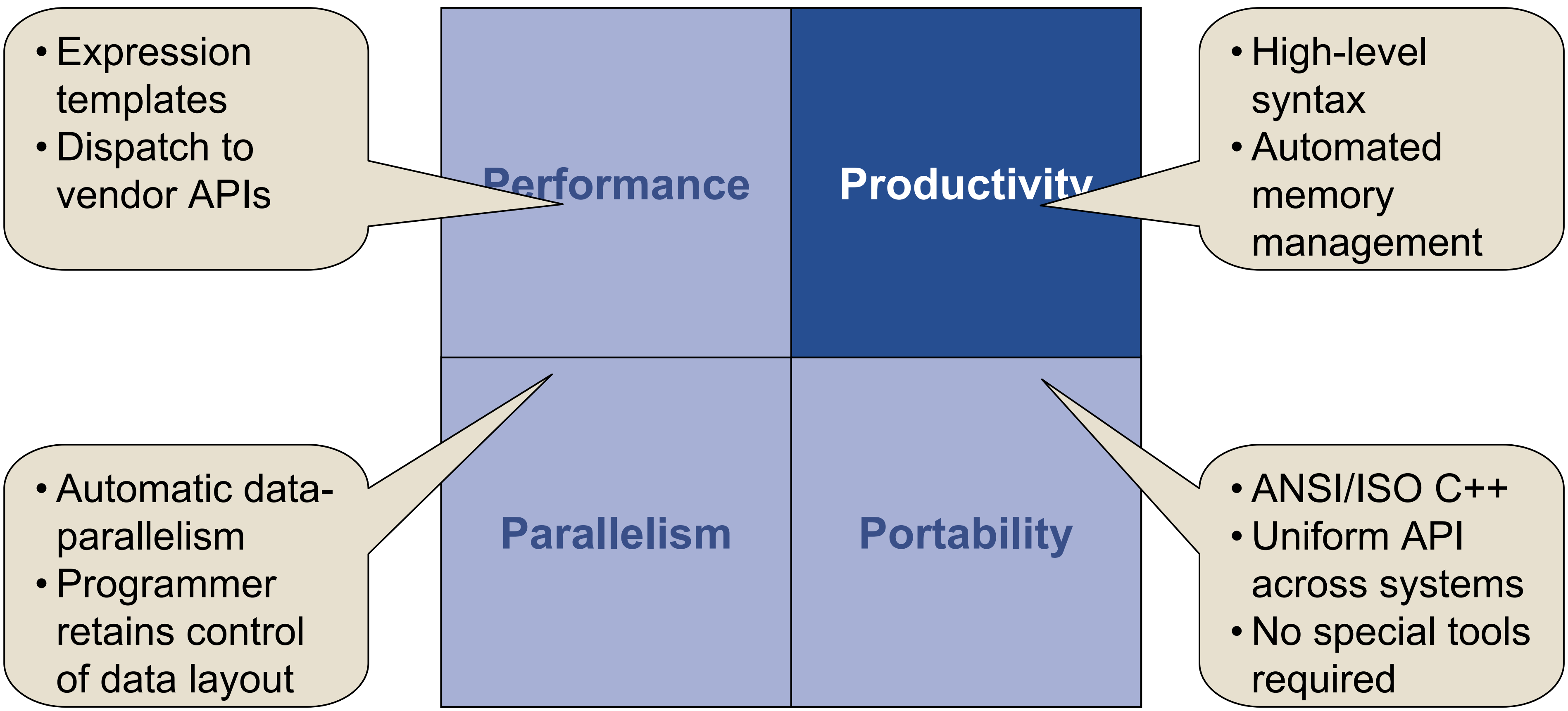
Sourcery VSIPL++

HPEC Workshop
Sep 21, 2005

Jules Bergmann
CodeSourcery, LLC
jules@codesourcery.com
650-704-4014

Sourcery VSIPL++

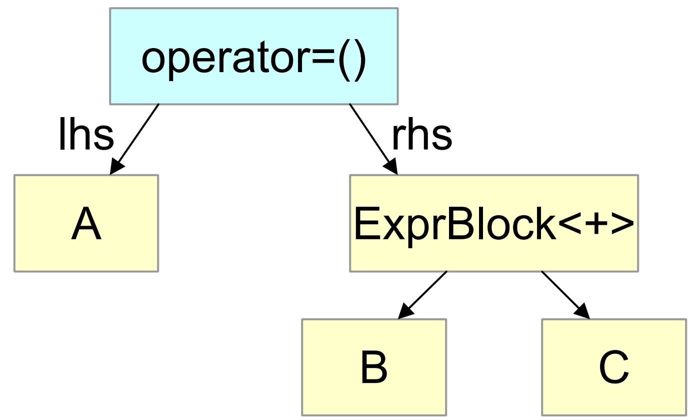
First optimized implementation of VSIPL++ API specification



Key Technology: Expression Templates

Source Code (what the user writes)	**Expression Template** (what the library sees)

```
A = B + C;
```



Information available:

- Expression type
- Data Layout
  - data distribution
  - dimension ordering
  - stride

Enables Optimized Performance:

- Dispatch to vendor library
  - Including strip-mining
- Optimal algorithm selection
  - transpose vs memcpy
- Loop fusion
- Decomposition

Expression Templates Enable Library to Perform Compiler-Type Optimizations

Low-Level Library Interface

Sourcery VSIPL++ Math Library Interface:

- Dispatch VSIPL++ constructs directly to vendor/platform libraries.
- Used for expression templates, signal processing, linear algebra.

Use best-performing vendor/platform libraries:

- Signal Processing (Intel IPP, Mercury SAL)
- Linear Algebra (ATLAS, Intel MKL)
- Fourier Transforms (FFTW, Spiral)

Extensible Dispatch:

- Designed to ease addition of new vendor/platform libraries
- Designed to allow mixing of multiple libraries and user routines.

