Simple, Portable and Fast SIMD Intrinsic Programming: Generic SIMD Library

Haichuan Wang
University of Illinois at Urbana-Champaign
hwang154@illinois.edu

Peng Wu Ilie Gabriel Tanase
Mauricio J. Serrano José E. Moreira
IBM T.J. Watson Research Center
{pengwu,igtanase,mserrano,jmoreira}@us.ibm.com

Abstract

Using SIMD (Single Instruction Multiple Data) is a cost-effective way to explore data parallelism on modern processors. Most processor vendors today provide SIMD engines, such as Altivec/VSX for POWER, SSE/AVX for Intel processors, and NEON for ARM. While high-level SIMD programming models are rapidly evolving, for many SIMD developers, the most effective way to get the performance out of SIMD is still by programming directly via vendor-provided SIMD intrinsics. However, intrinsics programming is both tedious and error-prone, and worst of all, introduces non-portable codes.

This paper presents the *Generic SIMD Library* (`https://github.com/genericsimd/generic_simd/`), an open-source, portable C++ interface that provides an abstraction of short vectors and overloads most C/C++ operators for short vectors. The library provides several mappings from platform-specific intrinsics to the generic SIMD intrinsic interface so that codes developed based on the library are portable across different SIMD platforms.

We have evaluated the library with several applications from the multimedia, data analytics and math domains. Compared with platform-specific intrinsics codes, using Generic SIMD Library results in less line-of-code, a 22% reduction on average, and achieves similar performance as platform-specific intrinsics versions.

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures*]: SIMD

General Terms Languages, Performance

WPMVP'14, February 15 - 19, 2014, Orlando, FL, USA.

http://dx.doi.org/10.1145/2568058.2568059

Keywords SIMD, Altivec, SSE, AVX, Generic Programming

1. Introduction

SIMD (Single Instruction Multiple Data) is a classic computation mode that performs the same operation over multiple data simultaneously. SIMD was firstly introduced for vector machines in the domain of supercomputing decades ago. Today, SIMD processing is an indispensable feature of most commercial micro-processors including processors on mobile platforms.

While conventional Instruction Set Architecture (ISA) for modern processors is stabilizing, ISA for SIMD processing is evolving rapidly over the last decade by enriching arithmetic operations, widening vector width, and adding support for predicated execution. For instance, between 2011 and 2013, SIMD vector width for Intel processors have gone from 128-bit for SSE, to 256-bit for AVX, and to 512-bit for MIC and AVX2. MIC and AVX2 have also introduced predicated execution and gather/scatter to future expand the scope of SIMDizable computation. Similar expansion of SIMD ISA also happens on POWER and ARM processors.

With the rapid development of SIMD hardware, programming models for SIMD have also evolved significantly:

- **Auto-vectorization.** This is the holy-grail of SIMD programming model, where the compiler automatically identifies parallel regions in an otherwise sequential program and generates SIMD codes for a target SIMD hardware. This approach was most actively pursued in the research and industry community before 2010. As the result, three major industry compilers, `gcc` [14, 15], `icc` [3, 4], and `xlc` [6, 7], all have auto-vectorization capabilities today.

  Auto-vectorization is a mature technology that works well on simple kernels and regular loops with no unknown side effects. However, it is often ineffective in real codes compiled under typical optimization flags.

- **SPMD (single-program-multiple-data) on SIMD.** This programming model includes OpenACC[18], OpenCL[10],

OpenMP[5, 11, 16], ISPC[19], and CUDA. Under the SPMD-on-SIMD model, programmers specify the parallelism and the compiler takes care of SIMD code generation to achieve portability and a reasonable performance.

We believe SPMD-on-SIMD is the SIMD programming model of the future, but will take some time to mature.

- **SIMD intrinsics.** This is a low-level SIMD programming interface, but also is the first and most basic programming interface that any SIMD platform provides. With this interface, programmers can write assembly style SIMD codes in a high level language.

  Despite the perceived superiority of the other two programming models, most SIMD programs today are still written using SIMD intrinsics because other programming models are either un-reliable or not very mature.

In this paper, we focus on improving the SIMD intrinsic programming model (the least interesting among the three described above). We believe intrinsics programming is the programming model for SIMD today. Furthermore, even after SPMD-on-SIMD has matured, intrinsics programming is still likely to be the choice for certain types of programmers (such as library developers) that demand maximum performance out of SIMD hardware.

