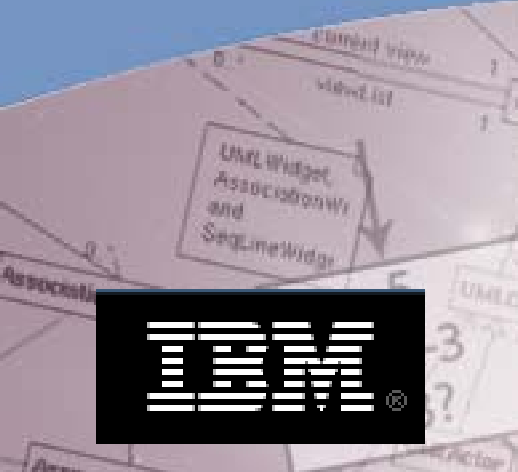
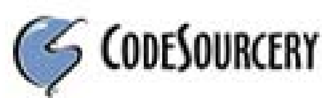


Sourcery VSIPL++ for Cell/B.E.

HPEC
Sep 20, 2007

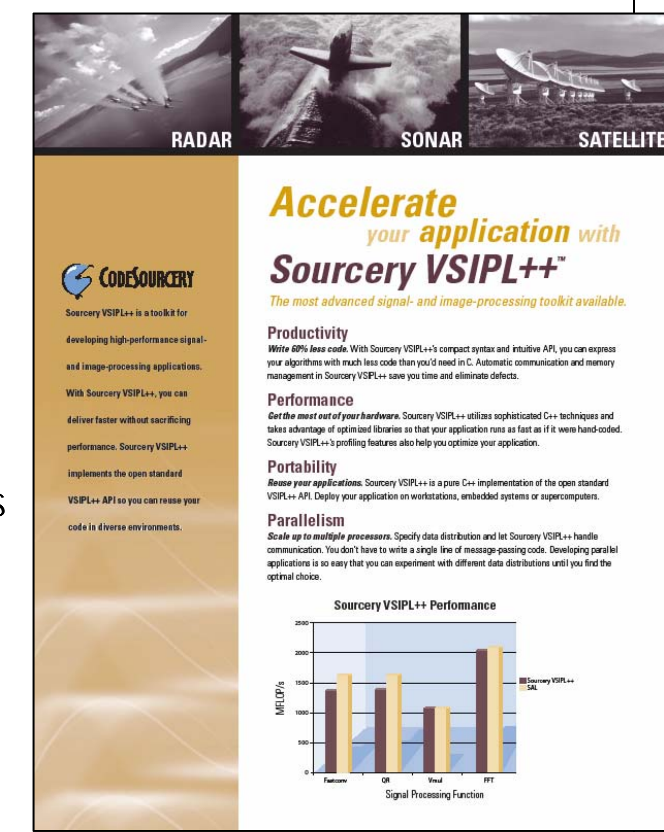
Jules Bergmann, Mark Mitchell, Don McCoy, Stefan Seefeld, Assem Salama - CodeSourcery, Inc
Fred Christensen - IBM
Rick Pancoast, Tom Steck - Lockheed Martin MS2

jules@codesourcery.com
888-776-0262 x705



Sourcery VSIPL++: Signal & Image-Processing Library

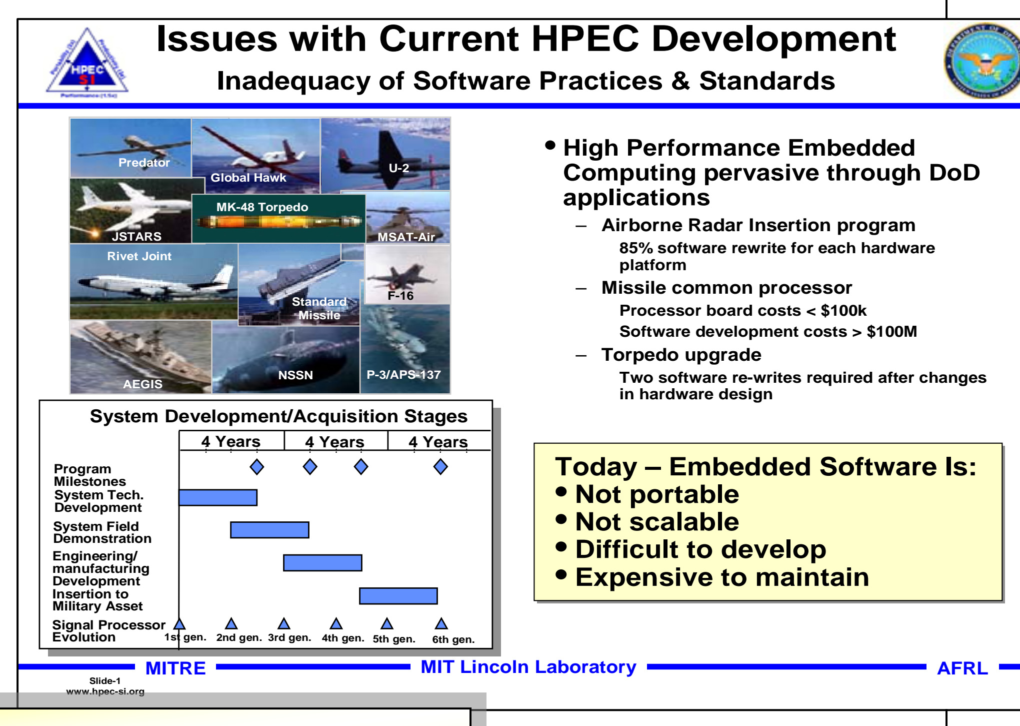
- **Comprehensive Functionality**
  - Signal-Processing: FFTs, convolutions, correlations, etc.
  - Solvers: QR, LU, Cholesky, etc.
  - Linear Algebra: matrix multiplication, Hermitians, etc.
  - Support for multi-processor computation
- **Simple C++ API**
  - No MPI programming required
  - No SPE programming required
  - No special tools required
  - Easy to port code across systems
  - Easy to compare performance across vendors/architectures
- **Performance**
  - Automatically fuses computations to run on SPEs
  - Single digit % "abstraction penalty" for simple primitives
- **Interoperability**
  - Leverages the vendor software stacks
  - Implements the open-standard VSIPL++ API