Direct Data Access:

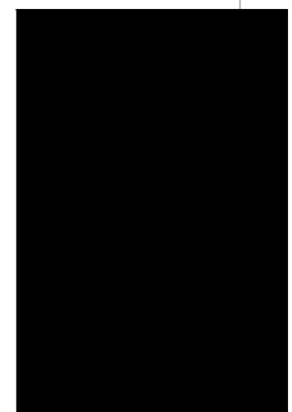
- Direct access to block's data: pointer & stride
- Request particular formats (dimension ordering, packing, etc).

Enable Portable Applications with
Platform-Optimized Performance

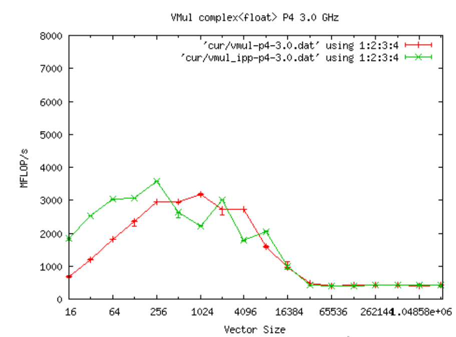
Instrumentation

Achieving good performance on today's architecture is challenging!

Sourcery VSIPL++ provides instrumentation to make this easier



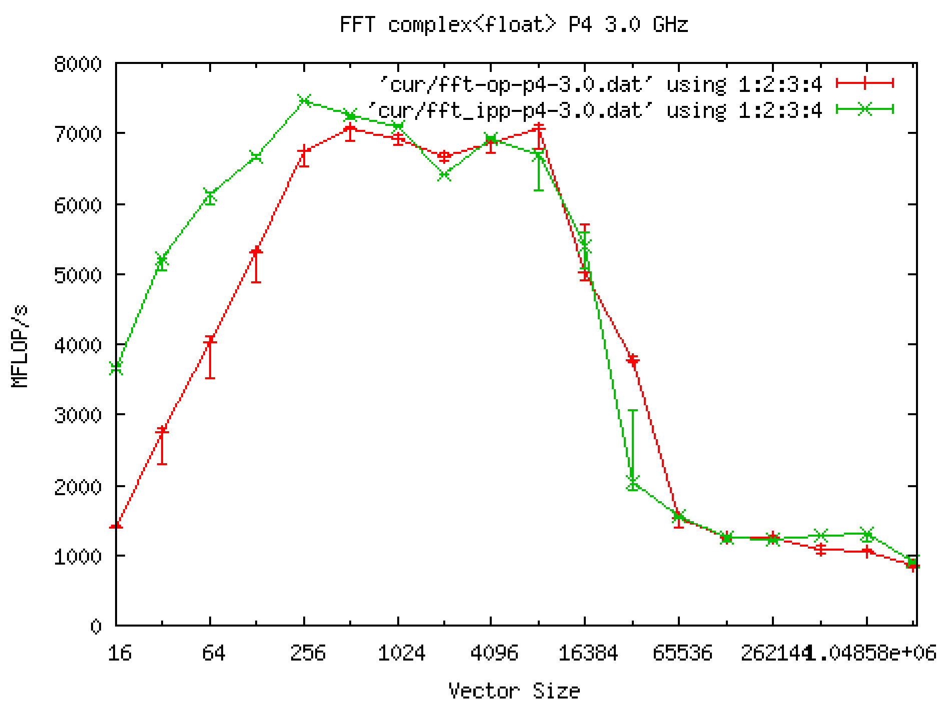
- **Event Profiling:**
  - User and Library events can be collected
  - Create a timeline
  - Record cumulative event information
- **Performance Measurements:**
  - Many signal processing objects record their performance
  - Query actual performance of operations

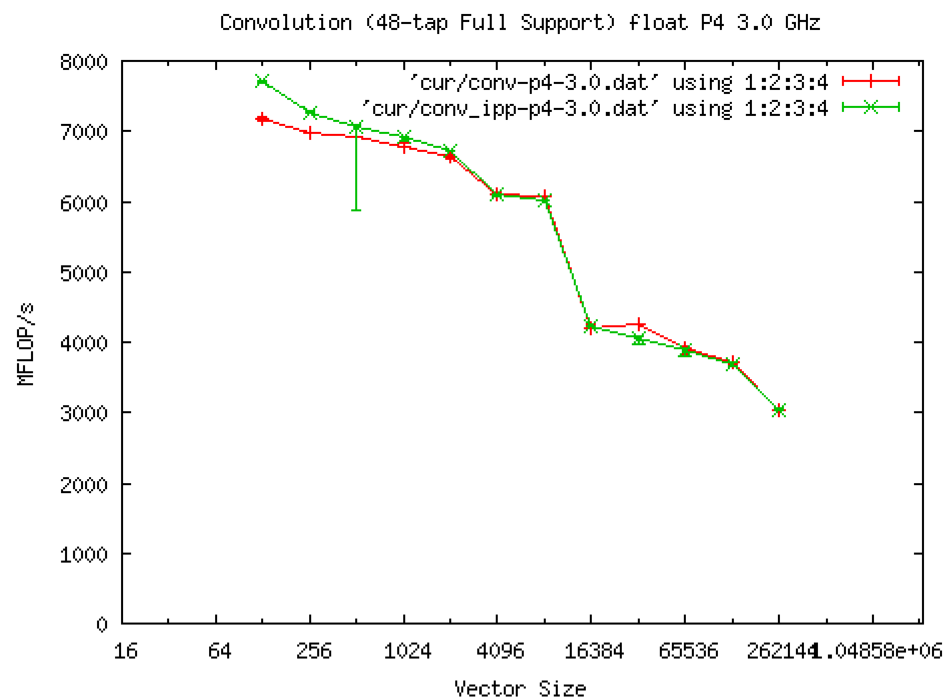


- **Development Mode:**
  - Issue warnings on sub-optimal operations.
  - Data copies, reorganizations, conversions
  - Memory allocations
- **Platform Characterization:**
  - Characterize performance of platform, compiler, vendor libraries.