Despite its importance, little consideration is put into the design of the SIMD intrinsics interface. All vendor-provided SIMD intrinsics are defined as a direct mapping to the underlying SIMD ISA. As a result, we observe three limitations of the current intrinsics interface:

- **Low-level and tedious** Because of the one-to-one mapping between intrinsics and the underlying SIMD ISA, intrinsics programming is like embedding assembly codes in a high-level language and is often tedious and complex. For example, there are no intrinsics to directly add a scalar to a SIMD vector despite the commonness of such an idiom. Instead, the programmer has to first create a SIMD vector from a scalar using a splat or smear intrinsic, and then apply a vector add intrinsic.
- **Non-portable** Each processor defines its own set of SIMD intrinsics based on its own SIMD ISA. As a result, programs using a SIMD intrinsics interface for one ISA are not portable to another SIMD platform. Furthermore, even on the same platform, programming with older generation intrinsics APIs cannot benefit from the new features introduced in the latest generation processors. Because intrinsics programming introduces platform-specific codes, some application developers resisted exploitation of SIMD in their codes.
- **Fixed byte-length vectors** The SIMD register width limits the maximal data set one SIMD instruction can process. For example a 128-bit SIMD instruction set can process two doubles or four integers in one operation. If a programmer wants to use SIMD to calculate the indices (integer type) and use them to process some double values, he either has to use only half of the integer vector, or need manually duplicate/unroll the double type operations.

We propose Generic SIMD Library as a simple solution to address the three limitations described above. The library offers a set of simple C++ based generic interfaces to program SIMD. The key idea here is to define the interface based on C/C++ language primitives instead of a SIMD instruction set. Our work is inspired by the internal generic intrinsics interface used in the ISPC compiler [19] and the Boost.SIMD library [8] (see Section 6 for details). This interface provides an abstraction of vector data types that hide platform specific register types and instructions, and common operations over them using C/C++ primitive operations. The data types and APIs in Generic SIMD Library are simpler and more intuitive compared with the low level instrincs. One important goal of programming SIMD is the performance. By carefully implementing the library and using compiler tricks, Generic SIMD Library introduces nearly no performance overhead.

We evaluated the library with several applications from different categories, including multimedia, data analytics and math, on both Intel platform and IBM Power platform. Compared with the original intrinsics implementation, the Generic SIMD Library based implementation has less line-of-code (22% reduction in average), and nearly the same performance as the intrinsics one. The same version of code can be compiled and run on multiple platforms without any modification, and achieve the similar speedup over the reference scalar code.

The remainder of this paper is organized as follows. Section 2 summaries the problems of directly programming intrinsics. Section 3 presents the design of the abstraction layer of Generic SIMD Library. Section 4 describes the implementation details. The preliminary evaluation results is reported in Section 5. Related work are described in Section 6 and Section 7 concludes with future work.

2. Motivation

Current SIMD intrinsics are designed primarily as a direct mapping of the underlying SIMD ISA. As a result, the interface incurs three inconveniences to a typical programmer: low-level assembly-like programming style, non-port codes, and dealing with fixed width vectors for codes that involve primitive data of different sizes.

To illustrate these limitations, consider a simple example of RGB2Gray computation. List 1 is the original serial code.

Listing 1. RGB2Gray - Original Serial Code Version

```
1 serial_rgb2gray(float* ra, float* ga, float* ba, float* gray) {
2   for(int i = 0; i < N; i++) {
3     gray[i] = 0.3f * ra[i] + 0.59f * ga[i] + 0.11f * ba[i];
4   }
5 }
```

Listing 2 and 3 are SIMD versions of the previous code for Intel's SSE and IBM POWER's VSX respectively (assuming N is a multiple of 4).

