A Performance Analysis of Vector Arithmetic with Expression Template in Supercomputer Fugaku

Yukihiro Ota

Dept. of HPC support, Research Organization for Information Science and Technology, Kobe, Japan

Abstract

Expression template (ET) is a template programming technique in C++ for efficient operator overloading on array-type objects. Compilers can interpret an ET-based code as a C-like one suitable for optimization. The expected behaviors of ET on performance can reply on its implementation and the characteristics of compilers. In this poster, we evaluate the performance in simple compute-bound kernels composed of vector arithmetic with ET in an Arm-based massively parallel computer, Supercomputer Fugaku, changing kinds of compilers available there. We show that Fujitsu Clang, GNU, and LLVM successfully apply compiler's optimization like auto-vectorization to ET-based kernels. Thus, we can understand the characteristics of the compilers in Fugaku.

Introduction

Expression templates (ET)

- ET [1] is a template programming technique in C++ for efficient operator overloading on array-type objects.
  - The key ideas are(i) to remove unnecessary temporal objects with the lazy evaluation and (ii) to complete function inlining in expressions composed of overloaded operators.
  - The code with ET can be regarded as C-like implementations (e.g., explicit loops), whose forms are suitable for the compiler optimization, such as auto-vectorization.
- Use of ET can lead to writing scientific codes with ease and performance comparable with low-level codes.
  - ET and its advanced version are used in various C++ libraries, such as Eigen (https://gitlab.com/libeigen/eigen) and Boost.Yap (https://www.boost.org/doc/libs/latest/doc/html/yap.html).
- This expected behavior of ET on performance replies on its implementation and kinds of compilers [1,2].

Motivation

- We desire to know the elementary properties of C++ compilers in Supercomputer Fugaku (A64FX), to enhanced performance and avoid for pitfalls on performance.
- To this end, we study a simple compute-bound kernel with ET on various kinds of C++ compilers in Fugaku.
  - Currently, Fujitsu (Clang/Trad), GNU, LLVM, and Arm are available in Fugaku [3].
    - REMARK: The authors previously found that ET with Fujitsu C++ in Supercomputer K (SPARCK VIIIfx) did not hide the penalties on performance of operator overloading (YO *et al.*, HPCI Research Report 3 (2018) 46-57 [in Japanese]).
- Understanding the strong points and characteristics of compilers can be helpful for the C++ users in Fugaku.
  - One can complementally use different kinds of compilers to obtain good performance in C++ applications.
    - cf. An optimization of GROMACS in Fugaku by combining Fujitsu with GNU/Arm is reported [4].

Goals

- We evaluate the performance in simple compute-bound kernels with ET in Fugaku, changing kinds of compilers.
- We study and understand how the compiler optimization enhances the performance in the ET-based kernels, depending on kinds of compilers in Fugaku.

