Data layout and SIMD abstraction layers: decoupling interfaces from implementations

Sylvain Jubertie
Univ. d'Orléans, INSA Centre Val de Loire
LIFO EA 4022, Orléans, France
sylvain.jubertie@univ-orleans.fr

Ian Masliah
Sorbonne University, CNRS
LIP6, Paris, France
ian.masliah@lip6.fr

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

Abstract—**From a high level point of view, developers define objects they manipulate in terms of structures or classes. For example, a pixel may be represented as a structure of three color components : red, green, blue and an image as an array of pixels. In such cases, the data layout is said to be organized as an array of structures (AoS). However, developing efficient applications on modern processors and accelerators often require to organize data in different ways. An image may also be stored as a structure of three arrays, one for each component. This data layout is called a structure of array (SoA) and is also mandatory to take advantage of SIMD units embedded in all modern processors.**

In this paper, we propose a lightweight C++ template-based framework to provide the high level representation most programmers use (AoS) on different data layouts fitted for SIMD vectorization. Some templated containers are provided for each proposed layout with a uniform AoS-like interface to access elements. Containers are transformed into different combinations of tuples and vectors from the C++ Standard Template Library (STL) at compile time. This way, we provide more optimization opportunities for the code, especially automatic vectorization. We study the performance of our data-layouts and compare them to their explicit versions, based on structures and vectors, for different algorithms and architectures (x86 and ARM). Results show that compilers do not always perform automatic vectorization on our data-layouts as with their explicit versions even if underlying containers and access patterns are similar. Thus, we investigate the use of SIMD intrinsics and of Boost.SIMD[1]/bSIMD libraries to vectorize the codes. We show that combining our approach with Boost.SIMD/bSIMD libraries ensures a similar performance as with a manual vectorization using intrinsics, and in almost all cases better performance than with automatic vectorization without increasing the code complexity.

Index Terms*—*data layouts, AoS SoA AoSoA, SIMD, vectorization

I. Introduction

When developing high performance applications, choosing the best data layout to extract full performance from the underlying architecture is a very critical and challenging task. Such a choice requires a deep knowledge of both the application data access patterns and the machine architecture. Historically, supercomputers rely on SIMD (Single Instruction on Multiple Data) units: a set of large registers and processing units able to apply a single instruction to multiple elements of the same register at the same time, to improve their computing power. The performance benefit brought by such units is important for algorithms containing suitable patterns (image/signal processing, linear algebra, ...). Today, almost all modern processors, whether they are in smartphones or supercomputers, contain SIMD units(SSE/AVX on x86, NEON on ARM, VMX on Power). Several ways exist to take advantage of these SIMD units. Compilers contain automatic vectorizers to detect patterns suitable for vectorization. If the compiler is not able to vectorize the code, it is possible to use intrinsics which are architecture specific functions mapping directly SIMD instructions or to use higher levels library like Boost.SIMD[1]. However, to take advantage of these units, data have in general to be organized using a SoA data layout.

When developing applications, the most popular programming paradigm, promoted by languages such as C++, Java, C#, is the Abstract Data Structures (ADS) one. In this case, data are organized in structures or classes with their attributes stored contiguously in memory, i.e. using a AoS data layout. Such a layout presents several advantages. Data in memory is accessed by blocks called cache lines (blocks of 64 bytes in common architectures), thus, attributes of a same structure are likely to be stored in the same cache line or in contiguous cache lines and accessing them may only require a single memory transaction. Since the data are likely to be accessed contiguously, the processor prefetcher can reduce cache misses by anticipating the load of the next cache line. Having a structure such as AoS where data can be traversed contiguously is the most cache friendly method. However, the cache is not the only aspect to take into account when optimizing the performance of an algorithm, SIMD units have also to be considered. Thus, it appears an antagonism between programming using the high level ADS paradigm and taking advantage of low level SIMD units i.e. between programming efficiency and performance.

In this paper, we propose to solve this problem by decoupling the high level view of the layout from the underlying implementation. We provide a single representation of the data layout to the developer based on the ADS paradigm, thus a AoS view. The developer only has to choose the underlying data layout implementation. As an example, an AoS representation of the data structure may be transformed in a different layout just by modifying a type definition. The benefits brought by this approach are threefold: **1) portability,**

[1]The Boost.SIMD library is available at https://github.com/NumScale/boost.simd

adaptability: the code remains the same and is independent of the underlying architecture, **2) productivity, readability**: we offer a single ADS view of the data layout, a unique code allows to test the different layouts, **3) performance**: the data layout may be optimized for a particular architecture, SIMD unit, etc. To be easily integrated and transparent to the developer, we choose to build our approach using a C++ header only library. This is made possible by using C++ template mechanisms and C++11 language features. We test our approach on several codes using different compilers on different architectures and we verify if the produced code is automatically vectorized. In all cases, we also consider manually vectorized versions of our codes using intrinsics and the Boost.SIMD library to compare to the versions produced by the compilers.

The paper is organized as follows. First, we discuss the related work on data layouts in HPC applications and on similar data layout abstraction approaches. Then, we present our approach and describe our implementation as well as its interface. We test several applications to validate our approach. With each test we explain the different data structure transformations and the vectorization possibilities. Experimental results on different architectures with several compilers are also presented and discussed. Finally, we conclude on our results and the benefits and limits of our approach and present some future work.

