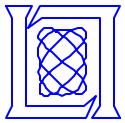
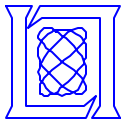


AltiVec Extensions to the Portable Expression Template Engine (PETE)*

Edward Rutledge

HPEC 2002
25 September, 2002
Lexington, MA

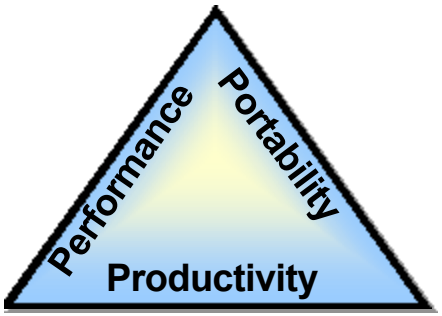
* This work is sponsored by the US Navy, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the Department of Defense.



Outline

- **Overview**
  - **Motivation for using C++**
  - **Expression Templates and PETE**
  - **AltiVec**
- **Combining PETE and AltiVec**
- **Experiments**
- **Future Work and Conclusions**

Programming Modern Signal Processors



Software Goals

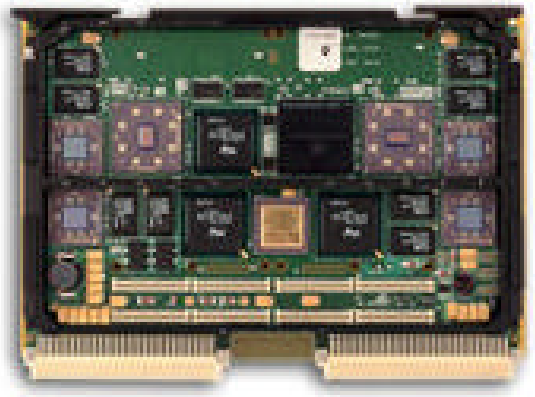
Software Technologies

	Hand coded loop	C (e.g. VSIPL)	C++ (with PETE)

	`for (i = 0; i < ROWS; i++)`
	`  for (j = 0; j < COLS; j++)`
	`  a[i][j] = b[i][j] + c[i][j];`
		`vsip_madd_f(b, c, a);`	`a = b + c;`

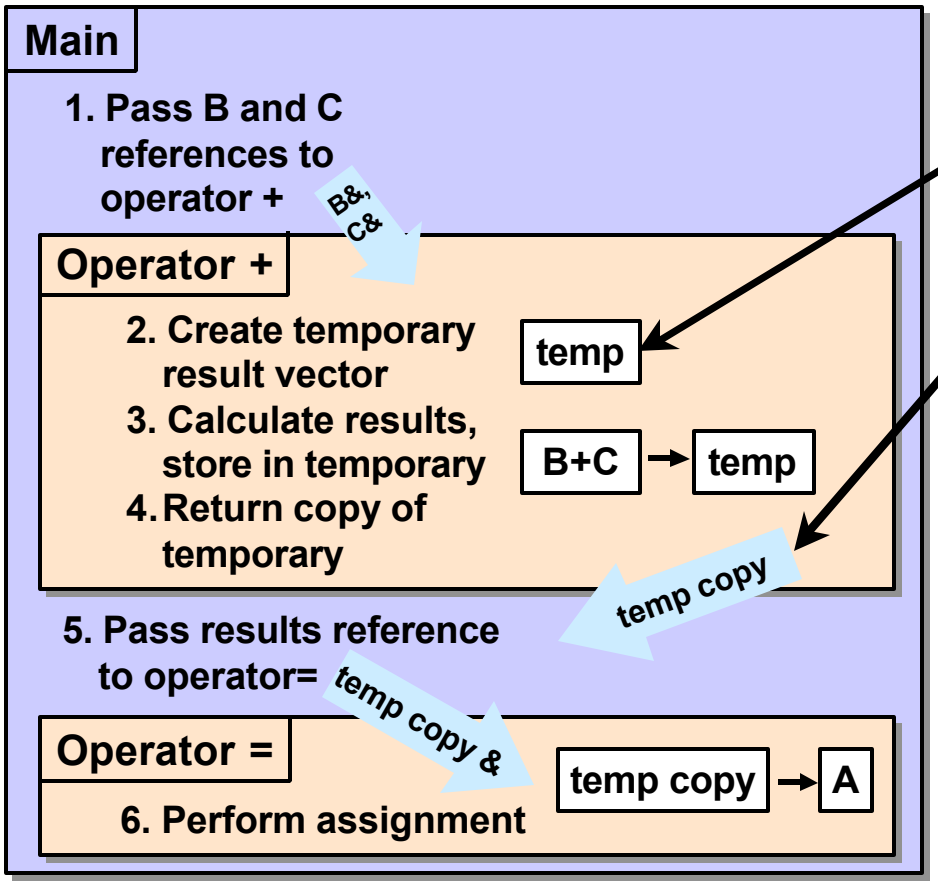
Portability	Low	High	High
Productivity	Low	Medium	High
Performance	High	Medium	High! (with PETE)

Challenge:
Translate high-level statements to architecture-specific implementations (e.g. use AltiVec C language extensions)



