalf(1, (unsigned short *) &x[2*nh - 1]);
_vishape(1, -1, 2*nh, 2*nh + 2);
_vihalf(2, (unsigned short *) &h[0]);
_vishape(2, 1, 2*nh, -2*nh);
_vohalf(0, r);

  // Start RSVP™ co-processor
_vloop2(&d_cfir32, nh, nr);  // execute DFG

}
```

In-line C Code

```
// DFG for the complex FIR
      vname      d_cfir32
      vbegin     L_end-L_start,0
L_start:
L0:   vconst      0      // clear the accumulators
L1:   vputa       L0, a0
L2:   vputa       L0, a1
   vinner
L3:   vld.u16    (v1)    // load the input data
L4:   vld.u16    (v1)    // load the input data
L5:   vld.u16    (v2)    // load the coefficients
L6:   vld.u16    (v2)    // load the coefficients
L7:   vmul.s32   L4,L5    // first multiply
L8:   vmul.s32   L3,L6    // second multiply
L9:   vsub.s32   L7,L8    // real part
L10:  vadd.s32   L7,L8    // imaginary part
      vadda      L9,a0    // accumulate real
      vadda      L10,a1   // accumulate imaginary
   vpost
L11:  vgeta.s16  a0
L12:  vgeta.s16  a1
L13:  vst        L11, (v0) // store the real
L14:  vst        L12, (v0) // store the imaginary
L_end:   vend
```

Linear DFG code



Graphical DFG

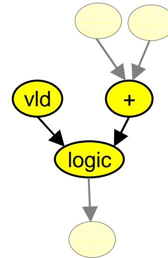
Proteus Scheduler Features

- Do not require separate prolog and epilog code
- Can handle nested loop constructs without having to create different schedules for different parts of the nested loop construct
- Supports tightly coupled memories (e.g. doubled buffered LUT)
- Handle resources (FUs, fabric, queues) for different application optimization

Hardware File Template

.hw

```
.DataPathBegin
// Functional Units
fu(adder0,  adder,    1);
fu(logic0,  logic,    1);
fu(sin1,    InStream, 1);
    ... more ...
// Functional Unit Slices
sfu(adder0.0,  {vsub, vabs, vnop},       {32,32,16},    1, 1);
sfu(logic0.0,  {vmin, vmax, vif, vnop}, {16,16,32,16}, 1, 1);
sfu(sin1.0,    {vld, vnop},              {16},          0, 1);
    ... more ...
// Line Queues
queue(Qadder0_0_0,  adder0.0,  [31..0], 1);
queue(Qlogic0_0_0,  logic0.0,  [15..0], 1);
queue(Qsin1_0_0,    sin1.0,    [15..0], 1);
    ... more ...
.DataPathEnd
.ControlPathBegin
cstep(3)
//
// Operations for each Functional Unit
ctl_ops(adder0, {{vsub.s32.u32.u16},     {vnop},
{vabs.u32.s32.u16}});
ctl_ops(logic0, {{vmax.u16.u16.u32.u16}, {vnop},
{vmin.u16.u16.u32.u16});
ctl_ops(sin1,   {{vld.u16},              {vnop}, {vnop});
    ... more ...
// Operands for each Functional Unit Slice Input
ctl_opnds(adder0.0.A, {{Qlogic2_0_0.0},  {vnop}, {Qadder0_0_0.0});
ctl_opnds(adder0.0.B, {{Qlogic3_0_0.0},  {vnop}, {vnop});
ctl_opnds(logic0.0.A, {{Qsin1_0_0.0},    {vnop}, {Qlogic0_0_0.0});
ctl_opnds(logic0.0.B, {{Qsca_z15_0_0.0}, {vnop}, {Qsca_z0_0_0.0});
ctl_opnds(logic0.0.C, {{vnop},           {vnop}, {vnop});
    ... more ...
// Controls for each Queue
ctl_queue(Qadder0_0_0, {{1}, {0}, {1}});
ctl_queue(Qlogic0_0_0, {{1}, {0}, {1}});
    ... more ...
.ControlPathEnd
```

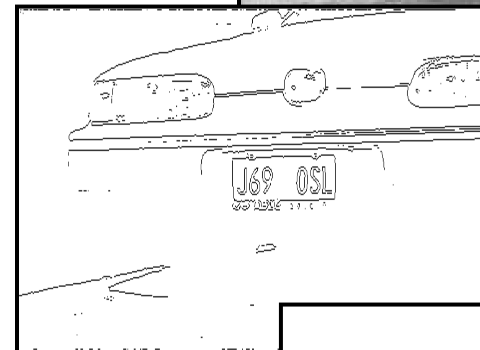


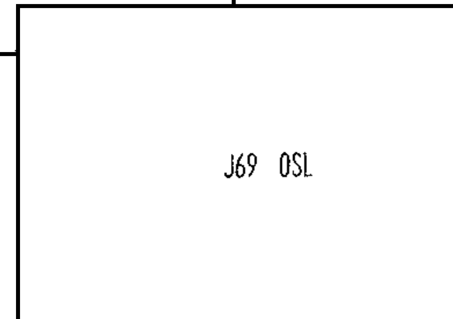
Proteus HW File features

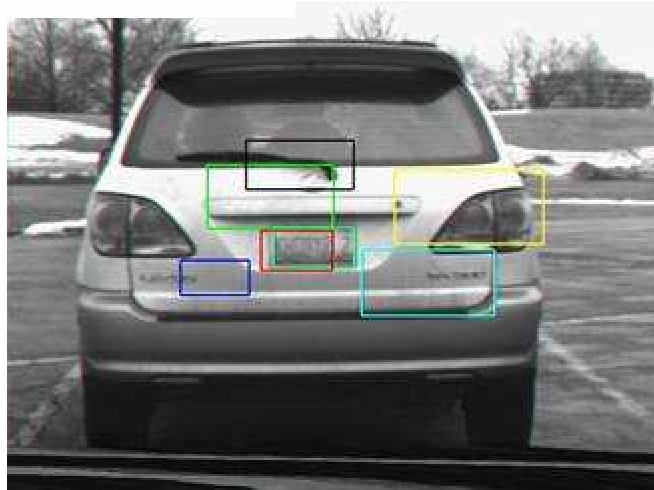
- Structured intermediate format (IF) for streaming data path
- Facilitates debugging with consistent naming/labels
- Describes resources such as FUs, fabric, queues, LUTs.
