nova.simd - A framework for architecture-independent SIMD development

Tim BLECHMANN
tim@klingt.org

Abstract

Most CPUs provide instruction set extensions to make use of the Single Instruction Multiple Data (SIMD) programming paradigm, examples are the SSE and AVX instruction set families for IA32 and X86_64, Altivec on PPC or NEON on ARM. While compilers can do limited auto-vectorization, developers usually have to target each instruction set manually in order to get the full performance out of their code.

Nova.simd provides a generic framework to write cross-platform code, that makes use of data level parallelism by utilizing instruction sets.

1 Introduction & Motivation

Most processors provide instructions to make use of data-level parallelism via the Single Instruction Multiple Data (SIMD) paradigm [3]. The instruction sets usually treat data as small vectors of scalars, which can be processed with single instructions.

In an ideal world compilers would generate these instructions manually without any contribution from the developers. Unfortunately, it is not that easy: some instructions require specific conditions like memory alignment and often the compiler is not able to infer, if certain memory regions overlap at run-time (aliasing). Some compilers avoid this by adding run-time checks, which of course introduce some overhead. Some more complex algorithms would require non-trivial program transformations, which are unlikely to be performed by the compiler.

To get the full performance out of the SIMD instruction sets, one usually has to target each instruction set manually, either by using compiler intrinsics or by writing assembler. Unfortunately, each instruction set implements different concepts. The SSE instruction set for example has been originally working only on single-precision floating point numbers, but has been extended to integer data and double-precision floating point types with SSE2, and literally each new CPU generation added some more instructions for specific use cases[1]. Some vendors provide specific libraries, but unfortunately most of them have specific restrictions, only abstract once specific instruction set or work only on a specific platform.

`Nova.simd` was designed to provide a generic and easy to use framework to easily write SIMD code, which is independent from the instruction set. It provides ready-to-use vector functions, but also an generic framework to write generic vector code. It is a header-only C++ library that makes heavy use of templates and template metaprogramming techniques and currently supports the SSE and AVX families on IA32 and X86_64, Altivec on PPC and NEON on ARM. But can be easily extended to other/-future instruction sets, if they provide a reasonable compiler support via intrinsics. For unsupported platforms, a generic C++ implementation is provided. The library is free software, released under the GPL-2+.

Section 2 explains the design decisions of `Nova.simd`, section 3 introduces the main part of the library, the `vec` class, section 4 explains the provided algorithms and we conclude in section section 5 with a discussion of related libraries and frameworks.

2 The Design of `Nova.simd`

`Nova.simd` was started, when the author worked on the 'nova' project for his bachelor thesis [1], which provided a simple form of abstraction for SIMD functions. After the 'nova' project was abandoned, the SIMD code was maintained separately and the unit generators of the computer music system SuperCollider [5] were adapted to use `nova.simd` instead of the

[1] The additional SSE instruction extensions have been SSE3, SSSE3, SSE4a, SSE4.1, SSE4.2. AMD's proposed SSE5 has never made it to real hardware.

old PPC-specific Altivec code. After some attempts to use Python to generate C++ code, it is currently implemented as a header-only C++ library, which makes heavy use of template and template meta-programming techniques to generate code at compile-time.

The main idea behind the design is to separate the library into a platform-specific and a platform-agnostic part. The platform-specific part is based on a generic template `vec<>` class, which represents a SIMD vector. Using template specialization, platform-specific versions of this class can be implemented for separate instruction sets, details can be found in the following section. The platform-agnostic part then uses the `vec<>` class to build more complex algorithms.

3 The `vec` class

The `vec<>` class is the heart of `nova.simd`. It represents a single SIMD vector similar to `__m128` on SSE, `__m256` on AVX, `vector float` on Altivec or the ARM specific `float32x4_t`. However unlike these platform-specific types, which are plain C-stype PODs, the `vec<>` class provides a feature-rich class interface, which can be used to compose more complex algorithms.

All `vec<>` class instatiations provide a common interface for aligned and non-aligned loads and stores, piecewise vector arithmetics, but also some functions for horizontal accumulation and some mathematical functions. Class specializations for SIMD extensions also provide common building blocks for vector code like comparison to bitmask, bitwise logical operations and other functionality, which is required to formulate vectorized code.

The advantage of this modular approach is portability: once an algorithm is implemented via the `vec<>` class interface, it will usually work out of the box on other platforms. The vector implementations for the mathematical functions for example are directly implemented via the `vec<>` class. The original algorithms for the approximations are adapted from cephes [6], while some of the approximation polynomials are improved with Sollya [2]. The main idea behind the vectorized implementation is to evaluate the approximation polynomial for every part of the function and select the result for the specific interval with bitmasks.