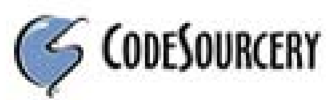
Open-Architecture API for Signal and Image Processing

DoD Motivation for VSIPL++: Faster, Better, Cheaper

- **Performance:**
  - Write fast code for particular CPUs once, then use it again and again
  - Let computers perform complex optimizations
- **Portability:**
  - Reuse code on multiple systems:
    - supercomputers
    - workstations
    - embedded systems
- **Productivity:**
  - Write new code faster
  - Repurpose existing code
  - Allow experimentation



COTS Benefits for Software



Cell / B.E. Architecture

Power Processing Element

- 64-bit general purpose RISC
- 2-way hardware multithreaded
- L1 Cache: 32KB I / 32KB D
- L2 Cache: 512KB combined
- VMX SIMD ISA
- 3.2 GHz

Synergistic Processor Elements

- SIMD Substrate
  - 128-bit wide SIMD Units
  - 128-word register file
  - 25.6 GF/s peak @ 3.2 GHz
- 256 KB Local Store
- DMA Controller

200+ GF/s Peak Performance

Cell / B.E. Programming Challenges

SPE | SPE | SPE | SPE

PPE — EIB (200+ GB/s sustained) — Mem, IO

SPE | SPE | SPE | SPE

25.6 GB/s Memory Bandwidth

20 GB/s Coherent 5 Gbps IO

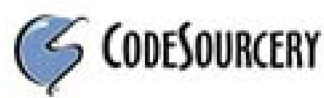
Usual Challenges

- SIMD Vectorization
- Instruction-Level Parallelism
  - Pipeline latency
  - Dual issue
- Memory Hierarchy
- Compute/IO

New Multi-core challenges

- Exploit SPE level parallelism
  - Algorithm Partitioning
- Manage explicit communication
  - Comp/Comm overlap
- Manage limited SPE memory

Complex Programming Model



Cell/B.E. SIP Application Development Models

- **Low-Level / Direct Access**
  - Write SPE and MPI code manually
  - Explicitly manage DMAs, double-buffering, etc.
  - Pros: theoretically optimal performance
  - Cons: challenging, time-consuming, not portable
  - Programming at this level is like programming in assembly language
- **Vendor Software Stack**
  - Write SPE and MPI code manually
  - Use SDK, ALF to manage DMAs and buffering
  - Pros: simpler programming model
  - Cons: not optimized for SIP, not portable
- **Sourcery VSIPL++**
  - Use high-level API to express algorithm
  - Let Sourcery VSIPL++ manage SDK, ALF, MPI, SPEs
  - Pros: simplest programming model, portable
  - Cons: may not provide maximum performance, cover all possible use cases

VSIPL++ Attributes for Multi-Core

Views / Blocks

- Separates concerns of data's logical view from its physical layout
  - Split/interleaved, dimension ordering, parallel distribution
- Initial functional development independent of subsequent optimization

Expression Templates

- Library has visibility to sequence of operations
- Greater optimization potential
- Operation Fusion – Locality

Dispatch Engine

- Flexible, low-overhead dispatch of operations to computation
- Based on run-time and compile-time attributes

VSIPL++ API and Sourcery VSIPL++ Implementation Provide Powerful Abstractions and Tools for Cell/B.E.

VSIPL++ Model for Cell/B.E.

PPE

User Application

User program runs on the PPE

SPE *N*

SPE *1*

Memory (RDRAM)

Fast Convolution

```
typedef complex<float> T;
Vector<T>         weights(size);
Matrix<T>         data(rows, size);

Fftm<T, T, row, fft_fwd>
   fwd(Domain<2>(rows, size), 1.);
Fftm<T, T, row, fft_inv>
   inv(Domain<2>(rows, size), 1./size);

fft_ip<fft_fwd>(weights);

data = inv(vmmul<row>(weights, fwd(data)));
```

VSIPL++ Model for Cell/B.E.

PPE

User Application

Sourcery VSIPL++

IBM SDK
(multi-core)

SPE *N*

SPE *1*

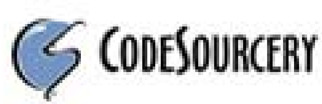
Memory (RDRAM)

Sourcery VSIPL++ manages the SPEs

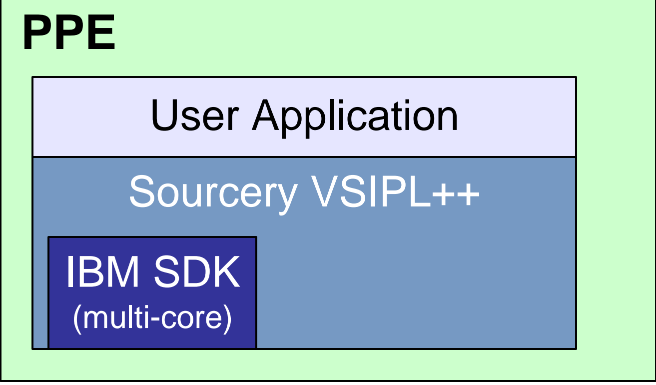
- Recognizes VSIPL++ routines suitable for SPEs
- Uses IBM SDK (ALF) to control SPEs

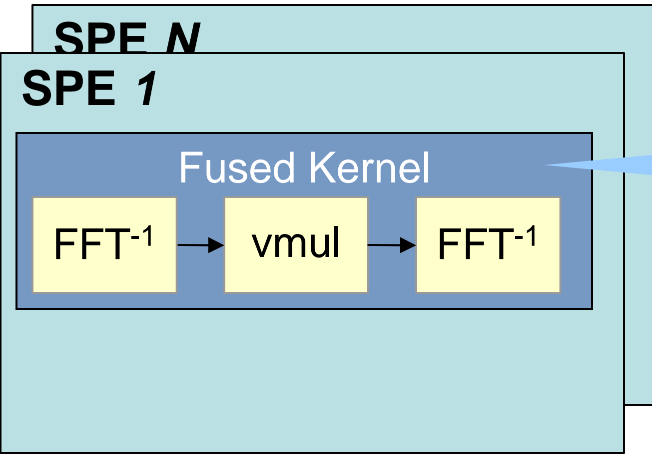
Fast Convolution

```
typedef complex<float> T;
Vector<T>        weights(size);
Matrix<T>        data(rows, size);

Fftm<T, T, row, fft_fwd>
   fwd(Domain<2>(rows, size), 1.);
Fftm<T, T, row, fft_inv>
   inv(Domain<2>(rows, size), 1./size);

fft_ip<fft_fwd>(weights);

data = inv(vmmul<row>(weights, fwd(data)));
```



VSIPL++ Model for Cell/B.E.