Listing 2. RGB2Gray - SSE Version

```
1  void sse_rgb2gray(float* ra, float* ga, float* ba, float*
       gray) {
2    __m128 c1 = _mm_set1_ps(0.3f);
3    __m128 c2 = _mm_set1_ps(0.59f);
4    __m128 c3 = _mm_set1_ps(0.11f);
5
6    for(int i = 0; i < N; i+=4) {
7      __m128 a = _mm_loadu_ps(ra+i);
8      __m128 b = _mm_loadu_ps(ga+i);
9      __m128 c = _mm_loadu_ps(ba+i);
10     __m128 ab = _mm_add_ps(_mm_mul_ps(c1, a),
           _mm_mul_ps(c2, b));
11     __m128 out = _mm_add_ps(ab, _mm_mul_ps(c3, c));
12     _mm_storeu_ps(gray+i, out);
13   }
14 }
```

Listing 3. RGB2Gray - VSX Version

```
1  void intrinsics_rgb2gray(float* ra, float* ga, float* ba,
       float* gray ) {
2    __vector float c1 = vec_splats(0.3f);
3    __vector float c2 = vec_splats(0.59f);
4    __vector float c3 = vec_splats(0.11f);
5
6    for(int i = 0; i < N; i+=4) {
7      __vector float a = vec_vsx_ld(0, ra+i);
8      __vector float b = vec_vsx_ld(0, ga+i);
9      __vector float c = vec_vsx_ld(0, ba+i);
10     __vector float out = c1 * a + c2 * b + c3 * c ;
11     vec_vsx_st(out, 0, gray+i);
12   }
13 }
```

The first problem is the assembly-like low-level programming style. For example, in order to multiply a vector with a scalar, the programmer has to construct a vector from the scalar first (Line 2,3,4 in Listing 2 and Listing 3). To access and operate on SIMD data, the programmer has to explicitly call the load and store intrinsics (Line 7,8,9,12 in Listing 2 and Line 7,8,9,11 in Listing 3) and other intrinsics for basic arithmetic operations over SIMD vectors (Line 10,11 in Listing 2). All of these combined result in a SIMD code that is much more complex and more difficult to maintain than the original scalar code. Furthermore, this level of intrinsics programming requires a deep knowledge of a target SIMD instruction set, which most typical programmers lack. The lack of programmers with the right skill sets creates a practical barrier to the adoption of this programming interface.

The second problem is the lack of portability across different SIMD platforms. In order to explore SIMD across platforms, application developers today have to develop multiple versions of a code, one for each target platform, such as the ones shown in Listing 2 and Listing 3. Sometimes, application developers are simply not willing to introduce platform-specific codes into the software, thus giving up the opportunity to explore SIMD. Furthermore, even if an application is intended for one processor family, a program written for an older generation of SIMD engine (e.g., SSE4) can not automatically benefit from a newer generation of SIMD hardware (e.g., AVX), adding to software maintenance cost.

The last limitation is the use of fixed-width vectors. The SIMD vector width defines the byte-length of data that a SIMD instruction can process. For example a 128-bit SIMD instruction can process two doubles or four integers in one operation. While a fixed byte-length vector is convenient to implement in the hardware, it is unnatural for parallelizing a software. Consider the case of a parallel loop that operates on two data types of different lengths, such as integers and doubles. The natural way to parallelize the loop for SIMD is to process a *fixed number* of data instead of a *fixed byte-length* of data. Consider a 128-bit SIMD intrinsics interface, in order to parallelize the loop with a blocking factor of 4, the programmer needs to fit 4 doubles to two SIMD vectors and manually split a vector of 4 integers to operate with two vectors of doubles. Such maneuver to map parallel regions with mixed data types to fixed byte-length SIMD vectors increase the complexity of SIMD codes.

3. Generic SIMD Library Design

In order to solve the problems of directly programming SIMD intrinsics, we designed and implemented the Generic SIMD Library. The key idea is to define an abstraction layer to hide platform specific register types and intrinsics, and provide short vector types and common operations over them. Figure 1 depicts the high level architecture of the library. The key features provided include **Short vector** data types and common operations over these types, **Fixed lanes**, not fixed-register-bit-width for basic data types, **Scalar like syntax**, and simple functions to provide C like semantics. We believe the library and computation kernel developers who want to use explicit vector style parallelism could benefit the most from the Generic SIMD Library.

