Abstraction for AoS and SoA Layout in C++

Robert Strzodka*

Memory access patterns are critical for performance, especially on parallel architectures such as GPUs. Because of this, the choice between an *array-of-structures (AoS)* data layout and a *structure-of-arrays (SoA)* layout has a large impact on overall program performance. However, it is not always obvious which layout will better serve a particular application, and testing both of them by hand in C++ is tedious because their syntax greatly differs. Not only is the syntax for defining the container different, but worse, the syntax for accessing the data within the container is different, leading to anywhere from tens to thousands of source code changes needed to switch any given container from the AoS to the SoA layout or vice versa.

This chapter presents an abstraction layer that allows switching between the AoS and SoA layouts in C++ without having to change the data access syntax. A few changes to the structure and container definitions allow for easy performance comparison of AoS vs. SoA on existing AoS code. This abstraction retains the more intuitive AoS syntax (`container[index].component`) for data access yet allows switching between the AoS and SoA layouts with a single template parameter in the container type definition on the CPU and GPU. In this way, code development becomes independent of the data layout and performance is improved by choosing the correct layout for the application's usage pattern.

A library called ASX (Array of Structs eXtended) that implements this abstraction layer, together with code examples that execute on both the CPU and the GPU, can be downloaded from the author's homepage [8].

*Max Planck Institut Informatik, 66123 Saarbrücken, Germany

1 Introduction

Often our data is not scalar but rather comprises multiple components, such as the x, y, and z coordinates of a 3D position or velocity vector, the principle components of a feature vector, or the red, green, and blue color channels of a pixel. When operating on such multi-valued data, we have two major choices for the data layout: AoS or SoA. For the AoS layout, we define the multi-valued structure and assemble many copies of it one after another in memory. In the case of SoA, we begin with all instances of the first component, then follow it with all instances of the second component, etc.

Collections of C++ class instances fall naturally into the AoS pattern, though this may not necessarily represent the best choice for performance for a given application (see Section 5.1). Consider the following examples of these two layouts. The same data is represented in both cases, but because of the different layout, the memory access patterns during computation are significantly different.

```
// Array of Structures (AoS)
struct Element {
  Type1 comp1;
  Type2 comp2;
  Type3 comp3;
};
typedef Element AoS_Container[CONT_SIZE];

// Structure of Arrays (SoA)
struct SoA_Container {
  Type1 comp1[CONT_SIZE];
  Type2 comp2[CONT_SIZE];
  Type3 comp3[CONT_SIZE];
};
```

Switching between these two layouts is labor-intensive, as the access syntax for the two forms differs, as seen below. Alternatively, a third possible access syntax could be used, which is the standard C++ solution to this problem: a class hides the data layout from the programmer by turning every structure component access into a call to a member function, thereby allowing the class implementation to switch between AoS and SoA layouts without having to change the access syntax.

```
value= container[index].component; // AoS access
value= container.component[index]; // SoA access
value= container.component(index); // C++ access
```

With the C++ member function access syntax, however, the definition of every structure (even simple structures with only a few components each) requires many additional lines of code, especially since the proper treatment of constness requires two access functions for every data member (or, alternatively, a getter and a setter function for each member):

```
const Type& component(int index) const; // read access
      Type& component(int index);       // write access
```

Besides the code bloat when defining accessor functions, the standard C++ solution has another problem: the AoS layout allows performing *in-place* updates on the container element at a certain index position with the *same* functions that are used on singleton elements:

```
Element single;                  // single element
AoS_Container container;         // AoS container
void update( Element& elm ) { elm.comp1= elm.comp2; }

update( single );                // OK
update( container[5] );          // OK
```

With the SoA layout or the standard C++ solution, this is not possible because the component must be selected first and the index afterwards. Consequently, in that case two different functions are necessary to perform the same operation as above:

```
Element single;                  // single element
Cpp_Container container;         // C++ container
void update( Element& elm ) { elm.comp1= elm.comp2; }
void update( Cpp_Container& con, int i )
{ con.comp1[i]= con.comp2[i]; }

update( single );                // the same operation ...
update( container, 5 );          // ...  but different syntax
```

Of course, this problem can also be solved with more C++ machinery by defining iterators and passing `Cpp_Container::iterator(single)` and

`Cpp_Container::iterator(container,5)` as arguments to functions, however, this only adds to the code bloat and hardly anyone is willing to implement so many classes for every small data structure. Moreover, many changes would have to be applied to an existing AoS code before it could work with this type of C++ solution.