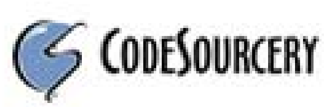




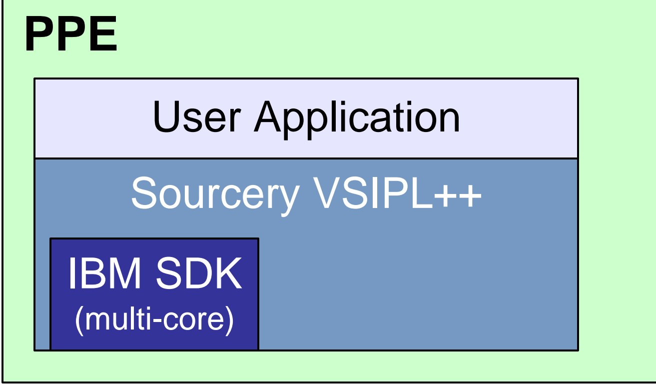
Memory (RDRAM)

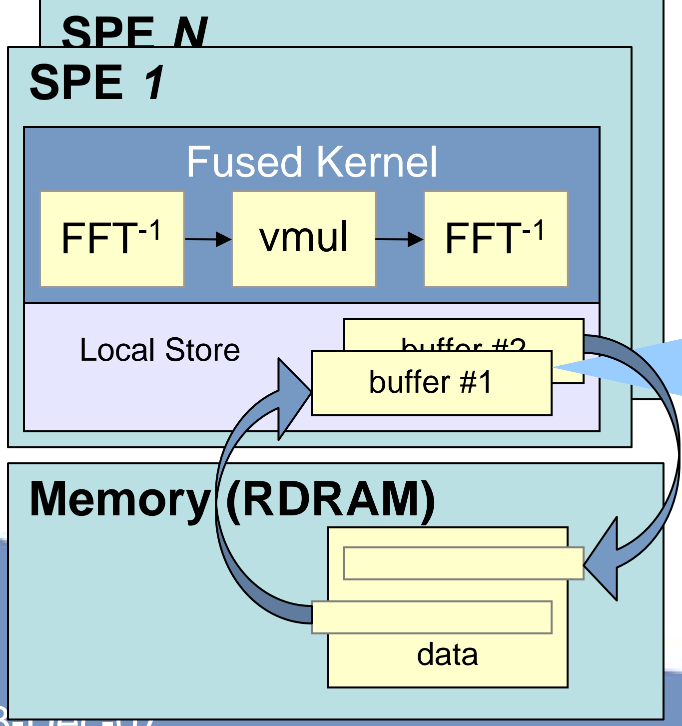
Compute kernels run on SPEs

```
data = inv(vmmul<row>(weights, fwd(data)));
```



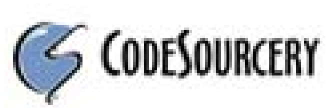
VSIPL++ Model for Cell/B.E.



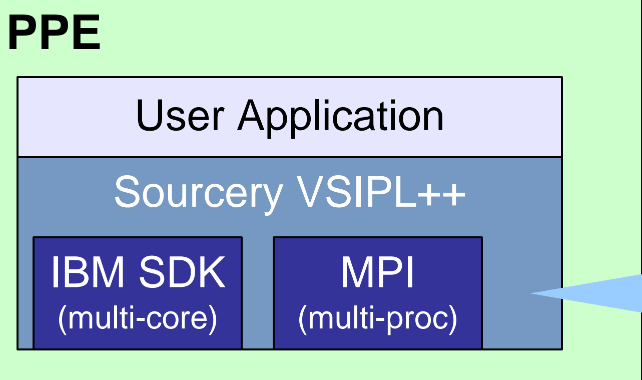


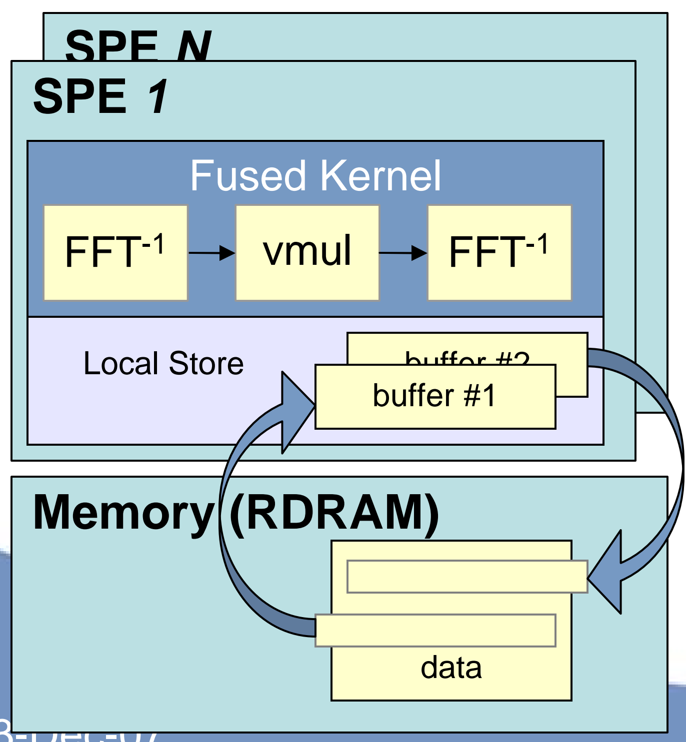
SPEs manage streaming

- DMA to/from memory
- Double buffering
- Computation/Communication overlap

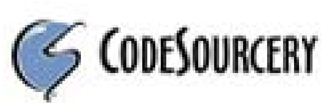


VSIPL++ Model for Cell/B.E.





