An Embedded Domain Specific Lanaguage for General Purpose Vectorization

Przemysław Karpiński (CERN, NUIM)

John McDonald (NUIM)
What to expect?

- Problem statement and our contribution
- arbitrary length vectors with SIMD execution
- EDSL design using Expression Templates
- Early performance evaluation
Problems with explicit (SIMD) vectorization

- Scientific **algorithms require vector** arithmetics, but C++ **only offers** scalars.

- Using explicit SIMD the user **has to manually handle peel and remainder loops**.

- Explicit SIMD model we presented before (UME::SIMD) still **might require full re-write** of specific codes for different architectures.

- True language extensions are **expensive to implement** and require **non-standard toolchain**.
1) Explicit vectorization gives best performance compared to auto-vectorization and directive based vectorization.

2) Expression templates (ET) already discussed as a pattern for expression based EDSL implementation.

3) Using ET for SIMD vectorization already presented, but heavily relying on template metaprogramming techniques

4) ET based linear algebra packages already exist, focusing on matrix processing with vectors being a special case.
   - Veldhuizen, T., Ponnambalam, K.: Linear algebra with C++ template metaprograms.
Contributions

1) Generalizing SIMD programming for arbitrary-length vectors:
   • Removes need for manual peeling
   • Improves portability
   • Linear algebra + array processing

2) Introducing Expression Coalescing pattern:
   • Enhances user-framework code interaction

3) Generalizing evaluation trigger:
   • Allows evaluation of more elaborate expressions (destructive, reductions, scatter)
   • Simultaneous evaluation of multiple expressions
SIMD programming for arbitrary-length vectors

```c
int LEN=19
float a[LEN], b[LEN], c[LEN], d[LEN];

for(int i = 0; i < LEN; i++) {
    d[i] = (a[i] + b[i]) * c[i];
}
```

SIMD programming for arbitrary-length vectors

```c
int LEN=19
float a[LEN], b[LEN], c[LEN], d[LEN];
SIMDVec<float, 4> a_v, b_v, c_v, d_v;

int PEEL_CNT = (LEN/4)*4;
// Peel loop
for(int i = 0; i < PEEL_CNT; i+=4) {
    a_v.load(&a[i]);
    b_v.load(&b[i]);
    c_v.load(&b[i]);
    d_v = (a_v + b_v)* c_v;
    d_v.store(&d[i]);
}
// Remainder loop
for(int i = PEEL_CNT; i < LEN; i++) {
    d[i] = (a[i] + b[i])* c[i];
}
```
int LEN=19
float a[LEN], b[LEN], c[LEN], d[LEN];
Vector<float>
a_v(LEN, a),
b_v(LEN, b),
c_v(LEN, c),
d_v(LEN, d);

auto temp1 = a_v + b_v;
auto temp2 = temp1 * c_v;
d_v = temp2
SIMD programming for arbitrary-length vectors

```c
int LEN=19
float a[LEN], b[LEN], c[LEN], d[LEN];
Vector<float>
   a_v(LEN, a),
   b_v(LEN, b),
   c_v(LEN, c),
   d_v(LEN, d);
auto temp1 = a_v + b_v;
auto temp2 = temp1 * c_v;
d_v = temp2
```

```
dcltype(temp1): ArithmeticADDExpression<Vector<float>, Vector<float>>
dcltype(temp2): ArithmeticMULExpression<
   ArithmeticADDExpression<Vector<float>, Vector<float>>,
   Vector<float>>
```
## Language overview

<table>
<thead>
<tr>
<th>Category</th>
<th>Some supported operations</th>
<th>Examples</th>
<th>Examples w/ operator syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic arithmetic</td>
<td><code>add, mul</code></td>
<td><code>C = A.add(B);</code></td>
<td><code>C = A + B;</code></td>
</tr>
<tr>
<td>Masked arithmetic</td>
<td><code>madd, msqrt</code></td>
<td><code>C = A.sqrt(mask, B);</code></td>
<td>-</td>
</tr>
<tr>
<td>Destructive arithmetic</td>
<td><code>adda, mula</code></td>
<td><code>A.adda(B);</code></td>
<td><code>A += B;</code></td>
</tr>
<tr>
<td>Horizontal arithmetic</td>
<td><code>hadd</code></td>
<td><code>c = A.hadd();</code></td>
<td>-</td>
</tr>
<tr>
<td>Arithmetic cast</td>
<td><code>ftoi, utof</code></td>
<td><code>C_u32 = A_f32.ftou();</code></td>
<td>-</td>
</tr>
<tr>
<td>Basic logical</td>
<td><code>land, lor</code></td>
<td><code>M_C = M_A.land(M_B);</code></td>
<td><code>M_C = M_A &amp;&amp; M_B;</code></td>
</tr>
<tr>
<td>Logical comparison</td>
<td><code>cmpeq</code></td>
<td><code>M = A.cmpeq(B);</code></td>
<td><code>M = A == B;</code></td>
</tr>
<tr>
<td>Gather/scatter</td>
<td><code>gather, scatter</code></td>
<td><code>A = B.gather(C);</code></td>
<td>-</td>
</tr>
</tbody>
</table>