II. Related work

We identify four different approaches to generate optimized data layouts:

1) Explicit implementations: The first one is a low level approach where the data layout is optimized for a given application on a given architecture and is explicitly exposed to the developer. This is the common approach traditionally used in HPC applications written in C, C++ or Fortran codes. For example, in [2], the authors investigate different data layouts to improve the performance of Lattice Boltzmann kernels on multi- and many-core architectures. In [3] the authors study the scalability provided by different data layouts on a molecular dynamic simulation written in Java on a multi-core architecture. In both cases, testing a different layout is performed by rewritting the computation kernel.

2) High level approaches: High level approaches consists in using specific languages and compilers to perform automatic data layout transformations. In [4] and [5], the authors use the Habanero-C framework to design a system that can automatically generate an efficient data layout and kernel mapping on heterogeneous architectures. Using an automatic approach is interesting and yields good results. However, it relies on specific programming environments i.e. compiler and languages, which may limit the integration of such a tool.

3) Guided approaches: Another approach is based on adding pragma extensions in the code. This is an in-between model that provides a semi automatic composition for data layout transformation. An example of this type of approach is described in [6]. The framework problem is still apparent with this approach, and the pragma extensions can be quite complex when mixing data layout and SIMD pragmas.

4) Domain Specific Languages: An approach followed recently by Intel relies on providing an embedded Domain Specific Library (eDSL) through a library that comes with their standard tools and C++ compiler. The Intel SIMD Data Layout Templates library (SDLT) [2] is a C++ template library which provides containers in both AoS and SoA versions. Changing the layout of the container only requires to change the type of the container. The container interface is very similar to the one of STL containers. The problem is that this library is tied to Intel processors and on its compiler, thus limiting portability.

The Kokkos library[7], [8] is also a C++ library which proposes multidimensional arrays parametrizable through the use of template parameters. It is possible to specify the data layout and the locality of the underlying memory to target different architectures like CPUs or GPUs. The vectorization process relies also on the underlying compiler.

We use a similar approach for our data layout abstraction layer as the one used by Kokkos and the SDLT library. To ensure proper vectorization, we propose to combine this layer with the Boost.SIMD[1] library as the SIMD abstraction layer. The VC[9] and the UME::SIMD[10] libraries proposes a similar approach with different interfaces and support SSE, AVX and AVX-512 instruction sets but are less generic as they still expose explicitly the size of the SIMD registers while the Boost.SIMD library provides a generic `pack` container. The Boost.SIMD library only support SSE and AVX instruction sets but a proprietary version, called bSIMD adds the support for the AVX-512, NEON and VMX instruction sets.

III. Abstracting data layouts in C++

Our approach is similar to the one used by the Intel SDLT library previously described. Indeed, we believe that using standard compilers is simpler than modifying a compiler or writing a new one and maintaining it. Using our library only requires to include some header files. Thus, it is also more accessible to the developer since it does not involve some compilation toolchain modifications. We also rely on C++11 features, especially on variadic templates, and on STL containers. Of course, like the SDLT library, we are also limited by the lack of an introspection mechanism in C++, but we propose another way to declare structures and avoid making use of macros. Instead of defining a type as a structure with attributes, then passing this type to the container, our approach consists in defining a container directly from the primitive types of the attributes it contains. The container concept is handled by default by the `std::vector` container. To ensure that data are correctly aligned for the SIMD units, we pass an aligned allocator provided by the Boost.Align library[3] to the vector. It is possible to replace it by a user defined container if it complies with the STL vector interface. The structure concept is handled by the `std::tuple` container.

[2]Data Layout Optimization using SDLT: https://software.intel.com/en-us/articles/data-layout-optimization-using-simd-data-layout-templates

[3]Boost.Align: http://www.boost.org/doc/libs/1_62_0/doc/html/align.html

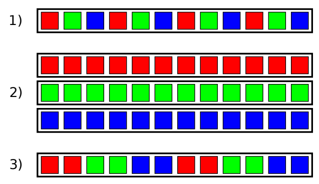

Fig. 1. Layouts: 1) AoS 2) SoA 3) AoSoA

a) Defining and using layouts: We provide three different basic layouts to the user: AoS, SoA, AoSoA (Array of Structures of Arrays, also called hybrid layout). The different layouts are represented in memory as depicted in figure 1. An AoSoA can be viewed as the interleaving of subarrays of a SoA. Thus it contains contiguous blocks of elements thus providing both a cache friendly layout and vectorization opportunities. Their implementations are presented below.

An Array of Structures is described as a vector of tuples:

```
template < typename... TS >
struct aos: std::vector< std::tuple< TS... > >
```

Listing 1. AoS implementation.

An RGB image may be defined as follows:

```
aos< uint8_t, uint8_t, uint8_t > img( size );
```

Listing 2. 3 8-bit color components image definition as an AoS.

