An Embedded Domain Specific Lanaguage for General Purpose Vectorization

Przemysław Karpiński (CERN, NUIM)

John McDonald (NUIM)

Performance Portable Programming Models for Accelerators (P^3MA)
Frankfurt, Germany, June 22, 2017

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**.

Prior art

1) Explicit vectorization gives best performance compared to auto-vectorization and directive based vectorization.

- Pohl, A., Cosenza, B., Mesa, M., Chi, C., Juurlink, B.: *An Evaluation of Current SIMD Programming Models for C++*.
- Karpiński, P., McDonald, J.: *A high-performance portable abstract interface for explicit SIMD vectorization.*

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

- Vandevoorde, D., Josuttis, N.: *C++ templates: The Complete Guide.*
- Hardtlein, J., Paum, C., Linke, A., Wolters, C. H.: *Advanced expression templates programming*.

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

- Falcou, J., Seerotz, J., Peche, L., Lapreste, J.: *Meta-programming Applied to Automatic SIMD Parallelization of Linear Algebra Code*.
- Niebler, E.: *Proto: A compiler Construction Toolkit for DSELs*.

4) ET based linear algebra packages already exist, focusing on matrix processing with vectors being a special case.

- Gunnabaus, G., Jacob, B., et al. *Eigen v3*.
- Veldhuizen, T., Ponnambalam, K.: *Linear algebra with C++ template metaprograms*.

Contributions

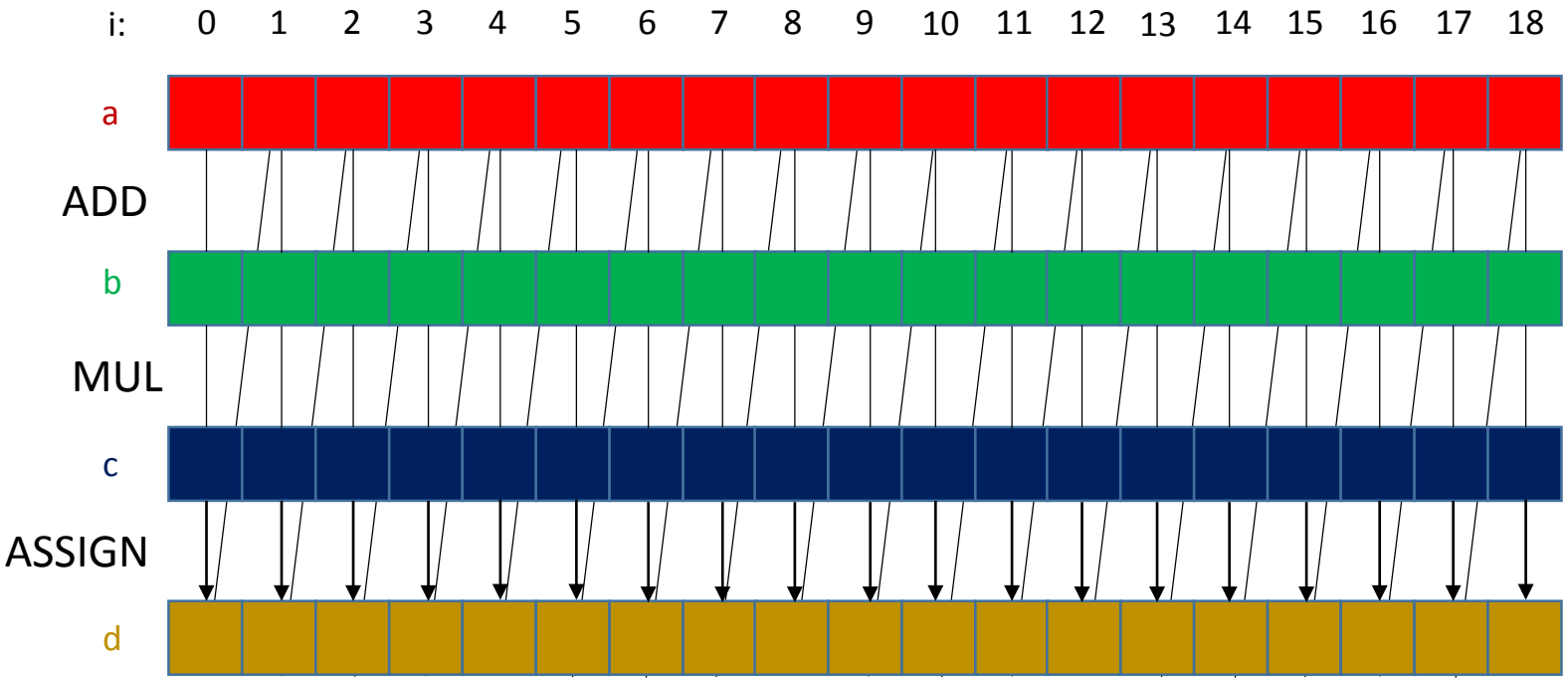
1) Generalizing SIMD programming for **arbitrary-length vectors**:
   - Removes need for manual peeling
   - Improves portability
   - Linear algebra + array processing