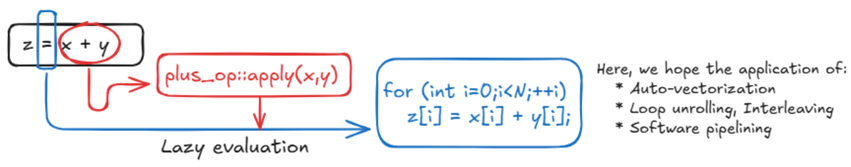
Methods

Implementation of vector arithmetic with ET

- We implement vector-arithmetic class with ET using std::array, according to the idea in [5], within C++14 standard.
  - 1. First, we define a class for "Expression" and arithmetic operations.
  - 2. Second, using these classes, we define a vector-arithmetic class as vector_et.

```cpp
// 1. To define "Expression" and arithmetic operations
template <typename LExp, typename BOp, typename RExp>
class Expression
{
  public:
  Expression(LExp _l, RExp _r) : l(std::forward<LExp>(_l)), r(std::forward<RExp>(_r)) {}
  // Expression for addition
  template <typename RE>
  inline auto operator+ (RE&& re) const -> decltype(auto) {
    return Expression<
              Expression<LExp,BOp,RExp> const &,
              plus_op<decltype(std::forward<RE>(re))>,
              decltype(std::forward<RE>(re)) > (*this, std::forward<RE>(re));
  }

  // Expression for multiplication is defined by a similar way

  inline auto operator [](int index) const -> decltype(auto) {
    return BOp::apply(l[index], r[index]);
  }

  private:
  LExp l;
  RExp r;
};

template <typename T> struct plus_op {
  static T apply(T const& a, T const& b) { return a + b; }
};
template <typename T> struct mult_op {
  static T apply(T const& a, T const& b) { return a * b; }
};
```

```cpp
// 2. Vector-arithmetic class with Expression Template, based on std::array
template <int N>
class vector_et
{
  using impl_type = std::array<double, N>;

  public:
  typedef typename impl_type::value_type value_type;

  template <typename R>
  inline vector_et& operator =(R&& re) {// Assignment (Lazy evaluation)
    for (int i=0; i<N; ++i) { v[i] = re[i]; }
    return *this;
  }

  template <typename R>
  inline auto operator +(R&& re) const -> decltype(auto){// Addition (No evaluation)
    return Expression<
              vector_et const&,
              plus_op<value_type>,
              decltype(std::forward<R>(re)) >(*this, std::forward<R>(re));
  }

  // Multiplication is defined by a similar way

  private:
  impl_type v;
};
```



Kernels

- The kernels to be measured are to calculate polynomials with Honer's method.
  - cf. 9th-degree polynomial in EuroBen (https://www.euroben.nl/).
- Increasing the degree of polynomials, the kernels become compute-bound ones.
  - The kernels are composed of FMA. The number of FMA is the degree of the polynomial.
  - The array elements are expected to be residents in registers.
- We measure the performance of the 1st-, 4th-, 8th-, and 16th-degree polynomials.
  - For reference, we measure that of the corresponding C-like implementations (i.e., use of explicit loops).

```cpp
// ET implementation: 8th-degree polynomial
vector_et<N> x, c0, c1;
x =  c0 +x*(c1 +x*(c0 +x*(c1 +x*(c0
    + x*(c1 +x*(c0 +x*(c1 +x*c0)))))));
```

```cpp
// Corresponding C-like implementation
double x[N],c0[N],c1[N];
for (int i=0; i<N; i++ ) {
   x[i] = c0[i]
        + x[i]*(c1[i] +x[i]*(c0[i] +x[i]*(c1[i] +x[i]*(c0[i]
        + x[i]*(c1[i] +x[i]*(c0[i] +x[i]*(c1[i] +x[i]*(c0[i]
        )))))));
}
```

Kinds of compilers to be investigated

Label	C++ compiler (version)	Vector length	Main options for compiler's optimization
fjclang	Fujitsu Clang (4.12.1)	512-bit	-Ofast -fvectorize -ffj-swp
fjclang.vla	Fujitsu Clang (4.12.1)	VLA	-Ofast -fvectorize -ffj-interleave-loop-insns=4
fjtrad	Fujitsu Trad (4.12.1)	512-bit	-Kfast -Ksimd=auto -Kswp -Kswp_policy=auto
gnu	GNU (12.2.0)	512-bit	-Ofast -free-vectorize -ftree-loop-vectorize -funroll-loops
llvm	LLVM (21.1.0)	512-bit	-O3 -ffast-math -fvectorize -mllvm -force-vector-interleave=4
arm	Arm (24.0)	512-bit	-Ofast -fvectorize -mllvm -force-vector-interleave=4

- As for all the cases, the instruction sets corresponds to armv8.3-a+sve.
- The options related to instruction scheduling are software pipelining (swp), interleaving, and unrolling.
- GNU, LLVM, and Arm are used only for generating the objects of the main kernels. Linking objects is performed by Fujitsu compiler, to use Fujitsu's profiler tools.

Settings

- Data size: We set the array size of each vector as 10000 (N=10000),
  - (Total amount of memory) < 8 MiB(Size of L2 cache per CMG in Fugaku).
- Precision: We take double precision in all the measurements.
- Power: We use "normal mode" in all the measurements.
  - Clock speed = 2.0 GHz, Eco mode of cores = 0 (eco_state=0) [3].
- Number of used cores: We run all the calculations with a single core in A64FX.

Number of vector elements	10000
Precision	double precision
Power setting	2.0GHz, eco_state=0
Number of used cores	1

Methods of measuring performance

- We use Fujitsu Advanced Performance Profiler (fapp) [3] to obtain performance date including elapsed time, GFlop/s, and so on, according to the measured hardware-counter information.