- Scalable with new ISA updates

Security & Surveillance

Smart Cameras read license plates and compare against database.





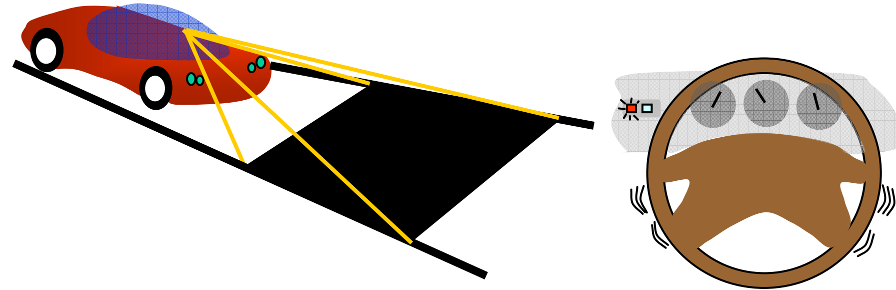


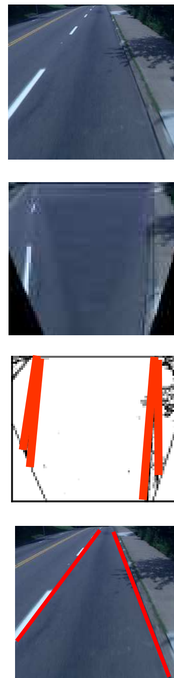
Sek Chai, et. al., “Reconfigurable Streaming Architectures for Embedded Smart Camera Applications”, Embedded Computer Vision Workshop, New York, June 2006.

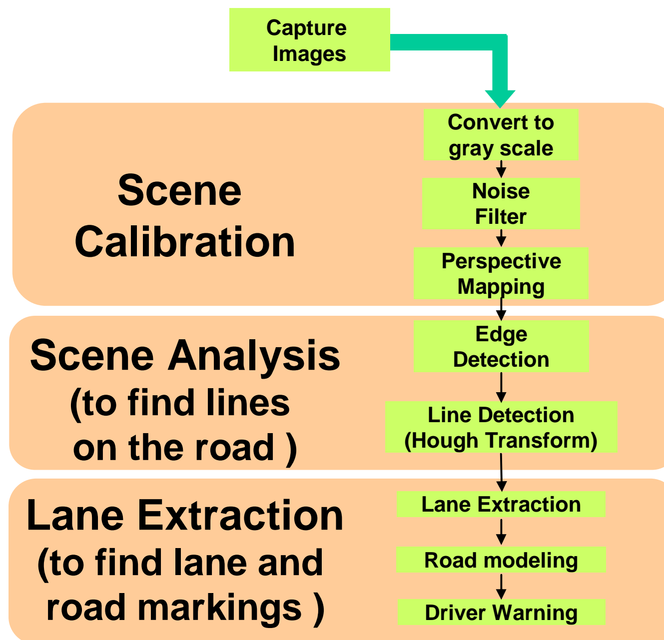
N. Bellas, Sek Chai, M. Dwyer, D. Linzmeier, “FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators”, Reconfigurable Architecture Workshop (RAW), April 2006.

Automotive

Smart Cameras detect and track lanes. Warn drivers on hazards and unexpected lane departures.