Sourcery VSIPL++ can utilize manages processors



Cell/B.E. Productivity

Fast convolution:

For each pulse: out = InvFFT(weights * FwdFFT(in))

In VSIPL++, this takes 7 lines (just 1 for computation):

```
typedef complex<float> T;
Vector<T>       weights(size);
Matrix<T>       data(rows, size);
```

Allocate Data Structures

```
Fftm<T, T, row, fft_fwd>
   fwd(Domain<2>(rows, size), 1.);
Fftm<T, T, row, fft_inv>
   inv(Domain<2>(rows, size), 1./size);
```

Create FFTM Objects

```
fft_ip<fft_fwd>(weights);
```

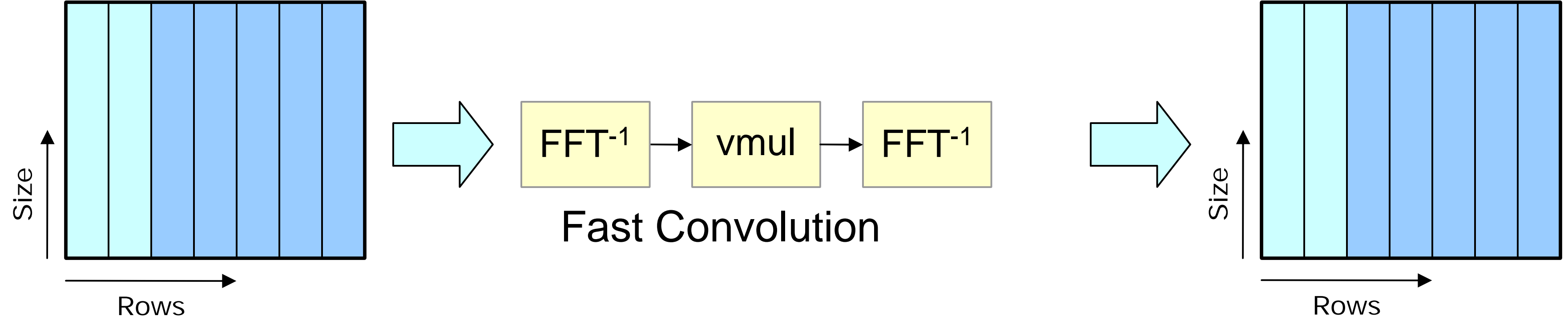
Transform Weights

```
data = inv(vmmul<row>(weights, fwd(data)));
```

Fast Convolution

No system/architecture specific statements required

Fast Convolution



Cell/B.E. Fast Convolution

PPE

SPE 1 | SPE 2 | SPE 8

Size

Rows

SPE 1

FFT^{-1} → vmul → FFT^{-1}

Fast Convolution

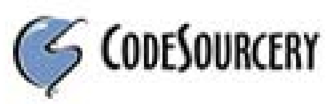
SPE 1 | SPE 2 | SPE 8

Size

Rows

Data is partitioned across SPEs

- Fused kernel runs on SPEs
- Data processed row at a time
- Double buffered DMA



Performance

VSIPL++ fast convolution *sustains* 80+ GFLOP/s (40% of SPE peak)

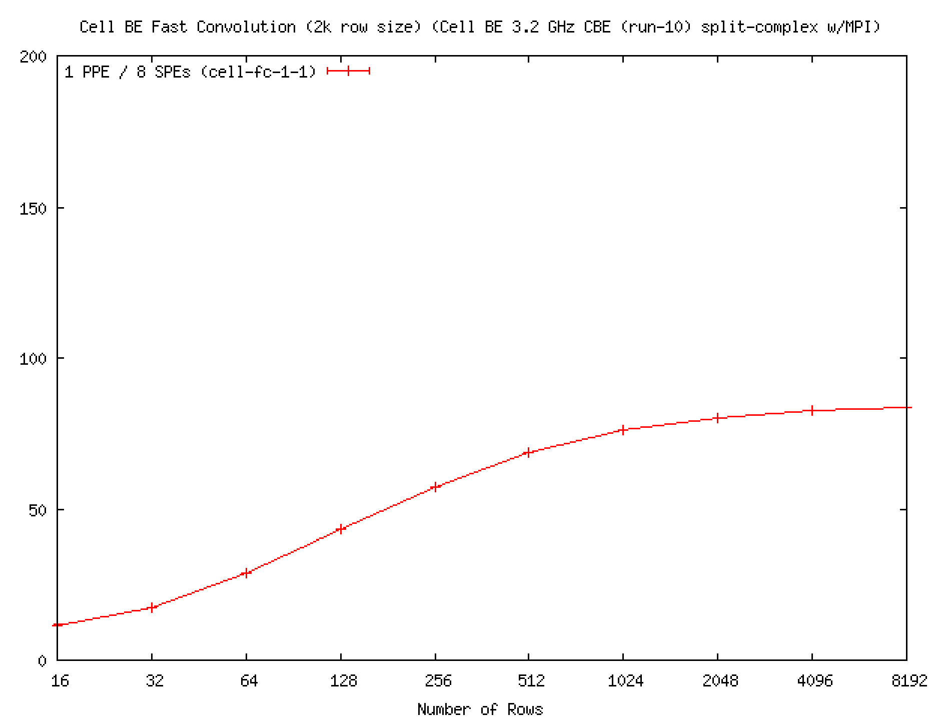
At 4096 rows of 2048 points

- 83 GFLOP/s (40% of peak)
- ~10 GB/s bandwidth

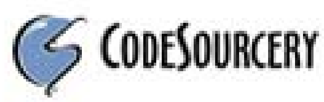
Performance Headroom

- FFT dominates computation.
- BW available: 20 GB/s demonstrated.