### Additional rules:
- Strong typing required.
- Control flow using logical masks.
- Operations only return DAG*, computation is deferred.
- Complex DAG’s require special evaluation schemes.

*Directed Acyclic Graph*
Expression coalescing

Listing 1.12: Static Expression Coalescence pattern merges user function written using Vector EDSL with framework-defined solver.

```cpp
template<typename USER_FUNC_T>
void rk4.framework_solver(auto & result, auto x, auto y, float dx, USER_FUNC_T& func) {
  float halfdx = dx * 0.5f;
  auto k1 = dx * func(x, y);
  auto k2 = dx * func(x + halfdx, y + k1 * halfdx);
  auto k3 = dx * func(x + halfdx, y + k2 * halfdx);
  auto k4 = dx * func(x + dx, y + k3 * dx);
  result = y + (1.0f / 6.0f) * (k1 + 2.0f * k2 + 2.0f * k3 + k4); // Evaluation starts with this statement
}

// User defined function has to be defined using the same Vector EDSL dialect.
auto userFunction = [](auto X, auto Y) {
  return X.sin() * Y.exp();
};

// User passes her function to solver
rk4.framework_solver(result_vec, x_exp, y_exp, timestep, userFunction);
...```

**Generic solver requires user-defined function**

**User defined function has no loops and no explicit SIMD**

**Generic solver is specialized depending on expression represented by the user function!**
Expression coalescing

```cpp
template<typename T_X, typename T_Y, typename T_DX, typename USER_FUNC_T>
auto rk4_framework_solver(T_X x_exp, T_Y y_exp, T_DX dx, USER_FUNC_T& func) {
    auto halfdx=dx*0.5f;
    auto k1=dx*func(x, y);
    auto k2=dx*func(x+halfdx, y+k1*halfdx);
    auto k3=dx*func(x+halfdx, y+k2*halfdx);
    auto k4=dx*func(x+dx, y+k3*dx);
    auto result = y+(1.0f/6.0f)*(k1+2.0f*k2+2.0f*k3+k4);
    return result; // No evaluation even at this point!
}

auto userFunction=[](auto X, auto Y) {
    return X.sin()*Y.exp();
}

auto solver_exp = rk4_framework_solver(x_exp, y_exp, timestep, userFunction);
result=solver_exp; // User triggers the evaluation when values needed
```
Expression coalescing

userFunction

For every element of X and Y calculate result as...
Expression coalescing

userFunction

rk4_framework_solver

userFunction

rk4_framework_solver
Evaluation

Listing 1.13: Default evaluator uses very straightforward evaluation scheme. Instead of traversing a vector in the data direction (horizontally), depth-first (vertical) traversal of the full expression is performed. The `elements` pointer refers to the memory location represented by an instance of 'FloatVector' type.

```
template<typename E>
UME_FORCE_INLINE FloatVector<SCALAR_TYPE>& operator= (ArithmeticExpression<SCALAR_TYPE,E>& vec)
{
  E & reinterpret_vec = static_cast<E&>(vec);

  // SIMD_STRIDE — a target specific library macro
  for(int i=0;i<LOOP_PEEL_OFFSET();i+=SIMD_STRIDE)
  {
    auto t0 = reinterpret_vec.evaluate_SIMD<SIMD_STRIDE>(i);
    t0.store(&this->elements[i]); // t0 needs to be a type respecting UME::SIMD interface.
  }
  for(int i=LOOP_PEEL_OFFSET();i<maxLength;i++)
  {
    auto t1 = reinterpret_vec.evaluate_SIMD<1>(i); // Evaluate remainder part using SIMD-1 (scalar) mode.
    t1.store(&this->elements[i]);
  }
  return *this;
}
```

Listing 1.14: Evaluation method can use a depth-first approach to calculate dependencies.