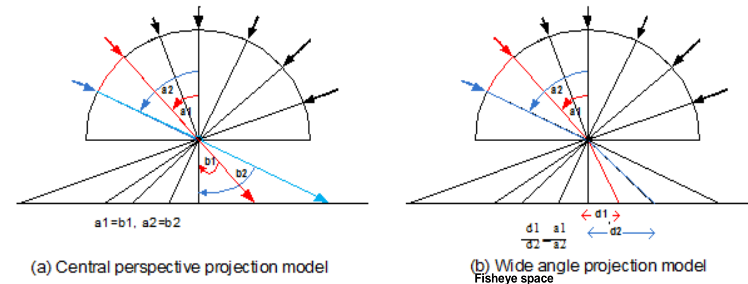


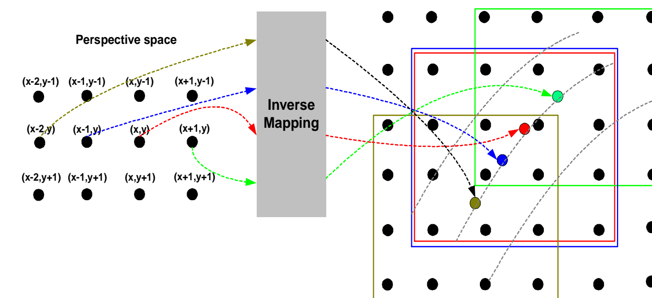


S. Chiricescu, S. Chai, K. Moat, B. Lucas, P. May, J. Norris, R. Essick, M. Schuette, "RSVP II: A Next generation Automotive Vector Processor," in IEEE Intelligent Vehicles Symposium, June 2005

Lens correction

Addresses issues related to filtering and lens-distortion correction for visual communications

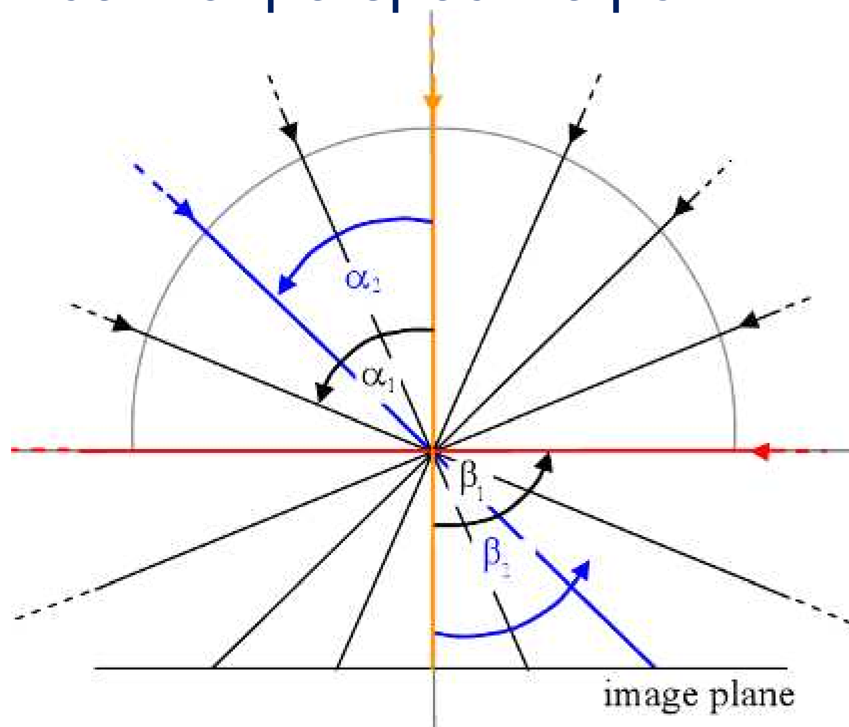




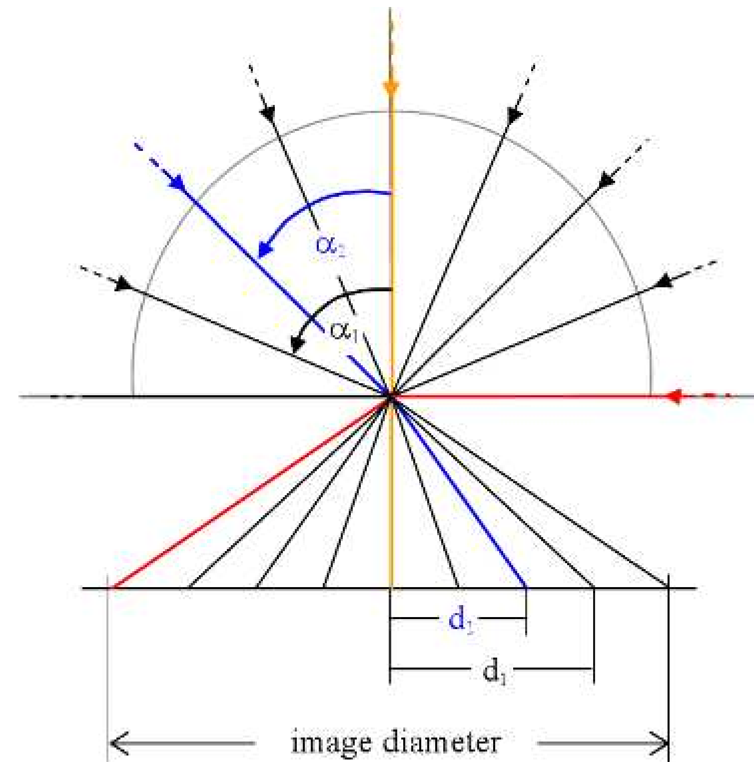
