1 Introduction

VSIPL++\(^1\) is the object-oriented “next-generation” version of the Vector Signal and Image Processing Library (VSIPL).\(^2\) Like VISPL, VSIPL++ specifies an Application Programming Interface (API) for use in the development of high-performance numerical applications, with a particular focus on embedded real-time systems performing signal processing and image processing. VSIPL++ contains a number of improvements relative to VISPL including a simpler, more intuitive programming model, simpler syntax, and greater flexibility and extensibility. The most significant of VSIPL++’s improvements is its support for multi-processor computation. This parallel support requires only that the user specify the way in which data should be distributed across processors. The VSIPL++ library automatically manages the transmission of data between the processors as necessary to effectively perform the desired computations.

CodeSourcery was awarded funding under the Air Force Small Business Investment Research (SBIR) program to develop a prototype version of the parallel functionality described in the VSIPL++ specification and to obtain measurements of VSIPL++ performance when executing on parallel systems.\(^3\) Our prototype implementation achieves a near-linear speedup on multi-processor systems demonstrating that, despite the high level of abstraction present in VSIPL++, it is nevertheless possible to obtain excellent performance. Thus, VSIPL++ has the potential to allow programmers to easily and rapidly develop systems that are both highly portable and highly efficient. In our presentation, we will describe the parallel VSIPL++ programming model, our parallel performance benchmark, and the results we obtained.

2 Benchmark Description

Beamforming is the detection of energy propagating in a particular direction while rejecting energy propagating in other directions. A beamformer consists of an array of sensors capturing signals and a signal processing algorithm to extract signals from one or more particular directions and one or more particular frequencies. The \(k\)-\(\Omega\) beamformer we consider assumes uniform spacing of omnidirectional individual sensors along the \(x\)-axis. No assumptions about the signal’s structure are made except that the signal is periodic and that the signal source is sufficiently far away that the signal appears planar to the sensors, and noise is assumed to be uniformly distributed across the signal.

The beamformer computes the power of the incoming signal for various bearings (\(k\)) and frequencies (\(\Omega\)). Those \(k\)-\(\Omega\) pairs where the power is strongest indicate incoming signals. Each sensor samples the input signal over time. After enough samples have been obtained, a computation is performed (involving the inputs from all of the sensors) to determine the power for the \(k\)-\(\Omega\) pairs. First, FIR filters remove higher-order frequencies from the signal matrix. Then a real-to-complex FFT is applied to the rows of the matrix, optionally the data is reordered into a column-major matrix, and finally a complex-to-complex FFT is applied to the columns. Generally, the collection of data and determining of power is repeated multiple times. The final power reported for a given \(k\)-\(\Omega\) pair is the average of that computed for the various iterations of the process.

\(^1\)http://www.hpec-si.org/private/vsipl++specification.html
\(^2\)http://www.vsipl.org
\(^3\)SBIR Contract FA8750-04-C-0017
**VSIPL++: Parallel Performance**

**AUTHOR(S)**

CodeSourcery, LLC

**SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

CodeSourcery, LLC

**ABSTRACT**

See also ADM00001742, HPEC-7 Volume 1, Proceedings of the Eighth Annual High Performance Embedded Computing (HPEC) Workshops, 28-30 September 2004 Volume 1., The original document contains color images.

**SUPPLEMENTARY NOTES**

Approved for public release, distribution unlimited

**DISTRIBUTION/AVAILABILITY STATEMENT**

Approved for public release, distribution unlimited

**REPORT DATE**

01 FEB 2005

**REPORT TYPE**

N/A

**DATES COVERED**

-
3 Implementation

We implemented the beamformer using several different programming methodologies. One implementation was written using C and VSIPL. After that implementation was complete, we developed a C++ and VSIPL++ implementation. The VSIPL and VSIPL++ implementations are similar in structure, but the VSIPL++ implementation is shorter than the VSIPL implementation because of the higher levels of abstraction provided by VSIPL++. Each implementation runs the $k$-$\Omega$ beamformer multiple times and computes a "running average" power spectra.