Typical C++ Operator Overloading

Example: A=B+C vector add



2 temporary vectors created

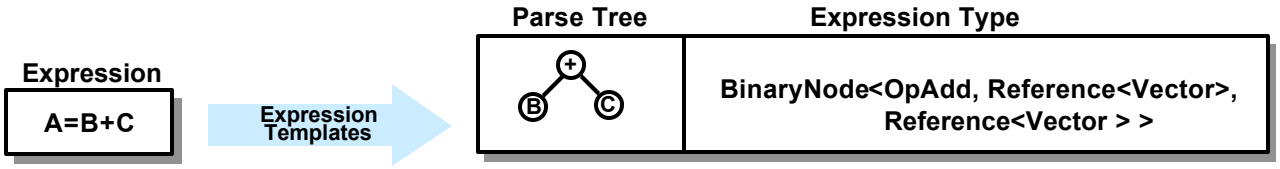
Additional Memory Use

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

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



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

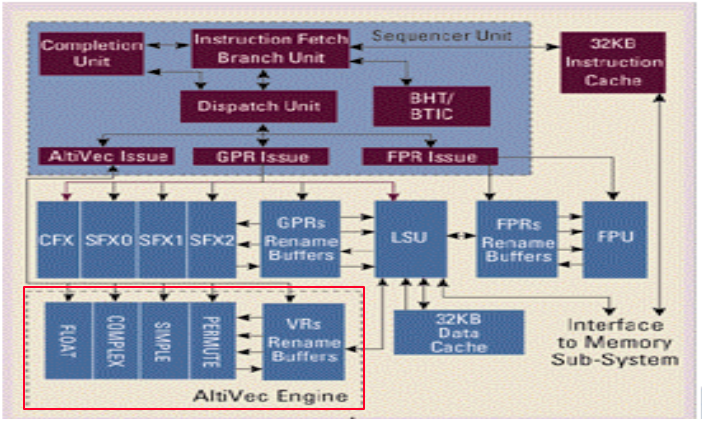
- PETE, the Portable Expression Template Engine, is available from the Advanced Computing Laboratory at Los Alamos National Laboratory
- PETE provides:
  - Expression template capability
  - Facilities to help navigate and evaluating parse trees

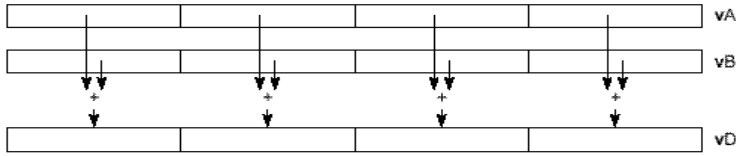
PETE: http://www.acl.lanl.gov/pete

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:

$$2\frac{\text{vector ops}}{\text{cycle}} \times 4\frac{\text{FLOP's}}{\text{vector op}}$$
$$= 8 \text{ FLOPS/clock cycle}$$





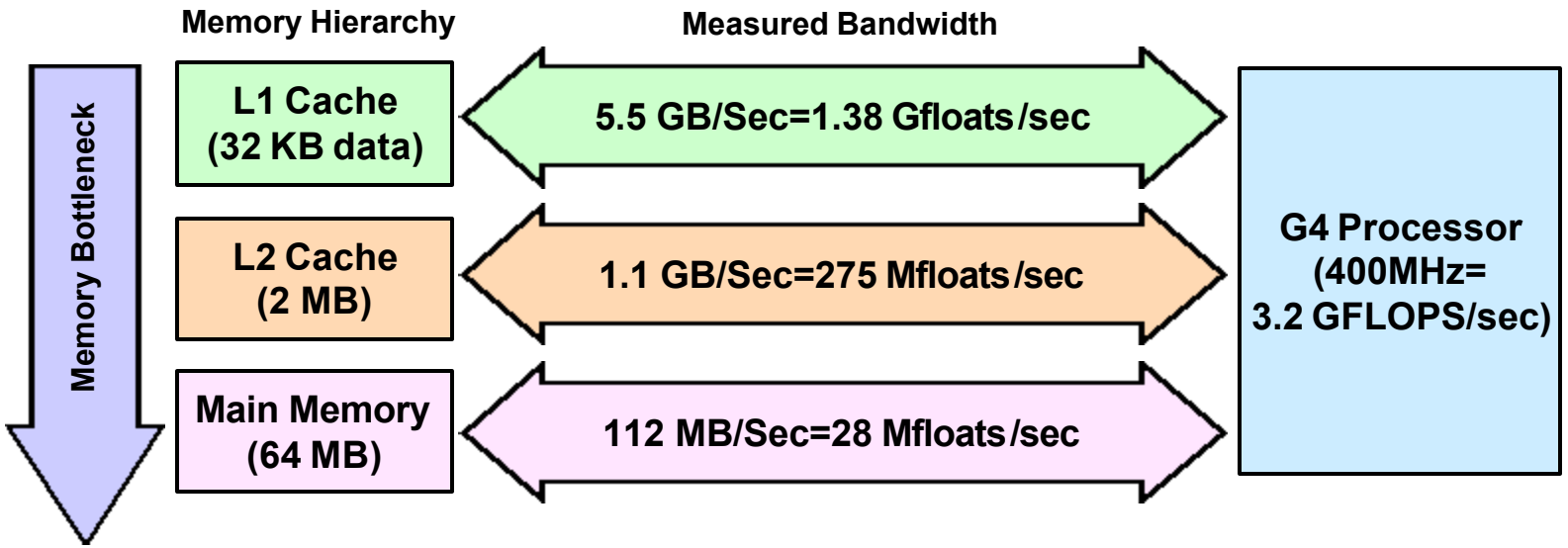
- AltiVec C/C++ language extensions
  - New “vector” keyword for new types
  - New operators for use on vector types
  - Vector types must be 16 byte aligned
  - Can cast from native “C” to vector type and vice versa

Example: a=b+c

```
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++);
```

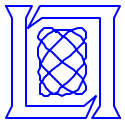
AltiVec Performance Issues

System Example: DY4 CHAMP-AV board



- 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**



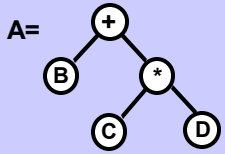
Outline

- 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*>>

User specifies what to store at the leaves

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++;
}
```

Action performed at leaves

Action performed at internal nodes

OpAdd + OpCombine

Dereference Leaf

```
*it= *blt + *clt * *dlt;
blt++; clt++; dlt++;
```

Increment Leaf

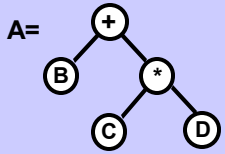
PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining

PETE: Adding AltiVec

Step 1: Form expression

A=B+C*D

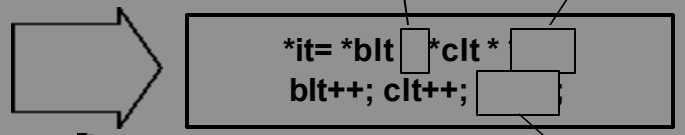


BinaryNode<OpAdd, float*, Binary Node <OpMul, float*, float*>>

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++;
}
```