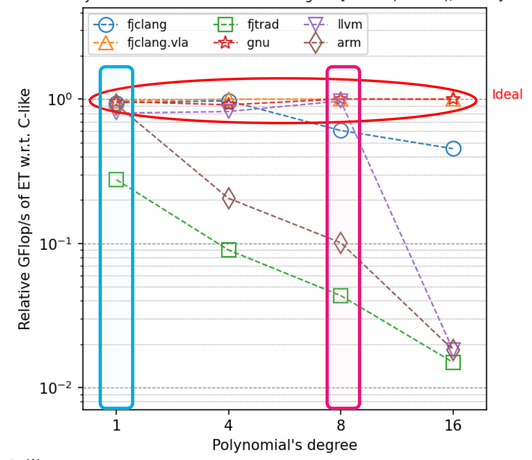
Results

TAKE HOME MESSAGE

- Fujitsu Clang, GNU, and LLVM are suitable for vector arithmetic with ET in Fugaku. Particularly, auto-vectorization is successfully applied to overloaded vector-arithmetic operations.
- Instruction scheduling by software pipelining does not positively contribute to performance in ET, within the current experiments. In contrast, interleaving works well.

GFlop/s: ET versus C-like

- The red circle indicates that ET is equivalent to C-like on performance, including fjclang (up to the 4th deg.) (◯), fjclang.vla (△), gnu (☆), llvm (up to the 8th deg.) (▽), and arm (up to the 1st deg.) (◊).
  - As for the 8th degree in fjclang, the performance is still compatible with fjclang.vla, gnu, and llvm (See the table and figure right below).
- Otherwise, ET shows significant decrease of performance.



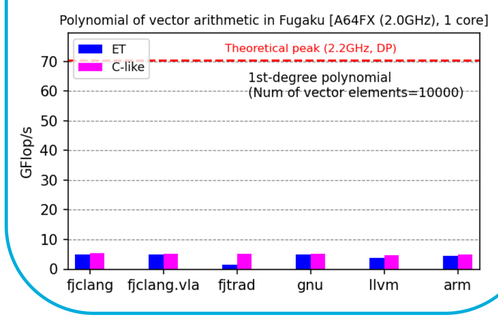
Auto-vectorization and instruction scheduling in ET

- To understand the results, we closely observe SIMD instruction rate and IPC, focusing on the 1st- and 8th-degree polynomials.
- On the 1st degree (▭):
  - SIMD inst. rates of ET are high, except for fjtrad.
  - On ET with fjtrad , auto-vectorization is suppressed. The significant lower performance than C-like occurs.
- On the 8th degree (▭):
  - Both of SIMD inst. rates and IPCs of ET are equivalent to C-like, except for fjclang, fjtrad, and arm.
  - On ET with fjclang, the lower IPC than C-like indicates that software pipelining does not work.
    - This is contract to interleaving (See IPCs in fjclang.vla and llvm).
  - On ET with fjtrad and arm, auto-vectorization is suppressed.

Effects of interleaving in Fujitsu Clang with ET
On the 8th degree polynomial:
- fjclang.vla: GFlop/s = 21.2
- fjclang.vla w/o interleaving: GFlop/s = 18.9

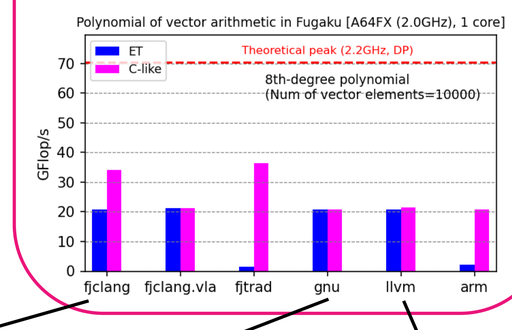
1st-degree polynomial

	ET		C-like	
	SIMD instruction rate (%)	IPC	SIMD instruction rate (%)	IPC
fjclang	90.3	0.868	86.11	0.964
fjclang.vla	72.4	1.27	72.5	1.31
fjtrad	0.00	2.87	78.4	1.03
gnu	81.7	0.966	66.6	1.34
llvm	89.3	0.669	75.6	0.987
arm	80.2	0.890	75.6	1.03

