Camelus: Scalable and Generic SIMD Programming for AI Accelerators

Xiaofeng Guan (xiaofeng.guan@enflame-tech.com), Hao Zhou (vincent.zhou@enflame-tech.com), Zhengping Hu (zhengping.hu@enflame-tech.com), Jianguo Yao (jianguo.yao@enflame-tech.com, jianguo.yao@sjtu.edu.cn)

Enflame Revolutionary Academy

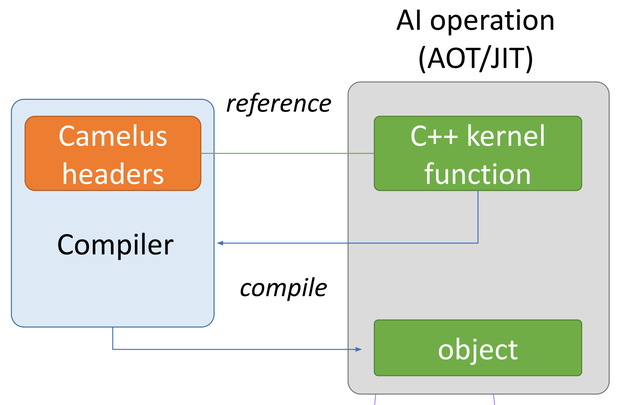

Figure 3 - Workflow with Camelus

Motivation

AI accelerators often come equipped with SIMD hardware, which allows them to exploit data-level parallelism. SIMD programs can be automatically generated by compilers through techniques like vectorization or explicitly programmed using compiler directives like OpenMP or SPMD programming languages like OpenCL.

Explicit SIMD programming involves programming SIMD intrinsic functions with vector-typed data. While it has the advantage of avoiding indirection, **maintaining SIMD program incurs high costs for compatibility/generality**.

One contributing factor is that the length of a SIMD vector register **varies based on its element types**, such as ARM Neon 128-bit vector register, which can contain 16x8-bit elements, 8x 16-bit elements, 4x 32-bit elements, or 2x 64-bit elements), making it difficult to create a unified implementation for multiple types of operations in a high-performance AI. This is because the operations are type-based and born from hardware instructions. As a result, duplicated code and higher maintenance costs may be necessary.

Another challenge is that the length of a specific SIMD vector register **varies between different hardware**, such as Intel's packed SIMD extension, which has evolved from SSE(4x float elements), to AVX(8x float elements), and AVX-512(16x float elements). Supporting new SIMD hardware requires re-engineering the same functionality in software for different vector lengths.

About Enflame GCU

Enflame GCU (General Computing Unit)* AI chips series supports matrix engine hardware to accelerate AI workloads. These chips come equipped with numerous SIMD units that can be explicitly programmed using SIMD intrinsics. In recent hardware changes, the vector length is reduced by half as the number of hardware threads increase.

Enflame 燧原科技

Considerations of the design

Our target of this work is to make SIMD programming **insensitive to different vector lengths**, as the vector length is critical for the generality and compatibility of SIMD code.

Unlike many known solutions, Camelus SIMD programming system offers a series of **programming APIs** that it neither requires ISA or hardware change (e.g. the support of PTX assembly) nor requires compiler re-design or enhancement (e.g. the variable vector-length IR support of ARM-SVE).

Instead, the Camelus system utilizes the **modern C++ language's template meta-programming** capability, which is already supported by many existing compilers. With this approach, programmers can conduct type-based template instantiation, which is suitable for scenarios involving "same/similar action with different types." Additionally, the compiler instantiates the templated code at compilation time, and with proper design, there may be no extra cost compared to SIMD intrinsic code.

Cost in compilation

Camelus relies much on C++ language features. Camelus may incur longer compilation times, particularly with the C++ compiler front-end. As a result, it is less suitable for building the JIT (Just-In-Time) compilation workload.

Camelus

This research is under the research project of "Compiler Design for Large Scale Heterogeneous Deep Learning", which is funded by the Science and Technology Committee of Shanghai, China, under award number 22QB1404600.

Methodology

Two High-level Abstraction

- **Abstract SIMD Vector Type (ASVT)**, which has NO FIXED vector length value.
- **Generic Operations** on top of the *ASVT*, which are implemented as C++ function templates.

With Camelus, instead of programming with a concrete vector type, code can be written using a unique **ASVT** ("`vector_type`" in the Fig.1), which represents the vector type suitable for the current hardware and its corresponding scalar type.

For example, on hardware with 512-bit SIMD registers, the meta function "*`scalar_to_vector_t`*" deduces the ASVT as *`<16 x float>`* from its scalar element type, *`float`*. If used on hardware with 1024-bit SIMD registers, the ASVT would be deduced as *`<32 x float>`* correspondingly.