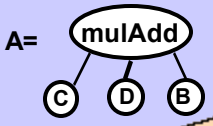
PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining

PETE: Adding AltiVec

Step 1: Form expression

A=B+C*D



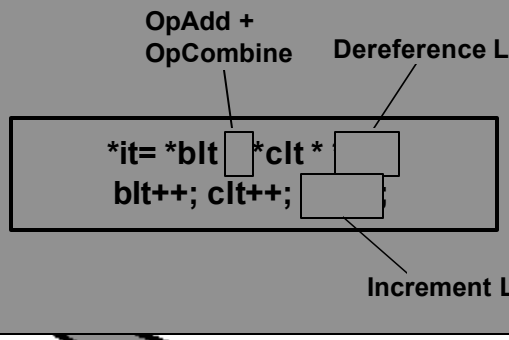
TrinaryNode<OpMulAdd, float*, float*, float*>>

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++;
}
```



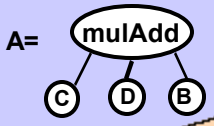
PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining

PETE: Adding AltiVec

Step 1: Form expression

A=B+C*D



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

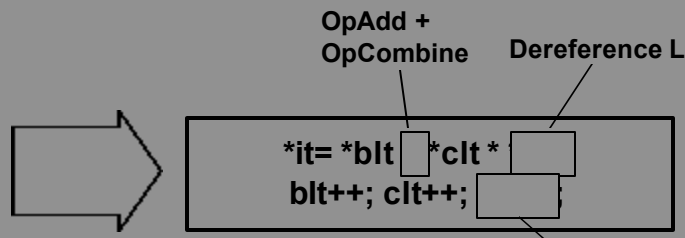
Multiply-add produces trinary node

vector float* instead of float* at leaves

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++;
}
```



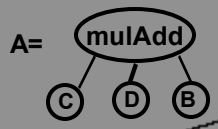
PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining