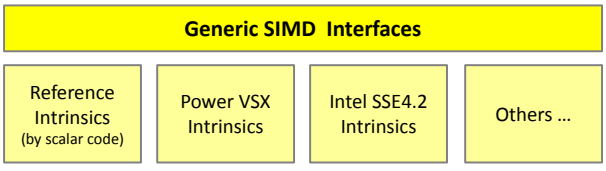


Figure 1. Generic SIMD Library Architecture

3.1 Short vector

Short vector is the basic data type abstraction in Generic SIMD Library. The basic syntax for short vector is `svec<N,type>`. `N` must be a power of 2, and `type` should be a basic primitive type, such as bool, integer, float, double, etc.. The library handles all the storage mapping from the abstraction data types to the platform dependent registers.

The library provides several ways to construct a short vector, including constructing from N scalars, splashing one scalar to a N-element vector and loading a short vector directly from a memory address, as shown in Listing 4.

Listing 4. Construct a Short Vector Version

```
1 svec<4, bool> mask(1); // vector with four true
2 svec<4, int> index(0,1,2,3); //vector with four integers
3 svec<4, double> data
4       = svec<4, double>::load((svec<4, double>*)aPtr);
5       //load a double vector from a memory address
```

Generic SIMD Library provides APIs to cover most of the common operations. Table 1 is a summary of these APIs. The detailed list is available in the library documentation.

Because the library is based on C++, we provide two styles of APIs: plain function calls, and object instance methods, to operate on short vectors. For example, both `svec_sqrt(aVec)` and `aVec.sqrt()` are valid APIs to get the square root of the short vector `aVec`. In fact, the second form is mapped to the first form internally. One important design decision is to have the short vector as a value object, which means that except the case of modifying single elements of a short vector (by `svec_insert()` or `[]` operator), all the operations on a short vector will return a new short vector. For example, the previous `aVec.sqrt()` returns a new short vector and the value of `aVec` is not changed.

3.2 Fixed lanes

The SIMD short vectors are defined based on the number of elements per vector (fixed-lane) instead of the byte-length of a vector (fixed-width). This is the key difference between our vector types and the ones defined in platform-specific intrinsics.

We choose fixed-lane vector because it is more natural to SIMDized parallel loops that involve data of different length such as int and double. We intend to support vectors with arbitrary power-of-two lanes, but currently only 4-element vectors are supported. Vector of 2-, and 8-elements are still under development.

3.3 Scalar like syntax

We define SIMD vector operations based on semantics of C++ operators instead of platform-specific ISA semantics. This is because the semantics of C++ operators are platform independent. Secondly, C++ operators provide higher semantics than platform-specific intrinsics and are more natural to program since most users understand C++ operators already. Thanks to the C++ operator overloading mechanism, programmer can write the scalar like code to add two vectors through '+', shift vectors by >>or <<, etc..

3.4 Platform Independent

The data types and interfaces defined in the library are completely platform neutral. The library provides mapping from the data type and interface to target SIMD platforms, which means the application based on our library can be compiled and run without any source code modification (only change a compilation flag) in different platforms. In fact, our library APIs is a super set of the platform dependent intrinsics. If one specific API operation has no corresponding SIMD instrinsics supported in one platform, we implement the API through scalar code.

Listing 5 is the RGB2Gray example based on our library. With all the features provided, the code is simpler and more intuitive compared with the intrinsics based implementation. Figure 2 presents the process to compile the code on different platforms.

Listing 5. RGB2Gray - Generic SIMD Version

```
1  #include <gsimd.h>
2
3  void svec4_rgb2gray_ptr(float* ra, float* ga, float* ba,
       float* gray ) {
4    for(int i = 0; i < N; i+=4) {
5      svec<4,float> a = *(svec<4,float>*)(ra+i);
6      svec<4,float> b = *(svec<4,float>*)(ga+i);
7      svec<4,float> c = *(svec<4,float>*)(ba+i);
8      svec<4,float> out = 0.3f * a + 0.59f * b + 0.11f * c ;
9      *(svec<4,float>*)(gray+i) = out;
10   }
11 }
```

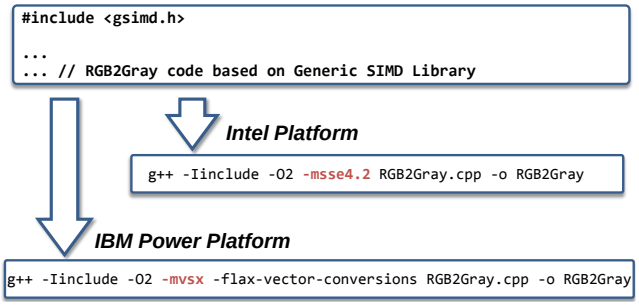


Figure 2. Compile Code to Different Platforms

4. Implementation

We have implemented Generic SIMD Library using C++. All the library code is purely inside header files, The source code is available at https://github.com/genericsimd/generic_simd/. The current version has IBM Power VSX, Intel SSE4.2 and a reference (scalar implementation) back-ends implemented.