Makes is Easier to Optimize Application and Library Usage for Target Platform

Kernel Performance





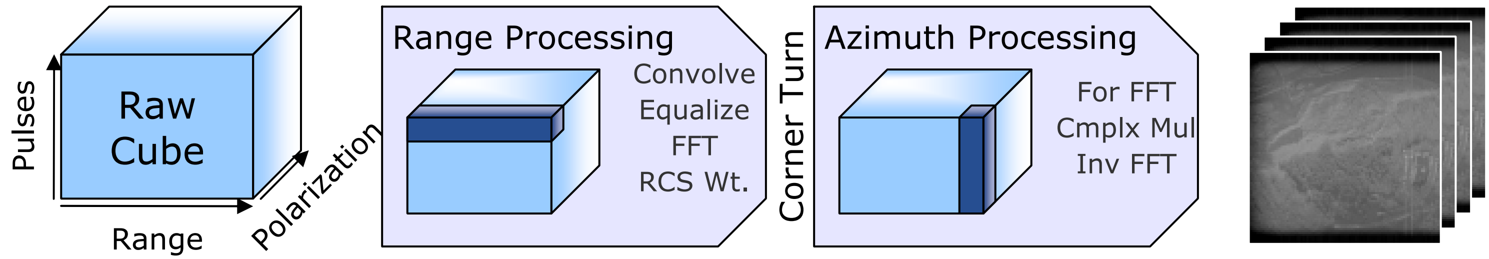
FFT Kernel
- Uses IPP, FFTW Libraries

VSIPL++ Performance
- 6927 MFLOPS @ 1024 pt (57.7%)
- 6655 MFLOPS @ 2048 pt (55.4%)

Convolution
- Uses IPP, C++ Implementation

VSIPL++ Performance
- 6647 MFLOPS @ 2048 pt (55.3%)

SAR Image Formation



Ported RASSP SARSIM to VSIPL++

- (Lines of C code)

Kernels used: FFT, convolution, vmul, corner-turn

Representative of SIP Applications

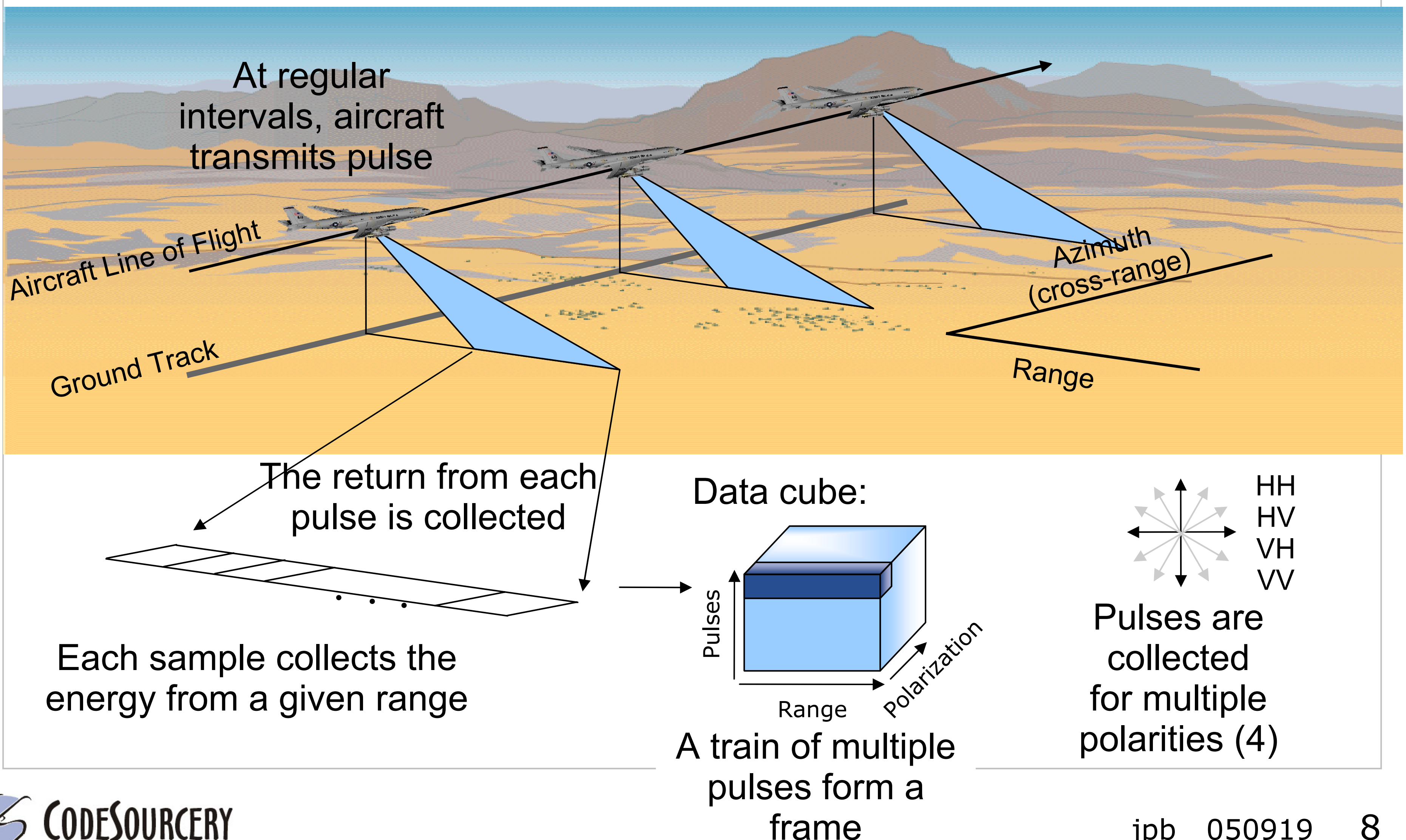
- Similar to MTI pulse-compression and doppler filtering