Memory to memory measurement

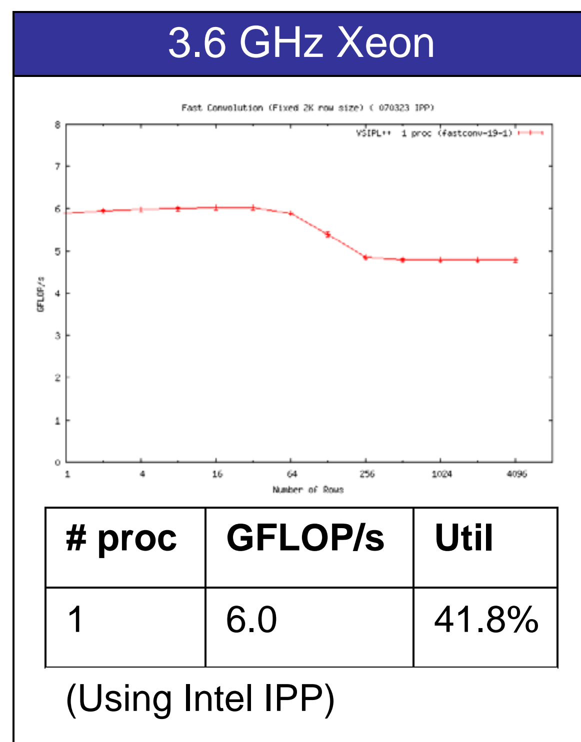


High Sustained Performance



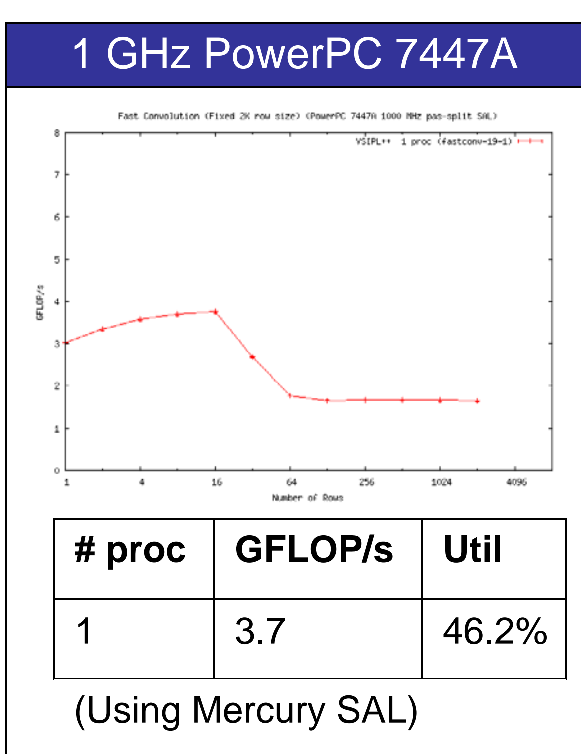
Portability

VSIPL++ fast convolution runs *unchanged* on Xeon and PowerPC



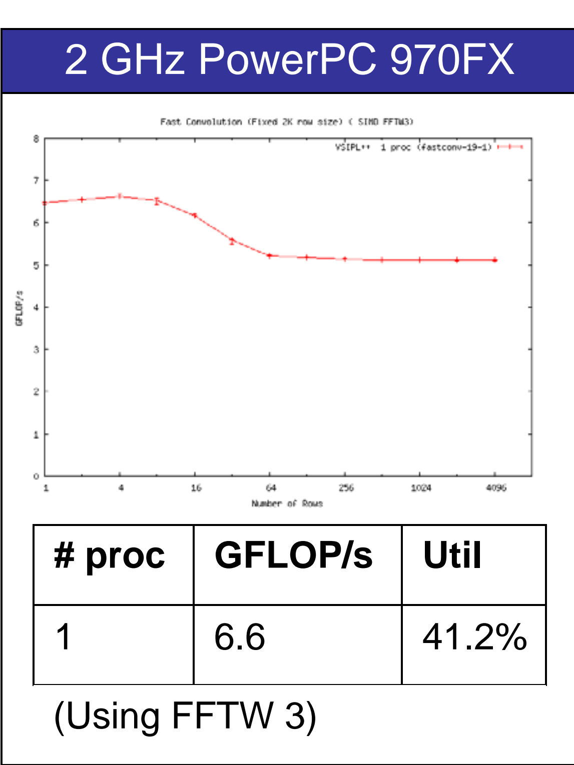
# proc	GFLOP/s	Util
1	6.0	41.8%

(Using Intel IPP)



# proc	GFLOP/s	Util
1	3.7	46.2%

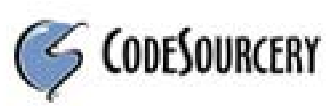
(Using Mercury SAL)



# proc	GFLOP/s	Util
1	6.6	41.2%

(Using FFTW 3)

Portable High Sustained Performance



Parallelism

Using multiple processors requires minor changes to data structures (blue):

```
typedef complex<float> T;
typedef Dense<2, T, row2_major, Map<> >          data_block_type;
typedef Dense<1, T, row1_major, Global_map<1> > weights_block_type;
Map<> map(num_processors());
Vector<T, weights_block_type> weights(size);
Matrix<T, data_block_type>    data(rows, size, map);
```

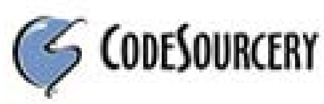
No changes to operations or computation:

```
Fftm<T, T, row, fft_fwd> fwd(Domain<2>(rows, size), 1.);
Fftm<T, T, row, fft_inv> inv(Domain<2>(rows, size), 1./size);
```

```
fft_ip<fwd_fft>(weights);
```

```
data = inv(vmmul<row>(weights, fwd(data)));
```

Expressing Data-Parallelism Straight-Forward



Parallelism

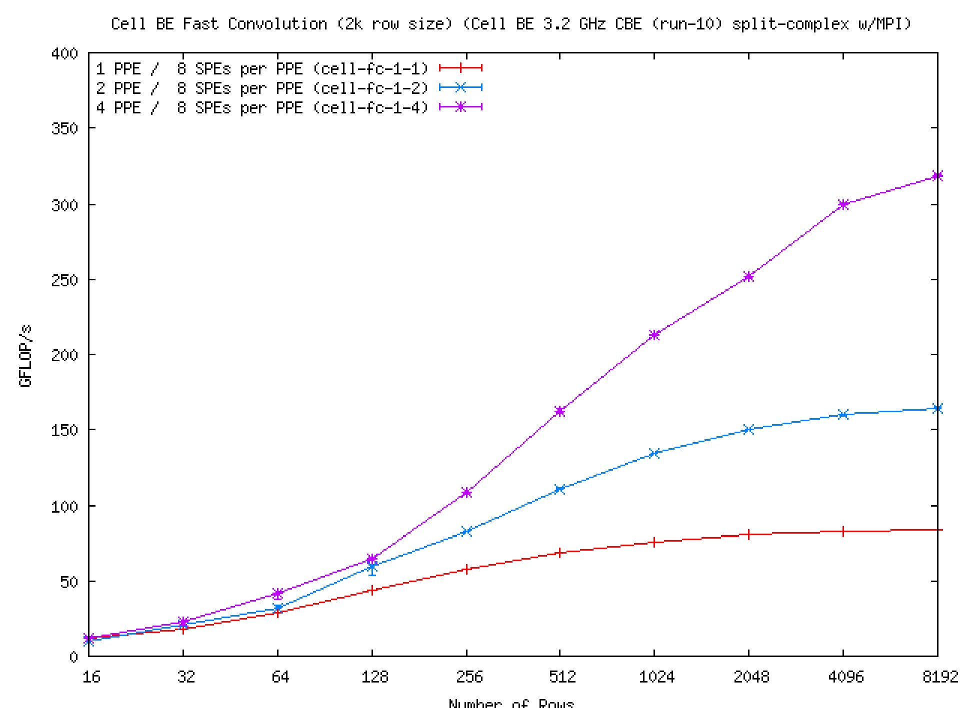
VSIPL++ fast convolution can take advantage of multiple processors

