Implementation of a Shipboard Ballistic Missile Defense Processing Application Using the High Performance Embedded Computing Software Initiative (HPEC-SI) API

Jane Kent
Joseph Cook
Rick Pancoast
Nathan Doss
Jordan Lusterman

Lockheed Martin
Maritime Systems & Sensors (MS2)

HPEC 2004
30 Sep 2004
Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion
Overview

HPEC Software Initiative (HPEC-SI) Goals

- Develop software technologies for embedded parallel systems to address:
  - Portability
  - Productivity
  - Performance
- Deliver quantifiable benefits

Current HPEC-SI Focus

- Development of the VSIPL++ and Parallel VSIPL++ Standards
- VSIPL++
  - A C++ API based on concepts from VSIPL (an existing, industry accepted standard for signal processing)
  - VSIPL++ allows us to take advantage of useful C++ features
- Parallel VSIPL++ is an extension to VSIPL++ for multi-processor execution

VSIPL++ Development Process

- Development of the VSIPL++ Reference Specification
- Creation of a reference implementation of VSIPL++
- Creation of demo applications
Lockheed Martin
Demonstration Goals

- Use CodeSourcery’s VSIPL++ reference implementation in a main-stream DoD Digital Signal Processor Application

- Utilize existing “real-world” tactical application Synthetic WideBand (SWB) Radar Mode. The original code was developed for the United States Navy and MDA under contract for improved S-Band Discrimination. SWB is continuing to be evolved by MDA for Aegis BMD signal processor.

- Identify areas for improved or expanded functionality and usability

**Milestone 1**
Successfully build VSIPL++ API
Unix, Linux, Mercury

**Milestone 2**
Convert SWB Application to use VSIPL++ API
Unix, Linux

**Milestone 3**
Port SWB Application to embedded platforms
Mercury, Sky

**Milestone 4**
Application analysis Feedback & recommendations

COMPLETE
During development, there was a continuous loop of change requests/feedback, and API updates and patches.
Outline

- Overview
- **Lockheed Martin Background and Experience**
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion
Lockheed Martin Software
Risk Reduction Issues

- General mission system requirements
  - Maximum use of COTS equipment, software and commercial standards
  - Support high degree of software portability and vendor interoperability

- Software Risk Issues
  - Real-time operation
    - Latency
    - Bandwidth
    - Throughput
  - Portability and re-use
    - Across architectures
    - Across vendors
    - With vendor upgrades
  - Real-time signal processor control
    - System initialization
    - Fault detection and isolation
    - Redundancy and reconfiguration
  - Scalability to full tactical signal processor
Lockheed Martin Software
Risk Reduction Efforts

- Benchmarks on vendor systems (CSPI, Mercury, HP, Cray, Sky, etc.)
  - Communication latency/throughput
  - Signal processing functions (e.g., FFTs)
  - Applications

- Use of and monitoring of industry standards
  - Communication standards: MPI, MPI-2, MPI/RT, Data Re-org, CORBA
  - Signal processing standards: VSIPL, VSIPL++

- Technology refresh experience with operating system, network, and processor upgrades (e.g., CSPI, SKY, Mercury)

- Experience with VSIPL
  - Participation in standardization effort
  - Implementation experience
    - Porting of VSIPL reference implementation to embedded systems
    - C++ wrappers
  - Application modes developed
    - Programmable Energy Search
    - Programmable Energy Track
    - Cancellation
    - Moving Target Indicator
    - Pulse Doppler
    - Synthetic Wideband
Lockheed Martin Math Library Experience

- **Vendor supplied math libraries**
  - **Advantages**
    - Performance
  - **Disadvantages**
    - Proprietary Interface
    - Portability

- **LM Proprietary C Wrappers**
  - **Advantages**
    - Performance
  - **Disadvantages**
    - Proprietary Interface

- **VSIPL**
  - **Advantages**
    - Performance
    - Portability
    - Standard interface
  - **Disadvantages**
    - Verbose interface (higher % of management SLOCS)