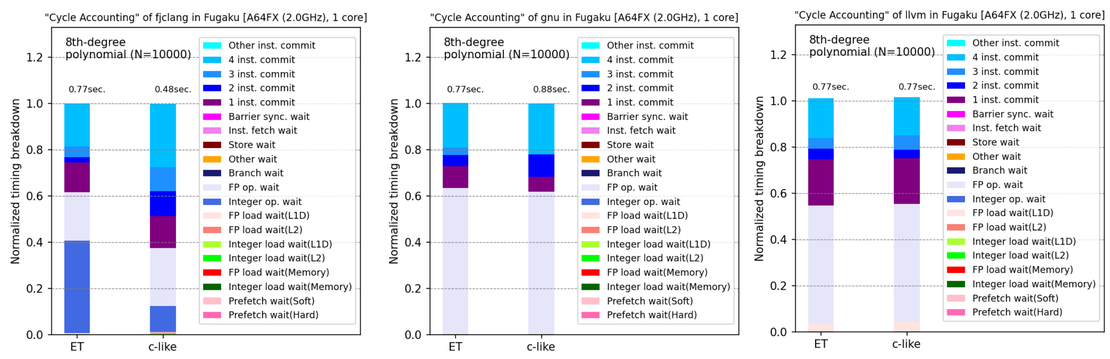


8th-degree polynomial

	ET		C-like	
	SIMD instruction rate (%)	IPC	SIMD instruction rate (%)	IPC
fjclang	92.1	1.05	88.5	1.81
fjclang.vla	85.0	1.27	85.1	1.26
fjtrad	0.00	1.56	89.4	1.91
gnu	92.2	1.06	81.2	1.18
llvm	87.6	1.11	87.6	1.15
arm	0.00	1.35	87.6	1.11



CPU performance analysis



- To fully understand instruction processing, we perform CPU performance analysis (PA) by fapp [3].
  - The timing data of the hardware counters are collected. The normalization is done by the elapsed time.
- On ET with fjclang
  - Wait time for integer operations (■) shows a marked behavior, compared to C-like.
  - This wait time does not appear in gnu and llvm.
- On ET with gnu
  - Timing breakdown is almost equal to C-like although slight differences on instruction commits exist.
  - Wait time for FP operations (■) is major. Instruction scheduling is not good, compared to fjclang.
- On ET with llvm
  - Again, there is no significant difference between ET and C-like.
  - Wait time for FP load (L1D) (■) slightly increases, compared to fjclang and gnu.

Perspectives

We have two future issues.
(1) To implement ET for software pipelining

- The result of CPU PA indicates that hiding or removing the latency of (unexpected) integer operations is required.

(2) To implement ET for compiler-based generation of GPU code.

- Use of directives in OpenACC or OpenMP offloading is a good candidate.
  - Indeed, we perform an initial experiment with OpenACC and NVIDIA HPC SDK (23.11) in NVIDIA A100. However:
    - The routine directive with gang for GPU parallelization does not work, within our trials.
    - Function inlining associated with the lazy evaluation leads to a sequential code.
- We will need to seek a suitable implementation of ET for GPU.

Summary

We study the characteristics of compilers in Fugaku via ET-based vector-arithmetic kernels. We find that Fujitsu Clang, GNU, and LLVM successfully apply auto-vectorization to the kernels. Implementing ET towards software pipelining is a future issue. Another future issue is the application of our approach with ET to GPU codes.

Contact address: yota@rist.or.jp
Acknowledgements: This research used computational resources of the supercomputer Fugaku provided by the RIKEN Center for Computational Science.

Reference

[1] D. Vandevoorde, N. M. Josuttis and D. Gregor, 2018, *C++ Templates: The complete Guide* (2nd ed.), Addison-Wesley, Boston.
[2] K. Iglberger, G. Hager, J. Treibig and U. Ruede, 2012, *Expression Templates Revisited: A Performance Analysis of the Current ET Methodology*, SIAM J. on Sci. Comput. 34(2), C42-C69. https://doi.org/10.1137/110830125
[3] *Fugaku User Guide - Language and development environment*. https://www.r-ccs.riken.jp/en/fugaku/user-manuals/
[4] G. Gouaillardet, *Porting and tuning GROMACS on Arm SVE*, in The 2025 Arm HPC User Group (AHUG) Workshop, https://github.com/arm-hpc-user-group/isc25-ahug-workshop
[5] Jiří Vyskočil, *Template Metaprogramming for Massively Parallel Scientific Computing* in Inverted CERN School of Computing 2016.. https://vyskocil.com/icsc-2016/