We modified the VSIPL++ reference implementation, developed by CodeSourcery under contract from MIT Lincoln Laboratory, to contain support for a subset of the functionality being considered for the parallel VSIPL++ specification. In particular, we created a data storage abstraction called DistributedBlock to represent a single one- or two-dimensional array that is stored across multiple processors. We specialized VSIPL++ algorithms, e.g., computing FIR filters and FFTs, to perform only local computations when operating on a DistributedBlock. We also modified VSIPL++ to implement a specialization of the two-dimensional FFT algorithm so that, when the input is a row-distributed DistributedBlock and the output is a column-distributed block, the algorithm performs the "corner-turn" required.

The VSIPL++ specification is written so as to be independent of any particular message-passing or threading system. However, in our implementation we chose to use the popular Message Passing Interface (MPI)\(^4\) to transmit data between cooperating processors.

After making these modifications to the VSIPL++ implementation, we made minor changes to our VSIPL++ benchmark program. These changes consisted only of modifications to the types used to declare particular arrays in the benchmark program. For example, some arrays were modified to use DistributedBlock to indicate distribution across processors. The types of these arrays indicate the arrays are distributed by rows or columns.

4 Results

The following table demonstrates that we were able to obtain a near-linear speedup with parallel VSIPL++ relative to serial VSIPL++ and VSIPL. Times are shown for the VSIPL implementation of the benchmark, the serial VSIPL++ implementation, and the parallel VSIPL++ implementation with one and two processors. Times for two hundred iterations of one problem instance are presented. Times for other instances are similar and will be presented in the extended version of this paper. In all cases, the times were obtained by running on a dual-processor Intel Pentium 4 Xeon GNU/Linux machine. The times given reflect only time spent in the execution of the beamforming computations. They do not include time required for initialization and finalization of the application and its libraries. All times shown are "wall-clock time," i.e., the total number of seconds required to execute the beamformer including time spent in the operating system kernel.

<table>
<thead>
<tr>
<th></th>
<th>serial, no corner turn</th>
<th>distributed VSIPL++</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VSIPL</td>
<td>VSIPL++</td>
<td>1-processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2-processor</td>
</tr>
<tr>
<td>FIR filter</td>
<td>20.7</td>
<td>20.7 + 0.1</td>
<td>20.7 + 0.3</td>
</tr>
<tr>
<td>1st FFT</td>
<td>12.7</td>
<td>12.7 + 0.1</td>
<td>12.7 + 0.0</td>
</tr>
<tr>
<td>Corner Turn</td>
<td>--</td>
<td>--</td>
<td>6.2 + 2.9</td>
</tr>
<tr>
<td>2nd FFT</td>
<td>10.0 + 9.2</td>
<td>10.0 + 9.1</td>
<td>10.0</td>
</tr>
</tbody>
</table>

The corner-turn times indicate the seconds required to transpose a row-major matrix to a column-major matrix plus the time for an MPI All-to-All computation. With one processor, only the transpose occurs. With two processors, MPI communication also occurs. The serial implementations do not perform the transpositions, leading to an increase in the time for the second FFT.

\(^4\)http://www-unix.mcs.anl.gov/mpi/
Abstract Submission Details

1. Title: VSIPL++: Parallel Performance

2. Authors:
   - Mr. Mark Mitchell
     CodeSourcery, LLC
     9978 Granite Point Ct
     Granite Bay, CA 95746
     +1.916.791.8304 (voice)
     +1.916.914.2066 (fax)
     mark@codesourcery.com
     USA citizenship
   - Dr. Jeffrey D. Oldham
     CodeSourcery, LLC
     144 Wyandotte Dr
     San Jose, CA 95123-3727
     +1.408.578.5684 (voice)
     +1.408.578.5684 (fax)
     oldham@codesourcery.com
     USA citizenship

3. First Author: Mitchell
   Corresponding Author: Oldham
   Presenting Author: Oldham

4. Submit for any session.

5. Prefer talk, not poster.

6. Work areas:
   - Middleware Libraries and Application Programming Interfaces
   - Software Architectures, Reusability, Scalability, and Standards
Challenge

“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 → Multiprocessor Support

SIMD Support

“Automated to reduce implementation cost.”