N. Bellas, Sek Chai, M. Dwyer, D. Linzmeier, "Real-Time Fisheye Lens Distortion Correction Using Automatically Generated Streaming Accelerators", IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2009

Stereoscopic geometry of Fisheye Lenses

- Fisheye lenses refract the incident light rays towards the central perspective point



Rectilinear Projection



Fisheye Projection

=

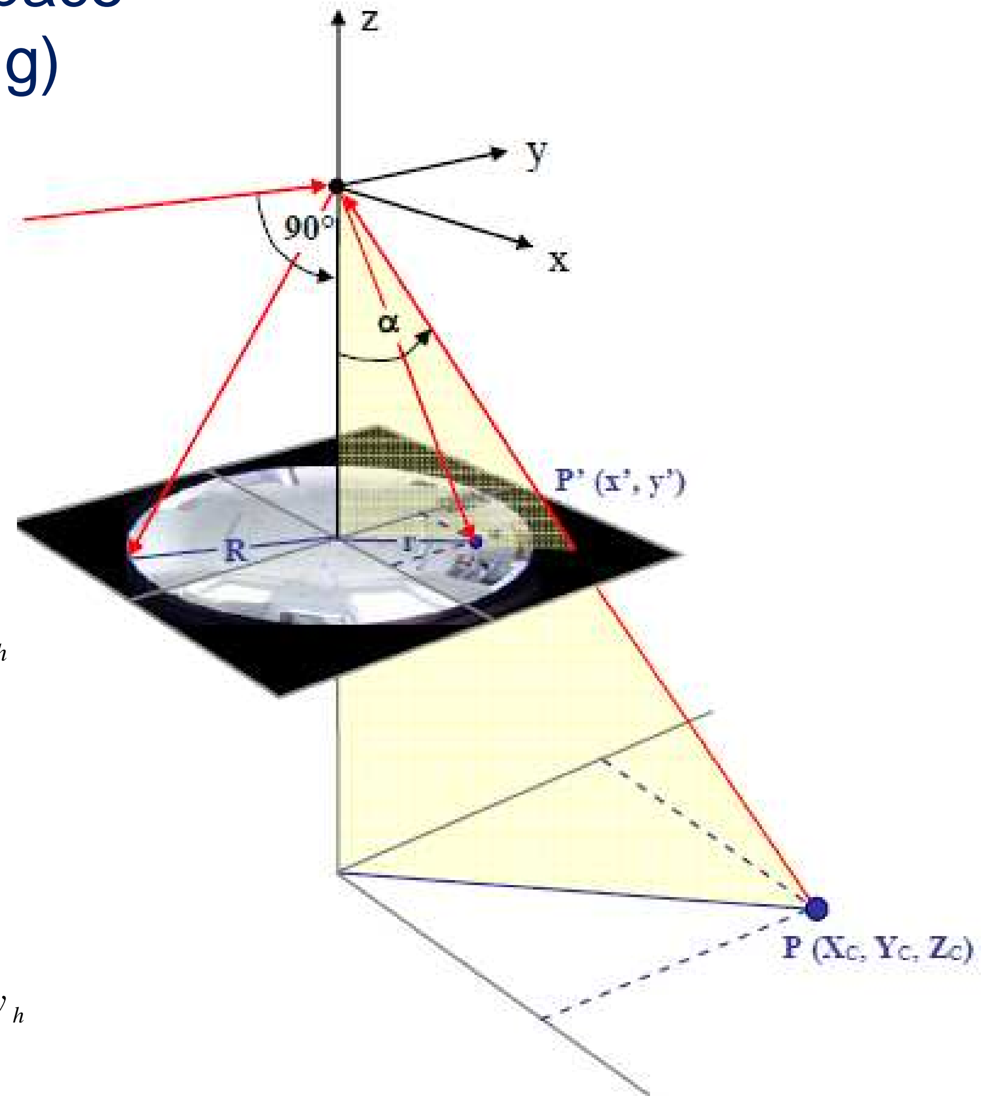
Algorithmic steps (A)

From rectilinear to fisheye space coordinates (inverse mapping)

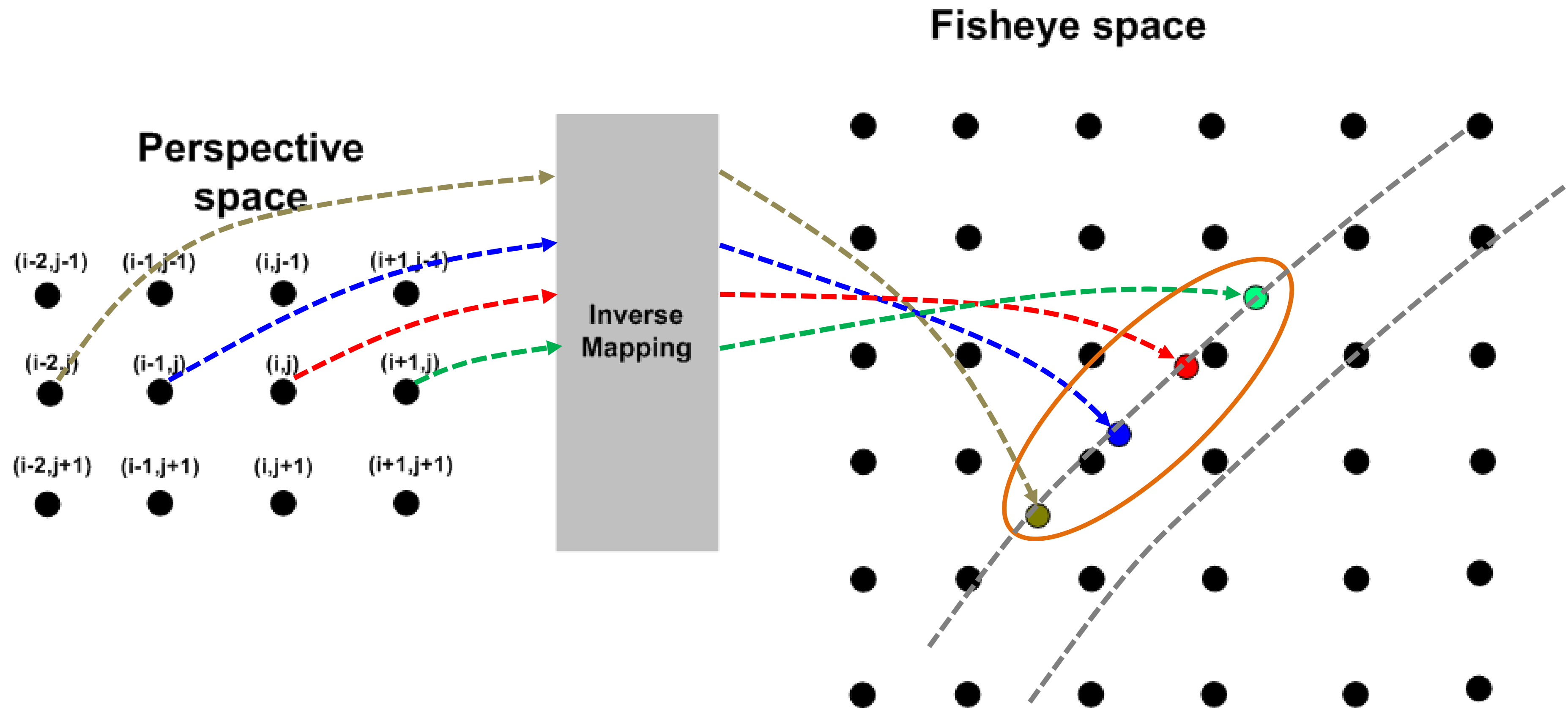