Using 4 Cell/B.E.s

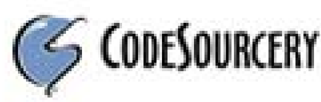
- Sustains 320 GFLOP/s

Speedup (expect linear):

- Fixed problem size: 3.6x speedup.
- Scaled problem size: 3.9x speedup.



Scalable **High Sustained Performance**



Trade-Space Exploration

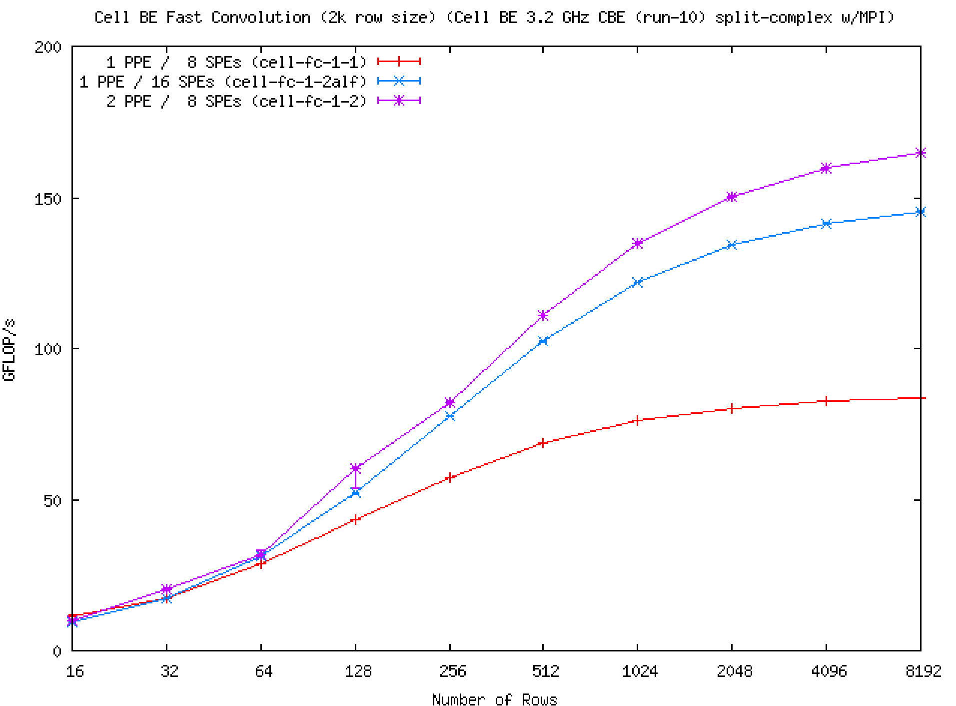
For coherently connected Cell/B.E.s, What is faster?

- 1 process - 1 PPE with 16 SPEs
- 2 processes - 2 PPEs with 8 SPEs each

Just try it!

Using 2 PPEs outperforms:

- Greater memory bandwidth
- Coherent interconnect bottleneck



Easy to Explore Implementation Trade-offs

Advantages of Sourcery VSIPL++ for Cell/B.E.

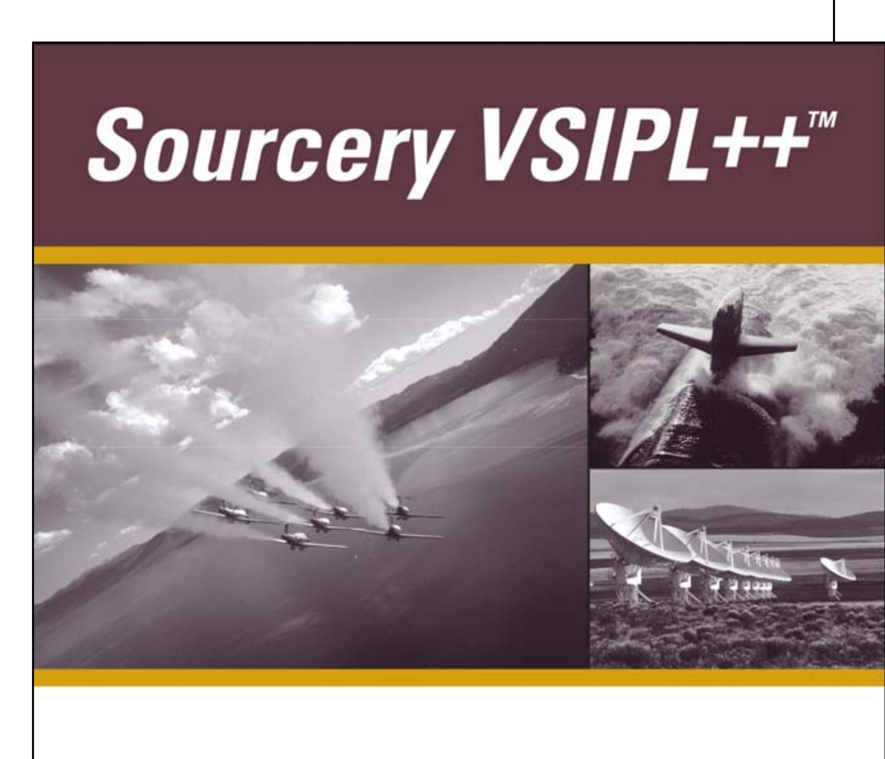
- Improves out-of-box experience
  - Code runs unchanged on Cell/B.E. with good performance
  - Programmer retains ability to tune for maximum performance
- Reduces software development costs
  - Fewer lines of code
  - Very little Cell-specific code
  - No direct SPE programming
  - Trade-space exploration
- Portability
  - Software can be easily migrated between Cell/B.E. and other systems

Performance, Productivity, Portability, Parallelism!