- **LM Proprietary C++ Library**
  - **Advantages**
    - Performance
    - Portability
    - Productivity
  - **Disadvantages**
    - Proprietary Interface
    - Partial implementation (didn’t wrap everything)

- **VSIPL++**
  - **Advantages**
    - Standard interface
    - Performance
    - Portability
    - Productivity
  - **To Be Determined**

- **Vendor supplied math libraries wrapped with #ifdef’s**
  - **Advantages**
    - Performance
  - **Disadvantages**
    - Proprietary Interface
    - Portability

- **Thin VSIPL-like C++ wrapper**
  - **Advantages**
    - Performance
    - Portability
    - Productivity (fewer SLOCS, better error handling)
  - **Disadvantages**
    - Proprietary interface
    - Partial implementation (didn’t wrap everything)
Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion
Application Overview

The Lockheed Martin team took existing Synthetic Wideband application, developed and targeted for Aegis BMD signal processor implementation, and rewrote it to use and take advantage of the VSIPL++

The SWB Application achieves a high bandwidth resolution using narrow bandwidth equipment, for the purposes of extracting target discriminant information from the processed range doppler image

Synthetic Wideband was chosen because:
- It exercises a number of algorithms and operations commonly used in our embedded signal processing applications
- Its scope is small enough to finish the task completely, yet provide meaningful feedback in a timely manner
- Main-stream DoD application
Application Overview — Synthetic WideBand Processing

By using “Stepped” medium band pulses, and specialized algorithms, an effective “synthetic” wide band measurement can be obtained.

1. Transmit and Receive Mediumband Pulses
2. Pulse Compress Mediumband Pulses
3. Coherently Combine Mediumband Pulses to Obtain Synthetic Wideband Response

- Requires accurate knowledge of target motion over waveform duration
- Requires phase calibration as a function of mediumband pulse center frequency
Application Interface

- SWB Application
  - Calibration Data
  - Hardware Mapping Information (How application is mapped to processors)
  - Algorithm Control Parameters
  - Control & Radar Data
  - Processing Results:
    - Images
    - Features
Processing Flow

PRI Processing (Repeated n times/CPI)

TrackWindow Processing
- Doppler Compensation
- Pulse Compression

SubWindow(1)
- Interpolation
- Range Walk Compensation
- Synthetic Up Mixing

SubWindow(2)
- Interpolation
- Range Walk Compensation
- Synthetic Up Mixing

SubWindow(x)
- Interpolation
- Range Walk Compensation
- Synthetic Up Mixing

CPI Processing
- Coherent Integration
- Coherent Integration
- Coherent Integration

Output
- Range Doppler Image
- Range Doppler Image
- Range Doppler Image

Industry Standards: MPI, VSI PL++

Input
- Radar & Control Data,
- Alg. Control Params,
- Cal. Data,
- Mapping

Block with Overlap Distribution
Replicated Distribution
Block Distribution

PRI = Pulse Repetition Interval
CPI = Coherent Pulse Interval

Lockheed Martin Corporation
Software Architecture

Application “main”
Ties together a set of tasks to build the overall application

Tasks
Data-parallel code that can be mapped to a set of processors and/or strung together into a data flow. Tasks are responsible for:
- Sending and/or receiving data
- Processing the data (using the algorithms)
- Reading the stimulus control data and passing any needed control parameters into the algorithms

Algorithms
Library of higher-level, application-oriented math functions with VSIPL-like interface
- Interface uses views for input/output
- Algorithms never deal explicitly with data distribution issues

HPEC-SI development involved modification of the algorithms
Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- Conclusion
Algorithm Case Study Overview

■ Goal
  ■ Show how we reached some of our VSIPL++ conclusions by walking through the series of steps needed to convert a part of our application from VSIPL to VSIPL++

■ Algorithm
  ■ Starting point
    ■ Simplified version of a pulse compression kernel
    ■ Math: output = ifft( fft(input) * reference)
  ■ Add requirements
    ■ Error handling
    ■ Decimate input
    ■ Support both single and double precision
    ■ Port application to embedded system
Algorithm Case Study

