Scalable SAR on the Cell/B.E.
with Sourcery VSIPL++

HPEC Workshop

Jules Bergmann, Don McCoy, Brooks Moses, Stefan Seefeld, Mike LeBlanc
CodeSourcery, Inc

jules@codesourcery.com
888-776-0262 x705
Outline

- SSCA3 SAR Algorithm
- Sourcery VSIPL++ Implementation
- Performance Analysis
- Optimization
- Results
SSCA3 SSAR Benchmark

Raw SAR Return → Fast-time Filter → Bandwidth Expand → Matched Filter → Range Loop

Digital Spotlighting

Interpolation

Formed SAR Image

Major Computations:
- FFT
- mmul
- mmul
- FFT
- pad
- FFT
- 2D FFT⁻¹
- magnitude

Scalable Synthetic SAR Benchmark
- Created by MIT/LL
- Realistic Kernels
- Scalable
- Focus on image formation kernel
- Matlab & C ref impl avail

Challenges
- Non-power of two data sizes (1072 point FFT – radix 67!)
- Polar -> Rectangular interpolation
- 5 corner-turns
- Usual kernels (FFTs, vmul)

Highly Representative Application
Matlab

# Filter echoed signal along fast-time
sFilt = fft( sRaw ) .* ( fastTimeFilter * ones(1,mc) );

# Compress signal along slow-time
sCompr = sFilt.* exp(ic*2*(ks(:)*ones(1,mc)) ... .*(ones(n,1)*sqrt(Xc^2+(-ucs).^2)) - ic*2*ks(:)*Xc*ones(1,mc));
C
ftx2d(S,Mc,N);
for(i=0;i<N;i++) {
    for(j=0;j<Mc;j++){
        tmp_real=S[i][j].real;
        tmp_image=S[i][j].image;
        S[i][j].real=tmp_real*Fast_time_filter[i].real-
                     tmp_image*Fast_time_filter[i].image;
        S[i][j].image=tmp_image*Fast_time_filter[i].real+
                      tmp_real*Fast_time_filter[i].image;
    }
}
for(i=0;i<N;i++) {
    for(j=0;j<Mc;j++){
        tmp_value=2*(state->K[i]*(sqrt(pow(Xc,2)+pow(Uc[j],2))-Xc));
        cos_value=cos(tmp_value);
        sin_value=sin(tmp_value);
        fp[i][j].real=S[i][j].real*cos_value-S[i][j].image*sin_value;
        fp[i][j].image=S[i][j].image*cos_value+S[i][j].real*sin_value;
    }
}
Fast-Time Filter: VSIPL++

**VSIPL++ Setup**

Matrix<complex_t> s_compr_filt(...);

\[
s_{\text{comprfilt}} = \text{vmmul<col>}(\text{fast_time_filter},
\quad \exp(\text{complex}_t(0, 2) \ast \text{vmmul<col>}(k_s, \text{nmc\_ones}) \ast
\quad (\sqrt{\text{sq}(X_c) + \text{sq}(\text{vmmul<row>}(u_c, \text{nmc\_ones}))}) - X_c)));
\]

\[\text{col\_fftm\_type \ ft\_fftm(Domain<2>(n, mc), 1);}\]

**VSIPL++ Compute**

// Filter echoed signal along fast time and compress
\[s_{\text{filt}} = \text{ft\_fftm}(s_{\text{raw}}) \ast s_{\text{compr\_filt}};\]
Source Lines of Code

<table>
<thead>
<tr>
<th>Function</th>
<th>Matlab</th>
<th>Unoptimized</th>
<th>VSIPL++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital Spotlighting</td>
<td>24</td>
<td>109</td>
<td>17</td>
</tr>
<tr>
<td>Interpolation</td>
<td>22</td>
<td>76</td>
<td>23</td>
</tr>
<tr>
<td>Setup</td>
<td>–</td>
<td>–</td>
<td>70</td>
</tr>
<tr>
<td>Other</td>
<td>4</td>
<td>206</td>
<td>93</td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>391</td>
<td>203</td>
</tr>
</tbody>
</table>

VSIPL++ computation routines comparable to Matlab, Optimized VSIPL++ significantly easier than unoptimized C

30-Sep-08
How Fast is SSAR Out of the Box?

