“Object oriented technology reduces software cost.”

“Fully utilizing HPEC systems for SIP applications requires managing operations at the lowest possible level.”

“There is great concern that these two approaches may be fundamentally at odds.”
“Parallel Performance Vision”

- SIP Program Code
- VSIPL++ API
- SIMD Support
- Multiprocessor Support

“Drastically reduce the performance penalties associated with deploying object-oriented software on high performance parallel embedded systems.”

“Automated to reduce implementation cost.”
Advantages of VSIPL

- **Portability**
  - Code can be reused on any system for which a VSIPL implementation is available.

- **Performance**
  - Vendor-optimized implementations perform better than most handwritten code.

- **Productivity**
  - Reduces SLOC count.
  - Code is easier to read.
  - Skills learned on one project are applicable to others.
  - Eliminates use of assembly code.
Limitations of VSIPL

- Uses C Programming Language
  - “Modern object oriented languages (e.g., C++) have consistently reduced the development time of software projects.”
  - Manual memory management.
  - Cumbersome syntax.

- Inflexible
  - Abstractions prevent users from adding new high-performance functionality.
  - No provisions for loop fusion.
  - No way to avoid unnecessary block copies.

- Not Scalable
  - No support for MPI or threads.
  - SIMD support must be entirely coded by vendor; user cannot take advantage of SIMD directly.
Parallelism: Current Practice

MPI used for communication, but:

- MPI code often a significant fraction of total program code.
- MPI code notoriously hard to debug.
- Tendency to hard-code number of processors, data sizes, etc.
- Reduces portability!

Conclusion: users should specify only data layout.
Atop VSIPL’s Foundation

VSIPL

- Serial
- Limited Extensibility
- C Programming Language
- Optimized Vendor Implementations: High Performance
- Open Standard: Specification, Reference Implementation

VSIPL++

- Scalable Multiprocessor Computation
- Extensible: Operators, data formats, etc.
- C++: OOP, memory management
Leverage VSIPL Model

Same terminology:
- Blocks store data.
- Views provide access to data.
- Etc.

Same basic functionality:
- Element-wise operations.
- Signal processing.
- Linear algebra.
Serial Specification: Version 1.0a
- Support for all functionality of VSIPL.
- Flexible block abstraction permits varying data storage formats.
- Specification permits loop fusion, efficient use of storage.
- Automated memory management.

Reference Implementation: Version 0.95
- Support for functionality in the specification.
- Used in several demo programs — see next talks.
- Built atop VSIPL reference implementation for maximum portability.

Parallel Specification: Version 0.5
- High-level design complete.
Input:
- Noisy signal arriving at a row of uniformly distributed sensors.

Output:
- Bearing and frequency of signal sources.
SIP Primitives Used

**Computation:**
- FIR filters
- Element-wise operations (e.g., magsq)
- FFTs
- Minimum/average values

**Communication:**
- Corner-turn
  - All-to-all communication
- Minimum/average values
  - Gather
Computation

1. Filter signal to remove high-frequency noise.  (FIR)

2. Remove side-lobes resulting from discretization of data.  (mult)

3. Apply Fourier transform in time domain.  (FFT)

4. Apply Fourier transform in space domain.  (FFT)

5. Compute power spectra.  (mult, magsq)
Diagram of the Kernel

- **Input**
  -one row per sensor

- **FIR**

- **Weights**
  -row-wise FFT
  -removes side lobes

- **magsq, *1/n**

- **Column-wise FFT**
  -“corner turn” from sensor domain to time domain

- **Optional corner turn**

Add this matrix to the sum.
VSIPL Kernel

Seven statements required:

```c
for (i = n; i > 0; --i) {
    filtered = filter (firs, signal);
    vsip_mmul_f (weights, filtered, filtered);
    vsip_rcfftmpop_f (space_fft, filtered,
                     fft_output);
    vsip_ccfftmpi_f (time_fft, fft_output);
    vsip_mcmagsq_f (fft_output, power);
    vsip_ssmul_f (1.0 / n, power);
    vsip_madd_f (power, spectra, spectra);
}
```
One statement required:

```cpp
for (i = n; i > 0; --i)
    spectra += 1/n *
    magsq (time_fft (space_fft (weights *
                        filter (firs,
                        signal))));
```

