A High-Level Signal Processing Library for Multicore Processors

Sharon Sacco, Nadya Bliss, Ryan Haney, Jeremy Kepner, Hahn Kim, Sanjeev Mohindra, Glenn Schrader and Edward Rutledge

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government
Outline

• Overview

• HPEC Challenge Benchmarks

• Parallel Vector Tile Optimizing Library

• Summary
Embedded Digital Systems

Next-Generation Warfighting Vision

Rapid System Prototyping
- Widebody Airborne Sensor Platform
- Greenbank
- Triple Canopy Foliage Penetration

Advanced Hardware Implementations
- Receiver on-Chip
- Very Large Scale Integration/Field Programmable Gate Array Hybrids

Open System Technologies
- High Performance Embedded Computing—Software Initiative
- Parallel Matlab
- Next Gen Radar Open Systems Architecture

Network & Decision Support Initiatives
- Integrated Sensing and Decision Support
- LLGrid

MIT Lincoln Laboratory

GT Cell Workshop-3
SMHS 6/19/2007
Embedded Processor Evolution

- 20 years of exponential growth in FLOPS / Watt
- Requires switching architectures every ~5 years
- Cell processor is current high performance architecture
Outline

- Overview
- HPEC Challenge Benchmarks
- Parallel Vector Tile Optimizing Library
- Summary

- Time-Domain FIR Filter
- Results
**Genetic Algorithm**

- Selection
- Crossover
- Mutation
- Evaluation

- Evaluate each chromosome
- Select chromosomes for next generation
- Crossover: randomly pair up chromosomes and exchange portions
- Mutation: randomly change each chromosome

**Pattern Match**

- Compute best match for a pattern out of set of candidate patterns
  - Uses weighted mean-square error

**Database Operations**

- Three generic database operations: 
  - search: find all items in a given range
  - insert: add items to the database
  - delete: remove item from the database

**Corner Turn**

- Memory rearrangement of matrix contents
  - Switch from row to column major layout

---

*Numbers denote memory content*
**HPEC Challenge**

**Signal and Image Processing Kernels**

**FIR**
- Input Matrix
- M Channels (N channels)
- M Filters (~10 coefficients)
- Bank of filters applied to input data
- FIR filters implemented in time and frequency domain

**QR**
- A \( (M \times N) \)
- \( Q \) \( (M \times M) \)
- \( R \) \( (M \times N) \)
- Computes the factorization of an input matrix, \( A = QR \)
- Implementation uses Fast Givens algorithm

**SVD**
- Input Matrix
- Bidiagonal Matrix
- Diagonal Matrix \( \Sigma \)
- Produces decomposition of an input matrix, \( X = U \Sigma V^H \)
- Classic Golub-Kahan SVD implementation

**CFAR**
- Dopplers
- Range
- Beams
- \( C(i,j,k) \)
- \( T(i,j,k) \)
- Target List \( (i,j,k) \)
- Normalize, Threshold
- Creates a target list given a data cube
- Calculates normalized power for each cell, thresholds for target detection

---

MIT Lincoln Laboratory

GT Cell Workshop-7
SMHS 6/19/2007
Time Domain FIR Algorithm

- **Single Filter** (example size 4)
  - Number of Operations: $k - \text{Filter size} \cdot n - \text{Input size} = n_f \cdot n_f$
  - Filter slides along reference to form dot products
  - Reference Input data
  - Output point
  - Total FOPs: $\sim 8 \times n_f \times n \times k$
  - Output Size: $n + k - 1$

- **TDFIR uses complex data**
- **TDFIR uses a bank of filters**
  - Each filter is used in a tapered convolution
  - A convolution is a series of dot products

- **HPEC Challenge Parameters TDFIR**
  - | Set | k   | n    | n_f |
  - |-----|-----|------|
  - | 1   | 128 | 4096 | 64  |
  - | 2   | 12  | 1024 | 20  |

- **FIR is one of the best ways to demonstrate FLOPS**
Performance Challenges

- Dual issue instructions
- Efficiently partition the application
- Keep the pipelines full
- Keep the data flowing
- Access memory efficiently
- Can the buses be controlled efficiently?
- Maximize the use of SIMD Registers
- Can the exact processor be selected?
- Cover memory transfers with computations
- Can information on disk be preloaded before needed?