```
template<int SIMD_STRIDE>
UME_FORCE_INLINE SIMDVec<SCALAR_T,SIMD_STRIDE> evaluate_SIMD(int index)
{
  SIMDVec<SCALAR_T,SIMD_STRIDE> t0 = o1.evaluate_SIMD(index); // Evaluate subexpressions
  SIMDVec<SCALAR_T,SIMD_STRIDE> t1 = o2.evaluate_SIMD(index);
  return t0.add(t1); // Evaluate current expression node
}
```
Expression divergence

Listing 1.16: Expression divergence happens when two expressions share a common sub-expression. This problem can cause memory locality issues, but can be solved with a very simple evaluation scheme.

```cpp
auto t0 = A + B;
auto t1 = C + D;
auto t2 = 6 * t1;
Ret = F; Get = H;
```

Listing 1.17: Dyadic evaluator calculates both expressions before updating destination values. This way data hazards are avoided.

```cpp
class DyadicEvaluator {
public:
  // Evaluate a pair of expressions simultaneously
  template<typename SCALAR_T_1, typename DST_T_1, typename EXP_T_1,
    typename SCALAR_T_2, typename DST_T_2, typename EXP_T_2>
  DyadicEvaluator( 
    DST_T_1& dst1, ArithmeticExpression<SCALAR_T_1, EXP_T_1>& exp1,
    DST_T_2& dst2, ArithmeticExpression<SCALAR_T_2, EXP_T_2>& exp2)
  {
    EXP_T_1& r_exp1 = static_cast<EXP_T_1&>(exp1);
    EXP_T_2& r_exp2 = static_cast<EXP_T_2&>(exp2);
    for (int i = 0; i < dst1LOOP_PEEL_OFFSET(), i = SIMD_STRIDE() {
      auto t1 = r_exp1.evaluate_SIMD<simd::Stride>(i1);
      dst1.update_SIMD(t1, i1);
    }
    for (int i = dst1_LOOP_PEEL_OFFSET(); i < dst1.LENGTH(); i++) {
      auto t1 = r_exp1.evaluate_SIMD<simd::Stride>(i1);
      auto t2 = r_exp2.evaluate_SIMD<simd::Stride>(i1);
      dst1.update_simd(t1, i1);
      dst2.update_simd(t2, i1);
    }
  }