In this case the image components are stored contiguously, as described in figure 1. However, this is not a practical way to use our layouts since it exposes the underlying structure at each instantiation. A better approach is to separate the definition from the instantiation by defining an intermediate type:

```
using imageRGB = aos< uint8_t, uint8_t, uint8_t >;
```

Listing 3. Shortcut to simplify image definition.

This way, changing the data layout only requires to modify its definition. It is also possible to put this definition in a separate header to hide it from the user. This separate header may also be automatically generated from micro-benchmarks to determine the definition offering the best performance.

A structure of arrays is described as a tuple of vectors:

```
template < typename... TS >
struct soa: std::tuple< std::vector< TS >... >
```

Listing 4. SoA implementation.

In this case, the image definition is transformed as:

```
using imageRGB = soa< uint8_t, uint8_t, uint8_t >
```

Listing 5. 3 8-bit color components image definition as a SoA.

Each component is stored in a different array, see figure 1.

An AoSoA is an array containing structures of arrays. Its definition must be restricted in order to optimize its implementation. We identify three constraints:

1) the need for a single contiguous container to reduce the work of the prefetcher by only managing contiguous cache lines. It is not possible to store an AoSoA as a vector of tuples of vectors since vectors are allocated dynamically. As a result, the generated layout would be non contiguous in memory. Thus, we describe an AoSoA as a single `std::vector`.

2) the size of the interleaved subarrays. In general, attributes may have different types thus different sizes, and storing them in an AoSoA involves adding padding to respect alignment constraints and to cast attributes when accessing them. Thus, we only consider AoSoA for structures containing the same attribute type. This type is given as the first template parameter of the AoSoA structure. The number of attributes is given as the second template parameter.

3) the fixed length of SIMD registers which impose the size of the subarrays to be a multiple of the register size. The size of the subarrays is given as the third template parameter of the hybrid structure.

This leads to define an AoSoA as follows:

```
template< typename T, size_t N, size_t S >
struct hyb: private std::vector< T >
```

Listing 6. AoSoA implementation.

An image with this layout, with components grouped by 32, is defined as:

```
using imageRGB = hyb< uint8_t, 3, 32 >
```

Listing 7. Image definition as an AoSoA.

The subarray size is chosen to match the size of an AVX register (256 bits) which can contain 32 8-bit unsigned integers. Choosing a size smaller than the register size may prevent the compiler from vectorizing the code. This size may also be determined automatically at compile time to fit the size of a SIMD register.

b) Accessing elements: Accessing elements is performed with the same interface, independently of the chosen layout. It is done through an overloaded version of the operator () which takes two parameters. The first parameter of the operator is the attribute number, ranging from 0 to the number of attributes less one. The second parameter is the position of the element to access. The following example shows how to access to the red component of an image using our containers:

```
for( size_t i = 0 ; i < img.size() ; ++i ) {
  img(integral_constant<size_t,0>{},i) = ...
}
```

Listing 8. Accessing elements using the operator ().

The integral constant is required since we need compile time information on the attribute accessed, but it increases the code verbosity. It is possible to name the attribute for a more explicit code. For example, if we want to name the RGB components of an image using the definition in listing 3, we can define the following integral constants:

```
std::integral_constant< std::size_t, 0 > R;
std::integral_constant< std::size_t, 1 > G;
std::integral_constant< std::size_t, 2 > B;
```

Listing 9. Integral constants for accessing components.

Note that these are independent of the data layout used. We can modify the code in listing 8 in a more simple version:

```
for( size_t i = 0 ; i < img.size() ; ++i ) {
  img(R,i) = ...
}
```

Listing 10. Accessing elements using named attributes.

IV. Experimental results

This section is divided into two parts. In the first one, we present several codes we have chosen for experimenting our approach. For each algorithm we choose several data layout implementations, we provide a version based on explicit structures and vectors, called **explicit**, and a version using our approach, called **abstract**. We explain for each one how it may affect vectorization and performance. We also study assembly codes produced by compilers to verify if they are able to automatically vectorize the codes. Thus, for each code with each layout we generate three different variants: 1) **novec** with vectorization options disabled, by adding the `-fno-tree-vectorize` flag, 2) **auto** with automatic vectorization enabled, by default with the `-O3` flag and with `-march=native` flag to select the proper SIMD unit, 3) **pragma** with an OpenMP SIMD pragma in front of the loops activated by the `-fopenmp` compilation flag. Then, we compare them to the ones generated from hand-written vectorized codes using intrinsics, called **intrinsics**, and Boost.SIMD, denoted **BS**. All data structures are aligned using the Boost.align library according to the SIMD unit considered i.e. aligned on 32 bytes boundaries for the AVX versions, 64 bytes for AVX-512 and 16 bytes for ARM NEON (even if it not a requirement on this architecture). We consider codes produced by different compilers: GCC(6.4), Clang(5.0). For the sake of conciseness, we only detail the complete study on the first code, and we highlight interesting details and unexpected results for the other codes. In the second part we study the performance of these different versions and confront our expectations with experimental results.