$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} r11 & r12 & r13 \\ r21 & r22 & r23 \\ r31 & r32 & r33 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

$$x = \frac{\frac{2R}{\pi} a\tan\left[\frac{\sqrt{(Xc)^2 + (Yc)^2}}{Zc}\right]}{\sqrt{\left(\frac{Yc}{Xc}\right)^2 + 1}} + d_x + x_h$$

$$y = \frac{\frac{2R}{\pi} a\tan\left[\frac{\sqrt{(Xc)^2 + (Yc)^2}}{Zc}\right]}{\sqrt{\left(\frac{Xc}{Yc}\right)^2 + 1}} + d_y + y_h$$



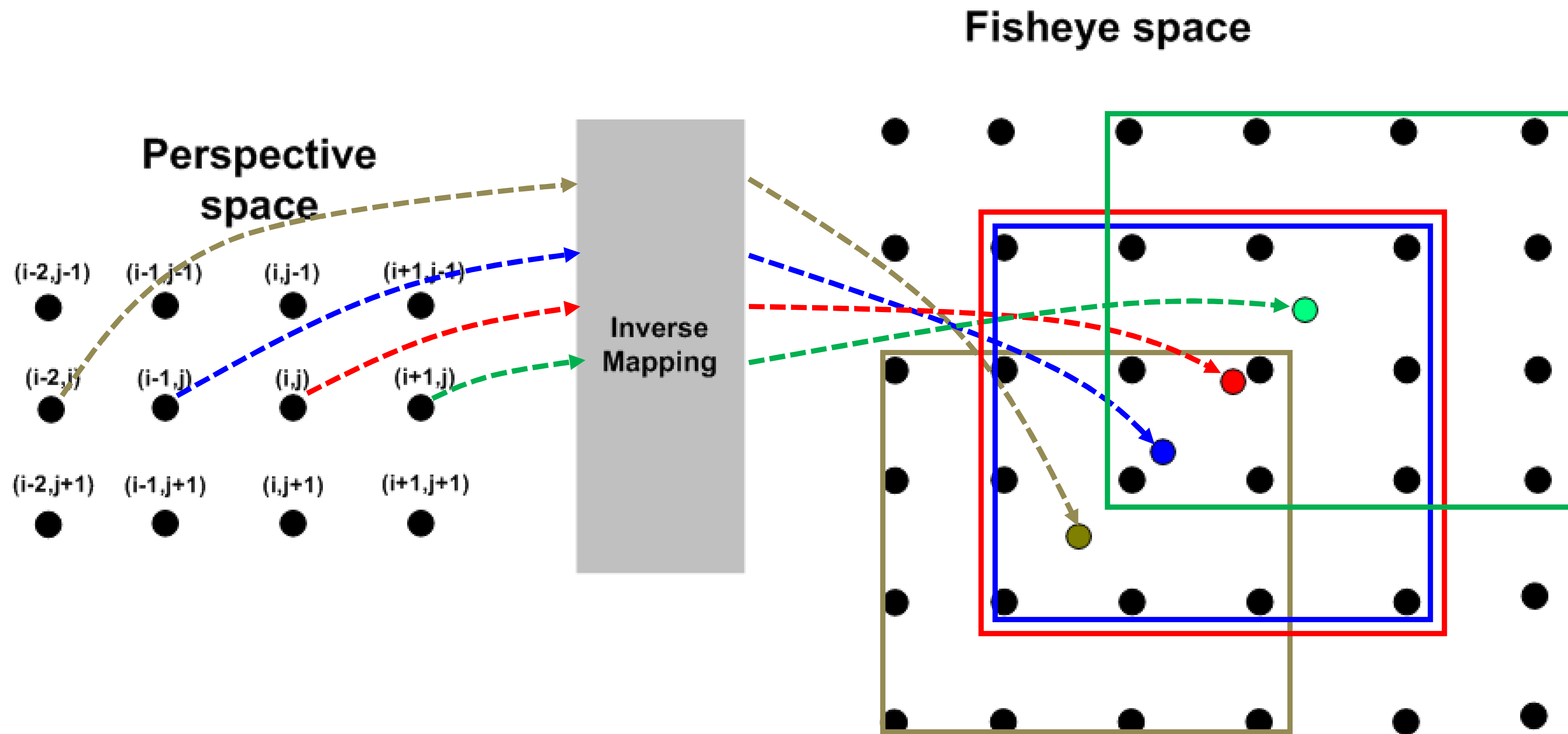
Algorithmic steps (B)