PETE: Adding AltiVec

Step 1: Form expression

A=B+C*D



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

Multiply-add produces trinary node

vector float* instead of float* at leaves

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++;
}
```

OpAdd + OpCombine

Dereference Leaf

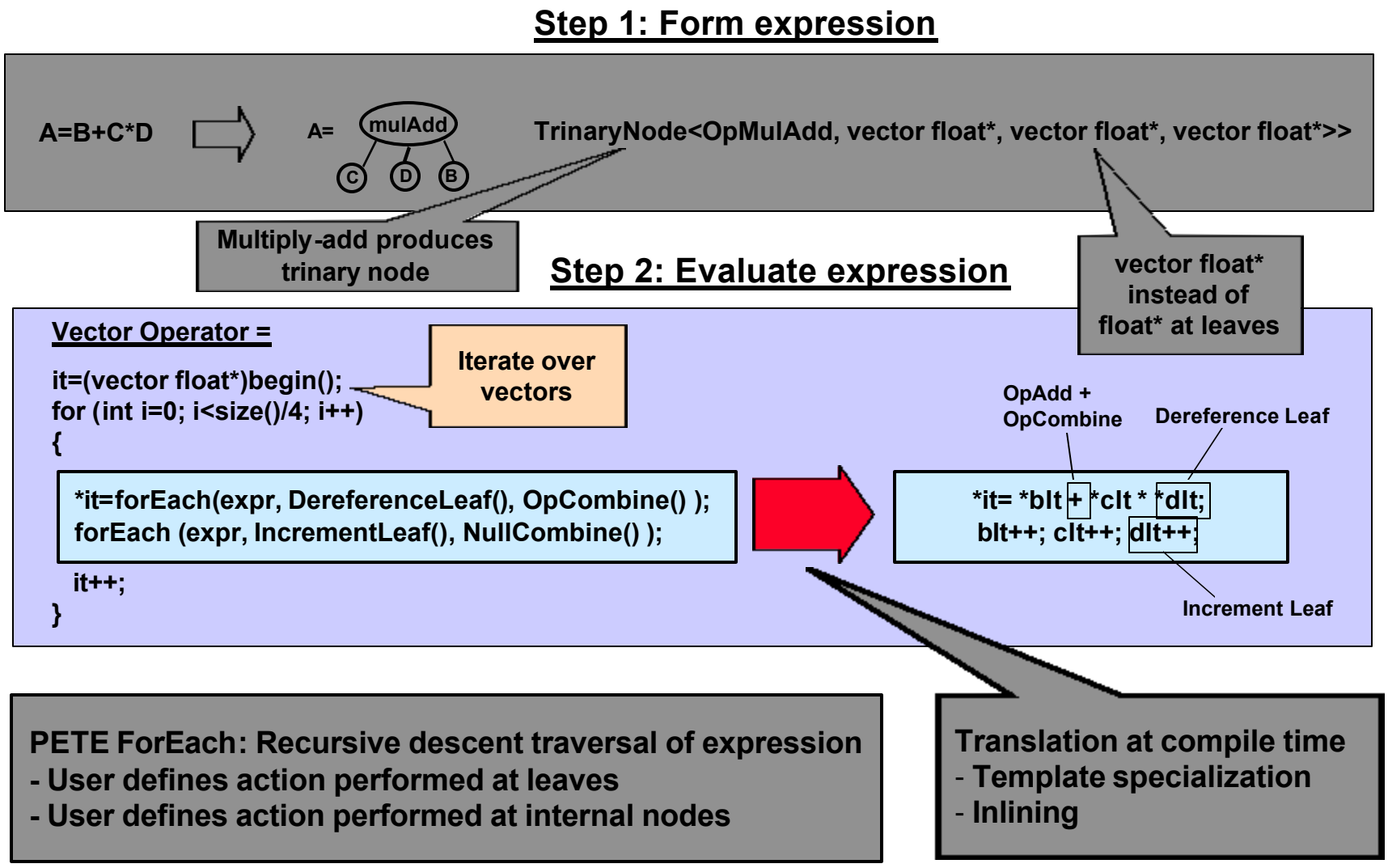
```
*it= *blt + *clt * *dlt;
blt++; clt++; dlt++;
```

Increment Leaf

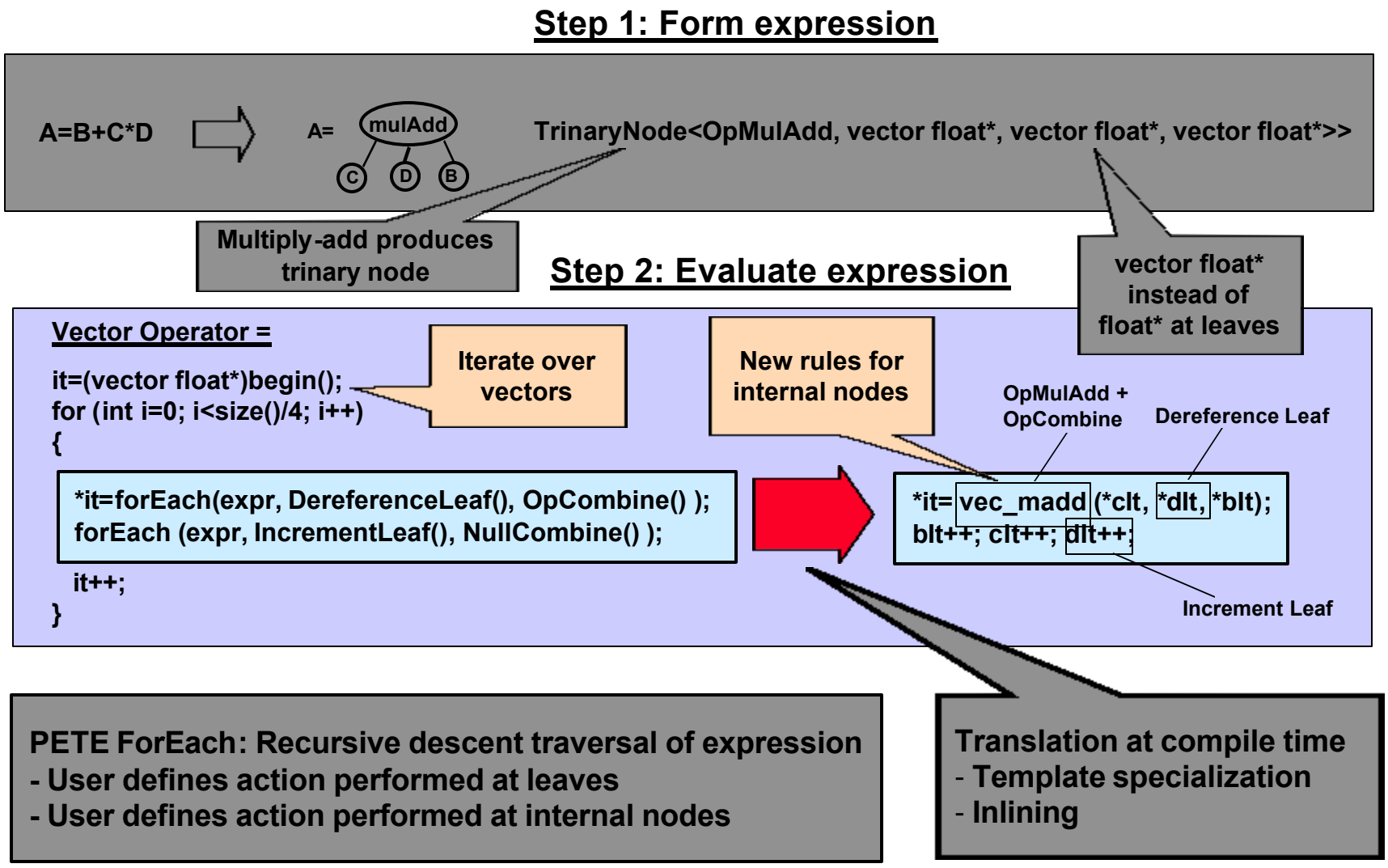
PETE ForEach: Recursive descent traversal of expression
- User defines action performed at leaves
- User defines action performed at internal nodes

Translation at compile time