A. Data layouts and automatic vectorization

We consider four algorithms in this study which exhibit opportunities for data layout transformations and vectorization. The two first ones are image conversion algorithms: RGB to grayscale and YUV to RGB. These algorithms are used in a lot of image and video processing filters. The third one is vector normalization, which requires few instructions but a square root and a division which are expensive. The last one is a N-Body simulation, which is more complex and known to be compute bound since its algorithmic complexity is $O(n^2)$, with n the number of bodies. In this section, we discuss the assembly codes generated by all compilers for a Broadwell processor, which supports the AVX2 instruction set, with vectorization and optimization flags activated.

a) Grayscale image conversion: It is a simple algorithm which consists for each pixel of an image to take its three components and compute the resulting level of gray component according to the following formula:

$$gray = \frac{307 * R + 604 * G + 113 * B}{1024} \tag{1}$$

Components are encoded as 8-bit unsigned integers. These values are accumulated, thus it is necessary to use 32-bit unsigned integers for intermediate results.

We define an image using our approach as one of the previous definitions in listings 3, 5 and 7. For the AoSoA version, the parameter `S` is set at compile time to the number of 8-bit values fitting into a SIMD register. We translate the above formula into the following code:

```
for( std::size_t i = 0 ; i < rows * cols ; ++i ) {
  uint32_t c = 307 * img(R, i) + 604 * img(G, i) + 113 *
       img(B, i);
  gray[ i ] = c >> 10;
}
```

Listing 11. Grayscale conversion using our approach.

AoS versions: Results show that GCC and Clang produce the exact same assembly code (same instructions in the same order, modulo register renaming) for the AoS code when using the **explicit** and **abstract** versions: the **explicit novec** version is the same as the **abstract novec** one, and the **explicit auto**, **explicit pragma**, **abstract auto** and **abstract pragma** are all the same. Thus, our approach does not introduce extra operations and must provide the exact same level of performance. We observe that GCC and Clang are able to vectorize the code for the **auto** and **pragma** variants, even if the data layout is not SIMD friendly, but the assembly codes are complex and most of the SIMD instructions are used to reorder data before and after performing packed arithmetic operations, thus limiting the possible speedup. It is also interesting to note that in this case our code appears in two different versions in the assembly code: one is inlined and vectorized, the other one is in a grayscale function and is not vectorized. As a consequence, it appears that if the code is compiled into a library, we can expect that the code will not be vectorized.

SoA versions: Both compilers transform the **explicit** and **abstract** versions into the same assembly code for each **novec**, **auto** and **pragma** variants, except for the explicit version using the OpemMP SIMD pragma which is only vectorized by Clang. In this case, we conclude that our approach may sometimes prevent vectorization.

AoSoA versions: For the explicit AoSoA version, we define an image as a vector of structures of static subarrays of size S, where S is set to the number of unsigned bytes fitting the SIMD register:

```
struct sub {
  uint8_t r[ S ];
  uint8_t g[ S ];
  uint8_t b[ S ];
};
using image = std::vector< sub, aligned_alloc< unsigned
     char, 32 > >;
```

Listing 12. AoSoA definition.

This way, components are stored contiguously in contiguous subarrays. This representation requires to add an internal for loop to iterate on subarrays:

```
size_t i = 0;
for( size_t s = 0 ; s < img.size() ; ++s ) {
  for( size_t j = 0 ; j < S ; ++j ) {
    uint32_t c = 307 * img[ s ].r[ j ]
```

```
                + 604 * img[ s ].g[ j ]
                + 113 * img[ s ].b[ j ];
    gray[ i++ ] = c >> 10;
  }
}
```

Listing 13. AoSoA code version.

Note that, since the size of subarrays is known at compile time we can expect the compiler to unroll the internal loop. Moreover, if we set this size to the SIMD register width, we can expect the compiler to vectorize the internal loop. Assembly codes show that our code differs from the one using structures since accessing an element in our hybrid data layout in a scalar way requires to compute the subarray in which an element is stored and its offset. It would require to use integer division and modulo instructions, however compilers are able to replace them with less expensive shift and logical instructions when `S` is a power of two, which is the case here.

As expected, the GCC compiler produces two different assembly codes for the **abstract** and **explicit** versions. However,it is not capable of vectorizing the code. The Clang compiler is the only one able to vectorize the code for both the **auto** and **pragma** variants, but only for the **explicit** layout.We note that Clang produces a different scalar code when using our approach with the pragma activated.

b) YUV to RGB image conversion: The YUV color space is used in lots of digital video equipments to compress and transmit images. The Y component represents the brightness, U and V are color differences. An YUV image needs to be converted into an RGB one to be used by image manipulation tools. Several encodings are possible, in our example we choose the YUV422 one which encodes components as 8-bit unsigned integers interleaved in the following order: Y_0UY_1V. From these four elements we obtain two RGB pixels i.e. from 32 bits we generate 48 bits of information.

AoS versions: Results show that GCC and Clang compilers generate different assembly codes for the **explicit** versions and for the **abstract** versions. Thus, our approach may have an impact on performance. The Clang compiler is able to vectorize the code for the **explicit auto** and **pragma** variants.