2 Core Method

Our goal is to introduce SoA functionality with minimal changes to the element and container definitions and without abandoning the more intuitive AoS syntax `container[index].component`. In this way it will be easy for programmers to transition from existing AoS code to a flexible code that supports both AoS and SoA. As the data access syntax remains the same, the main additional work required from the programmer is a more flexible structure definition based on the *ASA (array of structs of arrays)* coding pattern.

```
// AoS pattern:  concise but restricted to AoS layout
struct Element {
  Type1 comp1;
  Type2 comp2;
  Type3 comp3;
};
typedef Element Container[CONT_SIZE];

// ASA pattern:  a bit longer but handles both AoS and SoA
template <ASX::ID t_id= ASX::ID_value>
struct FlexibleElement {
  typedef ASX::ASAGroup<Type1,t_id> ASX_ASA;
  union{ Type1 comp1;  ASX_ASA dummy1; };
  union{ Type2 comp2;  ASX_ASA dummy2; };
  union{ Type3 comp3;  ASX_ASA dummy3; };
};
typedef FlexibleElement<> Element;

// specify ASX::SOA instead for an SoA container
typedef ASX::Array<FlexibleElement, CONT_SIZE, ASX::AOS>
    Container;

// common access syntax:  normal AoS code continues to function
Element single= {1, 2, 3};       // OK, no change
Container container;             // OK, no change
```

```
single.comp2= single.comp3;      // OK, no change
container[5].comp3= 2;           // OK, no change
```

The layout and behavior of the types `Element` and `Container` are exactly the same no matter whether the AoS pattern or the ASA pattern with AoS layout (parameter `ASX::AOS`) is used. However, the ASA pattern allows changing the in-memory data layout to SoA (corresponding to `SoA_Container` in Section 1) simply by replacing `ASX::AOS` with `ASX::SOA`. The normal AoS access syntax works in either case without any changes as shown above.

For some code, the above change to the container definition is everything that is required from the programmer. However, sometimes additional code adjustments are needed. First, references to ASA elements have necessarily different types depending on whether they refer to a single element, a container element, or possibly both. It is still possible to write a single function for handling of single element and container element references similar to the standard AoS solution, but the syntax of references must be adjusted, see Section 4.1. Second, the above ASA container definition assumes that `Type1`, which is used in `ASX::ASAGroup<...>`, is greater than or equal to `Type2` and `Type3` in size. The handling of differently-sized components, array-valued components and nesting of structures are discussed in Sections 4.2 through 4.4.

In this article, we focus on the ASA (array of structs of arrays) coding pattern. The first three examples included in the ASX library (`simpleAoS`, `simpleSoA`, and `simpleASA`) deal with the simple three-valued structure from above and allow an easy comparison of a standard AoS solution, a standard SoA solution and the ASA solution that supports both layouts. In particular, the transition from the standard AoS solution to the ASA solution requires only few changes, while the transition from the standard AoS solution to the standard SoA solution is far more time-consuming even for this simple structure.

The code above uses the `ASX::Array` class, which provides static allocation based on a size that is known at compile-time, i.e., the equivalent of a normal array in C. All examples using `ASX::Array` are located in the directory `examples/staticCUDA` of the ASX library.

For cases requiring dynamic allocation, the ASX library also provides the class `ASX::Vector`, which uses a constructor to allocate memory in a manner similar to STL vectors. The corresponding examples are located in the

directory `examples/dynamicCUDA`. Section 3.2 discusses the few differences between `ASX::Array` and `ASX::Vector`.