Simple pulse compression kernel

Main Algorithm

\[ \text{output} = \text{ifft} (\text{fft} (\text{input}) \ast \text{ref}) \]

VSIPL

```c
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length size = vsip_cvgetlength_f(in);
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    vsip_cvalldestroy_f(tmpView1);
    vsip_fft_destroy_f(forwardFft);
    vsip_fft_destroy_f(inverseFft);
}
```

VSIPL++

```c
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                    const vsip::Vector< std::complex<float> > &ref,
                    const vsip::Vector< std::complex<float> > &out) {
    int size = in.size();
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);
    inverseFft( ref \ast forwardFft(in), out );
}
```

Observations

- **VSIPL++ code has fewer SLOCS than VSIPL code**
  - (5 VSIPL++ SLOCS vs. 13 VSIPL SLOCS)
- **VSIPL++ syntax is more complex than VSIPL syntax**
  - Syntax for FFT object creation
  - Extra set of parenthesis needed in defining Domain argument for FFT objects
- **VSIPL code includes more management SLOCS**
  - VSIPL code must explicitly manage temporaries
  - Must remember to free temporary objects and FFT operators in VSIPL code
- **VSIPL++ code expresses core algorithm in fewer SLOCS**
  - VSIPL++ code expresses algorithm in one line, VSIPL code in three lines
  - Performance of VSIPL++ code may be better than VSIPL code
**Algorithm Case Study**

**Simple pulse compression kernel**

**Main Algorithm**

\[
\text{output} = \text{ifft}( \text{fft}(\text{input}) \ast \text{ref} )
\]

**Additional requirement**

Catch any errors and propagate error status

---

```
int pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    int valid = 0;
    vsip_length size = vsip_cvgetlength_f(in);
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    if (forwardFft && inverseFft && tmpView1 && tmpView2) {
        vsip_ccfftop_f(forwardFft, in, tmpView1);
        vsip_cvmul_f(tmpView1, ref, tmpView2);
        vsip_ccfftop_f(inverseFft, tmpView2, out);
        valid=1;
    }
    if (tmpView1) vsip_cvalldestroy_f(tmpView1);
    if (tmpView2) vsip_cvalldestroy_f(tmpView2);
    if (forwardFft) vsip_fft_destroy_f(forwardFft);
    if (inverseFft) vsip_fft_destroy_f(inverseFft);
    return valid;
}
```

---

**VSIFL**

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
const vsip::Vector< std::complex<float> > &ref,
const vsip::Vector< std::complex<float> > &out) {
    int size = in.size();
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD>  forwardFft ((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE>  inverseFft ((vsip::Domain<1>(size)), 1.0/size);
    inverseFft( ref * forwardFft(in), out );
}
```

---

**VSIFL++**

```
void pulseCompress(const vsip::Vector< std::complex<float> > &in,
const vsip::Vector< std::complex<float> > &ref,
const vsip::Vector< std::complex<float> > &out) {
    int size = in.size();
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD>  forwardFft ((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE>  inverseFft ((vsip::Domain<1>(size)), 1.0/size);
    inverseFft( ref * forwardFft(in), out );
}
```

---

**Observations**

- VSIPL code additions are highlighted
  - No changes to VSIPL++ function due to VSIPL++ support for C++ exceptions
  - 5 VSIPL++ SLOCS vs. 17 VSIPL SLOCS

- VSIPL behavior not defined by specification if there are errors in fft and vector multiplication calls
  - For example, if lengths of vector arguments unequal, implementation may core dump, stop with error message, silently write past end of vector memory, etc
  - FFT and vector multiplication calls do not return error codes
Algorithm Case Study

Simple pulse compression kernel

Main Algorithm
output = ifft( fft(input) * ref )

Additional requirement
Decimate input by N prior to first FFT

void pulseCompress( int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    vsip_cvalldestroy_f(tmpView2);
    vsip_fft_destroy_f(forwardFft);
}

void pulseCompress( int decimationFactor, const vsip::Vector< std::complex<float> > &in, const vsip::Vector< std::complex<float> > &ref, const vsip::Vector< std::complex<float> > &out) {
    int size = in.size() / decimationFactor;
    vsip::Domain<1> decimatedDomain(0, decimationFactor, size);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
    inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}

Observations

- SLOC count doesn’t change all that much for VSIPL or VSIPL++ code
  - 2 changed line for VSIPL
  - 3 changed lines for VSIPL++
  - 2 additional SLOCS for VSIPL
  - 1 additional SLOC for VSIPL++

- VSIPL version of code has a side-effect
  - The input vector was modified and not restored to original state
  - This type of side-effect was the cause of many problems/bugs when we first started working with VSIPL
Algorithm Case Study

**Simple pulse compression kernel**

**Main Algorithm**

\[ \text{output} = \text{ifft} (\text{fft} (\text{input}) \ast \text{ref}) \]

**Additional requirement**

- Decimate input by N prior to first FFT, no side-effects