SoA versions: GCC generates the same exact assembly codes for the **explicit novec**, **auto**, **omp** variants and their respective **abstract** versions. The **auto** and **omp** variants are the same but are different from the **novec** variant. Thus, for this layout with the GCC compiler, our approach does not induced any overhead. With Clang, the **explicit novec** and **auto** variants are the same. The **abstract novec**, **auto** and **pragma** variants are also the same but are different from their **explicit** versions. We observe that the **explicit pragma** variant is different from the others but is not vectorized.

c) Vector normalization: Normalizing a vector consists in dividing each ot its components by the square root of its length. We consider three dimensional vectors.

The generated assembly files show that none of the compilers is able to vectorize these codes, even in the SoA format.

AoS versions: GCC produces the same exact code for the **explicit novec**, **auto** and **pragma** variants. However, the **abstract** versions are different from their **explicit** versions. This is not the case with the Clang compiler which produces the same exact assembly code for all the **novec**, **auto** and **pragma** variants independently of the considered approach.

SoA versions: For SoA versions, the assembly codes produced by GCC for the **explicit** and **abstract** versions are all the same for all tha variants. This is also true for all versions produced by Clang but the codes are different from those produced by GCC. In this case using our approach does not affect the compilation process, but GCC and Clang generate very different assembly codes.

AoSoA versions: All the **explicit** variants generated by GCC lead to the same assembly code. This is also the case for Clang. Thus no compiler is able to vectorize this code.

d) N-Body simulation: An N-Body simulation consists in computing forces bodies of a system applied to each other. Several steps are performed to determine the evolution of the system. Each body is defined by its position, speed, and mass. In this study we do not consider AoSoA versions of this code since vectorization of such a layout requires a complex algorithm to access contiguous values without mixing components in the same SIMD register.

AoS versions: The generated assembly codes show that GCC produces the exact same code for all **explicit** and **abstract** variants. This is also the case for Clang.

SoA versions: The conclusion is the same, the **abstract** approach does not impact the generation of assembly codes. Only the compiler has an impact on the code generation.

Once again, compilers are not able to vectorize this code. Regarding the previous results, it appears that transforming the data layout does not always help compilers to vectorize the code except in a few cases. It also appears that using our approach may impact the performance on some cases, depending on the algorithm, the chosen layout and on the chosen compiler. However, this is also true for explicit versions of the codes, but our approach does not require to rewrite the code to test different layouts and compilers.

We now also study the use of explicit vectorization for the previous codes to determine if the automatic vectorization may lead to comparable speedups.

B. *Vectorisation with intrinsics and Boost.SIMD library*

Explicit vectorization can be achieved by using intrinsics or specific libraries. In this section, we present both approaches.

a) Intrinsics: The following code is the AVX version of the vector normalization:

```
for( std::size_t i = 0 ; i < v.size() ; i += 8 ) {
  auto x = _mm256_load_ps( &v(VX, i ) ) ;
  auto y = _mm256_load_ps( &v(VY, i ) ) ;
  auto z = _mm256_load_ps( &v(VZ, i ) ) ;
  auto norm = _mm256_sqrt_ps( x * x + y * y + z * z );
  _mm256_store_ps( &v( VX, i ), x / norm );
  _mm256_store_ps( &v( VY, i ), y / norm );
  _mm256_store_ps( &v( VZ, i ), z / norm );
}
```

Listing 14. Vector normalization vectorized version with AVX intrinsics.

We do not detail the vectorization of other algorithms but we note that, moving from scalar operations to intrinsics increases the code complexity especially when converting types for our image algorithms or when permutations within registers are required. It also affects its portability since some AVX intrinsics are not always directly translatable into NEON ones. We consider using Boost.SIMD to alleviate these problems.

b) Boost.SIMD: This library offer a single generic type called `pack`, to represent a single or a set of SIMD registers, and provides arithmetic operators and functions. With this library, the code is transformed as follows:

```
for( std::size_t i = 0 ; i < v.size() ; i+=S ) {
  pack< float > x( &v( VX, i ) );
  pack< float > y( &v( VY, i ) );
  pack< float > z( &v( VZ, i ) );
  auto norm = sqrt( x * x + y * y + z * z );
  aligned_store( x / norm, &v( VX, i ) );
  aligned_store( y / norm, &v( VY, i ) );
  aligned_store( z / norm, &v( VZ, i ) );
}
```

Listing 15. Vector normalization, Boost.SIMD version.

Arithmetic operators and the `sqrt` function are overloaded, thus producing a code similar to the scalar one. Only the explicit call to the `aligned_store` function remains. The other algorithms are also easily converted to Boost.SIMD. Note that, for our image processing algorithms, conversion between 8-bit to 32-bit values is automatically performed by using a `cast` function. Thus, the resulting code length is similar to the scalar one. The grayscale conversion is a bit more complex to write:

```
for( size_t i=0 ; i<(rows*cols / S)*S ; i+=S ) {
  pack< uint8_t > r( &v_in( R, i ) );
  pack< uint8_t > g( &v_in( G, i ) );
  pack< uint8_t > b( &v_in( B, i ) );
  auto r0 = pack_cast<uint32_t>( r );
  auto g0 = pack_cast<uint32_t>( g );
  auto b0 = pack_cast<uint32_t>( b );
  auto gray = (r0*307 + g0*604 + b0*113)>>10;
  aligned_store(pack_cast<uint8_t>(gray), &v_out[i]);
}
```

Listing 16. Grayscale conversion, Boost.SIMD version.