- Template specialization
- Inlining

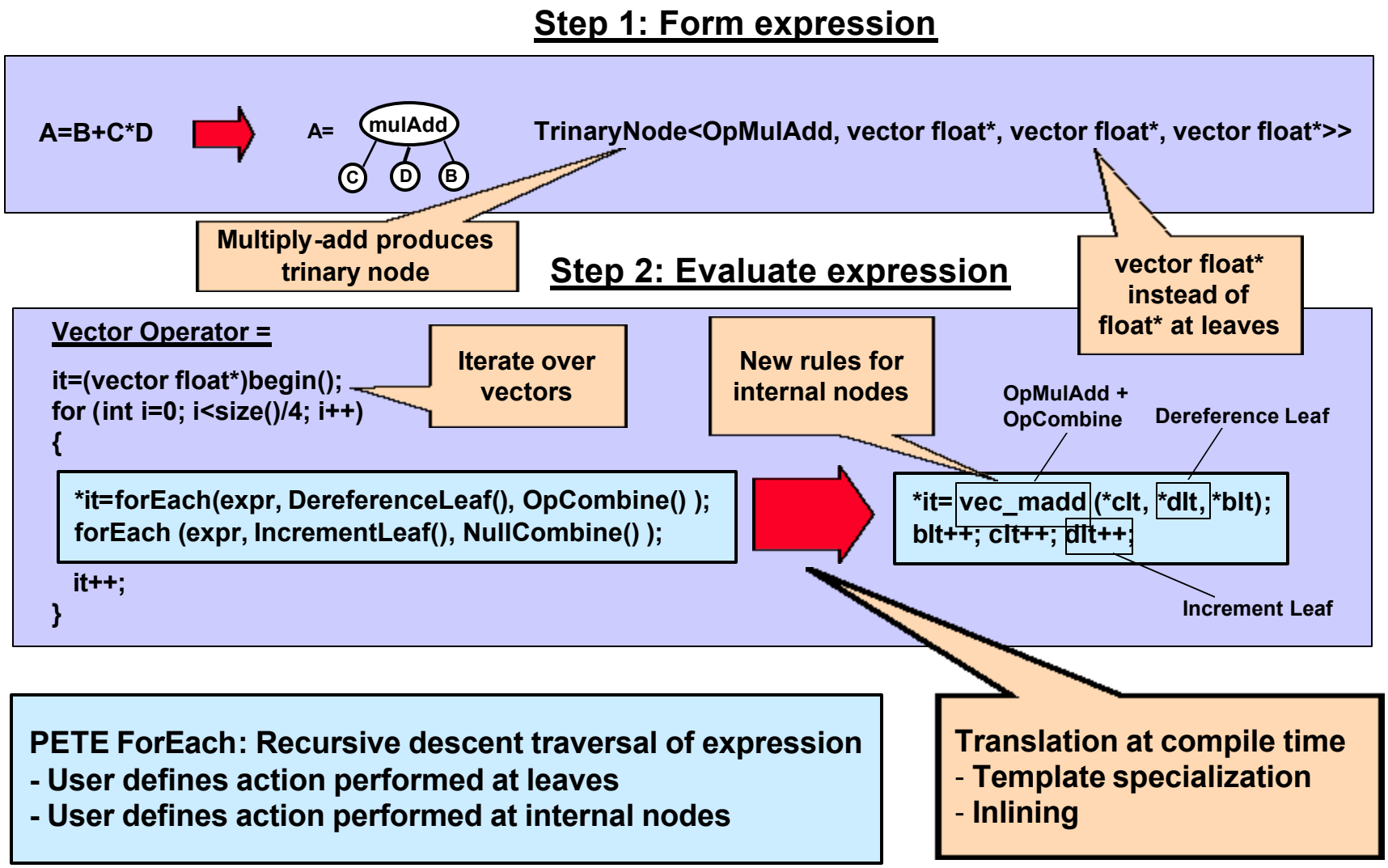
PETE: Adding AltiVec

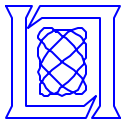


PETE: Adding AltiVec



PETE: Adding AltiVec





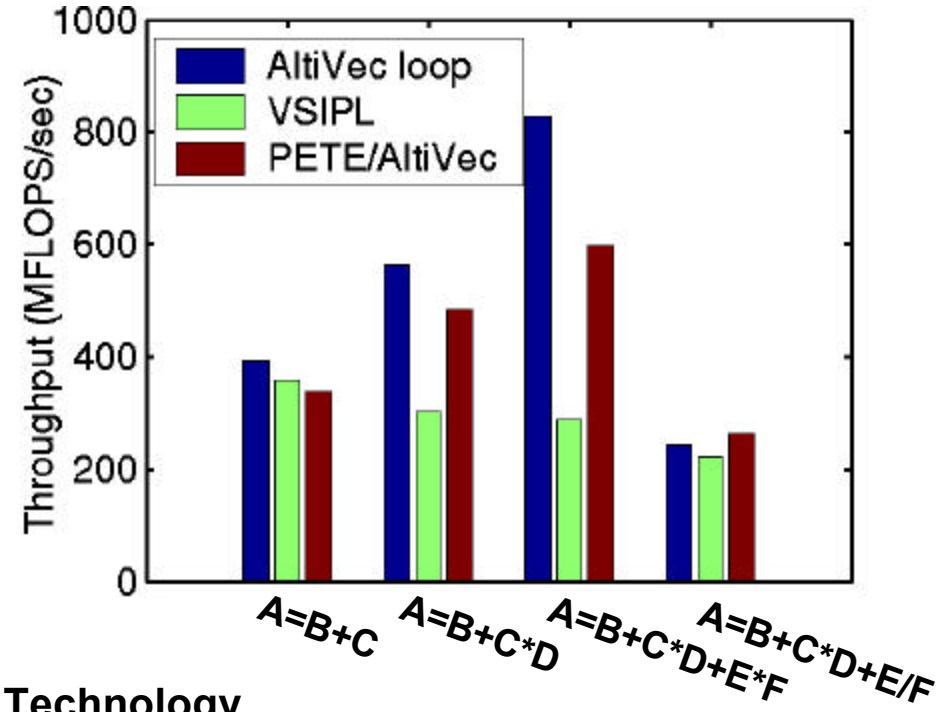
Outline

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

Experiments

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



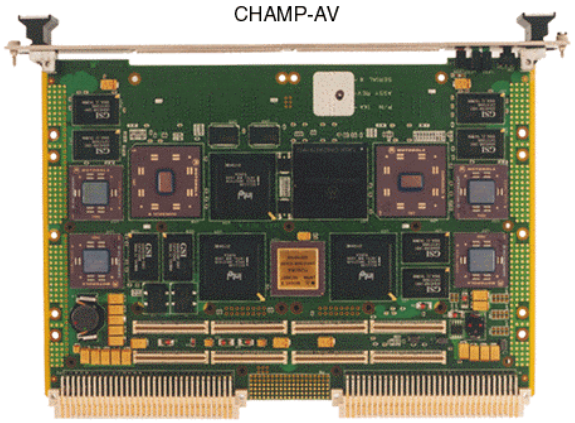
Software Technology

AltiVec loop	VSIPL	PETE with AltiVec
• C • For loop • Direct use of AltiVec extensions • Assumes unit stride • Assumes vector alignment	• C • AltiVec aware VSIPro Core Lite (www.mpi-softtech.com) • No