<table>
<thead>
<tr>
<th>Function</th>
<th>VSIPL++ Cell/B.E.</th>
<th>VSIPL++ Xeon</th>
<th>C Cell/B.E.</th>
<th>C Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital Spotlighting</td>
<td>0.11 s</td>
<td>1.46 s</td>
<td>429 s</td>
<td>141 s</td>
</tr>
<tr>
<td>Interpolation</td>
<td>4.32 s</td>
<td>1.71 s</td>
<td>217 s</td>
<td>74 s</td>
</tr>
<tr>
<td>Overall</td>
<td>4.43 s</td>
<td>3.15 s</td>
<td>647 s</td>
<td>215 s</td>
</tr>
</tbody>
</table>

Baseline VSIPL++ vs C reference implementation
146 x speedup on Cell/B.E., 68 x speedup on Xeon

Cell/B.E. 3.2 GHz
- 204.8 GF/s peak (SP)
- Sourcery VSIPL++ 2.0
- CML 1.0
- FFTW 3.2-alpha3
- IBM ALF

Intel Xeon 3.6 GHz
- 14.4 GF/s peak (SP)
- Sourcery VSIPL++ 2.0
- IPP 5, MKL 7.21,
- FFTW 3.1.2
Recall the Fast-time Filter:

\[ s_{filt} = ft_{fftm}(s_{raw}) \times s_{compr_filt}; \]
Recall the Fast-time Filter:
\[ s_{\text{filt}} = \text{ft fftm}(s_{\text{raw}}) \times s_{\text{compr filt}}; \]

\[ T_{\text{total}} = 2 T_{\text{DMA}} + T_{\text{FFTm}} + T_{\text{mmul}} \]

Sourcery VSIPL++ fused kernels to improve performance
Can It Go Faster?

Or, what exactly is it doing, and how close is that to peak?

Use Sourcery VSIPL++ profiling to find out:

- Insert profiling statements:
  
  ```cpp
  {  
    Scope<user> scope("ft-halfast", fast_time_filter_ops_);  
    s_filt_ = s_compr_filt_shift_ * ft_fftm_(s_filt_);  
  }
  ```

- Analyze the profiling output:

```plaintext
  doppler to spatial transform : 0.038539 : 10 : 85284728 : 22129.600000
  Fftm row Inv C-C by_ref 756x1144 : 0.014943 : 10 : 43934184 : 29401.100000
  Fftm col Inv C-C by_ref 756x1144 : 0.012679 : 10 : 41349880 : 32614.000000
```
## Performance

<table>
<thead>
<tr>
<th>Cell Performance</th>
<th>Xeon Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Time</strong></td>
</tr>
<tr>
<td>Digital Spotlight</td>
<td></td>
</tr>
<tr>
<td>Fast-time filter</td>
<td>0.018 s</td>
</tr>
<tr>
<td>BW expansion</td>
<td>0.026 s</td>
</tr>
<tr>
<td>Matched filter</td>
<td>0.020 s</td>
</tr>
<tr>
<td>Interpolation</td>
<td></td>
</tr>
<tr>
<td>Range loop</td>
<td>4.25 s</td>
</tr>
<tr>
<td>2D IFFT</td>
<td>0.038 s</td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.069 s</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>4.43 s</td>
</tr>
</tbody>
</table>

Cell/B.E. spends 96% of time in range loop
for (index_type j = 0; j < m; ++j) {
    for (index_type i = 0; i < n; ++i) {
        index_type ikxrows = icKX(i, j);
        index_type i_shift = (i + n/2) % n;
        for (index_type h = 0; h < I; ++h)
            F(ikxrows + h, j) += fsm_t(i_shift, j)
                * SINC_HAM(i, j, h);
    }
    F.col(j)(Domain<1>(j%2, 2, nx/2)) *= -1.0;
}
User-Defined Kernels

- **User provides custom code to run on SPEs**
  - Using CML SPE primitives
  - Hand-coded
- **Sourcery VSIPL++ manages data movement**
  - Dividing computation among SPEs
  - Streaming data to/from SPEs
- **Advantages**
  - Take advantage of SPEs for non-standard algorithms
  - Without having to deal with full complexity of Cell/B.E
  - Intermix seamlessly with Sourcery VSIPL++ code.
User-Defined Kernel Example: SSAR Interpolation

Defining the Kernel

```cpp
for (size_t i = 0; i < out_size; ++i)
    F[i] = std::complex<float>();
for (size_t i = 0; i < n; ++i)
    for (size_t h = 0; h < I; ++h)
        F[icKX[i] + h] += fsm_t[i] * SINC_HAM[i*I + h];
for (size_t i = 0; i < out_size; i+=2)
    F[i] *= -1.0;
```

Using the Kernel

```cpp
Interp_kernel obj;
ukernel::Ukernel<Interp_kernel> uk(obj);
uk(icKX.transpose(),
    SINC_HAM.template transpose<1, 0, 2>(),
    fsm_t.transpose(),
    F.transpose());
```
## User-Defined Kernel SLOCs