Adjustable Waveform:

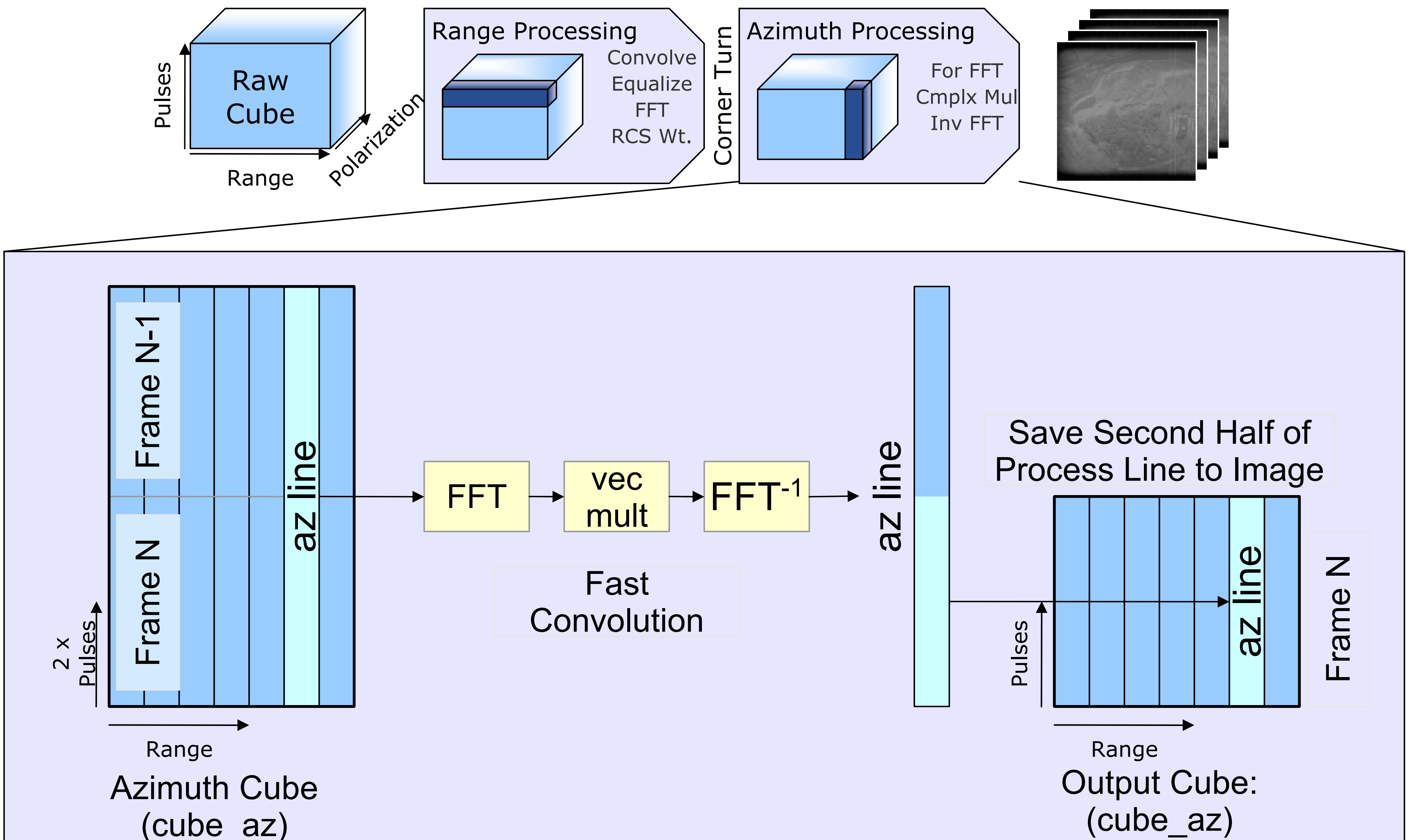
- 4 polarizations x 64 pulses x 256 range cells
- 4 polarizations x 512 pulses x 2048 range cells

SAR Data Collection

Cartoon of SAR data collection ...