3 Implementation

3.1 Statically Sized Containers

The technical goal of the implementation is to enable the use of the intuitive AoS syntax `container[index].component` for both data layouts. Clearly, using AoS syntax for an AoS layout is simple; the challenge lies in defining an `operator[]` such that the AoS syntax can access an SoA container. The following code explains why this is possible under certain conditions:

```
struct SoA_Container {
  Type1 comp1[CONT_SIZE];
  Type2 comp2[CONT_SIZE];
  Type3 comp3[CONT_SIZE];
};
SoA_Container container;            // SoA container

Type3& soa= container.comp3[5];     // normal SoA syntax
Type3& aos= reinterpret_cast<SoA_Container&>
            (container.comp1[5]).comp3[0]; // AoS-like
```

The main insight is that `soa` and `aos` reference the same value if `Type1`, `Type2`, and `Type3` all have the same size in memory, because then it does not matter in which order the index offset and the structure member offset are applied. This solves the problem of swapping the order of `operator[]` and `operator.` for the data access on the SoA layout. Now we only need to hide the ugly typecast in the `operator[]` function and eliminate the trailing `[0]` by giving `comp3[0]` a fixed name with the help of an anonymous union:

```
struct SoA_Container {
  SoA_Container& operator[](int index) { return
    reinterpret_cast<SoA_Container&>(comp1[index]); };
  union{ Type1 comp1val, comp1[CONT_SIZE]; };
  union{ Type2 comp2val, comp2[CONT_SIZE]; };
  union{ Type3 comp3val, comp3[CONT_SIZE]; };
};
SoA_Container container;            // SoA container
```

```
Type3& soa= container.comp3[5];     // normal SoA syntax
Type3& aos= container[5].comp3val; // normal AoS syntax
```

The last line here does exactly the same as the last line in the previous listing, except it is much cleaner using normal AoS syntax. Clearly, we can also access an AoS layout with AoS syntax, so we have achieved our goal of showing that the syntax can stay the same even though the layout changes. Only the `operator[]` function must be defined differently depending on the layout, and this can be achieved with a template specialization. Although no further technical tricks are involved in the implementation, the actual code of `ASX::Array` is much longer because of the template specialization and the partial compatibility of the interface with the containers of the STL library, i.e., many types and query functions are defined.

`ASX::Array` behaves like a normal array in C and therefore shares its disadvantage that the container size `CONT_SIZE` must be a compile-time constant as well as its advantages that it can be used to allocate on-chip shared memory in CUDA and that it contains nothing but the user data, so to copy a `ASX::Array` to the GPU, we simply invoke `cudaMemcpy`:

```
typedef ASX::Array<Flex,CONT_SIZE,ASX::AOS> Container;
Container host_A, *device_pA;
cudaMalloc(&device_pA, sizeof(host_A));
cudaMemcpy(device_pA, &host_A,
           sizeof(host_A), cudaMemcpyHostToDevice);
```

3.2 Dynamically Sized Containers

When dynamic container sizing is required, `ASX::Vector` can be used in place of `ASX::Array`. It circumvents the static size limitation by defining an `ASX::Array` of fixed size (granularity) and then allocating a dynamically defined number of them. The indexing in the `operator[]` function becomes slightly more complicated, but otherwise little changes. This is similar to the way an STL vector works.

Consequently, as with an STL vector, `ASX::Vector` itself contains only control logic and metadata, while the user data is stored at a different address. Therefore, moving the container between the host and device requires two copies: one for the metadata and one for the user data, and an adjustment of the pointer to the user data. The library provides the convenience function `ASX::deepCopyVector` for these actions:

```
typedef ASX::Vector<Flex,GRANULARITY,ASX::AOS> Container;
Container host_A(CONT_SIZE), *device_pA, *device_pAdata;
cudaMalloc(&device_pA, sizeof(host_A));
cudaMalloc(&device_pAdata, host_A.memory_size())
ASX::deepCopyVector(device_pA, device_pAdata,
                    &host_A, cudaMemcpyHostToDevice);
```

Note that the container size `CONT_SIZE` in this example can be dynamically defined. The constant `GRANULARITY` controls the granularity of the allocation, or it can be set to zero to use the default setting.

Apart from the above differences, the containers defined by `ASX::Array` and `ASX::Vector` utilize exactly the same syntax as described in the other sections. Code examples with `ASX::Array` are located in `examples/staticCUDA` and with `ASX::Vector` in `examples/dynamicCUDA`.

4 ASA in Practice

In the following sections, we discuss some caveats to be aware of when switching to the ASA pattern as well as support for more complex data layouts as they appear in practice.