```c
void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length savedSize = vsip_cvgetlength_f(in);
    vsip_length savedStride = vsip_cvgetstride_f(in);
    vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;

    vsip_ft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_ft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);

    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFactor);
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);

    vsip_cvalldestroy_f(tmpView1);
    vsip_cvalldestroy_f(tmpView2);
    vsip_fft_destroy_f(forwardFft);
    vsip_fft_destroy_f(inverseFft);
}
```

### Observations

- **VSIPL code** must save away the input vector state prior to use and restore it before returning.
- **Code size changes**
  - VSIPL code requires 4 additional SLOCS
  - VSIPL++ code does not change from prior version

```c
void pulseCompress(int decimationFactor, const vsip::Vector< std::complex<float> > &in,
                   const vsip::Vector< std::complex<float> > &ref,
                   const vsip::Vector< std::complex<float> > &out) {
    int size = in.size() / decimationFactor;
    vsip::Domain<1> decimatedDomain(0, decimationFactor, size);

    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);

    inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```
Algorithm Case Study

Simple pulse compression kernel

Main Algorithm
output = ifft( fft(input) * ref )

Additional requirement Support both single and double precision floating point

VSIPL

uint pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in);
  vsip_f_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_f_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}

VSIPL++

template<class T, class U, class V> void pulseCompress(const T &in, const U &ref, const V &out) {
  int size = in.size();
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);
  inverseFft( ref * forwardFft(in), out );
}

Observations

- VSIPL++ code has same SLOC count as original
  - Uses c++ templates (3 lines changed)
  - Syntax is slightly more complicated
- VSIPL code doubles in size
  - Function must first be duplicated
  - Small changes must then be made to code (i.e., changing _f to _d)

Lockheed Martin Corporation
Simple pulse compression kernel

Main Algorithm

output = ifft( fft(input) * ref )

Additional requirement

Support all previously stated requirements

```c
void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    void pulseCompress(int decimationFactor, vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
        vsip_length savedSize = vsip_cvgetlength_f(in);
        vsip_length savedStride = vsip_cvgetstride_f(in);
        vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;
        vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
        vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
        vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
        vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
        if (forwardFft && inverseFft && tmpView1 && tmpView2)
            vsip_cvputlength_f(in, size);
            vsip_cvputstride_f(in, decimationFactor);
            vsip_cvcreate_f(size, VSIP_MEM_NONE);
            vsip_ccfftop_f(forwardFft, in, tmpView1);
            vsip_cvmul_f(tmpView1, ref, tmpView2);
            vsip_ccfftop_d(forwardFft, in, tmpView2);
            vsip_cvputlength_d(in, savedSize);
            vsip_cvputstride_d(in, savedStride);
            if (tmpView1)   vsip_cvalldestroy_f(tmpView1);
            if (tmpView2)   vsip_cvalldestroy_f(tmpView2);
            if (forwardFft) vsip_fft_destroy_f(forwardFft);
            if (inverseFft) vsip_fft_destroy_f(inverseFft);
        }
    }
}
```
**Algorithm Case Study**