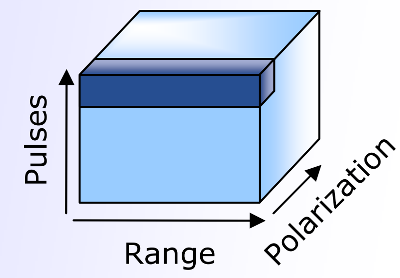


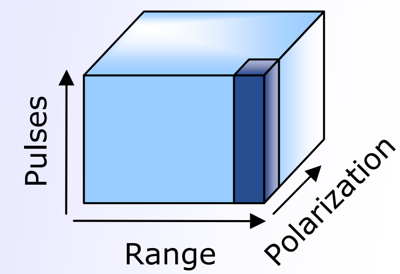
Azimuth Processing



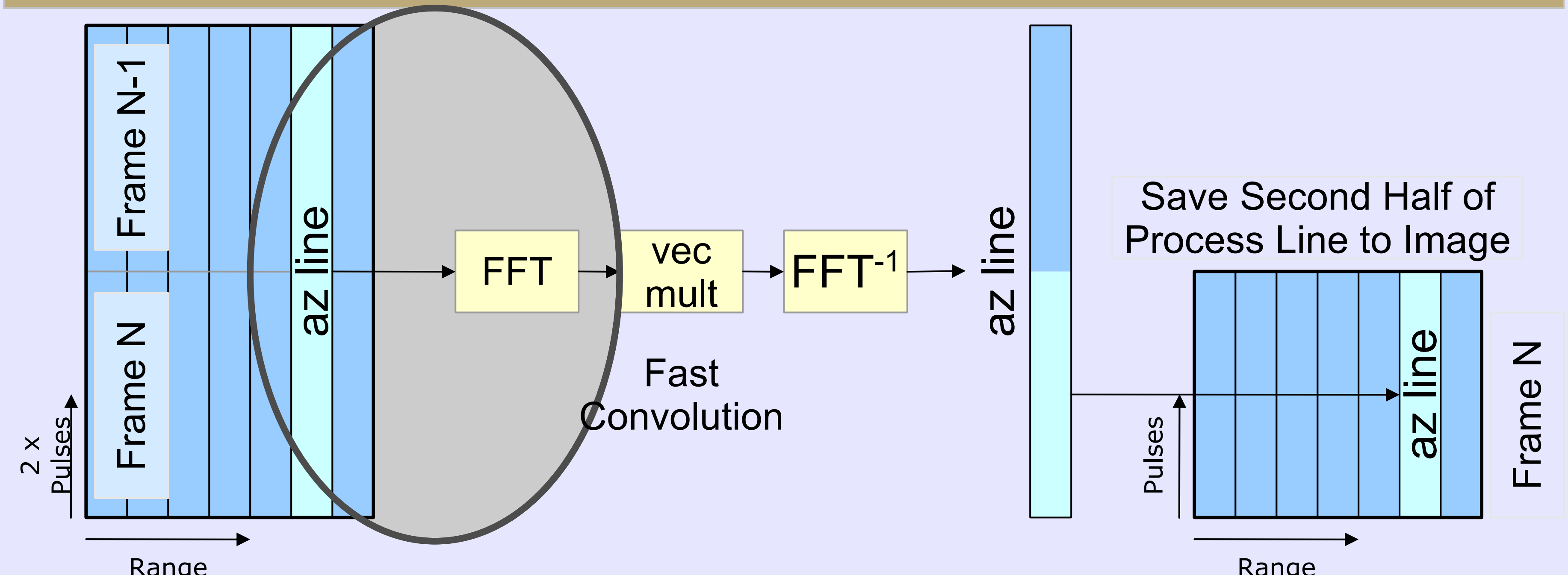
Data Layout

```
typedef complex<float> cval_type;

typedef Dense<3, cval_type, tuple<0, 1, 2> >
        rg_block_type;

typedef Tensor<cval_type, rg_block_type> cube_rg_type;

typedef Dense<3, cval_type, tuple<0, 2, 1> >
        az_block_type;

typedef Tensor<cval_type, az_block_type> cube_az_type;

cube_in_type  cube_in (npol, npulse, ncsamples);
cube_rg_type  cube_rg (npol, npulse, nrange);
cube_az_type  cube_az (npol, 2*npulse, nrange);
cube_out_type cube_out(npol, npulse, nrange);
```