“Drastically reduce the performance penalties associated with deploying object-oriented software on high performance parallel embedded systems.”
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.
<table>
<thead>
<tr>
<th>Limitations of VSIPL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Uses C Programming Language</strong></td>
</tr>
<tr>
<td>- “Modern object oriented languages (e.g., C++) have consistently reduced the development time of software projects.”</td>
</tr>
<tr>
<td>- Manual memory management.</td>
</tr>
<tr>
<td>- Cumbersome syntax.</td>
</tr>
<tr>
<td><strong>Inflexible</strong></td>
</tr>
<tr>
<td>- Abstractions prevent users from adding new high-performance functionality.</td>
</tr>
<tr>
<td>- No provisions for loop fusion.</td>
</tr>
<tr>
<td>- No way to avoid unnecessary block copies.</td>
</tr>
<tr>
<td><strong>Not Scalable</strong></td>
</tr>
<tr>
<td>- No support for MPI or threads.</td>
</tr>
<tr>
<td>- SIMD support must be entirely coded by vendor; user cannot take advantage of SIMD directly.</td>
</tr>
</tbody>
</table>
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.
VSIPL++ Status

- **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**
  - Removes side lobes

- **Weights**
  - Row-wise FFT
  - “Corner turn” from sensor domain to time domain
  - Optional corner turn

- **Magsq, \( \times \frac{1}{n} \)**
  - Add this matrix to the sum

- **Column-wise FFT**
VSIPL Kernel

Seven statements required:

```c
for (i = n; i > 0; --i) {
    filtered = filter (firs, signal);
    vsip_mmul_f (weights, filtered, filtered);
    vsip_rcffftmpop_f (space_fft, filtered, fft_output);
    vsip_ccffttmpi_f (time_fft, fft_output);
    vsip_mcmagsq_f (fft_output, power);
    vsip_ssmul_f (1.0 / n, power);
    vsip_madd_f (power, spectra, spectra);
}
```
VSIPL++ Kernel

One statement required:

```
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:

```cpp
Matrix<float_t, Dense<2, float_t> > signal_matrix;
```

Parallel case:

```cpp
typedef Dense<2, float_t> subblock;
typedef Distributed<2, float_t, subblock, ROW> Block2R_t;
Matrix<float_t, Block2R_t> signal_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

![Graph showing performance comparison between VSIPL and VSIPL++.](image)

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>VSIPL</th>
<th>VSIPL++</th>
<th>VSIPL++ (1)</th>
<th>VSIPL++ (2)</th>
<th>VSIPL++ (4)</th>
<th>VSIPL++ (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x512</td>
<td>3.3</td>
<td>3.5</td>
<td>3.6</td>
<td>1.9</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>512x1024</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>9</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1024x2048</td>
<td>64</td>
<td>66</td>
<td>63</td>
<td>37</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>2048x4096</td>
<td>306</td>
<td>314</td>
<td>277</td>
<td>165</td>
<td>78</td>
<td>40</td>
</tr>
</tbody>
</table>

**Notes:**
- The table above summarizes the performance comparison between VSIPL and VSIPL++ for different problem sizes.
- The times are in seconds.
- VSIPL++ shows improved performance compared to VSIPL.
Parallel Speedup

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

Problem Size

Overhead Compared to Linear Speedup

VSIPL 0% 0% 0% 0%
VSIPL++ 1% 0% 0% 0%
VSIPL++ (1) 6% 0% -6% -14%
VSIPL++ (2) 8% 14% 6% -6%
VSIPL++ (4) 3% 11% 5% -7%
VSIPL++ (8) 1% 4% 5% -8%

Corner turn improves execution.

10% slower than perfect linear speedup

10% faster than perfect linear speedup

Minimal parallel overhead.
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