multiply-add • Cannot assume unit stride • Cannot assume vector alignment	• C++ • PETE operators • Indirect use of AltiVec extensions • Assumes unit stride • Assumes vector alignment

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:

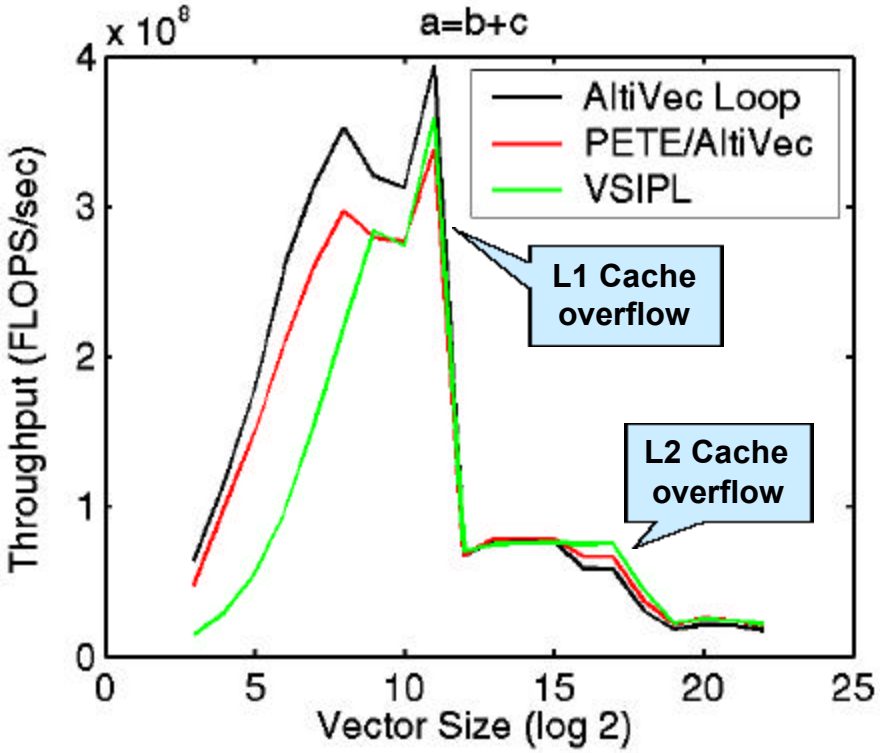
    -O3 -funroll-loops
    -fstrict-aliasing

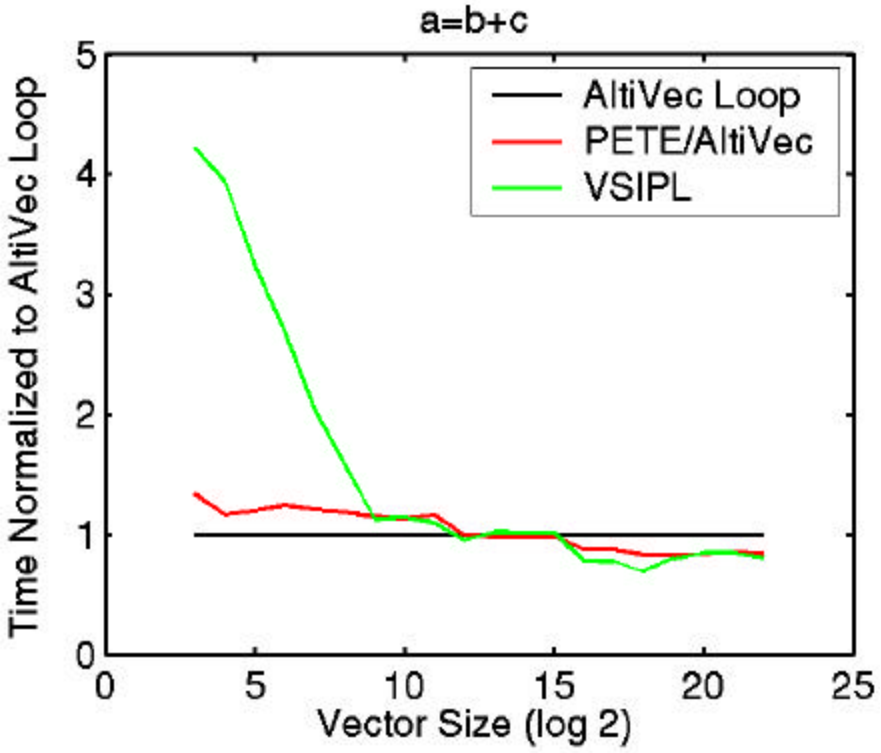
Method

- 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

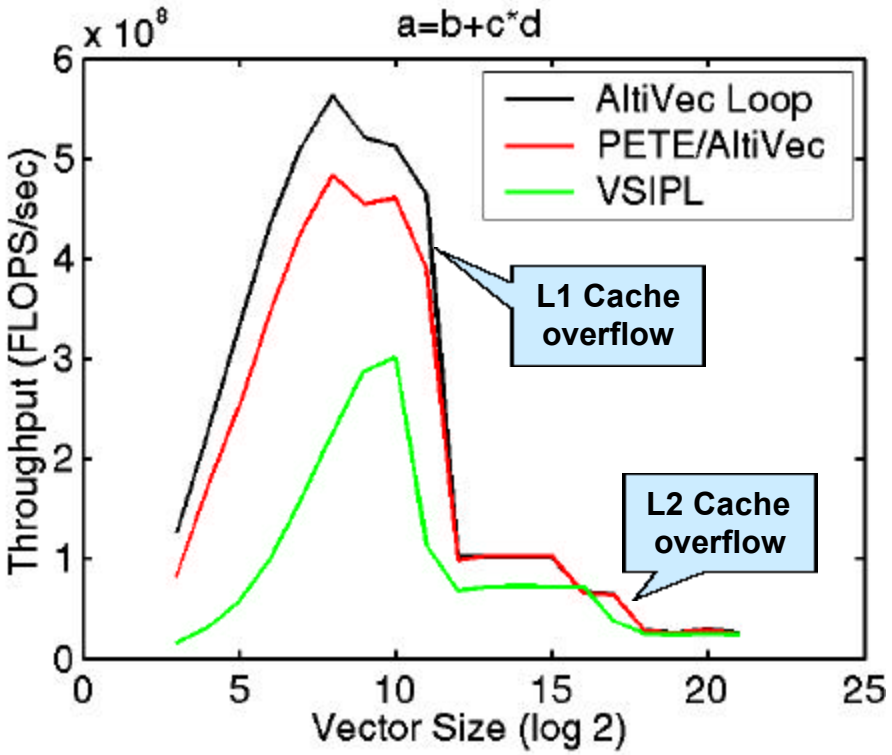
Experiment 1: A=B+C

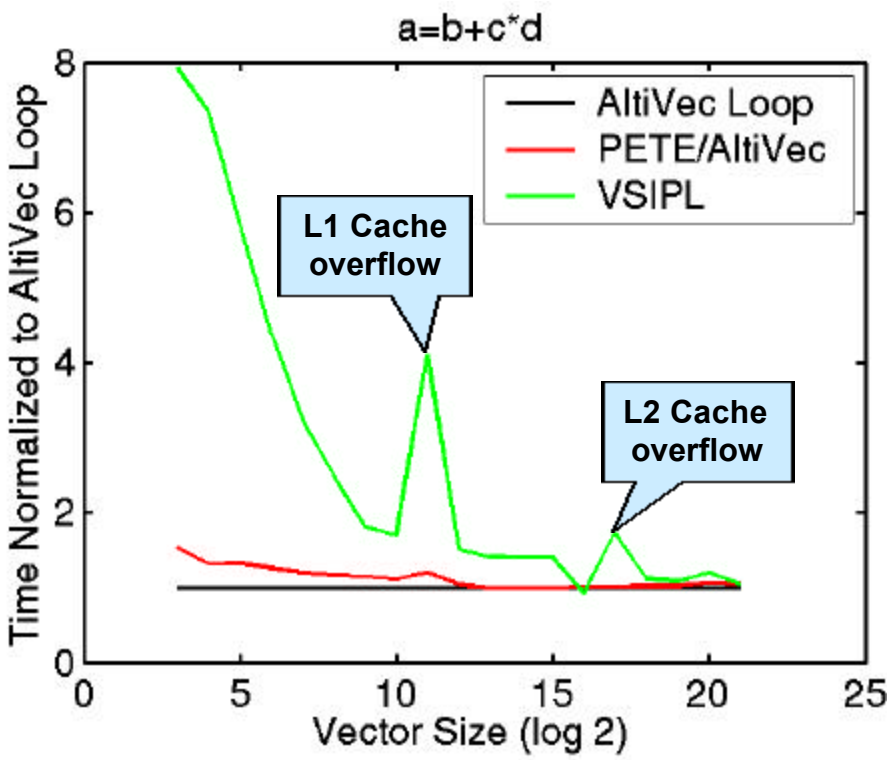




- **Peak throughput similar for all approaches**
