Sourcery VSIL++
HPEC Workshop
Sep 21, 2005

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

First optimized implementation of VSIPL++ API specification

- Expression templates
- Dispatch to vendor APIs
- High-level syntax
- Automated memory management
- ANSI/ISO C++
- Uniform API across systems
- No special tools required

- Automatic data-parallelism
- Programmer retains control of data layout
- Productivity
- Portability
- Performance
Key Technology: Expression Templates

Source Code
(what the user writes)

Expression Template
(what the library sees)

A = B + C;

operator=()

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

Information available:
- Expression type
- Data Layout
  - data distribution
  - dimension ordering
  - stride

Expression Templates Enable Library
to Perform Compiler-Type Optimizations
Low-Level Library Interface

Sourcery VSIPPL++ Math Library Interface:
- Dispatch VSIPPL++ 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 it 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

At regular intervals, aircraft transmits pulse

The return from each pulse is collected

Each sample collects the energy from a given range

A train of multiple pulses form a frame

Data cube:

Pulses are collected for multiple polarities (4)

Electromagnetic pulse

Ground Track

Azimuth (cross-range)

Range

Aircraft Line of Flight

Cartoon of SAR data collection...
Azimuth Processing

Raw Cube

Range Processing

Azimuth Processing

Pulses

Range

Polarization

Convolve

Equalize

FFT

RCS Wt.

Corner Turn

For FFT

Cmplx Mul

Inv FFT

FFT

vec mult

FFT⁻¹

Fast Convolution

Save Second Half of Process Line to Image

Output Cube:

(cube_az)

jpb 050919
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

!! Forward FFT into azbuf1

```c
az_for_fft(cube_az(pol, whole, i), azbuf1);
```

!! Apply Kernel.

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

!! Inverse FFT into azbuf2

```c
az_inv_fft(azbuf1, azbuf2);
```

!! Save second half into cube_img.

```c
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Size</th>
<th>MFLOPS</th>
<th>Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv</td>
<td>2048 pt</td>
<td>4891</td>
<td>6647</td>
</tr>
<tr>
<td>Vec Mult</td>
<td>2048 pt</td>
<td>2934</td>
<td>3184</td>
</tr>
<tr>
<td>FFT</td>
<td>2048 pt</td>
<td>6670</td>
<td>6655</td>
</tr>
</tbody>
</table>

### Azimuth Processing

<table>
<thead>
<tr>
<th>Function</th>
<th>Size</th>
<th>MFLOPS</th>
<th>Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>For FFT</td>
<td>1024 pt</td>
<td>4022</td>
<td>6927</td>
</tr>
<tr>
<td>Vec Mult</td>
<td>1024 pt</td>
<td>2425</td>
<td>3184</td>
</tr>
<tr>
<td>Inv FFT</td>
<td>1024 pt</td>
<td>4110</td>
<td>6927</td>
</tr>
</tbody>
</table>

### Performance for Large Dataset (4 x 512 x 2048)

- **Range Total**: 3189
- **Azimuth Total**: 2818
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.
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