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

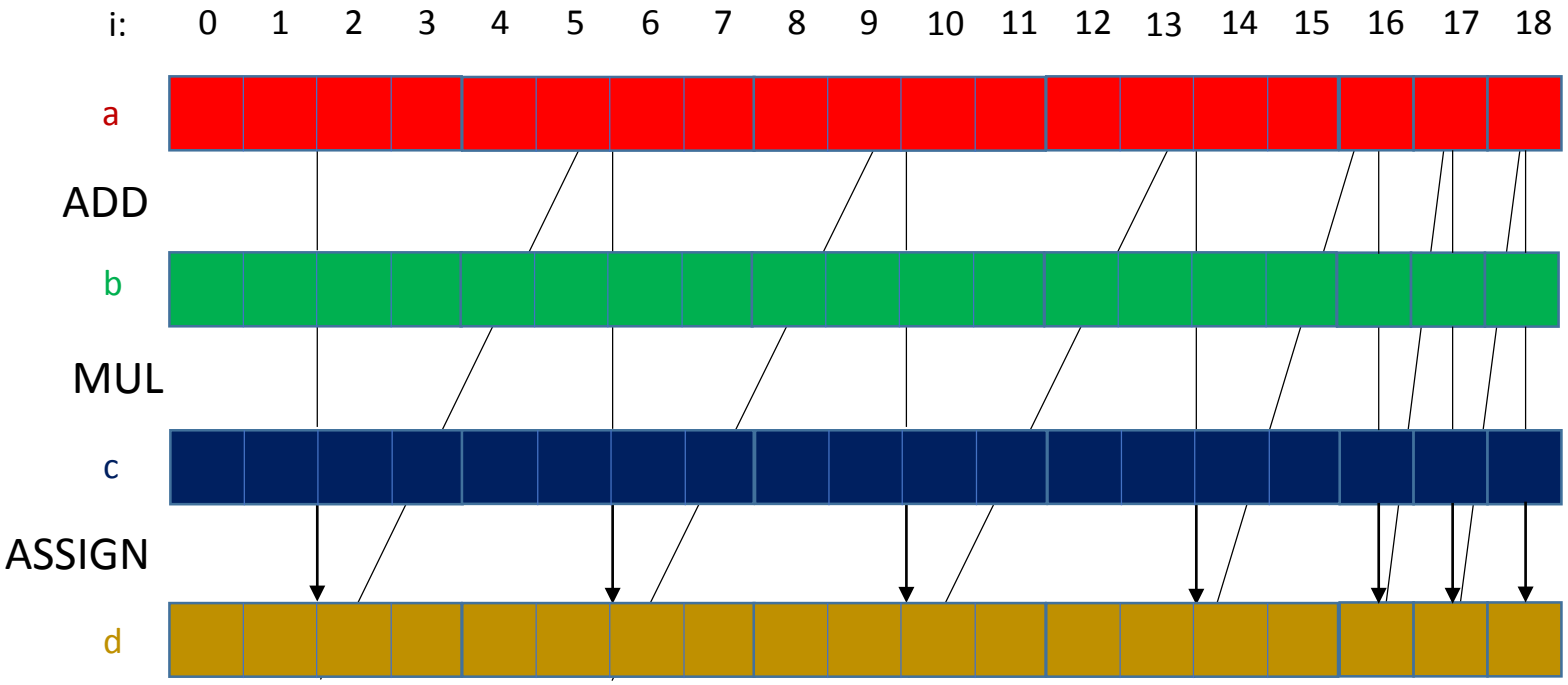
SIMD programming for **arbitrary-length vectors**

```
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**

```
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];
}
```

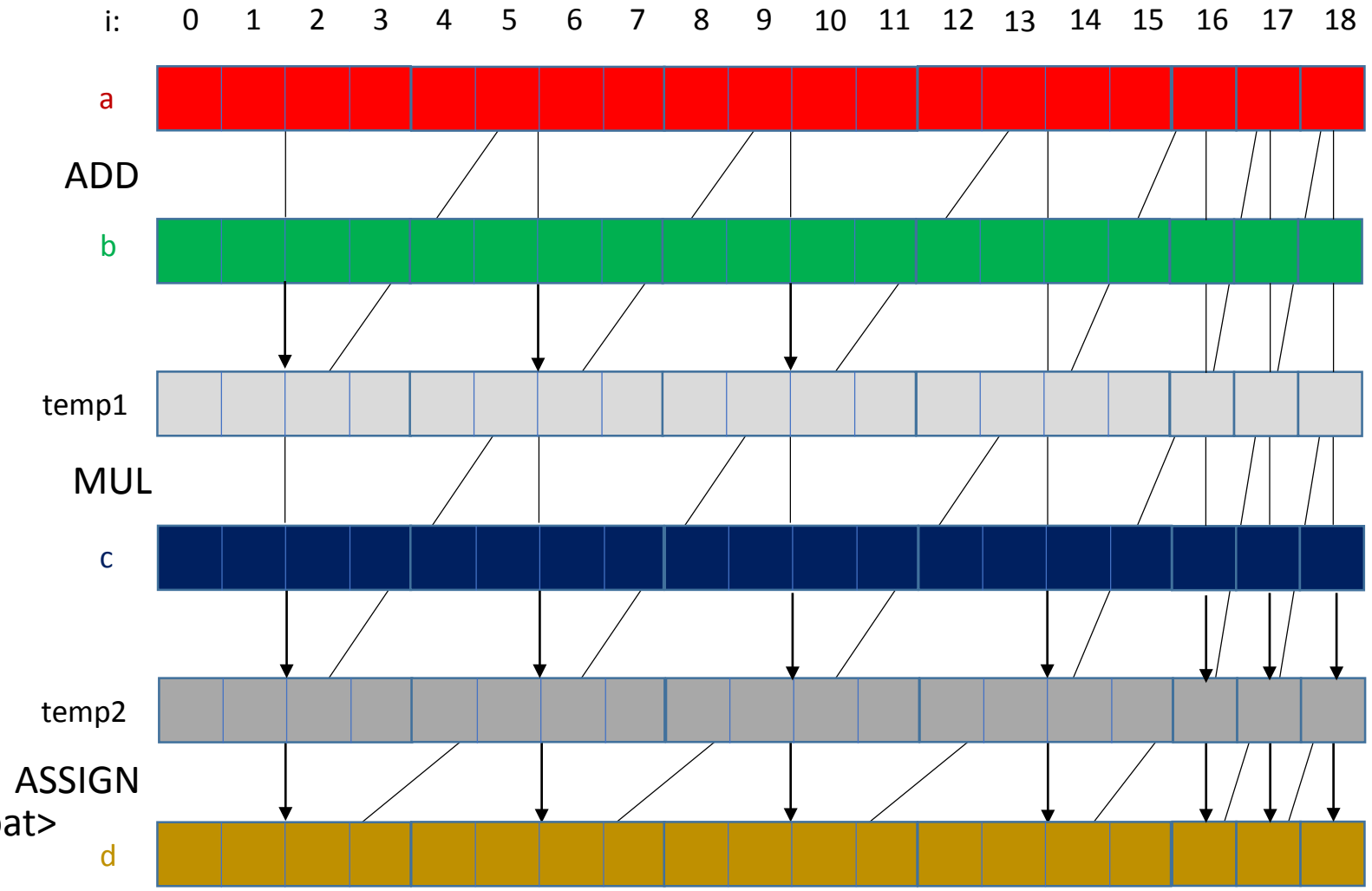


SIMD programming for **arbitrary-length vectors**