- Approximate pixel values in fractional positions in *Fisheye space*
- Complex memory access pattern due to non-linear projection trajectory

Algorithmic steps (B)

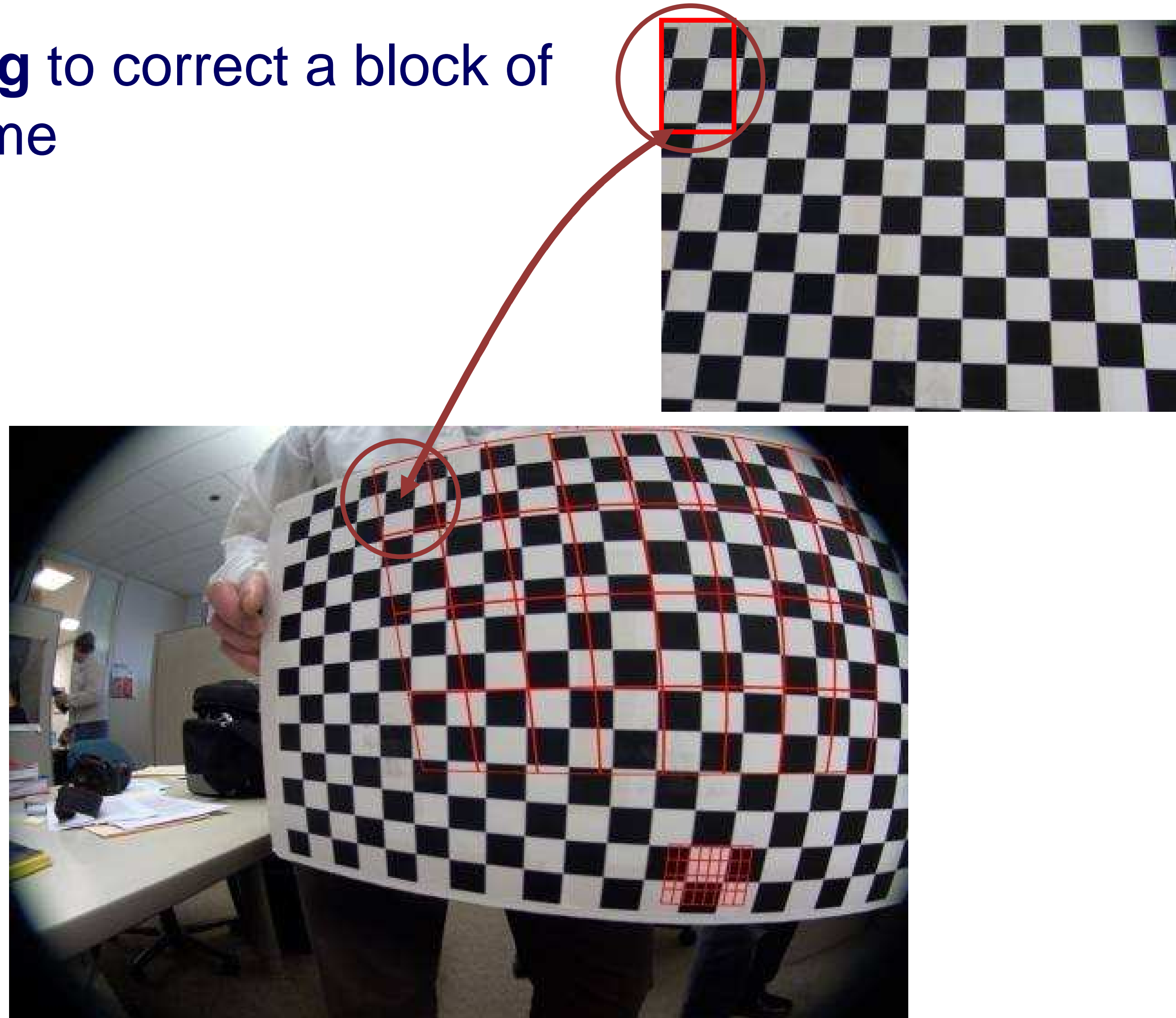
Bicubic interpolation



- **Bicubic interpolation** uses a 4x4 window of pixels to approximate intermediate points
- Interpolation weights depend on the relative position of the intermediate point

Architectural Optimization Strategies

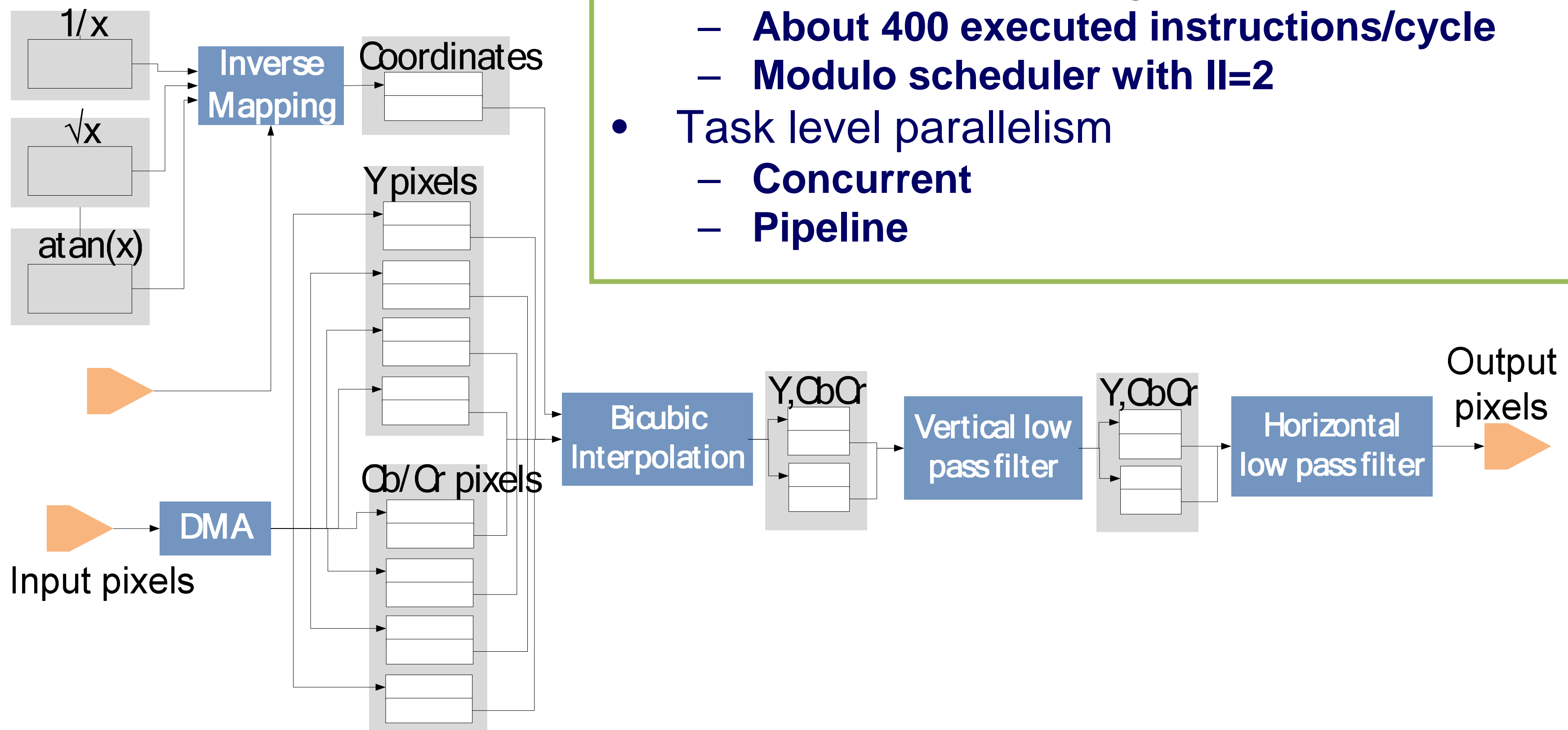