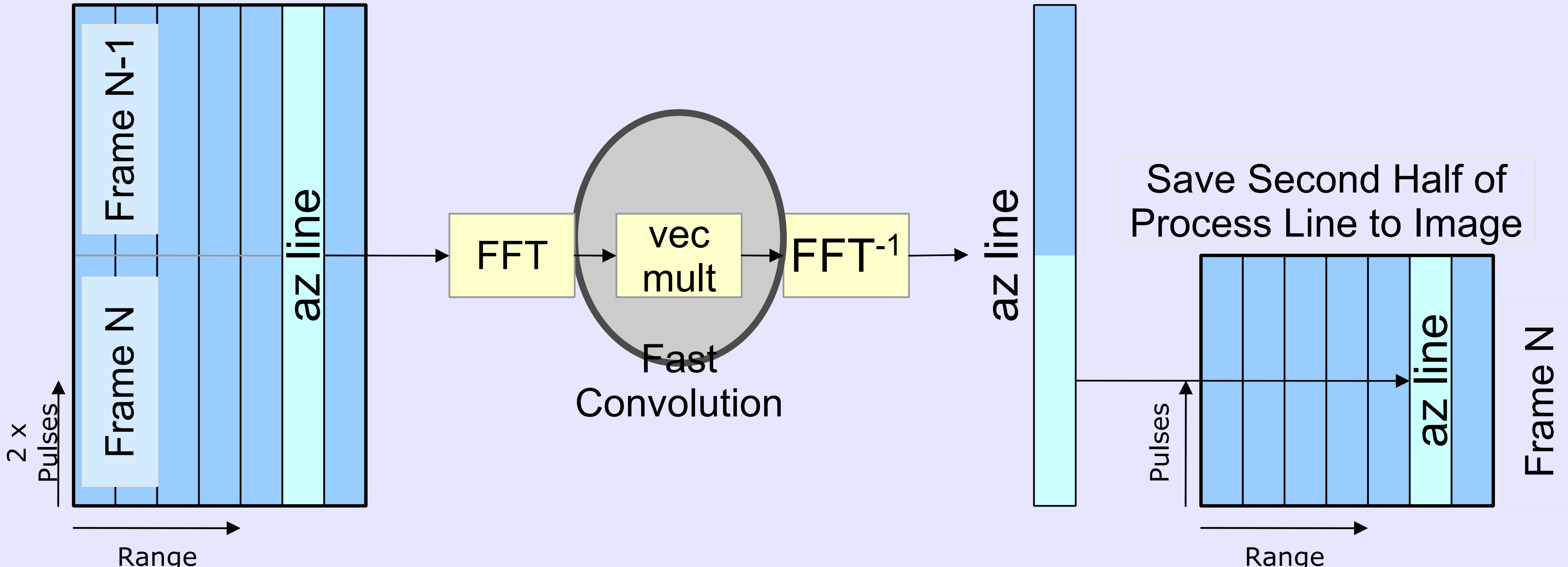


Azimuth Processing



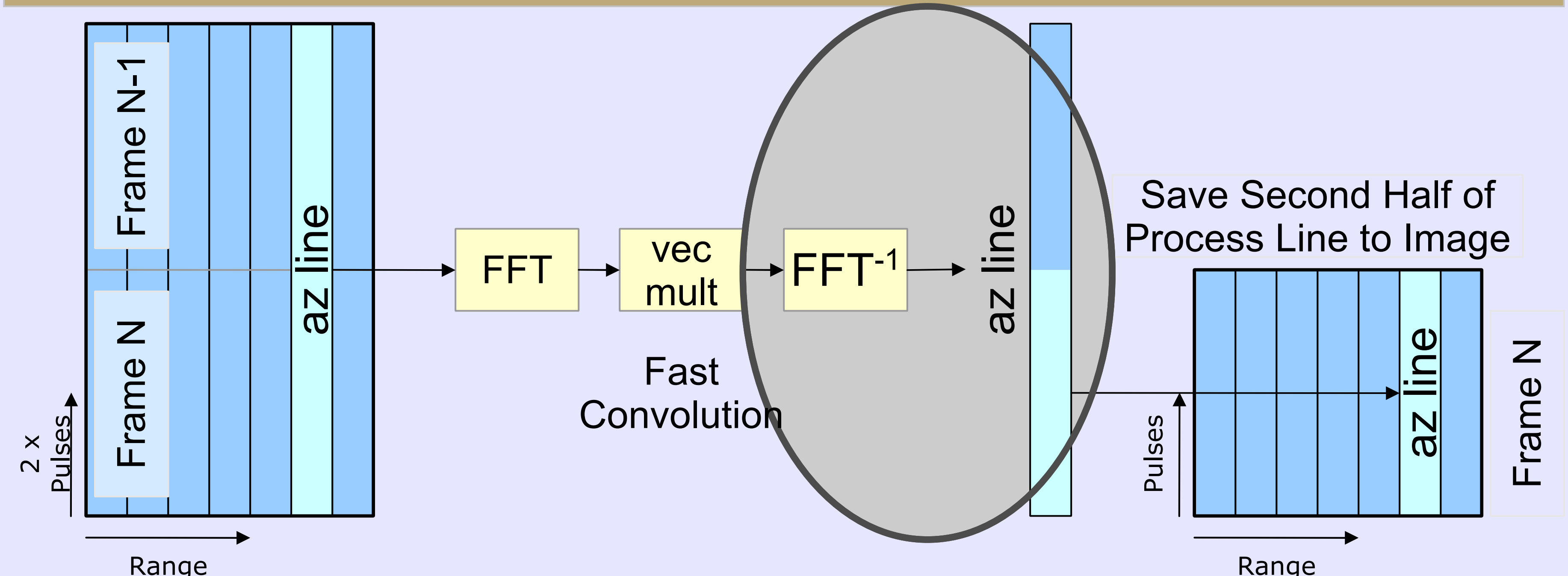

```
// Forward FFT into azbuf1
az_for_fft(cube_az(pol, whole, i), azbuf1);
// Apply Kernel.
index_type kernel = k0 + i * 16 / nrange_;
azbuf1 *= cphase.row(kernel);
// Inverse FFT into azbuf2
az_inv_fft(azbuf1, azbuf2);
// Save second half into cube_img.
cube_img(pol, whole, i) = azbuf2(second_dom);
```

Azimuth Processing




```
// Forward FFT into azbuf1
az_for_fft(cube_az(pol, whole, i), azbuf1);

// Apply Kernel.
index_type kernel = k0 + i * 16 / nrange_;
azbuf1 *= cphase.row(kernel);

// Inverse FFT into azbuf2
az_inv_fft(azbuf1, azbuf2);

// Save second half into cube_img.
cube_img(pol, whole, i) = azbuf2(second_dom);
```