```
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
```

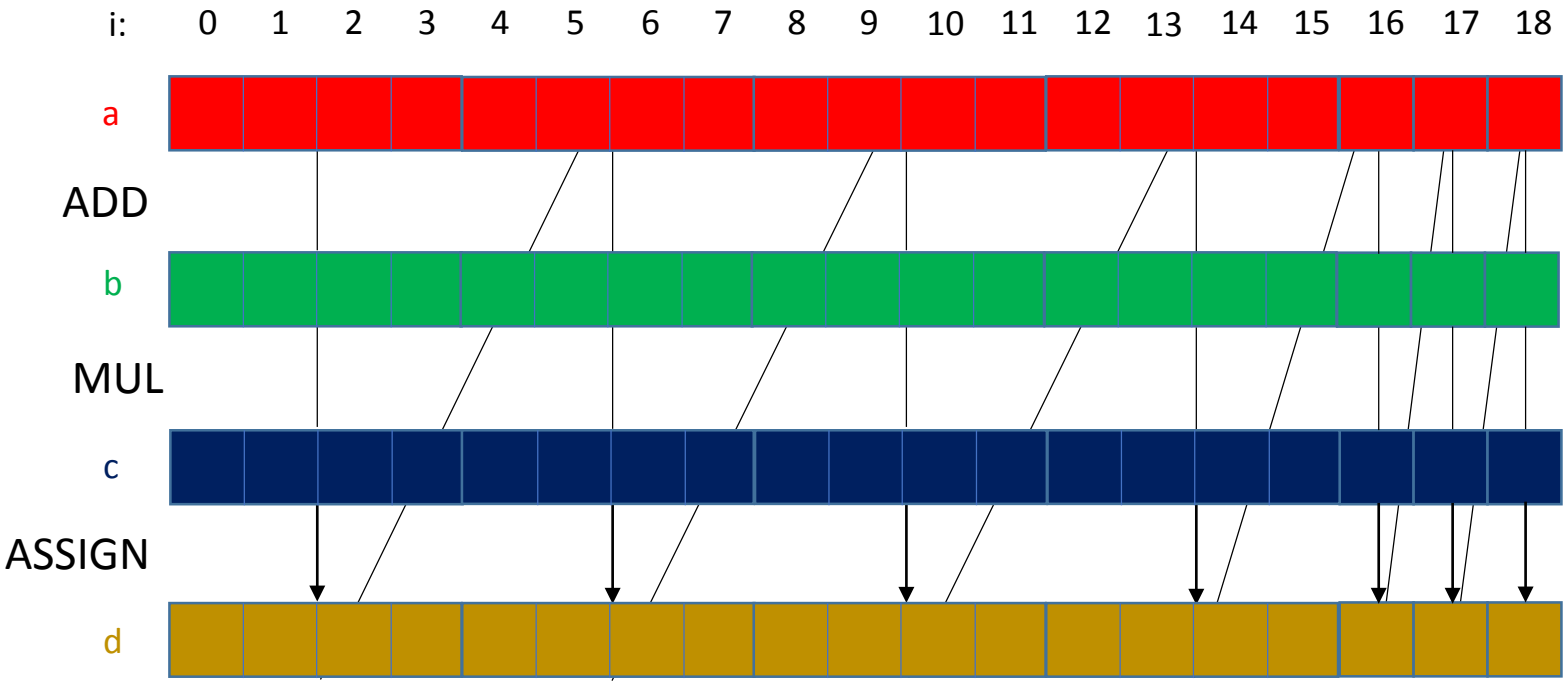
decltype(temp1): Vector<float>

decltype(temp2): Vector<float>



SIMD programming for **arbitrary-length vectors**

```
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
```



decltype(temp1): ArithmeticADDExpression<Vector<float>, Vector<float>>

decltype(temp2):
ArithmeticMULExpression<
ArithmeticADDExpression<Vector<float>, Vector<float>>,
Vector<float>>

Language overview

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

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.

```
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 C++14

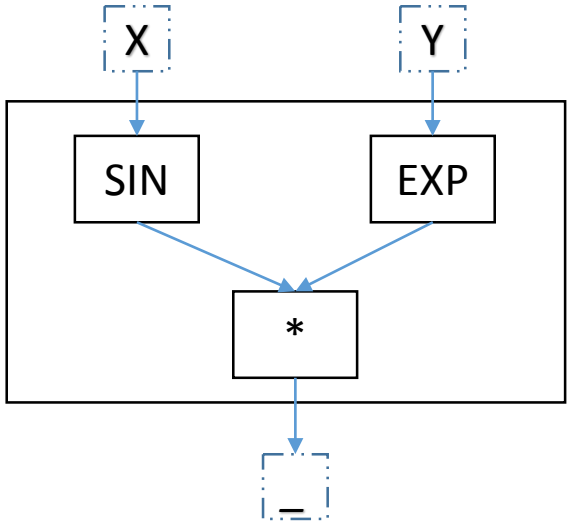
```
template<typename T_X, typename T_Y, typename T_DX, typename USER_FUNC_T>
auto rk4_framework_solver(T_X x_exp, T_Y y_exp, T_DX dx, USER_FUNC_T& func) {
    auto halfdx=dx*0.5f;
    auto k1=dx*func(x, y);
    auto k2=dx*func(x+halfdx, y+k1*halfdx);
    auto k3=dx*func(x+halfdx, y+k2*halfdx);
    auto 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_sover(x_exp, y_exp, timestep, userFunction);

result=solver_exp; // User triggers the evaluation when values needed
```