3.1 Class Interface

The `vec<>` class provides a feature rich interface, which provides the means to implement more complex algorithms. It covers the following aspects:

Operators
: `vec<>` provides C++ operators for arithmetics and comparison. However it omits logical operators, which are provided as bitmasking functions (see below). Multiply-Accumulate is provided as a function, as it is supported by several SIMD instruction sets.

Bitwise Operators and Bitmasks
: All template specializations which actually map to SIMD implememntations provide comparison functions, that yield bitmasks. These bitmasks can then be used with bitwise operators or a `select()` function, which selects a specific value from two arguments, depending on the value of a selection bitmask.

Element Access
: It is possible to read and write different vector elements. However there is no guarantee if the instruction set implements this efficiently.

Math Functions
: `vec<>` implements many, but not all functions that are part of libm. However it provides some functions that are useful for signal processing like signed square root or a signed power function.

Horizontal Functions
: Certain algorithms require some form of horizontal accumulation. The `vec<>` class provides functions that return the minimum or maximum element and the sum of all elements.

Constant Generation
: Some specific constants can efficiently be generated via bit twiddling tricks, which can be more efficient than loading a constant from memory.

Listing 1: SuperCollider's soft clipping

```
template <typename Float>
Float sc_softclip(Float arg)
{
  Float abs_arg = std::fabs(arg);
  if (abs_arg < 0.5)
    return arg;
  else
    return (abs_arg - 0.25) / arg
}
```

Listing 2: SuperCollider's soft clipping via `nova.simd`

```
template <typename Float>
void softclip_vec(Float * out,
  Float const * in, int count)
{
  typedef vec<Float> vec_type;
  vec_type const05 = vec_type::gen_05();
  vec_type const025 = vec_type::gen_025();
  const int vs = vec<Float>::size;

  int loops = count / vs;
  for (int i = 0; i != loops; ++i) {
    vec_type arg;
    arg.load_aligned(in + i*vs);
    vec_type abs = abs(arg);
    vec_type mask = mask_lt(abs, const05);
    vec_type alt_ret
      = (abs - const025) / arg;

    vec_type result
       = select(alt_ret, arg, mask);
    result.store_aligned(out + i*vs);
  }
}
```

3.2 Example: Soft Clipping

As example we have a closer look, how SuperCollider's softclip operator could be implemented using the `vec<>` class. Softclip is a simple wave shaper that maps values with an absolute value from 0.5 to infinity to 0.5 to 1 (compare Listing 1). The vectorized implementation shown in Listing 2. For the sake of simplicity, we assume that both pointers are reasonably aligned and that `count` is a multiple of the `vec<>` size. In the beginning, we generate two constants. Then in the loop, we load the argument and compute its absolute value. In the following line, we generate a bitmask that denotes that the argument is less than 0.5. Then we compute the result that should be used in the case that the absolute value is equal or greater 0.5 using the arithmetic operators. We finally use the `select()` function to select the result depending on the bitmask and store the result to the output.

This example shows one simple, but very useful code transformation: instead of branching, we compute both sides of the branch and use a bitmask to select the result. Attentive readers might argue that vectorized code has to perform a (slow) division, which the branching implementation could avoid in many cases. However our benchmarks show that the vectorized code is faster than the branching version. The out-of-order execution engines of recent CPUs are even able to cancel the division, if the bitmask of the select statement implies that the result of the division won't be used.

Listing 3: Signatures for Vector Additions

```
template <typename FloatType,
  typename Arg1Type, typename Arg2Type>
inline void plus_vec(FloatType * out,
  Arg1Type arg1, Arg2Type arg2,
  unsigned int n);

template <typename FloatType,
  typename Arg1Type, typename Arg2Type>
inline void plus_vec_simd(FloatType * out,
  Arg1Type arg1, Arg2Type arg2,
  unsigned int n);

template <unsigned int n,
  typename FloatType, typename Arg1Type,
  typename Arg2Type>
inline void plus_vec_simd(FloatType * out,
  Arg1Type arg1, Arg2Type arg2);
```

4 Generic Vector Algorithms

The `vec<>` class is used to implement a number of higher-level vector algorithms, most of them are similar of other frameworks for vector arithmetics. However all vector arithmetic functions come with a templated C++ interface, which provides some very handy features for audio synthesis applications. The basic signatures for the arithmetic vector addition is shown in Listing 3.