- **VSIPL has some overhead for small data sizes**
  - **VSIPL calls cannot be inlined by the compiler**
  - **VSIPL makes no assumptions about data alignment/stride**

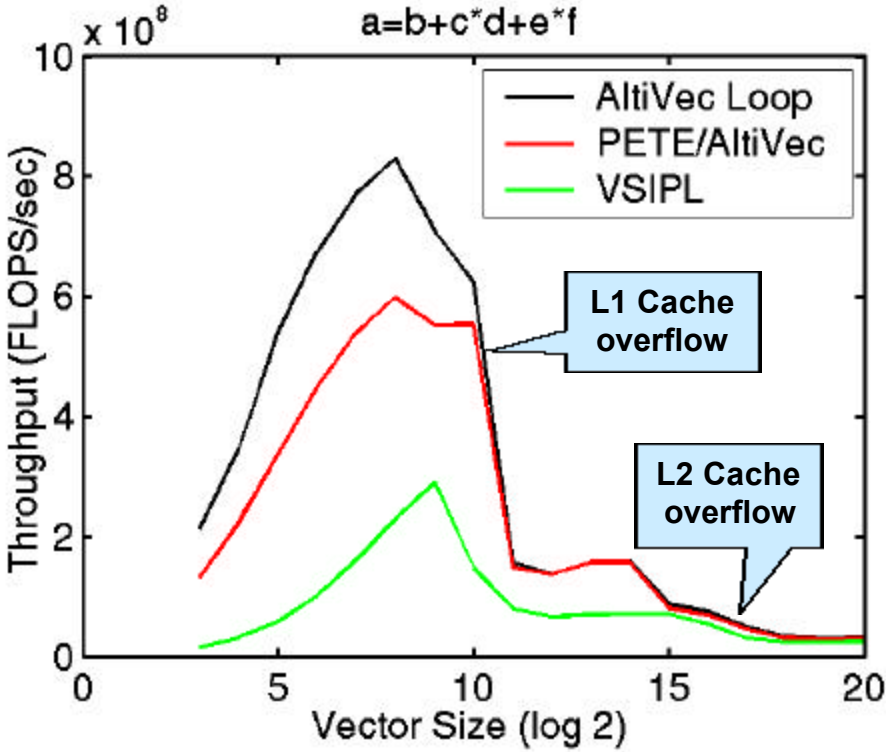
Experiment 2: A=B+C*D

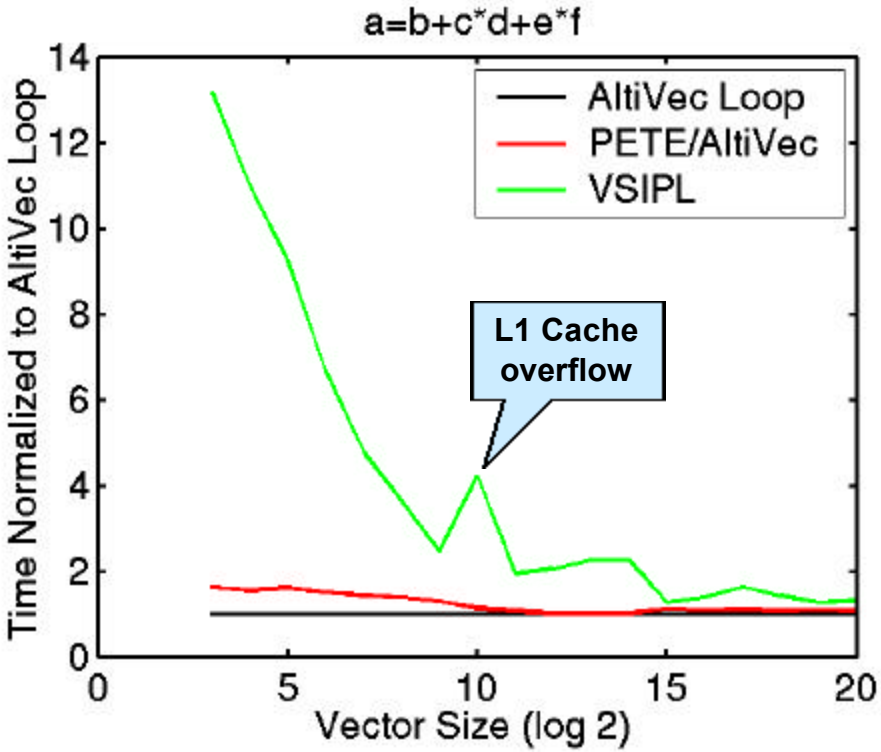




- **Loop and PETE/AltiVec both outperform VSIPL**
  - VSIPL implementation creates a temporary to hold multiply result (no multiply-add in Core Lite)
- **All approaches have similar performance for very large data sizes**
- **PETE/AltiVec adds little overhead compared to hand coded loop**

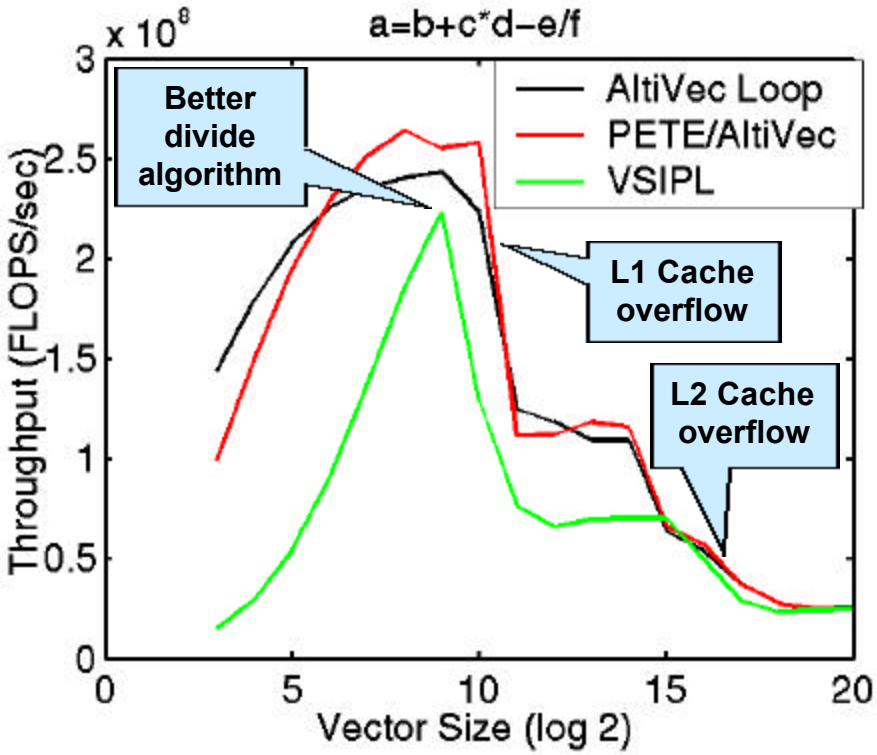
Experiment 3: A=B+C*D+E*F