Use **block tiling** to correct a block of pixels at a time



Streaming Accelerator

Parallelism extracted by Proteus

- Instruction Level Parallelism (ILP) naturally expressed in streaming
  - **About 400 executed instructions/cycle**
  - **Modulo scheduler with II=2**
- Task level parallelism
  - **Concurrent**
  - **Pipeline**

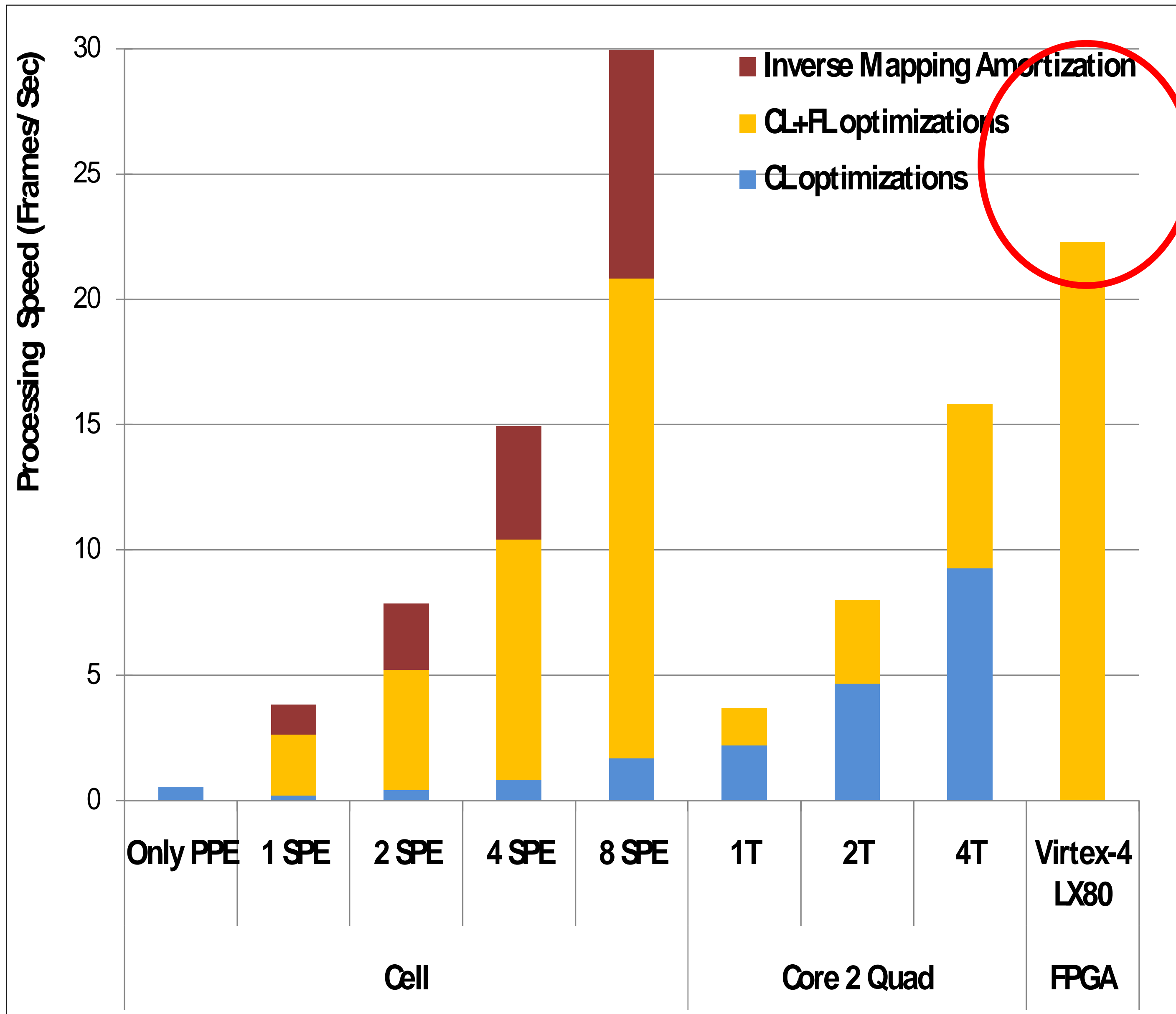


Hardware details

Virtex-4, LX-80 FPGA

SPEED	**Clock freq.**	62.5 MHz (single clock)
	Throughput	22 frames/sec
AREA	**Logic Slices**	11082 (30%)
	DSP48 units	71 (88%)