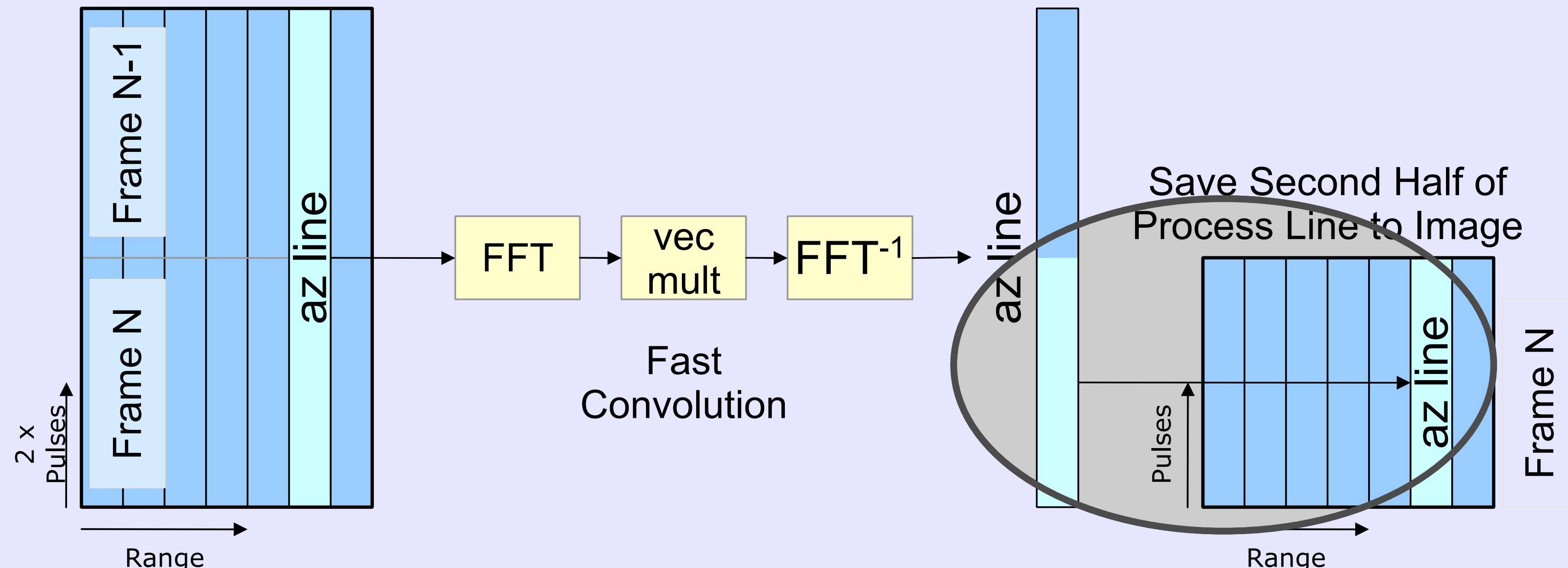
Azimuth Processing



Az

```
// Forward FFT into azbuf1
az_for_fft(cube_az(pol, whole, i), azbuf1);

// Apply Kernel.
index_type kernel = k0 + i * 16 / nrange_;
azbuf1 *= cphase.row(kernel);

// Inverse FFT into azbuf2
az_inv_fft(azbuf1, azbuf2);

// Save second half into cube_img.
cube_img(pol, whole, i) = azbuf2(second_dom);
```

Azimuth Processing

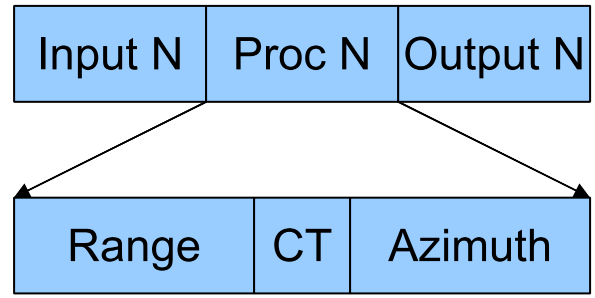



```
// Forward FFT into azbuf1
az_for_fft(cube_az(pol, whole, i), azbuf1);

// Apply Kernel.
index_type kernel = k0 + i * 16 / nrange_;
azbuf1 *= cphase.row(kernel);

// Inverse FFT into azbuf2
az_inv_fft(azbuf1, azbuf2);

// Save second half into cube_img.
cube_img(pol, whole, i) = azbuf2(second_dom);
```

Multi-Bufferring

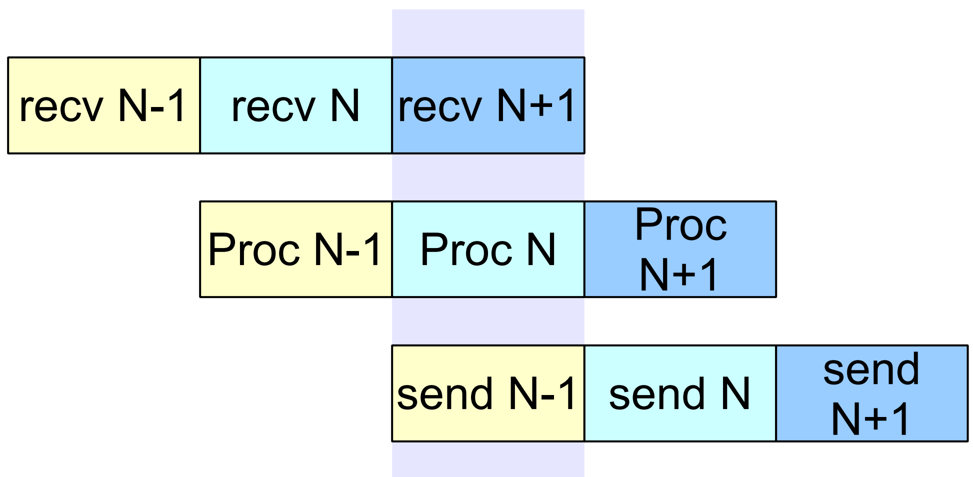
Processing organized by frame

- Receive input frame
- Range process
- Azimuth process
- Send output frame



Model input and output as multi-buffered IO

- While processing frame N ...
- ... input frame N+1 received
- ... output frame N-1 sent.
- Use admit/release to switch between buffers



Serial Performance

Range Processing

Function	Size	MFLOPS	Predicted
Conv	2048 pt	4891	6647
Vec Mult	2048 pt	2934	3184
FFT	2048 pt	6670	6655

Range Total	**3189**

Azimuth Processing

Function	Size	MFLOPS	Predicted
For FFT	1024 pt	4022	6927
Vec Mult	1024 pt	2425	3184
Inv FFT	1024 pt	4110	6927

Azimuth Total	**2818**

Performance for Large Dataset (4 x 512 x 2048)

Parallel SAR

Changes necessary to parallelize:

Apply maps to data structures:

- Determine how to distribute data:
  - Undistributed
  - Block-cyclic distributions
- Determine where to distribute:
  - Processor set

Code with explicit data-parallelism:

- Translate to work on local views:
  - Get local view from global view
  - Index translation.
- Converted code still works in serial.

Code with implicit data-parallelism (for example Fftm, vmmul):

- No conversion necessary.