4.1 References to Elements

In standard AoS code, a reference to a singleton element or to an indexed element in a container always has the type `Element&`. For the ASA pattern, these two have different types when an SoA layout is selected: a single element has its components laid out consecutively in memory, hence the reference is of type `Element&`, whereas an element within a container has its components spread out in memory, requiring the reference to be of type `Container::reference`. If an argument to a function could be either a singleton element or an indexed container element, then a templated type for that argument is required.

```
template <ASX::ID t_id>
void update( FlexibleElement<t_id>& elm )
{ elm.comp1= elm.comp2; }                    // OK, no change

Element single= {1, 2, 3};
```

```
Container container;

Element& refSE= single;                        // OK, no change
Container::reference refSE= single;            // Type error

Element& refCE= container[5];                  // Type error
Container::reference refCE= container[5];      // OK, changed

update( single );                              // OK, no change
update( container[5] );                        // OK, no change
```

Therefore, when transitioning from AoS code to ASA code, we need to replace `Element&` by `Container::reference` wherever it refers to a container element, and element references in function parameters must become template types if both single and container elements may be passed. The function calls themselves do not need to be changed, because the compiler deduces the corresponding template instantiation automatically from the passed parameter.

Besides the new container definition, the different treatment of references are the only changes that are necessary to turn the original AoS code `simpleAoS` into the flexible ASA code `simpleASA` in the example folder of the ASX library.

4.2 Components of Different Sizes

If the structure components have different sizes, they should be grouped such that the component groups have the same or at least similar sizes. While this sort of grouping is not strictly necessary, it is advisable, as it benefits performance and minimizes the memory footprint.

```
// AoS pattern:  concise but restricted to AoS layout
struct Element {
  float  c1;
  short  c2[2];
  double c3;
};
typedef Element Container[CONT_SIZE];

// ASA pattern:  a bit longer but handles both AoS and SoA
template <ASX::ID t_id= ASX::ID_value>
struct FlexibleElement {
```

```
  typedef ASX::ASAGroup<double,t_id> ASX_ASA;
  union{struct{float c1; short c2[2];}; ASX_ASA dummy1;};
  union{double c3;                      ASX_ASA dummy3;};
};
typedef FlexibleElement<> Element;
typedef ASX::Array<FlexibleElement, CONT_SIZE,
        ASX::AOS> Container; // ASX::SOA for an SoA container

Container container;
container[5].c3= 2;          // OK, no change
```

The above code shows the two critical code features that must be present in every flexible structure definition.

- The structure must contain a `typedef ASX::ASAGroup<GroupType,t_id> ASX_ASA`, where the user chosen type `GroupType` defines the *grouping size*. The size of each component group must be smaller or equal than the grouping size.
- All components must be a member of a component group. Each component group must be embedded in an *anonymous union*, which includes a dummy component of type `ASX_ASA`. Memory is wasted if the component groups have different sizes.

Grouping of components within a `union` should be performed with an *anonymous struct*; otherwise, the grouped components (`c1` and `c2` in the example above) would occupy the same memory and could not be used simultaneously. Most compilers support this use of an anonymous structure, although it is not part of the C++ standard. If this is not supported, a normal named structure can be used instead, however, then this name has also to appear in the access syntax.

Note that in the example above, had we required just one element of type `short` (i.e., if `float c1` were grouped with `short c2` rather than `short c2[2]`), then both the AoS and the ASA code would still execute correctly, even though the elements in the group would have different sizes. Moreover, the overall size of the container would not change in either case: for the AoS pattern, this is because of data alignment by the compiler; for the ASA pattern, it is because of the explicit alignment according to the grouping size. The grouping of differently-sized components is demonstrated in more detail in the example `groupingASA` in the ASX library.

4.3 Array-Valued Container

In the previous example, we have already used an array of components (`shorts`), however, the array components were located in the same group, so they are not separated in memory if we switch to the SoA layout. If we want to put the array components into different groups then we must use a different syntax. In the simplest case we want a large array-valued container with flexible layout. This is achieved with the provided class `ASX::FlexibleArray`, which interacts with the container class `ASX::Array` in almost the same way as the user defined structures `FlexibleElement` in the examples before, here we merely need an additional `::TTypeASA` because that is how C++ supports template typedefs.

```
// AoS pattern:  concise but restricted to AoS layout
typedef float Element[ARR_SIZE];
typedef Element Container[CONT_SIZE];

// ASA pattern:  a bit longer but handles both AoS and SoA
typedef ASX::FlexibleArray<float, ARR_SIZE> FlexElement;
typedef FlexElement::TTypeASA<> Element;
typedef ASX::Array<FlexElement::TTypeASA, CONT_SIZE,
        ASX::AOS> Container; // ASX::SOA for an SoA container

Container container;
container[5][1]= 2;           // OK, no change
```

The full example code is available in the folder `arrayASA` in the ASX library.