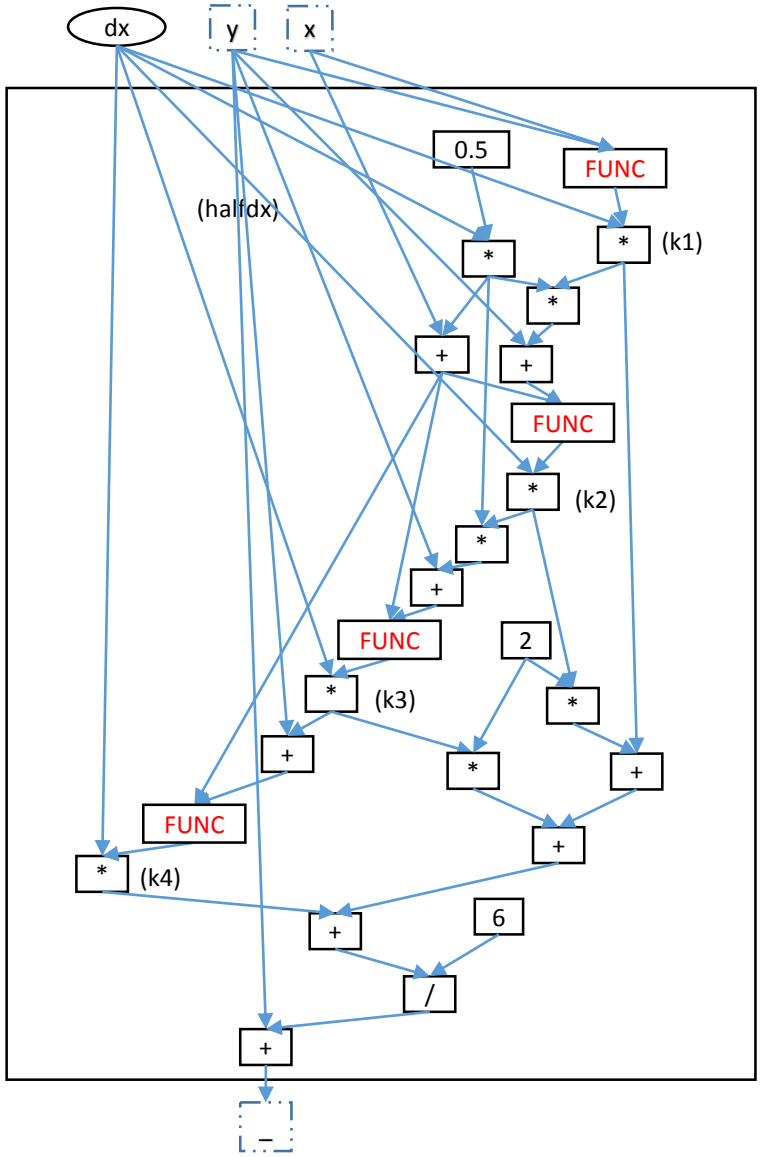
Expression coalescing

userFunction

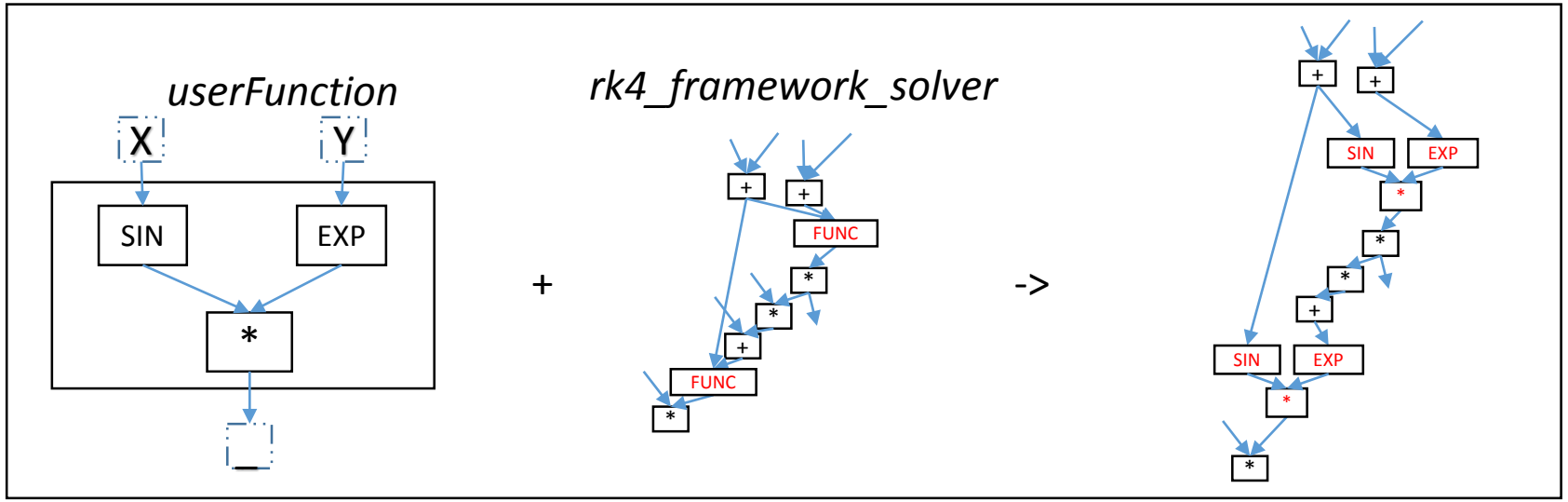


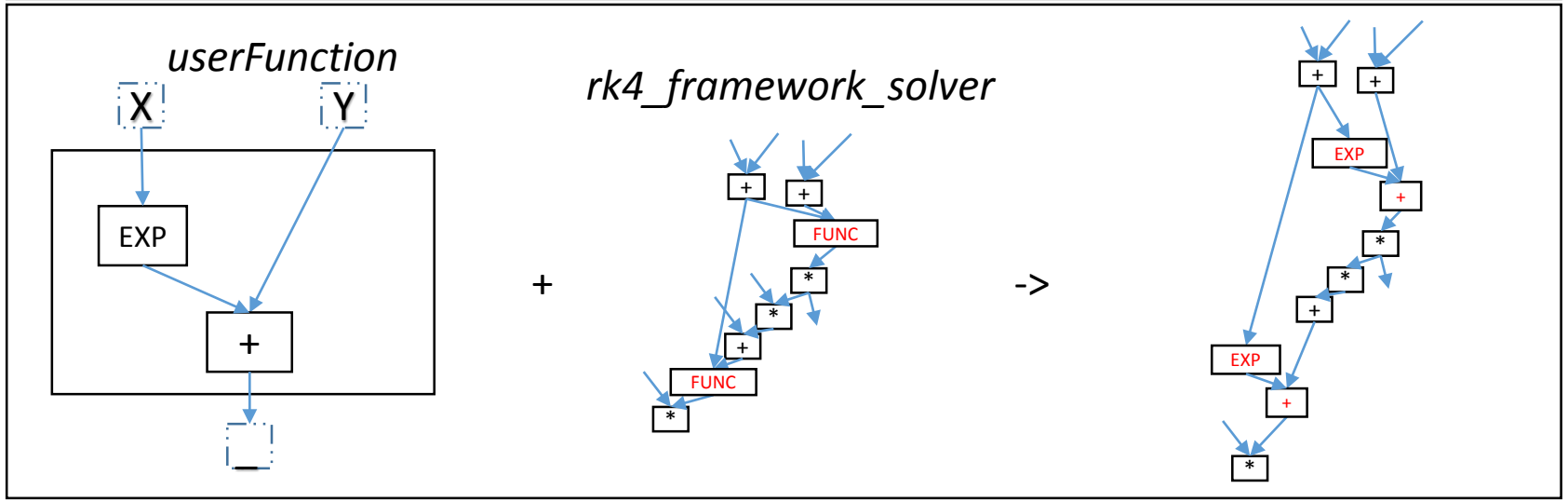
For every element of X and Y calculate result as…

rk4_framework_solver



Expression coalescing





Evaluation

```
result=y+(1.0f/6.0f)*(k1+2.0f*k2+2.0f*k3+k4);
```

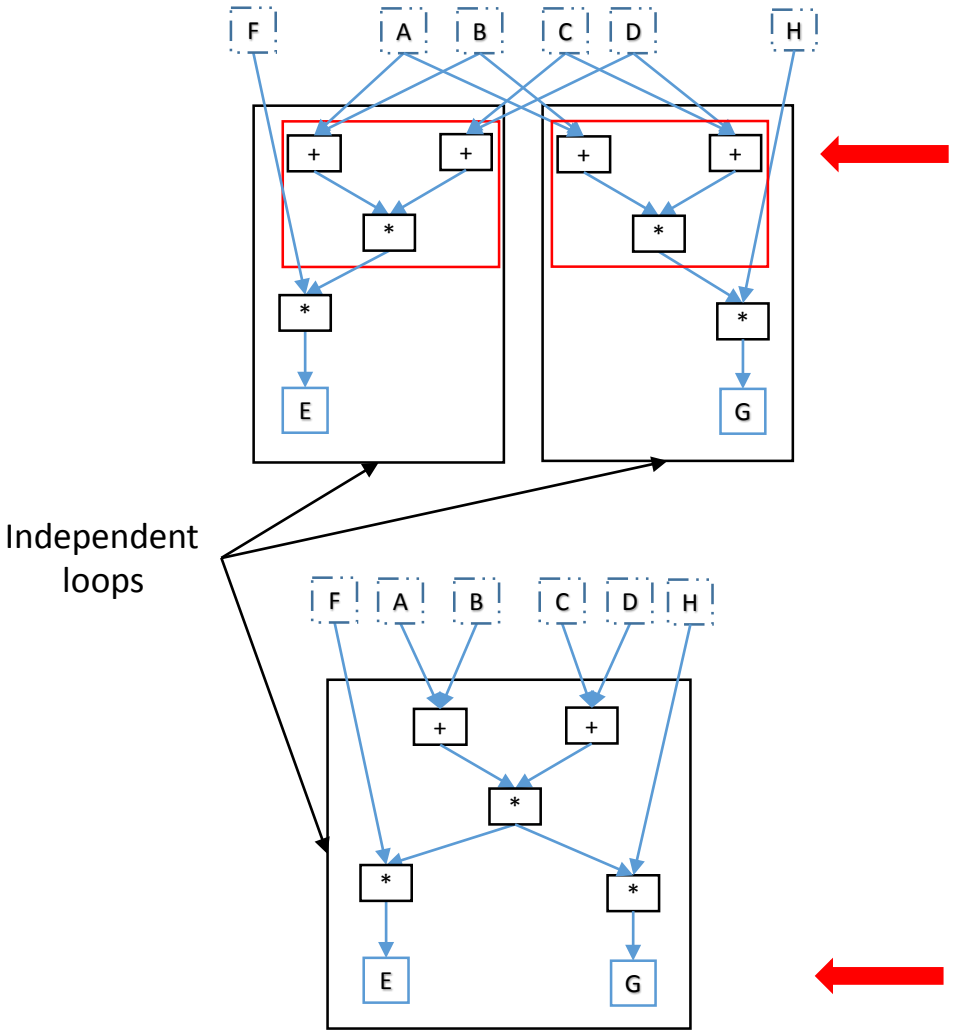
Listing 1.13: Default evaluator uses very straightforward evaluation scheme. Instead of traversing a vector in the data direction (horizontally), depth-first (vertical) traversal of the full expression is performed. The 'elements' pointer refers to the memory location represented by an instance of 'FloatVector' 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<mLength;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=_e1.evaluate_SIMD(index); // Evaluate subexpressions
  SIMDVec<SCALAR_T,SIMD_STRIDE> t1=_e2.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.

```
auto t0=A+B;
auto t1=C+D;
auto t2=t0*t1;
E=t2*F;
G=t2*H;
```