The meta function "*`vector_length_t`*" extracts the vector length directly from ASVT. The vector length is for stepping up the loops, etc., which allows programmers to write generic functions as necessary.

```
template <typename T>
void scalar_version(T* a, T* b, T *c, int n) {
  for (int i = 0; i < n; ++i) {
    if (a[i] < b[i])
      c[i] += b[i];
    else
      c[i] -= a[i];
  }
}

template <typename T>
void vector_version(T* a, T* b, T *c, int n) {
  using vector_type = scalar_to_vector_t<T>;
  constexpr int vl = vector_length_v<vector_type>;

  vector_type * va = (vector_type *)a;
  vector_type * vb = (vector_type *)b;
  vector_type * vc = (vector_type *)c;

  // ... check for alignment and elements count

  for (int i = 0; i < n / vl; ++i) {
    auto mask = less_than(va[i], vb[i]);  // generate the vector mask
    add_t(&c[i], c[i], b[i], mask);   // add only if the mask bit is set
    sub_f(&c[i], c[i], a[i], mask);   // sub only if the mask bit is unset
  }
}
```

Generic operations with the vector mask

Figure 2 – Generic operations for ASVTs and the vector mask

Abstract SIMD Vector Type | Meta functions

```
void foo(float * pa, int n) {
  // get the ASVT
  using vector_type = scalar_to_vector_t<float>;
  // get the vector length for the ASVT
  constexpr int vl = vector_length_v<vector_type>;

  if (valigned(pa) && n % vl == 0) {
    vector_type * va = (vector_type *) pa;
    for (int i = 0; i < n / vl; ++i) {
      va[i] = va[i] * va[i];
    }
  }
}
```

Figure 1 - Abstract SIMD Vetor Type (ASVT) and its type and vector-length decuction

In the example of Fig. 1, function "*`foo`*" is compatible with different hardware by using the ASVT to deduce corresponding vector types. If we change "*`foo`*" to be a function template, it turns "*`foo`*" to be more generic code. This is shown in Fig. 2.

Limitation

While the use of C++ templates incurs a one-time compile-time cost rather than a runtime cost, it also means that code compatibility is limited to the source code level.

The **generic operations** in Camelus are implemented as function templates that take ASVTs as their function parameters. This approach allows for identical semantics across different vector types. C++ Concept feature also allows Camelus **to detect errors at compile-time** when a generic operation has not been implemented yet or missing for a specific type or hardware, making programming with Camelus more productive than normal SIMD programming that requires searching for intrinsic names to program.

To support branch divergence in SIMD code, Camelus introduces the **execution mask** type and also the generic operations using execution masks. In the example of Fig. 2, the system generates a vector "*`mask`*" using the "*`less_than`*" operation, and then use the "*`add_t`*" (If the vector mask bit is not set, the operation is not performed in the vector lane) and "*`sub_f`*" operations to achieve a similar execution flow to that of the "*`scalar_version`*" function, effectively serialized the *`if-else`* statement.

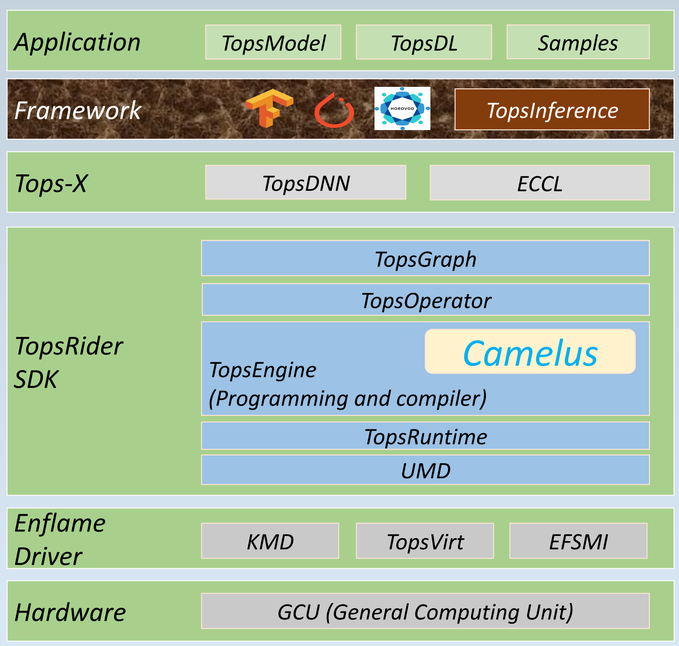

Figure 4 – Camelus in Enflame TOPS software stack

Camelus in Enflame

The Camelus header library was developed specifically for programming the Enflame high-performance AI library on top of the GCU AI accelerator. It supports programming for AI AOT (ahead-of-time) training/inference workloads, with nine vector types and 24 generic operations. This provides a generalization of over 400 SIMD intrinsics and greatly improves coding productivity.

Furthermore, since Camelus has no runtime overhead, code written using Camelus achieves identical performance with existing SIMD code implementations. As a result, Camelus has been adopted in the development of the AI library for Enflame's next-gen hardware. Early feedback has shown productivity improvements already.