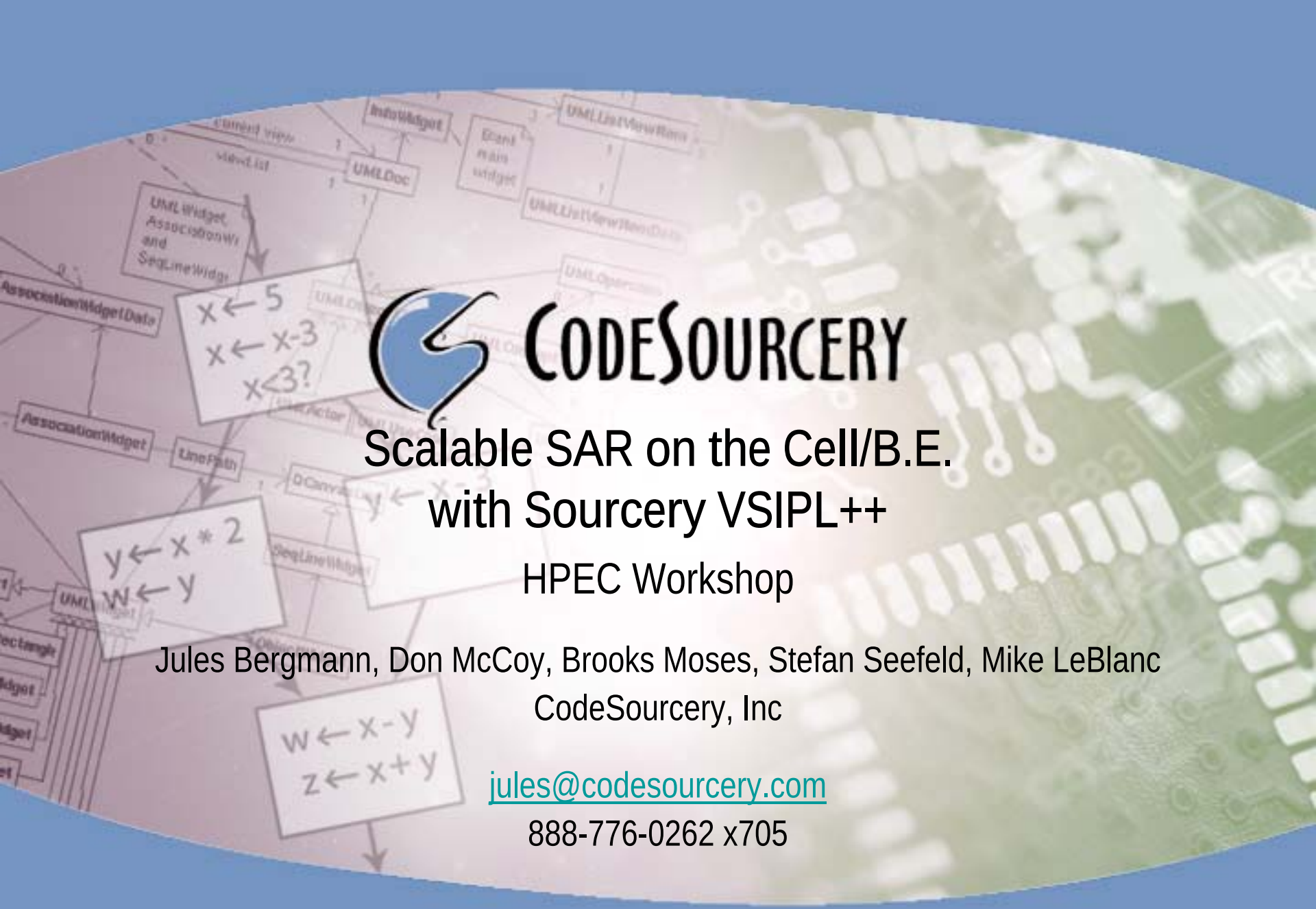


CodeSourcery

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

HPEC Workshop

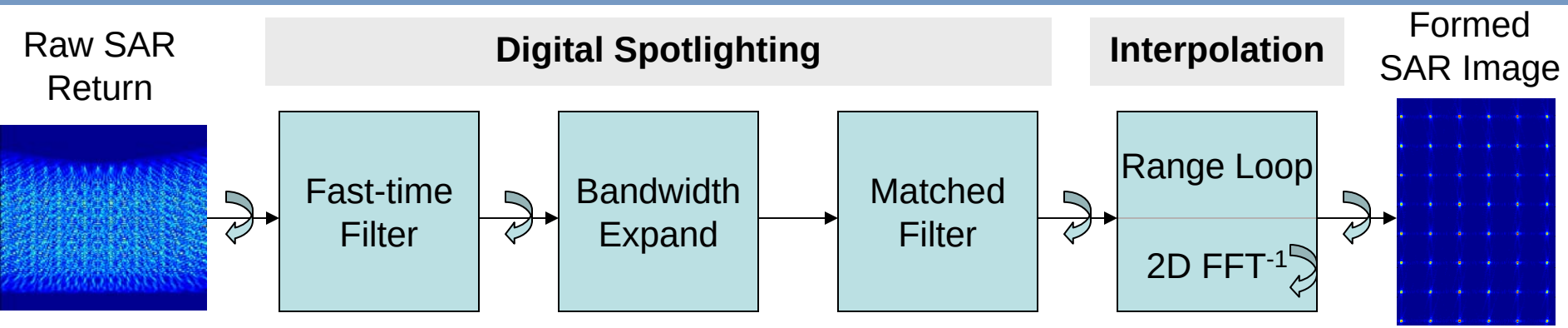
Jules Bergmann, Don McCoy, Brooks Moses, Stefan Seefeld, Mike LeBlanc
CodeSourcery, Inc

jules@codesourcery.com
888-776-0262 x705

Outline

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

SSCA3 SSAR Benchmark



Major Computations:	FFT mmul	mmul FFT pad FFT^{-1}	FFT mmul	interpolate 2D FFT^{-1} magnitude

Scalable Synthetic SAR Benchmark

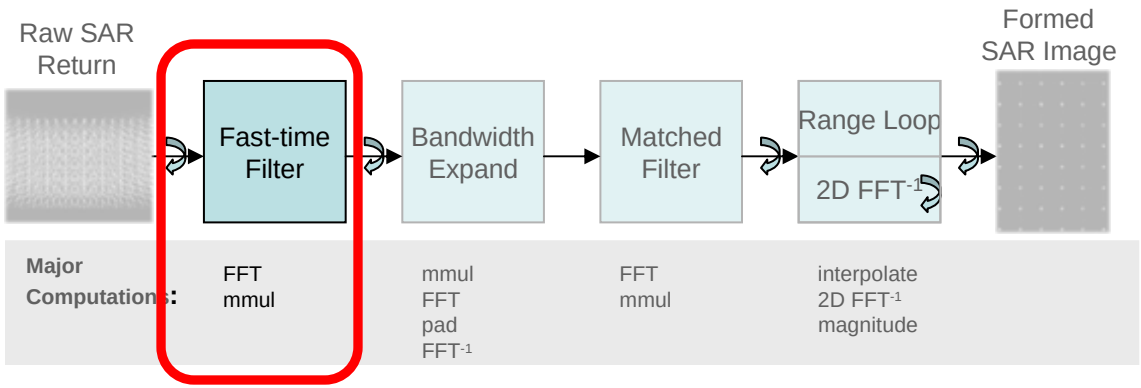
- Created by MIT/LL
- Realistic Kernels
- Scalable
- Focus on image formation kernel
- Matlab & C ref impl avail

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

Fast-Time Filter: Matlab Reference Impl



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));
```

Matlab Fast-Time Filter: 3 Lines

Fast-Time Filter: C Reference Implementation

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;
    }
  }
```

C Fast-Time Filter: 18 Lines

Fast-Time Filter: VSIPL++

VSIPL++ Setup

```
Matrix<complex_t> s_compr_filt(…);

s_compr_filt = vmmul<col>(fast_time_filter,
    exp(complex_t(0, 2) * vmmul<col>(ks, nmc_ones) *
      (sqrt(sq(Xc) + sq(vmmul<row>(ucs, nmc_ones))) - Xc)));

col_fftm_type ft_fftm(Domain<2>(n, mc), 1);
```

VSIPL++ Compute

```
// Filter echoed signal along fast time and compress
s_filt = ft_fftm(s_raw) * s_compr_filt;
```

VSIPL++ Fast-Time Filter: 6 Lines

Source Lines of Code

Function	Matlab	Unoptimized C	VSIPL++
Digital Spotlighting	24	109	17
Interpolation	22	76	23
Setup	--	--	70
Other	4	206	93
Total	50	391	203