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

```
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<dst1.LOOP_PEEL_OFFSET();i+=SIMD_STRIDE){
      auto t0= r_exp1.evaluate_SIMD<SIMD_STRIDE>(i); // evaluate multiple results at a time
      auto t1= r_exp2.evaluate_SIMD<SIMD_STRIDE>(i);
      dst1.update_SIMD(t0,i);
      dst2.update_SIMD(t1,i);
    }
    for(int i=dst1.LOOP_PEEL_OFFSET();i<dst1.LENGTH();i++){
      auto t0= r_exp1.evaluate_SIMD<1>(i); // evaluate single result at a time
      auto t1= r_exp2.evaluate_SIMD<1>(i);
      dst1.update_scalar(t0,i);
      dst2.update_scalar(t1,i);
    }
  }
};
...
auto t0=A+B;
auto t1=C+D;
auto t2=t0*t1;

DyadicEvaluator eval(E,t2*F,G,t2*H); // Evaluation trigger
```

Generalized evaluators

Monadic evaluators		
Mapping class	**Provisional name**	**Behavioiur**
expression -> vector	Assignment	Same as operator=
expression -> scalar	Reduction	The last operation in graph is a reduction
expression -> -	Destructive	Operation has only an implicit destination (e.g. operator+=)
(expression, indices) -> vector	Scatter	Last operation scatters the result.

Polyadic evaluators?
???

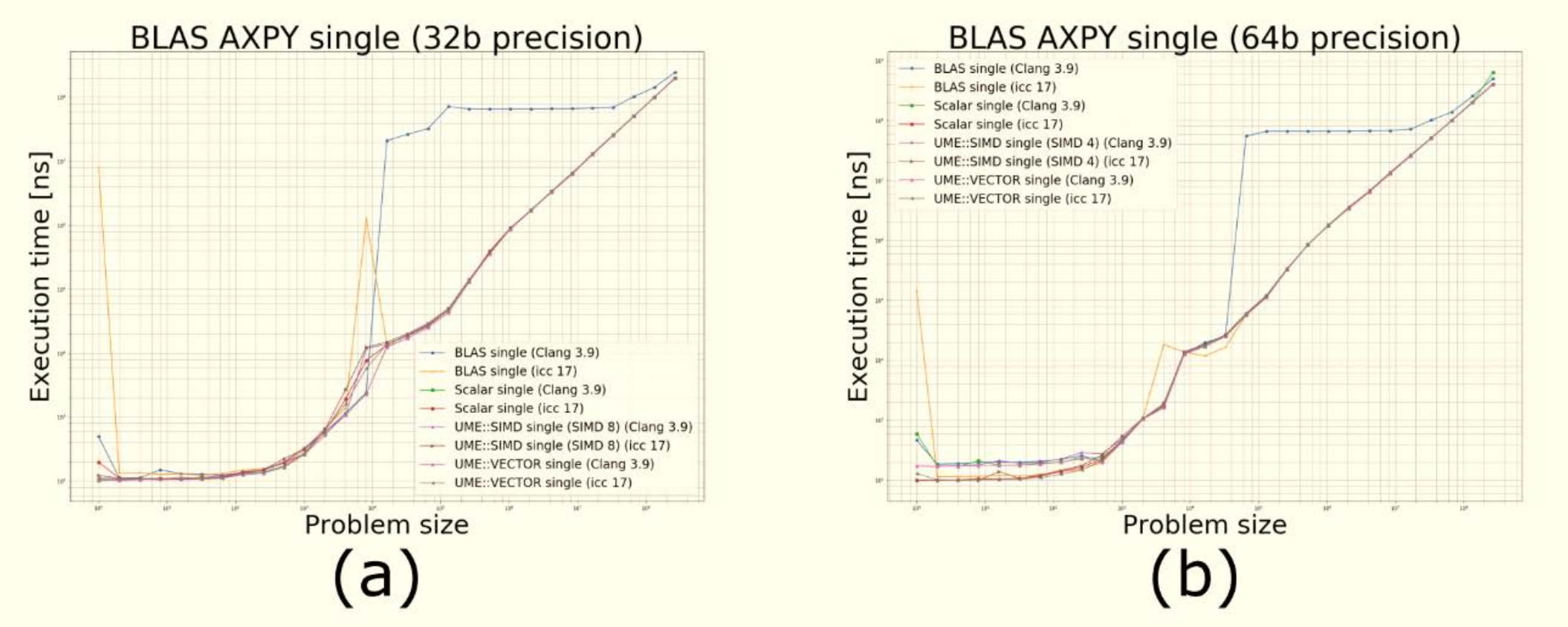
Dyadic evaluators	
Mapping class	**Provisional name**
(expression, expression) -> (vector, vector)	Assignment-assignment
(expression, expression) -> (vector, scalar)	Assignment-reduction
...	
(expression, indices, expression, indices) -> (vector, vector)	Scatter-scatter

Triadic evaluators	
Mapping class	**Provisional name**
(expression, expression, expression) -> (vector, vector, vector)	Assignment-assignment-assignment
(expression, expression, expression) -> (vector, vector, scalar)	Assignment-assignment-reduction
...	
(expression, indices, expression, indices, expression, indices) -> (vector, vector, vector)	Scatter-Scatter-Scatter