	BRAMs	109 (54.5%)
	BRAM types (number per type)	4096x8 (15) 13728x50 (1) 256x16, 512x7, 256x17, 256x7, 256x17, 256x7 6864 x 8 (2) 6864 x 16 (2) 3432 x 8 (2) 3432 x 16 (2)

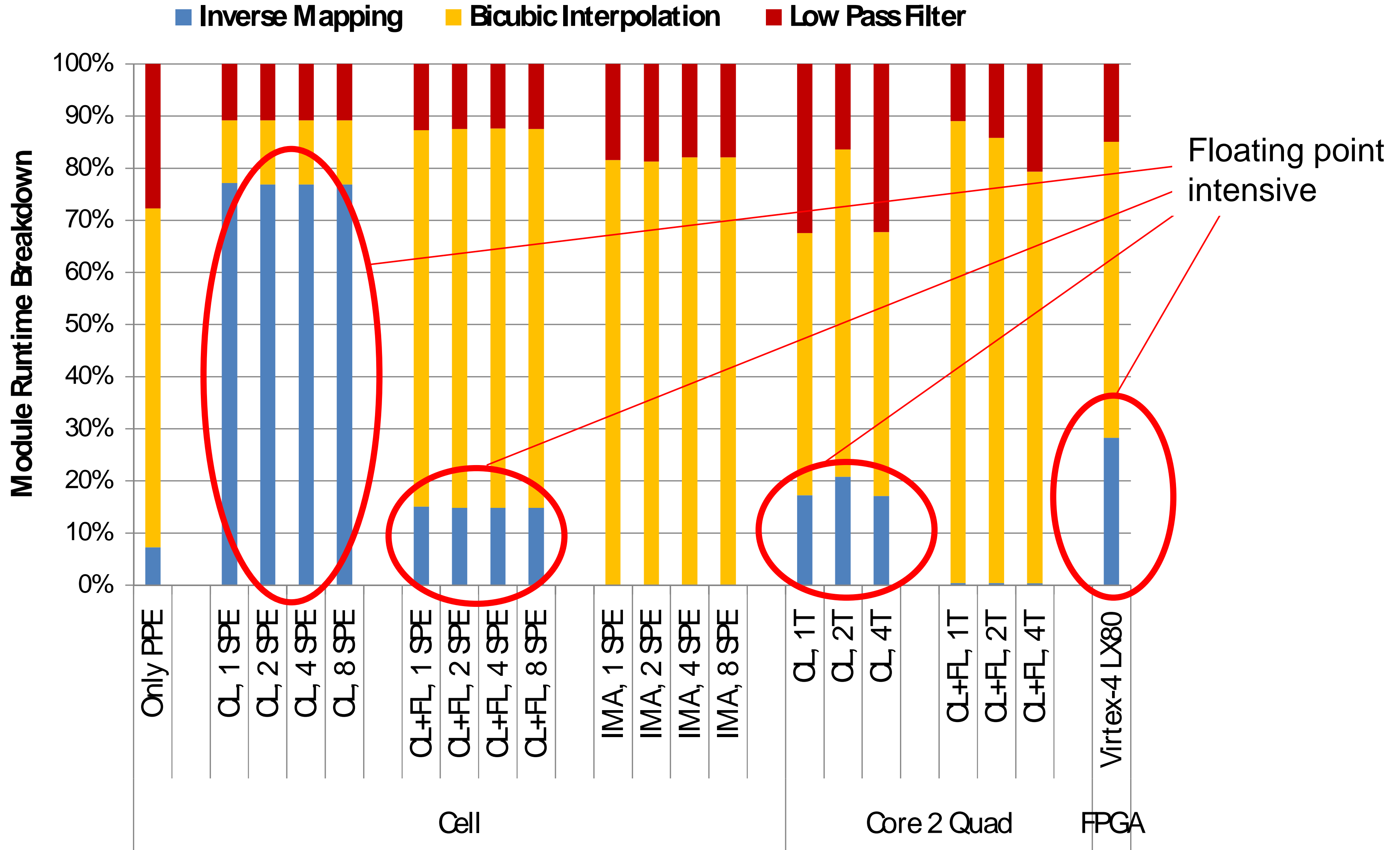
Performance



Bounded by available FPGA SRAM

(unpublished results)

Performance



(unpublished results)

Research summary

- Separation of concerns
  - **Memory access patterns are defined explicitly by programmer for RSVP and Proteus**
- Extend stream descriptors
- Plan to open source Proteus tool