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));
```

VSIPL++ Setup

```
Matrix<complex_t> s_compr_filt(…);

s_compr_filt = vmmul<col>(fast_time_filter,
    exp(complex_t(0, 2) * vmmul<col>(ks, nmc_ones) *
      (sqrt(sq(Xc) + sq(vmmul<row>(ucs, nmc_ones))) - Xc)));

col_fftm_type ft_fftm(Domain<2>(n, mc), 1);
```

VSIPL++ Compute

```
// Filter echoed signal along fast time and compress
s_filt = ft_fftm(s_raw) * s_compr_filt;
```

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*Fa
        st_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;
    }
  }
```

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

How Fast is SSAR Out of the Box?

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

Function	VSIPL++ Cell/B.E.	VSIPL++ Xeon	C Cell/B.E.	C Xeon
Digital Spotlighting	0.11 s	1.46 s	429 s	141 s
Interpolation	4.32 s	1.71 s	217 s	74 s
Overall	4.43 s	3.15 s	647 s	215 s

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

Kernel Fusion

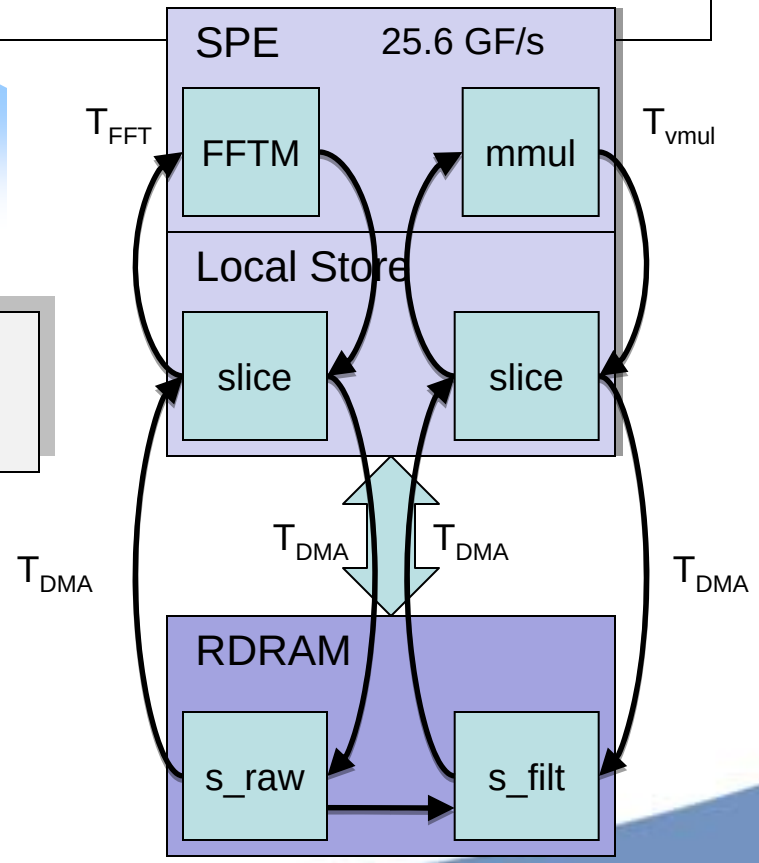
Recall the Fast-time Filter:

```
s_filt = ft_fftm(s_raw) * s_compr_filt;
```