Availability

Sourcery VSIPL++ is available today

- 1.3 for GNU/Linux, Mercury Power and Windows systems
- Technology preview for Cell/B.E.

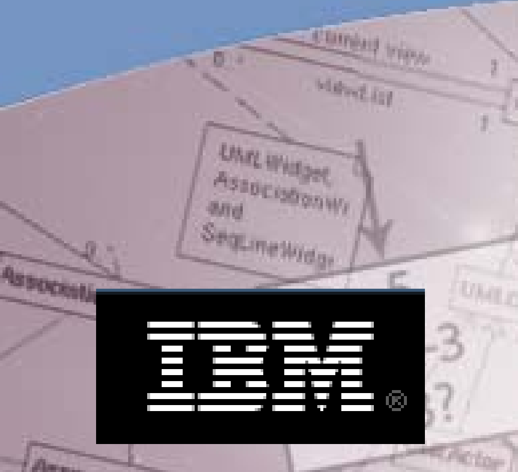


For more information and download:

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

Join our mailing list:

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



Sourcery VSIPL++ for Cell/B.E.

HPEC

Sep 20, 2007

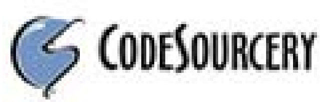
Jules Bergmann, Mark Mitchell, Don McCoy, Stefan Seefeld, Assem Salama - CodeSourcery, Inc

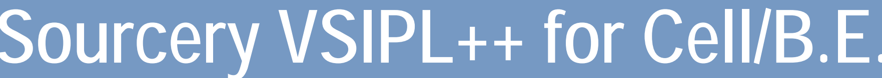
Fred Christensen - IBM

Rick Pancoast, Tom Steck - Lockheed Martin MS2

jules@codesourcery.com

888-776-0262 x705





Sourcery VSIPL++ for Cell/B.E.

Status

- IBM Teaming Agreement
  - VSIPL++ Proof of Concept (Complete): Optimize fast convolution (FFT, vector-multiply)
  - Cell Math Library
- Current Performance:
  - 1 Cell: 83 GFLOPS (~40% utilization)
  - 4 Cells (2 blades): 318 GFLOPS (~39% utilization)
- Completely Portable:
  - User needs no knowledge of Cell/B.E. (SPEs, etc.)
  - Porting from another system is just recompilation

Model

- Users program the PPE
  - User code does not directly run on SPEs, do DMAs, etc.
- Sourcery VSIPL++ manages the SPEs
  - Streaming kernel accelerator
  - Translates VSIPL++ API calls into SPE routines
  - Manages DMAs, double-buffering, etc.
- Sourcery VSIPL++ manages multi-processors
  - Uses MPI to communicate data between processors
- Leverages IBM Software Stack

Sourcery VSIPL++ delivers the performance of Cell/B.E. in a simple, portable, high-level API.

Productivity

Compute BLAS zherk:

$$C \leftarrow \alpha A \, conjug(A)^t + \beta C$$

VSIPL

```
A = vsip_cmcreate_d
     (10,15, VSIP_ROW,MEM_NONE);
C = vsip_cmcreate_d
     (10,10, VSIP_ROW,MEM_NONE);
tmp = vsip_cmcreate_d
     (10,10, VSIP_ROW,MEM_NONE);
vsip_cmprodh_d(A,A,tmp);
vsip_rscmmul_d(alpha,tmp,tmp);
vsip_rscmmul_d(beta,C,C);
vsip_cmadd_d(tmp,C,C);
vsip_cblockdestroy(
  vsip_cmdestroy_d(tmp));
vsip_cblockdestroy(
  vsip_cmdestroy_d(C));
vsip_cblockdestroy(
  vsip_cmdestroy_d(A));
```

Sourcery VSIPL++

```
Matrix<complex<double> > A(10,15);
Matrix<complex<double> > C(10,10);
C = alpha * prodh(A,A) + beta * C;
```

Advantages

- ✓ 70% fewer lines of code
- ✓ No explicit memory management
- ✓ Better optimization opportunities

Productivity

Vector Threshold

$$Z \leftarrow (A > B)\ ?\ A : 0$$

SAL

```
float* A[size];
float* B[size];
float* Z[size];

lvgtx(A, 1, B, 1, Z, 1, size, 0);
vmulx(Z, 1, A, 1, Z, 1, size, 0);
```

Sourcery VSIPL++

```
Vector<float> A(size);
Vector<float> B(size);
Vector<float> C(size);

C = ite(A > B, A, 0.0);
```

Advantages

- ✓ Not limited to API
- ✓ Fewer lines of code
- ✓ Better performance
  - Better cache locality

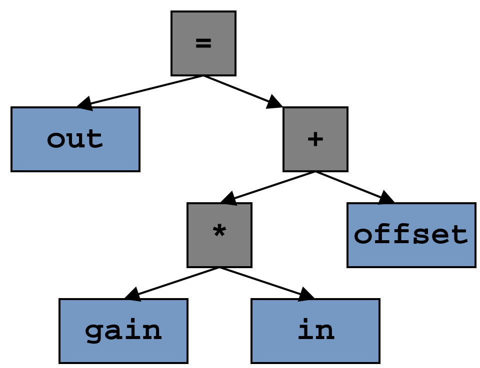
Performance

Fused multiply-add (aka non-uniformity correction):

```
out = gain * img + offset;
```

Expression Templates

- Represent expression as parse tree



- Library can examine, manipulate, evaluate parse tree at compile-time

Dispatch Engine

- Determine best way to evaluate expression

Operation Fusion

- Fuse multiple operations into single loop:

```
for (i=0; i<rows*cols; ++i)
  out[i] = gain[i]*img[i] + offset[i];
```