**Simple pulse compression kernel**

**Main Algorithm**

```cpp
do nothing (input not seen)
```

**Additional requirement**

*Port application to high performance embedded systems*

### Observations

- **Port to embedded Mercury system**
  - **Hardware:** Mercury VME chassis with PowerPC compute nodes
  - **Software:** Mercury beta release of MCOE 6.0 with linux operating system. Mercury provided us with instructions for using GNU g++ compiler
  - No lines of application code had to be changed

- **Port to embedded Sky system**
  - **Hardware:** Sky VME chassis with PowerPC compute nodes
  - **Software:** Sky provided us with a modified version of their standard compiler (added a GNU g++ based front-end)
  - No lines of application code had to be changed

- **Future availability of C++ with support for C++ standard**
  - Improved C++ support is in Sky and Mercury product roadmaps
  - Support for C++ standard appears to be improving industry wide

---

**Lockheed Martin Corporation**
Outline

- Overview
- Lockheed Martin Background and Experience
- VSIPL++ Application
  - Overview
  - Application Interface
  - Processing Flow
  - Software Architecture
- Algorithm Case Study
- **Conclusion**
Lockheed Martin Math Library Experience

- **Vendor supplied math libraries**
  - Advantages
    - Performance
  - Disadvantages
    - Proprietary Interface
    - Portability

- **LM Proprietary C Wrappers**

- **VSIPL standard**
  - Advantages
    - Performance
    - Portability
    - Standard interface
  - Disadvantages
    - Verbose interface (higher % of management SLOCS)

- **LM Proprietary C++ Library**

- **VSIPL++ standard**
  - Advantages
    - Standard interface
  - To Be Determined
    - Performance
    - Portability
    - Productivity

- **Thin VSIPL-like C++ wrapper**
  - Advantages
    - Performance
    - Portability
    - Productivity (fewer SLOCS, better error handling)
  - Disadvantages
    - Proprietary interface
    - Partial implementation (didn’t wrap everything)
Conclusion

- **Standard interface**
- **Productivity**
  - A VSIPL++ user’s guide, including a set of examples would have been helpful
  - The learning curve for VSIPL++ can be somewhat steep initially
  - Fewer lines of code are needed to express mathematical algorithms in VSIPL++
  - Fewer maintenance SLOCS are required for VSIPL++ programs
- **Portability**
  - VSIPL++ is portable to platforms with support for standard C++
  - Most vendors have plans to support advanced C++ features required by VSIPL++
- **Performance**
  - VSIPL++ provides greater opportunity for performance
  - Performance-oriented implementation is not currently available to verify performance

*Lockheed Martin goals are well aligned with VSIPL++ goals*
UNANIMATED BACKUPS
Algorithm Case Study

Simple pulse compression kernel
Main Algorithm
output = ifft( fft(input) * ref )

void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length size = vsip_cvgetlength_f(in);
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    vsip_cvalldestroy_f(tmpView1);
    vsip_cvalldestroy_f(tmpView2);
    vsip_fft_destroy_f(forwardFft);
    vsip_fft_destroy_f(inverseFft);
}

void pulseCompress(const vsip::Vector< std::complex<float> > &in,
                    const vsip::Vector< std::complex<float> > &ref,
                    const vsip::Vector< std::complex<float> > &out) {
    int size = in.size();
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft((vsip::Domain<1>(size)), 1.0/size);
    inverseFft(ref * forwardFft(in), out);
}

Observations

- VSIPL++ code has fewer SLOCs than VSIPL code
  (5 VSIPL++ SLOCs vs. 13 VSIPL SLOCs)
- VSIPL++ syntax is more complex than VSIPL syntax
  - Syntax for FFT object creation
  - Extra set of parenthesis needed in defining Domain argument for FFT objects
- VSIPL code includes more management SLOCs
  - VSIPL code must explicitly manage temporaries
  - Must remember to free temporary objects and FFT operators in VSIPL code
- VSIPL++ code expresses core algorithm in fewer SLOCs
  - VSIPL++ code expresses algorithm in one line,
    VSIPL code in three lines
- Performance of VSIPL++ code may be better than VSIPL code

Lockheed Martin Corporation
Algorithm Case Study

Simple pulse compression kernel

Main Algorithm

output = ifft( fft(input) * ref )