FFTM
•FFT on each column

Matrix-multiply

$$T_{total} = 4\,T_{DMA} + T_{FFTM} + T_{mmul}$$



Kernel Fusion

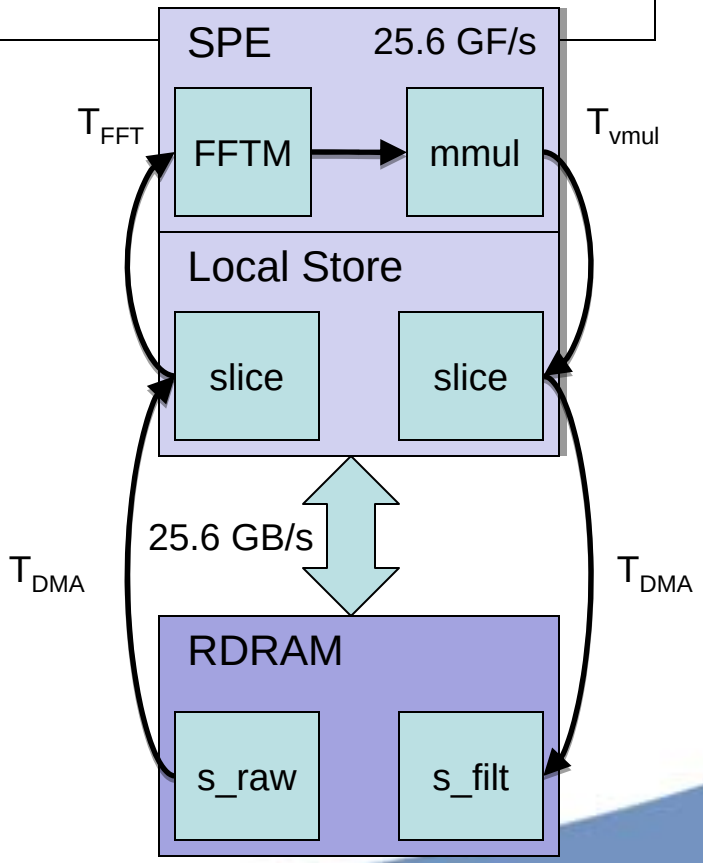
Recall the Fast-time Filter:

```
s_filt = ft_fftm(s_raw) * s_compr_filt;
```

FFTM
•FFT on each column

Matrix-multiply

$$T_{total} = \mathbf{2}\, T_{DMA} + T_{FFTM} + T_{mmul}$$



Sourcery VSIPL++ fused kernels to improve performance

Can It Go Faster?

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

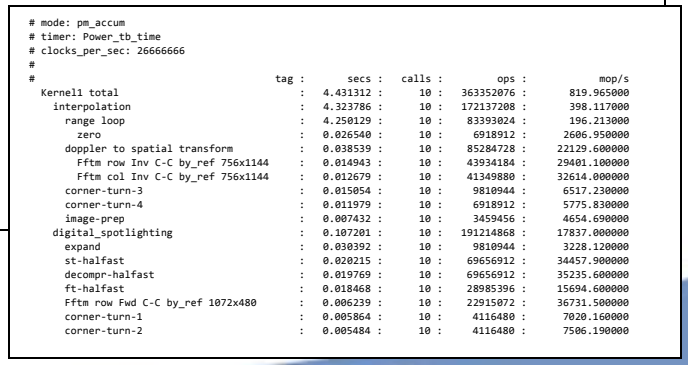
Use Sourcery VSIPL++ profiling to find out:

- Insert profiling statements:

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

- Analyze the profiling output:

```
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
```



```
# mode: pm_accum
# timer: Power_tb_time
# clocks_per_sec: 26666666
#
#                                                tag :      secs :   calls :       ops :          mop/s
  Kernel1 total                                    :  4.431312 :      10 :  363352076 :      819.965000
    interpolation                                  :  4.323786 :      10 :  172137208 :      398.117000
      range loop                                   :  4.250129 :      10 :   83393024 :      196.213000
        zero                                       :  0.026540 :      10 :    6918912 :     2606.950000
      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
      corner-turn-3                                :  0.015054 :      10 :    9810944 :      6517.230000
      corner-turn-4                                :  0.011979 :      10 :    6918912 :      5775.830000
      image-prep                                   :  0.007432 :      10 :    3459456 :      4654.690000
    digital_spotlighting                           :  0.107201 :      10 :  191214868 :     17837.000000
      expand                                       :  0.030392 :      10 :    9810944 :      3228.120000
      st-halfast                                   :  0.020215 :      10 :   69656912 :     34457.900000
      decompr-halfast                              :  0.019769 :      10 :   69656912 :     35235.600000
      ft-halfast                                   :  0.018468 :      10 :   28985396 :     15694.600000
      Fftm row Fwd C-C by_ref 1072x480             :  0.006239 :      10 :   22915072 :     36731.500000
      corner-turn-1                                :  0.005864 :      10 :    4116480 :      7020.160000
      corner-turn-2                                :  0.005484 :      10 :    4116480 :      7506.190000
```