- Don’t let the cache slow things down
- Watch out for race conditions
- Don’t let the cache slow things down
- Don’t let the cache slow things down

- Price of performance is increased programming complexity
Computations take 2 lines
Mostly loop control, pointers, and initialization
Output initialization assumed
SPE needs split complex
  • Separate real and imaginary vectors

Reference C FIR is easy to understand

```c
for (i = K; i > 0; i--){
    /* Set accumulators and pointers for dot product
       for output point */
    r1 = Rin;
    r2 = Iin;
    o1 = Rout;
    o2 = Iout;

    /* calculate contributions from a single kernel point */
    for (j = 0; j < N; j++){
        *o1 += *k1 * r1 - *k2 * r2;
        *o2 += *k2 * r1 + *k1 * r2;
        r1++; r2++; o1++; o2++;
    }

    /* update input pointers */
    k1++; k2++;
    Rout++;
    Iout++;
}
```
C with SIMD Extensions

- Inner loop contributes to 4 output points per pass
- SIMD registers in use
- Shuffling of values in registers is a requirement
  - Compilers are unlikely to recognize this type of code
- Can rival assembly code with more effort

/* load reference data and shift */
ir0 = *Rin++;
ii0 = *Iin++;
ir1 = (vector float) spu_shuffle(irOld, ir0, shift1);
ii1 = (vector float) spu_shuffle(iiOld, ii0, shift1);
ir2 = (vector float) spu_shuffle(irOld, ir0, shift2);
ii2 = (vector float) spu_shuffle(iiOld, ii0, shift2);
ir3 = (vector float) spu_shuffle(irOld, ir0, shift3);
ii3 = (vector float) spu_shuffle(iiOld, ii0, shift3);

Rtemp = kr0 * ir0 + Rtemp; ltemp = kr0 * ii0 + Itemp;
Rtemp = -(ki0 * ii0 - Rtemp); ltemp = ki0 * ir0 + Itemp;

Rtemp = kr1 * ir1 + Rtemp; ltemp = kr1 * ii1 + Itemp;
Rtemp = -(ki1 * ii1 - Rtemp); ltemp = ki1 * ir1 + Itemp;

Rtemp = kr2 * ir2 + Rtemp; ltemp = kr2 * ii2 + Itemp;
Rtemp = -(ki2 * ii2 - Rtemp); ltemp = ki2 * ir2 + Itemp;

Rtemp = kr3 * ir3 + Rtemp; ltemp = kr3 * ii3 + Itemp;
Rtemp = -(ki3 * ii3 - Rtemp); ltemp = ki3 * ir3 + Itemp;

/* update old values */

*Rout++ = Rtemp; *Iout++ = Itemp;
irOld = ir0; iiOld = ii0;

SIMD C extensions increase code complexity
  - Hardware needs consideration

Contents of inner loop of convolution

MIT Lincoln Laboratory
/* Fourth kernel point contribution */

- fma $50,$30,$66,$50 /* reout''' + rein * rek [4i+16j+3]-[4i+16j+6] */
- lqd $45,32($15) /* load imin[16j+24]-[16j+27] */
- fma $51,$31,$66,$51 /* imout''' + rein * imk [4i+16j+7]-[4i+16j+10] */
- shufb $66,$40,$42,$16 /* rein[4i+16j+17]-[4i+16j+20] in $66 */
- lqd $47,48($15) /* load imin[16j+28]-[16j+31] */
- fma $53,$31,$68,$53 /* imout''' + rein * imk [4i+16j+7]-[4i+16j+10] */
- shufb $68,$42,$44,$16 /* rein[4i+16j+21]-[4i+16j+24] in $68 */
- and $43,$43,$19 /* clear $43 for taper if necessary */
- lnop
- fma $54,$30,$70,$54 /* reout''' + rein * rek [4i+16j+11]-[4i+16j+14] */
- lqd $49,64($15) /* load imin[4i+16j+32]-[4i+16j+35] */
- fma $55,$31,$70,$55 /* imout''' + rein * imk [4i+16j+11]-[4i+16j+14] */
- shufb $70,$44,$46,$16 /* rein[4i+16j+25]-[4i+16j+28] in $70 */
- fma $56,$30,$72,$56 /* reout''' + rein * rek [4i+16j+15]-[4i+16j+18] */
- ai $14,$14,64 /* update rein address */
- fma $57,$31,$72,$57 /* imout''' + rein * imk [4i+16j+15]-[4i+16j+18] */
- shufb $72,$46,$48,$16 /* rein[4i+16j+29]-[4i+16j+32] in $72 */

Code fragment from inner loop

- Inner loop operates on 16 output values with 4 kernel values
- 78% dual issue cycles in inner loop with 4 nops
- 74 registers used