Depending on the targeted architecture, this code can be transformed at compile time into SSE/AVX optimized codes or in AVX-512, NEON or VMX codes if the bSIMD library is used. We now verify that Boost.SIMD codes provide the same level of performance as their intrinsics versions.

C. Performance results

We have verified that in many cases, compilers generate similar assembly codes for our approach compared to standard implementations. We now verify if our approach gives the same performance results. Tests are performed on four different architectures. Two x86 platforms: one with a Broadwell Core i5-5300U processor with AVX2 and one with a Skylake XEON 6148 processor with AVX-512. Two ARM platforms: a Jetson TK1 board with a quad-core ARM Cortex A15 processor (32-bit) and a Jetson TX1 board with a quad-core ARM Cortex A57 processor (64-bit). We also study performance brought by the use of intrinsics and the Boost.SIMD library but in this case only for the Broadwell platform since Boost.SIMD only generates SSE/AVX instructions. We also test the ICC 17 compiler on the Skylake platform with the vectorization flags enabled and by replacing the `pragma omp simd` directive by the `pragma simd` directive.

a) Automatic vectorization: Figure 2 contains the execution times, given in milliseconds, for the four algorithms using different combinations. The **novec** variants are given as a baseline to study the impact of the layout independently of the vectorization. Tests are performed with the AoS, SoA and AoSoA data layouts using **explicit** and **abstract** approaches denoted respectively **exp** and **abs** in the figure.

As shown in the study of the generated assembly codes, compilers generate very different codes for an algorithm depending on the architecture, the chosen layout and the way to implement it, and the compilation flags. Thus, we may expect very different levels of performance. Results discussed below are highlighted in figure 2. For example, if we consider the results for the grayscale algorithm, on the Broadwell platform the best performance is obtained with the AoSoA explicit layout using the Clang compiler. On the Skylake platform, we obtain a different conclusion, the best performance is still obtained with the Clang compiler but with the AoS explicit or abstract layouts, if we only consider the GCC and Clang compilers, or with the SoA explicit or abstract layouts if we also consider the Intel compiler. On the ARM platforms, results are also different, since the best combination is explicit SoA layout with Clang on the TK1 and the explicit AoSoA layout also with Clang on the TX1 platforms.

For the other algorithms, best results are written in bold numbers. Several results may be identified as the best results when the execution times are similar. For the YUV2RGB algorithm on the Skylake platforms, the best result is obtained with ICC with an explicit AoSoA layout. It is two times faster than the Clang compiler which is the second fastest result but with an AoS layout. On both ARM platforms, the best result is obtained with GCC using an explicit AoSoA layout.

If we consider the normalize algorithm on the x86 platforms, we note that for each compiler, execution times are nearly the same independently of the layout. We observe a similar result on the ARM platforms. In this case we may assume that the code is compute bound i.e. the computation masks the memory accesses. However, if we compare the results on the x86 platforms from the Clang and the GCC compiler, we note that the former produces a nearly two times faster code. On ARM platforms, results obtained with both compilers are nearly identical, except for AoSoA layout.

For the nbody algorithm, results are really close on x86 platforms independently of the compiler or layout. Interestingly, best results on the TX1 platform are obtained with the abstract approach using the GCC compiler. We also note that ICC is able to vectorize the code which is indeed running 16 times faster, which corresponds to the theoretical speedup we can expect with AVX-512. This confirm that the code is also memory bound on these platforms.