By leveraging the operator overloading and object oriented features of C++, the library's data types and APIs can

Table 1. Common Operations of the Short Vector

Category	Description
Construction	With N scalars or 1 scalar to splat
Elements access	[] operators, or svec_extract(), svec_insert()
Load and Store	User pointer de-reference or use load/store methods
Compare	Operators ==, !=, >, =, <, >=, and return a svec<N, bool>vector
Bit and logical operations	&, |, ^ for all integer and bool vector types; !, &&, ||for bool vector types
Math operations	+, -, *, /, >>, <<: between vector and vector, vector and scalar. FMA among three vectors
Instance methods(Map)	broadcast(), rotate(), shuffle(), round(), floor(), ceil(), sqrt(), exp(), log(), ...
Instance methods(Reduce)	reduce_add(), reduce_max(), reduce_min(), ...
Gather and scatter	Load and store non-continuous elements
Select operations	Compose a new vector by selecting elements from two
Type Cast	Explicit type casting functions between any two vector types

be used in a simple and clear style. Putting all code inside the header files greatly lowers the distribution and usage complexity. The user only needs to include the "gsimd.h" header file to use the library. There is no installation required.

We briefly describe some implementation techniques used in the library.

4.1 Mapping Data Types

Because our library provides fixed-lane SIMD vectors, it's impossible to directly do a 1-to-1 mapping from the short vectors of Generic SIMD Library to the underlying architecture registers. Let's consider 128bit width architecture register, and 4-element Generic SIMD Library short vector as example. If the bit width of the scalar is less or equal than 32 bits, we always map four scalars into one architected SIMD register. Based on the implementation, the architecture SIMD register may waste some space, for example `svec<4,short>` only uses half of the space. If the bit width of the scalar is larger than 32bit, for example double and long long, we use more than one architected SIMD registers to represent the Generic SIMD Library vector. All the operations of these types will iterate over all the architected registers, similar to unrolling the SIMD intrinsic operations. `bool` type is mapped to `unsigned int` type. Based on the mapping relationship, most of the commonly used data types bool, integer, float, long long and double are all perfectly mapped to the underlying architecture registers without wasting.

4.2 Mapping API Operations

Mapping API operations of the library to the underlying platform intrinsics is relatively straight forward. Many operations can be mapped to the intrinsics APIs directly. We use more than one intrinsics API calls to implement one generic operation sometimes. For example, adding a scalar to a vector requires 1) splatting the scalar to a vector, 2) adding the new vector to the other input vector.

If one generic operation has no corresponding intrinsics support, we use scalar code to implement the semantics. This way, we implement the full gather/scatter operations, and predicated (masked) operations of load/store and vector compare. Users can utilize these APIs to simplify their SIMD programming. When the processor has the hardware support for these semantics, the program can directly get the performance benefit without source code modification.

4.3 Use Compiler Internal APIs

Some of the intrinsics require compile-time constant values as operand. For example, `vec_splat_s32(a)` from Power AVX intrinsics construct a SIMD register by splatting value **a** in one instruction. But the requirement is that **a** must be a compile time constant value ranging from -16 to 15. We cannot directly call this intrinsics in constructing a vector from an arbitrary scalar value. We can always use standard way to construct a vector by repeating the scalar four times. But the later method requires more than one instructions, and suffers the performance loss if `a` is a compile time constant ranging from -16 to 15. Thanks to the GCC's `__builtin_constant_p()` compiler API, we can use the proper method to construct a vector based on the value of the scalar. List 6 is such an example.