Performance comparison*

```
cblas_saxpy(N, a, x, 1, y, 1);
```

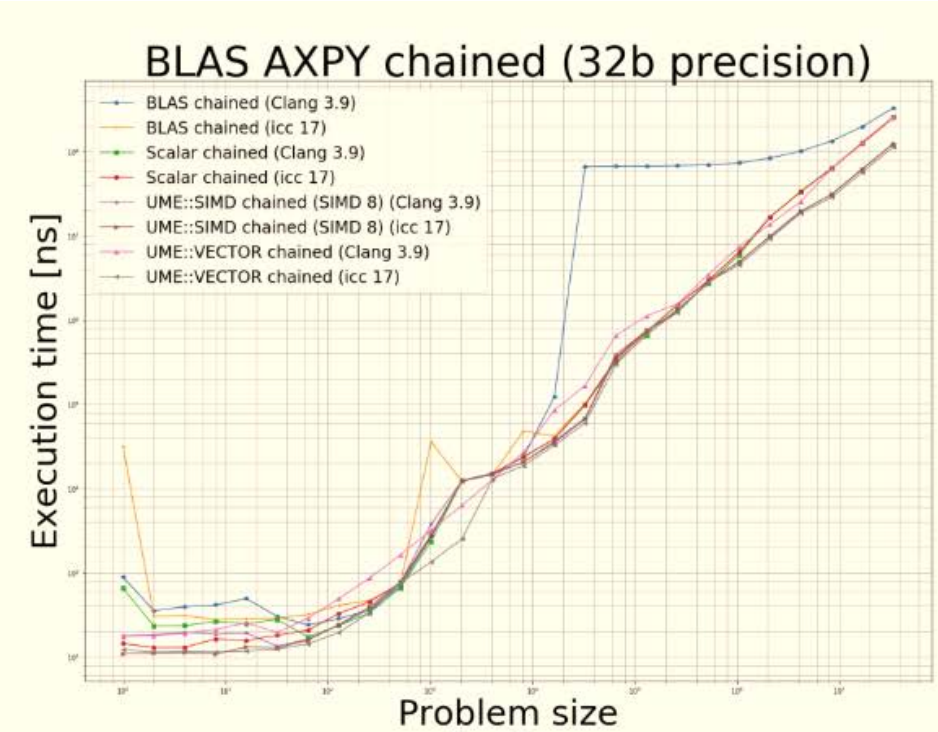
y[i] = a*x[i] + y[i];



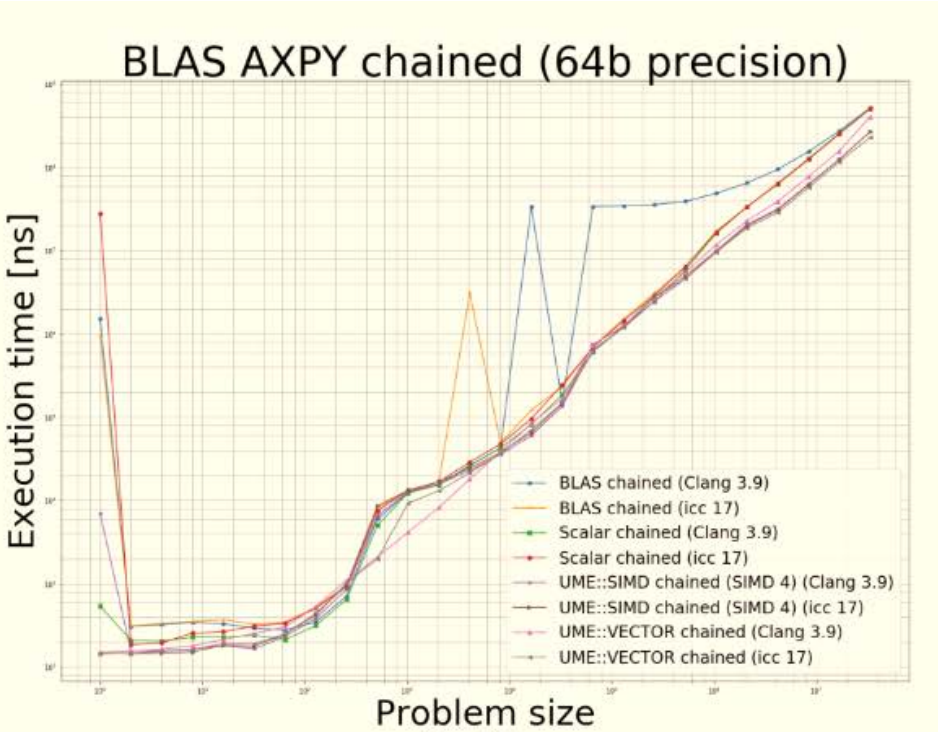
* All measurements with Intel Xeon E3-1280v3, 16GB of DDRAM, running SLC6 operating system

Performance comparison

```
cblas_saxpy(N, a[0], x0, 1, y, 1);
cblas_saxpy(N, a[1], x1, 1, y, 1);
cblas_saxpy(N, a[2], x2, 1, y, 1);
cblas_saxpy(N, a[3], x3, 1, y, 1);
cblas_saxpy(N, a[4], x4, 1, y, 1);
cblas_saxpy(N, a[5], x5, 1, y, 1);
cblas_saxpy(N, a[6], x6, 1, y, 1);
cblas_saxpy(N, a[7], x7, 1, y, 1);
cblas_saxpy(N, a[8], x8, 1, y, 1);
cblas_saxpy(N, a[9], x9, 1, y, 1);
```



(c)

