Boost.SIMD: Generic Programming for Portable SIMDization

Pierre Estérie
LRI, Université Paris-Sud XI
Orsay, France
pierre.esterie@lri.fr

Mathias Gaunard
Metascale
Orsay, France
mathias.gaunard@metascale.org

Joel Falcou
LRI, Université Paris-Sud XI
Orsay, France
joel.falcou@lri.fr

Jean-Thierry Lapresté
IP (Institut Pascal), Université Blaise Pascal
Clermont-Ferrand, France
lapreste@univ-bpclermont.fr

Brigitte Rozoy
LRI, Université Paris-Sud XI
Orsay, France
rozoy@lri.fr

ABSTRACT

SIMD extensions have been a feature of choice for processor manufacturers for a couple of decades. Designed to exploit data parallelism in applications at the instruction level and provide significant accelerations, these extensions still require a high level of expertise or the use of potentially fragile compiler support or vendor-specific libraries. In this poster, we present Boost.SIMD, a C++ template library that simplifies the exploitation of SIMD hardware within a standard C++ programming model.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD)*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

Keywords

SIMD, C++, Generic Programming, Template Metaprogramming

1. MOTIVATIONS

Since the late 90's, processor manufacturers have provided specialized processing units called multimedia extensions or Single Instruction Multiple Data (SIMD) extensions. The introduction of this feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. Today's processor architectures offer rich SIMD instruction sets working with larger and larger SIMD registers. For example, the AVX extension introduced in 2011 enhances the x86 instruction set for the Intel Sandy Bridge and AMD Bulldozer microarchitectures by providing a distinct set of 16 256-bit registers. Similary, the forthcoming Intel MIC Architecture will embed 512-bit SIMD registers and embedded systems also integrate such features like NEON ARM extensions. However, programming applications that take advantage of the SIMD extension remains a complex task. Programmers that use low-level intrinsics have to deal with a verbose programming style which is burying the initial algorithms in architecture specific implementation details. Furthermore, these efforts have to be repeated for every different extension that one may want to support, making design and maintenance of such applications very time consuming. Different approaches have been suggested to limit these shortcomings. On the compiler side, **autovectorizers** implement code analysis and transform phases to generate vectorized code [2, 4]. Compilers are able to detect code fragments that can be vectorized but they struggle when the classical code is not presenting a clear vectorizable pattern (complex data dependencies, non-contiguous memory accesses, aliasing or control flows). Other compiler-based systems use code directives to guide the vectorization process by enforcing loop vectorization. The ICC and GCC extension **`#pragma simd`** is such a system. To use this mechanism, developers explicitly introduce directives in their code, thus having a fine grain control on where to apply SIMDization. However, the generated code quality will greatly depend on the used compiler. Another approach is to use **libraries** like Intel MKL. Those libraries offer a set of domain-specific routines that are optimized for a given architecture. This solution suffers from a lack of flexibility as the proposed routines are optimized for specific use-cases that may not fulfill arbitrary code constraints.

PACT'12, September 19–23, 2012, Minneapolis, MN, USA.
ACM 978-1-4503-1182-3/12/09.

2. THE BOOST.SIMD LIBRARY

The main issue of SIMD programming is the lack of proper abstractions over the usage of SIMD registers. This abstraction should not only provide a portable way to use hardware-specific registers but also enable the use of common programming idioms when designing SIMD-aware algorithms.

2.1 SIMD register abstraction

The first level of abstraction introduced by Boost.SIMD is the `pack` class. For a given type `T` and a given static integral value `N` (`N` being a power of 2), a `pack` encapsulates the best type able to store a sequence of `N` elements of type `T`.

For arbitrary `T` and `N`, this type is simply `std::array<T,N>` but when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used instead. By default, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. The `pack` class handles these low-level SIMD register type as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value, list of scalar values, iterator or range. In each case, the proper register loading strategy (splat, set, load or gather) will be issued. This abstraction is coupled with a large set of functions covering the classical set of operators along with a sensible amount (200+) of mathematical functions and utility functions like constant generators and arithmetic/IEEE 754/reductions functions. A fundamental aspect of SIMD programming relies on the effective use of fused operations like multiply-add on VMX extensions or sum of absolute differences on SSE extensions. `pack` relies on *Expression Templates* [1] to capture the Abstract Syntax Tree (AST) of large `pack`-based expressions and performs compile-time optimization on this AST. These optimizations include the detection of fused operation and replacement or reordering of reductions versus elementwise operations. Moreover, the AST-based evaluation process is able to merge multiple function calls into a single inlined one, contrary to solutions like MKL where each function can only be applied on the whole data range at a time. This increases data locality and ensures high performance for any combination of functions.