4.4 Nesting of Composite Types

The flexible structure definitions can be nested to form a larger flexible structure. For this purpose, we use the `FlexibleElement` defined in Section 2 and the `FlexElement` defined in Section 4.3 and add one more component.

```
// AoS pattern:  concise but restricted to AoS layout
struct Element {
  struct{ Type1 c1; Type2 c2; Type3 c3; } comp1;
  float comp2[ARR_SIZE];
  int comp3;
};
typedef Element Container[CONT_SIZE];
```

```
// ASA pattern:  a bit longer but handles both AoS and SoA
template <ASX::ID t_id= ASX::ID_value>
struct LargeFlex {
  typedef ASX::ASAGroup<int,t_id> ASX_ASA;
  FlexibleElement<t_id>       comp1;
  FlexElement::TTypeASA<t_id> comp2;
  union{ int                  comp3;  ASX_ASA dummy3; };
};
typedef LargeFlex<> Element;
typedef ASX::Array<LargeFlex, CONT_SIZE,
        ASX::AOS> Container; // ASX::SOA for an SoA container

Container container;
container[5].comp1.comp3= 2; // OK, no change
```

Since the flexible structures already consist of component groups internally, they are not embedded in a union; only the last component of native type requires one.

Because the grouping size inside a flexible structure is hard-coded into the `ASX::ASAGroup<...>` typedef, the nesting of flexible structures is only possible if all their grouping sizes are the same. In the above example, `sizeof(Type1)` is the grouping size of `FlexibleElement` from Section 2 and `sizeof(float)` is the grouping size of `FlexElement` from Section 4.3 so for the above code to function correctly these sizes must be equal. This restriction could be removed by making the grouping size a template parameter of every flexible structure definition. But this would create considerable additional code complexity for all cases in order to resolve few exceptional cases of mismatched nested grouping sizes and therefore has not been implemented.

The corresponding example in the ASX library is named `nestingASA`.

5 Final Evaluation

5.1 Performance

Memory access patterns are critical for performance, especially on the GPU where caches are smaller. The data layout greatly influences the memory access patterns and therefore the choice of AoS or SoA layout has a large

