Altivec Extensions to the Portable Expression Template Engine (PETE)*

Edward Rutledge

HPEC 2002
25 September, 2002
Lexington, MA

* This work is sponsored by the US Navy, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the Department of Defense.
Outline

• Overview
  – Motivation for using C++
  – Expression Templates and PETE
  – AltiVec

• Combining PETE and AltiVec
• Experiments
• Future Work and Conclusions
### Challenge:
Translate high-level statements to architecture-specific implementations (e.g. use AltiVec C language extensions)

#### Software Technologies

<table>
<thead>
<tr>
<th>Hand coded loop</th>
<th>C (e.g. VSIPL)</th>
<th>C++ (with PETE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; ROWS; i++) for (j = 0; j &lt; COLS; j++) a[i][j] = b[i][j] + c[i][j];</td>
<td>vsip_madd_f(b, c, a);</td>
<td>a = b + c;</td>
</tr>
<tr>
<td>Portability</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Productivity</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Performance</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>
Typical C++ Operator Overloading

Example: A = B + C vector add

Main

1. Pass B and C references to operator +

Operator +

2. Create temporary result vector
3. Calculate results, store in temporary
4. Return copy of temporary

5. Pass results reference to operator =

Operator =

6. Perform assignment

2 temporary vectors created

Main

Additional Memory Use

- Static memory
- Dynamic memory (also affects execution time)

Main

Additional Execution Time

- Cache misses/page faults
- Time to create a new vector
- Time to create a copy of a vector
- Time to destruct both temporaries
C++ Expression Templates and PETE

- PETE, the Portable Expression Template Engine, is available from the Advanced Computing Laboratory at Los Alamos National Laboratory
- PETE provides:
  - Expression template capability
  - Facilities to help navigate and evaluating parse trees

**Expression Example:**

\[ A = B + C \]

**Expression Type:**

```
BinaryNode<OpAdd, Reference<Vector>, Reference<Vector> >
```

**Parse Trees:**

- Parse trees, not vectors, created

**Reduced Memory Use:**

- Parse tree only contains references

**Reduced Execution Time:**

- Better cache use
- Loop fusion style optimization
- Compile-time expression tree manipulation

PETE: [http://www.acl.lanl.gov/pete](http://www.acl.lanl.gov/pete)
**Altivec Overview**

- **Altivec Architecture**
  - SIMD extensions to PowerPC (PPC) architecture
  - Uses 128-bit “vectors” == 4 32-bit floats
  - API allows programmers to directly insert Altivec code into programs
  - Theoretical max FLOP rate:
    \[
    \frac{2 \text{ vector ops}}{\text{cycle}} \times \frac{4 \text{ FLOP's}}{\text{vector op}} = 8 \text{ FLOPS /clock cycle}
    \]

- **Altivec C/C++ language extensions**
  - New “vector” keyword for new types
  - New operators for use on vector types
  - Vector types must be 16 byte aligned
  - Can cast from native “C” to vector type and vice versa

Example: \(a = b + c\)

```c
int i;
vector float *avec, *bvec, *cvec;
avec=(vector float*) a;
bvec=(vector float*) b;
cvec=(vector float*) c;
for (i = 0; i < VEC_SIZE/4; i++)
    *avec++ = vec_add(*bvec++, *cvec++);
```
AltiVec Performance Issues

System Example: DY4 CHAMP-AV board

Memory Hierarchy

- L1 Cache (32 KB data)
- L2 Cache (2 MB)
- Main Memory (64 MB)

Measured Bandwidth

- G4 Processor (400MHz=3.2 GFLOPS/sec)
- 5.5 GB/Sec=1.38 Gfloats/sec
- 1.1 GB/Sec=275 Mfloats/sec
- 112 MB/Sec=28 Mfloats/sec

• Bottleneck at every level of memory hierarchy
• Bottleneck more pronounced lower in the hierarchy

Key to good performance: avoid frequent loads/stores
• PETE helps by keeping intermediate results in registers
Outline

• Overview
  – Motivation for using C++
  – Expression Templates and PETE
  – AltiVec

• Combining PETE and AltiVec

• Experiments

• Future Work and Conclusions
PETE: A Closer Look

Step 1: Form expression

A = B + C * D

BinaryNode<OpAdd, float*, Binary Node <OpMul, float*, float*>>

Step 2: Evaluate expression

Vector Operator =

it = begin();
for (int i = 0; i < size(); i++)
{
    *it = forEach(expr, DereferenceLeaf(), OpCombine());
    forEach(expr, IncrementLeaf(), NullCombine());
    it++;
}

PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

User specifies what to store at the leaves

OpAdd + OpCombine

Dereference Leaf

Increment Leaf

Translation at compile time
- Template specialization
- Inlining
PETE: Adding AltiVec

Step 1: Form expression

A=B+C*D

BinaryNode<OpAdd, float*, Binary Node <OpMul, float*, float*>>

Step 2: Evaluate expression

Vector Operator =

```c
it = begin();
for (int i=0; i<size(); i++)
{
    *it = forEach(expr, DereferenceLeaf(), OpCombine());
    forEach (expr, IncrementLeaf(), NullCombine());
    it++;
}
```

PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining
PETE: Adding AltiVec

Step 1: Form expression

\[ A = B + C \times D \]

\[ A = \text{mulAdd} \]

\[ \text{TrinaryNode\langle OpMulAdd, float*, float*, float*\rangle} \]

Multiply-add produces trinary node

Step 2: Evaluate expression

```
Vector Operator =

it = begin();
for (int i = 0; i < size(); i++)
{
    *it = forEach(expr, DereferenceLeaf(), OpCombine());
    forEach(expr, IncrementLeaf(), NullCombine());
    it++;
}
```

PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining
Pete: Adding Altivec

**Step 1: Form expression**

A = B + C * D

A = TrinaryNode<OpMulAdd, vector float*, vector float*, vector float*>>

Multiply-add produces trinary node

**Step 2: Evaluate expression**

Vector Operator =

```c
it = begin ();
for (int i = 0; i < size (); i++ ) {
    *it = forEach (expr, DereferenceLeaf (), OpCombine () );
    forEach (expr, IncrementLeaf (), NullCombine () );
    it ++ ;
}
```

PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining

vector float* instead of float* at leaves

OpAdd + OpCombine Dereference Leaf

Increment Leaf
Step 1: Form expression

A = B + C*D

TrinaryNode<OpMulAdd, vector float*, vector float*, vector float*>>

Multiply-add produces trinary node

Step 2: Evaluate expression

it = begin();
for (int i = 0; i < size(); i++)
{
    *it = forEach(expr, DereferenceLeaf(), OpCombine());
    forEach (expr, IncrementLeaf(), NullCombine());
    it++;
}

PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining
**PETE: Adding AltiVec**

**Step 1: Form expression**

\[ A = B + C \times D \]

\( \text{TrinaryNode<OpMulAdd, vector float*, vector float*, vector float*>} \)

Multiply-add produces trinary node

**Step 2: Evaluate expression**

**Vector Operator =**

```c
it = (vector float*)begin();
for (int i = 0; i < size()/4; i++)
{
    *it = forEach(expr, DereferenceLeaf(), OpCombine());
    forEach (expr, IncrementLeaf(), NullCombine());
    it++;
}
```

**PETE ForEach:** Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

**Translation at compile time**
- Template specialization
- Inlining

**vector float* instead of float* at leaves**
**PETE: Adding AltiVec**

**Step 1: Form expression**

A = B + C*D  

A = \[ \text{mulAdd} \]

Multiply-add produces trinary node

**Step 2: Evaluate expression**

**Vector Operator**

\[
\text{it} = \text{(vector float*)begin();}
\text{for (int i = 0; i < size()/4; i++)}
\{
    \text{it} = \text{forEach(expr, DereferenceLeaf(), OpCombine());}
    \text{forEach} (expr, IncrementLeaf(), NullCombine());
    \text{it}++; 
\}
\]

**Iterate over vectors**

**New rules for internal nodes**

\[
\text{*it} = \text{vec_madd (*clt, *dlt, *blt);}
\text{blt++; clt++; dlit++;}
\]

**Translation at compile time**
- Template specialization
- Inlining

**PETE ForEach: Recursive descent traversal of expression**
- User defines action performed at leaves
- User defines action performed at internal nodes
PETE: Adding AltiVec

Step 1: Form expression

A = B + C*D

A = \text{mulAdd}

TrinaryNode<OpMulAdd, vector float*, vector float*, vector float*>>

Multiply-add produces trinary node

Step 2: Evaluate expression

Vector Operator =

it = (vector float*)begin();
for (int i = 0; i < size()/4; i++)
{
    *it = forEach(expr, DereferenceLeaf(), OpCombine());
    each(expr, IncrementLeaf(), NullCombine());
    it++;
}

PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining
Outline

• Overview
  – Motivation for using C++
  – Expression Templates and PETE
  – AltiVec
• Combining PETE and AltiVec
• Experiments
• Future Work and Conclusions
Results

- Hand coded loop achieves good performance, but is problem specific and low level
- Optimized VSIPL performs well for simple expressions, worse for more complex expressions
- PETE style array operators perform almost as well as the hand-coded loop and are general, can be composed, and are high-level

Software Technology

<table>
<thead>
<tr>
<th>AltiVec loop</th>
<th>VSIPL</th>
<th>PETE with AltiVec</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C++</td>
</tr>
<tr>
<td>For loop</td>
<td>Altivec aware VSIPRO Core Lite (<a href="http://www.mpi-softech.com">www.mpi-softech.com</a>)</td>
<td>PETE operators</td>
</tr>
<tr>
<td>Direct use of AltiVec extensions</td>
<td>No multiply-add</td>
<td>Indirect use of AltiVec extensions</td>
</tr>
<tr>
<td>Assumes unit stride</td>
<td>Cannot assume unit stride</td>
<td>Assumes unit stride</td>
</tr>
<tr>
<td>Assumes vector alignment</td>
<td>Cannot assume vector alignment</td>
<td>Assumes vector alignment</td>
</tr>
</tbody>
</table>
Experimental Platform and Method

Hardware
- **DY4 CHAMP-AV Board**
  - Contains 4 MPC7400’s and 1 MPC 8420
- **MPC7400 (G4)**
  - 450 MHz
  - 32 KB L1 data cache
  - 2 MB L2 cache
  - 64 MB memory/processor

Software
- **VxWorks 5.2**
  - Real-time OS
- **GCC 2.95.4 (non-official release)**
  - GCC 2.95.3 with patches for VxWorks
  - Optimization flags:
    - -O3
    - -funroll-loops
    - -fstrict-aliasing

Method
- Run many iterations, report average, minimum, maximum time
  - From 10,000,000 iterations for small data sizes, to 1000 for large data sizes
- All approaches run on same data
- Only average times shown here
- Only one G4 processor used

- Use of the VxWorks OS resulted in very low variability in timing
- High degree of confidence in results
Experiment 1: A=B+C

- Peak throughput similar for all approaches
- VSIPL has some overhead for small data sizes
  - VSIPL calls cannot be inlined by the compiler
  - VSIPL makes no assumptions about data alignment/stride
Experiment 2: A=B+C*D

- Loop and PETE/AltiVec both outperform VSIPL
  - VSIPL implementation creates a temporary to hold multiply result
    (no multiply-add in Core Lite)
- All approaches have similar performance for very large data sizes
- PETE/AltiVec adds little overhead compared to hand coded loop
Experiment 3: \( A = B + C \times D + E \times F \)

- Loop and PETE/AltiVec both outperform VSIPL
  - VSIPL implementation must create temporaries to hold intermediate results (no multiply-add in Core Lite)
- All approaches have similar performance for very large data sizes
- PETE/AltiVec has some overhead compared to hand coded loop
Experiment 4: \( A = B + C \times D - E / F \)

- Loop and PETE/AltiVec have similar performance
  - PETE/AltiVec actually outperforms loop for some sizes
- Peak throughput similar for all approaches
  - VSIPL implementation must create temporaries to hold intermediate results
  - VSIPL divide algorithm is probably better
Outline

• Overview
  – Motivation for using C++
  – Expression Templates and PETE
  – AltiVec
• Combining PETE and AltiVec
• Experiments
  • Future Work and Conclusions
Expression Templates and VSIPL++

VSIPL
- C binding
- Object based
- Procedural

VSIPL++ (Proposal)
- C++ binding
- Object oriented
- Generic

HPEC-SI¹

Goals:
- Simplify interface
- Improve performance

Implementation can and should use expression templates to achieve these goals

¹ HPEC-SI = High Performance Embedded Computing Software Initiative
Conclusions

- Expression templates support a high-level API
- Expression templates can take advantage of the SIMD AltiVec C/C++ language extensions
- Expression templates provide the ability to compose complex operations from simple operations without sacrificing performance
- C libraries cannot provide this ability to compose complex operations while retaining performance
  - C lacks templates and template specialization capability
  - C library calls cannot be inlined
- The C++ VSIPL binding (VSIPL++) should allow implementors to take advantage of expression template technology