2.2 C++ Standard integration

Realistic applications usually require functions applied over a large set of data. To support such a use case, the library provides a set of classes to integrate SIMD computation inside C++ code relying on the C++ Standard Template Library (STL) components, thus reusing its generic aspect to the fullest. Based on Generic Programming [3], the STL is based on the separation between data, stored in various `Containers`, and the way one can traverse these data, thanks to `Iterators` and algorithms. BOOST.SIMD reuses existing STL Concepts to adapt STL-based code to SIMD computations by providing SIMD-aware allocators, iterators for regular SIMD computations and hardware-optimized algorithms. The hardware implementation of SIMD processing units introduces constraints related to memory handling. Performance is guaranteed by accessing the memory through dedicated `load` and `store` intrinsics that perform register-length aligned memory accesses. This constraint requires a special memory allocation strategy via OS and compiler-specific function calls. BOOST.SIMD provides a STL compliant allocator dealing with this kind of alignment. The `simd::allocator` class wraps these OS and compiler functions in a simple STL-compliant allocator. Usually, modern C++ programming style based on Generic Programming leads to an intensive use of various STL components like Iterators. BOOST.SIMD provides iterator adaptors that turn regular random access iterators into iterators suitable for SIMD processing. These adaptors act as free functions taking regular iterators as parameters and return iterators that output `pack` whenever dereferenced. These iterators are then usable directly with common STL algorithms such as `transform` or `fold`. The code keeps a conventional structure which facilitates the usage of template functors for both scalar and SIMD cases, maximizing code reuse.

3. THE RGB2YUV ALGORITHM

The RGB and YUV models are both color spaces with three components that encode images or videos. Unlike RGB, the YUV color space takes into account the human perception of the colors. The RGB2YUV algorithm fits the data parallelism requirement for SIMD computation. The comparison shown in Table 1 shows the performance of BOOST.SIMD against scalar C++ code and SIMD reference code. The implementation is realized in single precision floating-point and the results are presented in cycles per point (cpp).

Table 1: Results for RGB2YUV algorithm in *cpp*

Size	Version	SSE4.2	AVX	Altivec
256^2	Scalar C++	29.23	21.46	42.51
	Ref. SIMD	6.48	2.80	29.05
	Boost.SIMD	6.51	2.45	29.01
	Speedup	4.49	8.76	1.47
	Overhead	0.04%	-14.3%	0.1%

The speedups obtained with SEE4.2 and AVX are superior to the expected $\times 4$ and $\times 8$ on such extensions and no overhead is added by BOOST.SIMD. A slowdown appears with Altivec due to the lack of a level 3 cache which causes the SIMD unit to wait constantly for data from the main memory. For a size of 64^2 on the PowerPC, BOOST.SIMD performs at 4.65 cpp against 36.32 cpp for the scalar version giving a speedup of 7.81 which confirms the memory limitation of such an architecture. The latent data parallelism in the algorithm is fully exploited and the benchmarks corroborate the ability of the library to generate effecient SIMD code.

4. CONCLUSION

SIMD instruction sets are a technology present in an ever growing number of architectures. Despite the performance boost that such extensions usually provide, SIMD has been usually underused. Losing the $\times 4$ to $\times 16$ speed-ups they may provide in HPC applications is starting to be glaring. We presented BOOST.SIMD, which aims at simplifying the design of SIMD-aware applications while providing a portable high-level API, integrated with the C++ Standard Template Library and solves the problem of portable SIMD code generation.

5. REFERENCES

[1] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39, 1998.

[2] J. Shin. Introducing control flow into vectorized code. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:280–291, 2007.

[3] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, 1995.

[4] H. Wang, H. Andrade, B. Gedik, and K.-L. Wu. A code generation approach for auto-vectorization in the spade compiler. In *LCPC'09*, pages 383–390, 2009.