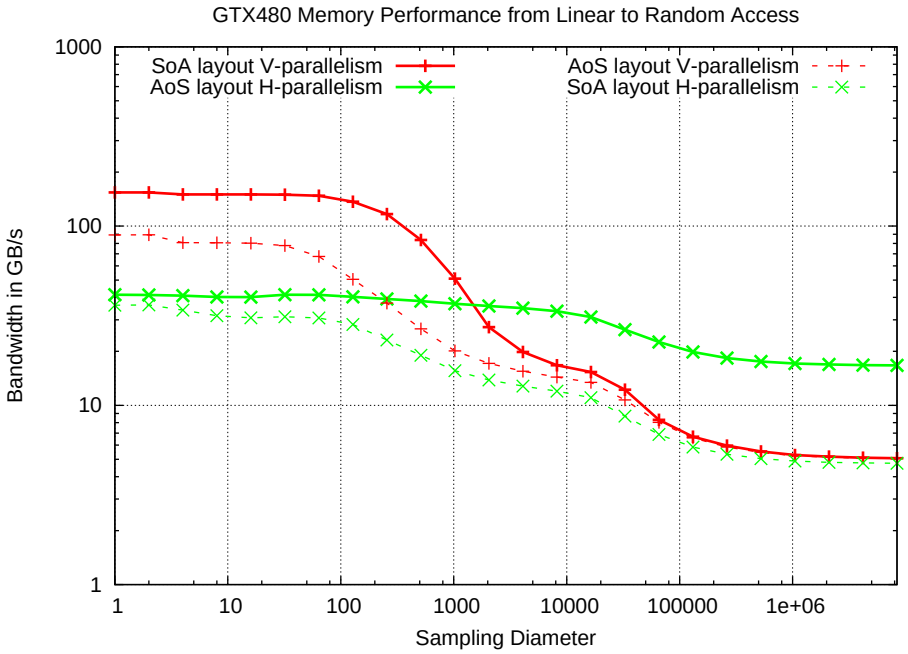

Figure 1: Memory performance on two large containers in which each element consists of four floats. Access patterns vary from linear element indexing on the left, over increasingly irregular access in the middle, to a completely random permutation on the right. All four combinations of two data layouts (AoS and SoA) and two parallelization strategies (horizontal and vertical) are presented. The mismatched combinations (dashed lines) perform clearly worse. The matching combinations (solid lines) differ by 3.7x on the left and 3.3x on the right, which demonstrates the large speedups that can be gained by choosing the correct data layout.

impact on overall program performance. This section discusses how to make the correct decision and reports on the resulting speedups.

For testing, we choose a kernel that performs a *saxpy*-type (scalar alpha X plus Y) operation, however, alpha and the components of the containers X and Y are not scalars but structures or short arrays themselves, therefore *gaxpy* (general axpy) is a more appropriate name. This is a good candidate to test the effective global memory performance for multi-valued containers because it has no on-chip data reuse. The tests cover the full spectrum of index access patterns from linear indexing to a random permutation. From the linear index list $0, 1, \ldots$ we create a permuted list by replacing index i with a random j that fullfills $|i - j| < d/2$ for a chosen sampling diameter d. By varying d we obtain access patterns of different irregularity and Figure 1

shows the results on a NVIDIA GeForce GTX 480 card in a 64-bit Linux system running the `g++-4.3.2` and `nvcc-3.1` compilers.

A SoA layout is almost always parallelized vertically, i.e., multiple instances of the same structure component are processed in parallel, whereas an AoS layout is almost always parallelized horizontally, i.e., multiple components of the same structure instance are processed in parallel. Figure 1 gives quantitative support for these choices, because the less typical combinations, shown as dashed lines, are clearly worse.

The comparison of the usual combinations, shown as solid lines, is dramatic. The SoA layout is 3.7x faster on the regular patterns as they typically appear in structured numerical simulations, but it loses by a similar factor of 3.3x on the irregular patterns that are typical of statistical and database processing. This clearly shows that one cannot rely on the same data layout for all purposes. A tool is required that can quickly switch from one layout to the other to find out which setting is best, and the ASX library provides exactly this functionality.

In practice, the situation is even more complex, because different application containers might require different data layouts. Up to now, finding the best data layout for each container was infeasible, because changing the syntax for even a single container from one layout to the other is very time consuming and error-prone. With the ASX library, each container can be easily configured to a different layout by setting a template parameter in the class definition. This ensures that the best possible memory performance can be obtained.

5.2 Related Work

AoS and SoA layouts are discussed in tutorials about SIMD processing for various architectures, e.g., CPU [4], GPU [6] or the Cell processor [3]. They advocate the use of SoA and vertical parallelism because this gives the fastest processing in case of large, linearly indexed data sets. Hybrid formats that adapt the data layout to the memory access granularity of the hardware by grouping of instances or components result in arrays of structures of arrays [1] or structures of arrays of structures [7] constructions, respectively. Such application specific data layout optimizations appear in many high performance codes, often leading to machine-specific code paths for the most critical parts of a program. The contribution in this paper is an abstraction

that allows the selection of different data layouts within the same code. The data layout can be changed at compile-time for each container.

A different approach is taken by Intel Array Building Blocks [5], which keeps the original layout but performs a transformation on collections of user defined structures into better-suited layouts at runtime. The library Blitz++ [9] allows the specification of compile-time user-defined storage orders for multi-dimensional arrays, which offers even more flexibility than the usual choices of row-major or column-major storage, but this solution does not apply to structures. Gou et al. [2] discuss related work and new ideas concerning a hardware solution for better support of different data layouts and strided memory access.

5.3 Limitations

Some limitations of the ASA abstraction remain; some are inherent to the abstraction, others are induced by the programming language.