- Possibly using AltiVec:

```
for (i=0; i<rows*cols; ++i)
  out = vec_madd(gain, img, offset);
  out+=4; gain+=4; img+=4; offset+=4;
```

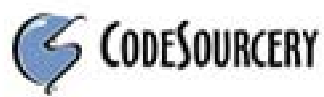
Math Library Interface

- Fuse operations into vendor library call(s):

```
vma(gain,1,in,1,offset,1,out,1,size);
```

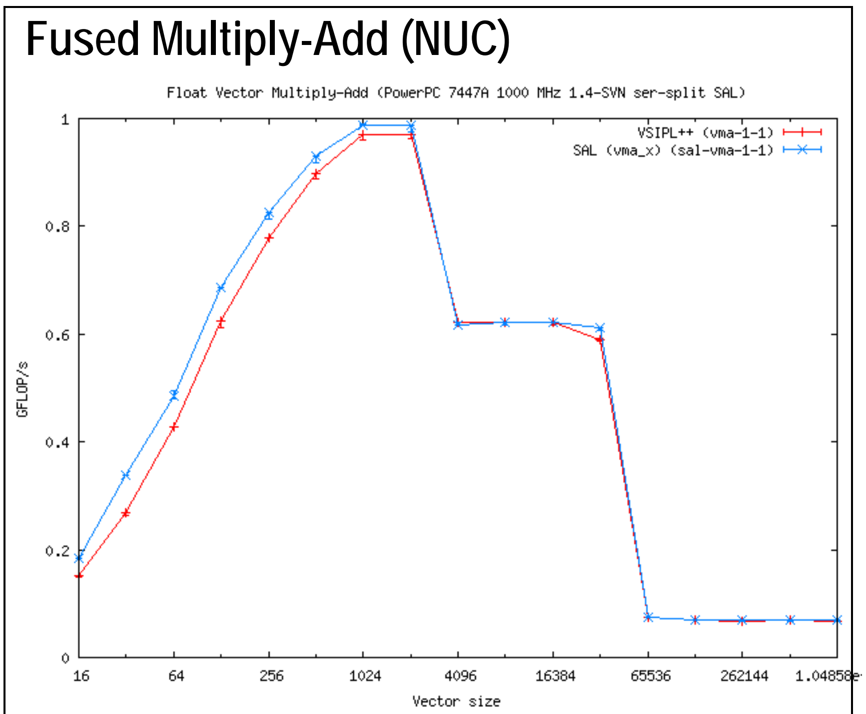
- Single digit overheads ~2%

Sophisticated Implementation Techniques for High-Performance



Performance

Fused Multiply-Add (NUC)

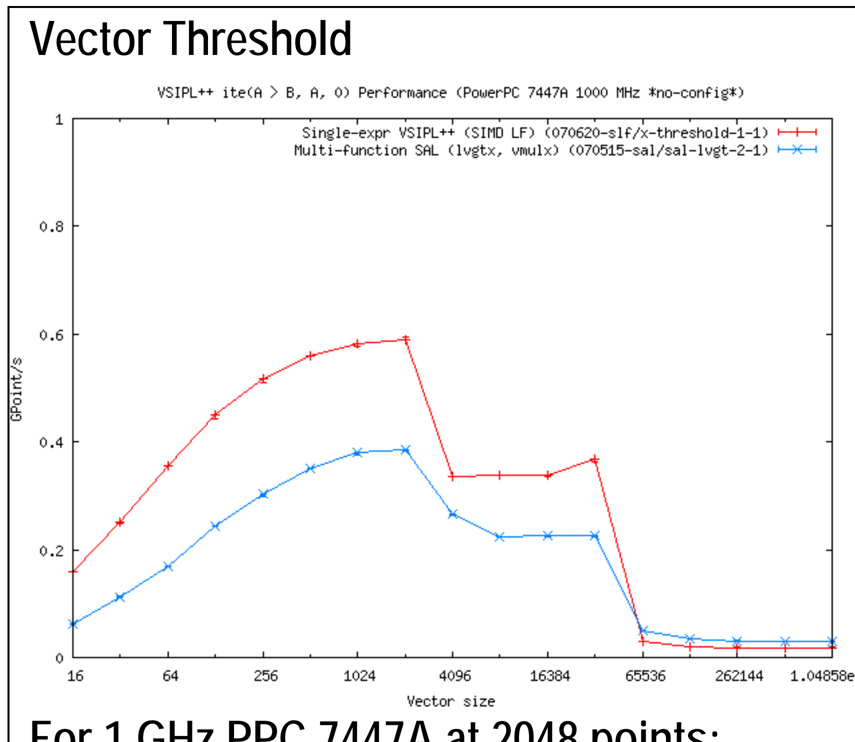


For 1 GHz PPC 7447A at 2048 points:

- VSIPL++ (red) 0.971 GFLOP/s
- Vendor (blue) 0.986 GFLOP/s

VSIPL++: 1.5% overhead

Vector Threshold

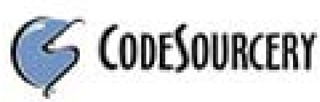


For 1 GHz PPC 7447A at 2048 points:

- VSIPL++ (red) 0.591 GPt/s
- Vendor (blue) 0.385 GPt/s

VSIPL++: 53% improvement w/fused Ops

Vendor Library Performance or Better



Portability

C++ API

- Developers use existing compilers, debuggers, etc.
- No special tools required
- No new programming languages to learn

CPUs

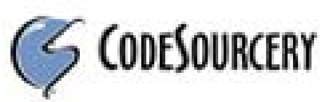
- IA32, EM64T, AMD64
- Power
- Cell/B.E.
- SPARC

Compilers

- Sourcery G++
- GNU
- Green Hills
- Intel

Advantages

✓ Compare multiple platforms
✓ Develop where convenient
✓ Deploy in multiple environments



Parallelism

Sourcery VSIPL++

- Simple Model
  - User specifies data distribution
  - VSIPL++ manages data movement
- Serial/Parallel Portability
  - Same algorithms run in serial and in parallel
  - Specify data distributions …
  - … recompile …
  - … run!

Advantages

✓ No MPI, PAS, etc. code required

✓ Same code runs on:
- Multiprocessor workstations
- GNU/Linux clusters
- Embedded multiprocessors

✓ Experimenting with data distributions is easy