  auto t0 = A + B;
  auto t1 = C + D;
  auto t2 = 6 * t1;
  DyadicEvaluator eval(E + F, C + H); // Evaluation trigger
```
### Generalized evaluators

#### Monadic evaluators

<table>
<thead>
<tr>
<th>Mapping class</th>
<th>Provisional name</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression -&gt; vector</td>
<td>Assignment</td>
<td>Same as operator=</td>
</tr>
<tr>
<td>expression -&gt; scalar</td>
<td>Reduction</td>
<td>The last operation in graph is a reduction</td>
</tr>
<tr>
<td>expression -&gt; -</td>
<td>Destructive</td>
<td>Operation has only an implicit destination (e.g. operator+=)</td>
</tr>
<tr>
<td>(expression, indices) -&gt; vector</td>
<td>Scatter</td>
<td>Last operation scatters the result.</td>
</tr>
</tbody>
</table>

#### Dyadic evaluators

<table>
<thead>
<tr>
<th>Mapping class</th>
<th>Provisional name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(expression, expression) -&gt; (vector, vector)</td>
<td>Assignment-assignment</td>
</tr>
<tr>
<td>(expression, expression) -&gt; (vector, scalar)</td>
<td>Assignment-reduction</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(expression, indices, expression, indices) -&gt; (vector, vector)</td>
<td>Scatter-scatter</td>
</tr>
</tbody>
</table>

#### Triadic evaluators

<table>
<thead>
<tr>
<th>Mapping class</th>
<th>Provisional name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(expression, expression, expression) -&gt; (vector, vector, vector)</td>
<td>Assignment-assignment-assignment</td>
</tr>
<tr>
<td>(expression, expression, expression) -&gt; (vector, vector, scalar)</td>
<td>Assignment-assignment-reduction</td>
</tr>
<tr>
<td>(expression, indices, expression, indices, expression, indices) -&gt; (vector, vector, vector)</td>
<td>Scatter-Scatter-Scatter</td>
</tr>
</tbody>
</table>

---

**Polyadic evaluators?**

???
Performance comparison*

\[ y[i] = a \times x[i] + y[i]; \]

\[ \text{cblas_saxpy}(N, a, x, 1, y, 1); \]

*All measurements with Intel Xeon E3-1280v3, 16GB of DDRAM, running SLC6 operating system*
Performance comparison

BLAS AXPY chained (32b precision)

BLAS AXPY chained (64b precision)
Performance comparison

\[
c\text{blas\_srot}(N, a, 1, b, 1, c, s);
\]

\[
x(t) = c\times x(t-1) + s\times y(t-1);
\]

\[
y(t) = c\times y(t-1) - s\times x(t-1);
\]
Table 1: Speedup of different implementations of RK4 solver vs. reference. Values given for single/double precision. Only Clang gives comparable results with auto-vectorization. Highest performance obtained with explicit SIMD and EDSL in all cases.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>1</th>
<th>10</th>
<th>10²</th>
<th>10³</th>
<th>10⁴</th>
<th>10⁵</th>
<th>10⁶</th>
<th>10⁷</th>
<th>Geomean</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC 5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
<td>1.00/1.00</td>
</tr>
<tr>
<td>UME::SIMD</td>
<td>0.97/0.98</td>
<td>2.66/2.43</td>
<td>5.72/3.72</td>
<td>7.33/3.09</td>
<td>6.39/3.26</td>
<td>6.13/3.17</td>
<td>6.12/2.68</td>
<td>5.78/2.99</td>
<td>4.43/2.63</td>
</tr>
<tr>
<td>UME::VECTOR</td>
<td>0.96/0.98</td>
<td>2.75/3.50</td>
<td>6.37/4.47</td>
<td>7.39/3.44</td>
<td>7.38/3.76</td>
<td>7.27/3.61</td>
<td>7.01/2.93</td>
<td>6.43/3.34</td>
<td>4.84/3.02</td>
</tr>
<tr>
<td>ICPC 17.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>1.29/1.78</td>
<td>1.58/1.98</td>
<td>1.58/1.53</td>
<td>1.58/1.77</td>
<td>1.65/1.76</td>
<td>1.65/1.76</td>
<td>1.66/1.90</td>
<td>1.63/1.60</td>
<td>1.57/1.75</td>
</tr>
<tr>
<td>UME::SIMD</td>
<td>1.17/1.60</td>
<td>2.85/3.61</td>
<td>8.27/5.74</td>
<td>10.89/4.67</td>
<td>8.85/5.04</td>
<td>8.73/4.80</td>
<td>8.68/3.84</td>
<td>7.07/3.69</td>
<td>5.88/3.9</td>
</tr>
<tr>
<td>UME::VECTOR</td>
<td>1.56/2.16</td>
<td>2.70/4.17</td>
<td>8.21/6.11</td>
<td>9.56/4.66</td>
<td>8.95/4.94</td>
<td>8.77/4.72</td>
<td>8.66/3.66</td>
<td>6.87/3.60</td>
<td>5.94/4</td>
</tr>
<tr>
<td>Clang 3.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>1.01/1.21</td>
<td>2.66/2.54</td>
<td>6.01/2.66</td>
<td>7.40/1.92</td>
<td>6.83/3.20</td>
<td>6.76/3.25</td>
<td>6.55/3.78</td>
<td>6.18/3.32</td>
<td>4.66/2.6</td>
</tr>
<tr>
<td>UME::SIMD</td>
<td>0.97/1.19</td>
<td>2.69/2.63</td>
<td>6.06/4.40</td>
<td>7.18/2.58</td>
<td>6.88/3.15</td>
<td>6.68/3.22</td>
<td>6.53/3.52</td>
<td>5.89/3.06</td>
<td>4.6/2.8</td>
</tr>
<tr>
<td>UME::VECTOR</td>
<td>0.98/1.15</td>
<td>2.76/3.15</td>
<td>6.13/4.41</td>
<td>7.21/2.88</td>
<td>7.15/3.20</td>
<td>6.90/3.29</td>
<td>6.70/3.28</td>
<td>5.86/3.09</td>
<td>4.68/2.89</td>
</tr>
</tbody>
</table>
Conclusions

• Implementation cost:
  • C++ 11/14/17 greatly improve ET applicability
  • EDSL can build upon existing compiler technology
  • Using code-generator to generate templates cuts the development costs significantly (and reduces compilation time)

• Portability:
  • The code can be ported ‘easily’ by providing target-specific evaluators (separate interface & implementation!)
  • Memory management left to the user (user allocates manually or passes a custom allocator)

• Performance:
  • Avoids building large temporaries
  • DAG built at compile time – no additional runtime overhead
  • Compiler helps with register management, including value re-use
  • Extensive inlining removes recursion costs

• Programmability:
  • Easier to use than explicit SIMD, more readable than ‘flat’ interfaces
  • Allows more flexible communication between user and framework codes
  • ET’s are still difficult to debug
Future directions

• Large temporaries should be created and managed by EDSL for complex expressions.

• Extension to handle matrices: requires JIT code generation.

• Some problems with compilers (e.g. ‘forceinline’, copy-elision).

• Parallel/distributed evaluators.

• CUDA support: custom evaluators implementation needed.