Algorithm The abstraction uses the AoS syntax for both data layouts. An SoA layout that can be accessed with a single `operator[]` necessarily requires a grouping of components (Section 4.2). So the abstraction does not offer all the data arrangement options that are possible in standard SoA code that uses multiple `operator[]`s for data access. However, in view of the standard compiler alignment, this is not a big restriction in practice. For example, a `struct{short a; int b;}` is padded by the compiler automatically as `struct{short a, pad; int b;}`, forming two groups `short a, pad` and `int b` of the same size as required by ASA.

An abstraction integrated into the programming language would have a smoother syntax and better checks on correctness, therefore a certain awareness about data type sizes and their alignment is required from the programmer as can be seen in the examples of Sections 4.2 to 4.4.

Coding The abstraction aims at leaving the AoS code unchanged and enabling the flexible layout by simply changing the container definition. However, references to container elements must also be changed as discussed in Section 4.1. Nonetheless, this is still much less effort (see e.g. the differences between the examples `simpleAoS` and `simpleASA` in the ASX library)

than changing a standard AoS code to a standard SoA code (see e.g. the differences between the examples `simpleAoS` and `simpleSoA`).

There are some restrictions with respect to dynamic memory allocation on the GPU. The statically sized `ASX::Array` can be used to allocate on-chip shared memory in CUDA. The dynamically sized `ASX::Vector` can also be stored in a previously statically allocated part of shared memory, but it cannot be used for its allocation. Concerning global memory, CUDA Toolkit 3.2 supports dynamic allocation of global memory on NVIDIA Fermi-based GPUs.

Performance The user-chosen grouping of components is hard-coded by the placement of the unions. Optimization of the grouping size requires code changes in the container definition. Luckily, the granularity of memory access is given by the hardware manufacturers, so often it is not difficult to choose the most efficient grouping size, e.g., selecting a size that avoids bank conflicts in on-chip shared memory.

5.4 Benefits

Section 5.1 showed the performance benefits of choosing the correct data layout. In theory, one could obtain these benefits manually, if one is willing to write an exponential number of code variants that correspond to the different assignments of data layouts to the containers: two choices for each container, 2^N code variants for N containers. Clearly, this is infeasible, and the flexible ASA code improves the situation on all levels: algorithm development, coding and performance:

Algorithm

- Algorithm development is independent of the data layout.
- Externally defined data layouts can be quickly integrated with new algorithms.

Coding

- The more intuitive AoS syntax can be used throughout the code.
- The same template function can be used to update single elements and indexed elements within a container.

Performance

- Performance of AoS vs. SoA for individual containers, groups of containers or the entire project can be evaluated rapidly by changing a few template parameters.
- AoS vs. SoA comparison and optimization can be integrated into an auto-tuning framework without the need for source code manipulation.

Acknowledgements

A big thank you goes to Cliff Woolley from NVIDIA for helping with the organization of the paper.

References

[1] James Abel, Kumar Balasubramanian, Mike Bargeron, Tom Craver, and Mike Phlipot. Applications tuning for streaming SIMD extensions. Technical report, Intel, 1999.

[2] Chunyang Gou, Georgi Kuzmanov, and Georgi N. Gaydadjiev. SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 49–59. ACM, 2010.

[3] IBM. *Developing code for Cell - SIMD*, 2006. http://-publib.boulder.ibm.com/infocenter/ieduasst/stgv1r0/topic/-com.ibm.iea.cbe/cbe/1.0/Programming/L3T2H1_37_DevelopingCode-ForCellSIMD.pdf.

[4] Intel. *A Guide to Vectorization with Intel C++ Compilers*, 2010. http://-software.intel.com/file/31848.

[5] Intel. Intel array building blocks. http://software.intel.com/en-us/-articles/intel-array-building-blocks/, 2011.

[6] Brent Oster. Advanced CUDA tutorial. http://www.nvidia.com/-content/cudazone/download/Advanced_CUDA_Training_NVI-SION08.pdf, 2008.

[7] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. CUDA memory optimizations for large data-structures in the Gravit simulator. In *Proc.*

Workshop on Simulation and Modelling in Emergent Computational Systems (SMECS) at ICPP 2009, pages 174–181. IEEE Computer Society, September 2009.

[8] Robert Strzodka. ASX (Array of Structs eXtended). http://www.mpi-inf.mpg.de/%7Estrzodka/software/ASX/, 2010.

[9] Todd Veldhuizen. Blitz++ library. http://www.oonumerics.org/blitz/, 2006.