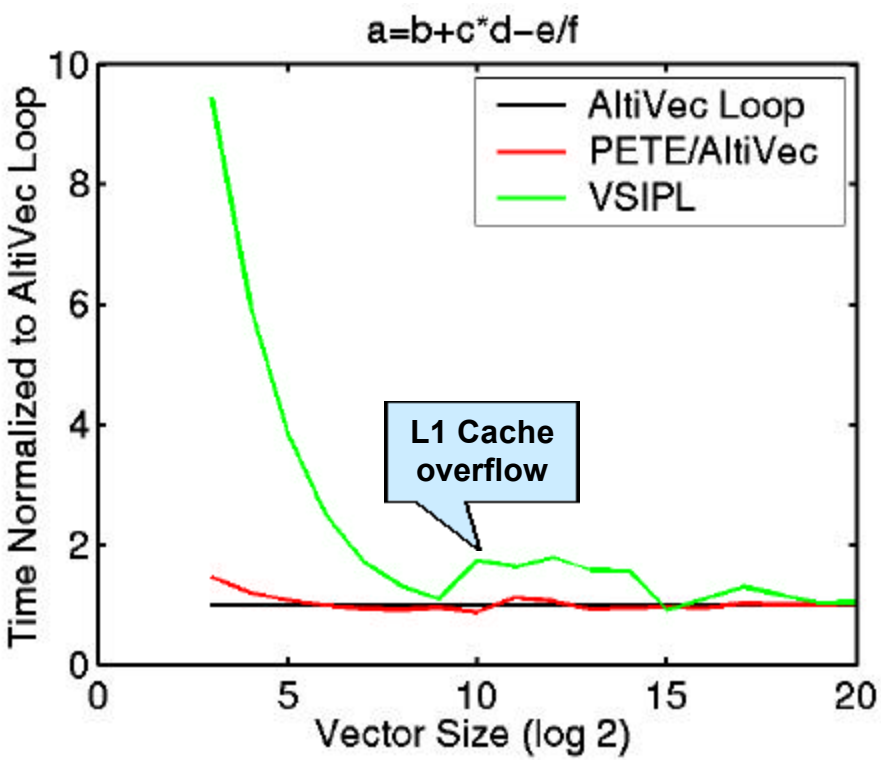




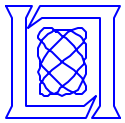
- **Loop and PETE/AltiVec both outperform VSIPL**
  - VSIPL implementation must create temporaries to hold intermediate results (no multiply-add in Core Lite)
- **All approaches have similar performance for very large data sizes**
- **PETE/AltiVec has some overhead compared to hand coded loop**

Experiment 4: A=B+C*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

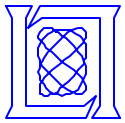
HPEC-SI[1]

VSIPL++ (Proposal)

- C++ binding
- Object oriented
- Generic

- **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