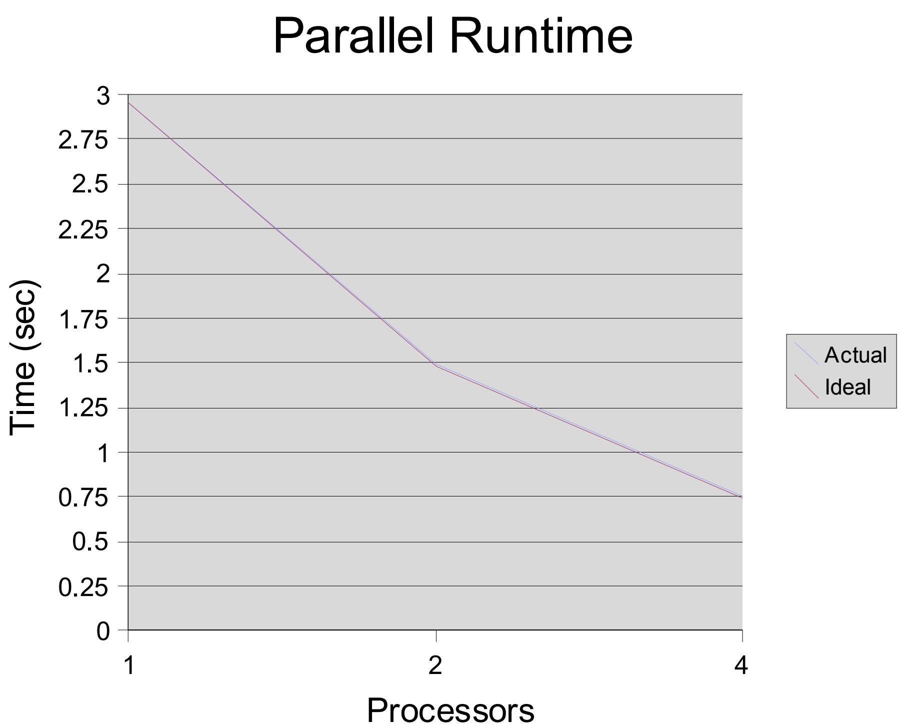
Apply Maps to Data Structures

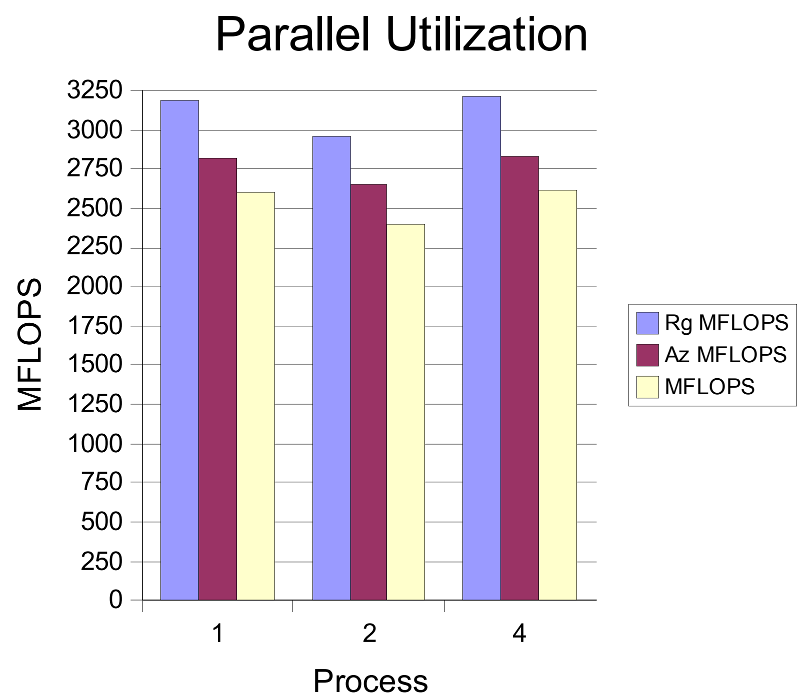
```
typedef complex<float> cval_type;

typedef Map<> map_az_type;

typedef Dense<3, cval_type, tuple<0, 2, 1>,
                map_az_type>
        az_block_type;

typedef Tensor<cval_type, az_block_type> cube_az_type;

map_az_type map_az(Block_dist(np), // pol    : distributed
                   Block_dist(1),  // pulses: whole
                   Block_dist(1)); // range : whole

cube_az_type cube_az(npol, 2*npulse, nrange, map_az);
```

Parallel Performance

Mapping Strategy: Process polarizations in parallel

- VSIPL++ delivers near ideal speedup to 4 processors





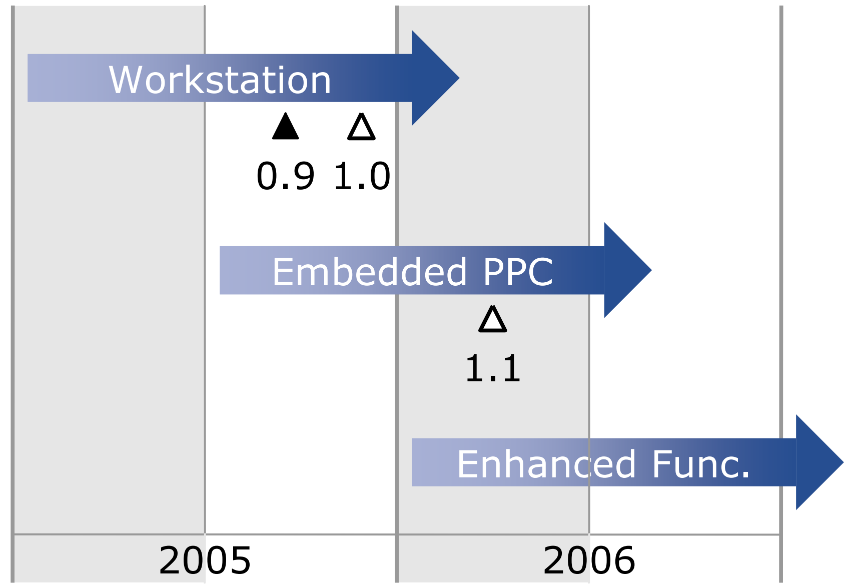
Near-linear speedup

Processor utilizations not affected by parallelization

Availability

Sourcery VSIPL++ is available today for download.

- Technology Preview – Version 0.9



For more information and download:

- **Visit our website: www.codesourcery.com/vsiplplusplus**

Join our mailing list:

- **Announcements: vsipl++-announce@codesourcery.com**

Sourcery VSIPL++

HPEC Workshop
Sep 21, 2005

Jules Bergmann
CodeSourcery, LLC
jules@codesourcery.com
650-704-4014