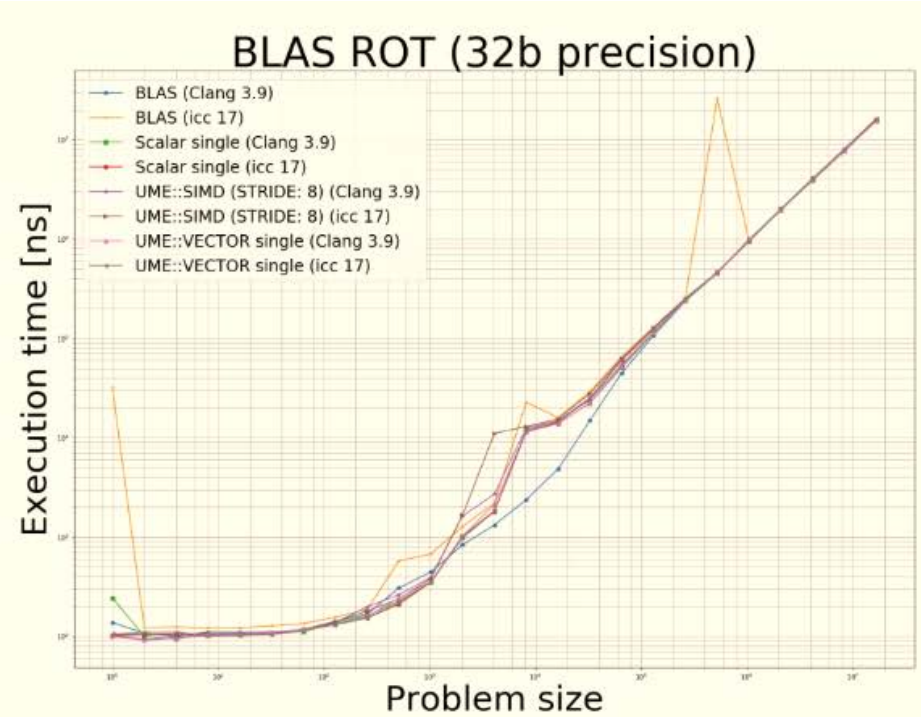


(d)

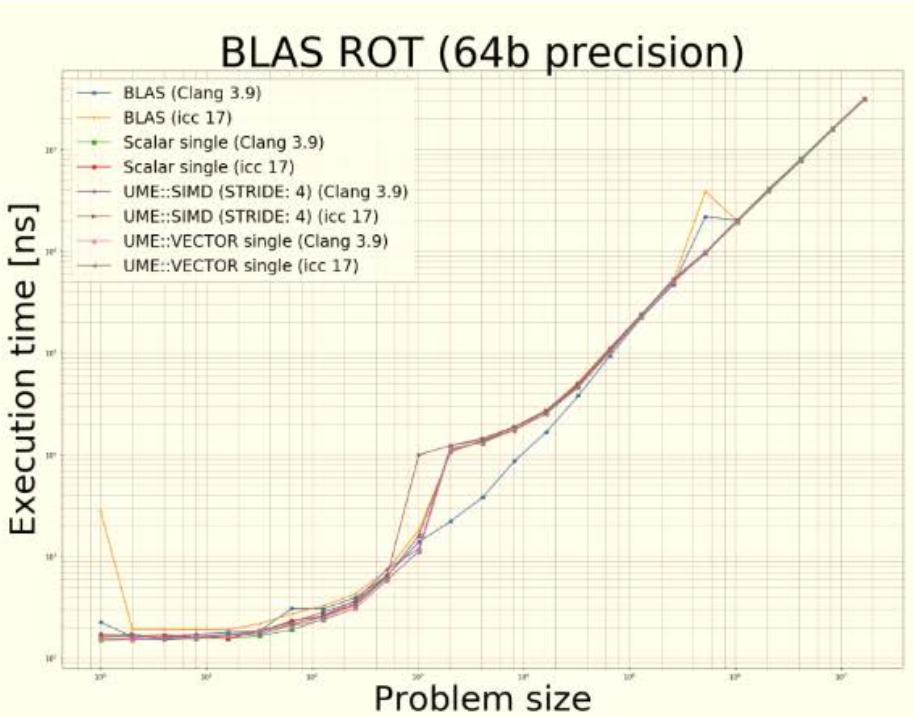
Performance comparison

```
cblas_srot(N, a, 1, b, 1, c, s);
```

$$x(t) = c*x(t-1) + s*y(t-1);$$
$$y(t) = c*y(t-1) - s*x(t-1);$$



(e)



(f)

Performance comparison

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.

Problem size	1	10	10^2	10^3	10^4	10^5	10^6	10^7	**Geomean**
GCC 5.2									
Scalar	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00
UME::SIMD	0.97/0.98	2.66/2.43	5.72/3.72	7.33/3.09	6.39/3.26	6.13/3.17	6.12/2.68	5.78/2.99	4.43/2.63
UME::VECTOR	0.96/0.98	2.75/3.50	6.37/4.47	7.39/3.44	7.38/3.76	7.27/3.61	7.01/2.93	6.43/3.34	**4.84/3.02**
ICPC 17.0									
Scalar	1.29/1.78	1.58/1.98	1.58/1.53	1.58/1.77	1.65/1.76	1.65/1.76	1.66/1.90	1.63/1.60	1.57/1.75
UME::SIMD	1.17/1.60	2.85/3.61	8.27/5.74	10.89/4.67	8.85/5.04	8.73/4.80	8.68/3.84	7.07/3.69	5.88/3.9
UME::VECTOR	1.56/2.16	2.70/4.17	8.21/6.11	9.56/4.66	8.95/4.94	8.77/4.72	8.66/3.66	6.87/3.60	**5.94/4.1**
Clang 3.9									
Scalar	1.01/1.21	2.66/2.54	6.01/2.66	7.40/1.92	6.83/3.20	6.76/3.25	6.55/3.78	6.18/3.32	4.66/2.6
UME::SIMD	0.97/1.19	2.69/2.63	6.06/4.40	7.18/2.58	6.88/3.15	6.68/3.22	6.53/3.52	5.89/3.06	4.6/2.81
UME::VECTOR	0.98/1.15	2.76/3.15	6.13/4.41	7.21/2.88	7.15/3.20	6.90/3.29	6.70/3.28	5.86/3.09	**4.68/2.89**

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.