Additional requirement

Catch any errors and propagate error status

VSIPL

int pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    int valid = 0;
    vsip_length size = vsip_cvgetlength_f(in);
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    if (forwardFft && inverseFft && tmpView1 && tmpView2) {
        vsip_ccfftop_f(forwardFft, in, tmpView1);
        vsip_cvmul_f(tmpView1, ref, tmpView2);
        vsip_ccfftop_f(inverseFft, tmpView2, out);
        valid=1;
    }
    if (tmpView1) vsip_cvalldestroy_f(tmpView1);
    if (tmpView2) vsip_cvalldestroy_f(tmpView2);
    if (forwardFft) vsip_fft_destory_f(forwardFft);
    if (inverseFft) vsip_fft_destory_f(inverseFft);
    return valid;
}

VSIPL++

void pulseCompress(const vsip::Vector< std::complex<float> > &in,
    const vsip::Vector< std::complex<float> > &ref,
    const vsip::Vector< std::complex<float> > &out) {
    int size = in.size();
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);
    inverseFft( ref * forwardFft(in), out );
}

Observations

- VSIPL code additions are highlighted
- No changes to VSIPL++ function due to VSIPL++ support for C++ exceptions
- 5 VSIPL++ SLOCs vs. 17 VSIPL SLOCs
- VSIPL behavior not defined by specification if there are errors in fft and vector multiplication calls
  - For example, if lengths of vector arguments unequal, implementation may core dump, stop with error message, silently write past end of vector memory, etc
- FFT and vector multiplication calls do not return error codes
Algorithm Case Study

Simple pulse compression kernel

Main Algorithm

output = ifft( fft(input) * ref )

Additional requirement

Decimate input by N prior to first FFT

void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
  vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;
  
  vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
  vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
  
  vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
  
  vsip_cvputstride_f(in, decimationFactor);
  vsip_cvputlength_f(in, size);
  
  vsip_ccfftop_f(forwardFft, in, tmpView1);
  vsip_cvmul_f(tmpView1, ref, tmpView2);
  vsip_ccfftop_f(inverseFft, tmpView2, out);
  
  vsip_cvalldestroy_f(tmpView1);
  vsip_cvalldestroy_f(tmpView2);
  vsip_fft_destroy_f(forwardFft);
  vsip_fft_destroy_f(inverseFft);
}

void pulseCompress(int decimationFactor, const vsip::Vector< std::complex<float> > &in,
                    const vsip::Vector< std::complex<float> > &ref
                    const vsip::Vector< std::complex<float> > &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);
  
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft((vsip::Domain<1>(size)), 1.0);
  vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft((vsip::Domain<1>(size)), 1.0/size);
  
  inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}

Observations

- SLOC count doesn’t change all that much for VSIPL or VSIPL++ code
  - 2 changed line for VSIPL
  - 3 changed lines for VSIPL++
  - 2 additional SLOCS for VSIPL
  - 1 additional SLOC for VSIPL++

- VSIPL version of code has a side-effect
  - The input vector was modified and not restored to original state
  - This type of side-effect was the cause of many problems/bugs when we first started working with VSIPL
Algorithm Case Study

Simple pulse compression kernel

Main Algorithm

\[
\text{output} = \text{ifft}( \text{fft}(\text{input}) \times \text{ref})
\]

Additional requirement

Decimate input by N prior to first FFT, no side-effects

VSIPL

```c
void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length savedSize = vsip_cvgetlength_f(in);
    vsip_length savedStride = vsip_cvgetstride_f(in);
    vsip_length size = vsip_cvgetlength_f(in) / decimationFactor;
    
    vsip_fftf *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fftf *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    
    vsip_cvputlength_f(in, size);
    vsip_cvputstride_f(in, decimationFactor);
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    
    vsip_cvputlength_f(in, savedSize);
    vsip_cvputstride_f(in, savedStride);
    vsip_cvalldestroy_f(tmpView1);
    vsip_cvalldestroy_f(tmpView2);
}
```