Listing 6. Using Compiler APIs to Improve Performa

```
1 svec( int16_t a) {
2   if(__builtin_constant_p(a) && (a <= 15) && (a >=
        -16)){
3     v = vec_splat_s16(a); //will gen one instr.vspltisb
4   } else {
5     //Standard method to construct a SIMD register
6     __vector signed short t = {a,a,a,a, 0,0,0,0};
7     v = t;
8   }
9 }
```

4.4 Support Sub Vector ISA under the Same Platform

The same platform may have several generations of intrinsics, such as SSE, AVX, AVX2 on the Intel platform. The same the interface in our library may be mapped to different generations of intrinsics. For example, the `add` opera-

tion of `svec<4,double>` is mapped to two intrinsics calls of SSE4.2 and only one intrinsics call of AVX. We still use compiler flags to control how the library operations are mapped to the corresponding intrinsics. Different compiler flags, such as -msse4.2 and -mavx, force the definition of macros, such as `__SSE4_2__` and `__AVX__`, which is used to control the library operation to sub vector ISA mapping.

5. Evaluation

We evaluated the effectiveness of Generic SIMD Library on several benchmarks across two platforms, and answered the following questions:

- **Simple** Does the library simplify the SIMD programming?
- **Fast** Does the abstraction layer in the library introduce additional overhead?
- **Portable** Does the application using the library can be compiled correctly and maintain the similar speedup over the serial code on different platforms?

5.1 Environment, Application and Methodology

Table 2. Evaluation Platforms

Platform	IBM Power	Intel
Processor	Power 7	i7-2600K Sandy Bridge
Frequency	3.1GHz	3.4GHz
OS kernel	Linux 3.0.1	Linux 3.5.3
Compiler	GCC 4.8.1	GCC 4.7.2
Compiling Flag	-O2	-O2

The evaluation was performed in two platforms, listed in Table 2. Our evaluation includes five applications from different categories. Table 3 summaries the applications. The last two columns indicate the availability of the original intrinsics version. For example, RGB2Gray has both IBM Power VSX intrinsics implementation and Intel SSE4.2 intrinsics implementation, while Mandelbrot only has Power VSX implementation.

Table 3. Applications

Name	Category	Data Type	VSX	SSE4.2
RGB2Gray	Multimedia	Float	X	X
Mandelbrot	Multimedia	Float	X	
SPSS-Sweep	Data analytics	Double	X	
MAG-CVA-Mean	Data analytics	Float		X
Dgemm	Math	Double	X	

We rewrote the intrinsics implementation to the version using Generic SIMD Library. The translation maintains the original algorithms.

The baseline is the serial version of all the applications. Each application has the same serial code for both Intel and IBM Power platforms. Please note that the serial code is based on the same algorithm of the SIMD version, which means the serial code may not be the best optimized implementation. But it is a good measurement of SIMD speedup compared to the serial version.

The performance number reported in the following sections are normalized to the speedup of the serial version on the corresponding platforms.

5.2 Simple

After rewriting with APIs of Generic SIMD Library, most of the applications have less line-of-code. Table 4 contains the detailed measurement of the SIMD functions. The original data type of SPSS-Sweep and Dgemm is double. The original intrinsics code uses 128bit registers, which means every operation can only process two elements. The Generic SIMD version uses four lanes. As a result, the two benchmarks have most of the LOC reduction rate. In average, the Generic SIMD version achieved 22% LOC reduction, which is an indicator that Generic SIMD Library simplifies the SIMD programming.

Table 4. Line-Of-Code of the SIMD functions

Application	Intrinsics	Generic SIMD	Reduction
RGB2Gray	14	10	28%
Mandelbrot	104	100	4%
SPSS-Sweep	418	256	39%
MAG-CVA-Mean	141	141	0%
Dgemm	83	51	39%

5.3 Fast

We compared the Generic SIMD version code with the original intrinsic version code in the platform where the original intrinsics version is developed. Figure 3 shows the SIMD versions' (intrinsics and Generic SIMD) performance speedup over the serial code on IBM Power platform, and Figure 4 shows the Intel one. The speedup of the Generic SIMD Library code is very similar to the intrinsics one, which proves that the abstraction layer in Generic SIMD Library introduces almost no additional overhead. Regarding the over 4x speedup of RGB2Gray on Intel Platform, it is because Intel Sandy Bridge processor has two 128-bit SIMD execution lanes.