We provide three different versions of the function: the first version `plus_vec` does not make any assumptions about arguments, vector alignment or vector size, so it can be called with any reasonable arguments, even overlapping arrays. The two versions of `plus_vec_simd` make certain assumptions concerning alignment and size: argument vectors are expected to be aligned and the vector size is expected to be a positive multiple of the cache line size. These requirements are stricter compared to other libraries, but have some advantages: cache lines of most current CPUs have a size of 64byte, enough space for 16 single-precision or 8 double-precision floating point numbers. `Nova.simd` takes advantage of this by unrolling the inner loop to work on one cache-line at a time. This reduces the number of branches which reduces the pressure to the branch predictor and helps the compiler to generate pipeline-friendly machine code. The second versions of `plus_vec_simd` goes even one step further as it requires to specify the vector size at compile time. Using C++ template metaprogram-

Listing 4: Expression Template Generators

```
template <typename FloatType>
inline ImplementationDefinedType
scalar_argument(FloatType const & f);

template <typename FloatType>
inline ImplementationDefinedType
vector_argument(const FloatType * f);

template <typename FloatType>
inline ImplementationDefinedType
slope_argument(FloatType const & value,
               FloatType const & slope);
```

ming techniques the compiler will the generate a branch-less version of this function. This is a tricky trade-off and has to be handled with care: on the one hand short loops may be optimized by the CPU (e.g. via the Loop Stream Detector of recent Intel CPUs [4]), but on the other hand, completely avoiding branches is much better for the pipeline. But since many audio synthesis applications work on rather small vector sizes like 64 samples, this can result in better runtime performance.

4.1 Expression Templates as Function Arguments

As seen in the function signatures, both arguments for the addition are templated: when passing a pointer, the argument will be interpreted as vector, when passing a floating point value, the value is interpreted as scalar. This has the advantage that the same function can be used for different argument combination of vector and scalar arguments. But the template approach is even more powerful as we can pass expression templates [7] to the function. `Nova.simd` currently provides three expression templates, two trivial to explicitly pass scalars or vectors as argument and one expression template that generates a linear ramp of scalars, which can be used to avoid zipper noise when changing scalar parameters of a unit generator. The synopsis can be found in Listing 4.

5 Related Works

`Nova.simd` shares some concepts with other libraries or frameworks. An incomplete list of the most common ones can be found in Table 1. Unfortunately many of them have certain limitations: most libraries are vendor-specific and/or target only a specific SIMD instruction set. The libraries of Intel only support x86 and x86_64 and may even require to use the Intel C Compiler. Apple's Accelerate framework is only provided for Apple's operating systems. Many libraries are proprietary, so they cannot be used by open source projects.

The most common APIs are C-style vector APIs, that work on arrays of scalars. Most of these functions are hidden behind a library interface, which makes them easy to use and avoids the burden of implementing run-time dispatching, but this prevents the library to be used for composing new efficient algorithms, because the compiler cannot perform any interprocedural optimization. A short vector interface composes much better, but only the upcoming Boost.SIMD library neither has any restrictions concerning architecture or platform nor is it proprietary.

6 Conclusion

We've presented `nova.simd`, a simple but powerful library for practical SIMD development. It supports abstracts different SIMD instruction set under a common interface and provides building blocks to write more advanced algorithms. It is heavily used in the audio synthesis core of SuperCollider.

7 Acknowledgements

This paper was peer reviewed accepted as poster for the ICMC 2012, but was not presented due to the lack of funding and therefore published independently.

References

[1] T. Blechmann, "pnpd/nova, a New Audio Synthesis Engine with a Dataflow Language," in *Proceedings of the 5th International Linux Audio Conference*. Technische Universitt Berlin, Germany, 2007, pp. 55–59.

[2] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An Environment for the Development of Numerical Codes," *International Congress on Mathematical Software 2010*, pp. 28–31, 2919.

[3] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 948–960, 1972.

[4] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

[5] J. McCartney, "SuperCollider, a new real time synthesis language," in *Proceedings of*

Figure 1: Related Vector Libraries

Library	Features	Restrictions
Accelerate framework (Apple)	Arithmetic, Math, Short Vectors	Proprietary, Apple-only
Integrated Performance Primitives (Intel)	Arithmetic	Proprietary, Intel-only
Math Kernel Library (Intel)	Math	Proprietary, Intel-only
Short Vector Math Library (Intel)	Math, Short Vectors	Intel Compiler Intrinsics, Intel-only
Framewave (AMD)	Arithmetic, Math	SSE family only
AMD libM (AMD)	Math, Short Vectors	SSE family / x86_64 only
Boost.SIMD	Short Vectors	unreleased, undocumented
SSEplus	Short Vectors	SSE family only

the International Computer Music Conference, 1996.

[6] S. Moshier, "Cephes mathematical library," http://www.netlib.org/cephes/, 1992.

[7] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.