<table>
<thead>
<tr>
<th>Function</th>
<th>SLOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Range Loop</td>
<td>9</td>
</tr>
<tr>
<td>Scalar Ukernel</td>
<td>72</td>
</tr>
<tr>
<td>Range Loop</td>
<td>11</td>
</tr>
<tr>
<td>Ukernel Framework</td>
<td>61</td>
</tr>
<tr>
<td>SIMD Ukernel</td>
<td>208</td>
</tr>
<tr>
<td>Range Loop</td>
<td>147</td>
</tr>
<tr>
<td>Ukernel Framework</td>
<td>61</td>
</tr>
</tbody>
</table>

VSIPL++ Ukernels provide high performance

Framework is the same for scalar and SIMD Ukerneals.
# Optimized Cell Results

<table>
<thead>
<tr>
<th>Optimized Cell</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Time</strong></td>
</tr>
<tr>
<td>Digital Spotlight</td>
<td>0.018 s</td>
</tr>
<tr>
<td>Fast-time filter</td>
<td>0.026 s</td>
</tr>
<tr>
<td>BW expansion</td>
<td>0.019 s</td>
</tr>
<tr>
<td>Matched filter</td>
<td>0.182 s</td>
</tr>
<tr>
<td>Interpolation</td>
<td>0.117 s</td>
</tr>
<tr>
<td>Range loop</td>
<td>0.071 s</td>
</tr>
<tr>
<td>Overall</td>
<td>0.458 s</td>
</tr>
</tbody>
</table>

User-Kernel results in 9.7 x speedup on Cell/B.E.
We can also do better on the Xeon

Optimize for Cache Locality

Instead of processing by function:

\[
\begin{align*}
\text{fsm} &= \text{s_decompr_filt_shift} \times \text{decompr_fftm(fsm)}; \\
\text{fsm} &= \text{fs_ref_preshift} \times \text{st_fftm(fsm)};
\end{align*}
\]

Process data by row/column:

\[
\begin{align*}
\text{for} \ (\text{index_type} \ i = 0; \ i < n_; \ ++i) \ { } & \ { } \\
\text{fsm.row(i)} &= \text{s_decompr_filt_shift.row(i)} \times \\
& \qquad \text{decompr_fftm(fsm.row(i))}; \\
\text{fsm.row(i)} &= \text{fs_ref_preshift.row(i)} \times \\
& \qquad \text{st_fftm(fsm.row(i))};
\end{align*}
\]

VSIPL++ respects locality

Good Locality results in Good Performance

30-Sep-08
### Optimized Xeon Results

<table>
<thead>
<tr>
<th>Function</th>
<th>Optimized Xeon</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Performance</td>
</tr>
<tr>
<td><strong>Digital Spotlight</strong></td>
<td>0.908 s</td>
<td>3.2 GF/s</td>
</tr>
<tr>
<td>Fast-time filter</td>
<td>0.160 s</td>
<td>1.8 GF/s</td>
</tr>
<tr>
<td>BW expansion</td>
<td>0.259 s</td>
<td>3.6 GF/s</td>
</tr>
<tr>
<td>Matched filter</td>
<td>0.184 s</td>
<td>3.8 GF/s</td>
</tr>
<tr>
<td><strong>Interpolation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Range loop</td>
<td>1.084 s</td>
<td>0.8 GF/s</td>
</tr>
<tr>
<td>2D IFFT</td>
<td>0.407 s</td>
<td>2.1 GF/s</td>
</tr>
<tr>
<td>Data Movement</td>
<td>-- --</td>
<td>-- --</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>2.60 s</td>
<td></td>
</tr>
</tbody>
</table>

Cache locality optimization: 1.2 x speedup on Xeon
Results

Productivity
- Sourcery VSIPL++ closely matches Matlab algorithm
- Optimized Sourcery VSIPL++ easier than unoptimized C

Performance
- Xeon: 82 x speedup vs C reference
- Cell: 1400 x speedup vs C reference, 5.7 x speedup vs Xeon

Portability
- “Baseline” Sourcery VSIPL++ runs well on both x86 and Cell
- Cell: User-kernel greatly improves performance
  - With minimal effort
- Xeon: Cache-locality requires modest transformation
Scalable SAR on the Cell/B.E. with Sourcery VSIPL++

HPEC Workshop

Jules Bergmann, Don McCoy, Brooks Moses, Stefan Seefeld, Mike LeBlanc
CodeSourcery, Inc

jules@codesourcery.com
888-776-0262 x705