**High performance demands software to leverage hardware**
Parallel Approach Time Domain FIR

- **HPEC Challenge Benchmark TDFIR** is a series of independent convolutions
  - “Embarrassingly” parallel problem is a good place to start
  - Independent convolutions are divided among the processors
  - Computation of one convolution can be overlapped with DMAs from others
Outline

• Overview

• HPEC Challenge Benchmarks

• Parallel Vector Tile Optimizing Library

• Summary

• Time-Domain FIR Filter
• Results
Mercury Cell Processor Test System

Mercury Cell Processor System

- Single Dual Cell Blade
  - Native tool chain
  - Two 2.4 GHz Cells running in SMP mode
  - Terra Soft Yellow Dog Linux 2.6.14
- Received 03/21/06
  - booted & running same day
  - integrated/w LL network < 1 wk
  - Octave (Matlab clone) running
  - Parallel VSIPLe++ compiled

- Each Cell has 153.6 GFLOPS (single precision)
  - 307.2 for system @ 2.4 GHz (maximum)

Software includes:

- IBM Software Development Kit (SDK)
  - Includes example programs
- Mercury Software Tools
  - MultiCore Framework (MCF)
  - Scientific Algorithms Library (SAL)
  - Trace Analysis Tool and Library (TATL)
Performance Time Domain FIR

Set 1 has a bank of 64 size 128 filters with size 4096 input vectors

- Octave runs TDFIR in a loop
  - Averages out overhead
  - Applications run convolutions many times typically

<table>
<thead>
<tr>
<th># SPE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>16</td>
<td>32</td>
<td>63</td>
<td>126</td>
<td>253</td>
</tr>
</tbody>
</table>

Maximum GFLOPS for TDFIR #1
Overhead

SPE Thread Spawning Overhead
2.4 GHz

- Thread spawn takes ~ 5.3 ms / SPE
  - Minimize thread spawns
  - Use middleware that avoids thread spawns
## SLOCs and Coding Effort

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>SIMD C</th>
<th>Hand Coding</th>
<th>Parallel (8 SPE)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines of Code</strong></td>
<td>33</td>
<td>110</td>
<td>371</td>
<td>546</td>
</tr>
<tr>
<td><strong>Design Time</strong></td>
<td>Minute</td>
<td>Hour</td>
<td>Hour</td>
<td>-</td>
</tr>
<tr>
<td><strong>Coding Time</strong></td>
<td>Minute</td>
<td>Hour</td>
<td>Day</td>
<td>-</td>
</tr>
<tr>
<td><strong>Debug Time</strong></td>
<td>Minute</td>
<td>Minute</td>
<td>Day</td>
<td>-</td>
</tr>
<tr>
<td><strong>Performance Efficiency</strong></td>
<td>0.014</td>
<td>0.27</td>
<td>0.88</td>
<td>0.82</td>
</tr>
<tr>
<td><strong>GFLOPS @ 2.4 GHz</strong></td>
<td>0.27</td>
<td>5.2</td>
<td>17</td>
<td>126</td>
</tr>
</tbody>
</table>

### Software Lines of Code (SLOC) and Performance for TDFIR

- **Clear tradeoff between performance and effort**
  - C code simple, poor performance
  - SIMD C, more complex to code, reasonable performance
  - Hand coding, very complex, excellent performance
Outline

• Overview

• HPEC Challenge Benchmarks

• Parallel Vector Tile Optimizing Library (PVTOL)
  • PVTOL Architecture
  • PVTOL Development Cycle
  • Contributing Technologies

• Summary
Parallel Vector Tile Optimizing Library (PVTOL)

- PVTOL is a portable and scalable middleware library for multicore processors
- Enables incremental development