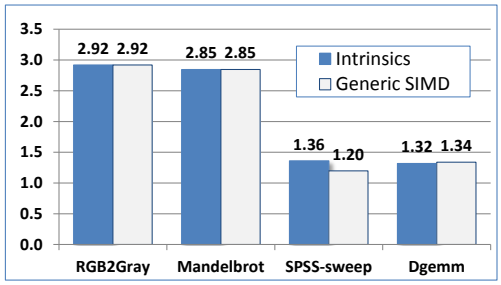


Figure 3. Speedup to Serial Code - IBM Power Platform

5.4 Portable

Finally, we compiled and run the the Generic SIMD version applications on the platforms without intrinsics implementation. For example, there is no Intel intrinsics implementation of Mandelbrot, SPSS-Sweep and Dgemm, and we just

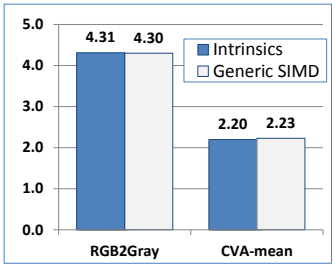
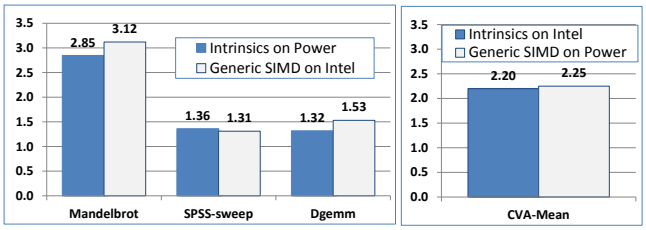


Figure 4. Speedup to Serial Code - Intel Platform



(a) Power to Intel Platform (b) Intel to Power

Figure 5. Speedup to Serial Code - Cross Platforms

compile the Generic SIMD version code ported from IBM Power intrinsics and run them directly on Intel Platform. Figure 5(a) shows the result. The bar on the left are the speedup of the Generic SIMD code on Power Platform over the serial code on Power Platform, and the bars on the right are the speedup of the Generic SIMD code on Intel Platform over the serial code on Intel Platform. All three applications still maintains the similar speedup over the serial code. Figure 5(b) shows the CVA-Mean ported from Intel to Power Platform. The result proves that the application using the library is portable and can maintain good speedup in different platforms.

6. Related Work

6.1 Auto SIMD and Auto Vectorization

Due to the complexity of directly programming SIMD intrinsics, there is a long history of using compilers to automatically identify code regions, and translate them into vectorized code with SIMD instructions. All the major commercial and open source compilers, such as ICC, XLC, GCC, LLVM, have the auto vectorization capability. However, a recent evaluation [13] shows that "despite all the work done in vectorization in the last 40 years, 45-71% of the loops in the synthetic benchmark and only a few loops from the real applications are vectorized by the compilers we evaluated". In order to full utilize the power of SIMD units, manually SIMD programming is still required.

6.2 SPMD on SIMD

SPMD (Single Program Multiple Data) on SIMD, such as OpenMP[5], OpenACC[18], OpenCL[10] and ISPC[19], enables the scalar style programming, and the constructs/compiler help map the scalar program to multiple data streams. OpenMP and OpenACC use *PRAGMA* compiler directives and additional functions to guide the compiler automatically identify the parallelizable region, and map the code into SIMD units or multi-threads. OpenCL and ISPC define a new programming model that let the programmer focus on programming the serial code of single unit's processing (computation kernel), and the compiler and runtime map the computation kernel into parallel execution units, including but not limited to SIMD units. Both of these approaches reduce the complexity of SIMD programming, maintain the flexibility, and achieves reasonable good performance.

6.3 Platform Specific Intrinsics Library

Because of the complexity of directly programming platform intrinsics, there are a lot of efforts of providing better abstractions to the intrinsics to lower the programming efforts, such as SSEPlus[2], CppVector[9], Cross Platform SIMD Math[1], and VMath [17]. SSEPlus provides additional virtual intrinsics that has no hardware support yet. Programs using these virtual intrinsics can simplify the programming, and immediately benefit from the hardware as soon as the intrinsics is supported in the processor. The other three have the similar high level idea, which provides easy to use interfaces, and hides the low level APIs. All of them use C++ to define the data abstraction by class, and use operator overloading to simplify the intrinsics API call.