VSIPL++

```cpp
void pulseCompress(int decimationFactor, const vsip::Vector< std::complex<float> > &in,
                  const vsip::Vector< std::complex<float> > &ref
                  const vsip::Vector< std::complex<float> > &out) {
    int size = in.size() / decimationFactor;
    vsip::Domain<1> decimatedDomain(0, decimationFactor, size);
    
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_FWD> forwardFft (vsip::Domain<1>(size), 1.0);
    vsip::FFT<vsip::Vector, vsip::cscalar_f, vsip::cscalar_f, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft (vsip::Domain<1>(size), 1.0/size);
    
    inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}
```

Observations

- VSIPL code must save away the input vector state prior to use and restore it before returning
- Code size changes
  - VSIPL code requires 4 additional SLOCS
  - VSIPL++ code does not change from prior version
Algorithm Case Study

**Simple pulse compression kernel**

**Main Algorithm**
\[ output = \text{ifft} ( \text{fft}(\text{input}) \times \text{ref} ) \]

**Additional requirement**
Support both single and double precision floating point

```c
void pulseCompress(vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length size = vsip_cvgetlength_f(in);
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(size, VSIP_MEM_NONE);
    vsip_ccfftop_f(forwardFft, in, tmpView1);
    vsip_cvmul_f(tmpView1, ref, tmpView2);
    vsip_ccfftop_f(inverseFft, tmpView2, out);
    vsip_cvalldestroy_f(tmpView1);
    vsip_fft_destroy_f(forwardFft);
    vsip_fft_destroy_f(inverseFft);
}
```

**VSILP**

Single Precision

**Double Precision**

```c
void pulseCompress(vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
    vsip_length size = vsip_cvgetlength_d(in);
    vsip_fft_d *forwardFft = vsip_ccfftop_create_d(size, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_d *inverseFft = vsip_ccfftop_create_d(size, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_d *tmpView1 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
    vsip_cvview_d *tmpView2 = vsip_cvcreate_d(size, VSIP_MEM_NONE);
    vsip_ccfftop_d(forwardFft, in, tmpView1);
    vsip_cvmul_d(tmpView1, ref, tmpView2);
    vsip_ccfftop_d(inverseFft, tmpView2, out);
    vsip_cvalldestroy_d(tmpView1);
    vsip_cvalldestroy_d(tmpView2);
    vsip_fft_destroy_d(forwardFft);
    vsip_fft_destroy_d(inverseFft);
}
```

**Observations**

- **VSILP++ code has same SLOC count as original**
  - Uses c++ templates (3 lines changed)
  - Syntax is slightly more complicated
- **VSILP code doubles in size**
  - Function must first be duplicated
  - Small changes must then be made to code (i.e., changing _f to _d)

```
template<class T, class U, class V> void pulseCompress(const T &in, const U &ref, const V &out) {
    int size = in.size();
    vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft((vsip::Domain<1>(size)), 1);
    vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft((vsip::Domain<1>(size)), 1.0/size);
    inverseFft(ref * forwardFft(in), out);
}
```
Algorithm Case Study

Simple pulse compression kernel
Main Algorithm
output = ifft( fft(input) * ref )

Additional requirement
Support all previously stated requirements

void pulseCompress(int decimationFactor, vsip_cvview_f *in, vsip_cvview_f *ref, vsip_cvview_f *out) {
    vsip_length savedSize = vsip_cvgetlength_f(in);
    vsip_length savedStride = vsip_cvgetstride_f(in);
    vsip_fft_f *forwardFft = vsip_ccfftop_create_f(savedSize, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_f *inverseFft = vsip_ccfftop_create_f(savedSize, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_f *tmpView1 = vsip_cvcreate_f(savedSize, VSIP_MEM_NONE);
    vsip_cvview_f *tmpView2 = vsip_cvcreate_f(savedSize, VSIP_MEM_NONE);
    if (forwardFft && inverseFft && tmpView1 && tmpView2) {
        vsip_cvputlength_f(in, size);
        vsip_cvputstride_f(in, decimationFactor);
        vsip_ccfftop_f(forwardFft, in, tmpView1);
        vsip_cvmul_f(tmpView1, ref, tmpView2);
        vsip_ccfftop_f(inverseFft, tmpView2, out);
    }
    if (tmpView1)   vsip_cvalldestroy_f(tmpView1);
    if (tmpView2)   vsip_cvalldestroy_f(tmpView2);
    if (forwardFft) vsip_fft_destroy_f(forwardFft);
    if (inverseFft) vsip_fft_destroy_f(inverseFft);
}