From these results we conclude that, even on simple codes,

	variant	Broadwell gcc	clang	Skylake gcc	clang	icc	TK1 gcc	clang	TX1 gcc	clang
grayscale		exp/abs	exp/abs	exp/abs	exp/abs	exp/abs	exp/abs	exp/abs	exp/abs	exp/abs
aos	novec	165/165	146/145	149/153	132/131		255/253	252/252	500/522	367/512
aos	auto	125/125	107/108	77/78	54/53		506/522	470/495	132/132	131/131
aos	pragma	125/125	107/107	77/78	53/53	190/239	506/522	470/496	132/132	131/131
soa	novec	212/212	174/173	180/184	154/154		440/440	394/386	448/449	386/386
soa	auto	212/212	175/175	180/184	154/153		443/438	389/392	443/447	385/386
soa	pragma	212/212	61/175	180/180	154/153	40/40	447/439	135/386	441/442	112/385
aosoa	novec	217/270	146/266	223/239	132/239		527/597	393/804	408/511	323/581
aosoa	auto	217/270	61/263	223/239	132/239		527/599	148/803	410/511	122/581
aosoa	pragma	217/270	57/350	220/239	132/286	41/277	523/597	408/1147	405/513	109/815
yuv2rgb										
aos	novec	450/543	614/637	443/533	486/495		591/1223	965/1378	1055/1194	989/1184
aos	auto	465/543	**270**/637	415/476	**145**/496		591/1223	663/1377	1059/1194	560/1183
aos	pragma	543/515	635/413	476/463	472/520	459/181	1221/1016	1378/1168	1191/1144	1183/640
soa	novec	595/595	613/572	531/531	520/517		1012/1015	1345/1388	1148/1145	1131/1185
soa	auto	515/515	613/570	463/457	518/518		1015/1015	1351/1389	1148/1145	1157/1128
soa	pragma	515/515	413/571	463/457	520/524	181/195	1016/1010	1168/1388	1144/1142	640/1135
aosoa	novec	476/657	671/754	448/622	541/596		724/1662	1629/1547	1096/1281	1283/1193
aosoa	auto	476/713	670/753	424/584	543/593		532/1660	1627/1545	**394**/1281	1283/1194
aosoa	pragma	476/713	669/748	424/584	543/631	**61**/526	**511**/1667	1569/1547	**396**/1281	1285/1193
normalize										
aos	novec	970/940	**543/542**	881/881	**515/521**		3797/3791	3778/3206	**1892/1895**	**1892/1892**
aos	auto	970/940	**541/544**	881/881	**516/520**		3797/3793	3765/3205	**1896/1892**	**1894/1893**
aos	pragma	970/956	**544/545**	881/881	**518/519**	166/398	3769/3765	3767/3765	**1892/1895**	**1892/1892**
soa	novec	954/954	574/573	884/884	523/521		**3092/3092**	**3081/3079**	**1895/1896**	**1901/1902**
soa	auto	954/954	571/572	884/884	522/521		**3096/3097**	**3087/3079**	**1895/1894**	**1903/1896**
soa	pragma	954/954	568/575	884/884	522/522	**123/123**	**3091/3091**	**3089/3081**	**1897/1890**	**1891/1895**
aosoa	novec	970/940	554/568	882/882	521/533		**3108**/3820	3760/3840	**1892**/1991	**1894**/2714
aosoa	auto	970/940	554/568	882/882	521/533		**3101**/3821	3760/3841	**1896**/2062	**1892**/2707
aosoa	pragma	970/940	553/568	882/882	519/533	129/382	3893/3820	3762/3840	**1893**/2501	**1894**/2737
nbody										
aos	novec	1185/1185	**1144/1158**	1106/1105	**1057/1066**		**3612/3682**	**3619/3730**	2681/2686	2966/3216
aos	auto	1185/1199	**1144/1134**	1104/1123	**1059/1049**		**3610/3594**	**3623/3604**	2681/**2380**	2969/4087
aos	pragma	1197/1199	1220/1228	1111/1111	1153/1168	396/690	**3612/3681**	**3645/3732**	2691/2687	2763/3418
soa	novec	1200/1199	**1134/1134**	1123/1123	**1061/1055**		3761/3598	3775/3608	2504/**2381**	4121/4088
soa	auto	1200/1199	**1136/1134**	1121/1123	**1049/1049**		3767/3594	3699/3604	2476/**2380**	4119/4087
soa	pragma	1200/1202	1213/1245	1119/1119	1172/1122	**67**/74	3765/3590	4313/4204	2440/**2373**	4207/4087

Fig. 2. Performance with different layout/variant/architecture/compiler combinations (ms).

we can not always rely on compilers to perform automatic vectorization. Furthermore, even if automatic vectorization occurs, we need to verify if it achieves the performance of an explicit vectorization. Thus, we propose to study explicitly vectorized versions of our codes.

b) Explicit vectorization: We now present results obtained with explicit vectorization approaches. Our goals are: 1) to evaluate the speedup provided by the code vectorization 2) to evaluate if, in the cases when compilers are able to automatically vectorize the code, it is as efficient as explicit vectorization, 3) to show that combining our data layout approach and the Boost.SIMD library can provide the same level of performance as an explicitly vectorized code with an explicit data layout.

Results are presented in figure 3 and show that our abstract layout approach offers the same performance as an explicit implementation. If we compare with results in figure 2, we also observe that GCC and Clang are not able to reach the same level of performance obtained with explicit vectorization. Only ICC is able to reach a similar performance, but not for the YUV2RGB code with the SoA layout. Note that, the Intel compiler is only available for the x86 platforms. Thus, considering explicit vectorization on ARM platforms seems mandatory to take advantage of the NEON unit. For the grayscale algorithm, Clang does not generate a correct code on both ARM platforms when using an explicit layout. This behavior is not observed on x86 platforms or when using our abstract approach. For both normalize and nbody codes, which are compute bound, we observe a speedup close to the theoretical speedup we can expect from the underlying SIMD unit i.e. a speedup of 4 with NEON on the TX1 and TK1 platforms, of 8 with AVX, and of 16 with AVX-512. If we consider using the Boost.SIMD library, we observe on the Broadwell platform that the generated assembly code for all compilers is identical to the one generated from the intrinsics. We do not report the results in the figure since we obtain the exact same level of performance.