1. Develop serial code
2. Parallelize code
3. Deploy code
4. Automatically parallelize code

Make parallel programming as easy as serial programming
PVTOL Architecture

- **Performance**
  - Achieves high performance

- **Portability**
  - Built on standards, e.g. VSIPL++

- **Productivity**
  - Minimizes effort at user level

PVTOL preservess the simple load-store programming model in software
Serial PVTOL code

```c
void main(int argc, char *argv[]) {
   // Initialize PVTOL
   process pvtol(argc, argv);

   // Create input, weights, and output matrices
   typedef Dense<2, float, tuple<0, 1> > dense_block_t;
   typedef Matrix<float, dense_block_t, LocalMap> matrix_t;
   matrix_t input(num_vects, len_vect),
               filts(num_vects, len_vect),
               output(num_vects, len_vect);

   // Initialize arrays
   ...

   // Perform TDFIR filter
   output = tdfir(input, filts);
}
```
Parallel PVTOL code

```c
void main(int argc, char *argv[]) {
  // Initialize PVTOL
  process pvtol(argc, argv);

  // Add parallel map
  RunTimeMap map1(...);

  // Create input, weights, and output matrices
  typedef Dense<2, float, tuple<0, 1> > dense_block_t;
  typedef Matrix<float, dense_block_t, RunTimeMap> matrix_t;
  matrix_t input(num_vects, len_vect, map1),
               filts(num_vects, len_vect, map1),
               output(num_vects, len_vect, map1);

  // Initialize arrays
  ...

  // Perform TDFIR filter
  output = tdfir(input, filts);
}
```
PVTOL Development Process

Embedded PVTOL code

```c
void main(int argc, char *argv[]) {
    // Initialize PVTOL
    process pvtol(argc, argv);

    // Add hierarchical map
    RuntimeMap map2(...);

    // Add parallel map
    RuntimeMap map1(..., map2);

    // Create input, weights, and output matrices
    typedef Dense<2, float, tuple<0, 1>> dense_block_t;
    typedef Matrix<float, dense_block_t, RuntimeMap> matrix_t;
    matrix_t input(num_vects, len_vect, map1),
                filts(num_vects, len_vect, map1),
                output(num_vects, len_vect, map1);

    // Initialize arrays
    ...

    // Perform TDFIR filter
    output = tdfir(input, filts);
}
```
Automapped PVTOL code

```c
void main(int argc, char *argv[]) {
  // Initialize PVTOL
  process pvtol(argc, argv);

  // Create input, weights, and output matrices
  typedef Dense<2, float, tuple<0, 1>> dense_block_t;
  typedef Matrix<float, dense_block_t, AutoMap> matrix_t;
  matrix_t input(num_vects, len_vect), filts(num_vects, len_vect), output(num_vects, len_vect);

  // Initialize arrays
  ...

  // Perform TDFIR filter
  output = tdfir(input, filts);
}
```
Outline

- Overview
- HPEC Challenge Benchmarks
- Parallel Vector Tile Optimizing Library (PVTOL)
  - PVTOL Architecture
  - PVTOL Development Cycle
  - Contributing Technologies
- Summary
pMapper Automated Mapping System**

Cell Processor Simulation

- Simulate the Cell processor using pMapper simulator infrastructure
- Use pMapper to automate mapping and predict performance on the Cell

** Note: Patent process underway
Hierarchical Arrays

- Hierarchical arrays hide details of the processor and memory hierarchy
- Hierarchical maps concisely describe data distribution at each level
The High Performance Embedded Computing Software Initiative

Program Goals

• Develop and integrate software technologies for embedded parallel systems to address portability, productivity, and performance
• Engage acquisition community to promote technology insertion
• Deliver quantifiable benefits

Portability: reduction in lines-of-code to change port/scale to new system
Productivity: reduction in overall lines-of-code
Performance: computation and communication benchmarks
VSIPL++

- Portable to workstations, embedded systems, FPGAs with minimal performance cost
- Applicable to simple and complex applications
- Easier upgrade cycle
- Reduced development time and cost

- Parallelism built in
- Object Oriented
- Basic Arithmetic, Matrix Algebra, Signal Processing, and Equation Solvers

VSIPL++ is freely available
www.hpec-si.org
Mercury SAL and MCF

- **Scientific Algorithms Library (SAL)** is an FPS based library available on most Mercury products
  - Program portability within Mercury products
  - Common signal processing algorithms

- **SAL** has over 100 functions optimized for single SPE
  - FFT (1D, multiple)
  - Convolution (real, complex)
  - Matrix multiply
  - Basic arithmetic
  - Trigonometric and transcendental
  - Transpose

- **MultiCore Frameworks (MCF)** manages multi-SPE programming
  - Function offload engine model
  - Stripmining
  - Intraprocessor communications
  - Overlays
  - Profiling

- Leveraging vendor libraries reduces development time
  - Provides optimization
  - Less debugging of application
Summary

• With basic tools high performance is achievable from the Cell processor
  – Hard to program: SIMD C extensions or assembly code are required
  – Development and debugging time can be long

• PVTOL is making programming easier for Cell
  – Leverages existing technologies
  – Packages common kernels needed by users
  – API simplifies application code