void pulseCompress(int decimationFactor, vsip_cvview_d *in, vsip_cvview_d *ref, vsip_cvview_d *out) {
    vsip_length savedSize = vsip_cvgetlength_d(in);
    vsip_length savedStride = vsip_cvgetstride_d(in);
    vsip_fft_d *forwardFft = vsip_ccfftop_create_d(savedSize, 1.0, VSIP_FFT_FWD, 1, VSIP_ALG_SPACE);
    vsip_fft_d *inverseFft = vsip_ccfftop_create_d(savedSize, 1.0/size, VSIP_FFT_INV, 1, VSIP_ALG_SPACE);
    vsip_cvview_d *tmpView1 = vsip_cvcreate_d(savedSize, VSIP_MEM_NONE);
    vsip_cvview_d *tmpView2 = vsip_cvcreate_d(savedSize, VSIP_MEM_NONE);
    if (forwardFft && inverseFft && tmpView1 && tmpView2) {
        vsip_cvputlength_d(in, size);
        vsip_cvputstride_d(in, decimationFactor);
        vsip_ccfftop_d(forwardFft, in, tmpView1);
        vsip_cvmul_d(tmpView1, ref, tmpView2);
        vsip_ccfftop_d(inverseFft, tmpView2, out);
    }
    if (tmpView1)   vsip_cvalldestroy_d(tmpView1);
    if (tmpView2)   vsip_cvalldestroy_d(tmpView2);
    if (forwardFft) vsip_fft_destroy_d(forwardFft);
    if (inverseFft) vsip_fft_destroy_d(inverseFft);
}

template<class T, class U, class V> void pulseCompress(int decimationFactor, const T &in, const U &ref, const V &out) {
    int size = in.size() / decimationFactor;
    vsip::Domain<1> decimatedDomain(0, decimationFactor, size);
    vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_FWD> forwardFft ((vsip::Domain<1>(size)), 1);
    vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft ((vsip::Domain<1>(size)), 1.0/size);
    inverseFft( ref * forwardFft( in(decimatedDomain) ), out );
}

Final SLOC count
- VSIP++ -- 6 SLOCS
- VSIP -- 40 SLOCS
(20 each for double and single precision versions)
Algorithm Case Study

Simple pulse compression kernel
Main Algorithm
output = ifft( fft(input) * ref )

Additional requirement
Port application to high performance embedded systems

Observations
- Port to embedded Mercury system
  - Hardware: Mercury VME chassis with PowerPC compute nodes
  - Software: Mercury beta release of MCOE 6.0 with linux operating system. Mercury provided us with instructions for using GNU g++ compiler
  - No lines of application code had to be changed

- Port to embedded Sky system
  - Hardware: Sky VME chasis with PowerPC compute nodes
  - Software: Sky provided us with a modified version of their standard compiler (added a GNU g++ based front-end)
  - No lines of application code had to be changed

- Future availability of C++ with support for C++ standard
  - Improved C++ support is in Sky and Mercury product roadmaps
  - Support for C++ standard appears to be improving industry wide

VSIPL++

template<class T, class U, class V> void pulseCompress(int decimationFactor, const T &in, const U &ref, const V &out) {
  int size = in.size() / decimationFactor;
  vsip::Domain<1> decimatedDomain(0, decimationFactor, size);
  vsip::FFT<vsip::Vector, typename T::value_type, typename V::value_type, vsip::FFT_INV, 0, vsip::SINGLE, vsip::BY_REFERENCE> inverseFft (vsip::Domain<1>{size}, 1.0/sizes);
  inverseFft(ref * forwardFft(in(decimatedDomain)), out);
}

Lockheed Martin Corporation