D. Concluding remarks on assembly codes and performance

a) GCC vs Clang vs ICC: On all platforms, GCC and Clang produce different assembly codes. The most notable differences are the way elements are accessed (index computation) and the order of instructions. For example, for the grayscale code on the Broadwell and Skylake platforms, the assembly code generated by Clang to access elements is more concise than GCC which uses additional registers to store offsets. This is also true for the nbody code but not for the YUV2RGB one. We also note that automatic vectorization is not always performed when expected. For example, GCC and Clang are able to vectorize the grayscale aos auto version but

	Broadwell		Skylake			TK1		TX1	
	gcc	clang	gcc	clang	icc	gcc	clang	gcc	clang
grayscale	47/48	48/48	34/34	34/34	34/34	143/141	-/134	154/153	-/101
yuv2rgb	87/87	83/82	51/51	43/48	42/43	386/387	282/283	290/291	206/222
normalize	284/284	266/267	124/124	128/124	123/122	1995/2038	651/658	541/517	519/534
nbody	174/174	197/199	74/74	90/91	65/73	2647/2649	892/910	1095/1083	1193/1196

Fig. 3. Performance (ms) of the vectorized version with the SoA layout.

not the soa one. This is not what we have expected since the soa access pattern is more simple to vectorize since it does not require transposition of color components before performing the computation.

The ICC compiler also generates different assembly codes and tend to unroll loops more frequently. It is the only one able to vectorize all codes but does not always better vectorization than GCC and Clang.

b) Automatic vectorization and pragmas: We observe that, in most cases, when the compiler vectorizes the code, adding a pragmas does not bring any further optimization. In few cases, it provides an additional speedup but only for the explicit versions. We also note that sometimes adding pragmas seems to break vectorization. This is the case for the YUV2RGB code with the aos layout using the Clang compiler on all the platforms. Sometimes, the generated code is even slower than the novec versions.

c) Processing of templates: Regarding our approach, we had expected the generated assembly code to be the same as the one obtained with the corresponding explicit versions. Indeed, since our containers are transformed at compile time into some combinations of tuples and vectors, then the access pattern is the same as the explicit one. However, we note that in lots of cases assembly codes differ. In a future study we propose to verify if the generated intermediate code is different to determine if this is due to the way templates are processed by the compiler or if further architectural optimizations are not performed the same way. Since automatic vectorization is often performed only for explicit versions, it suggests that the template mechanisms are more likely to alter optimization.

V. Conclusion

The main objective of our approach is to separate the data layout interface from its underlying representation to improve the portability and the performance of codes on current and future architectures. This work is also motivated by the presence of SIMD units in all modern processors from smartphones to supercomputers. Thus, we present a single interface to the user while the data layout is transformed to take advantage of optimizations like automatic vectorization by the compiler. The results of our experiments show that automatic vectorization occurs only in few cases and is not performed by all compilers, even on simple codes. Thus, we have investigated the combination of our approach with the Boost.SIMD library which abstracts the use of SIMD intrinsics for code vectorization. Using these two abstractions, we are able to reach the same level of performance as hand written code with intrinsics and SoA data layout, while keeping a scalar vision of the code, and ensuring portability on different architectures. Results show that we are able to reach speedups of 8 on several codes. In the future, we consider extending this work to accelerators like GPUs or Intel Xeon Phi, which are very specific architectures. We also plan to study more complex layouts for specific applications. Since current standard compilers are not able to automatically vectorize codes and to transform data layouts, we believe that extending our approach is the right way to go.

Acknowledgment

The authors thank Philippe Thierry, principal engineer at Intel for providing us access to Skylake Xeon platforms and to Intel tools and compilers.

We also thank Nvidia for providing us the Jetson TK1 board in the context of an Academic Partnership program.

We also thank Numscale SAS for giving us access to the bSIMD library.

References

[1] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté, "Boost.SIMD: generic programming for portable simdization," in Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing. ACM, 2014, pp. 1–8.

[2] E. Calore, N. Demo, S. F. Schifano, and R. Tripiccione, Experience on Vectorizing Lattice Boltzmann Kernels for Multi- and Many-Core Architectures. Cham: Springer International Publishing, 2016, pp. 53–62.

[3] N. Faria, R. Silva, and J. L. Sobral, "Impact of data structure layout on performance," in Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, ser. PDP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 116–120.

[4] D. Majeti, K. S. Meel, R. Barik, and V. Sarkar, "Automatic data layout generation and kernel mapping for cpu+gpu architectures," in Proceedings of the 25th International Conference on Compiler Construction, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 240–250.

[5] D. Majeti, R. Barik, J. Zhao, M. Grossman, and V. Sarkar, "Compiler-driven data layout transformation for heterogeneous platforms," in Euro-Par 2013: Parallel Processing Workshops. Springer, 2013, pp. 188–197.

[6] S. Xu and D. Gregg, "Semi-automatic composition of data layout transformations for loop vectorization," in Proceedings of the 11th International Conference on Network and Parallel Computing, NPC 2014. Springer, 2014, pp. 485–496.

[7] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-portability: Kokkos multidimensional array library," Sci. Program., vol. 20, no. 2, pp. 89–114, Apr. 2012.

[8] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in Proceedings of the 2013 Extreme Scaling Workshop (Xsw 2013), ser. XSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 18–24.

[9] M. Kretz and V. Lindenstruth, "Vc: A c++ library for explicit vectorization," Softw. Pract. Exper., vol. 42, no. 11, pp. 1409–1430, Nov. 2012.

[10] P. Karpiński and J. McDonald, "A high-performance portable abstract interface for explicit simd vectorization," in Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, ser. PMAM'17. New York, NY, USA: ACM, 2017, pp. 21–28.