Portability

- Operating System and Architecture Independence
  - Solaris - Ultra Sparc
  - Linux - PPC/Pentium
  - VxWorks - PPC-Altivec
  - MAC OS X - PPC-Altivec
  - Windows - Pentium

- If the machine supports a C++ compiler....
Reusability

- First signal processing application -- 6 months development
- Second (comparable) application -- 6 weeks development
  - Original development of second application in ADA - minimum of 6 months.
Rapid/Stable Applications Development

Powerful Expressibility:

\[
D = C / 2.0 - A + B * A;
\]

\[
A = B ^ 3.5;
\]

\[
A = B.\text{abs}();
\]

\[
A = B.\text{sin}();
\]

\[
A = B.\text{var}();
\]

\[
A = B.\text{fft}().\text{ffts}hift().\text{abs}(); \quad // \ A = \text{abs}(\text{ffts}hift(\text{fft}(B)));
\]

\[
A = (B.\text{fft}() * (C.\text{fft}().\text{conj}())).\text{ifft}();
\]

\[
A = B.\text{xcorr}(C);
\]

Memory Management:

most dynamic memory usage, including VSIPL, is transparent
Shared Objects

- Signal processing objects are generated once and are automatically shared:
  - FFT coefficients
  - Window Functions
  - Filters
  - Tuners
Standard Template Library

- The C++ STL containers provide an efficient means to organize, access, and process information/data.

- Most modern large-scale C++ signal processing development efforts will use the STL extensively.

- The math API interface can be made to mirror STL operation to provide a more intuitive use of basic API operations.
Extensibility

Inherit most functionality, modify some, add other functions

- Applications
- Extended C++ Math API
- C++ Math API
- C VSIPL Interface
- VSIPL library

This could be Company Proprietary

This could be unlimited distribution
VSIPL Transparency .... Memory Management

- VSIPL codelet to perform $A = B \times C$ for vectors of 512 samples

```c
float_complex A[512], B[512], C[512];
    // data must be placed in B and C
vsip_cblock_f *Ab, *Bb, *Cb;
vsip_cvview_f *Av, *Bv, *Cv;
    // must bind user data to blocks
Ab = vsip_cblockbind_f(A, 0, 512, 0);    // omitting error checking
Bb = vsip_cblockbind_f(B, 0, 512, 0);
Cb = vsip_cblockbind_f(C, 0, 512, 0);
    // must create view to blocks
Av = vsip_cvbind_f(Ab, 0, 1, 512);    // omitting error checking
Bv = vsip_cvbind_f(Bb, 0, 1, 512);
Cv = vsip_cvbind_f(Cb, 0, 1, 512);
```
VSIPL Transparency .... Memory Management (cont)

- VSIPL codelet to perform $A = B \times C$ for vectors of 512 samples (cont)

  // must admit blocks to VSIPL memory space
  vsip_cblockadmit_f(Ab, 0); // omitting error checking
  vsip_cblockadmit_f(Bb, 0);
  vsip_cblockadmit_f(Cb, 0);

  // finally, we get to the multiply
  vsip_vmul_f(Bv, Cv, Av);

  // must destroy blocks, views, etc:
  vsip_cvalldestroy_f(Av);
  vsip_cvalldestroy_f(Bv);
  vsip_cvalldestroy_f(Cv);
VSIPL Transparency .... Memory Management (cont)

- VSIPL codelet to perform $A = B \times C$ for vectors of 512 samples (cont)
  
  Clearly, VSIPL $\neq$ VSIMPLE
  Clearly, direct VSIPL coding is prone to memory leaks/errors

- The “CVector” equivalent code is:
  
  CVector A(512), B(512), C(512);
  
  // something puts data in B and C
  
  A = B * C;
### Performance (complex data)

<table>
<thead>
<tr>
<th>Function</th>
<th>Size</th>
<th>Application</th>
<th>VSIPL kernel</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = B \times C$</td>
<td>256</td>
<td>13.4 usec</td>
<td>5.3 usec</td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>38</td>
<td>18</td>
<td>47%</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>59</td>
<td>32</td>
<td>54%</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>351</td>
<td>205</td>
<td>58%</td>
</tr>
<tr>
<td>$A = B.fft()$</td>
<td>256</td>
<td>81.5</td>
<td>67.7</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>192</td>
<td>163</td>
<td>85%</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>426</td>
<td>383</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>1949</td>
<td>1746</td>
<td>90%</td>
</tr>
</tbody>
</table>
Performance (cont)

<table>
<thead>
<tr>
<th>Function</th>
<th>Size</th>
<th>Application</th>
<th>VSIPL kernel</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B.fir(F)</td>
<td>256</td>
<td>524 usec</td>
<td>498 usec</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>1011</td>
<td>974</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>2040</td>
<td>1972</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>9048</td>
<td>8702</td>
<td>97%</td>
</tr>
</tbody>
</table>

| A = B.xcorr(C) | 256 | 306 | 230 | 75% |
| 512 | 616 | 486 | 79% |
| 1024 | 1529 | 1228 | 80% |
| 4096 | 7643 | 6090 | 80% |
The API can provide:

- Development and Performance modes of operation
- An ability to view detailed state information for math objects
- Easy mobility of object data to/from objects and the file system
- Profiling of the application software to facilitate performance tuning
- Exception handling interface