No changes are required for distributed operation.
Distribution in User Code

Serial case:

\[
\text{Matrix}<\text{float}_t, \text{Dense}<2, \text{float}_t> \rightarrow \\
\text{signal}_\text{matrix};
\]

Parallel case:

\[
\text{typedef Dense}<2, \text{float}_t> \text{ subblock}; \\
\text{typedef Distributed}<2, \text{float}_t, \text{subblock}, \text{ROW}> \\
\text{Block2R}_t; \\
\text{Matrix}<\text{float}_t, \text{Block2R}_t> \text{ signal}_\text{matrix};
\]

User writes no MPI code.
VSIPL++ Implementation

**Added** DistributedBlock:
- Uses a “standard” VSIPL++ block on each processor.
- Uses MPI routines for communication when performing block assignment.

**Added specializations:**
- FFT, FIR, etc. modified to handle DistributedBlock.
Performance Measurement

Test system:
- AFRL HPC system
- 2.2GHz Pentium 4 cluster

Measured only main loop
- No input/output

Used Pentium Timestamp Counter

MPI All-to-all not included in timings
- Accounts for 10-25%
### VSIPL++ Performance

| Problem Size | VSIPL | VSIPL++ | || VSIPL++ (1) | || VSIPL++ (2) | || VSIPL++ (4) | || VSIPL++ (8) |
|--------------|-------|---------|--------------|--------------|--------------|--------------|--------------|
| 256x512      | 3.3   | 3.5     | 3.6          | 1.9          | 0.9          | 0.4          |
| 512x1024     | 15    | 15      | 15           | 9            | 4            | 2            |
| 1024x2048    | 64    | 66      | 63           | 37           | 19           | 9            |
| 2048x4096    | 306   | 314     | 277          | 165          | 78           | 40           |

**Note:**
- **VSIPL:** A baseline benchmark.
- **VSIPL++:** The performance of the optimized VSIPL++ implementation.
- **|| VSIPL++ (1):** Performance with a specific configuration.
- **|| VSIPL++ (2):** Performance with another specific configuration.
- **|| VSIPL++ (4):** Performance with yet another specific configuration.
- **|| VSIPL++ (8):** Performance with the highest specific configuration.
Parallel Speedup

Overhead Compared to Linear Speedup

-15% -10% -5% 0% 5% 10% 15%

2 processes
4 processes
8 processes
1 process

VSIPL, VSIPL++

Minimal parallel overhead.
10% slower than perfect linear speedup
10% faster than perfect linear speedup

Corner turn improves execution.

Problem Size

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>VSIPL</th>
<th>VSIPL++</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x512</td>
<td>0%</td>
<td>1%</td>
</tr>
<tr>
<td>512x1024</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>1024x2048</td>
<td>0%</td>
<td>-6%</td>
</tr>
<tr>
<td>2048x4096</td>
<td>0%</td>
<td>-14%</td>
</tr>
<tr>
<td>8 processes</td>
<td>6%</td>
<td>14%</td>
</tr>
<tr>
<td>1 process</td>
<td>0%</td>
<td>6%</td>
</tr>
<tr>
<td>VSIPL++, (1)</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>VSIPL++, (2)</td>
<td>0%</td>
<td>-6%</td>
</tr>
<tr>
<td>VSIPL++, (4)</td>
<td>0%</td>
<td>5%</td>
</tr>
<tr>
<td>VSIPL++, (8)</td>
<td>0%</td>
<td>-8%</td>
</tr>
</tbody>
</table>

VSIPL, VSIPL++

2 processes
4 processes
8 processes
1 process

Perfect linear speedup
Conclusions

- VSIPL++ imposes no overhead:
  - VSIPL++ performance nearly identical to VSIPL performance.

- VSIPL++ achieves near-linear parallel speedup:
  - No tuning of MPI, VSIPL++, or application code.

- Absolute performance limited by VSIPL implementation, MPI implementation, compiler.
Visit the HPEC-SI website
http://www.hpec-si.org
for VSIPL++ specifications
for VSIPL++ reference implementation
to participate in VSIPL++ development