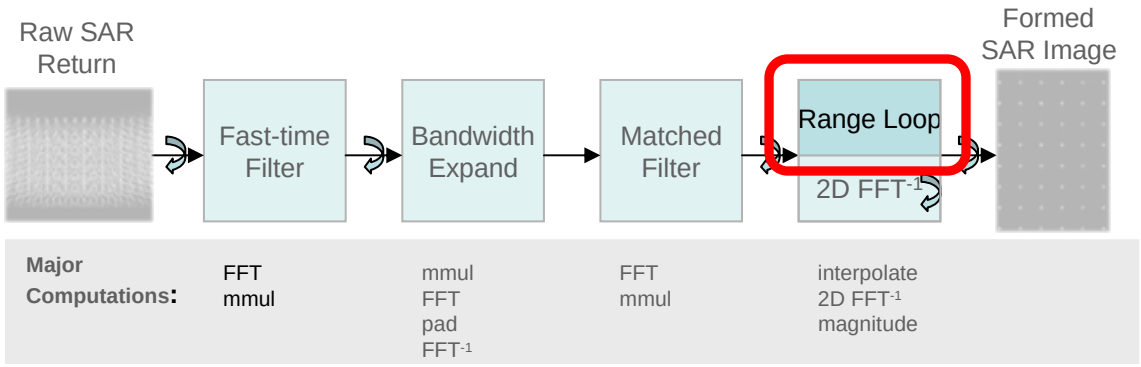
Performance

Cell Performance		
Function	Time	Performance
Digital Spotlight		
Fast-time filter	0.018 s	15.7 GF/s
BW expansion	0.026 s	35.6 GF/s
Matched filter	0.020 s	34.5 GF/s
Interpolation		
Range loop	4.25 s	0.2 GF/s
2D IFFT	0.038 s	22.1 GF/s
Data Movement	0.069 s	5.1 GB/s
Overall	4.43 s	

Xeon Performance		
Function	Time	Performance
Digital Spotlight		
Fast-time filter	0.34 s	0.9 GF/s
BW expansion	0.46 s	2.0 GF/s
Matched filter	0.41 s	1.7 GF/s
Interpolation		
Range loop	1.09 s	0.8 GF/s
2D IFFT	0.41 s	2.1 GF/s
Data Movement	0.32 s	1.1 GB/s
Overall	3.16 s	

Cell/B.E. spends 96% of time in range loop

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;
  }
```

Data dependency (prevents vectorization)

Short inner loop

Hard for VSIPL++ to extract parallelism

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

```
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

```
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

Function	SLOCs
Original Range Loop	9
Scalar Ukernel	72
Range Loop	11
Ukernel Framework	61
SIMD Ukernel	208
Range Loop	147
Ukernel Framework	61

Framework is the same for scalar and SIMD Ukernels.

VSIPL++ Ukernels provide high performance

Optimized Cell Results

Optimized Cell			Baseline	
Function	Time	Performance	Time	Speedup
Digital Spotlight				
Fast-time filter	0.018 s	16.2 GF/s	0.018 s	1.03
BW expansion	0.026 s	36.3 GF/s	0.026 s	1.02
Matched filter	0.019 s	36.5 GF/s	0.020 s	1.06
Interpolation				
Range loop	0.182 s	4.6 GF/s	4.25 s	23.33
2D IFFT	0.117 s	7.3 GF/s	0.038 s	0.33
Data Movement	0.071 s	4.9 GB/s	0.069 s	0.97
Overall	0.458 s		4.43 s	9.68

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:

```
fsm = s_decompr_filt_shift * decompr_fftm(fsm);
fsm = fs_ref_preshift * st_fftm(fsm);
```

Process data by row/column:

```
for (index_type i = 0; i < n_; ++i) {
  fsm.row(i) = s_decompr_filt_shift.row(i) *
               decompr_fftm(fsm.row(i));
  fsm.row(i) = fs_ref_preshift.row(i) *
               st_fftm(fsm.row(i));
  }
```

VSIPL++ respects locality

Good Locality results in Good Performance

Optimized Xeon Results

Optimized Xeon			Baseline	
Function	Time	Performance	Time	Speedup
Digital Spotlight	0.908 s	3.2 GF/s	1.456 s	1.60
Fast-time filter	0.160 s	1.8 GF/s		
BW expansion	0.259 s	3.6 GF/s		
Matched filter	0.184 s	3.8 GF/s		
Interpolation				
Range loop	1.084 s	0.8 GF/s	1.095 s	1.01
2D IFFT	0.407 s	2.1 GF/s	0.405 s	1.00
Data Movement	-- --	-- --		
Overall	2.60 s		3.157 s	1.22

Cache locality optimization: 1.2 x speedup on Xeon

Results

Productivity

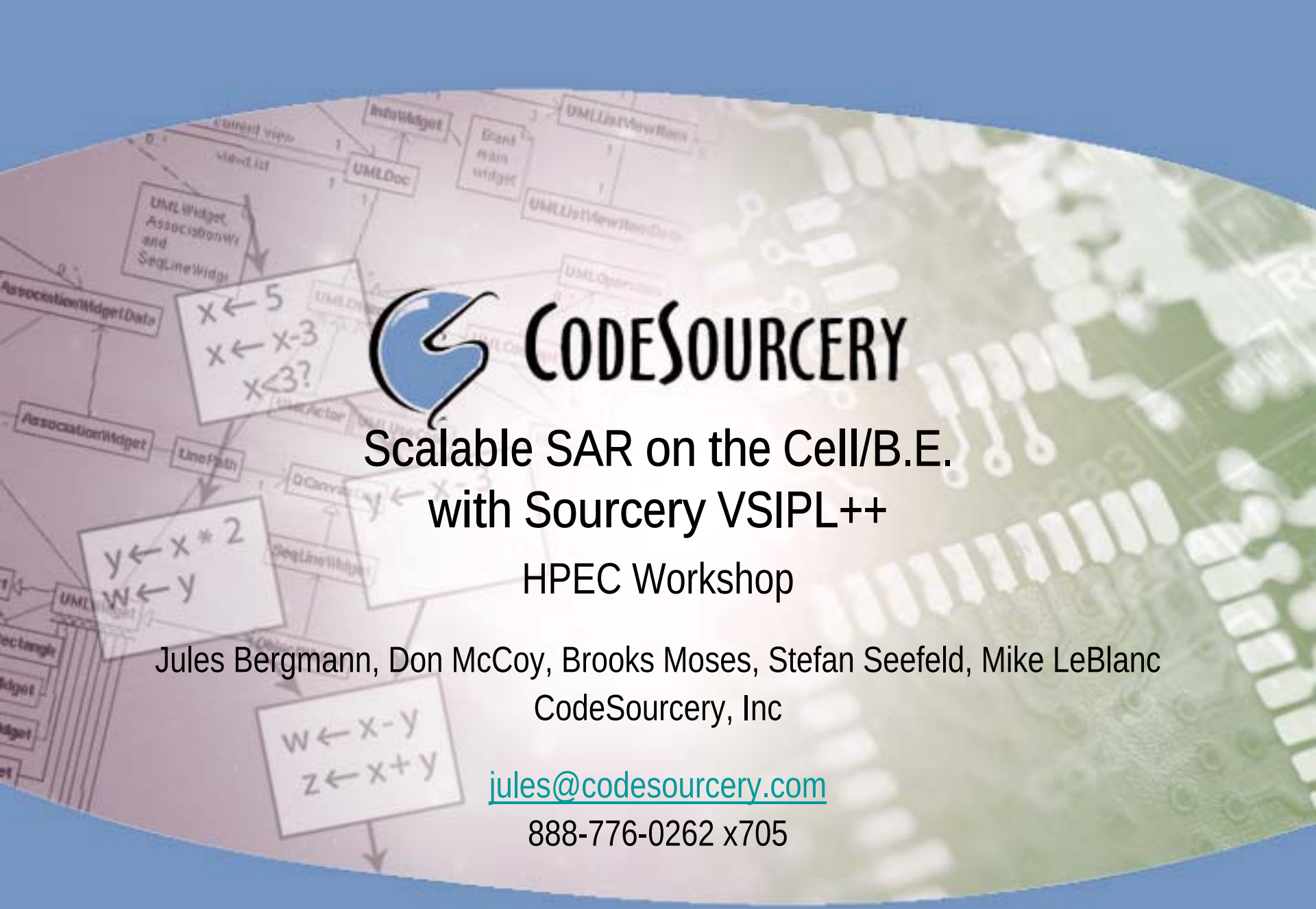
- Sourcery VSIPL++ closely matches Matlab algorithm
- Optimized Sourcery VSIPL++ easier than unoptimized C

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



CodeSourcery

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