However, all the above only target X86 platform, and have no portable feature compared with Generic SIMD Library.

6.4 Platform Independent Intrinsics Library

Platform independent intrinsics library also has a layer to hide the detail of the platform intrinsics. The design of the abstraction also considers the requirements of crossing different hardware platforms. Generic SIMD Library is such a platform independent library.

Cross-platform SIMD C Headers[20] is a simple platform independent SIMD library. It has Intel and ARM intrinsics back-ends. However, it only uses macro to map high level interfaces to the intrinsics APIs, no data abstraction layer. It only provides a small subset of APIs, no operator overloading. The programming is still similar to the original intrinsics programming,

Similar to our library, Boost.SIMD[8] offers `Pack` to express short vectors, and provide different APIs and operators over the `Pack`. But Boost.SIMD focuses more on the flexibility and integration with C++ STL components. It aggressively uses many other C++ libraries to achieve the goal. As a result, Boost.SIMD's code base is much larger than our library, and it requires installation. The syntax and APIs are also complex than our library's APIs. Our library focuses more on simple and self contained, and there is no additional installation process required.

Most of the libraries only provide abstractions of vector data objects and operations over them. The complexity of coveting serial control flow code to vector version still limits the productivity of SIMD programming. IVL language [12] tried to solve the problem by using a type system to steer vectorization of control structures. Our library does not have the feature, and we plan to explore the possibility to simplify control flow coding in a pure library approach in the future.

7. Conclusion

This paper has described Generic SIMD Library. The library defines an abstraction layer to hide platform specific data types and intrinsics, and provides short vector types and common operations over them. The code based on generic SIMD library is portable across different hardware platforms. The evaluation shows the application using Generic SIMD Library is simple, fast and portable.

Traditionally, SIMD is mainly targeted to workloads of multimedia and regular math routines. In the era of Big Data, the new emerging workload is from data analytics domain. The characteristics of the workload has shifted from regular data access and unified control flows to irregular data access and control divergences. On the other hand, new SIMD features are added into the processors, such as gather and scatter, and predicated (masked) operations. There are many open questions raised in the context. Is SIMD still able to accelerate data analytics workload? What's the additional support we require from the hardware? Our future work will focus on these two questions to improve Generic SIMD Library's capability.

References

[1] AllEightUp. Pratical cross platform simd math. http://www.gamedev.net/page/resources/_/technical/general-programming/practical-cross-platform-simd-math-r3068.

[2] AMD. Sseplus project. http://sseplus.sourceforge.net/.

[3] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, April 2002.

[4] Aart J. C. Bik, David L. Kreitzer, and Xinmin Tian. A case study on compiler optimizations for the intel core 2 duo processor. *Int. J. Parallel Program.*, 36(6):571–591, December 2008.

[5] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. http://www.openacc-standard.org/.

[6] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the CELL processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[8] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost.SIMD: generic programming for portable SIMDization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 431–432, New York, NY, USA, 2012. ACM.

[9] Agner Fog. C++ vector class library. http://www.agner.org/optimize/#vectorclass.

[10] Khronos Group. The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[11] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. Extending OpenMP* with vector constructs for modern multicore SIMD architectures. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 59–72, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Roland Lei, Sebastian Hack, and Ingo Wald. Extending a c-like language for portable simd programming. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 65–74, New York, NY, USA, 2012. ACM.

[13] S. Maleki, Yaoqing Gao, M.J. Garzaran, T. Wong, and D.A. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, 2011.

[14] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[16] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on CELL. *Int. J. Parallel Program.*, 36(3):289–311, June 2008.

[17] Gustavo Oliveira. Designing fast cross-platform simd vector libraries. http://www.gamasutra.com/view/feature/132636/designing_fast_crossplatform_simd_.php?print=1.

[18] Open ACC Standard Orgnaization. OpenACC directives for accelerator. http://openmp.org/wp/.

[19] M. Pharr and W.R. Mark. ISPC: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, 2012.

[20] Patrick Roberts. Cross-platform SIMD c headers. http://simd-cph.sourceforge.net/.